formation-angularjs

Travaux Pratiques

zenika
ARCHITECTURE INFORMATIQUE

Avant-propos

Afin de mettre en oeuvre les concepts vus dans le cours, nous vous proposons de développer une application de gestion de carnet d'adresses : ZenContacts.

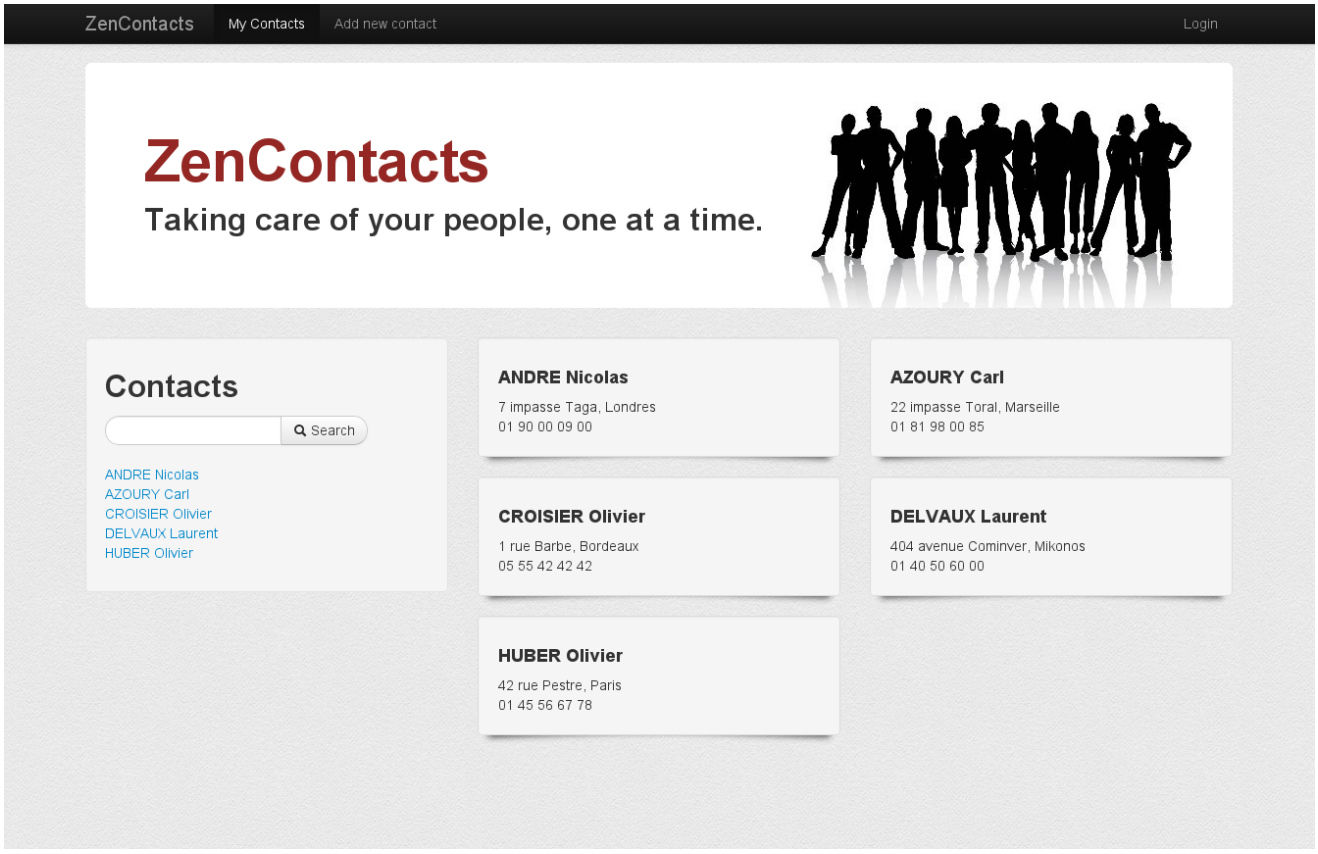
Nous partirons d'une simple maquette statique, qui sera enrichie au fur et à mesure des chapitres. Le résultat final sera une application moderne, riche et réactive.

La maquette est composée de 2 pages HTML statiques :

- Une page de liste des cartes de visite, avec un formulaire de recherche
- Une page de détail d'un contact permettant également de le créer/modifier

La maquette inclut le framework CSS Twitter Bootstrap permettant simplement d'avoir un style CSS "propre" de base.

Aucun code javascript n'est inclut dans ces 2 maquettes.



TP1 : Premier pas

Nous allons pour le moment nous concentrer sur la liste des contacts, page `contactList.html`, et commencer à rendre la maquette dynamique.

Objectifs :

- Apprendre à mettre en place AngularJS
- Déclarer un modèle et effectuer un binding inline ou via une directive
- Boucler sur une liste d'éléments
- Utiliser des filtres pour modifier l'apparence ou le comportement de l'application

Préparation du TP

Vous avez reçu l'archive TP.zip qui une fois décompressée affiche l'arborescence suivante :

```
TP.zip
├─────server                (Serveur de l'application)
│   └─────node_modules      (Généré - dépendances du serveur)
├─────tp0
│   ├─────node_modules      (Généré - dépendances de Karma)
│   │   └─────  ...
│   ├─────src
│   │   ├─────app           (Code de l'application)
│   │   │   ├─────components
│   │   │   ├─────core
│   │   │   └─────sections
│   │   ├─────assets
│   │   │   ├─────css
│   │   │   ├─────fonts
│   │   │   └─────img
│   │   ├─────e2e
│   │   └─────vendor        (Biliothèques externes)
│   │       ├─────angular
│   │       ├─────angular-ui
│   │       ├─────bootstrap
│   │       ├─────fuse
│   │       ├─────jquery
│   │       └─────showdown
│   └─────testrunner        (Runner de tests Jasmine)
└─────corrections           (Corrections de TP)
    └─────  ...
```

Lancer Chrome à l'adresse locale (file://) du fichier `contactList.html` .Exemple : `file:///TP/tp0/contactList.html` .

Intégration d'AngularJS

Dans le fichier `contactList.html` :

- Ajoutez la librairie Javascript `angular.js` (présent dans le répertoire `/src/vendor/angular/`) juste avant la balise de fermeture `</body>`.
- Activez le framework AngularJS pour l'ensemble de la page, en plaçant la directive `ng-app` sur la balise `<html>`.

