

SYMPHONY



1. INSTALLATION

1.1 Avec lando

Récupérer la recipe lando pour symfony

Exécuter: `lando start`

Puis: `lando composer create-project symfony/website-skeleton mon_projet`

N.B: toutes les commandes s'exécuteront avec lando (`lando composer`, `lando console`, ...)

1.2 Avec docker

Récupérer la config de `docker-compose.yml` (ainsi que le dossier PHP et NGINX)

Exécuter: `docker-compose up -d`

Puis: `docker-compose exec php composer create-project symfony/website-skeleton mon_projet`

N.B: toutes les commandes s'exécuteront avec `docker-compose exec php` (`docker-compose exec php composer`, `docker-compose exec php php bin/console`, ...)

1.3 Avec le cli de symfony

Il s'agit d'installer tout l'environnement en local sur sa machine, il faut installer PHP8, Composer, NPM ou Yarn et Symfony Cli => Se reporter aux différentes doc pour l'installation sur vos machine

2. INITIALISATION DU PROJET

Pour le cours on utilisera la méthode lando.

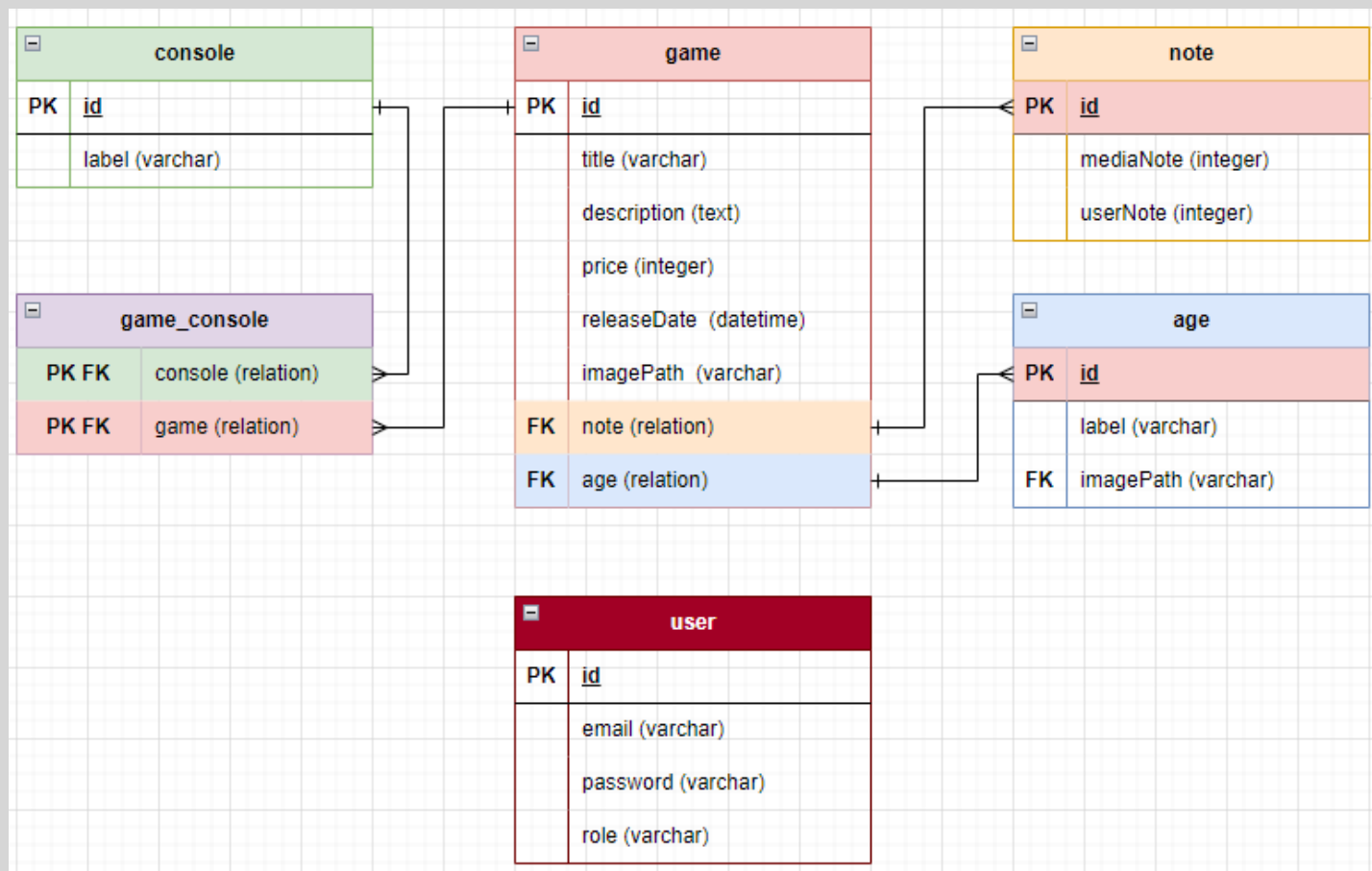
Une fois le projet installé, la première chose à faire est de configurer notre .env pour la connexion à la BDD.

```
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8&charset=utf8mb4"
#DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
DATABASE_URL="mysql://julien:admin@database:3306/symfony"
```

Maintenant l'objectif va être de définir toutes nos entités suivant le diagramme de BDD

N.B: Un diagramme de BDD est OBLIGATOIRE pour concevoir ses entités proprement.

3. LES ENTITES



On commence toujours par les tables qui n'ont pas de relation.

Commande pour créer une entité: **lando console make:entity**

Ordre de création:

- Console
- Note
- Age
- Game

Pour l'instant, on laisse la table User de coté, on verra après comment la créer.

Les principaux type de propriétés que l'on trouvera dans notre projet symfony

- String
- Intégrer
- Boolean
- Datetime
- Relation

N.B: Le nom des entités prennent toujours une majuscule, le nom des propriétés en minuscule et en camelCase (exemple: createdAt) et ne pas ajouter _id pour les relation, il s'en chargera tout seul (sinon on aura _id en double)

Une fois les entités créées on peut passer la commande

lando console make:migration

Cette commande va nous générer un fichier dans le dossier migration (ce fichier contient ni plus ni moi que du script sql pour effectuer les modification à apporter)

ATTENTION: ne pas modifier ou effacer de migration au risque de ne plus fonctionner, c'est un historique de vos modification en BDD

Maintenant s'il on veut envoyer les migrations à notre BDD il suffit de passer la commande

lando console d:m:m (pour doctrine:migrations:migrate).

