# Spécifications techniques

## I. Choix techniques

### A. Langages

Le développement du site nécessitera l'utilisation des langages HTML5 pour la structuration des pages, CSS3 pour le design et Javascript ES6 pour certains points de design et éléments interactifs.

Le cœur de l'application sera développé en PHP 8.X avec certains éléments utilisant Javascript pour du contenu dynamique tel que la barre de recherche.

La base de données utilisera le système MySQL et sera administrée via une page dédiée ainsi que phpmyadmin au besoin.

### B. Frameworks & Bibliothèques

Pour le développement du projet, nous utiliserons le framework Symfony 5 pour faciliter et accélérer le développement back de l'application.

La partie front utilisera le framework Angular pour traiter les données envoyées par l'API Symfony.

Enfin nous utiliserons SCSS afin de faciliter l'écriture du CSS et afin d'étendre la compatibilité front aux différents navigateurs disponibles.

## II. API

### A. Choix

Le choix d'une API s'est imposé par la volonté d'obtenir un code le plus clair et distinct possible, et donc le plus facile à comprendre et à entretenir même pour un développeur externe au projet. Nous utiliserons une API Rest car largement plus populaire et donc plus documentée, et également car utilisant le protocole HTTP.

## 1. Séparation client-serveur

Le premier avantage d'une API est la distinction entre la partie client ( front ) et la partie serveur ( back ). De ce fait, indépendamment de la manière dont le cœur de l'application est structuré, et indépendamment du code client, les deux parties pourront communiquer entre elles dès lors qu'elles utilisent le même protocole, ici HTTP, et la même architecture, ici REST.

## 2. Stateless

Dans une utilisation Rest, les API sont "stateless", ce qui signifie que la partie serveur ne sauvegarde ni les requêtes, ni les résultats qui lui auront été communiqués. De ce fait, chaque requête s'adressant au serveur se doit d'être le plus précis et déterminé possible afin de rendre le code parfaitement lisible pour un développeur tiers, limiter les risques en termes de sécurité et afin d'obtenir que et uniquement que le résultat attendu.

## 3. Cacheable

L'utilisation du cache permet à un client de pouvoir mettre en mémoire un résultat qui serait identique à un nouveau résultat reçu. Par exemple, la page d'accueil d'un site statique. Ainsi, le serveur enverra une première fois les données avec un identifiant de version, le client stockera ces données de son côté en local; Lorsqu'il revisitera la même page, le client ne fera qu'interroger le serveur sur la version des données qu'il possède, si la version est identique le client chargera les mêmes données qu'il possède de son côté. Les interactions client-serveur sont donc limitées et les temps de chargement accélérés.

## 4. Uniforme Interface

Un point important de l'utilisation d'une API Rest et la standardisation de l'interface. Comme précédemment évoqué, cela permet à un développeur tiers habitué à l'utilisation de l'architecture Rest de comprendre facilement le comportement de l'application, et cela permet également de pouvoir faire interagir deux applications totalement différentes entre elles en un minimum de code.

### 5. Layered System

Le système de couches d'une API Rest est la manière dont chaque composant de cette API agit indépendamment de l'autre. Ainsi, le fonctionnement de l'API devrait rester opaque au client, dans le sens où si celui-ci fait une requête, par exemple, pour récupérer les likes de sa photo de profil, il ne devrait obtenir que et uniquement que les likes sans avoir idée des composants autour de cet item. Ce fonctionnement permet de renforcer la sécurité et également de créer des composants indépendants faciles à entretenir et/ou remplacer.

## B. Routes

Les routes ci-après sont définies uniquement telles que nécessaires à la partie client, les méthodes nécessaires en sus à la partie administration seront notées en rouge.

| Routes | Méthodes |
|---|---|
| / | ❖ GET |
| /brands | ❖ GET<br>❖ POST<br>❖ PUT<br>❖ DELETE |
| /brands/{id} | ❖ GET<br>❖ PUT<br>❖ DELETE |
| /brands/{id}/products | ❖ GET<br>❖ POST<br>❖ PUT<br>❖ DELETE |
| /brands/{id}/products/{id} | ❖ GET<br>❖ POST<br>❖ PUT<br>❖ DELETE |
| /profile | ❖ GET<br>❖ POST<br>❖ PUT<br>❖ DELETE |

| | |
|---|---|
| /cart | ❖ GET<br>❖ POST<br>❖ PUT<br>❖ DELETE |
| /checkout | ❖ GET<br>❖ POST |

# III. Authentification ( JWT )

L'authentification à l'application sera gérée via l'utilisation du système JWT ( Json Web Token ). Le JWT est chargé de lier l'identité de l'application, l'identité du client et les autorisations qui lui sont accordées. Ce jeton sera transmis avec chaque requête communiquant avec le serveur afin d'authentifier l'utilisateur à son origine.

Un JWT est composé de 3 parties distinctes :

- Le Header, objet JSON contenant l'algorithme utilisé pour la signature du jeton.
- Le Payload, objet JSON contenant l'ensemble des informations embarquées dans le jeton.
- La Signature, résultat de l'encodage des deux parties précédentes avec l'algorithme contenu dans le Header.

Il convient toutefois de prendre certaines précautions quant à l'utilisation d'un JWT :

- Ne jamais mettre d'informations sensibles dans le Payload car le JWT est un token facilement décodable.
- Toujours valider la signature, et ne jamais entamer le décodage d'un token sans avoir fait cette étape au préalable.
- Ne jamais permettre l'utilisation alg=none ( Ou toute variation de casse du terme ), ce qui signifie que le token ne serait pas signé et donc pas vérifiable.
- Se documenter sur les injections de code à l'intérieur du header via les paramètres kid et jku.