Nuit de l'Info

Ensemble de tutos pour aider les participants de la nuit de l'info

- Docker
- Développement web avec C#
- Développement Web avec Java (et Spring)

Docker

Qu'est ce que Docker ?

Meme

Docker est un système de virtualisation légère. En gros, les containeurs docker sont des machines virtuelles où on ne virtualise pas le matériel (contrairement à une VM VirtualBox ou VMWare), ni le système d'exploitation. On utilise les capacités du noyau linux (comme LXC) pour conteneuriser les processus.

L'idée, c'est de créer des environnements applicatifs portables et prêt à l'emploi. Et ducoup c'est super pratique pour gérer une infrastructure sur un serveur, mais aussi pour lancer rapidement des applications (comme des bases de données notamment) pour développer sur notre poste.

Pour résumer, un conteneur docker c'est une machine virutelle légère, qui contient les dépendances nécessaire à l'exécution d'un processus.

Installer Docker

Sur Linux (par exemple sur le serveur)

```
sudo apt update # Mettre à jours les paquets
sudo apt install apt-transport-https ca-certificates curl software-properties-common # Installer de quoi récupérer la clé de sécurité de docker
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - # Récupération de la clé de sécurité de docker
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable" # Ajout de dépôt APT de docker
sudo apt update # Mettre à jours les paquets
sudo apt install docker-ce # Installation de docker
sudo systemctl status docker # Pour vérifier que le service tourne bien
```

Sur windows (par exemple votre poste)

Il faut avoir l'édition "Pro" de Windows ainsi qu'avoir son système à jours. Il faut tout d'abord installer WSL (Windows Subsystem for Linux) :

```
wsl --install
```

Installation via Winget

```
winget install Docker.DockerDesktop
```

Installation via Chocolatey

```
choco install docker-desktop
```

Installation manuelle

Téléchargez et exécuter l'installeur via ce lien.

Concepts généraux

Notion d'image

Une image docker, c'est la recette du conteneur, on la constuit à partir d'un Dockerfile, qui va décrire ce que contient l'image en terme de dépendances, librairies et exécutable, ainsi que comment le processus doit être exécuté.

Notion de conteneur

Un conteneur, c'est une instance d'une image en train de tourner.

Docker Hub

Docker Hub, est un dépôt central qui permet de publier des images docker, pour permettre à d'autre gens de les utiliser facilement.

Lancer un conteneur à partir d'une image

> Pour la suite du tutoriel, ajoutez `sudo` à toutes les commandes docker si vous êtes sur Linux. Si vous êtes sur Windows, n'oubliez pas le lancer le docker

> daemon en lançant l'app "Docker Desktop".

Pour lancer une image docker, il faut utiliser la commande `docker run`. Essayons de lancer un contenur à partir de l'image MySQL (au hasard), disponible à ce lien.

La commande pour lancer le conteneur est la suivante :

```
docker run -p 3306:3306 --name mon-mysql -e MYSQL_ROOT_PASSWORD=monMdpTresSecret -d mysql:latest
```

Explications :

- `-p` : permet de spécifier les ouvertures de port du conteneur sous la forme `externe:interne` (externe = port de la machine hôte, interne = port du conteneur). Ici on forward le port `3306`, qui est utilisé par MySQL
- `--name mon-mysql` : donner le nom "mon-mysql" au conteneur
- `-e` : permet de passer una variable d'environnement au conteneur. Ici on passe `MYSQL_ROOT_PASSWORD` qui d'après la documentation de l'image permet de paramètrer le mot de passe de l'utilisateur `root` de la base de donnée. Les variables d'environnement suivantes de cettes image peuvent aussi être intéressantes :
  - `MYSQL_DATABASE` : crée automatiquement dans le conteneur une base de donnée au nom correspondant à la valeur de cette variable
  - `MYSQL_USER` et `MYSQL_PASSWORD` : créer automatiquement dans le conteneur un utilisateur avec le login et password correspondant à la valeur de ces variables
- `-d` : permet de préciser l'image à utiliser et le tag, sous la forme `image:tag`. Le tag permet de préciser la version. On met souvent `latest` pour avoir le plus récent.

Et voilà, le tour est joué ! Le conteneur est lancé. Pour voir vos conteneurs actifs, vous pouvez faire :

```
docker ps
```

Pour voir les logs du conteneur vous pouvez faire :

```
docker logs -f mon-mysql
```

Construire une image

Pour construire une image docker, il faut définir un `Dockerfile`. Pour démonstration, nous allons créer une image pour une application web Java faisant juste un "Hello World".

Setup du projet

Pour commencer, récupérez le projet :

```
git clone https://github.com/Ombrelin/java-springboot-helloworld.git
cd java-springboot-helloworld
```

Pour le construire (Java 11 requis):

Windows	Linux
`./mvnw.cmd clean package`	`./mvnw clean package`

Cela va vous construire un JAR exécutable dans `target/build`.

Le Dockerfile

Pour commencer à la racine du projet un dossier `Dockerfile`.

On commence par hériter notre image d'une autre, pour ré-utiliser ce qu'elle contient :

```
FROM adoptopenjdk/openjdk11
```

Cette image contient un JDK 11 déjà installé. On va ensuite exposer le port de l'application web, pour pouvoir y accéder en dehors de l'image :

```
EXPOSE 8080
```

Ensuite on copie notre fichier JAR de notre application dans l'image :

```
ADD ./target/demo-0.0.1-SNAPSHOT.jar app.jar
```

Enfin, on précise le point d'entrée de notre processus, tout simplement la commande qui exécute l'application :

```
ENTRYPOINT ["java","-jar","/app.jar"]
```

Constuction à partir du Dockerfile

Pour construire une image à partir de votre Dockerfile :

```
docker build -t mon-image .
```

Vous pouvez ensuite lister vos images locales pour y voir apparaitre la votre :

```
docker images
```

Pour lancer votre image pour la tester, un simple `docker run` :

```
docker run -p 8080:8080 -t mon-image
```

Voilà, vous avez lancé un conteneur à partir de votre image, vous pouvez lancer votre navigateur à http://localhost:8080 pour vérifier que l'application est bien lancée.

