Rapport du projet de conception

Application Web de reconnaissance de visage sur *WebRTC*

Shiyi GU, Silin WANG

Proposé par Frédéric Pennerath

Sommaire

1. **Introduction** 3
   - **1.1. Reconnaissance de visage** 3
   - **1.2. WebRTC** 3
   - **1.3. Opencv** 4
   - **1.4. Node.js** 4
   - **1.5. Sails.js** 4
   - **1.6. MongoDB** 4
2. **Conception** 5
   - **2.1. Architecture globale** 5
   - **2.2. Architecture de la côté serveur** 5
   - **2.3. Modèle** 6
   - **2.4. View** 8
   - **2.5. Contrôleur** 8
3. **Logique** 9
   - **3.1. Logique globale** 9
   - **3.2. Logique de sign-up** 10
   - **3.3. Logique de Log-in** 11
   - **3.4. Logique de comparaison** 12
4. **Interface d'application** 13
   - **4.1. Page d'accueil** 13
   - **4.2. Page d'inscription** 13
   - **4.3. Page de connexion** 14

1. Introduction

Ce projet conception a pour but de réaliser en langage *Javascript* une application web qui peut reconnaître de visage, et nous utilisons cette application pour enregistrer et se connecter le compte de réseau, elle peut être divisé globalement en deux parties, le côté de client et le côté de serveur, pendant la réalisation, nous utilisons les technologies de *WebRTC* en côté de client, et les technologies de *Opencv*, *Node.js*, Sails.js et *MongoDB* en côté de serveur.

1.1. Reconnaissance de visage

La reconnaissance de visage est un domaine de la vision par ordinateur consistant à reconnaitre automatiquement une personne à partir d'une image de son visage. Dans ce projet, nous mettons en œuvre la reconnaissance de visage sur *WebRTC*.

1.2. WebRTC

WebRTC(Web Real-Time Communication, littéralement communication web en temps réel) est une interface de programmation (API) JavaScript actuellement au stade de brouillon (*Draft*) développée au sein du W3C et de l'IETF. C'est aussi un canevas logiciel avec des implémentations précoces dans différents navigateurs web pour permettre une communication en temps réel. Le but du *WebRTC* est de lier des applications comme la voix sur IP, le partage de fichiers en pair à pair en s'affranchissant des plugins propriétaires jusqu'alors nécessaires.

L'API repose sur une architecture triangulaire puis pair à pair dans laquelle un serveur central est utilisé pour mettre en relation deux pairs désirant échanger des flux de médias ou de données qui échangent ensuite sans autre relais. Cette architecture et la pile de protocoles utilisée posent des questions de sécurité et d'utilisation en relation avec d'autres technologies (comme les NAT ou les pare-feux) qui sont pour la plupart en cours de résolution par l'IETF et le W3C.

La technologie *WebRTC* étant récente, son intégration au sein des différents navigateurs internet est encore inégale.

Dans ce projet, nous appliquer deux API de *WebRTC* pour obtenir le média de utilisateur, ils sont *getUserMedia()* et *toDateUrl()*.

1.3. Opencv

OpenCV (pour Open Computer Vision) est une bibliothèque graphique libre, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel. Dans ce projet, nous utilisons l'algorithme d'*Opencv* pour reconnaître de visage.

1.4. Node.js

Node.js est une plateforme logicielle libre et événementielle en JavaScript orientée vers les applications réseau qui doivent pouvoir monter en charge. Elle utilise la machine virtuelle V8 et implémente sous licence MIT les spécifications *CommonJS*. *Node.js* contient une bibliothèque de serveur HTTP intégrée, ce qui rend possible de faire tourner un serveur web sans avoir besoin d'un logiciel externe comme *Apache* ou *Lighttpd*, et permettant de mieux contrôler la façon dont le serveur web fonctionne. Dans ce projet, nous mettons en pratique le *Node.js* comme serveur.

1.5. Sails.js

Sails.js est une architecture MVC pour développer des applications temps réel modernes et robustes avec *Node.js*.

1.6. MongoDB

MongoDB (de l'anglais humongous qui peut être traduit par « énorme ») est un système de gestion de base de données orientée documents, répartissable sur un nombre quelconque d'ordinateurs, efficace pour les requêtes simples, et ne nécessitant pas de schéma prédéfini des données. Dans ce projet, nous implémentons *MongoDB* comme la basse de données.

