CESI_
ÉCOLE D'INGÉNIEURS

MAALSI 2022/24 - BLOC 3 - Groupe 5

Superviser et assurer le développement des applications logicielles

François ANDRZEJEWSKI, Paul BOIREAU, Julien HUYGHE

février 2024

Table des matières

Table des figures

Liste des tableaux

Introduction

Figure 1: Logo de Breizhsport

La direction de Breizhsport, société spécialisée dans la vente de matériel de sport, a décidé d'étendre ses activités aux travers du lancement d'une plateforme en ligne de e-commerce sur internet. Située à Rennes en région bretonne, elle dispose de bureaux à Brest et Lorient, d'une usine de fabrication sur Brest, ainsi que des magasins dans les principales villes de Bretagne : Brest, Quimper, Lorient, Vannes, Rennes, Saint-Malo, etc.
Suite à cette envie de s'étendre, elle a demandé à sa DSI[1] de moderniser le système d'information existant, ainsi que les pratiques de développement qui n'ont guère évoluées lors de la dernière décenie. Celle-si se compose d'environ cinquant personnes, dont vingt-cinq développeurs, cinq chefs de projets, dix administrateurs système, cinq SRE et cinq responsables.

À partir de l'existant, à savoir des applications en PHP/MySQL sur des serveurs virtuels chez un hébergeur français et méthodologie de cycle en V, la direction souhaite orienter la modernisation du système d'information vers une architecture « Cloud Native » avec déploiement sur un cloud public, tout en profitant de l'occasion pour industrialiser un environnement de développement en y incluant un dépôt de code source basé sur Git, ainsi qu'une pipeline de CI/CD[2].

Au cours de ce rapport, nous définirons dans un premier temps les nouveaux principes d'architecture, de conception, de déploiement et de maintenance des applications du futur SI[3] de Breizhsport. Puis, nous verrons l'application développée à partir de ses principes.

[1] Direction des Systèmes d'Informations
[2] CI/CD - Continuous Integration, Continuous Development
[3] SI - Système d'Informations

Système de veille active

En plus de l'expression du besoin de la modernisation du SI de Breizhsport, la direction a constaté que les pratiques de développement de l'entreprise sont vieillissantes et souhaite également les morderniser. Étant à la tête de l'équipe de développeur, il est nécessaire de mettre en place un système permettant la montée puis le maintien en compétences des parties prenantes concernant le développement et la mise en production des éléments du sI.

Définition du système de veille

Un système de veille peut être vu comme un ensemble d'éléments qui permettent le suivi des actualités afin de rester au courant des bonnes pratiques, d'outils, de failles de sécurité, etc. On peut également définir la veille active, qui est la surveillance de l'environnement d'une entreprise. On peut par exemple avoir une veille stratégique pour déterminer les modes de consommation, une veille concurrentielle, sectorielle et commerciale afin de surveiller ses concurrents, leurs stratégies marketing et commerciales, etc. Nous cherchons donc à mettre en place un système de veille informatique au sens large, afin de se tenir au courant des bonnes pratiques et outils permettant de gagner en qualité, réduire les coûts ou combler les failles de sécurité, ceci dans le but d'éviter une nouvelle refonte dans quelques années.

Système de veille active

Pour mettre en place un système de veille active, il faut que celui-ci soit bien défini au préalable, avec des objectifs, des rôles et des méthodes de fonctionnement claires.

Objectifs

Dans un premier temps, il convient de définir les objectifs de celui-ci dans le cadre de la modernisation du SI de Breizhsport :

- Améliorer la qualité des produits et services développés;
- Actualiser les compétences des membres de l'équipe sur les dernières tendances et innovations;
- Réduire les risques de dysfonctionnement lors de la mise en production;
- Pouvoir réagir rapidement en cas de faille de sécurité découverte sur un ou plusieurs outils.

Organisation

Dans un deuxième temps, on souhaite organiser notre système de veille de la façon suivante :

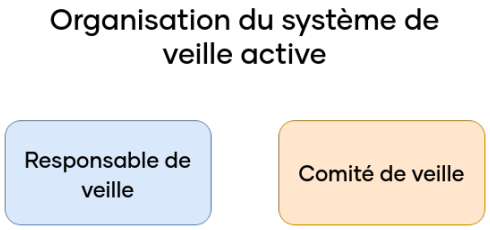

Figure 2: Organisation du système de veille active

- Un responsable de la veille active, qui sera chargé de la coordination du processus. Il sera responsable de la définition des thèmes de veille, de la collecte des informations, de leur traitement et de leur diffusion;
- Un comité de veille qui sera constitué de membres de l'équipe. Il sera consulté par le responsable de la veille pour la définition des thèmes de veille et pour la validation des informations collectées.

Ce système à deux acteurs permet une définition claire des rôles tout en impliquant le plus d'individus possible, ce qui contribue au partage de connaissances. Dans notre cas, le tech leader est le responsable de la veille active, tandis que le comité de veille est constitué des cinq développeurs de notre équipe.

Méthodes de collecte d'informations

Dans un troisième temps, nous allons nous intéresser aux méthodes pour collecter des informations depuis certaines sources. Parmi les sources qui peuvent nous intéresser, nous retrouvons :

- Les publications scientifiques, comme les revues spécialisées : *IEEE Software*, *ACM Transactions on Software Engineering and Methodology*, *Journal of Systems and Software*.
- Les conférences et événements professionnels : *Microsoft*, *Google*, *IBM*, *ACM SIGSOFT Conference on Software Engineering*, *IEEE International Conference on Software Engineering*, *European Conference on Software Engineering*.
- Les blogs et sites web spécialisés : *Medium*, *Wired.com*, *Google Feed*, *StackExchange*.
- Les réseaux sociaux : *LinkedIn*, *Quora*.

En variant les sources, nous pouvons faire varier les informations et la façon dont elles sont apportées. Pour pouvoir les collecter, il existe différentes méthodes plus ou moins automatisées, comme par exemple la connexion à des flux RSS [4] ou Atom, ou encore l'utilisation d'application dédiée comme Feedly. Il est également intéressant, selon notre point de vue, d'organiser une réunion bi-mensuelle afin de pouvoir discuter librement de sujets afin de les partager à l'ensemble de l'équipe. Enfin, on peut proposer des webinaires et conférences proposés par des entreprises externes aux membres de l'équipe pour qu'ils collectent des informations et les fassent remonter au reponsable de veille. Un tableau récapitulatif des sources est disponible plus bas.

Traitement des informations

Les informations collectées sont à traiter par le responsable de la veille. Son but est alors d'analyser les informations pertinentes et utiles, de les synthétiser, et de les rendre accessibles aux membres de l'équipe au travers d'une plateforme de diffusion que l'on verra par la suite. On souhaite que les informations, après traitement, soient conformes à la Réglementation Générale sur la Protection des Données (RGPD), ainsi qu'elles restent sourcées.

Diffusion des informations

Enfin, la dernière étape pour les informations est de les partager à l'ensemble des membres de l'équipe. Le format et la fréquence de partage peuvent varier, comme son accès. Il peut soit être sur la base du volontariat en utilisant un système de tableau de bord pour retrouver des informations, ou bien plus poussé en leur envoyant une newsletter et les faisant participer à des réunions de partage évoquées précédemment. Le format idéal selon nous, est de fournir un résumé des informations à l'ensemble des collaborateurs, et centraliser des informations pour ceux qui souhaiteraient en savoir plus.

Utilisation

Le système que l'on vient de décrire permet de collecter des informations de la part de plusieurs utilisateurs, qui seront à l'origine de décisions lors de la vie du projet, ainsi que lors de son maintien une fois la modernisation accomplie. Cependant, celui-ci est amené à évoluer, tant en terme de sources de veille, que de mode de fonctionnement, que de partage de l'information.
Celui-ci peut être adapté afin de cibler des thèmes de veille plus spécifiques en fonction des besoins, ou des modes de communication qui seraient mieux integrés dans le fonctionnement des équipes.

[4] RSS - Really Simple Syndication

Liste des principales sources d'informations

Voici une liste non-exhaustive de sources sur lesquelles on souhaite s'appuyer afin d'organiser la veille active. Cette liste sera amenée à évoluer dans le temps en fonction des besoins et des retours d'expérience.

Nom	Type	Commentaire
IEEE Software	Revue spécialisée	Revue bi-mensuelle sur des thématiques informatiques.
ACM Transactions on Software Engineering and Methodology	Revue spécialisée	Revue quadri-annuelle sur des thématiques informatiques.
Journal of Systems and Software	Revue spécialisée	Revue mensuelle sur des thématiques informatiques.
Microsoft Envision / Ignite / Secure / Inspire / Build / Virtual Training Days	Conférence	Lot de conférences Microsoft annuelles sur des thématiques variées.
Google Next, Techbyte	Conférences et ateliers	Lot de conférences et Ateliers organisés par Google.
IBM TechXchange et Think	Conférences	Lot de conférences organisés par IBM.
ACM International Conference on the Foundations of Software Engineering (FSE)	Conférence	Conférence organisée par le FSE autour des tendances, innovations et défis actuels du développement logiciel.
International Conference on Software Engineering	Conférence	Conférence internationale autour du développement logiciel.
European Symposium on Software Engineering	Conférence	Conférence européenne autour du développement logiciel.
Medium	Site web	Ensemble d'articles spécialisés sur des thèmes variés.
Wired	Site web	Site assez grand public qui regroupe des articles sourcés sur des thèmes variés.
Google News	Site web	Google News rassemble un ensemble d'articles de sites variés autour d'une thématique.
StackExchange	Site web	StackExchange regroupe un ensemble de sites dédiés à des thématiques, où la communauté peut poser des questions et y répondre.
LinkedIn	Réseau social	LinkedIn est le réseau social professionnel, où des entreprises et particuliers partagent des informations sur des sujets d'actualité.
Quora	Réseau social	Quora est un réseau social où l'on retrouve un système de questions réponses, ainsi que des articles écrits par des amateurs sur des thématiques données.

Figure 3: Liste non-exhaustive de sources de veille.

Architecture proposée

Comme vu précédemment lors de l'introduction, la direction souhaitant se lancer sur le créneau de la vente en ligne, nous a demandé de mettre en place une plateforme de e-commerce. Cette application est faite et maintenue par les développeurs en interne de l'entreprise. Au cours des pages précédentes, nous verrons les critères nous permettant de choisir une solution pour répondre à ce besoin, puis nous verrons comment nous allons mener le projet avec le choix d'une méthode de travail, ainsi que l'architecture et l'hebergement de cette application.

Benchmark des méthodes de développement

À la demande de la direction, on souhaite moderniser la méthodologie de développement en passant d'une méthode de cycle en V, à une méthode plus adaptée à des besoins informatiques.
Ainsi, nous allons établir un comparatif afin d'orienter et d'évaluer nos choix, pour qu'ils correspondent au mieux à notre besoin de modernisation du SI de Breizhsport.

Critères

Nous souhaitons évaluer différentes méthodes de travail sur des critères en adéquation avec l'environnement où elle sera utilisée : le développement informatique. Cet environnement est connu pour avoir des besoins et des priorités qui changent très rapidement, et auxquels il est nécessaire de devoir s'adapter afin de mener correctement le projet à terme.

Traditionnellement, on dispose de deux types de méthodes de travail : celles utilisant une approche dite prédictive, et celles utilisant une approche dite itérative.

- Le premier est le plus simple, au début du projet on va lister et ordonner toutes les tâches du projet qui sont nécessaires pour mener le projet à terme. Cette approche est surtout utile lorsque le besoin ne risque pas d'évoluer et que l'on souhaite avoir une bonne estimation des coûts et du temps de travail comme ça pourrait être le cas dans le BTP par exemple.
- La seconde est dit itératif, car à chaque cycle, chaque itération de celui-ci, on va évaluer le besoin pour définir les priorités de travail. Les cycles et la liste des tâches sont définis au début du projet, mais ils peuvent évoluer au cours de la vie du projet.

