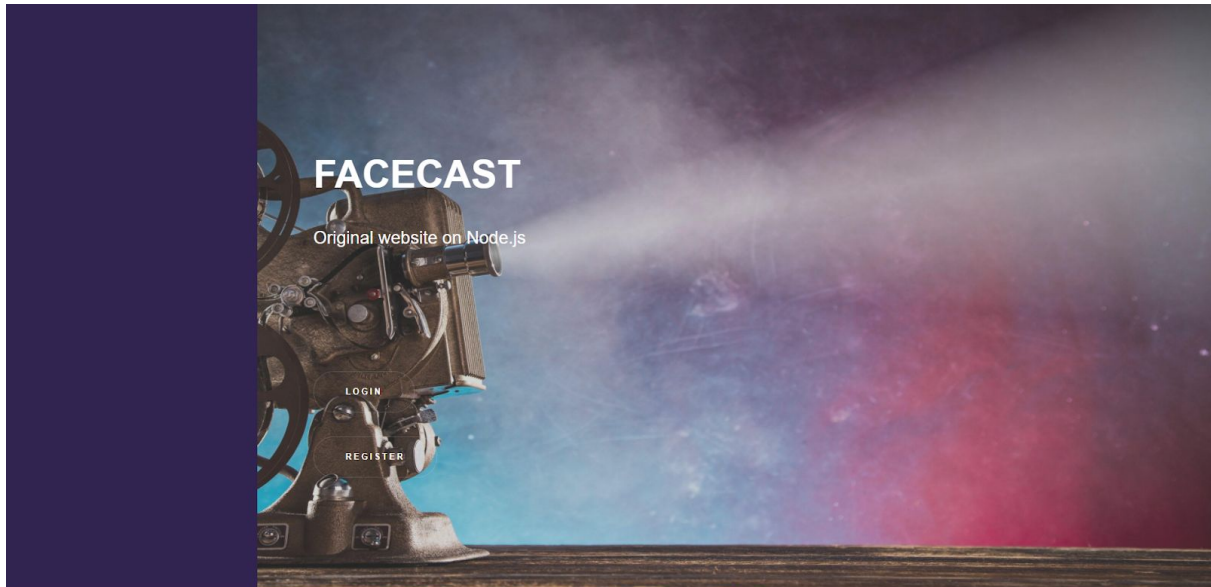


# Documentation de l'application intranet : 'faceCast'



## Situation professionnelle – 'faceCast'

Développement d'une application web basé sur un serveur Node.js, le framework Express.js ainsi qu'une base de données MongoDB via l'approche par Model de Mongoose

**Mai-Juin 2017 / Janvier-Février 2018**

Crété Jonathan – Adelaide Donovan

# Sommaire

- 1. Conception de l'application**
  - a. Contexte
  - b. Cahier des charges
  - c. Analyses préliminaires
- 2. Mise en oeuvre de la mission (développement et tests)**
  - a. Structure de l'application
  - b. Extrait d'implémentation
  - c. API REST
  - d. Contrôle des formulaires
  - e. Tests unitaires
- 3. Mise en production (?)**

# 1. Conception de l'application

## a. Contexte

### **PRÉSENTATION:**

L'agence cliente FaceCast, demandeuse de l'application, a besoin d'une application web afin de traiter au mieux ses candidatures. Le projet se découpe en 2 applications (web & Android) , l'application android permettra au figurant de postuler via l'application dont les données seront envoyées à l'application web nous intervenons alors dans la conception de l'application destinée au « back Office » à savoir l'application web.

Cette application étant destinée au gestionnaire de l'agence, celui-ci doit pouvoir ajouter une nouvelle offre ainsi que gérer en temps réel le traitement des candidatures (différents états).

### **CONTEXTE:**

L'application Web du projet s'appuie sur un serveur Node.js, le framework Express.js et une base de données MongoDB via l'approche par Model de Mongoose.

L'application mobile dialogue avec l'application via le protocole HTTP et des messages au format JSON. De fait, nous avons développés une API basée sur REST pour les échanges entre l'application Android (développée par une autre entreprise) et l'application Node JS. Afin de tester le bon comportement de notre application web vis à vis de l'application mobile, nous avons programmés des tests fonctionnels avec java et Unirest . Cette solution nous as permis de programmer des requêtes JSON (envoi) et analyser les réponses JSON (réception) de notre API.

