Rapport d’audit

&

Propositions pour la refonte

De l’application

Rendez-vous

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Auteur | Date | Commentaires |
| 1.0 | Laurent Payen | 18/11/2016 | Initialisation du document, premier audit |
| 1.1 | Laurent Payen | 23/12/2016 | Ajout de propositions, du modèle de données ainsi que d’une maquette de quelques écrans |
| 1.2 | Laurent Payen | 06/01/2016 | Ajout de correctifs suite à l’entrevue du 5 janvier + ajout de propositions d’évolution |

Table des matières

[1) Introduction 3](#_Toc470272403)

[2) Fonctionnel 3](#_Toc470272404)

[a) Front Office – Côté usager 3](#_Toc470272405)

[b) Back Office – Côté administrateur et agent 4](#_Toc470272406)

[c) Timezone 9](#_Toc470272407)

[d) Workflow 9](#_Toc470272408)

[e) Questions ouvertes / Propositions 10](#_Toc470272409)

[3) Modèle de données 11](#_Toc470272410)

[a) Les clés étrangères 11](#_Toc470272411)

[b) Les dates 11](#_Toc470272412)

[c) Les jours d’ouverture 12](#_Toc470272413)

[d) Le script d’initialisation 13](#_Toc470272414)

[e) La gestion du CRUD 13](#_Toc470272415)

[f) Nomenclature 13](#_Toc470272416)

[g) Proposition modèle de données 14](#_Toc470272417)

[h) Questions ouvertes 14](#_Toc470272418)

[4) Architecture 15](#_Toc470272419)

[a) Dao 15](#_Toc470272420)

[b) Les beans 19](#_Toc470272421)

[c) Divers 19](#_Toc470272422)

[5) Serveurs / Infra 21](#_Toc470272423)

[a) Serveurs 21](#_Toc470272424)

[b) SGBD 21](#_Toc470272425)

[6) Conclusion 22](#_Toc470272426)

# Introduction

Ce document a pour but de mettre en lumière les points qui pourraient être corrigés ou améliorés sur le plugin Rendez-Vous, que ce soit fonctionnellement ou au niveau de sa mise en œuvre, avec pour chaque point soulevé, une proposition d’amélioration.

# Fonctionnel

Avant tout développement d’application, une définition claire et précise du besoin métier est essentielle. Quelles sont les fonctionnalités essentielles que l’on souhaite ?

Dans tous les cas il faut que cela reste simple, rapide, ergonomique et facile d’accès.

Il est préférable d’avoir une application qui ne fasse qu’une seule chose mais qui le fasse bien.

Il n’existe pas de spécifications fonctionnelles décrivant le besoin métier ou encore les cas d’utilisation. Cela rend plus difficile la compréhension du besoin.

## Front Office – Côté usager

Côté front office, c’est-à-dire côté utilisateur, l’application rendez-vous est dans son ensemble simple, ergonomique et assez clair (pour un modèle de calendrier « Calendrier »).

Ça l’est moins pour le modèle de calendrier « Liste des créneaux disponibles » par exemple, car rien n’indique à l’utilisateur que ce sont des créneaux libres, il faut qu’il le devine (plus facile à deviner si de nombreux créneaux sont déjà pris, auquel cas il ne reste que quelques horaires affichés pour une journée)

D’autre part, il est possible de naviguer d’une semaine à l’autre en cliquant sur « semaine suivante » ou « semaine précédente » mais également sur les flèches gauche et droite présentes en haut à gauche du calendrier. Pourquoi ce double choix ? D’autant que l’utilisateur peut également sélectionner directement une date.

Enfin, petite remarque sur le fonctionnement, au moment de l'enregistrement en base d’un rendez-vous, aucune vérification n’est faite sur la disponibilité du créneau. Il est donc possible de faire du surbooking ...

**Proposition 2a1 :** Ok mais pas prioritaire

* Rafraichissement du calendrier (par call Ajax) toutes les minutes (ou temps à définir), afin d’éviter le surbooking.

**Proposition 2a2 :** Ok

* Simplification de la navigation par semaine (un seul moyen de naviguer, soit par les flèches ou les liens nommés, en plus de l’accès direct par calendrier)

## Back Office – Côté administrateur et agent

Côté Back-Office, c’est-à-dire côté administrateur de l’application, il existe beaucoup trop de possibilités de paramétrages. Les multiples combinaisons et possibilités, associées à un modèle de données non pertinent, se traduisent par une architecture mal distribuées et un code trop lourd et trop complexe, entraînant de nombreux bugs et une maintenabilité très difficile.

Par exemple, au niveau des onglets, il y a les jours d’ouverture, les jours de fermeture, les semaines spécifiques et les semaines types. Pourquoi avoir séparé jour d’ouverture et jour de fermeture ? Un jour non ouvert est un jour fermé non ? Pourquoi spécifier jour par jour un jour d’ouverture dans l'onglet jour d'ouverture lorsque l'on peut le faire à la semaine dans semaine spécifique ?

Autant de questions qui traduisent une trop grande diversité de manipulation pour un même besoin principal côté back office : définir les jours ouvrés.

**Proposition 2b1 :** Ok

Afin de simplifier la définition des jours ouvrés pour l’administrateur, les 4 onglets désignés par jour d’ouverture, jours de fermeture, semaine type et semaines spécifiques peuvent être rationalisés et fusionnés en deux onglets, l’un pour la semaine type, l’autre pour la semaine spécifique. Sur ce dernier, il faudra ajouter la possibilité de modifier une journée (1) en plus de celle de modifier un créneau (2).

* Semaine Type :

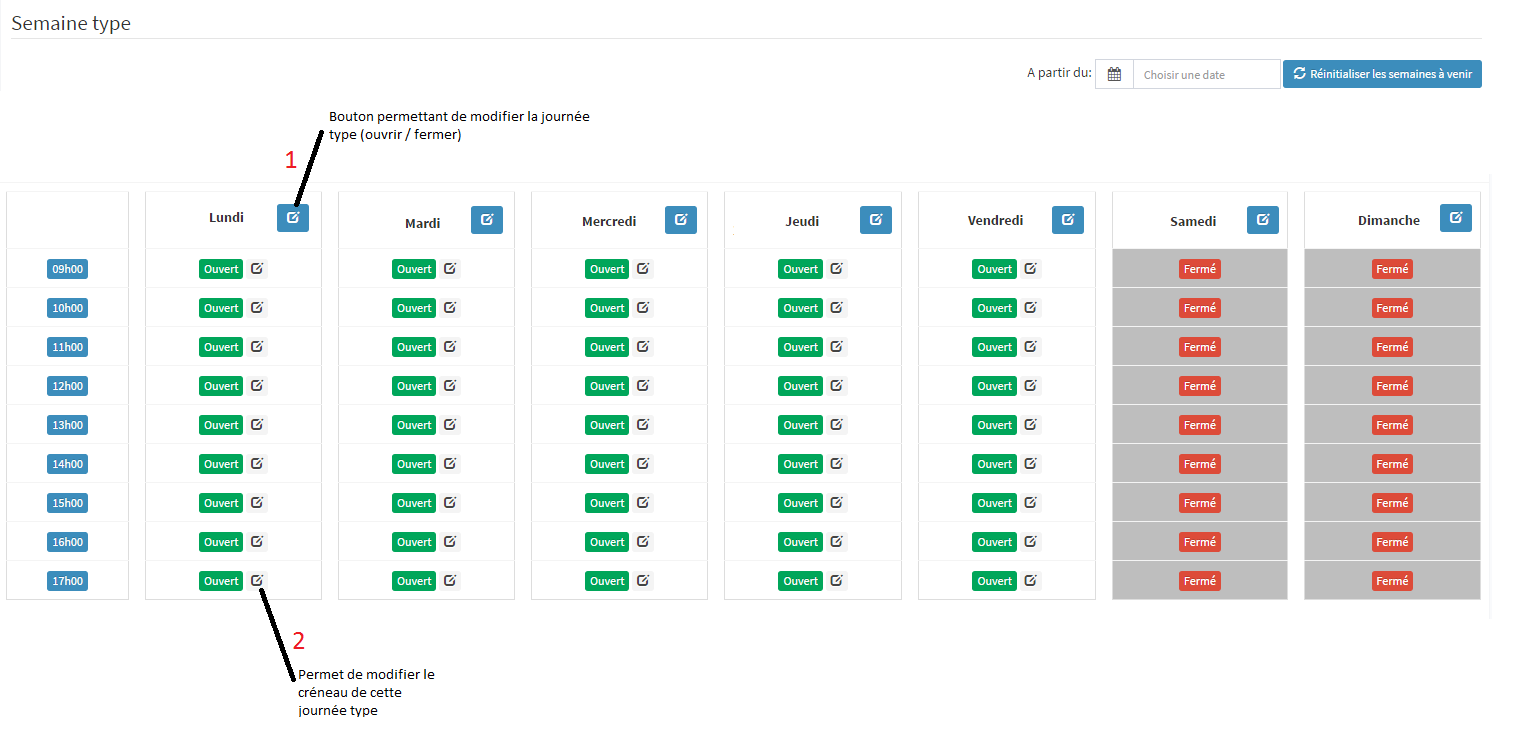


Figure : Semaine Type

Un clic sur (1) permet de fermer ou ouvrir directement une journée de la semaine type, sans avoir à aller dans les paramètres avancés du formulaire.

