

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

21 juillet 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 21 juillet 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 21 juillet 2023

# Résumé

Cliquez ou appuyez ici pour entrer du texte.

**Table des matières**

[Préambule I](#_Toc140847909)

[Authentification III](#_Toc140847910)

[Résumé V](#_Toc140847911)

[Table des figures X](#_Toc140847912)

[Liste des codes sources XII](#_Toc140847913)

[Chapitre 1 Introduction 1](#_Toc140847914)

[1.1 Contexte 1](#_Toc140847915)

[1.2 Cahier des charges 2](#_Toc140847916)

[1.2.1 Besoins fonctionnels 2](#_Toc140847917)

[1.2.2 Besoins non fonctionnels 3](#_Toc140847918)

[1.2.3 Extensions 4](#_Toc140847919)

[1.3 Contraintes client 4](#_Toc140847920)

[1.4 Solutions existantes 5](#_Toc140847921)

[Chapitre 2 Analyse 7](#_Toc140847922)

[2.1 Modules 7](#_Toc140847923)

[2.1.1 Comportement 8](#_Toc140847924)

[2.1.2 Configuration 10](#_Toc140847925)

[2.1.3 Affichage 13](#_Toc140847926)

[2.1.4 Modélisation 14](#_Toc140847927)

[2.1.5 Exemple 15](#_Toc140847928)

[2.2 Gestionnaire de modules 16](#_Toc140847929)

[2.3 Application Web 19](#_Toc140847930)

[2.3.1 Interface 20](#_Toc140847931)

[2.3.2 API 22](#_Toc140847932)

[2.4 Base de données 23](#_Toc140847933)

[2.4.1 Modèle conceptuel 24](#_Toc140847934)

[2.4.2 Synthèse 25](#_Toc140847935)

[2.5 Module « Proof Of Concept » 26](#_Toc140847936)

[2.5.1 Intégration avec le logiciel Composal 26](#_Toc140847937)

[2.6 Planification 29](#_Toc140847938)

[2.6.1 Livrables 29](#_Toc140847939)

[2.6.2 Étapes 29](#_Toc140847940)

[2.7 Prototypes et essais effectués 31](#_Toc140847941)

[2.7.1 Architecture serveur HTTP 31](#_Toc140847942)

[2.7.2 Architecture Electron.js 36](#_Toc140847943)

[2.7.3 Choix 37](#_Toc140847944)

[2.8 Choix technologiques 38](#_Toc140847945)

[2.8.1 Choix généraux 38](#_Toc140847946)

[2.8.2 Backend 38](#_Toc140847947)

[2.8.3 Frontend 39](#_Toc140847948)

[2.9 Méthodologies et outils 41](#_Toc140847949)

[2.9.1 Monorepo 41](#_Toc140847950)

[2.9.2 Contrôle de version 41](#_Toc140847951)

[2.9.3 CI / CD 41](#_Toc140847952)

[2.9.4 Production 42](#_Toc140847953)

[Chapitre 3 Modules 45](#_Toc140847954)

[3.1 Comportement 46](#_Toc140847955)

[3.2 Configuration 49](#_Toc140847956)

[3.2.1 Utilisation 52](#_Toc140847957)

[3.2.2 Validation 53](#_Toc140847958)

[3.3 Affichage 55](#_Toc140847959)

[3.3.1 Définition du composant 56](#_Toc140847960)

[3.3.2 Processus de rendu 57](#_Toc140847961)

[3.3.3 Style 58](#_Toc140847962)

[3.4 Accès aux dispositifs 60](#_Toc140847963)

[3.5 Gestionnaire de module 63](#_Toc140847964)

[3.5.1 Identifiant 63](#_Toc140847965)

[3.5.2 État et stockage 64](#_Toc140847966)

[3.6 Module d’exemple 66](#_Toc140847967)

[Chapitre 4 Application Web 67](#_Toc140847968)

[4.1 Backend 68](#_Toc140847969)

[4.1.1 Base de données 68](#_Toc140847970)

[4.1.2 API REST 72](#_Toc140847971)

[4.1.3 Authentification 73](#_Toc140847972)

[4.1.4 Architecture du code 77](#_Toc140847973)

[4.1.5 Évènements en temps réel 78](#_Toc140847974)

[4.2 Interface 80](#_Toc140847975)

[4.2.1 React 80](#_Toc140847976)

[4.2.2 Fonctionnalités 80](#_Toc140847977)

[4.2.3 Internationalisation 80](#_Toc140847978)

[4.3 Production 82](#_Toc140847979)

[Chapitre 5 Proof of Concept 83](#_Toc140847980)

[5.1 Matériel et ressources 83](#_Toc140847981)

[5.2 Module 83](#_Toc140847982)

[Chapitre 6 Bilan et perspectives 84](#_Toc140847983)

[Chapitre 7 Conclusion 85](#_Toc140847984)

[Bibliographie 87](#_Toc140847985)

[Annexes 88](#_Toc140847986)

# Table des figures

[Figure 1: Contexte 1](#_Toc140699330)

[Figure 2 : Structure d’un module 8](#_Toc140699331)

[Figure 3: Cycle de vie d'un module 8](#_Toc140699332)

[Figure 4 : Modélisation initiale de la classe Module 9](#_Toc140699333)

[Figure 5: Pattern Observer, selon le GoF 10](#_Toc140699334)

[Figure 6: Modélisation du domaine : configuration d'un module 12](#_Toc140699335)

[Figure 7: Séquence de "rendering" d'un "templating engine" 13](#_Toc140699336)

[Figure 8: Modélisation du domaine : module 14](#_Toc140699337)

[Figure 9: Module d'exemple, diagramme de séquence 15](#_Toc140699338)

[Figure 10: Modélisation du domaine : gestionnaire de modules 16](#_Toc140699339)

[Figure 11: Diagramme d'activité, import d'un module 18](#_Toc140699340)

[Figure 12: Architecture trois tiers 19](#_Toc140699341)

[Figure 13: Base de données, modèle conceptuel 24](#_Toc140699342)

[Figure 14: Architecture logique de l'application 25](#_Toc140699343)

[Figure 15: Timbrage sur l'application Composal 26](#_Toc140699344)

[Figure 16: Lecteur RFID avec carte 27](#_Toc140699345)

[Figure 17: Idéalisation du module "Proof Of Concept" 27](#_Toc140699346)

[Figure 18: Prototype, architecture Server Side Events – WebSocket 35](#_Toc140699347)

[Figure 19: Prototype, architecture Electron 36](#_Toc140699348)

[Figure 20: Résultat commun aux deux approches 40](#_Toc140699349)

[Figure 21: Pipeline CI/CD 42](#_Toc140699350)

[Figure 22: Raspberry Pi 4 Model B, présentation du constructeur 42](#_Toc140699351)

[Figure 23: Validation immédiate par JSON Schema dans un IDE, ici VSCode 54](#_Toc140699352)

[Figure 24: UML des "Accessors" 60](#_Toc140699353)

# Liste des codes sources

[Listing 1 - Exemple de configuration d’un module 11](#_Toc136012478)

[Listing 2: HTTP Polling, codec client 32](#_Toc136012479)

[Listing 3: HTTP Long Polling, code client 33](#_Toc136012480)

[Listing 4: Server Sent Events, code client 33](#_Toc136012481)

[Listing 5: Server Sent Events, code serveur 34](#_Toc136012482)

[Listing 6: WebSockets, code client 35](#_Toc136012483)

[Listing 7: JSX, composant React 39](#_Toc136012484)

[Listing 8: Exemple de l'approche utilitaire, Tailwind CSS 40](#_Toc136012485)

[Listing 9: Équivalant du listing 8, Daisy UI 40](#_Toc136012486)

## 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

### 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.**

### Cahier des charges

#### 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.**

#### 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.**

#### 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 et permettre de sélectionner les écrans à afficher.**

*****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.**

### 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.**

### 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]](#footnote-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]](#footnote-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]](#footnote-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]](#footnote-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.**

**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.**

## 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.**

### 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.**

#### 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.**

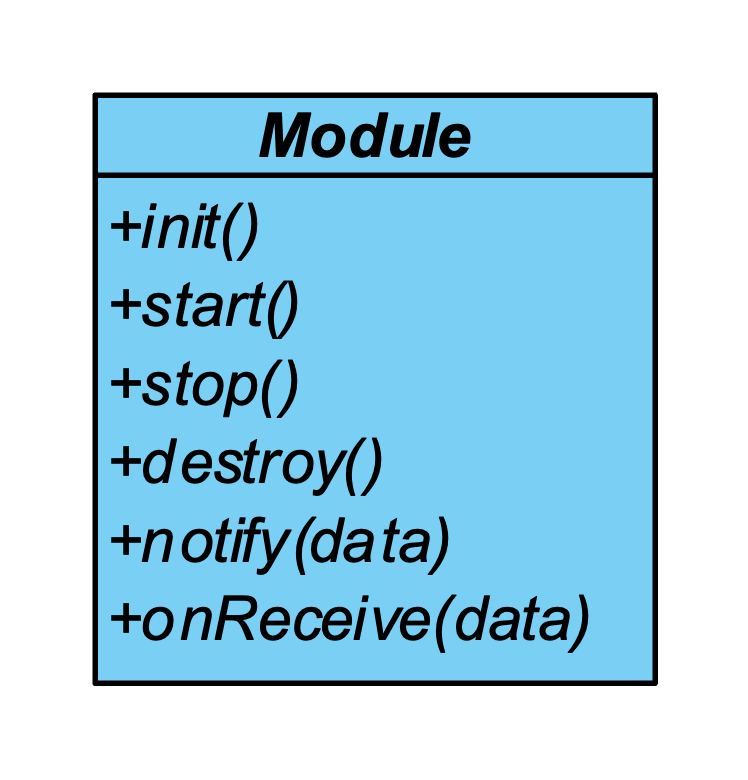


Figure 4 : Modélisation initiale de la classe Module

**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.**

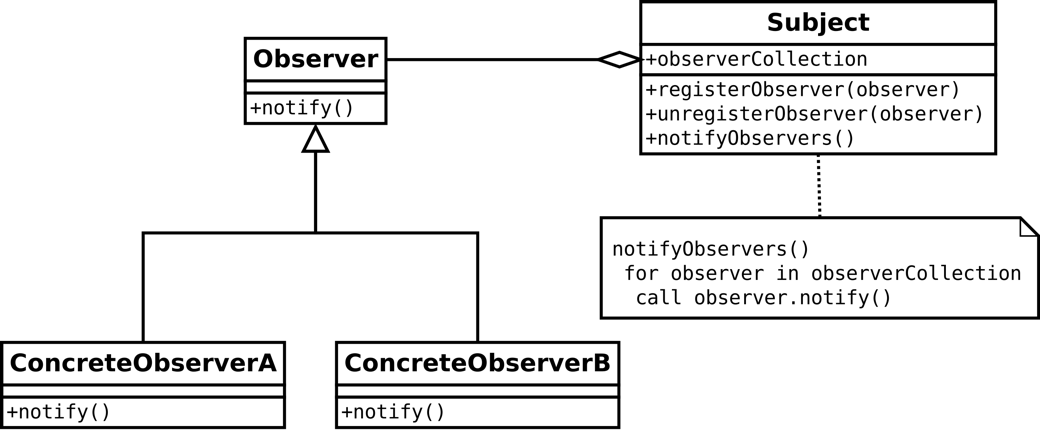


Figure 5: Pattern Observer, selon le GoF[[5]](#footnote-5)