Ainsi, chaque méthode de travail est basée sur une approche qui dispose d'avantages par rapport à l'autre. Les méthodes traditionnelles ou de cycle en V utilisent des approches prédictives, là où des méthodes comme le DevOps ou SCRUM sont itératives.
Chaque projet étant unique tant concernant la taille, la complexité ou encore les contraintes de temps

et de budget, il nous faut trouver celle qui est le plus adaptée à notre projet de modernisation du SI de Breizhsport.

Afin d'appuyer nos propos, il nous faut nous baser sur des critères de comparaison. Dans notre cas, nous avons choisi les suivants :

- **Flexibilité** : La méthode permet-elle de s'adapter aux changements de besoins ou de priorités ?
- **Réactivité** : La méthode permet-elle de délivrer des livrables rapidement ?
- **Qualité** : La méthode permet-elle de produire des logiciels de qualité ?
- **Collaboration et communication** : La méthode favorise-t-elle la collaboration et la communication entre les différents membres de l'équipe ?

Comparaison

Nous allons comparer quatre méthodes de développement qui sont couramment utilisées : Cascade (aussi appelée traditionnelle ou waterfall), SCRUM, DevOps et Kanban.

Cascade La méthode en Cascade est probablement la méthode prédictive la plus simple : Toutes les tâches sont définies en avance, elles se suivent dans un ordre séquentiel : chaque tâche doit être complétée avant de pouvoir passer à la suivante. Pour du développement informatique, on pourrait la décomposer en quatre phases : étude, conception, développement, tests et déploiement de la solution. L'avantage étant que cette méthode est simple à comprendre et à mettre en place. Cependant, elle s'avère inadaptée sur plusieurs points; elle est peu flexible dans le sens où il est difficile de s'adapter à un changement de besoin ou de priorité, ce qui est courant dans notre contexte. De plus, elle peut s'avérer longue et coûteuse, et est plutôt dimmensionnée pour des petits projets.

SCRUM SCRUM est une méthode ayant une approche itérative qui comporte des cycles courts (appelés sprints) d'environ deux à quatre semaines. Un sprint est composé des phases suivantes :

- **Sprint planning** (avant): Planification des tâches à effecter durant le sprint.
- **Daily** (pendant) : Réunion quotidienne pour suivre l'avancement du sprint.
- **Sprint Review** (après): Présentation du travail accompli durant le sprint.
- **Retrospective** (après): Réflexion sur le sprint dans une optique d'amélioration continue.

Cette méthode presente plusieurs avantages. Pour commencer, elle favorise le changement de besoins ou de priorités car elle se veut de nature flexible et réactive. De plus, elle favorise la collaboration et la communication entre les différents membres de l'équipe qui ont la liberté de s'auto-organiser pendant un sprint.
Cependant, c'est une méthode assez différente de ce que les équipes peuvent connaître, et qui nécessite souvent une réorganisation des équipes pour pouvoir l'exploiter.

DevOps Le DevOps n'est pas à proprement parler une méthode de travail, mais plutôt un concept sur lequel les équipes peuvent se baser pour adapter leurs méthodes de développement. Le but est de rapprocher les équipes de développement et celles d'exploitation afin d'améliorer la collaboration et la communication entre ces deux équipes, pour réduire le temps de mise en production des logiciels.

Celle-ci favorise la collaboration et la communication entre les équipes de développement et d'exploitation, réduit le temps de mise en production, et améliore la qualité des logiciels. Cependant, comme SCRUM, elle peut être compliquée à mettre en place dans certaines entreprises car elle nécessite souvent une réorganisation des équipes.

Kanban Kanban est une autre méthode de type itérative. Celle-ci est beaucoup lié au principe de lean management, qui vise à mieux gérer les flux de travail afin de favoriser la qualité tout en réduisant les déchets.

Son principal avantage est d'être très visuelle. Au moyen d'un tableau, les tâches représentées sous la forme d'une carte sont disposées en colonnes en fonction des phases du projet. Ainsi, non seulement on gagne en flexibilité en pouvant s'adapter aux besoins et priorités, mais on gagne en communication entre les équipes puisque chacun connait l'état du projet. Cependant, ce mode de travail peut vite être surchargé pour les gros projets, et n'est pas toujours facile à mettre en place.

Choix

À partir des éléments précédents, nous avons pu établir une matrice de choix basée sur les critères choisis précédemment. La notation se faisant de 1 à 4 afin d'éviter une note moyenne sans être trop simple ni trop complexe, avec 1 représentant un choix inadapté, et 4 un choix adapté.

Table 1: Matrice de choix de la méthode de travail

Critère/Méthode	Cascade	SCRUM	DevOps	Kanban
Flexibilité	1	4	3	3
Réactivité	1	4	4	3
Qualité	4	3	4	3
Collaboration et communication	1	4	3	2
Total	**7**	**15**	**14**	**11**

À partir de cette matrice, nous en concluant donc que la méthode la plus adaptée pour nos besoins de modernisation du SI de Breizhsport est la méthode SCRUM. Ce choix semble logique puisque la méthode SCRUM a été pensée pour des besoins de développement informatique contrairement à une méthode en Cascade. Néanmoins cela ne signifie pas que cette dernière est mauvaise, elle est juste inadaptée à nos besoins pour ce projet.

Benchmark de l'architecture

Toujours dans le cadre de la modernisation du SI de Breizhsport qui nous a été confié, la direction nous a demandé de baser le développement de l'application de vente en ligne sur des technologies modernes. Ainsi, elle souhaite que l'on oriente celle-ci sur les piliers suivants :

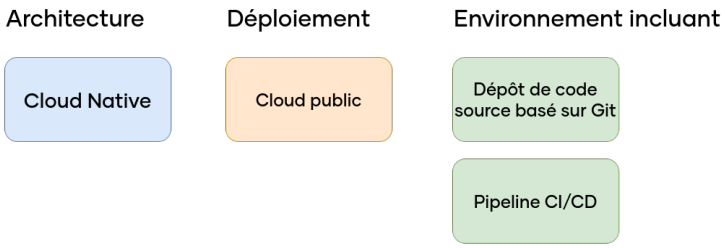

Figure 4: Piliers sur lesquels baser l'hébergement de l'application.

Choix de l'architecture

L'architecture d'une application représente la façon dont celle-ci est construite, est organisée, la façon dont elle fonctionne. Bien que chaque application soit unique, cela permet nos choix en terme de maintenance, complexité, de sécurité ou de redondance, chaque architecture ayant ses avantages et ses défauts.

Les différents critères que nous avons retenus sont les suivants. Ceux-ci sont liés à la fois au cycle de développement et au cycle de maintenabilité au cours de la vie de l'application.

- **Complexité** : L'architecture est peu complexe et permet une organisation assez simple ?
- **Flexibilité** : L'architecture est aisément adaptable à de nouveaux besoins ?
- **Maintenabilité** : L'architecture permet une maintenance aisée de l'application ?
- **Adaptée à de gros projets (AGP)** : L'architecture facilite le développement d'application complexe ?

Concernant les différentes architectures, nous allons en comparer quatre :

- **Monolithique** : Toutes les fonctionnalités de l'application sont regroupées dans un seul même bloc.
- **Microservices** : Les fonctionnalités sont découpées en petits blocs indépendants et capable de communiquer entre eux.
- **Headless** : La partie « front-end » (visible des utilisateurs) est séparée de la partie « back-end » (traitement et stockage de données).
- **Modèle-vue-contrôleur (MVC)** : Les données, la présentation et la logique métier sont séparées.

Architecture monolithique L'architecture monolithique est l'architecture la plus simple, toutes les fonctionnalités sont regroupées ensemble. Cette solution présente l'avantage d'être simple à développer et à maintenir, mais au prix d'une faible flexibilité où il est compliqué de la faire évoluer en cas d'évolution du besoin.

Architecture en microservices L'architecture en microservices est un peu l'opposé de l'architecture monolithique, toutes les fonctionnalités sont organisées en blocs indépendants entre eux, parfois même programmés avec des technologies différentes. Bien qu'une telle architecture complexifie le développement et la maintenance de l'application, on gagne en flexibilité et capacité d'évolution. Il est très aisé de rajouter un bloc de fonctionnalités correspondant à un nouveau besoin, puisque les blocs sont indépendants et ne font que communiquer entre-eux.

Architecture headless Ce format d'architecture est un peu moins courant que les deux précédentes. On peut la voir comme un mélange entre les deux précédents : toutes les fonctionnalités liées au stockage et le traitement des données (back-end) va être regroupée dans un bloc commun, mais les interfaces utilisateurs vont être indépendantes et communiquer avec le back-end au travers d'une API [5]. Ce format permet une grande flexibilité, car le font-end et le back-end sont isolés, mais présente l'inconvenient d'être assez complexe, surtout si l'on dispose de plusieurs interfaces utilisateurs.

Modèle-vue-contrôleur (MVC) Comme son nom l'indique, l'architecture modèle-vue-contrôleur sépare les données, de l'interface, et de la logique métier. Ces modules séparés facilitent le développement et la maintenance. Il faut toutefois garder en tête qu'un tel modèle ne favorise pas la flexibilité de l'application en cas de nouveaux besoins. Néanmoins, cela reste un modèle couramment utilisé pour les applications web et mobiles.

[5] API - Application Programming Interface - Boite à outils qui permet de communiquer depuis et vers un logiciel

Grille de choix

Maintenant que nous avons différents critères et solutions, nous pouvons les évaluer dans le tableau ci-dessous. La notation est sur une échelle de 1 à 4, 1 étant un « non », 4 étant un « oui » aux questions définies pour les critères.

Table 2: Grille de choix de l'architecture de l'application.

Critère / Architecture…	Monolithique	Microservices	Headless	MVC
Complexité	4	2	2	2
Flexibilité	1	3	3	2
Maintenabilité	2	3	3	3
AGP	1	4	3	3
Total	**8**	**12**	**11**	**10**

Cette grille de choix nous permet de déterminer qu'une architecture en microservices est la plus adaptée en fonction de nos critères et de notre contexte de projet. De plus, ce choix d'architecture s'associe bien avec le mode d'hébergement choisi que nous verrons par la suite. Cependant, comme nous le verrons par la suite nous allons privilégier l'utilisation d'un CMS à une application en microservices, pour des questions de simplicité en lien avec le besoin de l'entreprise.

Recherche de solution

Pour créer un site internet avec des fonctionnalités de vente en ligne, il nous faut impérativement trouver une solution vers laquelle nous orienter. Les possibilités sont vastes, très vastes mêmes puisque l'on estime qu'il y aurait eu 207 000 sites de vente en ligne actifs à la fin de l'année 2022 en France.[6] Ainsi, comme nous l'avons vu auparavant, l'idéal serait de trouver une solution avec une architecture en microservices. Cependant, afin de trouver une solution qui soit adaptée au besoin et au contexte de Breizhsport, il nous faut mettre en place des critères à évaluer pour différentes solutions que l'on envisage, sans prendre en compte le choix de l'architecture vu précédemment.

Nous définissons les critères suivants :

[6] Étude réalisée par FEVAD (Fédération du e-commerce et de la vente à distance)

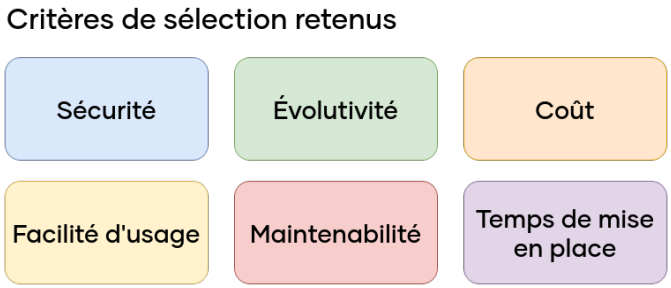

Figure 5: Critères retenus pour notre recherche de solution.

- **Sécurité** : Concerne le niveau de sécurité de la solution.
- **Évolutivité** : Est-ce que la solution peut évoluer aisément dans le temps pour obtenir de nouvelles fonctionnalités ?
- **Coût** : La solution est-elle peu onéreuse ?
- **Facilité d'usage** : La solution est-elle simple à mettre en place ? Ceci est lié aux compétences des développeurs qui aura été établi.
- **Maintenabilité** : Est-il facile de réaliser la maintenance de la solution ?
- **Temps de mise en place** : Dans le contexte de Breizhsport qui souhaite se lancer rapidement sur le marché du e-commerce, un POC de la solution est-il envisageable à court-terme ?

