NRCP Concepteur développeur d’applications

[1 Contexte SOS Immo 4](#_Toc123917096)

[1.1 Fonctionnement des services immobiliers 4](#_Toc123917097)

[1.2 Transformation de la gestion de suivi d’incident 4](#_Toc123917098)

[1.3 Objectifs 5](#_Toc123917099)

[1.4 Besoins et limites de l’application 5](#_Toc123917100)

[2 Conception 6](#_Toc123917101)

[2.1 Use Case 6](#_Toc123917102)

[2.2 Profils utilisateurs 6](#_Toc123917103)

[2.3 Architecture client-serveur 7](#_Toc123917104)

[2.3.1 Front End et React 8](#_Toc123917105)

[ Composant 8](#_Toc123917106)

[ UseState ou variable d’état 8](#_Toc123917107)

[2.3.2 Back End – structure à couches 12](#_Toc123917108)

[2.3.3 Base de données 14](#_Toc123917109)

[ Utilisateurs - Habilitations 15](#_Toc123917110)

[ Prestataires - Types d’incidents 16](#_Toc123917111)

[ Emplacements – Types d’emplacements 18](#_Toc123917112)

[ Incidents 19](#_Toc123917113)

[ Journaux 21](#_Toc123917114)

[3 Cinématiques utilisateur et écrans principaux 22](#_Toc123917115)

[3.1 Usager 23](#_Toc123917116)

[3.2 Technicien 28](#_Toc123917117)

[3.3 Valideur 29](#_Toc123917118)

[3.4 Admin 30](#_Toc123917119)

[4 Sécurité 32](#_Toc123917120)

[4.1 Utilisateur 32](#_Toc123917121)

[4.1.1 création 32](#_Toc123917122)

[4.1.2 Authentification 34](#_Toc123917123)

[4.2 Gestion de mot de passe 36](#_Toc123917124)

[4.2.1 Hachage 36](#_Toc123917125)

[4.2.2 Salage 36](#_Toc123917126)

[4.3 Identifiants UUID 37](#_Toc123917127)

[4.4 Cookie 39](#_Toc123917128)

[4.4.1 Besoin 39](#_Toc123917129)

[4.4.2 Principe 39](#_Toc123917130)

[4.4.3 Exemples 39](#_Toc123917131)

[4.5 Variables d’environnement 41](#_Toc123917132)

[4.6 Archivage et sauvegarde 42](#_Toc123917133)

[4.6.1 Archivage 42](#_Toc123917134)

[4.6.2 Base de sauvegarde 43](#_Toc123917135)

[4.6.3 Utilisation de triggers 43](#_Toc123917136)

[5 Tests 45](#_Toc123917137)

[5.1 Tests unitaires 45](#_Toc123917138)

[5.2 Base de tests 47](#_Toc123917139)

[5.3 Tests d’intégration et de non-régression 47](#_Toc123917140)

[6 Organisation 47](#_Toc123917141)

[7 Evolutions 48](#_Toc123917142)

[7.1 Fonctionnelles 48](#_Toc123917143)

[7.2 Techniques 49](#_Toc123917144)

[8 Bilan 50](#_Toc123917145)

[Ce que je savais déjà 50](#_Toc123917146)

[8.1 Découvertes de la formation mise en pratique pour le projet 50](#_Toc123917147)

[8.2 Recherches personnelles. 50](#_Toc123917148)

[8.3 Point sur l’application à aujourd’hui 51](#_Toc123917149)

[ANNEXES 52](#_Toc123917150)

[Annexe 1 – rendu du pipeline GitHub 52](#_Toc123917151)

[Annexe 2 – exemple de scénario complexe 53](#_Toc123917152)

[Annexe 3 - Collections Postman 55](#_Toc123917153)

[Annexe 4 - schéma déploiement 56](#_Toc123917154)

Introduction

Je m’appelle Sophie Joffre. Pendant près d’1 an, j’ai suivi un cursus de formation pour apprendre le métier de développeur en informatique.

Cette formation m’a été proposée par mon employeur Société Générale qui, grâce à un partenariat avec l’Ecole Efrei Paris, propose à ses salariés de changer d’orientation professionnelle.

L’investissement pour l’entreprise et pour le salarier est énorme et je suis heureuse de pouvoir maintenant présenter mon projet de fin d’étude à l’examen du Titre Professionnel de « Concepteur développeur d’applications ».

Le projet s’appelle SOS Immo.

C’est un outil de gestion de tickets, ces tickets étant des incidents techniques pouvant survenir dans un immeuble de bureau de grande taille.

Grace à cet outil, les occupants de l’immeuble pourront, signaler un incident, le commenter, demander à tel ou tel technicien de s’en occuper, signaler la résolution, donner leur avis.

Le choix d’une telle application s’est imposé pour plusieurs raisons :

* Elle correspond parfaitement à mes appétences : mettre en place des applications qui facilitent la vie des gens et les aide dans leur travail.
* Après 25 ans à la Défense dont environ 10 ans comme élue au CSSCT, anciennement CHSCT, je connais le fonctionnement « technique » d’un tel immeuble.
* Ce projet contient les éléments nécessaires au passage du NRCP.

Remerciements

Je tiens à remercier bien sûr les professeurs et les autres élèves de la cohorte, mes nouveaux managers et toutes leurs équipes. Pour leur confiance, leur patience et leur pédagogie.

Sans oublier les responsables du service immobilier de l’immeuble Basalte de la Défense qui m’ont beaucoup aidée lors de la conception du projet sans rien attendre en retour.

Abstract

My name is Sophie. I followed a training course for one year to learn the job of IT developer.

This training was proposed to me by my employer Société Générale which, thanks to a partenership with the Effrei Paris school, offers its employees, just like me, the possibility to change of professional orientation.

This is a big investment for the company and I and I’m now happy to be able to present my end-of -study project for the Professional Title of « Designer and developer of applications » exam.

The project is called SOS Immo.

It’s a ticket manager tool, these tickets are technical incidents that may append in a tall office building.

With this tool, every occupants of the building will be able to report an incident and comment it.

The technicians will be able to take care and report the resolutions.

The managers will be able to know which technicians take care which indicents.

# Contexte SOS Immo

## Fonctionnement des services immobiliers

Bien sûr, je ne me permettrais pas d’affirmer que toutes les entreprises fonctionnent sur le même modèle.

Je prends ici l’exemple que je connais.

Les tours de grandes tailles sont de véritables fourmilières qui abritent de nombreux corps de métiers.

Le service de gestion d’immeuble nécessite une organisation bien rodée afin qu’employés de bureau et ouvriers puissent chacun effectuer leur travail sans se gêner mutuellement. Les uns, la plupart du temps, totalement ignorants de la présence des autres.

L’entreprise occupante (propriétaire ou locataire de l’immeuble) n’intervient pas elle-même lorsque survient un incident technique. Elle utilise les services d’entreprises prestataires spécialisées dans leur domaine (plomberie, ascensoriste, ménage, etc.).

Des contrats sont passés à la suite d’appels d’offre. Ils sont signés pour plusieurs années.

Ces entreprises retenues emploient des équipes qui ont leur propre ligne hiérarchique et rendent compte à l’entreprise occupante.

Les techniciens « prestataires » ont, sur place et à demeure, leurs propres vestiaires, bureaux, locaux pour pouvoir intervenir à tout moment dans un délai convenu.

## Transformation de la gestion de suivi d’incident

Historiquement la centralisation des incidents est assurée par un standard téléphonique, des cahiers, post’it, dash board, etc.

Dans un contexte de digitalisation des process et pour gagner en efficacité, on a imaginé une application qui remplacerait ce standard en allant plus loin encore : une plateforme unique pour tous les occupants de l’immeuble, qu’ils soient employés de bureau, intervenants extérieurs ou responsables d’immeuble.

L’information serait alors rendue disponible en temps réel à tous les acteurs concernés, sans intermédiaire.

Une plateforme de ce genre est en place aux services centraux de Société Générale depuis quelques années. Je reprends l’idée à mon compte pour ce projet de fin d’étude

Pour construire ce projet, je suis allée rencontrer les premiers concernés : des responsables d’immeuble de grande taille[[1]](#footnote-1).

Ils m’ont expliqué comment ils utilisent l’outil et leurs besoins en la matière.

## Objectifs

Objectif principal

Aider au suivi des incidents bien sûr.

Objectifs secondaires

Aider les services techniques à identifier les points de vigilances : prestaires potentiellement en sous-effectif, étages ou types d’incidents particulièrement récurrents.

## Besoins et limites de l’application

Les équipes prestataires sont autonomes mais le service de gestion d’immeuble reste maître et garde le dernier mot.

Le service de gestion d’immeuble est également responsable des données et droits d’accès de l’application.

Il a donc un profil Admin.

N’importe quel occupant de l’immeuble, quel que soit son profil, doit pouvoir faire un signalement et le suivre jusqu’à résolution.

Un incident doit être résolu dans un délai convenu, ici 24 heures.

L’affectation d’un technicien à un incident peut se faire de 3 façons :

* Auto-affectation.
* Affectation par un responsable de l’équipe prestataire.
* Affectation par le service de gestion d’immeuble.

L’attribution d’un incident à une entreprise prestataire se fait de 2 façons :

* Au signalement, calcul par l’outil d’après le type d’incident.
* Réattribution par le service de gestion d’immeuble le cas échéant.

À tout moment, de la création à la clôture d’un incident, un commentaire doit pouvoir être ajouté.

* Usager : sur ses signalements.
* Technicien : sur les incidents qui lui sont affectés.
* Valideur : sur les incidents affectés à son équipe.
* Admin : sur n’importe quel incident.

Seuls les incidents techniques courant sont suivis. Pas les gros travaux, ni les travaux planifiés.

L’outil ne doit contenir qu’un minimum d’informations sensibles.

Il est administré par le service d’immeuble, pas le service d’achat ni les ressources humaines. Il ne s’agit donc pas de gérer les contrats.

Les occupants n’ont pas accès directement à leurs données, données qui devront être réduit au strict minimum pour le suivi des incidents.

En amont, l’administrateur à connaissance de chaque nouvelle arrivée et créée les comptes avec droits d’accès selon le profil de l’utilisateur. Ce process n’est pas géré par l’application.

# Conception

## Use Case



## Profils utilisateurs

**2 profils internes** – salariés de l’entreprise occupantes

* Admin :
  + A la fois service de gestion d’immeuble et administrateur de l’application.
  + A tous les droits sur l’outil.
* Usager :
  + Tout autre employé interne.
  + Utilise l’outil pour signaler les incidents et valider la fin d’intervention.
  + Seul profil non « technique ».

**2 profils externes** – salariés des entreprises prestataires[[2]](#footnote-2)

* Techniciens
  + En charge de la résolution des incidents.
  + Utilise l’outil pour suivre les incidents non résolus le concernant.
* Valideur :
  + Responsable d’une équipe de techniciens.
  + Utilise l’outil pour la coordination de son équipe.

## Architecture client-serveur

Une application moderne doit permet à plusieurs utilisateurs, chacun sur leur propre machine, de communiquer entre eux et d’accéder aux mêmes données en même temps.

Tout cela se fait par l’intermédiaire d’une machine centrale, appelée « serveur ». Cet intermédiaire est une étape totalement transparente pour les utilisateurs.

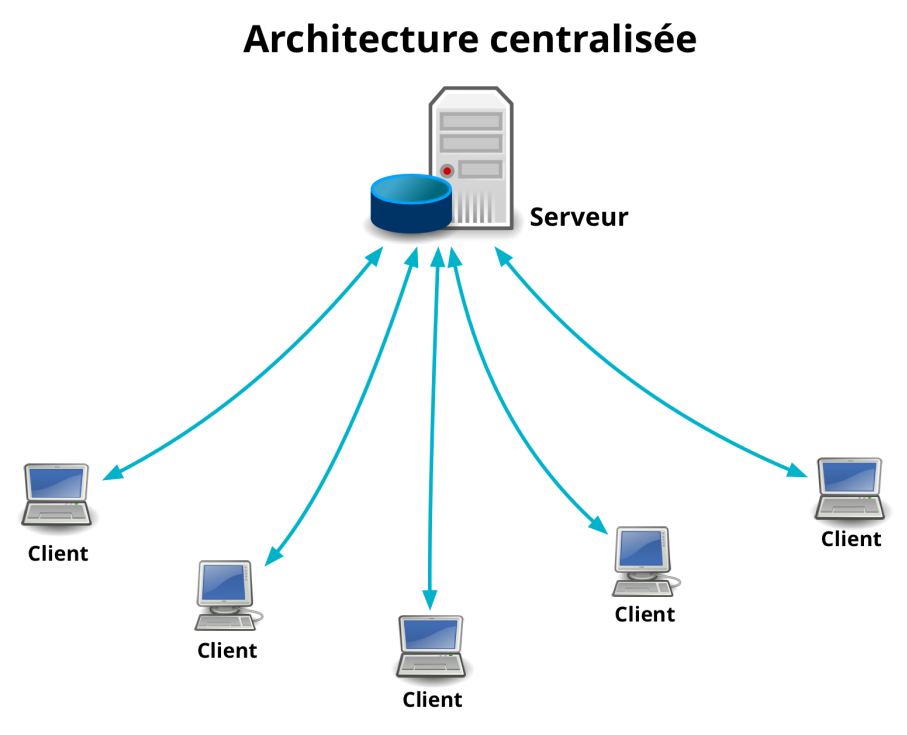
L’architecture client-serveur est composée de 2 parties totalement distinctes :

1. Le **Front end** est implanté dans chaque machine d’utilisateur.

C’est la partie visible du programme, l’interface qui permet de naviguer entre les différentes fonctionnalités.

1. Le **Back end** est implanté 1 seule fois, dans le serveur.

Il s’occupe de tous les calculs et du stockage des données.

Ainsi tous les utilisateurs ont le même niveau d’information.

Le client envoie une requête au serveur, généralement par le biais d’une interface graphique.

Le serveur traite la requête et retourne une réponse.

SOS Immo est conçu avec une architecture similaire.

Les 2 parties back et front sont développés comme 2 projets indépendants.

La communication entre back et front se fait au moyen d’adresses URL, via un navigateur web au choix de l’utilisateur.

Les 2 projets sont développés en Java Script.

Ils ont nécessité l’installation de packages, npm et node notamment. Ces packages permettent à leur tour l’installation et l’appel aux librairies javaScript.

### Front End et React

Le Front utilise la librairie React qui permet la création d’interfaces utilisateurs interactives.



Le concept de React est assez simple : 1 seule page « index.js » pour tout le site et une multitude de composants pour l’habiller.

Ces composants sont déclenchés si les conditions sont réunies et peuvent s’appeler les uns les autres. On parle alors de parents et d’enfants.

