

Rapport Technique : Echangéo

par Guiraud Maxime
professeur référents : Ali Ed-Dbali

Introduction :

N'ayant malencontreusement pas trouvé de stage de fin d'année, j'ai demandé à monsieur Ed-Dbali s'il avait projet à me proposer. C'est donc dans le cadre d'un projet de fin d'étude que j'ai travaillé sur cette application.

Il existe déjà plusieurs plateforme de services entre particuliers, cependant elles reposent bien souvent des échanges monétaires. Le but de ce projet est donc de créer une plateforme permettant à tout un chacun de mettre à disposition ses services ou de demander les services d'autrui sans qu'il soit question d'argent.

Ce rapport a donc pour but d'expliquer en profondeur le fonctionnement de l'application pour permettre son amélioration futur.

Sommaire :

- I. Introduction
- II. Le fonctionnement de l'application
 - a. Analyse et Conception du sujet
 - b. La structure de l'application : les entités
 - c. La gestion des pages : Le duo contrôleurs/templates
 - d. La recherche d'annonce
 - e. Répondre à une annonce et noter le service
 - f. Gestion des services
 - g. Gestion des utilisateurs (et des bundles)
 - h. L'interface
 - i. Les améliorations possibles
- III. Les détails de l'application
 - a. L'arborescence du projet
 - b. Les entités
 - c. Les contrôleurs
 - d. Les templates et les Scripts
 - e. Les bibliothèques
 - f. L'utilisation de git

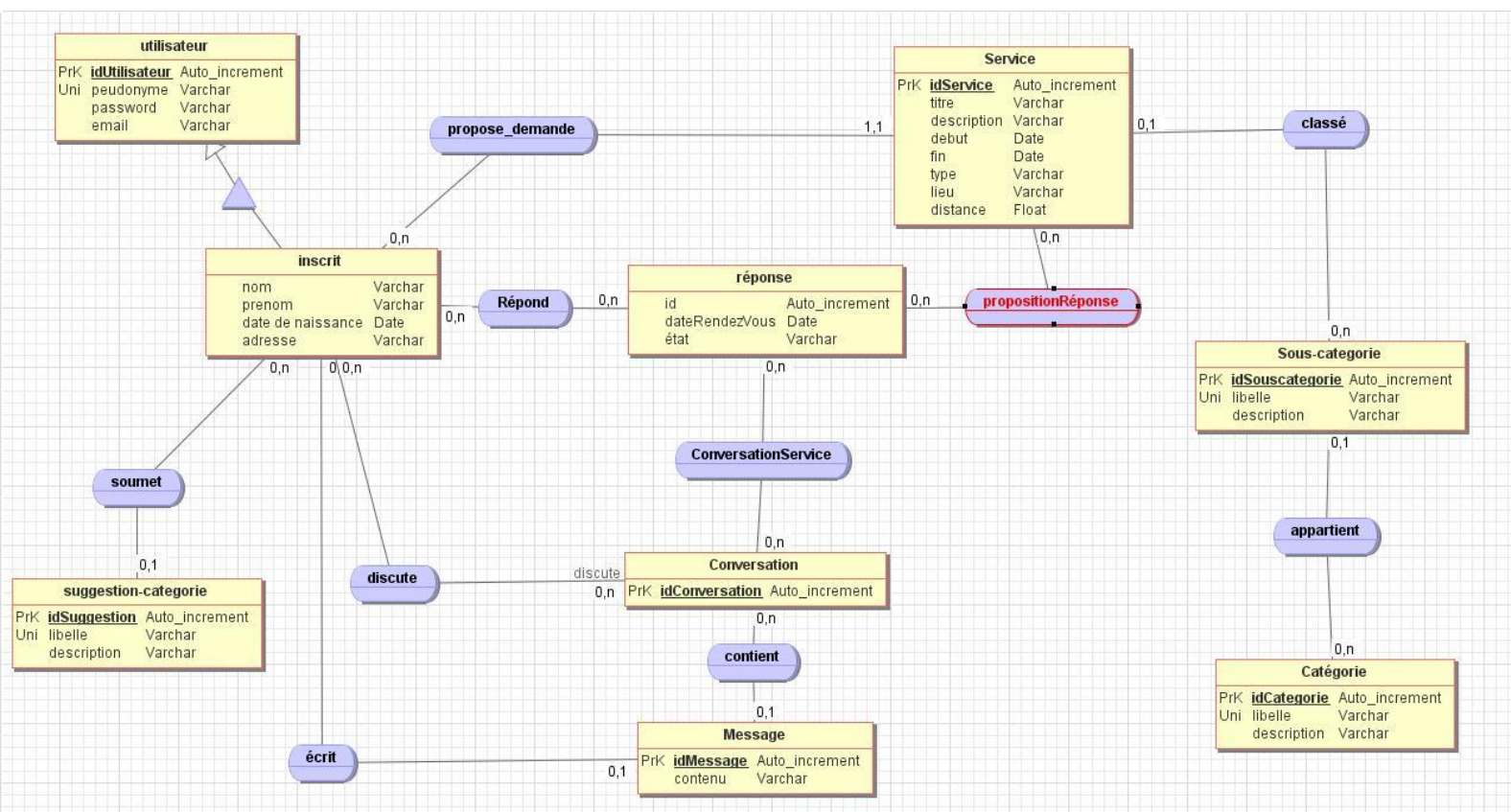
II. Le fonctionnement de l'application

a. Analyse et Conception du sujet

Le but du projet est donc de créer une plateforme d'échange de service entre particulier.

Elle doit être simple d'utilisation et basée sur la proximité. Pour être viable, les services reposent sur un système de notation réciproque. L'utilisateur qui répond à l'annonce note son créateur mais se fait lui aussi noter par ce dernier.

Pour comprendre le fonctionnement du site et de sa base de données, au cœur du système, un MCD a été créé.



Le service est au cœur de l'application. Un inscrit peut donc créer plusieurs services. Il peut aussi répondre à un service existant. Pour permettre la communication entre les deux utilisateurs (dans le cas où l'un des inscrits aurait des questions), un fois une réponse envoyée, une conversation est créée et peut contenir plusieurs messages.

Pour rendre plus accessible la recherche, les services sont triés par catégories et sous-catégories. (On trouve par exemple la catégorie « Cours particulier » qui comprend les sous-catégories « Maths », « Français », « Histoire », etc.).

b. La structure de l'application : les entités

Pour réaliser cette application, le choix a été fait d'utiliser le *Framework* Symfony dans sa version 2.8. La version 2.8 a été privilégiée face à la version 3.0 car elle dispose d'une documentation riche et est une version stable.

Outre le fait de permettre un développement selon la structure Modèle-Vue-Contrôleur particulièrement claire, ce *Framework* dispose d'un système d'entité simple et puissant.

Les entités sont les objets php servant d'interface avec la base de données. Pour interagir avec la BD, on utilise donc les entités et Doctrine, une bibliothèque native à Symfony gérant la connexion à BD et les requêtes.

