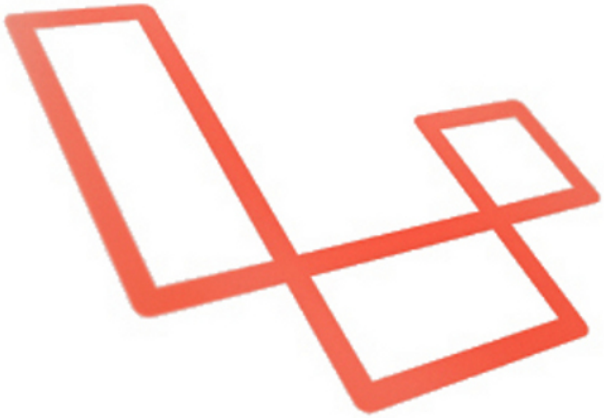
INSA ROUEN
À taille humaine,
à l'échelle du monde



laravel

SOSI

ASI4 2014-2015

Badgeur de cantine

Backoffice

Julien Baron
Thibaud Dauce
Florian Lepetit

Table des matières

Introduction

Le projet présenté consiste à identifier des utilisateurs au travers de tags NFC, le contexte étant celui d'un badgeur de cantine. Nous avons développé la partie backoffice. Cette partie comprend un webservice retournant les noms des utilisateurs identifiés via une requête comprenant une liste de tags. La partie frontoffice étant gérée par une autre équipe.

De ce fait, la documentation a donc été importante, cette dernière est disponible sur Github à l'adresse `https://github.com/ThibaudDauce/SOSILeoCard`. Nous avons également mis en place une démarche de tests unitaires. Enfin, un gestionnaire de versions a été utilisé pour permettre un développement plus rapide.

Une version en ligne est présente à l'adresse suivante : `http://sosi.thibaud-dauce.fr`

Chapitre 1

Choix des technologies

Pour répondre aux besoins du projet, nous avons dû déterminer rapidement vers quels outils nous devions nous tourner. Nos critères étant la facilité de prise en main et la rapidité des développements proposée.

1.1 Un projet web

Notre équipe s'est naturellement tournée vers PHP. Ce choix s'explique principalement par le fait que ce langage nous a semblé accessible (ressemblances au C). Aussi, le fait d'avoir certains membres déjà familiers avec cette technologie nous a finalement motivés.

1.2 Gestion de versions

Afin de mutualiser nos développements, nous avons choisi Git pour simplifier notre travail. Cela nous a permis tout d'abord de travailler de concert sur les mêmes sources, mais aussi de livrer continuellement l'équipe chargée de travailler sur la partie frontoffice.

1.3 Des contraintes de temps

La principale difficulté du projet SOSI est la durée allouée à la réalisation du projet. Nous avons opté pour un framework dans le but de gagner en temps de développement. Laravel s'est présenté comme une solution convenable puisqu'il propose des modules nous étant pratiques tels qu'une gestion facile des routes, un ORM très pratique (Eloquent) ainsi qu'une structure claire.

1.4 Tester notre solution

Nous avons décidé de soumettre notre application à des tests unitaires dans le but d'être garants de sa fiabilité. Laravel étant fortement compatible avec PHPUnit, nous avons opté pour ce framework de tests. Il nous a permis de maîtriser de manière simple et rapide la qualité du code produit.

1.5 Échanges des données

Pour cette partie backoffice de l'application de carte de cantine, nous avions deux contraintes à respecter. La première était d'utiliser un protocole d'échange de données compatible avec Android. La seconde était de choisir un format de données facilement exploitable.

Comme vous allez le voir maintenant, nous avons choisi le couple REST/JSON pour cette tâche.

1.5.1 Le REpresentational State Transfert ou REST

La première chose à préciser est que REST n'est pas un protocole en lui même, mais plutôt une architecture utilisant le protocole HTTP. L'architecture est de donc de type CRUD (Create, Retrieve, Update, Delete), mais nous n'utiliserons que le GET de HTTP, qui correspond à la récupération, à la lecture d'une ou plusieurs entrées en base.

De par son universalité, le HTTP ne pose aucun problème quant à la compatibilité avec la partie Android du frontoffice. Pour accéder aux données, chaque ressource doit avoir une URI (Uniform Ressource Identifier), à laquelle on fait un appel via une des méthodes HTTP en fonction de l'opération CRUD à réaliser. Côté Laravel, cette gestion des routes et méthodes est gérée nativement, ce qui permet une mise en place rapide et efficace de cette architecture.

1.5.2 Le format de données

Mais le choix du protocole et de l'architecture n'est pas suffisant pour définir la communication entre notre serveur et la partie frontoffice. En effet, nous devons aussi choisir sous quel format ces données sont échangées. Plusieurs possibilités se sont offertes à nous.