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*.**

#### 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.**

**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 :**

{

"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.**

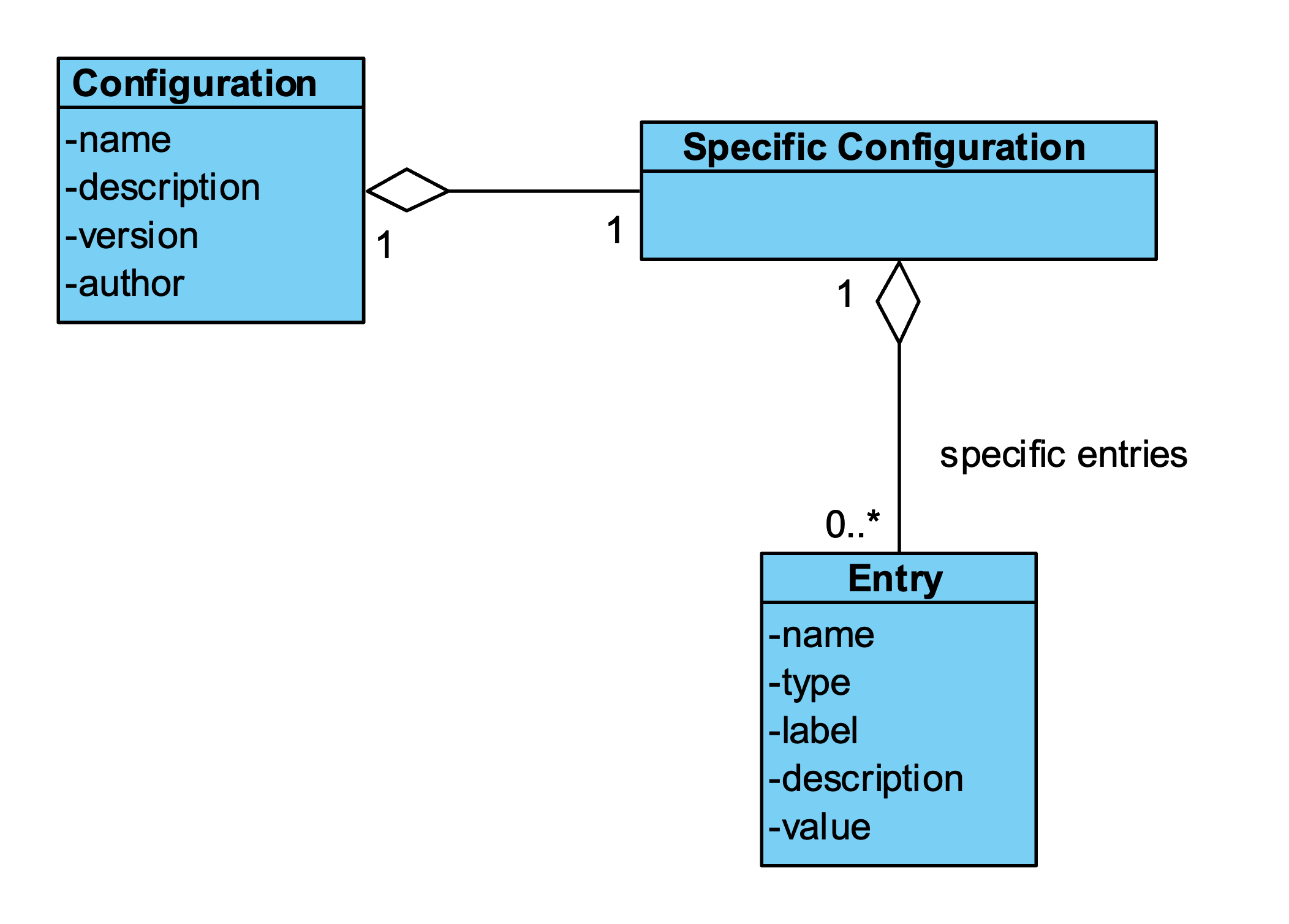


Figure 6: Modélisation du domaine : configuration d'un module

**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*).**

#### 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"

#### Modélisation

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

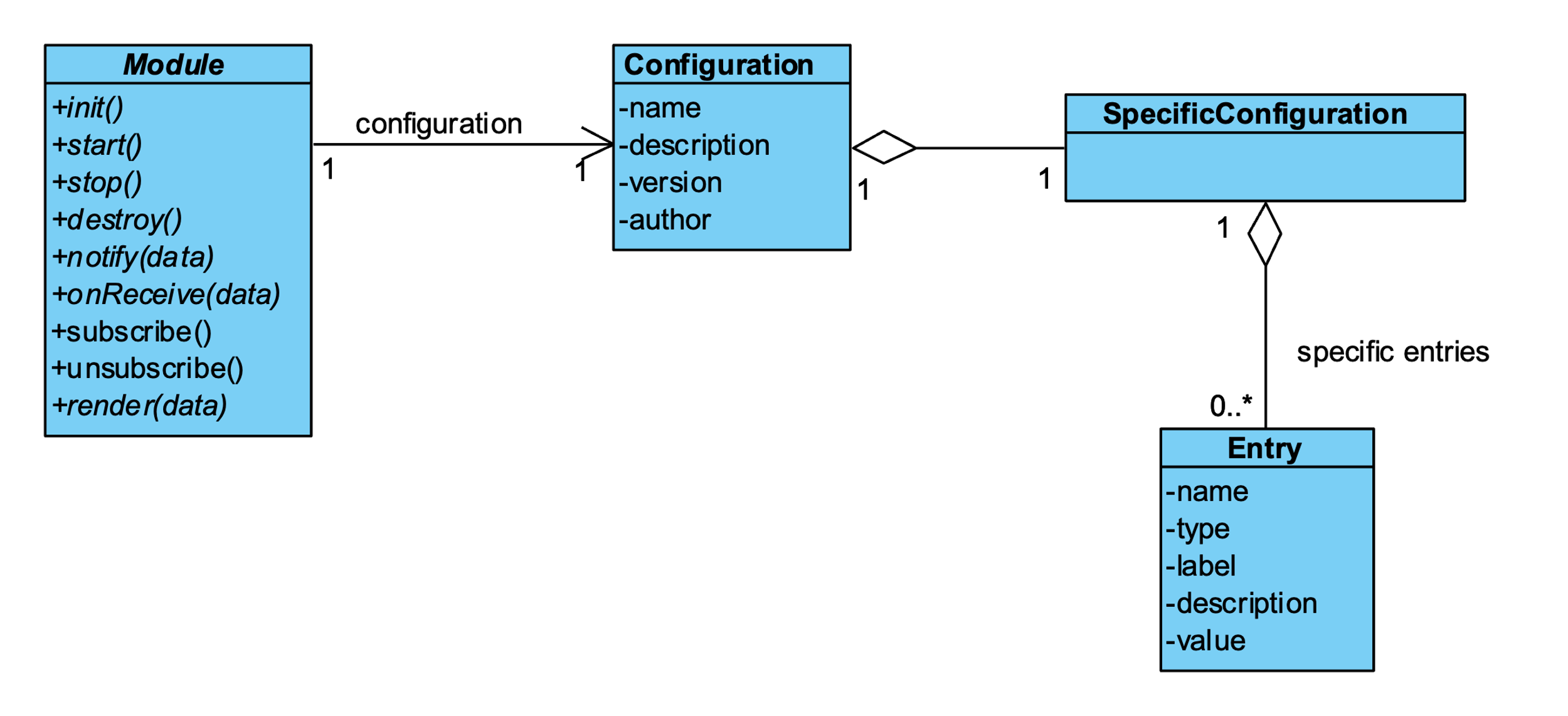


Figure 8: Modélisation du domaine : module

**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*.**

#### 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 :**

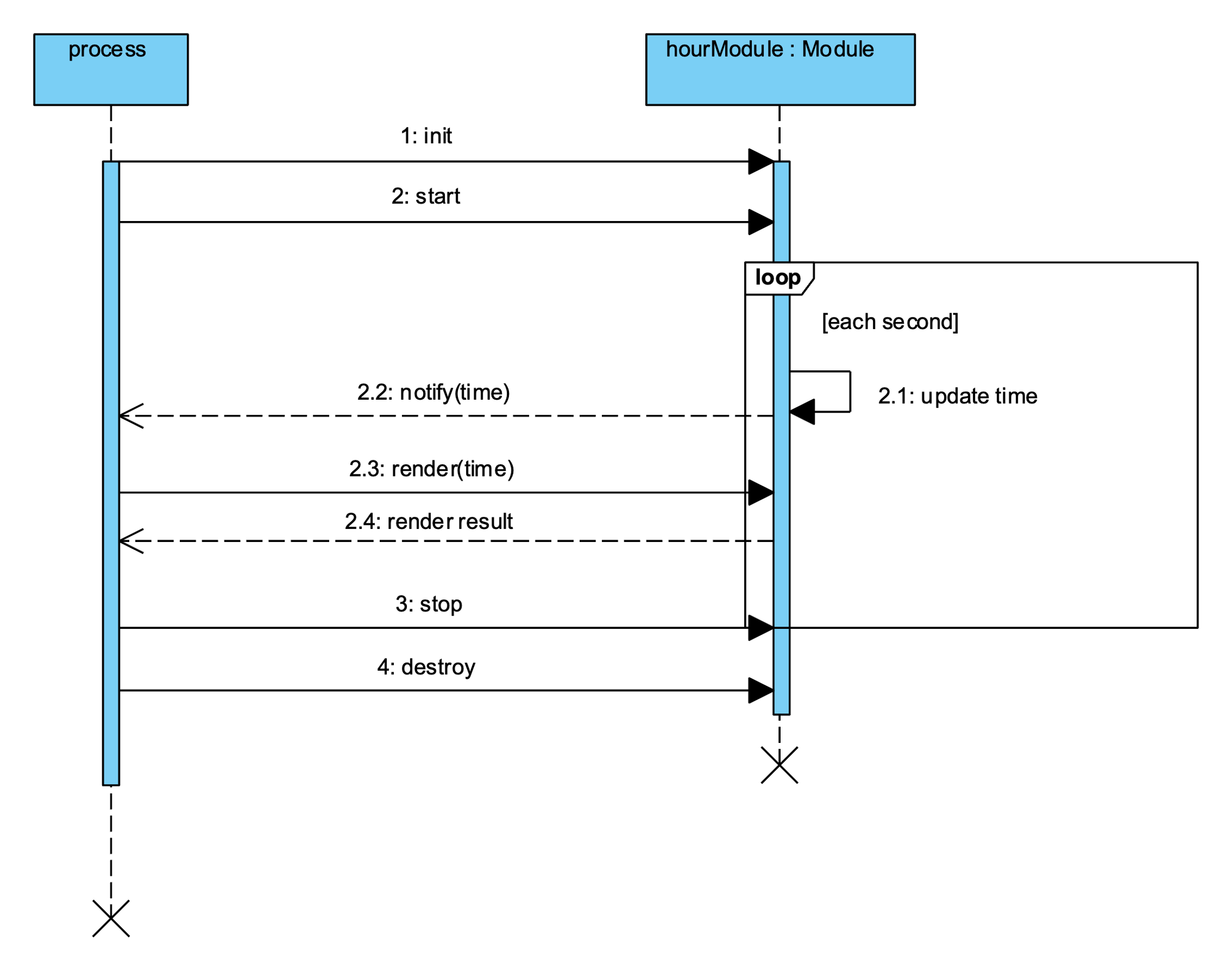


Figure 9: Module d'exemple, diagramme de séquence

**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.**

### 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 :**

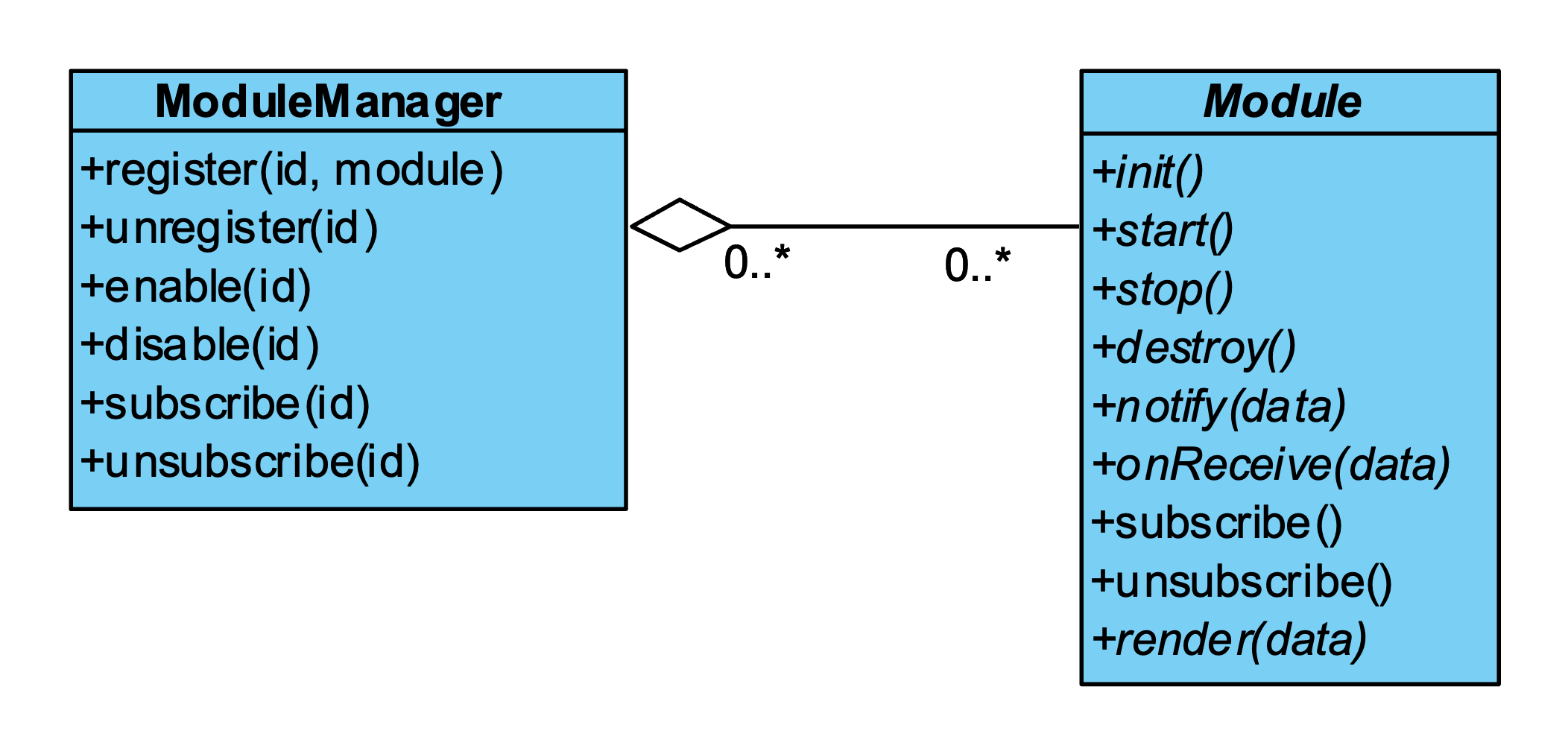


Figure 10: Modélisation du domaine : gestionnaire de modules

**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 :**

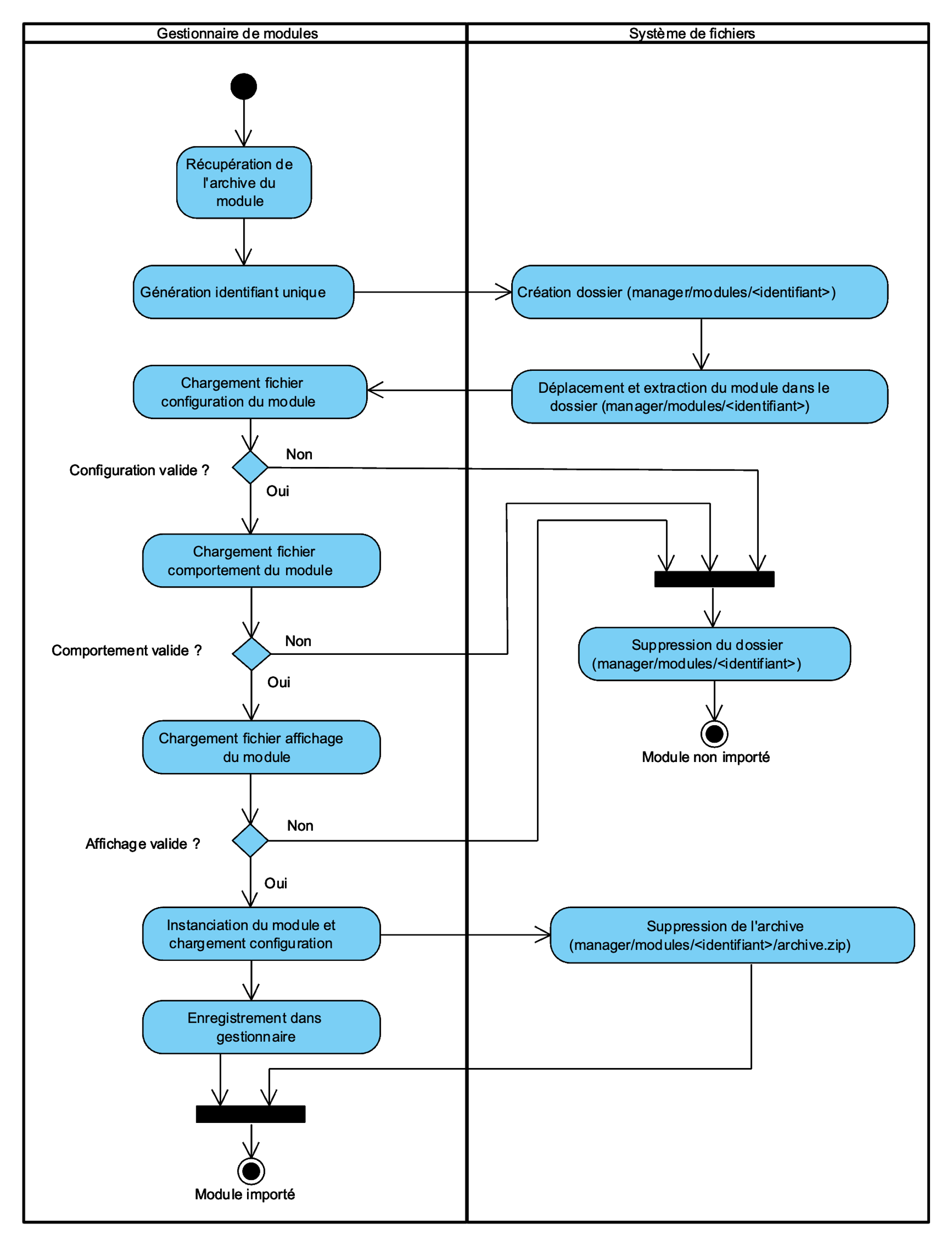


Figure 11: Diagramme d'activité, import d'un module

### 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.**

#### 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.**

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

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

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

**Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement**

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

**Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement**

* ***Tableau de bord (édition)* : Configuration de la disposition des affichages des modules selon une grille. Possibilité d’avoir une disposition différente pour chaque écran. Les différentes cellules de la grille peuvent être déplacées et redimensionnées pour créer des dispositions personnalisées.**

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

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

**Une image contenant texte, capture d’écran, Police, carte de visite

Description générée automatiquement**

* ***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.)**

#### 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.**

### 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.**

#### 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 :**

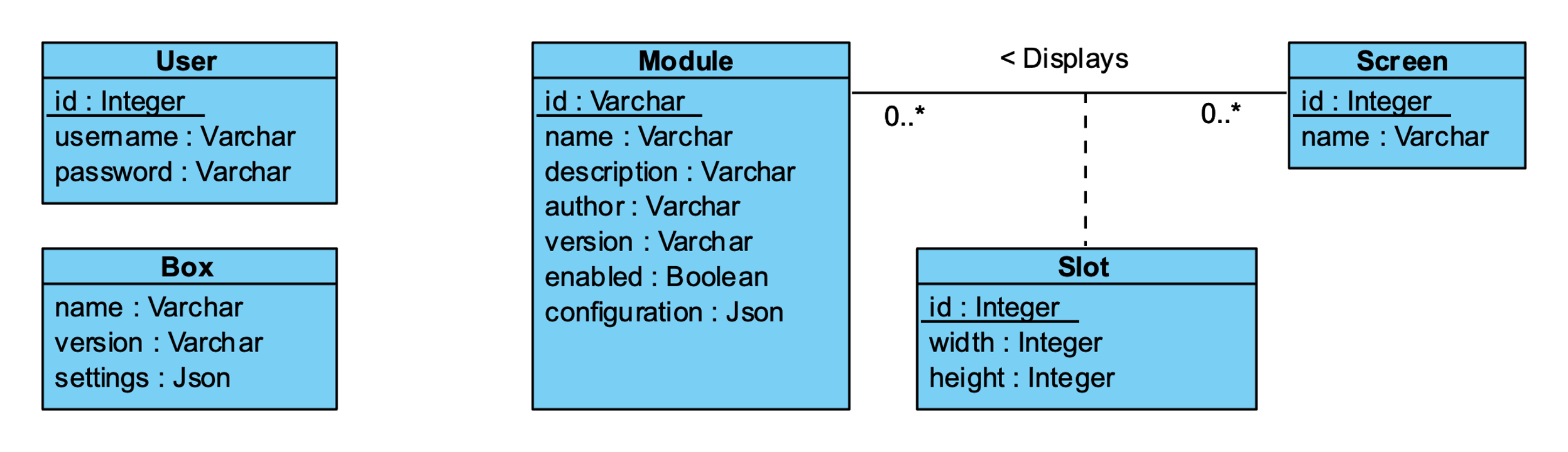


Figure 13: Base de données, modèle conceptuel

* *****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.**

#### 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

### 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.**

#### **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.**

Une image contenant texte, capture d’écran, cercle, carte de visite

Description générée automatiquement

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.**

Une image contenant carte de visite, Propriété matérielle, texte, logo

Description générée automatiquement

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.**

### Planification

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

#### 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.**

#### 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 25.06 – 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 à partir de cette date.**

*****Étape 8*** *– 26.06 au 31.06 – 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. A ce stade l’application devrait être globalement fonctionnelle.**

*****Étape 9*** *– 01.07 au 16.07 – 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. Différents consolidations et améliorations pourront être appliquées aux fonctionnalités existantes. Le temps à disposition permettra également la rédaction de la documentation et du rapport de projet.**

*****Étape 10*** *– 17.07 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.**

### 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.**

#### 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]](#footnote-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.**

**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

const 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]](#footnote-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

**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

#### 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) :**

Une image contenant texte

Description générée automatiquement

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 inter-processus (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.**

#### 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.**

### Choix technologiques

#### 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.**

#### Backend

##### 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.**

##### 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.**

##### 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.**

#### Frontend

##### 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 :**

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.**

##### 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 classes utilitaires à définir :**

Une image contenant texte, Police, logo, symbole