Un clic sur (2) permet d’accéder à la modification du créneau de cette journée type.

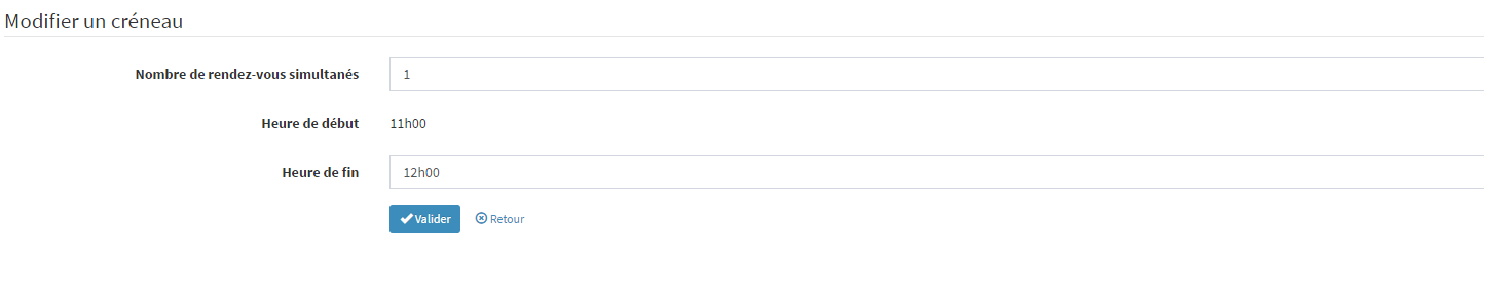


Figure : Modifier un créneau d'une journée type (2)

* Semaine spécifique

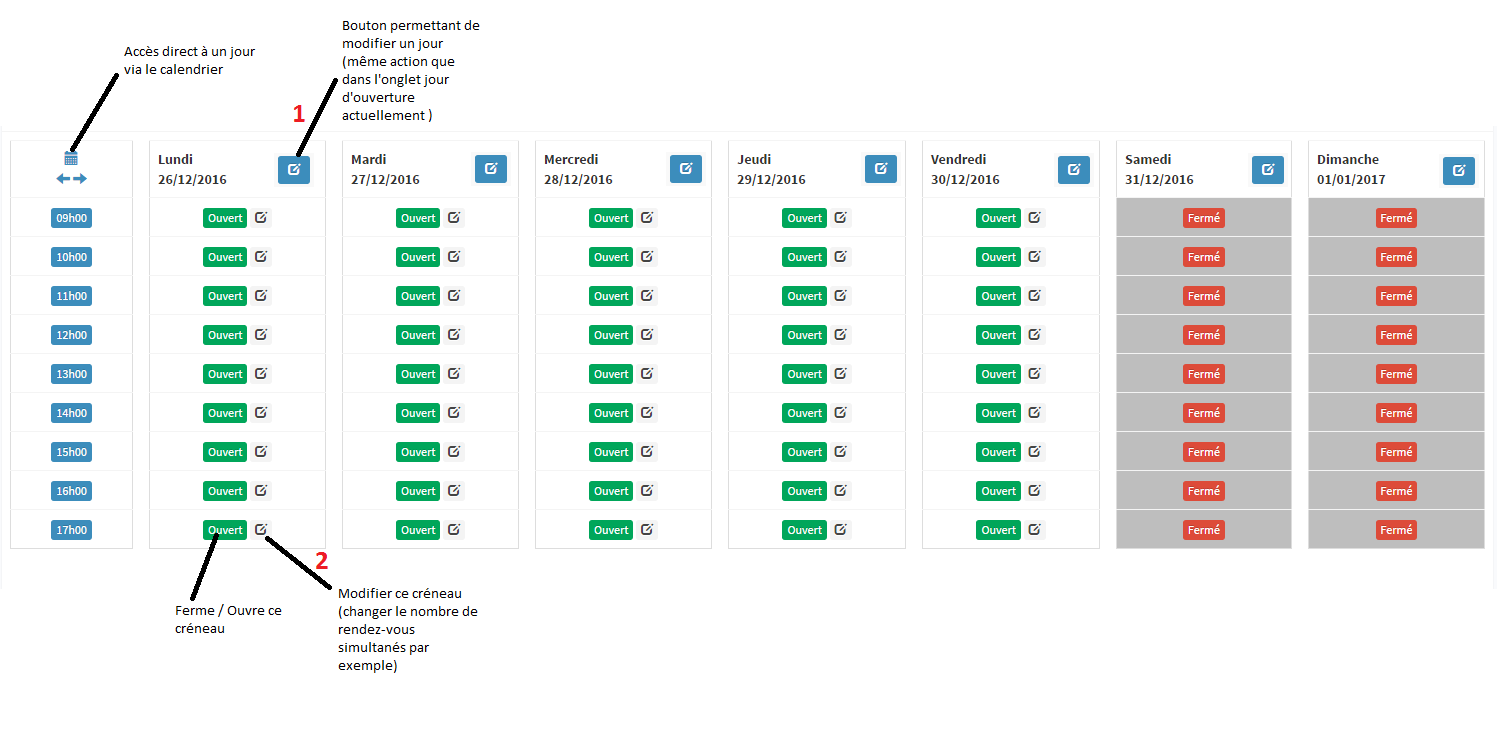


Figure : semaine spécifique

Possibilité de sélectionner directement un jour depuis l’icône calendrier et de le définir ouvert ou fermé.



Figure : Ecran de modification d'une journée (Clic sur 1)

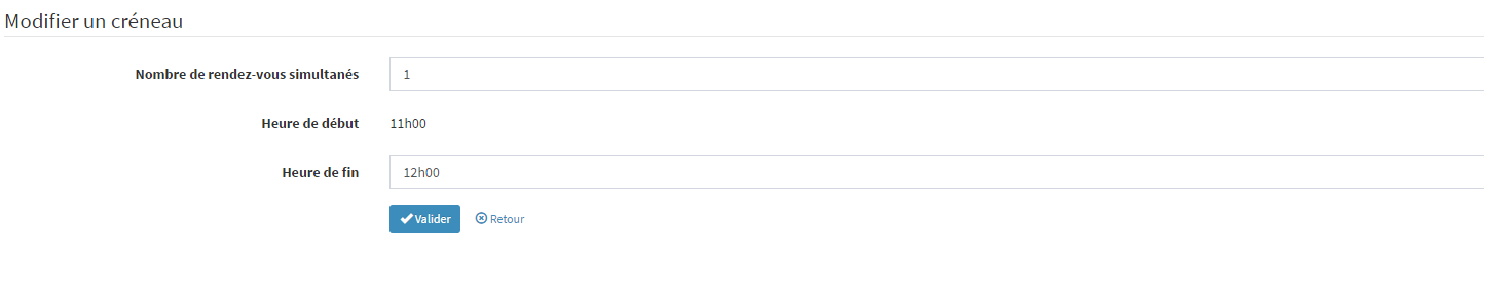


Figure : Ecran de modification d'un créneau (Clic sur 2)

**Proposition 2b2 :** Ok

Pour la fonctionnalité d’import des jours fermés/fériés, celle-ci serait déplacée dans l’écran des paramètres du formulaire, afin de supprimer l’onglet jour de fermeture.