Le XML

Format connu et le plus utilisé, le XML est intégré par défaut dans de nombreux langages. Il permet d'exploiter les données même si l'on ne connait pas leur structure. Il a en effet vocation à être un langage de présentation.

L'inconvénient cependant est qu'il est plutôt compliqué à générer et transformer en objet.

Le JSON

JSON est le format qui monte en ce moment et qui vient de JavaScript où il est géré nativement. Il n'est pas intégré de base dans la plupart des langages mais dispose de nombreuses implémentations qui permettent de le gérer facilement. Il est plus léger que le XML et permet d'être lisible plus facilement par l'homme.

L'inconvénient, c'est qu'il faut s'entendre sur le format de la ressource car celle-ci n'est pas décrite et structurée explicitement comme dans le xml.

L'application frontend ne requetant que de simples tags NFC au serveur, notre choix c'est porté sur le JSON car les données échangées sont simples (nom, prénom, tag NFC) et que ce format de données est géré par android et Laravel.

Chapitre 2

Utilisation du framework

Comme énoncé dans la première partie, nous avons fait le choix d'utiliser un framework nommé Laravel. Ce framework est assez récent et a pour objectif de faciliter le développement. Son slogan est `"PHP That Doesn't Hurt. Code Happy & Enjoy The Fresh Air."`

2.1 Routes

Dans la partie précédente, nous avons les différentes routes nécessaire à la mise en place d'un CRUD. Laravel implémente cela facilement via une façade nommée tout simplement `Route`.

```
<?php
Route::get('users/batch', 'UsersController@batch');
```

Notre application contient une seule route dans le fichier `app/route.php`, définie comme ci-dessus.

Tout d'abord, nous appelons la méthode `get` parce que dans une optique CRUD, nous cherchons à obtenir des informations. Cette méthode prend en premier paramètre l'URI associé à la route et en deuxième paramètre une chaîne de caractère formatée comme `ControllerObject@method`. Dans notre cas, lorsque l'utilisateur fait une requête GET sur l'URI `users/batch`, Laravel va appeler la méthode `batch` sur l'objet `UsersController`.

Qu'est-ce qu'une façade

Une façade est un classe spécifique qui permet d'appeler des méthodes publiques d'une instance d'un objet via des appels statiques sur la façade.

Par exemple, lorsque j'appelle une méthode statique sur la façade `Route`, Laravel va en réalité appeler une méthode publique d'une instance d'une classe appelée `Illuminate\Routing\Router` stockée dans l'application.

Vous pouvez en savoir plus sur les façades Laravel dans la documentation `http://laravel.com/docs/facades`

2.2 Le modèle

Laravel utilise son propre ORM appelé Eloquent. Ce dernier nécessite la création d'une classe par ressource de l'application. Notre application a donc un modèle nommé `User` situé dans le répertoire

`app/models`. Ce dernier hérite du modèle Eloquent qui fournit toutes les méthodes d'accès à la base de données telles que `where`, `find`, `save`...

Dans notre modèle, nous devons définir plusieurs attributs :
- `$table` : cet attribut permet de définir la table associée au modèle. Même si Eloquent est assez intelligent pour définir le pluriel du nom de l'objet (User => users), pour des raisons de lisibilité, nous l'avons redéfini.
- `$fillable` : afin de répondre au problème de `MassAssigment`, Eloquent demande de fournir un tableau contenant les attributs du modèle qu'il sera possible de sauvegarder directement via la méthode `create` afin d'éviter qu'un utilisateur mal intentionné insère dans la base de données des champs non voulus. (plus d'informations sur `http://laravel.com/docs/eloquent#mass-assignment`)

2.3 Les contrôleurs

Toujours dans une optique CRUD, Laravel nous encourage à créer un contrôleur pour chaque ressource. Notre application ne contient qu'une seule ressource pour le moment appelée `User`, notre contrôleur (par convention) aura donc le nom de `UsersController` et sera situé dans le répertoire `app/controllers`.

Cet objet aura une seule méthode, la méthode `batch` définie dans les routes comme la méthode à appeler.

2.4 La méthode batch

Dans notre unique méthode, nous pouvons définir cinq phases.
- Récupération des données fournies en GET ;
- Test des données et renvoi d'une erreur en cas d'absence d'identifiants ;
- Récupération des données en BDD via un répertoire (repository) ;
- Validation de ces données et information en cas de données manquantes ;
- Retour de l'ensemble du résultat si tout s'est bien passé.

