HEIG VD
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Conception d'une application modulaire pour une passerelle de visualisation intégrant des données issues de divers dispositifs

Travail de Bachelor

Non confidentiel

Département : TIC
Filière : Informatique et systèmes de communication
Orientation : Informatique logicielle

Nicolas Crausaz
26 mai 2023

Travail proposé par :
Loris Gavillet
YALK
Rue Basse 43, 1422 Grandson

Supervisé par :
Patrick Lachaize

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.
En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.
Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD
Le Chef du Département

Yverdon-les-Bains, le 26 mai 2023

Authentification

Le soussigné, Nicolas Crausaz, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Nicolas Crausaz

Yverdon-les-Bains, le 26 mai 2023

Résumé

Cliquez ou appuyez ici pour entrer du texte.

Table des matières

Table des figures

Liste des codes sources

Chapitre 1
Introduction

Ce travail de Bachelor consiste en la mise en place d'une application Web, destinée à s'exécuter sur une passerelle physique dans le réseau local d'un utilisateur. L'intérêt de cette passerelle est d'offrir un outil central pour mettre en place des interactions avec des services externes et dispositifs réseau, sous forme de modules. Cette solution permet à des utilisateurs d'automatiser des actions en utilisant et/ou développant des modules interagissant avec leurs propres services existants. La passerelle reliée à un ou plusieurs moniteurs permettra l'affichage d'informations issues de ces interactions, sous forme d'un tableau de bord modulable. L'aspect central du projet est la conception d'une architecture permettant l'exécution et la gestion des interactions. La finalité est que la passerelle puisse être considère comme un « *Proof of Concept* » permettant de démontrer les possibilités et les limites d'une telle solution, notamment grâce à la réalisation d'un module exploitant les possibilités de la passerelle pour interagir avec une application existante.

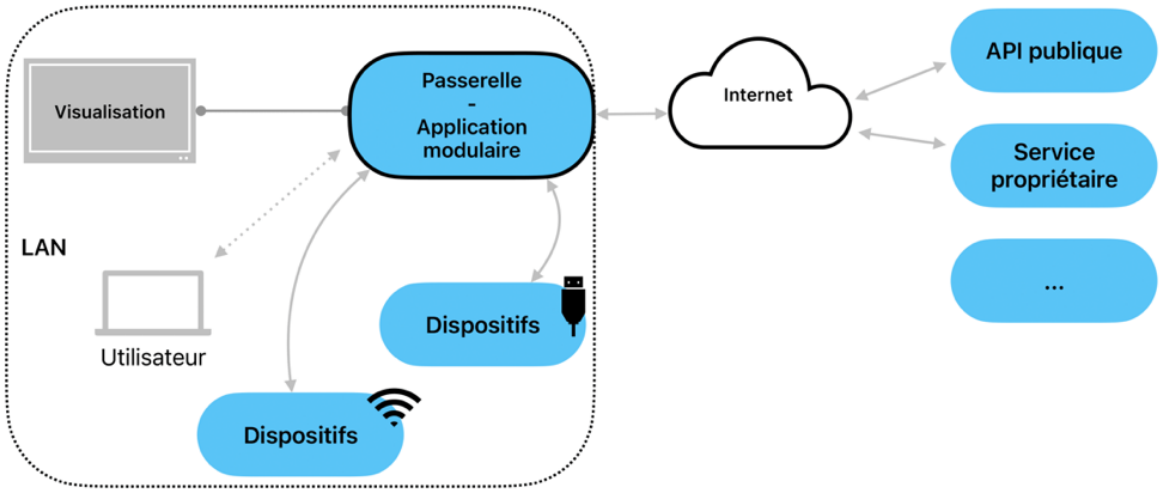


Figure 1: Contexte

1.1 Contexte

L'entreprise *YALK* a développé un logiciel nommé « *Composal* », un outil destiné aux entreprises, proposant entre autres des fonctionnalités suivantes :

- Gestion des interventions de technicien chez les clients
- Gestion de stock de matériel et de leur utilisation
- Planification des horaires de travail et timbrage des employés
- Génération de factures et de rapport d'intervention

L'intérêt de ce travail de Bachelor est d'agrandir l'écosystème autour de ce logiciel en mettant en place une passerelle offrant une application permettant des intégrations avec des dispositifs locaux, ainsi que des fonctionnalités de visualisation.
L'implémentation de ce projet est cependant totalement découplée des aspects du logiciel *Composal*. L'approche suivie dans ce travail a été de créer une architecture générique pouvant s'intégrer dans un écosystème totalement différent, ceci est rendu possible par la notion de « modules », composants autonomes pouvant être importés dans l'application afin d'ajouter des intégrations entre des composants totalement libres.
L'application *Composal* aura néanmoins un intérêt dans ce travail, puisque cette application existante pourra servir de référence pour tester les possibilités d'intégrations de dispositifs avec un logiciel concret.

1.2 Cahier des charges

1.2.1 Besoins fonctionnels

Les besoins fonctionnels suivants sont considérés comme des éléments de priorité, composant les objectifs fondamentaux du travail.

Modules
Un module est un composant logiciel destiné à offrir des fonctionnalités à la passerelle, principalement de gérer les interactions avec un dispositif précis : une API externe, un dispositif USB, WiFi etc. Les besoins liés aux modules sont :

1. Gestionnaire : Permettant l'activation / désactivation et la configuration du comportement d'un module.
2. Conception module « *Proof of Concept* » : Développement d'au moins un module spécifique permettant de gérer l'interaction entre un dispositif local et un service externe, permettant de démontrer l'intérêt de la solution.

Application Web

1. Structure : L'application offre deux interfaces : administration et visualisation. L'interface d'administration nécessite une authentification de l'utilisateur, la visualisation est accessible sans authentification, car elle ne propose pas de fonctionnalité d'édition. Les deux interfaces sont accessibles uniquement dans le réseau local.

2. Interface d'administration : permet l'activation / désactivation et la configuration des modules par l'utilisateur. Permet aussi la configuration du tableau de bord modulable, en laissant le choix à l'utilisateur de disposer les affichages des modules selon ses préférences.

3. Interface de visualisation : L'interface de visualisation affiche le tableau de bord selon la configuration effectuée dans l'administration. Lors d'une modification de la disposition des modules depuis l'administration, le tableau de bord devra se mettre à jour sans nécessiter d'interaction supplémentaire de l'utilisateur.

1.2.2 Besoins non fonctionnels

Support physique et système d'exploitation
La passerelle sera un *Raspberry Pi* relié au minimum à une source de courant, une connectivité réseau (avec ou sans fil) et pouvant être reliée à un ou deux moniteurs externes. Le système d'exploitation contiendra le nécessaire pour faire fonctionner l'application Web.

Architecture locale
Tout le nécessaire au bon fonctionnement de l'application s'exécutera sur la passerelle. La passerelle et son application web ne seront pas exposée en dehors du réseau local, mais pourront interagir avec des services externes (HTTP).

Performance
Du fait de l'architecture (ressources) de la passerelle relativement limitée, l'application s'exécutera sans consommation trop excessive de ressources et répondra dans un temps acceptable de manière à offrir une utilisation fluide.

Interface intuitive
L'interface de l'application web sera intuitive à utiliser pour un utilisateur ayant un bagage technique lui permettant de comprendre et configurer les modules et interactions.

Application fiable
L'application doit être capable de gérer des erreurs pouvant survenir et de pouvoir continuer à fonctionner. En cas de perte de connectivité puis de reconnexion réseau, l'application devra pouvoir refonctionner correctement sans interaction utilisateur.
En cas d'erreur non récupérable, il sera acceptable dans ce projet de simplement redémarrer l'application ou la passerelle.

Sécurité
L'application n'est pas conçue pour gérer des interactions critiques pouvant avoir des aspects de sécurité importants (transactions financières, etc.). Dans le cadre de ce travail, il n'est pas non plus nécessaire d'implémenter un système de sauvegarde de la configuration de la passerelle ou de sa réplication.

1.2.3 Extensions

Les extensions ci-dessous sont des besoins fonctionnels complémentaires faisant partie de la solution complète souhaitée par l'entreprise. Ces dernières ne seront pas réalisées dans le cadre de ce projet, sauf si le temps le permet, mais qu'il faut garder en tête lors de la conception des autres besoins pour prévoir leur ajout ultérieur.

Modules
Dans la version souhaitée par les besoins fonctionnels, tous les modules sont stockés localement sur la passerelle. Il serait intéressant de mettre en place un système d'import de nouveaux modules par l'utilisateur, afin de mettre en avant l'aspect modulaire de l'application.

Autonomie
Mise en place et configuration automatique / assistée de la passerelle dans le réseau local de l'utilisateur. Ceci permet de s'assurer une configuration correcte et connue pour l'exécution de l'application.

Gestion des écrans
Détection des écrans reliés à la passerelle pour pouvoir adapter le contenu du tableau de bord, selon l'orientation, la taille, la disposition relatives des écrans. Ceci permettrait d'ajouter un indicateur visuel et une aide à l'utilisateur configurant le tableau de bord.

Système d'exploitation
Il serait imaginable de créer une image d'un système d'exploitation contenant le strict nécessaire au fonctionnement de l'application. Ceci permettrait de limiter les interactions possibles de l'utilisateur final avec le logiciel, améliorant la sécurité et la stabilité de l'application.

1.3 Contraintes client

Les contraintes fixées par l'entreprise sont peu nombreuses. Le cadre du projet, bien que décrit dans le cahier des charges, ne représente que la première partie des fonctionnalités que l'application modulaire pourrait offrir. Il est donc nécessaire de concevoir l'application en ayant l'optique de réaliser une base solide en vue d'un développement futur.

L'entreprise *YALK* a fourni tout le matériel nécessaire au bon fonctionnement de la passerelle, entre autres le *Raspberry Pi* et toute sa connectique nécessaire, ainsi que du matériel destiné à réaliser le module « *Proof of Concept* ». Ce module devra s'intégrer à une application existante de l'entreprise *YALK*.
Aucune contrainte n'a été définie au niveau des aspect techniques et technologiques.

1.4 Solutions existantes

Sur le marché, il existe différents produits et applications orientées vers les mêmes objectifs que celui de ce travail. Ces produits ont chacun un public cible ou une portée différente et plus ou moins spécialisé dans un domaine particulier. Certains sont plus axés sur la domotique, et certains sont plus basé sur les aspects communautaire / Open Source. Il pourrait ainsi être intéressant de s'en inspirer. En voici un échantillon :

Odoo IoT Box

La *IoT Box* de *Odoo*[1] permet de connecter différents types de dispositifs (USB, Bluetooth...), elle a été conçue pour être utilisée qu'avec les applications de l'ERP[2] *Odoo*. Le support physique de cette IoT Box est un *Raspberry Pi*.
Contrairement à l'approche choisir dans ce projet, l'*Odoo IoT Box* restreint son utilisation à une série d'applications développées par la même entreprise. Il est donc possible d'en retenir que le choix du *Raspberry Pi* comme support pour l'application de ce projet est un choix viable.

Home Assistant

Home Assistant[3] est une plateforme Open Source, orienté sur la domotique, permettant de contrôler et d'automatiser une vaste gamme de dispositifs IoT. *Home Assistant* est une application développée en *Python* qui est destiné à être exécuté sur un *Raspberry Pi*, un NAS ou dans un container *Docker*. Cette application est très personnalisable et propose un grand catalogue d'intégrations, leur nombre étant propulsé par l'aspect communautaire du logiciel.

Node RED

Node-RED[4] est un outil de programmation visuelle Open Source. Il permet de créer des automatisations et des flux de données de manière graphique, en reliant des nœuds représentant différentes actions et en configurant leurs paramètres. Cette approche se différencie des autres produits du fait de la liberté complète donnée à l'utilisateur au niveau de la création de flux personnalisés.

Ces différentes solutions offrent chacun une approche relativement différente, selon le public cible et l'utilisation proposée. Tout de même, Les trois solutions se veulent suivre une approche « Low-Code » ou « No code », c'est-à-dire en amenant une couche de modèles graphiques permettant de réduite la complexité du code source, en ajoutant un aspect visuellement configurable pour l'utilisateur. Cela en fait des applications faciles à prendre en main pour l'utilisateur final sans besoin de compétences techniques avancées.

---

[1] https://www.odoo.com/fr_FR/app/iot-faq
[2] **E**nterprise **R**esource **P**lanning, ensemble d'applications permettant de gérer les processus de gestion d'une entreprise
[3] https://www.home-assistant.io/
[4] https://nodered.org/

L'approche choisie dans ce travail de Bachelor retrouve certains points communs avec les autres solutions présentées. En effet, l'application se veut être générique et personnalisable afin de pouvoir cibler un panel de personnes relativement large :

- Les entreprises, pour leur permettre de développer et exécuter des intégrations avec leurs propres logiciels.
- Les développeurs, pouvant concevoir des intégrations spécifiques et les mettre à disposition.
- Des utilisateurs non techniques, pouvant utiliser des modules préconçus et les configurer selon leur besoin.

La portée de l'application est ainsi très vaste. L'étude et la conception de l'architecture et des fonctionnalités de celle-ci dans ce travail permettront de spécifier les possibilités et limitations d'une telle approche.

Chapitre 2
Analyse

La première partie de l'analyse a consisté en la définition des fonctionnalités selon les besoins explicités au chapitre précédent. Cette division a permis de décomposer les bases de l'architecture de l'application, en séparant les fonctionnalités dans plusieurs composants ayant chaque leur rôle respectif et d'établir une première modélisation de leur fonctionnement et interactions. Cette analyse survole les différents concepts abordés lors de ce travail, en faisant abstraction des détails purement technique et d'implémentation ainsi que des choix technologiques.