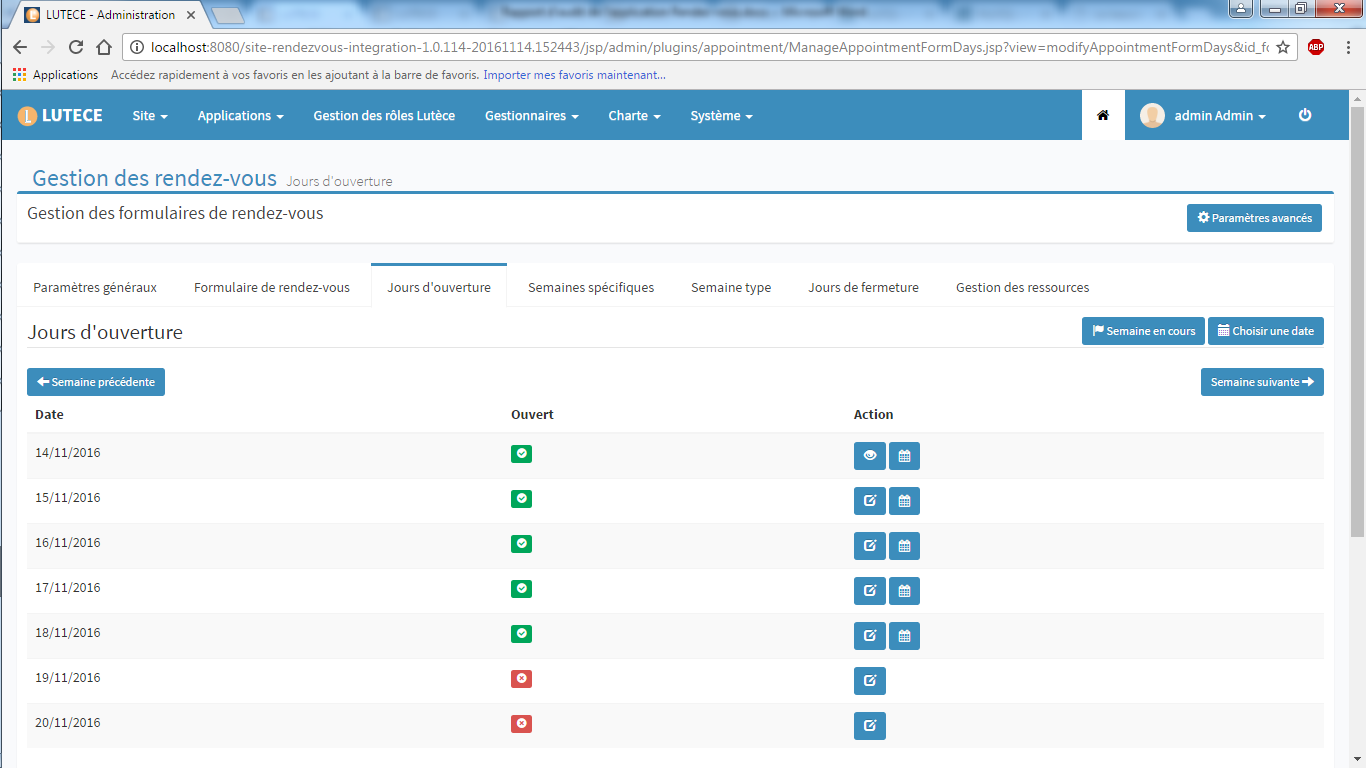
Ceci parait également plus fonctionnel d’importer les jours fériés directement à la création du formulaire, au même moment que la définition des jours ouvrés de la semaine type.

* Exemple de complexité de manipulation actuellement :

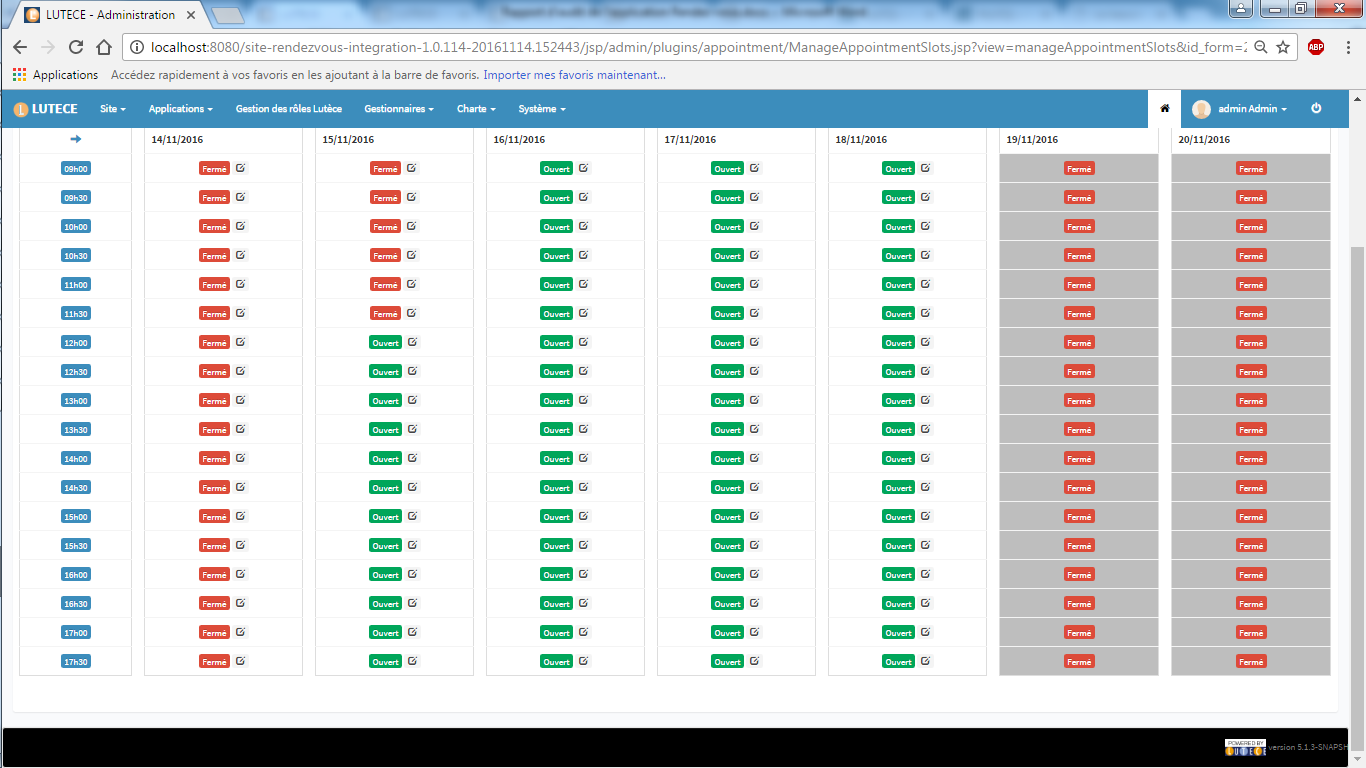
Certaines manipulations ne sont pas intuitives :

Dans cet exemple, un formulaire de rendez-vous avec des jours d’ouverture du lundi au vendredi a été défini.