Par convention, « index.js » appelle un composant unique « App.js » qui servira de gare de triage pour les autres composants.

Index.js – page unique du front

Une image contenant texte

Description générée automatiquement

#### Composant

Un composant est un élément de page. Il contient

* Du code html pour l’affichage.

Le rendu visuel peut être complexe ou très simple : un tableau, un bouton, un formulaire, etc.

* Des fonctions qui lui permettent d’interagir avec le Back.
* Des variables ou autres éléments propres à React, tels que des useStates.

#### UseState ou variable d’état

Ce sont des éléments composés d’une variable[[3]](#footnote-3) (l’état) et d’une fonction (pour la mise à jour de l’état). Ils ont la particularité d’être transmissibles d’un composant parent à un composant enfant, et leur valeur peut être mises à jour par un enfant.

Quand un useState est mise à jour, le composant parent dans lequel il a été déclaré est recalculé.

L’état de la useState est préservée tant que le composant en fonction[[4]](#footnote-4).

Lors de l’appel à un enfant, le parent lui transmet des attributs (qui peuvent ou non être des useStates). L’ensemble de ces attributs est récupéré par l’enfant dans un seul objet « props ».

App.js – composant principal

Une image contenant texte

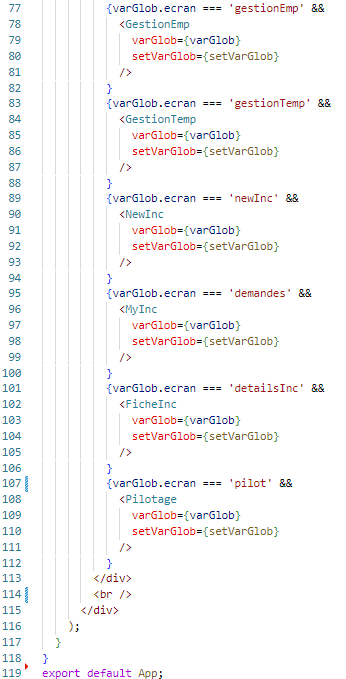
Description générée automatiquement

Import : librairie React et bootstrap, fichier .css et ensemble des composants enfants appelés par App.js

Une image contenant texte

Description générée automatiquement

Déclaration du useState. Pour des raisons pratiques j’ai créé un seul useState varGlob, de type objet, contenant les éléments influant l’appel aux composants enfants et donc susceptibles d’être mis à jour par ces enfants.



Retour du code http : lancement de 2 composants à la fois

* « Ribbon.js » automatiquement
* Un autre composant conditionné aux valeurs de varGlob.

Tous les composants enfants reçoivent en attribut le useState varGlob. Sa mise à jour modifie les conditions de lancement du second composant.

**Mise en pratique**

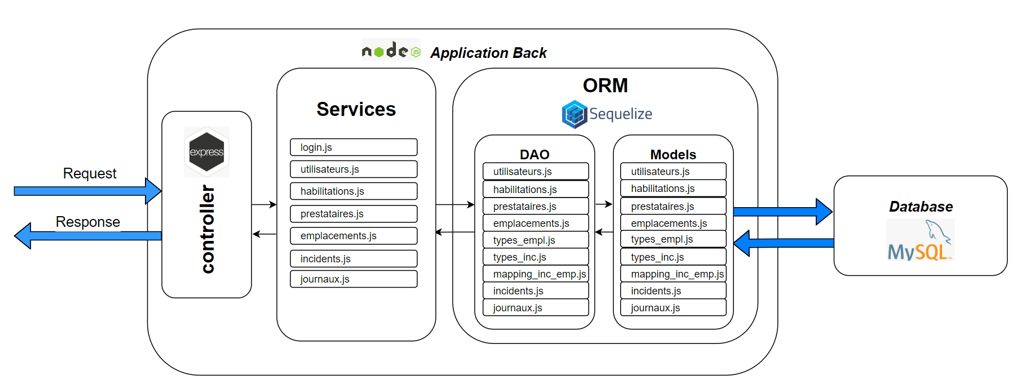
|  |  |  |  |
| --- | --- | --- | --- |
| Etape | varGlob | Composants actifs | Visuel |
| **Démarrage** | ecran : «login»  nom : «»  profil : «» | Ribbon.js  Login.js |  |
| varGlob.ecran est initialisé : « login ».  Déclenchement des composants Ribbon et Login. | | | |
| **Authentification réussie** | ecran : «Menu»  nom : «toto»  profil : «usager» | Ribbon.js  Menu.js[[5]](#footnote-5) |  |
| varGlobal.ecran est mis à jour : « Menu ».  Déclenchement des composants et Menu.  Menu affiche les actions accessibles à l’utilisateur selon son profit. | | | |

La sélection d’un des boutons d’actions met de nouveau à jour varGlobal.ecran…

### Back End – structure à couches

Le back est un serveur NodeJS, contrôlé par la librairie Express.

Il est conçu avec une structure à couche, chacune ayant son propre rôle dans l’organisation global du Back.



**«controller»**

Le cerveau de l’application.

Son rôle est de recevoir les requêtes/demandes du Front (via des adresses url) et de les traiter avec les fonctions adéquates.

**Couche «services»**

Fonctions appelées par le controller. Il y en a donc 1 par adresse url.

J’ai regroupé les services par catégorie (login, utilisateurs, incidents) et logé chaque catégorie dans un fichier .js distinct.

**Base de données**

Pour fonctionner, le Back a besoin d’une mémoire. Celle-ci est stockée dans une base de données MySQL.

**Couche «persistance»**

Les services n’interrogent pas directement la base de données.

L’interaction entre les 2 est assuré par un ORM[[6]](#footnote-6).

C’est un système qui converti les tables de la base de données en autant de classes objets.

Chaque table est une classe d’objet. Chacun colonne est 1 attribut de l’objet.

Ce sont les fichiers Models. J’ai choisi de créer 1 fichier model par classe, donc par table.

J’ai choisi d’utiliser la librairie Sequelize de NodeJS.

L’ORM Sequelize possède ses propres fonctions pour récupérer les données de la base, les convertir en objet et vice versa : un objet peut être converti en enregistrement.

Ainsi, on évite au maximum la saisie de requêtes SQL, ce qui rend le code plus lisible. Et, quand ces requêtes sont inévitables, les fonctions de ORM empêchent les problèmes d’injections SQL.

J’ai mis ces fonctions dans des fichiers DAO[[7]](#footnote-7) distincts des Models, pour clarifier le code.

Ce sont les fonctions DAO qui sont utilisées par les services.

**En conclusion**

Le controller reçoit les demandes

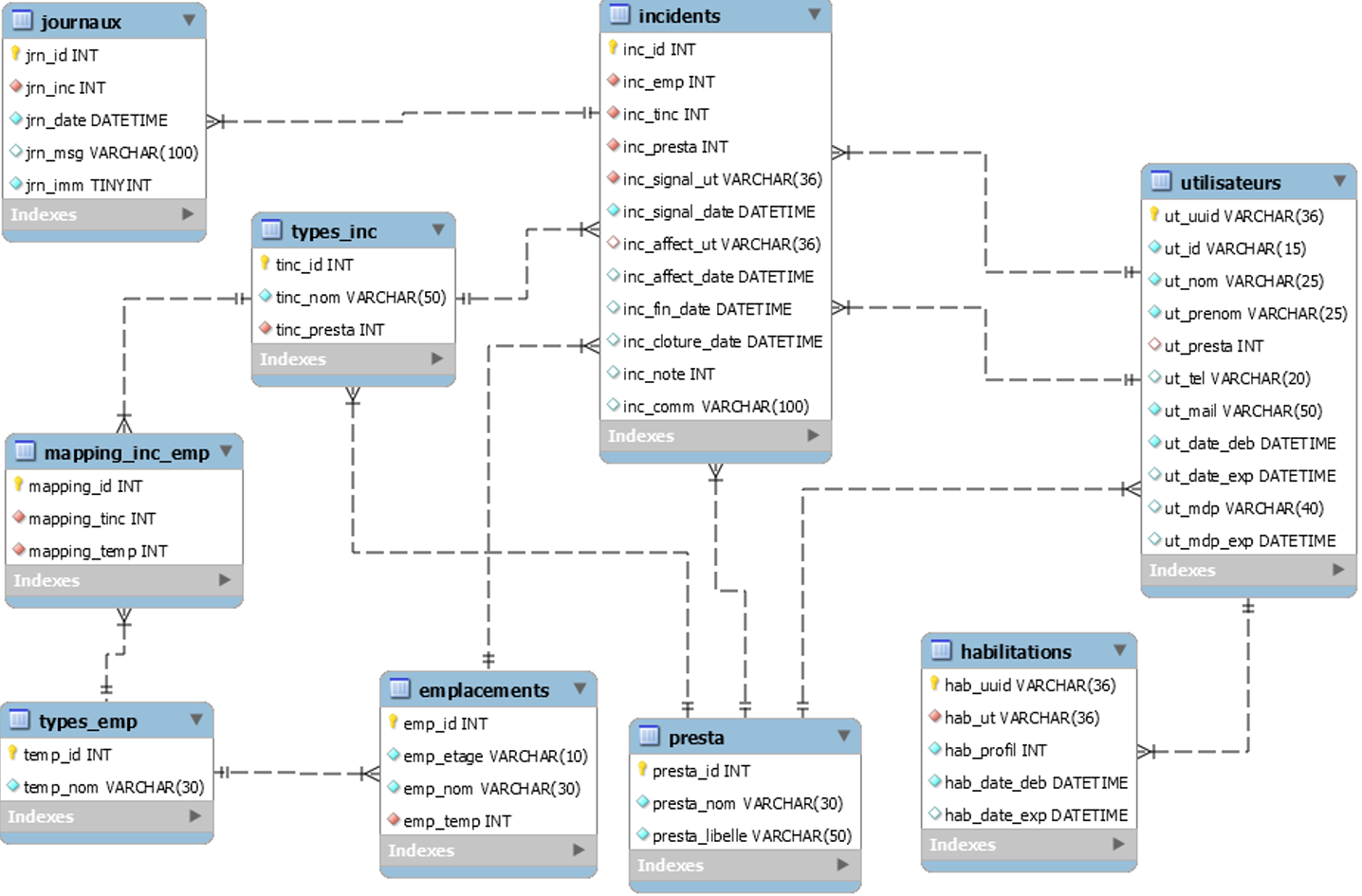
qu’il transmet aux services

qui utilisent les fonctions d’ORM (DAO)

qui utilisent les objets (models)

qui représentent les tables de la base de données.

### Base de données



Après avoir établi un premier dictionnaire de données, j’ai établi un modèle de base de données qui s’est étoffé avec le temps.

Puis j’ai créé une base de données mySQL nommée tout simplement « sos\_immo ».

Le résultat final comporte 9 tables.

Bien entendu, chaque table a sa clé primaire, pour la plupart un entier en auto-incrément et pour 2 d’entre-elles, un UUID. Je reviendrais sur cette distinction dans le chapitre sur la [sécurité](#_Identifiants_UUID).

Après avoir créé les tables, j’y ai inséré des données de test afin d’avancer plus vite dans la mise en place front et back.

Par la suite j’ai étoffé cette base en y ajoutant 4 tables « archives ».

En réalité, 3 bases de données identiques sont créées :

1. « sos\_immo » qui est la base de production finale.
2. « sos\_immo\_test » qui sert pour tester les mises à jour back et front avant leur mise en production.
3. « sos\_immo\_sauv » qui est le reflet de la base de production.

Tout cela sera détaillé dans les chapitres [Achivage et sauvegarde](#_Archivage_et_sauvegarde) et [Base de tests](#_Base_de_tests).

#### Utilisateurs - Habilitations

Tous les occupants de l’immeuble, internes ou externes, sont des utilisateurs. Dans la table Utilisateurs, la distinction se verra au niveau du champs ut\_presta, contenant l’identifiant de la table Presta (vide pour l’utilisateur interne).

Le profil d’un utilisateur peut être amener à évoluer (un technicien devient valideur, un usager devient admin…). Mais si la plupart des utilisateurs gardera le même profil de son entrée à sa sortir de la base.

Le cas échéant, il faudra créer une nouvelle habilitation (en date jour) et renseigner la date d’expiration de l’habilitation précédente (même date).

Pas de table pour les niveaux d’habilitations. La liste de ces niveaux est « en dur » dans le code du programme.

1. Utilisateur inactivé
2. Usager (externe)
3. Technicien (interne)
4. Valideur (externe)
5. Admin (interne)

Un utilisateur interne (champs ut\_presta vide) ne peut pas accéder aux profils 2 et 3.

De même, un utilisateur externe (champs ut\_presta rempli) n’a pas accès aux profils 1 et 4.

Extrait mySql : Utilisateurs et historique d’habilitations

Une image contenant table

Description générée automatiquement

Un utilisateur ayant quitté l’entreprise peut être amené à revenir. Ce sera surtout le cas pour les prestataires.

Son compte ne sera donc pas supprimé immédiatement.

Il sera dans un premier temps inactivé. Pour cela un profil « neutre » lui sera affecté et son mot de passe détruit.

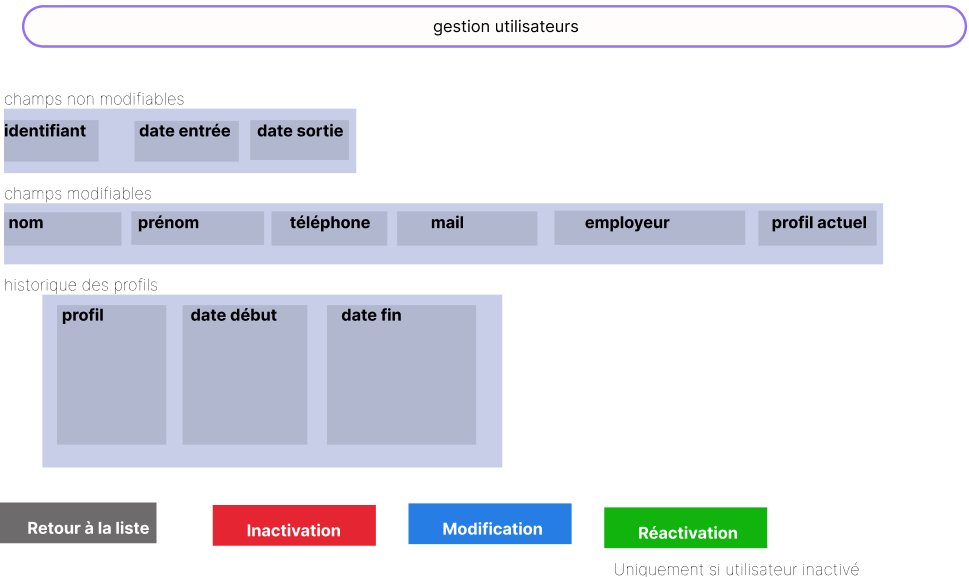
Les comptes inactivés devront être détruits au maximum 36 mois après inactivation.

