

Application JAM

~ Application de vente de confiture ~

Promotion 2022-2023

3WAcademy

Ghinda Cristina-Maria

Document explicatif de l'

Application JAM

pour le passage du titre

Expert informatique et systèmes d'information
(3WAcademy)

Table des matières

Introduction.....	5
1. Transformation de l'application Jam Symfony en API.....	7
1.1. Architecture basée sur une API (Application Programming Interface)	7
1.1. Facteurs clés pour le choix technologique dans la réalisation d'un projet.....	8
1.2. Choix technologique pour le backend en API de l'application JAM.....	9
2. Développement d'un front-end pour consommer l'API	11
2.1. Choix technologiques de Vue.js :.....	11
2.2. Choix d'utiliser Typescript	12
3. Phases de tests de l'application Jam	13
3.1. Importance des tests dans le développement d'une application web.....	13
3.2. Types de tests envisageables pour une application web.....	13
3.2.1. Types de tests envisageables pour un backend DJANGO :	15
3.2.2. Types de tests envisageables pour un front-end Vue.js :	15
3.2.3. Les tests end-to-end (E2E).....	15
3.3. Tests mise en place sur l'application JAM :	16
3.3.1. Backend	16
3.3.2. Frontend	22
4. Gestion d'un changement de version LTS majeur	28
4.1. LTS (Long-Term Support) - notions generals.....	28
4.2. Transition vers une version LTS majeure de l'application JAM	29
4.2.1. Transition vers une version LTS majeure du front-end (Vue.js)	29
4.2.2. Transition vers une version LTS majeure du backend Django Rest.....	30
5. Implémentation d'un filtre à facettes	32
5.1. Base de données existante, création d'une nouvelle base de données et améliorations apportées	32
5.2. Filtres qui ont été mis en place sur l'application JAM :	35
6. Premier démarrage de l'application JAM	39
7. Docker.....	41
7.1. Fichiers Dockerfile front et back.....	42
7.2. Fichiers entrypoint_front.sh et entrypoint_back.sh.....	43
8. Backend DJANGO REST FRAMEWORK	44
8.1. Arborescence des fichiers.....	44

8.2.	Authentification.....	47
8.3.	Stripe	50
9.	Frontend Vue.JS-Typescript	52
9.1.	Arborescence des fichiers.....	52
9.2.	Fonctionnalités de l'application.....	55
9.3.	Vues de l'application.....	57
10.	Sécurité.....	62
10.1.	La sécurité et les frameworks	62
10.2.	La sécurité de l'application JAM	62
	Conclusion	71

Introduction

Dans le cadre de la formation « **EXPERT EN INFORMATIQUE ET SYSTEME D'INFORMATION** » proposée par 3WAcademy, pour pouvoir valider les compétences du BLOC 3 - Conception et développement d'une application informatique, une application, nommé **application Jam** doit être soumise sous forme **d'un dépôt Git** ou d'une archive Zip, excluant les dépendances. De plus, **un document explicatif au format PDF** est requis pour accompagner cette soumission.

Le présent document, **structuré en 10 chapitres**, vise à fournir des réponses détaillées aux cinq questions directrices fournies, dans le cadre de la modification de l'application "Jam", et de mettre en évidence les étapes suivies pour adapter cette application selon les directives reçues, en mettant l'accent sur les améliorations apportées, les défis relevés et les solutions mises en place. La structuration en 10 chapitres garantit une présentation claire et complète du processus de modification.

Au sein du **premier chapitre**, nous explorons initialement les nombreux atouts que procurent les backends basés sur une architecture API. Ensuite, nous dévoilons les éléments clés à considérer lors du choix d'une technologie pour la réalisation d'un projet. Enfin, nous plongeons dans la justification du choix technologique spécifique pour la transformation de notre backend en une architecture API, mettant en lumière les avantages substantiels que cette technologie apporte. Ce chapitre met en lumière les considérations essentielles qui ont guidé le choix de la technologie spécifique dans la rénovation de l'application.

Le deuxième chapitre met en avant la sélection spécifique opérée pour le frontend chargé de consommer l'API. Nous mettons en lumière les avantages significatifs de ce choix, en soulignant les raisons fondamentales qui ont motivé cette décision.

Le troisième chapitre met en évidence l'importance cruciale des tests dans le processus de développement d'une application, en mettant en avant les phases de tests qui peuvent être mises en place pour assurer le bon fonctionnement global de l'application. Nous examinons en détail la mise en place des tests sur l'application "Jam", pour le backend ainsi que pour le front-end, en décrivant de manière exhaustive le déroulement de ces tests et en ajoutant d'exemples concrets expliqués en détail.

Le chapitre quatre aborde en détail la notion de Long-Term Support (LTS) et examine la procédure de mise à jour d'une telle version dans le contexte d'une application web. Nous explorons ensuite le processus de migration vers une version LTS pour notre backend et notre frontend, en mettant en évidence les implications de cette transition.

Le chapitre cinq détaille la conception et l'implémentation d'un filtre à facettes dans l'application. En commençant par l'analyse de la base de données d'origine, nous présentons ensuite la nouvelle structure de la base de données enrichie qui permet la mise en place d'un filtre à facettes convivial et performant, ainsi que les étapes impliquées dans la construction de la nouvelle base de données.

Le chapitre six présente les étapes essentielles à suivre pour lancer avec succès l'application, tout en mettant en évidence les outils et les applications nécessaires pour assurer un démarrage sans problème. En soulignant l'importance de la configuration préalable, nous fournissons des instructions détaillées pour faciliter le processus de démarrage.

Le chapitre sept fournit une explication détaillée de Docker et son utilisation dans le contexte spécifique de l'application JAM.

Le chapitre huit met en avant le backend de l'application JAM, offrant un aperçu détaillé de l'architecture du projet et expliquant en profondeur le fonctionnement de cette infrastructure essentielle. En mettant l'accent sur les paquets installés spécifiquement pour le backend de l'application JAM, nous abordons également des aspects clés tels que l'authentification et l'intégration du module de paiement Stripe, en détaillant leur implémentation et leur fonctionnement dans le contexte de l'application.

Le chapitre neuf se concentre sur l'arborescence détaillée du frontend de l'application JAM, offrant une analyse approfondie de la structure et du fonctionnement de cette interface consommant l'API. Les améliorations apportées pour optimiser l'expérience utilisateur sont présentées et expliquées en détail. Dans ce chapitre on trouve également la présentation des différentes pages de l'application accompagnées d'explications pertinentes pour une meilleure compréhension.

Le dernier chapitre de ce document est consacré à la sécurité. Ce chapitre commence par une discussion sur l'importance de la sécurité. Le premier sous-chapitre décrit la sécurité dans le contexte des frameworks en général, soulignant les principaux principes et pratiques de sécurité qui sous-tendent le développement d'applications robustes. Ensuite nous présentons en détail les mesures de sécurité mises en place à la fois du côté backend et du côté frontend de l'application JAM, en mettant en évidence les protocoles et les fonctionnalités clés qui contribuent à garantir la protection des données et la confidentialité des utilisateurs.

L'application peut être trouvée à l'adresse : https://github.com/Cristina-MariaG/JAM_APP

1. Transformation de l'application Jam Symfony en API

1.1. Architecture basée sur une API (Application Programming Interface)

Le choix de passer une application vers une API peut apporter plusieurs avantages.

Nous allons voir par la suite quelques des avantages apportés par une application avec un backend API :

- En utilisant une API, il est possible de séparer la logique métier de l'application de l'interface utilisateur. Cela permet de développer différentes interfaces utilisateur (applications web, mobiles, IoT, etc.) qui partagent la même API, facilitant ainsi la maintenance et la mise à jour de l'application.
- Une API bien conçue permet à une application d'interagir plus facilement avec d'autres systèmes, services ou applications tierces. Cela peut être essentiel si nous allons devoir intégrer l'application avec d'autres systèmes ou si nous souhaitons ouvrir les données **et** fonctionnalités à des partenaires externes.
- Les API sont plus évolutives et flexibles en termes de gestion de la charge et d'ajout de nouvelles fonctionnalités. Il est possible de développer de nouvelles fonctionnalités de manière indépendante sans perturber le fonctionnement de l'ensemble de l'application.
- Une API bien conçue peut fournir une expérience utilisateur plus rapide et réactive, car elle permet de réduire la charge de traitement côté serveur et d'effectuer des opérations en parallèle.
- En exposant l'application sous forme d'API, il est possible de réutiliser les données et les fonctionnalités de l'application dans divers contextes, ce qui peut économiser du temps et des ressources de développement.
- Une API peut offrir un niveau de sécurité supplémentaire en permettant un contrôle plus précis sur les autorisations d'accès aux données et aux fonctionnalités. Vous pouvez mettre en place des mécanismes d'authentification et d'autorisation spécifiques à l'API.
- Avec une API, il est possible de développer des clients pour différentes plateformes (iOS, Android, applications web, etc.) tout en offrant une expérience cohérente aux utilisateurs.
- Les mises à jour de l'application peuvent être plus faciles à gérer, car elles peuvent être déployées indépendamment de l'interface utilisateur. Cela réduit le risque d'interruption du service pour les utilisateurs finaux.
- Les API sont plus faciles à mettre à l'échelle horizontalement pour gérer une augmentation du trafic. Il est possible d'ajouter de nouveaux serveurs ou ressources pour répondre aux besoins en matière de performances.

- Les architectures basées sur des API sont bien adaptées aux tendances technologiques actuelles, telles que les applications à architecture microservices, le développement sans serveur (serverless), et les architectures cloud.

1.1. Facteurs clés pour le choix technologique dans la réalisation d'un projet

Le choix des technologies pour la mise en place d'un backend dépend de plusieurs facteurs clés, notamment :

- Comprendre clairement les exigences fonctionnelles et non fonctionnelles du projet est essentiel pour choisir la bonne technologie qui répondra le mieux aux besoins du projet.
- La capacité du système à évoluer avec la croissance du projet et l'augmentation de la charge de travail est un facteur important à considérer pour assurer la durabilité du projet à long terme.
- Évaluer les coûts associés à l'adoption d'une technologie spécifique, y compris les coûts de mise en œuvre, de maintenance et de formation du personnel, est crucial pour déterminer la viabilité économique du projet.
- Choisir une technologie qui est facile à maintenir et à mettre à jour est important pour assurer la stabilité et la durabilité du projet à long terme.
- La compatibilité avec d'autres systèmes existants ou futurs peut jouer un rôle crucial dans l'intégration harmonieuse du nouveau projet dans l'écosystème informatique de l'entreprise.
- La disponibilité d'une communauté active de développeurs et d'utilisateurs peut faciliter le support technique, l'échange de bonnes pratiques et la résolution rapide des problèmes.
- Les performances du système, y compris la vitesse, la capacité de traitement et la réactivité, doivent être évaluées en fonction des besoins spécifiques du projet.
- Évaluer la complexité technique de la mise en œuvre de la technologie, ainsi que la capacité de l'équipe à gérer cette complexité, est crucial pour éviter les retards et les problèmes potentiels.
- La possibilité de faire évoluer le système en fonction des besoins changeants de l'entreprise et des utilisateurs est essentielle pour garantir la pertinence à long terme de la technologie choisie.

L'expérience de l'équipe est un aspect crucial lorsqu'il s'agit de prendre des décisions technologiques. Nous allons voir par la suite quelques raisons pour lesquelles l'expérience de l'équipe est un point important dans le processus de prise de décision technologique :

- Une équipe expérimentée est susceptible de maîtriser les technologies qu'elle utilise, ce qui peut conduire à une mise en œuvre plus efficace et à une résolution rapide des problèmes techniques.
- L'expérience de l'équipe dans l'utilisation d'une technologie particulière peut conduire à une augmentation de la productivité. Une équipe familière avec une technologie peut travailler plus efficacement, ce qui se traduit par une livraison plus rapide des fonctionnalités et une meilleure gestion des délais.
- Une équipe expérimentée est mieux équipée pour évaluer les avantages et les inconvénients des différentes technologies disponibles. Elle est en mesure de prendre des décisions éclairées en fonction des exigences du projet, des contraintes de temps et de budget, ainsi que de la complexité du système.
- Face à des défis techniques ou des problèmes inattendus, une équipe expérimentée est en mesure de gérer ces situations de manière plus efficace. Elle peut anticiper les problèmes potentiels liés à une technologie spécifique et mettre en place des stratégies pour les atténuer ou les résoudre rapidement.
- L'expérience de l'équipe peut garantir une meilleure maintenabilité et évolutivité du système. Une compréhension approfondie de la technologie permet à l'équipe de développer des solutions évolutives et faciles à entretenir, ce qui réduit les coûts de maintenance à long terme.

1.2.Choix technologique pour le backend en API de l'application JAM

Pour répondre à la demande de migration de l'application JAM, vers une architecture basée sur une API, j'ai fait le choix de me diriger vers DJANGO Rest Framework, une extension populaire et puissante du framework Django qui facilite la création de services web RESTful (Representational State Transfer) en utilisant Python.

Outre les avantages mentionnés ci-dessous, j'ai choisi ce framework pour approfondir ma compréhension et ma maîtrise de ses fonctionnalités, ainsi que pour explorer de manière approfondie ses capacités et ses possibilités.

Avantages que ce framework apporte :

- Django REST framework se distingue par ses performances élevées et sa capacité à gérer des charges de trafic importantes.
- Django REST framework offre des outils puissants qui simplifient la création de mon API REST. Un avantage significatif est représenté par les classes de sérialisation qui me permettent de convertir facilement des objets Python en formats de données comme JSON ou XML. De plus, les vues basées sur des classes facilitent la gestion des opérations CRUD (Create, Read, Update, Delete), et les fonctionnalités pour la validation des données et l'authentification sont un vrai plus.
- La sécurité est essentielle et Django REST framework le comprend bien. Il intègre des fonctionnalités solides pour protéger l'API, notamment des mécanismes

d'authentification robustes tels que l'authentification basée sur des jetons (token-based authentication) et OAuth. Il assure également une protection contre les attaques CSRF (Cross-Site Request Forgery) et autres menaces de sécurité.

- J'aime avoir le contrôle, et c'est l'un des points forts de Django REST framework. Je peux personnaliser chaque aspect de l'API pour qu'elle corresponde exactement à mes besoins spécifiques.
- Je trouve très bénéfique de travailler avec un framework qui dispose d'une grande communauté de développeurs, de nombreuses ressources en ligne, ainsi que de bibliothèques tierces et de plugins. Cela fait gagner un temps précieux dans le développement de l'API.
- La gestion des versions est essentielle pour garantir la compatibilité avec les clients existants. Avec Django REST framework on peut gérer différentes versions de mon API de manière fluide.
- L'API sera compatible avec une variété de clients, que ce soient des applications web, des applications mobiles, ou même des clients en ligne de commande.
- La possibilité de mettre en place des tests unitaires et de tests d'intégration est un atout majeur pour assurer la qualité du backend.
- L'application va évoluer avec le temps, et la possibilité de développer mon API de manière modulaire pour répondre à ces évolutions est un plus.

Pour résumer, Django REST framework est un choix judicieux afin de développer le backend de cette application. Il permet de créer une API RESTful robuste, personnalisable, tout en garantissant la sécurité, la testabilité, et en profitant de l'écosystème Django.

2. Développement d'un front-end pour consommer l'API

En ce qui concerne le développement d'un front-end qui consommera l'API Django REST, j'ai opté pour l'utilisation de Vue.js. Ce choix découle non seulement de mon désir de réaliser un projet mettant en œuvre Vue 3, et de mieux appréhender son fonctionnement, mais également de plusieurs facteurs clés que je présenterai par la suite.

2.1.Choix technologiques de Vue.js :

Le choix d'utiliser Vue.js 3 repose sur plusieurs considérations importantes :

- Vue.js est un framework JavaScript convivial. Ayant déjà une solide base en HTML, CSS et JavaScript, son apprentissage est fluide, ce qui permet d'accélérer le développement initial de l'application.
- La réactivité de Vue.js facilite le développement. La gestion automatique des mises à jour de l'interface utilisateur lorsque les données de l'API changent simplifie grandement la gestion des états dynamiques.
- Le concept de composants réutilisables permet de créer une interface utilisateur modulaire et de réutiliser efficacement des éléments d'interface dans différentes parties de notre application.
- Vue.js dispose d'une communauté active et d'un écosystème croissant de bibliothèques et de plugins. Cela nous offre de nombreuses options pour étendre les fonctionnalités de notre application de manière efficace.
- L'interface en ligne de commande de Vue (Vue CLI) offre des outils puissants pour faciliter le développement, le test et le déploiement de notre application Vue.js. Elle simplifie considérablement le cycle de vie du projet.
- Vue.js est suffisamment souple pour s'adapter à nos besoins. Que nous développions une petite application simple ou une application web plus complexe, Vue.js peut être ajusté en fonction de nos exigences.
- La documentation de Vue.js est extrêmement complète, ce qui facilite la recherche de solutions aux problèmes rencontrés et la résolution de questions techniques.
- Vue.js offre de bonnes performances grâce à ses mécanismes d'optimisation. Cela garantira une expérience utilisateur fluide, même lorsque l'application deviendra plus complexe.

2.2.Choix d'utiliser Typescript

Le choix d'utiliser TypeScript est motivé par plusieurs facteurs et avantages inhérents à cette technologie :

- TypeScript est un sur-ensemble de JavaScript qui ajoute la vérification statique des types. Cela signifie que nous pouvons détecter et corriger les erreurs potentielles au moment de la compilation, ce qui améliore considérablement la fiabilité de notre code. Dans le contexte de la consommation d'une API, cela réduit le risque de bugs liés à la manipulation incorrecte des données reçues de l'API.
- TypeScript offre une meilleure aide à la saisie, une autocomplétion plus précise et une documentation intégrée dans les éditeurs de code. Cela accélère le développement en réduisant les erreurs de saisie et en offrant une expérience de développement plus fluide.
- À mesure que notre application se développe, la maintenance devient un enjeu majeur. TypeScript facilite la gestion de projets de grande envergure grâce à une meilleure organisation du code, une réduction des conflits de noms et une documentation plus claire.
- TypeScript améliore la lisibilité du code pour les développeurs qui travaillent sur le projet, ce qui facilite la collaboration. En outre, les types explicites rendent le code plus compréhensible pour toute l'équipe de développement.
- TypeScript est compatible avec JavaScript, ce qui signifie que nous pouvons progressivement ajouter du code TypeScript à notre projet existant. Cela nous donne la flexibilité d'adopter TypeScript à notre propre rythme.
- TypeScript bénéficie de la popularité croissante et de l'adoption généralisée. Il est largement pris en charge par des outils, des bibliothèques et des frameworks, y compris Vue.js, ce qui garantit une compatibilité harmonieuse avec notre stack technologique actuelle.
- En détectant les erreurs au moment de la compilation, TypeScript réduit les risques d'erreurs coûteuses et de comportements inattendus dans notre application, ce qui contribue à la robustesse de notre front-end.