A ce moment la vous verrez les modification sur votre BDD.

Création de l'entité User:

lando console make:user

On répondra à une série de questions

Le nom de l'entité, si on veut la stocker dans la BDD, par quoi on veut reconnaître notre user et si on veut encrypter le MDP

```
PS C:\Users\jul89\Desktop\jeu-video-symfony> php bin/console make:user
```

```
The name of the security user class (e.g. User) [User]:
```

```
>
```

```
Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
```

```
>
```

```
Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
```

```
>
```

```
Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed by some other system (e.g. a single sign-on server).
```

```
Does this app need to hash/check user passwords? (yes/no) [yes]:
```

```
>
```

```
created: src/Entity/User.php
```

```
created: src/Repository/UserRepository.php
```

```
updated: src/Entity/User.php
```

```
updated: config/packages/security.yaml
```

On peut repasser nos 2 commandes: `lando console make:migration` et `lando console d:m:m`

4. LES FIXTURES

Les fixtures sont des données factices ou pas que l'on crée pour alimenter notre base de données. Nous allons ajouter 2 nouveaux bundles pour créer nos fixtures

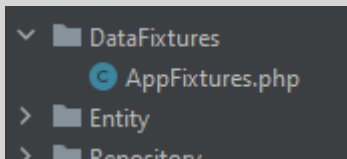
Pour créer les fixtures:

lando composer require --dev doctrine/doctrine-fixtures-bundle

Pour générer des fausses données :

lando composer require fzaninotto/faker

Un nouveau dossier DataFixtures a dû apparaître dans l'application



C'est dans ce fichier que nous allons créer nos fixtures.

Dans AppFixtures.php nous allons créer un utilisateur

```
class AppFixtures extends Fixture
{
    //propriété pour encoder le MDP
    private $encoder;

    public function __construct(UserPasswordHasherInterface $encoder)
    {
        $this->encoder = $encoder;
    }

    public function load(ObjectManager $manager): void
    {
        //on instancie la classe
        $user = new User();
        //On set les valeurs que l'on souhaite ajouter
        $user->setEmail( email: "admin@admin.com")
            ->setPassword($this->encoder->hashPassword($user, plainPassword: 'admin'))
            ->setRoles(['ROLE_ADMIN']);

        //persist permet de préparer les datas
        $manager->persist($user);
        //flush permet d'exécuter
        $manager->flush();
    }
}
```

On crée un nouveau fichier dans fixtures, GameFixtures.php pour créer tout ce qui concerne les jeux

On commence par les consoles

```
1 usage
public function loadConsole(ObjectManager $manager)
{
    //On définit notre tableau de valeurs pour console
    $consoleArray = ['PS4', 'PS5', '360', 'Xbox Series', 'ONE', 'Switch', 'PC'];
    //on boucle notre tableau et on affiche les valeurs
    foreach ($consoleArray as $key => $cons) {
        $console = new Console();
        $console->setLabel($cons);
        $manager->persist($console);
        //on définit une référence pour pouvoir faire nos relations avec console
        $this->addReference( name: "console-" . $key + 1, $console);
    }
}
```

On continue avec age

```
public function loadAge(ObjectManager $manager)
{
    //On définit notre tableau de valeurs pour console
    $ageArray = [
        ["key" => 1, "label" => "3", "imagePath" => "pegi3.png"],
        ["key" => 2, "label" => "7", "imagePath" => "pegi7.png"],
        ["key" => 3, "label" => "12", "imagePath" => "pegi12.png"],
        ["key" => 4, "label" => "16", "imagePath" => "pegi16.png"],
        ["key" => 5, "label" => "18", "imagePath" => "pegi18.png"],
    ];
    //on boucle notre tableau et on affiche les valeurs
    foreach ($ageArray as $key => $value) {
        $age = new Age();
        $age->setLabel($value['label']);
        $age->setImagePath($value['imagePath']);
        $manager->persist($age);
        //on définit une référence pour pouvoir faire nos relations avec age
        $this->addReference( name: "age-" . $value['key'], $age);
    }
}
```

On continue avec note

```
public function loadNote(ObjectManager $manager)
{
    //on crée une boucle for pour générer des notes comprises entre 10 et 20
    for ($i = 1; $i <= 15; $i++) {
        $note = new Note();
        $note->setMediaNote(rand(10, 20));
        $note->setUserNote(rand(10, 20));
        $manager->persist($note);
        //on définit une référence pour pouvoir faire nos relations avec age
        $this->addReference( name: "note-" . $i, $note);
    }
}
```

On fini avec game, car il contient toutes les relations

On crée notre tableau de data que l'on souhaite avoir et on générera des fausses data pour certaines valeurs

```
public function loadGame(ObjectManager $manager)
{
    $gameArray = [
        ["note" => 1, "age" => 1, "title" => "Animal Crossing : New Horizons", "imagePath" => "animal-crossing.jpg", "console" => [6]],
        ["note" => 2, "age" => 5, "title" => "Call of Duty : Modern Warfare 2", "imagePath" => "call-of-duty.jpg", "console" => [1,2,4,5,7]],
        ["note" => 3, "age" => 1, "title" => "Fall Guys : Ultimate Knockout", "imagePath" => "fall-guys.jpg", "console" => [1,2,4,5,7]],
        ["note" => 4, "age" => 1, "title" => "FIFA 23", "imagePath" => "fifa-23.jpg", "console" => [1,2,4,5,7]],
        ["note" => 5, "age" => 5, "title" => "Grand Theft Auto V", "imagePath" => "gta-v.jpg", "console" => [1,2,3,4,5,7]],
        ["note" => 6, "age" => 2, "title" => "Human Fall Flat", "imagePath" => "Human-Fall-Flat.jpg", "console" => [1,2,4,5,7]],
        ["note" => 7, "age" => 1, "title" => "Mario Kart 8 Deluxe", "imagePath" => "mario-kart-8.jpg", "console" => [6]],
        ["note" => 8, "age" => 1, "title" => "Super Mario Odyssey", "imagePath" => "mario-odyssey.jpg", "console" => [6]],
        ["note" => 9, "age" => 2, "title" => "Minecraft", "imagePath" => "minecraft.jpg", "console" => [1,2,3,4,5,7]],
        ["note" => 10, "age" => 2, "title" => "Légendes Pokémon: Arceus", "imagePath" => "pokemon.jpg", "console" => [6]],
        ["note" => 11, "age" => 4, "title" => "PlayerUnknown's Battlegrounds", "imagePath" => "PUBG-Battlegrounds.jpg", "console" => [1,5,7]],
        ["note" => 12, "age" => 5, "title" => "Red Dead Redemption II", "imagePath" => "red-dead-redemption.jpg", "console" => [1,5,7]],
        ["note" => 13, "age" => 5, "title" => "The Elder Scrolls V : Skyrim", "imagePath" => "The-Elder-Scrolls-Skyrim.jpg", "console" => [1,2,3,4,5,7]],
        ["note" => 14, "age" => 3, "title" => "The Legend of Zelda : Breath of the Wild", "imagePath" => "zelda.jpg", "console" => [6]],
    ];
}
```