Premier binding

Pour vérifier que tout fonctionne, nous allons mettre en place un premier binding.

Dans la maquette, un champ de recherche permet de filtrer la liste des contacts. Afin de fournir un feedback à l'utilisateur, nous allons afficher sous ce champ la mention "Search for: XXX", XXX étant le texte entré dans le champ de saisie.

- Indiquez que le champ de saisie est associé à un modèle nommé `search`
- Sous le champ, affichez `Search for:` et utilisez un binding inline `{{search}}` pour afficher le contenu du modèle `search`
- Rafraîchissez rapidement la page et constatez que le binding est visible un court instant avant le démarrage d'AngularJS. Remplacez le binding inline par un binding sur une balise `span ng-bind` et comparez.

Gestion des listes

Nous allons maintenant gérer l'affichage de la liste des contacts. Dans un premier temps cette liste sera fixe et déclarée dans la page même. Nous verrons plus tard comment la récupérer depuis un contrôleur, puis depuis un serveur.

- Une liste de contacts au format JSON est disponible dans le fichier `contacts.json` fourni dans le répertoire `server`
- A l'aide de la directive `ng-init` placée sur la balise `<body>`, déclarez et initialisez un modèle nommé `contacts`, décrivant une liste de personnes (copier/coller les données du fichier `contacts.json`). Attention aux conflits de guillemets.

Il faut maintenant afficher le nom des contacts dans la colonne de gauche, ainsi que les cartes de visite dans la partie centrale de la page. Ces listes viennent remplacer les listes statiques de noms déjà présent dans la page.

- Utilisez la directive `ng-repeat` pour boucler sur la liste des contacts, et des bindings (inline ou sous forme de balises) pour afficher leurs propriétés aux endroits appropriés.

Intégration de filtres

Dans la maquette, les contacts sont triés par ordre alphabétique, et les noms de famille sont affichés en majuscules.

- Utilisez les filtres `orderBy` et `uppercase` pour implémenter ces spécifications.

Pour finir, nous allons filtrer les contacts en fonction de ce que l'utilisateur saisit dans le champ de recherche.

- Utilisez le filtre `filter` sur la directive `ng-repeat` pour implémenter la fonction de recherche.

TP2 : Contrôleurs

Dans ce second exercice, nous allons mettre en place un contrôleur, qui sera responsable de la gestion des données affichées par le template (la page HTML). Le découplage entre contrôleur et template est important car il facilite le développement, le test et la maintenance des applications.

Objectifs :

- Développer un contrôleur et l'associer à un template.

Pré-requis

Nous allons a présent continuer la suite du TP en environnement réel, c'est-à-dire en accédant à notre application depuis un serveur HTTP/REST accessible depuis une URL : http://localhost:8080. Suivez les instructions du formateur pour installer en local un serveur HTTP. Les indications se trouvent dans l'annexe de ce cahier.

Création du contrôleur

Nous allons créer le contrôleur `ContactListController`, responsable de la gestion de la liste des contacts.

- Créez le fichier `/src/app/zen-contact.module.js` contiendra notre module principal.
- Dans ce fichier, déclarez un module AngularJS nommé `zen-contact`.
  `angular.module('zen-contact', ['zen-contact.sections']);`.
- Créez le fichier `/src/app/sections/sections.module.js` contiendra les modules des sections.
- Dans ce fichier, déclarez un module AngularJS nommé `zen-contact.sections`.
  `angular.module('zen-contact.sections', []);`.
- Créez le dossier `/src/app/sections/list`.
- Dans ce dossier, créez le fichier `contact-list.controller.js` qui contiendra le contrôleur `ContactListController`.
- Implémentez le contrôleur `ContactListController` avec comme module `zen-contact.sections`.

```
tp0
├──src
│  └──app
│     ├──zen-contact.module.js                (module `zen-contact`)
│     ├──sections
│     │  ├──sections.module.js          (module `zen-contact.sections`)
│     │  └──list
│     │     └──contact-list.controller.js
│     └── ...
└── ...
```

Nous allons utiliser la syntaxe « controllerAs » pour ce premier contrôleur et dans l'ensemble des exercices. Notre liste de

contacts doit donc être déclarée comme propriété de (l'instance) de notre contrôleur.

- Affectez la référence du contrôleur (`this`) à une variabe `vm` .
- Associez à `vm` une variable `contacts` , contenant une liste de personnes (remplace le `ng-init`).

Intégration du contrôleur

Il faut maintenant intégrer le contrôleur à l'application.

- Dans la page HTML, importez les scripts créés à la suite de la librairie `angular.js` .
- Complétez la directive `ng-app` existante pour lui demander d'utiliser notre module `zen-contact` .
- Associez le contrôleur à une section de la page à l'aide de la directive `ng-controller` : `ng-controller="ContactListController as contactList"` . La balise `<div>` qui suit immédiatement la `<div>` avec classe `jumbotron` semble un bon emplacement.
- Modifier toutes vos variables utilisées dans la template pour pointer vers ce nouveau contrôleur. Par exemple, `contacts` devients `contactList.contacts` .

Les données étant maintenant gérées par le contrôleur, la déclaration de la liste dans la page même n'est plus nécessaire.

- Supprimez la directive `ng-init` introduite à l'exercice précédent sur la balise `<body>` .
- Testez le bon fonctionnement de la page.

Bonus : ngPluralize

Pour finir, nous allons améliorer le feedback associé au champ de recherche en indiquant combien de résultats correspondent au critère, sous la forme "[nombre] résultat(s) pour '[critère]'". Bien sûr, cette information ne doit être affichée que si le champ de recherche n'est pas vide.

- Insérez une balise `<div>` sous le champ de saisie, qui contiendra le texte de feedback. Utilisez la directive `ng-show` pour ne l'afficher que si le champ de recherche n'est pas vide (c'est-à-dire si son modèle n'est pas vide).
- Utilisez la directive `ng-pluralize` pour sélectionner la bonne formule parmi "No contact", "One contact" ou "X contacts". **Cette directive est détaillée dans les diapos du chapitre I18N**.

Question : comment connaître dynamiquement le nombre de résultats correspondant au critère de recherche ? (Astuce : au sein de la directive `ng-repeat` , il est possible de créer de nouvelles variables).