2.1 Modules

Les modules sont l'aspect central de l'application, ils permettent d'offrir des fonctionnalités à la passerelle. Leur rôle est de fournir le comportement nécessaire afin de permettre l'interaction entre deux services ou dispositifs n'ayant pas initialement été conçu ou ne pouvant pas nativement interagir entre eux.
On peut isoler du cahier des charges les aspects fondamentaux d'un module afin d'établir les différentes fonctionnalités à mettre en place, un module est ainsi composé de plusieurs parties :

- *Un comportement* : la logique et les interactions offertes par le module y sont définie, sous forme de code.
- *Une configuration* : Afin de rendre le comportement d'un module personnalisable par l'utilisateur final, une configuration du module pourrait être plus ou moins spécifique dans l'objectif d'offrir une plus grande portée au comportement du module.
- *Un affichage* : il s'agit du rendu du module pouvant être affiché sur l'interface de visualisation de l'application.

Un module est destiné à être stocké localement sur la passerelle, il doit ainsi pouvoir être intégré à l'application sans autre transformation ou ajout de dépendances externes au projet, à condition de respecter une structure définie. Les trois parties ci-dessous forment ainsi la structure complète d'un module.

Le code nécessaire d'un module est structuré de manière à ce qu'il puisse être facilement packagé afin de pouvoir être ajouté à l'application et stocké de manière locale, mais également à pouvoir être publié sur un repository distant (hors du contexte de ce travail). La première étape de modélisation d'un module a donc consisté à établir une abstraction permettant d'unifier la structure d'un module, afin de pouvoir l'intégrer de manière simple et automatique dans l'application.

La structure de module suivante a été retenue :

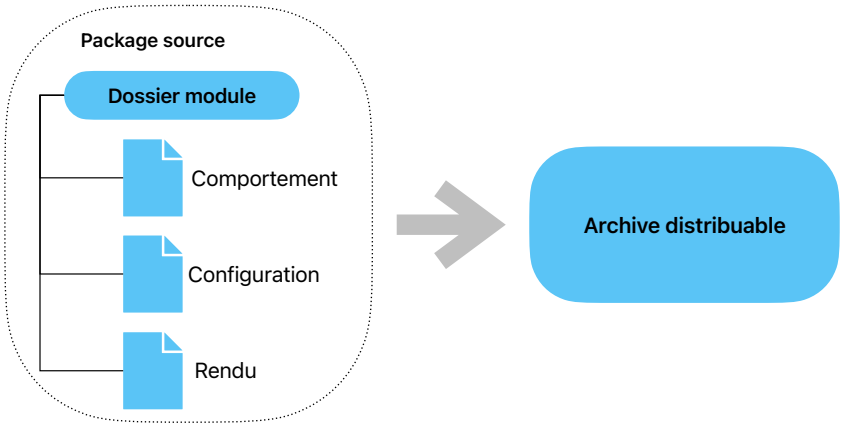


Figure 2 : Structure d'un module

Grâce à cette structure simple, un module peut être distribué facilement sous la forme d'un paquet. Dans le cadre de ce projet, j'ai décidé d'utiliser le format *zip* pour permettre le partage et l'import de module dans l'application.

D'un point de vue du développeur souhaitant mettre en place une nouvelle intégration grâce à un module, j'ai choisi de simplifier le processus de création en mettant à disposition un repository contenant un module d'exemple, permettant de démarrer avec une base fonctionnelle.

2.1.1 Comportement

Le comportement défini la logique du module, c'est-à-dire les interactions qu'ils va pouvoir offrir, sans ou avec une dépendance à un dispositif physique. Cette logique est définie selon son implémentation au niveau du code du module. Un module est capable de s'exécuter de manière autonome et de communiquer dans l'objectif de mettre à jour son affichage (*voir 2.1.3*). Afin de mettre en avant l'aspect indépendant du module, la modélisation de son comportement est établie sous la forme d'un cycle de vie. Le module peut être ajouté et supprimé de l'application, puis son comportement peut être activé ou désactivé.

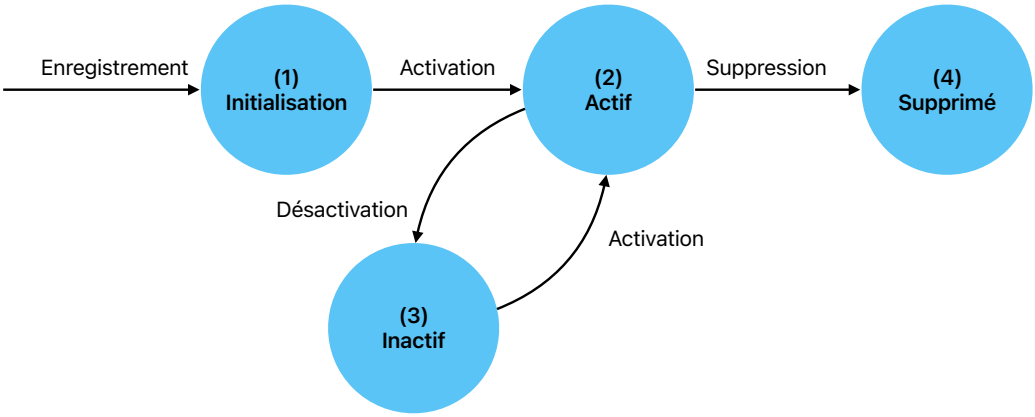


Figure 3: Cycle de vie d'un module

Le schéma ci-dessus présente les différents états possibles d'un module ainsi que les actions permettant de passer d'un état à l'autre. Le passage aux différents états déclenche des actions au niveau du module :

(1) L'état **initialisé** indique que le module à correctement été installé dans l'application. Dans cet état le module ne s'exécute pas, mais il déclenche une action d'initialisation. Celle-ci est définie au niveau du code par le développeur et permet d'effectuer de manière ponctuelle une série d'instructions, permettant par exemple d'établir une authentification à un service ou d'établir une configuration nécessaire au bon fonctionnement du module.

(2) L'état **actif** indique que le module est en cours d'exécution. Il réagit aux évènements reçus et notifie son affichage afin de le mettre à jour.

(3) L'état **inactif** indique que le module ne s'exécute pas, mais qu'il a auparavant été actif et initialisé, il peut ainsi être réactivé en tout temps. Lors du passage à cet état, le module stoppe toute réaction aux évènements et libère les ressources qu'il utilisait.

(4) L'état **supprimé** indique la suppression du module de l'application. Pour être réutilisé, il devra être réajouté dans l'application et repasser par l'état (1). Lors du passage à cet état, une action est déclenchée, permettant au développeur de libérer les dernières ressources non supprimées, généralement ce sont les ressources initialisées lors du passage à l'état (1).

Ces quatre états distincts permettent d'établir une première modélisation d'un module, sous la forme d'une classe abstraite. Le fait que cette classe soit abstraite impose au développeur d'hériter de cette classe et d'y implémenter le comportement attendu de son module.

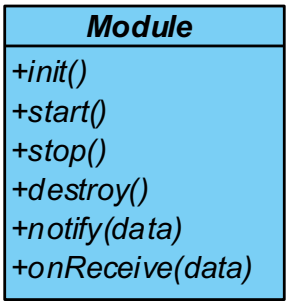


Figure 4 : Modélisation initiale de la classe Module

Chacune des méthodes de la classe Module est appelée lors d'un changement d'état spécifique, comme indiqué sur le diagramme précédent.
Lorsqu'un module se trouve dans l'état actif, il est nécessaire que celui-ci puisse notifier tout changement de son état ou une réception de données afin qu'il soit possible de savoir si son affichage nécessite d'être mis à jour. L'approche choisie pour implémenter cela est le design pattern *Observer* : Le pattern *Observer* est un modèle de conception comportemental qui permet à un objet, appelé sujet, de maintenir une liste de ses observateurs et de les notifier automatiquement en cas de changement d'état.

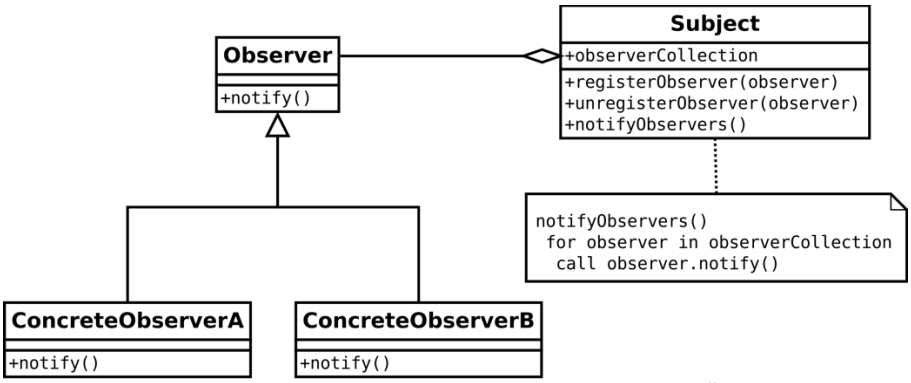


Figure 5: Pattern Observer, selon le GoF[5]

Afin de pouvoir observer les changements d'un module ou de s'en désinscrire, deux méthodes supplémentaires ont été ajouté à la modélisation de la classe Module : *subscribe* et *unsubscribe.*

Pour permettre au module de recevoir des données externes, la classe module offre une méthode *onReceive*, qui devra également être implémentée par le développeur pour définir un comportement lors de la réception de données. Il est donc nécessaire d'avoir une entité supplémentaire capable d'observer les évènements du module (rôle *Observer* du pattern) ainsi que de lui transmettre les données qui lui sont destinée par appel à la méthode *onReceive*. Cet aspect est détaillé au point *2.2*.

2.1.2 Configuration

Toujours dans l'objectif de rendre les modules les plus génériques possible afin de ne pas limiter la portée des interactions implémentables par un développeur, les modules sont liés à une configuration. Cette configuration permet tout d'abord de spécifier les valeurs obligatoires à leur fonctionnement ainsi que des valeurs relatives au comportement du module de manière spécifique. L'approche choisie pour établir cette notion de configuration est d'établir deux contextes.
Premièrement, la configuration principale défini les valeurs obligatoires à un module, celles-ci sont :

- Un nom
- Un auteur
- Une version
- Une description

Ces éléments permettront à l'utilisateur d'identifier les différents modules au travers de l'application.

[5] "Gang of Four", auteurs du livre Design Patterns: Elements of Reusable Object-Oriented Software

Chaque configuration principale contient une sous configuration, appelée configuration spécifique, ensemble valeurs dont la structure est cette fois définie par le développeur, qui doit respecter certaines contraintes. Le développeur a la liberté d'ajouter autant de valeurs configurables.

La configuration complète doit être spécifiée dans le fichier (JSON) de configuration du module, en voici un exemple :

```json
{
  "name": "Hello Module",
  "description": "A simple module to say hello",
  "version": "1.0.0",
  "author": "Nicolas Crausaz",
  "specificConfig": {
    "message": {
      "type": "text",
      "label": "Message",
      "description": "The message to say",
      "value": "Hello"
    },
    "refreshRate": {
      "type": "number",
      "label": "Refresh rate",
      "description": "The refresh rate in ms",
      "value": 5000
    },
    "showDate": {
      "type": "bool",
      "label": "Show date",
      "description": "Show the date",
      "value": true
    },
    "color": {
      "type": "option",
      "label": "Background color",
      "description": "The color of the background",
      "value": "#FFFFFF",
      "options": ["#FFFFFF", "#000000", "#FF0000"]
    }
  }
}
```

Listing 1 - Exemple de configuration d'un module

La structure de configuration est représentée par un objet JSON, contentant les quatre valeurs obligatoires (nom, description, version et auteur) et d'un sous-objet (*specificConfig*) contenant la définition des valeurs relative au comportement du module. Chaque valeur doit être spécifiée sous forme d'un objet JSON, ceci permettant d'assurer que chaque clé est unique.

Une valeur est obligatoirement structurée de la manière suivante :

- *type* : représente le type de la valeur, j'ai choisi de définir les types :
  - 'text' : Une chaine de caractère
  - 'number' : Un nombre
  - 'bool' : Une valeur vrai ou faux
  - 'option' : Une valeur de type 'text' à choisir parmi une liste prédéfinie de choix
- *label* : décrit brièvement la valeur.
- *description (optionnelle)* : décrit plus en détail la valeur et son influence sur le comportement du module.
- *value* : défini la valeur par défaut.
- *options* : défini la liste de choix possible, pris en compte si le type est 'option'.

Dans le futur, il serait intéressant d'ajouter d'autres champs spécifiques selon le type de valeur, par exemple une plage de valeur admissible pour les 'number', une longueur maximale pour les 'text' etc. Ceci permettrait notamment d'ajouter un système de validation des valeurs. Pour rester à l'essentiel, cela n'a pas été implémenté dans ce projet.