À tout moment, un Admin ou valideur peut affecter n’importe quel technicien (de la spécialité) à un incident.

Un admin peut clôturer automatiquement un incident résolu à la place de l’usager.

Ainsi, l’inactivation d’un utilisateur n’a pas d’impact sur le suivi des incidents. Il est tout de même nécessaire d’avoir au moins 1 compte Admin actif.

Maquette fiche utilisateur



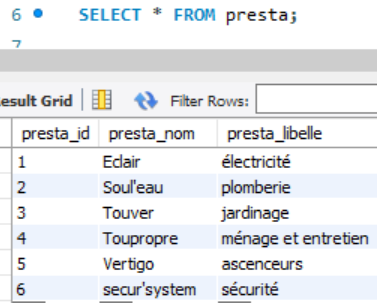
#### Prestataires - Types d’incidents

Comme on peut le constater, il n’y a pas de données d’entrée/sortie des entreprises prestataires. Si une entreprise perd le contrat, il suffira de remplacer son nom par un autre dans la table « presta ».

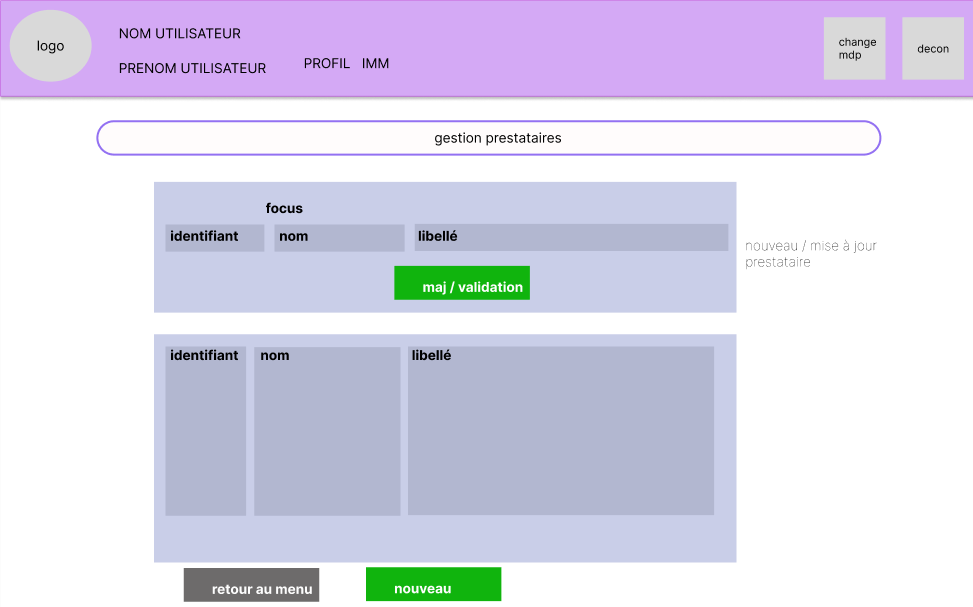
Nous sommes dans un outil de gestion d’incident. Le nom exact de l’entreprise a assez peu d’importance. Ce qui compte, c’est qu’elle soit rattachée aux bons types d’incidents et que ses techniciens/valideurs soient bien identifiés[[8]](#footnote-8).

Extrait mySql : Presta - utilisateurs

Une image contenant table

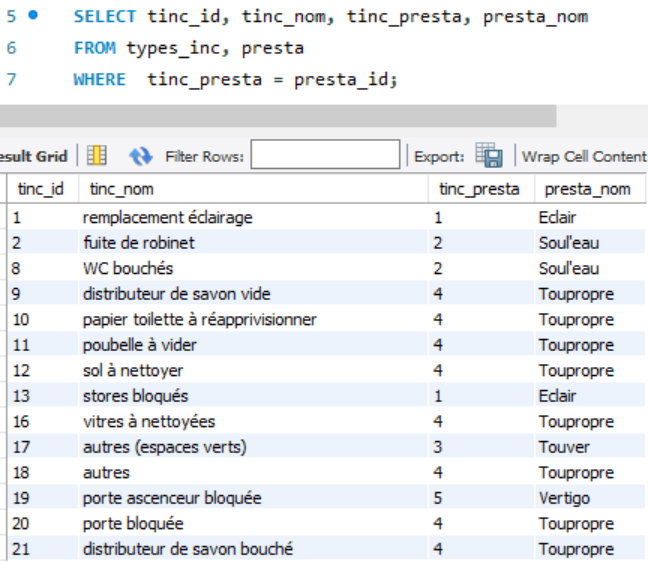
Description générée automatiquement

Maquette gestion prestataires – similaire pour gestion emplacements et types d’incidents



Chaque type d’incidents ne peut être traité que par une entreprise prestataire en particulier, ces entreprises étant spécialisées.

Extrait mySql : Presta – types d’incidents



#### Emplacements – Types d’emplacements

Chaque emplacement appartient à un type d’emplacement particulier. Il ne peut pas arriver n’importe quel type d’incident de n’importe quel type d’emplacement.

Extrait mySql : Emplacement - type d’emplacements

Une image contenant table

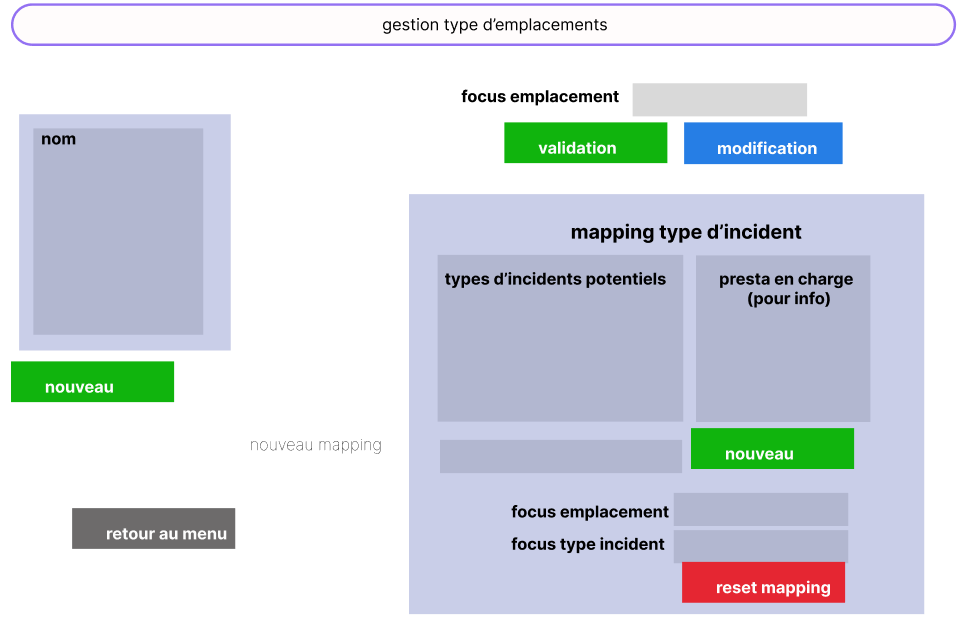
Description générée automatiquement

Extrait mySql : Mapping type d’emplacement-type d’incident

Une image contenant table

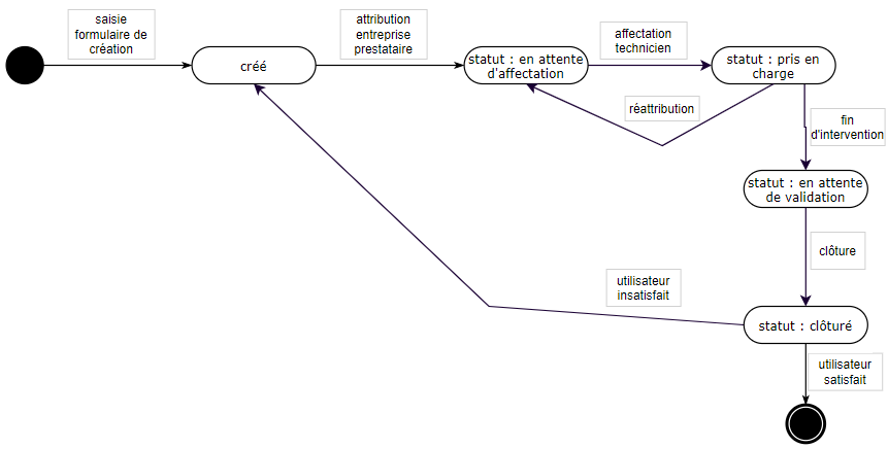
Description générée automatiquement

Maquette gestion type d’emplacements



#### Incidents

Cycle de vie d’un incident



Tous les incidents suivent le même parcourt et passent par 4 états, ou statuts, successifs.

* A sa création, 1 incident est en attente d’affection. Personne ne s’occupe encore de lui.
* Puis un technicien lui est affecté. Il est donc pris en charge.
* A la fin de l’intervention, l’incident est en attente de validation par l’utilisateur auteur du signalement.
* Après validation, l’incident est clôturé.

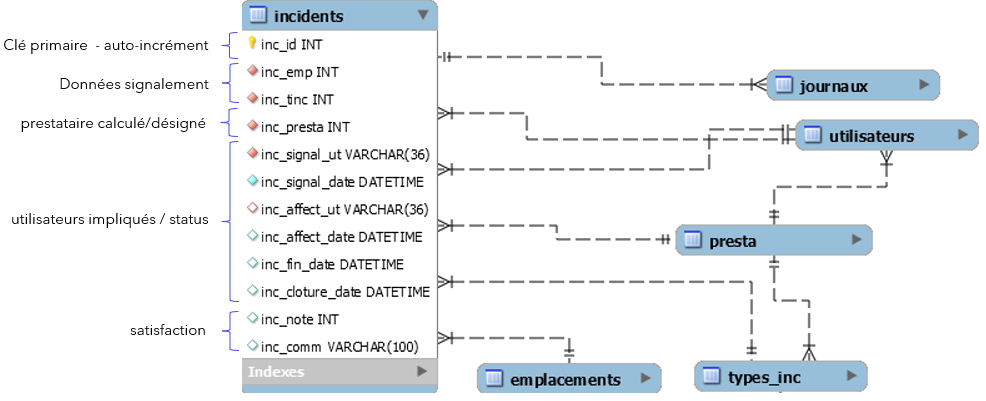
Le process s’arrête là si l’utilisateur est satisfait du résultat de l’intervention.

S’il est mécontent, il le signale et un nouvel incident est automatiquement généré avec les mêmes paramètres que le précédent, avec le motif d’insatisfaction en commentaire.

Le cycle de vie recommence avec ce nouvel incident jusqu’à résolution du problème.

L’incident est donc le cœur de l’application, la table Incidents est le cœur de la base de données.

Focus table Incidents



* Données signalement

L’utilisateur renseigne un emplacement et un type d’incident, éventuellement un commentaire[[9]](#footnote-9).

* Prestataire

L’entreprise responsable (champ inc\_presta) est dans un premier temps calculé par le back d’après le type d’incident (inc\_tinc).

S’il s’avère que le type d’incident a été mal renseigné ou évalué lors du signalement, un Admin à la possibilité de mettre ce champ à jour.

Cependant, il a été arbitré de garder la trace du signalement original, d’où la nécessité de garder les 2 champs inc\_tinc et inc\_presta, ce qui aurait pu ressembler à une redondance au premier abord.

* Utilisateurs impliqués / status

Les 4 champs « Datetime » représentent les 4 statuts du cycle de vie.

On ne conserve en base que 2 liens avec la table Utilisateurs.

* L’auteur du signalement (pour validation).
* Le technicien actuellement affecté (pour le pilotage de l’équipe).

Les noms des différents techniciens et entreprises prestataires qui se sont éventuellement succédés sur 1 même incident ne sont pas conservés.

* Satisfaction usager

Facultatif. Une note et un commentaire laissé par l’usager à la clôture de l’incident.

#### Journaux

Les utilisateurs techniques communiquent en laissant des messages et commentaires. Les usagers aussi dans une moindre mesure.

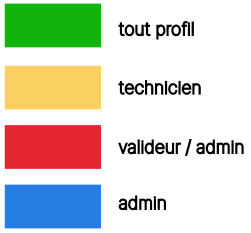
Un message ne concerne qu’un seul incident. Le choix a été arbitré de ne pas en mémoriser l’auteur dans un champs particulier mais plutôt dans le texte du message. Pour plusieurs raisons :

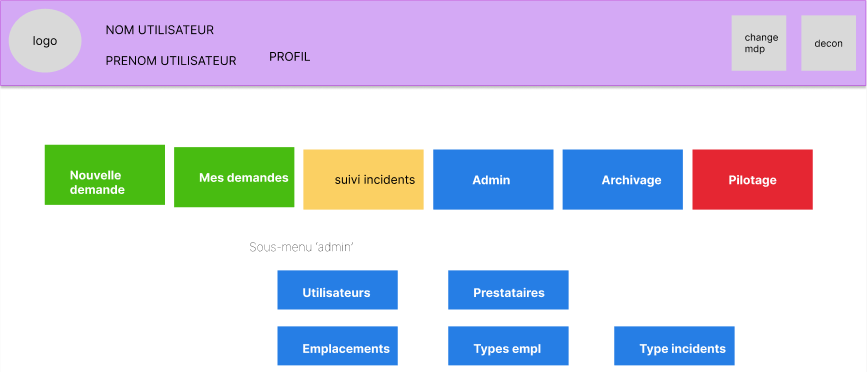
* L’auteur n’est pas forcément une information pertinente.
* Pas de statistique à faire sur cette donnée.
* Inutile de prévoir un espace dédié dans l’interface graphique.
* Gain de temps : il faudrait interroger systématiquement la table Utilisateurs pour chaque commentaire.

Le booléen jrn\_imm indique si le message est destiné aux équipes techniques (true) ou peut être rendu public (false). Par exemple, l’usager n’a pas à connaitre le nom du technicien.

# Cinématiques utilisateur et écrans principaux

Voici le menu complet des actions possibles de l’application.





Aucun profil n’a accès à l’ensemble de ces actions.