Le principe est de créer toutes les entités et de définir les relations entre elles. Ces relations sont définies par annotation (commentaire précédant un attribut) grâce à une série de mots clés.

```
38  /**
39  *
40  *  @ORM\ManyToOne(targetEntity="Inscrit", inversedBy="reponses")
41  *  @ORM\JoinColumn(name="Inscrit_id", referencedColumnName="id")
42  */
43  private $inscrit;
44
```

Figure 1 - Exemple d'annotation

Il existe trois types de relations possibles :

- OneToOne, correspond aux cardinalité (0,1) – (0,1)
- OneToMany, correspond aux cardinalité (0,1) – (0,N)
- ManyToOne, correspond aux cardinalité (0,N) – (0,1)
- ManyToMany, correspond aux cardinalité (0,N) – (0,N)

A chaque fois, il faut définir vers quel entité se fait la relation (paramètre «targetEntity ») et par quel attribut de l'entité ciblée notre entité est défini (paramètre « inversedBy »). On peut aussi préciser le comportement à avoir en cas de suppression (paramètre « cascade »).

En plus de ça, il faut aussi définir avec `Column()` ou `JoinColumn()` comment sera traduit l'attribut dans la base de données.

Toutes les entités fonctionnent sur le même principe, je vais seulement expliquer la construction de l'entité `Reponse.php` qui servira d'exemple pour les autres.

Elle dispose donc des attributs :

- `Id`
- `dateRendezVous`
- `etat`

On les définit à chaque fois de la manière suivante :

```
24  /**
25   * @var \DateTime
26   *
27   * @ORM\Column(name="dateRendezVous", type="date", nullable=true)
28   */
29  private $dateRendezVous;
30
```

Figure 2 - définition attribut entité

« `@var` » sert à définir le type de la variable en php.

« `@ORM\Column()` » sert à définir comment la stocker dans la BD (nullable permet à la colonne d'avoir une valeur nulle).

Une fois tous les attributs créés, on crée les relations entre entités.

```
38  /**
39   *
40   * @ORM\ManyToOne(targetEntity="Inscrit", inversedBy="reponses")
41   * @ORM\JoinColumn(name="Inscrit_id", referencedColumnName="id")
42   */
43  private $inscrit;
44
45  /**
46   *
47   * @ORM\ManyToOne(targetEntity="Service", inversedBy="reponses")
48   * @ORM\JoinColumn(name="Service_id", referencedColumnName="id")
49   */
50  private $service;
51
```

Un membre et un service possèdent tous deux (0,N) réponses. On a donc une relation du type **ManyToOne** (dans le sens où plusieurs réponses font référence au même inscrit/service).

Il faut donc définir dans réponse les attributs inscrit et service. Et inversement définir réponses dans les entités Inscrit et Service.

```
/**
 *
 * @ORM\ManyToOne(targetEntity="Reponse", mappedBy="service", cascade={"remove", "persist"})
 */
private $reponses;
```

Là on définit la relation contraire, donc OneToMany, ainsi que les stratégies de suppression.

Les autres relations, entre réponse et conversation et réponse et évaluation fonctionnent sur le même principe.

```
52  /**
53  *
54  * @ORM\OneToOne(targetEntity="Conversation", inversedBy="reponse", cascade={"remove"})
55  * @ORM\JoinColumn(name="Conversation_id", referencedColumnName="id")
56  */
57  private $conversation ;
58
59  /**
60  *
61  * @ORM\OneToMany(targetEntity="Evaluation", mappedBy="reponse", cascade={"remove", "persist"})
62  *
63  */
64  private $evaluations;
65
```

Une fois toutes nos entités créées et nos relations définies, nous pouvons **créer notre base de données** avec les commandes suivantes :

```
> php app/console doctrine:generate:entities EchangeoBundle
```

```
> php app/console doctrine:schema:update --force
```

La première permet de créer automatiquement les getters et les setters des objets. Et la seconde permet de mettre à jour la BD.

c. La gestion des pages : Le duo contrôleur/template

Symfony2 utilise une structure MVC, on sépare donc le contrôleur qui gère les données et les Template qui les affichent. Pour cette application, on utilise trois contrôleurs différents :

- ApiContrôleur.php qui regroupe les fonctions utilisées par les appels AJAX.
- DashBoardContrôleur.php qui gère le dashboard, la partie administration.
- DefaultContrôleur.php qui gère le reste des pages.

Pour créer un contrôleur il faut faire un objet qui hérite de la classe *Controller*. Ensuite, il suffit d'ajouter des fonctions, pour lequel on définit par annotation des routes, qui gèrent le rendu de Template.

Ci-dessous, on a le contrôleur DefaultContrôleur.php et la fonction de rendu de la page d'accueil.

```
18 class DefaultController extends Controller
19 {
20     /*FONCTIONS DE TEST*/
36
37     /*ACCUEIL*/
38
39     /**
40      * Page d'accueil
41      * @Route("/",
42      *         name="index")
43      */
44     public function indexAction()
45     {
46         $docServices = $this->getDoctrine()->getRepository('EchangeoBundle:Service');
47         $services = $docServices->findBy(array(), array('id' => 'desc'), 4, null);
48         return $this->render('EchangeoBundle:Default:index.html.twig',array(
49             "services"=>$services)
50         );
51     }
```

Les Template quant à eux sont écrits en Twig, le langage de Template de Symfony. Ce langage permet l'héritage entre plusieurs Template et fonctionne sur un système de « block » de code.

Cela permet donc de créer des layouts de bases dont hériteront les templates spécifique à chaque page.

Par exemple le template index.html.twig (qui permet l'affichage de la page d'accueil) hérite de base.html.twig. Ce dernier offre une structure qui sera enrichie en contenu par index.html.twig grâce au système de block.

```
<!-- BODY ----- -->
<body id="body">
  {% block content %}
    le body
  {% endblock content %}
  {% block footer %}
    <footer>
      <div> Echangéo - 2016 </div>
    </footer>
  {% endblock footer %}
</body>
```

On crée donc différents blocks, ici les blocks *content* et *footer*. Ils sont ensuite complétés dans le template fils où l'on ne redéfini que ces blocks.

```
{% block content %}
  <div class="container">
    <div id="googlemaps" class="googlemaps"></div>
    {% if app.user %}
    {% else %}
    {% endif %}
    <h1 class="titre">Echangeo</h1>
    <div class="link"><a href="{{ path('recherche_service') }}" class="btn btn-primary">voir les annonces</a></div>
  <!-- dernières annonces -->
  <div id="main" class="main">
    <h2>Dernières annonces passées :</h2>
    <div class="annonces">
      {% for service in services %}
        <div class="annonce">
          <h3>{{service.titre}}</h3>
          <p>{{service.description}}</p>
          <p>Sous-catégorie : <b>{{service.sousCategorie.libelle}}</b></p>
          <p>Auteur : {{service.inscrit.username}} <!-- / {{service.inscrit.nom}} {{service.inscrit.prenom}} --></p>
          <a href="{{ path('recherche_service_id',{ 'id': service.id }) }}" class="btn btn-primary">voir en detail</a>
        </div>
      {% endfor %}
    </div>
  </div>
</div>
{% endblock content %}
```

Cependant, on souhaite parfois aussi afficher les informations du block défini dans le template Père ; on utilise donc la fonction `{{ parent() }}`.

Pour plus de détails, la hiérarchie des templates est défini dans la section *Les templates et Scripts* de la Partie III.

