Dossier de projet

BlaBlaQuest, le site où tu peux trouver facilement ta table de jeux !

Projet réalisé dans le cadre de la présentation du Titre Professionnel Développeur Web & Web Mobile

Présenté par Brice Correia

Ecole O'Clock, promotion Wonderland - 2021

Sommaire :

Présentation personnelle & Remerciements

Après 10 ans de bons et loyaux services dans la Santé, j'ai décidé de me lancer un grand défi : changer de métier et devenir développeur web ! Pendant plusieurs années, je me suis interdit cette idée car j'étais persuadé qu'il fallait être hyper bon en mathématiques, alors qu'en fait… c'est en logique et en débrouillardise qu'il faut être armé ! Et comme j'en étais rempli, j'ai décidé de rechercher l'école qui me permettrait de passer à l'étape d'après, et une évidence s'est vite imposée : O'Clock ! Le programme, les valeurs, la méthode pédagogique, tout collait sur la présentation et… il s'est avéré que la publicité n'était pas mensongère.

Je remercie mes camarades de la promotion Wonderland qui, grâce à l'entraide, l'humour et la bienveillance, ont fait de cette aventure un moment plaisant et très enrichissant (en particulier la team #spica & Sarah) tant sur le plan technique qu'humain.
Je tiens aussi à remercier particulièrement l'équipe avec laquelle j'ai travaillé sur ce projet : Cédric, Khaled, Erwann, et ma coéquipière de back, Jennifer.
Je remercie bien entendu nos référents pédagogiques Damien et Morgane qui ont su apporter leurs aides aux moments les plus délicats de ma formation.
Merci aussi à tous les professeurs de cette grande école, ainsi qu'à tous son staff de l'ombre (équipe admission, équipe placement, etc.) qui ont tous participé à ce que mon expérience soit la meilleure possible.
Merci O'Clock.
Merci à mes amis de la team #titans.

Et surtout, merci Laurie.

Introduction

Liste des compétences du référentiels couvertes par le projet

Activité-type 1 : Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

- Maquetter une application
    - Par la production de wireframes (sur whimsical.com) pour répondre au mieux aux demandes du *product owner* et des user stories, nourri par une réflexion sur l'*UI*(ex : map interactive) et l'*UX*(ex : dashboard) nécessaire à notre site. Dans cette étape, nous avons aussi travaillé sur la façon dont le site répondra lors du changement de support, pour qu'il garde un aspect agréable et fonctionnel en respectant les critères du *responsive*.

- Réaliser une interface web statique et adaptable
    - Création de pages contenant du html/scss (sass) responsives. Nous avons ici utilisé pour les vues des fichier en .jsx pour injecter dans notre JS le html à afficher.

- Développer une interface utilisateur web dynamique
    - Création de pages .jsx (sucre syntaxique pour JS) utilisant des composants React, permettant un affichage asynchrone de nos pages pour une meilleure expérience utilisateur.
    - Communication avec une base de données via *API* pour une injection de données dynamique.

Activité-type 2 : Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

- Créer une base de données
    - Production d'un *MCD/MLD* avec http://mocodo.wingi.net, d'un dictionnaire de données, d'un dictionnaire des routes *API*.
    - Création des tables, de leurs relations et des champs qui les composent dans la base de données (via l'*ORM* de Symfony Doctrine).
    - Remplissage de la base de données via fixtures et scraping.

- Développer les composants d'accès aux données.
    - Utilisation du modèle *MVC*. Création d'un *CRUD/BREAD* pertinent pour le front et pour la partie backOffice. La partie Modèle a été générée via l'*ORM* (Object Relational Mapper) Doctrine.
    - Création de routes *API* pour transférer les données à la partie front du projet.
    - Sécurisation des données.

- Développer la partie back-end d'une application web ou web mobile
    - Création d'un backOffice pour la gestion du site, déployé sur le serveur contenant la base de données.
    - Création d'outils de modération adaptés.
    - Sécurisation de l'accès à la base de données.

Résumé du projet

Le site BlaBlaQuest a été proposé par une amie d'une de nos camarades.
Il propose de mettre en relation des passionnés de jeux de société grâce à un site communautaire, visuellement agréable, intuitif et responsive pour qu'il soit consultable sur tous les supports existants sans perdre de ses qualités.

On peut y créer des événements à la date désirée, avec le jeu désiré (parmi ceux dans la base de données), et en définissant le nombre de participants.

Tous ces événements sont consultables par l'utilisateur via la home page sans qu'il ait besoin d'être enregistré. Cette dernière est constituée d'une carte interactive de la France permettant de lister toutes les parties en cours de création (événement où il reste des places disponibles), département par département.

Une fois qu'un utilisateur enregistré décide de participer à un événement ou lorsqu'il désire avoir plus d'informations sur l'événement, il peut discuter avec l'organisateur et tous les autres utilisateurs participant à l'événement grâce à une partie commentaire.
Si l'utilisateur enregistré décide de participer, il peut faire une demande au créateur de l'événement, celui-ci peut l'accepter ou le refuser. Une liste des participants est consultable par le créateur de l'événement. S'il est accepté, le nombre de participants s'incrémente, et si ce nombre est égal au nombre de places disponibles (choisi par le propriétaire au moment de sa création), le statut de l'événement devient "complet", et il n'est plus visible dans la liste des événements proposés par le site.
La section commentaire permet aussi l'organisation entre tous les participants une fois l'événement complet.

Pour gérer toutes ces fonctionnalités, l'utilisateur enregistré a accès à un dashboard. Il peut y consulter ses événements créés, et les événements qu'il a rejoint en tant que simple participant. Pour chaque événement, il peut y voir le nombre de participants actuel, son statut, et accéder aux détails et à la partie commentaire en cliquant dessus.
Une liste des prochains événements dans son département y est affichée, et la carte interactive est aussi disponible en miniature afin de rechercher des événements dans d'autres départements s'il en a le besoin.

Équipe et rôles

Notre équipe était constituée de cinq étudiants : deux pour la partie back-end, et trois pour la partie front-end :

(screenshot tiré du rendu de notre site pour la page de présentation de l'équipe)

Après une réflexion commune sur les besoins de notre projet, nos rôles ont été définis pour répondre au mieux à ses exigences tout en prenant en compte nos forces et compétences. Après 5 mois de formation, nous nous connaissions déjà, et le poste de *Product Owner* a été endossé par Jennifer Chaul, qui était à l'origine de la demande.

Liste des rôles choisis (en plus de développeur front & back) et fonctions :

- Product Owner :
    - Responsable de la vision du produit.
    - Représente le client (enjeux, intérêts, priorités).
    - Il s'assure du ROI (Return on investment).
- Git Master :
    - Garant du bon fonctionnement du versioning avec Git.
    - Vérifie les Pull-Requests et merges.
    - Gère les conflits entre les différents codes.
- Lead Developer :
    - Garant de la réussite du développement d'un point de vue technique.

    - Analyse les contraintes et les besoins du projet.
    - S’assure de la qualité du code, et est le principal interlocuteur lors de la résolution de problèmes rencontrés par son équipe.

- Technical lead :
    - Il prend en charge les dossiers complexes.
    - Il conseille, apporte un appui technique à l’unité de travail dans l'activité quotidienne de la production.
    - Il assure une veille réglementaire et technique.

- Scrum Master :
    - Il est le responsable du projet.
    - Il s’occupe de faciliter l’organisation du projet.
    - Il s’assure du respect de la méthode Scrum (sprint, tâches, responsabilités...).
    - Il organise et anime le daily scrum.

J’ai eu le rôle de développeur back-end, et celui de *Scrum Master* (garant de la méthode sur laquelle je reviens dans la section #Méthode d’organisation).

Cahier des charges

Présentation du projet

Le Product Owner nous a *pitch* ce qu'elle attendait du projet (cf. #Résumé du projet).

Les besoins et objectifs

Ici, nous avons synthétisé les objectifs concrets du site :

- Trouver une communauté de personnes partageant la même passion.
- Créer du lien social près de chez soi, mais aussi à travers la France entière.
- Trouver des joueurs pour jouer à ses jeux préférés.
- Découvrir de nouveaux jeux.
- Que le site soit consultable via les navigateurs Chrome et Firefox.

La cible

La Product Owner a décrit la cible visée par le site dans sa version initiale :

- Les personnes passionnées de jeux de sociétés cherchant à créer un cercle de joueurs proche de chez lui, ou cherchant à étoffer son cercle de joueurs.
- Les personnes désirant trouver des joueurs de façon sporadique lors de leurs déplacements en France.

Les évolutions possibles

Plusieurs perspectives d'évolution ont été imaginées et proposées :

- Possibilité pour les bars à jeux de créer un compte spécial pour y afficher leurs événements.
    - Possibilité de consulter la liste de tous les bars à jeux partenaires.
- Possibilité pour les festivals de jeux de société de créer un compte spécial pour y afficher leurs informations.
    - Possibilité de consulter la liste de tous les festivals de jeux de société partenaires.
- Possibilité de créer sa ludothèque en ligne pour que chaque joueur puisse montrer les jeux de sociétés qu'il possède.
    - Possibilité de consulter la ludothèque des autres joueurs.
- Implémenter un *dark mode.*
- Possibilité de consulter par l'utilisateur les archives de ses événements passés accessibles via son dashboard.

- Ajout d'un calendrier des événements futurs d'un utilisateur, accessible via son dashboard, pour une meilleure visibilité (V1 ayant uniquement une liste ordonnée par date ascendante).

Les différents rôles au sein du site

- Guest :
    - Peut consulter la liste des événements par département.
    - Peut consulter la description d'un événement.

- User : A les mêmes droits qu'un Guest + :
    - Peut créer un événement.
    - Peut écrire un commentaire sur un événement.
    - Peut demander à rejoindre un événement.

- Admin : A les mêmes droits qu'un User + :
    - Peut modérer les commentaires d'un événement via la page de l'événement.
    - Peut accéder à l'outil de gestion du site via le backOffice.

Les users stories

- Front :
    - En tant que guest/user, je peux voir les derniers événements créés.
    - En tant que guest/user, je peux voir le détail des événements.
    - En tant que guest, je peux me créer un compte.
    - En tant que user, je peux me connecter à mon compte.
    - En tant que user, je peux modifier mon profil.
    - En tant que user, je peux créer/modifier/supprimer un événement.
    - En tant que user, je peux participer à un événement.
    - En tant que user, je peux interagir avec le créateur d'événements.
    - En tant que user, je peux consulter mon dashboard.
    - En tant que user, je peux valider la participation d'un joueur.
    - En tant que user, je peux répondre aux interrogations des participants.

- BackOffice :
    - En tant qu'admin, je veux pouvoir me connecter à l'interface d'admin du site.
    - En tant qu'admin, je peux supprimer/modifier un user.
    - En tant qu'admin, je peux pouvoir consulter tous les événements et commentaires d'un user.
    - En tant qu'admin, je peux supprimer/modifier un événement.
    - En tant qu'admin, je peux ajouter/modifier/supprimer un jeu.

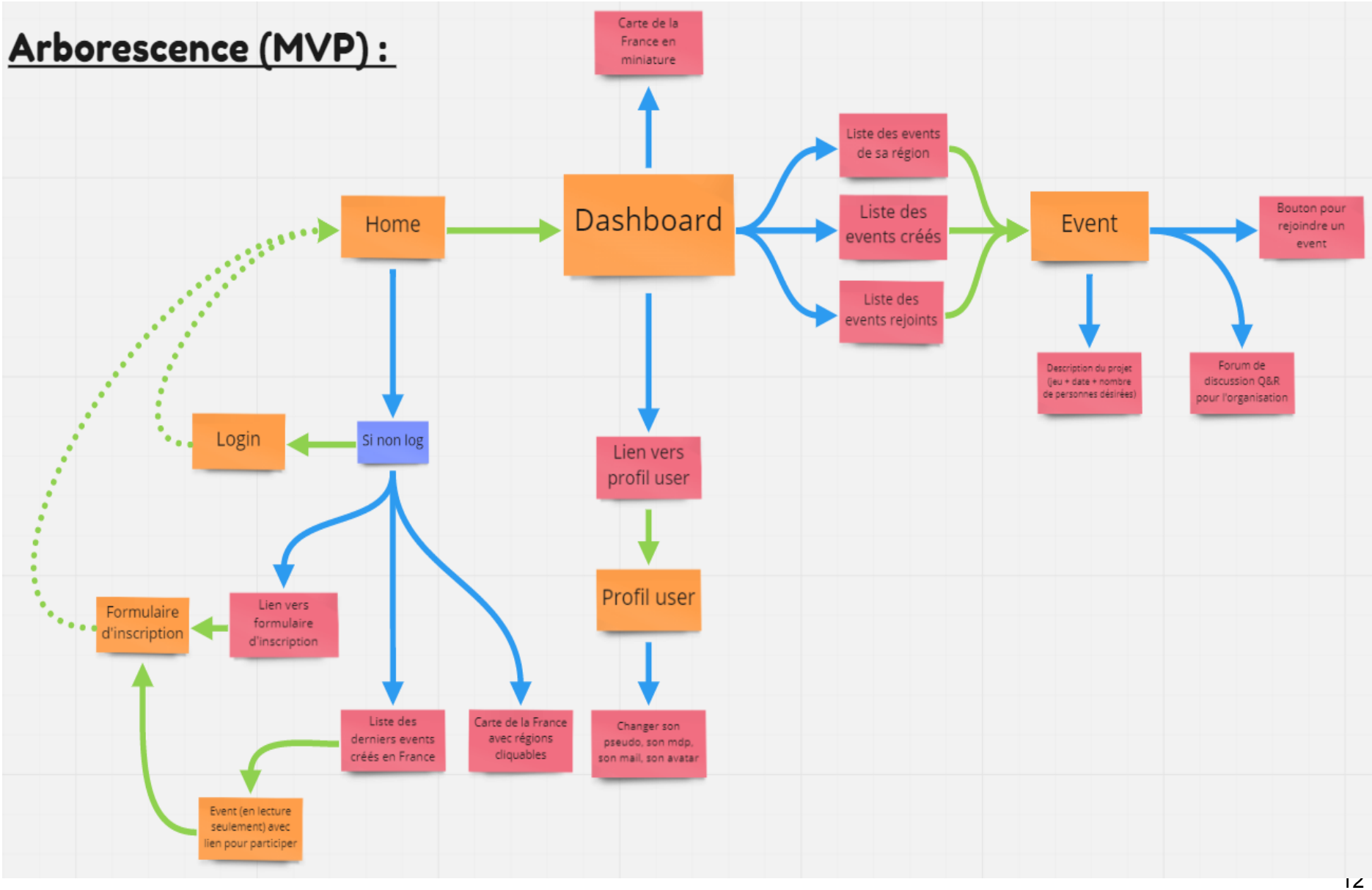
- En tant qu'admin, je peux ajouter/modifier/supprimer une catégorie.

MVP : fonctionnalités et schéma

Le site doit permettre à un utilisateur :

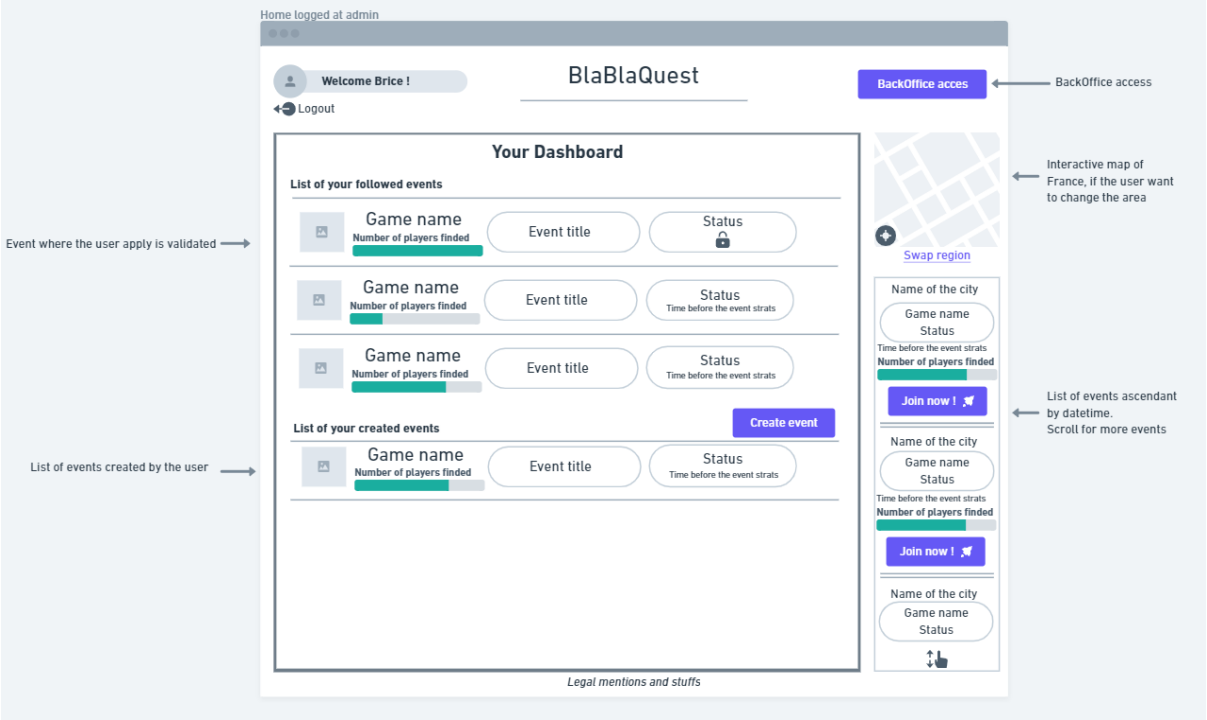
- De créer/modifier un compte.
- De se connecter à ce compte.
- De consulter les événements qui vont se passer dans chaque région de France, ou dans la France entière.
- De créer et/ou modifier un événement s'il en est le créateur.
- De participer à un événement.
- De pouvoir écrire un commentaire sur un événement s’il est connecté.
- De consulter tous les événements qu’il a créés ou qu’il a rejoints.

