Rapport d'audit VueJS flop!EDT (draft)

Mathieu Dartigues, expert VueJS

MAKINA CORPUS

version : Avril 2023 DRAFT

dernière révision : 14 Avril 2023

Introduction

Ce rapport est à destination de l'équipe du projet flop!EDT.

Il contient l'analyse effectuée début Avril 2023 du projet, principalement autour du code front afin d'évaluer l'usage de VueJS, et l'architecture mise en place avec le backend Django.

Cet audit a été réalisé sur un PC (i5 2 coeurs 4 threads, 8Go RAM, disque SSD) sous Linux Mint, navigateur Firefox 112.0, après avoir cloné le dépôt framagit et utilisé la branche `vue-vite`.

Le dernier commit de la branche étudiée est le `5ec61268da33fc000e3f9cef55f63ca9c87f7cb7`

Le backend a été instancié avec docker, et un backup de base a été fourni par Pablo Seban pour que je puisse effectuer mes essais / analyses.

Ce rapport a pour intérêt d'étudier le code VueJS et recense toute information qui m'a paru pertinente concernant le déroulé de l'audit, des recommandations au niveau du code ou de l'architecture.

Attention toutefois, le travail présenté ici est lié à ma compréhension personnelle du projet, du code, et des technologies utilisées. Il est tout à fait admissible que plusieurs de mes interprétations soient erronées, et par conséquent les préconisations corrélées.

Certains passages du rapport pointent sur des liens wikipédia ou de documentation. Ils sont écrits en *italique* et parfois en *`taille fixe`*.

dernière révision 14 Avril 2023

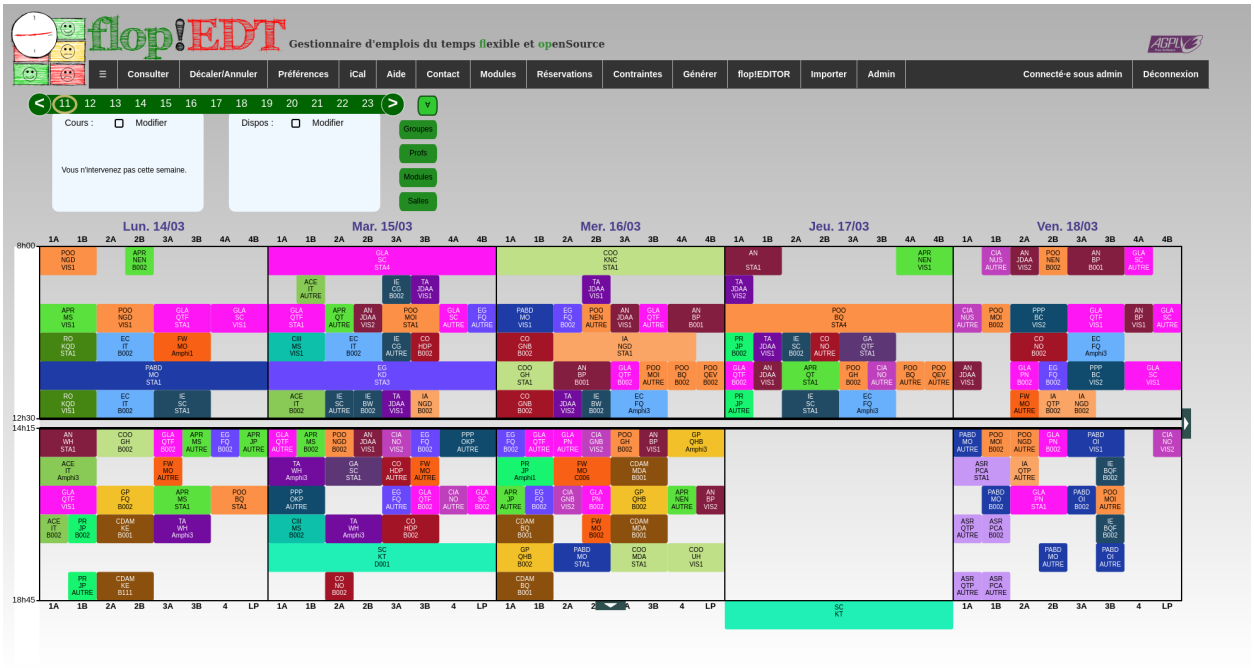

Table des matières

I Installation

Avant de commencer l'analyse du projet, j'ai procédé à l'installation.

Habitué des stacks docker, j'ai privilégié un build de container.

J'ai suivi le README.md en appliquant deux commandes dans le dossier projet :

```
make config
make build
```

À cet instant, la base de données est encore vide.

La commande `make init` charge un fichier `dump.json.bz2` qui semble problématique, avec une erreur de duplication de clé.

J'ai souhaité procéder à un import manuel des données via `./manage.py loaddata` à travers le docker.

Je me suis connecté avec `docker exec -it flopedt_backend_1 bash` :

```
root@aad8ce628a58:/code/FlOpEDT# ./manage.py loaddata ../dump.json.bz2
Traceback (most recent call last):
  File "/usr/local/lib/python3.7/site-packages/django/db/backends/utils.py", line 86, in
  ↪  _execute
    return self.cursor.execute(sql, params)
psycopg2.errors.UniqueViolation: duplicate key value violates unique constraint
↪  "django_content_type_app_label_model_76bd3d3b_uniq"
DETAIL:  Key (app_label, model)=(base, course) already exists.

...(stack trace)

django.db.utils.IntegrityError: Problem installing fixture
↪  '/code/FlOpEDT/../dump.json.bz2': Could not load contenttypes.ContentType(pk=6):
↪  duplicate key value violates unique constraint
↪  "django_content_type_app_label_model_76bd3d3b_uniq"
DETAIL:  Key (app_label, model)=(base, course) already exists.
```

Ce fichier semble définitivement provoquer une erreur de duplication.

J'ai ensuite procédé à l'import du fichier dump.json2.bz2, le format n'est pas reconnu.

Je pense qu'un renommage en `dump2.json.bz2` doit suffire pour l'importer. (non testé)

J'ai procédé ensuite à l'import du fichier `dump_anonyme.json.bz2`, qui a pris plusieurs dizaines de minutes (pour un fichier de 2.3 Mo). En dehors du fait que je ne savais pas si cela importait vraiment les données - j'ai toutefois pu observer une activité côté base de données avec des transactions et un volume de données croissant - il y a probablement quelque chose à vérifier concernant le temps d'importation. Un fichier de quelques méga octets ne doit pas prendre plusieurs minutes d'import. Pour l'heure, un message dans le README permettrait d'informer le développeur que cela est "normal".

Après fourniture d'un dump maîtrisé par Pablo, et plus rapide à importer, la base a été peuplée avec des données stabilisées.

En accédant à la base, j'ai pu découvrir la table `people_user` et tous les comptes utilisateurs. Ne connaissant pas les mots de passe de ces utilisateurs, j'ai procédé à la création d'un super user avec la commande `./manage.py createsuperuser` et définit un mot de passe basique "`passe`".