Persister les données des conteneurs

Pour l'instant les données de vos conteneurs ne sont pas persistées à l'extérieur, elle disparaitront si vous supprimez le conteneur, ce qui n'est pas très pratique pour une base de donnée par exemple.

Pour remédier à cela, on utilise la notion de volumes, afin de monter un dossier de notre système hote, dans le conteneur docker, persmettant à celui-ci de persister des fichiers.

Pour ce faire on va utiliser l'option `-v DossierMachineHote:DossierConteneur` . Par exemple pour persister les données de MySQL:

```
docker run -p 3306:3306 --name mon-mysql -v /un/dossier/sur/mon/ordi:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=monMdpTresSecret -d mysql:latest
```

Pour aller plus loin

Les sujets suivants peuvent être intéressants :

- Docker Compose
- Publier ses conteneurs sur Docker Hub

Développement web avec C#

Qu'est ce que le C#

C# ou csharp, est un langage développé par Microsoft pour concurrencer Java. Comme Java, c'est un langage semi-compilé et interprété par un environnement d'exécution. L'environnement d'exécution du C# est le .NET Runtime (prononcé dotnet), l'équivalent de la JVM de Java.

En Java, on compile les sources (fichiers `.java`) pour obtenir le bytecode (fichiers `.class`) que l'on package en exécutable (fichier `.jar`) pour les exécuter avec le Java Runtime Environnement. En C#, oncompile les sources (fichiers `.cs`) pour obtenir le code en Intermediate Language (fichiers `.il`) que l'on package en exécutable (fichiers `.dll` - attention les DLL .NET n'ont rien à voir avec les DLL natives).

Contrairement à certaines idées reçues, C# et .NET sont :

- Entièrement gratuits et open-source (licences MIT)
- Cross-platforme (comme Java, *write once, run anywhere*)

Mettre en place son environnement de développement

Installer .NET CLI

Avec Winget

```
winget install Microsoft.dotnet
```

Avec Chocolatey

```
choco install dotnet
```

Manuellement

Téléchargez et exécutez l'installer du **.NET SDK** via ce lien

Créer un projet web

Vous pouvez consulter les templates de projet en faisant :

```
dotnet new -l
```

Pour créer un nouveau projet d'application web :

```
dotnet new mvc -o "mon-projet"
```

Nous venons de créer une application web à page avec le framework ASP .NET Core MVC. Vous pouvez

Environnement de développement

Ce tutoial n'est pas dépendant d'un IDE particulier, il utilise la CLI .NET pour créer et exécuter les projets. Ainsi, vous pouvez utiliser n'importe quel IDE ou éditeur de texte. Je recommande les suivants :

- Jetbrains Rider : IDE Jetbrains dédié au développement .NET
- Visual Studio Code avec le plugin "C#" de Microsoft

Présentation de ASP .NET Core MVC

Le pattern MVC

Comme son nom l'indique, ce framework se base sur le pattern MVC, qui divise l'architecture de l'application en trois parties :

- Modèle : les données de l'application
- Vue : les pages web de l'application
- Controlleur : gestion des actions des utilisateurs, coordonne données et pages

Explication du layout du projet

- Dossier `Controllers` : Contient les classes controlleurs de votre application. Vous pouvez supprimer le fichier d'exemple créé par le projet
- Dossier `Models` : Contient les classes de logique et de données de votre application. Vous pouvez supprimer le fichier d'exemple créé par le projet

- Dossier `Views` : Contient les templates HTML du projet. Vous pouvez supprimer le dossier `Home` d'exemple créé par le projet
  - Dossier `Shared` : Contient les éléments de template ré-utilisables :
    - `_Layout.cshtml` : Squelette de toutes vos pages, l'appel `@RenderBody()` rend le contenu de la page. Vous pouvez vider les `<header>` et `<footer>` d'exemple créés par le projet, et les remplacer par votre propre `<header>` et `<footer>`. Tout ce que vous mettrez dans ce template sera rendu sur toutes les pages de votre application
    - `_Layout.css` : le CSS propre à votre squelette de pages
    - `_ValidationScriptsPartial.cshtml` et `Erros.cshtml` : vous pouvez les supprimer, ce sont des exemples du projet
  - `_ViewImports.cshtml` : Imports des namespaces dans les templates. Comme on a vider le namespace `Models`, son import ne compile plus, vous pouvez le commenter pour l'instant (la syntaxe pour les commenaire ici est `@* commenaire *@`
  - `_ViewStart.cshtml` est le point d'entrée de vos template
- Dossier `wwwroot` : les fichiers statiquement servis par votre application
  - `css` : le CSS de votre application
  - `js` : le JS de votre application
  - `lib` : les libraires JS et CSS téléchargées localement
  - `favicon.ico` : l'icone de votre site
- `Program.cs` : le point d'entré de votre application

Controllers & Routes

Pour créer un controlleur, créez une classe dont le nom finit par "Controller" dans le dossier `Controllers`. Par exemple, "HelloWorldController", qui étend la classe `Controller`, par exemple :

```
namespace net_web_tuto_2.Controllers
{
    public class HelloWorldController : Controller
    {

    }
}
```

Pour ajouter un préfixe de route à votre controleur, mettez lui l'attribut [Route("maroute")], par exemple :

```
    [Route("hello-world")]
    public class HelloWorldController : Controller
    {
```

```
    }
```

Ensuite, vous pouvez créer une méthode endpoint, avec l'annotation `[HttpGet]` :

```
        [HttpGet]
        public string HelloWorld()
        {
            return "Hello world !";
        }
```

La route de ce endpoint est dont `GET /hello-world`. On peut constater en lançant l'application avec un :

```
dotnet run
```

Et en en allant à https://localhost:7229/hello-world

On peut donner des suites de route particulières aux méthodes endpoint :

```
    [HttpGet("fr")]
    public string HelloWorldFr()
    {
        return "Bonjour, le monde";
    }

    [HttpGet("en")]
    public string HelloWorldEn()
    {
        return "Hello, world";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à :

- https://localhost:7229/hello-world/fr
- https://localhost:7229/hello-world/en

Paramètres de requête et endpoints

Paramètre de requête

Pour récupérer une paramètre de requête, il suffit de mettre un paramètre avec le même nom à la méthode endpoint :

```
    [HttpGet]
    public string HelloWorld(string name)
    {
        return $"Hello, {name}";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world?name=Arsène.

