POLYTECH
Relevons les défis de demain

IG

Rapport de projet


# OldLantern, application de e-commerce

Réalisé par
**Bastien RICOEUR**

Sous la direction de
**Anne LAURENT / Arnaud CASTELLTOR**

Année universitaire 2015-2016

# Sommaire

# Tables des figures

# Introduction

Dans le cadre de mes études à Polytech Montpellier, j'ai à réaliser une application web.
Après avoir analysé les contraintes du projet, j'ai estimé qu'un site de e-commerce serait une application en adéquation avec le sujet.

OldLantern est une entreprise (fictive) de fabrication et de vente d'articles de décoration d'intérieur.
Dans nos ateliers, nous créons des produits design et ergonomiques pour notre clientèle.
Grâce à cette nouvelle application web nous pourrons toucher un public plus important. Les clients pourront se connecter pour commander nos articles et être livrés chez eux en moins 72h.

# 1 Cahier des charges

## 1.1 Présentation du sujet et analyse du contexte

OldLantern permet aux utilisateurs de se connecter pour consulter notre catalogue et passer commande. Il va donc falloir gérer les paniers des clients, le nombre d'articles en stock ainsi que le paiement des commandes. Par conséquent, il va falloir que notre application soit sécurisée.

De plus nous devrons gérer plusieurs types d'utilisateurs (les clients et les administrateurs) qui auront des droits différents. En effet, les administrateurs devront avoir la possibilité de régler certains problèmes sur les commandes des clients à tout moment.

## 1.2 Analyse des besoins fonctionnels

Un utilisateur devra dans un premier temps s'inscrire au site pour pouvoir par la suite visualiser notre catalogue et passer des commandes. Une fois l'inscription réalisée, l'utilisateur pourra se connecter au site.

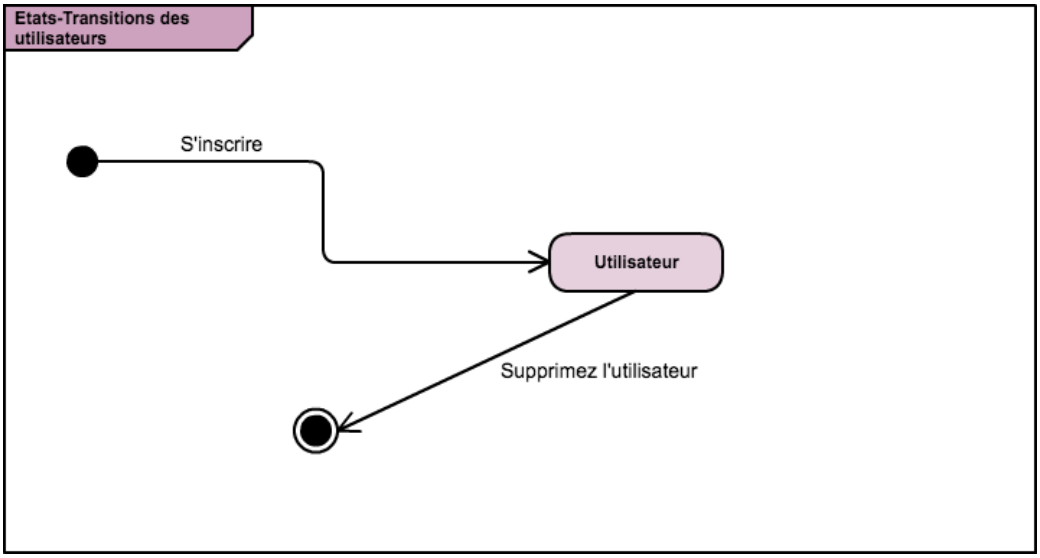


*Illustration 1: Diagramme Etats-Transitions des utilisateurs*

OldLantern offre de nombreuses fonctionnalités aux utilisateurs. Une fois connectés, ils peuvent consulter le catalogue de nos produits, passer commande mais aussi voir et modifier leur profil ainsi que gérer leurs cartes bancaires.

Les administrateurs ont plus de privilèges sur ce site. Ils peuvent supprimer des utilisateurs, modifier le rôle d'un utilisateur (sans pouvoir changer les informations personnelles des utilisateurs) et gérer le stock des artic les ( ajouter de nouveaux articles, les modifier, augmenter leur nombre d'exemplaires).