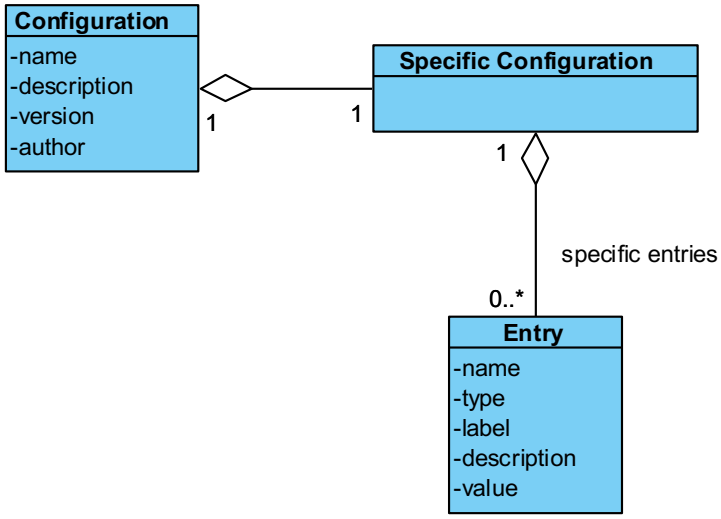


Figure 6: Modélisation du domaine : configuration d'un module

Les informations faisant partie de la configuration standard ne sont pas modifiables par l'utilisateur. Les valeurs définies dans la configuration spécifique sont quant à elles modifiables et sont accessibles depuis le comportement du module, ce qui permet de faire varier son flux d'exécution. L'interface de l'application a pour but de permettre leur modification depuis l'interface d'administration des modules (*voir 2.3.1*).

2.1.3 Affichage

L'affichage d'un module correspond à sa représentation affichée sur le tableau de bord de l'interface de visualisation de l'application. Cet affichage est soumis à quelques contraintes : il doit être interprétable par l'interface web afin d'être affiché de manière dynamique, sans pour autant que le code nécessaire doive être conservé du côté client. Les modules étant conservés sur le système de fichiers du serveur de l'application, l'accès direct à cette définition depuis l'interface n'est pas possible sans passer par un serveur Web.

Le comportement du module étant responsable de réagir à des évènements (dispositifs), le rôle de l'affichage est quant à lui de se mettre à jour selon les données qu'il reçoit. Ces données peuvent ainsi être structurée d'une manière totalement différente selon l'implémentation du module.

Pour laisser le choix du rendu visuel de cet affichage au développeur du module, l'approche choisie est de représenter cet affichage sous la forme d'un fichier HTML (ou tout autre format pouvant être transformé en HTML) capable d'inclure des variables qui pourront ensuite être remplacées par les données du module lors d'une phase de « rendering ». Cette approche permet une liberté dans le résultat visuel final car le développeur ne sera que très peu contraint sur la structure de l'affichage. La transformation de contenu dynamique vers du contenu statique est une fonctionnalité offerte par des outils appelés « templating engines ». Ces outils permettent de définir une structure statique souhaitée, selon leur syntaxe respective, puis d'y injecter dynamiquement des données durant l'exécution. Les templating engines ont souvent leur propre syntaxe et extension de fichier, la définition de la structure sera donc conservée dans un fichier spécifique de l'archive du module.

Le module offre une méthode chargée d'effectuer le rendering de l'affichage du module vers une structure statique en y injectant les données, passées en paramètre de cette méthode. Ce rendu pourra ensuite être utilisé par le programme principal, dans l'objectif de pouvoir être affichés sur l'interface de l'application. Le processus de génération de l'affichage s'effectue donc selon le modèle suivant :

Figure 7: Séquence de "rendering" d'un "templating engine"

2.1.4 Modélisation

La modélisation du domaine suivante récapitule les concepts expliqués dans ce chapitre :

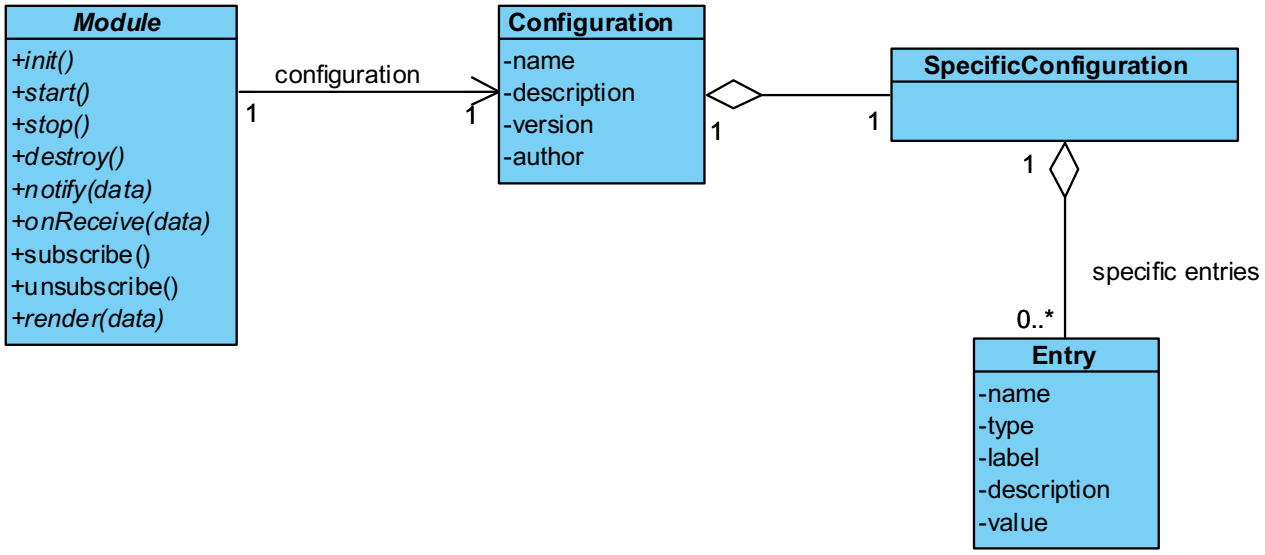


Figure 8: Modélisation du domaine : module

Cette modélisation constitue une base pour l'implémentation d'un module. Cette approche fait abstraction de toute structure externe ou englobant le module, ce qui en fait un composant logiciel autonome pouvant s'intégrer dans des contextes d'application différents. Il sera de la responsabilité du client (code utilisant un module) de s'abonner et de traiter les événements du module, en tant d'*Observer*.

2.1.5 Exemple

Afin d'avoir une vue d'ensemble sur le fonctionnement d'un module et de ces composants, prenons un exemple. Imaginons un module dont l'unique but est de donner l'heure. Son comportement est très simple, il contient une boucle interne s'exécutant à un intervalle régulier d'une seconde mettant à jour l'heure actuelle. Lors de la mise à jour de l'heure, le module notifie *l'observer* (le processus qui a initié le module) afin que ce dernier puisse déclencher le processus de rendu du module pour générer l'affichage avec la nouvelle heure.
Le diagramme de séquence suivant propose un exemple d'interactions avec modules durant son cycle de vie :

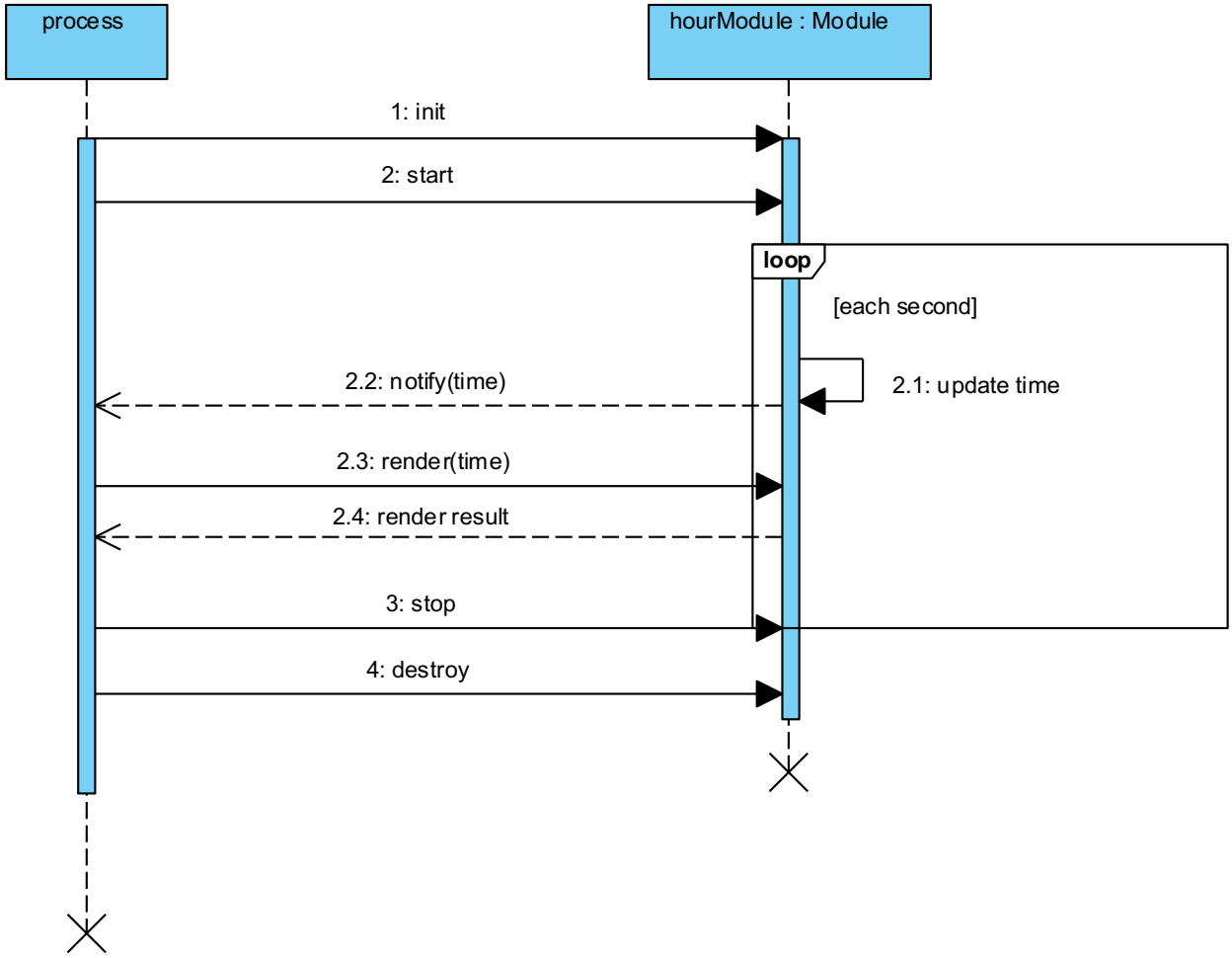


Figure 9: Module d'exemple, diagramme de séquence

Dans ce schéma, un seul module est initialisé et utilisé par le un processus arbitraire, jouant le rôle d'observer et déclenchant le processus de rendu lors de mise à jour du module. L'application ayant un besoin de pouvoir gérer un grand nombre de modules de manière concurrente. Les différents besoins liés aux modules nécessitent donc une entité qui permet leur gestion, notamment pour s'assurer de la cohérence de leurs états, ceci afin d'également faciliter leur ajout à la plateforme et d'éviter les erreurs. Ainsi, un gestionnaire de module doit être mis en place.

2.2 Gestionnaire de modules

Le gestionnaire de modules permet d'assurer le bon fonctionnement et la cohabitation des différents modules au sein de l'application. Il prend le rôle de façade en tant que point central d'accès aux modules.
Il encapsule la gestion des états des modules et définit les méthodes liées, telles que :

- Permettre l'ajout et la suppression de modules, en s'assurant de les initialiser correctement et de charger leur configuration.
- Permettre l'activation et la désactivation de modules
- Permettre le chargement dynamique des modules, sans devoir effectuer quelconque compilation ou redémarrage de l'application.
- Mettre à disposition des méthodes pour s'enregistrer aux évènements des modules (Observer) et d'y transmettre des données.

Le gestionnaire de module est l'entité responsable du bon fonctionnement des modules et du respect de leur cycle de vie (*voir 2.1.1*), il propose donc une abstraction des opérations offertes par les modules en assurant l'appel à leur méthode correcte du cycle de vie. Il permet principalement de stocker et gérer les modules d'une manière centrale. Il conserve les paquets des modules importés dans un répertoire du système de fichier du serveur et est capable de les charger dynamiquement en mémoire afin d'assurer leur exécution.
Afin de pouvoir identifier les modules importés et de connaître leur état, le gestionnaire de module conserve deux informations pour chaque module : un identifiant unique et un booléen permettant de savoir s'il est actif (en cours d'exécution). L'identifiant unique permet de différencier les instances des modules et permettre à l'application de communiquer avec un module spécifique.

Il est ainsi tout à fait possible d'importer le même module plusieurs fois (le même code source depuis son archive), mais chacun d'entre eux sera considéré comme une instance différente, il est donc possible d'avoir plusieurs fois le même module s'exécutant avec une configuration différente.
Le gestionnaire offre donc une abstraction des opérations possibles sur les modules, celles-ci sont définies dans la modélisation du domaine ci-dessous :

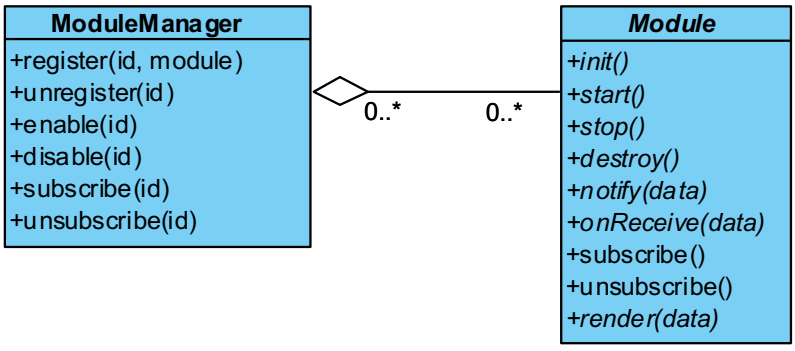


Figure 10: Modélisation du domaine : gestionnaire de modules

Lors des appels aux différentes méthodes offertes par le gestionnaire (*ModuleManager*), celui-ci est responsable de déléguer les appels aux modules (selon les identifiants) et de maintenir leur état d'activation à jour.