Arborescence (MVP) :

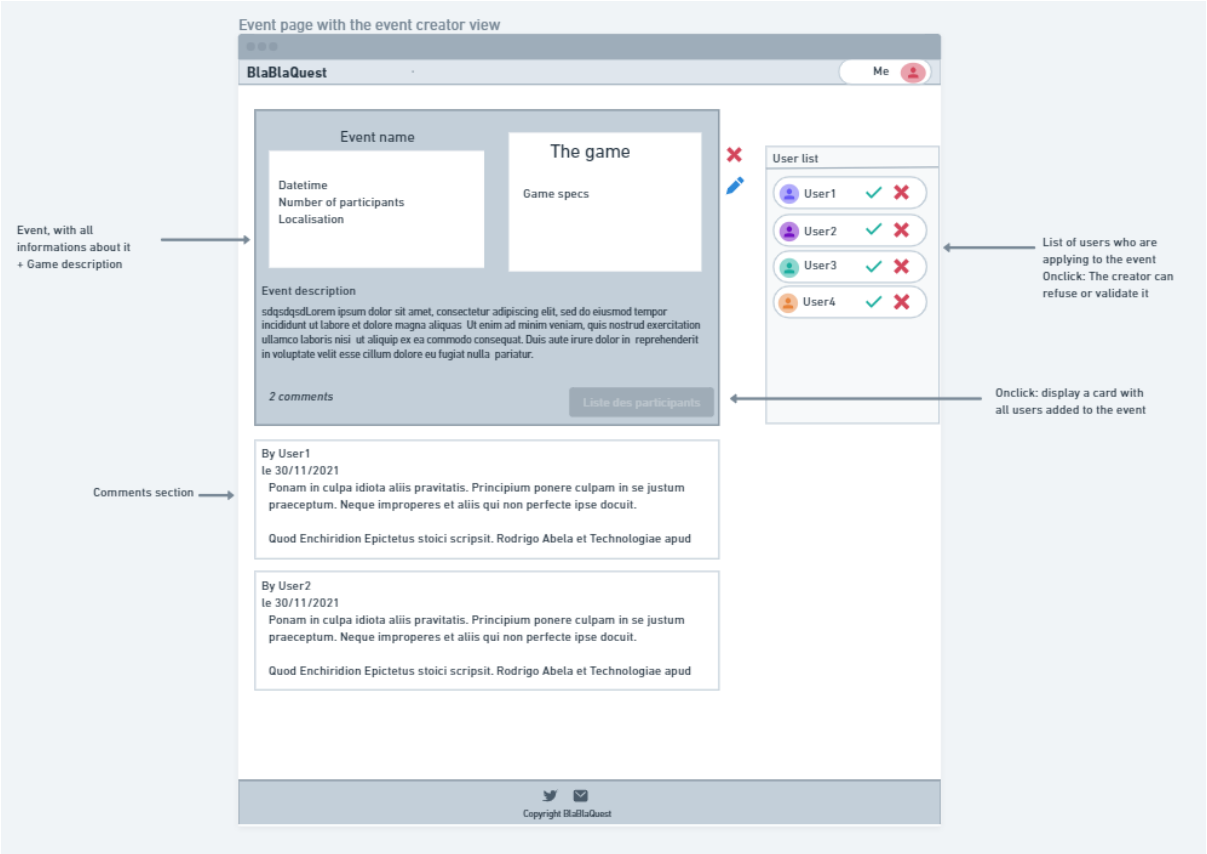


Wireframes

Une grande quantité de wireframes a été produite par notre équipe, voici deux wireframes tirés de mes productions :



(Wireframe du dashboard)



(wireframe de la page event)

Spécifications techniques

Technologies front

Pour la partie front du projet, nous avons principalement utilisé le framework **React JS**, car il était adapté à nos besoins en termes de rapidité, d'évolutivité et de simplicité de mise en oeuvre et donc correspondant parfaitement à notre désir de faire un dashboard dynamique et asynchrone (besoin d'apporter des modifications dans le *DOM* de façon ciblée et répétée). Pour un pont plus simple et plus agréable entre **JS** et les vues de la partie publique de notre site, nous avons opté pour l'utilisation de **jsx** .

La partie routing a été gérée par **React Router-DOM**, pour une utilisation de routing dynamique.

Pour une discussion plus fluide entre les composants React et les données, nous avons utilisé **React-Redux** pour créer le **Redux Store**.

Les échanges asynchrones avec la base de données via API sont gérés par **Axios** (basé sur Promise, similaire à JS Fetch).
La librairie **Formikform** a été utilisée pour les formulaires, et **Material ui** pour quelques éléments.

Pour le déploiement front, nous avons choisi **surge** (https://surge.sh/). Le site propose un service simple pour déployer une partie front en très peu de manipulation.

Technologies back

L'architecture backend *MVC* a été créée avec le framework **Symfony**. Nous l'avons choisi pour sa fiabilité, sa sécurité et sa richesse de librairies proposées. La création des entités de la base de données et le routing a été géré par son *ORM* **Doctrine**.

Côté serveur, le choix de langage des requêtes est **SQL**, et le système de gestion de base de données que nous avons choisi est **MariaDB** car il est open source, efficace, et est une référence. Pour tester nos routes API, nous avons utilisé le logiciel **Insomnia**.

Les vues ont été générées avec **twig** car c'est le moteur de templates par défaut de Symfony pour de bonnes raisons : flexible, rapide et sécurisé. De plus, sa syntaxe est agréable et facilement accessible pour les développeurs front. La librairie **bootstrap** a été utilisée pour une génération rapide, agréable et *responsive* de l'interface utilisateur.

Le scraping a été réalisé en PHP avec le bundle **Goutte**. Le langage PHP n'est pas le plus utilisé pour ce genre de fonction, mais comme ici nous n'avions pas un gros besoin de performance et que la syntaxe PHP était la plus maîtrisée par l'équipe, nous avons choisi de rester sur ce langage.

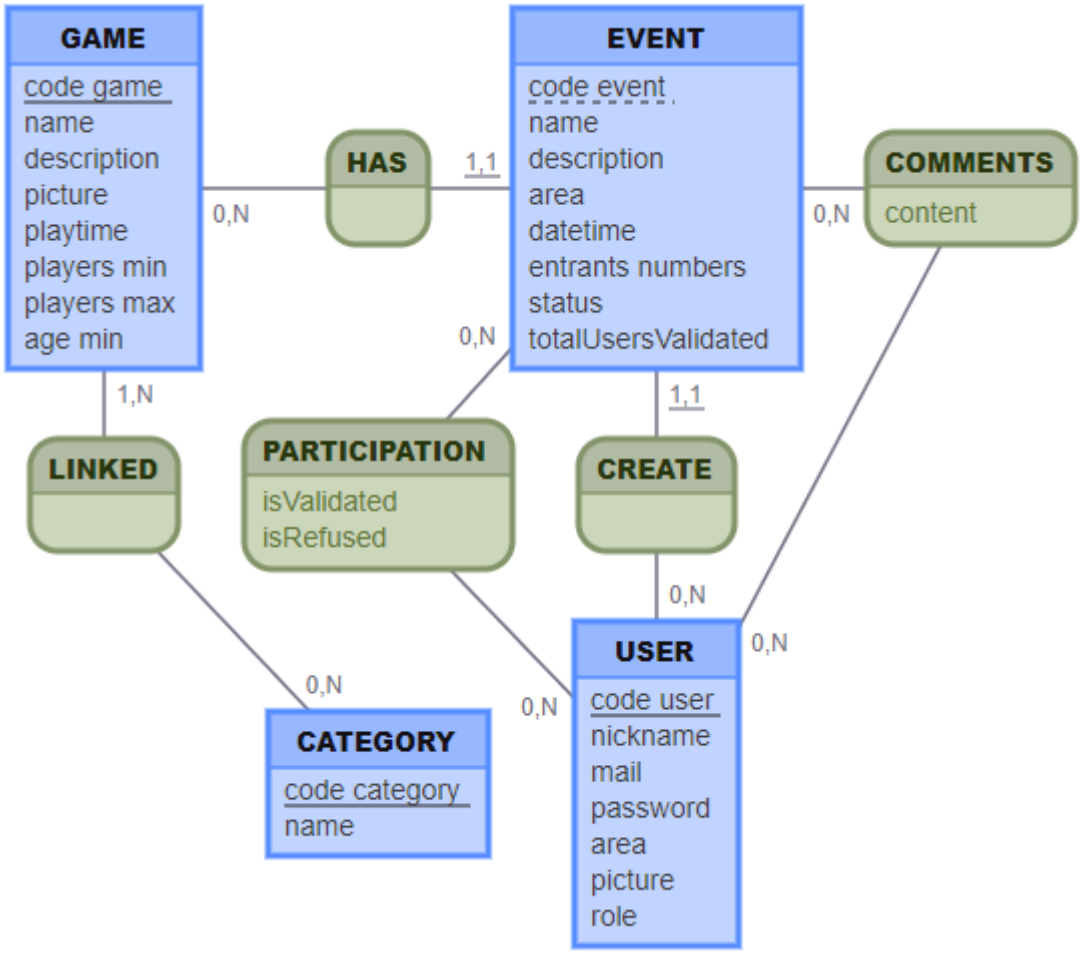
Pour la sécurité, nous avons choisi le **bundle lexik** (et son **JWT** - Json Web Token -) pour les échanges en json via *API REST*, et le bundle **nelmio/cors** pour la protection contre les requêtes externes.

Pour le déploiement back, nous avons choisi **AWS**, référence dans le domaine, car nous possédions un compte étudiant dessus.

Modélisation de la base de données

Après avoir défini le cahier des charges, les technologies dont nous avions besoin et l'architecture globale du site, nous avons modélisé la base de données en nous inspirant de la méthode *Merise* (cf #références n°2). Nous avons identifié 4 tables nécessaires à la réalisation de notre MVP : "GAME", "EVENT", "USER" et "CATEGORY".

Nous avons créé un MCD (Modèle Conceptuel de Données) afin de représenter les informations gérées par notre système de gestion de base de données. Il permet d'avoir une vue claire et graphique des futures entités de notre base de données, et les relations qu'elles auront. Le diagramme que nous avons créé :

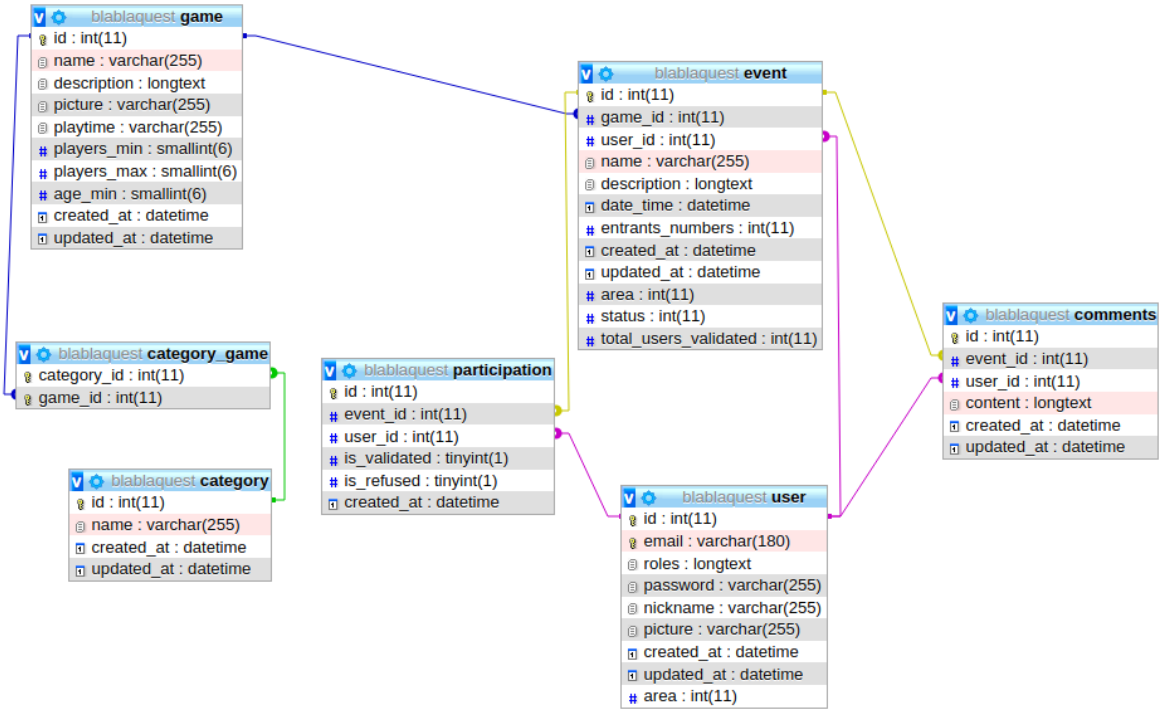


Nous y avons indiqué les cardinalités entre chaque entité, et donc identifié les relations Many-To-Many qui nécessitent des tables intermédiaires. Elles sont au nombre de 3 : “LINKED” en table intermédiaire stockant uniquement des clefs étrangères, “COMMENTS” et “PARTICIPATION” qui auront en plus des deux clefs étrangères de la relation qu’elles représentent, du contenu, et donc une clef primaire.

Une fois le MCD créé, nous sommes passé au MLD (Modèls Logique de Données) afin de représenter la localisation de toutes les clefs (en tirant les conclusions de l’analyse du MCD) :

```
GAME ( id, name, description, picture, playtime, players min, players max, age min )
EVENT ( id, game_id, user_id, name, description, area, datetime, entrants numbers, status,
totalUsersValidated )
COMMENTS ( id, user_id, event_id, content )
LINKED ( game_id, category_id )
PARTICIPATION ( id, user_id, event_id, isValidated, isRefused )
CATEGORY ( id, name )
USER ( id, nickname, mail, password, area, picture, role )
```

Nous n'avons pas fait de MPD (Modèls Physique de Données), et l'image qui suit est tirée de phpMyAdmin, mais si nous avions tenu à le faire, nous aurions pu utiliser un site comme https://dbdiagram.io/home pour faire le modèle physique.



Architecture et circulation des données côté back :

Nous avons utilisé le design pattern MVC (Model View Controller) que ce soit pour la gestion des données du site ou pour le backOffice. Pour le réaliser, nous avons utilisé l'*ORM* (Object Relational Mapping) de Symfony Doctrine. La communication s'est faite via API REST, géré côté front avec Axios.

Lorsque la vue demande des données à afficher, des requêtes HTTP sont effectuées sur les URLs que nous avons listées dans le dictionnaire des routes. Comme c'est un service web RESTful, nous utilisons les méthodes HTTP GET, POST, PUT, PATCH, DELETE. Pour le site, comme la communication se fait via API, ces méthodes sont stipulées dans l'entête du fichier *json*. Si son contenu a besoin d'être manipulé, il est *deserialize* (transformé en objet) grâce au composant *Serializer* de Symfony (cf. références 4).
Le composant Route s'occupe du mapping des routes, et permet d'orienter vers les controllers en lien avec la requête.
La circulation des données front/back se fera toujours de la même façon, à l'exception de la fonction de login, qui prendra un autre chemin. Nous commencerons par traiter ce cas.