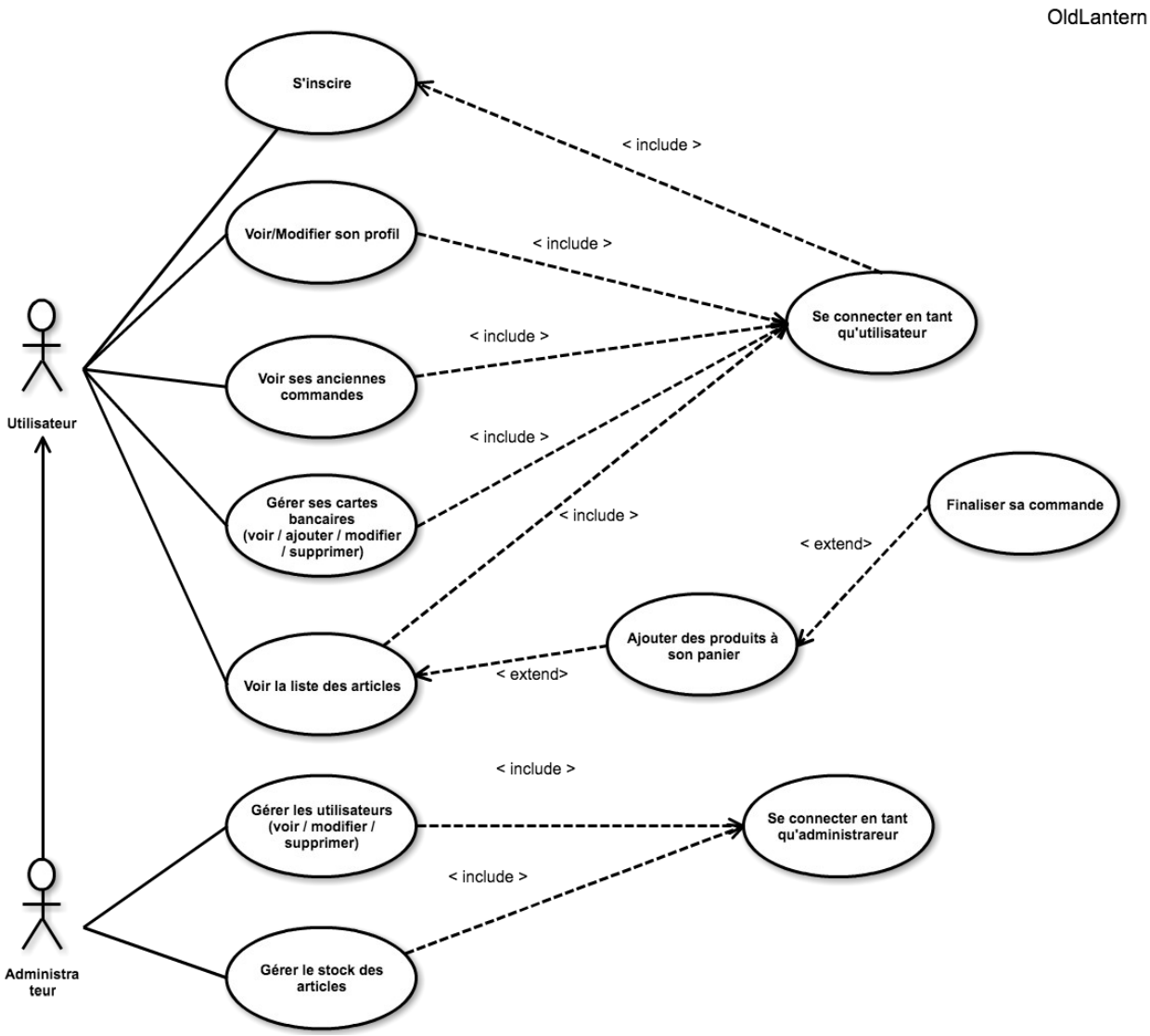


*Illustration 2: Diagramme de cas d'utilisation de OldLantern*

## 1.3 Analyse des besoins non-fonctionnels

### 1.3.1 Spécifications techniques

La nouvelle application web permettra à l'entreprise OldLantern de toucher un plus grand nombre de clients. Et pour ce faire, ce site doit être accessib le depuis les différents navigateurs web disponibles (Chrome, Firefox, Safari et IE 11).

De plus, comme tout site de e-commerce, une problématique très importante est la sécurité. Nous devons protéger au maximum les données des utilisateurs du site pour éviter tous problèmes.

### 1.3.2 Contraintes ergonomiques

Pour répondre au problème d'accessibilit é, nous devons penser cette application de façon à ce qu'elle soit responsive et ainsi être accessible depuis un ordinateur mais aussi depuis une tablette ou un téléphone pour toucher le maximum de futurs clients.

Et enfin, pour offrir une utilisation simple et agréable du site, il est important de développer un site ergonomique et facile d'utilisation.

# 2 Conception

## 2.1 Architecture de l'application et langages utilisés

Tout d'abord, j'avais comme but de développer une application maintenable dans le temps et permettant de futurs développements sans devoir modifier le code excitant. J'ai donc décidé de baser mon architecture générale sur une architecture REST ( *Representational State Transfer* ). Et effet, cette architecture permet de séparer totalement le client du serveur; on pourrait donc par la suite développer une application mobile de OldLantern. Elle communiquerait avec la même API serveur que ce site pour récupérer les données de la base de données. Voici ci-dessous le schéma de l'architecture de mon application qui évoluera au fil du rapport.

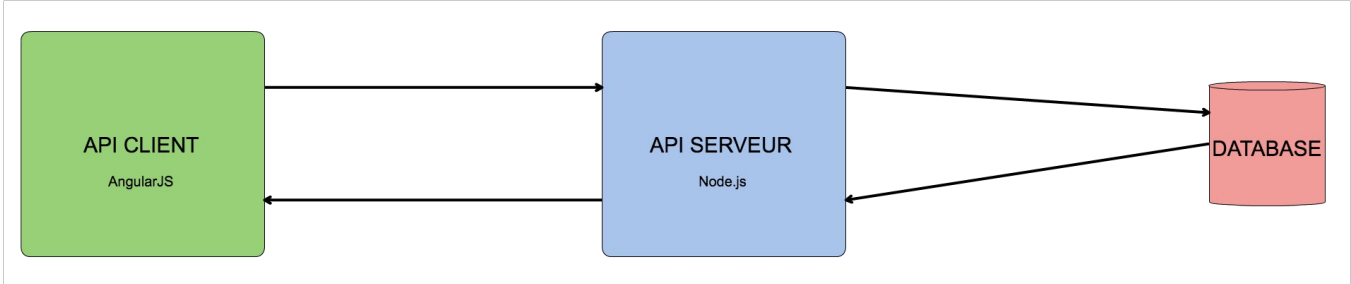


*Illustration 3: Schéma de l'architecture générale de l'application (Étape 1)*

Pour ce projet, j'ai décidé d'utiliser les langages de programmation AngularJS (pour l'API côté client) et Node.js (l'API côté serveur).