Pour utiliser faker il va falloir le déclarer dans le constructeur

```
class GameFixtures extends Fixture
{
    private \Faker\Generator $faker;

    public function __construct()
    {
        $this->faker = Factory::create();
    }

    public function load(ObjectManager $manager): void
    {
```

Ensuite on affectes les datas avec le tableau crée et des fausses datas de faker

```
//on crée une boucle
foreach ($gameArray as $value) {
    $game = new Game();
    $game->setTitle($value['title']);
    //fausse data de description
    $game->setDescription(implode( separator: ', ', $this->faker->words( nb: 8)));
    $game->setImagePath($value['imagePath']);
    //fausse data de date de création
    $game->setReleaseDate($this->faker->dateTimeBetween('2016-01-01 00:00:00', '2022-12-31 23:59:59'));
    $game->setPrice(rand(0, 6000));
    //on rappelle nos référence pour effectuer les relations
    $game->setNote($this->getReference( name: 'note-' . $value['note']));
    $game->setAge($this->getReference( name: 'age-' . $value['age']));
    //on boucle notre tableau de console pour notre relation many to many
    foreach ($value['console'] as $console){
        $game->addConsoles($this->getReference( name: 'console-' . $console));
    }
    $manager->persist($game);
}
```

Nous reste plus qu'à appeler nos fonctions que l'on vient de créer dans la fonction load()

```
public function load(ObjectManager $manager): void
{
    $this->loadConsole($manager);
    $this->loadAge($manager);
    $this->loadNote($manager);
    $this->loadGame($manager);

    $manager->flush();
}
```

N.B: l'ordre des fonction a un rôle très important (exemple: Game est à la fin car il a besoin de Console, de Age et de Note)

Nous reste plus qu'à passer la commande : **lando console d:f:l** (doctrine:fixtures:load)

5. CONTROLEURS ET VUES

Les controllers, comme pour un modèle MVC standard sert d'aiguillage entre nos repositories et nos vues. Ils servent aussi pour la création de routes.

Dans Contollers on va créer un nouveau fichier, HomeController.php

```
class HomeController extends AbstractController
{
    //on déclare la route
    #[Route('/', name: 'index')]
    //on crée notre méthode
    public function index(GameRepository $gameRepository, Request $request): Response
    {
        // on déclare nos variables
        $title = "Tous les jeux";
        //on récupère les datas du repository
        $games = $gameRepository->findAll();
        //on retourne le rendu de la vue et on lui passe les datas
        return $this->render(view: 'home/index.html.twig', [
            'games' => $games,
            'title' => $title
        ]);
    }
}
```

Puisque nous voulons faire un rendu de la vue home/index.html.twig on va devoir créer le dossier home (dans templates) et le fichier home.html.twig

```
{# on récupère la base #}
{% extends 'base.html.twig' %}
{# on redéfinit le titre de la page (l'onglet) #}
{% block title %}Liste des jeux vidéos{% endblock %}
{# on ouvre le block body qui correspond à l'affichage du contenu #}
{% block body %}
    {# ici on peut récupérer title car le controller nous l'a fournit #}
    <h2>{{ title }}</h2>
    {#   grace à games envoyé par le controller
    on peut boucler dessus pour afficher les datas #}
    {% for game in games %}
        <div class="card mb-2">
            <div class="card-body">
                <h4 class="card-title">{{ game.title }}</h4>
                <p class="card-text">{{ game.description }}</p>
            </div>
        </div>
    {% endfor %}
{% endblock %}
```

INSTALLATION DE BOOTSTRAP

On utilisera webpack:

Utiliser la commande : **composer require symfony/webpack-encore-bundle**

Puis: npm install

Renommer le fichier app.css dans le dossier assets/styles/ en app.sass et modifier dans le fichier app.js :

```
// any CSS you import will output into a single css file (app.css in this case)
import './styles/app.sass';
```

Dans webpack.config.json on va décommenter « enableSassLoader»

Puis on passe la commande: **npm install sass-loader node-sass --save-dev**


INSTALLATION DE BOOTSTRAP (SUITE)

Ensuite on modifie notre `base.html.twig`

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>{% block title %}Welcome!{% endblock %}</title>
  {% block stylesheets %}{{ encore_entry_link_tags('app') }}{% endblock %}
</head>
<body>
  {% block body %}{% endblock %}
  {% block javascripts %}{{ encore_entry_script_tags('app') }}{% endblock %}
</body>
</html>
```

On passe la commande : **npm run build**

Et on installe bootstrap: **npm install bootstrap**
dans `app.js`, on importe bootstrap



```
// start the Stimulus application
import './bootstrap';
```

On repasse la commande : **npm run build**

On va améliorer notre fichier home.html.twig

```
{% block body %}
    {# ici on peut récupérer title car le controller nous l'a fournit #}
    <h2>{{ title }}</h2>
    <div class="d-flex flex-wrap col-sm-10 justify-content-center mt-5">

        {# grace à games envoyé par le controller on peut boucler dessus pour afficher les datas #}
        {% for game in games %}
            <div class="card m-2 d-flex flex-column" style="width: 18rem;">
                
                <div class="card-body">
                    <h5 class="card-title">{{ game.title }} </h5>
                    {# formatage de game.price ~ : sert à concaténer et | à utiliser une fonction#}
                    <p class="card-text">{{ game.price != 0 ? (game.price/100)|number_format(2, ',', '.') ~ '€' : 'GRATUIT' }}</p>
                    <a href="" class="btn btn-primary">Voir détail</a>
                </div>
            </div>
        {% endfor %}
    </div>
{% endblock %}
```

Et dans base.html.twig, on ajoute des classes bootstrap à body