Paramètre de chemin

Pour récupérer un paramètre de chemin, il suffit de mettre ce paramètre entre accolades dans la route du endpoint, puis ajouter n paramètre avec le même nom à la méthode endpoint :

```
    [HttpGet("{id}")]
    public string HelloWorld(int id)
    {
        return $"Id : {id}";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world/1.

Vues

Retourner une vue

Avant de retourner une vue, il faut d'abord créer un dossier dans `Views` pour notre controleur qui correspond à son nom. Ici notre controleur s'appelle `HelloWorldController`, notre dossier va donc s'appeller `HelloWorld`.

On va ensuite créer dedans un fichier `hello-world.cshtml`, qui sera notre template :

```
<h1>Hello, world</h1>
```

Pour le retourner depuis une méthode endpoint, il faut utiliser la fonction `View()` du controleur :

```
    [HttpGet]
    public IActionResult HelloWorld()
    {
        return View("hello-world");
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Templater une vue

Avec une classe modèle

On peut templater une vue avec un modèle de façon fortement typée, ce qui est pratique car cela permet de répérer d'eventuelles erreurs plus vites.

Soit la classe de modèle (dans le dossier `Model`, et n'oublier pas de décomenter l'import du namespace dans `_ViewImports.cshtml`) :

```
namespace net_web_tuto.Models {
    public class Person {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

Dans la méthode endpoint, on ajouter le modèle dans l'appel à `View` :

```
    [HttpGet]
    public IActionResult HelloWorld()
    {
        var person = new Person(){ FirstName = "John", LastName = "Shepard"};
        return View("hello-world", person);
    }
```

Et pour le template, la syntaxe est la suivante :

```
<h1>Hello, @Model.FirstName @Model.LastName</h1>
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Avec d'autres données

Pour templater la vue avec d'autres infos que celles d'un modèle, on peut utiliser le `ViewBag`, exemple :

Dans la méthode endpoint, on ajouter le modèle dans l'appel à `View` :

```
    [HttpGet]
    public IActionResult HelloWorld()
    {
        var person = new Person(){ FirstName = "John", LastName = "Shepard"};
      ViewBag.Today = DateTime.Now;
        return View("hello-world", person);
    }
```

```
<h1>Hello, @Model.FirstName @Model.LastName, The Date is : @ViewBag.Today</h1>
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Rendu conditionnel

Pour faire du rendu conditionnel il faut utiliser `@if` :

```
@if((20 % 2) == 0) {
<p> 20 is even </p>
}

@if((21 % 2) == 0) {
<p> 21 is not even </p>
}
```

Seulement "20 is even" sera rendu sur la page.

Boucles

Pour faire du rendu en boucle, il faut utiliser `@foreach`, exemple :

```
@foreach (var person in Model.Persons)

    <div>@item.FirstName @item.LastName</div>
}
```

Soumission de formulaire

Pour gérer une soumission de formulaire, il faut un modèle dont les noms des champs correspondent aux champs du formulaire, puis ajouter ce modèle en paramètre la méthode enpoint. La méthode endpoint doit également utiliser la méhotde HTTP POST, et donc l'attribut `[HttpPost]`. Exemple :

Les deux templates, pour le formulaire et le résultat :

`hello-world-form.cshtml` :

```
<form action="/hello-world/salute" method="post">
    <input name="FirstName" type="text"/>
    <input name="lastName" type="text"/>
    <input type="submit" value="Say hello !"/>
</form>
```

`hello-world.cshtml` :

```
<h1>Hello, @Model.FirstName @Model.LastName</h1>
```

Les méthodes endpoint :

```
    [HttpGet]
    public IActionResult HelloWorldForm()
    {
        return View("hello-world-form");
    }

    [HttpPost("salute")]
    public IActionResult SayHello(Person person)
    {
        return View("hello-world", person);
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `dotnet run`). Et en allant à https://localhost:7229/hello-world.

Session HTTP

On peut stocker des données dans la session HTTP. Pour cela il faut d'abord la convigurer en rajoutant dans le `Program.cs` (après la ligne `var builder = WebApplication.CreateBuilder(args);`) :

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

Et après `app.UseAuthorization();` :

```
app.UseSession();
```

Pour enregistrer des données dans la session :

```
HttpContext.Session.SetString("key", "value");
```

Pour récupérer des données de la session :

```
string value = HttpContext.Session.GetString("key");
```

Interaction avec la base de donnée

L'interaction avec la base de donnée se fait via l'ORM (Object-Relationnal Mapper) officiel de Microsoft, Entity Framework Core.

