

API APPLI



1 installation avec lando

Configuration de .lando.yml

lando start

Puis:

lando composer create-project symfony/website-skeleton appli_musique

*Une fois le projet initialisé,
remonter tout le contenu du dossier d'un cran*

Puis refaire lando start

Configuration du .env pour connecter la database

User:MDP@host:port/nom_de_db

```
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
DATABASE_URL="mysql://root:admin@127.0.0.1:3306/spotify"
#DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
###< doctrine/doctrine-bundle ###
###> symfony/messenger ###
```

```
name: api_appli_musique
recipe: symfony
config:
  php: '8.1'
  via: nginx
  webroot: public
services:
  database:
    type: mysql
    portforward: 3307
    creds:
      user: julien
      password: admin
      database: appli_musique
  node:
    type: node:14
    build:
      - yarn install
      - npm install
  tooling:
    yarn:
      service: node
    node:
      service: node
    npm:
      service: node
```

2 installation avec le CLI

Téléchargement du CLI

<https://symfony.com/download>

Info: il faut avoir au préalable PHP8 d'installé

[Installation PHP8](#)

Puis:

composer create-project symfony/website-skeleton appli_musique

Configuration du .env pour connecter la database

User:MDP@host:port/nom_de_db

```
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
DATABASE_URL="mysql://root:admin@127.0.0.1:3306/spotify"
#DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
###< doctrine/doctrine-bundle ###

###> symfony/messenger ###
```

Puis on passe la commande php bin/console d:d:c (doctrine:database:create)

3 Création des entités

lando console make:entity
php bin/console make:entity

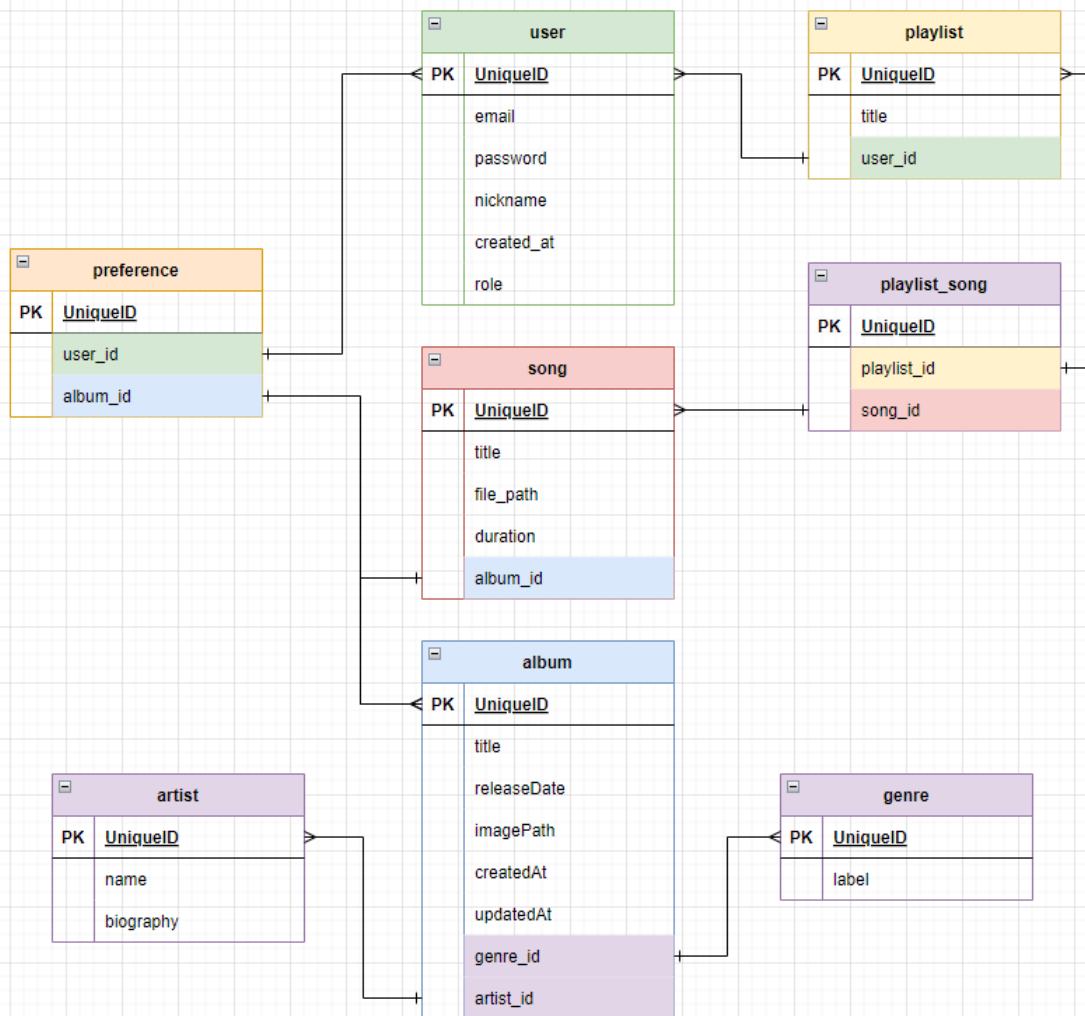
N.B: on crée toutes les entités
sans aucune relation, elles
seront faites à la fin pour éviter
les conflits