Comme indiqué plus haut, le gestionnaire de module conserve les codes sources des modules dans un dossier spécifique, il est donc capable de charger tout module ajouté dans ce dossier. Lors du lancement de l'application, les modules seront chargés par le gestionnaire. Grâce à la base de données (*voir 2.4*), l'état des différents modules peuvent être restaurés et les modules auparavant actifs réactivés lors de l'initialisation.

L'import d'un module depuis son archive *zip* est également possible. Pour que l'ajout d'un module soit possible, son archive doit être composée des trois fichiers obligatoires (comportement, configuration et affichage), lesquels doivent également respecter la structure imposée : pour la configuration les valeurs obligatoires (nom, description, auteur et version) doivent être spécifiés. Pour le comportement, le code doit être exécutable et respecter l'interface définie par la classe abstraite Module.

Le processus d'ajout d'un module au gestionnaire de module s'effectue selon la séquence suivante :

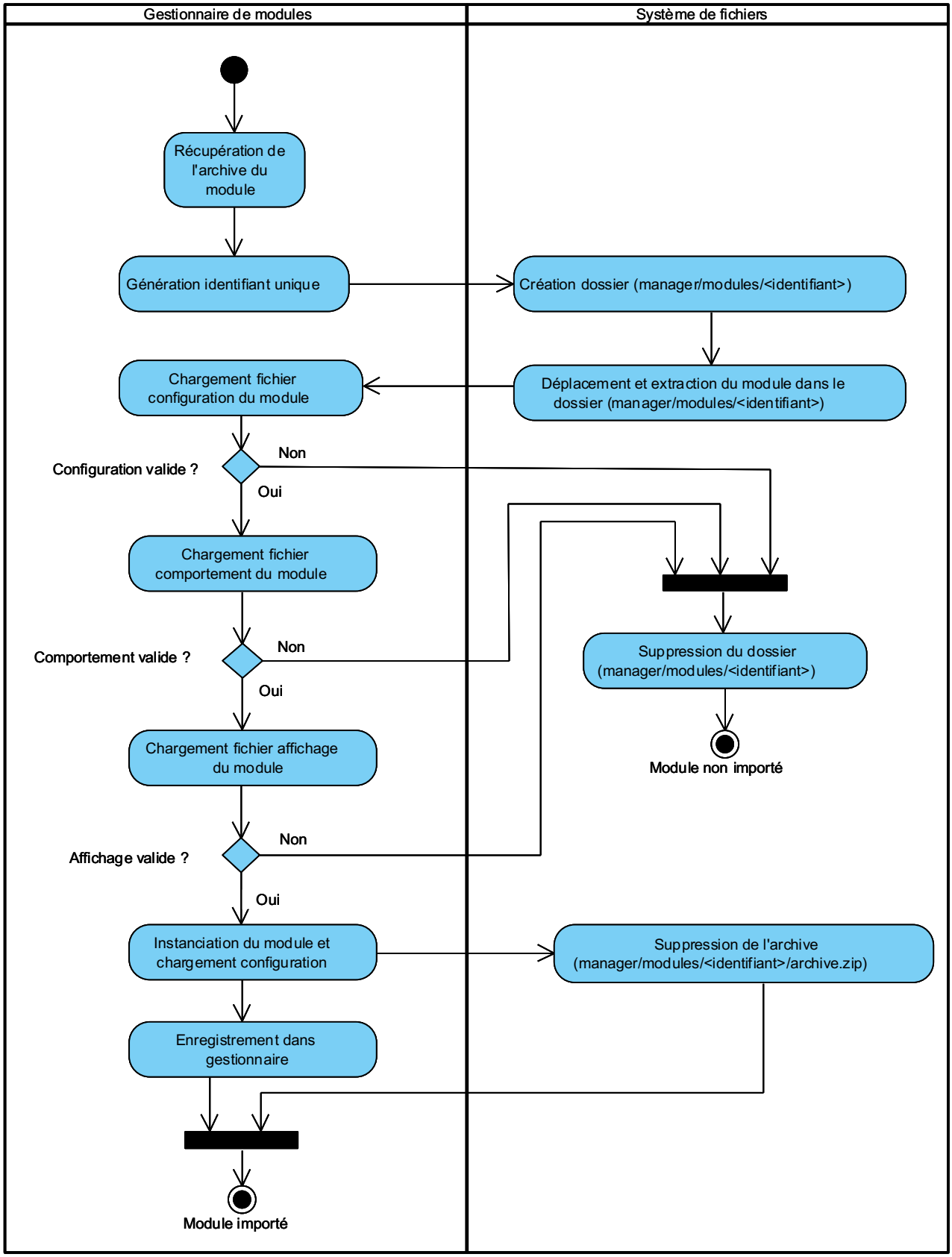


Figure 11: Diagramme d'activité, import d'un module

2.3 Application Web

L'application Web, destinée à être hébergée sur la passerelle, doit être accessible aux utilisateurs du réseau local. De ce fait, la passerelle exécute un serveur web, celui-ci est chargé de rendre disponible l'interface de l'application ainsi qu'une API offrant des fonctionnalités à l'application.

L'application est structurée selon le modèle d'architecture trois tiers (*3-tier*). Une architecture trois tiers divise l'application en trois couches logiques et/ou physique, chacune ayant son propre rôle strictement défini :

- *Présentation* : Correspond à l'interface client de l'application, dans ce projet il s'agit d'une interface web.
- *Logique métier* : Correspond à la partie fonctionnelle qui applique la logique métier et les opérations sur les données. Dans ce projet, il s'agit de l'API.
- *Accès aux données* : Correspond au stockage des données permanentes de l'application (base de données). Cette couche peut d'exécuter sur un tier physique différent de l'application.

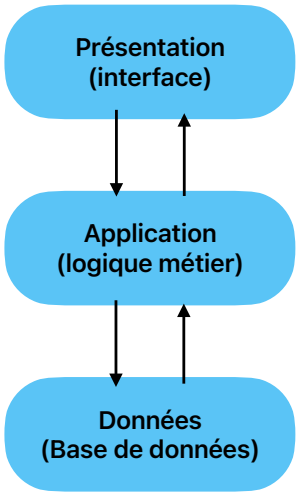


Figure 12: Architecture trois tiers

Chacune des couches ne communique qu'avec ses voisins directs au travers des services que ceux-ci proposent (la couche de présentation ne communiquera donc jamais directement avec la couche d'accès aux données). Avec ce type d'architecture, on cherche à mieux répartir la charge et alléger le travail du client en lui ôtant toute gestion métier.

Les détails de l'analyse de chacun de ces trois tiers et leurs fonctionnalités sont décrits dans les points suivants de ce chapitre.

2.3.1 Interface

L'application offre aux utilisateurs les diverses fonctionnalités liées aux modules cités dans les points précédents et plus encore. L'interface web correspond au tier de présentation de l'architecture trois tiers.

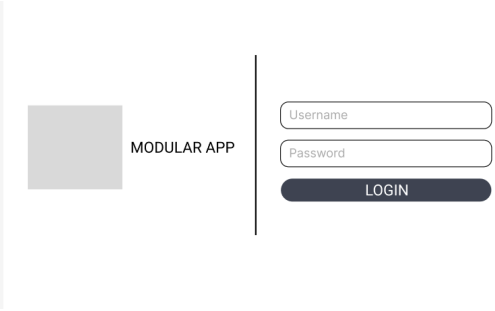
L'interface web de l'application est séparée conceptuellement en deux parties, chacune jouant un rôle distinct :

- *Interface d'administration* : permet toutes les opérations liées aux modules citées dans les points précédents, telle que l'ajout, la gestion des états et la configuration des modules, ainsi que la configuration du tableau de bord. Le tableau de bord permet de disposer les différents affichages spécifiques aux modules, sous forme d'une grille personnalisable par l'utilisateur. Pour accéder à cette interface, l'authentification de l'utilisateur est nécessaire.

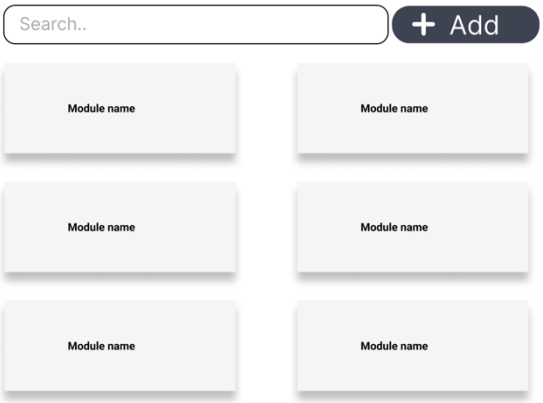
- *Interface de visualisation* : permet l'affichage en lecture seule du tableau de bord configuré dans l'interface d'administration. Cette interface est destinée à être affichée sur l'écran directement relié à la passerelle, elle ne nécessite donc pas d'authentification utilisateur et ne propose pas d'interactions directe avec l'utilisateur.

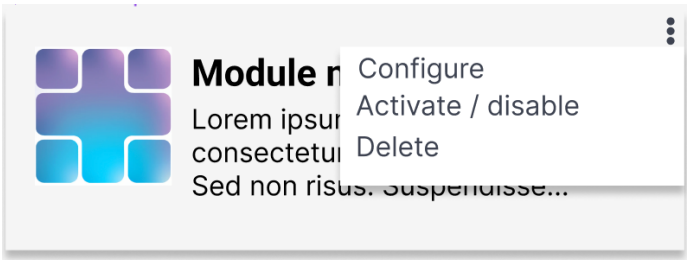
L'interface complète sera donc composée des pages suivantes :

- *Connexion* : Authentification de l'utilisateur par nom d'utilisateur et mot de passe.

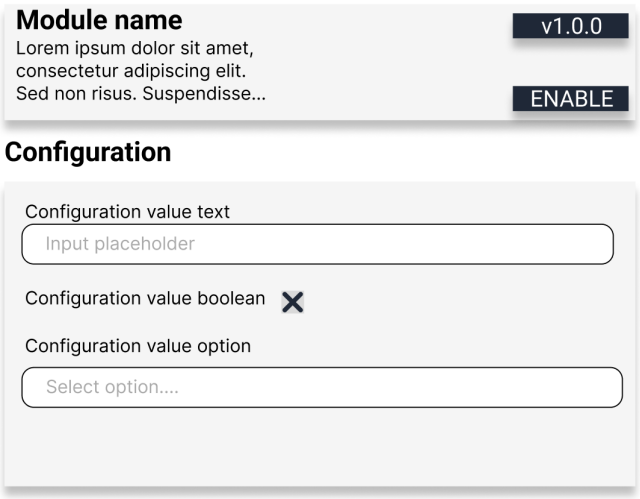


- *Modules* : Listes des modules disponibles localement sur la passerelle, possibilités d'activation, désactivation, ajout (extension) et suppression (extension) des modules.

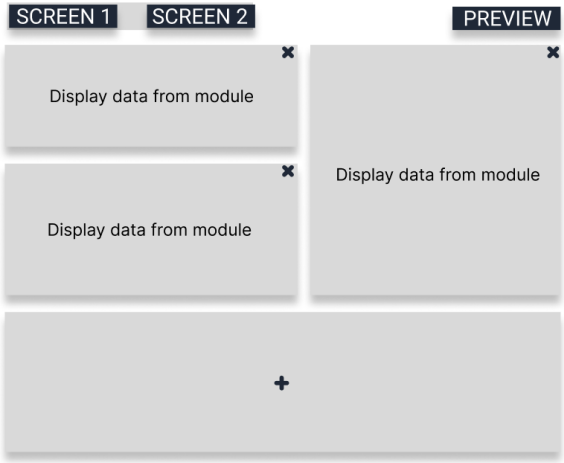




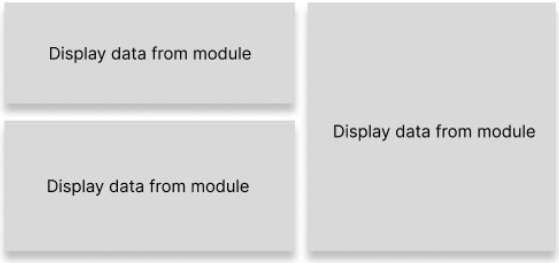
- *Configuration de module* : Affichage d'un unique module, modification de sa configuration, activation et désactivation.



- *Tableau de bord (édition)* : Configuration de la disposition des affichages des modules selon une grille. Possibilité d'avoir une disposition différente pour chaque écran.



- *Tableau de bord (affichage)* : Affichage du tableau de bord selon la configuration effectuée et selon l'écran choisi.



- *Paramètres* : Configuration générale des informations de la passerelle. Cette page n'est pas explicitée dans le cahier des charges mais permettrait dans le futur plusieurs ajouts (gestion de la configuration de la passerelle, utilisateurs etc.)