TP3 : Routage

Maintenant que la page listant les contacts fonctionne, nous allons intégrer à l'application l'autre page de la maquette, et mettre en place un système de navigation.

Objectifs :

- Mettre en place une application de type "single-page".
- Associer des URLs aux différents états de l'application.
- Naviguer au sein de l'application.

Modularisation

L'application que nous développons est de type "single-page", c'est-à-dire qu'elle est composée d'une unique page hôte dont certaines portions seront remplacées dynamiquement en fonction du contexte.

Nous allons maintenant préparer cette page hôte, ainsi que les fragments de vue qui constitueront les différents états de l'application (liste, détails, ajout). La page `contactList.html` nous servira de base. Nous en conserverons la structure générale, la barre de navigation supérieure, la zone de titre, ainsi que les scripts. La partie centrale (tout le `div` suivant le `div.jumbotron`) sera intégralement remplacée selon le contexte.

- Dupliquez la page `contactList.html` pour créer la page hôte `index.html`.
- Dans la page `index.html`, remplacez toute la zone centrale par une directive `ng-view`, qui indiquera à AngularJS où insérer les fragments de template.
- Dans le répertoire `/src/app/sections/list/`, créez le fichier `list.html`.
- Créez le répertoire `/src/app/sections/edit/`, dans ce dossier créez le fichier `edit.html`.
- Dans chaque fichier, placez les zones centrales (à partir de la balise `<div class="row">`) des maquettes correspondantes (`contactList.html` pour `list.html` et `contactEdit.html` pour `edit.html`).

Puisque nous gérons deux pages supplémentaires, il nous faut également un nouveau contrôleur.

- Dans le répertoire `/src/app/sections/edit/`, créez le fichier `contact-edit.controller.js` et déclarez le contrôleur `ContactEditController`, vide pour le moment.
- Le contrôleur sera contenu dans le module `zenContact.sections`. N'oubliez pas d'importer le fichier à la fin de `index.html`.

Routeur

Il faut maintenant indiquer à AngularJS quand afficher quel fragment de vue. Pour cela, nous allons mettre en place un routeur, qui est l'équivalent du front-controller dans les frameworks MVC conventionnels. Il nous permettra d'associer des URLs à des contrôleurs et des fragments de pages (vues).

Pour configurer un routeur, nous devons déclarer la dépendance au module `ngRoute`.

- `angular.module('zenContact.sections', ['ngRoute', ...]);`

Créez le fichier `/src/app/sections/sections.routes.js`. Dans la fonction `config()` de pré-configuration du module `zenContact.sections`, utilisez le service `$routeProvider` pour configurer les routes suivantes :

- `/list` -> liste des contacts
- `/edit` -> nouveau contact
- `/edit/:id` -> consulter et modifier le contact
- Sinon, renvoyer vers la liste.

Il faut maintenant demander à AngularJS d'utiliser notre module.

- Dans la page `index.html`, importez la librairie javascript `angular-route.js` (fournie dans le répertoire js) à la suite de la librairie `angular.js`.
- Dans la page `index.html`, importez les scripts du module à la suite de la librairie `angular.js`.
- Retirez l'association du contrôleur `ng-controller` sur la balise `div` ajoutée lors du TP2.3 sur la page `list.html`.

Testez le bon fonctionnement de l'application en entrant manuellement les URLs correspondant aux différents états de l'application.

Navigation

Il ne nous reste plus qu'à faire fonctionner les liens HTML présents dans l'application, afin de permettre à l'utilisateur de naviguer entre ses différentes "pages".

- Dans la page hôte `index.html`, complétez les liens de la barre de navigation.
- Dans la vue `list.html`, activez les liens sur les noms des contacts, pointant sur la page de détails.
- Dans la vue `edit.html`, changer le bouton "Cancel" en lien qui doit permettre de revenir à la liste des contacts.
- Testez la navigation au sein de l'application.

Pour finir, il reste un petit détail cosmétique à régler : dans la barre de navigation, il faut que la bonne classe CSS (`.active`) soit affectée au lien correspondant à la vue courante.

Pour cela, nous devons interroger le service `$location` mais celui-ci n'est pas directement accessible dans les vues, et la barre de navigation n'est pas couverte par nos contrôleurs.

Nous allons donc créer un contrôleur `NavBarController` spécifique pour la barre de navigation. Placer ce contrôleur dans un fichier `contact-navbar.controller.js`, dans un repertoire `/src/app/sections/navbar/`.

- Créer un nouveau contrôleur contenant le code suivant :

```
var vm = this;
vm.isActive = function(path) {
    return $location.path().indexOf(path) > -1;
}
```

- Dans la page hôte `index.html`, placer la référence à ce nouveau contrôleur.
- Dans la page hôte `index.html`, utilisez la directive `ng-class` pour modifier dynamiquement la classe CSS des liens de la barre de navigation en fonction de la vue courante en utilisant la fonction `isActive`, maintenant accessible dans le contrôleur.
- Testez l'application pour vérifier l'affichage des liens de la barre de navigation.

TP4 : Services et IOC

Les vues composant l'application (liste, détail, ajout de contacts) interagissent avec la liste des contacts. Pour l'instant, celle-ci est gérée par le contrôleur `ContactListController`, et n'est donc pas accessible aux autres contrôleurs.

Nous allons donc extraire la liste et en confier la gestion à un service indépendant, sur lequel s'appuieront tous les contrôleurs.

Objectifs :

- Développer un service
- L'injecter et l'utiliser dans les contrôleurs
- Implémenter la page de détail d'un contact

Développement du service

Afin de faciliter le découplage et le test de notre application, nous allons créer un nouveau module AngularJS, nommé `zenContact.core`, correspondant à la couche "services" d'une application trois-tiers traditionnelle.

Ce module sera utilisé par notre module `zenContact.sections`.

- Créez le fichier `/src/app/core/core.module.js`.
- Dans ce fichier, déclarer le module `zenContact.core`.
- Dans la page hôte, importez ce script à la suite de la librairie `angular.js`.
- Dans le module `zenContact.sections`, déclarez le module `zenContact.core` comme dépendance.

Nous pouvons maintenant nous servir de notre tout nouveau module pour déclarer un service `ContactService`, responsable de la gestion des contacts.

- Créez le fichier `/src/app/core/contact/contact.service.js`.
- Dans ce fichier en utilisant le module `zenContact.core`, utilisez la fonction `service` pour déclarer `ContactService`.
- Ajoutez au service une variable `contacts` contenant la liste des contacts.
- Développez également les méthodes `getAllContacts()`, `getContactById(id)` et `saveContact(contact)`.