Puis j'ai fait un `UPDATE` en base de données sur l'ensemble des lignes de la table `people_user` pour utiliser le chiffrement obtenu en base :

```
update people_user set "password" =
↪  'pbkdf2_sha256$180000$f2peD7GrNlSW$MfCvDaDMG64SQ030iIMLMtAR7TvWjWfjIwfZew2eRDw=';
```

Je peux ainsi me connecter avec n'importe quel utilisateur.

Cette phase d'installation s'est clôturée avec le lancement du backend et du frontend, et une petite phase de debug (merci à Pablo pour son assistance) suite à une mise à jour d'un modèle de données Django sur la branche `vue-vite`.

II Utilisation / Découverte de l'outil

Cette phase de l'audit a consisté à faire un tour du propriétaire pour mieux comprendre les différents écrans et fonctionnalités offertes par l'application.

Plusieurs interfaces ont été essayées :

- http://localhost:8000 qui correspond au site généré par Django, avec une partie d3 et VueJS
- http://localhost:5173 qui correspond à la SPA VueJS assemblée par le bundler *`vite`*

Ces essais m'ont permis de prendre conscience de plusieurs notions métiers :

- les intervenants
- les salles
- les modules
- les groupes d'étudiants
- les cours, qui relient les 4 informations précédentes pour fabriquer l'emploi du temps

Concernant l'expérience d'usage sur la SPA VueJS, plusieurs pages sont pour l'heure "proxyfiées" depuis le backend Django.

Il me semble que l'authentification via session / cookie semble partagée entre les domaines, mais la page de login côté SPA ne fonctionne pas ? Mon cas d'usage est de passer par la page http://localhost:5173/login, mais je n'arrive pas à persister d'authentification.

En passant par http://localhost:8000, j'arrive bien à m'authentifier, et cette authent semble "fonctionner" côté http://localhost:5173 pour les appels à l'API ainsi que les pages qui sont proxyfiées. (le cookie de session et le jeton CSRF sont envoyés à chaque requête)

Une autre méthode de communication entre le backend et le frontend peut être avec un JSON Web Token (JWT). Ce jeton est récupéré après une première authentification, et est utilisé avec un header HTTP dédié (généralement `Authorization`) comme le cookie de session.

Ce jeton peut ensuite être persisté dans une zone de mémoire navigateur comme le `sessionStorage` ou `localStorage`.

III Analyse du code front VueJS

Cette section vise à présenter les résultats de l'analyse du code front.

Pour cela, une première phase a été d'analyser au regard des 4 pages présentes dans la SPA.

Puis, dans un deuxième temps, en analysant la structure du projet front.

III.A Analyse des pages de la SPA

En l'état du projet, j'ai pu accéder à 4 pages de la SPA :

- roomreservation
- contact
- home
- login

Chacune de ces pages dispose d'un fichier correspondant dans le répertoire `/views`.

III.A.1 Contact, Home, Login

Ces 3 pages sont assez simples et présentent une structure maîtrisée.

Attention cependant à essayer de respecter au mieux la sémantique HTML, en incluant les `input` dans une balise `form` pour la page `Login`.

III.A.2 Room Reservation

Cette page est gérée par le fichier `/views/RoomReservationView.vue`.

Ce composant fait parti des "gros" composants, avec +1000 lignes de code.

Les gros fichiers sont délicats à la maintenabilité ; il est souvent complexe pour un développeur de se retrouver dans un fichier aussi grand. Il y a probablement des simplifications à amener à ce fichier, en pensant modularité, en regroupant visuellement les portions de code qui traite des mêmes problématiques, pour les externaliser dans des composables ou dans les stores adéquats. Il peut être également pertinent d'ajouter des commentaires pour faciliter la compréhension du composant. Extraire des fonctions utilitaires pour les déplacer dans un fichier dédié dans le répertoire `/helpers` pourrait également alléger le code.

Si nous procédons à l'analyse du code, nous pouvons constater un nombre important d'imports de typage (~30), de variables réactives (~30), de computed (~30), et plusieurs méthodes qui semblent créer des composants/slots (`createRoomReservationSlot`, `createScheduledCourseSlot`).

Cette première lecture me laisse penser qu'il y a beaucoup de données gérées dans cette vue, qui concerne des cours, salles, réservations, dont plusieurs dérivent de données du store. Une question que nous devons nous poser est est ce que ces données sont au bon endroit, et ne mériteraient pas d'être dans un store dédié. Cela fait écho aux principes `SRP` (Single Responsibility Principle) et `SoC` (Separation of Concerns). Au delà du fait d'isoler et d'alléger les traitements de ces données, cela permettrait également d'asseoir une structure de données au sein des stores.

Concernant la création de slots, sans arriver à comprendre précisément l'origine de cette implémentation, je me permets de rappeler que le framework VueJS est fortement inspiré du pattern *MVVM Model > View, View > Model*. En VueJS, cela signifie que le modèle de données génère la vue (Model > View), et que la vue peut également impacter le modèle de données (View > Model) via la mécanique de réactivité et le two-way data binding. Pour faire le parallèle aux slots, je me demande si une partie du code TypeScript pourrait être remplacé par du code de template en utilisant les directives VueJS et en se basant sur le modèle de données.

Nous pouvons également constater la présence de plusieurs fonctions de traitement métier.

Prenons le cas de `reservationRemoveCurrentAndFutureSamePeriodicity`.

```
function reservationRemoveCurrentAndFutureSamePeriodicity(reservation: RoomReservation) {
  if (!reservation.periodicity || reservation.periodicity.periodicity.id < 0) {
    return
  }
  const periodicityId = reservation.periodicity.periodicity.id
  const reservationDate = new Date(reservation.date)
  // Remove all the future reservations of the same periodicity
  reservationRemoveFutureSamePeriodicity(reservation)
    // Remove the current reservation
    ?.then((_: any) => {
      deleteRoomReservation(reservation)
    })
    // Reduce the periodicity end date to the day before the current reservation
    .then((_: any) => {
      // Get the day before the reservation
      const dayBefore = new Date(reservationDate)
      dayBefore.setDate(dayBefore.getDate() - 1)

      // Get the periodicity data
      const periodicity = reservationPeriodicities.perId.value[periodicityId].periodicity
      if (periodicity.periodicity_type === '') {
        // Should never arrive here as an existing periodicity always has a type. Written
        ↪  for the type checks.
        return
      }
      // Match the api path with the periodicity type
      let apiCall
      switch (periodicity.periodicity_type) {
        case 'BM':
          apiCall = api.patch.reservationPeriodicityByMonth
          break
        case 'EM':
          apiCall = api.patch.reservationPeriodicityEachMonthSameDate
          break
        case 'BW':
          apiCall = api.patch.reservationPeriodicityByWeek
          break
      }
      // Finally apply the patch
      return apiCall(periodicity.id, { end: dayBefore.toISOString().split('T')[0] })
    })
    ?.then((_: any) => {
      updateRoomReservations(selectedDate.value)
      updateReservationPeriodicities()
    })
    .finally(closeReservationDeletionDialog)
}
```