Installer Entity Framework Core

Il faut installer les packages nugets via les commandes suivantes :

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 6.0.0
dotnet add package Pomelo.EntityFrameworkCore.MySql --version 6.0.0
```

Il faut également installer l'outil en ligne de commande d'EF Core pour gérer les migrations :

```
dotnet tool install --global dotnet-ef
```

Le `DbContext`

Il faut ensuite créer notre classe `DbContext`, point d'entrée de notre base de donnée. Créez un dossier `Database` et dans ce dossier une classe `ApplicationDbContext` qui étend `DbContext`:

```
using Microsoft.EntityFrameworkCore;

namespace net_web_tuto.Database {

    public class ApplicationDbContext : DbContext {

        public ApplicationDbContext(DbContextOptions options) : base(options) {
        }
    }

}
```

Il faut ensuite rattacher notre `ApplicationDbContext` à notre application, dans votre `Program.cs` (après la ligne `var builder = WebApplication.CreateBuilder(args);`), rajoutez :

```
// Database
builder.Services.AddDbContext<ApplicationDbContext>(options => {
    string databaseHost = Environment.GetEnvironmentVariable("DATABASE_HOST");
    string databaseName = Environment.GetEnvironmentVariable("DATABASE_NAME");
    string databaseUsername = Environment.GetEnvironmentVariable("DATABASE_USERNAME");
    string databasePassword = Environment.GetEnvironmentVariable("DATABASE_PASSWORD");

    var connectionString =
$"server={databaseHost};database={databaseName};user={databaseUsername};password={databasePass
word}";
    options.UseMySql(connectionString, ServerVersion.AutoDetect(connectionString));
});
```

On récupère les identifiant depuis des variables d'environnement, c'est plus projet que des les avoir dans le code. Vous devriez aussi rajouter des `using` au début du fichier :

```
using Microsoft.EntityFrameworkCore;
using net_web_tuto.Database;
```

Définir les entités

Revenons à notre `ApplicationDbContext` auquel on va rajouter la méthode suivante pour définir nos entités :

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {


        }
```

Il faut aussi rajouter en attributs un `DbSet` qui va resprésenter notre table :

```
public DbSet<Person> Persons {get;set;}
```

Ensuite dans la méthode `OnModelCreating`, on peut définir notre entité (on a ajouté un champs `Id` de type `Guid` à la classe `Person` pour servir d'identifiant dans la base :

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
    modelBuilder.Entity<Person>().HasKey(person => person.Id);
            modelBuilder.Entity<Person>().Property(person => person.FirstName);
            modelBuilder.Entity<Person>().Property(person => person.LastName);
        }
```

On peut également définir des contraintes sur les attributs de l'entité, avec des méthodes comme `HasMaxLength`, `HasMinLength` chainées après l'appel à `Property`. Vous pouvez trouver plus de précisions sur la documention d'EF Core

Migrations

Après avoir défini notre modèle, il faut créer notre migration vers la base de donnée et l'exécuter. On va utiliser l'invite de commande EF Core pour ça.

Dans un premier temps on définit notre variables d'environnement (exemple avec Powershell, la syntaxe change selon votre shell):

```
$Env:DATABASE_HOST = "localhost"
$Env:DATABASE_NAME = "nomDeLaDb"
```

```
$Env:DATABASE_USERNAME = "userDeLaDb"
$Env:DATABASE_PASSWORD = "mdpDeLaDb"
```

> Avant de faire une migration, faite bien attention d'avoir éteint votre application, sinon vous allez avoir des err

Ensuite, exécutez la commande suivante pour créer la migrations :

```
dotnet ef migrations add maMigration
```

Enfin pour exécuter la migration sur votre base :

```
dotnet ef database update
```

Si vous vous connectez à votre base avec un client, vous deviez voir le schéma déployé.

Utiliser le `DbContext` dans le controleur

> Attention, si vous utiliser Rider, reportez les varibales d'environnement dans le fichier `Properties/launchSettings.json` dans l'objet `environmentVariables` de l'objet profile que vous utilisez pour lancer votre app (si vous avez un doute vous pouvez le mettre dans les deux).

Pour interagir avec la base de donnée depuis notre controleur, on peut demander au framework de nous l'injecter :

```
using Microsoft.AspNetCore.Mvc;
using net_web_tuto.Database;
using net_web_tuto.Models;

[Route("persons")]
public class PersonsController : Controller {

    private readonly ApplicationDbContext context;

    public PersonsController(ApplicationDbContext context){
        this.context = context;
    }
```

```
}
```

Créons maintenant un endpoint servant une vue avec :

- Une liste des personnes dans la base
- Un formulaire pour ajouter une personne

```
@foreach (Person person in @Model)
{
    <p>@person.FirstName - @person.LastName</p>
}

<form action="/persons" method="post">
    <input name="FirstName" type="text"/>
    <input name="LastName" type="text"/>
    <input value="Create" type="submit"/>
</form>
```

D'abord, pour récupérer les personnes, nous allons accèder à notre `DbSet` et le transformer en liste, pour EF Core, cela revient à faire une requête pour récuprer tous les enregistrement de la table :

```
    [HttpGet]
    public IActionResult PersonsPage(){
        List<Person> persons = this.context.Persons.ToList();

        return View("persons", persons);
    }
```

Ensuite, on peut ajouter notre endpoint d'insertion :

```
    [HttpPost]
    public IActionResult AddPerson(Person p){
        this.context.Persons.Add(p);
        this.context.SaveChanges();

        return new RedirectResult("/persons");
    }
```

On ajoute la `Person` récupérée depuis le formulaire au `DbSet`, puis on appelle `SaveChanges` afin de valider la transaction sur la base.

Enfin, on retourne une redirection vers notre endpoint d'affichage pour réafficher la page avec les données mises à jours.

> Vous pouvez regarder dans la console les requêtes effectuées par EF Core sur la base de données, qui correspondent à nos appels

On va ensuite ajouter une vue pour voir une seule personne, avec un lien dans la liste. Créons d'abord notre endpoint :

```
    [HttpGet("{id}")]
    public IActionResult GetOnePerson(Guid id){
        Person person = this.context.Persons
            .First(p => p.Id == id);

        return View("person", person);
    }
```

On utilise un paramètre de route pour récupérer l'identifiant. Ensuite, la méthode `First` prend un prédicat (fonction retournant un booléen) sur les personnes et permet de filtrer la liste pour ne retourner que le premier enregistrement qui valide le prédicat. EF Core compile cette méthode sous la forme d'une clause `WHERE` dans la requête SQL. L'appel à `First` permet ensuite de récupérer le premier élément du résultat (ici logiquement il n'y en a qu'un).

Pour faire un filtrage sur la table et récupérer une collection d'enregistrement, il faut utiliser la méthode `Where` qui prend également un prédicat, mais retourne tous les enregistrement qui le valident :