Puis:

lando console make:migration
php bin/console make:migration

Puis:

lando console d:m:m
php bin/console d:m:m



2 Création des entités (suite)

On commence avec l'entité User

```
php bin/console make:user
```

Ensuite on reviens dessus pour y ajouter ses autres propriété

```
php bin/console make:entity User
```

On crée ensuite les autres entités

- Genre
- Artist
- Album
- Song
- Playlist
- PlaylistSong
- Preference

3 Mise en place easyadmin bundle

Une fois toutes nos entités créées, on installe le bundle easy admin
composer require easycorp/easyadmin-bundle

Ensuite on va créer le menu d'administration

```
php bin/console make:admin:dashboard
```

On accepte les valeurs par défaut

On peut démarrer notre serveur symfony avec la commande:
symfony server:start

Et ajouter /admin à notre url

Welcome to EasyAdmin 4

You have successfully installed EasyAdmin 4 in your application.



[Read EasyAdmin Docs](#)

4 Création de notre back office

Nous allons maintenant exposer nos entités au back office, nous allons rentrer:

php bin/console make:admin:crud

Et commencer avec artiste (NB: on peut laisser les valeurs par défaut)

On recommence cette étape pour les entités suivantes:

- Genre
- Album
- Song

5 Configuration de l'accueil notre back office

Dans Controller, Admin, DashboardController.php nous allons créer les redirections pour les différentes entités exposées à notre BO (nous allons utiliser l'option 1)

```
class DashboardController extends AbstractDashboardController
{
    //dans un premier temps on crée le construct
    //qui prend comme paramètre une instance de AdminUrlGenerator
    public function __construct(
        private AdminUrlGenerator $adminUrlGenerator
    )
    {
    }

    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
        //on donne l'entité que l'on veut afficher au point d'entr"e de notre BO
        $url = $this->adminUrlGenerator
            ->setController( crudControllerFqn: GenreCrudController::class)
            ->generateUrl();

        return $this->redirect($url);
    }

    // return parent::index();
}
```


5 Configuration de l'accueil notre back office (suite)

On modifie le titre de notre BO

Nous allons ajouter un logo dans le dossier public (on crée un dossier image)

Nous allons ensuite afficher le logo

```
no usages
public function configureDashboard(): Dashboard
{
    return Dashboard::new()
        ->setTitle( title: '<span class="text-small"> Spotify</span>')
        ->setFaviconPath( path: '/images/logo2.png');
}
```