Essayons de refactoriser cette fonction en favorisant l'usage de `async` / `await` en lieu et place des `Promise` :

```
async function reservationRemoveCurrentAndFutureSamePeriodicity (reservation:
↪  RoomReservation) {
  if (!reservation.periodicity || reservation.periodicity.periodicity.id < 0) {
    return
  }
  const periodicityId = reservation.periodicity.periodicity.id
  const reservationDate = new Date(reservation.date)

  try {

    // Remove all the future reservations of the same periodicity
    await reservationRemoveFutureSamePeriodicity(reservation)

    // Remove the current reservation
    await deleteRoomReservation(reservation)

    // Reduce the periodicity end date to the day before the current reservation
    // Get the day before the reservation
    const dayBefore = new Date(reservationDate)
    dayBefore.setDate(dayBefore.getDate() - 1)

    // Get the periodicity data
    const periodicity = reservationPeriodicities.perId.value[periodicityId].periodicity
    if (periodicity.periodicity_type === '') {
      // Should never arrive here as an existing periodicity always has a type. Written
      ↪  for the type checks.
      return
    }
    // Match the api path with the periodicity type
    let apiCall
    switch (periodicity.periodicity_type) {
      case 'BM':
        apiCall = api.patch.reservationPeriodicityByMonth
        break
      case 'EM':
        apiCall = api.patch.reservationPeriodicityEachMonthSameDate
        break
      case 'BW':
        apiCall = api.patch.reservationPeriodicityByWeek
        break
    }
    // Finally apply the patch
    await apiCall(periodicity.id, { end: dayBefore.toISOString().split('T')[0] })
    await updateRoomReservations(selectedDate.value)
    await updateReservationPeriodicities()
  } catch (error) {
    // do sthg with the error ?
  }
  closeReservationDeletionDialog()
}
```

À cet instant de la refactorisation, nous avons certes un nombre de ligne équivalent, mais nous pouvons mieux prendre conscience que cette fonction effectue des traitements en séquence.

Néanmoins, que se passe t'il si l'appel à `apiCall` renvoie une erreur ? Nous ne ferons pas les appels suivant à `updateRoomReservations` et `updateReservationPeriodicities`.

Cela semble faire plutôt l'objet d'une transaction au sens SQL.

Si nous patchons les réservations (par mois / semaine / autre), alors il faut également mettre à jour les réservations. L'un ne va pas sans l'autre. Si l'un échoue, l'autre ne doit pas être fait, mais surtout, nous devons revenir à l'état d'avant suppression : une erreur est survenue durant la transaction, ce n'est pas normal, nous ne pouvons pas rester dans un état instable / incohérent.

Il y a peut-être ici une anomalie en devenir par inconsistance de la base de données.

Le frontend ne doit pas se substituer au backend dans son rôle de gestionnaire métier. Le frontend est uniquement un point d'accès à l'information et aux processus métiers. Il sert de mise en forme d'une information, d'aggrégateur, de gestionnaire, de déclencheur de processus, mais il ne doit pas gérer le processus en lui-même. Cette responsabilité appartient au backend, qui est seul garant de l'exécution du processus. À lui de mettre les protections nécessaires et suffisantes (notamment les transactions) pour garantir l'état du système, et notamment la consistance de la base de données.

Nous avons à faire ici à un traitement métier, un cas d'usage, qui doit être traité entièrement côté backend pour gérer une vraie transaction en base de données, pouvoir faire un `COMMIT` ou un `ROLLBACK` en cas de besoin, et ainsi garantir la consistance de la base.

III.B Structure du projet front

En rentrant plus en détail dans le code, et l'arborescence fichier, nous pouvons constater une architecture découpée selon les briques utilisées :

- un routeur avec sa définition dans le répertoire `/router`
- plusieurs stores avec leur définition dans le répertoire `/stores`
- des composants définis dans le répertoire `/components`
- un composable défini dans le répertoire `/composables`
- la gestion de l'internationalisation à travers les répertoires `/i18n` et`/locales`
- des types définis dans le répertoire `/ts`
- les pages de l'application dans le répertoire `/views`

Cette architecture suit les standards des applications VueJS, ce qui favorise une bonne compréhension à l'abordage du projet.

L'usage de TypeScript est aussi un très bon point pour favoriser l'auto complétion dans des IDE tels que VS Code, IntelliJ ou des éditeurs tels que Sublime Text. Cela réduit également les erreurs de codage, et favorise l'apprentissage des objets métiers.

Les prochains paragraphes vont s'attacher à préciser certains points sur l'ensemble des briques.

III.B.1 Routeur

III.B.1.a Locale en début de path

Contrairement à l'application front présente dans l'app Django, les chemins des routes n'intègrent pas la locale en début de chemin.

Le fait de privilégier les routes de type `:locale/path` - au lieu de glisser la locale dans le path - permet de fabriquer des routes enfants, et de réagir plus facilement au changement de locale.

Cela peut donner des routes déclarées plutôt comme suit :

```
const routes = [{
  path: '/:locale',
  children: [{
    path: '/roomreservation/:dept?',
    ...
  }, {
    path: '/contact/:dept?',
    ...
  }, {
    ...
  }]
}, {
  ...
}]
```

Et ce qui paraît être la norme côté Django avec le routage actuel.

III.B.1.b Récupération des paramètres nommés

Dans le fichier `stores/department.ts`, au sein de la fonction `getDepartmentFromURL`, je pense qu'il est question de récupérer le département de l'utilisateur, à travers la route courante.

D'après la configuration des routes, un paramètre nommé `dept` est déjà présent.

Si l'on se réfère à la documentation sur les *paramètres de route*, il est possible d'accéder à la valeur du département via la `$route.params.dept`.

Avec le composable `useRoute` déjà utilisé, voici ce que pourrait donner le code :

```
function getDepartmentFromURL() {
  const route = useRoute()
  return route.params.dept
}
```

Si l'objectif est également de réagir à ce changement, il est tout à fait possible de modifier une variable réactive du store `department` avec des *navigation guard*.

L'usage de `.params` peut également être mis en place dans le navigation guard `beforeEach` présent dans le fichier `router/index.ts` pour connaître la locale choisie par l'utilisateur.

Par exemple, le code

```
    availableLocales.forEach((currentLocale: string) => {
        to.fullPath.split('/').forEach(arg => {
            if(arg.includes(currentLocale) && arg.length === currentLocale.length) {
                locale.value = currentLocale
            }
        })
    })
```

pourrait se transformer en

```
    if (availableLocales.indexOf(to.params.locale) > -1) {
      locale.value = to.params.locale
    }
```

Ce code paraît plus concis et moins complexe pour le navigateur, probablement pour le développeur également. Plus rapide à l'exécution donc, et probablement plus facilement maintenable.