```
        List<Person> persons = this.context.Persons
            .Where(p => p.FirstName.Contains("e"))
            .ToList();
```

On retourne ensuite la vue suivante :

```
@Model.FirstName - @Model.LastName
```

On rajoute dans notre vue `persons` des liens vers notre nouveau endpoint :

```
@foreach (Person person in @Model)
{
    <p><a href="/persons/@person.Id">@person.FirstName - @person.LastName</a></p>
```

```
}
```

Les relations avec EF Core

Pour créer des relations, il faut utiliser les méthodes `HasOne`, `HasMany`, `WithOne`, `WithMany` dans la méthode `OnModelCreating` du `DbContext`. Les méthodes "Has" permettent de définir le premier dans de la relation, et les méthodes With, permettent de définir l'autre coté. Cela permet de créer toutes les relations possibles. Exemple avec une relation "1 lié à n" (avec un livre lié à des pages) :

```
        modelBuilder.Entity<Book>()
            .HasMany(book => book.Pages) // Premier sens
            .WithMany(page => page.Book); // Sens inverse
```

On est pas obligé de passer de paramètre dans la méthode de sens inverse, si on a pas besoin de la navigation en sens inverse (ici : si on a pas besoin d'avoir de référence au livre dans la page).

Ensuite, pour récupérer dans une requête le conteu de la propriété de navigation, il faut utiliser `Include` :

```
context.Books
    .Include(book => book.Pages)
    .ToList();
```

Dockeriser une application ASP .NET Core

D'abord on build un exécutable de notre application :

```
dotnet publish -c Release
```

On crée ensuite le Dockerfile suivante à la racine du projet

```
# On part d'une image microsoft pour ASP .NET Core
FROM mcr.microsoft.com/dotnet/aspnet:6.0

# On copie notre résultat de publication dans le conteneur
COPY bin/Release/net6.0/publish/ App/

# On se place là où on a copié
```

```
WORKDIR /App

# On expose le port 80
EXPOSE 80

# On lance l'application
ENTRYPOINT ["dotnet", "net-web-tuto.dll"]
```

On peut ensuite build l'image :

```
docker build -t mon-image -f Dockerfile .
```

Et enfin lancer l'image :

> Attention, si votre base de donnée est aussi un conteneur Docker tournant sur votre machine, dans l'hote de la base de donnée, notez non pas "localhost" mais l'adresse locale de votre machine pour permettre à la connexion de loopback entre vos conteneurs.

```
docker run -p 80:80 -e DATABASE_HOST=<IP de votre base de donnée> -e DATABASE_NAME=testdb -e DATABASE_USERNAME=root -e DATABASE_PASSWORD=monMdpTresSecret -t mon-image
```

Sources

- Documentation d'ASP .NET Core MVC
- Documentation d'Entity Framework Core

Pour aller plus loin

Si vous voulez aller plus loin, je vous conseille de vous intéresser aux sujet suivants :

- API ReST avec ASP .NET Core WebAPI
- Architectures Logicielles Backend :
  - N-Tiers
  - Clean Architecture

Des tutos sur ces sujets apparaitront peut être sur ce site à l'avenir !

Développement Web avec Java (et Spring)

Introduction

Pour faire du Web en Java, il ya plusieurs possibilités les principales sont :

- JakartaEE (anciennement JavaEE) : un ensemble de spécification pour des APIs de développement d'applications d'entreprise en Java
- Spring : un framework Java pour faciliter le développement de tout types d'applications

Ici, nous allons utiliser Spring car il est très facile à aborder et on est très rapidement productif avec. Il est très utilisé dans l'industrie. JakartaEE est aussi intéressant à connaître, vous aurez un cours d'introduction dessus au S4.

Spring est un très grand framework avec beaucoup de fonctionnalités, dont des fonctionnalités de développement d'application web MVC, que nous allons utiliser dans ce tutoriel.

Mettre en place un environnement de développement Java

Installer un JDK

Avec Winget