## b. Cahier des charges

### Besoins essentiels & contraintes :

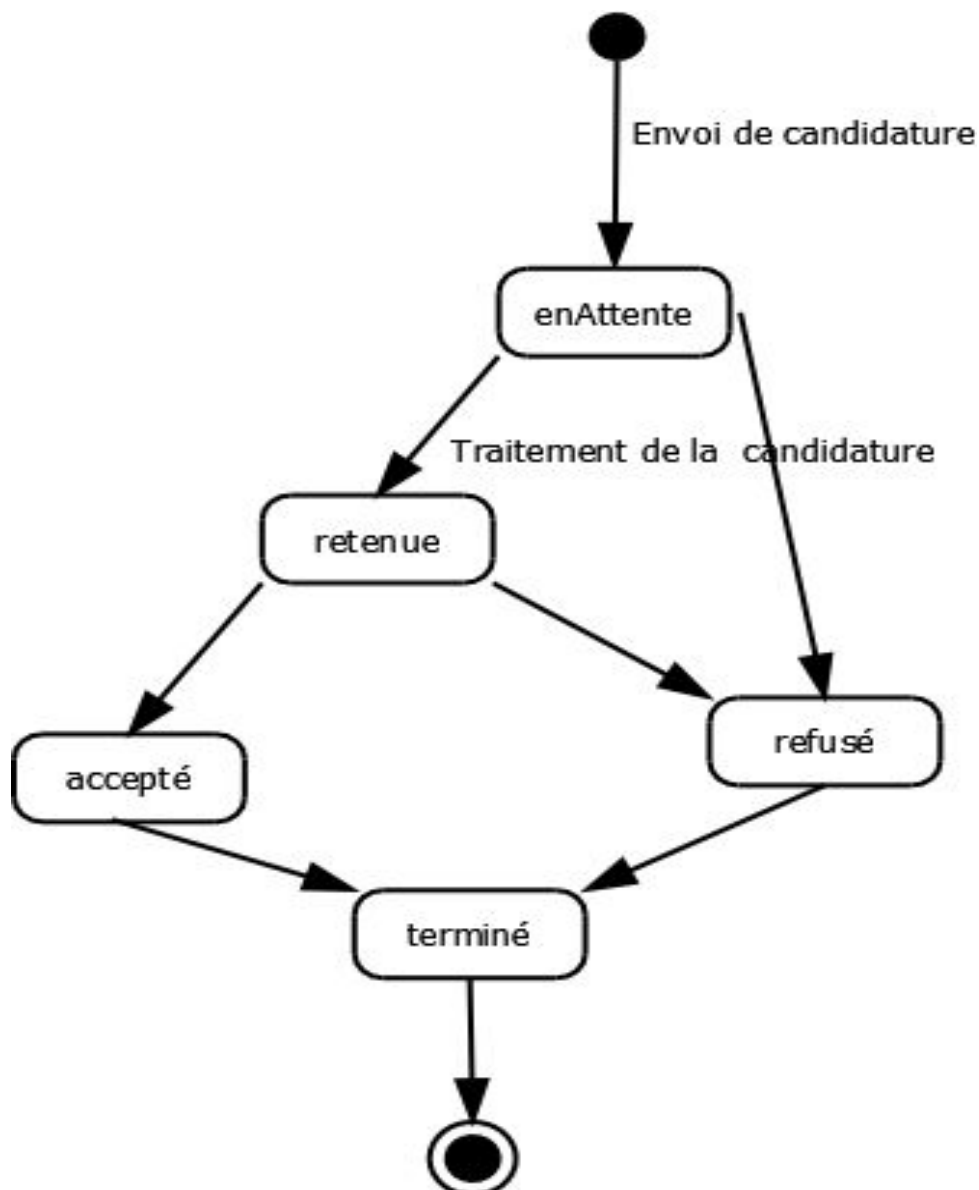
- ❑ Cette application étant destinée au gestionnaire de l'agence, celui-ci doit pouvoir ajouter une nouvelle offre ainsi que gérer en temps réel le traitement.
- ❑ L'application doit être responsive
- ❑ L'application doit s'appuyer sur un serveur Node.js, le Framework Express, et une base de données MongoDB via l'approche par Model de Mongoose.
- ❑ L'application mobile dialogue avec l'application via le protocole HTTP et des messages au format JSON.
- ❑ Afin de tester le bon comportement de notre application web, le développement de tests fonctionnels avec java et Unirest est nécessaire.

