UEL313-G6-S4 : projet de groupe Symfony

WATSON est une application de gestion et de mutualisation de liens. Son code source est disponible sur https://github.com/Clouddy23/UEL313-G6-S4/

Objectifs

- [x] Prise en main du cadriciel Symfony 6.4 (LTS)
- [x] Développement d'une API
- [x] Création de templates TWIG
- [x] Fonctionnalités MVP
  - [x] listing de liens stockés en base de données ;
  - [x] ajout de lien en base de donnés à partir d'un formulaire, un lien étant composé au minimum d'un titre et d'une URL et d'un descriptif ;
  - [x] mise à jour des liens stockés en base de données
  - [x] suppression de liens en base de données
- [x] Fonctionnalités supplémentaires
  - [x] proposer une gestion de mots clés à associer à un lien;
  - [x] permettre un espace "back office" avec accès restreint pour gérer les liens;
  - [x] permettre la gestion d'utilisateurs pour le "back office";
  - [x] proposer un rendu Twig/CSS de base.

Principe général de collaboration

Membres du groupe

Étudiant.e	Alias
Mathilde C.	Clouddy23
Kamo G.	Spaghette5
Mathieu L.	mathleys
Filippos K.	filkat34

Répartition du travail

Activité	Responsable(s)
Initialisation et configuration de l'environnement dev	Mathilde
Modèle, interfaaces, et manipulation des données	Filippos
Authentification/inscription, Webroutes	Filippos
API Liens	Mathieu
API Tags	Mathilde

Activité	Responsable(s)
UI Twig/CSS	Kamo
Tests fonctionnels	Filippos, Mathilde, Mathieu, Kamo
Documentation	Filippos, Mathilde

Chacun des membres du groupe a contribué au projet selon ses disponibilités et compétences.Tout le monde s'est montré impliqué et investi dans le travail demandé.

Calendrier de suivi du projet

Échéance	Objectif
15/12	Phase d'installation : installation Symfony 6.4 LTS, préparation du repository.
16/12	Visio d'organisation : répartition des tâches, création des issues/branches.
17/12	Phase de développement
19/12	Phase de relecture : Review et correction des branches (PR).
20/12	Fin du projet : Tests manuels fonctionnels, fusion des branches vers `main`, finalisation du PDF.

L'environnement de développement

Tous les outils suivants doivent être disponibles “globalement” dans le terminal depuis n’importe quel dossier :

- PHP
- Composer
- Symfony CLI
- SQLite
- DB Browser

Installation de Symfony CLI 6.4 LTS

Installation Homebrew (nécessaire sur MacOS)

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew --version
```

Installation de PHP

Sur MacOS :

```
brew update
brew install php
```

```
php -v
```

Sur Debian :

```
sudo apt install php php-zip php-sqlite3 php-mbstring php-xml php-curl
php -v
```

Extensions PHP requises

Vérifier si les extensions nécessaires sont actives :

```
php -m | grep -E "zip|sqlite|pdo_sqlite"
```

Si ce n'est pas le cas, procéder à l'activation des extensions dans le fichier `php.ini` en décommentant (suppression de ";") les extensions suivantes : `extension=zip`, `extension=pdo_sqlite`, `extension=sqlite3`.

Installation de Composer

Sur MacOS :

```
brew install composer
sudo composer self-update
composer -V
```

Sur Debian :

```
sudo apt install composer
composer -V
```

Installation de Symfony CLI

Sur Mac :

```
brew install symfony-cli/tap/symfony-cli
symfony -V
symfony check:requirements
```

Sur Debian :

```
curl -1sLf
&#039;https://dl.cloudsmith.io/public/symfony/stable/setup.deb.sh&#039; |
sudo -E bash &amp;&amp;
sudo apt install symfony-cli
```

Création du projet Watson-Symfony

Initialisation

Création d'un nouveau projet Symfony dans un dossier `watson-symfony` en forçant la version 6.4 LTS :

```
symfony new watson-symfony --version="6.4.*" --webapp
cd watson-symfony
```

Configurer de SQLite dans .env.local

On copie .env vers .env.local afin de : préserver la configuration par défaut (.env) et permettre à chaque membre du groupe d'avoir sa configuration locale (.env.local) :

```
cp .env .env.local
code .env.local
```

Dans .env.local, activer DATABASE_URL="SQLite..." (supprimer #) et désactiver DATABASE_URL="postgesql..." (ajouter #).

Installer DB Browser

Avec SQLite, la BDD est un fichier .db qui sera créé lors des migrations (après création des entités).

La visualisation et les manipulations de la BDD peuvent ensuite se faire grâce à l'installation de DBBrowser qui peut se faire via le terminal.

Sur Mac :