Ces critères nous permettent de considérer plusieurs points-clés assez facilement, nous permettant de nous guider dans notre prise de décision.
Pour les solutions envisagées, nous en avons trois :

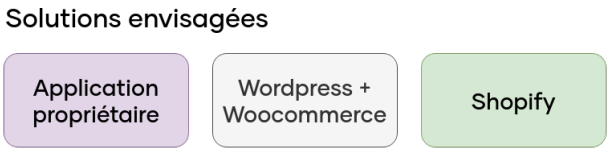

Figure 6: Solutions envisagées.

Application propriétaire

L'application propriétaire serait une solution faite en interne par notre équipe de développement en utilisant uniquement des bibliothèques externes pour la réaliser. Il y a plusieurs avantages à la choisir, la principale étant qu'on aurait une compréhension totale de son fonctionnement, et que l'on a le choix de son architecture, ce qui lui confère une grande évolutivité, maintenabilité, et facilité d'usage pour les développeurs. Cependant, cela se ferait au prix d'un temps de développement assez long, et coûterait bien plus cher que des solutions alternatives. En terme de sécurité, elle reste assez neutre puisque d'un côté on ne disposerait que des fonctionnalités dont on a besoin, ce qui permet de limiter les failles de sécurité potentielles, mais notre équipe ne diposant pas de membre dédié à la sécurité, il sera difficile de découvrir des failles et de les combler sans devoir faire appel à un prestataire externe.

Wordpress et Woocommerce

Une des solutions clé-en-main que l'on aurait serait d'utiliser un CMS [7] comme Wordpress afin de créer un site internet. À cela, on rajouterait une extension du nom de Woocommerce afin de pouvoir gérer une boutique en ligne. Cette combinaison offres plusieurs avantages, dans un premier temps Wordpress est la solution la plus utilisée au monde : selon W3Techs, Wordpress faisait tourner 42% de tous les sites web sur internet en 2022. Étant un outil open-source et maintenu par une communauté assez active, il peut être considéré comme assez bien sécurisé. Il dispose d'un magasin d'extensions afin de pouvoir l'adapter pour faire ce dont on a besoin, comme par exemple une boutique e-commerce. Gratuit, il est également très simple d'utilisation avec une documentation fournie. Enfin, il peut rapidement être mis en place afin de répondre à un besoin court. Le principal désavantage qu'on pourrait lui trouver est qu'il embarque plus de fonctionnalités qu'on aurait besoin, ce qui complexifie sa compréhension par les équipes dans le cas où il y aurait un problème avec celui-ci.

Shopify

Comme Wordpress, Shopify est un service en ligne qui permet de créer une boutique en ligne sans avoir à taper une seule ligne de code. Étant un outil dédié au e-commerce, il dispose d'une bonne documentation, un bon support, et une bonne sécurité face aux menaces. Il est également très rapide de créer sa boutique en ligne avec celui-ci, qui s'occupe de tout. L'idée est de fournir une solution tout en un que des non-informaticiens peuvent utiliser pour ouvrir une boutique en ligne. Cependant, il vient avec quelques inconvénients, pour commencer il s'avère assez onéreux tant en terme de coûts que de commission sur les transactions. De plus, il est difficile de le faire évoluer à l'avenir pour faire autre-chose que du e-commerce, ce qui peut être bloquant en cas d'un besoin changeant.

[7] CMS - Content Management System / Système de Gestion de Contenu

Matrice de choix

À partir des éléments, il nous est possible de comparer ces solutions en utilisant une matrice de choix. La notation est 1-4, 1 étant défavorable, 4 étant favorable.

Critère / solution	Solution propriétaire	Wordpress + Woocommerce	Shopify
Sécurité	3	4	4
Évolutivité	4	4	2
Coût	1	4	2
Facilité d'usage	2	3	4
Maintenabilité	4	2	2
Temps de mise en place	1	3	4
Total	15	20	18

On peut ainsi déterminer que la solution la plus adaptée à notre besoin est Wordpress et l'extension *WooCommerce*, sur les différents critères évoqués précédemment. Ce CMS dispose de sa propre architecture qui n'est pas celle que l'on a estimée auparavant. Cependant, nous estimons que sa complexité, son support et le fait que ce soit une solution mature prime sur une solution propriétaire basée sur une architecture en microservices, qui peut s'avérer être une tâche complexe étant donné le nombre de membres de l'équipe.

Par la suite, nous allons faire un comparatif des solutions pour pouvoir l'héberger, et le déployer pour répondre à notre besoin de modernisation du SI.

Choix du mode d'hébergement

Outre le développement de l'application en elle-même, il est également important de déterminer le moyen de distribution auprès des utilisateurs finaux (les clients) le plus adapté à notre besoin, en se basant sur certains critères.
La direction nous demande d'envisager une solution cloud, qui présente de nombreux avantages comme une très bonne flexibilité, une grande évolutivitée et une sécurité renforcée.

Ainsi, nous allons définir les critères suivants pour orienter notre choix :

- **Sécurité** : La solution propose-t-elle des outils pour se protéger des attaques extérieures ?
- **Flexibilité** : La solution peut-elle évoluer facilement face à de nouveaux besoins ?
- **Coûts** : La solution est-elle peu onéreuse, tout service compris ?

Il existe trois méthodes principales pour héberger une application web afin de la distribuer en ligne :

- **Hébergement sur site (on-premise)**
- **Hébergement mutualisé**
- **Hébergement cloud**

Par la suite, nous allons évaluer ces solutions sur les critères précédémment choisis afin d'orienter notre choix.

Hébergement sur site (on-premise) Comme son nom l'indique, l'hébergement sur site consiste à disposer de sa propre infrastructure dans les locaux de l'entreprise afin de distribuer des solutions. Cette solution est la plus flexible, dans le sens où l'on fait ce que l'on veut dessus et elle ne présente pas de problèmes quant à la confidentialité des données que l'on pourrait avoir avec un hébergeur public. Cependant, cela signifie également que l'on ne dispose d'aucun support, et si l'on ne dispose pas d'équipe liée à la sécurité, elle peut présenter de grandes vulnérabilités. Également, le coût est plus cher à l'acquisition et nous faisons face à des frais de fonctionnement et de maintenance. En somme, cette solution est surtout adaptée pour les grandes qui disposent d'équipes dédiées, et dont l'aspect sécurité et confidentialité est plus important que le critère de coûts.

Hébergement mutualisé Son nom étant encore une fois explicite, un hébergement mutualisé est un hébergement partagé entre plusieurs entreprises et clients. Le but est de réduire un maximum les coûts en optimisant les ressources disponibles, mais cela se fait au détriment de la flexibilité, des performances, et du contrôle de l'infrastructure. Cette solution est surtout adaptée à de petis projets qui ne nécessitent pas beaucoup de ressources.

Hébergement cloud Aussi appelé « cloud computing », le but est d'utiliser des serveurs à distance pour distribuer une solution en stockant et traitant les données associées. La différence avec les deux solutions précédents étant que l'on ne dispose pas d'un serveur en particulier, mais d'un ensemble de serveurs. Le principal avantage étant que la solution peut s'adapter en temps réel à la demande, en étant disponible sur plusieurs serveurs à la fois afin de répartir la charge. Les coûts sont notamment dépendants du traffic, c'est-à-dire que plus il y aura de traffic, plus les coûts seront élevés. Cependant, cette solution reste moins cher qu'une solution sur site. Enfin, tous les principaux fournisseurs de services cloud (Google, Amazon, Microsoft, IBM, OVH, etc.) disposent d'équipes de support et de sécurité. Ainsi, on externalise ces activités pour se concentrer sur l'application en elle-même.

Grille de choix

Afin de visualiser et de comparer ces différentes solutions et critères, nous pouvons les mettre sous forme de tableau comme ci-dessous. La notation est sur une échelle de 1 à 4, 1 étant un « non », 4 étant un « oui » aux questions définies pour les critères.

Table 4: Grille de choix de la solution d'hébergement.

Critère / Hébergement…	Sur site	Mutualisé	Cloud
Sécurité	4	2	3
Flexibilité	2	1	4
Coûts	1	4	3
Total	**7**	**7**	**10**

Ainsi, à l'aide de ces trois critères de sécurité, flexibilité et coûts, nous envisageons d'orienter le choix de la solution d'hébergement vers un hébergement cloud. Cela correspond bien à la volonté de l'entreprise, tout en étant en adéquation avec notre contexte. Pour rappel, nous ne disposons pas d'équipe pouvant gérer un hébergement en interne, et nous souhaitons avoir de la flexibilité au niveau de notre hébergement.

Ces choix, celui de la solution et de la méthode d'hébergement, nous permettent de valider la demande de la direction d'envisager une infrastructure cloud avec des fonctionnalités cloud native. Pour valider ce fonctionnement, nous mettrons en place dans un premier temps un POC [8] afin de valider son fonctionnement et ses fonctionnalités pour ensuite proposer l'application de production.

[8] POC - Proof Of Concept

Schéma d'architecture et de déploiement

Suite aux choix fonctionnels, nous souhaitons mettre en place un POC qui reprend les fonctionnalités principales de l'application, pour nous permettre d'à la fois valider les choix techniques, et de prendre des décisions afin de les adapter à notre besoin et au contexte de l'entreprise.

Par la suite, nous verrons comment les choix techniques effectués précédémment ont été traduits afin de répondre au besoin de Breizhsport de développer une application de e-commerce permettant la vente en ligne.

Schéma d'architecture commenté

Comme vu précédémment, nous souhaitons mettre en place une solution Wordpress avec une extension Woocommerce pour répondre au besoin de la direction.

Wordpress en lui-même a une architecture assez proche d'une architecture MVC vu précédemment, à quelques différences prêt.

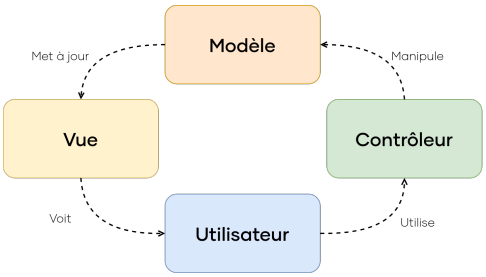

Figure 7: Schéma d'une architecture MVC

Tel que Wordpress a été fait, la « Vue » est au centre de l'application, où qui va pouvoir obtenir des données du « Modèle » et non l'inverse dans une architecture MVC. De plus, la « Vue » fourni une certaine logique métier au « Contrôleur », sans nécessairement passer par l'utilisateur.
Pour vulgariser, on peut imaginer que Wordpress reçoive une URL. Un module de Wordpress, le « Core » agît en tant que contrôleur et détermine les requêtes initiales à exécuter sur la base de données, et par extension, la vue qui doit être chargée. Il met ensuite en forme la réponse à la requête initiale et l'envoie au fichier de la vue. Ce fichier de vue peut être un fichier strictement réservé à l'affichage, ou bien il peut demander des informations supplémentaires par rapport à celles qui sont intégrées. C'est le type « pull » du MVC, où la vue tire les données du modèle au lieu que le « contrôleur » « pousse » les données du modèle dans la vue.

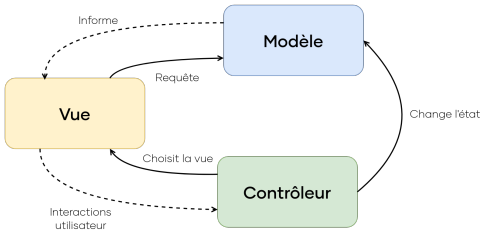

Figure 8: Schéma de vulgarisation de l'architecture de WordPress

Ainsi, pour fonctionner, notre WordPress a besoin de plusieurs éléments :

- Lui-même, à savoir ses fichiers « core », mais également un thème pour la vue et d'éventuelles extensions pour ajouter des fonctionnalités à notre application.
- Woocommerce, qui est l'extension nous permettant de gérer notre boutique en ligne.
- Une base de données, pour stocker et organiser toutes les informations nécessaires de notre application.
- Un serveur web, lui permettant de s'éxécuter, de recevoir et d'émettre des requêtes. Ce serveur web certains prérequis, il lui faut notamment PHP[9], langage dans lequel WordPress est en grande partie programmé.

