

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

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

Principe général de collaboration

Membres du groupe

| Étudiant.e | Alias |
|-------------|------------|
| Mathilde C. | Cloudy23 |
| Kamo G. | Spaghette5 |
| Mathieu L. | mathleys |
| Filippos K. | filkat34 |

Répartition du travail

| Activité | Responsable(s) |
|--|----------------|
| Initialisation et configuration de l'environnement dev | Mathilde |
| Modèle, interfaces, repositories et bdd | Filippos |
| Authentification, Inscription, Webroutes | Filippos |
| API Liens | Mathieu |
| API Tags | Mathilde |

| Activité | Responsable(s) |
|--------------------|-----------------------------------|
| UI Twig/CSS | Kamo |
| Tests fonctionnels | Filippes, Mathilde, Mathieu, Kamo |
| Documentation | Filippes, 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;&;
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
```

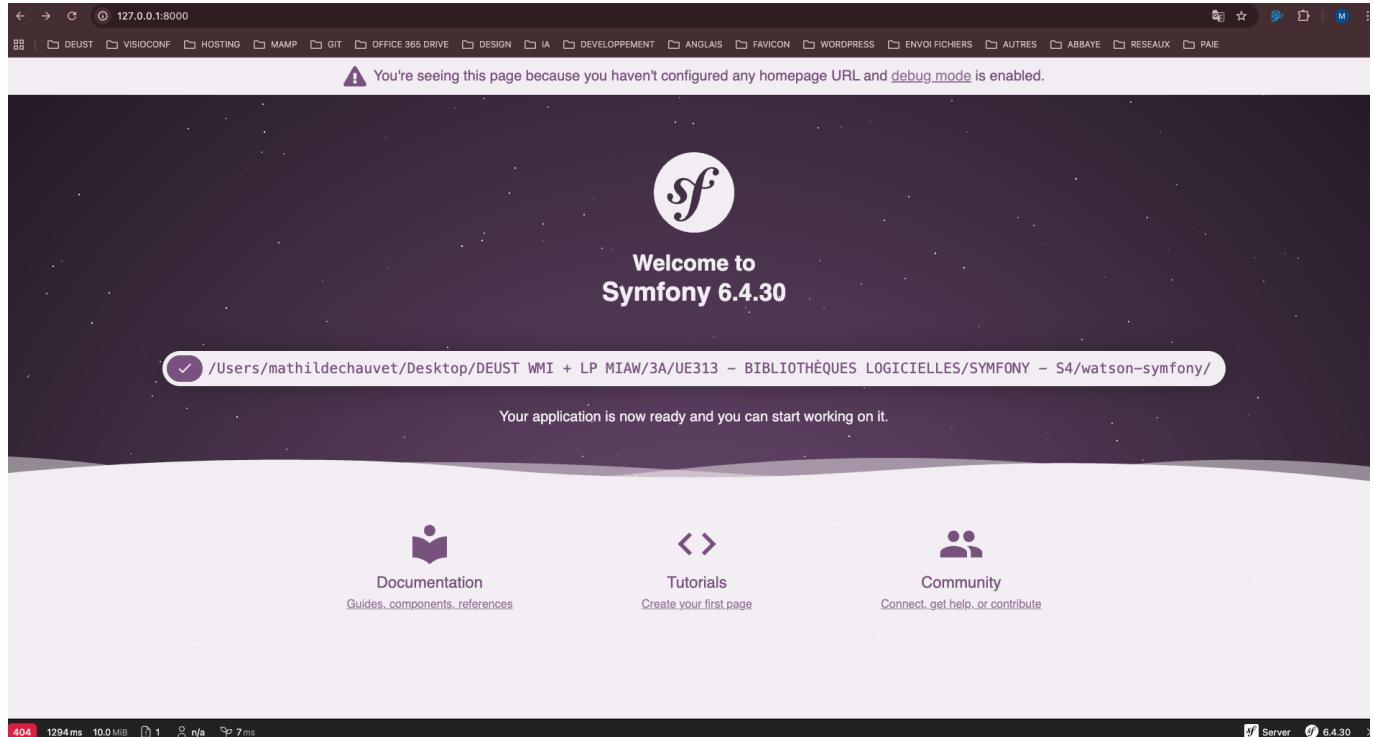
Sur Debian :

```
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
administrateur 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.

The screenshot shows the DB Browser for SQLite interface. On the left, there's a table view titled 'link' with columns: id, url, title, desc, created_at, and user_id. The table contains 31 rows of data, each representing a URL entry with its details. On the right, the 'DB Schema' tab is open, displaying the database structure. It includes tables like 'Tables (6)' such as 'link', 'tag', and 'user', and indexes like 'Index (7)' such as 'IDX_36AC99F1A76ED395'. The schema also shows constraints like 'UNIQ_IDENTIFIER_USERNAME'.