J'ai trouvé pertinent d'utiliser AngularJS car ce langage donne la possibilté de standardiser l'application client, ce qui permettra à notre API d'avoir une structure solide et adaptable aux futures demandes. De plus, AngularJS favorise particulièrement la création d'éléments visuels, le résultat étant une navigation fluide et rapide sur le site, ce qui est important pour un site de e-commerce.

Pour l'API du serveur, sachant que potentiellement plusieurs API clients pourraient se connecter à celle-ci, j'ai opté pour Node.js. V8 JavaScript Engine, développé pour Chrome et noyau de Node.js, est très performant. Sa conception asynchrone évite une programmation itérative « bloquante » qui limite les attentes et maximise le traitement rapide et immédiat.

Une autre raison qui m'a amené à utiliser ces langages  est mon désir d'approfondir, de développer mes connaissances en vue de mon futur stage que je réaliserai l'été prochain pour RedFabriq. Je devrai développer une application de gestion pour des

clients comme Mc Donald, l'Oréal et Dassault Aviation. Pour ce projet j'utiliserai AngularJS et Node.js et n'ayant aucune connaissance dans ces langages j'ai trouvé approprié de mettre à profil ce projet scolaire pour me former en vue de mon stage.

## 2.2API client

Le pattern d'AngularJS mis en œuvre est un peu différent du MVC ( *Model View Controller* ) classique, puisqu'il n'y a pas de controller qui s'occupe d'orchestrer les échanges entre les éléments du MVC. On a plutôt un système géré par les événements. On parle alors de MVW pour Whatever parce qu'il n'est finalement pas important de se concentrer sur le composant qui fait le lien. Ce qui est important, c'est le two-way Data Binding basé sur les événements.

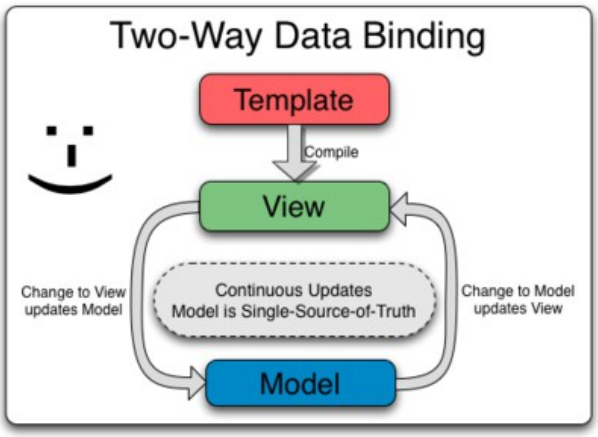


*Illustration 4: Schéma du Two-Way Data Binding*

Pour communiquer avec le serveur, l'API client envoie des requêtes http en GET, POST, PUT et DELETE. Par exemple: Pour récupérer des informations concernant les produits, l'API enverra la requête: GET: http://exemple.com/api/v1/products

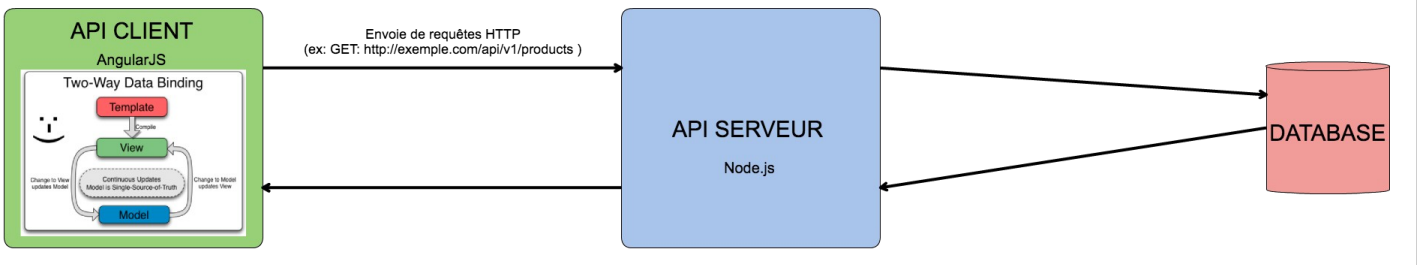


*Illustration 5: Schéma de l'architecture générale de l'application (Étape 2)*

## 2.3API serveur

Pour l'API serveur, développé en Node.js, j'ai décidé d'utiliser Express.js. Il s'agit d'un cadre d'application web Node.js minimal et flexible qui fournit un en semble robuste de fonctionnalités pour développer applications web et mobiles. Il permet de mettre en place des middlewares pour répondre aux requêtes HTTP. Il définit également une table de routage qui est utilisée pour effectuer une action différente en fonction de la méthode HTTP et URL.

J'ai donc configur é les routes de mon API pour répondre aux besoins de l'application. Et j'ai différencié plusieurs catégories de routes:

- les routes accessibles sans être connecté
  (ex : POST http://exemple.com/login)
- les routes accessibles en étant connecté en tant qu'utilisateur
  (ex : GET http://exemple.com/api/v1/products)
- les routes accessibles en étant connecté en tant qu'administrateur
  (ex : POST http://exemple.com/api/v1/admin/product)