Pour se connecter, l'utilisateur fera une requête en POST sur notre URL ^/api/v1/login_check. Cette requête sera interceptée par les *firewalls,* car nous avons ajouté la fonction de connexion via json, contrôlée par le bundle lexik_jwt. Il permet une fois la vérification des données effectuée, de rajouter à la réponse un jeton d'authentification que l'utilisateur gardera pendant le temps de son utilisation de notre service ou avant son expiration (ici paramétré par défaut à 3600 secondes, soit une heure). Ce jeton perd sa validité en cas de déconnection de l'utilisateur.

Extrait du code des firewalls :

```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    login:
        pattern: ^/api/v1/login
        stateless: true
        json_login:
            check_path: /api/v1/login_check
            success_handler: lexik_jwt_authentication.handler.authentication_success
            failure_handler: lexik_jwt_authentication.handler.authentication_failure

    api:
        pattern: ^/api
        stateless: true
        jwt: ~
```

Pour des raisons de praticité pour la partie front, nous avons en plus de ça ajouté un écouteur d'événement sur la classe AuthentificationSuccessEvent (qui est déclenchée lorsque l'authentification est un succès), pour ajouter à la réponse retournée les données de l'utilisateur venant de se connecter (stockée dans un tableau) :

```php
class AuthenticationSuccessListener
{
    /**
     * @param AuthenticationSuccessEvent $event
     */
    public function onAuthenticationSuccessResponse(AuthenticationSuccessEvent $event)
    {
        $data = $event->getData();
        $user = $event->getUser();

        if (!$user instanceof UserInterface) {
            return;
        }

        $data['data'] = array(
            'roles' => $user->getRoles(),
            'email' => $user->getUserIdentifier(),
            'id' => $user->getId(),
            'nickname' => $user->getNickname(),
            'area' => $user->getArea(),
            'picture' => $user->getPicture(),
        );

        $event->setData($data);
    }
}
```

Puis la réponse est *serialize* (toujours avec le composant Symfony *Serializer*) afin d'être renvoyée à la vue sous format json.

Pour toute autre action que login, la circulation des données sera différente:

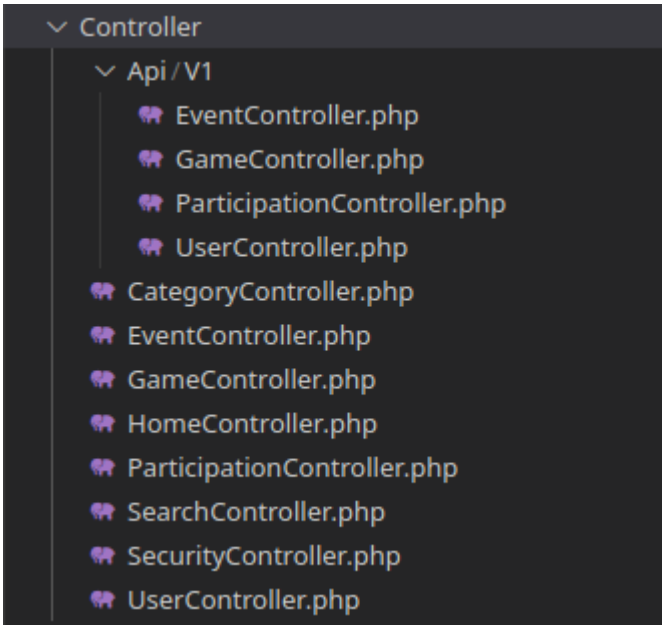
Dans un premier temps, toutes les requêtes devront passer par l'access control que nous avons mis en place pour la sécurité. Il va vérifier, selon l'URL demandée, si l'utilisateur a les droits d'y accéder. Pour ça, il va comparer les informations qu'il a avec ce que l'utilisateur lui envoi :

```
access_control:
    - { path: ^/api/v1/login, roles: PUBLIC_ACCESS }
    - { path: ^/api/v1/user/registration, roles: PUBLIC_ACCESS }
    - { path: ^/api, roles: PUBLIC_ACCESS, methods: ["GET"] }
    - { path: ^/api/v1, roles: IS_AUTHENTICATED_FULLY }
    - { path: ^/admin/login, roles: PUBLIC_ACCESS }
    - { path: ^/admin, roles: ROLE_ADMIN }
```

Il agit comme un *switch/break* : il prend la liste et la lit en descendant. S'il trouve une ligne qui correspond à l'URL demandée, il regarde la suite de la ligne et laisse passer si les conditions demandées sont validées. Pour cela, il utilise le *jeton JWT* (ou son absence) lié à toutes les requêtes une fois l'utilisateur connecté au front, ou le *jeton csrf* (ou son absence) si l'utilisateur tente de se connecter au backOffice.

Une fois cette étape passée, Doctrine continue son mapping et vient chercher le bon controller pour y déclencher la méthode demandée via la requête HTTP.

Pour plus de lisibilité, nous avons fait le choix de séparer nos controllers en 2 parties: Une pour la communication via API, et l'autre pour la communication avec les vues du backOffice:



Pour les requêtes HTTP en POST, PUT, PATCH, nous les faisons passer à travers les formulaires de *constraints* que nous avons créés, pour vérifier si les données correspondent à ce que nous attendons avant de tenter de les inscrire en base de données. Pour les

requêtes HTTP autres que GET nous utilisons la classe EntityManagerInterface de Doctrine pour interagir avec la base de données.

La partie Model du MVC de Symfony est scindée en 2 par Doctrine : Une partie *Entity*, et une partie *Repository*.

Dans *Entity,* se trouvent toutes les classes d'objets contenues dans notre base de données.
Dans *Repository,* se trouvent toutes les requêtes SQL classiques, qu'elles soient déjà créées par Doctrine (find(), findAll(), findOneBy() et findBy()), ou des requêtes que nous avons créées (avec *DQL* pour Doctrine Query Language).

Pour les requêtes HTTP autres que DELETE, nous renvoyons les données demandées, ajoutées ou modifiées à la vue via un fichier json dont les objets sont *serialize* si demandée par le site, ou une vue en twig pour le backOffice.

Quoi qu'il arrive, les réponses seront accompagnées de code HTTP pour signaler à la vue la réussite ou l'échec de l'action demandée, avec si possible un message pour expliquer avec des mots les raisons de ce code.

Les données renvoyées sont alors récupérées et traitées par la vue.

Mécanisme et cycle au déclenchement d'événement côté front

Lorsque l'utilisateur déclenche par ses actions un événement écouté par notre JS, l'action va être suivie de cycles entre nos différentes fonctions et composants. Ici, nous suivrons un événement de clic sur la map interactive de la page d'accueil, pour afficher les événements de la région sélectionnée, et qui doit donc changer les événements affichés de façon dynamique et asynchrone.

Le terme "state" sera utilisé pour décrire l'état d'une instance d'un objet contenant les paramètres des événements (dont les événements qui nous intéressent). Ce *state* est stocké dans le fichier src/reducer/event.js, et est suivi d'un switch/case qui sera déclenché pour trouver quelle fonction est appelée (et donc quel paramètre de l'objet est modifié).
Initialement, ses paramètres sont par défaut :

```
export const initialState = {
    eventsHome: [],
    eventByArea: [],
    eventSaved: [],
    eventSavedLoaded: false,
    gameSaved: [],
    eventCreated: eventCreated,
    eventClose: eventClose,
    mapArea: null,
    currentEvent: [],
    currentEventParticipation: [],
    loadedParticipation: false,
    currentEventLoaded: false,
};
```

Tout d'abord, on utilise un *handleClick* pour éviter que la page ne se recharge. Ce dernier va appeler la fonction setAreaMap(), afin de stocker dans la variable mapArea le numéro du

département sur lequel l'utilisateur a cliqué. Cette variable sera stockée dans une constante

```
export const SET_AREA_MAP = "SET_AREA_MAP"

export const setAreaMap = (mapArea) => ({
    type: SET_AREA_MAP,
    mapArea
});
```

(ici SET_AREA_MAP) :

La constante SET_AREA_MAP va être utilisée pour déclencher le switch/case du *state* de l'événement et va venir placer sa valeur dans le *state* de l'événement.

Maintenant que nous avons récupéré l'*integer* du département provenant du clic de l'utilisateur, nous allons devoir aller chercher tous les événements de ce département dans notre base de données, de façon asynchrone via Axios (clients HTTP basés sur les *promesses*), et les envoyer dans Components/HomeGrid qui est notre composant s'occupant d'injecter le *html* des *cards* relatifs aux événements dans notre vue.
Pour ça, nous allons utiliser le nouveau *state* de l'event, y récupérer la valeur de mapArea et l'envoyer en argument de getEventByArea(), fonction appelée par le useEffect dans HomeGrid avant l'envoi du *html* :

```
const {eventsHome, eventByArea, mapArea} = useSelector(state => state.event);

const dispatch = useDispatch();

useEffect(
  () => {
    dispatch(getEventHomePageFromApi())
  }, [dispatch]
);

useEffect(
  () => {
    dispatch(getEventByArea(mapArea))
  }, [dispatch, mapArea]
);
```

getEventByArea() est dans le dossier src/action et va stocker l'argument qu'on lui a envoyé (mapArea) dans une variable (area), nommer l'action GET_EVENT_BY_AREA. Le switch/case de la fonction apiMiddleware() (fonction qui import Axios) va alors créer une instanciation d'Axios pour les données demandées par l'action GET_EVENT_BY_AREA, et faire un appel sur l'URL désirée, en stipulant le type de requête HTTP REST (ici GET) :

```
case GET_EVENT_BY_AREA:
    axiosInstance.get(`/event/area/${action.area}`)
        .then((res) => {
            store.dispatch(saveEventByArea(res.data));
        })
        .catch((err) => console.log(err))
    next(action);
    break;
```

S'il y a un retour de données le .then se déclenche, lance la fonction saveEventByArea (nouvelle action se trouvant dans src/action) qui va recevoir la réponse en argument (en format json) et ensuite déclencher à son tour SAVE_EVENT_BY_AREA. Sinon un message d'erreur s'affiche dans les logs.

SAVE_EVENT_BY_AREA va retourner dans le switch/case du fichier reducers/event.js et stocker dans un tableau toutes les données dans le *state* du paramètre eventByArea de l'objet event :

```
case SAVE_EVENT_BY_AREA:
    return {
        ...state,
        eventByArea: action.events
    }
```

Maintenant que la fonction HomeGrid du composant HomeGrid appelée plus haut à reçu les données désirées, elle retourne son html avec les données récupérées injectées dedans lorsque le composant HomeGrid est appelé par la vue. Ici, la fonction native .map agit comme un foreach sur le tableau contenu dans le paramètre eventByArea du nouveau *state* de reducers/event.js :

```
  const events = eventByArea.length > 0 ? eventByArea : eventsHome;
  console.log(events);

    return (

    <div className="container-bot">
      <h1 className='container-bot__title'>Liste des événements départementaux</h1>
      <div className="container-bot__main">
        {
          events.slice(0,6).map((event) => (
            <NavLink to={`/event/${event.id}`} >
              {
                console.log(event)
              }
              <div key={event.id} className="container-bot__card">
                <div className='container-bot__card__image-container'>
                  <img className='container-bot__card__image' src={event.game.picture} alt=
{event.game.name} />
                </div>
                <h2 className='container-bot__card__event-name'>{event.name}</h2>
                <h3 className='container-bot__card__game-name'>{event.game.name}</h3>
                <div className='container-bot__card__info'>
                  <p className='container-bot__card__date'>Le
{moment(event.dateTime).format('DD/MM/YYYY')}</p>
                  <p className='container-bot__card__area'>
{convertNumberToDepartement(departements, event.area).dep_name}</p>
                </div>
              </div>
            </NavLink>
          ))
        }
      </div>
    </div>
    )
```

Un opérateur ternaire choisi en amont s'il doit afficher ce qui se trouve dans le paramètre eventByArea s’il contient des données, ou dans eventsHome dans le cas contraire.

Méthodologie

Méthode d'organisation

Pour le développement du projet, nous avons utilisé une méthode agile inspirée de la méthode *Scrum* (cf #références n°2). Nous avons découpé notre projet en plusieurs segments appelé *sprint* (cf section #synthèse des sprints), et nous avons découpé à l'intérieur de ces segments les fonctionnalités que nous avions à développer, selon l'appréciation de l'équipe sur la difficulté et le temps qu'elles prendraient, tout en essayant de les réaliser de façon efficiente pour avoir un produit minimum viable le plus rapidement possible.

Tous les matins, nous commencions la journée par des *daily scrum* (ou daily meeting), organisés par le *scrum master*. Ils permettent de partager le travail effectué par les équipes la veille (leurs avancées, leurs validations ou non des fonctionnalités prévues, et les difficultés qu'elles ont rencontrées), puis d'aborder les objectifs du jour. C'est à ce moment-là que le scrum master rédigeait le journal de projet pour garder une trace écrite de la progression globale.

La méthode *Scrum* nous a permis une grande fluidité dans les échanges, une meilleure capacité de compréhension de l'avancée globale du projet et donc de cohésion dans notre progression, et une meilleure répartition des tâches à effectuer. Les difficultés sont rapidement remontées et partagées, et les objectifs (même non atteints) sont plus facilement appréhendables (et d'un point de vue humain agréable, car la découpe par fonctionnalité procure la satisfaction de voir concrètement le projet avancer).

Outils

- Pour la communication écrite, nous avons utilisé **Slack**. Il a surtout servi à discuter entre le front et le back, et a aussi été utile lorsque les équipes ne travaillaient pas lors des mêmes heures.
- Pour les échanges vocaux, nous avons utilisé des salons **Discord**, avec généralement 2 salons : un pour le front et un pour le back, afin d'éviter le parasitage des discussions entre deux équipes qui travaillent sur des choses très différentes.
- Pour l'organisation globale, nous avons utilisé la plateforme **Miro** (miro.com), qui nous a paru comme une évidence lorsque nous cherchions des outils de travail en distanciel. Il permet de créer un grand espace interactif où il est possible d'importer les **kanbans**, les **notions**, des images, mais aussi de créer des boards/paper-boards où chacun peut travailler simultanément. Une organisation en "frame" permet aussi de classer et d'accéder rapidement aux différentes informations et travaux en cours ou passés. Cet outil nous a permis de se rapprocher le plus possible de ce qu'on peut vivre dans un open-space classique.

- Comme évoqué précédemment, nous avons utilisé **kanban** pour l'organisation des sprints, ce qui nous a permis une meilleure vision des objectifs en cours, du travail réalisé et des fonctionnalités restant à coder.

Versioning

Nous avons utilisé la plateforme *github* pour partager nos codes. Si initialement, nous pensions utiliser un seul *repository*, très rapidement nous avons opté pour deux : un pour le front et un pour le back. Cela a permis une plus grande souplesse dans le développement, et une fluidité dans la communication à l'intérieur des équipes. Le *Git Master* y a joué un rôle crucial, car même si nous avions utilisé *git* et *github* dès le début de la formation (cette dernière abordant *git* dès le jour 1), la synchronisation et l'utilisation de ces outils en équipe étaient différentes de la façon dont on gérait nos projets jusque-là. Car, à part quelques sessions en peer-programming, nous les avons surtout utilisés seul pour nos projets, exercices et "contrôles" (le terme *parcours* est préféré par notre école).

Le *git Master* a donc préparé avant le sprint 1 un résumé des bonnes pratiques en .md, que nous nous sommes efforcés de suivre tout au long du projet. Utilisation de branche différente pour chaque fonctionnalité, le pull en début de journée pour éviter les conflits (ou si un problème de communication était survenu, par exemple après qu'un des participants ait codé le soir/la nuit) et le push en fin de fonctionnalité et/ou fin de journée. Il a aussi insisté sur l'importance de la communication dans l'équipe (surtout à l'intérieur des équipes front et des équipes back), et l'importance des commits réguliers (pratique que nous avions déjà beaucoup développée tout au long de notre formation). Il nous a créé une "nomenclature" de la syntaxe des titres des commits, pour que nous ayons une pratique commune. Cela a permis une homogénéité dans nos repositories et une rapidité de compréhension du travail des collègues. Voilà un extrait de ce .md traitant de la syntaxe :

La syntaxe des commits

exemple

```
<type>(<scope>): <subject>
```

"type"

- `build` : modifications liées à la construction (Par exemple : liées à npm/ajout de dépendances externes).
- `chore` : un changement de code que l'utilisateur externe ne verra pas (par exemple : changement vers le fichier .gitignore ou le fichier .prettierrc).
- `feat` : une nouvelle fonctionnalité.
- `fix` : une correction de bug.
- `docs` : modifications liées à la documentation.
- `refactor` : un code qui ne corrige pas de bug ni n'ajoute de fonctionnalité. (par exemple : vous pouvez l'utiliser lorsqu'il y a des changements sémantiques comme renommer un nom de variable/fonction).
- `perf` : un code qui améliore les performances.
- `style` : un code lié au style.
- `test` : ajout d'un nouveau test ou modification d'un test existant.