Avec ces éléments, nous pouvons représenter notre application de la façon suivante :

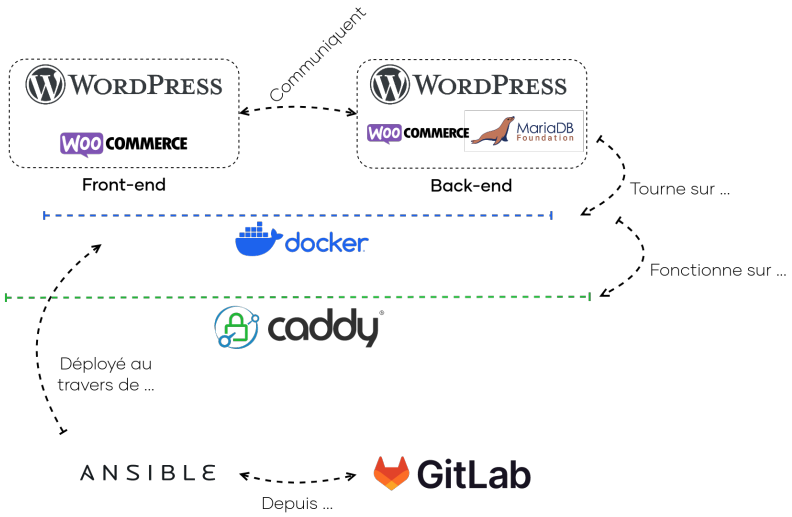

Figure 9: Schema de principe du POC de l'application de e-commerce

[9]PHP - PHP: Hypertext Preprocessor est un langage de script utilisé le plus souvent côté serveur.

Sur le schéma précédent, la même instance de WordPress et de Woocommerce sont bien à la fois côté « Front-end » et « Back-end ». Voici les choix technologiques que nous avons fait en vue de mettre en place le POC :

Base de données - MariaDB **MariaDB** est un système de gestion de base de données open-source, qui est basé sur **MySQL**, un autre système de gestion de base de données. C'est une solution populaire qui outre le fait d'être assez performant et bien suivi, à plusieurs avantages; il dispose notamment d'outils contre des injections SQL ou par force brut, et se veut stable et fiable.

Serveur web - Caddy Notre choix de serveur web s'est porté sur **Caddy**, pour plusieurs raisons. Bien que Apache et NGinx soient les plus répandus, Caddy présente plusieurs avantages assez intéressant pour nous. Dans un premier temps, celui-ci est simple à configurer, dans le sens où celui-ci utilisant un système de configuration déclaratif, il n'est pas aussi verbeux et chargé qu'Apache. Il utilise des techniques de mise en cache et d'optimisation pour à la fois réduire la charge du serveur, mais également améliorer les performances. Enfin, il est conçu pour être sécurisé en proposant un mode HTTPS par défaut, en plus d'autre outils d'authentification, d'autorisation, ou de chiffrement.

Docker Afin de gérer plus efficacement la solution, nous avons choisis d'utiliser **Docker**, qui est un outil permettant d'empaqueter une application et ses dépendances dans un conteneur isolé, qui peut être exécuté sur n'importe quel serveur. L'objectif est d'isoler les services afin de le exécuter de façon indépendante, uniquement liés entre-eux par des réseaux virtuels. Dans notre cas, c'est particulièrement utile pour deux raisons. Dans un premier temps, on dispose d'images Docker prêtent à l'emploi, ce qui simplifie le travail d'intégration et de déploiement continu (CI/CD) que l'on verra par la suite. La seconde raison, est qu'il est plus facile de travailler avec plusieurs environnements différents. Dans notre cas, nous en avons un de développement, un de pré-production, et un de production.

Ansible Toujours dans une optique de déploiement continu, nous avons choisis d'utiliser **Ansible**, qui nous permet d'automatiser le déploiement, la configuration et la gestion de notre infrastructure informatique. Il est surtout utile lorsque l'on dispose de plusieurs infrastructures différentes, comme c'est le cas pour nous avec la solution cloud. Encore une fois, il nous procure plusieurs avantages, notamment en terme de flexibilité où il est possible pour nous d'automatiser une grande variété de tâches, tout en nous laissant la possibilité de pouvoir le faire évoluer en cas de besoins changeants. De plus, il nous offre des fonctionnalités de sécurité nous permettant d'automatiser certaines tâches liées à sécuriser nos infrastructures.

GitLab Enfin, **GitLab** est une forge basé sur *Git*, qui nous sert de dépôt de code-source distant. Son but est de rassembler les projets et les développeurs, pour qu'ils puissent travailler ensemble sur un ou plusieurs projets commun. Nous retrouvons ainsi un système de gestion des versions, de suivi de tâches et de bugs, un wiki, etc. La particularité de GitLab est de très bien intégrer des outils de CI/CD, ce qui nous facilite la tâche pour les créer et les gérer par la suite.

Fonctionnement Nous pouvons résumer le fonctionnement du POC de la façon suivante : nous avons une seule et même instance WordPress, WooCommerce et MariaDB qui fonctionnent dans un conteneur Docker, lui même exécuté sur un serveur Caddy.
Lorsque nous apportons des modifications sur l'environnement de développement, comme des nouvelles fonctionnalités, des pipelines de CI/CD sont exécutées et font des opérations de tests, de validation, de nettoyage, etc., dans le but de fournir un environnement standardisé et « propre » contenant ces nouvelles fonctionnalités.

Une fois estimé que l'environnement de développement est assez stable et dispose des nouvelles fonctionnalités, on peut le basculer automatiquement sur l'environnement de pré-production en utilisant une nouvelle pipeline. Cette pipeline va exécuter de nouveaux tests avec cette fois-ci des données de production qui ont été anonymisées et traitées. Cet environnement nous permet d'avoir une bonne idée du fonctionnement de l'application, sans toutefois la rendre visible aux utilisateurs finaux.

Enfin, lorsque l'on estime que l'on peut peut publier l'application, on utilise une nouvelle pipeline qui va effectuer de nouvelles actions comme des tests de sécurité ou de performance avant de la publier.

Les déploiements entre environnements sont réalisés avec Ansible, qui automatise les tâches de configuration de nos infrastructures. Bien que similaires, il faut voir ces trois environnements comme distincts, en terme de fonctionnalités et de données.

Schéma de déploiement commenté

Pour nous permettre d'intégrer de nouvelles fonctionnalités et de les déployer sans impacter les utilisateurs, nous nous devons d'avoir plusieurs environnements distincts, appelés « branches ». Nous en avons quatre principales, plus une par fonctionnalité que nous souhaitons intégrer. Ainsi, imaginons que nous souhaitons développer puis intégrer une nouvelle fonctionnalité :

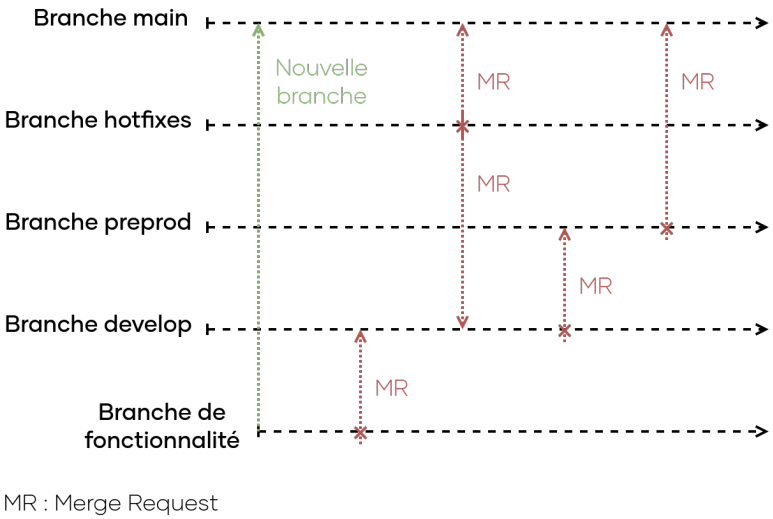

Figure 10: Schéma de vulgarisation du déploiement

Dans un premier temps, nous allons créer une nouvelle branche à partir de la branche principale de production, la branche « main ». C'est sur cette branche que nous allons développer notre fonctionnalité, laissant les autres intactes.
Une fois notre fonctionnalité prête, on peut demander à la fusionner avec la branche de développement, au travers d'un « Merge Request ». Cette étape sert à résoudre les conflits, et s'assurer que l'on a pas de régression.

Imaginons que l'on se rende compte d'un problème sur une fonctionnalité de production. Un correctif sera alors apporté depuis la branche « hotfixes » à la fois sur la branche « main », et aussi sur la branche « develop », encore une fois au travers d'une « Merge Request », qui va exécuter une pipeline comprenant des essais de non-régression, de qualité de code, etc.

Lorsque l'on estime qu'assez de fonctionnalités sont présentes sur la branche de développement, on exécute la pipeline permettant de la déplacer vers la branche de pré-production. Cette étape va tester notre fonctionnalité dans un environnement proche de celui de production, avec un jeu de données différent. Enfin, lorsque celle-ci est également suffisament stable, on exécute la pipeline permettant de la déplacer en production. Cette fois-ci, la fonctionnalité est confrontées à des données réelles et est visible des utilisateurs finaux.

Si jamais on rencontre un problème avec la nouvelle fonctionnalité, et qui serait passée au travers des mailles du fillet des essais, il est toujours possible d'apporter un correctif en faisant une copie de la branche de production sur la branche « hotfixes », avant de la redéployer sur la branche de production.

Mise en place d'une chaîne CI/CD

Comme nous l'avons abordé auparavant, suite à une demande de la direction, nous souhaitons passer d'un système de test, de validation et de déploiement manuel à un système d'intégration et de déploiement continue (CI/CD). Bien que similaires, les deux ont des objectifs différents que nous allons présenter par la suite.

CI - L'intégration continue

De l'anglais *Continuous Integration*, il faut le voir comme un ensemble de pratiques pour vérifier que les modifications de code-source que l'on fait n'amène pas de régression dans l'application. Dans notre cas, nous allons ajouter une brique logicielle automatisant certaines tâches, comme de la compilation, des tests unitaires et fonctionnels, des tests de performance, etc. À chaque changement dans le code-source, cette brique va s'exécuter et produire un ensemble de résultats. Cela nous permet de ne pas oublier d'éléments, de corriger des éventuelles erreurs, et par conséquent améliorer la qualité du code tout en réduisant la dette technique.

Pour pouvoir la mettre en place, il faut que notre code-source soit partagé (sur GitLab dans notre cas), et que des tests d'intégration soient développés pour valider l'application. Il nous faut également un outil d'intégration continue, les plus connus étant *Jenkins*, *Travis CI*, *GitLab CI* et *CruiseControl*. Dans notre cas, nous avons utilisé GitLab CI pour son intégration avec la forge GitLab.

Le principe est le suivant :

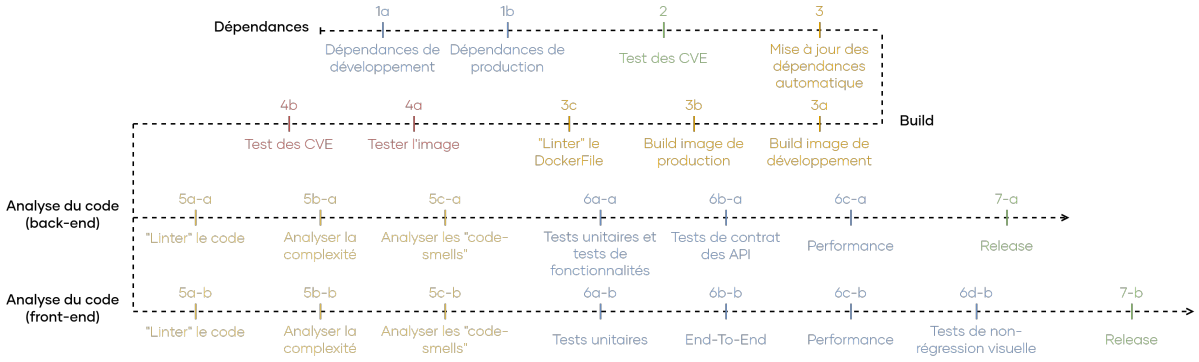

Figure 11: Schéma de principe d'une chaîne d'intégration continue

- **1a**/**1b** : Dans un premier temps, il nous faut récupérer les dépendances de nos outils (et les bonnes versions), comme par exemple *Composer* pour *PHP*. Cette étape est optimisée en détectant les changements sur les fichiers de configuration.