Description générée automatiquement

Figure 20: Résultat commun aux deux approches

### 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.**

#### 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.**

#### 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**

#### 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]](#footnote-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, 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

#### Production

##### 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]](#footnote-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

**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.**

##### 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]](#footnote-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 les 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.**

## Modules

**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 les fonctionnalités liées aux modules.**

**Comme défini dans l’analyse, les modules sont des composants indépendants permettant d’ajouter des fonctionnalités d’intégrations entre des services et des dispositifs au sein de l’application.**

**Ils disposent d’un comportement, qui définit la logique du module, c’est-à-dire les interactions qu’ils va pouvoir offrir, ainsi que d’un affichage et une configuration.**

**La particularité du développement des différentes fonctionnalités relatives aux modules est que celles-ci-visent à répondre aux besoins de l’application tout en étant utilisable principalement par deux types d’utilisateurs :**

* **Les développeurs : ces derniers souhaitent pourvoir prendre en main l’approche de développement d’un module afin de pouvoir eux-mêmes concevoir des intégrations entre des dispositifs et services quelconques. La conception d’un module se veut donc être relativement simple à prendre en main et ne pas imposer de limitations insensées sur les intégrations qui pourront être développées.**
* **Les utilisateurs finaux : ces derniers sont les utilisateurs de l’application web en elle-même, ayant un bagage technique moins important (mais tout de même suffisant pour comprendre les intérêts de l’utilisation de la passerelle). En finalité, ils doivent être en mesure d’acquérir, importer et exploiter les modules au sein de l’application pour profiter de différentes intégrations.**

**Toutes les fonctionnalités relatives aux modules ont été englobées dans un module TypeScript grâce à un workspace du monorepo (voir XXX). Ce module est nommé @yalk/module. Ce module sera une dépendance pour tout développeur souhaitant développer une nouvelle intégration.**

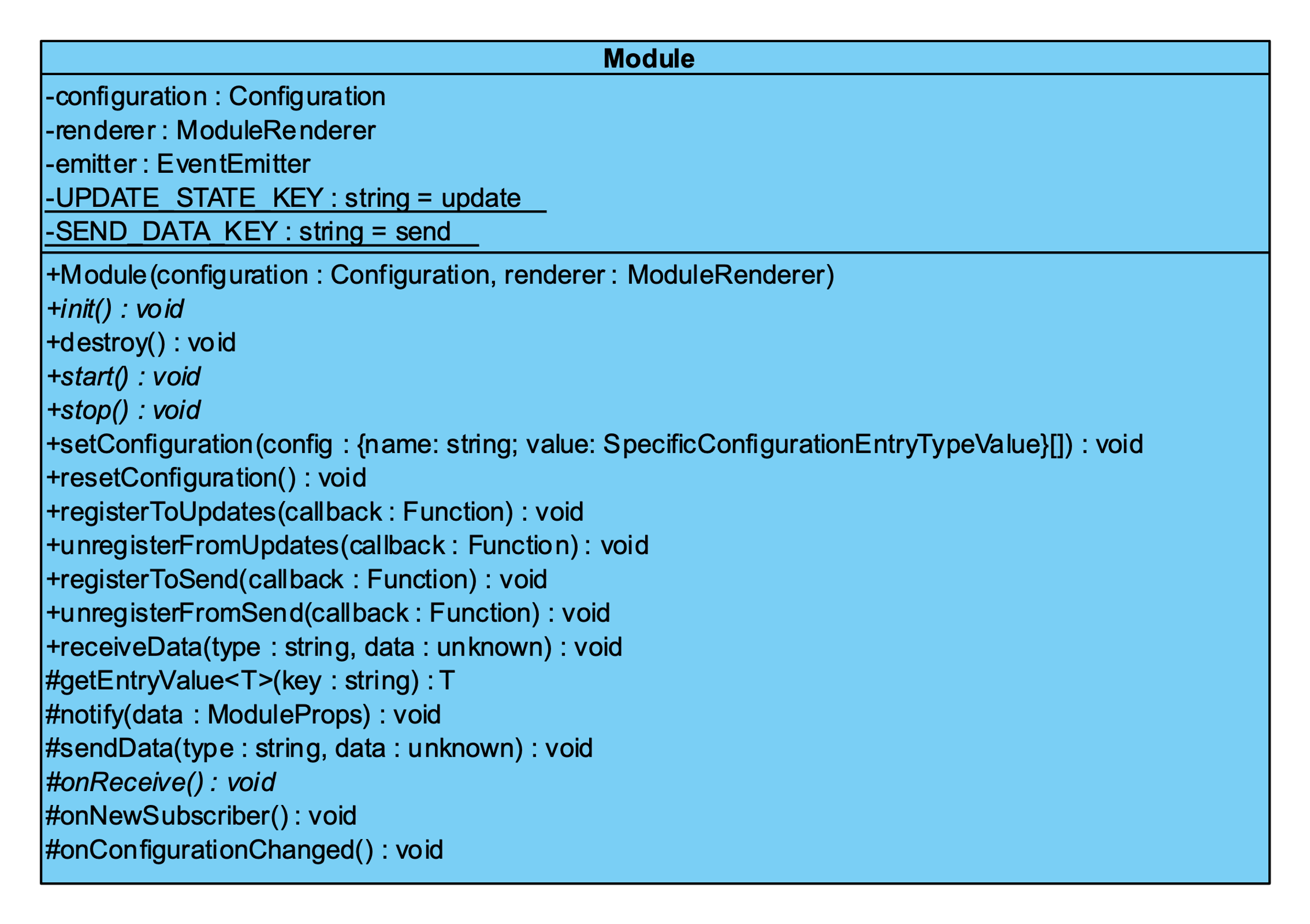
### **Comportement**

**Le comportement d’un module en est la pièce principale, il définit la logique d’interaction et défini un point central d’accès et de gestion des autres aspects d’un module : l’affichage et la configuration. Chaque module dispose de son propre cycle de vie (voir figure 3) et doit être capable de d’émettre des signaux à destination de ses observateurs, ces signaux servent à indiquer des mises à jour de l’état du module qui hésitent l’attention des observer afin de déclencher des actions.**

**Dans l’implémentation de cette approche, il existe deux types de signaux :**

* **Rendu : le module déclenche un rendu de son affichage selon son état actuel et notifie ses observeurs avec le résultat de ce nouveau rendu (voir 3.2).**
* **Envoi de données : un module peut nécessiter l’envoi de données vers un service ou un dispositif. Pour cela, il peut notifier ses observers avec des données. L’intérêt de ce type de signal est expliqué au point 3.5.**

**La classe abstraite Module englobe donc la définition et offre toutes les méthodes permettant de gérer le cycle de vie du module ainsi que s’abonner aux signaux qu’il envoie. Le développeur souhaitant concevoir une intégration grâce à un module va hériter de cette classe pour pouvoir définir sa logique.**



**Cet UML présente les différentes méthodes de la classe module. Pour le moment, notons les méthodes abstraites init, start et stop, ainsi que la méthode destroy. Ce sont les méthodes chargées de modifier l’état du cycle de vie du module :**

* ***Init* : permet l’initialisation de ressources de manière ponctuelle. Cette méthode est prévue pour être appelé une seule fois : lors de la première utilisation du module dans son contexte (dans ce cas, lors de l’ajout dans le gestionnaire de module, ou lors d’un redémarrage de l’application)**
* ***Start* : déclenche le comportement d’un module, c’est dans cette méthode que la logique de l’intégration va être implémentée. Cette méthode doit être appelée à chaque démarrage du module.**
* ***Stop* : déclenche la mise en suspend et une éventuelle libération de certaines ressources liées à la logique du module.**
* ***Destroy* : déclenche la destruction de toutes les ressources restantes en vue d’une suppression du module. Cette méthode n’est pas abstraite car elle implémente la désinscription totale des observer au sein de la classe.**

**Au niveau de signaux émis par le module et des données qui peuvent lui être envoyées, plusieurs méthodes sont fournies, fonctionnant selon l’approche du pattern Observer :**