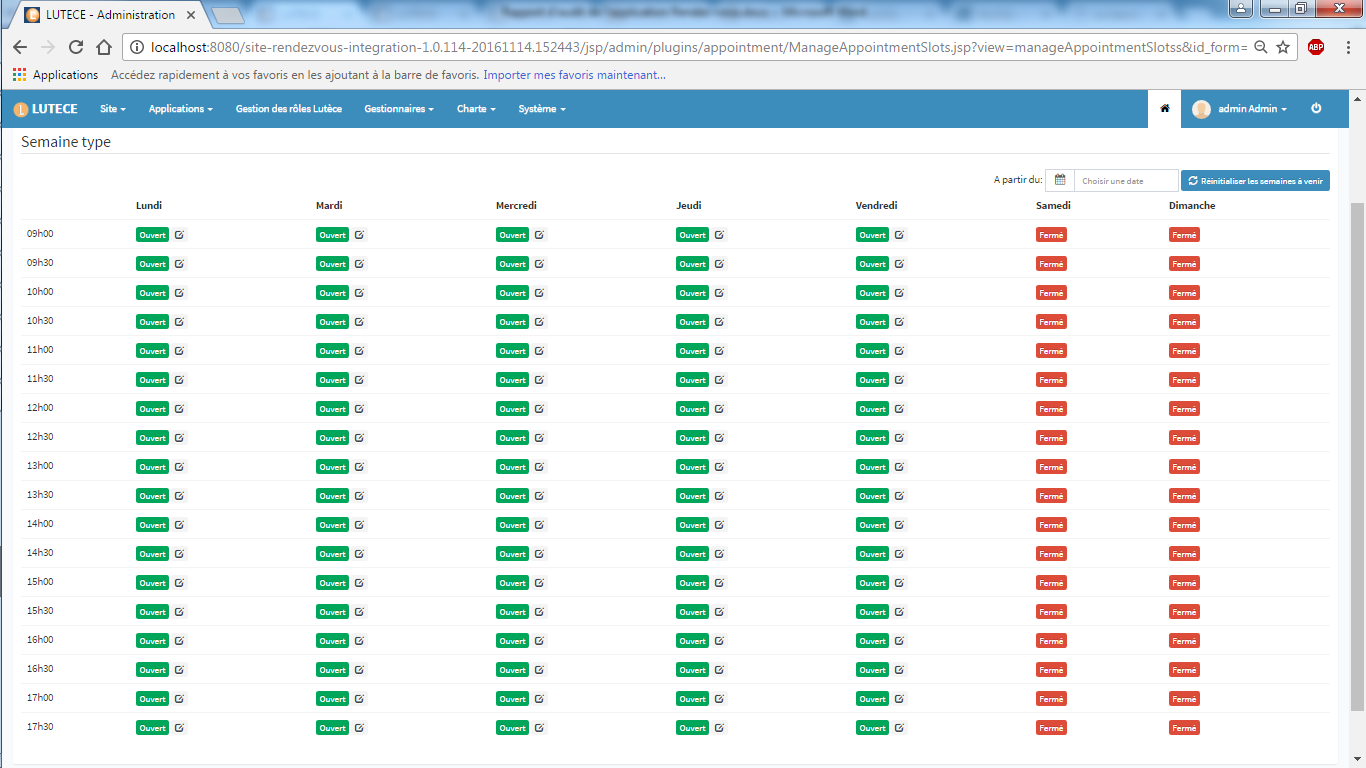
Dans l’onglet jours d’ouverture, pour la semaine courante, les jours ouverts apparaissent donc en vert :



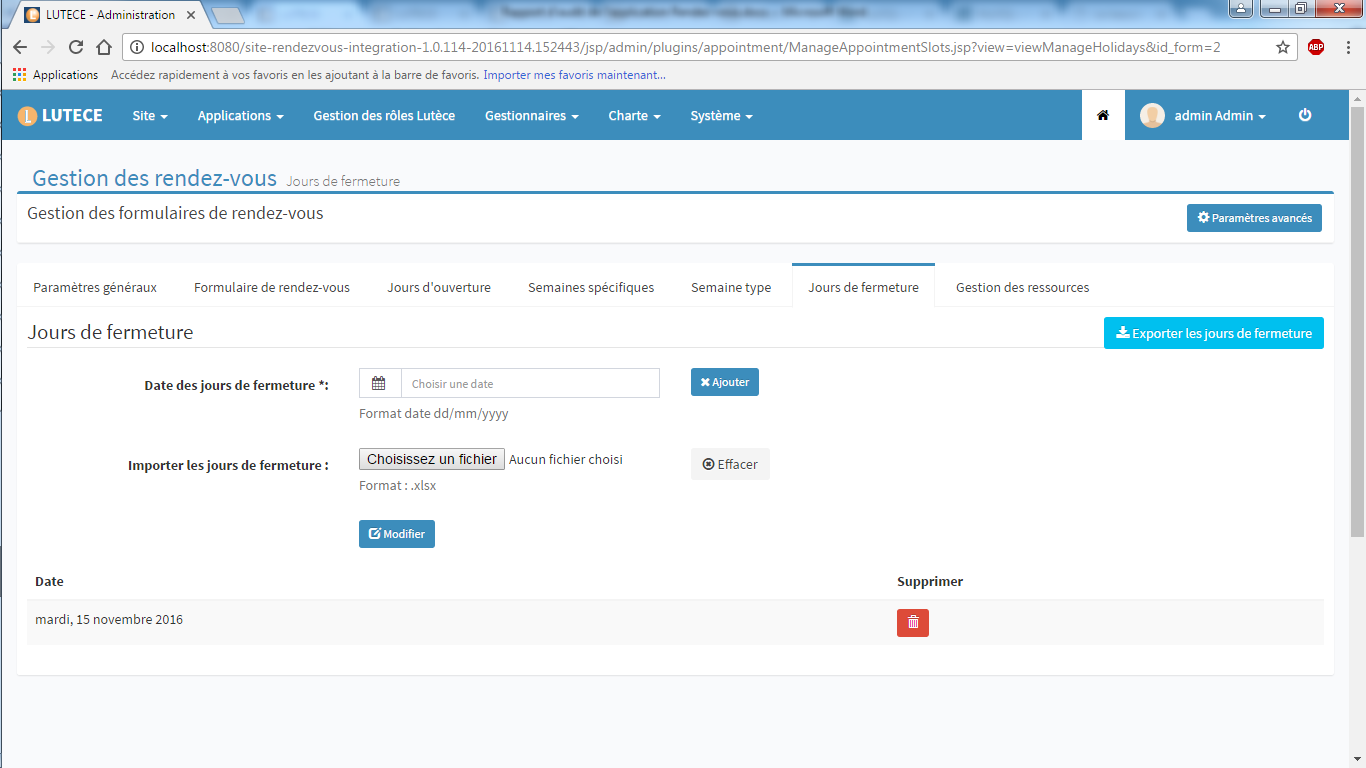
Dans l’onglet semaine spécifique, pour l’instant, il est calqué sur la semaine type (on voit juste le lundi 14/11 et le mardi 15/11 matin fermés car passés au moment du screenshot) :



Dans l’onglet semaine type, les jours du lundi au vendredi sont en ouverts, tels que définis lors de la création du formulaire :



On souhaite ajouter un jour de fermeture exceptionnelle le vendredi 15 Novembre :



Ce jour ne sera pris en compte qu’en retournant dans les paramètres avancés, en saisissant une date de modification à partir de laquelle la modification sera prise en compte et en cliquant sur modifier.

Seul problème, si des rendez-vous ont été pris après cette date de modification, la modification n’est pas possible, et donc le jour de fermeture n’est pas pris en compte.

**Proposition 2b3 :** Ok

Afficher les créneaux réservés impactés. L’utilisateur aura ensuite la possibilité d’effectuer une action sur chacun des créneaux concernés. (Ou alors de manière automatique, selon la définition du workflow).

* Autre exemple de complexité :

Il est possible de définir une durée avant le prochain rendez-vous possible (par exemple, il est possible de dire qu'il faut au moins 2h avant de prendre rendez-vous, donc s'il est 10h, on ne pourra prendre rendez-vous qu'à partir de midi, même s'il y a des créneaux disponibles avant midi).

Imaginons que les jours d'ouverture soient mardi et jeudi, et que l’on change cette valeur un mercredi, et que les plages d'ouverture soient de 9h à 18h, et que la valeur de délai avant un prochain rendez-vous soit de 2h, le comportement sera que les créneaux du jeudi et du mardi suivant (mais pas ceux d'après) seront dispo à partir de 11h, même si on consulte le formulaire le mercredi, une fois la modification faite. Donc les 2 heures s'appliquent à partir du prochain créneau (dispo ou non) et non de l'heure qu'il est actuellement. Ce qui n’est pas forcément le cas d’usage réellement souhaité. Cette fonctionnalité doit plus refléter les conditions réelles d’utilisation d’un tel outil. Il parait pertinent de penser que l’on souhaite interdire la prise de rendez-vous pendant x temps pour une absence exceptionnelle, donc à partir du moment présent, et non pas du prochain créneau horaire disponible.