- **2** : Cette étape consiste à tester les *CVE*, les *Common Vulnerabilities and Exposures* sur notre code mais aussi les dépendances. Il faut le voir comme des séries d'essais pour détecter si notre code est concerné par des failles connues de sécurité.
- **3** : Enfin, les dépendances sont mises à jour, ou supprimées, au besoin.
- **3a**/**3b** : À cette étape, nous construisons les images *Docker* à partir des dépendances testées auparavant.
- **3c**/**4a**/**4b** : Une fois en possession de l'image, nous allons effectuer plusieurs essais dessus. Pour commencer, nous allons utiliser un *Linter*, un outil mettant en évidence les erreurs de syntax et de convention de code pour limiter les erreurs et la dette technique. Puis, le fonctionnement ainsi que les CVE de l'image sont testées.

Par la suite, nous avons un embranchement qui s'exécute en parallèle, un pour le *front-end*, l'autre pour le *back-end*.

- **5a-a** à **5c-b** : Ces étapes sont identiques mais sur des données différentes. On souhaite à nouveau « linter » le code avec des technologies comme *ESLint* pour le *front-end*, et *PHP-cs-Fixer* pour le *back-end*. À la suite de ça, la complexité du code est analysée, ainsi que les *code-smells*, qui sont des erreurs liées à des mauvaises pratiques qui peuvent créer des défauts. Cela est fait dans un but de sécuriser l'application, et de prévenir la dette technique.
- **6a-a**/**6a-b** : À cette étape, nous pouvons effectuer des tests unitaires, c'est-à-dire tester des éléments en particulier de notre application. Ces tests nous permettent de valider le bon fonctionnement et prévenir d'éventuels problèmes.
- **6b-a**/**6b-b** : Côté *back-end*, nous effectuons un *test de contrat d'API* pour vérifier que notre API répond bien à nos attentes, en utilisant un outil dédié comme *Newman*. Côté *front-end*, nous avons un test *End-To-End* (E2E), qui ressemble aux tests unitaires que nous avons effectués auparavant, mais cette fois-ci sur l'ensemble de l'application. Cela inclut des tests sur l'interface utilisateur (GUI), des tests d'intégration, de bases de données, de performance, etc. Plusieurs technologies existent pour cela, comme *Cypress* ou *Selenium*.
- **6c-a**/**6c-b** : Une fois tous les tests de fonctionnalité effectués et validés, nous pouvons effectuer un test de performance de l'application, en utilisant *K6* ou *Lighthouse*.
- **6d-b** : Uniquement sur la partie *front-end*, nous effectuons un test de régression visuelle avec *BackspotJS*, pour nous assurer que l'ajout ou la modification de fonctionnalité n'altère pas visuellement une autre fonctionnalité, et sous différents scénarii (pc, mobile, avec un lecteur d'écran, etc.).
- **7-a**/**7-b** : Une fois que toutes les étapes ont été validées, nous pouvons publier l'image qui a été générée auparavant. Avec cette étape, une nouvelle version est crée et incrémentée en fonction des changements. De plus, un changelog est produit afin de permettre à tout le monde de suivre les changements associés à cette version.

CD - Le déploiement continu

Toujours de l'anglais *Continuous Deployment*, est une pratique de développement logiciel dans laquelle le logiciel est généré de manière à pouvoir être mis en production à tout moment. On va différencier *Distribution Continue* de *Déploiement Continu*. Pour ce faire, un modèle de distribution continue implique des environnements de test similaires à ceux de la production. Les nouvelles générations réalisées dans le cadre d'une solution de distribution continue sont automatiquement déployées dans un environnement de test automatique d'assurance qualité qui recherche les erreurs et les incohérences. Une fois que le code a réussi tous les tests, la distribution continue nécessite une intervention humaine pour approuver le déploiement en production. Le déploiement lui-même est ensuite exécuté par l'automatisation.

Le déploiement continu fait aller un peu plus loin l'automatisation et supprime l'intervention manuelle. Les tests et les développeurs sont considérés comme suffisamment fiables pour qu'aucune approbation de la mise en production ne soit pas nécessaire. Si les tests aboutissent, le nouveau code est considéré comme approuvé, et le déploiement en production a lieu.

Le déploiement continu est le résultat naturel d'une distribution continue efficace. Finalement, l'approbation manuelle apporte peu ou pas de valeur et ne fait que ralentir les choses. À ce stade, elle est supprimée, et la distribution continue devient un déploiement continu.

Il existe plusieurs outils permettant de faire du déploiement continu, le plus populaire étant probablement **Kubernetes**. Ci-dessous, nous pouvons retrouver un schéma de principe d'une pipeline CD :

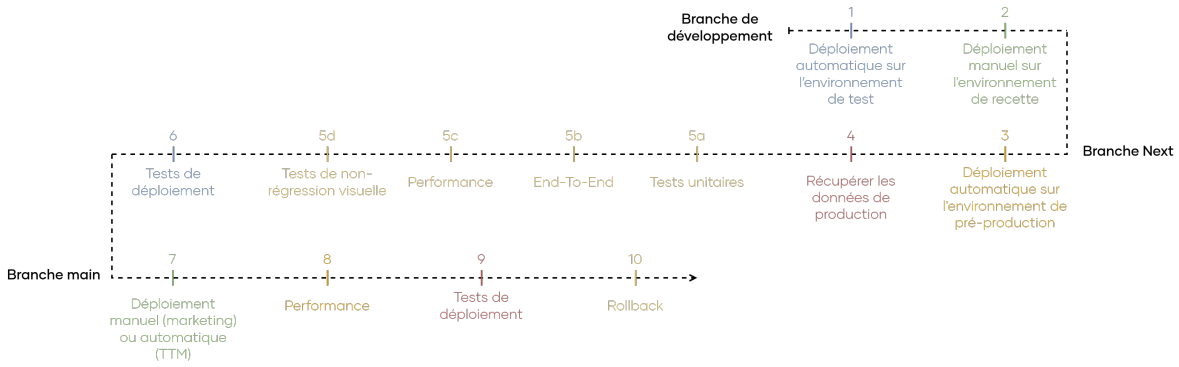

Figure 12: Schéma de principe d'une chaïne de déploiement continu

Nous allons reprendre l'exemple d'une nouvelle fonctionnalité qui a été développée sur l'environnement de développement, et qui a été intégrée.

- **1** : Le déploiement est réalisé automatiquement à l'aide d'*Ansible* sur l'environnement de test. Il est également possible d'utiliser *Autok8s* qui facilite et automatise l'utilisation de *Kubernetes*.

- **2** : De façon similaire, le déploiement est réalisé manuellement sur l'environnement de recette.
- **3** : Une fois réalisé, le déploiement est automatiquement réalisé sur l'environnement de pré-production, toujours à l'aide d'*Ansible*.
- **4** : Il s'agit maintenant de récupérer les données de production en vue de tester la solution à déployer. Cependant, dans le respect du *RGPD*[10], il est nécessaire de les anonymiser ou à minima de les pseudonymiser.
- **5a**/**5b**/**5c**/**5d** : À l'instar de la pipeline *CI*, on souhaite procéder à des tests sur la solution à déployer, à commencer par des tests unitaires réalisés avec des outils comme *JEST* ou *Vitest*. Une fois les essais unitaires complétés, on effectue des essais sur l'ensemble de la solution (E2E) grâce à *Cypress*, avant de procéder à des tests de performance à l'aide de *K6* ou encore *Lighthouse*. Enfin, nous retrouvons quelques tests de non régression visuelle, grâce à *BackspotJS*.
- **6** : Ultime action sur l'environnement de pré-production, on effectue des essais de déploiement sur l'environnement de production, en vérifiant notamment les routes critiques de l'*API*.
- **7** : Maintenant que notre solution à passé avec succès les essais de pré-production, il est temps de la déployer en production. Pour cela, soit on l'effectue automatiquement pour réduire le *time-to-market* et proposer les nouvelles fonctionnalités ou correction au plus vite, soit on peut déployer manuellement, ce qui est pratique pour des raisons marketing. Par exemple, X fonctionnalité sera disponible à telle date.
- **8**/**9** : Le déploiement complété, on réalise des essais de performance sur l'environnement de production, toujours avec *K6* ou *Lighthouse*. On en profite pour réaliser des tests suite au déploiement en vérifiant les routes et parcours critique de la solution et de son *API*.
- **10** : Cette étape est optionnelle et on souhaite l'éviter. Néanmoins si l'on découvre des problèmes sur la solution suite à son déploiement en production, il faut pouvoir restaurer une version antérieur de la solution.

Ces étapes de chaîne de déploiement continu permettent de fournir de façon automatisée et fréquente de nouvelles versions de la solution, tout en effectuant des essais pour perturber le moins possible la production, avec la possibilité de revenir en arrière si besoin.

[10] RGPD - Réglementation Générale sur la Protection des Données

Intégration sur le POC

> Dans le cadre du brouillon, cette section n'est pas complétée. Ce que nous souhaitons faire, suite à la présentation de la chaîne d'intégration continue et de déploiement continu, est de détailler la façon dont nous l'avons implémentée, en incluant la stack technique et justifiant son utilisation sur la façon dont il nous permet de répondre à notre besoin.

Pour les différentes étapes vues précédemment, voici les outils que nous allons utiliser :

- *Lintage* de code : **Hadolint**
- Test des vulnérabilitées des dépendances : **DependencyCheck**
- Tests de sécurité et pentests : **Zaproxy**
- Tests concernant la qualité du code : **Sonarqube**
- Tests de performance : **K6**

Passage du POC à l'application finale

La direction nous a demandé de développer une application de e-commerce permettant la vente en ligne, afin de permettre à l'entreprise de diversifier ses créneaux de vente.
Afin de valider que les choix techniques répondent au besoin, il a été entrepris de réaliser un POC reprenant les fonctionnalités principales de l'application finale.

Le POC étant maintenant réalisé, il est temps de chercher à déployer l'application finale à partir du POC. Cependant, le POC et notre application finale ont des finalités différentes : un POC est destiné à être utilisé par un faible nombre d'utilisateurs, avec un faible nombre d'interactions. On sait que ce dernier ne va pas avoir un trafic qui évoluer dans le temps, les risques de sécurité sont limités, et les besoins spécifiques des utilisateurs restreints.
Ainsi, si nous devions penser dès à présent à adapter notre POC pour qu'il réponde à des besoins différents de ceux pour lesquels il a été conçu, on risque de se retrouver avec une solution inadaptée aux besoins de la direction.

Le POC a été conçu avec la finalité de l'application finale dès le début. C'est-à-dire que celui-ci intègre des stratégies et outils pour que l'application finale soit évolutive et résiliente dans le temps. De plus, le terme « application finale » est un peu trompeur, puisque l'application ne sera pas finie lorsqu'elle sera publiée. Une application dispose, comme tout produit, d'un cycle de vie dont son développement et lancement ne représente qu'une partie. Sa maintenance, son évolution et à terme, son décommissionnement, en représente également une autre partie. Dans ce sens, elle est pensée dès le début pour être mise à jour et faire des changements rapide pour nous adapter à de nouveaux besoins et exigences.