* ***registerToUpdates*: permet d’enregistrer un observer pour être notifié d’un nouveau rendu de l’affichage du module.**
* ***unregisterFromUpdates* : permet de se désinscrire des signaux concernant l’affichage du module.**
* ***registerToSend*: permet d’enregistrer un observer pour être notifié d’un envoie de données en provenance du comportement du module.**
* ***unregisterFromSend* : permet de se désinscrire des signaux concernant l’envoi de donnée du module.**
* ***receiveData* : permet d’envoyer des données vers le module. En interne, cette méthode appellera la méthode *onReceive* qui doit être redéfinie par le développeur, c’est dans celle-ci que le traitement des données reçues devra être implémenté.**
* ***sendData* : cette méthode est accessible uniquement au sein de la classe module (*protected*), elle permet d’envoyer des données en dehors du module, c’est-à-dire de notifier ses observers.**

**Il est tout à fait possible de s’enregistrer aux évènements d’un module inactif (dans l’état initialisé ou stoppé), cela permet notamment de réagir à son activation.**

**Le pattern Observer a été implémenté à l’aide de la classe EventEmiter, intégrée à Node.js. Cette classe permet de créer, déclencher et écouter des évènements asynchrones grâce à l’utilisation de callbacks. En voici un simple exemple d’utilisation :**

import { EventEmitter } from 'events'

const emitter = new EventEmitter()

// Définition du callback

const listener = (value: number) => {

console.log("Got:", value)

}

// Ecouter un événement

emitter.on('event', listener)

emitter.on('event2', () => {})

// Plusieurs événements peuvent être attribués au même EventEmitter

// Emettre un événement

emitter.emit('event', 42)

// Se désabonner d'un événement

emitter.off('event', listener)

**Notez que pour que la désinscription fonctionne correctement, la référence du callback listener doit être identique à celle fournie à l’enregistrement. Dans le cas contraire, l’observeur se sera pas désinscrit, ce qui peut causer des fuites de mémoires. Pour cette raison, il est souvent nécessaire de maintenir une structure de données qui référence les callbacks des observers pour pouvoir les désinscrire après coup.**

**Les autres méthodes et types de la classe *Module* concerne la configuration et l’affichage, elles seront détaillées au fil du chapitre.**

### **Configuration**

**Au niveau de la configuration d’un module, l’approche définie dans le chapitre d’analyse a été implémentée et de nouvelles fonctionnalités ont pu être ajoutées. La distinction entre les deux parties de la configuration d’un module reste en place :**

**La configuration principale défini les valeurs obligatoires d’un module :**

{

"name": "Example Module",

"description": "An example description",

"version": "1.0.0",

"author": "Nicolas Crausaz",

"icon": "icon.svg",

"requires": ["http", "keyboard"],

// ...

}

**Nous pouvons constater l’ajout de deux nouvelles valeurs par rapport à la proposition de l’analyse :**

* ***icon* : permet de spécifier l’icône du module, celui-ci servira à illustrer le module dans l’interface de l’application web. Le nom du fichier doit y être spécifié et ce fichier doit se trouver à la racine de l’archive zip du module pour pouvoir être importé dans l’application. Si cette valeur est vide, ou si le fichier spécifié n’existe pas, l’application utilisera une icône de module par défaut.**
* ***requires* : permet de spécifier les différents accès aux dispositifs d’un module. Cette fonctionnalité est détaillée plus tard au point *3.5*. Pour le moment, il faut garder en tête que les valeurs spécifiées détermineront les possibilités d’interactions depuis et vers différents dispositifs externes.**

**Au niveau de la configuration spécifique, l’implémentation suis également la structure définie lors de l’analyse. Un nouveau type de donnée à cependant été ajouté : le type « secret » correspondant à une valeur configurable de type texte, dont l’utilisation vise à indiquer que cette valeur est de nature plus ou moins confidentielle. Cela permettra notamment de prendre des mesures quant à son traitement, par exemple en masquant la valeur sur l’interface pour éviter les fuites involontaires. Ce type a été ajouté pour permettre par exemple la configuration de clé API ou de jetons d’authentification pour des créer des intégrations vers des applications tierces.**

"specificConfig": {

"anExampleSecret": {

"type": "secret",

"label": "Example secret",

"description": "An example value for a secret",

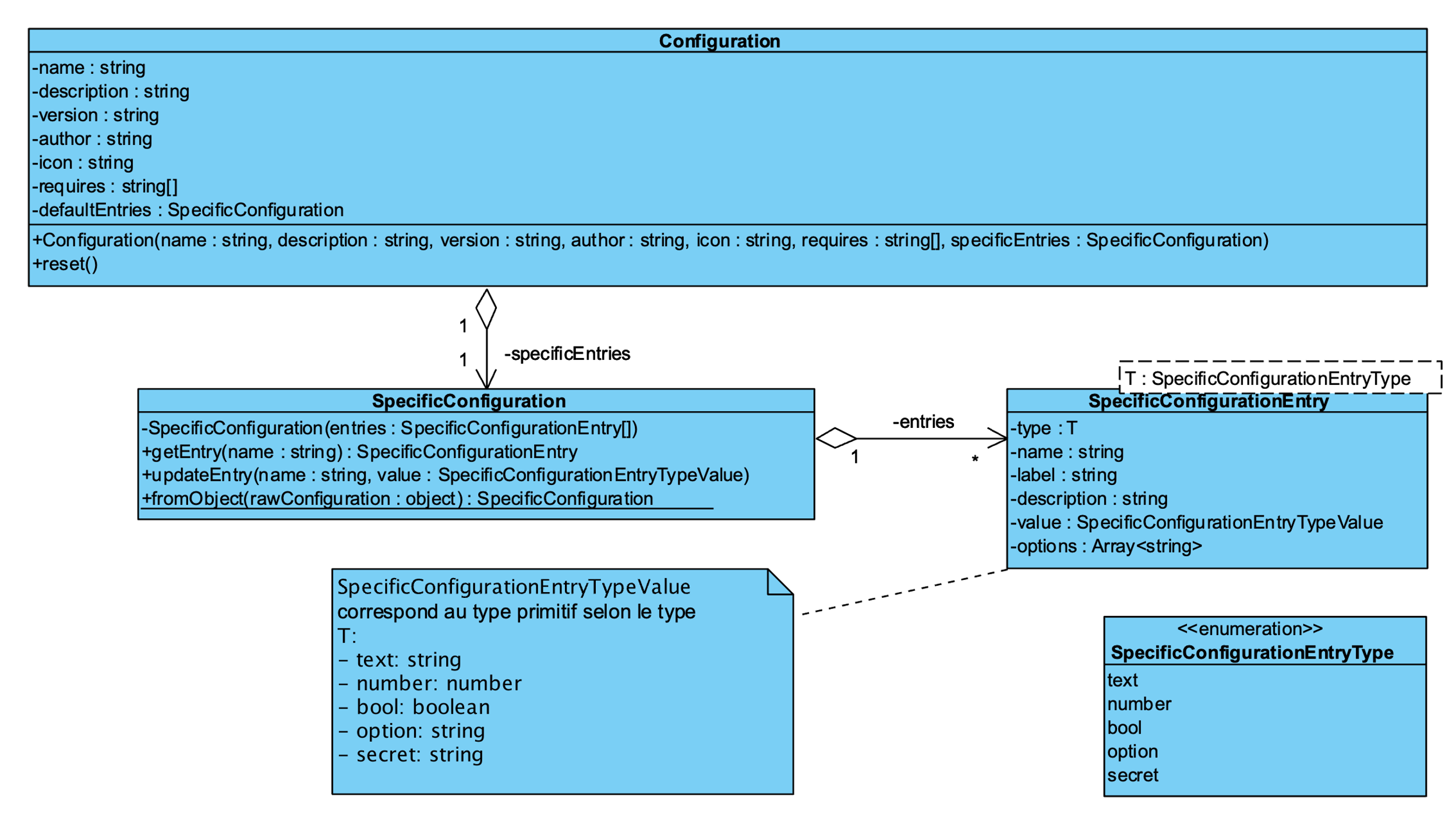
"value": "secret"

}

// ...

}

**Pour avoir un aperçu global de l’implémentation de cette notion de configuration, intéressons-nous à sa représentation UML :**



**Chaque attribut de Configuration possède un getter qui permet sa lecture, non représenté sur le diagramme pour des questions de visibilité. Il est important de noter les détails d’implémentations suivants : Lors de l’appel au constructeur de *Configuration*, celui effectue une copie profonde de la configuration spécifique (grâce à la fonction *structuredClone***[[11]](#footnote-11) **offerte par JavaScript), afin de conserver en mémoire les valeurs initiales de celle-ci. Cela permet de pouvoir réinitialiser la configuration par défaut en appelant la méthode *reset.***

**Lors de la modification de la configuration (avec la méthode *updateEntry*), seules les clés existantes dans la configuration spécifique initiale peuvent être modifiées, ceci pour respecter la structure définie par le développeur.**

**Au niveau de l’implémentation, j’ai exploité différentes fonctionnalités offertes par TypeScript telles que la généricité, les Conditionnal Types et Union Types pour assurer que le typage des types de SpecificConfigurationEntry soit respecté. J’ai effectué un mapping des valeurs possible de SpecificConfigurationEntryType pour leur assigner assigner le type de valeurs acceptées.**

export type SpecificConfigurationEntryType = 'text' | 'number' | 'bool' | 'option' | 'secret' // Union type

type SpecificConfigurationEntryTypeValueMap = {

text: string

number: number

bool: boolean

option: string

secret: string

}

export type SpecificConfigurationEntry<T extends SpecificConfigurationEntryType = SpecificConfigurationEntryType> = {

type: T

name: string

label: string

description?: string

value: SpecificConfigurationEntryTypeValueMap[T]

options?: T extends 'option' ? Array<string> : undefined // Conditional type

}

**Cette implémentation générique permet d’exploiter le typage statique et détecter ce type d’erreur à la compilation :**

const test: SpecificConfigurationEntry<'text'> = {

type: 'text',

name: 'test',

label: 'test',

value: 10, // Erreur détectée ici car 10 n'est pas une chaîne de caractères

}

#### **Utilisation**

**Bien que les différentes valeurs de la configuration standard servent principalement à identifier le module et que leur usage prendra plus de sens dans l’application web, la configuration spécifique a pour objectif de servir au développeur à permettre de faire varier le comportement d’un module selon certains valeurs définies par un utilisateur.**

**L’objectif de cette approche est bien entendu d’augmenter la portée et la réutilisabilité d’une intégration pour la rendre utilisable dans différents contextes. Il est donc nécessaire de permettre l’accès à l’état de la configuration actuelle depuis le comportement d’un module. Ceci est rendu possible grâce à la méthode *getEntryValue* de la classe *Module*:**

protected getEntryValue<T>(key: string): T {

return this.\_configuration.specificConfig.getEntry(key)?.value as T

}

**Celle-ci va accéder à la configuration spécifique, pour récupérer la valeur de la clé (unique) spécifiée. Dans le cas d’une clé spécifiée inexistante, la valeur retournée sera *undefined*, grâce à l’utilisation de l’opérateur JavaScript *Optional chaining (?.)****[[12]](#footnote-12)****,* plutôt de lever une exception. C’est donc la responsabilité du développeur de s’assurer que la clé spécifiée existe réellement dans la configuration spécifique qu’il a définie.**

start(): void {

const configValue = this.getEntryValue<string>('exampleText')

// ...

}

exemple d’accès à la configuration spécifique depuis le comportment d’un module

#### **Validation**

**Pour assurer que les différentes valeurs requises ont été spécifiées et que la structure de la configuration définie par le développeur respecte la structure attendue, un système de validation a été mis en place. Ce système de validation consiste à valider la structure du fichier de configuration JSON fourni par le développeur.**

**Cette validation est rendue possible par l’utilisation de JSON Schema**[[13]](#footnote-13)**, une spécification standardisée permettant de définir une structure attendue pour un objet JSON ainsi qu’un certain nombre de validations sur le type des données et les valeurs autorisés. Un JSON Schema est lui-même défini au format JSON, ce qui le rend sa lecture et son écriture relativement simple. Cette approche apporte de grands avantages, entre autres :**

* **Validation et structure : la structure d’un JSON Schema est structurellement assez proche du l’objet que l’on souhaite valider. Un schéma offre un grand nombre de possibilités de validation tant au niveau de types que de la structure d’objets, même imbriqués.**
* **Réutilisabilité : un JSON Schema peut être référencé au travers d’un fichier ou d’une URL, ce qui permet de partager facilement un schéma pour plusieurs services. Ceci assure la cohérences des validations entre les différents systèmes. Il peut être utilisé tant du coté serveur que client.**

**Voici un exemple de JSON Schema, extrait du Schema utilisé dans ce projet**

{

"$ref": "#/definitions/module",

"definitions": {

"module": {

"type": "object",

"additionalProperties": false,

"properties": {

"name": {

"type": "string",

"minLength": 2

},

// ...

},

"required": ["name"],

"title": "Module"

}, // ...

}

**Le schéma d’exemple présenté permet simplement d’appliquer une règle de validation sur la l’attribut *name* de la configuration en obligeant une chaine de caractère de longeur d’au moins deux caractères.**

**Il serait tout en fait envisageable de se passer ce genre de spécification pour effectuer la validation d’objet JSON, une simple fonction de validation « custom » aurait fait l’affaire. Cependant, l’utilisation du standard JSON Schema apporte une fonctionnalité très intéressante qui a été exploitée dans ce projet : il est possible de spécifier la référence du schéma dans le fichier à valider pour profiter d’une détection immédiate des erreurs dans un environnement de développement (IDE).**

**On spécifie le schéma de validation de cette manière :**

{

**"$schema": "https://test.crausaz.click/schema.json",**

"name": "Example Module",

// Reste de la configuration ...