Les filtres de recherche de la gestion des rendez-vous par filtre n’est pas ergonome. D’un côté, dans le cadre « rechercher » il existe plusieurs filtres, mais il existe également une liste déroulante à droite de l’écran (Accéder aux rendez-vous à venir, du jour, tous les rdv).

Proposition : fusionner tous les critères de recherche en un seul élément de filtre et conserver en mémoire les critères saisis, de sorte que lors de la modification ou la suppression d’un des rendez-vous apparaissant dans les résultats de la recherche, les critères saisis soient conservés.

## Timezone

A aucun moment dans l’application il n’est pris en compte la question du fuseau horaire.

Cela pose problème pour l’envoi de confirmation de rendez-vous avec un fichier ics attaché par exemple.

De plus, cela limite l’utilisation de l’application (pas d’internationalisation possible)

En effet les heures des créneaux sont stockées en ne prenant pas en compte le fuseau horaire.

**Proposition 3c1 :** Non pertinent

L’idéal serait de stocker les heures au format UTC (gmt+0) et la gestion de l’affichage dépendrait du fuseau horaire du client.

## Workflow

Du côté du module workflow appointment, l’utilisateur doit définir un workflow pour chaque formulaire. Par exemple, la Direction Logement et Habitat a créé 20 worflows (un par arrondissement). Les workflows sont identiques, la seule différence tient dans le corps du mail de confirmation de rendez-vous, avec la citation de l’arrondissement concerné ainsi que l’adresse. Une modification sur la procédure de rendez-vous ou de validation implique la modification de ces 20 workflows.

**Proposition 3d1 :** Nok Hors scope plugin

Une solution, afin de n’avoir qu’un seul workflow à définir dans ce genre de cas, serait d’avoir la possibilité d’ajouter des champs custom dans le formulaire. L’utilisateur aurait alors à définir un seul workflow, ainsi qu’un corps de mail type avec les champs custom définis. Ceux-ci seront remplacés par les valeurs correspondantes, définies dans chaque formulaire.

Côté workflow, un aperçu pourrait être disponible afin de visualiser le rendu final, une fois les champs custom remplacés.

**Proposition 3d2 :** Ok

D’autre part, actuellement il est permis de modifier le workflow rattaché au formulaire. Ceci implique la perte de tous les états des rendez-vous actuellement pris. Il apparait plus prudent d’interdire le changement de workflow d’un formulaire à partir du moment où celui-ci a des rendez-vous.

## Questions ouvertes / Propositions

* **Proposition 2e1** : Ok, à voir dans le détail (catégorie par ex)

A la création d’un formulaire, de nombreux champs et options sont disponibles, dont quelques-uns pourraient être supprimés ou rendus obligatoires.

* + « catégorie » : à quoi sert ce champ ? Il apparait qu’il n’est pas réutilisé par la suite pour classifier les formulaires.
  + « Autoriser les utilisateurs à annuler un rendez-vous » : coché par défaut, il ne devrait pas pouvoir être décoché. De plus, toute la gestion de l’annulation se fait côté workflow.
  + « Jusqu'à une date précise » : il s’agit ici d’une date butoir à partir de laquelle le calendrier du formulaire ne sera plus affiché, et donc sur lequel il ne sera plus possible de prendre rendez-vous. Dans ce cas, cette option fait double emploi avec la date de fin de validité du formulaire. De plus, il apparait beaucoup plus usuel de définir un nombre de semaine à afficher (l’autre option disponible).
  + Il peut exister une confusion entre la durée de validité d’un formulaire et le nombre de semaines à afficher aux utilisateurs.
  + « Affichage formulaire priorité » : si cette case est cochée, le formulaire sera affiché avant le calendrier, ce qui dans le cas pratique n’est ni intuitif ni efficace (sauf peut-être dans le cas du remplissage automatique après une redirection, mais dans ce cas, ne peut-il être prérempli après avoir sélectionné la date ?). L’utilisateur devra en effet renseigner tous les champs du formulaire pour ensuite choisir une date et vérifier que celle-ci est disponible.
  + « Activer la confirmation d'email » : dans le cas où l’email est demandé sur un formulaire, il est d’usage et plus que recommandé de demander à l’utilisateur une confirmation de l’email renseigné (en interdisant le copier-coller). Une faute de frappe est vite arrivée et ne permettrait pas d’envoyer une confirmation ou un rappel pour le rendez-vous pris.
* **Proposition 2e2** : Nok, pas de durée variable

De base, tous les créneaux ont une durée fixe, définie à l’origine de la création du formulaire. Quid d’une durée variable ?

* **Proposition 2e3** : Nok, permet de différencier les paramètres applicables immédiatement des paramètres qui peuvent être appliqués à partir d’une certaine date.

Différencier les paramètres qui doivent être appliqués à partir d’une certaine date (dans les paramètres avancés) des paramètres simples (paramètres généraux) amène une complexité à l’utilisation. Il faut en effet connaitre la répartition des paramètres dans ces deux pages. Pourquoi ne pas tout réunir dans la même page, avec la même présentation que lors de la création du formulaire. L’utilisateur ne sera pas désorienté ainsi.

# Modèle de données

Quand le besoin métier est clairement défini, des entités se détachent naturellement de ces spécifications. Le modèle de données doit s’en rapprocher au maximum et les traduire le plus simplement possible.

Dans le modèle de données du plugin rendez-vous, de nombreuses erreurs de conceptions et de nombreuses absences d’optimisations ont été commises.

## Les clés étrangères

Le framework Lutèce utilise peu ou pas de clés étrangères entre les tables. Elles permettent pourtant une cohérence des données.

**Proposition 3a1 :** Ok

Définir les clés étrangères là où elles sont nécessaires (cf modèle de données).

## Les dates

Le découpage des dates dans le plugin rendez-vous n’est pas judicieux.

Par exemple, pour définir un horaire d’ouverture dans la table appointment\_day, un champ opening\_hour en Integer et opening minute en Integer ont été créés. Même chose pour l’horaire de fermeture. Pourquoi ne pas avoir utilisé un champ de type Datetime ou Timestamp, qui comprend la date et l’heure ?

Du côté du formulaire, pour la table appointment\_form, on gère la date de début et la date de fin en varchar(10). Ensuite ces champs seront découpés avec les heures avant le h et les minutes après le h.

Côté DAO et Controller, cela se traduit par énormément de parsing et de conversions de integer en date et inversement.

Un champ de type Datetime peut être utilisé nativement par son entité comme une date java, sans aucune conversion.

Pourquoi stocker la durée d’un appointment quand on a l’heure de début et l’heure de fin ?

Même chose pour les places libres, on stocke les places libres restantes sur un créneau.

Dans les deux cas on est obligé d'aller lire en base, soit le nombre de créneaux, soit la valeur du champ place libre. Mais si on utilise le champ places libres, il faudra le mettre à jour ensuite.

**Proposition 3b1 :** Ok

Utiliser les types Time et Timestamp en base de données. Indexer ce champ dans la table des créneaux pour améliorer les performances de recherches.

Limiter au maximum les calculs de dates.

## Les jours d’ouverture

Pour la table appointment\_form, qui représente le formulaire de rendez-vous, on a traduit les jours d’ouverture par un champ pour chaque jour (is\_open\_monday, is\_open\_tuesday, …, is\_open\_sunday).

En plus d’être encodés en SmallInt(6) (il eût été plus judicieux d’émployer un Tinyint(1) puisqu’il s’agit d’un booléen ouvert/fermé), il apparait rapidement que chaque formulaire va contenir les 7 jours de la semaine. Les données ne seront pas facilement exploitables.

**Proposition 3c1 :** Ok, inclus dans la refonte du modèle

Une association entre le formulaire et les jours de la semaine par une table contenant un champ ouvert/fermé permettrait de mutualiser les jours de la semaine pour l’ensemble des formulaires.

De nombreuses erreurs peuvent subvenir par l’utilisation d’une même donnée pour deux sens différents fonctionnellement. Cette erreur de conception a été commise sur la table appointment\_slot. Cette table définit les créneaux d’ouverture pour chaque jour, donc en relation avec la table appointment\_day par son id id\_day.