```
<body class="d-flex flex-column align-items-center">
{% block body %}{% endblock %}
{% block javascripts %}{ { encore_entry_script_tags('app') } }{% endblock %}
</body>
</html>
```

Nous allons maintenant s'occuper du détail d'un jeu:

- 1: créer la route
- 2: créer le controller
- 3: créer le requête pour récupérer les bonnes infos dans le repository
- 4: fournir les infos au controller
- 5: appeler la vue depuis le controller

Création de la route et du controlleur pour afficher le détail d'un jeu

Dans HomeController.php

```
//on déclare la route détail, on donne le paramètre id entre {}  
no usages  
#[Route('/detail/{id}', name: 'detail')]  
public function gameById(GameRepository $gameRepository, int $id): Response  
{  
  
    //ici, on appelle la vue et on lui passera ses données nécessaires  
    return $this->render( view: 'home/detail.html.twig', [  
    ]);  
}
```

Création de la requête pour récupérer les infos du détail d'un jeu

Dans GameRepository.php

Le DQL est du SQL simplifié

On choisit l'entité de base

Ici: App\Entity\Game

Les jointures se font sur les propriétés en relation

La table Age est jointe grâce à la propriété age
de Game

Idem pour Note qui est lié grâce à note de Game

```
/**
 * @throws NonUniqueResultException
 * Création de la requête pour récupérer les infos d'un jeu
 * On utilisera du DQL (doctrine query language)
 */
1 usage
public function getGamesWithInfo($id): mixed
{
    $entityManager = $this->getEntityManager();
    $query = $entityManager->createQuery(
        dql: 'SELECT
            g.title,
            g.id,
            g.description,
            g.price,
            g.releaseDate,
            g.imagePath,
            a.label,
            a.imagePath as pegi,
            n.userNote,
            n.mediaNote
        FROM App\Entity\Game g
        JOIN g.age a
        JOIN g.note n
        WHERE g.id = :id'
    )->setParameter( key: 'id', $id);

    return $query->getOneOrNullResult();
}
```


On retourne dans le controlleur HomeController.php pour récupérer la méthode que l'on vient de créer

```
//on déclare la route détail
no usages
#[Route('/detail/{id}', name: 'detail')]
public function gameById(GameRepository $gameRepository, int $id)
{
    //on crée une variable qui récupère le résultat de notre requête
    $game = $gameRepository->getGamesWithInfo($id);

    //dans le rendu de notre vue, on envoie les datas sous forme de tableau
    return $this->render( view: 'home/detail.html.twig', [
        'game' => $game,
    ]);
}
```

Il nous reste plus qu'à créer la vue, dans templates/home on crée un nouveau fichier: detail.html.twig

```
{# on récupère la base #}
{% extends 'base.html.twig' %}
{# on redéfinit le titre de la page (l'onglet) #}
{% block title %}{ { game.title } }{% endblock %}
{# on ouvre le block body qui correspond à l'affichage du contenu #}
{% block body %}
    <h2>{ { game.title } }</h2>
    <div class="d-flex flex-wrap col-sm-10 justify-content-center mt-5">
        {# on peut redefinir une variable avec le mot clé set #}
        {% set date = game.releaseDate|date("d/m/Y") %}

        <div class="card my-3" style="max-width: 90%;">
            <div class="row g-0">
                <div class="col-md-4">
                    
                </div>
                <div class="col-md-8">
                    <div class="card-body">
                        <h5 class="card-title text-primary">{ { game.title } }</h5>
                    </div>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

detail.html.twig
(suite)

```
<p class="card-text pt-2"><span class="fw-bold">Synopsis:</span> {{ game.description }}</p>
<p class="card-text"><small class="text-muted">Date de sortie: <span
    class="text-primary fw-bold">{{ date }}</span></small></p>
<p class="card-text pt-2">
    
    age: {{ game.label }}+
</p>

<div class="d-flex flex-row justify-content-around ">
    <div class="d-flex align-items-baseline">
        <p class="text-warning fs-4">#9733;</p>
        <p class="card-text"> Avis presse: <span
            class="text-primary fw-bold">{{ game.mediaNote }}/20</span></p>
    </div>
    <div class="d-flex align-items-baseline">
        <p class="text-warning fs-4">#9733;</p>
        <p class="card-text"> Avis utilisateur: <span
            class="text-primary fw-bold">{{ game.userNote }}/20</span></p>
    </div>
</div>
</div>
</div>
</div>

</div>
{% endblock %}
```

Nous avons bien le détail de notre jeu mais il nous manque l'affichage des différentes consoles par jeu.
Nous allons créer une seconde requête pour récupérer la liste des consoles par jeu puis on les donnera à la vue

```
//requête qui récupère la liste des consoles par jeu  
no usages  
public function getConsoleByGame($id)  
{  
    $entityManager = $this->getEntityManager();  
    $query = $entityManager->createQuery(  
        dql: 'SELECT c.label, c.id  
            FROM App\Entity\Game g  
            JOIN g.consoles c  
            WHERE g.id = :id'  
    )->setParameter( key: 'id', $id);  
  
    return $query->getResult();  
}
```

On récupère le résultat de la requête dans notre contrôleur

```
//on déclare la route détail
no usages
#[Route('/detail/{id}', name: 'detail')]
public function gameById(GameRepository $gameRepository, int $id): Response
{
    $game = $gameRepository->getGamesWithInfo($id);
    //on récupère la liste des consoles par jeu
    $consoles = $gameRepository->getConsoleByGame($id);

    return $this->render(view: 'home/detail.html.twig', [
        'game' => $game,
        //On envoi les data des consoles
        'consoles' => $consoles,
    ]);
}
```

Dans detail.html.twig

On ajoute la liste des consoles

```
<div class="card-body">
  <h5 class="card-title text-primary">{{ game.title }}</h5>

  {# on boucle sur consoles pour récupérer la liste#}
  <p class="card-text">
    {% for console in consoles %}
      <span class="badge rounded-pill bg-primary">{{ console.label }}</span>
    {% endfor %}
  </p>

  <p class="card-text pt-2"><span class="fw-bold">Synopsis:</span> {{ game.description }}</p>
  <p class="card-text"><small class="text-muted">Date de sortie: <span
```

6. NAVBAR

Nous allons nous occuper de la navbar , on crée un nouveau dossier « base » dans template.

