# Méthodologie

Il n'est pas nécessaire de connaître tous les détails d'implémentation dès le début d'un projet, mais une réflexion préliminaire peut aider à avoir une meilleure vision de l'architecture de l'application et ainsi mieux démarrer un projet.

Ci-dessous, les étapes de développement suggérées pour ce travail pratique.

## 1. Prendre connaissance du projet

* Lire et comprendre tous les récits utilisateurs qui sont à développer.
* Voir les maquettes qui doivent être créées.
* Prendre connaissance du code de départ et comprendre comment est structuré le projet.

**À noter que :**

* Le projet a été créé avec VueCli, avec comme options VueJs 2, tests unitaires Jest et VueRouter (projet similaire au lab 7).
* Le dossier ./tests/unit n'est pas utilisé dans ce projet. Les tests doivent se retrouver dans les dossiers \_\_tests\_\_ des unités à tester. Par exemple : ./src/components/\_\_tests\_\_, ./src/router/\_\_tests\_\_, ./src/views/\_\_tests\_\_ et ./src/services/\_\_tests\_\_.
* Les librairies suivantes ont été installées :
  + Bootstrap-vue (la configuration a été ajoutée dans main.js).
  + Axios
* Les librairies de développement suivantes ont été installées :
  + flushpromise
  + jest-when
  + json-server
  + axios-mock-adapter
  + concurrently
  + copy-files-from-to
* Un serveur backend local (API-REST), tel que présenté dans l'un des vidéos du lab 7, a été configuré dans le projet.
* Des fichiers de variables d'environnement, tel que présenté dans l'un des vidéos du lab 7, ont été créées afin de faciliter la maintenabilité du code.
* Afin d'externaliser tout le texte qu'on retrouve dans les interfaces, un plugin d'externalisation (./src/externalization/uiTextPlugin.js) a été ajouté au projet. Pour en savoir plus sur son utilisation voir [Bonnes pratiques : Externalisation](https://ymazieres.gitlab.io/W40-H22/bonnes-pratiques/externalisation/) => site de Yannick.

## 2. Environnement de développement

L'environnement de travail doit être 100% fonctionnel **avant de commencer à coder**. Pour s'en assurer, il faut être capable de...

### L'API REST

* Démarrer uniquement le serveur backend (l'API REST).
* Faire une requête à l'API REST et voir le résultat (pour des exemples, voir le fichier /requests/requests.http).

### Les tests

* Exécuter les tests unitaires en ligne de commande (ou dans VSCode).
* Débogueur les tests (à essayer avec un point d'arrêt dans example.spec.js).

### L'application

* Démarrer en une seule commande l'application et le serveur (API REST).
* Déboguer le code de l'application qui s'exécute dans un navigateur (à essayer avec un point d'arrêt à la ligne 112 du fichier HelloWorld.vue et en démarrant le débogueur de VSCode).

### Console.log

Éviter l'utilisation de console.log pour déboguer. Le débogueur est un outil essentiel que tout bon programmeur doit maîtriser, et ce, peu importe le langage de programmation.

## 3. Identifier les différents éléments du projet

Avant de commencer à coder, identifier quelles seront dans l'application :

* Les pages (composants de types views)
* Les composants
* Les routes
* Les services et les modules (par exemple un service qui permet de communiquer avec l'API REST, un module qui gère l'affichage des notifications, etc.).

Pour chacun des composants, répondre aux questions suivantes :

* Quel est son emplacement dans [l'arbre des composants](https://fr.vuejs.org/v2/guide/components.html#Organisation-des-composants) ?
* Quelles sont les données qui y seront stockées (data) ?
* Comment communique-t-il communique avec les autres composants (props et emits) ?

## 4. Squelette de l'application

Pour chacun des éléments identifiés à l'étape 3, créer un **fichier JavaScript** (coquille vide) et son **fichier de tests** correspondant. Exemples :

### HelloWorld.vue

<template>

<div>

<h1>Composant HelloWorld</h1>

</div>

</template>

<script>

export default {}

</script>

### HelloWorld.spec.js

import { shallowMount } from '@vue/test-utils'

import HelloWorld from '@/components/HelloWorld.vue'

describe('HelloWorld.vue', () => {

test('test non implémenté', async () => {

throw new Error('Test non implémenté.')

})

})

Mettre en place une navigation simple (pas de validation) entre les pages.

## 5. Écrire le code

Prendre un récit utilisateur :

* Identifier les composants, routes, services et modules utilisés par le récit.
* Écrire la description des tests en lien avec le récit à développer. Ce peut être, par exemple, sous forme de commentaires dans les fichiers de tests.
* Pour chacun des tests identifiés, écrire le code en TDD ou appliquer certains principes d'écriture de code.
* Faire une revue de code lorsque le récit (avec tests unitaires associés) sont terminés.

**Évaluation de la revue de code**

Le processus de revue de code indiqué **ci-dessous** n'est pas évalué dans ce TP, mais le sera dans le travail pratique final. Il est donc **fortement suggéré de faire des revues de code dans ce travail** afin de se familiariser avec le processus.

# Revue de code

## Bonnes pratiques

* Faire des revues de code régulièrement. Par exemple, dès qu'une fonctionnalité a été ajoutée ou qu'une partie de code est fonctionnelle. **Ne pas attendre à la fin du projet pour faire une revue de code**.
* Toujours commenter une revue de code. Les commentaires doivent être **constructifs** (points à améliorer dans le code). Si rien n’est à signaler, donner un commentaire positif (exemples : bravo, beau travail, code très propre, etc.).
* Les commentaires doivent être clairs, pertinents et sans ambiguïté.

## Questions à se poser

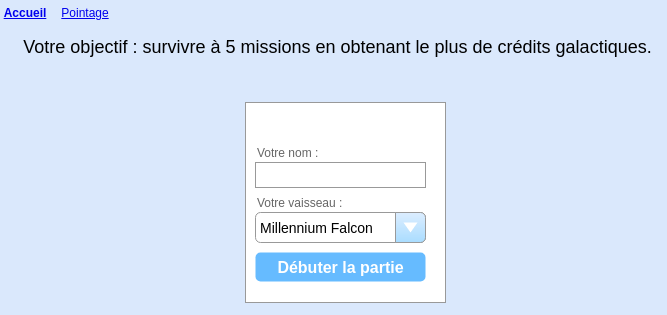
Liste non exhaustive de questions à se poser lors d'une revue de code :

* Le code est-il [propre et les standards](https://ymazieres.gitlab.io/W40-H22/bonnes-pratiques/code/) respectés ?
* Est-ce que [la sécurité](https://ymazieres.gitlab.io/W40-H22/bonnes-pratiques/securite/) a été considérée (données sensibles, connexion HTTPS, etc.) ?
* Est-ce que le code est facile à comprendre ou est-il trop complexe ?
* Est-ce que les exceptions sont gérées ?
* Est-ce qu'un algorithme peut être amélioré ?
* Est-ce que les noms des méthodes et fonctions représentent ce qu'elles font ?
* Est-ce que les tests unitaires sont fragiles ?
* Est-ce que les tests vérifient le bon comportement ?

# Maquettes

Ci-dessous les maquettes des trois pages de l'application. Il n'y a aucune obligation à suivre ce design, mais l'utilisation de l'application doit rester intuitif. Soyez créatif !

## Accueil



## En mission

Une image contenant texte

Description générée automatiquement

## Pointage

Une image contenant texte

Description générée automatiquement