"scope" (facultatif)

- La portée doit être un nom et elle représente la section de la section de la base de code.

"subject"

- utilisez l'impératif, le présent (par exemple : utilisez « add » au lieu de « added » ou « adds »).
- ne pas utiliser de point (.) à la fin.
- ne pas mettre la première lettre en majuscule.

Nous avons peu utilisé les *pull-requests* du fait de la taille de nos équipes.

Et enfin, nous avons suivi son ultime recommandation : ne pas *merge* seul, mais toujours attendre un membre de son équipe pour effectuer cette action, car c'est la partie la plus sensible et à risque dans le versionning.

Synthèse des sprints

Tous les sprints de ce projet se sont terminés par une présentation en stream de l'avancée de nos travaux devant nos référents pédagogiques, et les autres groupes s'ils le désiraient. Chaque sprint comprend donc une phase de préparation de cette présentation.

La durée de chaque sprint a été d'une semaine, sauf le dernier (n°3) qui a duré 3 jours.

Sprint 0

Phase de conception du projet.

Une fois le pitch du site effectué par le Product Owner et notre outil de gestion et de suivi de projet choisi, nous avons fait une grande session de partage d'idées sur *miro* ("paperboard" - tiré de nos archives - en #annexes). Nous y avons partagé nos idées sans restriction, couvrant tous les aspects du site : Idée de fonctionnalité, idée de page, nécessité technique, ressources techniques nécessaires, technologies nécessaires, mais aussi iconographie et idées de design. Nous avons ensuite ordonné ensemble toutes ces idées pour créer un début d'arborescence du site.

Nous avons alors commencé la rédaction du Cahier des Charges (CDC) en partant de l'utilisateur pour aller vers le technique (Objectifs/cible/user-stories puis arborescence, pour aller vers fonctionnalités/Wireframes/liste des technologies nécessaires). Le CDC a été le fil rouge de ce sprint, et nous l'avons régulièrement mis à jour selon l'avancée de notre travail.

Il nous a donc fallu différencier ce que serait notre Minimum Viable Product (MVP), et ce qui serait gardé comme fonctionnalité à développer pour des versions ultérieures. Nous avons visé une version utilisable, atteignable dans le temps imparti en prenant en compte les besoins de veille/recherche afin d'appréhender certains besoins non maîtrisés par l'équipe avant le début du projet.

Avec cette vision globale (et surtout commune), nous avons commencé à produire des wireframes grossiers, puis de plus en plus affinés afin d'être certain que nous allions dans la même direction par rapport aux fonctionnalités et besoins.

Une fois le MVP défini et nos fonctionnalités nécessaires pour sa réalisation identifiées, nous avons pu planifier nos futurs sprints. Nous avons découpé les fonctionnalités à développer en les répartissant selon leurs priorités pour que l'ensemble garde une structure logique, et que les fonctionnalités interdépendantes marchent le plus rapidement possible.

Nous avons ensuite modélisé la base de données en créant un MCD, puis un MLD et le dictionnaire des données (cf. #Modélisation de la base de données)

Enfin, nous avons créé le dictionnaire des routes. Il contient une partie pour les routes front, une partie pour les routes back, et une partie pour les routes API.

Sprint 1

Architecture front et back.

Grâce au dictionnaire des routes, que nous avons fait dans le Sprint 0 et en s'appuyant sur le découpage et la priorisation des fonctionnalités à développer, nous avons pu commencer à coder en scindant l'équipe front et l'équipe back sur deux canaux Discord pour une meilleure communication.

Côté front, l'objectif était d'avoir toutes les pages de l'arborescence du MVP. Ils ont codé le html nécessaire et créé une base de scss pour un travail plus agréable. Le html de la carte interactive a nécessité beaucoup de temps.
Pour pouvoir travailler en attendant que la base de données soit utilisable et déployée, ils ont créé un .json avec de fausses données (bouchon) pour simuler une discussion via API. L'architecture de la circulation des données a été créée, et les premières fonctions de dynamisation ont été codées.

Côté back, notre objectif du sprint était de fournir une base de données utilisable via API. Cela comprend donc la création de la base de données, la création des controllers, et la création de *fixtures* pour que les données renvoyées respectent le dictionnaire des données et soient les plus crédibles possible une fois que nous l'aurions déployé.

Nous désirions donc aussi terminer le sprint par le déploiement de notre base de données afin que l'équipe front puisse tester les fonctionnalités de discussion via API le plus tôt possible.

Sprint 2

Dynamisation, sécurité et backOffice.

Pour l'équipe front, ce sprint a été surtout dédié aux connexions avec les routes API, à l'élaboration de scripts pour que les bonnes informations soient affichées, et que toutes les dépendances réagissent correctement entre elles.

Côté back, nous avons commencé à corriger les premières erreurs remontées par le front en rapport avec les données échangées via API.

Nous avons commencé le sprint 2 par la résolution d'un énorme oubli dans la modélisation de notre base de données. Cela nous a pris beaucoup de temps. Nous nous sommes rendu compte qu'un énorme oubli avait été fait concernant les entités de la base de données, et qu'une table intermédiaire avec du contenu n'avait pas été faite dans la modélisation, et que nous devions la créer. C'était la relation "Participation" entre l'entité Event et l'entité User. Elle devait contenir un champ "isValidated" et un champ "isRefused". Nous avons ici créé 2 *booléens*, mais nous aurions pu choisir un *integer* pour représenter les 2 états. Cet oubli a

entraîné une correction de la base de données, des fixtures. Nous avons alors pu coder la méthode pour envoyer l'information nécessaire au front.

Une fois ce souci réglé, nous avons créé le backOffice de notre site. Nous avons créé les controllers (BREAD), fait le visuel du backOffice avec Bootstrap et twig, et connecté la base de données aux vues.

Nous avons sécurisé la base de données avec des access controls, ajouté les *constraints* aux données entrantes pour tripler le filtrage (une fois en front, une fois avant l'envoi de la requête SQL avec les *constraints*, et une fois à l'entrée de la base de données). Nous avons aussi ajouté un système de *voters* pour gérer les droits sur le backOffice, et avons déployé l'utilisation de token csrf afin de se protéger des vols de sessions.

Nous avons en bonus ajouté un script de *scraping* pour les jeux de société contenu dans notre base de données pour que le site puisse être réellement utilisable.

Sprint 3

Fin de dynamisation, fix, scss et nettoyage du code.

La partie front a terminé la connexion dynamique du dashboard, la dynamisation du formulaire d'inscription et celle de la création d'un événement.
La qualité de l'aspect visuel a été augmentée avec un énorme travail sur le scss et le *responsive* du site.

Côté back, nous avons continué à *fixer* les soucis remontés par le front.
Nous avons ajouté un script pour modifier le statut de l'événement si sa date est dépassée, afin qu'il ne s'affiche plus en front (et aussi pour que cette donnée puisse être utilisée dans une version suivante, lors de l'implémentation de la fonctionnalité d'archivage des événements de chaque utilisateur).
En parallèle, nous avons nettoyé notre code, ajouté des commentaires et factorisé quelques bout de script lorsque nous avions le temps.

Réalisation personnelle

Pour cette section, j'ai décidé de vous présenter 4 fonctionnalités que j'ai codées.

Service pour script Scraping

Pour avoir une base de données utilisable par les utilisateurs, nous avons opté pour un script de scraping. Il a pour but de récupérer pour les jeux les plus joués du site philibertnet.com le nom, l'image, la description, le temps de jeu estimé, l'âge minimum, le nombre de joueurs minimum et maximum, ainsi que les catégories qui leur sont liées.

Pour coder ce script, après quelques recherches, j'ai trouvé un bundle simple et en PHP (donc avec une syntaxe qui m'était très familière): Goutte (cf. références 4).

Pour son utilisation en rapport avec les faux événements créés pour les tests, le script a été relié à la création des fixtures. À terme, il sera lié à une commande si le site est utilisé sans fixtures. Dans le fichier DataFixtures/appFixtures, j'ai placé dans le __construct() de l'objet un nouveau paramètre pour paramétrer le nombre de pages à scraper. Puis j'ai appelé la fonction qui va chercher le Service contenant le script createCategoriesAndGames(), et appelé dans cette dernière la fonction lançant le script initialScraping(). Tous les jeux ont été stockés dans un tableau pour être réutilisés plus tard comme évoqué plus haut (lors de la création de faux événements) :

```php
    // Making the parameter for passwords hashing in the construct
    public function __construct(UserPasswordHasherInterface $passwordHasher, $pageToScrap =
1)
    {
        $this->passwordHasher = $passwordHasher;
        $this->pageToScrap = $pageToScrap;
    }

//////////////////////////////////////////////////////

$gamesEntities = $this->createCategoriesAndGames($manager, $pageToScrap);

//////////////////////////////////////////////////////


public function createCategoriesAndGames(EntityManagerInterface $manager, $pageToScrap)
    {
        return Scraping::initialScraping($manager, $pageToScrap);
    }
```

Comme j'avais à interagir avec la base de données pour insérer dans les tables `game` et `category` les données récupérées, j'ai appelé dans le __construct l'entityManager de l'ORM Doctrine.

Puis j'ai créé via Goutte un client http, pouvant lancer des requêtes (ci-dessous stocké dans la variable $client). J'ai pu utiliser la fonction native request() (qui appelle une URI) en y renseignant en paramètre la méthode HTTP et l'URI désirée, et ai stocké le résultat dans une variable $crawler. Cette variable contient tout le code HTML de la page ciblé :

```
        // Create a Goutte Client instance (which extends
Symfony\Component\BrowserKit\HttpBrowser) with custom setings
        $client = new Client(HttpClient::create(['timeout' => 60]));

        // Go to the philibert.com page for all board games classed by notes, ordered by
DESC
        // The method returns a Crawler object (Symfony\Component\DomCrawler\Crawler)
        $crawler = $client->request('GET', 'https://www.philibertnet.com/fr/50-jeux-de-
societe/s-3/langue-francais/categorie-jeux_de_societe?orderby=sales&orderway=desc');
```

Pour plus de confort, et parce que le script peut prendre du temps, j'ai ajouté une progress bar dans le terminal (cf. références 5). J'y ai ajouté des informations une fois le script terminé, pour indiquer dans le terminal le nombre de jeux entrés en base de données et le nombre d'échecs rencontrés (car parfois, il y avait des jeux qui n'arrivaient pas à être récupérés : cas spéciaux, ou temps de requêtes serveur malgré l'utilisation de la fonction sleep() - qui permet de configurer un temps en seconde avant de continuer la lecture du code - cf. références 6). Pour ne pas arrêter le script, j'ai ajouté un try/catch qui en cas d'échec incrémente le chiffre des échecs, et n'arrête pas le script.

Puis à l'aide d'un *sélecteur css* (cf. références 7) et de la fonction filter() de la classe DomCrawler (cd. références 8), j'ai récupéré tous les *nodes* des liens vers les pages de jeux que j'ai stocké dans la variable $links. Puis avec un foreach, j'ai parcouru tous ces liens avec la fonction click(). À chacun d'entre eux, j'ai dans un try/catch (expliqué plus haut) instancié la classe Game() dans $game, et pour chaque information désirée, rechercher leur emplacement dans la page à l'aide de sélecteurs css et *set* le contenu (soit via la fonction text(), soit getUri() pour les liens - pour récupérer l'image -) dans le nouvel objet Game() créé préalablement :

```php
$links = $crawler->filter('p.s_title_block > a')->links();

// [foreach links]

$crawler = $client->click($link);

$game = new Game();

$name = $crawler->filter('h1')->text();
$game->setName($name);

$description = $crawler->filter('#short_description_content')->text();
$game->setDescription($description);

$picture = $crawler->selectImage($name)->image()->getUri();
$game->setPicture($picture);
```

Des exceptions sont arrivées, et j'ai dû adapter mon script :

- Pour l'âge, nous désirions seulement le chiffre, hors le <*li*> contenait un texte. J'ai utilisé une *regex* pour ne récupérer que les chiffres après avoir découpé le texte via preg_split(), fonction qui reçoit en paramètre le masque à rechercher dans la *string* (ici un chiffre donc la *regex* "/\D+/" vient cibler tout ce qui n'est pas digit) et le texte dans lequel chercher (cf. références 9). Une recherche avec preg_match() et \d pour extraire seulement les digits aurait été une autre solution.

- Pour la durée de la partie, car parfois, il n'y en avait pas. J'ai donc utilisé un if/else pour remplacer le chiffre en cas d'absence de durée :

```php
if (null === ($crawler->filter('li.duree_partie')->text())) {
    $playtime = 'Jusqu\'à l\'infini est au delà !';
    $game->setPlaytime($playtime);
} else {
    $playtime = $crawler->filter('li.duree_partie')->text();
    $game->setPlaytime($playtime);
};
```

- Pour le nombre de joueurs minimum et maximum, car il faut tirer 2 chiffres d'une phrase, sauf que parfois, il n'y a qu'un seul chiffre (pour les jeux où seuls 2 joueurs sont attendus par exemple). J'ai utilisé la même fonction preg_split() que pour l'âge, la même *regex* pour isoler les chiffres que j'ai pour cette fonction stocké dans un

tableau car potentiellement plusieurs :

```php
$playersNbInString = $crawler->filter('li.nb_joueurs')->text();

 $playersNb = array_filter(preg_split("/\D+/", $playersNbInString));

if (array_key_exists(1, $playersNb)) {
    $playersMax = $playersNb[1];
}

$playersMin = $playersNb[0];

$game->setPlayersMin($playersMin);
$game->setPlayersMax($playersMax);
```

J'ai effectué le même processus pour les catégories en créant des objets Category().
A la fin de chaque page, j'ai incrémenter une variable contenant initialement un *integer* de 1 pour pouvoir créer une nouvelle requête URL pour la page suivante :

```php
// At the begining of the script
$urlIncrementation = 1;

// After one page scraped

$urlIncrementation++;

$crawler = $client->request('GET', 'https://www.philibertnet.com/fr/50-jeux-de-societe/s-3/langue-francais/categorie-jeux_de_societe?orderby=sales&orderway=desc&p=' . $urlIncrementation);
```

Et pour terminer, j'ai demandé à Doctrine de tout envoyer dans la base de données.

SearchBar pour users dans backOffice

Pour un outil de modération plus pertinent, j'ai créé une *search bar* dans le backOffice, afin de pouvoir retrouver un utilisateur précis dans la base de données. J'ai créé un nouveau controller, et ajouté en haut de la page affichant la liste de tous les utilisateurs un *<form>* avec pour type "search" et menant à une page d'affichage des résultats sur le *submit :*

```
<form class="d-flex mb-3 mt-3" action="{{ path('search_results') }}">
    <input class="form-control form-control-sm me-2" name="search" type="search"
placeholder="Rechercher un utilisateur...">
    <button class="btn btn-danger" type="submit">
        <i class="fa fa-search"></i>
    </button>
</form>
```

Lorsque le controller répondant à l'URL vers lequel ce formulaire est déclenché, il récupère la requête, extrait le terme stocké dans "search" et le stock dans une variable $searchTerm. Il fait ensuite une requête à la base de données via Doctrine, en utilisant une requête DQL que j'ai codée dans le UserRepository, avant de renvoyer le tout à une vue twig se trouvant dans search/results.html.twig :

```php
/**
 * @Route("/admin", name="search_")
 */
class SearchController extends AbstractController
{
    /**
     * @Route("/search", name="results")
     */
    public function results(Request $request, UserRepository $userRepository): Response
    {

        $searchTerm = $request->get('search');

        $results = $userRepository->findAllBySearchTerm($searchTerm);

        return $this->render('search/results.html.twig', [
            'results' => $results,
            'searchTerm' => $searchTerm
        ]);
    }
}
```

```php
    public function findAllBySearchTerm($searchTerm)
    {
        return

            $this->createQueryBuilder('user')

            // WHERE user LIKE searchTerm
            ->andWhere('user.nickname LIKE :searchTerm')
            ->setParameter(':searchTerm', "%$searchTerm%")

            ->getQuery()
            ->getResult();
    }
```

(Nb : les %[...]% entourant la recherche contenue dans $searchTerm permet de rechercher tous les utilisateurs ayant un *nickname* contenant la valeur de la variable, pour plus de flexibilité)

```
{% extends 'base.html.twig' %}

{% block title %}
Recherche pour {{ searchTerm }}
{% endblock %}

{% block body %}
<div class="container">
    <h1>Résultat pour la recherche "{{ searchTerm }}" :</h1>
    <ul>
    {% for user in results %}
        <a class="btn mt-3 mb-3 btn-lg btn-dark" href="{{ path('user_read', {id:user.id})
}}">
            {{ user.nickname }}
        </a>
    {% endfor %}
    </ul>
</div>
{% endblock %}
```

Fixtures: creatUsers() et refactorisation du code

Comme notre script de création de *fixtures* commençait à être long, j'ai décidé de refactoriser une partie de celui-ci. J'avais codé un script de génération aléatoire d'utilisateurs, mais nous voulions aussi la création de comptes particuliers et constants pour nos tests (user/admin et un autre utilisateur pour notre présentation finale - pour une blague précise -). Cet ajout augmentant la lourdeur du code, j'ai décidé de le découper et de le soustraire au script de base.