### ❑ Besoins secondaires (évolutifs) :

- ❑ L'application pourra être destinée à plusieurs utilisateurs de l'agence : Authentification utilisateurs.
- ❑ L'application pourra être accessible à distance (hébergement).

## c. Analyses préliminaires

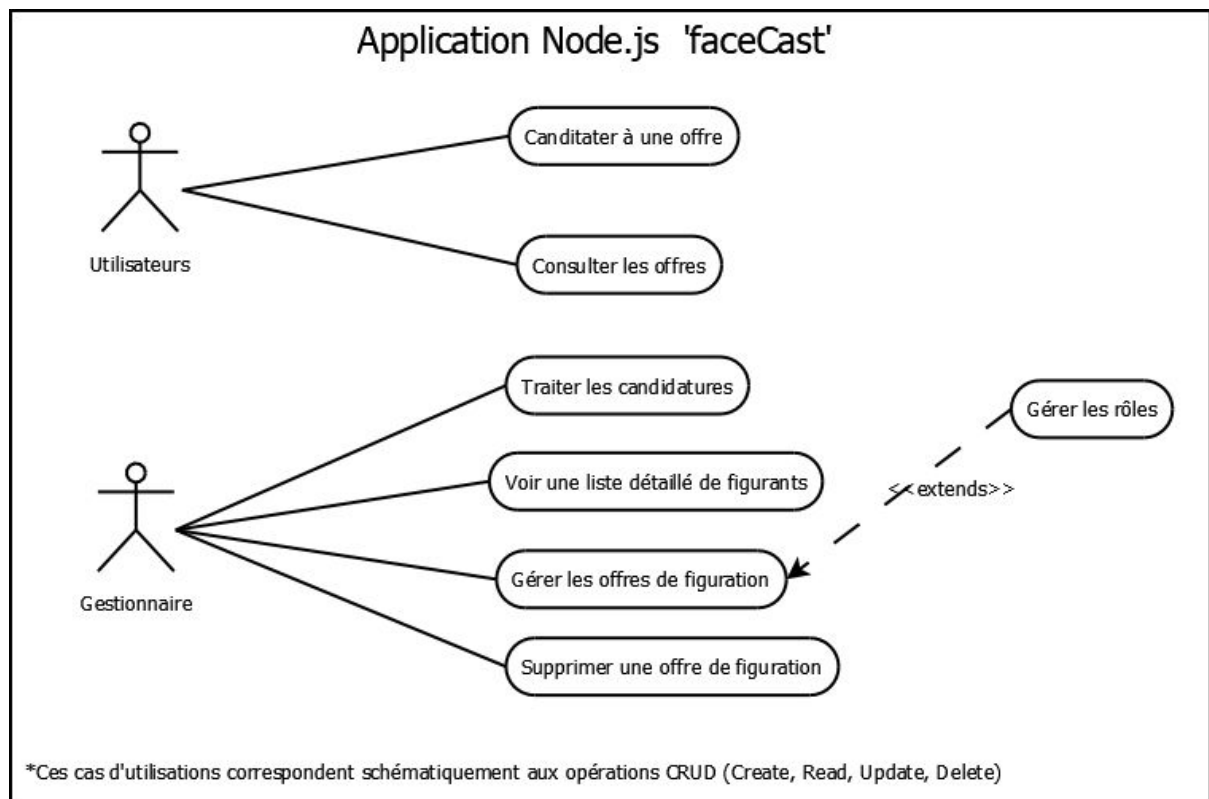
### Diagramme UML E/T du cycle de vie de FaceCast



Le diagramme de cycle de vie désigne toutes les étapes du développement d'une application, de sa conception à sa disparition.

L'objectif d'un tel découpage est de permettre de définir les différentes étapes de l'application qui sont en adéquation avec les besoins exprimés, et l'adéquation des méthodes mises en œuvre.