2. Conception

2.1. Architecture globale

Afin de réaliser les fonctionnements que nous avons conçu, nous avons besoin de construire un serveur qui permet de interaction avec le client. Parce ce que *Node.js* est une plateforme poids-léger, donc on l'a choisi en tant que le serveur, Sails.js est un architecture MVC qui est basé sur *Node.js*, par conséquent on peut utiliser le modèle de conception MVC pour réaliser l'interaction entre le serveur et le client. Comme le montre la Figure 1 ci-dessous, c'est l'architecture globale de notre application.

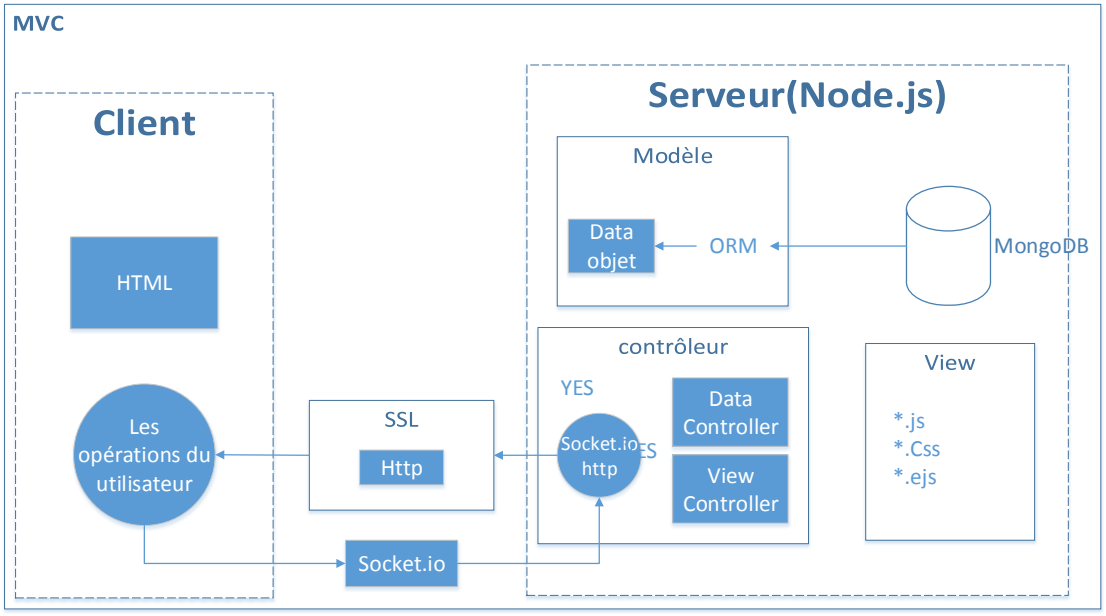


Figure 1

2.2. Architecture de la côté serveur

Comme mentionné précédemment, *Sails.js* est un cadre du modèle *MVC*, ainsi nous pouvons naturellement considérer la côté serveur comme trois parties des *M(modèle)*, *V(View)*, *C(contrôleur)*.
Dans *Sails.js*, Nous utilisons un modèle pour définir les types de données correspondants de la base de données, de ce fait il peut échanger des données avec la base de données.

View est principalement responsable de l'interface client, dans notre application, il est par *Html*, *Css* et *javascript* pour mettre en œuvre le Web. Contrôleur est responsable de la transmission de la demande, et traiter des requêtes.

Les différentes opérations sont contrôlées par les contrôleurs différents, c'est-à-dire de traiter les informations interactifs qui sont obtenu par *View*, ainsi que les opérations correspondantes sur la base de données. *Sail.js* peut manipuler des réponses de la page par le protocole *http* peut, également, il peut aussi gérer des services de connexion longs via *WebSocket* (réalisation en utilisant socket.io).

2.3. Modèle

Maintenant, on va présenter précisément les Modèle, Contrôleur et *View*, le Modèle de notre application est en utilisant de trois types de données, ces sont *Face*, *User* et Training data.