Nous allons voir pour construire notre menu de back office

6 Configuration des menus du back office

```
public function configureMenuItems(): iterable
{
    //Section principale
    yield MenuItem::section( label: 'Gestion Discographie');
    //Liste des sous-menu
    yield MenuItem::subMenu( label: 'Gestion Catégories', icon: 'fa fa-star')->setSubItems([
        MenuItem::linkToCrud( label: 'Ajouter une catégorie', icon: 'fa fa-plus', entityFqcn: Genre::class)->setAction( actionName: Crud::PAGE_NEW),
        MenuItem::linkToCrud( label: 'Voir les catégories', icon: 'fa fa-eye', entityFqcn: Genre::class)
    ]);
    yield MenuItem::subMenu( label: 'Gestion Albums', icon: 'fa fa-music')->setSubItems([
        MenuItem::linkToCrud( label: 'Ajouter un album', icon: 'fa fa-plus', entityFqcn: Album::class)->setAction( actionName: Crud::PAGE_NEW),
        MenuItem::linkToCrud( label: 'Voir les albums', icon: 'fa fa-eye', entityFqcn: Album::class)
    ]);
    yield MenuItem::subMenu( label: 'Gestion Chansons', icon: 'fa fa-play')->setSubItems([
        MenuItem::linkToCrud( label: 'Ajouter une chanson', icon: 'fa fa-plus', entityFqcn: Song::class)->setAction( actionName: Crud::PAGE_NEW),
        MenuItem::linkToCrud( label: 'Voir les chansons', icon: 'fa fa-eye', entityFqcn: Song::class)
    ]);
    yield MenuItem::subMenu( label: 'Gestion Artistes', icon: 'fa fa-user')->setSubItems([
        MenuItem::linkToCrud( label: 'Ajouter un artiste', icon: 'fa fa-plus', entityFqcn: Artist::class)->setAction( actionName: Crud::PAGE_NEW),
        MenuItem::linkToCrud( label: 'Voir les artistes', icon: 'fa fa-eye', entityFqcn: Artist::class)
    ]);
}
```

7 Modification des CRUD Controller (GenreCrudController)

On va modifier les CRUD, on commencera par GenreCrudController.php

```
class GenreCrudController extends AbstractCrudController
{
    no usages
    public static function getEntityFqcn(): string
    {
        return Genre::class;
    }

    public function configureCrud(Crud $crud): Crud
    {
        //permet de renommer les différentes pages
        return $crud
            ->setPageTitle(
                pageName: Crud::PAGE_INDEX, title: 'Catégories de chanson')
            ->setPageTitle(
                pageName: Crud::PAGE_EDIT, title: 'Modifier une catégorie')
            ->setPageTitle(
                pageName: Crud::PAGE_NEW, title: 'Ajouter une catégorie de chanson');
    }
}
```

7 Modification des CRUD Controller (GenreCrudController)

On va modifier le formulaire, on commencera par GenreCrudController.php

```
no usages
public function configureFields(string $pageName): iterable
{
    //permet de redéfinir le formulaire
    return [
        IdField::new( propertyName: 'id')->hideOnForm(),
        TextField::new( propertyName: 'label'),
    ];
}
```

```
//Fonction pour agir sur les boutons d'actions
```

7 Modification des CRUD Controller (suite)

On va modifier rendu des différents boutons d'actions (on peut appliquer ce code sur tous les CRUD)

```
//Fonction pour agir sur les boutons d'actions
public function configureActions(Actions $actions): Actions
{
    //permet de configurer les différentes actions
    return $actions
        // Permet de customiser les champs de la page index
        ->update( Crud::PAGE_INDEX, Action::NEW,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-add')->setLabel( label: 'Ajouter')->setCssClass( cssClass: 'btn btn-success'))
        ->update( Crud::PAGE_INDEX, Action::EDIT,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-pen')->setLabel( label: 'Modifier'))
        ->update( Crud::PAGE_INDEX, Action::DELETE,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-trash')->setLabel( label: 'Supprimer'))
        //Page edition
        ->update( Crud::PAGE_EDIT, Action::SAVE_AND_RETURN,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer et quitter'))
        ->update( Crud::PAGE_EDIT, Action::SAVE_AND_CONTINUE,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer et continuer'))
        //page création
        ->update( Crud::PAGE_NEW, Action::SAVE_AND_RETURN,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer'))
        ->update( Crud::PAGE_NEW, Action::SAVE_AND_ADD_ANOTHER,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer et ajouter un nouveau'));
}
```