Lorsque le script de *scraping* se lance, il appelle une fonction nommée createUsers() (il stockera les résultats dans un tableau pour pouvoir lier ultérieurement des événements et commentaires aux utilisateurs créés) :

```php
// At the begining of the script
$usersEntities = $this->createUsers($manager);

// At the and of the file

    public function createUsers(ObjectManager $manager)
    {
        $users = [];

        foreach (UserDataFixtures::getData() as $userData) {
            $user = new User();
            $user->setNickname($userData['nickname']);
            $user->setEmail($userData['email']);
            $user->setPassword($this->passwordHasher->hashPassword($user,
$userData['password']));
            $user->setPicture($userData['picture']);
            $user->setRoles(($userData['roles']));
            $user->setArea(mt_rand(1, 95));

            $manager->persist($user);

            $users[] = $user;
        }

        return $users;
    }
```

La fonction createUsers() va venir extraire les données que j'ai stocké dans un fichier à part (DataFixtures/UserDataFixtures) afin de *set* toutes les données attendues en paramètre pendant la création d'un compte utilisateur en base de données.

Code se trouvant dans UserDataFixtures :

```php
class UserDataFixtures
{
    public static function getData() {

        return [
            [
                'nickname' => 'user',
                'picture' => 'picture',
                'email' => 'user@bbq.bbq',
                'password' => 'bbq',
                'roles' => ['ROLE_USER'],
            ],
            [
                'nickname' => 'admin',
                'picture' => 'picture',
                'email' => 'admin@bbq.bbq',
                'password' => 'bbq',
                'roles' => ['ROLE_ADMIN'],
            ],
            [
                'nickname' => 'darknasus',
                'picture' => 'picture',
                'email' => 'darknasus@bbq.bbq',
                'password' => 'darknasus',
                'roles' => ['ROLE_USER'],
            ]
        ];
    }
}
```

Ainsi, pour chaque tableau contenant les données utilisateurs désirées, la fonction va créer un utilisateur, l'inscrire à la base de données, et à la fin du *foreach*, renvoyer le tableau les contenant pour les lier à des événements et commentaires aléatoires dans la suite du script de création de *fixtures*.

Méthode Validate de ParticipationController

Pour tout de même présenter une des méthodes de *controller* autre que celle activée par la *search bar*, j'ai décidé de présenter un extrait controller intéressant : celui des participations. En plus de faire un ADD dans la base de données, il doit contenir un moyen de modifier le statut s'il est complet (afin de ne plus l'afficher dans les événements disponibles dans les vues).

Lorsque le propriétaire d'un événement accepte la participation d'un utilisateur, une requête HTTP est faite sur [URI]/api/v1/participation/{id}/validate/{event} en PUT ({id} = l'id de la participation, {event} = l'id de l'événement), et la fonction validate() est lancée.

Dans un premier temps, elle va venir modifier en base de données la participation de l'utilisateur en modifiant les booléens de is_validated en "true" et is_refused en "false" grâce à l'EntityManager de Doctrine.

Ensuite, elle va venir interroger la base de données pour récupérer tous les utilisateurs validés pour l'événement via une requête DQL, et stocker ce nombre dans la variable $totalUsersValidated. Grâce à la fonction native PHP count(), le nombre total d'utilisateurs validés va être trouvé. Après l'avoir set dans total_users_validated il va être comparé au nombre total de participants accepté par l'événement (entrants_numbers). Avec un if/else, on va venir set le statut de l'événement à 1 s'ils sont égaux (ce qui représente un événement complet). Sinon, on set l'événement à 0 (ce qui veut dire qu'il reste de la place dans l'événement et qu'il est encore ouvert).

Enfin, on vient inscrire dans la base de données le statut de l'événement, et on retourne une réponse http 200 pour signaler au front que la validation a été validé en base de données :

```
    public function validate(EntityManagerInterface $manager, Request $request, Event
$event, Participation $participation, ParticipationRepository $participationRepository)
    {

        $participation->setIsValidated(true);
        $participation->setIsRefused(false);

        $manager->persist($participation);
        $manager->flush();

        $totalUsersValidated = $participationRepository->findByValidated($participation-
>getEvent());

        $totalUsersValidated = count($totalUsersValidated);

        $event->setTotalUsersValidated($totalUsersValidated);

        if ($event->getEntrantsNumbers() == $totalUsersValidated) {

            $event->setStatus(1);

        } else {

            $event->setStatus(0);
        }

        $manager->persist($event);
        $manager->flush();

        return $this->json(null, 200);
    }
```

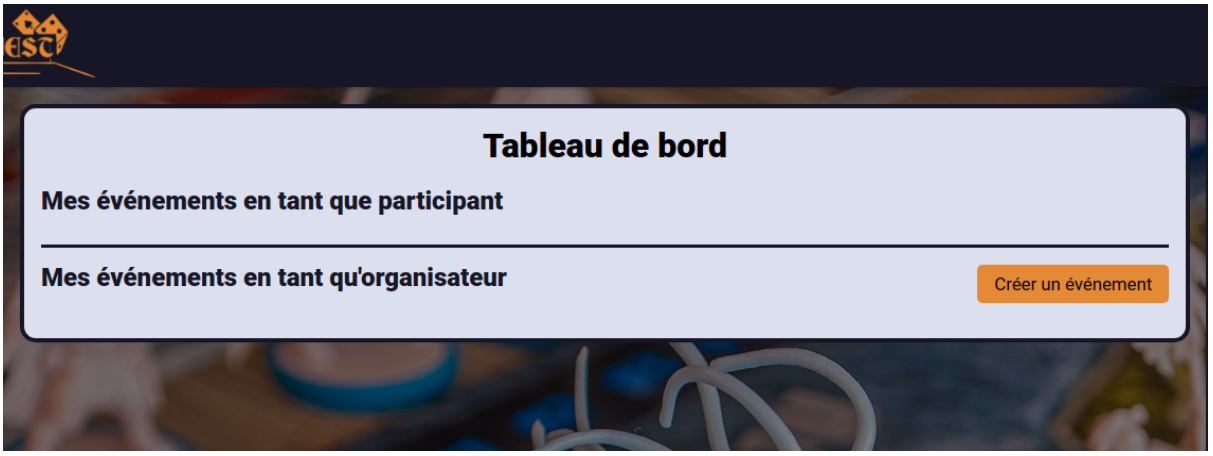
Jeu d’essai significatif

Tout au long du projet, nous avons effectué énormément de tests. Que ce soit pour les routes API, les inscriptions en base de données, les actions utilisateurs. Ici, je vais présenter une action utilisateur représentative de notre site : La création d’un événement.

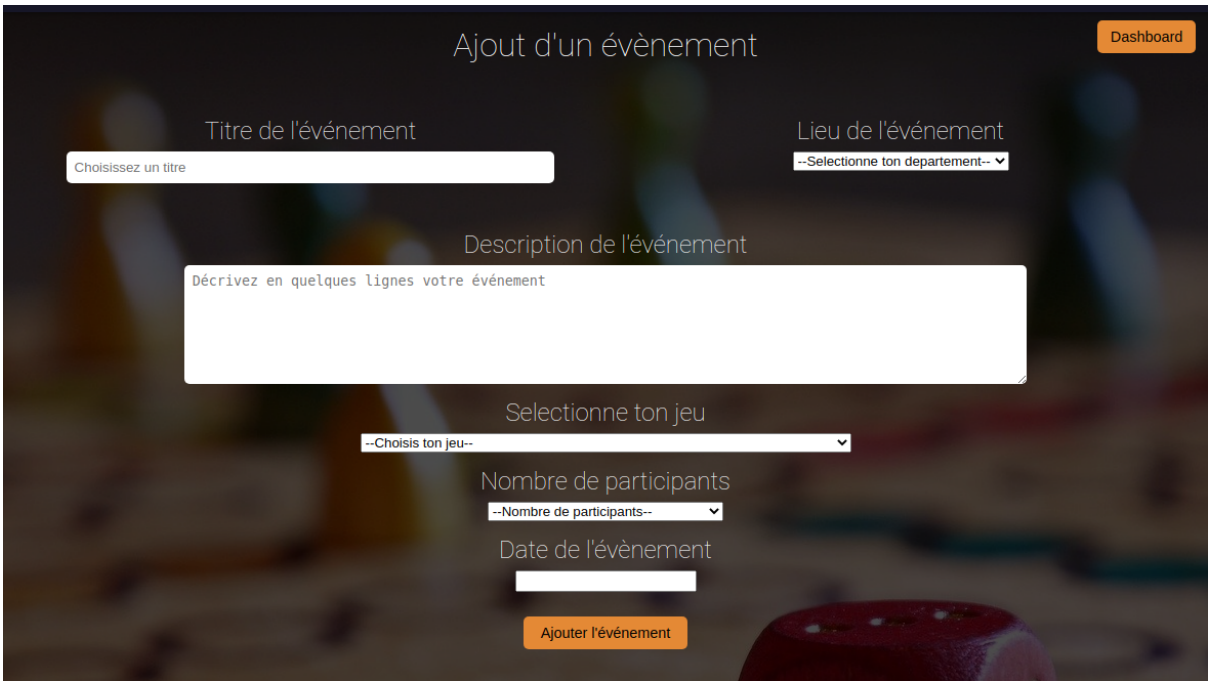
Avec un compte d'utilisateur enregistré, créé pour le test, je vais essayer de créer un événement. Pour être concluant, le test devra valider les points suivants :

- L’affichage du formulaire de création d’événement lorsque je clique sur “Créer un événement”.
- L’affichage de tous les jeux disponibles en base de données lorsque je désire sélectionner un jeu pour mon événement.
- L’affichage d’un message flash pour informer l’utilisateur que l’événement a bien été créé.
- Confirmation que les données sont correctement inscrites en base de données.
- Constater que l’événement apparaît bien sur le dashboard de l’utilisateur, et que les données affichées correspondent à ce qui a été entré par l’utilisateur lors de la création de l’événement.
- Confirmer que la page “événement” est consultable, et que toutes les données à afficher correspondent à ce qui est attendu.

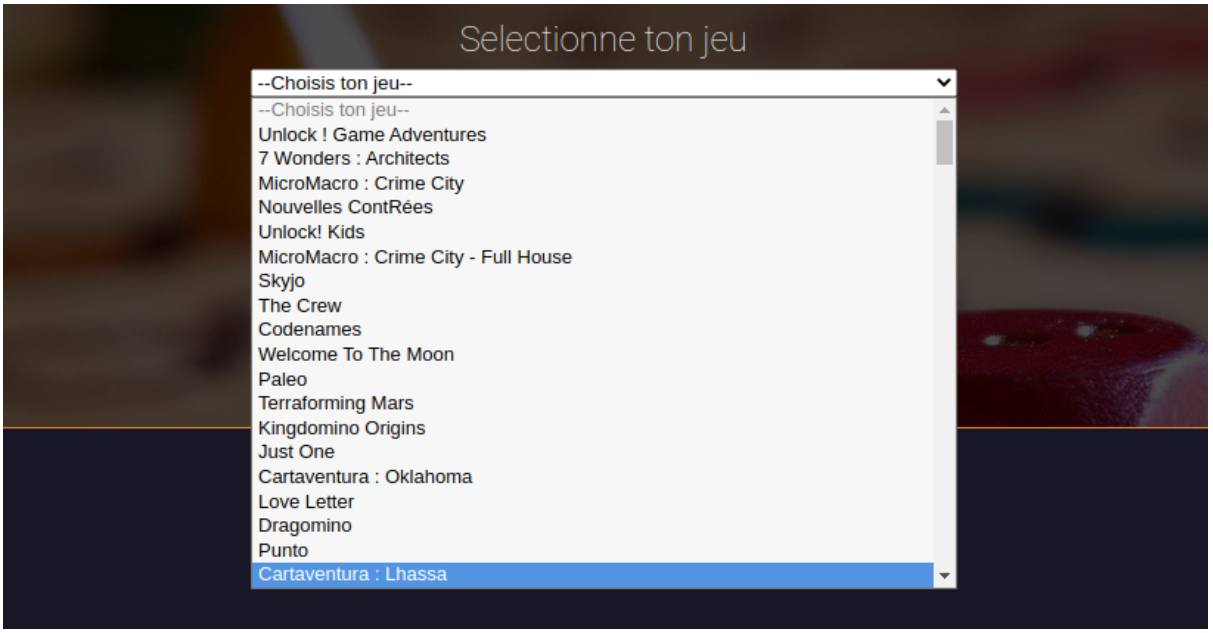
Initialement, le dashboard est vide :



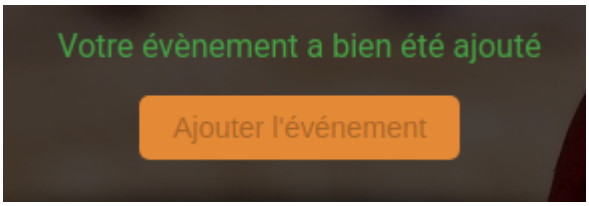
Lorsque je clique sur “Créer un événement” le formulaire de création d’événement s’affiche correctement :



Lorsque je désire sélectionner un jeu pour mon événement, la liste de ceux qui sont disponibles correspond bien à ceux qui sont inscrits en base de données. La communication avec cette dernière fonctionne donc bien :



Lorsque je valide la création de mon événement, un message flash apparaît pour me confirmer sa création. Le code http 200 a donc bien été envoyé par le back, et l'inscription en base de données doit être effectuée :



Côté base de données, je vais vérifier que l'inscription a été effectuée. Voilà les données attendues :

- id : id auto incrémenté
- game_id : 5293 (id du jeu Scythe)
- user_id : 3433 (id du compte user créé pour le test)
- name : Test Event
- description : Test for freedom
- date_time : 2022-01-26
- entrants_numbers : 6
- created_at : la date et l'heure à laquelle j'ai créé l'event (+/- 11h55)
- updated_at : null
- area : 44
- status : 0 (représente le statut "ouvert" pour l'événement)
- total_users_validated : 1 (pour l'instant, je suis seul)

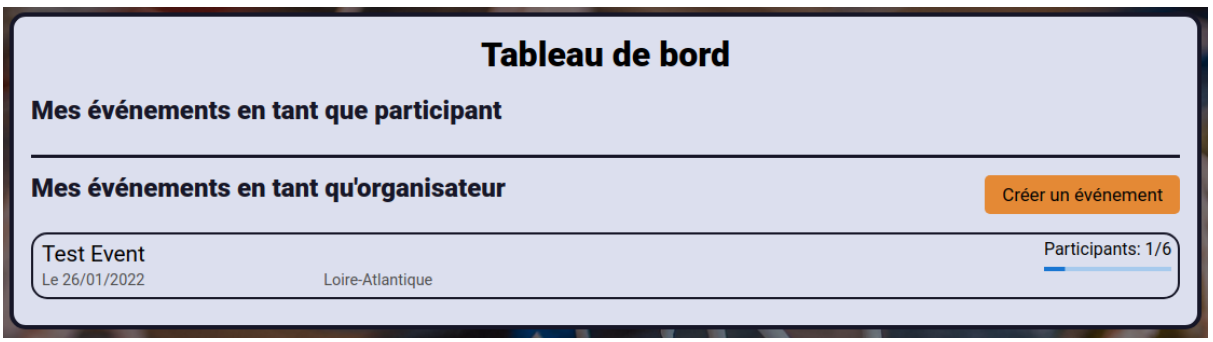
Voilà ce qui est trouvé dans phpMyAdmin :

id 1	game_id	user_id	name	description	date_time	entrants_numbers	created_at (DC2Type:datetime_immutable)	updated_at (DC2Type:datetime_immutable)	area	status	total_users_validated
10601	5293	3433	Test Event	Test for freedom	2022-01-26 00:00:00	6	2022-01-21 11:52:48	*NULL*	44	0	1

Je constate que les données attendues sont valides et conformes à mes attentes.
Il ne me reste plus qu'à aller confirmer la création de l'event côté vues.