Enfin, pour tester le bon fonctionnement, modifions le contrôleur `ContactListController` pour qu'il utilise le service au lieu de stocker la liste des contacts "en dur".

- Injectez le service `ContactService` dans le contrôleur `ContactListController`.
- Récupérez la liste des contacts depuis le service.

- Testez le bon fonctionnement de l'écran listant les contacts.

Développement de la page d'affichage d'un contact

Passons maintenant à l'écran affichant les détails des contacts. Puisqu'on affiche un seul contact, il faut décider de la façon de l'identifier. Le plus simple est de passer son ID dans l'URL associée à la vue.

- Dans la vue `list.html`, modifiez les liens de façon à passer l'ID en paramètre.
- Dans le contrôleur `ContactEditController`, injectez le service `$routeParams`.
- Utilisez-le pour récupérer l'ID du contact à afficher.

Intéressons-nous maintenant à l'affichage de ce contact.

- Dans le contrôleur `ContactEditController`, injectez le service `ContactService`.
- Déclarez une propriété `contact` dans le contrôleur, et initialisez-la en interrogeant le service `ContactService`.
- Dans la vue `edit.html`, utilisez des bindings pour afficher les propriétés du contact dans les différentes `input`.

Pour finir, testez la navigation vers la page de détail, et le bon affichage des propriétés du contact sélectionné.

TP5 : Tests

Ce TP a pour objectif d'illustrer les outils de tests unitaires fournis avec AngularJS à travers la réalisation de tests sur l'application qui a été réalisée.

Préparation de Karma

Nous allons utiliser Karma pour exécuter les tests. Karma est une application Node.js.

En fonction de l'environnement de votre ordinateur :

- Si vous avez la possibilité d'installer Node.js :
  - Utiliser l'installeur avec les paramètres par défaut.
  - Après l'installation, vous aurez les commandes `node` et `npm` de disponible.
  - Dans le répertoire, taper la commande `npm install` pour télécharger les dépendances.
  - Taper la commande `npm install -g karma-cli` pour disposer de la commande `karma` .
- Si vous n'avez pas la possibilité d'installer Node.js :
  - Un fichier « `karma.bat` » se trouve le répertoire test et utilise l'exécutable `node.exe` fournit.
  - Cette commande permet donc de se substituer à l'installation standard de karma. Pour la suite, remplacer simplement la commande `karma` par `karma.bat` .

Lancer la commande `karma start` pour éxecuter Karma « à vide » afin de vérifier qu'il fonctionne puis le couper avec `Ctrl-C` .

Tester un service unitairement

Créez un nouveau fichier `/src/app/core/contact/contact.service.spec.js` .

Le premier test aura simplement pour objectif de vérifier que la méthode `getAllContacts` du service `ContactService` retourne bien une liste de 8 contacts.

- Injectez le service `ContactService` dans le `beforeEach` et stockez le service dans une variable `ContactService` .
- Créez un test avec la fonction `it` .
- Testez que la fonction `getAllContacts` retourne bien une liste de 8 contacts.
- Faîtes de même pour les méthodes `getContactById` et `saveContact` .

Retournez dans la console, Karma a dû détecter les changements dans les fichiers et ré-executer les tests. Les tests doivent bien sûr passer.

Tester un contrôleur unitairement

Ce test aura pour objectif de vérifier que le contrôleur `ContactListController` définit bien une propriété `contacts` et récupère la liste des contacts depuis le service `ContactService`, que nous *mockerons*.

- Créez un nouveau fichier `/src/app/sections/list/list.controller.spec.js`.
- De la même manière que lors des tests du service, ajoutez une fonction `beforeEach` qui aura pour but d'injecter un mock du service `ContactService`.
- Dans cette fonction, créez un object `ContactServiceMock` qui contient une fonction `getAllContacts`. Retournez de cette fonction, quelques contacts de la liste originale.
- Testez que `contacts` n'est pas encore défini dans le contrôleur.
- Instanciez le contrôleur `ContactListController` grâce au service `$controller`. Associez-y l'objet `ContactServiceMock`.
- Dans une fonction `it`, vérifiez que la propriété `contacts` du contrôleur contient bien la liste retournée par le *mock* du service `ContactService`.

Retournez dans votre console, Karma a dû détecter les changements dans les fichiers et ré-executer les tests. Les tests doivent bien sûr passer.

Test avec *mock* HTTP

Nous allons maintenant mocker la requête HTTP qui est exécutée dans le contrôleur pour fournir un jeu de données de test et valider que les données sont bien positionnées dans le dit contrôleur.

- Copiez une partie du fichier `contacts.json` dans une variable du test. Changer ensuite le *mock* HTTP pour retourner ces données lors de la requête.
- Dans un nouveau test, vérifiez que `contacts` est vide avant de commencer.
- Instanciez le contrôleur puis vérifiez que la liste de contact est définie mais est de taille nulle.
- *Flusher* `$httpBackend` pour libérer la réponse à la requête HTTP. Puis vérifiez que la propriété `contacts` à la bonne taille.

Test End-2-end (bonus)

Un squelette de test e2e est proposé pour aller plus loin. Les tests e2e nécessitent l'installation de protractor. C'est un module Node.js qui s'installe via la commande suivante : `npm install -g protractor` . Comme pour karma, des scripts sont proposés au cas où l'installation globale est impossible.

Le programme protractor s'accompagne du programme webdriver-manager qui permet de piloter les navigateurs.

Lancez la commande suivante pour initialiser webdriver la première fois : `webdriver-manager update` .

Ouvrir un nouveau terminal et lancer la commande `webdriver-manager start` . Le processus devra rester actif pendant l'exécution des tests e2e.

Exécutez les tests e2e en exécutant la commande `protractor protractor.conf.js` .

Complétez le fichier `e2e/search.scenario.js` en ajoutant un test qui exécute une recherche sur le mot "Wayne".

- Vérifiez que le message 'Search for: XXX' correspond bien au message attendu.
- Vérifiez que la liste de contacts sous le formulaire de recherche contient une balise de type `li` et contient le mot "WAYNE".

TP6 : Gestion des formulaires

Jusqu'à présent, nous n'avons fait qu'afficher des données. Il est temps de donner la possibilité à l'utilisateur de saisir de nouveaux contacts.