On peut donc lancer le server Symfony avec la commande (valable à la racine du projet)

```
> php app/console server:run
```

Le fonctionnement de base ayant été défini, passons au fonctionnement de l'application à proprement parlé.

d. La recherche d'annonces

L'application se base entièrement des services que proposent les gens. Cependant pour y répondre, il faut trouver ces services. C'est le but de la page /recherche.

La recherche peut se faire selon plusieurs critères :

- La catégorie
- Le département
- Des mots clés



Toutes catégories ▼ Toute la France ▼ mots clés ☐ recherche en profondeur Rechercher

Une fois les critères choisis, on valide avec le bouton Rechercher. Cela appelle la fonction javascript **displayRecherche()**, présente dans le fichier rechercheScript.html.twig qui regroupe tous les scripts de la page.

En premier lieu, la fonction, avec JQuery, récupère les valeurs des champs de recherche et les passe en paramètre de l'url utilisé par l'appel AJAX.

```
36 function displayRecherche() {
37     /*récupération des critères de recherche choisie*/
38     idCategorie = $("#sousCategorie option:selected").val();
39     departement = $("#departement option:selected").val();
40     keyword = $("#keyword").val();
41     if (keyword == ""){ keyword = "null"; }
42     profondeur = $("#profondeur").prop('checked');
43     console.log(idCategorie+" "+departement+" "+keyword+" "+profondeur);
44     /*création de l'url*/
45     var url = "{{ path('getRecherche',{ 'categorie': 'catPlaceholder' , 'departement': 'dptPlaceholder' , 'keyword': 'kwPlaceholder' , 'profondeur': 'proPlaceholder' , '_format': 'json'}) }}" ;
46     url = url.replace("catPlaceholder", idCategorie);
47     url = url.replace("dptPlaceholder", departement);
48     url = url.replace("kwPlaceholder", keyword);
49     url = url.replace("proPlaceholder", profondeur);
```

On vide ensuite la liste des dernières annonces pour préparer l'affichage de la recherche et on enlève les marqueurs de la carte avec la fonction **clearMarkers()**.

Début ensuite la fonction **Ajax**. Ajax permet de faire des appels asynchrones au contrôleur et donc de récupérer des données sans recharger la page.

```
56 $.ajax({  
57     url: url,  
58     type: "GET",  
59     dataType: "json",  
60     success: function(recherche){  
94     },  
95     error: function (req, status, err){  
96         console.log("erreur");  
97         console.log(err);  
98     }  
99 }
```

La requête utilise donc l'url précédemment créé et qui permet d'appeler la fonction `getRecherche()` de l'`ApiController`.

Cette fonction récupère en premier lieu les sous-catégories de la catégorie choisie (si une catégorie a été sélectionnée). Il y a ensuite quatre requête possible selon qu'une catégorie ou/et un département ont été renseigné ou non.

```
62 if($categorie != "null" && $departement != "null"){  
63     $em = $this->getDoctrine()->getManager();  
64     $query = $em->createQuery(  
65         'SELECT s  
66         FROM EchangeoBundle:Service s  
67         WHERE EXISTS(  
68             SELECT sc  
69             FROM EchangeoBundle:sousCategorie sc  
70             WHERE sc.categorie = :categorie  
71             AND s.sousCategorie = sc.id  
72         )  
73         AND s.departement = :dpt  
74         ORDER BY s.id DESC'  
75     )->setParameter('categorie', $categorie)  
76     ->setParameter('dpt', $departement);  
77     $servicesQuery = $query->getResult();  
78 }
```

Prenons le cas où une catégorie et un département ont été renseignés. On exécute une requête imbriquée pour obtenir les services souhaité. On souhaite en effet les services d'une catégorie précise, seulement, un service n'a pour attribut qu'une sous-catégorie. On ne prend donc que les services dont les sous-catégories appartiennent à la catégorie choisie.

```

118  /*Tri des services répondant au mots clés rentrés*/
119  $services = array();
120  $listeKeyword = explode(" ", $keyword);
121  if ($keyword != "null") {
122      foreach ($listeKeyword as $word) {
123          $index = 0;
124          foreach ($servicesQuery as $s) {
125              /*Si la recherche en profondeur est activé, la recherche des mots clés se fait dans le titre et dans la
126              description des services*/
127              if ($profondeur=="true" && (strpos(strtolower($s->getTitre()), strtolower($word)) || strpos(strtolower($s
128              ->getDescription()), strtolower($word)) )) {
129                  $services[] = $s;
130                  /*Une fois le service ajouté, on le retire de la liste parcouru pour éviter les doublons*/
131                  unset($servicesQuery[$index]);
132                  $index++;
133              }
134              elseif ($profondeur=="false" && strpos(strtolower($s->getTitre()), strtolower($word)) ) {
135                  $services[] = $s;
136                  unset($servicesQuery[$index]);
137                  $index++;
138              }
139          }
140      }
141      $services = $servicesQuery;
142  }
143

```

Une fois les services récupérés, on ne garde que ceux qui correspondent aux mots clés. De base, on ne recherche les mots clés que dans le titre de l'annonce, cependant il est possible de faire une recherche plus large mais moins précise qui cherche aussi dans la description.

Il faut ensuite optimiser notre liste de services pour transfert en JSON. En effet, on manipule pour l'heure des objets PHP complexe (dû aux relations entre les entités défini plus tôt). La conversion en JSON créerais des objets beaucoup trop lourds, on en crée donc de nouveaux avec seulement les attributs qui nous sont nécessaires.

```

154  $services_res = array();
155  foreach ($services as $s) {
156      $service = array('id' => $s->getId(),
157                      'titre'=> $s->getTitre(),
158                      'description'=> $s->getDescription(),
159                      'debut'=> $s->getDebut(),
160                      'fin'=> $s->getFin(),
161                      'type'=> $s->getType(),
162                      'distance'=> $s->getDistance(),
163                      'adresse'=> $s->getAdresse(),
164                      'lieu'=> $s->getLieu(),
165                      'username'=> $s->getInscrit()->getUsername(),
166                      'icone'=> $s->getSousCategorie()->getIcone(),
167                      );
168      $services_res[]=$service;
169  }
170  $res = array('sousCategories' => $sousCategories_res,
171              'services' => $services_res,
172              );

```

Il faut ensuite convertir \$res en objet JSON pour pouvoir l'utiliser en javascript. Pour la conversion, on utilise le bundle JMS/Serializer qui offre une meilleure conversion et plus simple que l'outil de base de Symfony.

```
173  /*passage en JSON avec Serializer*/
174  $serializer = $this->get('serializer');
175  $jsonContent = $serializer->serialize($res, 'json');
176  return new Response(
177      $jsonContent,
178      200,
179      array('Content-Type' => 'application/json')
180  );
181  }
182
```

On renvoie donc un objet JSON pouvant être utilisé par la fonction **displayRecherche()**. Le but est maintenant d'afficher les données reçus. On parcourt donc les sous-catégories et les services pour les afficher grâce à la fonction append() de JQuery.

```
78  for(var i=0; i<recherche.services.length; i++){
79      var service = recherche.services[i];
80
81      /*affichage des annonces*/
82      $('#titre').text("Annonces disponibles :");
83      $('#liste').append($("<div id='service'+service.id+'\" class='tuile'>"));
84      $('#service'+service.id).append($("<h3>").text(service.titre));
85      $('#service'+service.id).append($("<p>").text("Proposé par : "+service.username));
86      $('#service'+service.id).append($("<p>").text("Du : "+moment(service.debut).locale('fr').format("LLL")+
87          au "+moment(service.fin).locale('fr').format("LLL"))));
88      $('#service'+service.id).append($("<a id='"+service.id+"\" href='#\" class='btn btn-primary'>").text('voir en
89          détails'));
90
91      /*affichage des annonces à la carte*/
92      markerServices(false,service,service.icone);
93
94      /*ajout de la fonction d'affichage des détails d'un service*/
95      displayDetails(service.id,false);
96  }
```