Sur mon dashboard, j'attends qu'il soit affiché avec :

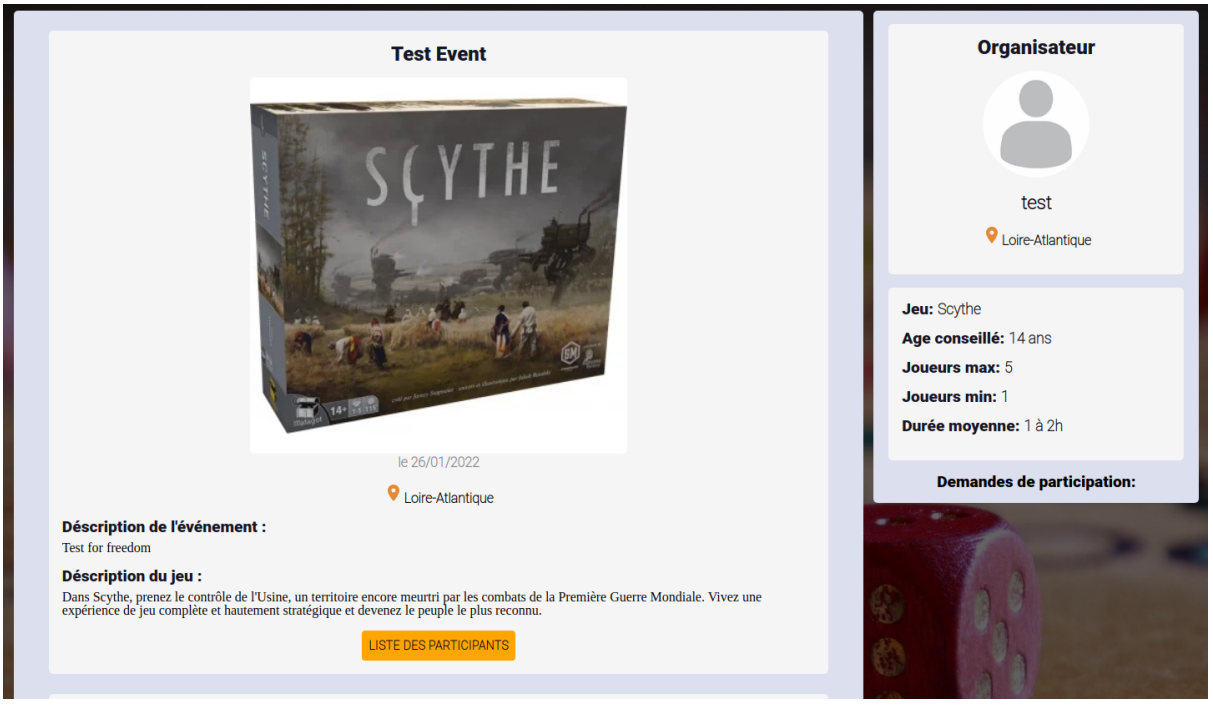
- nom : Test Event
- Date : 26/01/2022
- Département : Loire-Atlantique
- Nombre de participant actuel/le nombre de participants total : 1/6



Sur mon dashboard, je constate que l'événement que je viens de créer est correctement affiché.

Lorsque je clique sur l'événement, je devrais avoir une page qui me l'affiche, avec en plus des informations du dashboard :

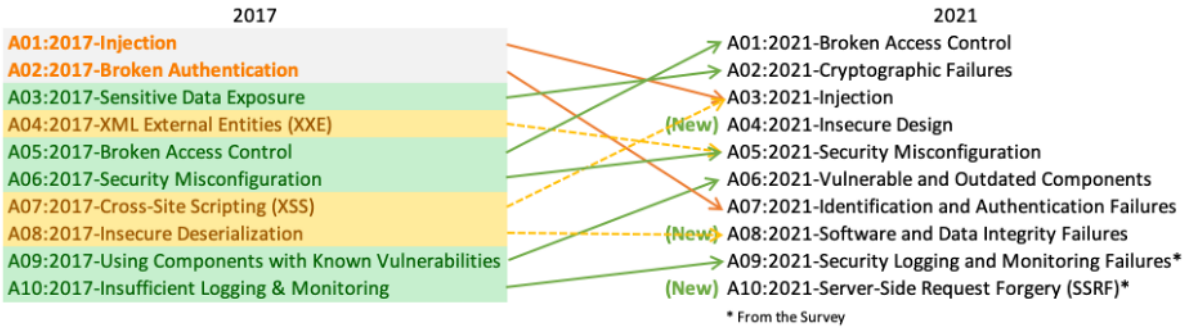
- Mon profil utilisateur : test
- L'image du jeu : Scythe
- La description du jeu tiré du site philibertnet.com
- Les informations du jeux
    - Jeu : Scythe
    - Age conseillé : 14 ans
    - Joueurs max : 5
    - Joueurs min : 1
    - Durée moyenne : 1 à 2h



Après tous ces tests, je peux valider la création de l'événement, car tous les affichages, valeurs, données et inscriptions en base de données ont fonctionné comme attendu initialement.

Veille technologique sur la sécurité

Pour ma veille technologique sur la sécurité, j'ai sur les conseils d'un ami consulté le site owasp.org (cf. références 10), qui fait un classement des 10 failles critiques de sécurité les plus répandues et utilisées, mise régulièrement à jour. La dernière liste date de 2021. Ce classement est tiré d'une analyse de plus de 500 000 applications, puis d'un travail de classement complexe. 8 failles du top 10 viennent de ces analyses de données, et 2 viennent d'un forum de discussion d'experts du développement et de la sécurité, afin de partager aussi les failles majeures actuelles qui n'ont pas eu le temps d'être collectées et donc mises à jour dans leur base de données.



Pour le backend, nous utilisons Symfony, qui est reconnu comme un framework robuste en termes de sécurité. Son **firewall** proprement configuré permet une grande sécurité sur l'accès des urls sensibles, et sur l'authentification. L'utilisation de Symfony 5.4 est notre premier choix de sécurité, puisque la *mise à niveau* contre les **CVE** (Common Vulnerabilities and Exposures) est suivie de près par *SensioLabs* et aussi par les développeurs des composants tiers.

Nous avons utilisé une liste d'**access control** plutôt stricte, qui l'a été d'ailleurs trop au début. Nous avons commencé par tout verrouiller avant d'ouvrir les droits au compte goutte selon les besoins d'accessibilité.

Pour l'authentification, comme nous passons par API pour communiquer avec le front, nous avons utilisé un **JWT** (Json Web Token - cf. références 13) afin de s'assurer de la validité de la connexion utilisateur et de ses droits. La durée de vie n'a pas été modifiée, elle est restée par défaut (3600 secondes // 1h), même si nous avons hésité à l'augmenter à une durée significative pendant que nous étions en train de travailler (particulièrement, lorsque nous testions pendant plusieurs heures les routes API). Mais le risque d'oublier de le reconfigurer a été jugé plus important que le confort de nos équipes.

Nous n'avons pas eu à utiliser de hiérarchisation des rôles, car nous n'avions que 2 rôles. Si nous développons le site en implémentant les idées d'évolution du site, plusieurs nouveaux rôles y sont prévus, et le **role_hierarchy** sera utilisé.

Toujours dans la configuration du firewall, le **security-bundle** de Symfony a configuré les **providers** lorsque nous l'avons appliqué aux entités User.

Pour le *hasher* de mot de passe, nous avons laissé celui choisi par défaut par Symfony (actuellement le **Bcrypt** - cf. références 11 -). En faisant nos recherches, nous avons constaté que pour des raisons d'évitement de confusion, le terme "encode" et toutes ses déclinaisons allaient être *deprecated* lors de la montée en version sur **Symfony 6.0**, remplacé par le mot "hash" communément adopté par la communauté des développeurs (cf. références 12). Nous avons donc utilisé la nouvelle interface proposée : au lieu d'utiliser la UserPasswordEncoderInterface, nous avons utilisé la UserPasswordHasherInterface.

Pour éviter le vol de session sur la partie backend, nous avons ajouté un **token csrf** a chaque action autre que GET. Ce jeton a usage unique permet de lier l'input à la requête, pour être certain qu'elle est désirée et ne vient pas de l'extérieur via un script (requête forgée).

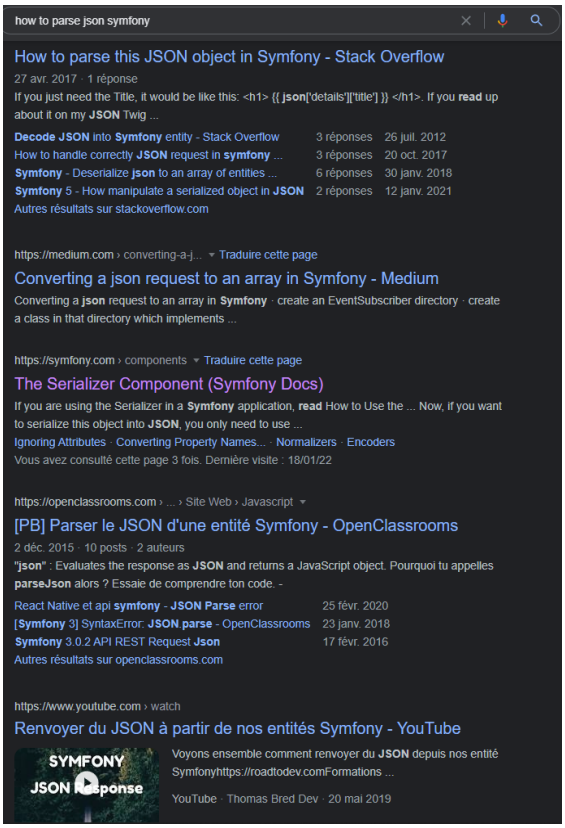
La protection contre les injections SQL de Symfony est solide, et **Doctrine** s'occupe de beaucoup de choses, dont l'échappement des données et l'encodage (Par exemple, en SQL, " -- " est le début d'un commentaire. Sans protection contre les injections, cela permet de mettre tout ce qui suit en facultatif, puisque mis en commentaire. La vérification d'un mot de passe peut donc sauter). Mais pour plus de protection face au XSS (Cross Site Scripting), nous avons ajouté des **constraints** à tous nos formulaires. Que ce soit des données du front ou celle du backOffice, les données transitent toutes par des contraintes que nous avons ajoutées à tous nos champs.

Symfony fournit aussi une gestion des headers **CSP** (Content Security Policy) avec **HttpFoundation**, qui maîtrise les instructions faites au navigateur.
Nous avons aussi installé le bundle nelmio pour ajouter au headers une autre couche de protection pour gérer les **CORS** (Cross-origin resource sharing), afin d'ouvrir l'accès à notre back à des échanges de données provenant d'autres plateformes de façon contrôlée.

Pour s'assurer de la sécurité vis-à-vis de l'équipe, nous avons utilisé des **clés SSH** pour nos connexions à *gitHub*, et pour l'accès au serveur, nous avons créé une clé **.pem**.

Difficultés rencontrées

Lors de notre mise en place de communication front/back via API, j'ai eu besoin de comprendre comment le format json allait être interprété, et comment les objets que j'y mettais allait être manipulé. Je me suis rappelé que nous utilisons la fonction JSON.parse() en JS. J'ai donc utilisé ce mot-clé dans une recherche google :



"how to parse json symfony"

J'ai choisi de lire la documentation de symfony.com, qui est la documentation officielle du framework et qui est mise à jour très régulièrement en suivant la montée des versions. Je me suis gardé les topics de stackoverflow pour plus tard, si jamais mes soucis persistaient.

La documentation m'a fourni une explication sur la façon dont le json est traité par Symfony et m'a permis de l'appliquer à notre projet.

Extrait site anglophone et traduction

“The Serializer component is meant to be used to turn objects into a specific format (XML, JSON, YAML, ...) and the other way around.

In order to do so, the Serializer component follows the following schema.

[...image…]

As you can see in the picture above, an array is used as an intermediary between objects and serialized contents. This way, encoders will only deal with turning specific formats into arrays and vice versa. The same way, Normalizers will deal with turning specific objects into arrays and vice versa.

Serialization is a complex topic. This component may not cover all your use cases out of the box, but it can be useful for developing tools to serialize and deserialize your objects.

Installation:
```
$ composer require symfony/serializer
```

If you install this component outside of a Symfony application, you must require the vendor/autoload.php file in your code to enable the class autoloading mechanism provided by Composer. Read this article for more details.

To use the ObjectNormalizer, the PropertyAccess component must also be installed.

Usage
This article explains the philosophy of the Serializer and gets you familiar with the concepts of normalizers and encoders. The code examples assume that you use the Serializer as an independent component. If you are using the Serializer in a Symfony application, read How to Use the Serializer after you finish this article.

To use the Serializer component, set up the Serializer specifying which encoders and normalizer are going to be available:

```
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
use Symfony\Component\Serializer\Serializer;

$encoders = [new XmlEncoder(), new JsonEncoder()];
```

```
$normalizers = [new ObjectNormalizer()];

$serializer = new Serializer($normalizers, $encoders);
```

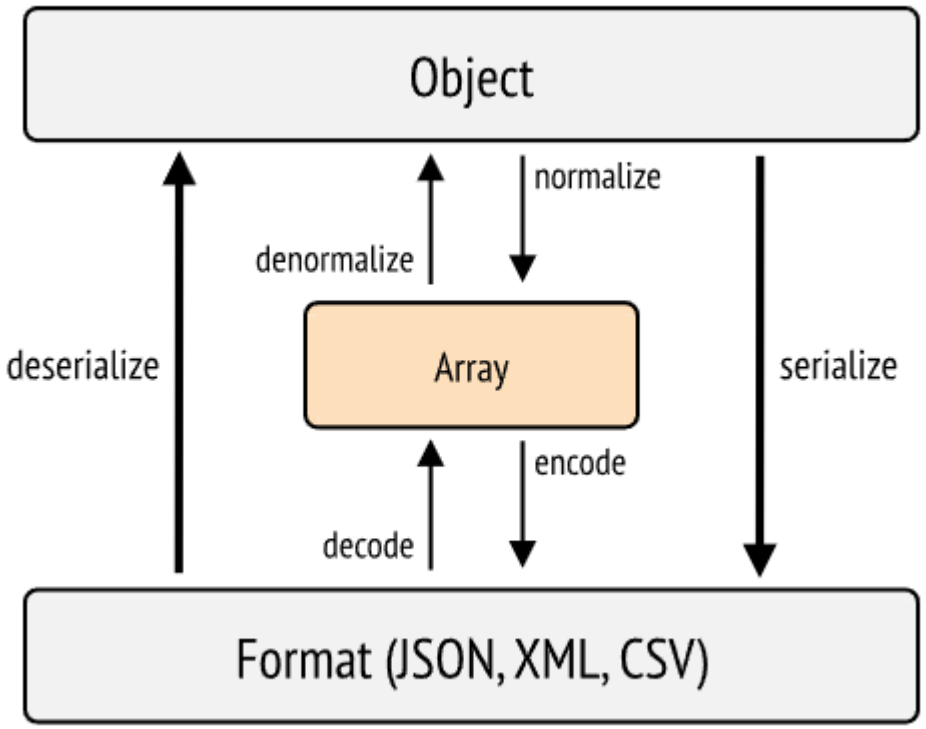
The preferred normalizer is the ObjectNormalizer, but other normalizers are available. All the examples shown below use the ObjectNormalizer.

”

Traduction

Le composant Serializer est destiné à être utilisé pour transformer des objets en un format spécifique (XML, JSON, YAML, …) et inversement.