En somme, l'ajout de TypeScript à notre front-end basé sur Vue.js renforce la sécurité, la productivité, la maintenance, la collaboration et la qualité de notre code, tout en améliorant la fiabilité de notre application. Je suis convaincu que cette combinaison nous permettra de développer un front-end exceptionnel pour notre application qui consomme une API.

3. Phases de tests de l'application Jam

3.1.Importance des tests dans le développement d'une application web

Pour assurer la qualité et la fiabilité d'une application il est essentiel de mettre en place un processus de test rigoureux.

Les tests jouent un rôle crucial dans le développement d'applications web, offrant divers avantages significatifs qui contribuent à la fiabilité, à la qualité et à la robustesse d'une application. Nous allons voir par la suite quelques raisons pour lesquelles les tests sont importants pour une application web.

La fiabilité du code : Les tests aident à identifier les bogues et les erreurs potentiels dans le code, ce qui garantit la fiabilité et la robustesse de l'application, réduisant ainsi les risques de dysfonctionnements et d'interruptions imprévues.

L'Assurance de la qualité : Les tests permettent de s'assurer que l'application fonctionne conformément aux spécifications définies, garantissant ainsi une meilleure qualité du produit final et une expérience utilisateur optimale.

La maintenance facilitée : Les tests facilitent la détection précoce des problèmes, ce qui permet de les résoudre rapidement et efficacement. Cela facilite également la maintenance continue de l'application et réduit les coûts associés à la résolution de problèmes complexes à un stade ultérieur du développement.

La réduction des coûts à long terme : L'investissement initial dans les tests peut contribuer à réduire les coûts à long terme en minimisant les erreurs et les problèmes qui pourraient nécessiter des corrections coûteuses une fois l'application déployée.

L'agilité du développement : Les tests automatisés permettent d'identifier rapidement les problèmes et de les résoudre efficacement, ce qui favorise un processus de développement agile et itératif, permettant ainsi aux équipes de livrer des fonctionnalités de manière plus rapide et efficace.

L'amélioration de la sécurité : Les tests peuvent contribuer à identifier les vulnérabilités de sécurité potentielles, ce qui permet de renforcer la sécurité de l'application en identifiant et en éliminant les failles de sécurité dès les premières phases du développement.

Les tests jouent, donc, un rôle crucial pour assurer la fiabilité, la qualité et la sécurité d'une application web, ce qui contribue à une meilleure expérience utilisateur, tout en favorisant un développement agile et rentable à long terme.

3.2.Types de tests envisageables pour une application web

Pour une application web, différents types de tests sont généralement utilisés pour garantir sa qualité, sa fiabilité et son bon fonctionnement. Ces tests incluent :

- **Tests unitaires :** Ils vérifient que des parties spécifiques du code (comme des fonctions ou des modules individuels) fonctionnent correctement. Les tests unitaires permettent de détecter rapidement les erreurs au niveau du code de base.
- **Tests d'intégration :** Ils vérifient que différentes parties de l'application fonctionnent ensemble sans problème. Ces tests sont essentiels pour garantir que les différentes fonctionnalités de l'application interagissent correctement les unes avec les autres.
- **Tests fonctionnels :** Ils vérifient que l'application fonctionne conformément aux spécifications fonctionnelles. Ces tests sont axés sur les scénarios utilisateur et s'assurent que l'application répond correctement aux actions et aux entrées des utilisateurs.
- **Tests de rendu :** Assure que les composants sont rendus correctement en fonction des données fournies, en vérifiant le rendu des éléments visuels et des interfaces utilisateur.
- **Tests de navigation :** Vérifie la navigation entre les différentes vues et composants pour garantir que les routes fonctionnent correctement et que les redirections se produisent comme prévu.

En ce qui concerne la sécurité, des différents types de tests peuvent être mise en place :

- **Tests d'intrusion :** Ces tests sont conçus pour simuler les attaques de pirates informatiques sur l'application afin de détecter les failles de sécurité potentielles et les vulnérabilités. Ils peuvent inclure des tests tels que l'injection SQL, les attaques par déni de service, et d'autres formes courantes d'attaques.
- **Tests de vulnérabilité :** Ces tests sont axés sur l'identification des vulnérabilités connues dans l'application, telles que des failles de sécurité connues dans les bibliothèques tierces, les plugins ou les composants utilisés dans le développement de l'application.
- **Tests de gestion des accès :** Ces tests vérifient que les contrôles d'accès et les autorisations sont correctement mis en œuvre dans l'application, garantissant que seules les personnes autorisées peuvent accéder aux fonctionnalités et aux données sensibles.
- **Tests de sécurité des données :** Ces tests évaluent la sécurité des données au repos et en transit, en s'assurant que les données sensibles sont correctement cryptées et protégées contre les accès non autorisés.

En adoptant une approche complète et équilibrée qui inclut ces différents types de tests, le fonctionnement d'une application web se fera d'une manière fiable, offrant une bonne expérience utilisateur et répondant aux attentes des utilisateurs.

3.2.1. Types de tests envisageables pour un backend DJANGO :

Pour tester un backend DJANGO on peut prévoir des différents types de tests :

- **Tests unitaires** : pour vérifier les fonctionnalités spécifiques des différentes parties de votre application Django, telles que les modèles, les vues et les formulaires, pour assurer qu'elles fonctionnent comme prévu.
- **Tests d'intégration** : pour vérifier l'interaction entre les différentes parties de l'application Django, comme la communication entre les vues et les modèles, pour garantir que l'application fonctionne de manière fluide dans son ensemble.
- **Tests fonctionnels** : pour simuler les interactions réelles avec l'application en testant les scénarios utilisateur courants, en vous assurant que toutes les fonctionnalités et les flux de travail fonctionnent correctement.
- **Tests de performances** : pour évaluer les performances de l'application en testant la vitesse de chargement, la capacité de traitement des requêtes et d'autres aspects critiques pour garantir une expérience utilisateur optimale, même en cas de charges élevées.

3.2.2. Types de tests envisageables pour un front-end Vue.js :

- **Tests unitaires des composants** : pour vérifier le bon fonctionnement des composants Vue.js individuellement en simulant des interactions avec des données simulées ou des mocks.
- **Tests d'intégration des composants** : pour vérifier que les composants interagissent correctement les uns avec les autres, en testant les flux de données et les communications entre les composants.
- **Tests end-to-end (E2E)** : pour vérifier le bon fonctionnement de l'application dans son ensemble, simulant les actions d'un utilisateur réel.
- **Tests de rendu** : pour assurer que les composants sont rendus correctement en fonction des données fournies, en vérifiant le rendu des éléments visuels et des interfaces utilisateur.
- **Tests de navigation** : pour vérifier la navigation entre les différentes vues et composants pour garantir que les routes fonctionnent correctement et que les redirections se produisent comme prévu.

3.2.3. Les tests end-to-end (E2E)

Dans le cadre des tests end-to-end, on peut réaliser des différents tests :

- **Les tests de navigation** : Vérifient que la navigation entre différentes pages et sections de l'application se déroule sans problème, et que les URL et les chemins de navigation fonctionnent correctement.
- **Les tests d'interaction** : Simulent des interactions utilisateur réelles telles que le remplissage de formulaires, le clic sur des boutons, le défilement, etc., afin de vérifier que les fonctionnalités interactives de l'application se comportent comme prévu.
- **Les tests de charge** : Évaluent les performances de l'application en simulant une charge élevée, et vérifient que l'application reste réactive et fonctionne correctement même sous une charge de travail importante.
- **Tests de compatibilité** : Vérifient que l'application fonctionne de manière cohérente sur différents navigateurs et dispositifs, en s'assurant que l'expérience utilisateur reste homogène quel que soit l'environnement d'utilisation.
- **Tests de bout en bout basés sur des scénarios** : Simulent des scénarios utilisateur réels complexes pour vérifier que les fonctionnalités clés de l'application fonctionnent de manière fluide du début à la fin.

3.3. Tests mise en place sur l'application JAM :

Pour l'application Jam réalisé avec Django REST et Vue.js 3 des tests des vues (API Endpoint), des tests unitaires et d'intégration ont été mis en place.

3.3.1. Backend

Avec les API REST c'est possible de développer très rapidement des tests fonctionnels complets qui ont 100% de couverture.

Le framework Django REST offre une infrastructure solide pour effectuer des tests sur les API RESTful créées avec Django.

Sur la documentation en ligne de Django REST framework (DRF) on a une section entière dédié aux tests des views API (<https://www.django-rest-framework.org/api-guide/testing/>).

Voici comment les tests fonctionnent dans le contexte de Django REST Framework :

1. **Configuration des tests** : Les classes de tests sont créés en héritant de `APITestCase` ou `TestCase` pour les tests d'API. Ces classes fournissent des méthodes d'assistance pour effectuer des requêtes HTTP et valider les réponses.
2. **Création de cas de test** : Nous créons de test pour vérifier différents aspects de l'API, tels que la création, la récupération, la mise à jour et la suppression de ressources. Il est possible de tester les autorisations, la validation des données, les réponses HTTP, etc.

3. **Exécution des tests :** Les tests sont exécutés à l'aide d'une commande exécutée dans le répertoire racine du projet Django. Cette commande exécute tous les tests définis dans les fichiers de tests, en s'assurant que chaque cas de test est vérifié et que les assertions définies réussissent. Les étapes à suivre pour démarrer les tests du backend sont décrites dans le fichier README.md (à voir le chapitre 6, pour plus de détails).
4. **Validation des résultats :** La vérification des résultats des tests assure que chaque aspect de l'API fonctionne comme prévu. Cela inclut la vérification des codes de statut HTTP, des données de réponse, des messages d'erreur et de tout comportement spécifique de l'API.

En utilisant les fonctionnalités de test de Django REST Framework, nous pouvons nous assurer que l'API fonctionne correctement, que les erreurs sont correctement gérées et que les cas de bord sont pris en compte. Cela garantit la fiabilité et la robustesse de l'API, ce qui est crucial pour le développement d'une application solide et stable.

Il faut préciser que Django crée une base de données de test distincte pour l'exécution des tests, afin de s'assurer que les tests n'interfèrent pas avec les données de la base de données de production ou de développement.

Lorsque les tests sont exécutés, le framework crée automatiquement une base de données de test en utilisant le moteur de base de données configuré pour les tests. La base de données de test est créée à partir de zéro ou en fonction des migrations de base de données définies dans l'application. Django configure ensuite la base de données de test pour chaque cas de test et l'utilise pour exécuter les tests définis.

Après l'exécution de chaque test, la base de données de test est nettoyée pour éliminer toutes les modifications apportées pendant les tests. Cela garantit que chaque exécution de test est effectuée dans un environnement propre et isolé, ce qui permet des tests reproductibles et fiables sans affecter la base de données de production ou de développement.

Dans l'application jam-backend, une fois que la commande pour tous les tests (on peut aussi exécuter chaque test à la fois) est exécutée nous pouvons voir dans le terminal des logs comme les suivants :

```
$ docker exec jam-back python3.10 manage.py test back_app/tests -v 2
Creating test database for alias 'default' ('file:memorydb_default?mode=memory&cache=shared')...
Operations to perform:
  Synchronize unmigrated apps: corsheaders, messages, rest_framework, rest_framework_simplejwt, staticfiles
  Apply all migrations: admin, auth, back_app, contenttypes, sessions
Synchronizing apps without migrations:
  Creating tables...
  Running deferred SQL...
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying back_app.0001_initial... OK
  Applying back_app.0002_alter_productingredient_quantity... OK
  Applying back_app.0003_alter_product_ingredients... OK
  Applying sessions.0001_initial... OK
System check identified no issues (0 silenced).
test create checkout and order and checkout_success_put (test_checkout.TestCheckoutSessionProducts) ... DEBUG 2023-10-28 14:38:43,292 jam.post:24 - Start CheckoutCollection post
```

```
test_get_user (test_user.TestUser) ... ok
test_getting_unexisting_user (test_user.TestUser) ... WARNING 2023-10-28 14:38:46,972 django.request.log_response:224 - Not Found: /login/
ok
test_getting_user_with_wrong_password (test_user.TestUser) ... WARNING 2023-10-28 14:38:47,164 django.request.log_response:224 - Unauthorized: /login/
ok
test_inscription_user (test_user.TestUser) ... DEBUG 2023-10-28 14:38:47,167 jam.post:55 - Start Collection post user
ok

-----
Ran 21 tests in 4.327s

OK
Destroying test database for alias 'default' ('file:memorydb_default?mode=memory&cache=shared')...
```

Ces logs correspondent à la création de la base de données de test, à la migration des modèles, à l'installation des fixtures, à l'exécution des tests et à la fin à la destruction de la base de données de test.

Par la suite, nous avons un exemple d'un test d'API pour le backend.

```
from django.test import TestCase
import logging

logger = logging.getLogger("jam")

class TestUser(TestCase):
    fixtures = ["initial_data_for_SQL_test_DB.json"]

    def setUp(cls):
        super().setUp()

    def tearDown(self):
        super().tearDown()

    def test_get_user(self):
        data = {
```

```

        "email": "admin@admin.com",
        "password": "adminlovejam",
    }
    response_post = self.client.post(
        "/login/",
        data=data,
    )
    self.assertEqual(response_post.status_code, 200)
    self.assertIn("token", response_post.data)
    self.assertIn("refresh_token", response_post.data["token"])
    self.assertIn("access_token", response_post.data["token"])
    self.assertEqual(response_post.data["user"]["email"], data["email"])
    self.assertEqual(response_post.data["user"]["role"], "admin")

def test_inscription_user(self):
    response_post = self.client.post(
        "/inscription/",
        data={
            "email": "arc_cris@yahoo.com",
            "password": "crislovejam",
        },
    )

    self.assertEqual(response_post.status_code, 200)

def test_creating_existing_user(self):
    response_post = self.client.post(
        "/inscription/",
        data={
            "email": "admin@admin.com",
            "password": "adminlovejam",
        },
    )

    self.assertEqual(response_post.status_code, 403)

def test_getting_unexisting_user(self):
    response_post = self.client.post(
        "/login/",
        data={
            "email": "admin_test@admin.com",
            "password": "adminlovejam",
        },
    )

    self.assertEqual(response_post.status_code, 404)

def test_getting_user_with_wrong_password(self):
    response_post = self.client.post(

```

```

        "/login/",
        data={
            "email": "admin@admin.com",
            "password": "admindoesnotlovejam",
        },
    )

    self.assertEqual(response_post.status_code, 401)

```

On peut voir que cette classe de test en Python utilise le module `unittest` de Django. La classe `TestUser` hérite de `django.test.TestCase`, ce qui permet l'exécution de tests unitaires sur des fonctionnalités spécifiques de l'API de Django. Voici une explication du code ligne par ligne :

- `from django.test import TestCase` : Cette ligne importe la classe `TestCase` du module `django.test`, qui fournit des fonctionnalités pour l'écriture et l'exécution de tests unitaires.
- `import logging` : Cette ligne importe le module `logging` de Python, qui permet d'enregistrer des messages de journalisation pour un suivi et un débogage ultérieurs.
- `logger = logging.getLogger("jam")` : Cette ligne crée une instance de logger nommée "jam", qui peut être utilisée pour enregistrer des événements ou des messages spécifiques lors de l'exécution des tests.
- `class TestUser(TestCase) :` : Cette ligne définit la classe de test `TestUser` qui hérite de la classe `TestCase`.
- `fixtures = ["initial_data_for_SQL_test_DB.json"]` : Cette ligne spécifie les fichiers de fixtures à charger avant l'exécution des tests. Les fixtures fournissent des données initiales pour les tests de base de données.
- `def setUp(cls) :` : Cette méthode est appelée avant l'exécution de chaque test. Elle peut être utilisée pour initialiser des variables ou des ressources nécessaires pour les tests.
- `def tearDown(self) :` : Cette méthode est appelée après l'exécution de chaque test. Elle est utilisée pour nettoyer les ressources ou les modifications apportées pendant les tests.
- **Test d'authentification d'un utilisateur** (`def test_get_user(self) :`) Cette méthode définit un test unitaire spécifique. Dans ce cas, il teste la fonctionnalité d'obtention d'un utilisateur à partir d'un point de terminaison API de connexion.

Nous allons examiner en détail ce test :

`response_post = self.client.post("/login/", data=data)` : Cette ligne envoie une requête POST à l'URL `/login/` avec les données spécifiées.

`self.assertEqual(response_post.status_code, 200)` : Cette ligne vérifie si le code d'état de la réponse est 200, ce qui indique que la requête a réussi.

`self.assertIn("token", response_post.data) :` Cette ligne vérifie si la clé "token" est présente dans les données de la réponse.

`self.assertIn("refresh_token", response_post.data["token"]) :` Cette ligne vérifie si la clé "refresh_token" est présente dans les données du champ "token" de la réponse.

`self.assertIn("access_token", response_post.data["token"]) :` Cette ligne vérifie si la clé "access_token" est présente dans les données du champ "token" de la réponse.

`self.assertEqual(response_post.data["user"]["email"], data["email"]) :` Cette ligne vérifie si l'e-mail dans les données de l'utilisateur de la réponse correspond à l'e-mail spécifié dans les données.

`self.assertEqual(response_post.data["user"]["role"], "admin") :` Cette ligne vérifie si le rôle de l'utilisateur dans les données de la réponse est "admin".

- **Test d'inscription d'utilisateur** (`def test_inscription_user(self):`)

Ce test vérifie si un nouvel utilisateur peut s'inscrire avec succès en envoyant une requête POST à l'URL `"/inscription/"` avec une adresse e-mail et un mot de passe. Il vérifie si le code de statut de la réponse est 200, indiquant une inscription réussie.

- **Test de création d'un utilisateur existant** (`def test_creating_existing_user(self):`)

Ce test vérifie si la tentative d'inscription d'un utilisateur existant échoue en envoyant une requête POST avec les mêmes identifiants d'utilisateur déjà existants. Il vérifie si le code de statut de la réponse est 403, indiquant un accès interdit.

- **Test d'obtention d'un utilisateur inexistant** (`def test_getting_unexisting_user(self):`)

Ce test vérifie si la tentative de connexion d'un utilisateur inexistant échoue en envoyant une requête POST à l'URL `"/login/"` avec des identifiants d'utilisateur qui n'existent pas dans la base de données. Il vérifie si le code de statut de la réponse est 404, indiquant que la ressource n'a pas été trouvée.