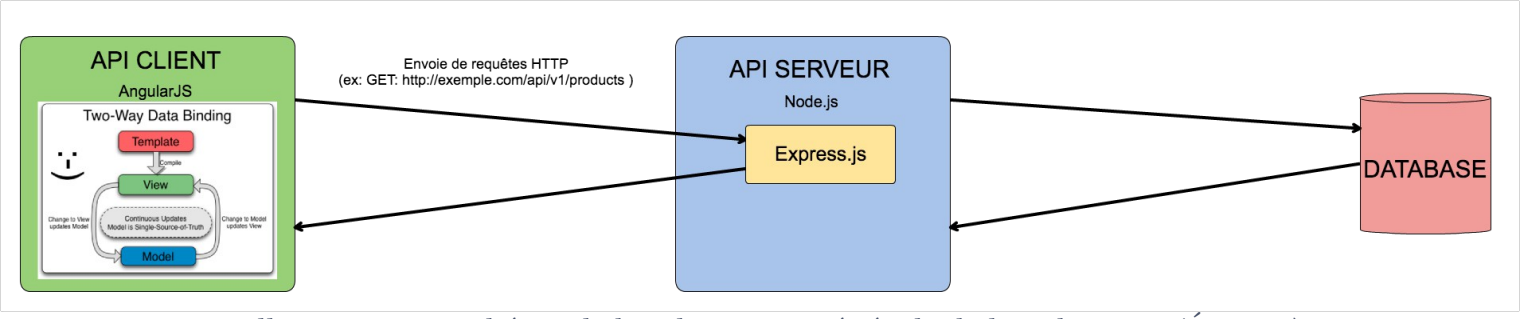


*Illustration 6: Schéma de l'architecture générale de l'application (Étape 3)*

Une fois la requête HTTP traitée, l'API server envoie le résultat de la requête au client avec un statut (200:succès, 400,500:erreurs) et un json. Dans le cas d'une erreur, le json sera composé du statut de la requête et d'un message expliquant l'erreur. En revanche, dans le cas d'un succès, le json comportera les informations propres à la requête envoyé e. Pour respecter les directives permettant de structure r

son API JSON ( cf: http://jsonapi.org/), notre résultat ressemblera à l'exemple du site de référence.

```
"links": {
   "self": "http://example.com/articles",
   "next": "http://example.com/articles?page[offset]=2",
   "last": "http://example.com/articles?page[offset]=10"
},
"data": [{
   "type": "articles",
   "id": "1",
   "attributes": {
      "title": "JSON API paints my bikeshed!"
   },
   "relationships": {
      "author": {
         "links": {
            "self": "http://example.com/articles/1/relationships/author",
            "related": "http://example.com/articles/1/author"
         },
         "data": { "type": "people", "id": "9" }
      },
      "comments": {
         "links": {
            "self": "http://example.com/articles/1/relationships/comments",
            "related": "http://example.com/articles/1/comments"
         },
         "data": [
            { "type": "comments", "id": "5" },
            { "type": "comments", "id": "12" }
         ]
      }
   },
   "links": {
      "self": "http://example.com/articles/1"
   }
}],
"included": [{
   "type": "people",
   "id": "9",
   "attributes": {
      "first-name": "Dan",
      "last-name": "Gebhardt",
      "twitter": "dgeb"
   },
   "links": {
      "self": "http://example.com/people/9"
   }
}, {
   "type": "comments",
   "id": "5",
   "attributes": {
      "body": "First!"
   },
   "relationships": {
      "author": {
         "data": { "type": "people", "id": "2" }
```

```
        }
      },
      "links": {
        "self": "http://example.com/comments/5"
      }
    }, {
      "type": "comments",
      "id": "12",
      "attributes": {
        "body": "I like XML better"
      },
      "relationships": {
        "author": {
          "data": { "type": "people", "id": "9" }
        }
      },
      "links": {
        "self": "http://example.com/comments/12"
      }
    }]
}
```


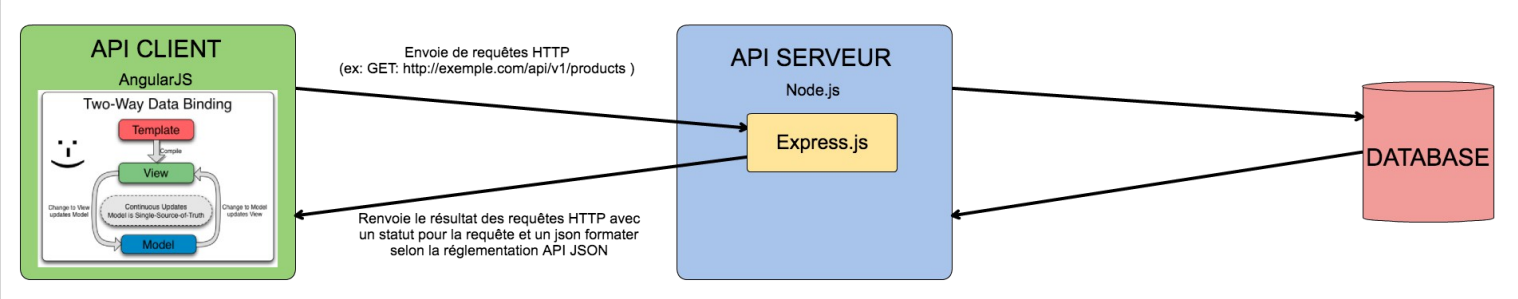


*Illustration 7: Schéma de l'architecture générale de l'application (Étape 4)*

Pour répondre aux demandes des API client, mon API serveur doit pouvoir se connecter à une base de données. J'ai choisi d'utiliser une base PostgreSql pour l'unique raison que c'est le langage SQL le plus compatible avec Node.js. Il a fallu par la suite installer le module Node.js '*pg*', qui nous fait la liaison entre l'API server (Express.js) et la base de données.