Pour procéder à cela, le composant Serializer suit le schéma suivant :



Comme vous pouvez le voir sur l’image au-dessus, un tableau est utilisé en intermédiaire entre les objets et le contenu “sérialisé”.
De cette façon, les "encodeurs" auront uniquement à transformer les formats spécifiques en array, et vice versa. De la même façon, les "normalisateurs" auront uniquement à transformer des objets en tableaux et vice-versa.

Installation :

$ composer require symfony/serializer

[!] Si vous installez ce composant en dehors de l'utilisation de Symfony, le fichier vendor/autoload.php est nécessaire dans votre dossier afin d'utiliser la classe de la mécanique d'auto chargement fournie par Composer. Lisez cet article pour plus d'informations [lien].
[/!]

Pour utiliser ObjectNormalizer, le composant PropertyAccess doit aussi être installé.

Usage :

[!] Cet article explique la philosophie derrière Serializer, et vous permet de mieux appréhender les concepts d'encodage et de normalisation. Le code proposé en exemple part du principe que vous utilisez le composant Serializer comme un composant indépendant. Si vous utilisez Serializer dans une application Symfony, lisez "Comment utiliser Serializer" [lien] après avoir terminé cet article.

Pour utiliser le composant Serializer, mettez en place les spécificités des encodeurs et normalisateurs que Serializer va utiliser :

```
use Symfony\Component\Serializer\Encoder\JsonEncoder;
use Symfony\Component\Serializer\Encoder\XmlEncoder;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
use Symfony\Component\Serializer\Serializer;

$encoders = [new XmlEncoder(), new JsonEncoder()];
$normalizers = [new ObjectNormalizer()];

$serializer = new Serializer($normalizers, $encoders);
```

On utilise de préférence le normalisateur ObjectNormalizer, mais d'autres normalisateurs sont disponibles. Tous les exemples montrés ci-dessous utilisent ObjectNormilizer.

Conclusion

Ce projet a été passionnant à réaliser, et très enrichissant d'un point de vue technique, organisationnel et humain.

Beaucoup de problèmes ont été rencontrés, mais le résultat final valait la peine de relever les défis que nous avons eu à surmonter. Ayant travaillé sur la partie back du projet, j'ai tiré beaucoup de satisfaction à voir notre travail payer lorsque les premières vues dynamiques nous sont apparues, car pendant plusieurs jours nous n'avons vu que du code, des analyses de nos bases de données, et les résultats de tests de nos routes API.

Le déploiement n'a pas été évident, et a aussi entraîné des erreurs que nous avons eu à fixer. J'avais déjà entendu des blagues sur le "Pourtant ça marche en local", mais là ça a été concrétisé. Par exemple, notre script de scraping ne fonctionnait pas lorsque nous le lançions sur le serveur, alors qu'en local, nous n'avions aucun souci. La solution est venue d'une idée après avoir réfléchi sur la différence entre le serveur et le local. Ma conclusion a été qu'il devait y avoir un souci de nombre de requêtes par secondes, et j'ai donc essayé d'ajouter la fonction sleep() dans mon script pour temporiser entre chaque page récupérée. Ce qui a fonctionné.

Le travail en équipe n'a pas toujours été simple non plus, mais grâce à beaucoup de communication, nous avons réussi à nous entendre et à nous coordonner. Travailler en méthode agile y a beaucoup joué, et j'ai constaté la puissance d'organisation de ces méthodes. Avancée régulière, satisfaction régulière, et communication régulière font que la progression globale n'est pas ralentie, et lorsqu'il y a des difficultés ou des divergences, elles sont corrigées/rectifiées rapidement.

D'un point de vue personnel, la phase de conception du projet (sprint 0) a été la partie du projet qui m'a le plus plu et le plus excité. Et rétrospectivement, je pense que c'est la partie du projet la plus importante, et donc à laquelle il faut porter le plus d'attention. Quand toute l'équipe a la même vision, les mêmes supports, la même documentation, la partie code est clairement plus simple et agréable, et les objectifs sont plus facilement réalisés.

Tous ces points m'ont énormément formés, et m'ont surtout confirmé que développeur web était le métier que je voulais faire. Chaque point a été passionnant et prenant, et la satisfaction que j'en ai tiré a été énorme. Cet univers est un puits infini de connaissances, et j'ai hâte de pouvoir continuer à les engranger.

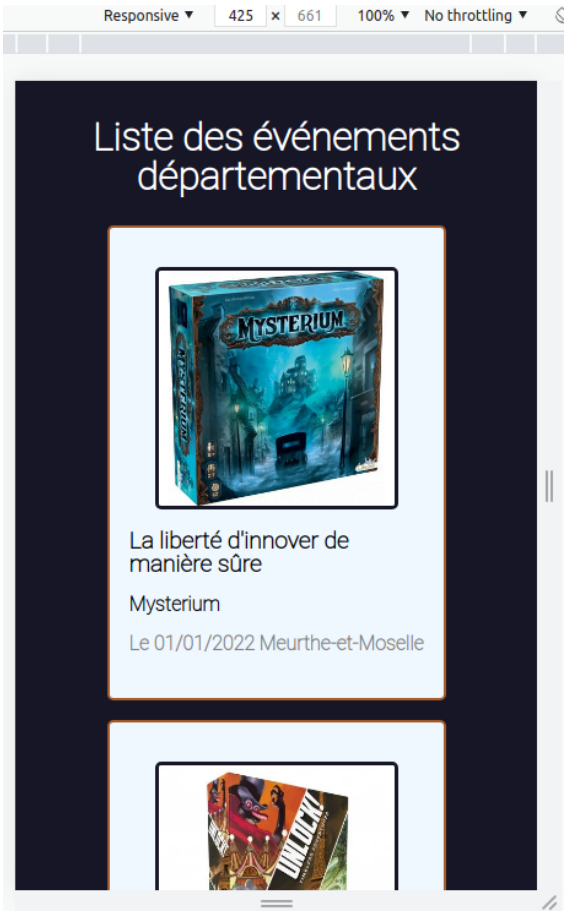
Annexes

Vues du site BlaBlaQuest & backOffice

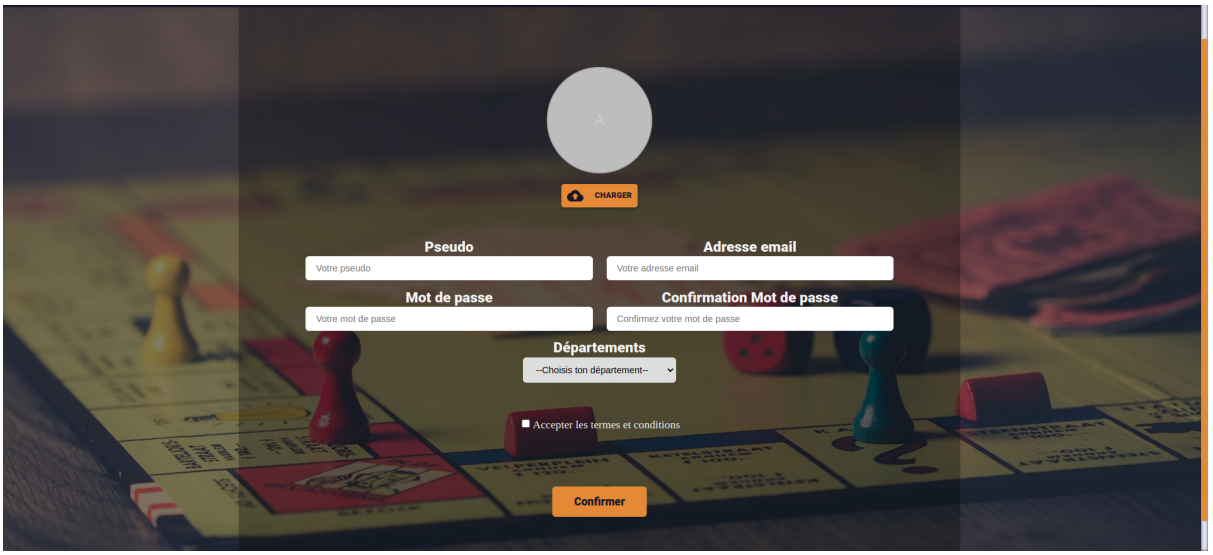
(Home page map - desktop)

(Home page map - mobile)

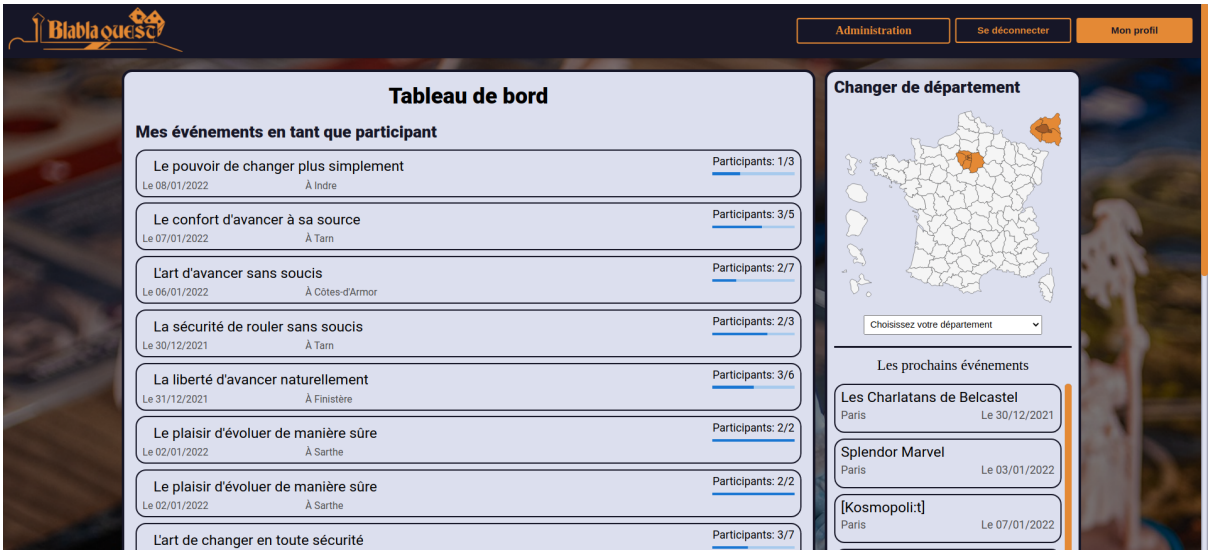
(Home page for events)



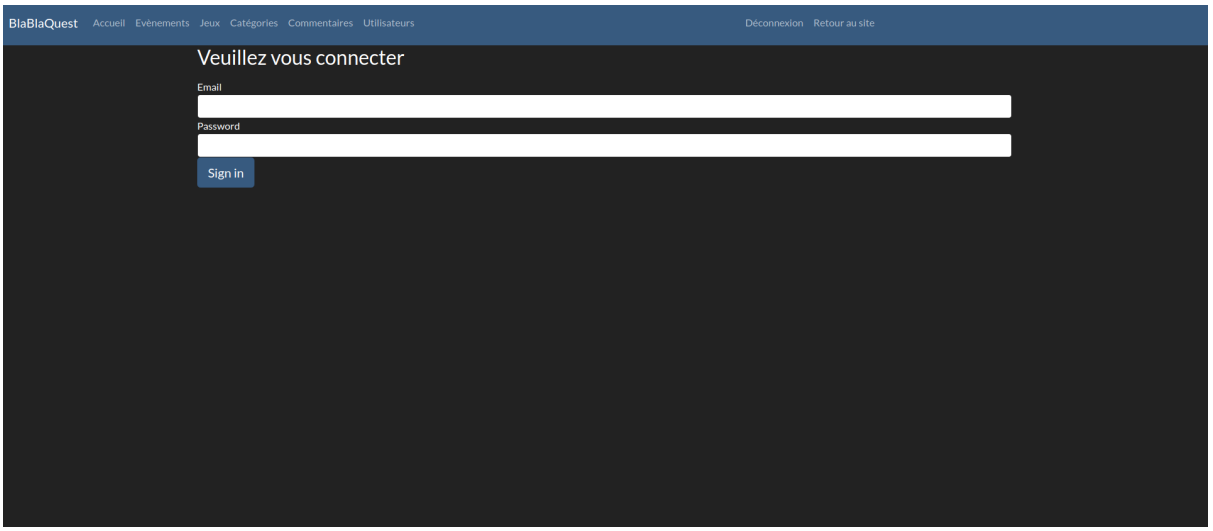
(Home page for events - mobile)



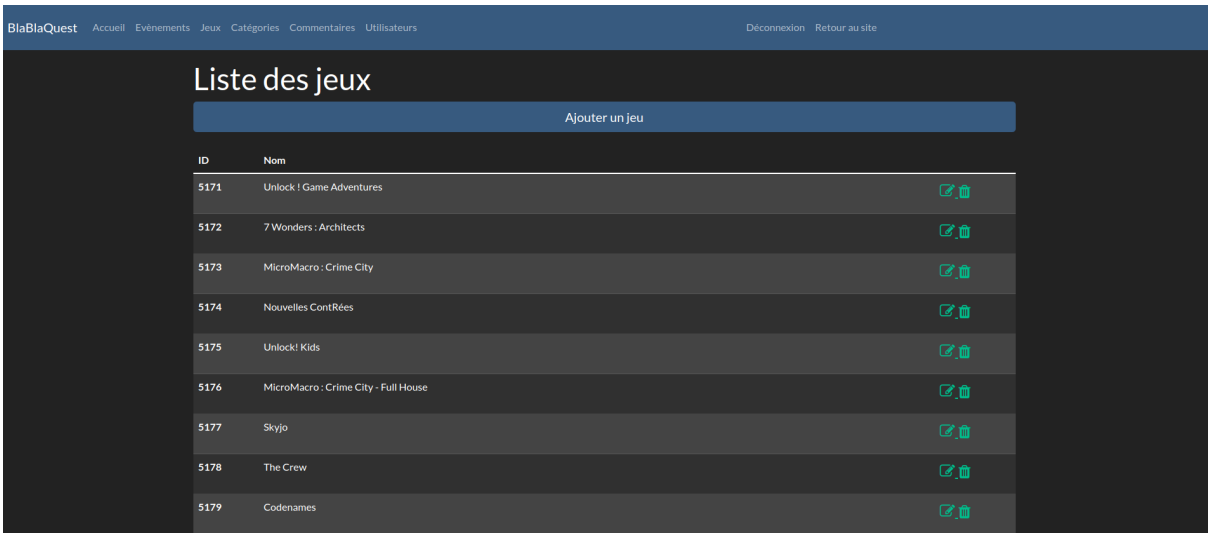
(Logon page)



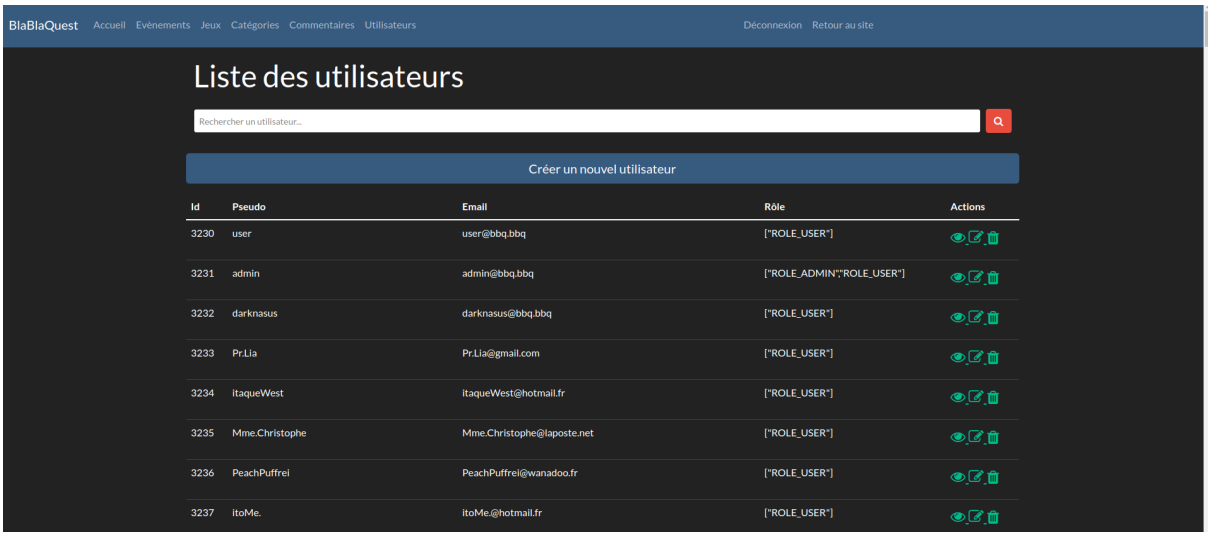
(Dashboard page)