Dans le dossier base nous allons créer un fichier nav.html.twig

Template récupérer sur bootstrap

```
<nav class="navbar navbar-expand-lg bg-light py-4 col-sm-12">
  <div class="container-fluid flex justify-content-around">
    <div>
      <a class="navbar-brand" href="/">
        
      </a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNavDropdown"
        aria-controls="navbarNavDropdown" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
    </div>
    <div>
      <div class="collapse navbar-collapse" id="navbarNavDropdown">
        <ul class="navbar-nav">
          <li class="nav-item">
            <a class="nav-link active text-dark fs-5" aria-current="page" href="/">Accueil</a>
          </li>
          <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle text-dark fs-5" href="#" role="button"
              data-bs-toggle="dropdown"
              aria-expanded="false">
              Par console
            </a>
            <ul class="dropdown-menu">
              <li><a class="dropdown-item" href="#">console 1</a></li>
              <li><a class="dropdown-item" href="#">console 2</a></li>
              <li><a class="dropdown-item" href="#">console 3</a></li>
            </ul>
          </li>
        </ul>
      </div>
    </div>
  </div>
</nav>
```

Nous allons maintenant appeler notre navbar dans le fichier base.html.twig

```
<body class="d-flex flex-column align-items-center">

    {# on importe la navbar #}
    {% block menu %}
        {% include 'base/nav.html.twig' %}
    {% endblock %}

    {#ici le block body#}
    {% block body %}{% endblock %}
```

Notre navbar s'affiche bien sur la page d'accueil et page détail, maintenant on aimerait bien afficher la vraie liste de consoles disponible avec son nombre de jeux associé (ex : PS5 (9))

Dans un premier temps, on va créer la requête dans GameRepository.php

Dans GameRepository.php

```
//requête qui récupère toutes les consoles avec le nombre de jeux associés  
2 usages  
public function getCountGameByConsole()  
{  
    $entityManager = $this->getEntityManager();  
    $query = $entityManager->createQuery(  
        dql: 'SELECT c.label, c.id, COUNT(g.id) AS total  
            FROM App\Entity\Game g  
            JOIN g.consoles c  
            GROUP BY c.id'  
    );  
  
    return $query->getResult();  
}
```

Comment maintenant passer les paramètres à la vue?

On pourrait faire la même manière que pour les autres en ajoutant un troisième tableau

```
#[Route('/detail/{id}', name: 'detail')]
public function gameById(GameRepository $gameRepository, int $id): Response
{
    $game = $gameRepository->getGamesWithInfo($id);
    //on récupère la liste des consoles par jeu
    $consoles = $gameRepository->getConsoleByGame($id);

    $menuItems = $gameRepository->getCountGameByConsole();

    return $this->render( view: 'home/detail.html.twig', [
        'game' => $game,
        'consoles' => $consoles,
        'menu_items' => $menuItems
    ]);
}
```

Mais on devrait dans chaque controller appeler cette méthode pour fournir a notre navbar les mêmes infos, ce qui n'est pas très maintenable dans la durée

Pour remédier à ce problème, nous allons passer par une extension Twig

On tape la commande suivant : `php bin/console make:twig-extension NavExtension`

Ca va nous générer un nouveau dossier Twig à la racine avec 2 dossiers l'intérieur: Extension et Runtime avec chacun un fichier php

Dans NavExtension.php on peut voir 2 méthode générer automatiquement

```
public function getFilters(): array //permet de générer un filtre de rendu (mettre du text en uppercase)
{
    return [
        // If your filter generates SAFE HTML, you should add a third
        // parameter: ['is_safe' => ['html']]
        // Reference: https://twig.symfony.com/doc/3.x/advanced.html#automatic-escaping
        new TwigFilter( name: 'filter_name', [TestExtensionRuntime::class, 'doSomething']),
    ];
}
```

no usages

```
public function getFunctions(): array // permet de récupérer une fonction directement dans le rendu
{
    return [
        new TwigFunction( name: 'function_name', [TestExtensionRuntime::class, 'doSomething']),
    ];
}
```

Dans notre cas nous allons utiliser la fonction twig et non le filtre, car on souhaite récupérer des informations en BDD, on va commenter la partie getFilter et nous allons coder getFunctions

Le principe est de donner un nom a notre fonction (ici: menu_items)

Et de lui passer son callable: [nom-de-class, méthode-de-la-class]

```
//nous allons créer une fonction twig qui va se charger de récupérer les infos en BDD
no usages
public function getFunctions(): array
{
    return [
        new TwigFunction( name: 'menu_items', [NavExtensionRuntime::class, 'menuItems']),
    ];
}
```

Dans NavExtensionRuntime, c'est ici que nous allons y mettre la mécanique, dans la fonction menuItems()

```
class NavExtensionRuntime implements RuntimeExtensionInterface
{
    //on déclare une variable privé pour stocker notre instance
    2 usages
    private $gameRepository;

    //dans le construct on instancie notre GameRepository
    public function __construct(GameRepository $gameRepository)
    {
        // Inject dependencies if needed
        $this->gameRepository = $gameRepository;
    }

    //on crée la méthode qui recupère les infos de la navbar
    2 usages
    public function menuItems( )
    {
        //on appelle la query depuis GameRepository
        return $this->gameRepository->getCountGameByConsole();
    }
}
```

Nous reste plus qu'à appeler notre fonction Twig dans notre rendu nav.html.twig

```
</a>
<ul class="dropdown-menu">
    {# On récupère ici notre fonction twig pour récupérer les éléments #}
    {% for nav in menu_items() %}
        <li><a class="dropdown-item" href="/console/{{ nav.id }}">{{ nav.label }} ({{ nav.total }})</a></li>
    {% endfor %}
</ul>
<li>
```

Voilà, maintenant à chaque fois que l'on voudra afficher la navbar, c'est elle-même qui se chargera d'aller récupérer ses informations pour construire son menu

EXERCICE: lorsqu'on clique sur une console dans la navbar, je veux le rendu..