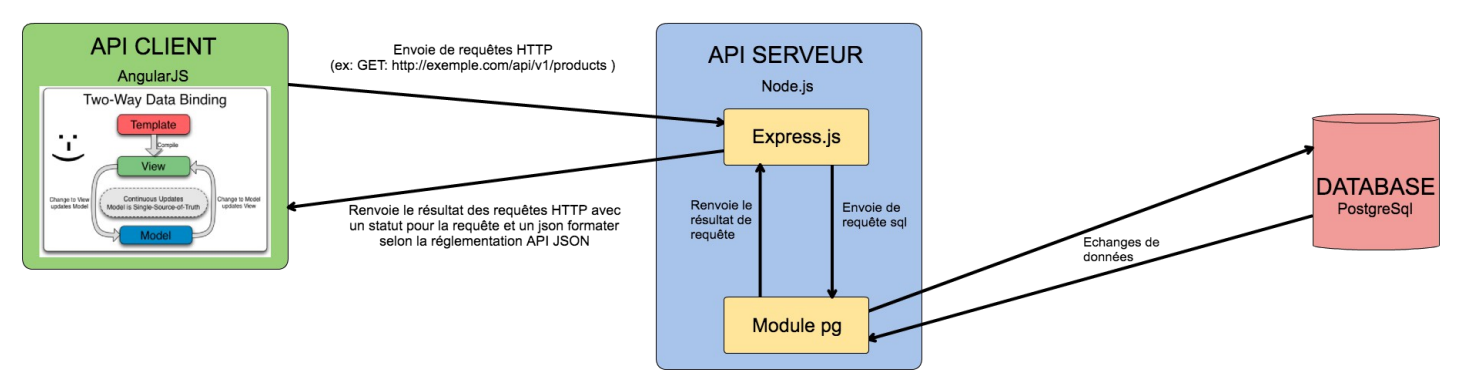


*Illustration 8 : Schéma de l'architecture générale de l'application (Étape 5)*

## 2.4Base de données

Suite à des recherches, j'ai identifié deux bases de données compatibles et performantes avec Node.js : PostgreSql et MySql. Et j'ai choisi d'utiliser le PostgreSql.

Grâce à l'étude des besoins de cette application, j'ai pu en déduire un schéma de base de données pour modéliser les relations entre les différents objets de mon application.
Tout d'abord, nous devons sauvegarder les utilisateurs avec leurs informations (comme leur prénom, nom, adresse, téléphone) et pour chaque utilisateur nous devons savoir s'il s'agit d'un administrateur ou d'un simple client.
Ensuite, dans le cadre de mon application, j'ai modélisé les articles de mon site dans une table avec un type d'articles pour chaque produit et les informations de l'article comme son prix.
Les utilisateurs pourront ainsi commander des articles en choisisant la quantité désirée. Une fois la commande terminée, le client devra la régler ce qui créera une facture propre à chaque commande.
Les utilisateurs devront enregistrer une (ou plusieurs) carte bancaire pour pouvoir payer ensuite leur commande. Je me suis renseigné sur le module de paiement par carte bancaire : Il faut créer une entreprise déclarée et répondre à plusieurs critères pour pouvoir l'utiliser sur son site. J'ai donc décidé de modéliser les cartes bancaires

des clients avec une table que j'ai créée. J'ai conscience que cela va générer un problème de sécurité, mais ces tables pourrraient être remplacées par un module de paiement si l'application est commercialisée. Et c'est pour cette raison que je n'approfondirai pas le développement des fonctionnalinités concernant les cartes bancaires.