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.

| Users | |
|--------|--|
| GET | <code>/api/users</code> Retourne la liste de tous les utilisateurs |
| POST | <code>/api/users</code> Ajoute un nouvel utilisateur |
| PUT | <code>/api/users/{id}</code> Modifie un utilisateur existant |
| DELETE | <code>/api/users/{id}</code> Supprime un utilisateur |

Contrôleur *Liens*

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

| Links | |
|--------|--|
| GET | <code>/api/links</code> Retourne la liste de tous les liens |
| POST | <code>/api/links</code> Ajoute un nouveau lien |
| PUT | <code>/api/links/{id}</code> Modifie un lien existant |
| DELETE | <code>/api/links/{id}</code> Supprime un lien |
| GET | <code>/api/links/user/{userId}</code> Retourne tous les liens d'un utilisateur |
| GET | <code>/api/links/tag/{tagId}</code> Retourne tous les liens d'un tag |
| GET | <code>/api/links/search</code> Recherche des liens par terme |
| GET | <code>/api/links/recent</code> Retourne les liens récents |
| GET | <code>/api/links/paginated</code> Retourne les liens avec pagination |

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.

| Tags Tags | | |
|-----------|----------------------------------|--|
| GET | /api/tags | Retourne la liste de tous les tags |
| POST | /api/tags | Crée un nouveau tag |
| GET | /api/tags/{id} | Retourne un tag par son ID |
| PUT | /api/tags/{id} | Modifie un tag existant |
| DELETE | /api/tags/{id} | Supprime un tag |
| POST | /api/links/{linkId}/tags/{tagId} | Associe un tag existant à un lien existant |
| DELETE | /api/links/{linkId}/tags/{tagId} | Dissocie un tag d'un lien |
| GET | /api/links/{linkId}/tags | Liste les tags d'un lien |

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` :

The screenshot shows a REST API documentation interface for the `/api/links` endpoint. At the top, it indicates a `GET` method and the URL `/api/links`. Below this, there's a section for "Parameters" which says "No parameters". On the right side, there are "Execute" and "Clear" buttons. The "Responses" section is expanded, showing:

- Curl:** A command-line example using curl to make a GET request to `http://localhost:8000/api/links` with an accept header of `/*`.
- Request URL:** `http://localhost:8000/api/links`
- Server response:** This section is collapsed.
- Code Details:** This section is collapsed.

Under "Server response", the status code is 200 and the response body is a JSON array of links. One link is shown in detail:

```

        },
        {
            "id": 29,
            "url": "https://www.npmjs.com/",
            "title": "npm",
            "desc": "Node package manager.",
            "user_id": 1,
            "tags": [
                {
                    "id": 16,
                    "name": "Bootstrap"
                },
                {
                    "id": 19,
                    "name": "Webpack"
                }
            ],
            "id": 28,
            "url": "https://webpack.js.org/",
            "title": "Webpack",
            "desc": "Module bundler for JavaScript.",
            "user_id": 1,
            "tags": [
                {
                    "id": 19,
                    "name": "Bootstrap"
                }
            ]
        }
    ]
}

```

Below the response body, there are download buttons for "Copy" and "Download".

Under "Response headers", the following HTTP headers are listed:

```

cache-control: no-cache,private
content-type: application/json
date: Fri,19 Dec 2025 17:15:30 GMT
transfer-encoding: chunked
x-debug-token: 836f8c
x-debug-token-link: http://localhost:8000/_profiler/836f8c
x-powered-by: PHP/8.4.13
x-robots-tag: noindex

```

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 manipulation des données au niveau de l'UI que les templates TWIG actuels.