2.3.2 API

L'application est composée d'un serveur (backend) permettant de traiter les requêtes venant de l'interface, ainsi que de permettre la communication vers et depuis les modules. Ceci est rendu possible par la mise à disposition d'une API respectant les contraintes d'architecture *REST* (Representational State Transfer). Ces contraintes forment un ensemble de directives à adopter sur une architecture. Une API respectant les contraintes explicitées est dite *RESTful* :

- Interface uniforme : les interactions entre le client et le serveur respectent des aspects communs au niveau de l'interface, tels que l'utilisation des méthodes HTTP standards (GET, POST...) et la manipulation des ressources au travers de leur représentation selon un format défini (JSON, XML...).
- Client-serveur : les deux composants du logiciel doivent être dissociés de manière à évoluer de manière indépendante.
- Sans-état (*stateless*) : Chaque requête du client doit contenir toutes les informations nécessaires à son traitement, car le serveur ne conserve aucune information entre les différentes requêtes.
- Système en couches : le système peut être composé de plusieurs couches, chacune étant responsable de ses propres fonctionnalités, ceci ajoute une évolutivité à l'architecture.
- Cache : Les réponses du serveur peuvent mise en cache par des tiers (le client ou d'autres serveurs) dans l'objectif d'alléger la charge.

L'objectif de l'API est d'offrir toutes les fonctionnalités de l'application, tant pour l'interface utilisateur que pour l'interactions avec les modules depuis le réseau local. Ceci permet d'étendre les fonctionnalités de l'application en permettant à d'autres applications, clients ou dispositifs de communiquer avec les modules au travers d'une interface HTTP définie, afin de pouvoir déclencher des actions spécifiques.

L'API propose des opérations sur plusieurs types de ressources :

- Les modules : afin de permettre leur ajout, activation, désactivation et suppression, mais également l'édition de leur configuration spécifique, la récupération de leur affichage et l'envoi de données vers un module.
- L'authentification : afin de sécuriser l'accès à l'interface de l'application et à l'API.
- Les écrans de visualisation : afin de permettre leur personnalisation grâce à l'ajout, l'édition et la suppression des dispositions des modules, destinés à être affichés sur l'interface de visualisation.

2.4 Base de données

La passerelle évoluant dans un environnement LAN de l'utilisateur final, aucun contrôle à distance n'est possible pour un quelconque administrateur. Il en va donc de soi que l'application sache réagir à des perturbations, notamment à l'arrêt de la passerelle. L'application doit donc être capable de retrouver un état cohérent et identique à celui précédent l'interruption.
Une base de données doit donc être mise en place pour assurer une persistance des données, principalement pour conserver les configurations des modules afin d'assurer leur bon fonctionnement, mais est également nécessaire à l'implémentation des autres fonctionnalités de l'application, telles que l'authentification des utilisateurs accédant à l'interface d'administration et la configuration des tableaux de visualisation.

Le premier dilemme a été de choisir quel type de base de données utiliser pour ce projet. Deux approches possibles ont été abordées :

- *Base de données relationnelles* : Une base de données relationnelle organise les données sous formes de tables en utilisant des relations et des contraintes, celles-ci assurant l'intégrité et la cohérence des données. Les bases de données relationnelles utilisent le langage SQL pour les requêtes. On peut citer comme système de gestion de bases de données relationnelle (SGBDR) PostgreSQL, MySQL, SQLite...

- *Base de données NoSQL orientées document* : Il existe de nombreux types différents de base de données NoSQL, parmi lesquelles l'approche orientée document correspond au cas d'utilisation de ce projet. Les informations sont stockées sous forme de document semi-structuré, généralement au format JSON. La représentation des données est très flexible, car non définie par des contraintes d'intégrité, ni par un modèle structurel. Généralement chaque SGBD utilise son propre langage de requête. On peut citer comme système de gestion de bases de données (SGBD) orienté document : MongoDB, Couchbase...

Dans le cadre de ce projet, les deux approches peuvent être envisagée, le choix final a principalement été motivé selon les contraintes d'utilisation de ressource de l'application. Le fait d'exécuter un serveur de base de données (SGBD) en parallèle de l'application est relativement couteux en termes de ressources et fait peu de sens puisque seule l'application y accélèrerait localement. Le choix d'un serveur est ainsi trop conséquent pour l'utilisation finale qui en sera faite. Cependant, il existe des systèmes de gestion de base de données dits « embarqué » (*embedded database*) qui proposent des bases de données sans serveur intégrée à l'application, la base de données est généralement stockée sous la forme d'un unique fichier. Parmi les systèmes de base de données relationnels embarquées, on peut citer le plus connu et très populaire SQLite.
Pour l'approche orienté document, l'offre est moins convaincante : le principe utilisé par les solutions existantes (PoloDB, UnQLite) revient à écrire dans un fichier JSON.

Dans l'approche relationnelle, la plupart des SGBDR, dont SQLite, permettent de stocker des attributs à structure variables grâce à des attributs de type JSON, ceci pourrait être utile notamment pour stocker la configuration dynamique des modules.

Le choix final s'est donc porté sur l'utilisation d'une base de données embarquée de type relationnelle, dans la volonté de tirer avantage de l'intégrité et la cohérence des données.

2.4.1 Modèle conceptuel

Le modèle conceptuel (modèle Entité - Association) suivant regroupe les aspects concernés par la base de données, connus à ce niveau de l'analyse :

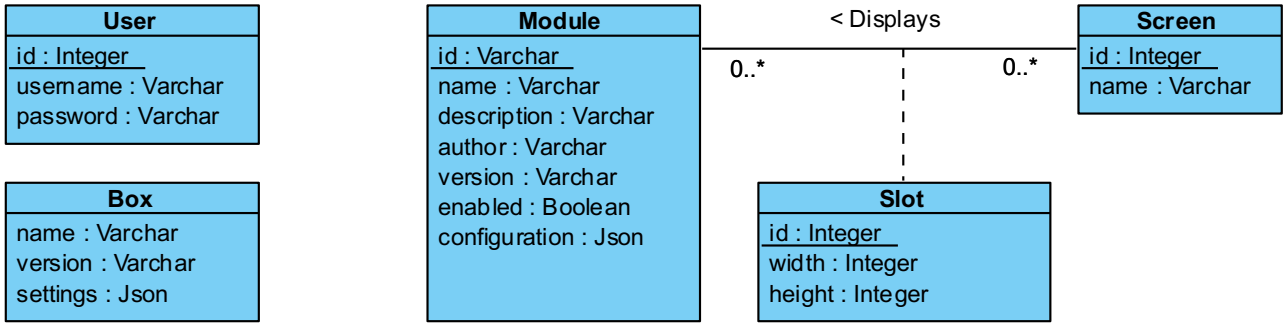


Figure 13: Base de données, modèle conceptuel

- ***User*** : entité représentant les utilisateurs, permettant d'authentifier l'accès à l'application.
- ***Box*** : entité représentant la passerelle, utilisée pour permettre la personnalisation de la passerelle et son application (fonctionnalités supplémentaires hors cahiers des charges).
- ***Module*** : entité représentant les modules ainsi que leur configuration. Les champs de la configuration spécifique sont stockés dans un attribut de type JSON, ceci permet de la représenter sous la forme une structure non stricte. Les champs obligatoires de la configuration sont quant à eux défini comme attributs directs de l'entité, ceci pour pouvoir y permettre l'accès à des fins de recherches (filtres), étant donné qu'effectuer des requêtes de sélection sur des champs JSON est moins efficace et généralement plus complexe au niveau syntaxique.
- ***Screen*** : entité représentant un écran, sur lequel seront disposés des ***Slots*** sous forme d'une grille.
- ***Slot*** : entité représentant un emplacement de la grille du tableau de visualisation, chaque slot contient l'affichage d'un ***Module*** spécifique et est affiché sur un ***Screen***.

Ce modèle est voué à être enrichi ultérieurement en cas de besoins nécessaires à l'implémentation de l'application, notamment au niveau de l'authentification et de la gestion des grilles d'affichage du tableau de bord.

2.4.2 Synthèse

Le schéma suivant récapitule le concept global de l'architecture selon les critères définis dans les chapitres précédents :

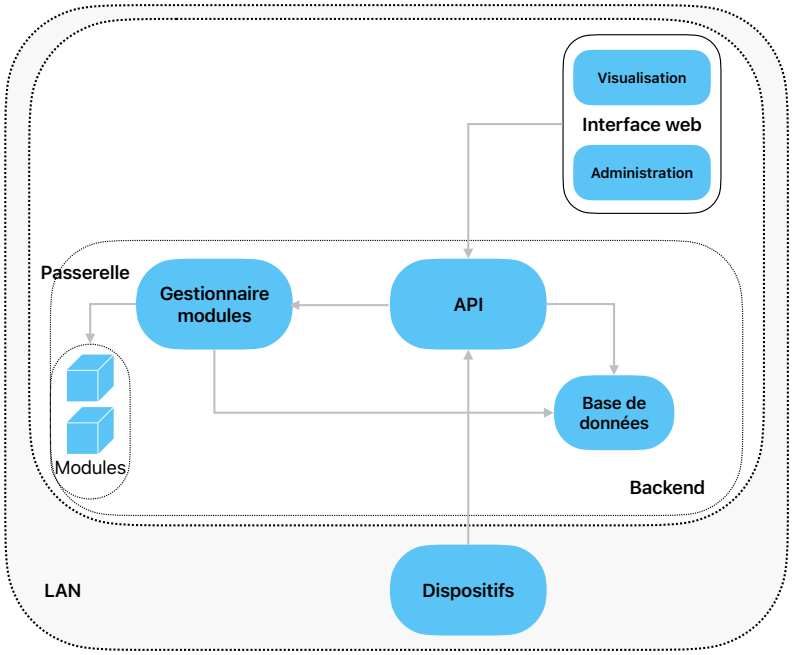


Figure 14: Architecture logique de l'application

2.5 Module « Proof Of Concept »

Un des objectifs de ce travail est de démontrer l'intérêt, les fonctionnalités et les limites de la solution proposée. Un des facteurs pouvant prouver ceci est la capacité à l'application d'intégrer des modules proposant des comportements variés, ayant chacun comme objectif de mettre en place des intégrations entre divers services.
Dans le cadre de ce projet, ceci sera démontré par l'implémentation de plusieurs modules, en mettant l'accent sur une intégration particulière exploitant au maximum les capacités de l'architecture. L'objectif est de proposer une intégration concrète avec le logiciel *Composal* de la société *YALK*. Ce module démontrera la capacité à pouvoir développer des modules permettant de créer des interactions entre des dispositifs internes (LAN) vers une application d'une entreprise (« *Proof Of Concept* » ou « *POC* »).

Quelques petits modules seront également développés en cours de projet, pour valider le fonctionnement des différents scénarios d'utilisation de la passerelle, à des fins de débogage et de tests. Si le temps le permet, d'autres modules plus concrets pourront être développés.

2.5.1 Intégration avec le logiciel Composal

Pour rappel, *Composal* est une application développée par l'entreprise *YALK*, ce logiciel est spécialisé dans la gestion du planning et des interventions pour les entreprises.
Le logiciel *Composal* est doté d'une fonctionnalité de timbrage. Lorsqu'un employé arrive au travail ou sur une intervention, il se rend sur l'application pour démarrer sa session de travail. Son temps de travail est ainsi chronométré jusqu'au prochain timbrage ou sa modification sur l'application.

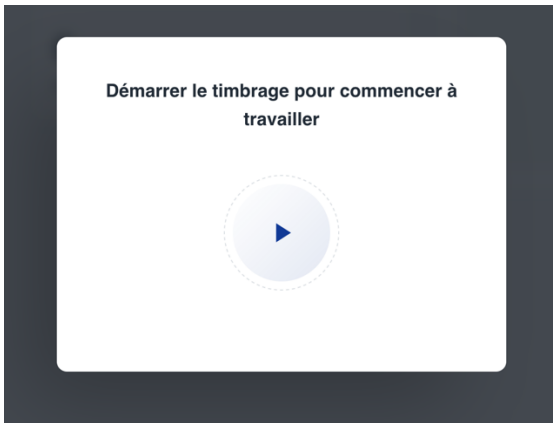


Figure 15: Timbrage sur l'application Composal

L'objectif de ce module *POC* sera de proposer l'intégration d'une timbreuse physique, fonctionnant sur la base d'un lecteur de carte RFID. Cette timbreuse, reliée par USB à la passerelle, lira la carte de l'employé et transmettra les données à la passerelle. Cette carte contient un identifiant unique, permettant d'identifier l'utilisateur dans le système *Composal* (cette fonctionnalité d'attribution d'un identifiant RFID à un utilisateur existe déjà dans l'application).

Le module sera ensuite chargé de lire et de traiter les données du lecteur puis d'envoyer les requêtes nécessaires sur l'API de *Composal* pour déclencher le timbrage de l'utilisateur concerné. Si ce dernier a déjà timbré en ce jour, la requête clôturera la période de travail. L'API de *Composal* propose déjà des endpoints permettant de d'effectuer le timbrage d'un utilisateur.

Figure 16: Lecteur RFID avec carte

Dans l'objectif d'exploiter au plus possible les fonctionnalités offertes par la passerelle, notamment les fonctionnalités de visualisation, l'affichage du module devra afficher les informations de l'utilisateur venant de timbrer et pourra afficher divers messages personnalisés.