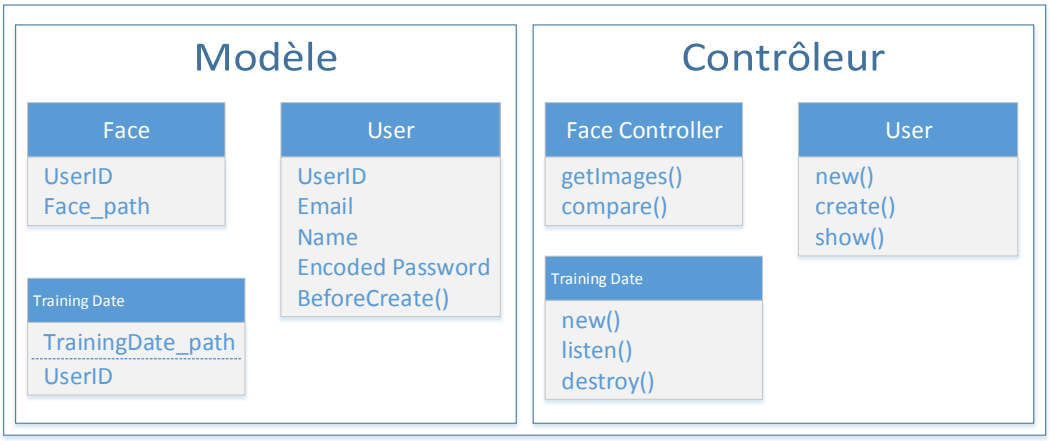


Figure 2

User permet de gérer les clients d'inscription, il inclure :

UserID : L'Identification unique de client *Email* : L'adresse électronique Le nom d'utilisateur Le mot de passe ---------- Le méthode *beforeCreate()*: pour traiter des données avant être produire

Face permet de lier le client et son visage, il inclure :

UserID : L'Identification unique de client
Facepath : l'adresse de visage

Trainingdata permet de lier le client et son visage, il inclure :

UserID : L'Identification unique de client
L'adresse de Traingdata

Grâce à la technologie de *ORM*, nous pouvons opérer les trois données en mode de orientée objet, le paquet *ORM* de *Sails.js* qui s'appelle *waterline* fournit les phrases d'opération simple, la méthode que nous utilisons principalement, ces sont :

create() : pour construire le phrase, on utilise le formule *json* pour régler le contexte d'information, par exemple :

```
User.create({
    UserID: ,
    Name:
})
```

find() : pour chercher les données, par exemple : find({UserID:xxxxxxxxx})
findByOne() : chercher en unitaire
remove() : pour supprimer
Tous les phrases peuvent être utiliser comme le méthode du modèle,
Par exemple : User.create(), Face.findByone({Email:xx@xx.com})

Afin de gérer les données que nous avons besoin de choisir la base de données appropriée, où nous avons choisi la base de données *MongoDB* qui utilise la technologie de *ORM*, les phrase d'opération de *MongoDB* et la technologie ORM de *Sails.js* sont très similaire, à cause de tous les deux utilisent *Javascript*, de sorte que la syntaxe est presque identique.

D'autre part, nous espérons de comprendre comment *MongoDB* fonctionner, donc nous avons choisi *MongoDB* comme le serveur de données. Afin de s'exécuter localement mongoDB, normalement il a besoin de faire fonctionner son serveur, puis créer des comptes avec des autorisations différentes pour manipuler. Lorsque le déploiement, on peut choisir une base de données locales et une autre base de données de réseau, ici nous avons choisi la basse de données qui est fourni par *MongoDB*. Ensuite, par la définition des informations nécessaires sur les fichiers de *Sail.js* qui sont *local* et adapter, tels que les noms d'utilisateur et mots de passe et l'adresse d'accès, etc, la liaison avec la base de données est fini.

2.4. View

Nous avons conçu trois pages: l'Accueil, la page d'inscription, la page de capture du visage, en raison de l'utilisation la moteur de *Ejs* et *Grunt*, nous pouvons utiliser *layout.ejs* pour définir les parties de l'information qui existent toujours sur la page, par exemple, nous fixons les fenêtre d'inscription et de connexion à *layout.ejs*, à cause de l'objectif de cette application est l'authentification par le reconnaissance de visage, donc nous avons écrit deux fichiers de *javascript* pour réaliser, ces sont getVideo.js et faceRecognition.js, le première permet de obtenir le vidéo, le deuxième est responsable de acquérir, envoyer et traiter la réponse de serveur.