```
winget install Microsoft.OpenJDK.17
```

Avec Chocolatey

```
choco install openjdk17
```

Avec APT

```
sudo apt install openjdk-17-jdk
```

Avec Homebrew

```
brew install --cask oracle-jdk
```

Manuellement

Suivez ce lien, téléchargez et exécutez l'installeur correspondant à votre plateforme.

Environnement de développement

Ce tutoial n'est pas dépendant d'un IDE particulier, il utilise la CLI Maven pour exécuter les projets. Ainsi, vous pouvez utiliser n'importe quel IDE ou éditeur de texte. Je recommande les suivants :

- Jetbrains IDEA : IDE Jetbrains dédié au développement Java
- Visual Studio Code avec le pack de plugins Java" de RedHat

Présentation de Spring et Spring Boot

Spring Boot ?

Spring Boot est une des fonctionnalités de Spring qui permet de créer une application "stand-alone" (sans avoir besoin de serveur d'application) et de l'autoconfigurer pour des besoin simple. Cela permet de démarrer rapidement sans passer trop de temps à configurer son projet.

Le pattern MVC

Comme son nom l'indique, ce framework se base sur le pattern MVC, qui divise l'architecture de l'application en trois parties :

- Modèle : les données de l'application
- Vue : les pages web de l'application
- Controlleur : gestion des actions des utilisateurs, coordonne données et pages

Créer un projet Web

D'abord rendez vous sur start.spring.io, il s'agit d'un configurateur de projets spring, très utile pour démarrer un projet rapidement.

Voici les différence options à prendre :

- Project : Maven Project
- Language : Java
- Spring Boot : 2.6.0
- Projet Metadata :
  - Group : sous la forme `fr.<votre pseudo>` (c'est une convention)
  - Artifact : nom de votre projet
  - Name : nom de votre projet
  - Description : un texte libre décrivant votre projet
  - Package Name : laissez la valeur générée
  - Packaging : JAR
  - Java : 17 (vérifiez bien que vous avez Java 17 si vous l'aviez déjà avant ce tuto)
- Dépendencies : cliquez sur "Add Dependencies" et séléctionnez "Spring Web", ainsi que "Thymeleaf" (Thymleaf est le moteur de template que nous allons utiliser pour les vues).

Explication du layout du projet

- Dossier `.mvn` : wrapper Maven qui nous permet d'utiliser Maven (le build tool du projet) sans avoir besoin de l'installer sur notre système (on ne va jamais y toucher).
- Dossier `src/main/java/<package name de votre projet>` : package racine de nos source, on vas y mettre nos packages et nos classes Java.
- `src/main/java/<nom du projet>Application.java` : fichier Java qui contient la méthode `main` de notre application Spring
- Dossier `src/main/resources/static` : fichiers servis statiquement par notre application
- Dossier `src/main/resources/templates` : c'est ici qu'on va ranger nos templates de vues
- Fichier `src/main/resources/application.properties` : dans ce fichier on va définir la configuration de notre application
- Dossier `src/main/test` : Dossier contenant les classes de tests. Nous ne couvrirons pas le sujet des tets dans ce tuto
- Fichier `mvnw` et `mvnw.cmd` : script du wrapper maven, on va l'appeler pour lancer et construire notre projet (`mvnw` pour mac/linux et `mvw.cmd` pour windows).
- Ficheir `pom.xml` : descripteur Maven de notre projet, il contient les méta-données du projet, ainsi que les dépendances

Controlleurs & Routes

Commencez par créer un nouveau package `controller` (sous `src/main/java/<package name de votre projet>`). Pour créer un controlleur, créez une classe dont le nom finit par "Controller" dans ce package. Par exemple, "HelloWorldController", et annotez cette classe avec `@Controller` par exemple :

```
package fr.arsenelapostolet.springmvcapptuto.controller;


import org.springframework.stereotype.Controller;
```

```
@Controller
public class HelloWorldController {

}
```

Pour ajouter un préfixe de route à votre controleur, mettez lui l'attribut [RequestMapping("maroute")], par exemple :

```
@RequestMapping("hello-world")
@Controller
public class HelloWorldController {

}
```

Ensuite, vous pouvez créer une méthode endpoint dans la classe controleur, avec l'annotation `@GetMapping` :

```
    @GetMapping
    @ResponseBody
    public String HelloWorld() {
        return "Hello world !";
    }
```

> Ici, on doit également ajouter `@ResponseBody` pour que notre string soit utilisée telle qu'elle et non comme un nom de template. Nous n'aurons plus besoin de la rajouter quand nous utiliserons les templates.

La route de ce endpoint est donc `GET /hello-world`. On peut constater en lançant l'application avec un :

```
./mvnw.cmd spring-boot:run
```

Et en en allant à https://localhost:8080/hello-world

On peut donner des suites de route particulières aux méthodes endpoint :

```
  @GetMapping("fr")
@ResponseBody
    public string HelloWorldFr() {
```

```
        return "Bonjour, le monde";
    }

    @GetMapping("en"))
    @ResponseBody
    public string HelloWorldEn() {
        return "Hello, world";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à :

- [https://localhost:8080/hello-world/fr]
- [https://localhost:8080/hello-world/en]

Paramètres de requête et endpoints

Paramètre de requête

Pour récupérer une paramètre de requête, il suffit de rajouter un paramètre annoté avec `@RequestParam` à la méthode endpoint, et de passer en paramètre de l'annotation le nom du paramètre :

```
    @GetMapping
    @ResponseBody
    public String HelloWorld(@RequestParam("name") String name) {
        return "Hello, " + name + " !";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world?name=Arsène.

Paramètre de chemin

Pour récupérer un paramètre de chemin, il suffit de mettre ce paramètre entre accolades dans la route du endpoint, puis ajouter un paramètre annoté avec `@PathVariable` à la méthode endpoint, et de passer en paramètre de l'annotation le nom du paramètre :

```
    @GetMapping("{id}")
    @ResponseBody
```

```
    public String HelloWorld(@PathVariable("id") int id) {
        return "Id : " + id;
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world/1.

Vues

Avant de retourner une vue, il faut d'abord en créer dans le dossier `src/main/resources/templates`. On va donc créer dedans un fichier `hello-world.html`, qui sera notre template :

```
<h1>Hello, world</h1>
```

Pour le retourner depuis une méthode endpoint, il faut retourner le nom du fichier (sans l'extension), et ne pas mettre l'annotation `@RequestBody` :

```
    @GetMapping
    public String HelloWorld() {
        return "hello-world";
    }
```

Templater une vue

Avec des données quelleconques

Pour templater une vue, il faut ajouter l'objet `Model` en paramètre de la méthode endpoint, et utiliser la méthode `addAttribute` pour ajouter des objets au modèle :

```
    @GetMapping
    public String HelloWorld(Model model) {
        model.addAttribute("today", LocalDate.now());

        return "hello-world";
    }
```

On peut ensuite utiliser la syntaxe suivante pour accèder au modèle dans la vue :

```
<h1>The date is : <span th:text="${today}"></span></h1>
```

Le moteur de template va appeler la méthode `toString` de l'objet `LocalDate` pour l'afficher.

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world.