- **Test d'obtention d'un utilisateur avec un mot de passe incorrect** (`def test_getting_user_with_wrong_password(self):`)

Ce test vérifie si la tentative de connexion avec un mot de passe incorrect échoue en envoyant une requête POST à l'URL `"/login/"` avec une adresse e-mail d'utilisateur valide mais un mot de passe incorrect. Il vérifie si le code de statut de la réponse est 401, indiquant une erreur d'authentification.

Ces tests garantissent que les fonctionnalités d'inscription et de connexion utilisateur fonctionnent correctement et renvoient les codes de statut HTTP appropriés en fonction des

résultats des opérations. Il définit et exécute donc des tests pour vérifier si le point de terminaison d'API de connexion fonctionne correctement en renvoyant les bonnes données pour un utilisateur spécifié.

3.3.2. Frontend

Des tests unitaires pour les composants Vue.js en utilisant Vue Test Utils et Vitest ont été écrits.

Vue Test Utils est une bibliothèque officielle pour tester les composants Vue.js. Elle fournit un ensemble d'utilitaires pour créer des tests unitaires et d'intégration complets pour les composants Vue. Voici quelques-unes de ses fonctionnalités clés :

1. **Montage des composants** : Vue Test Utils permet de monter les composants Vue individuellement ou avec d'autres composants, ce qui facilite les tests isolés ou les tests de composants imbriqués.
2. **Manipulation du DOM** : Il offre des outils pour simuler des événements DOM et observer le rendu du DOM après l'exécution d'une action, ce qui est utile pour vérifier les effets des interactions utilisateur.
3. **Assertions Vue** : La bibliothèque fournit des méthodes d'assertion spécifiques à Vue pour vérifier les états des composants, les propriétés, les données, les événements, etc.
4. **Prise en charge de Vue 3** : Elle est entièrement compatible avec Vue 3 et prend en charge les fonctionnalités avancées de la dernière version du framework Vue.js.
5. **Intégration avec des bibliothèques de test** : Vue Test Utils s'intègre facilement avec des bibliothèques de test populaires telles que Jest, Mocha, Karma, etc., offrant ainsi une flexibilité pour les développeurs dans le choix de leurs outils de test.

En combinant Vue Test Utils avec d'autres bibliothèques de test et des outils d'assertion, nous pouvons créer des tests robustes pour les composants Vue.js, assurant ainsi la fiabilité et la stabilité des applications.

Dans ce projet, nous combinons Vue Test Utils avec Vitest.

Vitest est une bibliothèque de test JavaScript légère et simple conçue pour faciliter l'écriture de tests unitaires et d'intégration. Quelques caractéristiques clés de cette bibliothèque sont :

1. **Syntaxe simple** : Vitest utilise une syntaxe simple et facile à comprendre, ce qui la rend accessible même pour les développeurs débutants.
2. **Assertions claires** : Il fournit des fonctions d'assertion claires telles que `expect` pour faciliter la vérification des résultats attendus par rapport aux résultats réels.
3. **Simulations d'appels** : Vitest permet de simuler facilement des appels de fonctions, ce qui est particulièrement utile pour tester des fonctions qui dépendent d'appels asynchrones ou d'interactions avec des API externes.
4. **Prise en charge de l'asynchronisme** : Il gère les cas de test asynchrones de manière simple en permettant l'utilisation de `async/await` ou de la fonction `then` pour gérer les promesses.
5. **Intégration facile** : Vitest s'intègre bien avec d'autres bibliothèques et frameworks JavaScript populaires, ce qui en fait un choix polyvalent pour les projets de développement web.

Globalement, ViTest est une bibliothèque de test légère mais robuste qui permet de créer et d'exécuter des tests unitaires de manière efficace, en mettant l'accent sur la lisibilité et la simplicité.

Pour exécuter tous les tests unitaires, il faut exécuter dans le container du front

npm run test, comme dans l'image suivante :

```
$ docker exec jam-front npm run test

> front@0.0.0 test
> vitest

RUN v0.34.6 /front

[@vue/compiler-sfc] `defineProps` is a compiler macro and no longer needs to be imported.

✓ src/components/__tests__/NoDataFound.spec.ts (1 test) 656ms
✓ src/stores/__tests__/auth.spec.ts (6 tests) 380ms
✓ src/components/__tests__/HRComponent.spec.ts (1 test) 633ms
✓ src/stores/__tests__/cart.spec.ts (8 tests) 30ms
✓ src/components/__tests__/filterComponent/ContenantType.spec.ts (4 tests) 465ms
✓ src/components/__tests__/filterComponent/Category.spec.ts (3 tests) 441ms
✓ src/components/__tests__/filterComponent/Brand.spec.ts (2 tests) 357ms
✓ src/components/__tests__/filterComponent/AvailableStock.spec.ts (1 test) 221ms
stdout | src/views/__tests__/Login.spec.ts > test login > mount Page login

✓ src/views/__tests__/Login.spec.ts (5 tests) 815ms
✓ src/composables/__tests__/filters.spec.ts (13 tests) 47ms
✓ src/composables/__tests__/validateForm.spec.ts (7 tests) 59ms
✓ src/components/__tests__/filterComponent/OrderByPrice.spec.ts (2 tests) 240ms

Test Files 13 passed (13)
Tests 55 passed (55)
Start at 10:04:22
Duration 37.05s (transform 24.30s, setup 77.97s, collect 72.18s, tests 3.72s, environment 123.66s, prepare 21.43s)
```

Nous pouvons voir dans les logs une série de résultats de tests, indiquant les fichiers et les tests qui ont été exécutés. Chaque ligne commence par un caractère ✓, indiquant que le test a réussi, suivi du chemin du fichier et du nombre de tests ainsi que la durée de chaque test.

La sortie indique que les 13 fichiers de test ont tous réussi, avec un total de 55 tests réussis. L'exécution a commencé à 10h04 et 22 secondes, et a duré au total 37,05 secondes, avec des phases telles que la transformation, la configuration, la collecte, les tests réels, la préparation de l'environnement et les préparatifs prenant des durées spécifiques.

Nous avons la possibilité d'exécuter un seul test (à voir README.md) avec la commande :

```
npm run test:unit fileName
```

Dans l'image suivante nous avons un exemple d'exécution d'un seul test dans le front :

```

$ docker exec jam-front npm run test:unit Category.spec.ts

> front@0.0.0 test:unit
> vitest Category.spec.ts

RUN v0.34.6 /front

✓ src/components/__tests__/filterComponent/Category.spec.ts (3 tests) 244ms

Test Files  1 passed (1)
Tests       3 passed (3)
Start at   10:00:43
Duration   28.72s (transform 8.61s, setup 6.67s, collect 4.17s, tests 244ms, environment 13.95s, prepare 1.99s)

```

La commande `$ docker exec jam-front npm run test:unit Category.spec.ts` exécute avec succès le test unitaire du fichier `Category.spec.ts` dans le répertoire des composants de filtrage, où trois tests ont réussi en 244 millisecondes.

Nous avons la possibilité de voir le taux de couverture de nos tests avec la commande :

```
npm run coverage
```

Après ce que tous les tests sont exécutés, une table avec le rapport de couverture de code fournit des détails sur la couverture du code source, indiquant le pourcentage de déclarations, de branches, de fonctions et de lignes couvertes par les tests. Dans ce cas, le pourcentage de déclarations, de branches, de fonctions et de lignes couvertes est de 79,89 %. Le rapport précise également que le fichier "HrComponent.vue" a une couverture de 100 % pour les déclarations, les branches, les fonctions et les lignes, ce qui signifie que toutes les parties de ce fichier ont été couvertes par les tests.

% Coverage report from v8					
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	79.89	88.07	59.49	79.89	
components	98.97	100	75	98.97	
HrComponent.vue	100	100	100	100	
JamCard.vue	98.75	100	75	98.75	48
NoDataFound.vue	100	100	100	100	
...lterComponents	98.93	100	33.33	98.93	
...ableStock.vue	95.65	100	0	95.65	9
Brand.vue	98.21	100	50	98.21	12
...Component.vue	100	100	0	100	
...enantType.vue	100	100	0	100	
...Component.vue	100	100	100	100	
composables	94.11	94.28	100	94.11	
filters.ts	91.75	90.47	100	91.75	55-58,67-70
validateForm.ts	100	100	100	100	
functions	85.71	14.28	50	85.71	
i18nMocks.ts	85.71	14.28	50	85.71	7,9,11
helpers	58.69	100	0	58.69	
api.ts	100	100	100	100	
error-handler.ts	13.63	100	0	13.63	4-22
...s/repositories	40	100	34.61	40	
...repository.ts	37.93	100	33.33	37.93	9-17,20-28
...repository.ts	50	100	50	50	8-15

Voici un exemple de test unitaire du front :


```

import { useAuthStore } from '../auth'
import { authRepository } from '@helpers/api'
import { expect, describe, it, vi, beforeEach } from 'vitest'
import { setActivePinia, createPinia } from 'pinia'

vi.mock('@helpers/api', () => ({
  authRepository: {
    login: vi.fn(),
    inscription: vi.fn()
  }
}))
let authStore: any

describe('Auth Store', () => {
  beforeEach(() => {
    // creates a fresh pinia and makes it active, so it's automatically picked
    // up by any useStore() call
    // without having to pass it to it: `useStore(pinia)`
    setActivePinia(createPinia())
    authStore = useAuthStore()
  })

  it('checks initial state values', () => {
    expect(authStore.accessToken).toBe('')
    expect(authStore.refreshToken).toBe('')
    expect(authStore.email).toBe('')
  })

  it('checks if user is authenticated', () => {
    expect(authStore.isAuthenticated).toBe(false) // Initially, user should not be
    authenticated
    authStore.accessToken = 'test_token'
    expect(authStore.isAuthenticated).toBe(true) // User should be authenticated after
    setting the token
  })

  it('checks login action', async () => {
    const mockPayload = { email: 'test@example.com', password: 'password' }
    const mockResponse = {
      token: { access_token: 'test_access_token', refresh_token:
'test_refresh_token' },
      user: { email: 'test@example.com' }
    }
    authRepository.login.mockResolvedValueOnce(mockResponse)

    const result = await authStore.login(mockPayload)
    expect(result).toBe(true)
    expect(authStore.accessToken).toBe('test_access_token')
    expect(authStore.refreshToken).toBe('test_refresh_token')
  })
})

```

```

    expect(authStore.email).toBe('test@example.com')
  })

  it('checks setRefreshedTokens action', async () => {
    const mockToken = { access_token: 'new_access_token', refresh_token:
'new_refresh_token' }
    await authStore.setRefreshedTokens(mockToken)
    expect(authStore.accessToken).toBe('new_access_token')
    expect(authStore.refreshToken).toBe('new_refresh_token')
  })

  it('checks inscription action', async () => {
    const mockPayload = { email: 'test@example.com', password: 'password' }
    authRepository.inscription.mockResolvedValueOnce()

    const result = await authStore.inscription(mockPayload)
    expect(result).toBe(true)
  })

  it('checks logout action', () => {
    authStore.accessToken = 'test_token'
    authStore.refreshToken = 'test_refresh_token'
    authStore.email = 'test@example.com'
    authStore.logout()
    expect(authStore.accessToken).toBe('')
    expect(authStore.refreshToken).toBe('')
    expect(authStore.email).toBe('')
  })
})

```

Ce fichier contient une suite de tests unitaires visant à évaluer le comportement de l'entrepôt d'authentification (authStore) dans notre front Vue.js. Pour ce faire, il utilise plusieurs bibliothèques de test, notamment vi pour les assertions et les simulations, ainsi que pinia pour la gestion de l'état de l'application.

Le premier test vérifie les valeurs par défaut de l'état de l'authentification pour s'assurer que les propriétés comme accessToken, refreshToken et email sont initialisées correctement. Ensuite, il teste si la propriété isAuth est correctement mise à false initialement, et devient true après avoir défini le jeton d'authentification dans authStore.

Ensuite, le test simule des appels à l'API d'authentification à l'aide de fonctions fictives login et inscription de l'objet authRepository. Il vérifie si les actions login, setRefreshedTokens, et inscription de l'entrepôt d'authentification réagissent de manière appropriée aux réponses simulées de l'API, en s'assurant que les propriétés d'authStore sont mises à jour en conséquence.

Enfin, le dernier test vérifie si l'action `logout` réinitialise correctement les propriétés d'authentification dans `authStore` après que l'utilisateur s'est déconnecté. Globalement, ces tests assurent le bon fonctionnement des fonctionnalités d'authentification de l'application et garantissent que les interactions avec l'API se déroulent comme prévu.

4. Gestion d'un changement de version LTS majeur

4.1.LTS (Long-Term Support)- notions generals

Dans le contexte du développement de logiciels et d'applications, une version LTS (Long-Term Support) est une version particulière d'un logiciel ou d'une application qui bénéficie d'un support prolongé et de mises à jour de sécurité pendant une période étendue, généralement sur plusieurs années.

Les versions LTS sont particulièrement importantes dans le contexte des systèmes d'exploitation, des frameworks de développement et des plates-formes logicielles utilisées par de nombreuses entreprises et organisations, car elles offrent une base solide et stable pour le développement et l'utilisation à long terme.

La transition vers une version majeure LTS d'une application web doit être soigneusement planifiée et exécutée pour assurer la continuité du service.

Il y a plusieurs étapes générales à suivre lors de la transition vers une version majeure LTS d'une application web.

- **Analyse approfondie de la version actuelle :**

Évaluer les caractéristiques et les fonctions clés de l'application web existante.

- **Évaluation de la version LTS :**

Examiner attentivement les changements et les améliorations offerts par la version LTS pour comprendre son impact sur l'application.

- **Planification de la transition :**

Élaborer un plan détaillé en identifiant les étapes clés de la transition et en définissant des objectifs clairs.

- **Tests rigoureux :**

Effectuer des tests complets pour évaluer la compatibilité de l'application avec la nouvelle version LTS et identifier tout dysfonctionnement potentiel.

- **Communication avec les parties prenantes :**

Informar les parties prenantes de la transition imminente, en fournissant des détails sur les avantages de la mise à niveau et en clarifiant les éventuels impacts sur l'expérience utilisateur.

- **Mise en œuvre de la transition :**

Mettre en place une fenêtre de maintenance pour effectuer la mise à niveau de manière transparente et minimiser les perturbations pour les utilisateurs finaux.

- **Tests post-mise à niveau :**

Effectuer des tests approfondis après la mise à niveau pour garantir la stabilité et la fonctionnalité de l'application dans son nouvel environnement.

- **Formation et support continu :**

Offrir une formation adéquate pour familiariser les utilisateurs avec les nouvelles fonctionnalités et assurer un support technique pour résoudre tout problème éventuel après la mise à niveau.

4.2. Transition vers une version LTS majeure de l'application JAM

La transition vers une version LTS majeure d'une application web réalisée avec Django et Vue.js nécessite une planification soignée pour assurer la stabilité et la compatibilité de l'application.

Car notre Application JAM est construite avec un backend en API et un frontend qui consommera l'API, nous allons voir les étapes qui sont prévues à suivre pour organiser une transition vers une version majeure de LTS de notre front-end Vue.js et après les étapes prévues à suivre pour le backend Django Rest Framework.

4.2.1. Transition vers une version LTS majeure du front-end (Vue.js)

Les étapes qui sont prévues à suivre pour organiser une transition vers une version majeure de LTS de notre front-end Vue.js sont :

- **Identification de la version cible LTS de Vue.js :** Identification de la version LTS de Vue.js vers laquelle nous souhaitons migrer.
- **Etudier et comprendre les différences et changements entre les versions**
- **Mise à jour des dépendances :** Mise à jour des dépendances du projet, y compris Vue.js et tous les plugins tiers, pour qu'ils soient compatibles avec la version cible de Vue.js.
- **Mise à jour graduelle des composants :** Mise à jour progressive des composants en testant chaque changement pour détecter tout problème de compatibilité.
- **Tests rigoureux :** Les tests automatisés aident à vérifier que les fonctionnalités clés de l'application fonctionnent correctement après chaque mise à jour.

- **Adaptation du code** : Modification du code obsolète pour qu'il soit compatible avec la nouvelle version de Vue.js. Il faut tenir compte des changements de comportement et des mises à jour nécessaires dans le code.
- **Sauvegarde des données importantes** : Avant toute mise à jour majeure, nous allons nous assurer de sauvegarder les données importantes de l'application pour éviter toute perte accidentelle.
- **Documentation** : Mettre à jour la documentation pour refléter les changements spécifiques à la version LTS de Vue.js et ajouter les changements majeurs à la documentation.

Il est également crucial de tester les fonctionnalités de l'application après la transition pour nous assurer qu'elle fonctionne correctement.

4.2.2. Transition vers une version LTS majeure du backend Django Rest

Pour effectuer avec succès la transition vers une version LTS (Long Term Support) de Django, il est recommandé de suivre les étapes suivantes :

- **Évaluation de la compatibilité de l'application** : Avant toute mise à niveau, il est crucial d'évaluer la compatibilité de votre application avec la version LTS de Django en analysant attentivement les changements, les mises à jour, et les éventuels impacts sur votre code existant.
- **Sauvegarde préalable des données** : Avant de commencer le processus de mise à niveau, assurez-vous de réaliser une sauvegarde complète de toutes les données essentielles de votre application. Cela vous permettra de restaurer les données en cas de problème lors de la transition.
- **Mise à niveau progressive recommandée** : Si l'application utilise actuellement une version de Django qui n'est pas directement compatible avec la version LTS visée, il est préférable d'effectuer une mise à niveau progressive, en passant par des versions intermédiaires compatibles, pour faciliter la transition en douceur.
- **Tests rigoureux après la mise à niveau** : Une fois la mise à niveau effectuée, il est impératif de mener des tests approfondis pour identifier et résoudre tout dysfonctionnement ou incompatibilité avec la nouvelle version de Django.
- **Mise à jour complète des dépendances** : Mettre à jour toutes les dépendances tierces pour assurer leur compatibilité avec la version LTS de Django. Cela garantira le bon fonctionnement de toutes les fonctionnalités de l'application.
- **Surveillance continue pour la stabilité** : Surveiller attentivement le fonctionnement de l'application après la mise à niveau pour détecter rapidement tout problème émergent et y remédier immédiatement afin de maintenir la stabilité de l'application.

- **Mise à jour détaillée de la documentation :** Une fois la mise à niveau effectuée, mettez à jour la documentation de l'application pour refléter les changements apportés par la nouvelle version de Django. Cette documentation actualisée aidera les développeurs à mieux comprendre les modifications apportées à l'API et aux fonctionnalités.

Peu importe la complexité d'une application, la migration vers des nouvelle version majeure de LTS, implique impérativement une planification minutieuse des transitions en lisant attentivement les notes de versions et en étudiant toutes les modifications susceptibles de causer des dysfonctionnements.