Afin d'anticiper ce passage du POC à l'application finale, nous avons établis les principaux risques que l'on pourrait rencontrer, en les évaluant, et en y apportant une solution préventive et corrective, ce qui nous permet de les anticiper. Ainsi, nous en retrouvons six :

ID	Nature du risque	Description	Gravité	Probabilité	Criticité	Conséquences si avéré	Solution préventive	Solution corrective
R1	Infrastructure	L'application est incapable de gérer un trafic important.	3	1	3	Utilisation perturbée voire impossible.	Penser le POC et son infrastructure dès le départ pour qu'ils puissent s'adapter à une hausse du trafic, comme une architecture distribuée ou cloud.	Migration de l'infrastucture vers une infrastructure capable de supporter la charge.
R2	Infrastructure	Un module de l'application tombe en panne.	3	2	6	Utilisation perturbée voire impossible.	Prévoir dans l'infrastructure des redondances pour les services critique, avec des possibilité de détection de panne et de basculement ni nécessaire.	Bascule sur un service de secours de redondance, et analyse des causes profondes ayant causé la panne.
R3	Infrastructure	Suite à une panne, l'application est indisponible	4	2	8	Utilisation perturbée voire impossible.	Prévoir des solutions de haute disponibilité 24h/24 et 7j/7. Utilisation de services de surveillance et de récupération d'urgence.	Bascule sur un service de secours de redondance, et analyse des causes profondes ayant causé la panne.
R4	Sécurité	L'application n'est pas sécurisée et est propice aux cyberattaques	3	3	9	Utilisation perturbée Pertes financières Réputation	Utilisation d'outils de détection de failles de sécurité et de détection d'attaque	Suspendre certains services de l'application le temps de combler la faille de sécurité ou de trouver une solution de contournement.
R5	Expérience utilisateur	L'application présente des soucis à l'utilisation (performance) ou à la navigation (expérience utilisateur)	1	3	3	Réputation Pertes financières	Utilisation d'outils de contrôle unitaires, de performance, et de pratiques d'expérience utilisateur (UX).	Publier un correctif en cas de problème de performance, modifier le design et des fonctionnalités.
R6	Conformité	L'application n'est pas conformes aux réglementations en vigueur.	3	2	6	Réputation Juridique	Appel à un DPO pour valider la conformité de l'application.	Publier un correctif pour se conformer à la réglementation.

Figure 13: Matrice des risques du passage du POC à l'application finale.

Voici le détail des risques. Nous en retrouvons trois liés à l'infrastructure de l'application finale :

- **R1** : Comme évoqué, le POC et l'application finale ont des finalités différentes. Là où le POC à affaire à un nombre limité d'utilisateurs et d'interactions, l'application finale doit pouvoir se mettre à l'échelle, s'adapter pour faire face à un éventuel trafic important, sans gêner la navigation pour les utilisateurs.
- **R2** : De plus, un POC est généralement conçu pour être résilient aux pannes de composants individuels, là où l'application doit être capable de résister à des pannes plus importantes, comme un fournisseur cloud par exemple. Pour cela, on peut faire appel à des techniques de redondance et de basculement pour anticiper un tel problème.
- **R3** : Enfin, l'application finale se doit d'être disponible et assurer une haute-disponibilité 24h/24 et 7j/7, même en cas de panne d'un service. Pour cela, il est intéressant d'utiliser des techniques de surveillance de l'état de santé des services, et de récupération d'urgence. Avec le risque R2, on peut ainsi mettre en place une infrastructure nous permettant d'être disponible à tout instant.

En plus de ces risques liés à l'infrastructure, nous déterminons des risques plus fonctionnels :

- **R4** : Le POC se veut avoir un faible nombre d'interactions, là où l'application finale serait exposée sur internet et avec un volume variable d'utilisateurs. Étant un site hébergeant des données clients et traitant avec des données financières, elle est une cible de choix pour les cybermenances. Il est important d'utiliser des outils nous permettant de détecter d'éventuelles vulnérabilitées pour pouvoir les anticiper. Il faut également avoir des protections contre les cyberattaques comme celles par déni de service, afin de garantir la sécurité des données.
- **R5** : Bien que celui-ci soit moins critique si avéré, nous sommes sur un nouveau créneau pour l'entreprise. Il est donc important de proposer une expérience agréable aux utilisateurs pour développer l'activité e-commerce de Breizhsport. Nous souhaitons donc nous assurer que toutes les fonctionnalités fonctionnent comme prévu, et que les performances de l'application permettent aux clients d'effectuer leurs achats sans techniques.
- **R6** : Enfin, puisque nous sommes une entreprise française et destinée à un public français, nous devons nous conformer au *RGPD*, Réglementation Générale pour la Protection des Données. Pour nous aider, nous pouvons faire appel à un *DPO*[11] qui peut nous aider dans cette tâche, ainsi que la *CNIL*[12].

Ces risques, avec leurs solutions de prévention et de correction, nous permettent d'anticiper et de prévenir certains problèmes que nous pourrions rencontrer lors du passage à l'application finale.

[11] DPO - Data Protection Officer - Délégué à la Protection des Données
[12] CNIL - Commission Nationale de l'Informatique et des Libertés

Démarches d'apprentissage et de montée en compétences

Pour faire face à la charge de travail liée au développement de l'application de vente en ligne demandé par la direction, l'équipe va s'agrandir pour accueillir de nouveaux développeurs, dont des développeurs juniors.
Pour permettre à l'ensemble des développeurs d'avoir des compétences similaires leurs permettant de travailler dans les meilleures conditions possibles, nous souhaitons mettre des mesures qui s'appuient sur le socle suivant :

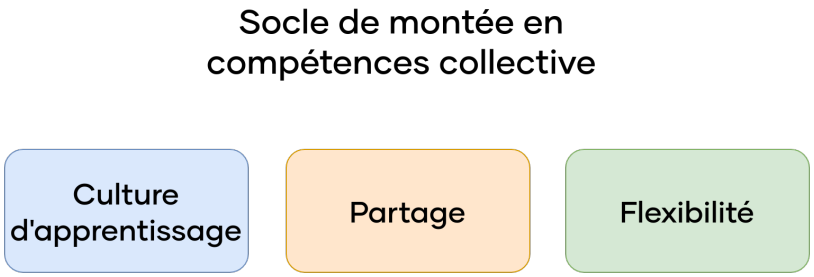

Figure 14: Pilliers du socle de montée en compétences collective

Nous souhaitons développer une culture d'apprentissage au sein de l'équipe. Il faut que l'apprentissage et le partage de connaissances soient valorisés. Cela se traduit de plusieurs façons que nous verrons par la suite. De plus, il est important de rester flexible, aussi bien pour les développeurs qui ont tous des courbes d'apprentissage différentes, que pour nous. Les besoins de montée en compétences évolueront au fil du temps, il est important d'adapter nos démarches à ces évolutions.

Concrètement, nous souhaitons que cela se traduise de la façon suivante :

- Chaque membre de l'équipe, qu'il soit nouveau ou non, participera à un **onboarding** le plus clair et le plus complet possible. À cette occasion, il sera présenté le projet, l'outillage utilisé, les processus et les réglementations en vigueur. Un rappel des bonnes pratiques peut également être établi, ce qui nous permettra de limiter un peu plus notre dette technique au fil du temps.
- À intervals réguliers, des **sessions de partage** seront organisées. Au début de ce rapport, nous évoquions des séances de partage afin de faciliter le maintien en compétences, qui a le même but, à savoir le transfert de connaissances entre membres de l'équipes. Nous pensons donc organiser un créneau commun pour tous les membres de l'équipe (ancien comme nouveau), ce qui favoriserait la communication et la cohésion des équipes, tout en créeant une culture d'apprentissage. Ces réunions pourraient être consacrées à des sujets techniques, des retours d'expérience, ou encore des discussions libres.

- Une autre piste, serait l'introduction de **tech talks**. Le format serait un peu similaire à une conférence, où des membres de l'équipe comme des partenaires et experts externes peuvent venir parler d'un sujet qui les concerne, et qui pourrait être utile à l'ensemble de l'équipe. Cela permettrait à certaines personnes d'être au courant des dernières nouveautés techniques, et de découvrir de nouveaux sujets.
- Nous souhaitons organiser des sessions de **pair programming**, c'est-à-dire des séances de programmation en binôme, qui est l'occasion d'aprendre les uns des autres et de résoudre des problèmes de façon commune. De façon idéale, nous aimerions associer des profils assez différents.
- Nous aimerions favoriser l'**auto-apprentissage**, en encourageant les développeurs à apprendre de manière autonome en lisant des livres, des articles ou des blogs, en suivant des formations ou en participant à des communautés en ligne.
- Enfin, nous aimerions organiser un programme de **cooptation** et de **mentorat**, c'est-à-dire recruter en externe un candidat qui a été recommandé par une personne en interne, non pas pour son lien de parenté, mais pour ses compétences. La personne l'ayant recommandé deviendrait son mentor pendant quelques temps, et aurait en partie à sa charge la montée en compétence du développeur, contre une rémunération ou des avantages supplémentaires.

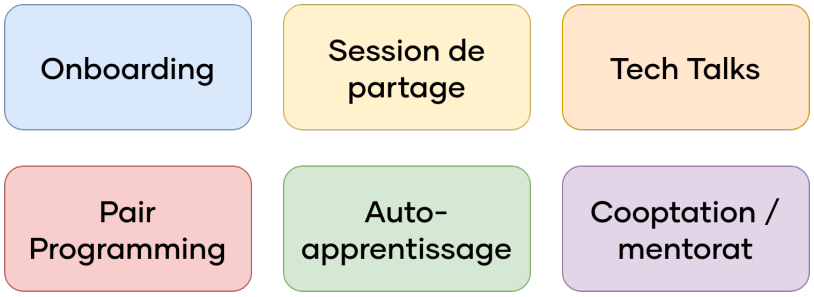

Figure 15: Nuage d'idées permettant une démarche d'apprentissage collective

Nous pensons que ces idées permettront un apprentissage collectif assez efficace, sans être trop contraignant pour les développeurs. Encore une fois, ces idées seront à adapter à chacun et au fil du temps, afin qu'elle reste efficace. Elle sera donc revue régulièrement pour y apporter des nouveautés, ou retirer des idées qui ne fonctionnent pas.

Assurance qualité

La direction nous a fourni le besoin suivant : créer une application web permettant la vente en ligne pour *Breizhsport*. Pour réaliser celui-ci, nous avons entrepris des étapes qui ont été détaillées plus tôt dans ce rapport. Cependant, il nous est nécessaire de formaliser les exigences attendues en terme de qualité, pour justifier que la solution réponde bien au besoin. Pour cela, on va utiliser un *Plan d'Assurance Qualité* (PAQ), qui est un document commun entre client et fournisseur, afin de s'accorder sur le contexte, le périmètre, les enjeux, et permettant de traduire les attentes et envies du client en besoin fonctionnel. Normalement réalisé et validé en début de projet, il est nécessaire de le refaire valider en cas d'évolution du besoin ou de dérogation qui sort du cadre de ce qui a été défini dans ce document.

Ce *PAQ* reprend divers points de ce rapport. Nous l'avons volontairement allégé afin de ne pas surcharger ce rapport, tout en synthétisant les idées. Pour l'instant nous nous concentrons sur les aspects fonctionnels, les détails techniques des tests seront établis par la suite.

Objectifs

Ce *PAQ* a trois objectifs :

- **Garantir que l'application réponde aux exigences spécifiées par la direction.**
- **Identifier et corriger les défauts et les erreurs avant la mise en production.**
- **Améliorer la qualité de l'application au fil du temps.**

Ces objectifs permettent de traduire la demande de la direction de développement d'une application de e-commerce en besoin. Ainsi, avec ce *PAQ*, nous définissons le niveau de qualité attendu afin d'encadrer le développement et la maintenance de l'application dans le temps, pour qu'elle réponde toujours au besoin après l'ajout et la modification de fonctionnalités. Bien évidemment, il sera nécessaire de régulièrement revoir ce PAQ afin de l'adapter à une évolution du besoin.