Avec un objet de modèle

Avec un objet de modèle, la syntaxe est la même, on peut accèder aux attributs avec un point. Ce qui va appeler les getters & setters. Par exempe avec la classe suivante :

```
public class Person {

    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

On peut l'ajouter au modèle de la même façon :

```
    @GetMapping
    public String HelloWorld(Model model) {
        Person p = new Person();
        p.setFirstName("John");
        p.setLastName("Shepard");
```

```
        model.addAttribute("today", LocalDate.now());
        model.addAttribute("person", p);

        return "hello-world";
    }
```

Et y accèder dans la vue :

```
<h1>The date is : <span th:text="${today}"></span></h1>

Hello, <span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
```

Rendu Conditionnel

Pour faire du rendu conditionnel, il faut utilise l'attribut `th:if` :

```
<span th:if="${(20 % 2) == 0}">20 is even</span>
<span th:if="${(21 % 2) == 0}">21 is even</span>
```

Seulement "20 is even" sera rendu sur la page.

Boucles

Pour faire du rendu en boucle, il faut utiliser l'attribut `th:each`, exemple :

```
<div th:each="person: ${persons}">
<span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
</div>
```

Soumission de formulaire

Pour soumettre un formulaire avec Thymleaf, il faut d'abord sur le formulaire :

- L'attribut `th:action` qui va indiquer l'URL de l'endpoint qui gère la soumission du formulaire
- L'attribut `th:object` qui va indiquer le nom de l'objet du modèle que l'on va modifier dans le formulaire
- L'attribut `method` qui indique le verbe HTTP utiliser pour la soumission du formulaire, il faut utiliser POST. La méthode endpoint qui gère le formulaire doit aussi gérer le verbe HTTP POST et va donc utiliser l'annotation `PostMapping`

Sur les champs du formulaire, il faut l'attribut `th:field` afin d'indiquer à quel attribut de la classe modèle le champs correspond.

Dans la méthode endpoint qui sert le formulaire, il faut ajouter dans le modèle l'objet que l'on veut modifier avec le formulaire (son nom doit correspondre avec l'attribut `th:object` du formulaire).

Exemple :

Les deux templates, pour le formulaire et le résultat :

`hello-world-form.html` :

```
<form th:action="@{/hello-world/salute}"  th:object="${person}" method="post">
    <input th:field="*{firstName}" type="text"/>
    <input th:field="*{lastName}" type="text"/>
    <input type="submit" value="Say hello !"/>
</form>
```

`hello-world.html` :

```
Hello, <span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
```

Les méthodes endpoint :

```
    @PostMapping("salute")
    public String SayHello(@ModelAttribute Person person) {
        return "hello-world";
    }

    @GetMapping
    public String HelloWorld(Model model) {
        model.addAttribute("person", new Person());
        return "hello-world-form";
    }
```

On peut ensuite constater le résultat en redémarrant l'app (CTRL + C dans la console puis `./mvnw.cmd spring-boot:run`). Et en allant à https://localhost:8080/hello-world.

Session HTTP

On peut stocker des données dans la session HTTP. Pour cela, ajoutez simplement une `HttpSession` en paramètre de votre méthode endpoint :

```
@GetMapping
    public String EndpointWithSession(HttpSession session){
      ...
    }
```

Pour enregistrer des données dans la session :

```
session.setAttribute("clé","valeur");
```

Pour récupérer des données de la session :

```
String value = session.getAttribute("clé");
```

Interaction avec la base de donnée

L'interaction avec la base de donnée se fait avec un ORM. Ici on va utiliser une implémentation de JPA (Java Persistence API). JPA est la spécification de JarkataEE pour les ORMs. Nous allons utiliser Hibernate, son implémentation la plus connue.

Installer Hibernate

Il faut installer le package nugger d'hibernate, ainsi que le driver de base de donnée en dépendance Maven. Pour ce faire, ouvrez votre `pom.xml` et ajoutez les balises suivantes à la balise `<dependencies>`:

Hibernate :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Driver JDBC MySQL :

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
```

```
    <version>8.0.22</version>
</dependency>
```

Configurer la connexion

Afin de configurer la connexion à la base de donnée, ouvrez le fichier `application.properties` situé dans `src/main/resources`. Et ajoutez les entrées suivantes :

```
spring.datasource.url=${DATABASE_URL}
spring.datasource.username=${DATABASE_USERNAME}
spring.datasource.password=${DATABASE_PASSWORD}

spring.jpa.generate-ddl=false
spring.jpa.open-in-view=false
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults=false
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL55Dialect
```

Dans le fichier `application.properties`, la syntaxe `${}` permet de récupérer la valeur d'une variable d'environnement.

> L'URL JDBC de la base de donnée MySQL se consruit comme cecii : `jdbc:mysql://<hote>:3306/<nomDeLaBase>`

Les Entités

Les entités sont les objets qui sont persistés par l'ORM dans la base de donnée. Pour définir nos classes d'objets qui seront enregistrées dans la base de donnée, il faut les définir en tant qu'entité. Cela se fait à l'aide de l'annotation `@Entity` sur la classe. Une entité doit avoir un champs qui correspondra à se clé primaire dans la base de donnée, il est désigné avec l'annotation `@Id`. l'ORM peut le générer automatiquement, et même aléatoirement dans le cas d'une UUID (voir exemple).

Les autres champs de la classe entités sont persistés automatiquement. Pour qu'un champs ne soit pas persisté, il faut l'annoter avec `@Transient`.

Exemple d'entité pour notre Todo :

```
@Entity
public class Person {
  @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String firstName;
    private String lastName;

  ... getters & setters ...
}
```

Les annotations `@GeneratedValue` et `GenericGenerator` permet de générer automatiquement un UUID aléatoire pour l'entité.

On peut aussi définir des contraintes sur les colonnes en rajoutant l'annotation `@Column` sur le champs et en lui passant des paramètres, par exemple :

```
@Column(unique = true, length = 32)
```

Cela permet de mettre la contrainte "unique" sur le champs et d'imposer une longueur maximum de 32 caractère

Les Repositories

Pour interagir avec les entités, il faut créer des Repository. Ce sont des interface que nous allons définir, mais qui seront implémentées non pas nous, mais pas l'ORM Hibernate. Ils peuvent ensuite être injectés dans les services par le conteneur d'injection de dépendances de Spring. Une interface Repository gère les interactions pour une entité et doit étendre l'interface `JpaRepository` en fournissant en paramètre de type, le type de l'entité, ainsi que le type de son Id.