III.B.2 Store

Un store est déjà présent, et son objectif est de centraliser la donnée et ses traitements associés pour la rendre accessible à un plus grand nombre.

Un découpage pertinent doit être effectué en fonction du métier.

À ce jour, j'observe un découpage pour :

- l'authentification, qui va probablement évoluer dans le temps, en intégrant des fonctions de login / logout
- les objets métiers (départements, salle, réservation, cours planifiés)
- un store générique et utilisable selon les objets métiers (`store.ts` avec `StoreAction`)

III.B.2.a Interdépendance

À la lecture du code, les stores semblent limités à du CRUD.

Il est possible de pousser un peu plus loin leur usage, avec - pour reprendre le cas de la page `RoomReservationView` - le calcul de `computed` (`getters` au sens Pinia) lié à des infos d'autres stores.

Par exemple, dans le hook `onMounted` de ce même composant, la variable `selectedDepartement` est affectée avec la valeur du département courant (ligne 1022 : `selectedDepartment.value = departmentStore.getCurrentDepartment`), puis est utilisée pour `selectedDepartments`, en utilisant les départements préalablement fetchés.

Ces deux variables sont parfaitement candidates pour migrer dans le store `department`, tout en laissant `RoomReservationView` muter la propriété `selectedDepartment` du store.

Double avantage : on regroupe ce qui parle de la même chose (les départements) et on simplifie le code d'un composant complexe.

La réflexion peut être étendue à d'autres variables, comme par exemple `scheduledCourses`.

Globalement, une réflexion doit être menée sur la structuration de ces stores et leurs interdépendances. Certains stores peuvent être limités à de simples CRUD, pour des glossaires par exemple. C'est déjà le cas pour les départements, et cela peut être le cas pour les intervenants, les salles, les modules. Pour des objets métiers plus élaborés, comme les réservations, l'usage d'autres stores peut s'avérer bénéfique pour disposer de la réactivité tout en isolant les données dans un store spécifique.

III.B.2.b Getter / State

Certaines computed (getters au sens Pinia) du store `department` renvoient le contenu d'une propriété du state.

C'est le cas de `getCurrentDepartment`, `getAllDepartmentsFetched`.

Sauf erreur, il est préférable d'utiliser directement la variable du state, `currentDepartment` ou `departments` dans les deux cas précédent, plutôt que d'écrire des getters dédiés : on réduit le code, et on bénéficie de la réactivité.

Pour rappel, la fonction *`storeToRefs`* de Pinia permet de déstructurer un store tout en conservant la réactivité.

Tout comme un getter qui renvoie une donnée du state, une fonction "setter" comme `setCurrentDepartement` qui affecte directement la donnée du state `currentDepartment` semble peu pertinente. La variable `currentDepartment` fait partie du state du store, elle est donc réactive. En l'exposant dans l'objet de retour, elle pourra être accessible dans les composants "smart" qui en ont besoin, et qui pourront la modifier directement.

III.B.2.c Convention de nommage

Afin de deviner par le nommage le type d'une donnée d'un store (getter vs state), j'ai l'impression que les getters ont été préfixés de `get`. Cela correspond plutôt à une fonction, or, une computed / getter est plutôt une propriété. Il ne paraît pas utile de préfixer par `get`, pour éviter une incompréhension.

Par contre, l'usage de `is` devant (`isCurrentDepartmentSelected`) permet de comprendre qu'il s'agit d'un booléen, et que donc il pourra être utilisé en tant que tel.

Dernier conseil également sur la consistance de nommage, une fois une "norme" adoptée, penser à l'adopter partout, même si ce n'est pas toujours évident.

Exemple des stores, avec un `useAuth` vs `useRoomReservationStore`. Privilégier le nommage `useAuthStore` pour garantir la consistance.

III.B.3 Composables

Un composable, *conformément à la documentation*, est **une fonction qui exploite la Composition API de Vue pour encapsuler et réutiliser une logique avec état**.

Son utilité est une factorisation de code pour réutilisation future.

C'est le cas des stores Pinia par exemple.

Actuellement, le dossier `composables` du projet contient le fichier `api.ts`, ce même fichier contenant des fonctions utilitaires.

Il ne s'agit pas à proprement d'un candidat pour un composable, il serait plus adapté de déplacer ce fichier dans un dossier `utils` par exemple, ou `helpers`.

III.B.4 Composants provider

Les deux composants localisés dans le dossier `components/provider` m'ont paru "provider / fournir" des données au sein des enfants, via la mécanique *`provide / inject`*.

Mais cela ne semble pas être le cas, car `provide` n'est pas utilisé dans les composants.

De ce fait, la pertinence de les positionner dans `provider` paraît contre intuitive. De même, nous pouvons remettre en question la pertinence d'un composant tel que `ProvideDepartment`, qui semble avoir pour objectif de déclencher un chargement des départements.

En reprenant le code `App.vue` qui utilise le composant `ProvideDepartment`, nous pourrions obtenir ce code :

```
<template>
  <header>
    <Header></Header>
  </header>
  <main>
    <router-view></router-view>
  </main>
</template>

<script setup lang="ts">
import Header from '@/components/Header.vue';
import { useDepartmentStore } from '@/stores/department';
import { ref } from 'vue';

const deptStore = useDepartmentStore()
const departmentsLoaded = ref(false)

await deptStore.fetchAllDepartments()
departmentsLoaded.value = true
```

```
</script>

<style>
...
</style>
```

La variable `departmentsLoaded` n'a pas beaucoup d'utilité à ce niveau. Si l'objectif est de disposer d'un loader, peut être que `Suspense` pourrait aider. Attention, cette fonctionnalité est encore en expérimentation et son API n'est pas considérée comme stable.

III.B.5 Composants

L'analyse des composants situés dans `/components` m'amène à reprendre certains principes clés de VueJS ou de l'approche composant.

III.B.5.a Props in events out

Un tag HTML, tout comme un composant VueJS (et d'autres frameworks), respecte le contrat "**props in, events out**" :

- des propriétés, en lecture seule, en entrée
- des événements en sortie

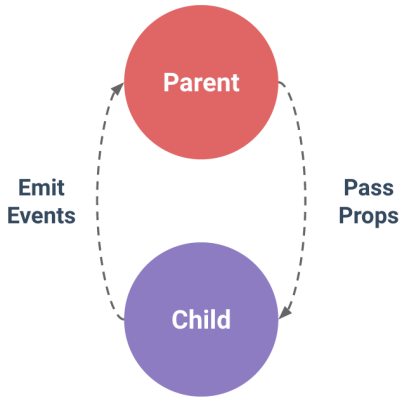

Figure 1: props in, events out

Certains cas d'usage "sortent" de ce schéma simplifié (cas des `provide/inject`) mais cela reste une bonne base d'écriture des composants.

La fonction `defineEmits` est très largement utilisé dans les composants de la base de code, et appuie ce principe de **props in, events out**.

J'ai pu noter une inconsistance dans `CalendarRoomReservationSlot`, avec certaines fonctions corrélées à une prop `actions` :

```
function onDelete() {
    props.actions.delete?.(props.data)
}
```

dans le script `setup` du composant.