2.5. Contrôleur

Nous avons deux contrôleurs sur Modèle, ces sont *FaceController* et *UserController* qui sont responsable de traiter les données de visage et gérer les données d'utilisateur, d'autre part, nous avons conçu *Sessioncontroller* à gérer les comptes utilisateurs, parmi :

Facecontroller :

getImages() : pour obtenir l'information de visage d'utilisateur et générer *Trainingdata* compare() : pour obtenir l'image du visage pendant connextion et le comparer avec *Trainingdata*

Usercontroller :

new() : pour afficher le *index*
create() : pour créer l'utilisateur inscrit
show() : pour afficher la page après la connexion

Sessioncontroller :

new() : pour afficher la page de connexion(juste pour tester)
listen() : pour surveiller le comportement de login de l'utilisateur, et transmettre l'information de visage reçu à compare() de Facecontroller pour comparer les visages
destroy() : pour réaliser la logique de quitter

3. Logique

3.1. Logique globale

La logique globale est : d'abord, s'inscrire et analyser les données de visage recueillies, puis pendant la connexion, comparer les données recueillies et *Trainingdata* et faire des jugements sur les résultats. En bref, la logique globale peut être divisé en trois parties : sign-up, login et compare.

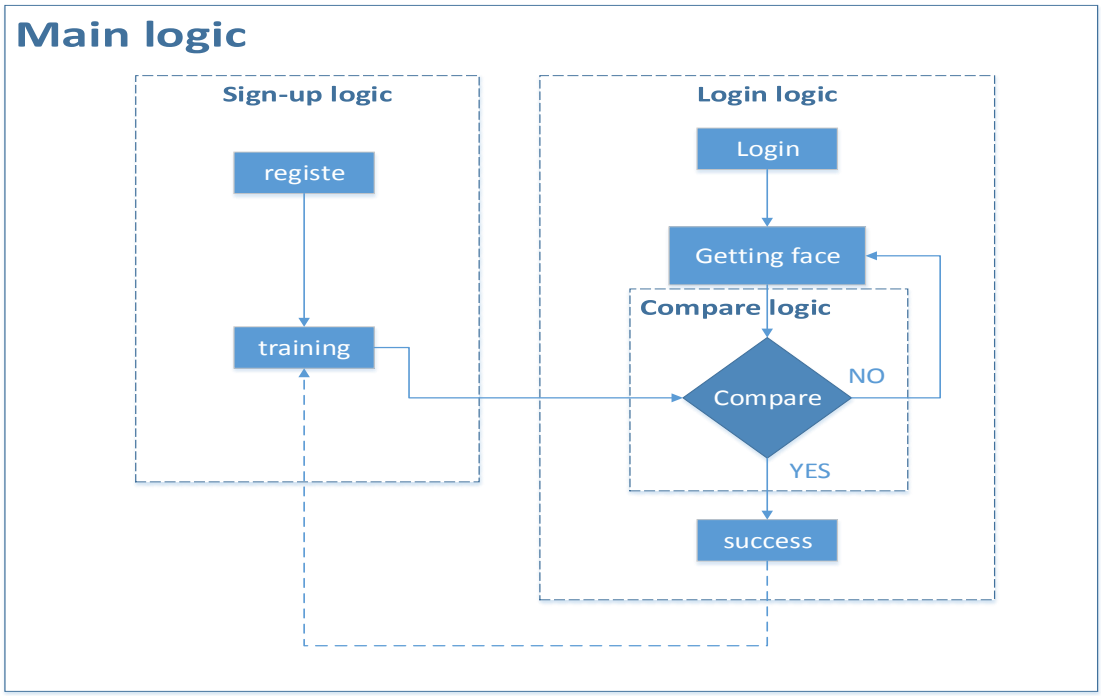


Figure 3

3.2. Logique de sign-up

D'abord, quand l'utilisateur clique le bouton de *sign-up*, *Http* reçoit une requête, à cause de la route par défaut, cette requête de *Url* sera dirigé vers user/new action et afficher la page d'inscription, puis l'utilisateur entre les informations voulu pour l'inscription. Lorsque l'utilisateur clique le bouton de *submit*, les données seront transférées vers le serveur, le serveur va appeler la méthode ORM du modèle pour créer des données de base de données, de cette façon un nouvel utilisateur sera été s'inscrit, en même temps, le serveur va orienter la réponse de URL à user/new action.