Correction:

Créer la route et la méthode dans HomeController

```
//on déclare la route
no usages
#[Route('/console/{id}', name: 'gameByConsole')]
//on crée notre méthode
public function gameByConsole(GameRepository $gameRepository, int $id): Response
{
    //on récupère les datas du repository
    $games = $gameRepository->getGamesByConsole($id);
    //on retourne le rendu de la vue et on lui passe les datas
    return $this->render(view: 'home/index.html.twig', [
        'games' => $games,
    ]);
}
```

Correction:

Créer la requête dans GameRepository

```
public function getGamesByConsole($consoleId): array
{
    $entityManager = $this->getEntityManager();
    $query = $entityManager->createQuery(
        dql: 'SELECT c.label, g.id, g.imagePath, g.title,g.price
              FROM App\Entity\Game g
              JOIN g.consoles c
              WHERE c.id = :id'
    )->setParameter(key: 'id', $consoleId);

    return $query->getResult();
}
```

On peut réutiliser la vue home/index.html.twig car on a le même rendu

7. AUTHENTICATION

Nous allons utiliser la commande `php bin/console make:auth`, cette commande va nous générer tout le système d'authentification

Il va nous poser des questions, savoir si on veut générer le formulaire, le nom du service de sécurité, le nom du controller,...

Une fois créé, ça va nous créer plusieurs fichiers et modifier certains. On va modifier `security/login.html.twig`

```
{% if app.user %}
    <div class="mb-3">
        Vous êtes enregistré en tant que {{ app.user.userIdentifier }}, <a href="{{ path('app_logout') }}">Logout</a>
    </div>
{% endif %}

<h1 class="h3 my-5 font-weight-normal">Connectez vous</h1>
<label for="inputEmail">Email</label>
<input type="email" value="{{ last_username }}" name="email" id="inputEmail" class="form-control" autocomplete="email" required autofocus>
<label for="inputPassword">Mot de passe</label>
<input type="password" name="password" id="inputPassword" class="form-control" autocomplete="current-password" required>

<input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">
<button class="btn btn-lg btn-primary my-3 p-1 form-control" type="submit">
    Se connecter
</button>
</form>
{% endblock %}
```

Maintenant, nous allons créer un nouveau controller, AdminController.php, ou on aura toutes nos routes d'administration uniquement accessible aux ayants droits.

On tape la commande : `php bin/console make:controller AdminController`

On va de suite modifier le controller:

On préfixe le controller par /admin

```
#[Route("/admin")] //ici tout le controller sera préfixé par /admin
class AdminController extends AbstractController
{
    //pour accéder à index la route sera /admin/dashboard
    #[Route('/dashboard', name: 'app_admin')]
    public function index(): Response
    {
        return $this->render( view: 'admin/index.html.twig', [
            'controller_name' => 'AdminController',
        ]);
    }
}
```

L'objectif va être de rendre accessible la route /admin uniquement aux utilisateur ayant un rôle « admin »
Dans config/packages/security.yaml

```
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    # ici, seul les role admin pourront accéder à la route /admin
    - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
```

Maintenant si on veut accéder à la route directement par l'URL, on est redirigé sur le formulaire de login si on est pas connecté et sur access denied si on est connecté avec un rôle différent d'admin

Dans template, on crée un nouveau dossier game et on ajoute un fichier index.html.twig (qui sera la page d'accueil de notre dashboard)

```
{% extends 'base.html.twig' %}

{% block title %}Liste de jeux{% endblock %}

{% block body %}
    <h1>Liste des jeux</h1>

    <table class="table">
        <thead>
            <tr>
                <th>Id</th>
                <th>Titre</th>
                <th>Description</th>
                <th>Prix</th>
                <th>Date de sortie</th>
                <th>Nom de l'image</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            {% for game in games %}
                <tr>
```

index.html.twig (suite)

```
<tbody>
{% for game in games %}
    <tr>
        <td>{{ game.id }}</td>
        <td>{{ game.title }}</td>
        <td>{{ game.description }}</td>
        <td>{{ game.price != 0 ? (game.price/100)|number_format(2, ',', '.') ~ '€' : 'GRATUIT' }}</td>
        <td>{{ game.releaseDate ? game.releaseDate|date('d-m-Y') : 'inconnu' }}</td>
        <td>{{ game.imagePath }}</td>
        <td>
            <a href="{{ path('app_game_show', {'id': game.id}) }}" class="btn btn-primary">Voir</a>
            <a href="{{ path('app_game_edit', {'id': game.id}) }}" class="btn btn-warning">Editer</a>
        </td>
    </tr>
{% else %}
    <tr>
        <td colspan="7">no records found</td>
    </tr>
{% endfor %}
</tbody>
</table>

<a href="{{ path('app_game_new') }}" class="btn btn-success">Créer un nouveau</a>
{% endblock %}
```

Nous reste qu'à appeler cette vue dans notre controlleur AdminController.php

```
#[Route("/admin")] //ici tout le controller sera préfixé par /admin
class AdminController extends AbstractController
{
    //pour accéder à index la route sera /admin/dashboard
    // julien.linard *
    #[Route('/dashboard', name: 'app_admin')]
    public function index(GameRepository $gameRepository): Response
    {
        return $this->render( view: 'game/index.html.twig', [
            'games' => $gameRepository->findAll(),
        ]);
    }
}
```

Dans templates/game, on crée un nouveau fichier show.html.twig

```
{% extends 'base.html.twig' %}

{% block title %}Game{% endblock %}

{% block body %}
    <h1>{{ game.title }}</h1>

    {# on peut redéfinir une variable avec le mot clé set #}
    {% set date = game.releaseDate|date("d/m/Y") %}

    <div class="card my-3" style="...">
        <div class="row g-0">
            <div class="col-md-4">
                
            </div>
            <div class="col-md-8">
                <div class="card-body">
                    <h5 class="card-title text-primary">{{ game.title }}</h5>

                    {# on boucle sur consoles pour récupérer la liste #}
                    <p class="card-text">
                        {% for console in consoles %}
                            <span class="badge rounded-pill bg-primary">{{ console.label }}</span>
                        {% endfor %}
                    </p>
                </div>
            </div>
        </div>
    </div>

```

Dans show.html.twig (suite)

```
<p class="card-text pt-2"><span class="fw-bold">Synopsis:</span> {{ game.description }}</p>
<p class="card-text"><small class="text-muted">Date de sortie: <span
    class="text-primary fw-bold">{{ date }}</span></small></p>
<p class="card-text pt-2">
    
    age: {{ game.label }}+
</p>

<div class="d-flex flex-row justify-content-around ">
    <div class="d-flex align-items-baseline">
        <p class="text-warning fs-4">&#9733;</p>
        <p class="card-text"> Avis presse: <span
            class="text-primary fw-bold">{{ game.mediaNote }}/20</span></p>
    </div>
    <div class="d-flex align-items-baseline">
        <p class="text-warning fs-4">&#9733;</p>
        <p class="card-text"> Avis utilisateur: <span
            class="text-primary fw-bold">{{ game.userNote }}/20</span></p>
    </div>
</div>
</div>
</div>
</div>
<div class="d-flex flex-row">
```