5. Implémentation d'un filtre à facettes

Un filtre à facettes (ou facette) est un moyen de permettre aux utilisateurs de filtrer et de trier des données de manière interactive dans une interface utilisateur, généralement sous forme de liste ou de tableau. Ce type de fonctionnalité est couramment utilisé dans les applications Web pour faciliter la recherche et la navigation à travers de grandes quantités de données.

La mise en place d'un filtre à facette permet aux utilisateurs de parcourir et de trouver rapidement et facilement les produits de confiture en fonction de leurs préférences, de leur budget, de leurs besoins alimentaires spécifiques, de la disponibilité des produits, etc. Lorsque les utilisateurs sélectionnent un ou plusieurs filtres, la liste des produits affichée est mise à jour en temps réel pour correspondre à leurs critères de recherche.

Pour garantir la mise en place d'un filtre à facettes assurant une expérience utilisateur performante, il a été nécessaire d'enrichir la base de données initiale. Par la suite, nous examinerons à la fois la base de données initiale et la nouvelle base de données enrichie, ainsi que les étapes suivies pour générer cette dernière.

5.1. Base de données existante, création d'une nouvelle base de données et améliorations apportées

Pour construire une nouvelle base de données pour cette application, j'ai choisi d'utiliser le logiciel MySQL Workbench (un outil de conception de base de données graphique, pour concevoir le modèle de base de données).

A partir d'un modèle conçu dans MySQL Workbench nous allons pouvoir créer une base de données dans le cadre de notre backend DJANGO, en suivant les étapes suivantes :

1. **Conception du modèle de base de données dans MySQL Workbench :** Dans MySQL Workbench, nous concevons notre modèle de base de données. Cela implique la création de la base de données, des tables, la définition de colonnes, la mise en place de relations entre les tables, la spécification des clés primaires et des clés étrangères, ainsi que d'autres propriétés nécessaires pour notre base de données. De plus, nous insérons des données initiales dans les tables si nécessaire.
2. **Exportation du modèle MySQL Workbench :** Une fois que la conception est terminée, nous exportons le modèle depuis MySQL Workbench vers un script SQL. Ce script SQL contient toutes les instructions nécessaires pour créer la structure de la base de données, y compris les tables et les contraintes.
3. **Configuration de la base de données dans Django :** Dans notre projet Django, nous configurons la connexion à la base de données en modifiant les paramètres appropriés dans le fichier `settings.py`. Nous spécifions le moteur de base de données (MySQL), le

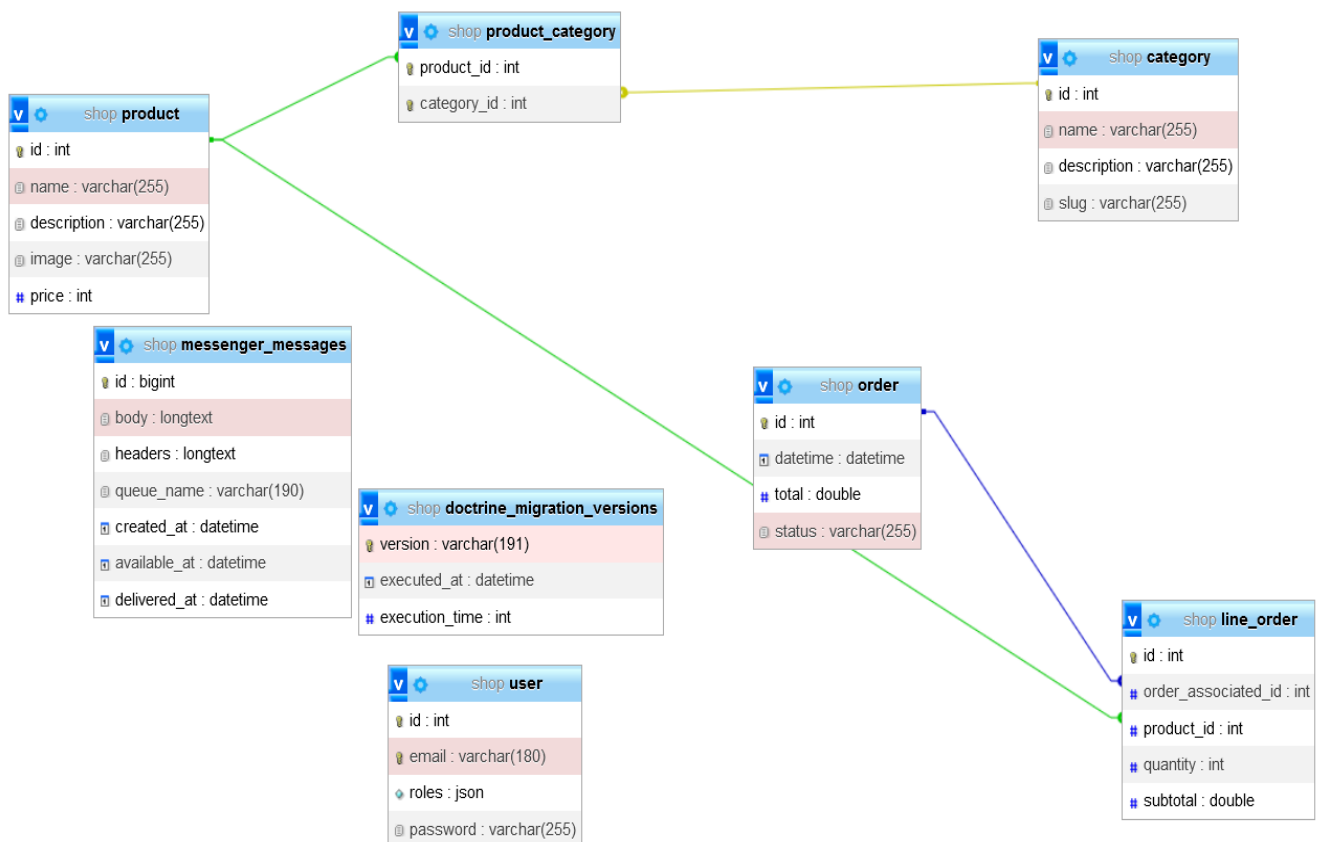
nom de la base de données, le nom d'utilisateur, le mot de passe, l'hôte et le port de la base de données MySQL. Dans MySQL Workbench, nous pouvons utiliser l'option "Forward Engineering" pour établir la connexion à notre application Django et pour générer la structure de la base de données à partir de notre modèle. Cela simplifie le processus de création de la base de données en synchronisant le modèle de données de notre application Django avec la base de données MySQL, garantissant ainsi la cohérence entre la structure de la base de données et notre modèle conceptuel. Une fois que cette configuration est correctement réalisée, Django peut créer, gérer et interagir avec la base de données de manière transparente, facilitant ainsi le développement de notre application.

4. **Création de modèles Django :** Nous créons des modèles Django qui correspondent aux tables que nous avons conçues dans MySQL Workbench. Chaque modèle Django est une classe Python qui définit les champs de la table et les relations avec d'autres tables. Nous utilisons le module `models` de Django pour définir ces modèles. Si nous avons utilisé le Forward engineering, nous pouvons générer les modèles à partir de la bdd MySQL Workbench, avec la commande `python3.10 manage.py inspectdb > back_app/models.py`
5. **Génération de migrations :** Nous utilisons la commande `python manage.py makemigrations` pour générer des fichiers de migration Django basés sur nos modèles. Ces fichiers contiennent les instructions nécessaires pour créer les tables correspondantes dans la base de données.
6. **Application des migrations :** À l'aide de la commande `python manage.py migrate`, nous appliquons les migrations pour créer les tables réelles dans la base de données MySQL. (à exécuter cette étape si on ne fait pas le forward engineering de l'étape 3)
7. **Génération de fixtures :** Une fois que la base de données est prête, nous pouvons générer des fixtures Django à partir des données introduites dans les tables de MySQL Workbench. Nous utilisons la commande `python manage.py dumpdata --format json > back/fixtures/initial-data.json`. Les fixtures sont des fichiers JSON, XML ou YAML contenant des données initiales pour notre application. Nous pouvons ensuite insérer ces données initiales dans la base de données à l'aide de la commande `python /back/manage.py loaddata initial-data.json`. À noter que si nous ajoutons de nouvelles données ou apportons des modifications au fichier de fixtures, Django est intelligent et saura préserver les données existantes dans les tables. Il effectuera uniquement les mises à jour nécessaires et ajoutera de nouvelles données lorsque cela est requis, sans supprimer les données préexistantes. Cette fonctionnalité garantit que nos données existantes restent intactes tout en permettant des mises à jour en douceur.

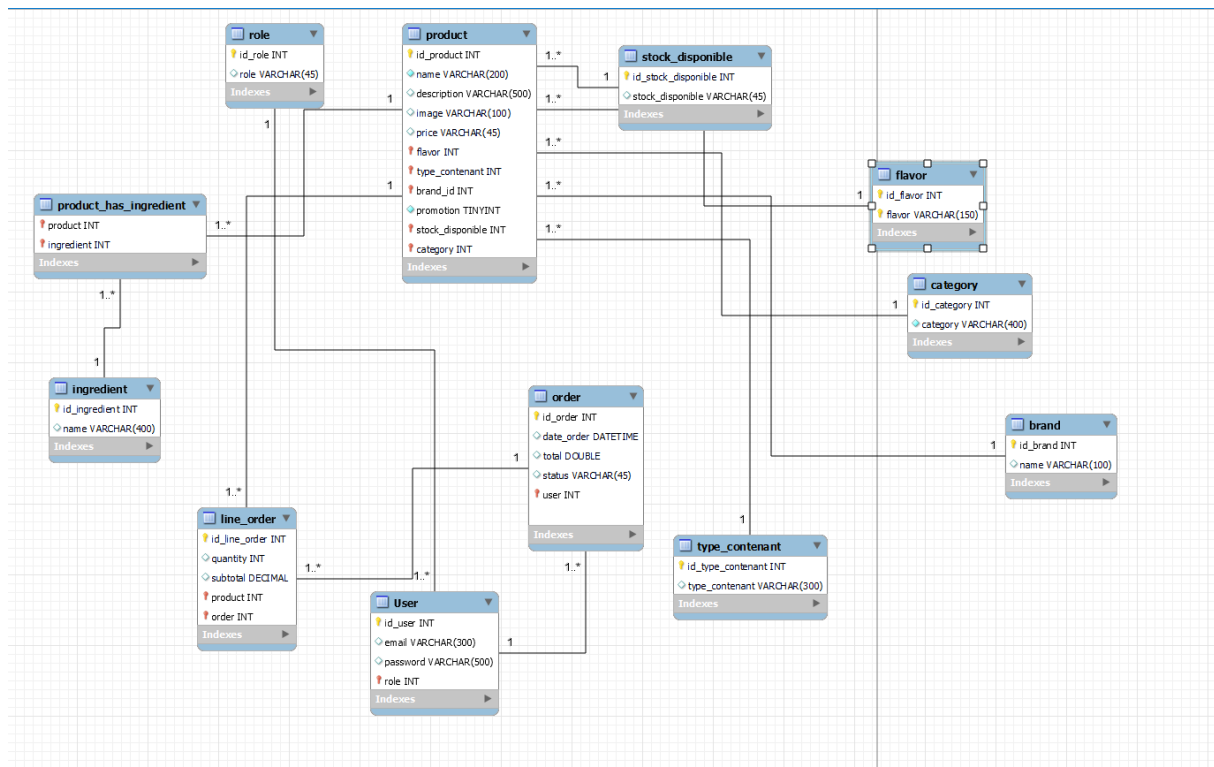
8. **Utilisation de la base de données :** Avec la structure de la base de données en place, nous pouvons utiliser Django pour interagir avec la base de données. Nous pouvons créer, lire, mettre à jour et supprimer des données en utilisant les modèles Django et les requêtes SQL générées automatiquement par Django.

Ce processus de création de base de données à partir d'un modèle MySQL Workbench dans Django permet de bénéficier de l'interface graphique conviviale de MySQL Workbench pour la conception initiale, tout en utilisant la puissance de Django pour la gestion de la base de données et le développement d'applications Web.

La base de données existante dans l'application JAM Symfony a la structure présentée dans l'image suivante :



Pour répondre à la question 5, un filtre à facettes doit être mise en place et le modèle de données existant doit être enrichie. La base de données que je vais utiliser pour cette application constituée à partir de la base de données existante, mais améliorée, aura la structure suivante :



À partir de notre nouvelle base de données, nous avons la possibilité de mettre en place une gamme complète de filtres, offrant ainsi une expérience utilisateur encore plus riche et personnalisée.

5.2. Filtres qui ont été mis en place sur l'application JAM :

Un filtre à facettes est une fonctionnalité de recherche et de filtrage avancée qui permet aux utilisateurs de raffiner leurs résultats en fonction de critères spécifiques.

Les filtres qui ont été mis en place sur l'application JAM sont les suivantes :

- **Filtre en fonction du Nom du Produit (Recherche en fonction du nom) :**

Pour que les utilisateurs puissent effectuer une recherche en fonction du nom de la confiture souhaité on va utiliser le champ nom de la table "Product". Ils pourront ainsi trier les produits en fonction du nom.

- **Filtre par Catégorie :**

On va se servir de la table "Category" pour montrer une liste des catégories disponibles.

Quand un utilisateur choisit une catégorie, on trie les produits en utilisant la clé étrangère "category" dans la table "Product".

- **Filtre par Marque :**

Pour ça, on utilise la table "Brand" pour présenter une liste de marques dispo.

Quand l'utilisateur sélectionne une marque, on trie les produits en utilisant la clé étrangère "brand" dans la table "Product".

- **Filtre par Saveur :**

On affiche une liste des saveurs dispo en se servant de la table "Flavor".

Quand un utilisateur choisit une saveur, on trie les produits avec la clé étrangère "flavor" dans la table "Product".

- **Filtre par Type de Contenant :**

On montre une liste des types de contenant avec la table "TypeContenant".

Quand l'utilisateur sélectionne un type de contenant, on trie les produits avec la clé étrangère "type_contenant" dans la table "Product".

- **Filtre par Prix :**

Pour que les utilisateurs puissent définir un prix min et max, on utilise le champ "price" dans la table "Product". Ils pourront ainsi trier les produits en fonction du prix.

- **Filtre par Prix ordre ascendante ou descendante :**

Pour que les utilisateurs puissent filtrer les produits en fonction du prix, on utilise le champ "price" dans la table "Product". Ils pourront ainsi trier les produits en fonction du prix en ordre ascendante ou descendante.

- **Filtre par Stock Disponible :**

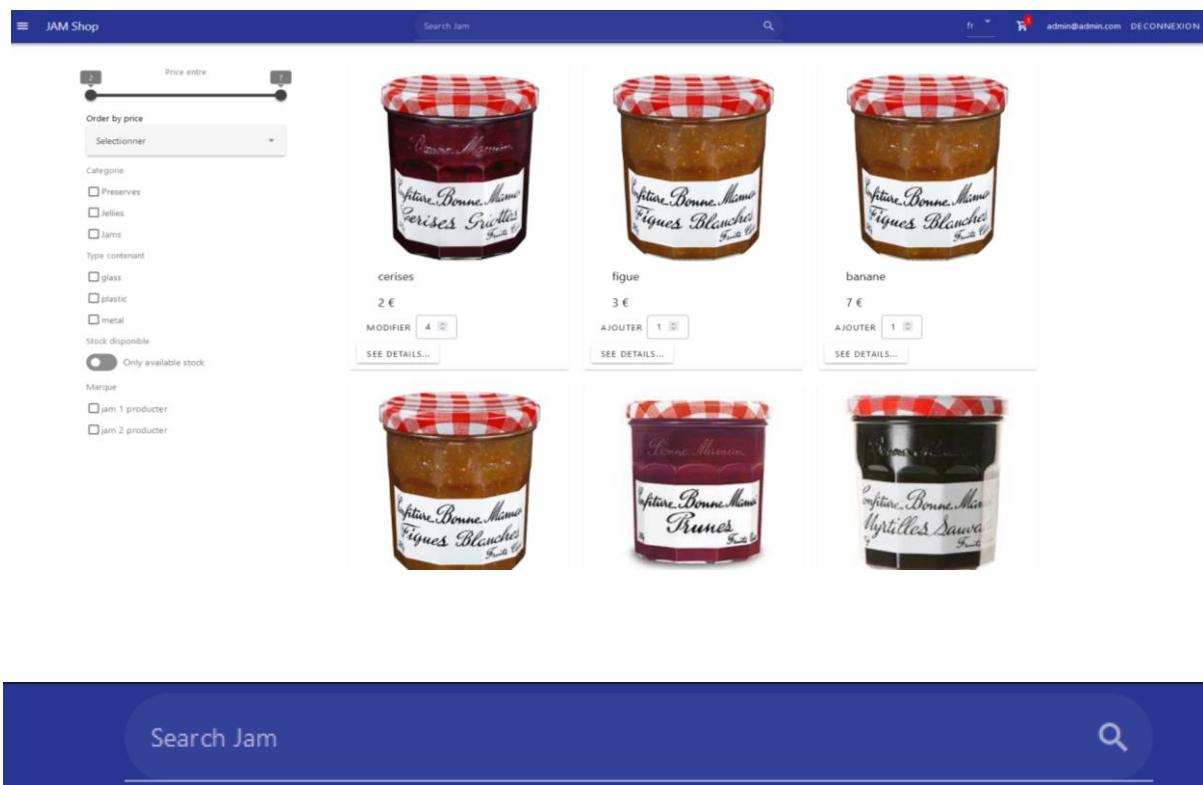
Pour ça, on utilise la table "StockDisponible" pour présenter les options de stock disponibles.

Quand l'utilisateur choisit une option, on trie les produits avec la clé étrangère "stock_disponible" dans la table "Product".

L'interaction entre tous les filtres fonctionne de manière conjointe. Par exemple, si une recherche est effectuée pour le terme "cerises", la liste affichée comprendra tous les produits


contenant ce terme. Si un autre filtre est ensuite ajouté, comme la catégorie "preserved", la liste sera restreinte aux produits qui contiennent le terme "cerises" dans leur nom et qui appartiennent à la catégorie "preserved". Ainsi, l'utilisation simultanée de plusieurs filtres permet de préciser et d'affiner les résultats en fonction de critères multiples spécifiés par l'utilisateur.

Dans notre front-end vue.js, ce filtre à facette est rendu comme dans l'image suivante :



Price entre

2 7

A horizontal slider bar with two black circular handles. The left handle is positioned at the value 2 and the right handle is at the value 7. The text "Price entre" is centered above the slider.