Mais cette table permet également de stocker les horaires de références définis lors de la création du formulaire. Dans ce cas, pour les horaires de référence, un id\_day à 0 est utilisé pour différencier ce créneau horaire d’un créneau horaire « classique ».

Seul problème, dans le code, à chaque requête sur cette table, il est nécessaire de différencier les résultats ayant un id\_day à 0 de ceux ayant un id\_day différents de 0. Et de nombreuses erreurs peuvent être commises (lors de la mise à jour ou sauvegarde par exemple)

**Proposition 3c2 :** Ok, inclus dans la refont avec la table working\_day

Les horaires de références doivent clairement être différenciés et sortis de cette table de créneaux disponibles, ce sont deux aspects métiers totalement différents.

## Le script d’initialisation

*Nota bene : la mise en place d’un framework ORM a déjà été discutée lors du premier point avec l’équipe Lutèce. Ce paragraphe est juste laissé à titre informatif.*

De nombreuses erreurs existent sur ces scripts (par exemple des primary key qui sont mis en default null, ce qui rend la création de la table impossible), là où un framework ORM est capable de générer automatiquement le script de création de table par rapport aux entités définies.

Cela permet également d’avoir le modèle de données clairement défini dans les entités, avec, pour chaque champs, l’ensemble de ses contraintes.

**Proposition 3d1 :** Nok

Générer automatiquement le script de création des tables via l’ORM.

## La gestion du CRUD

*Nota bene : la mise en place d’un framework ORM a déjà été discutée lors du premier point avec l’équipe Lutèce. Ce paragraphe est juste laissé à titre informatif.*

Actuellement, toutes les écritures/lectures sont gérées manuellement dans les DAO. Ceci rend le code très lourd, souvent illisible, et source de nombreuses erreurs.

Un framework ORM permet de s’abstraire de toutes ces opérations et de fiabiliser l’intégrité des données et des transactions.

## Nomenclature

Sur certaines tables, il n’existe pas de concordance forte entre les champs affichés et le nommage de celui-ci en base de données.

Par exemple, pour le formulaire, le « délai minimum de prise de rendez-vous » (en heures) est nommé min\_days\_before\_app.

Pour le « nombre maximum de rendez-vous valides » c'est max\_appointment\_mail et pour la « durée en nombre de jours », nb\_appointment\_week

Il peut exister également des erreurs dans les traductions, par exemple, pour le nombre de rendez-vous simultanés, celui-ci se traduit en base par people\_per\_appointment, ce qui peut prêter à confusion.

**Proposition 3f1 :** Ok

Sans reprendre l’ensemble du champ, en plus de règles de nommages strictes (des majuscules apparaissent dans certains champs de table), il est indispensable d’avoir une corrélation forte entre le nom du champ affiché et celui correspondant en base de données.

## Proposition modèle de données

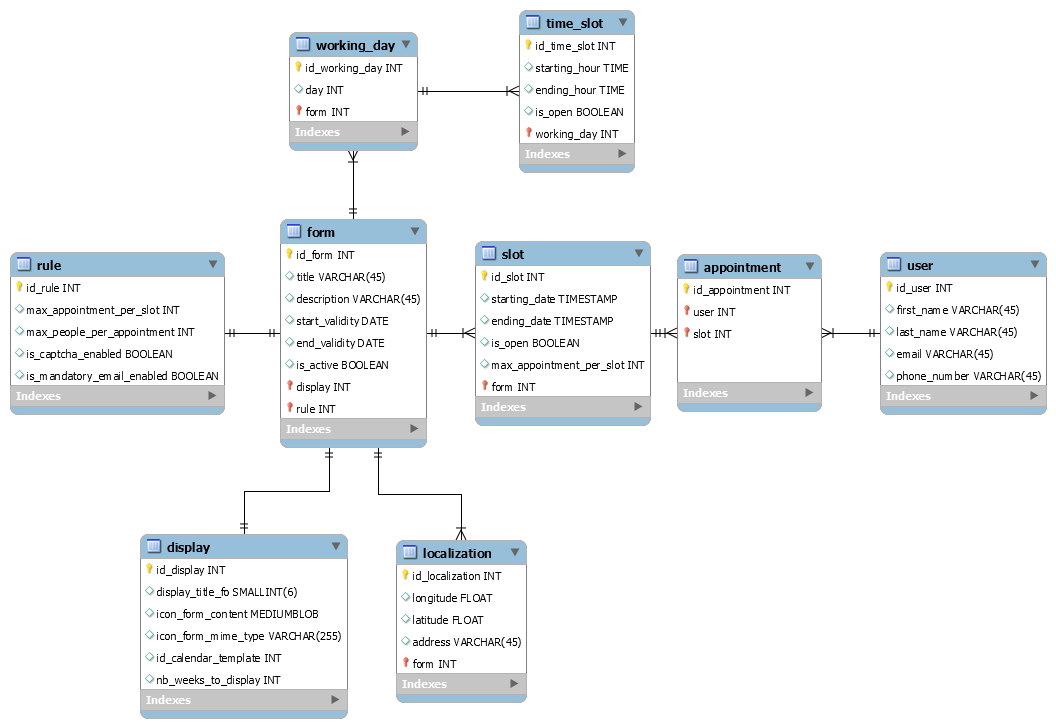


Figure : Proposition 3g1

## Questions ouvertes

* Lors de la création d’un formulaire, avec les heures de début et de fin des rendez-vous et leur durée moyenne, une première création en masse est effectuée dans la table appointment\_slot, afin de créer tous les créneaux « type » (id\_day=0).

Une fois le formulaire validé, ce sont tous les créneaux possibles qui vont être créés dans cette table, rattachés aux dates réelles.

Par exemple, avec un formulaire sur 10 semaines, ouvert du lundi au vendredi, de 9h00 à 18h00 et une durée des créneaux de 30 minutes, la création du formulaire créé en base 90 entrées, sa validation 2250 entrées uniquement sur cette table (175 dans la table appointment\_day)

Cette création à l’avance de tous les créneaux possibles sera à recréer en cas de modification de la semaine type ou de tout autre élément impactant les horaires/jours d’ouverture ou la durée des créneaux.

**Proposition 3h1 :** Ok mais à voir pour les jours fériés définis par l’utilisateur

Créer uniquement les créneaux fermés à l’avance. Les créneaux réservés seront créés au fur et à mesure des réservations. Ainsi, si un créneau n’apparait ni dans ceux fermés, ni dans ceux réservés (selon le nombre de rendez-vous simultanés), alors ce créneau est disponible.

Pour optimiser au mieux la création de créneaux, il faut donc gérer 3 types de créneaux :

* + les créneaux type, qui définissent ceux ouverts lors de la création du formulaire (non rattachés à une date mais à un jour de la semaine avec une plage horaire), stockés dans *working\_day* pour les jours et *time\_slot* pour les créneaux de ce jour
  + les créneaux spécifiques qui définissent ceux fermés par l'utilisateur (dans la table *slot*, avec *is\_open* à *false*)
  + et enfin les créneaux réservés par les utilisateurs (dans la table *slot*, avec *is\_open* à *true*)

Avec ces 3 données, on optimise les entrées en base de données. Il suffit d’indexer ces trois tables sur les champs de type timestamp/time et les recherches seront optimales.

Lors d’un changement de la semaine type ou du nombre de places possible, la modification est immédiate, aucune information n’étant lié à une date concernant la semaine type ou les règles associées au formulaire.

**Proposition 3h2 :** Nok, Il faudra dans tous les cas créér des créneaux à l’avance du fait de la possibilité de modifier par la suite la capacité d’accueil par exemple.

Si l’on souhaite conserver la possibilité de modifier les paramètres avancés à partir d’une certaine date, sans prise d’effet immédiate, il est nécessaire de créer les créneaux en avance de phase, ceci afin de permettre de modifier les créneaux ultérieurs à la date de modification sélectionnée et uniquement ceux-ci. Afin de limiter la création en masse, il est néanmoins possible de ne créer que les créneaux ouverts. Ceux qui ne seront pas stockés seront automatiquement considérés comme des créneaux fermés. (Dans la table *slot*)

# Architecture

Le framework Lutece impose une architecture assez stricte (Couche présentation, avec les JSP, les JSP Bean et les templates, couche métier avec les DAO et la couche service).

Au fur et à mesure des développements, des dérives ont été commises sur le plugin rendez-vous.