}

**Comme indiqué précédemment, un schéma peut être référencer de manière distante grâce à une URL (ici, le schéma est hébergé sur un serveur web). On profite désormais de la détection d’erreur immédiate :**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Figure 23: Validation immédiate par JSON Schema dans un IDE, ici VSCode

**Dans cet exemple, la propriété « name » ne respecte pas la règle de longueur minimale définie.**

**Grâce à cette fonctionnalité, il a été possible d’offrir une validation stricte pour la configuration des modules. Cette validation s’oriente sur deux axes :**

* **Une validation indicative, effectuée dans l’IDE du développeur grâce l’attribut *$schema*, visant à réduire la marge d’erreur et de détecter les erreurs lors du développement, avant que le développeur importe son module dans l’application.**
* **Une validation « stricte » : effectuée au sein même de l’application, lors de l’import du module, permettant de refuser l’import d’un module si sa configuration n’est pas valide, évitant de potentielles erreurs d’exécution. Cette validation est effectuée grâce à un « JSON Schema Validator ». Dans le cadre de ce projet, le module JavaScript AJV**[[14]](#footnote-14) **a été utilisé. Celui-ci permet de valider un objet JSON selon un schéma spécifié.**

const ajv = new Ajv()

const validate = ajv.compile(ModuleValidationSchema)

return validate(configToValidate)

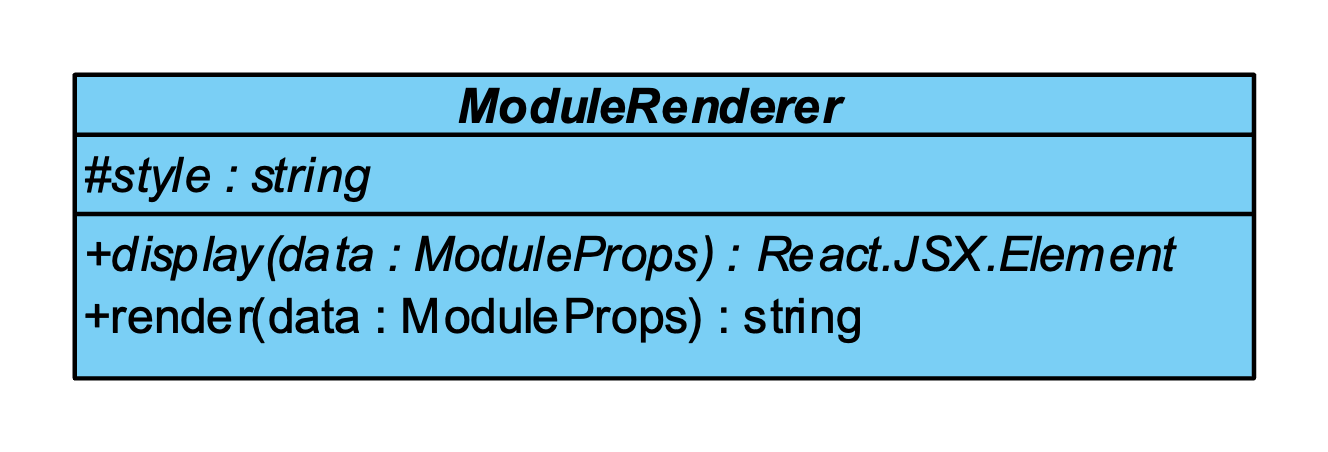
### **Affichage**

**L’affichage d’un module correspond à sa partie visuelle, destinée à être affichée sur les différents écrans de visualisation de l’interface web. Cet affichage est défini par le développeur du module dans un fichier réservé à cet usage. Comme défini dans l’analyse, cette définition doit se faire dans un format pouvant être transformé en HTML et permettant l’injection de données durant une étape de « rendering », afin de pouvoir être dynamiquement ajouté à l’interface.**

**Pour l’implémentation des fonctionnalités d’affichage, le choix s’est porté sur l’utilisation de React et de JSX. JSX n’est pas un moteur de templating à proprement parlé, mais il propose toutes les fonctionnalités nécessaires aux besoin des modules. React embarque des fonctionnalités de rendu côté serveur permettant de transformer un composant React (JSX) sous format HTML en y injectant des données, notamment grâce à la fonction *renderToStaticMarkup****[[15]](#footnote-15)***.**

**Plutôt que le laisser la responsabilité du rendu au module (comme prévu dans l’étape d’analyse), j’ai décidé de dissocier cette responsabilité dans une nouvelle classe abstraite, *ModuleRenderer*.**

**En héritant de cette dernière, le développeur implémente les méthodes nécessaires à la définition d’un affichage de module. L’utilisation d’un fichier *.tsx* est nécessaire à la définition de l’affichage afin de profiter de JSX et TypeScript.**



**Le *ModuleRenderer* implémente ainsi toutes les méthodes liées au rendu de l’affichage, celles-ci sont expliquées dans les sections suivantes.**

#### **Définition du composant**

**Comme défini au point précédent, un affichage est conçu avec la librairie React, dans lequel les composants utilisent le format JSX (TSX avec TypeScript) est utilisé (voir 2.8.3.1).**

**Dans React, il existe un mécanisme fondamental appelé « props » (pour propriétés) servant à fournir des données en entrée à un composant sous la forme d’un objet JavaScript. Celles-ci permettent de définir une structure de données attendue plus ou moins complexe dans l’optique de rendre les composants générique et réutilisables. Grâce à l’utilisation de TypeScript, il est possible de typer ces props et d’assurer que les données soient cohérentes et respectent les valeurs attendues. Cela augmente fortement la qualité de code et sert d’auto-documentation du composant.**

**Au niveau de la définition de la classe ModuleRenderer, on aperçoit la définition de type des props dans les deux méthodes display et render, dont les données sont de type ModuleProps.**

**ModuleProps est une interface TypeScript vide, celle-ci peut être augmentée par le développeur pour définir les données attendues par son composant React :**

export interface ModuleProps {

[key: string]: unknown // Permet l’accès aux propriétés avec l’opérateur []

}

// Code du dévellopeur

import { ModuleProps } from '@yalk/module'

interface ExampleProps extends ModuleProps {

message: string

}

**Au sujet des deux méthodes offertes par cette classe :**

* ***Display* est la méthode ou le développeur va définir le contenu JSX de son affichage. Celle-ci est abstraite pour forcer le développeur à l’implémenter**
* ***Render* permet de déclencher le processus de rendu, le CSS définit dans l’attribut style et le JSX retourné par la méthode display vont être combiné dans un JSX et transformé en HTML (voir *3.3.2*).**

**Prenons un exemple de l’affichage d’un module affichant un simple message :**

import React from 'react'

import { ModuleRenderer } from '@yalk/module'

interface ExampleProps extends ModuleProps {

message: string

}

export default class ExampleRenderer extends ModuleRenderer {

protected style: string = ``

display({ message }: ExampleProps): JSX.Element {

return (

<p>{message}</p>

)

}

}

**Ceci est le code minimal nécessaire à la création d’un affichage fonctionnel dans le cadre de l’application. On aperçoit l’utilisation de *props* définies selon une interface TypeScript.**

#### **Processus de rendu**

**Dès lors que le comportement du module notifie la nécessité de rafraichir l’affichage du module, il va se dérouler une série d’action permettant de générer un nouveau rendu de celui-ci, qui pourra ensuite être transi à l’interface client pour y être affiché. Ce processus est déclenché par le comportement du module mais son exécution est complètement gérée par le *ModuleRenderer*.**

**React offre des fonctionnalités de rendu coté serveur (Server Side Rendering). Dans notre cas, c’est l’outil *renderToStaticMarkup* qui est utilisé. Cette fonction permet de convertir un composant React en son rendu HTML en tendant compte de son état actuel.**

**Nous avons vu auparavant l’existence de la méthode notify qui permet déclencher un nouveau processus de rendu en spécifiant les données à utiliser (selon le sous-type de *ModuleProps* définit). Le processus de rendu démarre alors dès l’appel à cette méthode :**

protected notify<T extends ModuleProps>(data: T): void {

this.emitter.emit(Module.UPDATE\_STATE\_KEY, this.\_renderer.render(data))

}

**La génération est ensuite déléguée par l’appel à la méthode render du *ModuleRenderer*.**

**La méthode render est chargée d’effectuer la transformation du JSX retourné par la méthode display, implémentée par le développeur, et d’injecter le style dans un nouvel éléments JSX. Finalement, le résultat est transformé en HTML sous la forme d’une chaine de caractère grâce à la méthode de rendu côté serveur de React, *renderToStaticMarkup*:**

import { renderToStaticMarkup } from 'react-dom/server'

public render(data: ModuleProps): string {

return renderToStaticMarkup(

<>

<style>{this.style}</style>

{this.display(data)}

</>

)

}

**Grâce à l’utilisation de l’emitter, le résultat va être notifiés aux observers.**

#### **Style**

**Initialement, il était prévu de permettre l’utilisation de TailwindCSS également dans le contexte des affichages des modules. Cependant, cela n’est pas totalement possible pour plusieurs raisons : TailwindCSS utilise un moteur JIT**[[16]](#footnote-16) **(Just-In-Time), lors de développement ainsi que de la « compliation » des fichiers de production, le moteur scanner le code source pour y détecter les classes utilisées pour ne conserver que celle-ci (purge). Cela permet de réduire grandement le temps de compilation et de chargement des pages en production. Cela a cependant un inconvénient, il n’est pas possible d’utiliser des classes TailwindCSS de manière dynamique si celles-ci n’ont pas été conservée par le JIT (c’est à-dire utilisées dans l’interface client). L’utilisation des classes TailwindCSS n’est donc pas viable pour l’affichage des modules, car les classes utilisables sont trop restreintes et on risque de se retrouver avec des affichages à moitié complets.**

**Une solution à ce problème est d’utiliser le CDN TailwindCSS dans l’application, mais cette pratique est fortement déconseillée par les développeurs TailwindCSS pour des questions de performance (on ne profite pas du JIT et le temps de chargement en sera impacté).**

**L’approche retenue pour ce projet a donc été d’utiliser du CSS, celui-ci étant directement écrit pas le développeur dans le fichier responsable de la définition de l’affichage. Ainsi, comme montré dans les exemples précédents, le développeur écrit son style dans l’attribut style afin qu’il puisse être automatiquement injecté lors du processus de rendu.**

export default class ExampleModuleRenderer extends ModuleRenderer {

protected style: string = `

.example-module {

background-color: lightgreen;

padding: 6px;

}

`

display({ message }: ExampleModuleProps): JSX.Element {

return (

<div className="example-module">

<h1>Hi ! This is the example module</h1>

<p>{message}</p>

</div>

)

}

**Exemple d’affichage de module utilisant du style**

**Il est également possible d’injecter sur style directement sur les éléments en utilisant la syntaxe JSX :**

<p style={{color: "red"}} >{message}</p>

### **Accès aux dispositifs**

**Pour répondre aux différents besoins d’interactions vers les modules et en prévision du développement du module PoC, la question de la gestion des communications vers les dispositifs physiques et l’utilisation de divers protocoles s’est rapidement posée.**

**Durant la phase de développement des premiers modules de tests, deux approches différentes ont été comparées pour la gestion des données envoyées par et vers les modules :**

**Imaginons deux modules différents dans leur objectif, nécessitant tous deux l’accès et la communication avec un même dispositif (par exemple un lecteur RFID via USB). Chacun des développeurs va donc implémenter son propre accès au dispositif, selon une approche pouvant être différentes (libraires différentes etc.). Ainsi, chaque module instanciera son propre accès aux dispositifs.**

**Cette approche peut amener à des erreurs non gérées dues à l’utilisation simultanée d’un périphérique, on peut penser à des problèmes de surcharge du dispositif et d’accès concurrents.**

**Au niveau du code du comportement du module, cette approche l’alourdi et augmente le travail nécessaire pour un développeur pour mettre en place une intégration qui plus est si ce dernier souhaite concevoir plusieurs modules utilisant le même dispositif va amener à de la duplication du code.**

**L’approche qui a donc été choisie et de créer des « Accessors », classes permettant de gérer la logique d’acquisition et d’envoi de données vers un dispositif précis ou un protocole (USB, saisie clavier, Bluetooth, WiFi).**

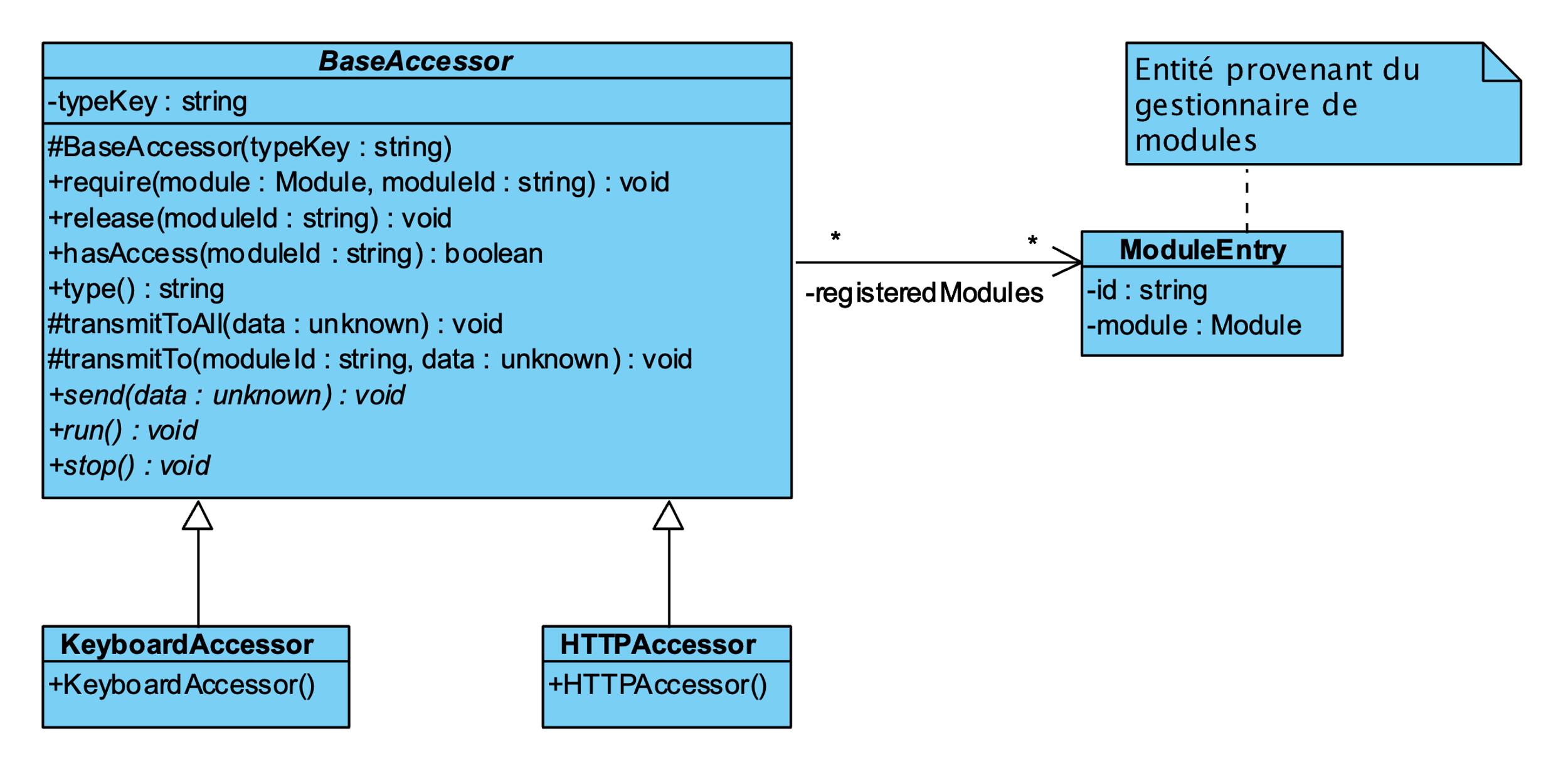


Figure 24: UML des "Accessors"