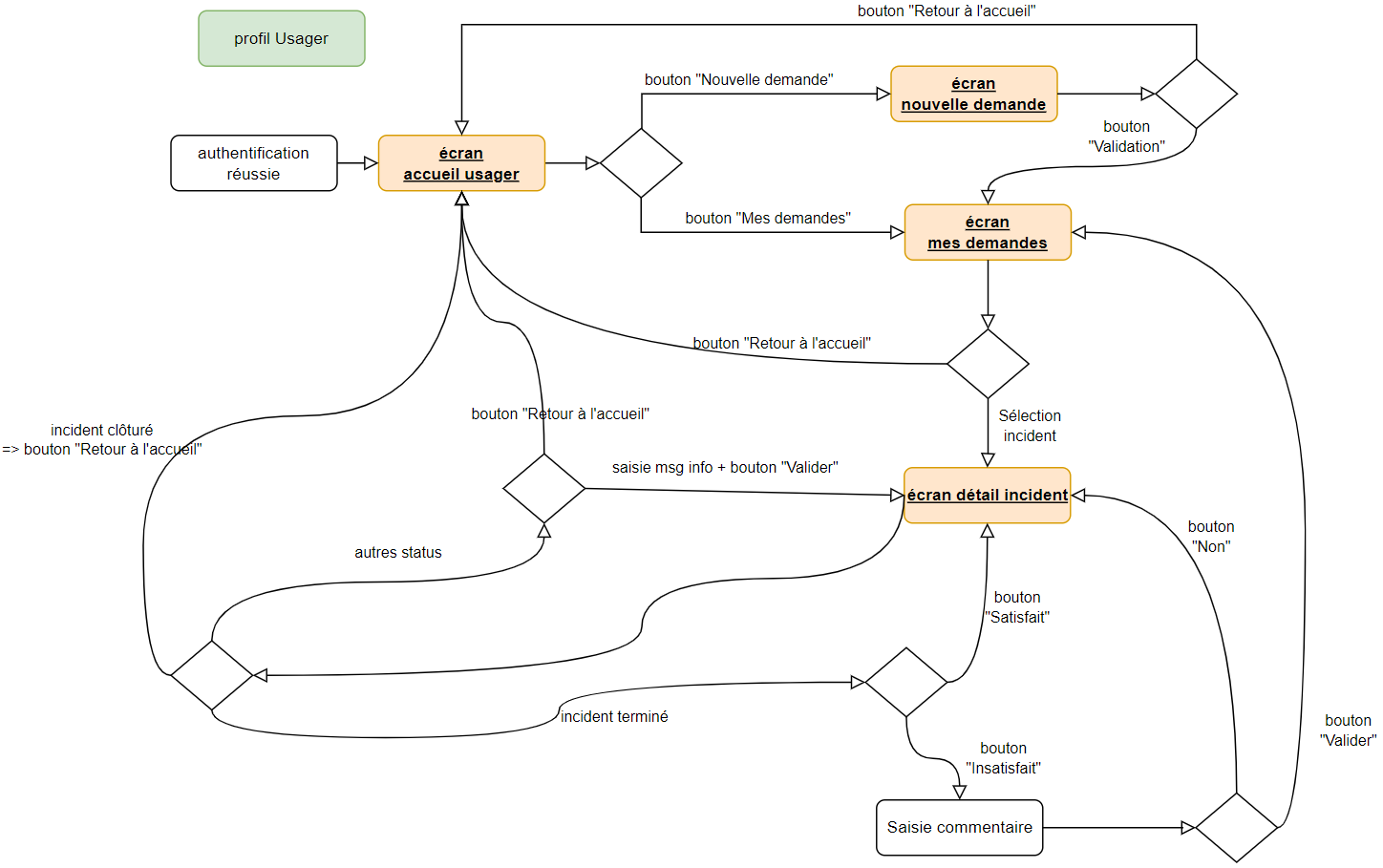
Tous les profils peuvent créer et suivre leurs propres signalements.

Le suivi d’incident est réservé aux techniciens. Il liste les interventions non terminées de l’entreprise prestataire en question.

L’écran pilotage qui contient tous les incidents listés par catégories.

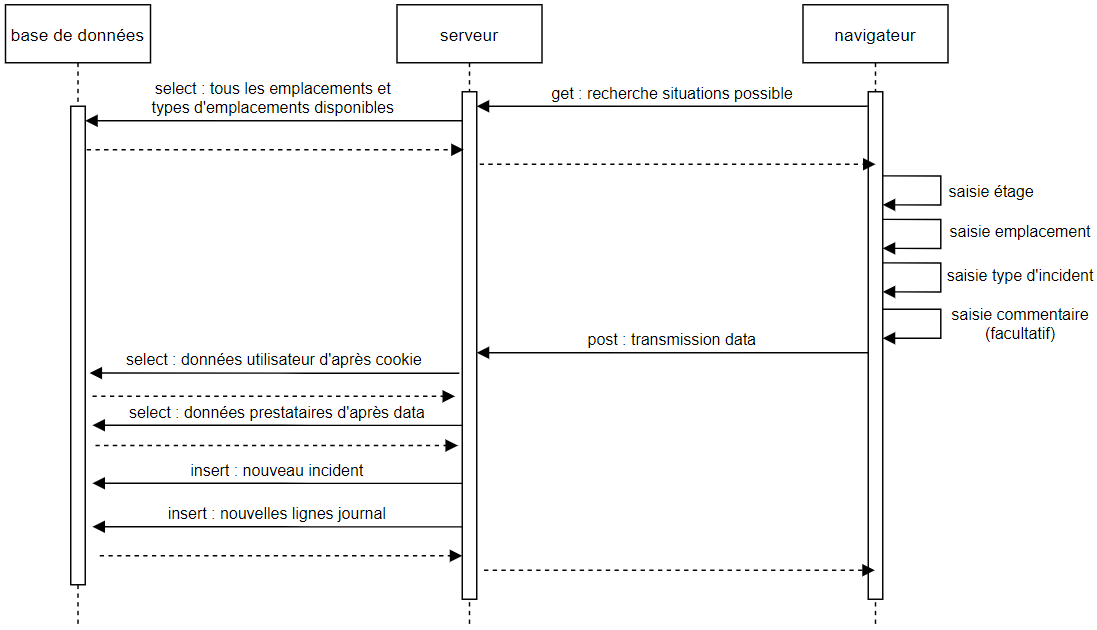
Les boutons « Admin », réservé au profil du même nom donne accès aux différents écrans de gestion de données ainsi qu’à la possibilité d’archiver les données Utilisateurs et Incidents les plus anciennes.

## Usager

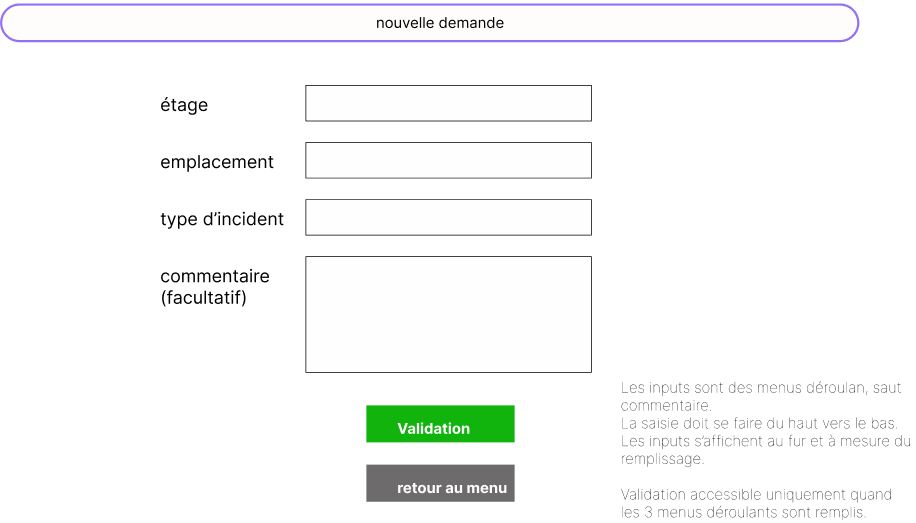


N’importe quel utilisateur peut donc saisir, suivre, commenter et valider ses propres signalements.

Diagramme de séquence : création d’un incident



Maquette nouvelle demande



Au lancement de cet écran, il y a échange front/back : le back envoie la liste de tous les emplacements possibles dans l’immeuble avec tous les types d’incidents possible dans chacun de ces emplacements.

Un premier menu déroulant s’affiche pour sélectionner l’étage.

Une fois l’étage renseigné, le deuxième menu déroulant propose les emplacements possibles à cet étage précis.

Ensuite, un dernier menu déroulant propose tous les types d’incidents pouvant survenir à l’emplacement sélectionné.

Le commentaire est facultatif.

A la validation, l’emplacement, le type d’incident et le commentaire sont envoyés au Back.

Celui-ci calcule l’entreprise prestataire attribuée à cet incident suivant le type renseigné.

Puis sauvegarde le nouvel incident en base.

L’utilisateur est redirigé vers le suivi d’avancement de ses propres signalements.

Maquette mes demandes

Une image contenant texte

Description générée automatiquement

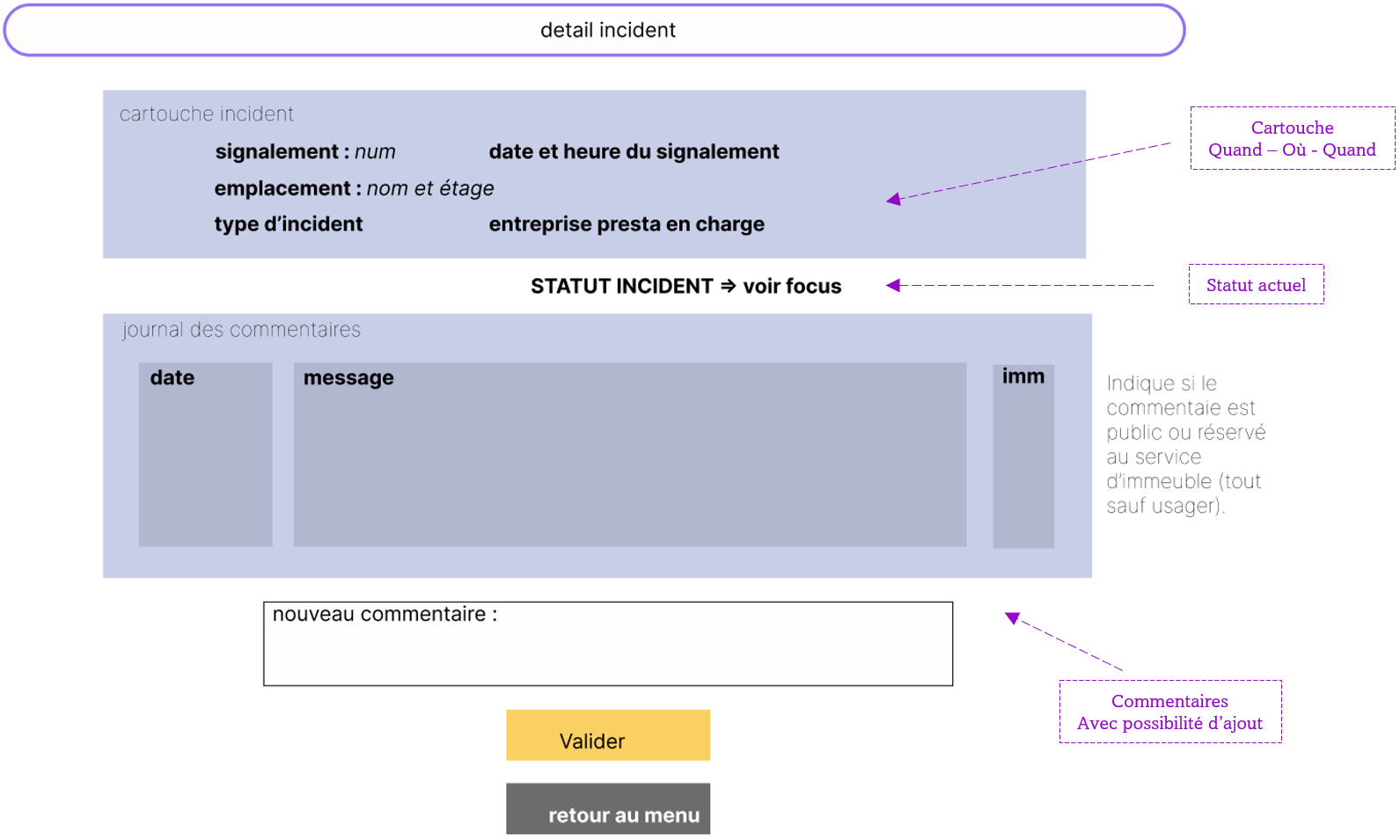
Chaque signalement est une vignette avec un petit résumé de l’incident.

La couleur indique le statut actuel de l’incident.

L’utilisateur peut sélectionner une vignette pour consulter le détail de l’incident.

Pour le visuel, on opte pour des vignettes les unes à côté des autres. Cette présentation est tout à fait adaptée à l’usager, qui fait assez peu de signalements individuellement.