```
brew install --cask db-browser-for-sqlite
```

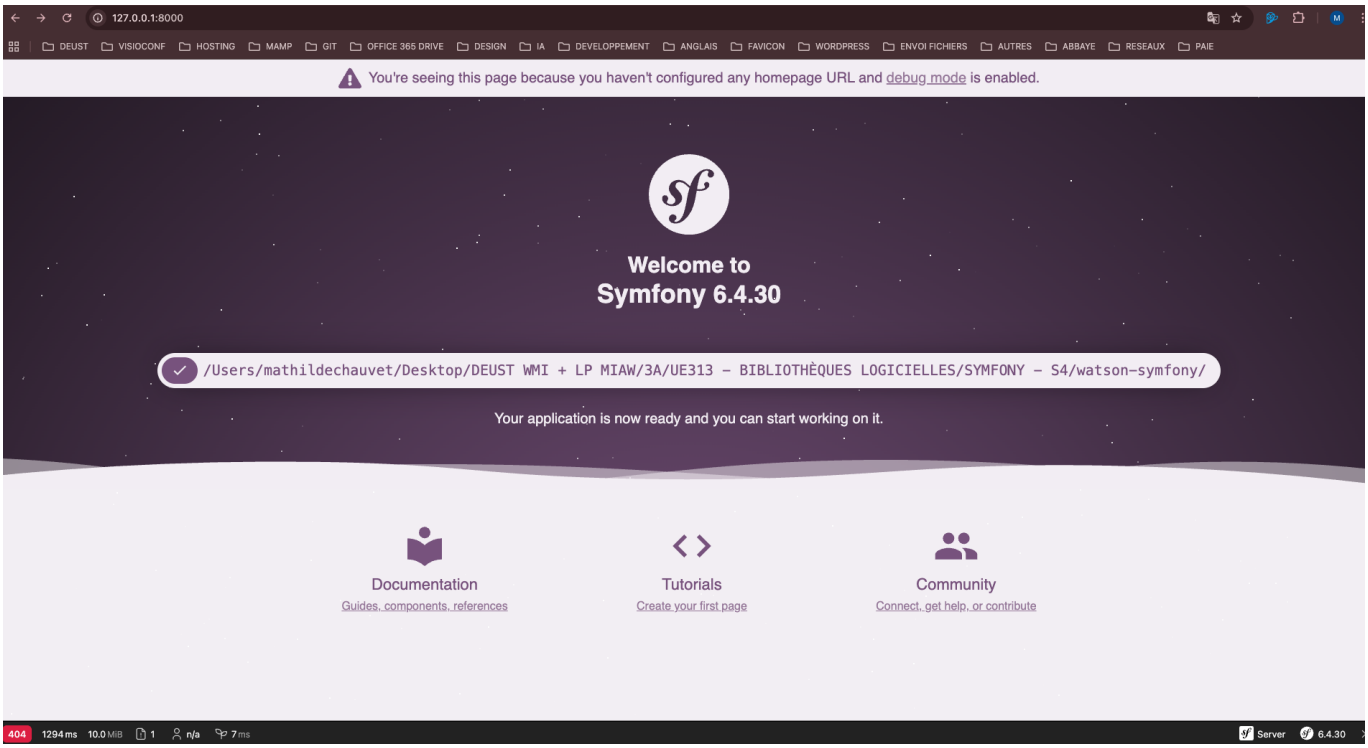
Sur Debian :

```
sudo apt install sqlitebrowser
```

Tester le serveur Symfony

```
cd "./S4/watson-symfony"
symfony server:start
```

Se rendre à l'URL indiquée par le serveur, par exemple : http://127.0.0.1:8000.



Mettre à jour le fichier .env

- Modification du fichier *.env* en renseignant l'URL correct de la base de données : `DATABASE_URL="sqlite:///%kernel.project_dir%/var/watson.db"`
- Modification du fichier *.gitignore* pour exclure le fichier *watson.db* afin qu'il puisse être partagé avec les collaborateurs.

Base de données

Le Modèle

Nous avons d'abord recréé les trois entités de l'application watson (User, Tag, Link) ainsi que leurs méthodes de gestion de base. Il a fallu refactoriser l'entité *User* d'origine et toutes ses références par la suite afin qu'elle implemente l'*UserInterface* de Symfony, nécessaire pour utiliser le système d'authentification du framework.

Dans une perspective de standardisation et de clarté du code, nous avons créé des interfaces pour chacune de ces trois entités à l'exception de *User* qui en possédait déjà une.

Deux *Repositories* ont été créés pour y placer les requêtes complexes à la base de données, impliquant des jointures.

Nous avons pour finir créé plusieurs *Commandes* pour trois commandes pour peupler la base de données.

Création et mise en place de la base

La base de données *SQLite* est déjà partagée dans le dépôt. Mais si l'on souhaite l'effacer et la réinitialiser avec des données de test, voici la démarche à suivre :