Portée

En plus des objectifs, il est nécessaire de définir les thématiques sur lesquelles le *PAQ* s'applique. À ce stade, nous sommes encore aux premières étapes de développement de l'application. Ainsi, notre PAQ se limite pour l'instant aux éléments suivants :

- **Chaîne de déploiement et d'intégration continue**
- **Infrastructure**
- **Fonctionnalités de sécurité**

- **Extensions associées à notre CMS (Wordpress)**
- **Organisation des équipes**
- **Méthodologie de travail**

D'autres aspects à venir, comme la navigation sur le site, le module de paiement et les produits proposés seront rajoutés par la suite, d'où la nécessité de revoir régulièrement ce document. De plus, nous allons aborder la sécurité d'un point de vue assez macro en prenant l'application dans son ensemble. Par la suite, on envisagerait de produire un *Plan d'Assurance Sécurité* (PAS) pour décrire les moyens garantissant la sécurité en détail.

Responsabilités

En termes de responsabilités des parties prenantes, nous pouvons le définir comme tel :

Figure 16: Schéma des parties prenantes de notre PAQ

Bien qu'avec avec une méthode de travail *AGILE*, comme *SCRUM*, la notion de chef de projet n'existe pas, nous définissons un responsable de la coordination des activités que nous appelerons **Service Owner**.
En plus de cela, nous avons deux équipes distinctes : l'**équipe de développement**, qui développe et intègre les fonctionnalités afin de les faire évoluer ou les maintenir ; l'**équipe de test**, qui développe et effectue les tests visant à garantir la qualité et la sécurité de notre application.

Si l'on souhaitait rentrer plus en détail, on pourrait attribuer un rôle des activités à chaque partie prenante, notamment au travers d'une matrice *RACI*. Cependant, ce document se veut fonctionnel et non technique, ainsi nous en restons-là pour le moment.

Méthodologie

Les équipes projet précédemment présentées sont amenées à travailler en mode *AGILE*, au-travers de la méthode de travail *SCRUM*. Cette méthode de travail est particulièrement adaptée à des projets de développement informatique, car elle offre une grande flexibilité pour faire face à d'éventuels besoins changeants pour notre application. De plus, les sprints sont cadencés de façon régulière, ce qui nous permet de définir aisément un planning pour vérifier et mettre à jour ce document pour l'adapter à un nouveau besoin.

Activités d'assurance qualité

Dans le cadre du développement de l'application web permettant la vente en ligne de *Breizhsport*, nous souhaitons mettre en place les activités suivantes afin de garantir la qualité de la solution pour qu'elle répondre au besoin de l'entreprise.

- **Vérification des exigences** : nous verifions les exigences imposées pour l'application, afin de vérifier qu'elles sont complètes, cohérentes, et correctes.
- **Vérification de la conception** : nous souhaitons également vérifier que la conception de l'application, ses choix architecturaux et sa stack technique puisse correspondre aux exigences.
- **Vérification du code** : pour s'assurer que le code de l'application ne comporte pas d'erreur, et respecte des standards et conventions de code.
- **Tests fonctionnels** : pour vérifier et s'assurer que l'application fonctionne comme demandé.
- **Tests non fonctionnels** : nous souhaitons vérifier que des aspects de performance, de sécurité ou encore de compatibilité fonctionnent comme demandé.
- **Tests d'acceptation** : concerne tous les tests effectués dans le cadre de la recette de l'application, qui permet de valider ou non la livraison de la solution à la direction.

Des détails des tests, et de certaines vérification comme le code, les outils utilisés, sont décrits par la suite dans ce rapport.

Outils et ressources

Pour nous accompagner afin de garantir la qualité de l'application, nous souhaitons mettre en place des outils.
En particulier, nous souhaitons utiliser **JIRA**, qui est un outil permettant de faire plusieurs choses et favorise la collaboration; il nous permet de gérer notre projet sous une méthodologie *SCRUM*, en nous proposant de quoi des tâches, des modules et des user-stories, et de les intégrer lors d'un sprint. De plus, il nous permet de réaliser un suivi des bugs et des tests de façon assez complète, tout en s'intégrant bien avec notre projet existant.

Nous avons d'autres outils en lien avec les tests, que nous détaillerons par la suite de ce rapport.

Planification

Comme évoqué précédemment, pour pouvoir garantir la qualité de l'application dans le temps, il est nécessaire pour nous de revoir ce document régulièrement pour mettre à jour les tests, les exigences, ou encore les ressources associées. Puisque nous sommes encore à la phase de développement de notre application, celui-ci ne sera pas amené à changer avant la livraison de notre premier livrable, sauf si la direction souhaite ré-évaluer son besoin ou ses exigences.

Cependant, par la suite nous souhaitons revoir celui-ci de façon à minima trimestrielle, et ce même si le besoin ou les exigences n'ont pas ou peu évolués. Puisque nous sommes sur un format *AGILE* avec des sprints, nous pouvons tout à fait envisager de l'inclure dans notre *sprint backlog* tous les cinq ou dix sprints, en fonction de la durée de nos sprints.
L'idéal serait de le faire à chaque jalon avec le client, la direction de *Breizhsport* dans notre cas. Par exemple, notre premier jalon étant le *09 février 2024*, nous souhaitons compléter et mettre à jour ce *PAQ* suite à ce jalon, avec validation du client.

Suivi et amélioration

Ce document est amené à vivre en parallèle de notre application. Tous les résultats des activités d'assurance qualité sont collectés et analysés, afin que d'éventuelles évolutions ou changements puissent être apportés à ce document.

Plan de communication

Enfin, nous souhaitons cadrer la communication de notre application auprès de notre client. Puisque nous avons un client interne, avec des jalons réguliers qui sont imposés, nous estimons ne pas avoir besoin d'effectuer de communication avec la direction en dehors de ces jalons. Une fois l'application livrée et avec des utilisateurs finaux, nous souhaiterons mettre à jour ce document, afin de prendre en compte les éventuelles pannes et opérations de maintenance à communiquer auprès des utilisateurs.

Concernant la communication entre les équipes, nous organisons déjà des sessions de partage de connaissances au sein de l'équipe projet. De plus, **JIRA** que nous souhaitons utiliser, permet une communication claire sur des sujets précis entre différents acteurs de nos équipes.

Dette technique

Un élément à prendre en compte dès à présent est ce que l'on appelle la dette technique :

> La dette technique est l'accumulation d'erreurs et de défauts sur une application, quand la rapidité d'un projet prime sur la qualité et demande ainsi un travail de correction supplémentaire par la suite.

Ainsi, nous souhaitons mettre en place dès à présent une stratégie de gestion de la dette technique, ce qui nous permettra de gagner du temps et par conséquent, réduire les coûts, par la suite.

Stratégie

Il existe plusieurs stratégies de gestion de la dette techniques. Celles-ci vont dépendre de la taille et la complexité de l'application, des ressources disponibles, des objectifs, etc.

Soit on adopte une stratégie :

- **Proactive** : dès que l'on identifie une erreur qui causerait de la dette technique, on va chercher à la corriger, ce qui peut se faire au détriment des objectifs et peut engendrer des délais, mais assure la qualité de l'application.
- **Réactive** : on s'occupe d'identifier et de corriger les erreurs liées à la dette technique uniquement si elles posent problème. Cela favorise les temps de développement, mais peut engendrer des problèmes à long-terme qui peuvent être coûteux.

Chaque stratégie a des avantages et des inconvénients, qui vont dépendre du souhait de la direction. Cependant, on peut imaginer adopter une solution hybride. Dès que possible, on va chercher à identifier et corriger la dette technique, en particulier pour les problèmes les plus importants. En revanche, pour ce qui pourrait poser des problèmes moins graves, on peut les laisser de côté pour le moment et jusqu'à ce qu'ils causent des problèmes.

Pour cette approche, nous proposons en plus quelques mesures à intégrer dans notre stratégie :

- **Intégrer la gestion de la dette technique dans nos processus de développement** : on souhaite utiliser des outils permettant de détecter et de corriger la dette technique dès le départ, comme des *linters*, des scanners de code et des outils de gestion des dépendances. Nous allons développer cette partie par la suite.
- **Planifier des audits de code réguliers** : à chaque interval de x sprints, nous mettons en place des actions visant à identifier et corriger les éléments de dette technique susceptibles de créer des problèmes majeurs par la suite.

- **Mettre en place un processus de réécriture du code** : bien que ce ne soit pas idéal, il est probable que nous ayons à réécrire complètement certaines parties de notre application. Il nous faut donc mettre dès à présent un processus qui permet de gérer cette étape, tout en garantissant qu'il n'y a pas de régression ni de nouvelle dette technique.
- **Former les développeurs aux bonnes pratiques de développement** : que ce soit sur des principes de conception, de programmation et de documentation, cela nous permettrait de réduire dès le départ notre dette technique.

Comme nous l'avons vu auparavant dans ce rapport, nous souhaitons mettre en place des sessions pour permettre aux collaborateurs d'échanger des connaissances. En favorisant la communication et la formation, nous espérons ainsi que l'ensemble des équipes soient formés dès le départ afin de réduire la dette technique, et ainsi favoriser la qualité de notre application.

Ce *PAQ*, nous permet d'établir les démarches visant à la qualité de notre application. Il sera amené à évoluer dans le temps, afin de refléter les besoins et exigences du moment. Lors de ces évolutions, dans une démarche d'amélioration continue, nous intégrerons également des éléments du retour d'expérience qu'il nous aura permis d'avoir pour pouvoir l'adapter un peu plus à notre équipe, notre mode de fonctionnement et nos outils.

Politique de tests

Un projet, que ce soit dans le monde matériel ou logiciel, comporte des risques plus ou moins importants et faciles à identifier. Ces risques sont liés au fait de produire une valeur que l'on voudra éviter de perdre du fait de « régressions » et d'un mauvais fonctionnement lors des différentes évolutions. Lors du développement d'un logiciel, plus que dans tout autre domaine, les régressions, bugs, ou effets indésirables sont un impondérable lors de l'ajout de fonctionnalités ou de cas d'utilisation. Pour compléter le *PAQ* il faut garantir d'une part que la valeur à livrer a été correctement produite et qu'elle sera maintenue sur la durée. La politique de tests a pour rôle de répondre à la question du « pourquoi teste-t-on ? » et « dans quel but, pour aller dans quelle direction ? » Cette dernière permettra aux équipes de s'appuyer dessus pour élaborer les plans de tests et concevoir des tests qui minimisent les risques identifiés, tout en adaptant le produit au rythme dicté par le marché ou l'organisation. Nous décidons de nous appuyer sur la pyramide des tests suivante dans l'ordre indiqué :

- Tests unitaires
- Tests d'intégration
- Tests de sécurité
- Test de non régressions
- Tests d'API
- Tests End to End
- Tests Manuels

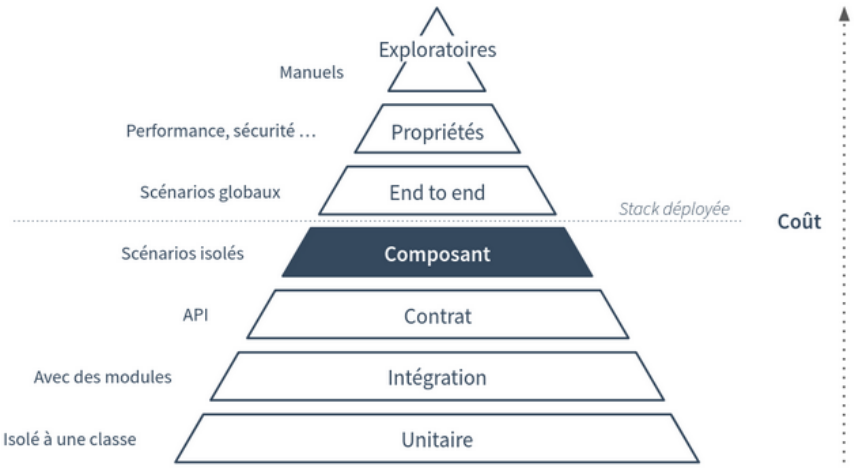

Figure 17: Pyramide des tests

En mode *Agile* et *DevOps*, ce sont les développeurs qui rédigent et exécutent les tests, pas les équipes d'assurance qualité. Consulter leur expertise peut permettre aux développeurs de rédiger de meilleurs tests qualité/sécurité. L'équipe qualité peut aussi aider à maintenir un certain cadre de tests, corriger les tests irréguliers.