# Diagramme de cas d'utilisation

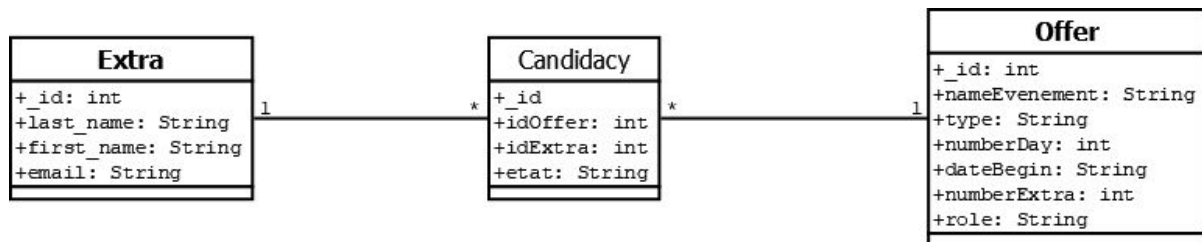


Les diagrammes de cas d'utilisation sont des diagrammes UML utilisés pour donner une vision globale du comportement fonctionnel d'une application.

Un cas d'utilisation représente une unité discrète d'interaction entre un utilisateur (humain ou machine) et un système.

Notre diagramme d'utilisation nous permet d'avoir une approche dite "utilisateur" afin de distinguer les différent rôle et cas d'utilisation lors de la conception de l'application web.

## Diagramme UML des entités du modèle (diagramme de classes métier)



Ce diagramme représente les entités (du modèle) manipulées par les utilisateurs.

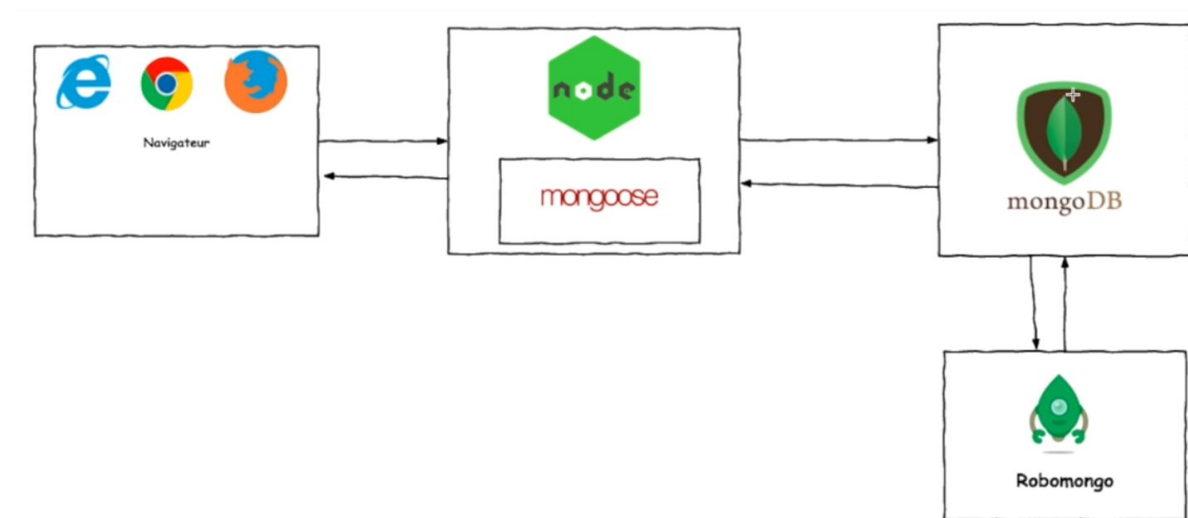
Dans la phase de conception, il représente la structure objet d'un développement orienté objet.

D'où l'utilisation de l'ORM mongoose.

## 4. Mise en oeuvre de la mission (développement et tests)

### a. Structure de l'application

Pour rappel, Node.js est un environnement permettant d'exécuter du code JavaScript hors d'un navigateur. Son architecture est modulaire et événementielle. Il est fortement orienté réseau en possédant pour les principaux systèmes d'exploitation (Unix/Linux, Windows...) de nombreux modules réseau (DNS, HTTP, TCP, TLS/SSL, UDP...). De ce fait, il remplace avantageusement, dans le cadre qui nous intéresse ici un serveur web tel qu'Apache.