Order by price

Selectionner ▼

Categorie

☐ Preserves

☐ Jellies

☐ Jams

Type contenant

☐ glass

☐ plastic

☐ metal

Stock disponible

☒ Only available stock

Marque

☐ jam 1 producter

☐ jam 2 producter

6. Premier démarrage de l'application JAM

Cette application est construite avec Docker, elle nécessite donc que Docker soit installé sur la machine ou elle doit se démarrer.

Si Docker n'est pas déjà installé, il faut commencer par son installation. Pour ça, les instructions spécifiques au système d'exploitation depuis le site officiel de Docker peuvent être suivies : <https://docs.docker.com/get-docker/>. Il faut s'assurer également d'installer Docker Compose si ce n'est pas déjà fait.

Ensuite, il faut cloner le repo GitHub en utilisant la commande suivante dans le terminal :

```
git clone https://github.com/Cristina-MariaG/JAM_APP
```

Il faut avoir Git installé sur la machine. Si ce n'est pas le cas, il est possible de le télécharger à partir de <https://git-scm.com/downloads>.

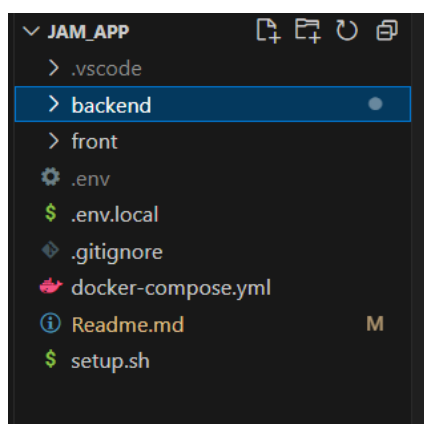
Ensuite, on peut accéder au répertoire du projet :

```
cd JAM_APP
```

Dans le dossier JAM_APP, un fichier on trouve un fichier README.md où toutes les étapes qui doivent être suivi pour pouvoir démarrer ce projet sont décrites.

Il est recommandé de lire attentivement le fichier README.md pour connaître les dépendances, les commandes et les configurations spécifiques.

Voici l'arborescence qu'on doit avoir après avoir effectué toutes les étapes présentées :



Après avoir suivis les instructions spécifiques fournies dans le fichier README.md pour lancer l'application en utilisant Docker, il faut exécuter dans le répertoire JAM_APP la commande :

```
docker-compose.up
```

de la première exécution de cette commande, toute d'abord les images docker seront créés (c'est pour ça que ça va prendre un peu plus de temps cette fois ci) et après l'application sera démarrée. Le front-end de l'application pourra être accessible à l'adresse <http://localhost:8000/> et le backend pourra être accessible à l'adresse <http://localhost:8213/>.

7. Docker

Docker est une plateforme open source conçue pour automatiser le déploiement, la gestion et le dimensionnement d'applications.

Il permet de packager une application avec toutes ses dépendances dans un conteneur standardisé, qui pourra s'exécuter sur n'importe quel environnement compatible.

Grâce à cette approche, Docker facilite le déploiement rapide et fiable d'applications, en garantissant la portabilité et la cohérence des environnements de développement et de production.

Cela permet également de simplifier le processus de développement, de test et de déploiement d'applications, tout en favorisant la collaboration entre les équipes de développement et d'exploitation.

Sur cette application on utilise docker.

Pour mettre en place docker, à la racine du projet qui peut être téléchargé depuis GitHub (https://github.com/Cristina-MariaG/JAM_APP/tree/main) nous avons un fichier docker-compose.yml, et pour le front et back, un fichier Dockerfile pour chaque partie et un fichier entrypoint_front.sh pour le front et un autre entrypoint_back.sh pour le back.

Le fichier `docker-compose.yml` est une configuration Docker Compose qui spécifie la mise en place de trois services : `jam-front`, `jam-back`, et `jam_sqldb` (base de données MySQL). Voici une explication de chaque section du fichier :

1. `x-logging` : Cette section définit les options de journalisation par défaut avec une taille maximale de fichier de 10 Mo ("max-size: "10m") et un nombre maximal de fichiers de journal de 10 ("max-file: "10"). Cela peut être utilisé pour configurer les paramètres de journalisation pour les services.
2. `services` : Cette section contient des trois services.
 - `jam-front` : Il s'agit du service frontend. Il utilise un fichier Dockerfile (défini par `context: ${HOST_DIR_FRONT}`) pour construire une image appelée `jam-front`. Il expose le port 8000 de la machine hôte et le fait correspondre au port 8000 du conteneur. Il monte un volume local dans le conteneur à l'emplacement spécifié et utilise le fichier `.env` pour les variables d'environnement.
 - `jam-back` : C'est le service backend. Il construit l'image en utilisant le répertoire spécifié par `HOST_DIR_BACK`. Il expose le port 8212 du conteneur et le fait correspondre au port 8213 de la machine hôte. Il monte également un volume local dans le conteneur à l'emplacement spécifié et utilise le fichier `.env` pour les variables d'environnement. Il dépend du service `jam_sqldb`.
 - `jam_sqldb` : Ce service est une base de données MySQL. Il utilise l'image `mysql` pour créer le conteneur, en spécifiant les variables d'environnement nécessaires pour la configuration de MySQL. De plus, il utilise le volume

- `jam_sqldb` pour stocker les données MySQL. Il expose le port 3306 du conteneur et le fait correspondre au port 3506 de la machine hôte.
3. `volumes` : Cette section définit le volume `jam_sqldb` comme un volume externe.
 4. `networks` : Cette section définit le réseau `jam_network` comme un réseau externe.

Le fichier `setup.sh` existe à la racine de ce projet contient le script qui une fois exécuté (avec `./setup.sh` dans le terminal, à la racine du projet) construira les volumes et les `networks` nécessaires pour le démarrage de cette application.

7.1. Fichiers Dockerfile front et back

Les fichiers Dockerfile jouent un rôle crucial dans la construction d'images Docker. Ils contiennent un ensemble d'instructions qui spécifient comment assembler une image Docker. Voici les rôles qu'un fichier Dockerfile peut jouer :

- **Définition de l'environnement d'exécution** : Les fichiers Dockerfile permettent de définir l'environnement d'exécution pour une application. Ils peuvent spécifier la version du système d'exploitation, les dépendances logicielles, les bibliothèques et autres composants nécessaires pour exécuter l'application dans un conteneur Docker.
- **Création d'une image Docker** : Les instructions dans un Dockerfile spécifient comment construire une image Docker à partir d'une base d'image ou d'une image existante. Les étapes peuvent inclure l'installation de logiciels supplémentaires, la copie de fichiers dans l'image, la définition de variables d'environnement, et la configuration de l'application pour l'exécution dans un conteneur.
- **Automatisation du processus de construction** : Les fichiers Dockerfile permettent d'automatiser le processus de construction d'images Docker. Ils peuvent être utilisés pour créer des images reproductibles et cohérentes à partir des spécifications fournies, ce qui facilite le déploiement et la distribution d'applications.
- **Gestion des dépendances** : En spécifiant les dépendances logicielles et en les installant dans le Dockerfile, il devient possible de gérer les dépendances de manière isolée par conteneur. Cela garantit que l'application a toutes les dépendances nécessaires pour fonctionner correctement sans affecter l'environnement hôte.
- **Configuration de l'application** : Les fichiers Dockerfile permettent de configurer divers aspects de l'application, tels que les variables d'environnement, les ports exposés, les commandes d'initialisation, etc., ce qui facilite le déploiement et l'exécution de l'application dans un environnement de conteneur Docker.

Dans le backend, le fichier `requirements.txt` est utilisé dans le processus de construction de l'image pour spécifier les dépendances Python nécessaires à l'exécution de l'application dans le conteneur Docker.

7.2. Fichiers `entrypoint_front.sh` et `entrypoint_back.sh`

Concernant les fichiers `entrypoint_front.sh` et `entrypoint_back.sh`, elle contient des scripts d'entrée pour les conteneurs Docker, qui sont utilisés pour définir le point d'entrée principal lors du démarrage du conteneur. Nous allons voir par la suite, leur rôle respectif :

- **`entrypoint_front.sh`** : est utilisé dans le contexte du service `jam-front` dans la configuration Docker Compose. Ce type de script est utilisé pour configurer l'environnement et démarrer les processus principaux nécessaires pour exécuter l'application front-end. Cela inclut des étapes telles que la configuration initiale, la gestion des dépendances, le démarrage du serveur Web ou d'autres services requis par l'application front-end.
- **`entrypoint_back.sh`** : Ce script d'entrée est utilisé dans le contexte du service `jam-back`. Il est utilisé pour configurer l'environnement et démarrer les processus principaux nécessaires pour exécuter l'application back-end. Cela peut inclure des étapes telles que la configuration de la base de données, le lancement du serveur d'application, l'initialisation de certaines tâches en arrière-plan ou d'autres opérations nécessaires pour faire fonctionner l'application back-end.

Ces scripts d'entrée sont utilisés pour effectuer les tâches de configuration et d'initialisation nécessaires avant le démarrage de l'application réelle. Ils sont utiles pour préparer l'environnement, exécuter des migrations de base de données, lancer les serveurs Web, ou démarrer d'autres services requis pour que l'application fonctionne correctement dans un conteneur Docker.

Toutes ces fichiers utilise des variables définies dans le fichier `.env`.

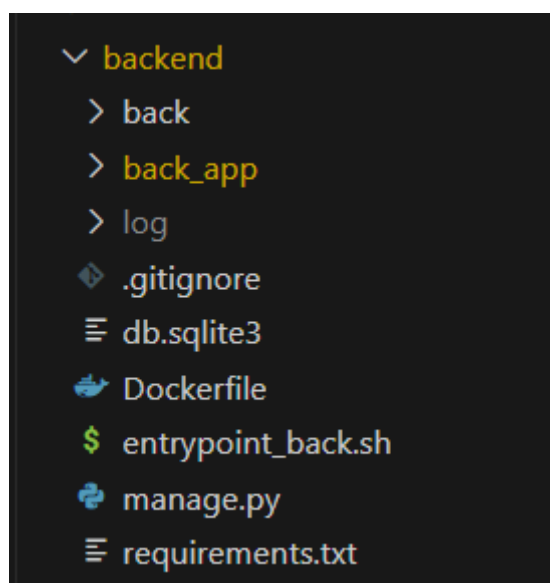
8. Backend DJANGO REST FRAMEWORK

Nous avons déjà vu la mise en place d'une base de données Mysql, ainsi que l'installation des fixtures avec Django Rest, dans le chapitre précédent.

Dans ce chapitre, nous allons voir l'arborescence des fichiers du backend, le fonctionnement et la logique qui a été utilisé dans le développement de ce backend en mode API.

8.1. Arborescence des fichiers

Voici l'arborescence du dossier backend :



- Le premier répertoire racine **backend/** est un contenant pour le projet.

Ici on retrouve :

- le dossier **back/** avec des fichiers qui correspondent au paquet Python effectif du projet
- le dossier **back_app/** qui contient des fichiers essentiels pour l'application.
- le fichier : **manage.py** : un utilitaire en ligne de commande qui permet d'interagir avec ce projet Django.
- les fichiers **Dockerfile**, **entrypoint_back.sh** et **requirements.txt** ont été déjà présentés dans le chapitre précédent ;
- le dossier **log** (le logging a été défini sur l'application) qui contient le fichier **debug.log** où on collecte des informations sur le fonctionnement de l'application.

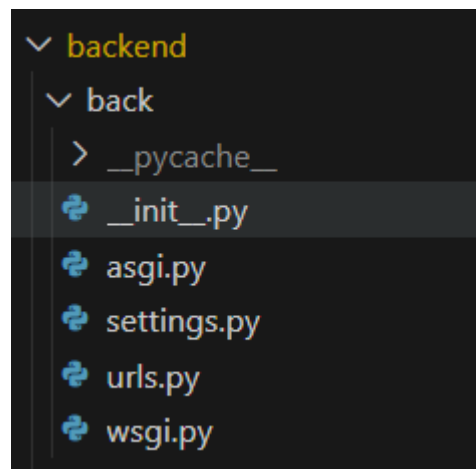
Le logging joue un rôle crucial en facilitant le débogage et la détection des erreurs, en permettant le suivi des activités pour l'analyse des performances, en assurant l'audit et la sécurité pour la conformité et la détection des menaces, ainsi qu'en contribuant à l'optimisation des performances en identifiant les goulots d'étranglement potentiels. Avec la capacité de configurer différents niveaux de journalisation, les développeurs peuvent

surveiller et contrôler efficacement le fonctionnement de l'application dans divers environnements et conditions.

Exemple des logs qu'on trouvera dans le fichier debug.log :

```
backend > log > debug.log
16088  DEBUG 2023-10-27 18:44:34,918 jam.get:16 - Start get prices
16089  DEBUG 2023-10-27 18:44:34,921 jam.get:48 - End ProductsCollection get
16090  DEBUG 2023-10-27 18:44:34,924 jam.get:20 - End get prices
16091  DEBUG 2023-10-27 18:44:34,956 jam.get:16 - Start CategoryCollection get
16092  DEBUG 2023-10-27 18:44:34,960 jam.get:24 - End CategoryCollection get
16093  DEBUG 2023-10-27 18:44:34,963 jam.get:16 - Start ContenantTypeCollection get
16094  DEBUG 2023-10-27 18:44:34,969 jam.get:25 - End ContenantTypeCollection get
16095  DEBUG 2023-10-27 18:44:34,971 jam.get:16 - Start BrandCollection get
16096  DEBUG 2023-10-27 18:44:34,976 jam.get:24 - End BrandCollection get
16097  DEBUG 2023-10-27 18:44:53,655 jam.get:30 - Start get ProductsCollection
16098  DEBUG 2023-10-27 18:44:53,656 jam.get:16 - Start get prices
16099  DEBUG 2023-10-27 18:44:53,660 jam.get:58 - End ProductsCollection get
16100  DEBUG 2023-10-27 18:44:53,663 jam.get:20 - End get prices
16101  DEBUG 2023-10-27 18:44:53,698 jam.get:16 - Start CategoryCollection get
16102  DEBUG 2023-10-27 18:44:53,702 jam.get:24 - End CategoryCollection get
16103  DEBUG 2023-10-27 18:44:53,705 jam.get:16 - Start ContenantTypeCollection get
16104  DEBUG 2023-10-27 18:44:53,709 jam.get:16 - Start BrandCollection get
16105  DEBUG 2023-10-27 18:44:53,711 jam.get:25 - End ContenantTypeCollection get
16106  DEBUG 2023-10-27 18:44:53,714 jam.get:24 - End BrandCollection get
```

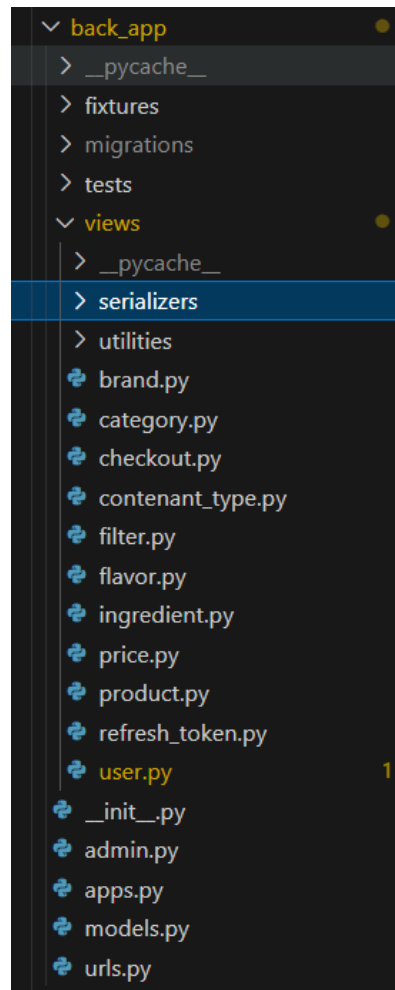
Le dossier backend/back/ :



Dans le repertoire back/ on trouve des fichiers qui correspondent au paquet Python effectif du projet. Ces fichiers sont :

- **backend/__init__.py** : un fichier vide qui indique à Python que ce répertoire doit être considéré comme un paquet.
- **backend/settings.py** : réglages et configuration du projet Django.
- **backend/urls.py** : les déclarations des URL de ce projet Django, une sorte de « table des matières » du site Django.
- **backend/asgi.py** et **application/wsgi.py** : des points d'entrée pour les serveurs Web compatibles ASGI et WSGI pour déployer le projet.

Le dossier backed/back_app/ :



Dans le deuxième répertoire back_app/ il contient des fichiers essentiels pour l'application qui représente l'application. La plupart des fichiers ont des noms caractéristiques de leur fonction dans le fonctionnement de Django :

- **fixtures/**: dossier qui contient un fichier permettant d'initialiser la base de données MySQL utilisé par l'application, mais aussi un fichier qui permet d'initialiser la base de données utilisé pour exécuter les tests.
- **migrations/** : dossier qui contient les fichiers liés à la migration des modèles déclarés
- **tests/** : dossier où on peut trouver les tests d'API mis en place pour cette application
- **views/** : dossier avec des fichiers `views_<nom-fichier>.py` où les vues sont traitées, mais aussi le dossier `utilities` qui contient un fichier `handle_errors.py` qui contient une classe de gestion d'erreur qui est utilisée dans toutes les méthodes des classes des `views` comme décorateur pour gérer les erreurs, et le

dossier serializers qui contient des fichiers avec des serializers. Le rôle des serializers est de permettre la conversion de données complexes telles que des querysets et des instances de modèle en types de données natifs de Python, qui peuvent ensuite être facilement rendus en JSON, XML ou d'autres types de contenu. Les serializers fournissent également la désérialisation, permettant la conversion de données analysées en types complexes, après avoir d'abord validé les données entrantes. Ils offrent une manière puissante et générique de contrôler la sortie de vos réponses, similaire aux classes Form et ModelForm de Django, ainsi qu'une manière pratique de gérer les instances de modèle et les querysets. Les serializers peuvent être perçus comme des filtres d'entrées/sorties. Ils vous permettent donc de **valider** (entrées) ou **formater** (sorties) vos données.

- **models.py** : fichier où on a déclaré le modèle de données pour constituer la base de données MySQL
- **admin.py** : fichier où on peut enregistrer des modèles
- **apps.py** : fichier exécuté à chaque démarrage du projet
- **urls.py** : déclarations des URL du projet

8.2. Authentification

Pour gérer une authentification basée sur les JWT le plugin `django-rest-framework-simplejwt` a été installé et utilisé dans notre backend Django.