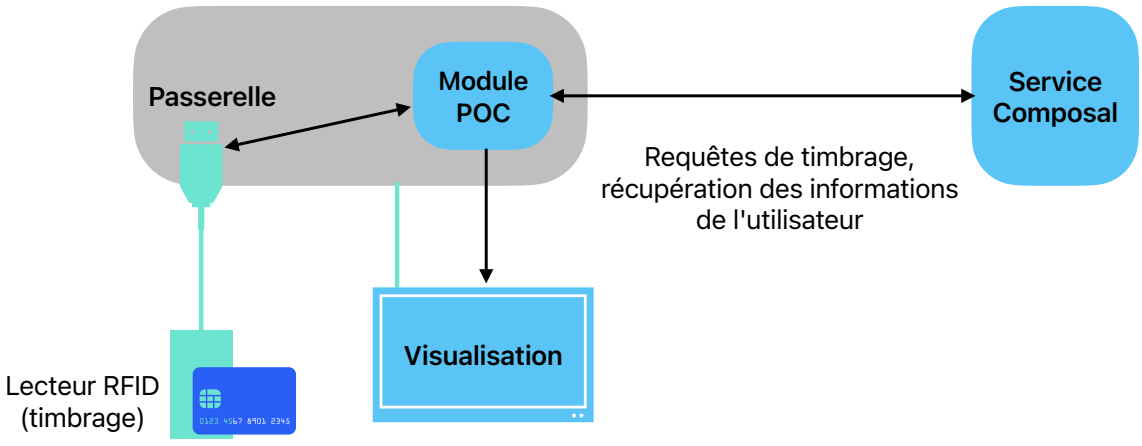


Figure 17: Idéalisation du module "Proof Of Concept"

Parmi les intégrations proposées par l'entreprise YALK, ce module est celui qui permet d'utiliser pleinement l'architecture de la passerelle, en utilisation un dispositif local (USB) et un service externe. D'autre idées de modules ont également été proposées :

Interventions en cours

Le principe est d'avoir un module permettant de visualiser les interventions du logiciel *Composal*. Cela permet, à la manière d'un outil de « ticketing », de visualiser rapidement les changements de statut des interventions dès leurs modifications dans l'application.

Ce module peut également se décliner sur plusieurs fonctionnalités de *Composal*, tels que les devis, planning journalier, etc.

2.6 Planification

Le projet se déroule du lundi 20 février au juillet 27 juillet 2023. Une présentation viendra finalement clôturer ce travail en septembre 2023.

2.6.1 Livrables

Plusieurs livrables sont attendus :

- Un rapport intermédiaire, détaillant le travail produit jusqu'à cette étape, au 26 mai.
- Un rapport final, détaillant le travail complet du projet, au 27 juillet.
- Un résumé publiable ainsi qu'une affiche, permettant de présenter et verbaliser le projet.
- L'application, selon les critères explicités précédemment.

2.6.2 Étapes

J'ai choisi de travailler sous forme d'étapes basées sur les diverses fonctionnalités de l'application. Ces étapes, généralement d'une durée de 1 à 3 semaines, planifiées tout au long du projet, me permettent d'avancer rapidement en priorisant les fonctionnalités principales définies de l'application.
Il s'agit d'un bon moyen pour prioriser les fonctionnalités ainsi que pour faire un retour concret au mandant et au professeur à chaque fin d'étape. Après avoir isolé tous les besoins (point 2.1), j'ai choisi de décomposer le projet selon le modèle suivant :

Étape 1 *– 20.02 au 03.03 – Cahier des charges*
Cette première étape consistera à établir les besoins avec le mandant, de manière à rédiger un cahier des charges du projet.

Étape 2 *– 04.03 au 17.03 – Architecture et outils*
L'objectif sera d'analyser les besoins pour en extraire les différents composants de l'application et de mettre en place une première architecture. Il s'agira ensuite de choisir les technologies et de créer la base de code utilisée pour le projet, ainsi que la mise en place des différents outils.

Étape 3 *– 18.03 au 07.04 – Prototypages, test et analyse*
Dans cette itération, l'objectif sera de d'effectuer différents prototypes, visant à évaluer la faisabilité et d'obtenir une base d'architecture exploitable. Il sera potentiellement nécessaire de comparer plusieurs approches et de comparer leur intérêt.

Étape 4 *– 08.04 au 21.04 – Modules*
L'étape portera sur la modélisation et l'implémentation des fonctionnalités relatives aux modules. Étant l'élément central du projet, la bonne conception de cet aspect est très critique. La création d'un premier module simple pour prouver le fonctionnement de l'intégration de modules est à prévoir.

Étape 5 *– 22.04 au 14.05 – API et backend*
L'accent sera mis sur les fonctionnalités côté serveur, telles que la définition de l'API, l'intégration de la base de données et l'authentification. Le développement d'un module « POC » sera effectué.

Étape 6 *– 15.05 au 26.05 – Refactoring et rapport intermédiaire*
Cette courte étape permettra d'effectuer du refactoring du code existant, puis de terminer la rédaction du rapport intermédiaire de projet. Le rendu de ce rapport marquera le passage à l'étape suivante.

Étape 7 *– 27.05 au 18.06 – Frontend*
L'objectif sera d'assembler l'interface de manière à ce qu'elle offre les fonctionnalités nécessaires qui n'ont pas encore été complètement implémentées dans les sprints précédents. Ceci comprendra entre autres la mise en place de l'authentification ainsi qu'en grande partie la configuration du tableau de bord et son affichage sur l'interface de visualisation. Le 18 juin marque la fin du semestre académique, le temps attribué à la réalisation du projet sera désormais de 5 jours par semaine.

Étape 8 *– 19.06 au 21.06 – Refactoring et documentation*
Le début du travail à plein temps sur le projet sera une bonne occasion de prendre du recul sur le travail accompli jusqu'à présent et d'effectuer une session de refactoring et de documentation.

Étape 9 *– 22.06 au 09.06 – Finalisation et tests*
Cette étape permettra de terminer les fonctionnalités si nécessaire, et d'effectuer des tests globaux sur l'application afin de pouvoir juger sa performance et sa capacité et s'exécuter de manière autonome. Le temps à disposition permettra également la rédaction de la documentation et du rapport de projet.

Étape 10 *– 10.06 au 27.07 – Livrables*
L'objectif cet ultime étape sera de clôturer le projet en retravaillant les éventuels derniers détails d'implémentation, mais principalement la finalisation de rédaction du rapport final et des autres livrables attendus.

2.7 Prototypes et essais effectués

Durant cette phase initiale d'analyse, j'ai effectué une série d'essais d'architecture et de technologies relativement variés. Ces essais, plus au moins concluants, m'ont permis d'évaluer et d'assurer mes choix de technologies et d'architecture finaux. Ce chapitre détaille les différents prototypes réalisés pour analyser la faisabilité des différentes approches.

Deux architectures pouvant répondre aux différents besoins de l'application ont été prototypées afin de départager le choix final. Ces deux architectures répondent au besoin de fonctionnement de manière locale sur un réseau privé de l'utilisateur, sans dépendre de la conception de services externes à ce réseau et permettent de répondre aux besoins finaux de l'application. Afin de pouvoir tester les différentes architectures, une première version de l'implémentation des modules et du gestionnaire de modules a été prototypée afin d'être intégré dans une architecture quelconque.

2.7.1 Architecture serveur HTTP

La première approche envisagée consiste en une architecture relativement classique, composée d'un serveur HTTP, offrant une interface web ainsi qu'une API. Ce serveur, destiné à s'exécuter sur la passerelle, offre une API pour gérer les intégrations avec l'interface utilisateur et également de permettre à des dispositifs locaux (LAN) de communiquer avec la passerelle au travers du protocole HTTP.
L'objectif de ce premier prototype est d'établir une communication entre l'interface web et l'API permettant de démontrer le fonctionnement des modules et de leur affichage.
Pour concevoir cette architecture, j'ai utilisé le Framework JavaScript Express.js offrant le nécessaire à la mise ne place rapide d'un serveur HTTP.
Dans ce prototype, un simple module a été mis en place afin de tester son intégration au système de module et de récupérer son affichage. Pour faciliter le prototype, l'affichage du module est défini au format HTML et ne contient pas de contenu dynamique.

Le serveur propose deux endpoints :

- / : affiche l'interface, un simple fichier HTML dont le rôle est d'effectuer une requête pour la récupération de l'affichage du module
- /module/<id> : Retourne l'affichage du module au format HTML

Cette interaction a pu facilement être mise en place en utilisant l'API Fetch[6] du navigateur. Elle n'est cependant pas vraiment adaptée à l'utilisation souhaitée. Pour simuler un module dont l'état pouvant changer et notifier le besoin de rafraichir son affichage, j'ai ajouté une boucle interne au module s'exécutant toutes les secondes et notifiant un changement d'état du module.
Le client doit donc être capable de récupérer l'affichage à jour du module, pour cela il est nécessaire que le client soit notifié de la disponibilité du nouvel affichage et que celui-ci lui soit transmis directement.

[6] https://developer.mozilla.org/fr/docs/Web/API/Fetch_API/Using_Fetch

Avec HTTP, il existe généralement quatre approches pouvant répondre à ce besoin de communications en temps réel :

Polling

Le polling consiste en l'envoi de requêtes HTTP du client vers le serveur à une intervalle de temps régulier afin de récupérer de nouvelles données. Cette approche est très simple à implémenter, en voici un exemple avec l'API Fetch du navigateur :

```
async function subscribe() {
  setInterval(async () => {
    const response = await fetch('https://api.example.com/data')
    const data = await response.json()
    // Traitement des données
  }, 5000)
}
```

Listing 2: HTTP Polling, codec client

Le client va ainsi envoyer une nouvelle requête toutes les 5 secondes au serveur pour récupérer de nouvelles données.

Il est évident que cette approche comporte trop d'inconvénients pour être viable dans une application conséquente : le client va envoyer une nouvelle requête au serveur toutes les cinq secondes, cela même si de nouvelles données ne sont pas disponibles sur le serveur. Ceci aura tendance à générer une surcharge du réseau et d'utilisation de ressources sur le serveur. Il est également nécessaire d'attendre cinq secondes pour obtenir les nouvelles données au travers d'une requête, ce qui crée une latence d'actualisation des données sur le client.

On en retient donc que l'amélioration idéale serait que le serveur soit responsable de notifier le client de la disponibilité de nouvelles données.

Long polling

Le client initie une connexion avec le serveur dans l'attente d'une réponse, cette connexion est maintenue ouverte tant que le serveur n'a pas envoyé de réponse, c'est-à-dire lorsque celui-ci dispose de nouvelles données qu'il souhaite transmettre au client. Lors de la réponse du serveur, ou d'un timeout, le client récupère la réponse et initie à nouveau une connexion avec le serveur pour être notifié lors de la prochaine mise à jour.

L'implémentation de cette approche peut se faire de manière relativement naïve en utilisant l'API Fetch du navigateur :

```
async function subscribe() {
  let response = await fetch("https://example.com/endpoint")

  if (response.status == 502) {
    // Timeout, tentative de reconnexion
    await subscribe()
  } else {
    // Données reçues
    let message = await response.text()
    // Nouvelle connexion
    await subscribe()
  }
}
```

Listing 3: HTTP Long Polling, code client

En comparaison avec le polling, le long polling saturera moins la bande passante et souffrira moins de latence, du fait du nombre moins important de requêtes envoyée vers le serveur. Cependant, le serveur doit maintenir ouvertes les connexions, ce qui devient problématique lorsqu'un grand nombre de clients sont connectés.

Server Sent Event (SSE)

Dans cette approche, le client initie une connexion avec le serveur, cette connexion est maintenue ouverte jusqu'à sa fermeture explicite par le client ou le serveur. Dès cet instant, le serveur pourra envoyer des mises à jour au client, sous forme d'événements. Cette approche propose donc une communication unidirectionnelle, le client ne peut pas envoyer de données sur cette connexion établie.

L'API du navigateur[7] propose une interface simple pour mettre en place une communication SSE, en utilisant l'API EventSource :

```
const evtSource = new EventSource("https://example.com/events")

evtSource.onmessage = (e) => {
  // Traitement des données
  console.log(`message: ${e.data}`)
}
```

Listing 4: Server Sent Events, code client

[7] https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events

Du côté serveur, il est nécessaire de respecter certaines contraintes, comme la définition d'entêtes et le respect d'une structure de message, comme démontré dans cet exemple :

```
res.setHeader('Content-Type', 'text/event-stream')
res.setHeader('Cache-Control', 'no-cache')
res.setHeader('Connection', 'keep-alive')

// L'envoi de données au client se fait grâce à la méthode write
// Le contenu doit être préfixé par data : et terminé par \n\n
res.write('data: Connected\n\n')
```

Listing 5: Server Sent Events, code serveur

Contrairement aux approches précédentes, les SSE permettent un flux en temps réel, ceci de manière unidirectionnelle car les évènements sont déclenchés par le serveur uniquement. La mise en œuvre est relativement simple au vu de l'API offerte par le navigateur. L'utilisation de la bande passante est largement réduite (une seule connexion par client). Les SSE offrent également un processus de reconnexion automatique au serveur en cas de perte de connexion par le client.

Cette approche est viable pour l'utilisation dans ce projet, l'utilisation des SSE permet aux clients de s'abonner aux évènements des différents modules au travers de l'API, ceci sans aucune latence. Les affichages des modules seront ainsi uniquement rafraichi lors ce que cela est nécessaire. Étant donné que les affichages des modules sont destinées à être affichés sur l'interface de visualisation qui n'offre pas d'interactions utilisateur, l'aspect unidirectionnel des SSE n'est pas un problème.