Implémentation des routes Web

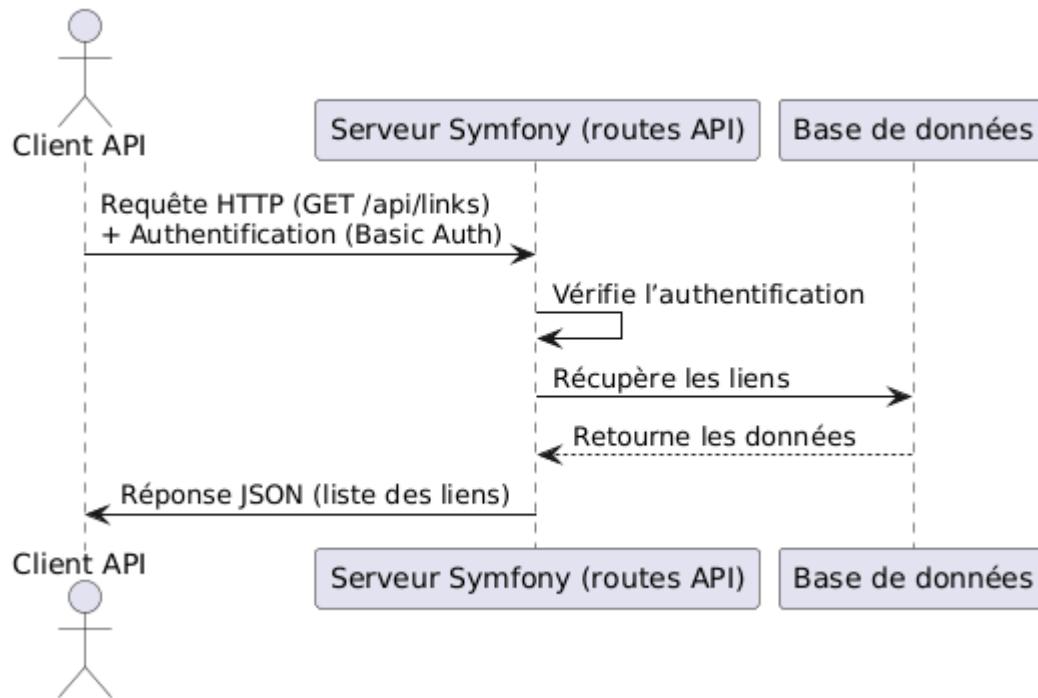
Après avoir testé le bon fonctionnement de toutes les routes de l'API, nous avons implémenté les deux contrôleurs qui servent à afficher les pages de l'application :

- 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`).

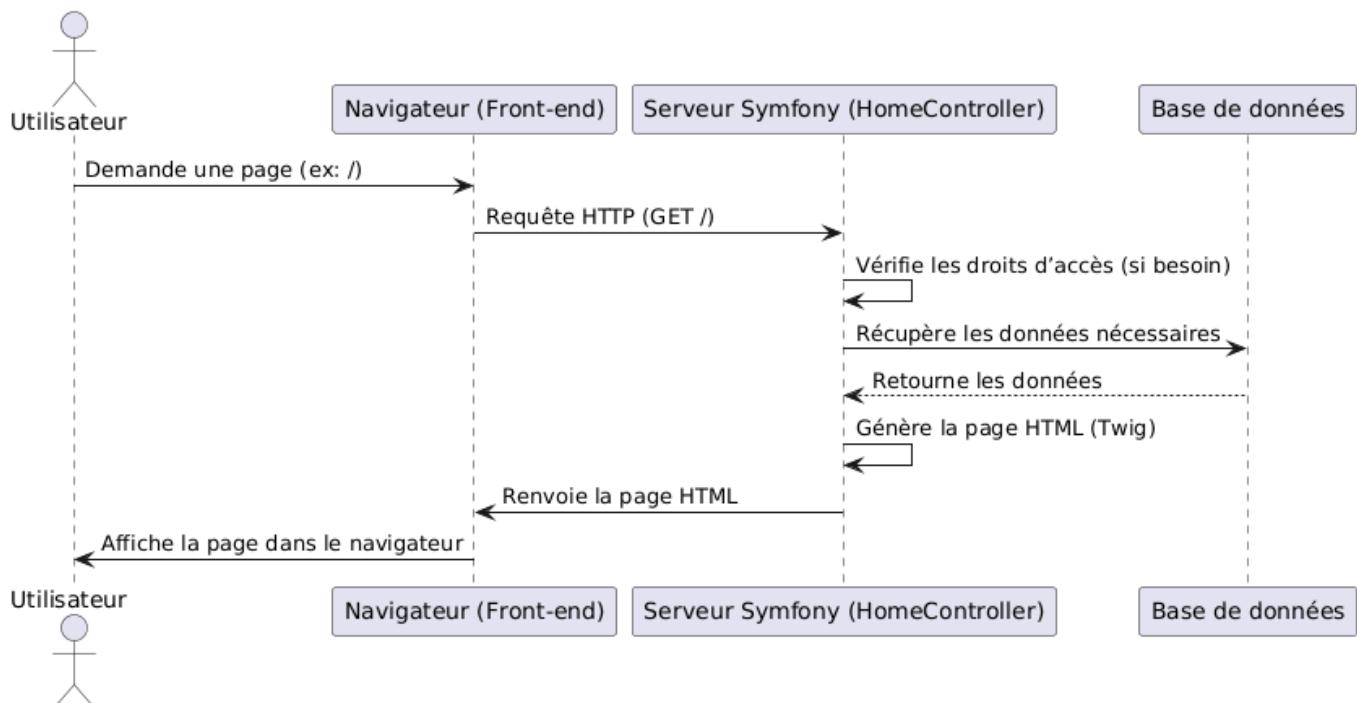
Architectures : SSR vs API REST

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.



Toutefois, la partie publique et "fonctionnelle" de l'application suit le modèle SSR (Server Side Rendering) telle qu'implémenté dans les contrôleurs *HomeController* et *BackOfficeCOnroller* : 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.