Maquette fiche incident



carte d’identité. Quand, où et quoi

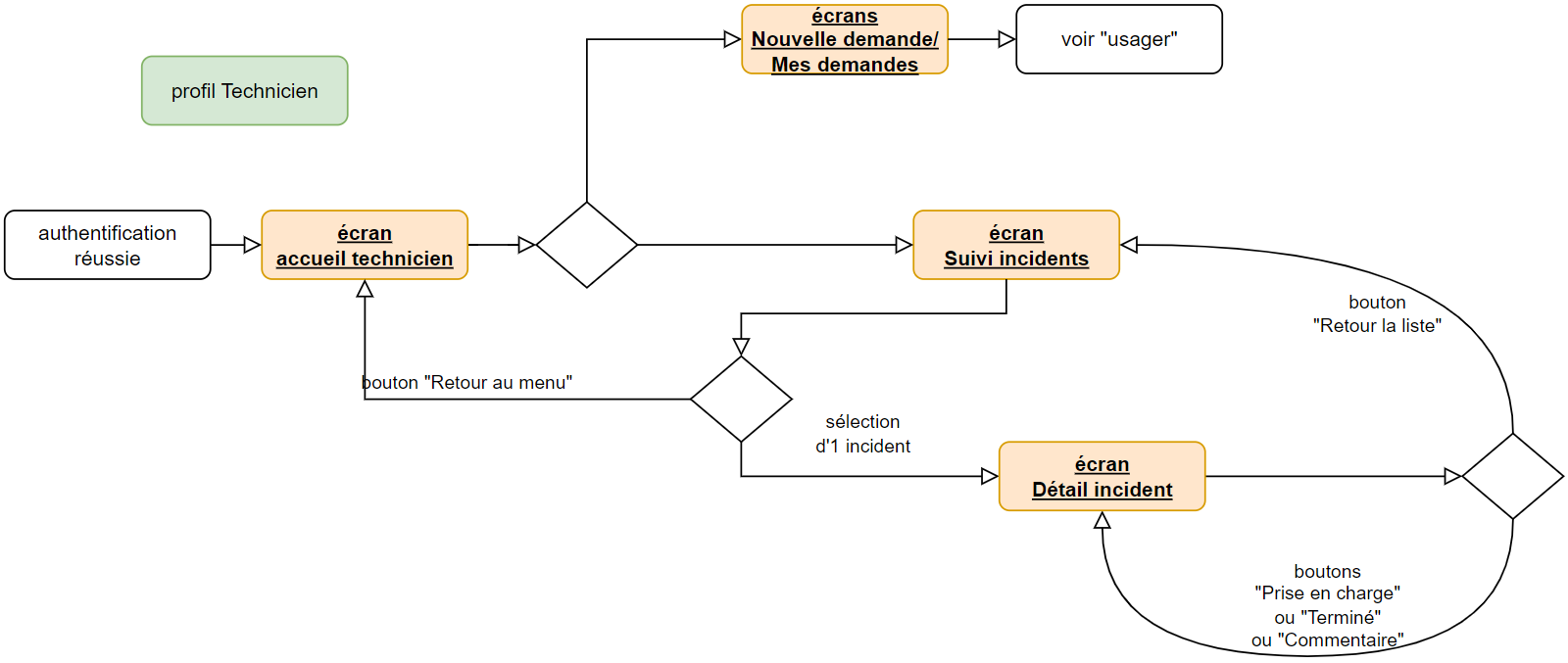
Maquettes focus statut incident

Une image contenant texte

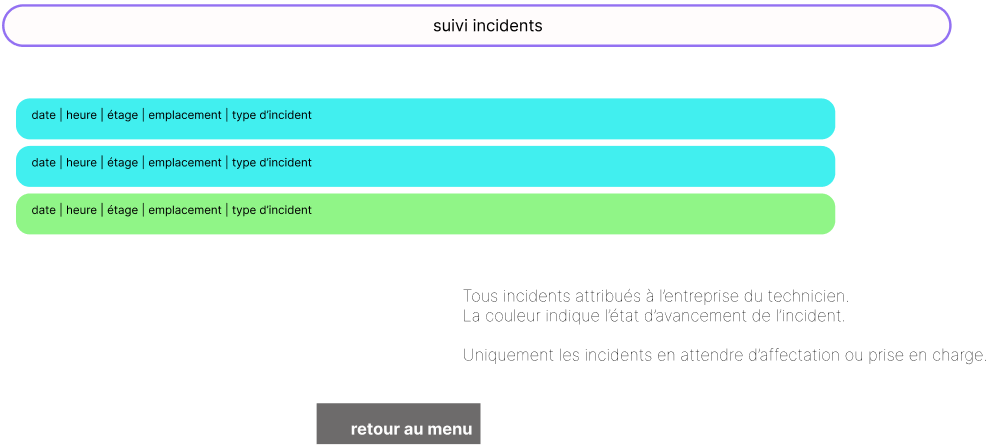
Description générée automatiquement

Tout le monde peut connaitre le statut. Mais les fonctionnalités qui le concerne ne sont accessibles que selon le profil de d’utilisateur.

## Technicien



Maquettes Suivi d’incidents



Sur le modèle de l’écran « mes demandes », des vignettes avec un résumé de l’incident et un code couleur pour le statut.

De même, à la sélection, on bascule sur la fiche incident.

Accès à tous les incidents « En attente d’affectation » de l’entreprise prestataire concernée.

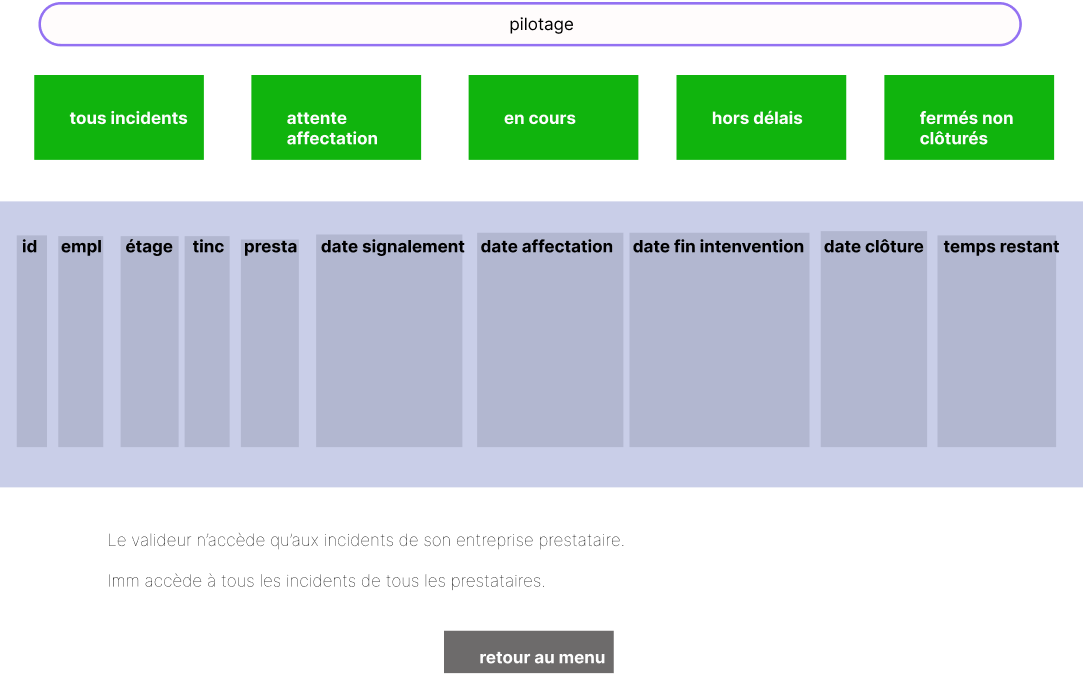
Accès aux incidents « Pris en charge » par le technicien.

Le visuel est un peu différent de « mes demandes ». Il est plus adapté à une utilisation de type « to do list ».

## Valideur



Maquettes Pilotage



Incidents classés par catégorie. Ecran disponible pour les profils Valideurs et Admin.

Les Valideurs n’ont accès qu’aux incidents attribués à leur propre entreprise.

Les incidents sont disponibles jusqu’à 30 jours après leur clôture.

## Admin

Une image contenant texte, carte, intérieur

Description générée automatiquement

Le profil Admin à tous les droits : il peut directement gérer les données de toutes les tables de la base, à l’exception d’Incidents et Journaux.

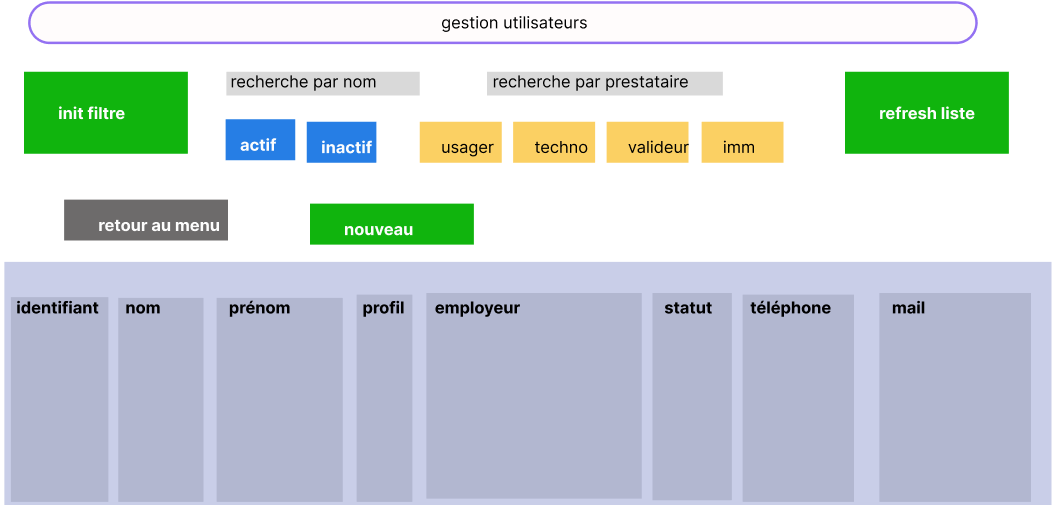
Il peut bien sûr intervenir sur les incidents et journaux, par l’intermédiaire de l’écran « Pilotage », qui lui donne accès à tous les incidents.

Focus gestion Utilisateurs

Les données Utilisateurs nécessiteront de fréquentes mises à jour.

Pour simplifier les recherches, on a installé un système de critères dans l’écran « gestion utilisateurs ».

Maquettes « Gestion utilisateurs »

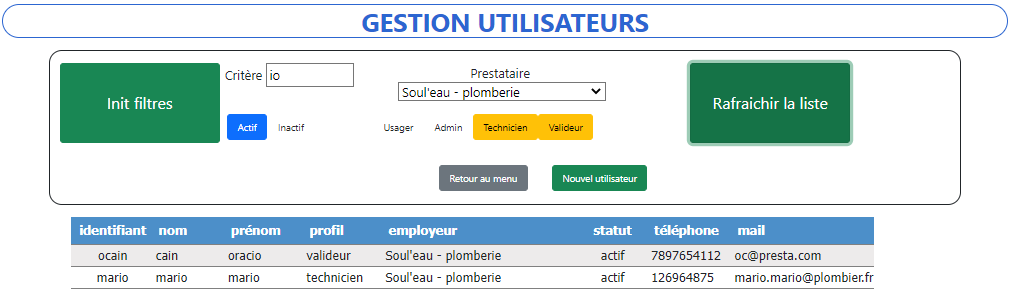


Critères :

* Statut (actif ou inactif)
* Profils
* Entreprise prestataires
* Recherche par mot-clé , présent dans les noms et/ou prénoms et/ou identifiants.

Ces critères peuvent se cumuler.

Page écran « Gestion utilisateurs »



Tous les techniciens et valideurs de l’entreprise « Soul’eau », actifs et dont le nom|prénom|identifiant contient « io ».

Pour une évolution future : une recherche par étage et type d’emplacement peut être envisagée dans l’écran « gestion de emplacements ». Il y a peu de champs mais la liste des emplacements va rapidement être très importante.

# Sécurité

## Utilisateur

### création

SOS Immo est conçu pour un usage professionnel au sein d’un immeuble de bureaux.

Ce n’est ni un réseau social ni pas un site marchand : l’utilisateur ne décide pas de lui-même de créer un compte. Il ne décide pas non plus de le supprimer.

Tous les occupants doivent pouvoir accéder à l’application, mais ce sont les services immobiliers qui gèrent les comptes.

Maquette nouvel utilisateur

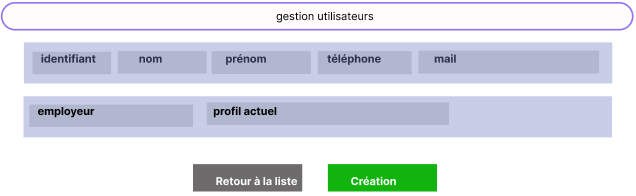
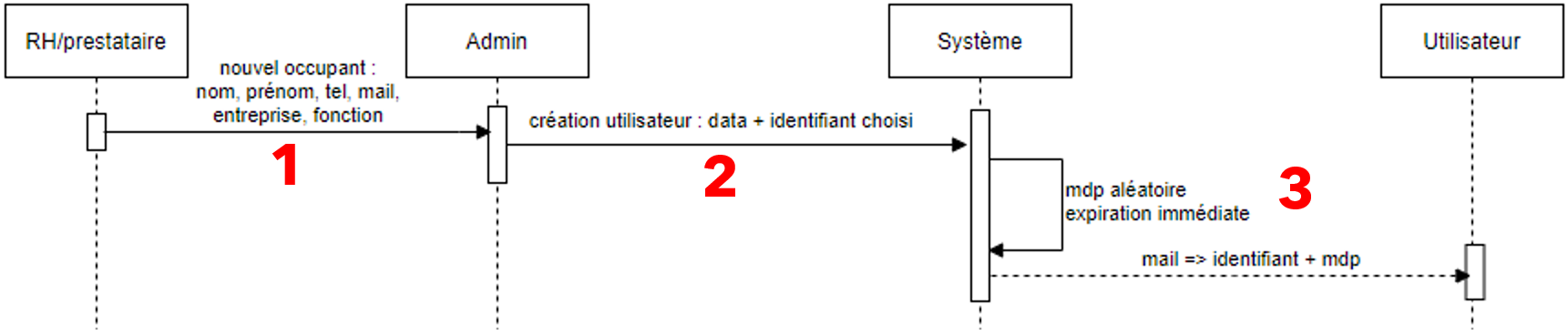


Diagramme de séquence : création d’un utilisateur



1. Un Admin a connaissance en amont d’une nouvelle arrivée. Ce process n’est pas géré dans SOS Immo.

Données obligatoire tous profils : nom, prénom, mail[[10]](#footnote-10), téléphone.

Données obligatoire profils prestataires : nom de l’entreprise, rôle hiérarchique.

1. L’Admin saisi ces données dans l’application.

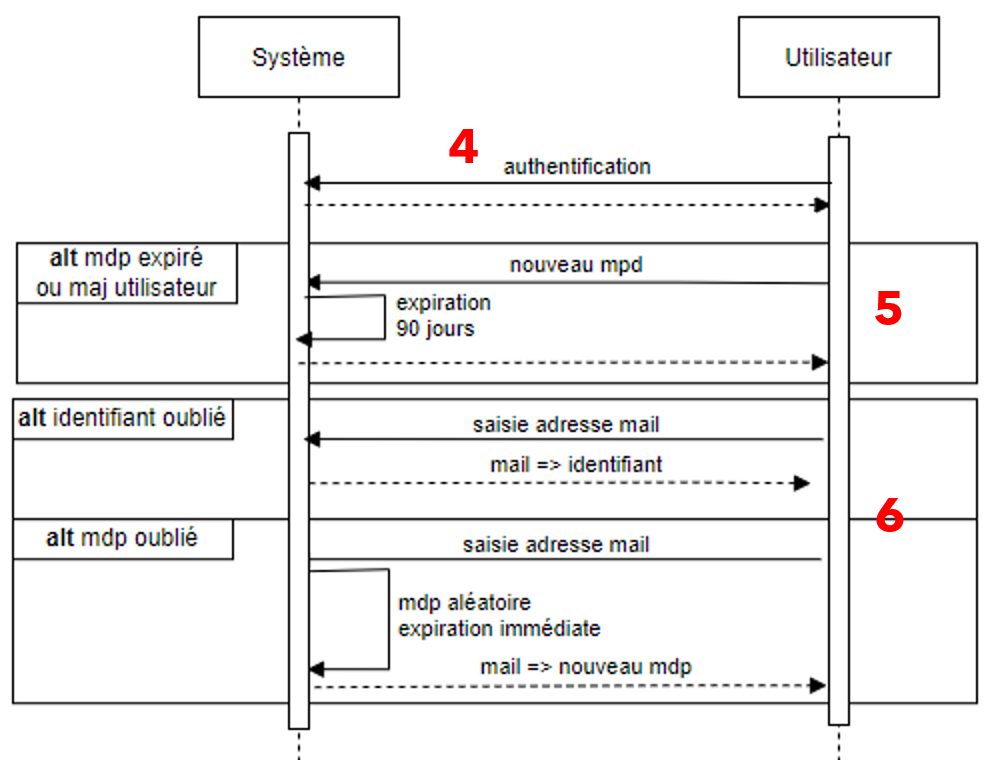
Il complète avec un identifiant unique et le profil, déterminé d’après le service (profil interne) et le rôle (profil externe).