Pour chaque service, on l'affiche sur la carte avec la fonction markerService() et on permet d'afficher les détails d'un service avec displayDetails(). Cette dernière fonction permet d'afficher un service dans son ensemble ainsi qu'un formulaire de réponse. Dans le cas où l'utilisateur n'est pas connecté, un lien vers la connexion s'affichera.

retour

Garde d'enfant

Proposé par : johnDoe

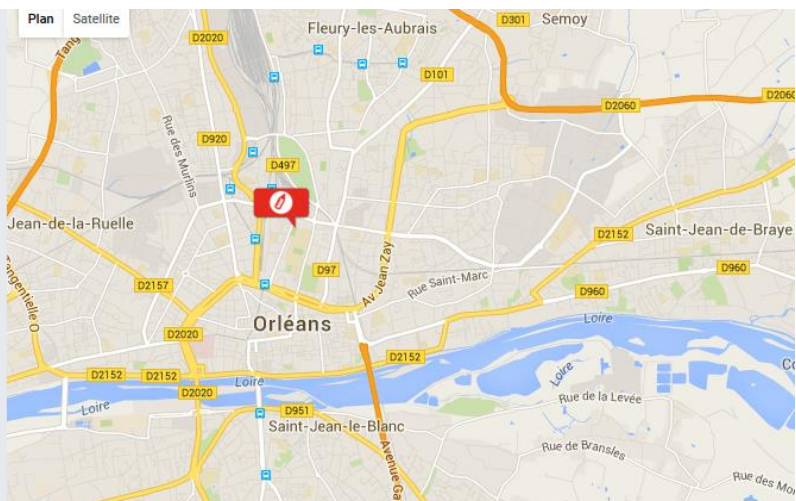
Du : 19 juin 2016 02:00 au 25 juin 2016 02:00

je me propose de garder des enfants

aaaa/mm/jj h:m

message

répondre



Pour continuer sa recherche, et voir les autres services répondant à notre recherche, on peut retourner à sa recherche avec le bouton retour. Il appelle la fonction `backDisplay()`.

Cette fonction a pour but de réafficher les services issus de la dernière recherche, stockés dans la variable globale **servicesCourants**.

Pour afficher la carte, l'api Google Maps est utilisée. La fonction `showGoogleMapsDefault()` affiche la carte par défaut. De base, elle est centrée sur Paris. Après une recherche, elle se centre sur la première annonce de la liste. De même lorsqu'on regarde les détails d'un service, la carte se centre sur son icône grâce à `updateMap()` qui utilise la méthode `panTo()` pour recentrer la carte.

De même, à chaque recherche ; les annonces sont affichées sur la carte avec la fonction `markerServices()`. À chaque recherche, les anciens services sont enlevés par `clearMarkers()`.

Enfin, `animationIcône()` permet grâce à la méthode `hover()` de JQuery d'animer l'icône du service qui est survolé dans la liste.

e. Répondre à une annonce et noter le service

Une fois les détails d'une annonce affichés, si elle nous intéresse, on peut y répondre avec le formulaire. Pour pouvoir préciser plus simplement la date du rendez-vous, un datepicker s'affiche lorsque le champ est activé. Ce datepicker utilise les bibliothèques Javascript Pickaday (<https://github.com/dbushell/Pikaday>) et Moment pour gérer les dates (<http://momentjs.com>).

Pour afficher le datepicker, il suffit de créer un nouvel objet Pickaday en lui passant en paramètre le champ texte où l'on veut que le calendrier s'affiche.

```
/*ajout du datePicker*/  
var picker = new Pikaday({ field: $('#date')[0] });
```

Un fois validé, le formulaire appelle `reponseAction()` dans `DefaultController.php`. Lorsqu'une réponse est envoyée, on crée un objet réponse, un objet message et un objet conversation (qui permet aux deux utilisateurs de discuter du service). On redirige ensuite vers la page de recherche. Dès lors, le créateur de l'annonce peut se rendre dans son **dashboard** et décider d'accepter ou non la réponse proposée.

Réponse :

[retour](#)

de **janneDoe**

date proposé pour le service :27 juin 2016 02:00

etat :attente

[accepter](#)

[decliner](#)

janneDoe

j'aimerais votre aide

message

[répondre](#)

Cela appelle la fonction `validationAction()` du contrôleur du dashboard. Si « décliner » a été cliqué, l'état de la réponse passe à « refuse » et la conversation se clôture. Dans le cas contraire, la réponse passe à « valide ».

Une fois la date du rendez-vous passée, la réponse passe à l'état « Notation ». Ce passage à l'état notation se fait lors de l'accès à réponse dans les fonctions `getReponseID()` et `getReponseUserID()` de l'ApiController.

L'état étant celui de notation ; le formulaire de notation sera affiché par la fonction `validation()` présente dans `serviceScript` et `reponseScript`. Cette fonction affiche les formulaires d'interactions et les informations de l'annonce en fonction de l'état de la réponse.

Le formulaire propose une notation par étoile grâce au fichier `star.css` et de laisser un commentaire.

Note :



L'application fonctionne sur le principe de service/réponse. Pour noter à son tour le service, l'utilisateur qui y a répondu doit se rendre dans l'onglet « Mes Réponses » de son dashboard.

Si les deux utilisateurs ont voté, l'état de la réponse passe à « cloture » lors de l'appel de `notationAction()`.

```
341
342 if (count($reponse->getEvaluations())>= 1) {
343     $reponse->setEtat("cloture");
344 }
345 else{
346     $reponse->setEtat("notation");
347 }
```

Cette fonction crée un nouvel objet `Evaluation`, l'enregistre dans la BD et redirige l'utilisateur vers la page d'où il vient.

Une fois clôturé, les utilisateurs ne peuvent plus communiquer. On peut tout de même renvoyer une nouvelle réponse à l'annonce.

f. Gestion des services

Nous avons vu comment fonctionne la recherche et la notation, voyons maintenant la création de service.

Les services se créent depuis l'onglet « Mes Services » et le bouton « créer un nouvelle annonce ». On est alors redirigé vers la page de création, générée par `addServicesAction()`. On crée un formulaire avec la fonction `createFormBuilder()` de symfony. Cela permet une gestion des réponses plus simple.

```
94     $formulaire = $this->createFormBuilder($service)
95         ->add('titre', TextType::class)
96         ->add('sousCategorie', EntityType::class, array('mapped' => true,
97             'required' => true,
98             'multiple' => false,
99             'expanded' => false,
100             'class' => 'EchangeoBundle:sousCategorie',
101             'choice_label' => 'libelle'))
102         //liste des sous-categorie avec les titres
```

Il suffit en effet, d'ajouter des champs avec la fonction `add()` pour créer le formulaire. Et en ayant passé en paramètre un objet service, celui-ci a se attribut directement paramétré avec les valeurs rentrées par l'utilisateur dans le formulaire.