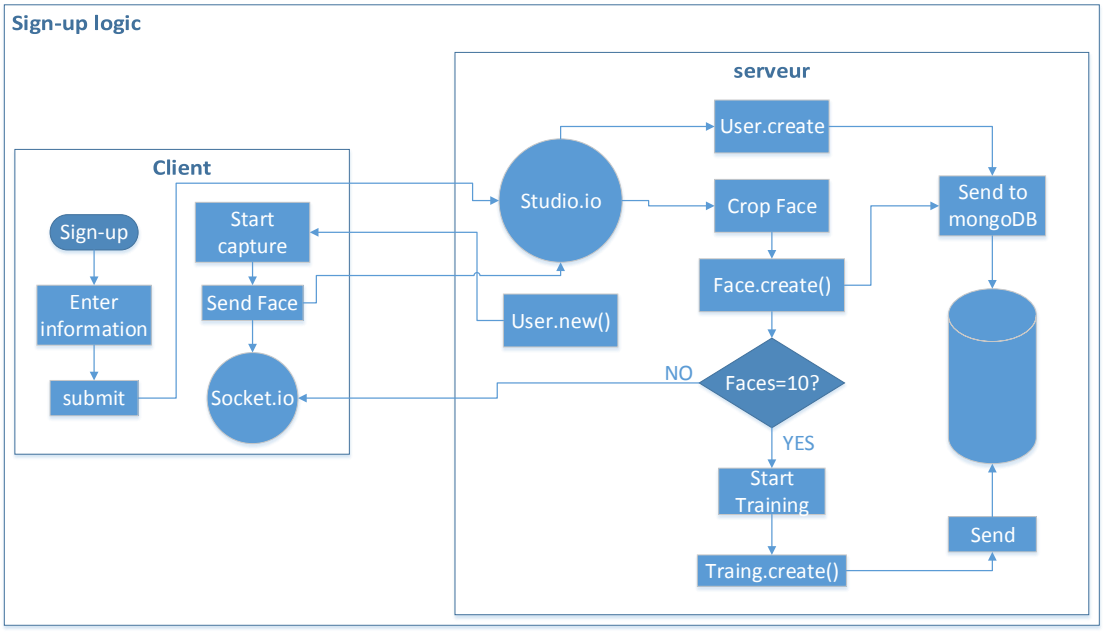


Figure 4

Après l'inscription, c'est l'acquisition du visage et *Training*. Nous pouvons obtenir le vidéo par *WebRTC* sur la page d'utilisateurs, puis lorsque l'utilisateur clique le bouton *startcaptrue*, nous transmettrons *Video frame en formule de image/jepg* à la base de données, ici nous utilisons *Socket.io* à accepter les données du visage, socket.io est une réalisation de *websocket*, il peut répondre les activités émis par la côté de client immédiatement. Après obtenir le visage de l'utilisateur, la côté de client va émettre une information d'activité *faceForRec* ce qui inclure l'adresse de mail et les données du visage de l'utilisateur, *Socket.io* va commencer à écouter les données émis quand la connexion sera établit, lorsque l'information d'activité reçu sera *faceForRec*, *Socket.io* va transmettre l'information de *json* à *getImages action* de Facecontroller

pour traiter les données du visage obtenues, afin de entraîner les données du visage en économisant de l'espace de stockage, nous découperons la partie du visage d'image en utilisant *haar feature*, si il ne peut pas obtenir exactement l'information du visage, le serveur va retourner un message faute, sinon il va retourner un message correct. Dans la le côté de serveur, *Socket.js* écoute également les activités, quand il recevra un message faute, il va traiter différemment pour les activités différents, si il racontera l'erreur de ne découper pas exactement la partie du visage d'image, il va indiquer l'utilisateur de régler la position du visage, si il n'y aura pas l'erreur, il va continuer obtenir le visage jusqu'à 10, chaque visage obtenu va appeler *create()* de *face*, lorsque il atteindra 10 visages, il va appeler l'algorithme *hbph* de *opencv* pour entraîner *Trainingdata, quand il finira, Trainingdata* va être transmettre à la basse de données.