Dans show.html.twig (suite)

```
</div>
{# partie avec les bouton d'actions #}
<div class="d-flex flex-row">
    {# bouton pour revenir à la liste #}
    <div>
        <a class="btn btn-primary m-2" href="{{ path('app_game_index') }}">Retour à la liste</a>
    </div>
    {# bouton pour éditer le jeu #}
    <div>
        <a class="btn btn-warning m-2" href="{{ path('app_game_edit', {'id': game.id}) }}">Modifier le jeu</a>
    </div>
    {# import du bouton pour supprimer le jeu #}
    <div>
        {{ include('game/_delete_form.html.twig') }}
    </div>
</div>
{% endblock %}
```

Dans game/_delete_form.html.twig (qu'il faut créer)

```
{#formulaire de suppression #}
<form method="post" action="{{ path('app_game_delete', {'id': game.id}) }}"
    onsubmit="return confirm('Êtes-vous certain de vouloir supprimer ce jeu?');">
    <input type="hidden" name="_token" value="{{ csrf_token('delete' ~ game.id) }}">
    <button class="btn btn-danger m-2">Supprimer</button>
</form>
```

On définit sa méthode dans GameController.php

```
//méthode de suppression d'un jeu
* julien.linard *
#[Route('/{id}', name: 'app_game_delete', methods: ['POST'])]
public function delete(Request $request, Game $game, GameRepository $gameRepository): Response
{
    if ($this->isCsrfTokenValid( id: 'delete' . $game->getId(), $request->request->get( key: '_token' ))) {
        $gameRepository->remove($game, flush: true);
    }

    return $this->redirectToRoute( route: 'app_game_index', [], status: Response::HTTP_SEE_OTHER);
}
```

On va maintenant s'occuper de créer un formulaire qui servira à créer un jeu et on pourra aussi l'utiliser pour la modification d'un jeu

Dans src, on va créer un nouveau dossier Form et à l'intérieur on crée un nouveau fichier GameType.php

Dans GameType.php, nous allons créer notre formulaire

```
class GameType extends AbstractType
{
    no usages  julien.linard *
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            //Champ input pour le titre
            ->add( child: 'title', type: TextType::class, [
                'label' => 'Title',
                'attr' => ['class' => 'form-control mb-3'],
            ])
            //Champ text pour la description
            ->add( child: 'description', type: TextareaType::class, [
                'label' => 'Description',
                'attr' => ['class' => 'form-control mb-3'],
            ])
            //Champ integer pour le prix
            ->add( child: 'price', type: IntegerType::class, [
                'label' => 'Prix (en cts)',
                'attr' => ['class' => 'form-control mb-3'],
            ])
            // Champ date pour la date de sortie
    }
}
```

GameType.php (suite)

```
// Champ date pour la date de sortie
->add( child: 'releaseDate', type: DateTimeType::class, [
    'label' => 'Date de sortie (jj/mm/aaaa)',
    'widget' => 'single_text',
    'html5' => false,
    'format' => 'dd/MM/yyyy',
    'attr' => [
        'class' => 'form-control mb-3 datepicker'
    ]
])

// Champ fichier pour l'upload d'image
->add( child: 'image', type: FileType::class, [
    'label' => 'Image (JPG, JPEG, PNG)',
    'mapped' => false,
    'required' => false,
    'attr' => ['class' => 'form-control mb-3'],
    'constraints' => [
        new File([
            'maxSize' => '5M',
            'mimeTypes' => [
                'image/jpeg',
                'image/png',
                'image/jpg',
            ],
            'mimeTypesMessage' => 'Please upload a valid JPG, JPEG, or PNG file',
        ])
    ]
])
])
```

GameType.php (suite)

```
)  
// ici on récupère le formulaire de note (NoteType)  
//c'est ce qu'on appelle de l'imbrication de formulaire  
->add( child: 'note', type: NoteType::class, [  
    'label' => false  
)  
// Champs de relation pour récupérer la liste de restriction
```

Pour affilier les notes à un jeu, on va devoir passer par un autre formulaire (NoteType) on crée ce fichier dans Form

```
class NoteType extends AbstractType  
{  
    no usages  julien.linard *  
    public function buildForm(FormBuilderInterface $builder, array $options)  
    {  
        $builder  
            // Champ select qui récupère la liste  
            ->add( child: 'userNote', type: ChoiceType::class, [  
                'choices' => $this->createNoteChoices(),  
                'label' => 'Note de l\'utilisateur : ',  
                'attr' => ['class' => 'form-control mb-3'],  
            ])  
            // Champ select qui récupère la liste  
            ->add( child: 'mediaNote', type: ChoiceType::class, [  
                'choices' => $this->createNoteChoices(),  
                'label' => 'Note de la presse : ',  
                'attr' => ['class' => 'form-control mb-3'],  
            ];  
    }  
}
```

NoteType.php (suite)

//méthode qui permet de générer une liste de 0 à 20

2 usages  julien.linard

private function createNoteChoices(): array

```
{  
    $choices = [];  
    for ($i = 0; $i <= 20; $i++) {  
        $choices[$i] = $i;  
    }  
    return $choices;  
}
```

no usages  julien.linard

public function configureOptions(OptionsResolver \$resolver): void

```
{  
    $resolver->setDefaults([  
        'data_class' => Note::class,  
        'attr' => ['class' => 'form'],  
    ]);  
}
```

}

On retourne dans GameType.php

```
})  
// Champs de relation pour récupérer la liste de restriction d'age  
// ici un seul choix possible  
->add( child: 'age', type: EntityType::class, [  
    'label' => 'Restriction d\'age',  
    'class' => Age::class,  
    'choice_label' => 'label',  
    'attr' => ['class' => 'form-control mb-3'],  
])  
// Champs de relation pour récupérer la liste de consoles  
// ici plusieurs choix possible  
->add( child: 'consoles', type: EntityType::class, [  
    'label' => 'Disponible sur : ',  
    'class' => Console::class,  
    'choice_label' => 'label',  
    'multiple' => true,  
    'attr' => ['class' => 'form-control mb-3'],  
]);
```

A la fin on lui donne sa classe associé

```
public function configureOptions(OptionsResolver $resolver)  
{  
    $resolver->setDefaults([  
        'data_class' => Game::class,  
        'attr' => ['class' => 'form'],  
    ]);  
}
```