**Un accessor encapsule la logique de communication vers en en provenance d’un dispositif arbitraire. Il est également capable de propager les données qu’il reçoit vers les modules qui s’y sont incrit (en spécifiant le champs requires dans leur configuration). Chaque accessor est doté d’un type (keyType) qui lui permet d’être identifié (la responsabilité d’éviter les collisions de type est gérée par le code qui instancie les différents accessors) et c’est ce même type qui doit être spécifié dans la configuration d’un module (requires) pour y autoriser l’accès.**

**Lorsqu’un module ayant mentionné un ou plusieurs accessors dans ses requires est démarré (appel à start), le gestionnaire de module va s’assurer d’abonner le module à l’accessor grâce à la méthode require. Réciproquement, le gestionnaire désabonnera le module lorsque celui-ci est stoppé en appelant la méthode release (voir point 3.5.2).**

**En interne, l’accessor doit implémenter trois méthodes :**

* ***Send :* implémente la logique d’envoi de données vers le dispositif ou protocole représenté pas l’accesseur.**
* ***Run :* permet d’initialiser et de déclarer les le comportement lié à l’acquisition des données reçues d’un dispositif.**
* ***Stop :* permet d’arrêter les fonctionnalités de réception et d’envoi de données et de nettoyer les ressources utilisées.**

**Comme le montre le diagramme ci-dessus, deux accesseurs ont été développé pour les besoins du projet :**

* ***KeyboardAccessor* : permet à un module de recevoir la saisie du clavier ou de tout dispositif écrivant sur l’entrée standard du système. Utilisé notamment dans le module Poc (voir chapitre *5*)**
* ***HTTPAccessor* permet à un module de recevoir des données http passant par l’API (voir point xx). Cet accesseur est particulier car il ne permet pas l’envoi de données. L’envoi requêtes http peut se faire directement depuis le comportement du module grâce à la l’api fetch.**

**C’est la responsabilité du gestionnaire de module d’instancier et démarrer et de stocker les différents accesseurs ainsi que de rediriger les données vers l’accesseur concerné (grâce au type). Ainsi, les signaux de données envoyés par les modules peuvent être redirigés vers le bon accessors selon le type de données spécifiés.**

**Afin de garantir un accès contrôlé aux dispositifs au travers de acccessors, un système de permission / droits a été implémenté : un module nécessitant l’accès à un accessor devra le spécifier explicitement dans sa configuration sous peine de ne pas pouvoir recevoir ou envoyer des données via ce canal.**

**Comme expliqué au point XX, la configuration d’un module a donc été modifiée en conséquence pour renseigner cette information :**

{

"name": "Example module",

"requires": ["keyboard", "http"],

// ...

}

* **Exemple de module demandant l’accès au accesseurs keyboard et http**

**Lors du chargement de l’application, les différents accessors existants sont instanciés et démarré (appel à run()). Lors du démarrage d’un module (start()), les dépendances (requires) du module sont lues. Chaque dépendance déclenchera l’inscription d’un module à un accessor afin de pouvoir récupérer ou envoyer les données au travers de celui-ci.**

**Bien entendu, cette approche à des limitation : bien qu’elle uniformise l’accès à un certain type de dispotif ou de protocole et limitant l’accès grâce au « require », cela n’en fait pas une sécurité d’accès absolue car un développeur pourrait sans autres implémenter lui-même l’accès au dispotifs dans le comportement de son module et outre-passer cette vérification. L’idée principale était surtout d’offrir des fonctionnalités déjà implémentée pour simplifier le travail des développeurs**

**Cela demande également au développeur qui maintenant l’application de développer les différentes intégrations afin de les inclure. L’avantage est que les différents modules accesseur sont un module a part (monorepo) et peuvent être modifier sans modifier directement l’application**

**Les accessors fournis suffisent à l’utilisation souhaitée dans le projet. Cependant leur implémentation peut sembler relativement simpliste. Dans le cas d’implémentation d’un accesseur pour un protocole plus complexe et moins générique (par exemple USB et Bluetooth), quelques modifications de l’interface BaseAccessor ne peuvent pas être excluent.**

### **Gestionnaire de module**

**Comme défini lors de l’analyse, le gestionnaire de module l’entité responsable du bon fonctionnement des modules et de la gestion de leur 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 et est capable de les charger dynamiquement en mémoire afin d’assurer leur exécution.**

**TODO : UML**

**Pour maintenir l’état actuel de chaque module, le gestionnaire attribue 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 de permettre à l’application de communiquer avec un module spécifique. Le gestionnaire est également responsable de la redirection des données vers les modules lors d’envoi et de réceptions (voir point 3.4).**

#### **Identifiant**

**La génération de l’identifiant se fait lors du premier ajout d’un module dans le gestionnaire. La génération est effectuée avec les *Universally Unique Identifier* (*UUID*). Un *UUID* est un identifiant unique universel représenté sous la forme d’une chaîne de 36 caractères (par exemple : *550e8400-e29b-41d4-a716-446655440000*).**

**La particularité des UUID est qu’ils assurent un risque de collision (le risque que deux identifiants générés soient identiques) extrêmement minime, même s’ils sont générés et utilisés par plusieurs systèmes différents, et ceci sans nécessiter un système de génération centralisé. Ils sont également très simples à intégrer dans une application grâce aux différentes libraires existantes.**

**Cette génération a été implémentée dans l’application grâce à la librairie native Node.js *crypto*, qui offre la méthode *randomUUID*.**

#### **État et stockage**

**Le gestionnaire de module doit stocker les différents fichiers compilés (affichage, comportement et configuration) de chaque module sur le système de fichiers afin que ceux-ci puissent être chargés en mémoire, notamment lors du démarrage de l’application et de l’ajout d’un module.**

**Afin de permettre de faire le lien entre les fichiers du module et ces fichiers respectifs, chaque module est stocké dans un dossier, nommé par son identifiant unique. Le chemin du dossier principal, contenant tous les dossiers des modules est donné en paramètre du constructeur (*modulesPath*).**

**Grâce à la méthode loadModule, un module peut être chargé dans le gestionnaire à condition que celui-ci se trouve dans un sous dossier du modulePath. Pour répondre à ce besoin au niveau de l’application la fonctionnalité d’import du module va désarchiver l’archive fournie directement dans ce dossier et générer un identifiant unique.**

**Le gestionnaire de module est conçu pour ne pas être forcément restreint à utiliser un uuid comme nom de dossier. Le fait d’utiliser des identifiants et un détail d’implémentation au niveau de l’application complète. C’est pour cette raison que la méthode loadModule prend un chemin d’accès en paramètre, qui peut être un nom de dossier quelconque.**

**Lors de l’import, les trois fichiers obligatoires d’un module vont être chargé en mémoire : app.js (affichage), index.js (comportement) et config.json (configuration). Cette fonctionnalité d’import à la demande est rendue possible grâce aux import dynamique offerts par Node.js :**

const module = await import('path\_to\_file')

module.default() // Utilisation du module

**Ceci permet de charger un module JavaScript de manière asynchrone (d’où l’utilisation du mot clé *await*).**

**En cas de succès du chargement des trois fichiers, la phase de validation débute, celle-ci est identique à celle définie dans l’analyse (voir figure XXX)**

**Une fois le module validé et chargé en mémoire, le gestionnaire va maintenir une référence sur chaque module chargé en mémoire, ainsi que son identifiant et son état grâce à l’utilisation d’une structure de données *Map*.**

type ModuleId = string

type ModuleEntry = {

module: Module

enabled: Boolean

}

export default class Manager {  
 private modules: Map<ModuleId, ModuleEntry> = new Map()

}

**A Chaque action demandée, le gestionnaire appellera la méthode respective du module, en s’assurant que l’action est cohérente (par exemple, un module doit être actif pour pouvoir être désactivé). Il maintiendra également à jour le Map.**

**Le gestionnaire instancie également les différents accessors et implémentera la redirection des données vers l’accessor concerné par la demande (selon son type). Il effectue également la vérification des accès d’un module à ceux-ci.**

**Lors de l’appel à enableModule, le module est inscrit aux différents accesors qu’il a spécifié dans sa configuration (à l’aide d’attribut requires) pour lui en autoriser l’accès. Ces accès seront relâchés lors de l’appel à disableModule.**

### Module d’exemple

**Les différents concepts liés aux modules ayant été implémenté et expliqués, il est intéressant de se pencher sur la question de leur utilisation et prise en main par les développeurs souhaitant créer de nouvelles intégrations.**

**Afin de faciliter le travail du développeur j’ai mis en place un module d’exemple sous la forme d’un template qui contient toutes les dépendances nécessaires au bon développement et fonctionnement d’un module. Ce template à dans l’optique d’être mis à disposition publiquement, par exemple sur un repository GitHub. Il permet de démarrer le développement rapidement et en disposant de toutes les informations nécessaires : il définit les méthodes et explique leur utilisation pour guider le développeur. Afin de le guider pas à pas le développeur dans son travail. Une documentation sous la forme d’un readme est également fournie. Le développeur doit avoir installé l’environnement Node.js sur sa machine pour l’utiliser.**

**TODO : en annexe…**

**Après avoir implémenter son module, le développeur souhaitera pouvoir le partager afin que des utilisateurs puissent l’utiliser. La marche à suivre est la suivante :**

* ***Transpilation* : le code source, développé en TypeScript, doit être transformé en code natif JavaScript afin d’être exécutable par Node.js. Il suffit d’exécuter la commande**

npm run build

* ***Archivage* : Pour être importé dans l’application, le module soit être englobé dans une archive au format zip. Il est nécessaire que l’archive contienne au moins les trois fichiers obligatoire (app.js, index.js et config.json). D’autres fichiers peuvent y être inclus si besoin, tel que l’icône du module (spécifié dans la configuration).**
* **L’archive peut ensuite être distribuée par n’importe quel canal.**

**Le processus de création de l’archive n’a pas été automatisé dans le cadre de ce projet, mais cela peut être une piste d’amélioration pour une version future.**

## Application Web

**Ce chapitre apporte des explications sur l’implémentation de l’application modulaire s’exécutant sur la passerelle. Après que les différentes fonctionnalités liées aux modules ont été implémentée, l’étape suivante a été de concevoir une application web permettant d’interagir avec ces dernières tout en étoffant l’écosystème autour d’elles.**

**TODO : D’un point de vue logiciel**

**au niveau du monorepo, le partage a été fait de la manière suivante ...**

**

Figure 25: Rappel, architecture logique de l'application

### **Backend**

**Comme défini lors de la phase d’analyse, l’application Web est composée d’un backend, permettant d’offrir les fonctionnalités à l’interface ainsi que de proposer des méthodes permettant l’interaction avec des dispositifs locaux et d’autres applications, tout ceci au travers d’une API REST. En plus de cela, le backend est responsable d’assembler les différents composants liés aux modules présentés aux chapitres précédents.**

*La principal interet du backend est évidemment d’exposer les diverses fonctionnalités offertes par les différents composant liés aux modules qui ont été présentés au chapitre précédent. Bien que ces modules aient été conçu pour pouvoir fonctionner de manière autonome, leur intégration dans l’application à demander d’agrandir l’écosystème autour d’eux.*

#### **Base de données**

**Les différents choix établis durant l’analyse ont été conservés : la base de données a été implémenté selon l’approche relationnelle et utilise le système SQLite. Le modèle entité-association proposé à également été gardé comme base de départ pour la réalisation du modèle physique.**

##### **Modèle**

**TODO : capture modèle physique DB**

**On peut constater l’ajout de certaines colonnes par rapport au modèle conceptuel établi durant l’analyse, ces différentes valeurs ont été ajouté pour parvenir à l’implémentation de certaines fonctionnalités prévues, ainsi que pour ajouter de nouvelles fonctionnalités secondaires à l’application :**

* **Table *Users***
  + ***isDefault* : indique que l’utilisateur est le compte par défaut de l’application.**
  + ***avatar* : nom du fichier de l’avatar de l’utilisateur**
* **Table *Modules***
  + ***importedAt* : date d’importation du module dans l’application**
  + ***nickname* : nom alternatif donné à un module par un utilisateur**
  + ***icon* : nom du fichier de l’icône du module**
* **Table *Screen***
  + ***enabled* : indique si un écran est actif**
* **Table *ScreenSlots***
  + ***x* : coordonnées du slot dans la grille (axe des abscisses)**
  + ***y* : coordonnées du slot dans la grille (axe des ordonnées)**
* **Table *APIKeys* : table permettant de stocker les informations sur les clés API (voir point *4.1.3*)**
  + ***name* : le nom de la clé**
  + ***key* : le hash de la clé**
  + ***display* : la valeur affichée pour représenter la clé**
  + ***createdAt* : la date de création de la clé**

**Les diverses utilisations de ces nouveaux attributs seront détaillées plus précisément au cours de ce chapitre.**

**Il est également nécessaire de spécifier quelques contraintes supplémentaires par rapport à ce schéma :**

* **Un utilisateur ayant la valeur *isDefault* à *true* ne peux pas être supprimé. Dans l’état actuel de l’application, il ne peut exister qu’un seul compte par défaut.**
* **Il doit toujours exister au moins 1 écran dans la table Screen. Ainsi, la suppression du dernier écran disponible doit être refusée**
* **Lors de la suppression d’un écran, tous les ScreenSlots qui lui sont liés doivent être supprimés.**