3.3. Logique de Log-in

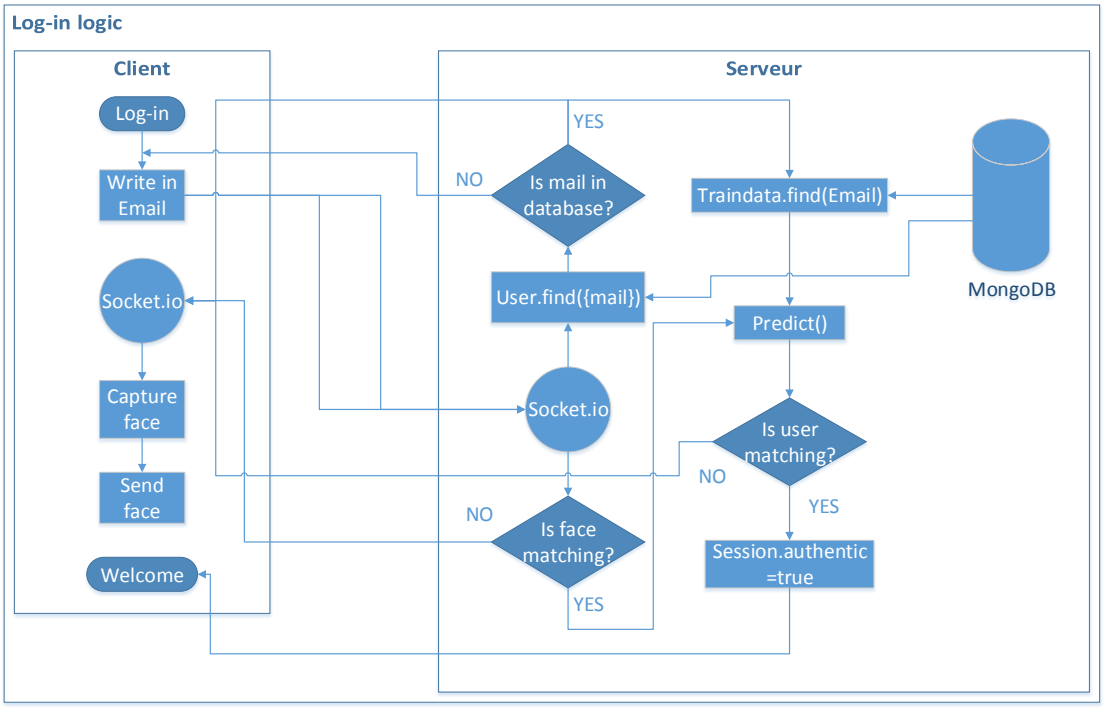


Figure 5

D'abord, l'utilisateur doit entrer Email, la côté de client va vérifier si l'Email entré est dans la formule correcte, puis l'Email de l'utilisateur va être émis à serveur lorsque le bouton *log-in* sera appuyer, ensuite le serveur va utiliser *user.findOneByEmail({email:addr})* à chercher si ce Email s'est inscrit avant, sinon il va retourner une erreur et demander l'utilisateur de entrer le Email une autre fois. Le serveur obtiendra

l'utilisateur correct lorsque le Email entré est correspondant à celui dans la basse de données, par *UserID on peut chercher le Trainingdata* correspondant, le même temps, le Webcam commence à obtenir les données du visage et les transmettre au serveur, enfin, le serveur transmettra les données du visage obtenu et *Trainingdata* à *compare action* de *facecontroller pour les comparer.*