WebSockets

A la différence des SSE, les WebSockets permettent une communication bidirectionnelle entre le client et le serveur. Les WebSockets ne font pas nativement partie du protocole HTTP, ils définissent une couche supplémentaire : lors de l'initialisation d'une connexion, une augmentation du protocole (HTTP Upgrade) est demandée et un handshake doit également être effectué (utilisation du protocole WS).
Le navigateur offre une *API* permettant de mettre en place la connexion au travers de WebSockets :

```
const socket = new WebSocket('ws://example.com/events')

socket.addEventListener('open', () => {
  // Connexion établie, envoi d'un message
  socket.send('example message')
})

socket.addEventListener('message', (event) => {
  const message = event.data
  // Réception d'un message et traitement
})
```

```
socket.addEventListener('close', () => {
  // Connexion fermée
})
```

Listing 6: WebSockets, code client

L'implémentation gagne cependant en complexité du côté serveur, ceci est du fait que les WebSockets utilisent leur propre protocole (WS), il est donc souvent nécessaire d'utiliser une librairie externe pour les implémenter.

Les WebSockets reprennent les avantages des SSE en y ajoutant cette fois la communication bidirectionnelle au travers d'une unique connexion. La complexité de l'implémentation des WebSockets est relativement plus grande que les SSE.

Tout comme l'approche SSE, les WebSockets conviennent au cas d'utilisation de ce projet, permettant la même approche en y ajoutant la possibilité de gérer des évènements envoyés par des affichage des modules de l'interface de visualisation.

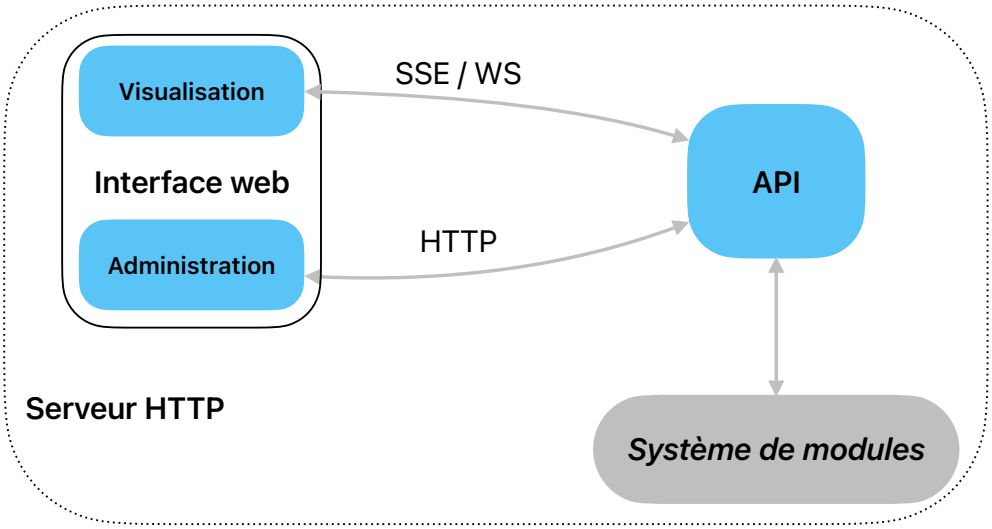


Figure 18: Prototype, architecture Server Side Events – WebSocket

2.7.2 Architecture Electron.js

La seconde approche consiste en l'utilisation de la plateforme Electron.js. L'utilité première d'Electron et de permettre de créer des applications multiplateformes en utilisant un écosystème de technologies web, telles que HTML, JavaScript, Node.js et divers Frameworks comme React ou Vue.js.
Une application Electron est basée sur Chromium, l'application s'exécute donc dans un navigateur sous la forme d'une application native sur un poste client.

N'étant pas familier avec cet écosystème, j'ai choisi de réaliser un prototype de mon application afin d'évaluer si une architecture correspondant aux attentes du projet pouvait être réalisé avec Electron.

L'environnement Electron impose une architecture stricte, notamment en différenciant de manière très distinctes les composants ayant accès à l'environnement Node.js. On distingue notamment une séparation entre les composants Process (coté serveur) et les Renderers (côté client) :

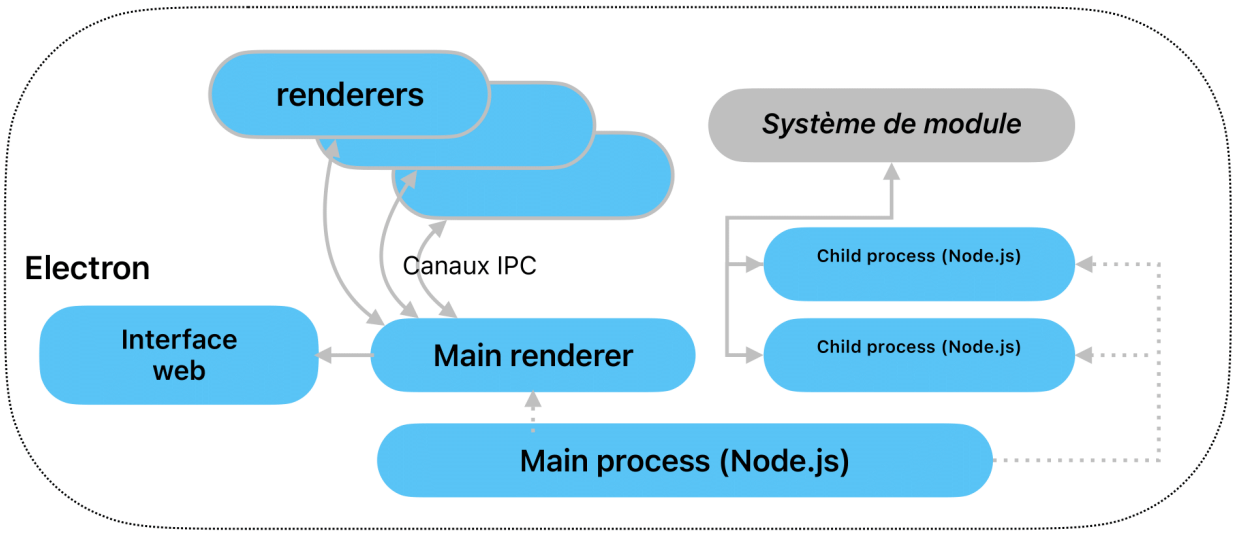


Figure 19: Prototype, architecture Electron

Les Process sont tout simplement des processus Node.js, chaque tâche nécessitant l'utilisation de ressources plus ou moins importante doit être exécuté dans un processus enfant. Dans le cas de ce projet, chaque module actif de la passerelle doit ainsi être exécuté dans son propre contexte.

Au niveau de l'interface, l'architecture est également séparée en plusieurs processus de rendu (renderer). Chacun de ces renderers s'exécute dans un contexte différent, sous la forme d'un onglet caché du navigateur, de ce fait les interactions directes entre les processus de rendus ne sont pas possible. Pour communiquer entre les processus et les rendus, Electron utilise des canaux interprocessus (IPC). Un IPC dans Electron est un mécanisme de communication permettant l'échange de données entre les différents processus de rendu et le main process, sous la forme de canaux bidirectionnels synchrone et asynchrone.

L'approche Electron propose des avantages intéressants dans le cadre de ce projet :

- Auto-updater : permet à l'application se mettre à jour automatiquement selon un dépôt distant
- Application packagée dans un unique exécutable : ceci facilite grandement le déploiement et l'installation de l'application sur la passerelle
- Une bonne isolation des contextes d'exécution de chaque module grâce à l'architecture imposée.

Sur ces points cités, cette architecture semblait prometteuse, mais les contraintes et inconvénients qu'impose Electron risquaient de pouvoir à tout moment menacer le bon déroulement du projet en cas de non-compatibilité des directives Electron, relativement strictes, avec l'entièreté des fonctionnalités de ce projet. Ce cas est d'ailleurs arrivé lors du prototype, car l'architecture d'Electron ne permet pas d'exposer l'application sur réseau, il est donc nécessaire s'ajouter un serveur HTTP pour rendre l'interface et l'API accessible. L'ajout de ce composant rend l'architecture de l'application incohérente et va à l'encontre de l'utilisation prévue d'Electron.

2.7.3 Choix

Comme décrit au point précédent, Les limitations de l'architecture Electron ont donc été atteinte au stade de prototype et ne remplissent pas les critères et fonctionnalités imposés par le projet. La première architecture, plus classique mais surtout moins restrictive a été retenue comme approche pour l'implémentation du projet.

Dans ce premier prototype, deux approches de communication se démarquent et correspondent au besoin de l'application : les Server Sent Events et les WebSockets. L'approche préférée a été celle de l'utilisation d'une communication unidirectionnelle orientée serveur, offerte par les SSE, tout en sachant que s'il s'avérait nécessaire de gérer des interactions de l'utilisateur avec l'interface de visualisation en cours de projet, l'ajout des WebSockets ne serait pas un élément bloquant et un travail trop laborieux.

2.8 Choix technologiques

2.8.1 Choix généraux

Pour l'implémentation de ce projet, j'ai choisi de travailler avec le langage TypeScript. TypeScript est une extension de JavaScript qui ajoute des fonctionnalités de typage statique au langage. Les avantages d'un typage statique sont bien connus : les erreurs sont détectées plus rapidement (à la compilation), la robustesse, la lisibilité et la maintenabilité du code en sont grandement améliorées. Grâce à cela, TypeScript corrige de nombreux problèmes que l'on peut rencontrer lors du développement avec JavaScript, ces avantages deviennent encore plus appréciables lors que l'application gagne en complexité.
Étant donné que TypeScript est une extension de JavaScript, il est un langage idéal dans la conception d'application web, principalement côté client, mais également du côté serveur. Ainsi, j'ai choisi d'utiliser ce langage pour l'entièreté du projet, de manière à conserver un écosystème cohérent et à pouvoir profiter de technologies et librairies communes au backend et au frontend.
De manière globale au projet, j'utilise la librairie de testing Jest. Il s'agit d'une librairie très complète et facile d'utilisation pour établir des suites de test automatisés et s'assurer d'un bon fonctionnement de l'application.

2.8.2 Backend

2.8.2.1 Node.js

Pour le développement d'application JavaScript / TypeScript côté serveur, il est nécessaire d'utiliser l'environnement d'exécution Node.js. Node.js est basé sur le moteur JavaScript V8 et offre un écosystème extrêmement complet, notamment réputé pour sa large collection de modules et sa performance. Node.js fonctionne sur un modèle d'asynchronisme, ce qui permet de gérer efficacement de nombreuses connexions simultanées sans bloquer le processus principal, ceci est particulièrement apprécié dans les applications web. Les fonctionnalités d'accès natif et unifié aux API du système d'exploitation est également un point fort de cet environnement.

2.8.2.2 Express.js

Pour implémenter le serveur HTTP, responsable de servir l'application ainsi que l'API au réseau local, j'ai choisi le Framework Express.js. Offrant toutes les fonctionnalités nécessaires à la conception d'une application web tout en étant très léger et simple d'utilisation, Express est un outil majoritaire adopté par la communauté Node.js, ce qui en fait un choix pertinent pour ce projet.
Bien qu'il existe de nombreux Framework Node.js offrant un bien plus grand écosystème (Nest.js, pour n'en citer qu'un), la simplicité et la flexibilité ont été préférée dans ce projet, afin de garder un contrôle de la courbe d'apprentissage liée à ce genre de Framework.

2.8.2.3 SQLite

Comme défini au point *2.4* de l'analyse, l'approche adoptée pour la conception de la base de données dans l'application est l'approche relationnelle. Le choix du système de gestion de bases de données relationnelle (SGBDR) s'est ainsi porté sur SQLite. L'intérêt de SQLite est de ne pas reposer sur un serveur, comme sont généralement conçus les autres SGBDR (PostgreSQL,

MySQL) mais d'être directement intégrable dans un logiciel sous forme d'un unique fichier, indépendamment de l'environnement d'exécution du programme. Ainsi, un important gain de ressource est obtenu grâce à l'utilisation de SQLite. L'application ayant un besoin d'autonomie et de s'exécuter sur une architecture relativement limitée en termes de ressources, le choix de SQLite, dont les objectifs répondent parfaitement aux besoins de l'application, est un choix évident et parfaitement adapté pour ce projet.

2.8.3 Frontend

2.8.3.1 React

React est une librairie JavaScript open source, conçue pour faciliter la création d'interface utilisateur. Les application React sont dites SPA (Single Page App) et se démarquent par les aspects suivants : Une SPA s'exécute entièrement sur le navigateur du client. Un seul fichier HTML est chargé par le navigateur, ceci explique le terme de Single Page App. Les interactions, la modification des données et la navigation s'effectuent ainsi sans rechargement de page grâce à un système de rendu virtuel.
L'approche de React est dite Component Based Development. Il s'agit d'un paradigme dans laquelle une application est conçue en assemblant des composants, dans l'objectif que chaque composant soit réutilisable, facilement maintenable et évolutif. React se différencie des autres Framework SPA comme Vue.js ou Angular, car il se défini plutôt comme une collection d'outils permettant de concevoir des composant, plutôt qu'être un Framework complet définissant une structure et approche stricte. La popularité actuelle de React et son écosystème très riche en fond un choix logique et sûr pour la création d'application web client.

Un composant React est défini en tant que fonction, retournant un élément JSX. JSX (JavaScript XML) est une extension de la syntaxe HTML permettant d'intégrer du code JavaScript et HTML dans un même fichier, ceci permettant d'avoir du contenu, des attributs et du style dynamique en fonction de l'état (données) d'un composant. Voici un exemple d'un composant React simple, retournant un bloc JSX :

```jsx
export default function Example() {
  return (
    <section className='example-section'>
      <h1>This is an example</h1>
      <OtherComponent property={'test'} />
    </section>
  )
}
```

Listing 7: JSX, composant React

Bien que la ressemblance entre le JSX et HTML soit forte, il est nécessaire de transformer le JSX en JavaScript afin que les navigateurs puissent l'interpréter. JSX n'est pas considéré un templating engine, mais plutôt comme une extension de syntaxe adoptée dans le Framework JavaScript comme React.