Il serait préférable d'utiliser un événement `delete` qui propagerait la donnée :

```
const emits = defineEmits<{
  (e: 'delete'): void
}>()
function onDelete () {
  emits('delete')
}
```

Il est également probable que `props.data` soit inutile, car le cablage sur le `@delete` par le parent saurait de quelle `data` il s'agit.

Le raisonnement est également applicable à `DeletePeriodicReservationDialog.vue`, où le code pourrait être recodé comme suit :

```
import { computed, onMounted, ref, watch } from 'vue'
import type { RoomReservation } from '@/ts/types'
import ModalDialog from '@/components/dialog/ModalDialog.vue'

const props = defineProps<{
  reservation: RoomReservation | undefined
}>()

const emit = defineEmits<{
  (e: 'cancel'): void
  (e: 'confirmCurrent'): void
  (e: 'confirmAll'): void
  (e: 'confirmFuture'): void
}>()

const isCurrentSelected = ref(false)
const isAllSelected = ref(false)
const isFutureSelected = ref(false)
const isLocked = computed(() => isCurrentSelected.value || isAllSelected.value ||
↪  isFutureSelected.value)

function cancel() {
  emit('cancel')
}

function confirmCurrent() {
  isCurrentSelected.value = true
  emit('confirmCurrent')
}

function confirmAll() {
  isAllSelected.value = true
  emit('confirmAll')
}

function confirmFuture() {
  isFutureSelected.value = true
  emit('confirmFuture')
}
```

avec une mise à jour du composant parent `RoomReservationView` :

```
<DeletePeriodicReservationDialog
  v-if="isReservationDeletionDialogOpen"
  :reservation="reservationToDelete"
  @confirm-current="reservationDeletionConfirmed(reservationToDelete)"
  @confirm-all="reservationRemoveAllSamePeriodicity(reservationToDelete)"
  @confirm-future="reservationRemoveCurrentAndFutureSamePeriodicity(reservationToDelete)"
  @cancel="closeReservationDeletionDialog(reservationToDelete)"
></DeletePeriodicReservationDialog>
```

Les composants ne doivent pas savoir ce que devient la donnée. Ils ne sont que des exécutants d'affichage et de propagation d'événements. Le parent lui sait ce qu'il doit en faire.

III.B.5.b Dumb vs Smart components

Dans les composants, que ce soit en React, Angular, VueJS, nous distinguons deux catégories de composant : les composants dumb / bêtes, et les composants smart / intelligents.

Pour faire une analogie, les composants bêtes sont comme les tags HTML.

Un tag HTML, par exemple `<input>`, reçoit des "props" au sens VueJS, ou des "attributs" au sens HTML. L'attribut `type='text'` lui indique de se comporter comme un champ de saisie texte. L'attribut `type='number'` lui indique qu'il sera plutôt un champ de saisie de nombre. Dans ce cas, il nous interdira d'écrire des caractères, selon l'implémentation du navigateur.

Cet `input` n'a aucune connaissance du contexte dans lequel il est placé. Cela peut être un formulaire `form`, ou pas, il ne sait absolument pas ce que deviendra la valeur qui sera saisie, mais il la rend accessible via sa propre API. Par exemple, il émet un événement `input` ou `change` à chaque fois que la valeur de l'`input` change. Charge à celui qui a instancié cet `input` de manipuler et capturer les événements et les données associées.

Cet `input` est "bête", dans le sens où il ne fait que remplir son contrat, et n'a pas d'impact dans le contexte dans lequel il a été positionné.

À l'inverse, un composant "smart" peut avoir une influence sur le contexte. En VueJS, le contexte correspond au modèle de données, au store, à des appels à des API, à des déclenchements de traitement. Un composant "bête" de formulaire de login ne déclenchera pas l'appel à l'API pour s'authentifier. Il ne fera que renvoyer les données login/password à travers un événement, qui pourra être capturé le parent. Le composant "smart", lui, à partir des données de cet événement, contactera l'API pour authentifier l'utilisateur. Et selon la réponse, affichera une erreur, ou poursuivra la navigation.

En VueJS, les composants bêtes sont localisés dans le répertoire `components`, là où les composants smarts sont positionnés dans `views`.

Une bonne pratique est de réserver l'accès au store uniquement aux `views`. Les `components` sont uniquement en mode `props in events out`, tout comme un tag HTML.

Quelques liens :

- https://www.digitalocean.com/community/tutorials/react-smart-dumb-components
- https://michelestieven.medium.com/components-stateful-stateless-dumb-and-smart-2847dd4092f2

Pour illustrer ces principes au sein de la base de code actuelle, nous pouvons trouver quelques composants qui utilisent les données du store directement :

- `Authentication.vue`
- `DepartmentSelection.vue`
- `Menu.vue`
- `ProvideDepartment.vue`
- `ProvideUser.vue`
- `RoomDepartmentFilters.vue`

Il me paraît préférable d'injecter les données via des props en respectant le fait que ces composants soient juste des dumb components. Cela permettra une meilleure maintenabilité sur le moyen long terme, pour éviter de se poser la question "quel composant a modifié / touché la donnée ?".

Par exemple, dans le fichier `RoomDepartmentFilters.vue`, nous pouvons trouver :

```
const filterRoomstoDisplay = computed(() => {
  return roomStore.rooms.filter((r : Room) => {
    if (props.selectedDepartment) {
      let belongsToDept = false
      if(r.departments.length === 0) {
        belongsToDept = true
      } else {
        r.departments.forEach((dept: Department) => {
          if(dept.id === props.selectedDepartment.id) {
            belongsToDept = true
          }
        })
      }
      return r.is_basic && belongsToDept || r.departments.length === 0
    }
    return r.is_basic
  })
  .sort((r1 : Room, r2 : Room) => {
    return r1.name.toLowerCase().localeCompare(r2.name.toLowerCase())
  })
})
```

Cette `computed` s'appuie sur les données du store `roomStore`, et donc, nous constatons que ce composant a connaissance de comment est structurée la donnée du store. Il me paraît plus "composant compatible" de fournir directement la donnée `filterRoomsToDisplay` en propriété à ce composant, afin qu'il n'est pas à connaître la structure interne du store. Cela appuie le principe "Separation of Concerns" (nous y reviendrons plus tard).

III.B.5.c usage des slots

Sans avoir tous les tenants et aboutissants de certains composants, plusieurs composants définissent des slots avec un seul niveau de profondeur, sans une réutilisation fréquente.