3.4. Logique de comparaison

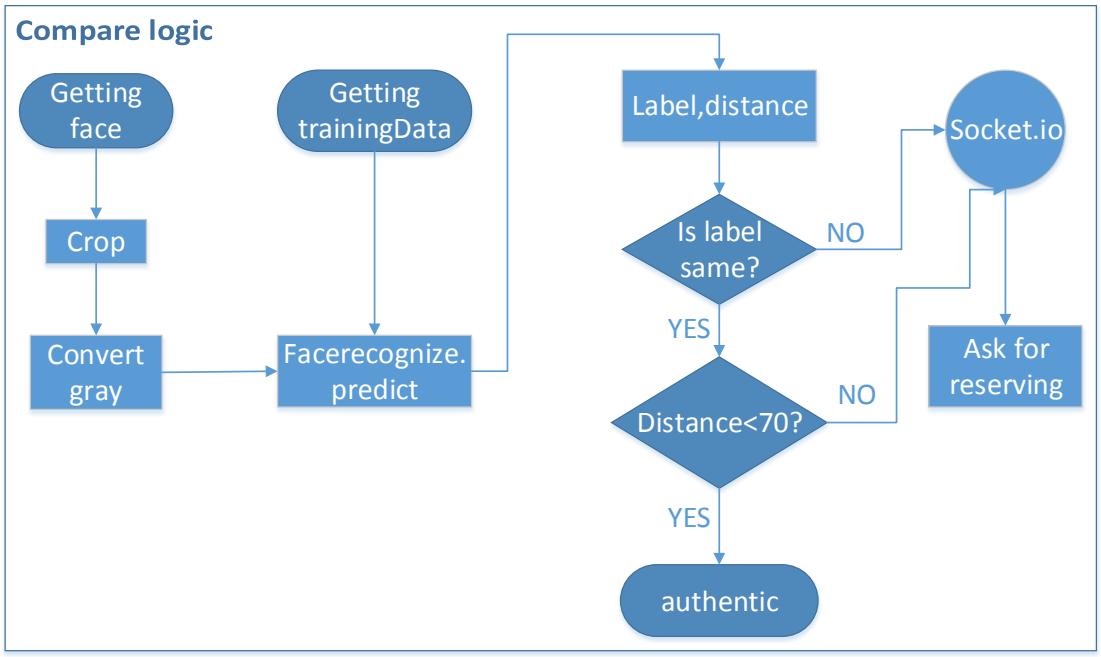
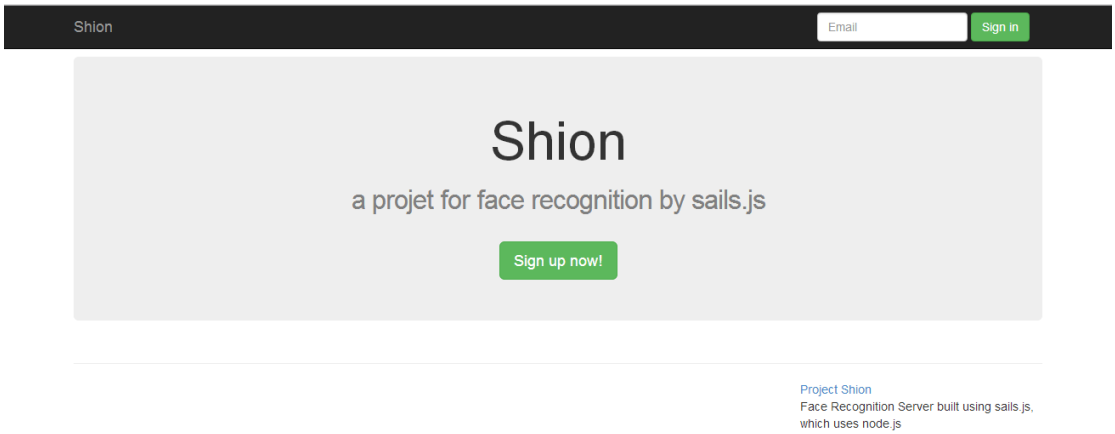