Objectifs :

- Développer la page de saisie d'un nouveau contact
- Gérer la saisie utilisateur via un formulaire
- Valider les données saisies

Gestion du formulaire d'ajout de contact

Commençons par gérer un formulaire basique, sans validation. Le principe est simple : il suffit de lier ("*binder*") les champs de saisie à des propriétés du contrôleur.

- Modifiez le contrôleur `ContactEditController` pour prendre en charge le cas où on crée un nouveau contact. C'est-à-dire que si `$routeParams` est `undefined`, il initialise la propriété `contact` avec un objet vide.
- Dans la vue `edit.html`, si ce n'est déjà fait, pour chaque champ de saisie, utilisez la directive `ng-model` pour lier le champ à la propriété correspondante du contact du contrôleur.

Soumettre le formulaire revient alors à appeler une simple méthode sur le contrôleur.

- Dans le contrôleur `ContactEditController`, implémentez la fonction `saveContact(contact)`. Celle-ci appelle le service `contactService` (à injecter) pour enregistrer le nouveau contact, puis le service `$location` (à injecter également) pour rediriger l'utilisateur vers la liste des contacts.
- Sur le bouton de validation du formulaire, appliquez la directive `ng-click` pour appeler la méthode `saveContact()` du contrôleur.

Vous pouvez maintenant tester le formulaire, et vérifier la présence d'une entrée supplémentaire dans la liste des contacts.

Validation des champs du formulaire

Les bonnes pratiques imposent que toutes les saisies utilisateur soient validées. Nous allons donc rajouter quelques contrôles sur le formulaire, ainsi que des messages de *feedback* en cas d'erreur.

Les règles sont les suivantes :

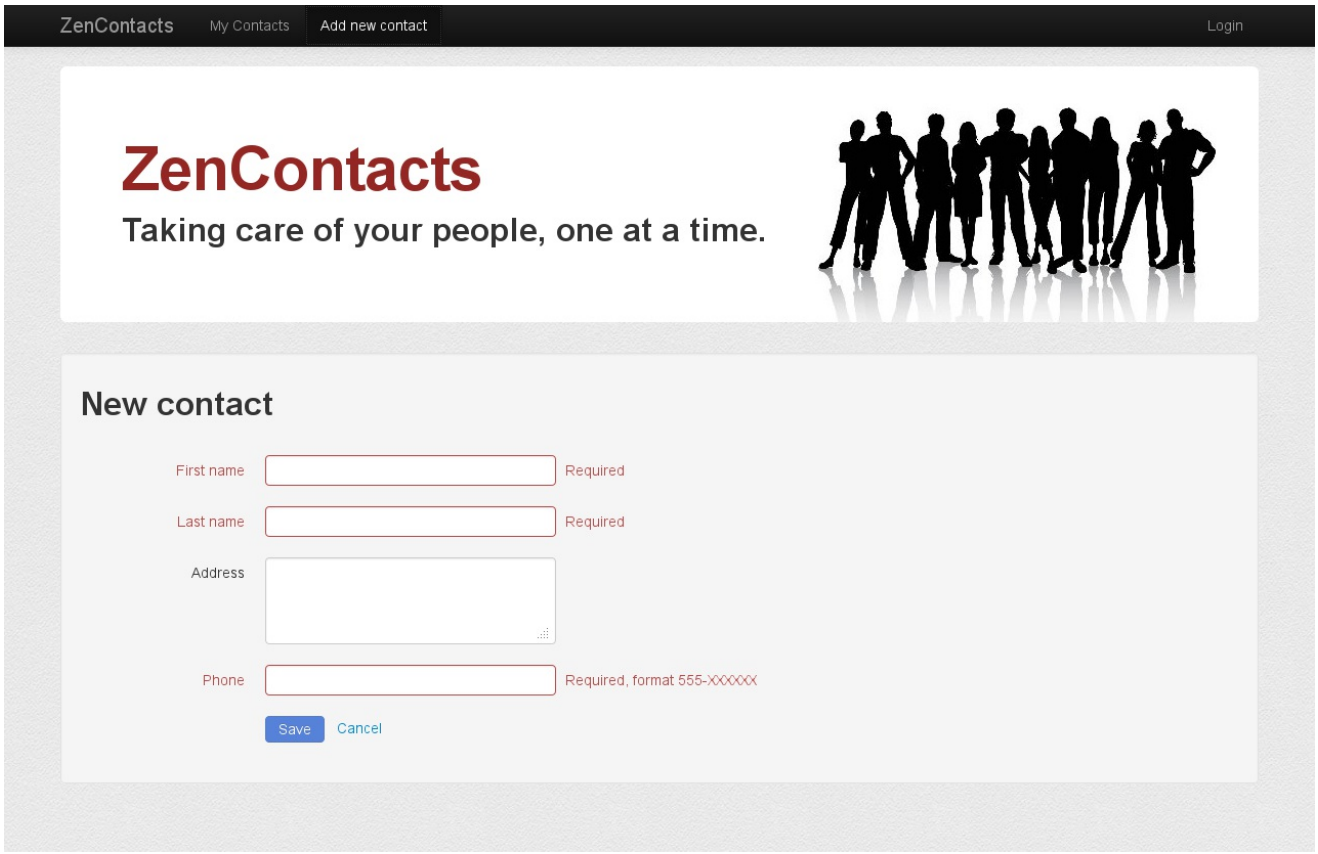
- Le nom et le prénom sont obligatoires.
- Le téléphone est obligatoire, au format `555-[A-Z]{5,6}`.

Instructions :

- Appliquez la directive `required` aux champs de saisie concernés pour les rendre obligatoires.

- Appliquez la directive `ng-pattern` au champ de saisie du téléphone pour valider son format.

Ajoutons un peu de feedback en cas d'erreur. Twitter Bootstrap propose une classe CSS `has-error` que nous allons utiliser, permettant de colorer en rouge les champs en erreur. Ensuite, nous allons ajouter à droite de chaque champ un petit message d'erreur visible uniquement en cas de saisie non conforme.



Pour chaque champ à valider :

- Modifiez sa balise `<div class="form-group">` englobante, afin de lui ajouter dynamiquement la classe CSS `has-error` en cas de saisie erronée (basez-vous sur la propriété `$error` du champ pour vérifier son état).
- Ajoutez, à la suite du champ, une balise `span` contenant un message d'erreur en lui appliquant la classe CSS `help-block`. Employez la directive `ng-show` pour ne l'afficher qu'en cas d'erreur.
- Ajoutez la classe CSS `control-label` à chaque balise `label` du formulaire.