**Ces différentes contraintes ont été appliquées au niveau de la logique métier de l’application.**

##### **Création**

**Lors du démarrage de l’application, le backend vérifie si le fichier de la base de données existe. Si ce n’est pas le cas, il va être créé et les différentes tables du schéma vont être créer.**

**Un « seeder » va également s’exécuter. Il insère des données de base essentielles au bon fonctionnement de l’application :**

* **Table *Box***
  + **Un nom : *Modular App***
  + **Le chemin de l’icône par défaut : *logo.svg***
  + **Le numéro de version : *1.0.0***
* **Table *Users***
  + **L’utilisateur par défaut, celui-ci ne peut pas être supprimé de l’application (*isDefault = true*)**
  + **Son nom d’utilisateur : *admin***
  + **Le hash de son mot de passe (*password1234*), il recommandé de le modifier après coup**
* **Table *Screens***
  + **Un écran par défaut, ne contenant aucun *ScreenSlot*, nommé *Default screen*.**

##### Intégration

**La base de données est désormais prête à intégrer à l’application. Pour pouvoir effectuer des requêtes vers celle-ci le module Node.js sqlite3[[17]](#footnote-17) a été utilisé. Ce module amène une série de méthodes facilitant l’utilisation d’une base de données embarquée SQLite, notamment en offrant :**

* **La gestion de la création, ouverture et fermeture de connexions vers la DB.**
* **L’exécution de requête au format SQL**
* **Permet l’utilisation de requête préparées et la *sanitization****[[18]](#footnote-18)* **des entrées.**
* **La gestion des transactions**
* **La gestion des erreurs**

**Voici un exemple de requête SQL, récupérant un utilisateur de la table *Users*, selon l’identifiant spécifié**

db.all(SELECT id, username FROM Users WHERE id = ?', [id], (err, rows) => {

if (err) {

console.error(err)

return

}

if (rows.length === 0) {

console.error(”Not found !”)

return

}

console.log(rows[0])

})

**Notez que les champs spécifiés en deuxième paramètre de la fonction *all* permet d’injecter les variables dans la requête, celles-ci vont être échappées (*sanitize*) afin d’éviter les vulnérabilités liées aux injection SQL, puis injectée pour remplacer les « ? » dans la requête.**

**Afin de pouvoir persister l’état des modules (leur état actif ou non et leur configuration), ces données doivent être persistées dans la base de données. Il est cependant nécessaire d’assurer que l’état du gestionnaire de module (qui connait l’état des modules actuel) et la base de données soient toujours synchronisés. Ce mécanisme de synchronisation permet, en cas d’interruption de l’application, que chaque module importé puisse être restauré dans l’état précédent l’interruption.**

**Pour implémenter cette synchronisation, la classe *ModuleDatabaseManager* conçue. Cette classe agit comme un Wrapper autour du gestionnaire de module *ModuleManager*. Elle définit publiquement la même interface que celui-ci, l’encapsule et lui délègue certaines responsabilité. Ce wrapper va s’assurer que pour chaque action sur un module (start, stop…), la base de données soie mise à jour, puis délègue ensuite l’action au ModuleManager qui appliquera l’action au module.**

**TODO : UML Wrapper + manager**

***ModuleDatabaseManager* est également responsable de faire en sorte que lors du démarrage de l’application, les modules soient rechargés dans l’état précédent l’interruption. En plus de cela, lors du démarrage, un contrôle supplémentaire s’effectue : une comparaison entre les modules présents en base de données et dans sur le système de fichier (géré par le gestionnaire de module) est faite :**

* **Si un module est présent en DB, mais pas sur le système de fichier, il est supprimé de la DB.**
* **Si un module est présent sur le système de fichier, mais pas en DB, il est inséré dans la DB.**

**Cette synchronisation est effectuée par la méthode *syncDBToLocalModules*.**

**Une fois cette vérification terminée, les différentes configurations enregistrées en DB sont chargées dans leur module respectif. Ainsi, les modules retrouvent l’état identique précédent l’interruption. Les modules qui étaient actifs sont ensuite démarrés.**

#### API REST

**L’API REST est le point d’accès qui offres les diverses fonctionnalités utilisées par l’interface web, permet également d’offrir des fonctionnalités à des dispositifs du réseau local. Les endpoints (points d’entrées des requêtes) proposés par l’API permettent globalement de proposer les fonctionnalités suivantes :**

* **Utilisateur : Récupération, création, modification, suppression et authentification d’utilisateurs.**
* **Modules : Récupération, import, modification, suppression, activation et envoi de données.**
* **Écrans : Récupération, création, modification et suppression**
* **Clés API : Récupération, génération et suppression ainsi que l’authentification.**
* **Box : Modification des informations**

**Les différents endpoints sont ainsi utilisés offerts aux intégrations sur le réseau local, notamment pour son utilisation au travers de l’interface client de l’application. Les différentes fonctionnalités offertes sont détaillées dans le chapitre détails l’interface client (4.2) afin d’apporter une explication visuelle supplémentaire.**

**La spécification, servant de documentation complète de cette API se trouve en annexe. Elle a été documentée selon la norme OpenAPI Specification (OAS). Il s’agit des normes et d’outils permettant principalement de décrire et de documenter des API RESTFul de manière standardisée afin d’établir une documentation exhaustive des « endpoints », du comportement et des différents formats utilisés par une API. Généralement, la documentation OpenAPI est décrite en utilisant le format YAML ou JSON.**

**Dans le cadre de ce projet, cette norme a été mise en place en utilisant Swagger. Swagger est une suite d’outils OpenSource offrant des fonctionnalités de documentation, génération et visualisation pour le développement d’API selon la norme OAS.**

**Grâce à l’utilisation de deux modules : swagger-ui et swagger-jsdoc, les spécifications de l’API peuvent être générées à partir de commentaires suivant une structure précise, spécifiés dans le code, afin de générer la documentation ainsi qu’un affichage documentation sur une interface dédiée qui offre différentes fonctionnalités telles que l’envoi de requêtes et affiche toutes les informations.**

#### Authentification

**L’API est destinée à être utilisée par des utilisateurs au travers de l’interface ainsi que par d’autres applications et dispositifs du réseau local. Elle se doit ainsi de proposer des méthodes d’authentification pour protéger les ressources et limiter les accès à certaines opérations.**

**Ce chapitre passe en revue les différentes implémentations du backend en relation avec l’authentification. Le premier choix qui a été effectué et de faire une distinction entre les différents utilisateurs potentiels de l’API :**

* ***Les utilisateurs humains* : ceux-ci vont se connecter à l’application au travers d’un formulaire (nom d’utilisateur et mot de passe) sur l’interface web afin d’être authentifier. C’est ensuite, par le biais de l’interface, qu’ils pourront effectuer des requêtes vers l’API en étant authentifié par un compte utilisateur.**
* ***Les application tierces, scripts ou dispositifs dont la transmission de données est basée sur le protocole http* : ces derniers n’étant pas utilisé directement pas un utilisateur, l’utilisation d’un nom d’utilisation n’est pas pertinente. De plus, cela demande de conserver ces informations dans l’application tierces souhaitant utiliser l’API. On doit donc faire confiance à l’application tierce quant à la sécurité qu’elle implémente, à risque de faire fuiter les informations de connexion d’utilisateur humains.**

**Pour répondre aux besoins spécifiques de chacune de ces deux catégories d'utilisateurs, deux approches distinctes d'authentification ont été adoptées : les JSON Web Token (JWT) et les clé API (API Key). Leur concepts et implémentation respective sont détaillés dans chapitres suivantes.**

##### Utilisateurs et JWT

**Le processus d’authentification des utilisateurs se fait sur la base d’un nom d’utilisateur et d’un mot de passe. Lors de la création d’un compte utilisateur, la valeur du mot de passe est hachée et enregistré dans la base de données, le mot de passe n’y est ainsi pas stocké en clair.**

**Cette approche est une obligation en termes de sécurité, car si l’on stocke les mots de passe en clair, on s’expose à de grands risques : les mots de passes peuvent être lus par un attaquant ayant accès gagné un à la base de données. Ce qui compromet toute notion de sécurité interne de l’application (niveaux de permissions, paiements etc.). Il est également fort probable que certains utilisateurs utilisent le même mot de passe dans d’autre services, ce qui augmente encore la portée des risques à d’autres systèmes.**

**Le hachage est effectué avec l’algorithme argon2, grâce au module Node.js argon2 (**[**https://www.npmjs.com/package/argon2**](https://www.npmjs.com/package/argon2)**). Argon2 est un algorithme de hashage dont l’utilisation vise spécifiquement le hachage des mots de passes**

**Lors de la connexion d’un utilisateur, la valeur du mot de passe saisie est hachée et comparée avec le hash stocké en base de données, grâce à argon2. Si la comparaison est valide, l’utilisateur va être authentifié et recevoir un JWT.**

**JSON Web Token est un standard permettant de créer des jetons autonomes, sous la forme d’un objet JSON encodé en base64, permettant de transmettre des informations de manière sécurisée entre divers services. L’intégralité des informations contenue dans un JWT peuvent être vérifiée grâce à un secret (défini par le serveur ayant généré le token (issuer)) ou par une paire de clés publique et privée. Voici un exemple de JWT :**

eyJhbGciOi-JIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik5pY29sYXMiLCJpYXQiOjE1MTYyMzkwMjJ9.wXeXE92HXWgd6tVSv-Ud4QLFipRhuV7HmScCUC6vHXM

**Un JWT est constitué de trois parties : l'en-tête (*header*), la charge utile (*payload*) et la signature. Chacune d’entre elle est séparée par un point.**

**L'en-tête contient des informations sur le type de token et l'algorithme de hachage utilisé :**

{

"alg": "HS256",

"typ": "JWT"

}

**La charge utile contient les données (*claims*) telles que l'identifiant de l'utilisateur et les autorisations**

{

"sub": "1234567890",

"name": "Nicolas",

"iat": 1516239022

}

**La signature permet de vérifier l'intégrité du JWT.**

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), ‘TB-2023’)

**Lorsqu’un serveur reçoit un JWT, il sépare les trois parties, notamment pour valider la signature : il recalcule la signature avec la clé de génération pour vérifier que son contenu n’a pas été modifié.**

**Il faut savoir qu’un JWT dispose d’une date d’expiration, fixée par le serveur issuer. Dès lors, il n’est pas possible de révoquer ce token. Pour cette raison il est généralement recommandé de définir une durée de validité relativement courte dans l’éventualité où un token serait volé par un attaquant, ce qui réduirait sa marge de manœuvre. Une durée trop courte importunera cependant l’utilisateur, qui devra s’authentifier à nouveau lors de l’expiration du token. Une autre solution peut-être de conserver les tokens générés en base de données et d’effectuer une vérification lors de son utilisation, cette approche permet une révocation de l’accès d’un certain jeton à l’application (sans révoquer le jeton en lui-même).**

**Revenons à l’implémentation dans le contexte de l’API. La mise en place de l’authentification par JWT à demander l’utilisation de du module JavaScript *jsonwebtoken****[[19]](#footnote-19)****.* Grâce à cette librairie et à un secret (clé privée) généré aléatoirement et conservé dans un fichier. env, des JWT peuvent être générés et vérifiés. Il a été choisi une durée de validité de 4 heures pour les tokens.**

**Dès lors, une requête sur l’API permet de récupérer un nouveau token en cas d’authentification valide par nom d’utilisateur et mot de passe. Ce token doit ensuite être transmi dans les en-tête http de chaque requête envoyée depuis l’interface web pour pouvoir identifier l’utilisateur connecter, on utilise le header Authorization avec le préfixe Bearer.**

"Authorization": Bearer <token>

**Lors de chaque requête, le serveur va vérifier la validité du token (signature et expiration) et les données qu’il contient peuvent également être décodées. Le mécanisme d’authentification de l’application inclus les données suivantes dans le token, permettant d’authentifier un utilisateur :**

export type ReqUser = {

id: string

username: string

}

##### Clés API

**L’authentification par clé API est un mécanisme bien plus simple que les JWT. Une clé API est une chaîne de caractères, généralement alphanumériques, générés aléatoirement, unique au sein de l’application qui les génère. Elle offre l’avantage de pouvoir servir d’identification pour les applications tierces ou les clients autorisés à accéder à une API. Une clé API ne contient pas d’informations, contrairement à un JWT.**

**Le processus d’authentification pour l’application tierce est très simple, elle se voit attribuer une clé d’API (processus de génération unique) qu’elle doit spécifier dans l’entête des requêtes http. La pratique courante dans les services RESTFul est de spécifier la clé dans le header x-api-key :**

"x-api-key": <key>

**Au niveau de l’implémentation dans ce projet, la génération de clés est mise en place en utilisant la librairie native Node.js *crypto*, qui offre la méthode *randomBytes****[[20]](#footnote-20)* **permettant de générer un nombre de bytes (32 dans notre cas). Le résultat est ensuite converti en hexadécimal pour obtenir une clé alphanumérique.**

const key = randomBytes(length).toString('hex')

**Un fois cette clé générée, elle va être hachée en utilisant, à nouveau, l’algorithme argon2. La clé n’est pas stockée en clair dans la base de données. Seul son hash, et une valeur d’affichage seront enregistré (cette valeur correspond aux quatre premiers caractères de la clé et des astérisques, par exemple : *db65f∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗*). La clé sera renvoyée une seule fois à l’utilisateur, il ne sera plus possible de connaitre sa valeur après cela.**

**Comme pour les mots de passe, le hachage des clés d’API permet d’éviter les fuites de données, principalement pour éviter la lecture des clés dans la base de données (étant une base de données embarquée, son accès n’a pas restreint pas une authentification). Même si dans le cadre de cette application, les risques liés au vol d’une clé API ont moins de conséquence que le vol d’un mot de passe.**

**Lors de l’utilisation d’une clé dans une requête, celle-ci va être hachée et comparée avec les autres hash des clés existantes stockés dans la base de données.**

#### Architecture du code

##### **Controller, service et Repository**

**Le code nécessaire à la réalisation de l’API REST a été implémentée en suivant l’approche Controller – Service – Repository. Cette architecture de décompose donc en trois couches :**