Simple JWT est une extension pour Django REST Framework et fournit un backend avec une implémentation simple et sécurisée de JSON Web Tokens (JWT) pour l'authentification dans les applications Django. vise à couvrir les cas d'utilisation les plus courants des JWT en offrant un ensemble conservateur de fonctionnalités par défaut. Il vise également à être

Voici quelques points importants concernant Django REST Framework SimpleJWT :

1. **JWT Authentication** : Il facilite l'authentification basée sur les JWT, permettant aux utilisateurs d'obtenir des jetons JWT après s'être authentifiés avec succès, et d'utiliser ces jetons pour accéder aux ressources protégées.
2. **Configuration Simple** : Il est relativement simple à mettre en place dans un projet Django existant, offrant des fonctionnalités prêtes à l'emploi pour l'authentification basée sur JWT sans nécessiter de configurations complexes.
3. **Personnalisation des vues** : Il permet de personnaliser les vues et les comportements d'authentification pour répondre aux besoins spécifiques de l'application.
4. **Gestion des tokens** : Il facilite la gestion des tokens JWT, y compris la création, la validation et le rafraîchissement des tokens, ainsi que la révocation en cas de besoin.
5. **Sécurité avancée** : Il offre des fonctionnalités de sécurité avancées telles que la validation et le chiffrement des tokens pour garantir l'intégrité des données et la sécurité des communications.

En utilisant Django REST Framework SimpleJWT, on peut mettre en place facilement une authentification basée sur les JWT dans une applications Django, ce qui simplifie la gestion de l'authentification et de l'autorisation des utilisateurs.

Voici comment le login des utilisateurs est réalisé dans notre application :

```
import logging

from rest_framework.response import Response
from rest_framework.views import APIView
from rest_framework import status
from django.contrib.auth.hashers import make_password, check_password
from back_app.models import Role, User
from back_app.views.utilities.handle_errors import HandleError
from back_app.views.serializers.user import UserSerializer
from rest_framework_simplejwt.tokens import RefreshToken

logger = logging.getLogger("jam")

class UserCollection(APIView):
    @HandleError.handle_error("User collection post -")
    def post(self, request, *args, **kwargs):
        user_serializer = UserSerializer(data=request.data)
        user_serializer.is_valid(raise_exception=True)
        validated_data = user_serializer.validated_data

        try:
            user = User.objects.get(email=validated_data["email"])
        except User.DoesNotExist:
            return Response(
                {"error": "User not found"}, status=status.HTTP_404_NOT_FOUND
            )

        if not check_password(validated_data["password"], user.password):
            return Response(
                {"error": "Incorrect password"},
                status=status.HTTP_401_UNAUTHORIZED
            )

        refresh = RefreshToken.for_user(user)

        response_data = {
            "token": {
                "refresh_token": str(refresh),
                "access_token": str(refresh.access_token),
            },
            "user": {"email": user.email, "role": user.role.role},
        }
```



```
return Response(response_data, status=status.HTTP_200_OK)
```

Ce script Python représente une vue d'API utilisant le framework Django REST pour gérer les utilisateurs et les connexions.

1. Imports : Les importations au début du fichier incluent les dépendances nécessaires, telles que ``logging``, ``Response`` de Django REST Framework, ``APIView`` de Django REST Framework, ``status`` de Django REST Framework, ``make_password`` et ``check_password`` de Django pour le chiffrement et la vérification des mots de passe, ``Role`` et ``User`` de ``back_app.models`` pour la gestion des utilisateurs, et ``UserSerializer`` pour la sérialisation des données d'utilisateur.

2. Initialisation du journal : Une instance de logger est créée pour enregistrer les événements, avec le nom "jam".

3. Définition de la vue API : La classe ``UserCollection`` est définie comme une vue API héritant de ``APIView``.

4. Méthode POST : La méthode ``post`` est décorée avec un gestionnaire d'erreurs et reçoit les paramètres ``request``, ``*args``, et ``**kwargs``. Cette méthode gère la vérification des utilisateurs en fonction des données de la requête.

5. Sérialisation des données d'utilisateur : Les données de la requête sont sérialisées à l'aide de ``UserSerializer`` pour valider les données entrantes.

6. Récupération de l'utilisateur : La méthode tente de récupérer un utilisateur en fonction de l'adresse e-mail fournie dans la requête.

7. Vérification du mot de passe si l'utilisateur est trouvé, la méthode vérifie si le mot de passe fourni correspond au mot de passe haché stocké en base de données.

8. Génération de tokens : Si le mot de passe est correct, la méthode génère un token d'actualisation (Refresh Token) et un token d'accès pour l'utilisateur en utilisant la bibliothèque ``rest_framework_simplejwt``. A noter que la durée de vie de ces tokens est défini dans les paramètres de la librairie dans `backend/back/settings.py`.

9. Préparation de la réponse : La méthode prépare les données de réponse, notamment les tokens générés et les détails de l'utilisateur, pour être renvoyées au client.

10. Réponse de la requête : Enfin, la méthode renvoie une réponse JSON contenant les tokens d'accès et de rafraîchissement ainsi que les détails de l'utilisateur connecté.

Ce code est utilisé pour la vérification des données de login des utilisateurs, ainsi que la génération de tokens d'authentification pour les connexions d'utilisateurs dans une application utilisant Django REST Framework.

8.3.Stripe

Stripe est une plateforme de paiement en ligne qui permet aux entreprises d'accepter des paiements sur Internet. Elle propose une gamme complète de produits et de services pour faciliter les transactions financières en ligne.

Pour ajouter le module de paiement sur notre application JAM, la librairie stripe a été installée.

Cette bibliothèque Python permet d'interagir avec l'API de Stripe. Elle facilite l'intégration des fonctionnalités de paiement de Stripe dans les applications Python en fournissant des méthodes et des classes pour interagir avec l'API de Stripe.

Voici dans notre application l'utilisation de stripe :

```
from back_app.models import Order
from back_app.views.serializers.order import LineOrderSerializer,
OrderSerializer
from back_app.views.utilities.handle_errors import HandleError
from datetime import datetime
from datetime import datetime
from django.conf import settings
from rest_framework import status
from rest_framework.response import Response
from rest_framework.views import APIView
import jwt
import logging
import stripe

logger = logging.getLogger("jam")

domain_url = settings.DOMAIN_URL

class CheckoutCollection(APIView):
    @HandleError.handle_error("Jam CheckoutCollection post -")
    def post(self, request, *args, **kwargs):
        logger.debug("Start CheckoutCollection post ")

        data = request.data
        line_items = []

        user_id = verify_validity_decode_token(data["accessToken"])

        if not user_id:
            return Response(status=status.HTTP_401_UNAUTHORIZED)

        order_id = create_order(request.data, user_id)

        for el in list(data["cart"]):
```

```

        line_items.append(
            {
                "price_data": {
                    # Convert the price from a string to an integer value
                    # in cents for Stripe
                    # (which displays prices in cents) and then cast it to
                    # an integer.
                    # This is done to avoid potential errors with Stripe
                    # due to non-integer values.
                    "unit_amount": int(float(el["price"]) * 100),
                    "currency": "eur",
                    "product_data": {"name": el["name"]},
                },
                "quantity": el["selectedQuantity"],
            }
        )
        create_line_order_for_order_id(order_id, el)

    stripe.api_key = settings.STRIPE_SECRET_KEY
    checkout_session = stripe.checkout.Session.create(
        success_url=domain_url +
        "success?session_id={CHECKOUT_SESSION_ID}",
        cancel_url=domain_url + "cancel-payment",
        payment_method_types=["card"],
        mode="payment",
        line_items=line_items,
    )
    logger.debug("End CheckoutCollection post ")
    return Response({"url": checkout_session["url"], "order_id":
order_id})

```

Ce script Python représente une vue d'API en utilisant Django et Stripe, pour gérer des transactions de paiement en ligne. Il inclut une méthode POST accessible à l'adresse : <http://localhost:8213/create-checkout-session/>, qui vérifie l'authentification, crée une commande (dans la bdd mysql), construit des éléments de ligne pour les articles du panier, configure Stripe pour le traitement du paiement, et renvoie une réponse avec l'URL de la session de paiement et l'ID de la commande qui a un status de : 'Payment_Waiting '.

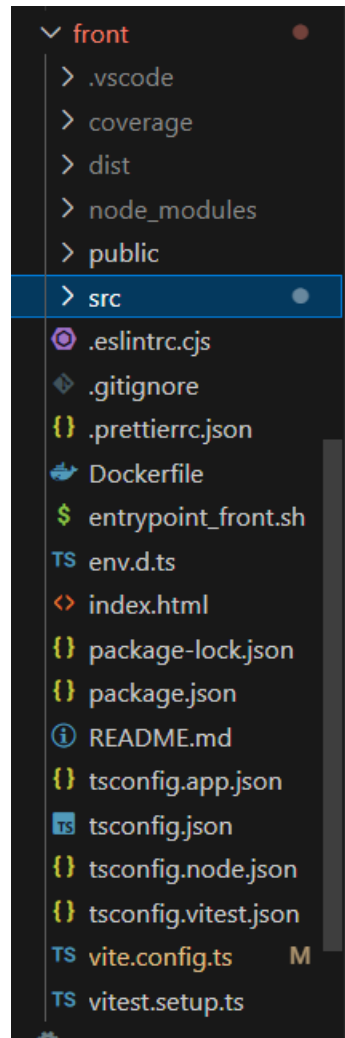
9. Frontend Vue.JS-TypeScript

Le front-end de cette application a été construit avec Vue.js 3.

Vue.js 3 est la dernière version majeure du framework JavaScript Vue.js.

9.1.Arborescence des fichiers

Nous allons voir par la suite l'arborescence de ce projet :



Voici une description générale des fichiers et dossiers dans le projet :

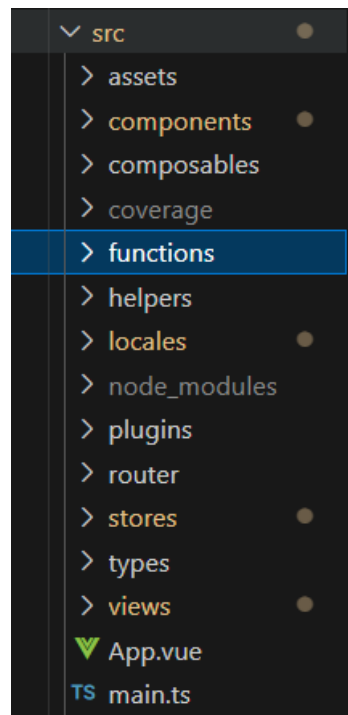
1. Dossiers :

- **src** : Le dossier principal de votre application où on trouve la majorité du code source Vue.js. Il contient des fichiers de composants, des fichiers de routage, des fichiers de magasin (store) et d'autres éléments de l'application.

- **public** : Ce dossier contient les ressources statiques de l'application, comme les fichiers HTML, les images ou les polices.
- **node_modules** : Le dossier qui contient toutes les dépendances du projet, installées via npm
- **dist** : Le dossier où les fichiers optimisés de production sont générés après la construction de l'application.
- **Fichiers** :
- **package.json** : Ce fichier contient les métadonnées du projet, y compris les dépendances, les scripts de démarrage, les versions, et d'autres informations pertinentes pour la configuration de l'application.
- **package-lock.json** : Ce fichier est généré automatiquement pour verrouiller les versions exactes des packages installés, assurant ainsi la reproductibilité des installations de dépendances.
- **tsconfig.json** : Ce fichier contient la configuration TypeScript pour le projet, y compris les options de compilation spécifiques.
- **vite.config.ts** : Ce fichier contient la configuration de l'outil de développement Vite, y compris les plugins, les options de construction et de développement, et d'autres réglages spécifiques à Vite.
- **babel.config.js** : Ce fichier est présent pour configurer Babel si nécessaire, bien que Vite prenne en charge nativement TypeScript.
- **.gitignore** : Ce fichier spécifie les fichiers et dossiers qui ne doivent pas être suivis par Git lors de la gestion des versions de projet.
- **README.md** : Ce fichier contient généralement la documentation de base du projet, décrivant son fonctionnement, ses dépendances, et fournissant des instructions d'installation et d'utilisation.

En ce qui concerne les fichiers Dockerfile et entrypoint_front.sh ce sont des fichiers de configuration Docker et nous avons vu leur rôle dans un chapitre précédent.

Dossier front/src/ :



Voici ce que nous avons à l'intérieur du dossier `src` :

- **assets/** : Un dossier où on stock les ressources statiques telles que des images, des polices, des fichiers CSS, etc.
- **components/**: Ce répertoire contient les fichiers de composants réutilisables qui constituent les éléments de base de l'interface utilisateur.
- **composables/** : Les "composables" sont une fonctionnalité introduite dans Vue 3, visant à faciliter la réutilisation de la logique métier dans les composants Vue. Ce dossier contient des fichiers contenant des fonctions qui encapsulent une logique métier spécifique.
- **coverage/**: le dossier "coverage" est utilisé pour stocker les rapports de couverture de code générés par des outils de test lors de l'exécution de tests unitaires. Ce dossier est généré automatiquement lors de l'exécution de tests, et il contient des informations détaillées sur la couverture du code, indiquant quelles parties du code ont été exécutées ou testées.
- **functions/** : Dans ce dossier des fonctions utilitaires globales qui ne sont pas liés à un composant spécifique mais qui sont utilisés à plusieurs endroits dans l'application.
- **helpers/** : Ce répertoire contient des fichiers de services qui gèrent les appels d'API pour récupérer ou manipuler des données. Ces helpers sont utilisés par les composants pour effectuer des opérations telles que la récupération et la manipulation de données distantes.

- **locales/** : Le dossier "locales" est généralement utilisé pour stocker des fichiers de localisation ou de traduction dans les applications multilingues. Il est souvent utilisé pour contenir des fichiers de texte contenant des traductions pour différents langages pris en charge par l'application
- **plugins/** : Le dossier "plugins" est utilisé pour stocker des plugins spécifiques utilisés dans l'application. Exemple : le plugins vuetify.
- **router/** : Ce répertoire contient le fichier index.ts, fichier de configuration de routage qui définit les différentes routes de l'application, permettant ainsi la navigation entre les différentes vues et composants.
- **store/** : Ce répertoire peut contenir des fichiers de magasin qui gèrent l'état global de l'application à l'aide de la bibliothèque de gestion de l'état Pinia. Ces fichiers contiennent des mutations, des actions et des getters pour gérer l'état de l'application de manière cohérente.
- **types/** : Le dossier "types" est utilisé pour stocker des fichiers de définition de type.
- **views/** : Ce répertoire contient les fichiers de vues qui représentent les pages de l'application. Les vues peuvent être composée de plusieurs composants et peut être liée à des routes spécifiques dans l'application.
- **App.vue** : Le composant racine de l'application Vue.js, contenant la structure de base de l'application.
- **main.ts** : Le point d'entrée de l'application où l'instance Vue est initialisé et d'autres configurations sont effectués.

9.2.Fonctionnalités de l'application

Toutes les fonctionnalités qui existaient sur le projet d'origine ont été implémenté dans le nouveau projet, et encore quelques-unes de plus :

- **Pagination**

La pagination est une fonctionnalité essentielle présente dans de nombreuses applications web, permettant de découper de vastes ensembles de données en plusieurs pages. Elle améliore la navigation et renforce les performances en évitant de charger toutes les données en une seule fois. À chaque changement de page, une requête supplémentaire est envoyée au serveur pour récupérer les données spécifiques à la page en question. Ce processus garantit une expérience utilisateur fluide et optimisée, en permettant une manipulation efficace des données volumineuses sans sacrifier la vitesse ou la fluidité de l'application.

- **Internationalisation**

L'internationalisation, abrégée en "i18n" dans le contexte du développement web, consiste à rendre une application accessible à un public international en prenant en charge plusieurs langues et cultures. Dans notre application l'internationalisation est implémenté à l'aide de bibliothèques "vue-i18n".

Un autre avantage majeur de l'internationalisation réside dans la centralisation de toutes les chaînes de texte dans un seul fichier. Cela facilite grandement la gestion et la maintenance des traductions, permettant de travailler efficacement sur les différentes versions linguistiques de l'application. En regroupant toutes les chaînes de texte dans un fichier dédié, nous pouvons facilement mettre à jour, ajouter ou supprimer des traductions sans avoir à parcourir tout le code de l'application. Cette approche simplifie également le processus de gestion de la localisation, ce qui se traduit par une expérience utilisateur plus cohérente et plus fluide, indépendamment de la langue sélectionnée.

- Possibilité d'ajouter un nouveau produit ou de supprimer un produit si l'utilisateur connecté a le rôle d'admin.

Sur la page "/dashboard", accessible depuis la page d'accueil pour les administrateurs, les utilisateurs disposant du rôle d'administrateur peuvent facilement ajouter de nouveaux produits à la base de données. De plus, pour les utilisateurs ayant le rôle d'administrateur, un bouton de suppression des produits est disponible sur la page d'accueil, à côté du bouton "Ajouter au panier", sur chaque carte représentant un produit. Cette fonctionnalité offre aux administrateurs un contrôle complet sur la gestion des produits, leur permettant d'ajouter de nouveaux articles et de supprimer ceux existants en toute simplicité, depuis une seule et même interface conviviale.

- Possibilité de voir les détails d'un produit

Chaque carte représentant un produit sur la page d'accueil est accompagnée d'un bouton "Voir les détails", redirigeant les utilisateurs vers une page dédiée affichant de manière exhaustive toutes les informations spécifiques au produit sélectionné. Cette fonctionnalité permet aux utilisateurs d'explorer plus en détail les caractéristiques, les spécifications et d'autres informations pertinentes relatives au produit, offrant ainsi une expérience plus approfondie et informative.

- Possibilité d'effacer un produit depuis le panier

Chaque élément de la liste de courses sur la page "/cart" est associé à un bouton "Supprimer" permettant aux utilisateurs de retirer facilement un produit de leur panier. Cette fonctionnalité simplifie la gestion des articles dans le panier, offrant aux utilisateurs un contrôle direct sur les articles qu'ils souhaitent conserver ou supprimer.

- Concernant le filtre à facettes, nous avons déjà vu dans un chapitre précédent qu'il a été enrichie et qu'on peut, donc, appliquer des filtres en plus sur nos données.

9.3. Vues de l'application

Nous aborderons par la suite les différentes vues de l'application.