```
php bin/console doctrine:schema:create # pour créer la base de données _SQLite_ "watson.db"
php bin/console app:create-admin-user # pour créer le premier adminitrateur de la base de données
php bin/console app:populate-tags # pour peupler la table des tags
php bin/console app:populate-links # pour peupler la table des liens
php bin/console app:populate-link-tag-associations # pour peupler la table des associations entre link et tags
```

Suites à ces commandes nous vérifions grâce à *DB Browser* que la base de données a été correctement remplie.

 DB Browser

Documentation et tests : *Nelmio*

Pour faciliter la documentation, le codage et l'implementation des méthodes CRUD, nous avons installé *Nelmio* avec la commande :

```
composer require nelmio/api-doc-bundle
```

Pour le configurer, nous avons suivi la documentation disponible sur le site de Symfony.

L'interface de test est disponible sur la route /api/doc.

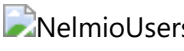
Codage de l'API

Nous avons par la suite procédé au codage des différentes composantes de l'API et notamment des contrôleurs qui gèrent les différentes requêtes HTTP pour interroger la base des données.

Afin que chacune de ces requêtes soit documentée sur *Nelmio*, nous avons ajouté au dessus de chacune d'elles l'attribut correspondant commençant par `#[OA\METHOD(...)]`

Contrôleur *Users*

Le contrôleur qui interroge la table des utilisateurs est exposé à la route `/api/users`. Quatre requêtes ont été codées.

 NelmioUsers

Contrôleur *Liens*

Le contrôleur qui interroge la table des liens est exposé à la route `/api/links`. Neuf requêtes ont été codées.


Nelmio routes links

Contrôleur *Tags*

Le contrôleur qui interroge la table des mots clés (tags) est exposé à la route `/api/tags`. Huit requêtes ont été codées. Nous avons également créé les différentes méthodes CRUD pour implementer le lien d'association entre les liens et les mots clés.


Nelmio routes tags

Plus précisément, nous avons modélisé la relation ManyToMany entre Link et Tag :

- Link possède une collection de Tag ($tags)
- Tag possède une collection de Link ($links)
- Une table link_tag gère les associations en BDD

Pour éviter des incohérences côté application, nous avons synchronisé les deux côtés de la relation dans les méthodes :

- Link::addTag() / Link::removeTag()
- Tag::addLink() / Tag::removeLink()

Nous avons, enfin, développé un contrôleur TagController (API JSON) exposant des routes :

- GET /tags : liste de tous les tags
- POST /tags : création d'un tag (JSON { "name": "..." })
- GET /tags/{id} : détail d'un tag
- PUT /tags/{id} : modification d'un tag
- DELETE /tags/{id} : suppression d'un tag

A cela nous avons ajouté des routes permettant de gérer la relation tag et lien :

- POST /links/{linkId}/tags/{tagId} : associer un tag existant à un lien
- DELETE /links/{linkId}/tags/{tagId} : dissocier un tag d'un lien
- GET /links/{linkId}/tags : lister les tags d'un lien

Il est à noter que nous avons également réalisé une protection contre la création de doublons :

- Lors de la création d'un tag (`POST /tags`) on vérifie qu'un tag du même nom n'existe pas déjà en base
- Lors de l'association d'un tag à un lien (`POST /links/{linkId}/tags/{tagId}`) une vérification empêche d'ajouter 2 fois le même tag au même lien.

Tests manuels fonctionnels

L'interface de *Nelmio*, nous a servi pour tester l'API et nous assurer que toutes les routes pointent vers des requêtes valides. Ci-dessous, un exemple de la requête GET sur la route */api/links* :


Nelmio exemple test

La création de cette API a été ambitieuse et seulement une petite partie de ses méthodes a été reprise pour l'implémentation des routes publiques. Cependant elle pourra éventuellement servir dans la maintenance

évolutive de l'application, notamment si l'on décide d'adopter une librairie frontend comme *React* qui permettrait plus de liberté dans la manupulation des données au niveau de l'UI que les templates TWIG actuels.

Implémentation des routes Web

Après avoir testé le bon fonctionnement de toutes les routes de l'API, nous avons implémenté les deux contrôleurs qui servent à afficher les pages de l'application :