L'application mobile dialogue alors avec l'application Node.js via le protocole HTTP. Le serveur Node.js va alors nous renvoyer des données. S'agissant d'un environnement JavaScript, ces données seront naturellement formatées en JSON : ce format est celui de la sérialisation des objets JavaScript (il est aussi utilisé dans le fichier package.json).

Pour assurer ceci, nous utiliserons une API basé sur REST (Representational State Transfer) pour les échanges entre l'application Android et l'application Node.js. Nous allons alors invoquer des routes REST en utilisant le module express qui est un micro-Framework. D'autre part, La gestion de routes REST permet d'associer des requêtes HTTP (dont la request path est exprimée dans une syntaxe orientée donnée) à une action déterminée par un contrôleur (et optionnellement par la méthode http utilisée).



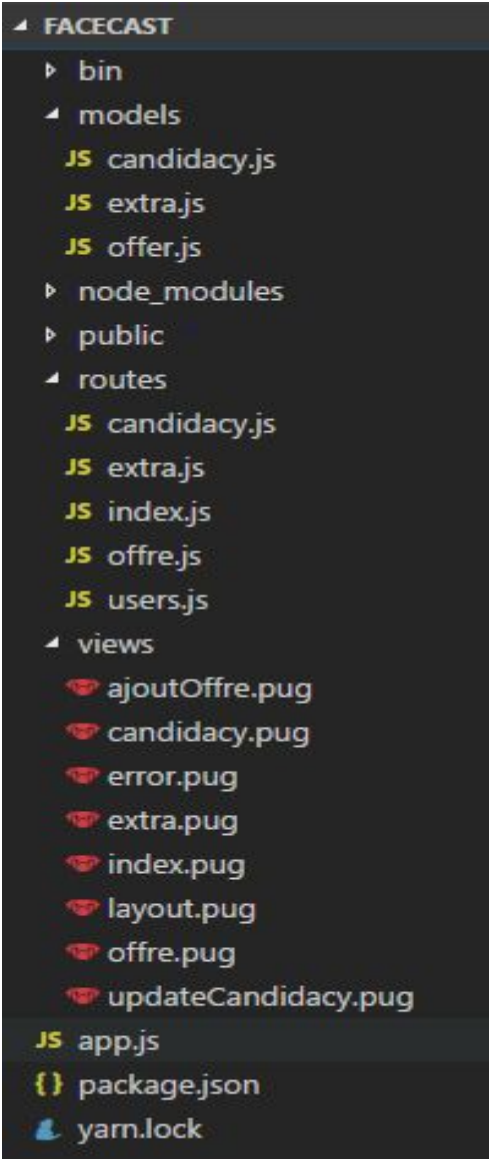
Le logiciel Robo3T sera utilisé afin de d'avoir une interface graphique pour la base de donnée NoSql.

**Une architecture MVC (modèle/Vue/Contrôleur) sera utilisée :**

- ❑ Le modèle est un objet "métier", regroupant des données et des méthodes.
- ❑ La vue est chargée de restituer le modèle au sein d'une interface graphique et de permettre à l'utilisateur d'interagir avec le modèle.
- ❑ Le contrôleur définit les règles de navigation. Le passage d'une vue à l'autre se fait au moyen d'actions conduites par le contrôleur.

URL	Méthode HTTP	Format échangé	Action résultante	Cas d'utilisation
/	GET	Json	Index	X
/offer/	GET	Json	Retourne la liste des offres	Consulter les offres
/offer/delete/:id	GET	Json	Suppression d'une offre selon son identifiant	Suppression d'une offre
/offer/ajoutOffre	POST	Json	Ajout d'une offre	Traitement d'ajout d'une nouvelle offre
/offer/formAjoutOffre	GET	Json	Retourne la vue du formulaire d'ajout d'offre	Ajout d'une nouvelle offre
/extra	GET	Json	Retourne la liste des figurants	Voir une liste détaillée de figurants
/candidacy/formUpdate/:id/:etat	GET	Json	Retourne le formulaire de mise à jour d'une candidature avec l'id et l'etat passer en paramètre	Mise à jour d'une candidature
/candidacy	GET	Json	Retourne la liste des candidatures	Traiter les candidatures
/candidacy/update	GET	Json	Modification de l'état d'une candidature selon son identifiant	Mise à jour d'une candidature
/candidacy/delete/:id	GET	Json	Suppression d'une candidature selon son identifiant	Suppression d'une candidature