## Dao

*Nota bene : la mise en place d’un framework ORM a déjà été discutée lors du premier point avec l’équipe Lutèce. Ce paragraphe est juste laissé à titre informatif.*

Un des principaux manque du framework Lutèce est l’absence de l’utilisation d’un framework de mapping objet relationnel. Cette absence impose d’écrire dans chaque DAO en string statique les requêtes SQL.

Cette écriture des requêtes peut être la cause de nombreuses erreurs.

Lorsque l’on souhaite récupérer des données liées, toutes les jointures sont à écrire dans la requête et rendent celle-ci rapidement illisible et difficilement maintenable.

De plus, la gestion des transactions et des connections se fait manuellement. Là encore, cible de nombreuses sources d’erreur, là où Hibernate (ou tout autre ORM implémentant JPA) est capable d’optimiser ces accès.

De plus, pour chaque DAO, des besoins spécifiques de requêtes existent, tels que des filtres, du tri ou de la pagination.

En combinant Hibernate et Spring Data, une fois l’entité correctement définie et en adéquation avec la table correspondante, tous les accès à ces données pourraient se faire depuis une interface repository.

L’affichage du calendrier semaine par semaine pourrait se faire en utilisant des objets Pageable par exemple, qui gèrent automatiquement la pagination et optimise les accès et requêtes en base de données. Même chose pour les tris.

Chaque nom de méthode décrit explicitement la requête que l’on souhaite faire, sans avoir besoin de l’implémenter, et retourne la (ou les) entités correspondantes.

Par exemple :

List<Appointement> findBySlot(Slot slot);

Retourne la liste des rendez-vous selon un créneau horaire.

Là où actuellement l’implémentation est :

On définit d’abord la requête dans une string :

*private static final String SQL\_QUERY\_SELECTALL = "SELECT app.id\_appointment, app.first\_name, app.last\_name, app.email, app.id\_user, app.authentication\_service, app.localization, app.date\_appointment, app.id\_slot, app.status, app.id\_action\_cancel, app.id\_admin\_user, app.has\_notify, slot.starting\_hour, slot.starting\_minute, slot.ending\_hour, slot.ending\_minute , app.nb\_place\_reserved FROM appointment\_appointment app INNER JOIN appointment\_slot slot on slot.id\_slot = app.id\_slot";*

Puis définition de la méthode qui va retourner la liste des rendez-vous en exécutant cette requête :

*public List<Appointment> selectAppointmentsList( Plugin plugin )*

*{*

*List<Appointment> appointmentList = new ArrayList<Appointment>( );*

*DAOUtil daoUtil = new DAOUtil( SQL\_QUERY\_SELECTALL, plugin );*

*daoUtil.executeQuery( );*

*while ( daoUtil.next( ) )*

*{*

*appointmentList.add( getAppointmentFormValues( daoUtil ) );*

*}*

*daoUtil.free( );*

*return appointmentList;*

*}*

Avec executeQuery définit tel que :

*public void executeQuery( )*

*{*

*try*

*{*

*\_resultSet = \_statement.executeQuery( );*

*}*

*catch( SQLException e )*

*{*

*free( );*

*throw new AppException( getErrorMessage( e ), e );*

*}*

*}*

Et DAOUtil, qui va récupérer la connexion et gérer la transaction :

*public DAOUtil( String strSQL, Plugin plugin )*

*{*

*\_bReleased = false;*

*\_strSQL = strSQL;*

*if ( plugin != null )*

*{*

*\_strPluginName = plugin.getName( );*

*\_connectionService = plugin.getConnectionService( );*

*}*

*else*

*{*

*\_strPluginName = DEFAULT\_MODULE\_NAME;*

*\_connectionService = AppConnectionService.getDefaultConnectionService( );*

*}*

*if ( \_connectionService == null )*

*{*

*throw new NoDatabaseException( "Database access error. Please check component installations and db.properties." );*

*}*

*// Use the logger name "lutece.debug.sql.<plugin\_name>" to filter logs by plugins*

*\_logger = Logger.getLogger( LOGGER\_DEBUG\_SQL + \_strPluginName );*

*if ( \_logger.isDebugEnabled( ) )*

*{*

*log( "Module : '" + \_strPluginName + "' - SQL Statement : " + \_strSQL );*

*}*

*try*

*{*

*MultiPluginTransaction transaction = TransactionManager.getCurrentTransaction( plugin );*

*if ( transaction != null )*

*{*

*\_bTransactionnal = true;*

*}*

*else*

*{*

*// We check if there is a managed transaction to get the transactionnal connection*

*if ( TransactionSynchronizationManager.isSynchronizationActive( ) )*

*{*

*\_bTransactionnal = true;*

*DataSource ds = AppConnectionService.getPoolManager( ).getDataSource( \_connectionService.getPoolName( ) );*

*\_connection = DataSourceUtils.getConnection( ds );*

*if ( \_logger.isDebugEnabled( ) )*

*{*

*\_logger.debug( "Transactionnal context is used for pool " + \_connectionService.getPoolName( ) );*

*}*

*}*

*else*

*{*

*// no transaction found, use the connection service directly*

*\_connection = \_connectionService.getConnection( );*

*}*

*}*

*if ( transaction != null )*

*{*

*\_statement = transaction.prepareStatement( \_strSQL );*

*}*

*else*

*if ( \_connection != null )*

*{*

*\_statement = \_connection.prepareStatement( \_strSQL );*

*}*

*else*

*{*

*throw new AppException( "Database access error for component '" + \_strPluginName + "'. Please check plugin installation and db.properties." );*

*}*

*}*

*catch( SQLException e )*

*{*

*free( );*

*throw new AppException( getErrorMessage( e ), e );*

*}*

*}*

Et enfin getAppointmentFormValues, qui va faire le mapping entre les données récupérées et l’objet que l’on souhaite manipuler et afficher, un rendez-vous (Appointment) :

*private Appointment getAppointmentFormValues( DAOUtil daoUtil )*

*{*

*Appointment appointment = new Appointment( );*

*Calendar dateAppointment = Calendar.getInstance( );*

*int nIndex = 1;*

*appointment.setIdAppointment( daoUtil.getInt( nIndex++ ) );*

*appointment.setFirstName( daoUtil.getString( nIndex++ ) );*

*appointment.setLastName( daoUtil.getString( nIndex++ ) );*

*appointment.setEmail( daoUtil.getString( nIndex++ ) );*

*appointment.setIdUser( daoUtil.getString( nIndex++ ) );*

*appointment.setAuthenticationService( daoUtil.getString( nIndex++ ) );*

*appointment.setLocation( daoUtil.getString( nIndex++ ) );*

*dateAppointment.setTime( daoUtil.getDate( nIndex++ ) );*

*appointment.setDateAppointment( new Date( dateAppointment.getTimeInMillis( ) ) );*

*appointment.setIdSlot( daoUtil.getInt( nIndex++ ) );*

*appointment.setStatus( daoUtil.getInt( nIndex++ ) );*

*appointment.setIdActionCancel( daoUtil.getInt( nIndex++ ) );*

*appointment.setIdAdminUser( daoUtil.getInt( nIndex++ ) );*

*appointment.setHasNotify( daoUtil.getInt( nIndex++ ) );*

*dateAppointment.set( Calendar.HOUR\_OF\_DAY, daoUtil.getInt( nIndex++ ) );*

*dateAppointment.set( Calendar.MINUTE, daoUtil.getInt( nIndex++ ) );*

*appointment.setStartAppointment( dateAppointment.getTime( ) );*

*dateAppointment.set( Calendar.HOUR\_OF\_DAY, daoUtil.getInt( nIndex++ ) );*

*dateAppointment.set( Calendar.MINUTE, daoUtil.getInt( nIndex++ ) );*

*appointment.setEndAppointment( dateAppointment.getTime( ) );*

*appointment.setNumberPlacesReserved( daoUtil.getInt( nIndex++ ) );*

*return appointment;*

*}*

Cet exemple démontre bien la complexité et la lourdeur de l’implémentation actuelle, là où un framework ORM tel que Hibernate est capable du même résultat en une seule ligne de code.

Cela permettrait également de se concentrer sur le cœur de métier de l’application.