La validation se fait avec en vérifiant la soumission du formulaire et en enregistrant le service.

Pour sélectionner l'adresse du service, une carte googleMaps est une nouvelle fois utilisée.

Elle créé de la même manière que pour la page de recherche avec

`showGoogleMapsDefault()`. Cependant un *action Listener* est ajouté pour pouvoir déplacer le marqueur sur la carte.

```
// Ajout d'un listener pour deplacer le marker en cliquant
google.maps.event.addListener(map, "click", function(event){
    radius = $("#form_distance").val();
    var lat = event.latLng.lat();
    var long = event.latLng.lng();
    var position = new google.maps.LatLng(lat, long);
    marker.setPosition(position);
    cityCircle.setMap(null);
});
```

Ce listener enregistre les coordonnées dans le champs caché « lieu » du formulaire et appelle la fonction `editAdresse()`. Cette fonction utilise l'api géocode de Google et affiche dans le champ « adresse » l'adresse où est positionné le marqueur. Elle enregistre aussi le département dans le champ caché du même nom.

Pour rendre la précision de l'adresse plus simple, on peut l'ajouter par géolocalisation.

```
316 $("#geoloc").on("click", function(){  
317     navigator.geolocation.getCurrentPosition(updateMap);  
318 });  
319
```

Une fois enregistré, un service peut toujours être modifié. On utilise le même template et la fonction `editServicesAction` crée le même formulaire, sauf qu'au lieu de passer en paramètre un service vierge, on place le service que l'on souhaite éditer, récupéré grâce à la méthode `find()` de Doctrine.

g. Gestion des utilisateurs (et des bundles)

Pour gérer les utilisateurs, le bundle Symfony2 FOSUserBundle (pour *Friend of Symfony User Bundle*) est utilisé. Ce bundle offre l'ensemble des pages et des fonctions permettant la gestion des utilisateurs.

Pour installer ce bundle, on utilise Composer.

Composer est un gestionnaire de dépendance php (<https://getcomposer.org/>). Cela permet de gérer les dépendances et l'installation de bundle. Pour ajouter un bundle on utilise la commande :

```
> composer require nom_du_bundle
```

Cela permet d'ajouter le bundle au fichier composer.json qui liste tous les bundle. Il faut ensuite l'installer avec la commande :

```
> composer install
```

Il est donc conseillé de lancer cette commande à chaque « pull » depuis le dépôt Git

FOSUserBundle est maintenant installé. On peut donc créer des membres.

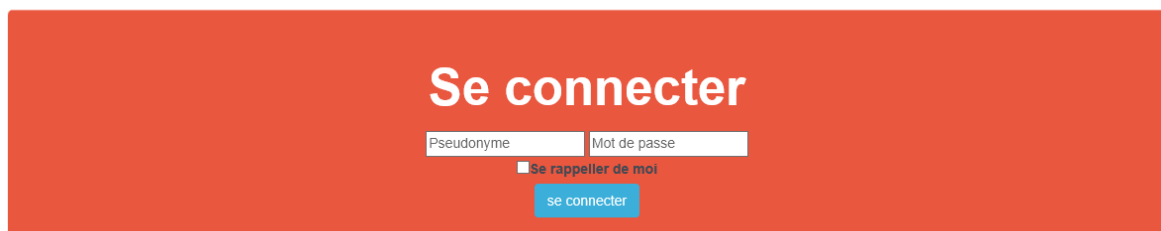
Cependant les membres de l'application nécessitent plus d'attributs que les utilisateurs de base proposés par le bundle. L'entité Inscrit hérite donc de l'objet BaseUser du FOSUBundle.

```
5 use FOS\UserBundle\Model\User as BaseUser;
```

De même, les templates des pages de connexion et d'inscription sont basique, on doit donc les surcharger pour pouvoir les modifier et ajouter du css.

Pour les surcharger, il faut recréer un FOSUserBundle dans app/Ressource et recréer les vues.

```
12 <div class="connexion">
13 <h1>Se connecter</h1>
14 <form action="{{ path('fos_user_security_check') }}" method="post">
15 <input type="hidden" name="_csrf_token" value="{{ csrf_token }}" />
16
17 <!-- <label for="username">Pseudonyme : </label> -->
18 <input type="text" class="from-control" id="username" name="_username" placeholder="Pseudonyme" value="{{
19 last_username }}" required="required" />
20
21 <!-- <label for="password">Mot de passe : </label> -->
22 <input type="password" class="from-control" id="password" placeholder="Mot de passe" name="_password" required="
23 required" />
24 <br>
25 <input type="checkbox" class="from-control" id="remember_me" name="_remember_me" value="on" /><label for="
26 remember_me">Se rappeler de moi</label>
27 <br>
28 <input type="submit" class="btn btn-primary" id="_submit" name="_submit" value="se connecter" />
29 </form>
30 </div>
```

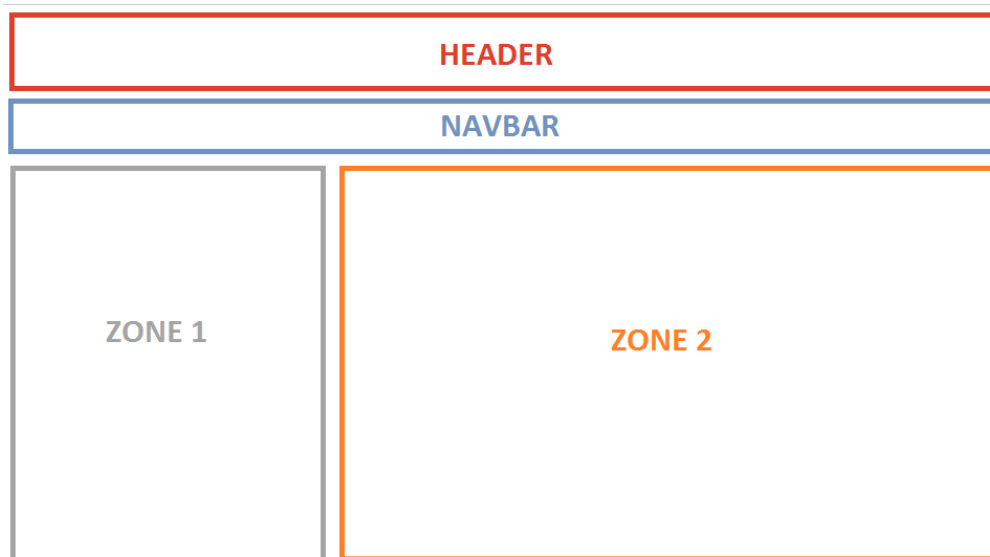


L'utilisateur peut ensuite éditer ses informations personnelles depuis le dashboard et la section « Mon profil ». Comme pour les services ; un formulaire est créé dans le contrôleur avec en paramètre l'inscrit souhaité.

h. L'interface

Pour l'interface, le choix a été fait d'adopter un style flat design. La bibliothèque Bootflat (<http://bootflat.github.io>) basé sur Bootstrap est donc utilisée. Cela permet d'avoir une interface simple et claire.