1. Le back calcule un mot de passe aléatoire à expiration immédiate.

Il enregistre le nouvel utilisateur et sa première habilitation en base[[11]](#footnote-11) puis transmet les données d’identification (identifiant unique + mot de passe) au nouvel utilisateur par mail.

### Authentification

Diagramme de séquence : authentification



1. A présent l’utilisateur peut saisir ses données d’authentification.

Maquette authentification

Une image contenant texte

Description générée automatiquement

Le Back contrôle que l’identifiant et le mot de passes fournis appartiennent bien à un même utilisateur.

1. Après la réussite de chaque authentification, la date d’expiration du mot de passe est contrôlée.

Le cas échéant, l’utilisateur est redirigé vers l’écran de changement de mot de passe.

Ce sera le cas systématiquement à la première connexion.

Maquette modification mot de passe

Une image contenant texte

Description générée automatiquement

L’utilisateur doit saisir 3 champs :

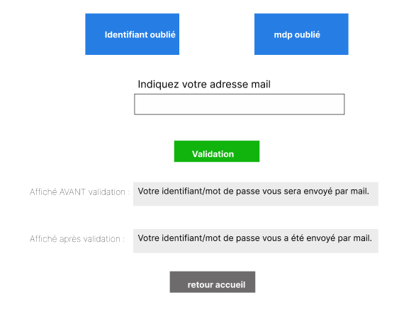
* Son mot de passe actuel.
* Le nouveau mot de passe une première fois. Il ne sera valide que s’il contient certaines caractéristiques minimum[[12]](#footnote-12).
* Une seconde fois, validé que s’il est identique à la première.

Le back reçoit les 2 mots de passe. Il enregistre le nouveau en base après avoir contrôlé que l’ancien appartient bien à l’auteur de la modification.

Ce nouveau mot de passe aura une validité de 90 jours. L’utilisateur pourra aussi décider de le changer à tout moment.

1. L’utilisateur est un être humain, il peut oublier ses données d’authentification et demander à les récupérer.

Maquette identifiant oublié



Le back contrôle que le mail fourni correspond bien à un utilisateur de la base.

Ensuite il transmet par mail à ce même utilisateur, soit son identifiant unique, soit un nouveau mot de passe aléatoire qu’il faudra changer à la première connexion.

## Gestion de mot de passe

Le mot de passe, qu’il ait été généré par l’application ou déterminé par l’utilisateur lui-même n’est pas immédiatement sauvegardé en base. Auparavant il passe par 2 étapes :

### Hachage

Principe

Le hachage est un moyen de crypter un message.

Des fonctions de hachage se trouvent dans des librairies dédiées. Pour SOS Immo, j’ai utilisé la librairie « sha1 », en JavaScript.

* En entrée : une chaine de caractères – que ce soit une simple lettre ou un document entier.
* En sortie : un nombre hexadécimal de 32 caractères. C’est l’empreinte numérique.

Avantages

* L’empreinte est particulièrement difficile à retenir et donc adaptée aux données confidentielles. De plus, la taille du champ peut être facilement calibrée dans la base de données.
* Le cryptage est à sens unique. Il n’existe pas de fonction de dé-hachage.
* Quelle que soit la librairie et quel que soit le langage utilisé, l’empreinte sera toujours la même et totalement spécifique. Deux documents (même très proche) produiront toujours les deux empreintes différentes.
* Le mot de passe est impossible à reconstituer. En cas d’oubli de l’utilisateur, le seul moyen est d’en générer un nouveau.

Limites

* Les humains sont assez prévisibles : un programme de craquage peut être utilisé pour retrouver les empreintes. De plus, des dictionnaires entiers de mots de passe vs empreintes sont disponibles plus ou moins facilement.
* Pour contrer ce défaut, des caractéristiques minimums pour les mots de passe sont exigées. Malgré tout, nous avons souvent tendance à utiliser les mêmes types de séquence (1 majuscule suivie de minuscules, le tout formant un mot, en terminant par les chiffres). Un programme de craquage sera tout de même beaucoup plus lent.

### Salage

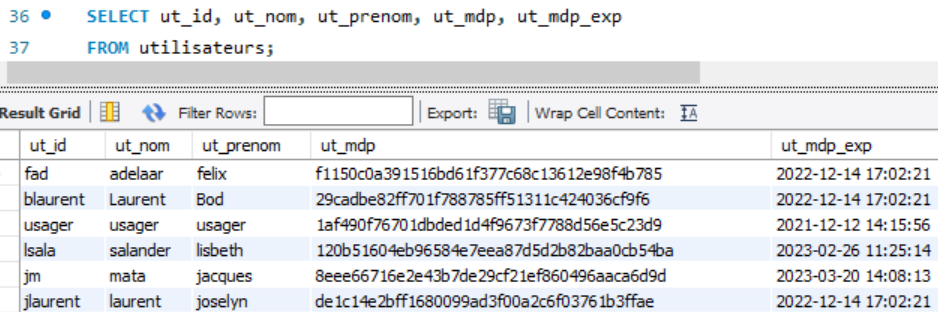
Pour ralentir le travail des hackers, on procède également au salage du mot de passe.

C’est un procédé purement mécanique : on ajoute à la chaine à crypter une autre chaine (de préférence unique) avant le hachage.

J’ai choisi de concaténer l’identifiant unique avec le mot de passe avant de hacher le tout.

* Un programme de craquage devra connaître l’identifiant de l’utilisateur pour pouvoir travailler.
* Plusieurs utilisateurs peuvent utiliser un même mot passe, les empreintes sauvegardées dans la base seront forcément différentes.

Extrait mySql : Utilisateurs – mots de passes

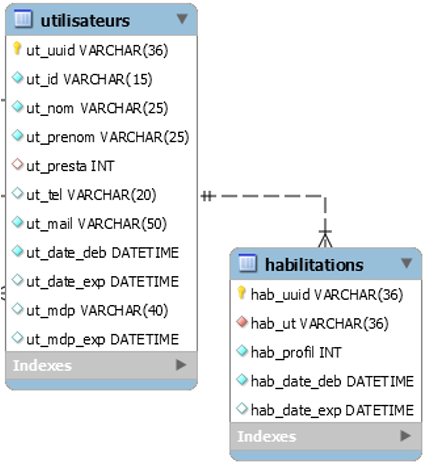


## Identifiants UUID

Il n’y a pas de données vraiment confidentielles dans la base de données, comme les coordonnées des prestataires par exemple. C’est pourquoi pour la plupart des tables ont comme clé primaire un entier en auto-incrément, très simple à mettre en place.

Les quelques données un peu sensibles se trouvent dans les tables Utilisateurs (noms, coordonnées professionnelles, mot de passes) et Habilitions (profil). J’ai donc choisi pour elles un UUID[[13]](#footnote-13) comme clé primaire.

Extrait de la base de données



Tout comme l’empreinte de hachage, l’UUID est un nombre hexadécimal de 32 caractères.

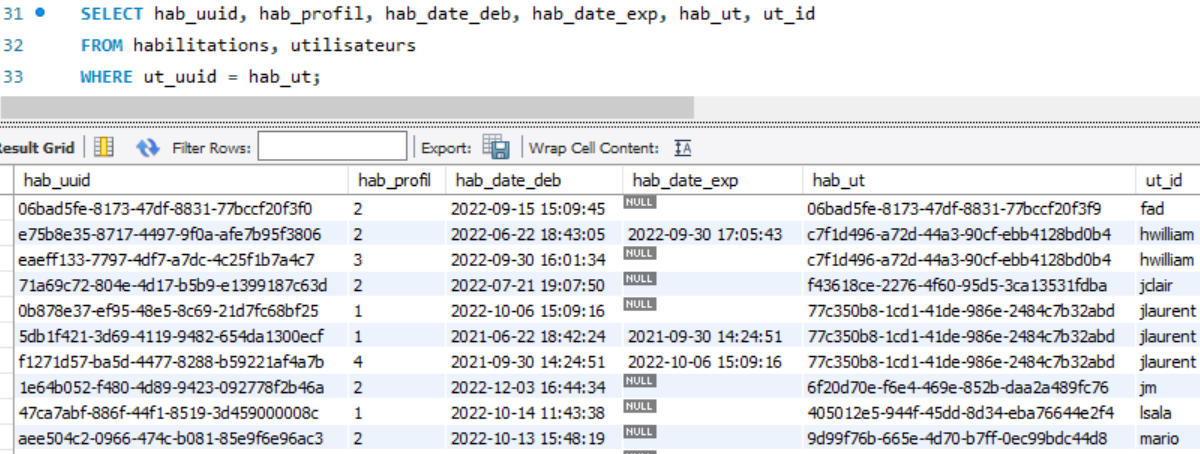
A la différence de l’empreinte, il n’est pas calculé d’après une chaîne mais généré aléatoirement par une fonction ad oc. Ici j’ai choisi la librairie « uuidv4 », en JavaScript.

La complexité de l’UUID lui garantit, quasiment, d’être unique.

L’intérêt d’utiliser un UUID comme clé primaire est encore une fois de ralentir les hackers.

Si un hacker découvre l’url nécessitant un identifiant utilisateur, il aura moins de chance d’arriver à ses fins avec « 06bad5fe-8173-47df-8831-77bccf20f3f9 » ou « 94de4d95-e933-46aa-b94a-99366e47b1a2 » qu’avec « 1 », « 2 », « 789 » ou « 42 ».

Extrait mySql : habilitations et Utilisateurs



## Cookie

### Besoin

Sécuriser les données, c’est bien, mais il faut aussi sécuriser leur accès.

Un premier filtre est fait par le biais de l’application front, puisque que c’est le profil de l’utilisateur qui lui donne accès ou non aux différentes fonctionnalités.

Mais ce n’est pas suffisant car on peut facilement envoyer des requêtes au back sans passer par l’appli front.

Il faut donc qu’à chaque fois qu’un service est sollicité par le serveur, il contrôle : l’auteur de la requête a-t-il le droit d’envoyer cette requête ?

### Principe

Pour cela, j’ai choisi d’utiliser un cookie de session.

Ce cookie est un petit fichier créé par le serveur et stocké dans le navigateur.

Ses création, destruction, utilisation doivent être définies dans le serveur. Il peut contenir toutes les informations que le développeur juge nécessaires.

Ici, le serveur crée un cookie de session après la réussite de l’authentification.

Y sont stockés les infos sur l’utilisateur qui seront nécessaires aux futurs contrôles :

* Un booléen[[14]](#footnote-14) qui signale réussite de l’authentification.
* UUID
* Identifiant unique[[15]](#footnote-15)
* Profil
* Prestaire employeur[[16]](#footnote-16) (le cas échéant, null par défaut)

Quand un service est appelé par le serveur, il peut tester le cookie à tout moment pour contrôler la validité de la demande.

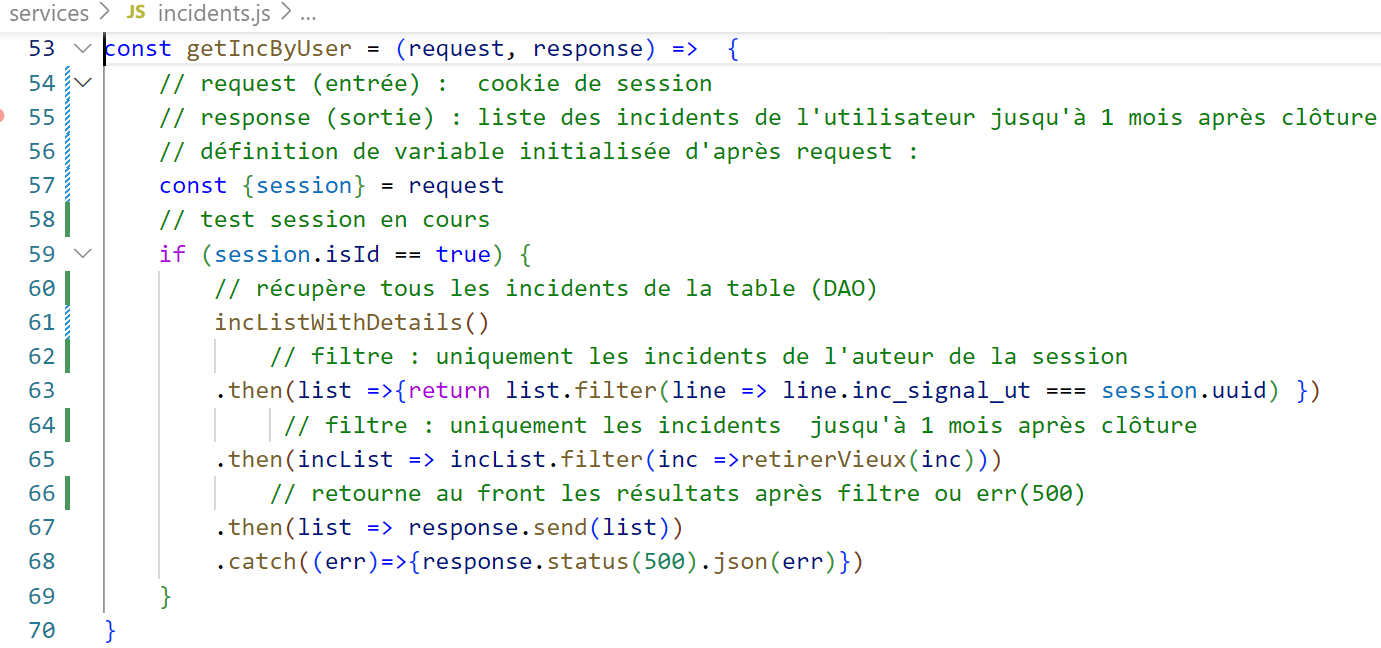
* Systématiquement au début.
* En cours de process si nécessaire.

Un cookie à une durée de vie limitée qu’il faut paramétrer avant sa création.

Pour SOS Immo, j’ai décidé qu’il s’auto-détruirait 10 minutes après la dernière connexion à la base.