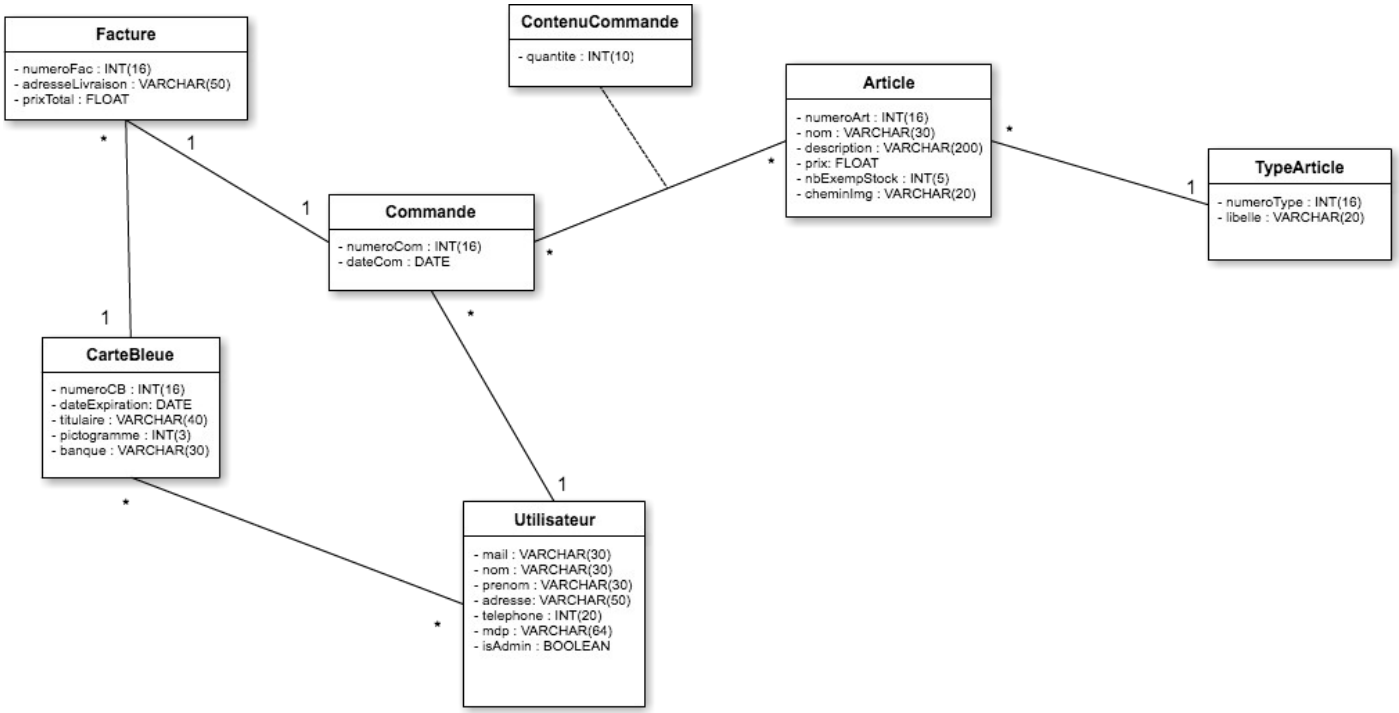


*Illustration 9: Diagramme de classe de l'application OldLantern*

Après avoir réalisé ce diagramme, j'en ai déduit le schéma relationnel suivant :

- Facture(numeroFac, adresseLivraison, prixTotal, #numeroCB, #numeroCom, #code)
- ContenuCommande(#numeroCom, #numeroArt, quantite)
- Commande(numeroCom, dateCom, #mail)
- Article(numeroArt, nom, description, prix, nbExempStock, cheminImg, #typeArticle)
- TypeArticle(numeroType, libelle)
- ProprietaireCarte(#numeroCB, #mail)

- CarteBleue(numeroCB, dateExpiration, titulaire, pictogramme, banque)
- Utilisateur(mail, nom, prenom, adresse, telephone, mdr, isAdmin)

J'ai ensuite développé un script pour la création de mes tables en PostgreSql.
Et pour finir, j'ai rajouté les triggers suivants:

- Quand un utilisateur ajoute un nouvel article à sa commande (et à la modification également), la quantité de l'article doit être supérieure à 0 sinon le trigger renvoie une erreur. De plus, si la quantité demandée par l'utilisateur est supérieure au nombre d'exemplaires du produit en stock, le trigger renvoie également une erreur.
- Quand un utilisateur commande un article, le nombre d'exemplaires en stock de cet article diminue en fonction de la quantité demandée par le client.
- À la création et la modification d'une facture, un trigger calcule le prix total de la commande en fonction du prix et de la quantité des article commandés.

## 2.5 Design de l'application

Pour le design de mon application, j'ai décidé de le concevoir en MobileFirst. Je suis donc parti du design mobile jusqu'au design sur ordinateur. Maitrisant bien Bootstrap, j'ai décidé d'utiliser seulement de l'HTML5 et CSS3 pur et du jQuery.

Pour réaliser une application responsive, j'ai utilisé des media queries CSS pour adapter le contenu de ma page à la taille de l'écran.

## 2.6 Déploiement de l'application

Lors du déploiement de mon application sur Internet, j'ai voulu mettre en avant le fait que mon API client et serveur sont vraiment indépendantes l'une de l'autre et peuvent donc être déployés à des endroits différents.

J'ai décidé d'héberger mon API serveur et ma base de données sur https://heroku.com. Je me suis renseigné et j'ai découvert que Heroku hébergeait nos

bases de données sur http://aws.amazon.com pour offrir un service performant et gratuit pour héberger les bases de données PostgreSql. De plus, le déploiement de l'API serveur en Node.js sur Heroku est très simple en passant par git. Et j'ai séparé mon API client du serveur en la déployant sur mon Raspberry.