Maintenant que les formulaires sont créés, il va falloir les appeler, dans templates/game on crée un nouveau fichier _form.html.twig

```
{# debut du formulaire #}  
{{ form_start(form) }}  
<div class="flex justify-content-center">  
    <div>  
        {# appelle de la fonction form_widget #}  
        {{ form_widget(form) }}  
    </div>  
    <div>  
        <button class="form-control btn btn-success mb-5">{{ button_label|default('Enregistrer') }}</button>  
    </div>  
</div>  
{#fin du formulaire#}  
{{ form_end(form) }}
```


Ensuite dans templates/game on crée un nouveau fichier new.html.twig

```
{% extends 'base.html.twig' %}

{% block title %}Création de jeu{% endblock %}

{% block body %}
    <h1>Créer un nouveau jeu</h1>
    {{ include('game/_form.html.twig') }}

    <a class="btn btn-primary" href="{{ path('app_game_index') }}">Retour à la liste</a>
{% endblock %}
```

Nous reste maintenant à déclarer notre méthode dans GameController.php

Dans GameController.php, on déclare notre route

```
//méthode pour créer un nouveau jeu
⤴ julien.linard *
#[Route('/new', name: 'app_game_new', methods: ['GET', 'POST'])]
public function new(Request $request, GameRepository $gameRepository, NoteRepository $noteRepository): Response
{
    $game = new Game();
    $form = $this->createForm( type: GameType::class, $game);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        //gestion de l'image uploadé
    }
}
```

Dans GameController.php, on s'occupe de l'upload de l'image

```
if ($form->isSubmitted() && $form->isValid()) {  
    //gestion de l'image uploadé  
    $imageFile = $form->get('image')->getData();  
    if ($imageFile) {  
        //si une image est uploadé on récupère son vrai nom  
        $originalFilename = pathinfo($imageFile->getClientOriginalName(), flags: PATHINFO_FILENAME);  
        //on génère un nouveau nom unique pour éviter d'écraser des images  
        $newFilename = $originalFilename . '-' . uniqid() . '.' . $imageFile->guessExtension();  
        try {  
            //on déplace l'image uploadé dans le bon dossier  
            $imageFile->move(  
                //games_images_directory est configuré dans config/service.yaml  
                $this->getParameter(name: 'game_images_directory'),  
                $newFilename  
            );  
        } catch (FileException $e) {  
            // handle exception if something happens during file upload  
        }  
        //on donne le nouveau nom pour la BDD  
        $game->setImagePath($newFilename);  
    }  
}
```

Dans GameController.php, on s'occupe d'enregistrer les note dans la table note et de récupérer son id, puis on enregistre le tout dans la table Game

```
}  
// On récupère les notes pour le jeu  
$userNote = $form->get('note')->get('userNote')->getData();  
$mediaNote = $form->get('note')->get('mediaNote')->getData();  
  
// On crée une nouvelle ligne dans note et on y ajoute les deux notes  
$note = new Note();  
$note->setUserNote($userNote);  
$note->setMediaNote($mediaNote);  
//on enregistre les info dans la table note  
$noteRepository->save($note, flush: true);  
  
// On récupère l'id de note pour le donner au jeu  
$game->setNote($noteRepository->find($note->getId()));  
//on enregistre dans la table game  
$gameRepository->save($game, flush: true);  
//si tout s'est bien passé on redirige sur la liste des jeux  
return $this->redirectToRoute( route: 'app_game_index', [], status: Response::HTTP_SEE_OTHER);  
}  
  
return $this->renderForm( view: 'game/new.html.twig', [  
    'game' => $game,  
    'form' => $form,  
]);  
}
```

Dans config/service.yaml on définit le path du dossier des images uploadées

```
parameters:
  game_images_directory: '%kernel.project_dir%/public/images/games'
services:
  # default configuration for services in *this* file
```

Dans templates/game on crée un nouveau fichier edit.html.twig pour la modification d'un jeu

```
{% extends 'base.html.twig' %}

{% block title %}Edit Game{% endblock %}

{% block body %}
  <h1>Modifier le jeu</h1>
  <h2> {{ game.title }}</h2>

  {{ include('game/_form.html.twig', {'button_label': 'Modifier'}) }}

  <div class="d-flex">
    <a class="btn btn-primary m-2" href="{{ path('app_game_index') }}">Retour à la liste</a>

    {{ include('game/_delete_form.html.twig') }}
  </div>
{% endblock %}
```

Et on crée la méthode dans GameController.php

```
//méthode de modification d'un jeu
@ julien.linard
#[Route('/{id}/edit', name: 'app_game_edit', methods: ['GET', 'POST'])]
public function edit(Request $request, Game $game, GameRepository $gameRepository): Response
{
    $form = $this->createForm( type: GameType::class, $game);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $gameRepository->save($game, flush: true);

        return $this->redirectToRoute( route: 'app_game_index', [], status: Response::HTTP_SEE_OTHER);
    }

    return $this->renderForm( view: 'game/edit.html.twig', [
        'game' => $game,
        'form' => $form,
    ]);
}
```

On va maintenant faire évoluer la navbar, dans templates/base/nav.html.twig

```
</div>
<div>
  <div>
    <div class="dropdown">
      <a class="nav-link dropdown-toggle " href="#" role="button" id="dropdownMenuLink"
        data-bs-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
        <i class="fa-regular fa-circle-user" style="font-size:40px"></i>
      </a>

      {% if app.user %}
      <div class="dropdown-menu" aria-labelledby="dropdownMenuLink">
        {% if is_granted('ROLE_USER') %}
          <a class="dropdown-item" href="{{ path('app_admin') }}">Dashboard</a>
        {% endif %}
        <a class="dropdown-item" href="{{ path('app_logout') }}">Déconnexion</a>
      </div>
      {% else %}
      <div class="dropdown-menu" aria-labelledby="dropdownMenuLink">
        <a class="dropdown-item" href="{{ path('app_login') }}">Connexion</a>
      </div>
    </div>
  </div>
  {% endif %}
</div>
</div>
</nav>
```

Pour avoir accès aux icônes de Font Awesome, on ajoute le link dans templates/base.html.twig

```
<meta charset="UTF-8">
<title>{% block title %}Welcome!{% endblock %}</title>
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v6.3.0/css/all.css">

{% block stylesheets %}{% encore_entry_link_tags('app') %}{% endblock %}
```