7 Modification des CRUD Controller (ArtistCrudController)

ArtistCrudController.php

```
public function configureCrud(Crud $crud): Crud
{
    //permet de renommer les différentes pages
    return $crud
        ->setPageTitle( pageName: Crud::PAGE_INDEX, title: 'Artistes')
        ->setPageTitle( pageName: Crud::PAGE_EDIT, title: 'Modifier un artiste')
        ->setPageTitle( pageName: Crud::PAGE_NEW, title: 'Ajouter un artiste');
}

//formulaire artiste
no usages
public function configureFields(string $pageName): iterable
{
    return [
        IdField::new( propertyName: 'id')->hideOnForm(),
        TextField::new( propertyName: 'name', label: 'Nom de l\'artiste'),
        TextEditorField::new( propertyName: 'biography', label: 'Biographie de l\'artiste'),
    ];
}
```

7 Modification des CRUD Controller (AlbumCrudController)

On va maintenant passer à AlbumCrudController.php, on aura un peu plus de traitement à effectuer au niveau du formulaire, on aura aussi des persister pour date de création et date de modification. On récupère le configureActions.

On déclare nos constantes

```
class AlbumCrudController extends AbstractCrudController
{
    //on crée nos constantes
    no usages
    public const ALBUM_BASE_PATH = 'upload/images/albums';
    no usages
    public const ALBUM_UPLOAD_DIR = 'public/upload/images/albums';

    no usages
}
```

7 Modification des CRUD Controller (AlbumCrudController)

On crée notre formulaire

```
public function configureFields(string $pageName): iterable
{
    return [
        IdField::new( propertyName: 'id')->hideOnForm(),
        TextField::new( propertyName: 'title', label: 'Titre de l\'album'),
        TextEditorField::new( propertyName: 'description', label: 'Description de l\'album'),
        //Champs d'association avec une autre table
        AssociationField::new( propertyName: 'genre', label: 'Catégorie de l\'album'),
        AssociationField::new( propertyName: 'artist', label: 'Nom de l\'artiste'),
        ImageField::new( propertyName: 'imagePath', label: 'Choisir une image de couverture')
            ->setBasePath( path: self::ALBUM_BASE_PATH)
            ->setUploadDir( uploadDirPath: self::ALBUM_UPLOAD_DIR)
            ->setUploadedFileNamePattern(
                //On donne un nom de fichier unique pour éviter de venir écraser une image en cas de même nom
                fn(UploadedFile $file): string => sprintf(
                    format: 'upload_%d_%s.%s',
                    random_int(1, 999),
                    $file->getFilename(),
                    $file->guessExtension()
                ),
            ),
        DateField::new( propertyName: 'releaseDate', label: 'Date de sortie'),
        //ici on cache createdAt et updatedAt on passera les données grace au persister
        DateField::new( propertyName: 'createdAt')->hideOnForm(),
        DateField::new( propertyName: 'updatedAt')->hideOnForm(),
    ];
}
```


7 Modification des CRUD Controller (AlbumCrudController)

Dans notre BO on se retrouve avec une erreur

Error

Object of class App\Entity\Genre could not be converted to string

Car notre input AssociationField ne sait pas récupérer la valeur que l'on veut afficher tout seul, il va falloir indiquer dans les entités mêmes, quelles propriétés il va falloir stringifier

Dans Genre.php

```
public function __toString(): string
{
    return $this->label;
}
```

Dans Album.php

```
no usages
public function __toString(): string
{
    return $this->title;
}
```