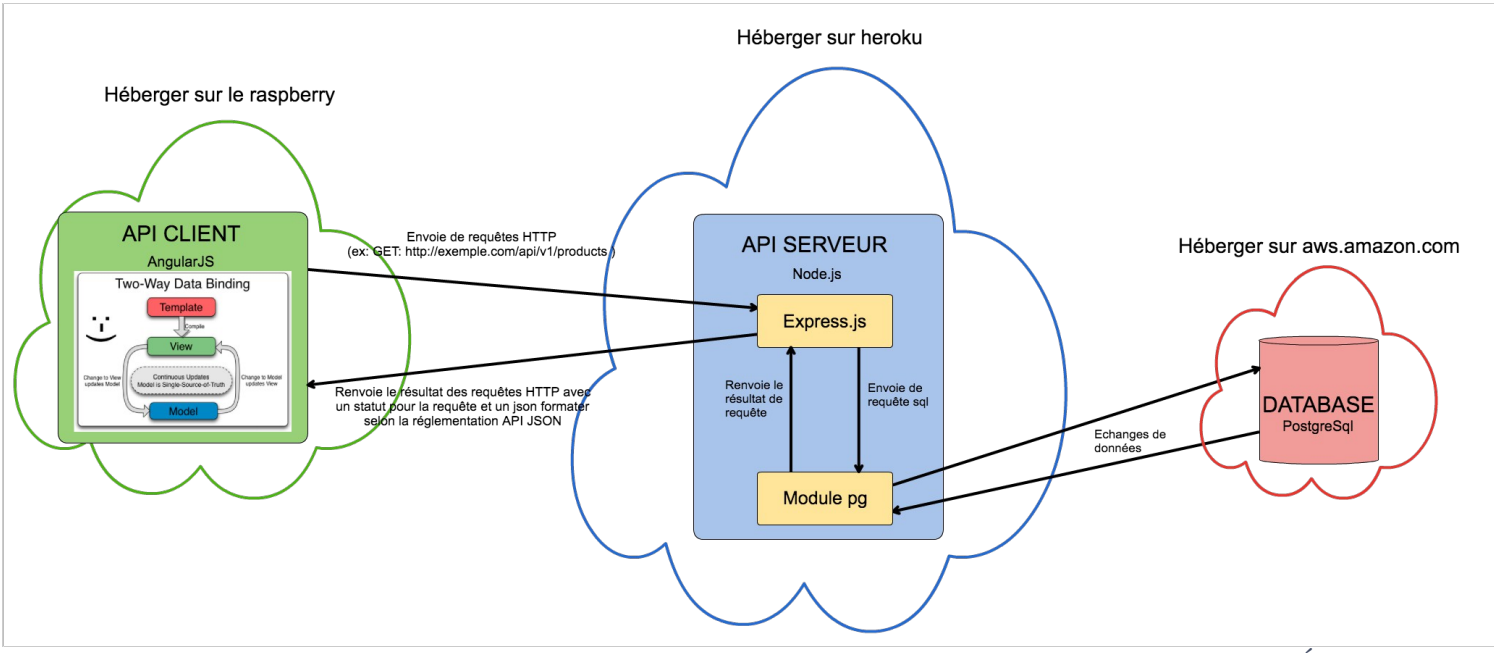


*Illustration 10: Schéma de l'architecture générale et de déploiement de l'application (Étape 6)*

# 3 Manuel d'utilisation

Tout d'abord, voici l'adresse pour accéder à mon application :

***http://109.30.180.96:2771* (API client)**

***https://oldlantern.herokuapp.com* (API server)**

Une fois qu'on a accédé à l'API client depuis un navigateur , on arrive sur le page d'accueil. Depuis celle-ci on peut soit se connecter, soit s'inscrire. Une fois connecté en tant qu'utilisateur, on accède au catalogue des produits proposés par OldLantern. On peut ensuite consulter la fiche technique d'un produit en cliquant sur le bouton « Détails » d'un des produits. Arrivé sur la page de détail du produit, on peut acheter ce produit en précisant au préalable la quantité souhaitée. On sera ensuite rediriger vers la page « Mon panier » pour régler sa commande (on peut retourner sur le page « Catalogue » pour continuer ses achats). Pour cela il faut au préalable avoir ajouter au moins une carte bancaire (Rendez-vous dans la rubrique « Mon profil » puis cliquez sur le bouton « Gérer mes cartes bleues»). Une fois la carte bancaire sélectionnée pour la commande, on peut finaliser notre commande en renseignant le champs pour l'adresse de livraison, on sera alors redirigé vers une page de récapitulatif de la commande.
En passant par le menu de gauche, on peut accéder à la rubrique « Mon profil » pour modifier les informations de notre compte ou réaliser d'autres actions (comme la gestion des cartes bancaires).

En étant connecté en tant qu'administrateur, on a accès à d'autres fonctionnalités alors qu'un simple client n'a pas les droits. En se rendant dans le menu de gauche, on peut accéder à la rubrique « Liste des membres » où l'administrateur peut gérer les membres du site, en les supprimant ou en modifiant leur rôle. L'administrateur peut également ajouter ou modifier les produits du site depuis la page de détail d'un produit.

# 4 Rapport d'activité

## 4.1 Gestion du projet

Tout au long du projet, j'ai opté pour la méthode de gestion Agile qui m'a permis de développer en cycles itératifs. En effet, je commençais par une phase de conception. Je relisais le cahier des charges puis j'en tirais les informations clés et j'en déduisais des fonctions et un design appropriés.

Je suis passé par la suite au développement des fonctionnalités tout en gardant toujours à l'esprit la conception faite précédemment pour répondre au mieux aux attentes du site.

Et enfin, j'ai fini par une phase de tests où je pouvais me rendre compte des problèmes qui existaient au niveau de mon code et je les corrigeais ensuite pour rendre l'application fonctionnelle.

Cette méthode de travail m'a permis de tester mon application au fur et à mesure du développement. J'ai pu ainsi régler au fur et à mesure les problèmes et par conséquent à chaque nouvelle phase de développement partir d'un code propre, fonctionnel, testé et sans erreur.

Une fois ces tests réalisés, j'ai déployé mes API sur les différents serveurs et j'étais ainsi sûr d'avoir une partie de mon projet déployée sur le web et qui fonctionnait.

## 4.2Outils utilisés

Pour la gestion de mon projet, je me suis servi de Trello qui met en application le concept de « tableau blanc » issu de la méthode Agile.

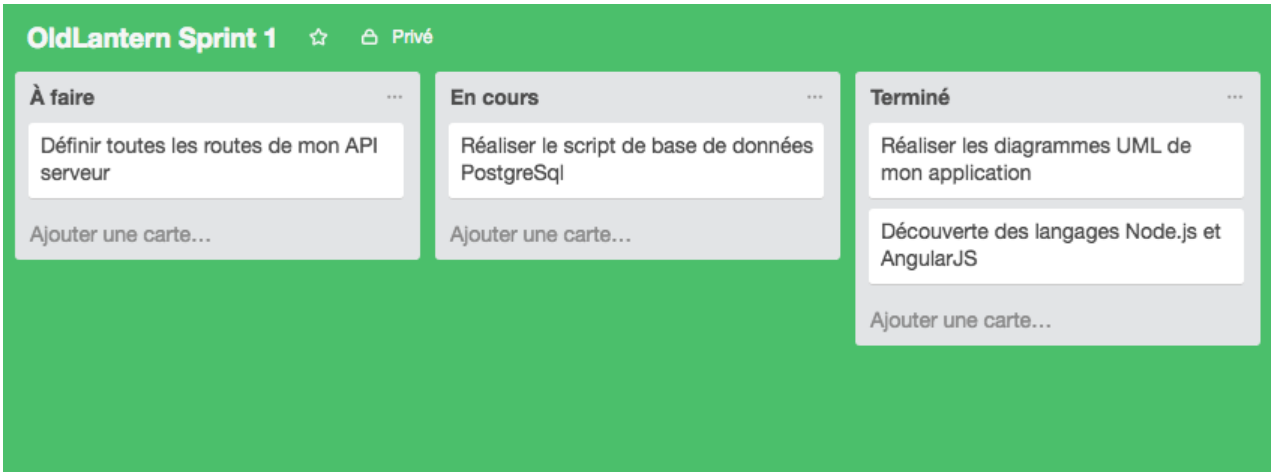


*Illustration 11: Capture d'écran du sprint 1 du projet sur Trello*

J'ai réalisé des sprints de 3 jours, à la fin desquels je devais avoir terminé la liste de tâches que je m'étais fixée au début. J'ai ainsi pu gérer mon temps de travail pour ce projet.