## ARBORESCENCE DE L'APPLICATION

 <pre>FACECAST ├── bin ├── models │   ├── candidacy.js │   ├── extra.js │   └── offer.js ├── node_modules ├── public ├── routes │   ├── candidacy.js │   ├── extra.js │   ├── index.js │   ├── offre.js │   └── users.js ├── views │   ├── ajoutOffre.pug │   ├── candidacy.pug │   ├── error.pug │   ├── extra.pug │   ├── index.pug │   ├── layout.pug │   ├── offre.pug │   └── updateCandidacy.pug ├── app.js ├── package.json └── yarn.lock</pre>	<p><b>/bin</b> dossier contenant le serveur node.js</p> <p><b>/node_modules</b> dossier des modules</p> <p><b>/public</b> dossier des éléments accessibles depuis le navigateur</p> <p><b>public/images</b> dossier contenant les images</p> <p><b>public/javascripts</b> dossier contenant les fichiers JavaScript</p> <p><b>public/stylesheets</b> dossier contenant les fichiers CSS</p> <p><b>/routes</b> dossier contenant les contrôleurs</p> <p><b>/views</b> dossier contenant les vues app.js moteur et point d'entrée de l'application</p> <p><b>package.json</b> fichier descriptif contenant les dépendances nécessaires à l'application</p> <p><b>yarn.lock</b> fichier réservé de Yarn</p>
--	--



## b. Extrait d'implémentation

L'application, réalisée avec le Framework express de Node.js(déjà évoqué), utilise le module Mongoose ORM (Object-Relational Mapping, qui est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle).

FaceCast utilise le SGBD NoSql (système de gestion de base de données) MongoDB.

Vous trouverez ci-dessous l'exemple d'un de nos modèles utilisé pour notre application web en l'occurrence le modèle candidacy (candidature), le modèle est un objet (candidacy) lié à notre collection candidacy dans notre base mongodb.

Il nous permet d'exploiter les données de notre base via l'orm mongoose.

JS candidacy.js ✕

```
1  var mongoose = require('mongoose');
2  var Schema = mongoose.Schema;
3
4  var candidacySchema = Schema ({
5    etat : {type : String , default: 'en attente'},
6    extra :{type:Schema.Types.ObjectId, ref : 'extras'},
7    offer :{type:Schema.Types.ObjectId, ref : 'offers'}
8  });
9
10
11  var Candidacy = mongoose.model('Candidacy', candidacySchema,'candidacys');
12
13
14  // export le modèle
15  module.exports = Candidacy;
16
```



Le contrôleur définit les règles de navigation. Le passage d'une vue à l'autre se fait au moyen d'actions conduites par le contrôleur.

Notre applications web contient différents contrôleur qui retourne une réponse selon les actions effectuées par l'utilisateur.

```
var express = require('express');
var router = express.Router();
var Candidacy = require('../models/candidacy');

// Modification de l'état d'une candidature selon son id passer par la méthode GET
router.get('/update/', function(req, res, next) {
  Candidacy.findByIdAndUpdate(req.query.id, {etat : req.query.etat},
    (err, candidacy) => {
      if (err) return res.status(500).send(err);
      res.redirect('/candidacy')
    }
  )
});
```

Ci dessus voici un extrait d'une méthode permettant d'effectuer une des opération de base CRUD (CREATE/READ/UPDATE/DELETE) en l'occurrence l'opération UPDATE.

Notre méthode récupère l'id d'une candidature passé à l'URL par la méthode HTTP GET afin de modifier son état. Ensuite, on redirige l'utilisateur vers une autre méthode qui lui, le renvoie vers une vue pug.

les *routes* sont les différentes URL auxquelles notre application doit répondre.