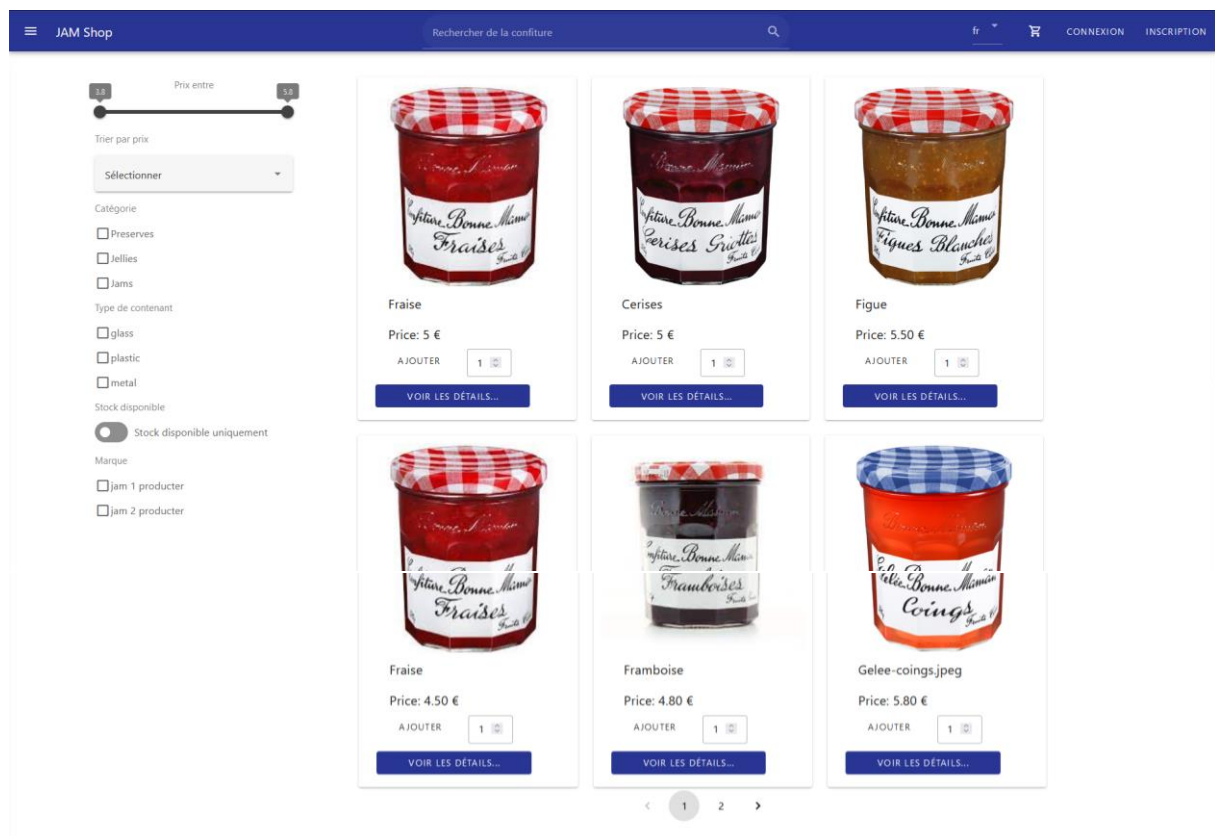
Parmi celles-ci, seules deux vues, à savoir `"/dashboard"` et la page de gestion des paiements générée par Stripe, ne sont accessibles qu'après une connexion préalable.

- La page située à l'adresse <http://localhost:8000> correspond à la vue `"HomeView.vue"`, qui constitue la page d'accueil principale de l'application JAM Shop.

À partir de cette page, nous pouvons accéder à l'ensemble des autres pages disponibles dans le projet. Toutes les pages de l'application intègrent l'en-tête tel qu'illustré dans l'image suivante, offrant ainsi une expérience de navigation cohérente et conviviale à travers les différentes sections de l'application.

Depuis l'en-tête, les utilisateurs ont la possibilité de modifier la langue, d'accéder aux pages de connexion ou d'inscription (`"/login"` ou `"/inscription"`), de consulter le panier (`"/cart"`), de revenir à la page d'accueil (en cliquant sur `"JAM Shop"` ou en sélectionnant les lignes du menu, ce qui ouvre un menu sur la gauche de la page).

De plus, les utilisateurs peuvent effectuer une recherche en fonction du nom, et depuis le menu déroulant accessible en cliquant sur les trois lignes horizontales, les administrateurs peuvent accéder à la page de tableau de bord (`"/dashboard"`) si une session est active. Cette disposition permet d'explorer facilement les différentes fonctionnalités en fonction des besoins spécifiques.

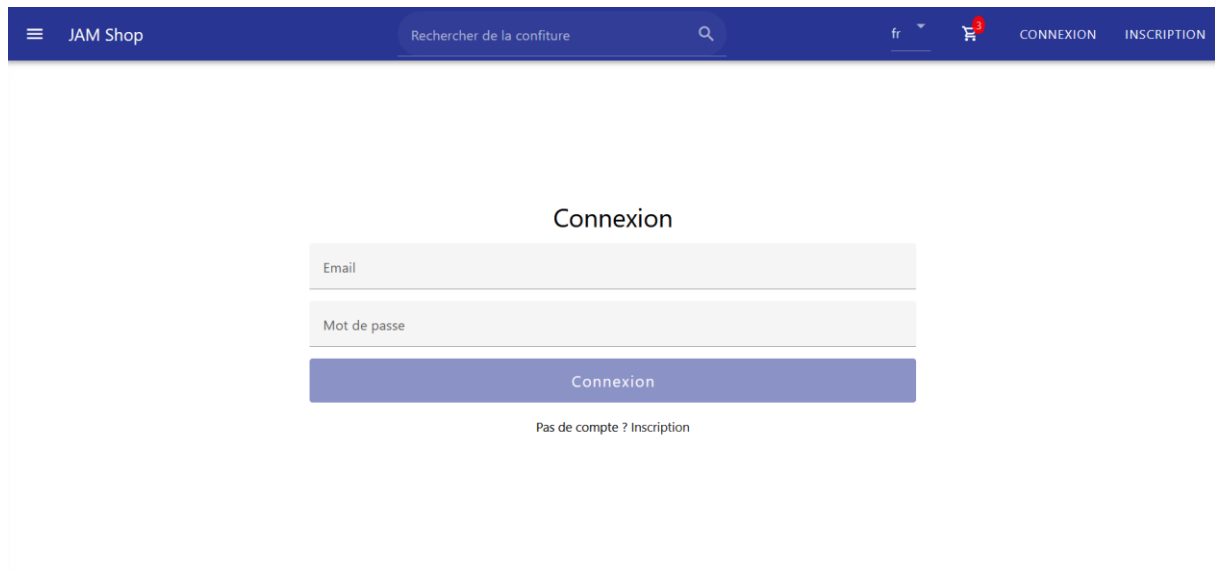


- La page située à l'adresse /product-details/<int :id> .

La page située à l'adresse "/product-details/int:id" est conçue pour afficher tous les détails du produit sélectionné. Lorsqu'on accède à cette page, l'identifiant du produit souhaité est transmis en arrière-plan, ce qui permet de récupérer toutes les informations spécifiques relatives à ce produit dans la base de données. Grâce à cette fonctionnalité, nous pouvons obtenir une vue détaillée et complète des caractéristiques, des images et d'autres informations pertinentes concernant le produit sélectionné, améliorant ainsi l'expérience d'achat globale sur la plateforme.

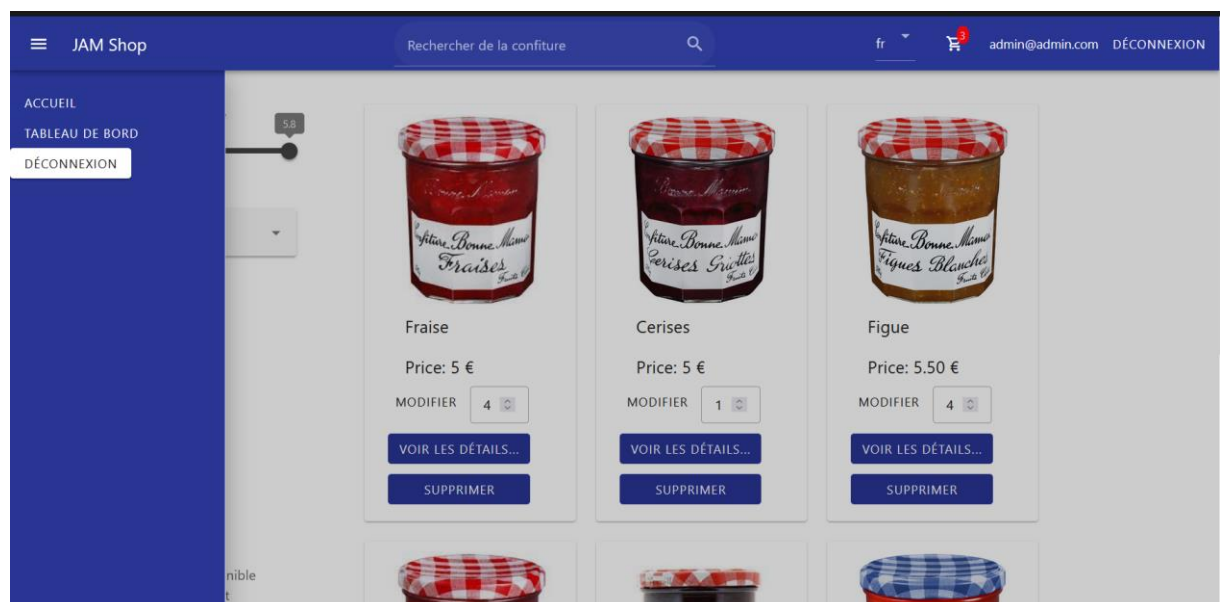


- La page Login



- La page Inscription

- La page d'accueil après une connexion avec d'un utilisateur qui a le rôle admin :



- La page /dashboard (si admin)

Cette page n'est pas une fierté au niveau design, mais elle est fonctionnelle et permet à un administrateur d'ajouter des nouveaux produits.

JAM Shop

Rechercher de la confiture

fr


 admin@admin.com DÉCONNEXION

Tableau de bord

Select

Marque

Select

☒ Stock disponible

Type de contenant

Select

Saveur

Select

Ingrédients

Quantité

sugar

lemon juice

strawberries

peaches

abricot

water

AJOUTER UN PRODUIT

AJOUTER UN PRODUIT

AJOUTER UN PRODUIT

AJOUTER UN PRODUIT

AJOUTER UN PRODUIT

AJOUTER UN PRODUIT

Name

Description

Image URL

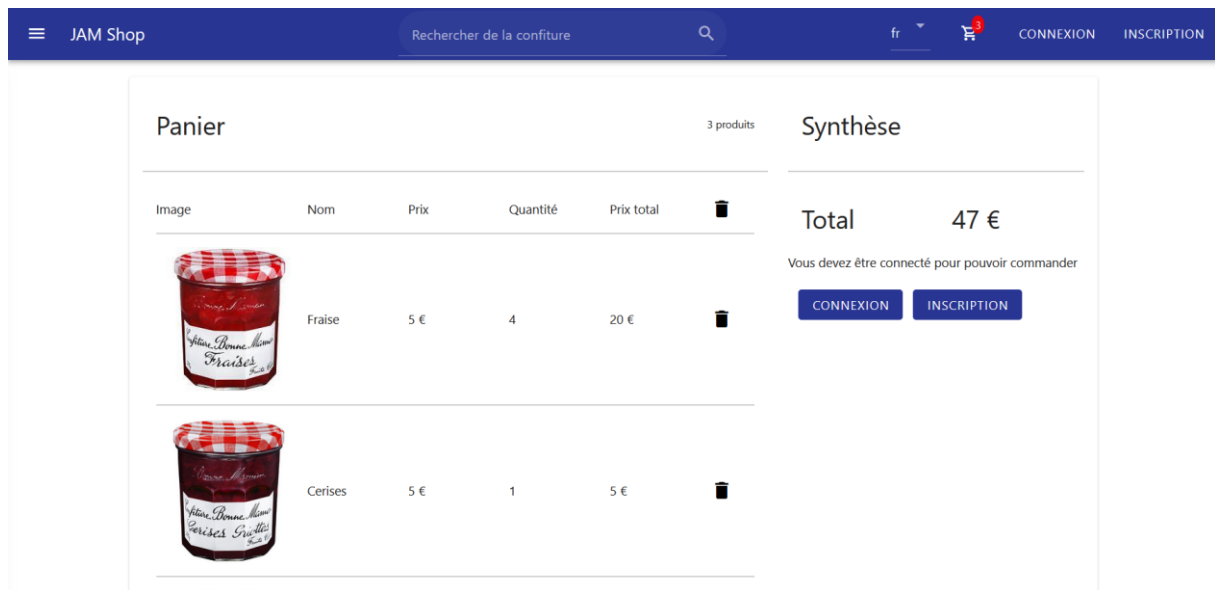
Price

Quantité

Promotion

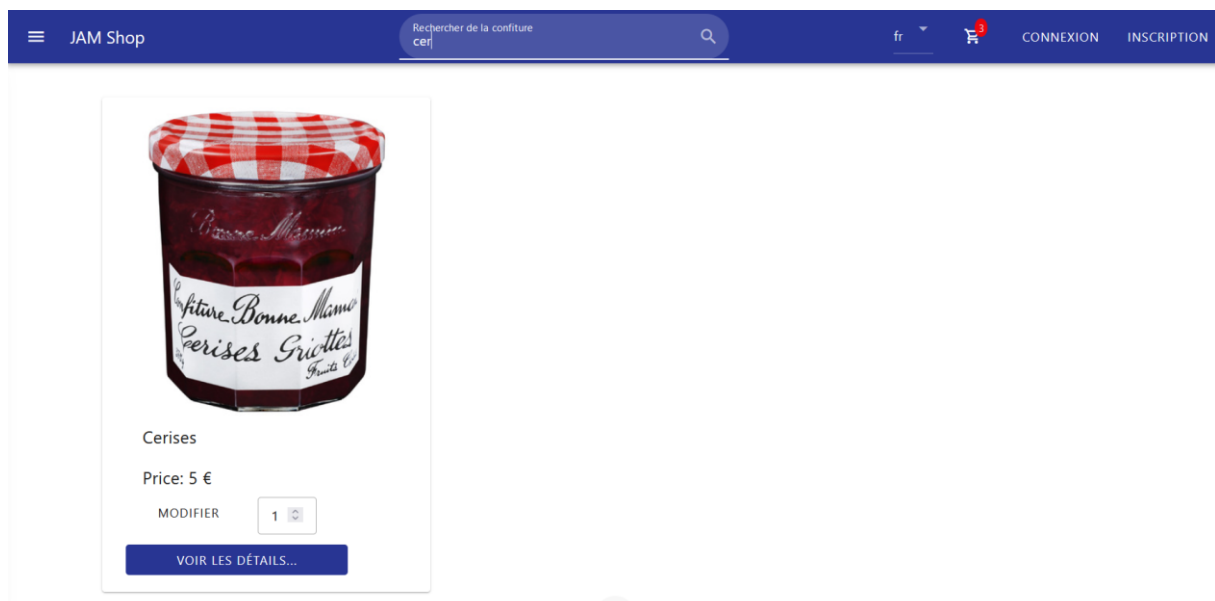
SAUVEGARDER

- La page panier :



- La page /search/string

La page de recherche est accessible à la fois depuis la page "/cart" et depuis la page "/product-details/id". Cette page a été créée pour garder le fonctionnement de l'application initiale. Auparavant, lorsqu'on effectuait une recherche de produit par nom, nous étions redirigé vers une page de recherche distincte. Les filtres ne pouvaient être appliqués qu'à l'ensemble des produits, ce qui limitait la possibilité d'appliquer des filtres après avoir effectué une recherche par nom. Pour offrir une fonctionnalité plus souple, j'ai mis à jour l'application pour permettre la recherche par nom, suivie de la possibilité d'appliquer des filtres aux résultats de la recherche (sur la page d'accueil). De plus, nous pouvons également appliquer des filtres à l'ensemble des produits sans avoir préalablement effectué de recherche. Avec cette approche j'ai pu conserver le fonctionnement initial de l'application tout en ajoutant de nouvelles fonctionnalités pour améliorer l'expérience de recherche et de filtrage des produits.



10. Sécurité

L'importance de la sécurité dans le contexte d'une application web ne peut être surestimée. Dans un paysage numérique en constante évolution, la protection des données et la confidentialité des utilisateurs sont des préoccupations critiques pour tout projet de développement d'application. Pour garantir la résilience des applications et atténuer les risques liés aux cybermenaces, il est impératif de suivre des principes et des pratiques de sécurité clés qui sous-tendent le développement d'applications robustes. Les points suivants soulignent l'importance ainsi que la mise en place de la sécurité dans le contexte des frameworks et de l'application JAM.

10.1. La sécurité et les frameworks

Lorsqu'il s'agit de sécurité dans le contexte des frameworks, il existe plusieurs aspects à considérer :

Prévention des failles de sécurité : Les frameworks modernes intègrent souvent des mécanismes de sécurité de base pour aider à prévenir les failles courantes telles que les attaques par injection SQL, les attaques XSS (Cross-Site Scripting) et d'autres types d'attaques de sécurité connus. Ces mécanismes incluent la validation des entrées, l'échappement des sorties, et la protection contre les requêtes falsifiées entre sites (CSRF).

Authentification et autorisation : De nombreux frameworks offrent des fonctionnalités d'authentification et d'autorisation intégrées pour gérer l'identification des utilisateurs, la vérification des droits d'accès, et le contrôle des privilèges de l'utilisateur.

Gestion des sessions : Les frameworks fournissent généralement des outils pour gérer les sessions des utilisateurs, ce qui inclut la sécurisation des identifiants de session, la limitation de la durée de validité des sessions, et la gestion sécurisée des cookies.

Chiffrement et sécurité des données : Certains frameworks intègrent des fonctionnalités pour le chiffrement des données sensibles, telles que les mots de passe d'utilisateurs, les informations de paiement et d'autres données confidentielles.

Protection contre les attaques de sécurité avancées : Les frameworks les plus récents peuvent également offrir des protections contre les menaces de sécurité avancées telles que les attaques par déni de service distribué (DDoS), les attaques de scripting côté client (XSS), et les tentatives d'intrusion malveillantes.

Il est important de noter que, bien que les frameworks puissent fournir des fonctionnalités de sécurité de base, la responsabilité ultime de la sécurité de l'application repose toujours sur les développeurs.

Il est essentiel de suivre les meilleures pratiques de sécurité et de tenir compte des vulnérabilités potentielles spécifiques à chaque projet lors du développement.