Pour empêcher la soumission du formulaire tant qu'il reste des champs en erreur, nous allons également désactiver conditionnellement le bouton `Save`.

- Sur le bouton `Save` du formulaire, appliquez la directive ng-disabled. Vous pouvez interroger la propriété `$invalid` du formulaire pour vérifier son état général de validité.

Pour finir, testez le formulaire et vérifiez la bonne validation des champs de saisie.

TP7 : Communication avec un serveur

AngularJS offre deux API pour effectuer des appels AJAX : `$http` et `$resource`. Nous allons utiliser la première, bas niveau, pour importer nos contacts directement depuis le fichier `contacts.json` ; puis nous mettrons en place la seconde, plus haut niveau, pour communiquer avec un serveur exposant nos contacts sous la forme d'une ressource REST.

Objectifs :

- Utiliser le service `$http` pour effectuer des appels AJAX.
- Consommer des ressources REST avec le service `$resource`.

Effectuer des appels AJAX avec $http

Actuellement, les contacts sont déclarés de manière statique dans un tableau. Il serait plus intéressant de les lire depuis une ressource externe, par exemple le fichier `contacts.json`. Le service `$http` permet d'émettre des requêtes AJAX. En raison de leur caractère asynchrone, leur résultat n'est pas disponible immédiatement. Il est possible de passer une fonction callback en cas de succès, qui sera appelée quand les données seront prêtes, mais cette fonctionnalité est dépréciée depuis la version 1.4.4 d'AngularJS. Désormais, il est recommandé d'utiliser les promesses, retournées par le service `$http`.

Dans le service `ContactService` :

- Déclarez une dépendance vers le service `$http`.
- Supprimez l'initialisation statique du tableau des contacts.
- Dans la méthode `getAllContacts()`, utilisez le service `$http` pour émettre une requête GET vers l'URL `/rest/contacts` et retournez simplement la promesse rendue.

Dans le contrôleur `ContactListController` :

- Utilisez la méthode `then` de la promesse retournée par `getAllContacts` pour obtenir la liste des contacts chargée par le service `ContactService`. Stockez cette liste dans la propriété `contacts` du contrôleur.

Vérifiez que l'application se comporte toujours normalement, et que les contacts ont bien été chargés depuis le fichier.

Puis faites de même pour `getContactById` avec GET `/rest/contacts/:id`, `saveContact` avec PUT `/rest/contacts/:id` et POST `/rest/contacts`.

Consommer une ressource REST avec $resource

Cette application expose une API de type REST proposant les mêmes fonctionnalités que notre service `ContactService`. Les données échangées seront représentées au format JSON.

Maintenant que la ressource est exposée côté serveur, nous pouvons la consommer côté client. Il suffit pour cela de déclarer une ressource AngularJS; il s'agit d'une extension d'AngularJS, qui doit être importée séparément.

- Dans la page hôte, importez la librairie javascript `angular-resource.js` (fournie dans le répertoire `/src/vendor/angular/`) à la suite de la librairie `angular.js`.

Nous pouvons maintenant déclarer une ressource AngularJS nommée `ContactResource`. Dans le module `zenContact.core` :

- Déclarez l'injection du module `ngResource` fourni par la librairie importée ci-dessus.
- Déclarez une Ressource AngularJS nommée `ContactResource`, en utilisant la fonction `factory` et en injectant le service `$resource` pour interagir avec les données exposées en REST à l'URL `/rest/contacts/:id`.

Il ne nous reste plus qu'à utiliser notre ressource `ContactResource` dans les contrôleurs, à la place de l'actuel `ContactService` statique. Dans chaque contrôleur :

- Remplacez la dépendance vers `ContactService` par une dépendance vers la ressource `ContactResource`.
- Utilisez les méthodes adéquates de `ContactResource` pour consulter ou mettre à jour les contacts côté serveur.

Pour terminer, vérifiez que l'application fonctionne toujours correctement. Utilisez la vue `Réseau` des outils de développement du navigateur pour constater que les données sont bien échangées avec le serveur.

TP8 : Directives

Nous allons enchaîner sur l'ajout de directives personnalisées. Leur intérêt fonctionnel peut être discutable mais l'attention a surtout été apportée sur la progression en terme de complexité entre la première et la dernière.

De plus, si certaines peuvent être gadget dans cette application, il s'agit d'illustration de mise en œuvre de directive qui peuvent s'avérer tout à fait concrète dans d'autres applications.

Ce TP requiert quelques connaissances jQuery et Bootstrap. Une référence des API à connaître est disponible en annexe.

Directive auto-height

Mettons que dans la vue du formulaire, nous souhaitions que le cadre du formulaire s'ajuste à la hauteur de la fenêtre du navigateur.

Définir une hauteur de 100 % en CSS ne fonctionne pas : les principes de positionnement du CSS supposent une page extensible verticalement et ne fonctionnent pas avec des hauteurs en pourcentage.

Le seul mécanisme pour utiliser une hauteur de 100 % est d'utiliser du JavaScript pour demander au navigateur quelle est la taille de la fenêtre à un moment donnée.

Le plus souvent, nous ne souhaitons pas utiliser réellement 100 % de la hauteur de la page mais 100 % moins une valeur fixe qui correspond à une barre de menu de hauteur fixe. Nous utiliserons donc la valeur de l'attribut correspondant à la directive pour paramètrer la valeur à soustraire pour la hauteur de l'élément.

- Créez un fichier `/src/app/components.module.js` qui contiendra la déclaration du module `zenContact.components`.
- Créez un fichier `/src/app/components/directives/auto-height.directives.js` pour ajouter une nouvelle directive `auto-height` au module `zenContact.components`.
- Utilisez la fonction directive sur le module avec le nom de la directive et une fonction pour la définir en second paramètre.
- Pour cette première directive, nous n'utiliserons pas l'API complète mais simplement sa fonction `link`.
- Dans la fonction donnée en retour de la définition de la directive, prendre en paramètre les variables `scope`, `element` et `attributes`.
- Dans la fonction, définir une fonction qui change la hauteur de l'élément associé à la directive par la hauteur souhaitée.
- Associez cette fonction à l'événement de modification de la hauteur de la fenêtre (utiliser jQuery).
- Lancez également cette fonction immédiatement pour que la taille soit ajustée dès le chargement de la directive et sans redimensionnement de la fenêtre du navigateur.