Exemple :

```
public interface PersonsRepository extends JpaRepository<Todo, String> {


}
```

Voilà, juste en étendant cette interface, on peut accèder à tout un tas de méthodes intéressantes. Les principales sont :

- `save` : Sauvegarder (créer ou mettre à jours) une instance d'une entités
- `findById` : récupérer une entité à partir de son Id
- `findAll` : récupérer toutes les entités de ce type contenues dans la base (⚠ attention ça peut faire beaucoup de tout charger dans la mémoire)
- `deletebyId` : supprimer une entité à partir de son Id

Et bien d'autre qui peuvent être utile, à retrouver dans la javadoc de l'interface

Utiliser le repository dans le contrôleur

Pour interagir avec la base de donnée depuis notre controleur, on peut demander au framework de nous l'injecter :

```
@Controller
@RequestMapping("persons")
public class PersonsController {


    private final PersonsRepository repository;


    public PersonsController(PersonsRepository repository) {
        this.repository = repository;
    }
}
```

Créons maintenant un endpoint servant une vue avec :

- Une liste des personnes dans la base
- Un formulaire pour ajouter une personne

```
<div th:each="person: ${persons}">
    <span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
</div>


<form th:action="@{/persons}" th:object="${person}" method="post">
    <input th:field="*{firstName}" type="text"/>
    <input th:field="*{lastName}" type="text"/>
    <input type="submit" value="Save"/>
</form>
```

D'abord, pour récupérer les personnes, nous allons appeler la méthode `findAll` de notre repository, pour l'ORM, cela revient à faire une requête pour récuprer tous les enregistrement de la table :

```
    @GetMapping
    public String persons(Model model){
        List<Person> persons = repository.findAll();

        model.addAttribute("persons", persons);
        model.addAttribute("person", new Person());

        return "persons";
    }
```

Ensuite, on peut ajouter notre endpoint d'insertion, qui redirige ensuite vers l'endpoint d'affichage :

```
    @PostMapping
    public RedirectView createPerson(@ModelAttribute Person person){
        this.repository.save(person);

        return new RedirectView("persons");
    }
```

On ajoute la `Person` récupérée depuis le formulaire au repository avec la méthode `save`.

> Vous pouvez regarder dans la console les requêtes effectuées par Hibernate sur la base de données, qui correspondent à nos appels

On va ensuite ajouter une vue pour voir une seule personne, avec un lien dans la liste. Créons d'abord notre endpoint :

```
    @GetMapping("{id}")
    public String personDetails(@PathVariable("id") String id, Model model){
        Person person = repository.getById(id);

        model.addAttribute("person", person);

        return "person";
    }
```

On utilise un paramètre de route pour récupérer l'identifiant, ainsi que la méthode `getById` du repository pour récupérer l'enregistrement.

On retourne ensuite la vue suivante :

```
<span th:text="${person.firstName}"></span> <span th:text="${person.lastName}"></span>
```

Enfin on modifie la vue d'affichage pour ajouter un lien :

```
<div th:each="person: ${persons}">
    <a th:href="'/persons/' + ${person.id}"><span  th:text="${person.firstName}"></span> <span
th:text="${person.lastName}"></span></a>
</div>
```

Relations

Les relations permettent de faire des liens entre les entités. Il existe quatres types de relations, définies par des annotations :

- `@ManyToOne` : plusieurs instances de cette classe entité sont en relation avec une unique instance d'une autre (ex: Pages d'un livre - plusieurs pages sont reliées à un unique livre)
- `OneToMany` : une unique instance de cette classe entité est en relations avec plusieurs instances d'une autre (ex: Livre qui contient des pages - un unique livre est relié à plusieurs pages)
- `@ManyToMany` : plusieurs instance de cette classe entité sont en relations avec plusieurs isntance d'une autre (ex: Classes et Professeurs - Les professeurs ont plusieurs classes et les classes ont plusieurs professeurs.
- `@OneToOne` : une unique instance de cette classe entités est reliée à une unique instance d'une autre (ex: Dircteur et Ecole - une directeur dirige une seule école et une école est dirrigée par un seul directeur).

Exemple :

```
@Entity
public class Todo {
  ... autres propriétés ...

  @ManyToOne
  private ApplicationUser owner;

  ... getters & setters ....
}
```

```
@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(generator = "uuid")
```

```
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    private String id;

  private String name;

  @OneToMany
    private Set<Todo> todos;

    ... getters & setters ...
}
```

Les annotations de relation possèdes plusieurs paramètres utiles à connaitres :

- `fetch` :
  - `FetchType.EAGER` : charge les données de la relation directement (défaut pour `@ManyToOne` et `@OneToOne`)
  - `FtechType.LAZY` : charge les données de la relation que quand le getter est appelé (défaut pour `@OneToMany` et `@ManyToMany`
- `orphanRemoval` : `true` ou `false`, détermine dans un `@OneToMany` si les enfant doivent être supprimés quand le parent est supprimé → un enfant ne peut exister sans parent.

Dockeriser une application Spring Boot

D'abord on build un JAR exécutable de notre application :

`./mvnw clean package`

Cela va vous construire un JAR exécutable dans `target/build`.

On crée ensuite le Dockerfile suivante à la racine du projet :

```
FROM openjdk:17.0-oracle
EXPOSE 8080
ADD ./target/mon-projet-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

On peut ensuite build l'image :

```
docker build -t mon-image -f Dockerfile .
```

Et enfin lancer l'image :

> Attention, si votre base de donnée est aussi un conteneur Docker tournant sur votre machine, dans l'hote de la base de donnée, notez non pas "localhost" mais l'adresse locale de votre machine pour permettre à la connexion de loopback entre vos conteneurs.

```
docker run -p 80:8080 -e DATABASE_URL=<URL JDBC de votre base de donnée> -e
DATABASE_USERNAME=root -e DATABASE_PASSWORD=monMdpTresSecret -t mon-image
```

Sources

- https://spring.io/guides/gs/serving-web-content/
- https://www.baeldung.com
- https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/http-session-param.html