Nous nous appuierons sur une **stratégie de test** qui est le **document de haut niveau** définissant, pour un programme, les niveaux de tests à exécuter et les tests dans chacun de ces niveaux (pour un ou plusieurs projets).
Il s'agit donc d'un document fixant les tests répartis par niveau dans la pyramide des tests, le *RACI* pour chacun des niveaux et les attendus en termes d'objectifs pour chacun des types de tests mis en œuvre.
C'est à partir de ce document que les équipes de développement du produit vont construire les plans de tests, ou simplement implémenter les tests internes au produit (tests unitaires et d'intégration par exemple). La stratégie de test porte réellement cette répartition des responsabilités.
Elle identifie pour chaque niveau de tests les éléments suivants :

- Le type de test
- Les limites de ce que ce niveau teste
- Qui implémente les tests du niveau et quand
- Le non testé et les risques associés (et assumés) qui est essentiel et sera demandé lors d'un audit ISO 9001 au titre de la maîtrise des risques, notamment si une partie du produit est fournie par un prestataire et doit faire l'objet de contrôles qualité.

La stratégie de test est un document collaboratif entre les différentes équipes et acteurs d'un développement. Elle regroupe les informations sur ce qui sera testé ou non, met en valeur la limite de risque communément acceptée.
De ce fait, c'est un document construit durant les phases de conception du produit dans sa globalité, de préférence de manière itérative pour faciliter l'émergence d'un compromis et une bonne correspondance avec l'évolution de la connaissance produit.

Chaque contributeur doit donc durant ces phases de découverte :

- Déterminer les niveaux de test qui les incombent.
- Déterminer les risques détectables via ces niveaux de tests.
- Déterminer si il est nécessaire de faire évoluer la conception pour réduire les facteurs de risque (probabilité, gravité, non-détectabilité) en adaptant la conception pour la rendre plus testable ?
- Déterminer le but visé par les tests (Détecter un défaut au plus tôt, valider la bonne implémentation d'un parcours client, détecter une dérive d'une variable de suivi, guider la conception, etc.)

- Déterminer les autres acteurs en interface avec les tests dont ils ont la responsabilité et déterminer les informations à échanger avec eux pour faciliter le codéveloppement de la stratégie et plus tard des tests.

Un des éléments importants dans la qualité d'un projet est la connaissance de la responsabilité de chaque acteur dans les étapes des différents processus conduisant à l'ajout de valeur au produit. Le *RACI* permettra l'attribution des pilotes aux différents niveaux de test. Nous utiliserons le *shift-left* aussi appelé décalage à gauche, il signifie que plus un défaut est détecté plus tôt, plus sa correction est facile à effectuer, et plus le coût de correction deviendra faible.
Enfin, nous fixerons des règles de sélection des tests en fonction de leur proximité avec le système et leur capacité à être mis en place tôt dans le processus de développement.

Tests unitaires

Les tests unitaires doivent être isolés et être exécutables sur n'importe quelle machine, dans n'importe quel ordre, sans affecter les uns les autres. Si possible, les tests ne doivent pas dépendre des facteurs environnementaux ou de l'état global / externe. Les tests qui ont ces dépendances sont plus difficiles à exécuter et généralement instables, ce qui les rend plus difficiles à déboguer et à corriger, et finit par coûter plus de temps qu'ils n'en économisent. Les tests unitaires doivent aussi être **automatisés** et constitueront la majeure partie de nos tests. Cela peut être quotidien, ou toutes les heures, ou dans le cadre d'un processus d'intégration ou de livraison continue. Les rapports devront être accessibles et examinés par tous les membres de l'équipe. Les équipes devront étudier les métriques comme : la couverture du code, couverture du code modifié, nombre de tests en cours d'exécution, performances, etc.

Couverture du code

Un outil de couverture de code utilisera un ou plusieurs critères pour déterminer comment notre code a été mis à l'épreuve ou non pendant l'exécution de notre suite de tests. Dans notre cas, nous utiliserons **SonarQube** comme outil de couverture du code.Les métriques courantes qui seront mentionnées dans nos rapports de couverture sont les suivantes :

- **Couverture des fonctions** : nombre de fonctions définies ayant été appelées.
- **Couverture des instructions** : combien d'instructions du programme ont été exécutées.
- **Couverture des branches** : combien de branches des structures de contrôle (les instructions « If » par exemple) ont été exécutées.
- **Couverture des conditions** : combien de sous-expressions booléennes ont été testées pour une valeur vraie et fausse (true/false).
- **Couverture des lignes** : combien de lignes du code source ont été testées.

Il ne serait pas raisonnable de fixer un seuil d'échec trop élevé, et une couverture de 90 % est susceptible de faire échouer nos build dans de nombreux cas. Si notre objectif est une couverture de 80 %, nous devrions envisager un seuil d'échec à 70 % comme filet de sécurité pour notre culture CI. Il faut que notre équipe comprenne comment l'application est censée se comporter lorsqu'une personne l'utilise correctement, mais aussi lorsque quelqu'un essaie de l'endommager. Les outils de couverture de code peuvent nous permettre de comprendre où notre attention doit ensuite se porter, mais ils ne nous diront pas si nos tests existants sont suffisamment robustes pour les comportements inattendus.

Couverture fonctionnelle

Notre *Wordpress* fournit des plugins tel que *Woocommerce* permettant de proposer un service de vente sur le site web. Le framework **Timber** permet de modifier le site web de manière simplifié et donc ajoute une réactivité accrue lors de corrections. Le but est d'anticiper sur les besoins futurs, et donc de choisir un outil évolutif.

Tests d'intégration

Le fait de mettre en place une chaine d'intégration continue permet de redéployer les composants et dépendances associés au site web ce qui facilite la détection de problèmes de compatibilités suite à une mise à jour ou modification du site web. *GitLab CI* fait partie de *GitLab*. Il s'agit d'une application web avec une *API* qui stocke son état dans une base de données. Le principe consiste à rendre systématique et de façon plus ou moins automatique l'intégration des différents composants d'un système dès que ses composants sont modifiés, pour faire en sorte que les effets produits par ces modifications soient rapidement mesurables. Ceci permet notamment d'éviter les gros bugs difficiles à déceler, et favorise l'agilité dans les projets. Des tests de dépendances sont inclus dans ces tests avec **OWASP**.

Tests de non régression

Enfin, ces tests permettent de valider que la mise en ligne d'une nouvelle fonctionnalité sur un logiciel n'impactera pas les fonctions déjà existantes. Les tests fonctionnels auront bien validé que la nouvelle fonction est opérationnelle mais c'est les tests de non régression qui valideront que cette dernière n'impacte pas les autres fonctionnalités du logiciel. Les tests de non régression sont déployés sur un environnement de recette et doivent vérifier au minimum que les fonctionnalités principales ou « critiques » du logiciel sont toujours disponibles après la livraison de nouveaux développements.

Tests de sécurité

Les tests de sécurité des logiciels sont le processus d'évaluation et de test d'un système afin de découvrir les risques et les vulnérabilités de sécurité du système et de ses données. Nous utiliserons **Zaproxy** pour effectuer nos tests de sécurité. Ils sont souvent répartis, de manière quelque peu arbitraire, en fonction du type de vulnérabilité testée ou du type de test effectué. Une répartition courante est la suivante :

- **Évaluation de la vulnérabilité** : Le système est scanné et analysé pour détecter les problèmes de sécurité.
- **Test de pénétration (pentesting)** : Le système est analysé et attaqué par des attaquants malveillants simulés.
- **Test d'exécution** : Le système fait l'objet d'une analyse et d'un test de sécurité de la part d'un utilisateur final.
- **Examen du code** : Le code du système fait l'objet d'un examen et d'une analyse détaillés à la recherche de vulnérabilités en matière de sécurité.

Le pentesting suit généralement les étapes suivantes :

- **Exploration** : Le testeur tente d'en savoir plus sur le système testé. Il s'agit notamment d'essayer de déterminer quels logiciels sont utilisés, quels points d'extrémité existent, quels sont les correctifs installés, etc. Il s'agit également de rechercher sur le site des contenus cachés, des vulnérabilités connues et d'autres indications de faiblesse.
- **Attaque** : Le testeur tente d'exploiter les vulnérabilités connues ou suspectées pour prouver leur existence.
- **Rapport** : Le testeur rend compte des résultats de ses tests, y compris les vulnérabilités, la manière dont il les a exploitées, la difficulté des exploits et la gravité de l'exploitation.

Sécurisation des applications

Au cours des précédentes pages de ce rapport, nous avons abordé dans un premier temps, d'un point de vue fonctionnel, le projet, ses enjeux et ses objectifs. Puis nous nous sommes dirigés vers des recherches de solutions, suivi de son implémentation et de la façon dont elle répond au besoin.
Nous allons maintenant nous intéresser à la sécurité de l'application, afin de les connaître, les évaluer, et mettre en place des actions pour les anticiper, et au besoin les corriger.

Analyse des risques de l'application

Le plus simple pour procéder, est d'établir un *Plan de Continuité d'Activité* (PCA), accompagné d'un *Plan de Reprise d'Activité* (PRA). Les deux étant sensiblement la même chose, ils nous permettent de déterminer des risques, les évaluer en fonction de leur impact et probabilité, et de proposer des solutions pour les anticiper et les corriger si ils sont amenés à se produire. Ainsi, nous retrouvons :

ID	Nature du risque	Description	Gravité	Probabilité	Criticité	Conséquences si avéré	Solution préventive	Solution corrective
R1	Technique	Vulnérabilités du code.	3	2	6	Vol de données Réputation	Mettre à jour régulièrement les solutions et les dépendances. Utiliser des pratiques fortes pour sécuriser l'application.	Il faut être en mesure de pouvoir corriger rapidement les failles de sécurité, et avoir un système de détection des intrusions.
R2	Technique	Attaques par injection	3	1	3	Vol de données Réputation	Prévoir des sécurités pour empêcher le problème.	Sauvegarde régulière de la base de données, et restauration de secours si besoin.
R3	Technique	Attaque par force brute	2	2	4	Vol de données Réputation	Imposer une politique de mots de passe forte, voire un système de double-authentification.	Bloquer un compte utilisateur qui aurait été usurpé suite à un vol de mot de passe par force brute.
R4	Technique	Attaque par déni de service (DdoS)	4	3	12	Utilisation perturbée Pertes financières Réputation	Utilisation d'outils de détection et de protection contre les DdoS, comme un pare-feu.	Suspendre certains services de l'application le temps de combler la faille de sécurité ou de trouver une solution de contournement.
R5	Humain	Accès non autorisé	1	2	2	Réputation Vol de données	Mettre en place des contrôles d'accès strictes pour protéger certains espaces sensibles.	N/D
R6	Humain	Perte de données	3	2	6	Réputation Juridique Pertes financières	Sauvegardes régulières des données afin de se protéger contre les pertes.	Restauration d'une sauvegarde de données utilisateurs.
R7	Humain	Ingénierie inverse	4	2	8	Réputation Pertes financières	Définir une licence et un copyright adapté à notre solution	Recours légaux.

Figure 18: Analyse des risques de notre application de vente en ligne.

> Suite à cette image, nous souhaitons décrire les risques, et les solutions à mettre en place pour les anticiper, afin d'établir un PCA et un PRA. Puisque les risques évoluent dans le temps, nous souhaitons également mettre en place un système d'audit à intervals réguliers, dans le but de réviser ces plans afin qu'ils soient toujours d'actualités concernant les besoins de sécurité de

> notre application.

Amélioration continue

> Enfin, cette dernière section sera finalisée vers la fin du projet, où l'on décrit une démarche d'amélioration continue de notre application. Comme expliqué auparavant, une application dispose d'un cycle de vie, et son développement n'en représente qu'une partie. Il est important de penser dès à présent à ses étapes de maintenance, afin qu'elle réponde toujours au besoin dans quelques années, mais également divers procédures pour transmettre les compétences et assurer son maintien opérationnel et son évolution dans le temps.