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 d'un 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`

```

firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  api:
    pattern: ^/api
    stateless: true
    provider: app_user_provider
    http_basic: ~
  main:
    lazy: true
    provider: app_user_provider
    entry_point: form_login
    form_login:
      login_path: app_login
      check_path: app_login
      enable_csrf: true
      default_target_path: app_home
    logout:
      path: app_logout
      target: app_home

```

```

# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
  - { path: ^/login, roles: PUBLIC_ACCESS }
  - { path: ^/register, roles: PUBLIC_ACCESS }
  - { path: ^/feed, roles: PUBLIC_ACCESS }

```

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.

```
/***
 * Backoffice pour la gestion des utilisateurs et des liens
 * @route("/backoffice", name="app_backoffice", methods={"GET"})
 */
#[Route('/backoffice', name: 'app_backoffice', methods: ['GET'])]
public function index(Request $request, EntityManagerInterface $entityManager): Response
{
    $this->denyAccessUnlessGranted('ROLE_USER');

    $activeTab = $request->query->get('tab', 'links');

    if (!$this->isGranted('ROLE_ADMIN') && $activeTab === 'users') {
        $activeTab = 'links';
    }

    if ($this->isGranted('ROLE_ADMIN') && !$request->query->has('tab')) {
        $activeTab = 'users';
    }
}
```

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 utilisateurs 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`.

The screenshot shows the main landing page of the Watson application. At the top, there is a navigation bar with the Watson logo on the left and a "CONNEXION" button on the right. Below the navigation bar, the heading "Bienvenue sur Watson" is displayed, followed by the subtext "Découvrez notre sélection de liens". A search bar with the placeholder "Rechercher par titre, description ou tags..." and a "RECHERCHER" button are located below the heading. The main content area consists of a grid of eight cards, each representing a different website or resource:

- Scrimba**: Partagé par admin. Cours en ligne avec IDE intégrée. Tags: #JAVASCRIPT, #REACT, #NODEJS, #API, #SQL. Button: OUVRIRE LE LIEN.
- MDN Web Docs**: Partagé par admin. Comprehensive web standards documentation. Tags: #REACT, #MYSQL. Button: OUVRIRE LE LIEN.
- W3Schools**: Partagé par admin. Web tutorials and references. Tags: #SQL, #MYSQL. Button: OUVRIRE LE LIEN.
- freeCodeCamp**: Partagé par admin. Learn to code for free. Tags: #BOOTSTRAP, #GIT. Button: OUVRIRE LE LIEN.
- CSS-Tricks**: Partagé par admin. Tips and tricks for CSS and frontend. Tags: #CSS. Button: OUVRIRE LE LIEN.
- Smashing Magazine**: Partagé par admin. Web design and development magazine. Tags: #HTML, #POSTGRESQL. Button: OUVRIRE LE LIEN.
- A List Apart**: Partagé par admin. Explores the design, development, and meaning of web content. Tags: #SQL, #TAILWIND. Button: OUVRIRE LE LIEN.
- Stack Overflow**: Partagé par admin. Programming Q&A site. Tags: #PHP, #NODEJS. Button: OUVRIRE LE LIEN.

The screenshot shows the login page of the Watson application. At the top, there is a navigation bar with the Watson logo on the left and a "CONNEXION" button on the right. The main content area features a central modal window with a green header bar containing the text "Veuillez vous connecter...". Inside the modal, there are two input fields: "NOM D'UTILISATEUR" and "MOT DE PASSE", both with placeholder text. Below the input fields is a blue "SE CONNECTER" button. At the bottom of the page, there is a footer bar with the text "Réalisé avec amour par le Groupe 6" and a link "S'inscrire".

The screenshot shows the backoffice user management interface of the Watson application. At the top, there is a navigation bar with the Watson logo on the left, a "Bonjour, admin" greeting, a "BACKOFFICE" button, and a "DÉCONNEXION" button on the right. The main content area features a central modal window titled "Backoffice" with the subtitle "Administration des utilisateurs et des liens". Inside the modal, there are two tabs: "UTILISATEURS (2)" (selected) and "LIENS (30)". Below the tabs is a blue "AJOUTER UN UTILISATEUR" button. A table lists the users:

| ID | NOM D'UTILISATEUR | RÔLES | NOMBRE DE LIENS | ACTIONS |
|----|-------------------|-----------------------|-----------------|-------------------|
| 1 | admin | ROLE_ADMIN, ROLE_USER | 30 | ÉDITER, SUPPRIMER |
| 2 | filippos | ROLE_USER | 0 | ÉDITER, SUPPRIMER |

At the bottom of the page, there is a footer bar with the text "Réalisé avec amour par le Groupe 6" and a link "S'inscrire".

Webographie

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