(Login backOffice page)

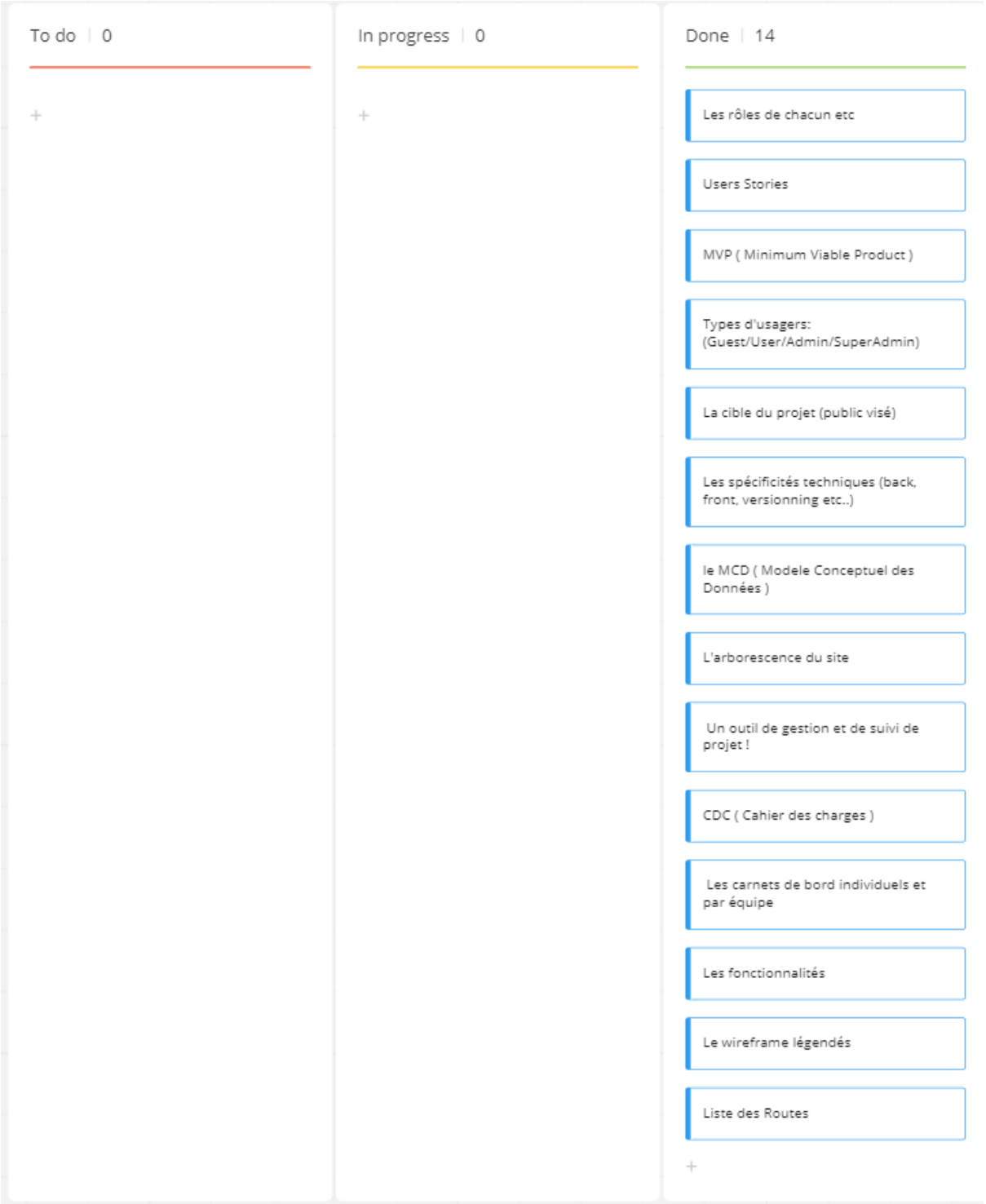


(BackOffice page for Games management)

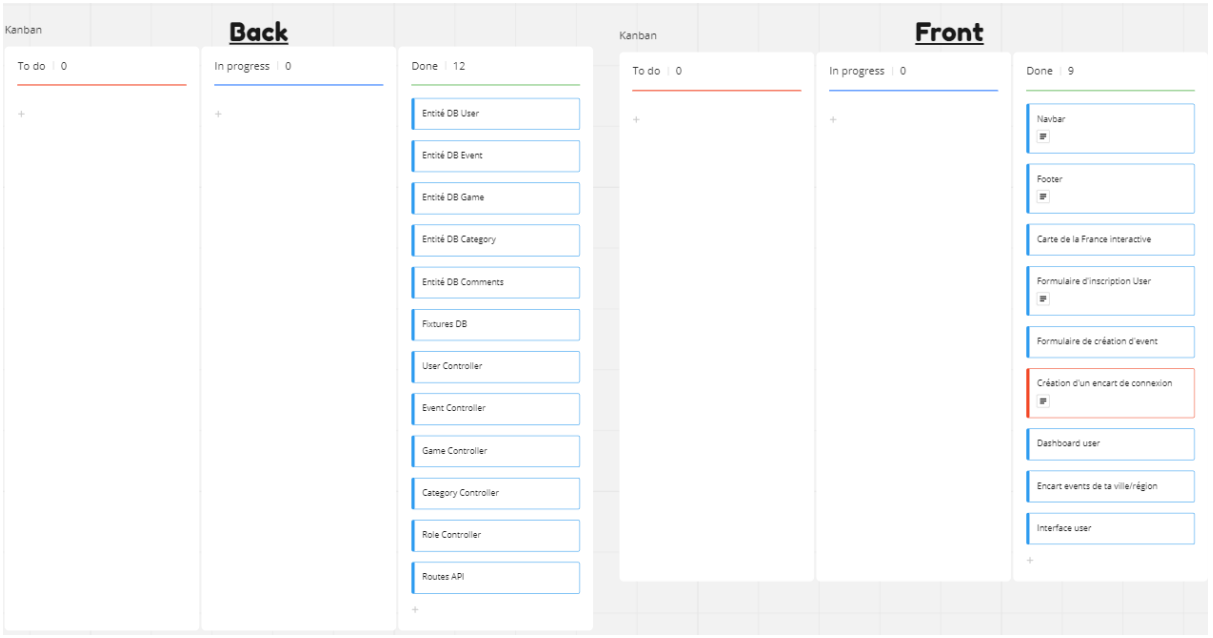


(BackOffice page for users with searchBar)

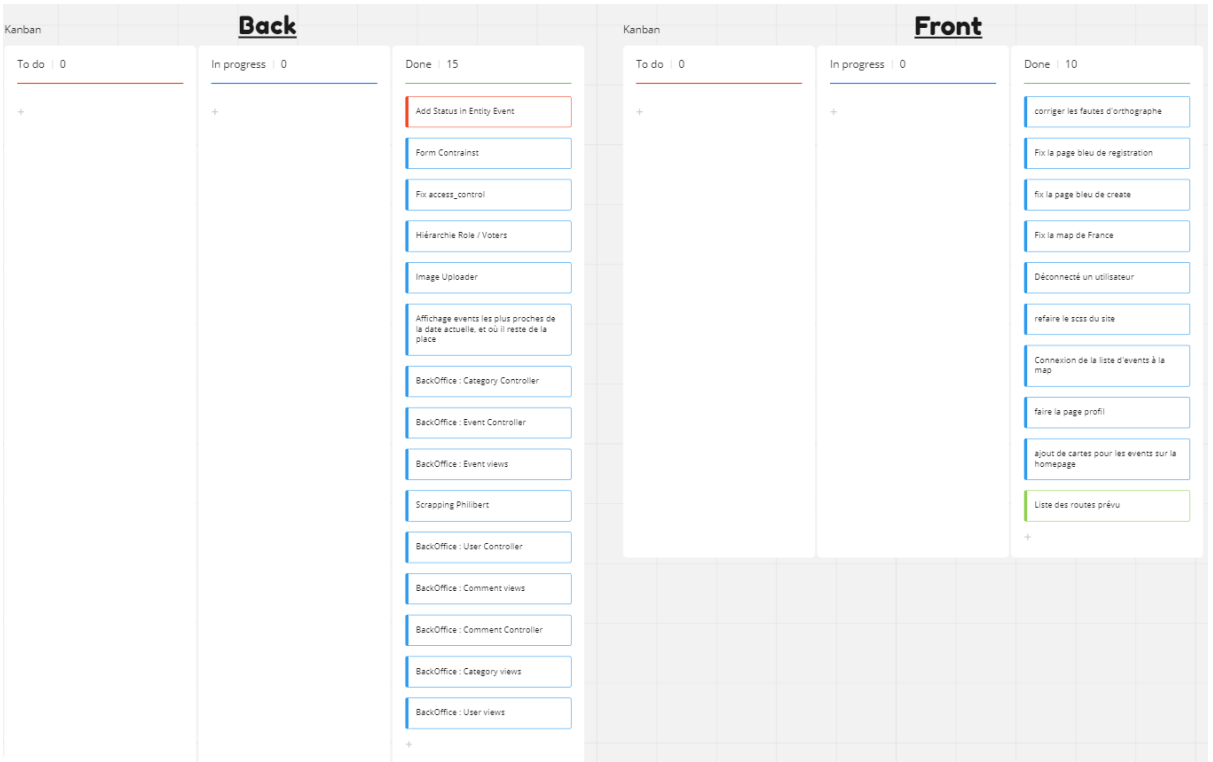
Kanban des sprints



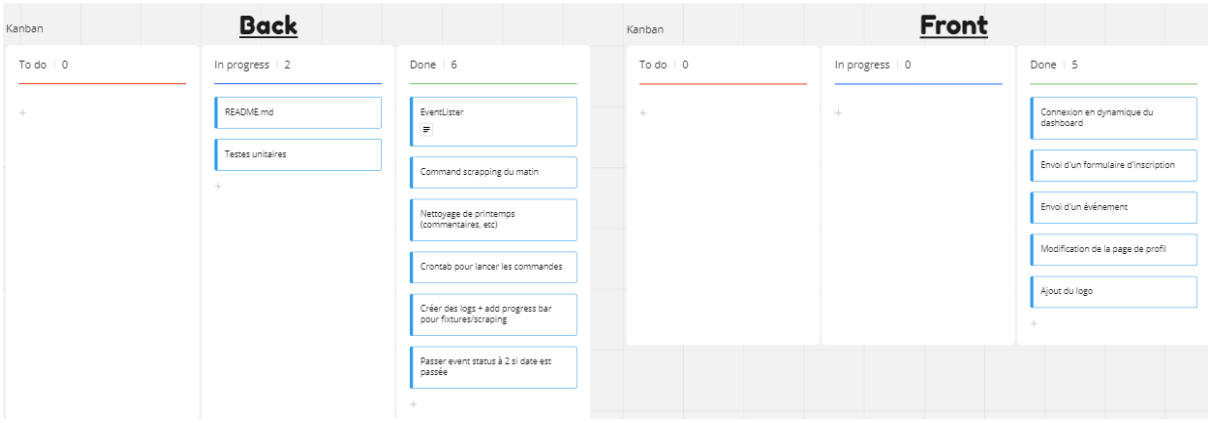
(Sprint 0)



(Sprint 1)

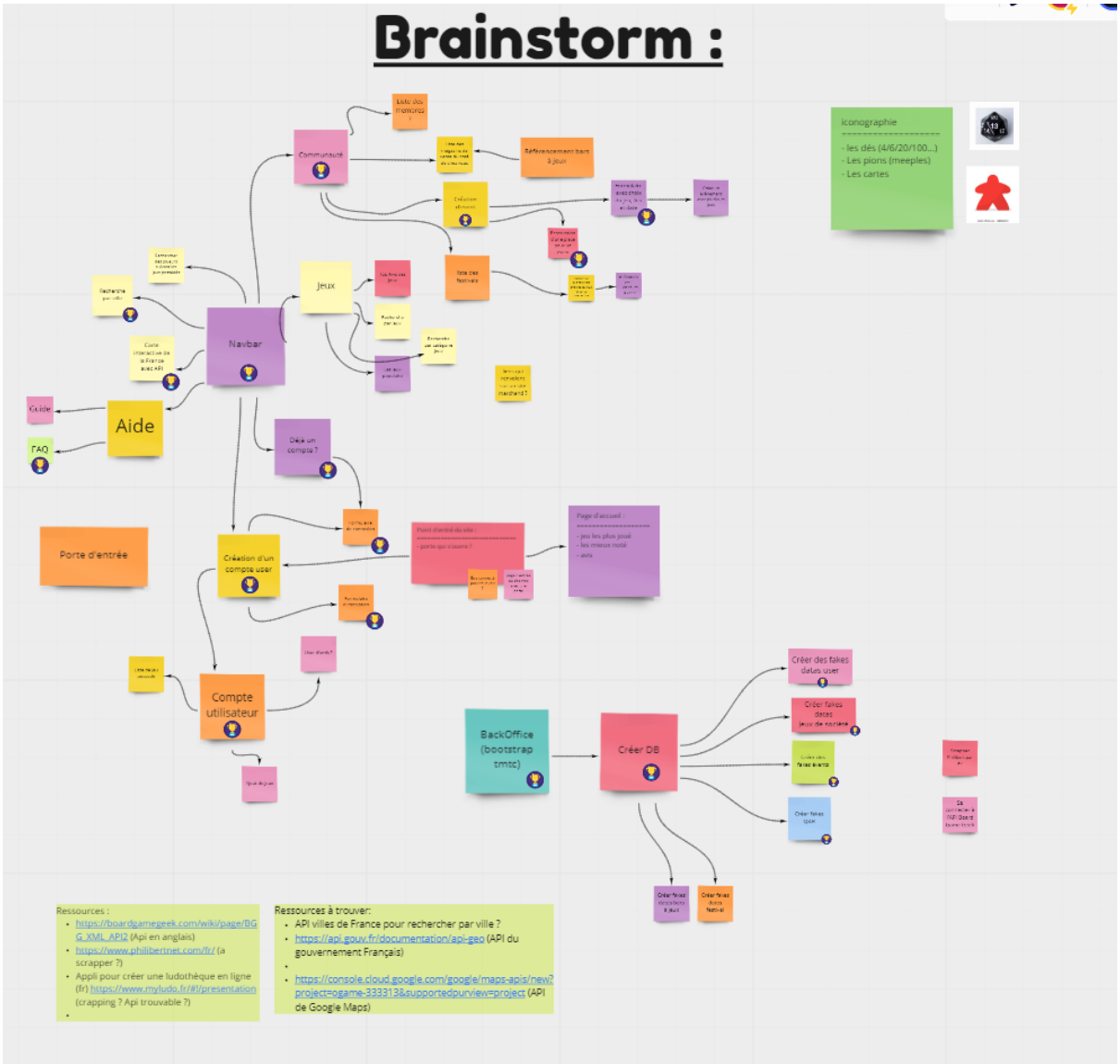


(Sprint 2)



(Sprint 3)

Archive brainstorm



Références

1. Méthode Scrum : https://scrumguides.org/scrum-guide.html
2. Méthode Merise : https://www.christian-roze.fr/for/formerise.pdf
3. Composant Serializer de Symfony : https://symfony.com/doc/current/components/serializer.html
4. Goutte : https://github.com/FriendsOfPHP/Goutte
5. Progress bar terminal : https://symfony.com/doc/current/components/console/helpers/progressbar.html
6. Fonction sleep() : https://www.php.net/manual/fr/function.sleep.php*
7. CSS Selector : https://symfony.com/doc/current/components/css_selector.html
8. domCrawler : https://symfony.com/doc/current/components/dom_crawler.html
9. fonction preg_split : https://www.php.net/manual/fr/function.preg-split.php
10. Top Ten OWASP : https://owasp.org/www-project-top-ten/
11. Hasher Symfony : https://symfony.com/doc/current/security/passwords.html#the-auto-hasher
12. Symfo 6.0 maj composant sécurité :https://symfony.com/blog/new-in-symfony-5-3-passwordhasher-component
13. Json Web Token : JWT : https://jwt.io/
14. CSP : https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
15. CORS : https://developer.mozilla.org/fr/docs/Glossary/CORS
16. Procédure stockée :https://sql.sh/cours/procedure-stockee