Figure 6

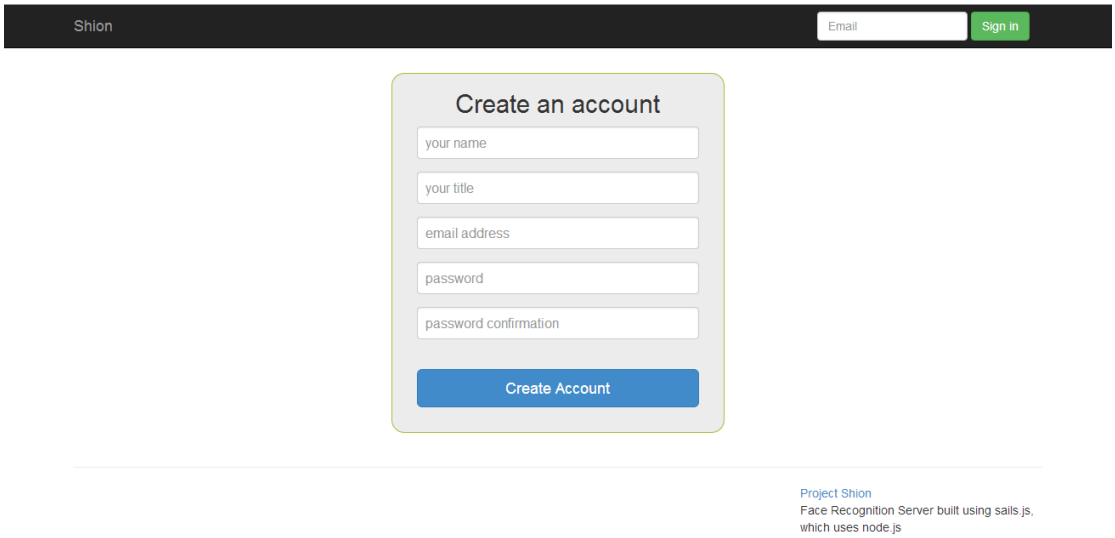
Au début, obtenir l'image et le convertir en niveaux de gris, et puis comparer Trainingdata aux données en niveaux de gris, ensuite il va retourner deux informations, le label et distance. Si le label et le label qui est obtenu avant seront identiques, dans ce cas, le serveur va authentifier, sinon, il ne va pas, et requêter la transmission encore une fois, en revanche, le serveur va continuer à juger si le distance est inférieur à seuil, si oui, le serveur va autoriser de se connecter.

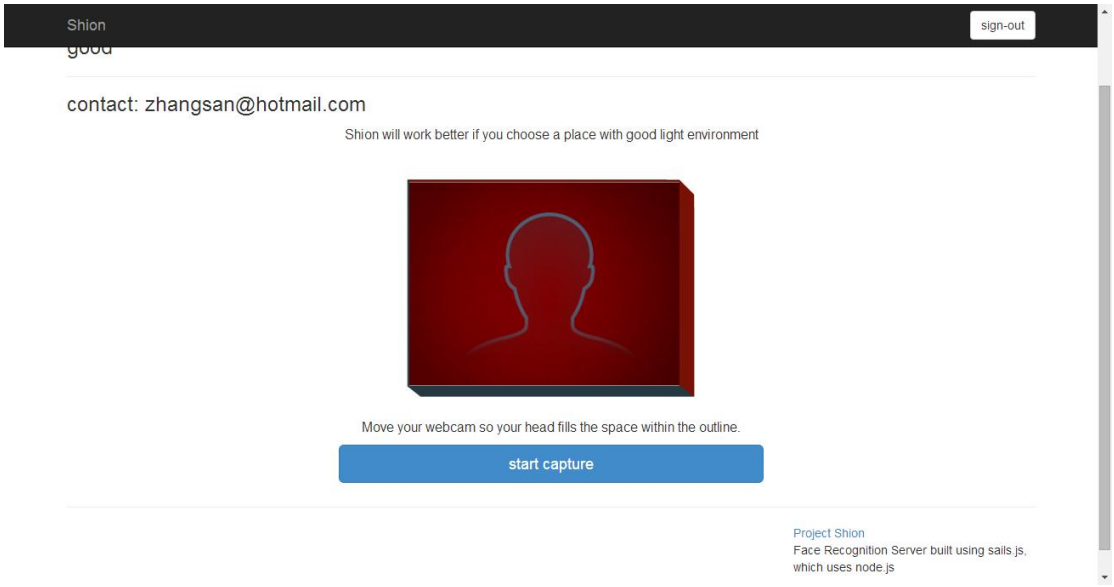
4. Interface d'application

4.1. Page d'accueil



4.2. Page d'inscription





4.3. Page de connexion