De plus, la mise en page de l'application suit toujours le schéma suivant :

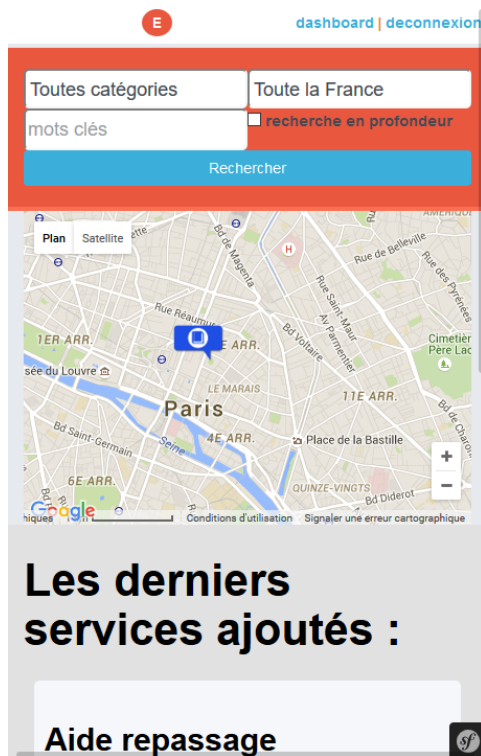


Le header regroupe toujours le logo et les liens de connexion ou vers le dashboard. La navbar sert à naviguer dans la section du site.

Dans la page de recherche, la zone 1 correspond à la liste des services et la zone 2 à la carte. Dans le dashboard, la zone 1 affiche aussi les services, mais la zone 2 affiche les détails du service sélectionné.

Un des points clés de l'application est d'être utilisable aussi bien sur ordinateur que sur mobile. Grâce à bootstrap, cette adaptabilité se fait facilement cependant certains détails doivent être gérés.

En version mobile, pour la page recherche, la carte s'affiche au-dessus de la liste des services. Pour cela on utilise flexbox et l'attribut *order* qui permet de réorganiser l'ordre des div.



De plus le header est lui aussi modifier pour prendre moins de place.

i. Les améliorations possibles

L'application est fonctionnelle. Il est possible de proposer des services, d'y répondre et de les noter. Malgré tout, certaines améliorations sont possibles.

Bien que l'application soit utilisable sur mobile, l'interface pourrait être optimisée pour ces appareils. Certaines interfaces sont pour le moment un peu ambiguës sur mobile.

Il pourrait aussi être intéressant de la proposer dans plusieurs langues.

Le passage en JSON des entités pourrait encore être optimisé.











Un système de notification pour les utilisateurs pourrait être intéressant.

III. Les détails de l'application

Le principe de cette partie est présenté en détails les fichiers qui compose l'application, leur arborescence et leur fonctionnement.

a. L'arborescence du projet

Pour mieux comprendre le fonctionnement, nous allons voir en détails l'arborescence du projet :

 .git	30/06/2016 22:20	Dossier de fichiers	
 app	20/05/2016 15:21	Dossier de fichiers	
 bin	22/04/2016 09:44	Dossier de fichiers	
 src	30/06/2016 16:00	Dossier de fichiers	
 vendor	10/05/2016 09:59	Dossier de fichiers	
 web	14/06/2016 11:40	Dossier de fichiers	
 .gitignore	24/05/2016 10:22	Document texte	1 Ko
 composer.json	12/05/2016 10:46	Fichier JSON	3 Ko
 composer.lock	12/05/2016 10:46	Fichier LOCK	95 Ko
 README.md	24/05/2016 10:22	Fichier MD	3 Ko

/app :

Ce dossier contient les fichiers de gestion de Symfony.

- Cache : contient les données mise en cache lors de l'utilisation de l'application en mode développement
- Config : contient les fichiers de configuration. On configure l'accès à la base de données dans le fichier **parameter.yml**
- Logs : regroupe les fichiers de logs

/app/Resources :

- FOSUserBundle : Contient les templates redéfini pour FOSUserBundle
- Views : Contient les templates par défaut de Symfony

/bin :

Contient les fichiers de doctrine.

/src :

Contient les bundles utilisés par le site.

/src/AppBundle :

Bundle de base de Symfony, il gère les pages par défaut.

/src/EchangeoBundle :

Bundle créé pour l'application et regroupant les fichiers nécessaire à son fonctionnement.

- Controller : regroupe les contrôleurs du site
- Entity : regroupe toutes les entités
- Form : regroupe les objets formulaires crée.
- Repository : regroupe les repository des entités
- Ressources : contient les dossiers config et views
- Ressources/config : contient les fichiers de configuration unique au bundle
- Ressources/views : contient les templates.
- Ressources/views/Dashboard : contient les templates du dashboard
- Ressources/views/Default : contient les templates du reste du site
- Ressources/views/Script : contient les Script Javascript (écrit avec twig)
- Test : contient les fichiers de test

/vendor :

Ce dossier contient les fichiers permettant le fonctionnement de Symfony. Il ne faut rien y modifier

/web :

Ce dossier contient les fichiers relatifs à la mise en forme des pages web tel que les fichiers css ou javascript.

- Assets : contient les images utilisées par l'application
- Bootflat : dossier de bootflat
- Bundles : contient les fichiers par défaut de symfony
- Css : contient les fichiers css
- Script : contient les bibliothèques javascript

b. Les entités

Les entités se trouvent dans le dossier *EchangeoBundle/Entity*.

Catégorie :

Attributs :

- Id : int
- Libelle : string
- Description : string
- sousCatégorie : relation *OneToMany* vers l'entité SousCatégorie

Conversation :

Attributs :

- Id : int
- reponse : relation *OneToOne* vers l'entité Reponse
- interlocuteur1 : relation *ManyToOne* vers l'entité Inscrit
- interlocuteur2 : relation *ManyToOne* vers l'entité Inscrit
- messages : relation *OneToMany* vers l'entité Message

Evaluation :

Attributs :

- Id : int
- note : int
- commentaire : string
- inscritNotant : relation *ManyToOne* vers l'entité Inscrit ; défini l'utilisateur qui note.
- inscritNote : relation *ManyToOne* vers l'entité Inscrit ; défini l'utilisateur qui est noté.
- Reponse : relation *ManyToOne* vers l'entité Reponse

Inscrit :

Hérite de *BaseUser*

Attributs :

- Id : int
- nom : string
- prenom : string
- dateNaissance : date
- adresse : string
- services : relation *OneToMany* vers l'entité Service ; défini les services créés par l'inscrit
- reponses : relation *OneToMany* vers l'entité Reponse ; défini les réponses posté par l'inscrit.
- Conversation1 : relation *OneToMany* vers l'entité Conversation ; défini les conversation où l'inscrit est l'interlocuteur1
- Conversation2 : relation *OneToMany* vers l'entité Conversation ; défini les conversation où l'inscrit est l'interlocuteur2
- messages : relation *OneToMany* vers l'entité Message ; défini les messages écrit par l'inscrit
- evaluationsDonnees: relation *OneToMany* vers l'entité Evaluation ; défini les évaluations données par l'inscrit
- evaluationsRecus: relation *OneToMany* vers l'entité Evaluation ; défini les évaluations reçus par l'inscrit

Message :

Attributs :

- Id : int

- contenu : string
- conversation : relation *ManyToOne* vers l'entité Conversation ; défini la conversation à laquelle appartient le message
- inscrit : relation *ManyToOne* vers l'entité Inscrit ; défini l'utilisateur qui a écrit le message.