Ci-dessous un extrait des variable correspondant aux contrôleur à son importation et à son association à une URL dans notre moteur et point d'entrée de l'application (fichier app.js)

```
var offre = require('./routes/offre');
var extra = require('./routes/extra');
var candidacy = require('./routes/candidacy');
```

```
app.use('/', index);
app.use('/users', users);
app.use('/offre', offre);
app.use('/extra', extra);
app.use('/candidacy', candidacy);
```

Lors de la conception de notre application web nous avons effectués une jointure entre le modèle candidature/offre et le modèle candidature/figurant afin de lier:

- Une offre pouvant être référencée par une ou plusieurs candidatures
- Un figurant pouvant être référencé par une ou plusieurs candidatures.

```
router.get('/', function(req, res, next) {
  Candidacy.aggregate([
    {
      $lookup: {
        from: "offers",
        localField: "idOffer",
        foreignField: "_id",
        as: "offer"
      }
    },
    {
      $lookup: {
        from: "extras",
        localField: "idExtra",
        foreignField: "_id",
        as: "extra"
      }
    }
  ]).exec(function (err, listCandidacy) {
    if (err) throw err;
    res.render('candidacy', { Candidacy: listCandidacy });
  });
});
```

La jointure est une action définie dans une suite d'actions (un pipeline) gérée par la méthode *aggregate()* appliquée sur une collection.

**\$match** : sélectionne les documents de la collection.

**\$lookup** : applique la jointure.

**from** : le nom de la collection sur laquelle est opérée la jointure.

**localField** : le nom de la propriété locale qui crée le lien de jointure.



**foreignField** : le nom de la propriété de la seconde collection qui crée le lien de jointure.

**as** : le nom de la nouvelle propriété qui accueille les documents.

Ici vous trouverez un exemple de vue pug de notre application web FaceCast (vue des candidature)

La vue est chargée de restituer le modèle au sein d'une interface graphique et de permettre à l'utilisateur d'interagir avec le modèle.

## Liste des candidature

etat	nom du figurant	prenom du figurant	nom de l'evenement	role	Mise a jour	Suppression
Accepter	Wicket	Harry	destiny	cabal		

## c. API REST

API est un acronyme pour Applications Programming Interface. Une API est une interface de programmation qui permet de se « brancher » sur une application pour échanger des données. Une API est ouverte et proposée par le propriétaire du programme.

Dans le cadre de la programmation de notre API, nous allons utiliser une architecture REST. L'architecture REST « Representational State Transfer » consiste en la mise en place de services basés sur le protocole HTTP pour les opérations standard de « CRUD ». En effet, le protocole HTTP comprend différents « verbes », comme « GET », « POST », etc... qu'on va utiliser pour implémenter un CRUD.

```
//Call the dependencies: modules, models and other
var express = require('express');
var router = express.Router();

var Candidacy = require('../models/candidacy');
var Extra = require('../models/extra');
var Offer = require('../models/offer');
var objectId = require('mongoose').Types.ObjectId;

router.get('/', function (req, res, next) {
  res.send('Hi ApiRest , Choose roads : /offer or /extra or /candidacy');
});
```

→ Retourne une simple réponse qui demande à l'utilisateur de choisir une route

```
router.get('/checking/:email', function (req, res, next) {
  Extra.find({}, { key: 1 }, function (err, extras) {
    if (err) throw err;
    res.json(extras);
  });
});
```



<https://github.com/JonathanCrt/FaceCast-2>

```
});
});
```

→ Retourne l'id des figurants selon l'email attendu par l'URL sous le format Json

```
//Return email
router.get('email/:email', function (req, res, next) {
  Extra.find(
    { email: req.params.email },
    function (err, extras) {
      if (err) throw err;
      liste1 = res.json(extras);

    });
});
```

→ Retourne les figurants selon l'email attendu par l'URL sous le format Json

```
//Get road to return all Json file (offer & extras with agregation)
router.get('/key/:key', function (req, res, next) {

  Offer.aggregate([
    {
      $lookup: {
        from: "extras",
        localField: "_id",
        foreignField: "key",
        as: "offers"
      }
    }
  ]).exec(function (err, currentextras) {
    if (err) throw err;
  });
  Offer.find({}, function (err, currentoffers) {
    if (err) {
      throw err;
    }
    liste = res.json(currentoffers);
  });
});
```

```
});  
});
```

// 1: Return the collection offer in a Json format

```
router.get('/offer/', function (req, res, next) {  
  Offer.find({}, function (err, offers) {  
    if (err) throw err;  
    res.json(offers);  
  });  
});
```