### Exemples

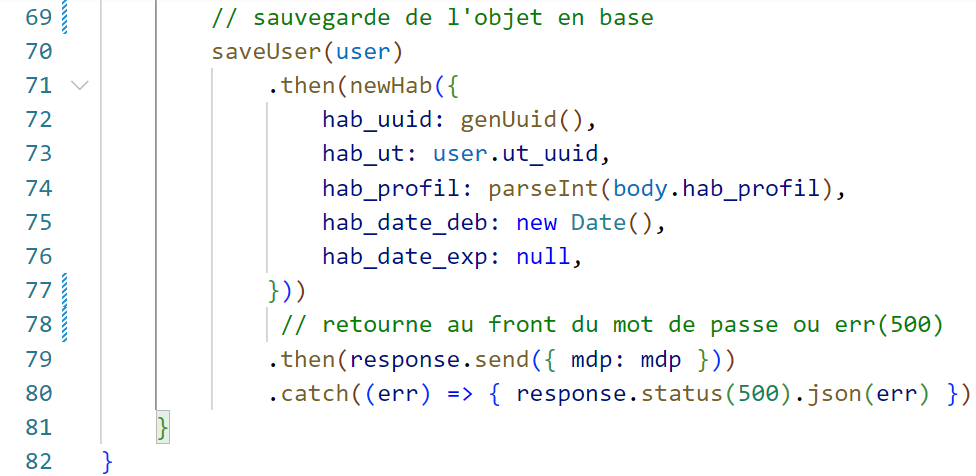
Extrait code : service « getIncByUser »



Cookie testé au début et au cours de process.

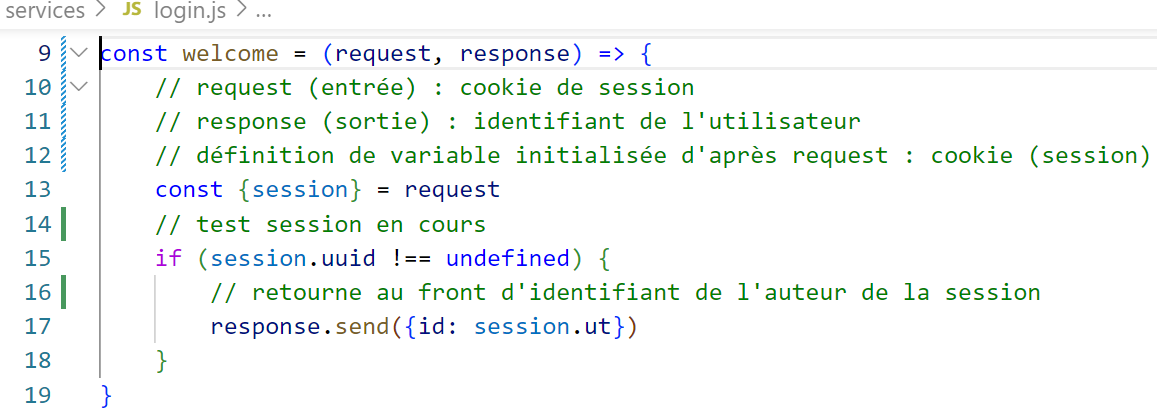
Extrait code : service « creaOneUser »





Cookie testé uniquement au début du process.

Extrait code : service « welcome » - appelé par l’écran d’authentification[[17]](#footnote-17)



Cookie testé au début puis retourné partiellement en fin de process.

## Variables d’environnement

Pour se connecter à une base de données, le back end a besoin de plusieurs informations : type de base (ici mySql), localisation (IP, ici localhost), nom, utilisateur et mot de passe de la base.

Il est très dangereux d’écrire ces informations, surtout les 3 dernières, directement dans le code.

C’est pourquoi on utilise des variables d’environnement.

Une image contenant texte

Description générée automatiquement

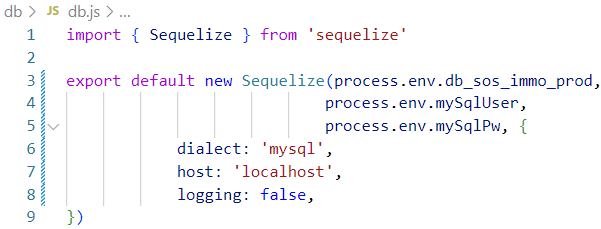
Elles sont enregistrées directement dans les paramètres de la machine. Un certain est défini automatiquement, mais on peut en ajouter manuellement si besoin.

Pour faire fonctionner le back end, les variables suivantes devront être définies au préalable :

* db\_sos\_immo\_prod
* mySQLUser
* mysSQLPw

En javaScript, l’appel à une variable d’environnement se faire par le mot-clé « process.env.nomVariable ».

Extrait code : définition de la base de données rattachées à l’ORM Sequelize



## Archivage et sauvegarde

### Archivage

Les tables Incidents et Utilisateurs seront rapidement très lourdes.

* Les incidents clôturés n’ont plus vraiment d’utilité.
* Les comptes des utilisateurs sortis des effectifs (inactifs) ne sont pas supprimés immédiatement, au cas où ils seraient amenés à revenir bientôt (surtout dans le cas des prestataires). Mais au bout d’un moment, on peut considérer qu’ils n’ont plus d’utilité non plus.

Il faut pouvoir archiver les incidents et utilisateurs et par conséquent les journaux et habilitations rattachés. C’est-à-dire, mettre de côté, sans pour autant supprimer, les incidents clôturés et les utilisateurs inactifs depuis un certain temps.

* Chaque fois qu’un enregistrement d’1 de ces 4 tables est supprimé, il est recopié à l’identique dans une table « \_arc ».

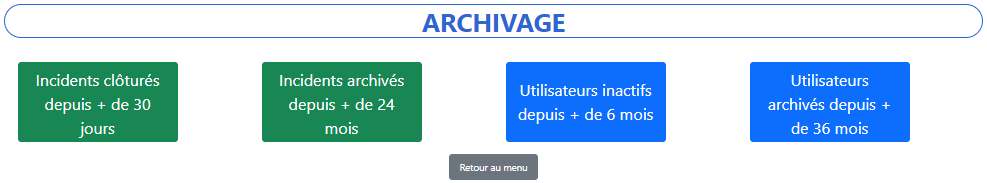
Les tables « \_arc » ont donc les mêmes champs que les tables d’origine plus 1 champs « \_arc » indiquant à quelle date a eu lieu l’archivage.

Ainsi, on dispose toujours des incidents (avec leurs journaux) et des utilisateurs (avec leur historique d’habilitations) en cas de besoin.

Bien sûr les tables \_arc auront finalement besoin d’être purgées à leur tour. Cette fois, on supprimera définitivement les données.

L’archivage et les purges seront gérés par le profil Admin.

Page écran « Archivage »



Les 5 autres tables[[18]](#footnote-18) n’ont pas d’archivage. Il n’est pas pour l’instant prévu de supprimer ces données et un archivage serait de toute façon sans intérêts.

### Base de sauvegarde

En cas d’accident, il est toujours bon d’avoir une sauvegarde de la base de données.

Pour cela, j’ai créé une base « sos\_immo\_sauv » qui est le reflet de la base de prod « sos\_immo ».

Elle se trouve dans mon ordinateur, comme la base de production. Bien entendu, dans une véritable application, les 2 bases seront logées dans des machines différentes.

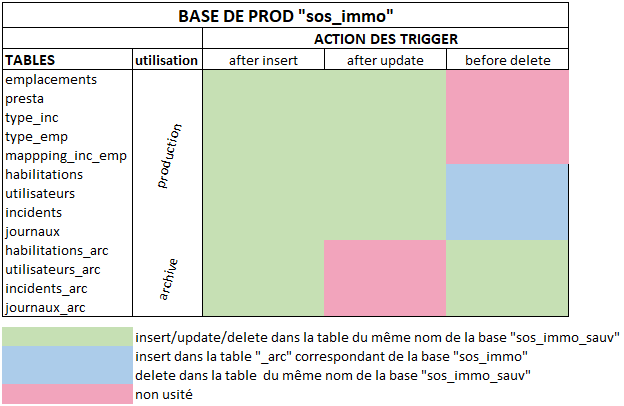
Les données sont recopiées, mises à jour et même supprimées dans la base de sauvegarde au fur et à mesure qu’elles le sont dans la base de prod grâce à l’utilisateur de triggers.

### Triggers

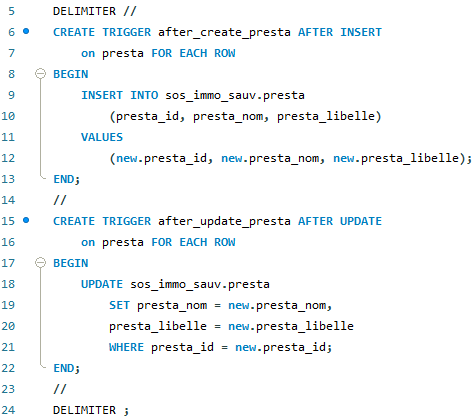
Les triggers/déclencheurs sont des scripts SQL rattachés à une base de données et programmés pour faire des actions lorsque survient un évènement précis (insert, update, delete).

Si la base venait à disparaitre, ils seraient détruits en même temps.

La gestion de l’archivage et de la sauvegarde est gérée par des triggers rattachés à la base de prod « sos\_immo ».



Script trigger : insert et update de la table Presta - similaires aux autres tables



Ici, 1 trigger pour chaque évènement Insert et Update.

Pas de trigger pour Delete : il n’est pas prévu de supprimer ces données[[19]](#footnote-19).

Script trigger : delete de la table Utilisateurs- similaire pour les 4 tables avec archivage

Une image contenant texte

Description générée automatiquement

Dans un premier temps, un insert est joué dans la table \_arc.

Ensuite, un autre insert est joué dans la même table de la base de sauvegarde.

Enfin, l’enregistrement est supprimé de la table « utilisateurs » de la base de sauvegarde.

Pas de trigger « update » pour les tables \_arc : on ne modifie pas une donnée archivée.

Script trigger : delete de la table Utilisateurs\_arc – similaire pour les 4 tables \_arc

Une image contenant texte

Description générée automatiquement

Une donnée supprimée d’une table d’archive l’est définitivement.

# Tests

Dans le métier de développeur, les tests unitaires sont réalisés au fur et à mesure que les fonctions sont mises en place. Dans un monde parfait, ils sont conçus avant le codage des fonctions.

Ici, pour mon premier projet, je me suis retrouvée confrontée à un problème :

* Côté front, comment tester les composants React ?
* Côté back, comment définir des tests unitaires alors que toutes les fonctions sont des services qui interagissent avec la base de données ?

La réflexion a été intense pour trouver arriver et les solutions ont souvent été conçues après la plupart du développement. J’ai tout de même réussi à tester le front et le back.

## Tests unitaires

La librairie Jest, en JavaScript, m’a permis de tester les fonctions front utilisées par les composants pour :

* Déterminer les « class » css du code html d’après certains paramètres.
* La mise en forme des dates.
* Les manipulations un peu fastidieuses et répétitives (ex. nom du profil d’après le numéro, retour Null si un champ est vide, etc.)

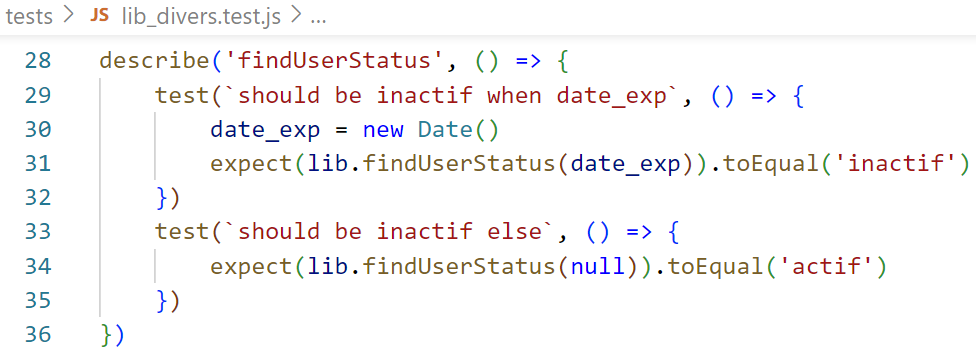
Exemple : la fonction « findUserStatus »

* La fonction retourne un string d’après la date entrée en argument.

Une image contenant texte

Description générée automatiquement

* Le test unitaire consiste à contrôler que le string de sortie est bien le bon, suivant que l’argument d’entrée est Null ou pas.



Utilisation d’un Pipeline GitHub

Un pipeline est une série d’instructions lancée automatiquement à la suite d’un élément déclencheur.

L’outil GitHub possède un système de pipeline grâce auquel les tests unitaires sont lancés à chaque push.

Les instructions doivent être saisies en Linux dans un fichier .yml.

Il suffit d’aller sur GitHub après chaque push pour savoir si les tests ont réussis ou échoués.

GitHub m’envoie un mail en cas d’échec.

Extrait code : pipeline Github .yml

Une image contenant texte

Description générée automatiquement

Dans un docker, le pipeline récupère les fichiers du repo GitHub, installe npm et lance les tests unitaires.

[Annexe 1 : rendu d pipeline GitHub](#_Annexe_1_–)

## Base de tests

Pour éviter les données fantômes et la corruption de données réelles, une base « sos\_immo\_test » est mise en place, identique à la base de prod « sos\_immo ».

Pour tester les évolutions back et front, il suffit de changer le nom de la base cible dans le code.

Plusieurs possibilités :

* Modifier la variable d’environnement « db\_sos\_immo\_prod » dans les paramètres de la machine
* Remplacer dans le code la variable d’environnement par le nom de la base de test.
* Définir une nouvelle variable d’environnement ad hoc[[20]](#footnote-20) et changer le code en conséquence le temps des tests.

J’ai choisi la dernière solution.

## Tests d’intégration et de non-régression[[21]](#footnote-21)

Côté back, c’est le déroulé des services qui doit être contrôlé.

Ces tests doivent être effectués dans la base test, après que celle-ci a été réinitialisée.

En effet, ils vont consister entres autres à insérer et mettre à jour de nouvelles données.

Un scénario est déterminé sur papier, avec des instructions à suivre à la lettre sur l’interface front.

[Annexe 2 - exemple de scénario complexe](#_Annexe_2_–)

Ce travail est fastidieux et l’interface front a des limites. J’ai utilisé l’outil PostMan qui permet de tester directement les url et d’établir de véritables scénarios qui enchaînent les services. Ces enchaînements sont nommés Collections.

[Annexe 3 - exemples de collections Postman](#_Annexe_3_-)

# Organisation

Un certain nombre d’outils ont été utilisés pour l’élaboration de ce projet.

Il est inutile de tous les énumérer, voici les principaux logos.