- reprise du contrôleur déjà implémenté `RSSFeedController` pour le flux RSS (`/feed`) ;
- création du contrôleur `HomeController`pour gérer la page d'accueil et retourner la liste des liens (`/`) ;
- création du contrôleur `BackOfficeController`pour gérer l'affichage du backoffice et les requêtes à la base de données (`/backoffice`).

Architectures : SSR vs API REST

Dans son état actuel, notre projet a une architecture hybride. Il a commencé, à des fins de test, comme un projet API REST comme on pourrait en trouver dans la plupart des frameworks *NodeJS* : on a créé des endpoints API (dans LinkController, UserController, TagsController) accessibles par des requêtes HTTP et qui ne servent qu'à la transmission pure d'informations entre le serveur et des clients sous format JSON.


API Diagram

Toutefois, la partie publique et "fonctionnelle" de l'application suit le modèle SSR (Server Side Rendering) telle qu'implémenté dans les contrôleurs *HomeController* et *BackOfficeCOntroller* : le serveur Symfony génère entièrement les pages HTML avant de les envoyer au navigateur. Plus précisément, ces contrôleurs reçoivent les requêtes HTTP, traitent les données, puis utilisent Twig pour produire le HTML final avant de l'envoyer au navigateur de l'utilisateur. Tout se passe côté serveur.


SSR Diagram

Authentification et inscription

Nous avons mis en place un système d'authentification et d'inscription en suivant la documentation de Symfony :

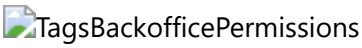
```
composer require symfony/security-bundle # pour installer le
_SecurityBundle_ qui ajoute toutes les fonctionnalités nécessaires à
l'authentification
php bin/console make:user # pour récréer une entité _User_ compatible avec
Symfony. Nous avons dû procéder à des refactorisations et à la
réinitialisation de la base de données
php bin/console make:registration-form # pour ajouter un formulaire
d'inscription et les méthodes nécessaires pour l'enregistrement dun nouvel
utilisateur
```

Nous sommes intervenus au niveau du fichier `config/packages/security.yaml` pour sécuriser les routes en exigeant une authentification en cas de requêtes sur les routes de l'API et pour donner un accès public aux routes `/login`, `register` et `feed`


firewallApi


PublicRoutes

La logique du *BackofficeController* a été refactorisée également pour discriminer, concernant la disponibilité des deux onglets (utilisateurs, liens), en fonction des rôles des utilisateurs connectés : le rôle USER ne peut que ajouter/modifier/supprimer des liens alors que le rôle ADMIN peut en plus gérer les comptes des utilisateurs.


TagsBackofficePermissions

Front-end

Templates TWIG

Les templates `base.html.twig`, `login.html.twig` et `register.html.twig` ont été générés automatiquement par *Symfony* pour le premier lors de la création du projet et, pour les deux autres, lors de l'installation du *SecurityBundle*.

Le template `base.html.twig` comporte la mise en page générale du site avec la *navbar* et le *footer* : tous les autres templates en dépendent.

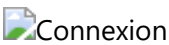
Nous avons crée des templates supplémentaires:

- `index.html.twig` qui constitue la page d'accueil du site permettant de visualiser l'ensemble des liens et offrant un champ de recherche ;
- `templates\backoffice.html.twig` qui constitue la page principale du backoffice qui liste les liens et les utiliateurs dans les onglets dédiés ;
- `templates\admin\link_create.html.twig` et `templates\admin\link_edit.html.twig`qui permettent d'ajouter ou modifier des liens ;
- `templates\admin\user_create.html.twig` et `templates\admin\user_edit.html.twig` qui permettent d'ajouter ou modifier des utilisateurs.

Interface utilisateur

L'UI de l'application a été faite grâce à la librairie *Bootstrap* installée dans les dépendances du projet (`assets/vendor/bootstrap/`) ainsi qu'au paquet *Bootstrap Icons* importé via un CDN au niveau du template de base `templates\base.html.twig`. Des classes CSS personnalisées ont également été créées dans le fichier `assets\styles\app.css`.


Accueil


Connexion


Backoffice

Webographie

- PHP

- Composer
- Symfony CLI
- Symfony SecurityBundle
- Mise en place d'un projet Symfony
- DB Browser
- Nelmio