//Return the number of offers in a Json format

```
router.get('/offer/:number', function (req, res, next) {  
  
  Offer.find({}, function (err, offers) {  
    if (err) throw err;  
    if (req.params.number == 0) {  
      res.json(offers[0]);  
    }  
    else {  
      res.json(offers.splice(0, req.params.number));  
    }  
  });  
});
```

// 2 : Return the collection extra in a Json format

```
router.get('/extra/', function (req, res, next) {  
  Extra.find({}, function (err, extras) {  
    if (err) throw err;  
    res.json(extras);  
  })  
});
```

//1 + 2 : Return the collection candidacy which is the result of aggregations between the two collections (extra +offer)

```
router.get('/candidacy', function (req, res, next) {  
  Candidacy.find({}, function (err, candidacys) {
```



```

        if (err) throw err;
        res.json(candidacys);
    });
});

//Save a new candidacy and return this in a json format
router.post('/candidacy', function (req, res, next) {
    var candidacy = new Candidacy({
        etat: req.body.etat,
        idOffer: req.body.idOffer,
        idExtra: req.body.idExtra
    });
    candidacy.save(function (err) {
        if (err) throw err;
        res.json(candidacy);
    });
});

// Return on candidacy with the ID an extra in parameter
router.get('/candidacy/:idExtra', function (req, res, next) {
    Candidacy.findById({idExtra }, function (err, listCandidacy) {});
    if (err) return res.status(500).send(err)
    return res.status(500).send(listCandidacy)

});
module.exports = router;

```

## d. Tests unitaires

Un test unitaire est un programme qui vérifie le bon fonctionnement d'un module (unité fonctionnelle) au travers de situations déduites des spécifications du module testé ; à partir de données d'entrée prédéterminées (l'état du système en entrée est connu), le test sollicite le module et confronte les données réellement obtenues avec celles théoriquement attendues, puis en déduit un état de succès ou d'échec. Un test est souvent accompagné d'une procédure d'exécution. L'activité de test d'un logiciel est un des processus du développement de logiciels.

```
package faceCast;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import com.mashape.unirest.http.HttpResponse;
import com.mashape.unirest.http.JsonNode;
import com.mashape.unirest.http.Unirest;
import com.mashape.unirest.http.exceptions.UnirestException;

public class TestUnit {

    JsonNode body;
    HttpResponse<JsonNode> response;
    HttpResponse<JsonNode> responseExtra;

    @Before
    public void setUp() throws UnirestException {

        response = Unirest.get("http://localhost:3000/rest/candidacy").asJson();
        body = response.getBody();
        responseExtra = Unirest.get("http://localhost:3000/rest/extra").asJson();
    }

    @Test
    public void testIdExtraAndIdOffer() {

        // Test IdExtra
        assertEquals("59f1dbd830e2ea92834cc568", body.getArray().getJSONObject(0).getString("idExtra"));
        // Test IdOffer
        assertEquals("59fed81b069a78003b0bc631", body.getArray().getJSONObject(0).getString("idOffer"));
    }
}
```



```

    }
    @Test
    public void testEtat() {

        assertNotNull(response);
        assertEquals("Waiting", body.getArray().getJSONObject(0).getString("etat"));
    }

    @Test
    public void myApplications() {
        try {
            HttpResponse<JsonNode> responseCandidacy = Unirest
                .get("http://localhost:3000/rest/candidacy/59f1dbd830e2ea92834cc568").asJson();
            assertEquals(2, responseCandidacy.getBody().getArray().length());
        } catch (UnirestException e) {
            // TODO Auto-generated catch block
            fail("Error : number of objects in the json");
            e.printStackTrace();
        }
    }
}

```

Le test *“testIdExtraAndIdOffer”* nous permet de vérifier que l’id Offer et l’id Extra est bien récupérer au sein de la collection.

Le test *“testEtat”* nous permet de vérifier que l’état d’une candidature n’est pas vide dans la collection et quelle est égal a la chaine de caractère *“Waiting”* (en attente).

### **Méthode de test mes candidatures (myApplications) :**

Le test *“myApplications”* nous permet de vérifier que la longueur du tableau correspond bien à celle attendue (soit 2 ici, correspondant aux 2 objets Json)

Le *assertEquals* nous permet de vérifier que la longueur du tableau *responseCandidacy* est bien de 2 (attendu).