**Proposition 4a1**

Combiner l’utilisation d’un ORM et de Spring Data.

## Les beans

Les JSP Bean (contrôleurs) sont trop volumineux (entre 1000 et 3000 lignes) et comportent peu ou pas de commentaires.

Le rôle d’un contrôleur est de gérer la présentation des informations issues des objets métier. Or ici, trop de logique métier se retrouve dans les contrôleurs. C’est au service d’assurer la répartition des fonctionnalités d’une application.

De plus, de nombreuses manipulation et transformations de date sont effectuées dans chaque méthode (dû aussi au modèle de données). Il n’existe aucune mutualisation de code. Par exemple toutes ces manipulations de date devraient se trouver dans des classes outils.

**Proposition 4b2 :** Ok

Interdire tout calcul ou manipulation complexe dans les controlers, ce ne sont pas leurs rôles.

Mieux définir les services. Créer des classes utilitaires.

## Divers

* Concernant le module workflow appointment, en ce qui concerne le code, celui-ci est mieux structuré et comporte peu de problèmes, le module utilisant ou surchargeant les méthodes définies dans le plugin workflow.
* Dans le plugin rendez-vous, il existe beaucoup de code mort, de méthodes inutilisées, méthodes en commentaire …

Exemple : Dans AppointmentSlotJspBean :

*// int nIdForm = Integer.parseInt( strIdForm );*

*// listSlots = AppointmentSlotHome.findByIdForm( nIdForm );*

*// form = AppointmentFormHome.findByPrimaryKey( nIdForm );*

*// nNbWeeks = form.getNbWeeksToDisplay( );*

*//*

*// boolean[] bArrayListDays =*

*// {*

*// form.getIsOpenMonday( ), form.getIsOpenTuesday( ), form.getIsOpenWednesday( ),*

*// form.getIsOpenThursday( ), form.getIsOpenFriday( ), form.getIsOpenSaturday( ),*

*// form.getIsOpenSunday( ),*

*// };*

*// AppointmentDay day = AppointmentService.getService( ).getAppointmentDayFromForm( form );*

*// day.setIsOpen( false );*

*// model.put( MARK\_READ\_ONLY, false );*

*//*

*// boolean bHasClosedDay = false;*

*//*

*// for ( int i = 0; i < bArrayListDays.length; i++ )*

*// {*

*// if ( !bArrayListDays[i] )*

*// {*

*// listSlots.addAll( AppointmentService.getService( ).computeDaySlots( day, i + 1 ) );*

*// bHasClosedDay = true;*

*// }*

*// }*

*//*

*// if ( bHasClosedDay )*

*// {*

*// Collections.sort( listSlots );*

*// }*

*// }*

*// else*

*// {*

*// String strIdDay = request.getParameter( PARAMETER\_ID\_DAY );*

*//*

*// if ( StringUtils.isNotBlank( strIdDay ) && StringUtils.isNumeric( strIdDay ) )*

*// {*

*// int nIdDay = Integer.parseInt( strIdDay );*

*// AppointmentDay day = AppointmentDayHome.findByPrimaryKey( nIdDay );*

*//*

*// if ( !day.getIsOpen( ) )*

*// {*

*// return redirect( request,*

*// AppointmentFormDayJspBean.getURLManageAppointmentFormDays( request, strIdDay ) );*

*// }*

*//*

*// listSlots = AppointmentSlotHome.findByIdDay( nIdDay );*

*// model.put( MARK\_DAY, day );*

*// model.put( MARK\_READ\_ONLY, isReadonly( day.getDate( ) ) );*

*// form = AppointmentFormHome.findByPrimaryKey( day.getIdForm( ) );*

*// }*

*// }*

*//*

*// if ( listSlots == null )*

*// {*

*// return redirect( request, AppointmentFormJspBean.getURLManageAppointmentForms( request ) );*

*// }*

*//*

*// int nDuration = 0;*

*// int nMinStartingTime = 0;*

*// int nMaxEndingTime = 0;*

*// int nMinStartingHour = 0;*

*// int nMinStartingMinute = 0;*

*// int nMaxEndingHour = 0;*

*// int nMaxEndingMinute = 0;*

*//*

*// for ( AppointmentSlot slot : listSlots )*

*// {*

*// int nSlotStartingTime = ( slot.getStartingHour( ) \* 60 ) + slot.getStartingMinute( );*

*// int nSlotEndingTime = ( slot.getEndingHour( ) \* 60 ) + slot.getEndingMinute( );*

*// int nSlotDuration = nSlotEndingTime - nSlotStartingTime;*

*//*

*// if ( nSlotDuration < 0 )*

*// {*

*// nSlotDuration = -1 \* nSlotDuration;*

*// }*

*//*

*// if ( ( nDuration == 0 ) || ( nSlotDuration < nDuration ) )*

*// {*

*// nDuration = nSlotDuration;*

*// }*

*//*

*// if ( ( nMinStartingTime == 0 ) || ( nSlotStartingTime < nMinStartingTime ) )*

*// {*

*// nMinStartingTime = nSlotStartingTime;*

*// nMinStartingHour = slot.getStartingHour( );*

*// nMinStartingMinute = slot.getStartingMinute( );*

*// }*

*//*

*// if ( nSlotEndingTime > nMaxEndingTime )*

*// {*

*// nMaxEndingTime = nSlotEndingTime;*

*// nMaxEndingHour = slot.getEndingHour( );*

*// nMaxEndingMinute = slot.getEndingMinute( );*

*// }*

*// }*

*//*

*// int nNbWeeksToCreate = AppPropertiesService.getPropertyInt( PROPERTY\_NB\_WEEKS\_TO\_CREATE\_FOR\_BO\_MANAGEMENT, 1 );*

*// model.put( MARK\_LIST\_SLOTS, listSlots );*

*// model.put( MARK\_MIN\_SLOT\_DURATION, nDuration );*

*// model.put( MARK\_MIN\_STARTING\_HOUR, nMinStartingHour );*

*// model.put( MARK\_MIN\_STARTING\_MINUTE, nMinStartingMinute );*

*// model.put( MARK\_MAX\_ENDING\_HOUR, nMaxEndingHour );*

*// model.put( MARK\_MAX\_ENDING\_MINUTE, nMaxEndingMinute );*

*// model.put( MARK\_LOCALE, getLocale( ) );*

*// model.put( MARK\_BORN\_DATE, getLimitedDate( nNbWeeks + nNbWeeksToCreate ) );*

*// AppointmentFormJspBean.addElementsToModelForLeftColumn( request, form, getUser( ), getLocale( ), model );*

* Duplication de code.

Exemple : La prise de rendez-vous, que ce soit côté front office ou côté back office, devrait faire appel aux mêmes méthodes. En effet, que ce soit l’utilisateur ou l’administrateur, les méthodes appelées et les contrôles effectués sont similaires. Hors ce n’est pas le cas. Toutes les méthodes côté FO sont regroupées dans un contrôleur nommé AppointmentApp (1816 lignes). Pour le Back Office, celles-ci sont regroupées dans le contrôleur AppointmentJspBean (2966 lignes).

Cette duplication résulte d’une erreur de conception et le code dupliqué expose de graves problèmes de maintenance dont la gravité augmente avec la quantité de code dupliqué.

**Proposition 4c1 :** Ok

Factoriser le code entre le back office et le front office.

**Proposition 4c2 :** Nok

Supprimer la gestion des rendez-vous côté back office. L’administrateur aura toujours accès au formulaire côté front office.

# Serveurs / Infra

## Serveurs

Question ouverte :

Est-ce que l’application peut être déployée sur n’importe quel type de solution ? Tomcat, Weblogic, Websphere, Jboss ?

Quid du mode cluster/distribué ?

## SGBD

Question ouverte :

Avec quel SGBD est-ce que l’application doit être compatible ? MySQL, Sql Server, Oracle, PostgreSql ?

# Conclusion

Après une revue du fonctionnement de l’application, de sa modélisation et de son implémentation, il apparait que le plugin rendez-vous peut être grandement simplifié au niveau fonctionnel. Cela simplifiera par la même occasion le code source. Mais cela ne suffit pas. Il faut repenser le modèle de données, afin de le rendre performant et fiable (utilisation de type timestamp, indexation, ne plus créer les créneaux en masse …).