Reponse :

Attributs :

- Id : int
- dateRendezVous : date ; date souhaité pour le service
- etat : string
- inscrit : relation *ManyToOne* vers l'entité Inscrit ; défini l'auteur de la réponse
- service : relation *ManyToOne* vers l'entité service ; défini le service auquel réponde la réponse
- conversation : relation *OneToOne* vers l'entité Conversation ; défini la conversation lié à la réponse
- evaluations : relation *OneToMany* vers l'entité Evaluation ; défini les évaluations qui sont données par l'auteur du service et l'auteur de la réponse.

Service :

Attributs :

- Id : int
- titre : string
- description : string
- debut : date ; défini la date à partir de laquelle peut être rendu le service
- fin : date ; défini la date jusqu'à laquelle peut être rendu le service
- type : string ; type du service (proposition ou demande)
- adresse : string ; adresse du service

- departement : string
- lieu : string ; position gps du service
- inscrit : relation *ManyToOne* vers l'entité Inscrit ; défini l'auteur du service
- sousCategorie : relation *ManyToOne* vers l'entité SousCategorie ; défini la sousCatégorie à laquelle appartient le service
- reponses : relation *OneToMany* vers l'entité Reponse ; défini les réponses au service

SousCategorie :

Attributs :

- Id : int
- libelle : string
- description : string
- icone : string ; nom de l'icône à utiliser pour afficher les services de la catégorie sur la carte.
- categorie : relation *ManyToOne* vers l'entité Categorie ; défini la catégorie mère.
- services : relation *OneToMany* vers l'entité Service ; défini les services de la sousCatégorie

SuggestionCategorie :

Entité représentant les suggestions de sousCatégorie faites par les utilisateurs

Attributs :

- Id : int
- cateogire : string ; catégorie mère
- libelle : string
- description : string
- inscrit : relation *ManyToOne* vers l'entité Inscrit ; défini l'auteur de la suggestion.

c. Les contrôleurs

Les contrôleurs se trouvent dans *EchangeoBundle/Controller*. Ils permettent de faire le lien entre les entités et les Template qui les affichent.

ApiController :

Ce contrôleur regroupe l'ensemble des routes renvoyant des données JSON, utilisées par les scripts JavaScript.

getAllServices() :

URL	/api/services
Name	getAllServices
Paramètres	aucun
Description	Fonction qui renvoie en JSON tous les services.

getRecherche() :

URL	/api/recherche/{categorie}_{departement}_{keyword}_{profondeur}.{_format}
Name	getRecherche
Paramètres	Id Catégorie, nom département, mot clés, profondeur
Description	Fonction qui renvoie en JSON les services répondant aux critères de recherche.

getServicesSousCategories () :

URL	api/serviceSousCategorie/{sousCategorie}_{categorie}_{departement}_{keyword}_{profondeur}.{_format}
Name	getServicesSousCategories
Paramètres	Id SousCategorie, Id Catégorie, nom département, mot clés, profondeur
Description	Fonction qui renvoie en JSON les services répondant aux critères de recherche. Prend en compte les sous-catégories

getServiceID () :

URL	/api/service/{id}.{_format}
Name	getServiceID
Paramètres	Id Service
Description	Fonction qui renvoie en JSON un service choisi par son id

getmonServiceID () :

URL	/api/dashboard/service/{id}.{_format}
Name	getmonServiceID
Paramètres	Id Service
Description	Fonction qui renvoie en JSON un service choisi par son id et les réponses qui lui sont liées

getReponseID () :

URL	/api/reponse/dashboard/{id}.{_format}
Name	getreponseID
Paramètres	Id Reponse
Description	Fonction qui renvoie en JSON une réponse choisi par son id et les messages qui lui sont liées

getReponseUserID () :

URL	api/reponseUser/dashboard/{id}.{_format}
Name	getreponseUserID
Paramètres	Id Reponse
Description	Fonction qui renvoie en JSON une réponse choisi par son id et les messages qui lui sont liées et le service auquel elle se rapporte.

DashboardController :

Ce contrôleur regroupe l'ensemble des routes gérant la partie dashboard

dashboardAction () :

URL	/dashboard
Name	dashboard
Paramètres	aucun
Description	Génère la page d'accueil du dashboard et renvoie la note moyenne de l'utilisateur

servicesUserAction () :

URL	/dashboard/services
Name	servicesUser
Paramètres	aucun
Description	Génère la page de gestion des services

addServicesAction () :

URL	/dashboard/services/new
Name	addServices
Paramètres	Requête du formulaire
Description	Génère la page et le formulaire de création de service.

editServicesAction () :

URL	/dashboard/services/edit/{id}
Name	editServices
Paramètres	Requête du formulaire, Id Service
Description	Génère la page et le formulaire d'édition d'un service ; utilise le même template que pour créer un service.

reponsesUserAction () :

URL	/dashboard/reponses
Name	reponsesUser
Paramètres	aucun
Description	Génère la page de gestion des réponses

sendAction () :

URL	/dashboard/send
Name	sendMessage
Paramètres	Réponse du formulaire
Description	Route d'envoi de message. Redirige vers la page depuis laquelle est envoyé le message

validationAction () :

URL	/dashboard/validation
Name	validation
Paramètres	Réponse du formulaire
Description	Route de validation d'une réponse

notationAction () :

URL	/dashboard/notation
Name	notation
Paramètres	Réponse du formulaire
Description	Route d'enregistrement d'une notation.

optionsAction () :

URL	/dashboard/options
Name	options
Paramètres	aucun
Description	Génère la page d'option du dashboard

editProfilAction () :

URL	/dashboard/options/profil/{id}
Name	editProfil
Paramètres	Réponse de formulaire, Id Inscrit
Description	Génère la page d'option du dashboard

optionsCategorieAction () :

URL	/dashboard/options/categorie
Name	optionsCategorie
Paramètres	Réponse de formulaire
Description	Génère la page et le formulaire de soumission d'une sous-catégorie

DefaultController :

Ce contrôleur regroupe l'ensemble des routes gérant la partie « front-office » de l'application

testAction () :

URL	/test
Name	test
Paramètres	aucun
Description	Génère une page utilisé pour tester le fonctionnement de doctrine

indexAction () :

URL	/
Name	index
Paramètres	aucun
Description	Génère la page d'accueil. Afficher les 4 dernières annonce créées.

rechercheAction () :

URL	/recherche
Name	recherche_service
Paramètres	aucun
Description	Génère la page de recherche des services.

rechercheServiceAction () :

URL	/recherche/service/{id}
Name	recherche_service_id
Paramètres	Id Service
Description	Génère la page d'un service. Affiche les détails d'un service et permet d'y répondre

reponseAction () :

URL	/recherche/reponse
Name	reponse_service
Paramètres	Réponse d'un formulaire
Description	Enregistre la réponse à une annonce ; crée la conversation et le message qui y sont liés.

d. Les Templates et les Scripts

Les templates servent à mettre en forme les informations envoyées par les contrôleurs. Ils utilisent le moteur de template Twig. Les scripts JavaScript permettent d'interagir avec l'application plus simplement. Ils sont écrits dans un fichier twig pour pouvoir utiliser les variables retournées par le contrôleur.