React offre également des fonctionnalités utilisable côté serveur, notamment le pré-rendu statique et le rendu côté serveur (SSR), permettant de transformer des composant React défini en JSX en contenu HTML statique afin de soulager une partie du travail de rendu du navigateur.

2.8.3.2 TailwindCSS et DaisyUI

Afin de rendre l'interface ergonomique et agréable d'utilisation tout en économisant du temps sur son implémentation, TailwindCSS a été choisi comme Framework CSS pour le développement de l'interface utilisateur. TailwindCSS adopte une approche dite utilitaire, à l'inverse des approches de composant prédéfinis généralement offertes par les Frameworks CSS concurrents (Bootstrap, Bulma...). Cette approche consiste à utiliser des classes CSS individuelles fournie par le Framework, plutôt que de choisir des composants parmi une collection prédéfinie. On obtient donc une liberté et personnalisation très flexible et un résultat visuel se distinguant des composants prédéfinis.
Le choix de TailwindCSS relève surtout d'un choix personnel, ayant déjà travaillé avec et préférant cette approche utilitaire. Afin de gagner du temps sur la conception de l'interface, le plugin DaisyUI a été ajouté à TailwindCSS afin d'avoir à disposition quelques composants prédéfinis offrant néanmoins une personnalisation totale.

Voici un exemple d'un bouton défini avec TailwindCSS :

```
<button class="inline-block cursor-pointer rounded-md bg-gray-800 px-4 py-3
text-center text-sm font-semibold uppercase text-white transition duration-200
ease-in-out hover:bg-gray-900">Button</button>
```

Listing 8: Exemple de l'approche utilitaire, Tailwind CSS

Et avec DaisyUI:

```
<button class="btn">Button</button>
```

Listing 9: Équivalant du listing 8, Daisy UI

Le résultat sera identique avec les deux approches, On constate donc que DaisyUI permet d'alléger le nombre de classe utilitaire à définir :

Figure 20: Résultat commun aux deux approches

2.9 Méthodologies et outils

Ce chapitre explore les aspects relatifs aux besoins non fonctionnels et aux méthodologies logicielles et de travail adopté durant la réalisation du projet.

2.9.1 Monorepo

L'approche Monorepo (monolithic repository) est une manière d'organiser la structure d'un projet, de façon à ce que tous les composants de l'applications soient regroupés sur un unique dépôt de code (repository). Cette approche offre beaucoup d'avantages dans le cadre de ce projet : la gestion centralisée facilite le partage des dépendances et de pouvoir travailler sur tous les composants de manière globale, sans avoir besoin de publier de nouvelles versions des paquets pour pouvoir les utiliser comme dépendances.

L'écosystème Node.js et son gestionnaire de packet npm permet de mettre en place ce principe de monorepo grâce aux npm Workspaces. Les workspaces permettent d'isoler des différents modules afin que chacun puisse avoir ses propres dépendances, mais également de regrouper les dépendances communes, ce qui réduit la taille de dépendances finales. Cela permet également d'exécuter des commandes (test, build, etc) sur tous les modules de manière centralisée. Les workspaces permettent tout de même de publier chacun des modules de manière indépendante.

2.9.2 Contrôle de version

Pour la gestion du code et de ses différentes versions, j'ai utilisé le système Git, ainsi qu'un référenciel distant (repository) hébergé sur la plateforme GitHub. Il existe beaucoup d'approche de gestion des branches Git pour le développement logiciel, chacun pouvant mieux correspondre à certains cas d'utilisation (taille de l'équipe, type de logiciel etc.). Parmi celles-ci, j'ai adopté l'approche *Trunk based*. Cette approche repose sur l'utilisation principal de la branche principale (master / main), appelé *trunk*. L'objectif est d'itérer très rapidement sur des petites fonctionnalités, en travaillant directement sur le *trunk* ou dans certains cas en utilisant des branches temporaires à très court terme. L'intérêt de cette solution est d'avoir un état de l'application mis en commun le plus souvent afin d'éviter les problèmes de conflit lors de *merge*. Cette approche est idéale pour les petites équipes, et donc convient parfaitement pour ce projet. Son succès repose également sur le concept d'intégration continue, puisque les mise en commun fréquentes du code permettent de délivrer fréquemment l'application

2.9.3 CI / CD

Sur le repository GitHub, un pipeline d'intégration continue (CI) a été mis en place grâce à l'utilisation des GitHub Actions[8]. L'intérêt du pipeline est de s'assurer du bon fonctionnement du code lors de sa publication sur le repository, grâce à l'exécution d'une série d'instructions définie,

[8] https://github.com/features/actions

tels que la compilation et l'exécutions des tests automatisés. Le pipeline comporte également un aspect de livraison continue (CD), permettant de publier le résultat de le CI sous la forme d'un image *Docker*. *Docker* est une plateforme permettant de d'exécuter des applications dans des containers isolés, contenant le nécessaire au bon fonctionnement de l'application, ce qui les rends autonomes. L'approche choisie dans ce projet et de créer une image (modèle de référence permettant de créer un container Docker) à l'issue de la phase d'intégration et de la publier sur une référentiel distant (DockerHub) afin que celle-ci puisse être récupérée par les utilisateurs. Le pipeline mis en place sur le repository du projet se présente de la manière suivante :

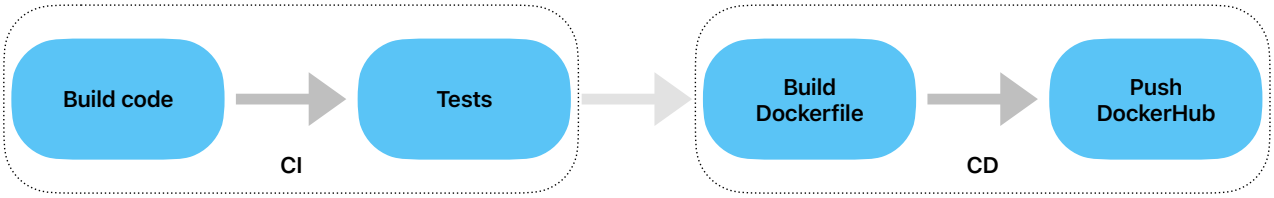


Figure 21: Pipeline CI/CD

2.9.4 Production

2.9.4.1 Support physique

L'application est destinée à s'exécuter sur une passerelle physique, installée dans un réseau local. Dans le cadre de ce projet, cette passerelle utilise comme matériel un Raspberry PI 4 Model B[9], disposant des caractéristiques suivantes :

- Système d'exploitation Raspberry PI OS, distribution Linux basée sur Debian.
- 8 GB RAM, Stockage flash sur carte micro-SD (128GB)
- Alimentation par USB-C
- 2x micro-HDMI
- 4x USB
- Connectivité Wi-Fi, Ethernet, Bluetooth,

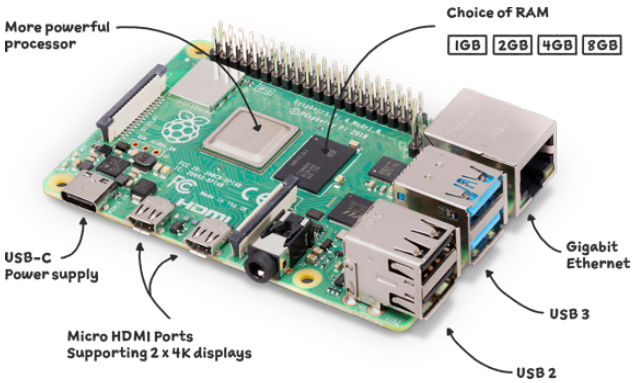


Figure 22: Raspberry Pi 4 Model B, présentation du constructeur

[9] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

L'avantage de cette machine est son format réduit permettant une installation facile dans des locaux d'une entreprise et l'embarquement de toutes les connectiques standards de manière native et physique (Bluetooth, Wifi, USB...), tout ceci à un prix relativement abordable.

Bien entendu, l'application n'est pas conçue de manière ne s'exécuter que restrictivement sur un Raspberry PI. Tout ordinateur peut potentiellement exécuter l'application en disposant des dépendances nécessaires.

2.9.4.2 Environnement d'exécution

Afin d'exécuter l'application sur la plateforme de production, plusieurs approches sont possibles. Avant d'énumérer les possibilités, il est intéressant de démontrer les transformations nécessaires du code sources jusqu'à un état utilisable en production.

Au niveau du frontend, les applications SPA React suivent généralement un processus de transformation assez complexe. Le code JavaScript et JSX est transformé en JavaScript standard, grâce à un outil appelé le transpileur. Cette transpilation est nécessaire pour assurer que tous les navigateurs puissent interpréter le code source de manière identique. En effet, le langage JavaScript à tendance à évoluer plus vite que le son support par les navigateurs, certains éléments de syntaxe ne sont pas pris en compte par tous les navigateurs.
Une fois cette étape achevée, les dépendances sont injectées et le code peut être optimisé grâce à divers mécanismes, comme la compression et la minification (réduction de la tailles des fichiers en effaçant les espaces, supprimant les commentaires et raccourcissement des noms de variables).

Le résultat final est généralement composé d'un unique fichier HTML, un fichier JavaScript et les autres ressources statiques (CSS, images). Ceci en fait des applications faciles à déployer en production puisque qu'il est uniquement nécessaire de disposer d'un simple serveur HTTP, nécessitant peu de configuration.

Côté backend, les applications Node.js ne sont généralement pas transformée, étant donné qu'il s'agit d'un langage interprété. Cependant, TypeScript est utilisé dans ce projet. Le sur-ensemble TypeScript n'est utilisé que pour le développement, car les navigateur et l'environnement Node.js ne sont pas capable d'interpréter cette syntaxe. Il est donc nécessaire de compiler le code TypeScript vers son équivalant JavaScript. Cette compilation apporte une analyse syntaxique, sémantique ainsi qu'un vérification des types dans toute la base de code.

L'application peut ainsi être déployée en un seul composant, étant donné que le frontend ne se résume qu'à quelques fichiers statiques, le backend pourra les rendre accessible directement, au travers du serveur HTTP Express.js, pour que l'interface soit accessible sur le réseau local.

Revenons aux alternatives d'exécution du code sur l'environnement de production. Il existe principalement deux axes possibles :

Exécution sur OS hôte
L'application est exécutée directement sur le système d'exploitation de l'hôte. Avec *Node.js*, cela impose généralement l'utilisation d'un gestionnaire de processus. Le plus populaire s'appelle *PM2*[10], il permet d'exécuter l'application sous la forme d'un *deamon* et de faciliter son exécution au démarrage du système. C'est une solution fiable et facile à mettre en place, mais elle demande d'installer un certain nombre de dépendances directement sur la machine hôte :

- L'environnement *Node.js* et son gestionnaire de paquets *NPM*, nécessaires à l'exécution, l'installation des dépendances et de la compilation
- *PM2* pour l'exécution de l'application en tant que service

Exécution dans un container
L'application est exécutée dans un container Docker contenant tout l'écosystème Node.js et les dépendances nécessaires. Le container peut être créé à partir de la définition d'une image, celle-ci est généré en fin de pipeline CI/CD (voir 2.9.3).

Les containers sont des instances pouvant être détruite et récrée à souhait, ils ne sont donc pas conçus pour conserver des données de manière persistantes. De base, les containers n'ont pas accès au système hôte, Docker embarque cependant un outil appelé « volumes » qui permettent d'explicitement mettre à disposition un répertoire du système d'exploitation au container. Ceci est par exemple utilisé pour conserver les données devant persister au remplacement du container (par exemple lors du déploiement d'une nouvelle version d'application), notamment la base de données et le modules.
Cette solution offre demande moins de configuration quand la solution précédente, puisqu'il suffit d'installer uniquement Docker sur l'environnement de production.

Les deux approches sont tout à fait viables et cohérentes, j'ai cependant privilégié la seconde approche. Ce choix a principalement été motivé par la simplicité de l'installation, en terme du nombre de dépendance. Il sera également plus simple de déployer des nouvelles versions, car il suffira de remplacer le container par la nouvelle version, dont l'image de référence est créée automatiquement par le pipeline d'intégration.

Il sera cependant nécessaire de revalider ce choix en évaluant les possibilités d'accès à la connectique Bluetooth ou USB depuis un container, afin d'assurer que les modules nécessitant ces technologies puissent fonctionner.

[10] https://pm2.keymetrics.io/

Chapitre 3
Implémentation

La deuxième phase du travail a consisté en la modélisation et l'implémentation concrète des différentes fonctionnalités définies lors de la phase d'analyse. Ce chapitre aborde les différents choix et détails d'implémentations effectués pour chacune des fonctionnalités.

A suivre...

Bibliographie

GAMMA, Erich., HELM, Richard., JOHNSON, Ralph., VLISSIDED, John., « Design Patterns: Elements of Reusable Object-Oriented Software », Addison-Wesley Professional, 1994

https://react.dev/
https://reactrouter.com/en/main

https://restfulapi.net/
https://tailwindcss.com/
https://daisyui.com/
https://www.raspberrypi.com/documentation/
https://docs.docker.com/
https://www.electronjs.org/fr/
https://sqlite.org/index.html
https://v8.dev/features/dynamic-import
https://www.typescriptlang.org/docs/
https://expressjs.com/
https://nodejs.org/en/docs
https://jwt.io/

Annexes

Glossaire

Index