Une fois cette directive définie, pensez à charger le fichier `auto-height.directives.js` dans `index.html` (ainsi que le

module `components.module.js`).

Ajoutez également `jquery.js` dans les balises scripts. Attention, il faut impérativement que la balise soit positionnée avant celle d'Angular, en effet, il n'y a que comme cela qu'Angular remplacera jqLite (la version allégée de jQuery embarquée dans Angular) par jQuery.

Dans la vue `edit.html`, utilisez la directive au niveau de la `div` avec la classe `well`. Ajustez la valeur de l'attribut pour que le cadre prenne la bonne hauteur.

Directive auto-popup

Une autre directive, régulièrement utile dans un projet, pourra compléter avantageusement cette application : la directive auto popup. Elle permet d'associer très simplement l'ouverture d'une popup lorsque l'on clique sur un élément.

Le contenu de la popup sera défini par une URL dans la valeur de l'attribut. La directive aura pour objectif de lancer une requête Ajax pour en récupérer le contenu et l'insérer dans une popup.

Cette directive peut toujours se suffire de la définition simplifiée avec la fonction `link` mais demande par contre un peu plus de code jQuery et Bootstrap à l'intérieur.

Créez le fichier `/src/app/components/directives/auto-popup.directives.js` pour ajouter une nouvelle directive auto-popup au module `zenContact.components` :

- Utilisez la fonction directive sur le module avec le nom de la directive et une fonction pour la définir en second paramètre.
- Avant de retourner la fonction `link`, créez à l'aide de jQuery un élément détaché du DOM servant de cadre à une popup Bootstrap.
- Sélectionnez dans cet élément, l'élément qui recevra le contenu de la popup.
- Dans la fonction donnée en retour de la définition de la directive, prendre en paramètre les variables `scope`, `element` et `attributes`.
- Associez à l'événement clic sur l'élément, le déclenchement du traitement.
- Lancez la requête Ajax avec le service `$http` pour récupérer le contenu de la popup sur le serveur.
- Au retour des données, positionnez le contenu de la popup dans l'élément préparé précédemment, puis utiliser l'API de Boostrap pour afficher la popup.

Utilisez cette nouvelle directive dans l'en-tête du site avec un lien `about` qui permet d'afficher un contenu `about.html`, créé pour l'occasion.

Directive Markdown

Cette nouvelle directive va permettre de saisir une description pour un contact. Cette description sera saisie dans un champ textarea multiligne. Nous enrichirons ce champ de saisie avec le format Markdown.

Ainsi, à l'édition d'un contact dans le formulaire, il sera possible de compléter une description qui aura une prévisualisation automatique, affichée à côté. Le rendu de cette prévisualisation sera au format Markdown.

Pour implémenter cette directive, nous aurons besoin de l'API complète de réalisation d'une directive. Ainsi, plutôt que de rendre uniquement la fonction `link`, il faudra retourner un objet contenant la description détaillée de la directive. Ici, il n'y aura pas traitement à effectuer sur le DOM. Ainsi, pour appliquer les bonnes pratiques en vigueur, nous implémenterons la logique de la directive dans un contrôleur associé, et non dans sa fonction `link`.

Cette fois ci, la directive sera appelée au niveau de la balise et non plus en tant qu'attribut comme c'était le cas précédemment.

Créez le fichier `/src/app/components/directives/markdown.directives.js`, pour ajouter une nouvelle directive `markdown` au module `zenContact.components` :

- Utilisez la fonction directive sur le module avec le nom de la directive et une fonction pour la définir en second paramètre.
- La fonction de définition doit, cette fois ci, rendre un objet.
- Paramétrez le fait que la directive fonctionne en tant que balise et non pas en tant qu'attribut.
- Ajoutez un template qui correspondra à une div contenante, un textarea qui permettra la saisie et une div qui réalisera la prévisualisation du résultat.
- Utilisez un scope privé et les propriétés `controllerAs` et `bindToController` dans laquelle on va binder la valeur de la propriété `ng-model` de l'élément.
- Définissez un contrôleur ayant pour objectif de surveiller la donnée associée au textarea pour déclencher un traitement à chacune de ses modifications. Ajouter ce contrôleur au module `zenContact.components` puis à la directive `markdown`.
- Utilisez la librairie `Showdown.js` fournie dans le projet pour transformer le code Markdown en HTML.

```
var converter = new Showdown.converter();
var html = converter.makeHtml(content);
```

- Pour afficher le résultat dans le template, stockez ce dernier dans une propriété du contrôleur, puis ajoutez l'attribut `ng-bind-html` au conteneur dans le template. Attention, `ng-bind-html` requiert que le code HTML soit sécurisé. Angular dispose du service `$sanitize` (module `ngSanitize`) pour cela. Toutefois, si le code est de confiance, comme c'est le cas dans le cadre de ce TP, on peut plus simplement appeler `$sce.trustAsHtml(html)` (en injectant le service `$sce` au contrôleur).

Une dernière librairie à inclure : `/src/vendor/showdown/Showdown.js`.

Ajouter ce champ au formulaire d'édition pour la saisie d'une propriété description.

TP9 : Filtres

Pour enrichir le système de recherche de la page d'accueil, nous allons modifier le filtre qui est utilisé pour, dans un premier temps, limiter les recherches au prénom et au nom du contact et, dans un second temps, être permissif face à d'éventuelles fautes de frappes.

Filtre sur le prénom et le nom du contact

La fonction à fournir au filtre `filter` doit être accessible depuis la vue, nous allons donc la localiser dans le contrôleur `ContactListController`.

- Positionnez dans le contrôleur une fonction `nameFilter` qui prend en paramètre un contact. Cette fonction sera appelée autant de fois qu'il y a de contacts dans la liste.
- Bien penser aux cas limites, notamment lorsqu'aucune recherche n'a encore été saisie.
- Rendre `true` si la propriété `firstName` ou `lastName` du contact *matche* la recherche courante, `false` sinon.
- Modifiez la vue `list.html` dans la directive `ng-repeat`, remplacez le paramètre `filter:vm.search` par `filter:vm.nameFilter`.