Etapes préparatoires :

* Maquettes des écrans principaux (PowerPoint et papier) pour validation du sujet par le professeur.
* Interview des services immobiliers de l’immeuble Basalte (Société Générale) à la Défense.
  + Vision usager
  + Vision admin
  + Utilisation par les prestataires
* Liste des fonctionnalités minimums pour les différents profils (Notes).
* Dictionnaire de données, regroupées par catégorie (Excel).
* Priorisation des fonctionnalités (Notes).
* Maquettes des écrans principaux (PowerPoint et papier).
* Choix du langage pour le code.
  + Dans un premier temps, je souhaitais utiliser JavaScript pour le front, car j’étais à l’aise avec React, et Python pour le back.
  + Après quelques essais avec mes connaissances du moment, j’ai finalement décidé de faire la totalité du code en JavaScript.

Figma m’a permis de transposer les maquettes déjà faites et de créer les autres plus facilement qu’avec PowerPoint.

GitHub – outil de versionning

Conçu pour permettre à plusieurs développeurs de travailler sur les mêmes fichiers de code. Les modifications sont fusionnées (mergés) pour former le code définitif.

GitHub m’a servi de 2 façons :

* Comme sauvegarde du code et des fichiers annexes (présentations, diagrammes, etc.)
* Lancement automatique des tests unitaires via un système de pipeline.

J’ai créé 2 dépôts distincts : back et front.

# Evolutions

## Fonctionnelles

Certaines fonctionnalités pourraient être poussée ou étudiées :

Relance d’un incident non satisfaisant au lieu de la création d’un nouvel incident

C’est une demande tout à fait légitime qui permettrait un meilleur suivi des incidents.

Après avoir étudié la question, j’en ai conclu qu’il faudrait scinder la table « incidents » en 2 tables « incidents » et « interventions », 1 incident pouvant contenir plusieurs interventions.

Réintégration d’un utilisateur inactif

Récupération des données archivées

Arrivées groupées

Il peut être fastidieux de saisir chaque utilisateur à la main. Surtout en période de fort turn-over.

On pourrait imaginer un tableau formalisé reçu en amont.

L’identifiant unique pourrait être calculé ou saisi directement dans le tableau.

Celui-ci serait alors chargé par l’application qui créerait les nouveaux utilisateurs et leur enverrait leur identifiant par mail.

Gestion de plusieurs immeubles

Il suffirait d’ajouter un champs « immeuble » dans la table « emplacements ». A ce stade, une table d’immeubles ne me semble pas pertinente.

Lors d’un signalement, il y aurait 1 liste déroulante supplémentaire en début de formulaire.

Enquête de satisfaction

2 champs sont prévus dans la table « incidents » pour que l’usager puisse entrer une note et un commentaire.

Ces données pourraient être récupérées par les Admin et les Valideurs.

KPI

Pour l’instant l’écran pilotage produit des listes.

Par la suite, il pourrait également produire des graphes et des statistiques[[22]](#footnote-22).

Application mobile

La seule réelle utilité serait pour les techniciens : pour qu’ils n’aient pas besoin de retourner à leur base pour notifier une fin d’intervention et

Les seuls écrans activables seraient : login, mot de passe oublié, suivi d’incident, fiche détail incidents.

J’aurais besoin d’en connaître plus sur les librairies à utiliser.

## Techniques

Pour être au plus proche des conditions d’utilisation réelles, SOS Immo devra être déployé sur des machines distantes, back et front.

* La machine back (ou serveur web) devra contenir :

La dernière version de l’application back.

La définition des variables d’environnement nécessaires au fonctionnement de l’application.

La base de données de production.

Un conteneur lancé en permanence qui contiendra :

Une copie de l’application

Un lien (volume) vers la base de données. Si le conteneur est détruit, la base de données reste intacte.

* Chaque utilisateur possèdera un lien vers la machine front.

Dans cette machine on trouvera la dernière version de l’application front.

Le lien va déclencher le lancement d’un conteneur.

Celui contiendra une copie de l’application.

Les requêtes seront envoyées vers le conteneur de la machine back.

Ainsi il ne sera pas nécessaire d’installer l’application front sur les ordinateurs de chaque utilisateur.

Pour un déploiement automatique, on utilisera un pipeline GitHub qui s’exécutera à chaque push.

* Exportation de la nouvelle version du programme sur la machine virtuelle dédiée, back ou front.
* Reconstitution de l’image docker si nécessaire.
* Destruction des dockers en cours et lancement d’un nouveau sur la machine back.

[Annexe 4 - schéma déploiement](#_Annexe_4_-)

# Bilan

## Ce que je savais déjà

Il y a maintenant un peu plus de 30 ans, j’ai entamé des études pour devenir ce qu’on appelait alors analyste programmeur. La vie a fait bifurquée mon parcourt professionnel. Je suis maintenant fière de revenir à ce métier, qu’on appelle désormais développeuse[[23]](#footnote-23).

En fait je n’ai jamais vraiment arrêté de programmer, que ce soit pour mon plaisir ou dans un cadre professionnel. Le plus souvent c’était des macro Excel et Access, certaines très complexes.

Aussi, j’avais déjà l’habitude

* D’organiser des projets.
  + Interroger les futurs utilisateurs pour recueillir leurs besoins
  + Etablir un dictionnaire de données et en faire une base.
  + Déterminer l’ordre des priorisations.
* Et bien sûr de les développer.
  + Algorithmique.
  + Choix de la techno.
* De faire les tests manuellement. Ou de les faire faire par les futurs utilisateurs.

A chaque fois j’étais déjà seule au développement. La grosse différence avec ce projet, c’est que la plupart du temps, il y avait un responsable qui exprimait ses besoins et validait les fonctionnalités, les priorités, et, finalement, le projet.

Ici j’ai assuré tous les rôles : j’ai porté, conçu, codé, testé, validé ce projet seul[[24]](#footnote-24).

## Découvertes de la formation mise en pratique pour le projet

La formation m’a permis de découvrir beaucoup de nouveaux concepts. Certains m’ont servi spécifiquement dans ce projet.

* Utilisation des diagrammes ulm.
* Les langages JavaScripts, css, html.
* Tout ce qui concerne l’architecture client/serveur, front/back, l’utilisation des api et de React.
* La POO[[25]](#footnote-25), notion au départ très floue pour moi et complétement indispensable à n’importe quel développeur.
* Le cours Devops m’a appris qu’il est possible d’automatiser les tests unitaires dans GitHub et d’utiliser les variables d’environnement. Il m’a aussi permis à commencer à étudier le déploiement sur serveurs distants.

## Recherches personnelles.

Les cours que nous avons reçus était les plus complet possibles, mais nous n’avons pas pu approfondir certaines notions qui m’ont beaucoup manquées après coup.

La première mouture du projet était fonctionnelle mais mal conçue du point de vue architecture back.

* Les couches serveur/service/DAO n’étaient pas bien définies.
* La base de données était directement interrogée par les fonctions services via des requêtes SQL, sans utilisation d’ORM.

J’ai passé beaucoup de temps comprendre exactement le principe de l’ORM et son application. Maintenant le back est beaucoup mieux construit, bien plus clair.

Cela m’a également permis de bien mieux comprendre ce que je fais au quotidien dans mon nouveau travail.

Beaucoup de temps aussi notamment pour les tests.

En théorie le concept est assez évident mais dans la pratique, on ne sait pas forcément quoi tester et comment le faire.

Le codage javaScript m’a posé pas mal de problème.

Finalement j’ai pu faire des tests unitaires sur les fonctions front utilisées par les composants React, et des tests d’intégration grâce à Postman.

J’ai découvert les triggers de base de données dans le cadre de mon travail. Bien sûr, chez Société Générale, nous utilisons des outils qui génèrent automatiquement les scripts.

J’ai pensé qu’ils pourraient m’être très utiles pour l’archivage et la sauvegarde des données. Il m’a fallu donc apprendre à les écrire manuellement.

## Point sur l’application à aujourd’hui

SOS Immo est opérationnelle dans sa mission principale (suivi des incidents).

Elle possède un front end, un back end et une base de données.

La partie authentification est terminée.

La grande majorité des fonctionnalités est prête, il me reste à finaliser l’administration de certaines données et l’archivage des données incidents et utilisateurs.

# ANNEXES

## Annexe 1 – rendu du pipeline GitHub

Une image contenant texte

Description générée automatiquement

## Annexe 2 – exemple de scénario complexe

Suivre à la lettre les indications dans l’application front avec un environnement de test.

Prérequis :

La base test doit posséder deux utilisateurs Admin et Technicien avec les caractéristiques indiquées aux étapes A1 et T1.

Elle ne doit pas posséder d’utilisateurs avec comme id « nduval ».

Recommandation :

Pour gagner du temps et simplifier la procédure, utiliser 1 navigeur par profil d’utilisateur, soit 3 navigateurs différents (Chrome, Edge, Mozilla…).

Tests :

Admin crée un nouvel utilisateur.

Il voit les incidents dans son écran pilotage.

Un nouvel utilisateur change son mot de passe à la première connexion.

Il fait un signalement et valide ou non l’intervention une fois terminée.

Technicien voit ce signalement dans son suivi.

Il prend en charge et termine une intervention.



## Annexe 3 - Collections Postman

Bien sûr, les requêtes POST nécessite des données prédéterminées.

Pour cela, Postman permet d’implémenter des variables manuellement par avance.

Il permet aussi de récupérer la réponse des requêtes dans des variables.

Pour chaque collection, on peut cocher/décocher les services à contrôler.

L’enchaînement est déclenché par le bouton Run.

Une image contenant texte

Description générée automatiquement

Résultat :

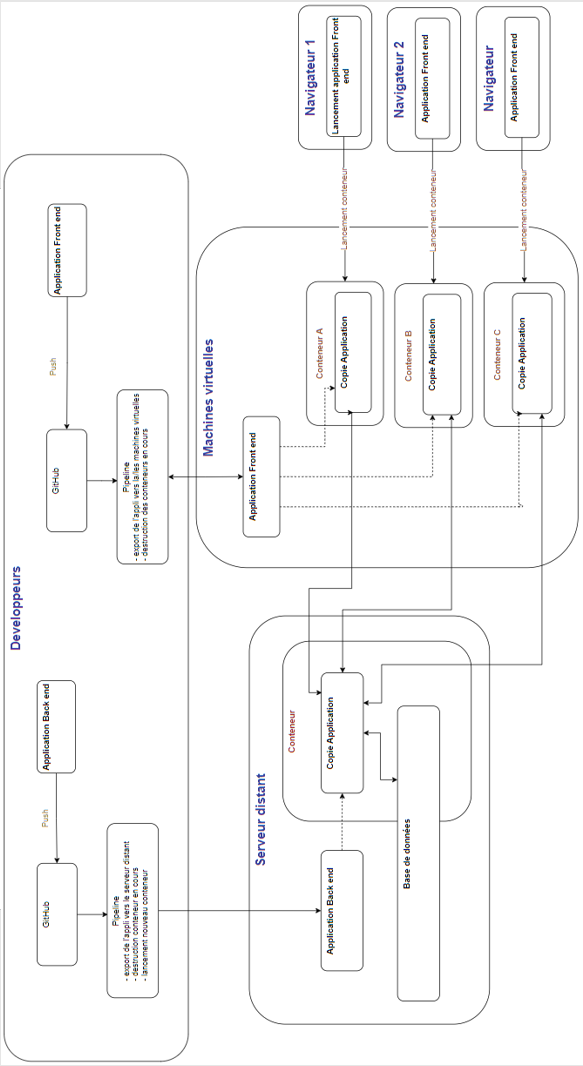
Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

## Annexe 4 - schéma déploiement



1. L’immeuble Basalte (Société Générale), à la Défense. [↑](#footnote-ref-1)
2. Un profil externe ne peut travailler que pour une seule entreprise prestataire. [↑](#footnote-ref-2)
3. De n’importe quel type. [↑](#footnote-ref-3)
4. Une useState déclarée dans App.js gardera tout le temps son état. [↑](#footnote-ref-4)
5. «Menu.js» contient toutes les fonctionnalités de l’application, accessibles par des boutons.

   L’affichage ou non de ces boutons est conditionné au profil de l’utilisateur. [↑](#footnote-ref-5)
6. Object Relational Mapping [↑](#footnote-ref-6)
7. Data Access Object [↑](#footnote-ref-7)
8. On peut noter au passage que la nouvelle entreprise prestataire va reprendre la plupart des salariés de l’ancienne entreprise. Il n’y a donc que très peu d’impact sur le suivi des incidents. [↑](#footnote-ref-8)
9. Conservé dans la table Journaux. [↑](#footnote-ref-9)
10. Le mai doit être unique. Il servira en cas d’oubli de donnée d’authentification. [↑](#footnote-ref-10)
11. Les modalités de sauvegarde du mot de passe seront traitées dans le chapitre « [gestion du mot passe](#_Gestion_de_mot) ». [↑](#footnote-ref-11)
12. 12 caractères dont 1 majuscule, 1 minuscule, 1 chiffre et 1 caractère spécial => ()!@#$+-\*&\_. [↑](#footnote-ref-12)
13. Universally Unique Identifier [↑](#footnote-ref-13)
14. « true » après réussite de l’authentification – « false » en cas de déconnexion. [↑](#footnote-ref-14)
15. Identifiant peut être retrouvé grâce à l’UUID et vice versa. Conserver les 2 données permet tout de même de simplifier le programme. [↑](#footnote-ref-15)
16. Là aussi, on peut retrouver l’information par l’UUID. L’idée est de simplifier les recherches. [↑](#footnote-ref-16)
17. Fonctionnalité annexe : l’identifiant du dernier utilisateur s’affiche directement si le navigateur possède encore un cookie (dernière connexion depuis moins de 10 minutes).

    Sinon, pas de retour, ce qui est parfaitement transparent pour l’utilisateur : le formulaire d’authentification reste vierge. [↑](#footnote-ref-17)
18. Prestataires, emplacements, type d’emplacements, types d’incidents, mapping type d’emplacements/types d’incidents [↑](#footnote-ref-18)
19. Ce qui ne signifie pas que l’on ne peut pas les supprimer. Il n’y aurait juste aucune d’incidence sur la base « sos\_immo\_sauv ». [↑](#footnote-ref-19)
20. « db\_sos\_immo\_test » par exemple [↑](#footnote-ref-20)
21. Intégration : première fois que l’on contrôle un service.

    Non-régression : contrôle d’un service déjà en production pour vérifier qu’il passe toujours bien. [↑](#footnote-ref-21)
22. Nombre d’incidents pour une période sur un étage, par prestataire, par type d’incident. Taux de satisfaction. [↑](#footnote-ref-22)
23. Contrairement au 20ème siècle, on peut féminiser le métier. [↑](#footnote-ref-23)
24. Sauf pour le recueil de besoin pour lequel je me suis fait aider par les services immobiliers Société Générale. [↑](#footnote-ref-24)
25. Programmation Orientée Objet [↑](#footnote-ref-25)