10.2. La sécurité de l'application JAM

Backend :

Django REST Framework (DRF) offre plusieurs fonctionnalités de sécurité intégrées pour garantir la protection des API contre les attaques potentielles. Voici quelques mesures de sécurité intégrées à Django REST Framework :

- **Authentification et autorisation** : DRF fournit plusieurs options d'authentification telles que la gestion des jetons (token-based authentication), l'authentification de base (basic authentication), l'authentification OAuth, l'authentification à l'aide de sessions, etc. Ces méthodes permettent de vérifier l'identité des utilisateurs et de garantir que seuls les utilisateurs autorisés ont accès aux ressources appropriées.
- **Protection contre les failles de sécurité courantes** : DRF intègre des protections contre les attaques courantes telles que les attaques par **injection SQL**, les attaques par injection de scripts entre sites (XSS), et les **attaques CSRF** (Cross-Site Request Forgery). Ces protections sont mises en place par défaut pour empêcher les attaques potentielles visant à compromettre la sécurité de l'application.
- **Contrôle d'accès** : DRF propose des mécanismes de contrôle d'accès granulaires qui permettent de restreindre l'accès à certaines ressources en fonction des permissions accordées à chaque utilisateur. Cela permet de définir des règles spécifiques pour contrôler qui peut voir, modifier ou supprimer certaines données dans l'API.
- **Validation des données** : DRF offre des outils de validation de données intégrés qui aident à garantir que les données entrantes sont conformes aux attentes de l'application, ce qui réduit les risques de vulnérabilités potentielles causées par des données incorrectes ou malveillantes.
- **Gestion des erreurs sécurisée** : DRF fournit des fonctionnalités pour gérer les erreurs de manière sécurisée, en veillant à ce que des informations sensibles ne soient pas divulguées en cas de problème ou d'erreur de traitement.
- **Protection contre l'injection SQL**

L'injection SQL est un type d'attaque où un utilisateur malveillant est capable d'exécuter du code SQL arbitraire sur une base de données. Il peut en résulter des suppressions d'enregistrements ou des divulgations de données.

Les jeux de requête de Django sont prémunis contre les injections SQL car leurs requêtes sont construites à l'aide de la paramétrisation des requêtes. Le code SQL d'une requête est défini séparément de ses paramètres. Comme ceux-ci peuvent provenir de l'utilisateur et donc non sécurisés, leur échappement est assuré par le pilote de base de données sous-jacent.

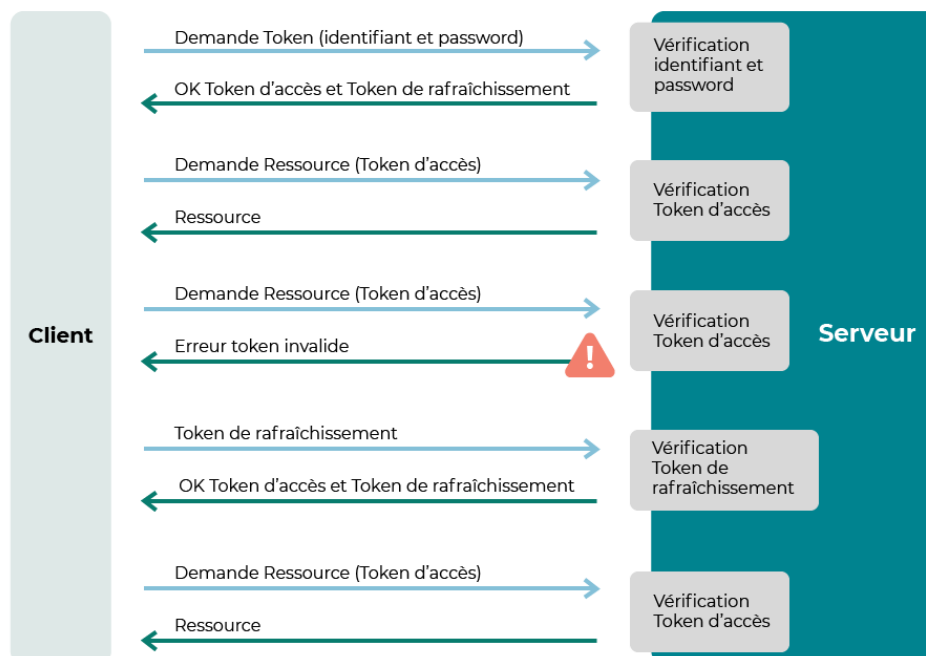
Django permet aussi d'écrire des [requêtes brutes](#) ou d'exécuter du [code SQL personnalisé](#). Ces possibilités devraient être exploitées de manière parcimonieuse et il faut toujours prendre la précaution d'échapper proprement tout paramètre pouvant être contrôlé par l'utilisateur. De plus, il faut être prudent en utilisant [extra\(\)](#) et [RawSQL](#).

Dans notre application on utilise le ORM de Django pour les requêtes, donc les requêtes seront échappées par le pilote de base de données.

En plus, sur l'application JAM la sécurisation des mots de passe, la gestion des autorisations de manière appropriée, la validation minutieuse des données et la configuration correcte des paramètres de sécurité ont été adoptés.

Nous avons vu dans le sous-chapitre 8.2., que le plugin `django-rest-framework-simplejwt` a été installé et utilisé pour mettre en place une authentification basée sur les JWT.

Grace à ce plugin, nous pouvons générer lors de l'authentification d'un utilisateur un token d'accès et un token de rafraîchissement qui sont utilisés pour sécuriser les routes : `/create-checkout-session/` et aussi pour le dashboard des utilisateurs avec le rôle d'admin, qui veulent enregistrer des nouveaux produits. L'utilisation de ces tokens pour les deux routes spécifiées précédemment, se fait comme dans l'image suivante :



Nous avons déjà vu dans un chapitre précédent comment la génération d'un token d'accès se fait.

Nous allons voir maintenant la route qui contient la vérification du token de rafraîchissement, qui est appelé si le token d'accès est expiré, ainsi que la génération d'un nouveau token d'accès et un nouveau token de rafraîchissement si le dernier est valide lors de la vérification.

```

import logging
from rest_framework.views import APIView
from rest_framework.response import Response
from back_app.views.utilities.handle_errors import (
    HandleError,
)

from rest_framework_simplejwt.tokens import RefreshToken
    
```



```

from rest_framework import status

logger = logging.getLogger("jam")

class RefreshTokenData(APIView):
    @HandleError.handle_error("Jam RefreshToken delete -")
    def post(self, request, *args, **kwargs):
        logger.debug("Start RefreshToken delete ")
        response = dict()

        refresh_token = request.data["refresh_token"]
        if refresh_token:
            refresh = RefreshToken(refresh_token)

            response["token"] = {
                "refresh_token": str(refresh),
                "access_token": str(refresh.access_token),
            }
            status_code = status.HTTP_200_OK
        else:
            status_code = status.HTTP_401_UNAUTHORIZED

        logger.debug("End RefreshToken delete")
        return Response(response, status=status_code)

```

Ce script importe des modules de journalisation, de gestion des erreurs et de gestion des réponses. Il utilise également des fonctionnalités de journalisation pour enregistrer des messages à des niveaux de gravité variés. En outre, il définit une classe pour gérer les requêtes POST (pour le url refresh_token/) et renvoyer des réponses HTTP en conséquence. Dans le cadre de cette methode , la récupération du "refresh_token" à partir des données de la requête est faite ainsi que la génération d'une réponse en fonction de la présence et la vérification du "refresh_token".

Voici le code qui correspond à la route /refresh-token :

```

import logging
from rest_framework.views import APIView
from rest_framework.response import Response
from back_app.views.utilities.handle_errors import (
    HandleError,
)
from rest_framework_simplejwt.tokens import RefreshToken
from rest_framework import status

logger = logging.getLogger("jam")

class RefreshTokenData(APIView):
    @HandleError.handle_error("Jam RefreshToken delete -")

```

```

def post(self, request, *args, **kwargs):
    logger.debug("Start RefreshToken delete ")
    response = dict()

    refresh_token = request.data["refresh_token"]
    if refresh_token:
        refresh = RefreshToken(refresh_token)

        response["token"] = {
            "refresh_token": str(refresh),
            "access_token": str(refresh.access_token),
        }
        status_code = status.HTTP_200_OK
    else:
        status_code = status.HTTP_401_UNAUTHORIZED

    logger.debug("End RefreshToken delete")
    return Response(response, status=status_code)

```

Voici une explication détaillée du code correspondant à la route /refresh-token/ :

1. ``import logging``: Cela importe le module de journalisation intégré de Python, qui permet de créer des journaux d'événements pour l'application.
2. ``from rest_framework.views import APIView``: Cela importe la classe `APIView` de Django REST framework, qui est utilisée pour créer des vues basées sur les API.
3. ``from rest_framework.response import Response``: Cela importe la classe `Response` de Django REST framework, qui est utilisée pour renvoyer des réponses HTTP.
4. ``from back_app.views.utilities.handle_errors import HandleError``: Cela importe la classe `HandleError` du module `handle_errors` dans le package `utilities` du package `views` de l'application `back_app`. Il est utilisé pour gérer les erreurs dans l'application.
5. ``from rest_framework_simplejwt.tokens import RefreshToken``: Cela importe la classe `RefreshToken` du module `tokens` de la bibliothèque Django REST framework `simplejwt`, qui est utilisée pour gérer les jetons de rafraîchissement.
6. ``from rest_framework import status``: Cela importe le module de statut de Django REST framework, qui contient des constantes pour les codes de statut HTTP.
7. ``logger = logging.getLogger("jam")``: Cela initialise un enregistreur avec le nom "jam" pour enregistrer les événements spécifiques à cette partie du code.

8. ``class RefreshTokenData(APIView)``: Cela définit une classe appelée `RefreshTokenData` qui hérite de la classe `APIView`, ce qui permet de gérer les requêtes liées au rafraîchissement des jetons.

9. ``@HandleError.handle_error("Jam RefreshToken delete -")``: Cela applique un décorateur à la méthode ``post`` pour gérer les erreurs liées à cette méthode de rafraîchissement du token.

10. ``logger.debug("Start RefreshToken delete ")``: Cela enregistre un message de débogage indiquant le début du processus de suppression du jeton de rafraîchissement.

11. La méthode ``post`` vérifie si un jeton de rafraîchissement est présent dans la requête. Si c'est le cas, elle crée un nouvel objet `RefreshToken` avec ce jeton, puis renvoie un dictionnaire contenant les nouveaux jetons d'accès et de rafraîchissement. Si aucun jeton de rafraîchissement n'est présent, elle renvoie un code d'état 401.

12. ``logger.debug("End RefreshToken delete")``: Cela enregistre un message de débogage indiquant la fin du processus de suppression du jeton de rafraîchissement.

13. Enfin, la méthode ``post`` renvoie la réponse avec le dictionnaire et le code de statut approprié.

Front-end :

Vue.js, en tant que framework JavaScript populaire pour la construction d'interfaces utilisateur, met l'accent sur la sécurité. Voici quelques points importants à considérer lors de l'utilisation de Vue.js :

- **Prévention des attaques XSS (Cross-Site Scripting)**

Vue.js est conçu pour prévenir les **attaques XSS** en échappant automatiquement les données interpolées. Cela signifie que les données insérées dans des modèles Vue.js sont automatiquement échappées pour éviter l'exécution de code malveillant provenant de sources non fiables.

Exemple :

Dans le script :

```
<h1>{{ userProvidedString }}</h1>
```

si `userProvidedString` contient:

```
'<script>alert("hi")</script>'
```

ce sera échappé en

```
&lt;script&gt;alert(&quot;hi&quot;)&lt;/script&gt;
```

- **Gestion de la sécurité des routes**

L'utilisation de bibliothèques complémentaires telles que Vue Router pour gérer la navigation et les routes dans une application Vue.js permet de mettre en place des mécanismes de sécurité tels que la vérification des autorisations avant d'accéder à certaines pages ou fonctionnalités sensibles.

- **Utilisation de directives de sécurité**

Vue.js permet la création de directives personnalisées qui peuvent être utilisées pour appliquer des mesures de sécurité spécifiques, telles que la désactivation de certaines fonctionnalités pour les utilisateurs non authentifiés ou la gestion de la visibilité basée sur les autorisations.

- **Gestion sécurisée de l'état de l'application**

Vue.js offre des fonctionnalités pour la gestion de l'état de l'application via des bibliothèques telles que Pinia. L'utilisation appropriée de Pinia permet de maintenir un état d'application sécurisé, notamment en gérant correctement les données sensibles et les informations d'identification des utilisateurs.

- **Attribute bindings**

De même, les liaisons d'attributs dynamiques sont automatiquement échappées en Vue.js. Cela signifie que dans ce modèle :

```
<h1 :title="userProvidedString">  
  hello  
</h1>
```

si `userProvidedString` contenait :

```
'" onclick="alert(\'hi\')'
```

alors il serait échappé au HTML suivant :

```
&quot; onclick=&quot;alert('hi')
```

empêchant ainsi la fermeture de l'attribut `title` d'injecter un nouveau code HTML arbitraire. Cet échappement est effectué à l'aide d'API de navigateur natives, comme `setAttribute`, de sorte qu'une vulnérabilité ne peut exister que si le navigateur lui-même est vulnérable.

- **Potential Dangers**

Dans toute application Web, le fait d'autoriser l'exécution de contenu non sécurisé fourni par l'utilisateur sous forme de HTML, CSS ou JavaScript est potentiellement dangereux et doit donc être évité dans la mesure du possible.

Il y a cependant des cas où un certain risque peut être acceptable.

Par exemple, des services tels que CodePen et JSFiddle permettent l'exécution de contenu fourni par l'utilisateur, mais dans un contexte où cela est attendu et encadré dans une certaine mesure par des iframes. Dans les cas où une fonctionnalité importante nécessite intrinsèquement un certain niveau de vulnérabilité, il appartient à notre équipe d'évaluer l'importance de la fonctionnalité par rapport aux pires scénarios que la vulnérabilité permet.

- **Injecting JavaScript**

Il est déconseillé fortement de rendre un élément `<script>` avec Vue, car les modèles et les fonctions de rendu ne devraient jamais avoir d'effets secondaires. Cependant, ce n'est pas le seul moyen d'inclure des chaînes qui seraient évaluées en tant que JavaScript au moment de l'exécution.

Chaque élément HTML a des attributs avec des valeurs acceptant des chaînes de JavaScript, telles que `onclick`, `onfocus` et `onmouseenter`. La liaison du code JavaScript fourni par l'utilisateur à l'un de ces attributs d'événement constitue un risque de sécurité potentiel et doit donc être évitée.

TIP

Notez que le JavaScript fourni par l'utilisateur ne peut jamais être considéré comme sûr à 100 %, sauf s'il se trouve dans une iframe protégée par un bac à sable ou dans une partie de l'application où seul l'utilisateur qui a écrit ce JavaScript peut y être exposé.

Il est possible de faire du cross-site scripting (XSS) dans les modèles Vue, mais ces cas ne sont pas considérés comme des vulnérabilités réelles, car il n'existe aucun moyen pratique de protéger les développeurs contre les deux scénarios qui permettraient le XSS :

1. Le développeur demande explicitement à Vue de rendre le contenu fourni par l'utilisateur, non nettoyé, sous forme de modèles Vue. Ceci est intrinsèquement dangereux et Vue n'a aucun moyen de connaître l'origine.
2. Le développeur monte Vue sur une page HTML entière qui contient du contenu rendu par le serveur et fourni par l'utilisateur. C'est fondamentalement le même problème que le numéro 1, mais parfois les développeurs peuvent le faire sans s'en rendre compte. Cela peut conduire à des vulnérabilités possibles où l'attaquant fournit du HTML qui est sûr en tant que HTML ordinaire mais non sûr en tant que modèle Vue. La meilleure pratique est de ne jamais monter Vue sur des nœuds qui peuvent contenir du contenu rendu par le serveur et fourni par l'utilisateur.

La règle de sécurité la plus fondamentale lors de l'utilisation de Vue est de **ne jamais utiliser de contenu non fiable comme modèle de composant**. Cela équivaut à autoriser l'exécution arbitraire de JavaScript dans votre application - et pire encore, cela pourrait entraîner des violations de serveur si le code est exécuté pendant le rendu côté serveur. Un exemple d'une telle utilisation :

```
Vue.createApp({
  template: `

` + userProvidedString + `</div>` // NE JAMAIS FAIRE CELA
}).mount('#app')


```

Les modèles Vue sont compilés en JavaScript et les expressions contenues dans les modèles seront exécutées dans le cadre du processus de rendu. Bien que les expressions soient évaluées par rapport à un contexte de rendu spécifique, en raison de la complexité des environnements d'exécution globaux potentiels, il n'est pas possible pour un framework comme Vue de vous protéger complètement de l'exécution potentielle de code malveillant sans encourir une surcharge de performance irréaliste. La façon la plus simple d'éviter complètement cette catégorie de problèmes est de s'assurer que le contenu de vos modèles Vue est toujours fiable et entièrement contrôlé par nous.

- **Best Practices**

En règle générale, si on autorise l'exécution d'un contenu non sécurisé fourni par l'utilisateur (HTML, JavaScript ou même CSS), nous nous exposons à des attaques. Ce fait est valable que nous utilisons Vue, un autre framework ou même aucun framework.

Il est donc conseillé dans Vue.js de **ne jamais utiliser de contenu non fiable comme modèle de composant, mais d'utiliser** des modèles ou des fonctions de rendu, ou le contenu est automatiquement échappé.

Pour garantir la sécurité de notre front-end Vue.js, des stratégies appropriées de gestion des autorisations et des identités ont été également mis en place. Par exemple la route /dashbord est accessible seulement par un utilisateur qui est connecté et qui a le rôle admin. Pour pouvoir passer une commande l'utilisateur doit être tout d'abord connecté, et le token doit être valide.

Conclusion

La refonte de cette application de vente de confitures, passant d'une architecture basée sur Symfony en une application Web moderne reposant sur une architecture API avec un backend Django et un frontend Vue.js, a été réalisée avec succès.

Au cours du processus de développement de cette application, j'ai acquis de nouvelles compétences et mis en place des fonctionnalités que je n'avais pas utilisé avant et que je voulais tester.

En modernisant l'infrastructure technologique, la flexibilité, la scalabilité et les performances de l'application, ont été améliorés et offre une expérience plus fluide et réactive.

L'intégration d'un filtre à facettes robuste, permettant de parcourir et de sélectionner facilement des produits en fonction de critères spécifiques, a grandement enrichi leur expérience de navigation

En utilisant la puissance de Docker, un environnement de développement cohérent et isolé a été construit et garantit la portabilité et la facilité de déploiement de l'application sur diverses plates-formes et systèmes d'exploitation.

En suivant attentivement les instructions d'installation, il est possible de mettre en place rapidement l'application JAM sur toute autre machine. Le fichier README.md ainsi que le chapitre 6 de ce document, fournissent des directives détaillées sur les étapes spécifiques à suivre, y compris les dépendances, les commandes et les configurations essentielles pour assurer un déploiement fluide.

En somme, cette refonte complète de l'application JAM m'a permis d'apprendre des nouvelles choses, et aussi de positionner l'application pour répondre aux exigences technologiques actuelles, tout en offrant une infrastructure robuste et évolutive pour soutenir la croissance continue de l'application.