Note : afin de rendre le filtre insensible à la casse, il est possible d'utiliser une expression régulière permettant de vérifier si le champ de recherche matche la propriété du contact : `string.match(new RegExp(searchstring, "i"))`.

Filtre Fuzzy

Pour cette seconde implémentation nous allons utiliser les possibilités de spécialisation du filtre `filter`.

L'objectif ici est de s'intéresser aux mécanismes des filtres d'AngularJS et non pas aux algorithmes de comparaison approximative. Nous allons donc nous appuyer sur des outils externes.

Vous trouverez dans les fichiers :

- `/src/vendor/fuse/fuse.js` : une librairie qui implémente la recherche dite fuzzy. La documentation de l'API de cette librairie se trouve ici : http://kiro.me/projects/fuse.html. En résumé, l'API qui sera nécessaire est celle ci :

```
var fuse = new Fuse(array, {
 threshold: threshold,
 keys : ['firstName', 'lastName']
});
var result = fuse.search(search);
```

N'oubliez pas de les inclure dans l'application via des balises scripts dans le fichier `index.html`.

- Créez le fichier `/src/app/components/filters/fuzzy.filter.js` pour ajouter un nouveau filtre `fuzzy` au module `zenContact.components`.

- Utilisez la fonction `filter` sur le module avec le nom du filtre et une fonction pour la définir en second paramètre.
- La fonction doit rendre une fonction qui sera appelée pour chaque application du filtre.
- Cette fonction doit prendre en paramètre le tableau à filtrer. Il faut ensuite y ajouter un paramètre qui correspond à la valeur du champ de recherche et un dernier correspondant au seuil (*threshold*) pour Fuse.
- Appliquez les traitements nécessaires et rendre en retour de la fonction, le tableau réordonné en fonction du score, et filtré des résultats ne correspondant pas à la recherche.
- Dans la directive `ng-repeat` de la vue `list.html`, remplacez le paramètre `filter:vm.search` par `fuzzy:vm.search`.

Ne pas oublier d'ajouter le fichier `fuzzy.filter.js` dans le fichier `index.html`.

Il faut noter que la librairie Fuse retourne plusieurs contacts identiques si le terme recherché se trouve à la fois dans le prénom et le nom. Si le cas se présente, une erreur AngularJS de type « Duplicates in a repeater are not allowed » se produit.

Il est possible de tolérer les doublons via l'option `track by` de `ng-repeat`. Mais cela n'est pas le comportement souhaité dans notre cas, nous souhaitons plutôt éliminer les doublons avant affichage.

Pour cela, un filtre `unique` est disponible dans le projet Angular-UI :

- Inclure le script `/src/vendor/angular-ui/unique.js` dans le fichier `index.html`.
- Importez le module `ui.unique`.
- Paramétrez le filtre : `... | unique:'[id]'`.

TP10 : Authentification

Ce TP va nous permettre d'ajouter l'authentification sur la suppression d'un contact. En effet, l'appel REST DELETE sur un contact va retourner le code HTTP `401 Unauthorized` si l'utilisateur ne s'est pas authentifié au préalable.

Un Appel REST en POST sur `/rest/login/:user` va permettre d'enregistrer l'utilisateur et la réponse HTTP contiendra l'en-tête HTTP `Auth-Token` que vous pourrez stocker en cookie. Les cookies sont gérés via le service `$cookies`, composant du module `ngCookies`, lui-même fourni dans `angular-cookies.js`.

Il faudra ensuite ajouter cet en-tête HTTP à chaque requête soumise à l'authentification (soit DELETE d'un contact) dans notre cas.

- Créez une nouvelle vue `/src/app/sections/login/login.html` munie d'un formulaire comprenant les champs `User`, `Password`, et un bouton `Login` et configurez le routage pour accéder à cette dernière.
- Créez un fichier `/src/app/core/security/security.config.js` qui contiendra une fonction de configuration que l'on associera au module `zenContact.core`. Ajoutez un intercepteur à l'aide du service injecté `$httpProvider`. Ce service va vous permettre de définir les actions suivantes :
    - Lors d'un statut `401 Unauthorized`, redirigez l'utilisateur vers la page de *login*.
    - Lors d'une réponse OK, stocker le *token* de session en cookie (utilisation du service injecté `$cookies`).
- Ajoutez un bouton `Supprimer` sur la fiche contact qui récupère le token dans les cookies. Ajoutez le token dans l'en-tête HTTP puis appellez la suppression du contact.
- Sur la page principale, rendre dynamique le lien `Login` pour qu'il affiche `Logout` et qu'il déconnecte l'utilisateur lorsque l'on clique dessus, si ce dernier est identifié.
- Tentez de refactorer l'application en créant un service `/src/app/core/security/authenticator.service.js` qui encapsulera toute la logique d'authentification.
- Pouvez-vous faire en sorte que suite au login, l'utilisateur soit redirigé vers la page qui a déclenché l'authentification ?

ANNEXES

Serveur HTTP/REST

Prérequis

- Node.js (http://nodejs.org/ → Install)

Installation

- Une fois que l'archive fournit `TP.zip` est décompressé
- En ligne de commande, positionnez vous dans le répertoire `server`
- Lancez la commande `npm install`
- Lancez le serveur avec la commande `node.exe server.js`

API

Après le démarrage du server, l'URL http://localhost:8080 mettra a disposition les ressources suivantes :

methode	URI	corps	description
GET	/rest/contacts		list collection
POST	/rest/contacts	Object	create item
GET	/rest/contacts/:id		read item
PUT	/rest/contacts/:id	Object	update item
DELETE	/rest/contacts/:id		delete item
POST	/rest/login/:user		return token

Référence DOM, jQuery, Bootstrap

- Récupérer la hauteur de la fenêtre du navigateur : `window.innerHeight`
- Modifier la hauteur d'un élément : `element.height(newHeight)`
- Exécuter une fonction à chaque redimensionnement de la fenêtre du navigateur : `angular.element(window).resize(func)`
- Créér un élément HTML à l'extérieur du DOM : `angular.element(html)`
- Trouver un sous-élément : `element.find(cssSelector)`

- Insérer du contenu HTML dans un élément : `element.html(content)`
- HTML pour une popup Bootstrap :

```
<div class="modal fade" tabindex="-1">
  <div class="modal-dialog">
    <div class="modal-content"><div class="modal-body"></div></div>
  </div>
</div>
```

- Afficher un élément en tant que popup : `element.modal('show')`