Prenons l'exemple du composant `RoomCalendarScheduledCourseSlot.vue`, qui est une instanciation de `CalendarScheduledCourseSlot` à qui l'on donne une prop `data`.

```
<CalendarScheduledCourseSlot :data="props.data">
  <template #text>
    <p class="col-xl-auto">{{ props.data.course.course.module.abbrev }}</p>
    <p class="col-xl-auto">{{ props.data.course.tutor }}</p>
    <p class="col-xl-auto ms-auto">{{ startHour }} - {{ endHour }}</p>
  </template>
</CalendarScheduledCourseSlot>
```

Et `CalendarScheduledCourseSlot` est :

```
<PopperComponent
  :id="props.data.id"
  class="frame"
  :show="isContextMenuOpened"
  :style="props.data.displayStyle"
```

```
  @click="onClick"
>
  <div class="row m-0 course">
    <slot name="text"></slot>
  </div>
  <template #content></template>
</PopperComponent>
```

Nous pouvons constater dans le code présent que `CalendarScheduledCourseSlot` est également utilisé dans `HourCalendarScheduledCourseSlot.vue`. Pourquoi ne pas écrire directement dans `RoomCalendarScheduledCourseSlot.vue` :

```
<PopperComponent
  :id="props.data.id"
  class="frame"
  :show="isContextMenuOpened"
  :style="props.data.displayStyle"
  @click="onClick"
>
  <div class="row m-0 course">
    <p class="col-xl-auto">{{ props.data.course.course.module.abbrev }}</p>
    <p class="col-xl-auto">{{ props.data.course.tutor }}</p>
    <p class="col-xl-auto ms-auto">{{ startHour }} - {{ endHour }}</p>
  </div>
  <template #content></template>
</PopperComponent>
```

Cela permet de réduire le nombre de composants, et donc la surface de maintenance.

Cette suggestion reste toutefois critiquable, ma compréhension de l'usage fait des slots étant encore obscure à ce jour.

III.B.6 Typage

L'usage de TypeScript au sein du projet permet de mieux prévenir sur du moyen long terme les régressions.

Pour autant, actuellement, la compilation du projet (`yarn build`) ne passe pas, et l'IDE VS Code râle beaucoup.

Le compilateur TypeScript n'arrive pas à bien s'y retrouver au niveau des alias.

En ajoutant la propriété `paths` dans le `tsconfig.json`, nous aidons le compilateur TypeScript à retrouver les chemins aliasés :

```
{
  "compilerOptions": {
    "target": "ESNext",
    "useDefineForClassFields": true,
    "module": "ESNext",
    "moduleResolution": "Node",
    "strict": true,
    "jsx": "preserve",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "esModuleInterop": true,
    "lib": ["ESNext", "DOM"],
    "skipLibCheck": true,
```

```
+    "noEmit": true,
+    "paths": {
+      "@/*": ["./src/*"],
+    }
  },
  "include": ["src/**/*.ts", "src/**/*.d.ts", "src/**/*.tsx", "src/**/*.vue"],
  "references": [{ "path": "./tsconfig.node.json" }]
}
```

Désormais, notre compilateur peut poursuivre son analyse et trouver de nouvelles anomalies.

Par exemple, dans le fichier `RoomReservationView.vue`, l'import du fichier `@/ts/types` est en erreur car il s'agit du fichier `@/ts/type`. Et ainsi de suite.

En exécutant la commande `yarn build` suite à cette modification, il reste encore 63 erreurs dans 27 fichiers.

Chacune de ces erreurs doit être adressée en vue de builder le projet.

III.B.7 Asynchronisme, async, await, Promise

JavaScript dispose de plusieurs méthodes pour gérer l'asynchronisme.

En 2023, la méthode conseillée est les `async` / `await`. En effet, elle offre au moins deux avantages :

- la lecture en séquence du code
- une simplification du code, et donc une meilleure maintenabilité sur le moyen long terme

Nous retrouvons dans la base de code également des `Promise` avec les `.then`, `.catch` et `.finally`.

Prenons l'exemple de la fonction `fetchAllDepartments`,

```
async function fetchAllDepartments() : Promise<void> {
  await api?.getAllDepartments().then((json: any) => {
    departments.value = json
  })
}
```

elle peut probablement être réécrite comme suit :

```
async function fetchAllDepartments() {
  departments.value = await api.getAllDepartments()
}
```

Plus simple, plus facile à maintenir, moins d'erreur potentielle. J'ai également enlevé le `?` après `api` car c'est une constante, elle doit donc toujours exister.

Il me paraît être une bonne pratique que d'abandonner l'usage des `Promise` quand cela est possible, et de favoriser `async` / `await`.

III.B.8 Syntaxes avancées

III.B.8.a provide / inject

Dans le code "ancien" du frontend, au sein du répertoire `FlOpEDT/frontend`, j'ai rencontré le répertoire `@/assets/js`.

Il est utilisé, par exemple dans `FlOpEDT/frontend/src/views/RoomReservationView.vue` :

```
  125  <script setup lang="ts">
  126: import type { FlopAPI } from '@/assets/js/api'
```

```
127  import { createTime, listGroupBy, parseReason, toStringAtLeastTwoDigits } from
↪  '@/helpers'
128  import { apiKey, currentWeekKey, requireInjection } from '@/assets/js/keys'
...
171  import { type Room, useRoomStore } from '@/stores/room'
172
173: const api = ref<FlopAPI>(requireInjection(apiKey))
174  const currentWeek = ref(requireInjection(currentWeekKey))
175  let currentDepartment = ''
```

Je constate l'usage de *inject* de l'API VueJS qui permet d'injecter / récupérer des données à travers les différents composants à partir de données fournies par un `provide` (même page de documentation).

L'usage de `provide` est effectué dans le fichier `main.ts`, et - a priori - part du principe qu'il pourrait y avoir plusieurs applications VueJS qui tournent, (ligne 27, `const apps = [{ appName: 'roomreservation', app: roomreservation }]`) afin de boucler dessus et `provide` les différentes clés (`apiKey` et `currentWeekKey`).

À mon sens, l'usage de `provide` et `inject` semble peu pertinent à ce stade

- parce que pour l'instant, il n'y a qu'une seule app
- le fonctionnement modulaire de JavaScript via des `import` / `export` suffirait à avoir un singleton disponible aux endroits où il est utilisé

III.B.8.b usage des types

Une autre remarque également, concerne l'export de l'interface `FlopAPI`, il est fort probable que l'export de cette interface et même son écriture deviennent inutiles dans le cas où l'export de l'objet api est fait directement.

L'inférence de code avec TypeScript et l'intellisense d'un IDE tel que VS Code devrait suffire.

Il faudra peut-être prévoir de bien signer les méthodes de l'objet api.

Il n'est pas indispensable de typer une variable en TypeScript, l'inférence suffit généralement, sauf si l'on veut expliciter les attendus ou les retours. La compilation TypeScript vérifie également la cohérence, même si les types n'ont pas été clairement définis.

III.C Conclusion de l'analyse

Ainsi se termine cette partie analyse du code.

L'ensemble des points soulevés, remarques, anomalies, inconsistances doivent être bien entendu mis en relief avec la qualité d'écriture du code. L'exercice de l'audit porte sur ce qui est améliorable, mais l'usage de briques comme Pinia, vue-router, vue-i18n démontre une réelle volonté de professionnaliser l'outil en utilisant des briques reconnues.