Notre formulaire s'affiche bien, par contre il va falloir créer nos persister pour createdAt et updatedAt

7 Modification des CRUD Controller (AlbumCrudController)

On crée nos persister dans AlbumCrudController

```
//persister lors de la création d'un album, on génère la date
public function persistEntity(EntityManagerInterface $em, $entityInstance): void
{
    if (!$entityInstance instanceof Album) return;
    $entityInstance->setCreatedAt(new \DateTimeImmutable());
    parent::persistEntity($em, $entityInstance);
}

//persister lors de la modification d'un album, on génère la date
public function updateEntity(EntityManagerInterface $em, $entityInstance): void
{
    if (!$entityInstance instanceof Album) return;
    $entityInstance->setUpdatedAt(new \DateTimeImmutable());
    parent::updateEntity($em, $entityInstance);
}
```

7 Modification des CRUD Controller (SongCrudController)

Pour SongCrudController on va devoir uploader des musiques, pour ce faire, nous allons installer une nouvelle librairie: composer require vich/uploader-bundle

Une fois installé, nous allons configurer le fichier config/packages/vich_uploader.yml

```
vich_uploader:
    db_driver: orm

    mappings:
        songs:
            uri_prefix: '%song_file%'
            upload_destination: '%kernel.project_dir%/public%song_file%'
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
            delete_on_update: false
            delete_on_remove: false
```

Et dans services.yaml

```
# https://symfony.com/doc/current/best-practices.html#parameters
parameters:
    song_file: /upload/files/music

services:
    # default configuration for service
```

7 Modification des CRUD Controller (SongCrudController)

Dans Song.php (entité) nous devons appeler Vich

```
use Doctrine\ORM\Mapping as ORM;
use Vich\UploaderBundle\Mapping\Annotation as Vich;

22 usages
#[ORM\Entity(repositoryClass: SongRepository::class)]
#[Vich\Uploadable]
class Song
{
```

Ensuite nous devons ajouter une nouvelle propriété dans Song.php (mapping correspond à la config .yaml)

```
//Ajout d'une propriété file
no usages
#[Vich\UploadableField(mapping: 'songs', fileNameProperty: 'filePath')]
private ?File $filePathFile = null;

#[ORM\ManyToOne(inversedBy: 'songs')]
```

7 Modification des CRUD Controller (SongCrudController)

On ajoute ensuite les méthodes get et set de la propriété

```
//ici on ajoute les propriétés de filePathFile
```

```
no usages
```

```
public function getFilePathFile(): ?File
```

```
{
```

```
    return $this->filePathFile;
```

```
}
```

```
no usages
```

```
public function setFilePathFile(?File $filePathFile = null): void
```

```
{
```

```
    $this->filePathFile = $filePathFile;
```

```
}
```

7 Modification des CRUD Controller (SongCrudController)

Dans SongCrudController, on crée notre formulaire, on oublie pas de créer les dossiers pour l'upload de mp3

```
public function configureFields(string $pageName): iterable
{
    return [
        IdField::new( propertyName: 'id')->hideOnForm(),
        TextField::new( propertyName: 'title', label: 'titre de la chanson'),
        TextField::new( propertyName: 'filePathFile', label: 'choisir un fichier mp3')
            ->setFormType( formTypeFqcn: VichFileType::class)
            ->hideOnDetail()
            ->hideOnIndex(),
        TextField::new( propertyName: 'filePath', label: 'Nom du fichier mp3')
            ->hideOnForm()
            ->hideOnIndex(),
        ImageField::new( propertyName: 'filePath', label: 'choisir un fichier mp3')
            ->setBasePath( path: '/upload/files/music')
            ->hideOnForm()
            ->hideOnIndex()
            ->hideOnDetail()
            ->addHtmlContentsToBody( ...contents: '<p>hello</p>'),
        NumberField::new( propertyName: 'duration', label: 'durée de la chanson'),
        AssociationField::new( propertyName: 'album', label: 'Album associé'),
    ];
}
```