Les templates :

test.html.twig :

arborescence	EchangeoBundle/Resources/views/Default
héritage	Aucun
Fichier Script	aucun
Description	Template de la page de test

Base.html.twig :

arborescence	EchangeoBundle/Resources/views/Default
héritage	Aucun
Fichier Script	aucun
Description	Template de base qui donne la structure de base aux pages.

index.html.twig :

arborescence	EchangeoBundle/Resources/views/Default
héritage	Base.html.twig
Fichier Script	aucun
Description	Template de la page d'accueil

recherche.html.twig :

arborescence	EchangeoBundle/Resources/views/Default
héritage	base.html.twig
Fichier Script	rechercheScript.html.twig
Description	Template de la page de recherche

rechercheService.html.twig :

arborescence	EchangeoBundle/Resources/views/Default
héritage	recherche.html.twig
Fichier Script	rechercheScript.html.twig
Description	Template d'un service précis.

dashboard.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	base.html.twig
Fichier Script	aucun
Description	Template qui donne la structure du dashboard

dashboardOptions.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	dashboard.html.twig
Fichier Script	aucun
Description	Template de l'onglet des options

dashboardProfile.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	dashboardOption.html.twig
Fichier Script	aucun
Description	Template de mise en forme du formulaire d'édition des données personnelles

dashboardSuggestion.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	dashboardOption.html.twig
Fichier Script	aucun
Description	Template de mise en forme du formulaire de soumission de nouvelle sous-catégorie.

dashboardReponses.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	dashboard.html.twig
Fichier Script	reponsesScript.html.twig
Description	Template d'affichage des réponses de l'utilisateur

dashboardServices.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	dashboard.html.twig
Fichier Script	servicesScript.html.twig
Description	Template d'affichage des services de l'utilisateur

dashboardServicesAjout.html.twig :

arborescence	EchangeoBundle/Resources/views/Dashboard
héritage	dashboardService.html.twig
Fichier Script	servicesAjoutScript.html.twig
Description	Template de mise en forme du formulaire de création et d'édition d'annonce

Les scripts :

recherche.html.twig :

Description : Ce script est utilisé par la page de recherche. Il permet l'affichage de la carte et la validation de la recherche.

displayRecherche() :

Route utilisée pour l'Ajout	getRecherche
Description	Cette fonction sert à afficher les services correspondant à la recherche effectuée.

displayServices():

Route utilisée pour l'Ajout	getServicesSousCategories
Description	Cette fonction sert à afficher les services correspondant à la recherche effectuée, prend on compte les sous-catégories sélectionnées

displayDetails (id, onload):

Route utilisée pour l'Ajout	getserviceID
Description	Cette fonction sert à afficher les détails d'un service

backDisplay ():

Route utilisée pour l'Ajout	aucune
Description	Cette fonction sert à réafficher les recherches issues de la recherche quand on a affiché les détails d'un service.

showGoogleMapsDefaut ():

Route utilisée pour l'Ajout	aucune
Description	Cette fonction affiche la carte en utilisant l'api googleMaps

updateMap (lieu):

Route utilisée pour l'Ajx	aucune
Description	Cette fonction centre la carte sur les coordonnées passées en paramètre.

markerServices (default, service, icone):

Route utilisée pour l'Ajx	aucune
Description	Cette fonction affiche les services sur la carte

clearMarkers ():

Route utilisée pour l'Ajx	aucune
Description	Cette fonction supprime les services de sur la carte

drawCircle (service):

Route utilisée pour l'Ajx	aucune
Description	Cette fonction trace un cercle représentant la portée d'un service sur la carte.

animationIcône ():

Route utilisée pour l'Ajx	aucune
Description	Cette fonction permet d'animer l'icône d'un service sur la carte lorsque qu'il est survolé dans la liste.

reponses.html.twig :

Description : Ce script est utilisé par la page de gestion des réponses. Il permet l'affichage des réponses dynamiquement.

\$(".tuile").on("click", function()) :

Route utilisée pour l'Ajx	getreponseUserID
Description	Cette fonction permet d'afficher le service auquel l'utilisateur a répondu et sa réponse.

Validation(data) :

Route utilisée pour l'Ajx	aucune
Description	Cette fonction affiche le formulaire de notation ou non en fonction de l'état de la réponse

displayMap () :

Route utilisée pour l'Ajaj	
Description	Cette fonction affiche la carte d'un service

services.html.twig :

Description : Ce script est utilisé par la page de gestion des services. Il permet l'affichage des services dynamiquement.

\$(".tuile").on("click", function()) :

Route utilisée pour l'Ajaj	getmonServiceID
Description	Cette fonction permet d'afficher le service sélectionné dans la partie cadre.

listeReponses(data):

Route utilisée pour l'Ajaj	getreponseID
Description	Cette fonction affiche les réponses envoyées pour le service.

validation(reponse) :

Route utilisée pour l'Ajaj	aucune
Description	Cette fonction affiche le formulaire de notation ou non en fonction de l'état de la réponse.

displayMap () :

Route utilisée pour l'Ajaj	aucune
Description	Cette fonction affiche la carte d'un service

servicesAjout.html.twig :

Description : Ce script est utilisé par la page de création de service. Il permet de gérer la carte.

\$ (« .tuile »).on (« click », function()) :

Voir service.html.twig

listeReponses(data):

Voir service.html.twig

validation(reponse) :

Voir service.html.twig

displayMap () :

Voir service.html.twig

showGoogleMapsDefaut () :

Route utilisée pour l'Ajax	aucune
Description	Cette fonction affiche la carte pour sélectionner l'adresse du service grâce à un listener

updateMap(position):

Route utilisée pour l'Ajax	aucune
Description	Cette fonction met à jour l'adresse par géolocalisation

updateMap(position):

Route utilisée pour l'Ajax	aucune
Description	Cette fonction met à jour l'adresse par géolocalisation

editAdresse(lat,long):

Route utilisée pour l'Ajax	https://maps.googleapis.com/maps/api/geocode/json?
Description	Cette fonction récupère l'adresse d'un lieu avec ses coordonnées GPS grâce à l'api Géocode de GoogleMaps. Elle met à jour les champs cachés du formulaire.

e. Les bibliothèques

Pour fonctionner, l'application a recourt à plusieurs bibliothèques externes.

jQuery :

Cette bibliothèque permet de simplifier l'utilisation du JavaScript. Cela permet notamment de réaliser des requêtes asynchrones avec ajax.

Pikaday :

Cette bibliothèque permet d'ajouter des datepicker à un champ texte.



Moment.js :

Bibliothèque permettant une meilleure gestion des dates.

f. L'utilisation de git

Pour réaliser ce projet, l'outil de versionnage Git. Le projet est donc accessible en public à l'adresse : <https://github.com/Joystikman/echangeo>

Pour ce projet le principe de git flow a été plus ou moins utilisé. Le développement ne se fait donc pas sur le master mais à chaque fois sur une branche ayant pour nom **feature/nom-de-la-fonctionnalité** .

```
> git checkout -b feature/le-nom-de-la-feature
```

Un fois la fonctionnalité fini, il faut la pousser vers le dépôt distant et créer une **Pull Request**. (En ayant pris soin de pull les nouveautés avant de la soumettre).

```
> git push -u origin feature/le-nom-de-la-feature
```