* ***Controller* : Le contrôleur est responsable de la gestion des requêtes entrantes et de la coordination des actions à effectuer. Le contrôleur prend les données d'entrée, effectue les validations nécessaires et appelle le service approprié pour exécuter l'action demandée. Le contrôleur renvoie ensuite les réponses selon un format définition (JSON).**
* ***Service* : Le service est responsable de l’application de la logique métier de l’application. Il est chargé de traiter les requêtes en provenance du contrôleur, d'effectuer des opérations, de gérer les règles de validation relative au domaine métier et de coordonner les interactions avec le repository approprié. Le service encapsule la logique de l'application et assure une abstraction entre le contrôleur et le repository, permettant une réutilisation facile et une meilleure séparation des préoccupations.**
* ***Repository* : Le repository est responsable de l'accès et de la manipulation des données. Il fournit des méthodes pour lire, mettre à jour et supprimer des données provenant d’une source de données, pouvant être une base de données, un système de fichiers, une API externe, etc. Le repository cache les détails de stockage des données et fournit une interface simple et abstraite pour le service métier afin que celui-ci n'ait pas besoin de connaître les détails d'accès aux données.**

**C’est une architecture standard dans la conception d’API et d’applications web, notamment car cette architecture est imposée par la plupart des Frameworks web utilisés dans l’industrie (Java Spring, Nest.js, Laravel etc.). Cette approche amène de nombreux avantages : elle offre une séparation claire des préoccupations et facilite la gestion du code en le divisant en différentes couches logiques. Cela permet de maintenir le code plus propre, plus modulaire et plus facilement testable.**

##### **Middlewares**

**L’API est réalisée avec le Framework Express.js, celui-ci se base sur l’utilisation de middlewares. Un middleware est une fonction s’exécutant généralement en début ou en fin d’une requête dans le but d’effectuer des opérations sur la requête ou la réponse. Cette approche a été très utile dans l’implémentation, notamment pour les fonctionnalités d’authentification (validation des JWT et clés des en-têtes de la requête) et la gestion des erreurs.**

Au niveau de la gestion des erreurs, en ensemble de classes ont été créés. Celles-ci permettent de lever des exceptions qui seront capturée par un middleware.

enum HTTPStatus {

FORBIDDEN = 403

// ...

}

export class HTTPError extends Error {

constructor(public message: string, public status: HTTPStatus, public name: string) {

super(message)

}

public toResponse() {

return {

message: this.message,

}

}

// ...

}

export class ForbiddenError extends HTTPError {

constructor(message: string) {

super(message, HTTPStatus.FORBIDDEN, 'Forbidden')

}

}

**Ces classes servent ainsi de base pour toutes les erreurs liées au contexte métier de l’application.**

export class ModuleDisabledException extends ForbiddenError {

constructor(id: string) {

super(`Module with id ${id} is disabled`)

}

}

**Lorsqu’elles sont capturées par le middleware, elles sont transformées pour être retournée à l’utilisation au format JSON grâce à la méthode *toResponse*.**

#### Évènements en temps réel

**Une des fonctionnalités principales de l’application modulaire est de pouvoir afficher en temps réel les mises à jour des affichages des modules ainsi que les modifications apportées aux écrans de visualisation.**

**Lors de la phase d’analyse, le choix avait été défini sur l’utilisation des SSE pour l’implémentation des fonctionnalités d’évènement en temps réel, mécanisme de communication unidirectionnel http dans lequel le serveur envoie des évènements sur une connexion maintenu ouverte avec le client. Malgré le fait que la communication unidirectionnelle convient bien à l’utilisation dans ce projet, après plusieurs semaines de développement, plusieurs limitations ont été atteinte :**

**Initialement, l’approche choisie prévoyait de créer une connexion SSE par affichage de module. Bien évidement je me suis rapidement rendu compte que cela n’était pas la bonne approche au niveau des performances et de l’utilisation de ressource, mais qu’en plus que les navigateurs ont des limites de connexions simultanées SSE qu’il est possible d’utiliser. Cette limite est fixée à 6 connexions par navigateur et par domaine. Il est donc devenu évident que la gestion des affichages des modules doit se faire dans une unique connexion.**

**TODO : à terminer**

Un autre problème rencontré, cette fois-ci lié à la manière de fonctionner des SSE : est qu’il n’est pas nativement possible de détecter la déconnexion d’un client (la connexion restant ouverte pour gérer les éventuelles reconnexions). Ceci est problématique dans l’approche d’implémentation des évènements en temps réels de l’application, principalement ceux des modules : lors qu’un module n’est pas affiché sur un écran, il n’est pas nécessaire de s’enregistrer à ces évènements de rendu. Il est donc nécessaire de pouvoir

Il y a des mécansimes (heartbart, timeout…)

il n’est pas possible de détecter la déconnexion d’un client, et donc de savoir si un module est toujours affichés sur un écran et si il est donc toujours nécessaire de transmettre des évenements de mise à jour de l’affichage au client. Cet aspect s’est révélé très problématique : il n’était pas possible de se désinscrire des événements des modules, cela donnait donc une gestion inefficace des observeurs et des potentielles fuites de mémoires

Il me fallait un moyen de pouvoir venir s’enregsitrer et relacher sans devoir relacher la connexion

Car il est fait en sorte que les évenements d’un module ou d’un écrans ne soient transmi que si un composant en à besoin (composant module render ou screen)

De plus, le passage aux web socket à permit de rehausser la limite de 6 onglets à plus de 200 (selon navigateur) (lien)

Ce passage au websocket à également permi d’implémenter la mise à jour des écrans de visualisation et de l’état (actif ou non) des modules en temps réel sur les clients

TODO : parler de la structure (type) utilisé pour les échanges ({type, subtype …..})

Et de la classe event front

**Points à traiter :**

**Gestion des événements en temps réel : passage aux websockets**

* **Websocket**

### Interface

L’interface de l’application web est la pièce finale qui vient offrir toutes les fonctionnalités offertes par le backend en fournissant une présentation interactive et ergonomique des différentes données et actions.

Dans ce chapitre, les différentes fonctionnalités offertes par l’interface seront présentées et des détails d’implémentation seront ajoutés.

Une première section, plus théorique, vient amener des informations de compréhension sur l’utilisation de la librairie React.

#### React

React : hook et contextes

#### Fonctionnalités

##### Authentification

Ce chapitre passe en revue les différentes fonctionnalités présentées sur les différentes pages.

#### Internationalisation

**Au niveau de l’interface de l’application, le concept d’internationalisation (généralement abrégé par *i18n*) a été appliqué. Il s’agit d’un processus de développement dont l’objectif est de rendre une application utilisable dans différentes langues et cultures. L’approche consiste principalement à faciliter la « localisation » de l’application » en séparant rigoureusement le contenu textuel du code source de l’application. Les différents éléments (textes, dates, devises etc.) deviennent ainsi facilement remplaçables.**

**Dans le cadre de ce projet, l’approche de l’internationalisation a été adoptée pour permettre ses fonctionnalités de base : les divers contenus textuels ont été isolé du code source vers des fichiers de langues de manière pour ajouter la possibilité de changer de langue sur l’interface. Les langues gérées actuellement sont le Français et l’Anglais. L’avantage est que l’ajout de nouvelle langue est désormais très simple, il suffira d’ajouter un nouveau fichier de langue contenant les traductions de la langue souhaitée.**

**Au niveau de l’implémentation, le module « react-18next**[[21]](#footnote-21)**» a été utilisé. Ce dernier permet de facilement mettre en place l’internationalisation d’une application frontend React. Les différents fichiers de langues sont définis au format JSON puis chargé dans le module pour les rendre accessibles :**

{

"menu": {

"home": "Home",

"user\_menu": {

"profile": "Profile",

"about": "About",

"logout": "Log out"

}

},

//...

}

Todo : exemple de fichier de langue (anglais) au format JSON

**La possibilité de changer la langue de l’interface a ensuite été ajoutée grâce à un sélecteur de langue, situé sur la page de paramètre de l’interface.**

**Il n’a cependant pas été mis en place d’internationalisation au niveau du backend.**

### Production

**Comme définir lors de l’analyse, la création d’une image Docker a permis de fournir un moyen d’utiliser l’application en production. Cette image défini le modèle de référence pour la création de containers destinés à héberger l’application. Les containers créés selon cette image embarquent toutes les dépendances nécessaires à l’exécution de l’application, notamment l’environnement Node.js (version 18). Lors de la création du container, les dépendances de l’application sont installées, le code TypeScript est transpilé et l’application (backend) est démarrée.**

**Deux volumes sont également utilisés, un pour stocker les modules (c’est le chemin vers ce volume qui sera géré par le gestionnaire de module) et un second pour y conserver le fichier de la base de données SQLite). L’interêt d’utiliser des volumes est d’assurer la persistance des données en cas d’extinction des containers. Il serait donc possbile de remplacer les containers pour utiliser une nouvelle version de l’application sans perdre les données et les modules.**

**Afin de rendre l’interface web accessible, un serveur web Nginx a été mis en place. Les fichiers statiques issus des différentes phases de transformation du code du frontend sont copié à la racine du serveur afin d’y être hébergé. L’utilisation de Nginx permet également de faire configuration nécessaire au bon fonctionnement du routage côté client (react-router), car pour que celui-ci soit fonctionnée, toutes les requêtes doivent être redirigée vers le fichier index.html.**

**Il aurait également été possible de gérer l’hébergement du frontend au travers du serveur Express mais j’ai préféré dissocier totalement le frontend du backend.**

**Ainsi, deux containers sont nécessaires pour exécuter l’application. Pour faciliter la création**

**TODO Docker compose pour orchestrer les**

Docker compose

ngnix

Afin de faciliter

Les containers

Parler des volumes

Image de l’application c

Ce chapitre parle du passage en production sur le raspberry. De l’image docker…

## Proof of Concept

**Le développement du module « Proof of Concept » amène un cas d’utilisation concret de l’application en créant une intégration entre un logiciel existant et un dispositif local relié à la passerelle. Il permet d’évaluer le bon fonctionnement de l’application, du processus de développement d’un module ainsi que son intégration dans l’écosystème de l’application modulaire.**

**L’objectif de ce module est de créer une intégration avec le logiciel Composal de l’entreprise YALK**

### **Matériel et ressources**

**Pour réaliser cette intégration, l’entreprise YALK a fourni du matériel, de la documentation et des accès à l’API du logiciel Composal sur une instance de test attribuée à ce projet.**

Rapsberry PI

Lecteur de carte RFID, parler des deux types

Pointage Théorique sur RFID

### Module

Pour évaluer la facilité de prise en main

**Parti du module d’exemple pour prouver que la définition du module fonctionne et que la structure peut être facilement prises en main pour créer des intégrations**

**Points à traiter :**

* **Lecteur RFID**
* **L’API de Composal (url, token)**
* Fonctionnement (shéma)
* Problèmes rencontrées (2 types de lecteur + marchait pas)

Affichage : nom, photo, heure actuelle et différence avec l’heure de timbrage estimé

Utiilisation des accesseurs http et keyboard

## Bilan et perspectives

TODO : Dans ce chapitre j’analyse la solution que j’ai proposé pour en tirer les points forts et les points à améliorer pour que la solution puisse être exploitable.

Je passe également en revu les différentes perspectives d’évolution liées à chacun des composant de l’application, dans l’objectif de démontrer le travail de réflexion qui ont justifier les différents choix d’implémentation.

Je pense que l’application à dépassé le stade de prototype. Malgré le fait que cetraines fonctionnalités peuvent encore être grandement améliorées, l’application pourrait tout à fait être utilisée en production, mais plutôt dans un cadre privé (utilisation loisir). Pour être exploitable en entreprise, je pense que plusieurs points sont à revoir.

Voici quelques axes sur lesquels s’est porté ma reflexion quand aux perspectives d’amélioration et de nouvelles fonctionnalités.

## Conclusion

TODO : Conclusion sur le projet, interet personnel, ressenti, satisfactions, remerciements, liberté du projet

# 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**

1. https://www.odoo.com/fr\_FR/app/iot-faq [↑](#footnote-ref-1)
2. ****E**nterprise **R**esource **P**lanning, ensemble d'applications permettant de gérer les processus de gestion d'une entreprise** [↑](#footnote-ref-2)
3. <https://www.home-assistant.io/> [↑](#footnote-ref-3)
4. https://nodered.org/ [↑](#footnote-ref-4)
5. “**Gang of Four”, auteurs du livre Design Patterns: Elements of Reusable Object-Oriented Software** [↑](#footnote-ref-5)
6. **https://developer.mozilla.org/fr/docs/Web/API/Fetch\_API/Using\_Fetch** [↑](#footnote-ref-6)
7. **https://developer.mozilla.org/en-US/docs/Web/API/Server-sent\_events** [↑](#footnote-ref-7)
8. **https://github.com/features/actions** [↑](#footnote-ref-8)
9. **https://www.raspberrypi.com/products/raspberry-pi-4-model-b/** [↑](#footnote-ref-9)
10. **https://pm2.keymetrics.io/**  [↑](#footnote-ref-10)
11. https://developer.mozilla.org/en-US/docs/Web/API/structuredClone [↑](#footnote-ref-11)
12. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional\_chaining [↑](#footnote-ref-12)
13. https://json-schema.org/ [↑](#footnote-ref-13)
14. https://ajv.js.org/ [↑](#footnote-ref-14)
15. **https://react.dev/reference/react-dom/server/renderToStaticMarkup** [↑](#footnote-ref-15)
16. https://tailwindcss.com/blog/tailwindcss-v3#just-in-time-all-the-time [↑](#footnote-ref-16)
17. https://github.com/TryGhost/node-sqlite3 [↑](#footnote-ref-17)
18. Action de nettoyer les données d'entrée utilisateur afin d'éliminer tout contenu malveillant ou caractères spéciaux potentiellement dangereux, afin d'empêcher les attaques d'injection SQL. [↑](#footnote-ref-18)
19. https://github.com/auth0/node-jsonwebtoken [↑](#footnote-ref-19)
20. **https://nodejs.org/api/crypto.html#cryptorandombytessize-callback** [↑](#footnote-ref-20)
21. https://react.i18next.com/ [↑](#footnote-ref-21)