7 Modification des CRUD Controller (SongCrudController)

On modifie configureActions pour ajouter une page détail

```
public function configureActions(Actions $actions): Actions
{
    return $actions
        // Permet de customiser les champs de la page index
        ->update( pageName: Crud::PAGE_INDEX,  actionName: Action::NEW,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-add')->setLabel( label: 'Ajouter')->setCssClass( cssClass: 'btn btn-success'))
        ->update( pageName: Crud::PAGE_INDEX,  actionName: Action::EDIT,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-pen')->setLabel( label: 'Modifier'))
        ->update( pageName: Crud::PAGE_INDEX,  actionName: Action::DELETE,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-trash')->setLabel( label: 'Supprimer'))
        ->add( pageName: Crud::PAGE_INDEX,  actionNameOrObject: Action::DETAIL)
        ->update( pageName: Crud::PAGE_INDEX,  actionName: Action::DETAIL,
            fn(Action $action) => $action->setIcon( icon: 'fa fa-info')->setLabel( label: 'Informations'))

        //Page edition
        ->update( pageName: Crud::PAGE_EDIT,  actionName: Action::SAVE_AND_RETURN,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer et quitter'))
        ->update( pageName: Crud::PAGE_EDIT,  actionName: Action::SAVE_AND_CONTINUE,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer et continuer'))

        //page création
        ->update( pageName: Crud::PAGE_NEW,  actionName: Action::SAVE_AND_RETURN,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer'))
        ->update( pageName: Crud::PAGE_NEW,  actionName: Action::SAVE_AND_ADD_ANOTHER,
            fn(Action $action) => $action->setLabel( label: 'Enregistrer et ajouter un nouveau'));
}
```

8 Création de l'API

On va avoir besoin [d'Api Platform](#)


composer require api

Maintenant que la librairie est installé, on va devoir l'étendre à nos entités.

Dans Album.php

```
1 inheritor  julien.linard *  
#[ORM\Entity(repositoryClass: AlbumRepository::class)]  
//On expose l'entité à l'api  
//il suffit simplement de faire référence à ApiResource  
#[ApiResource()]  
class Album  
{
```

Maintenant sur l'adresse local/api vous avez accès au swagger de l'API pour tester les requêtes

Album		^
GET	/api/albums Retrieves the collection of Album resources.	v
POST	/api/albums Creates a Album resource.	v
GET	/api/albums/{id} Retrieves a Album resource.	v
PUT	/api/albums/{id} Replaces the Album resource.	v 
DELETE	/api/albums/{id} Removes the Album resource.	v
PATCH	/api/albums/{id} Updates the Album resource.	v

9 Création de groupes pour l'API

Les groupes permettent de « regrouper » toutes les propriétés que l'on souhaite recevoir (relations comprises) ou envoyer à notre BDD. Il se base sur le principe de normalisation et dénormalisation.

Dans un 1^{er} temps on déclare notre groupe dans `apiRessource`, ensuite on affine le groupe aux différentes propriétés que l'on souhaite inclure dans le groupe.

```
1 inheritor  julien.linard *  
#[ORM\Entity(repositoryClass: AlbumRepository::class)]  
//On expose l'entité à l'api  
#[ApiResponse(  
    //On déclare les groupes de serialisation et de désérialisation  
    normalizationContext: ['groups' => ['album:read']],  
    denormalizationContext: ['groups' => ['album:write']],  
)]  
  
class Album  
{
```

Maintenant dans l'interface on a notre objet avec toutes les infos nécessaires

On peut aussi ajouter des filtres

```
#[ApiFilter(  
    SearchFilter::class, properties: ['id' => 'exact', 'title' => 'exact', 'description' => 'partial', 'genre.label' => 'exact']  
)]  
  
class Album
```

et au dessus des propriétés que l'on veut ajouter

```
#[ORM\Column(length: 150)]  
#[Groups("album:read")]  
private ?string $name = null;
```