Toutes ces annotations sont des améliorations potentielles, certaines nécessaires, mais rarement indispensables.

Également, cette analyse s'est faite avec l'appui initial de Pablo Seban qui a pu m'expliquer le fonctionnement général de l'application, et une phase "archéologie en autonomie" où j'ai pris parti sur l'interprétation du code. Cela pourra rendre caduque certains de mes conseils.

IV Préconisations

Après avoir essayé l'application, observé l'architecture du projet, analysé des morceaux de code plus en détail, voici un résumé des préconisations que je vous propose.

IV.A Architecture découplée

À ce jour, le backend Django sert génère une grande partie des pages, et l'API grossit progressivement.

Je vous encourage à poursuivre ces développements pour pouvoir disposer d'une architecture découplée entre un backend qui pourrait devenir interopérable avec, certes, la SPA en cours de développement, mais également d'autres outils, tels l'application mobile, ou bien encore des workflows d'automate type Talend / n8n.

Il pourrait être également pertinent d'ajouter la gestion des JWT, ce qui permettrait une communication avec l'API également sans Cookie, uniquement avec un header d'`Authorization`.

IV.B Migration du code d3 en VueJS

Le code d3 utilisé côté Django pour générer le calendrier est assez fourni, et présente des cas d'usages assez diversifiés.

La mécanique de réactivité de VueJS apporte normalement une souplesse lorsque la tuyauterie / plomberie est bien effectuée.

Il me paraîtrait pertinent de ne pas migrer le code d3 et de convertir les cas d'usage en template + TypeScript, en s'appuyant fortement sur la mécanique de réactivité de VueJS.

Cependant, à la lumière des cas d'usage déjà implémentés, je vous encourage à faire une petite étude des frameworks existant, ou des bibliothèques spécialisées dans les calendrier.

Voici quelques liens à explorer :

- Vuetify : https://github.com/vuetifyjs/vuetify
- Prime Vue : primevue.org/
- listing des frameworks VueJS : https://next.awesome-vue.js.org/components-and-libraries/frameworks.html
- FullCalendar : https://fullcalendar.io/docs/vue

IV.C Qualité

À ce jour, je n'ai pas identifié d'éléments encourageant des bonnes pratiques de qualité :

- pas de lint
- pas de test unitaire, d'intégration
- pas de design system / storybook
- pas / peu de documentation / commentaires dans le code (front)

À Makina Corpus, nous encourageons les pratiques favorisant la qualité d'un projet web, dans l'objectif de favoriser la maintenabilité du projet, garantir au mieux les non régression dans les évolutions, et permettre un abordage plus serein de nouveaux développeurs dans l'équipe.

IV.C.1 Lint

Plusieurs outils existent, dont deux que nous utilisons au quotidien :

- ESLint https://eslint.org/
- Prettier https://prettier.io/

Prettier est un formateur de code assez "opinionated", ce qui permet de partager un formatage de code à travers l'ensemble des développeurs.

ESLint va permettre d'analyser plus la syntaxe du code et prévenir d'erreurs d'exécutions.

Ces deux outils devront disposer de scripts `npm` dédiés, et si être intégré dans la chaîne d'intégration continue le cas échéant.

IV.C.2 Tests

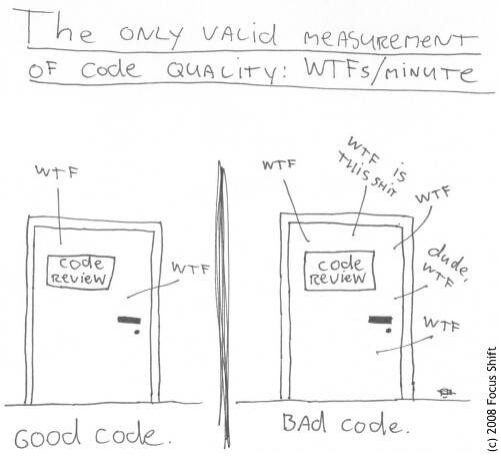

Figure 2: La qualité du code...

Sur un projet de cette envergure, il paraît prudent de prendre le temps d'ajouter des tests unitaires, de composants, de fonctions, afin de garantir la non régression et une bonne couverture de tests.

Au sein du projet front, il est possible d'ajouter la dépendance `vitest` (https://vitest.dev). Il s'agit de la bibliothèque portée par la communauté VueJS suite à la naissance du module bundler `vite`.

`vitest` offre une syntaxe similaire à `jest`, et est compatible avec `vue-test-utils`, paquet proposant des fonctions utilitaires pour manipuler des composants VueJS.

Une fois ces tests écrits, ils devront être joués à chaque modification de code source dans la chaîne d'intégration continue, si elle existe.

IV.C.3 Storybook

Qu'est ce qu'un **storybook** ?

C'est un outil permettant de travailler, développer, montrer les composants d'un projet.

Deux solutions existent pour les projets VueJS :

- Histoire : https://histoire.dev/
- Storybook : https://storybook.js.org/

Histoire est bien plus récent, et donc moins riche, que son prédécesseur Storybook. Par contre, il est plus compatible et rapide (car basé sur `vite`) pour les projets VueJS, là où Storybook est écrit en React, et est donc plus adapté pour les projets React, bien que compatible avec beaucoup de frameworks et disposant d'une communauté d'utilisateur / développeur bien plus grande.

Quel est l'intérêt d'utiliser un storybook ?

L'intérêt principal réside dans l'amélioration du développement et la maintenabilité. Pouvoir développer les composants de manière unitaire, sans être contextualisé de l'application, permet de mieux prendre conscience des paramètres nécessaires au composant (au sens `props in events out`) pour afficher correctement les données.

Dans le cas de Storybook (et bientôt côté Histoire), il est même possible de générer des tests de non régression visuelle en "rendant" les composants à travers les différentes stories écrites, puis en capturant un screenshot

du rendu, et en le comparant au précédent rendu de ce composant. Une différence de pixel trop importante fera échouer le test de non régression visuelle.

À intégrer dans la chaîne d'intégration également.

IV.C.4 Documentation

Le projet dispose de README expliquant l'installation, mais peu d'information sur l'architecture, les responsabilités de chaque brique, etc.

Sur la taille de ce projet, il paraît opportun d'écrire une documentation au plus proche du code, et pourquoi pas la publier en ligne sur un site Internet.

Il existe des outils côté Python, mais également côté Node, issus de la communauté VueJS :

- VuePress, ancêtre de VitePress : https://vuepress.vuejs.org/
- VitePress, qui est le moteur faisant tourner la documentation de vuejs : https://vitepress.dev/

À partir d'un répertoire `/docs` et de fichier markdown `*.md`, il est assez aisé de construire une documentation, progressivement, de fabriquer un site Internet statique, et de le publier sur des hébergeurs type surge ou netlify.

Par ailleurs, ajouter des commentaires dans le code, au niveau des entêtes de fonctions complexes par exemple, serait un plus pour disposer de la documentation "au bon endroit au bon moment".