2.5 Utilisation d'un répertoire

2.5.1 Pourquoi utiliser un répertoire

Dans le cas d'une application simple comme la notre, il aurait été parfaitement correct d'appeler les méthodes de l'ORM directement dans le contrôleur comme simplifié ci-dessous :

```php
<?php
public function batch()
{
  return User::whereIn('serial', Input::get('data'));
}
```

Mais dans le cas d'une application plus complexe, il est préférable de dissocier ses contrôleurs de l'ORM afin d'améliorer la maintenabilité de l'application et de permettre l'utilisation d'un autre système de stockage.

2.5.2 Utilisation d'un répertoire dans notre projet

Afin de montrer les possibilités du framework Laravel ainsi que de permettre aux futurs étudiants allant travailler sur l'application de partir sur de bonnes bases, nous avons décidé d'implémenter un répertoire pour les utilisateurs.

Ce repertoire se présente sous la forme d'une classe nommée `DatabaseUserRepository` proposant une seule méthode : `getBySerialBatch`. Cette méthode prend en paramètre un tableau contenant les différents numéros de carte NFC et retourne une collection d'objets `User` via la méthode `whereIn` de l'ORM Eloquent.

Maintenant, afin de découpler nos contrôleurs de l'ORM, nous devons utiliser notre répertoire. Pour cela, Laravel nous propose deux solutions :
— Instancier l'objet normalement dans la méthode ;
— Utiliser l'outil d'injection de dépendance de Laravel.

2.5.3 Outil d'injection de dépendance

Nous n'instançons jamais notre contrôleur, Laravel s'en charge à notre place lorsque la route enregistrée est demandée. Pour instancier une classe, Laravel utilise un outil très puissant de résolution de dépendance grâce à l'introspection de PHP. Avant de créer l'objet nécessaire, elle détermine les classes requises par le constructeur.

Si le constructeur nécessite un objet `DatabaseUserRepository` comme dans notre cas, Laravel va l'instancier et le fournir au constructeur du contrôleur `UsersController`.

```php
<?php
public function __construct(DatabaseUserRepository $userRepository)
{
  $this->userRepository = $userRepository;
}
```

De cette manière, nous avons accès au répertoire dans notre contrôleur sans effort.

2.5.4 "Code to an interface"

La méthode présentée dans le paragraphe précédent présente l'avantage d'être simple à implémenter mais présente aussi un gros défaut : notre contrôleur est toujours lié à l'ORM via l'objet `DatabaseUserRepositor`

En réalité, notre contrôleur n'a pas besoin de savoir quel type de répertoire il utilise, l'unique besoin est une fonction appelée `getBySerialBatch` prenant en paramètre un tableau d'identifiants NFC et retournant une collection d'utilisateurs. Ce qui est décrit ici est le principe même d'une interface. Nous avons donc créé une interface appelée `UserRepository` et fournissant la signature de méthode requise. Bien évidement, notre répertoire `DatabaseUserRepository` doit maintenant implémenter cette interface.

Nous pouvons donc changer le pré-requis du constructeur de notre contrôleur par l'interface `UserRepository`.

```
public function __construct(UserRepository $userRepository)
{
  $this->userRepository = $userRepository;
}
```

Mais que va donc faire Laravel lorsqu'il va rencontrer l'interface. En effet, si le framework cherche à résoudre la dépendance, il va se heurter au problème qu'il est impossible d'instancier une interface. Nous avons donc besoin de définir un lien entre notre interface et notre implémentation.

2.5.5 Service provider

Les `services provider` de Laravel sont de simples objets fournissant de la configuration pour l'application. Ils sont définis dans le fichier `app/config/app.php`.

Dans notre cas, nous avons créé un simple service provider `LeoCardServiceProvider` qui effectue le lien de l'interface vers l'implémentation dans sa méthode `register`.

```php
<?php
public function register()
{
  $this->app->bind('UserRepository', 'DatabaseUserRepository');
}
```

2.5.6 Conclusion

Cette approche par répertoire et par interface nous a permis de créer une application très solide et sans grande dépendance. Par exemple, si nous souhaitons changer d'implémentation, remplacer par exemple la base de données par un système de fichier, il suffit de définir un nouveau repertoire `FileUserRepository` implémentant l'interface `UserRepository` et de compléter les méthodes nécessaires. Ensuite, en changant le lien dans le service provider nous obtenons une nouvelle implémentation sans jamais avoir à ouvrir nos contrôleurs ce qui aurait pu être une source de bugs lors du changement.

La principale limitation de ce système provient du language PHP, qui ne permet pas de spécifier des types de retour. La nouvelle implémentation doit donc se référer à la documentation et suivre à la lettre les instructions de retour spécifiées dans l'interface même si ces dernières ne sont pas obligatoire au niveau du language. Ce problème peut être contourné via l'utilisation de la machine virtuelle `HHVM` de Facebook et du language `Hack`, compatible avec PHP et introduisant un typage statique.