J'ai également utilisé l'outil Postman pour tester mon API serveur. Ce logiciel permet de simuler des GET, POST, PUT.., j'ai ainsi pu tester les cas où mon application devait me renvoyer des erreurs et le cas où mon application devait me renvoyer le résultat de la requête HTTP.

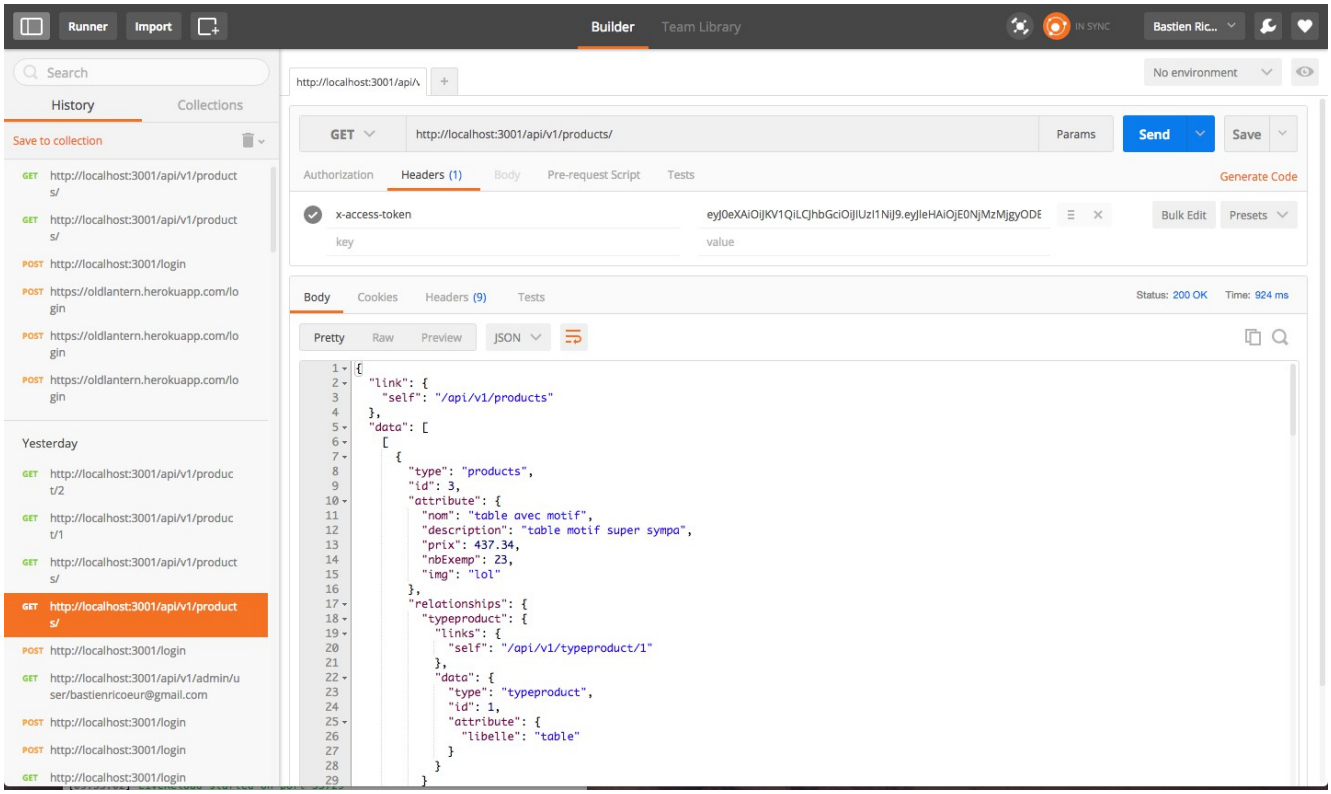


*Illustration 12: Capture d'écran de l'outil Postman*

## 4.3Perspectives d'évolution

Après avoir terminé mon projet, j'ai réalisé une autoévaluation de mon travail afin d'identifier les points à améliorer pour le futur et ceux qui n'auront pas besoin d'être modifiés.

Ma gestion de projet en cycles itératifs m'a permis d'apporter au fil du temps de la plus value à mon travail et d'être certain d'avoir un projet fonctionnel.

Je trouve également mes choix de conception  justes et adaptés au projet. Cependant si le projet était en refaire, il y a certains choses que je modifierai.
En effet, côté client, je n'ai pas mis en pratique le concept de directives d'AngularJS par faute de temps alors que cela aurait tout à fait approprié. Et côté serveur, j'aurais dû réaliser des fonctions plus génériques pour éviter de réécrire à chaque fois certaines parties de code.

# Conclusion

OldLantern possède désormais une application web de e-commerce pour pouvoir se faire connaître sur internet et ainsi  augmenter son nombre de clients potentiels.

Grâce à ce travail, j'ai pu acquérir de nouvelles connaissances qui me serviront pour mon futur stage.

Ma gestion de projet m'a permis de mener à terme ce projet en respectant les délais.