IV.D Automatisation, chaîne d'intégration continue

Les commandes de build existent déjà au sein du projet, ce qui démontre une phase déjà avancée d'automatisation.

Il me paraît intéressant de pousser le concept plus loin en utilisant le potentiel de Gitlab et de son intégration continue à travers les pipelines.

En mettant une chaîne d'intégration continue, voire de déploiement continu (ou au moins automatisé), le projet pourra garantir :

- garantir une qualité d'écriture (règles de lint)
- garantir que les tests écrits continus de passer
- valider le fait que l'on peut toujours builder le projet (front ET back)
- bénéficier d'outils tiers vérifiant des règles de sécurité, de duplication de code (*SonarCloud* par ex.)

Pour les développeurs / administrateurs en place, cela permet de gagner du temps, de se rassurer, et de déployer facilement sur les environnements cibles.

IV.E Refactorisation / Simplification

Certains morceaux de code peuvent être améliorés / simplifier en usant de ce que le langage JS / TS nous permet.

Exemple, dans le fichier `api.ts` :

```
async function putData<T>(url: string, id: number, data: unknown): Promise<T | never> {
  const optional: { [key: string]: unknown } = {
    id: id,
    data: data,
  }
  return await sendData('PUT', url, optional)
}
```

peut être refacto comme suit :

```
async function putData<T>(url: string, id: number, data: unknown): Promise<T | never> {
  return await sendData('PUT', url, { id, data })
}
```

Nous utilisons ici la notation abrégée d'auto affectation `{ id, data }` qui correspond à `{ id: id, data: data }`.

Concernant la fonction `doFetch` du même fichier :

```
async function doFetch(url: string, requestInit: RequestInit) {
  return await fetch(url, requestInit)
    .then(async (response) => {
      const data = await response.json()
      if (!response.ok) {
        const error = data || `Error ${response.status}: ${response.statusText}`
        return Promise.reject(error)
      }
      return data
    })
    .catch((reason) => {
      return Promise.reject(reason)
    })
}
```

On pourrait plutôt coder ainsi :

```
async function doFetch(url: string, requestInit: RequestInit) {
  const response = await fetch(url, requestInit)
  const data = await response.json()
  if (!response.ok) {
    const error = data || `Error ${response.status}: ${response.statusText}`
    return error
  }
  return data
}
```

C'est à tester, mais l'erreur de `fetch` n'a pas nécessairement besoin d'être catchée si elle est rejettée telle quelle.

L'usage du `await` en lieu et place des `.then` et `.catch` permet également de mettre en séquence le code, ce qui peut avoir pour intérêt une amélioration de la lisibilité et de la maintenabilité.

IV.F Principes / Patterns

Quelques rappels de quelques principes qui me paraissent utiles pour ce projet

IV.F.1 Single Responsability Principle

https://fr.wikipedia.org/wiki/Principe_de_responsabilit%C3%A9_unique

Qui fait quoi.

Ce raisonnement peut s'appliquer pour les stores, au niveau du découpage.

Mais également au niveau des composants, qui ne doivent pas connaître leur environnement.

IV.F.2 Separation of Concerns

https://en.wikipedia.org/wiki/Separation_of_concerns

Attention, les composants sont "bêtes", les views sont "intelligentes". dumb vs smart.

Les views peuvent avoir accès au store et l'utiliser.

Les composants ne savent pas dans quel contexte ils sont. Ils sont bêtes, et font du `props in events out`.

IV.F.3 Model View View Model

C'est bien les données du modèle qui génèrent la vue.

Et la vue peut mettre à jour le modèle dans le cas de la directive `v-model` qui implémente le **two-way data binding**.

Attention à ne pas sur réagir à l'évolution du modèle, et privilégier le système de réactivité de VueJS qui permet de fabriquer le DOM en fonction de l'évolution du modèle.

IV.G Remarques annexes

Le dépôt git semble assez gros : ~200Mo pour le cloner. Il y a probablement un nettoyage du repo à effectuer pour purger d'anciens fichiers volumineux ?

L'import de données anonymes (seulement 2.3Mo) a été très très lent (plusieurs dizaines de minutes). Il y a probablement une anomalie de performance, mais je ne peux pas préciser laquelle.

Conclusion générale

Le projet flop!EDT est un projet d'envergure.

Par cet audit, restreint au code front, j'ai pu apercevoir la partie visible de l'iceberg, et imagine la complexité du backend...

Ce document n'a pas vocation à être exhaustif, ni parfaitement juste. Il est fort probable que certaines de mes observations soient erronées. Il peut parfois être également délicat de recevoir un document comme celui-ci quand beaucoup d'énergie personnelle, et peut-être de l'affect, a été mise dedans. C'est pourquoi j'ai essayé de m'attacher à fournir des exemples, le plus précis possible, le plus souvent possible, avec des directions à suivre.

Le projet frontend a embrassé plusieurs briques du framework VueJS, ce qui démontre une volonté de construire un projet sur la durée, avec l'internationalisation, l'approche composant, les stores, le typage fort avec TypeScript...

Des questions doivent cependant encore être posées et discutées, notamment autour de l'organisation des stores, et des responsabilités, partagées ou exclusives, entre les composants, les vues et les stores. En documentant ces choix, cela permettra de partir sur des bases saines pour agrandir progressivement le scope fonctionnel de l'applicatif VueJS.

Au delà du code, certains principes / patterns de développement (SRP/SoC) doivent être suivis comme des guides. Cela permet, je crois, d'avoir et de partager une vision sur la façon dont doit être conçue une application avec l'approche composants. C'est d'autant plus utile quand on utilise des briques modernes comme le framework VueJS : les principes peuvent toujours s'appliquer, peu importe la technologie employée.

Une attention doit également être apportée concernant la qualité. Sur un projet d'une telle envergure, et avec une vision moyen-long terme, il paraît indispensable de muscler cette partie, avec de la documentation, des règles d'écritures, des tests unitaires, et de l'automatisation avec une CI/CD.

Au delà du frontend, ces même principes peuvent s'appliquer, notamment sur le découpage backend / frontend. Il appartient bien au backend de gérer les processus métiers, et notamment les cas d'erreurs. Attention à ne pas donner trop de responsabilité au frontend, sous peine d'avoir des inconsistances ensuite en base de données.

J'espère que ce document permettra de mettre en lumière ces axes d'améliorations, et qu'il vous permettra de porter et maintenir ce projet dans les années à venir !

Questions avant remise définitive du rapport

Pourquoi utiliser des fonctions aussi avancées telles que :

- shallowRef, markRaw
- fonctions avancées telles que defineExposes, defineProps
- création de slots pour la partie calendrier ?

Proposition pour un accompagnement :

- refactorisation du code RoomReservation en version simplifiée avec découpage des données dans les stores et remise en question des slots
- audit UX pour les parcours d'usage
- démarrage de l'écriture du composant principal d'affichage à travers Histoire ou StoryBook