Au secours mon code Angular est pourri

Speaker : **Thierry Chatel** – consultant formateur Angular sur Montpelier

Devoxx France 2014 – Jeudi 17/04/2014 – 14h30 - 15h20

Au début, sur une petite application, chose magique avec très peu de code.

Angular n'évite pas la complexité que peut avoir une grosse application

Papier « Not silver bullet » de 1986.

Angular n'empêche pas de faire de mauvais choix. N'indique pas comment structurer proprement une application.

Objectifs de la conf : garder une application maintenable.

Généralités

- Attention aux composants graphiques open source. Il y'en a des bons comme de très mauvais. Exemple de ng-grid à éviter à cause des perfs.
- Optimiser que lorsque c'est vraiment nécessaire. Le code reste simple lorsqu'il n'est pas optimisé. Cout négligeable sur le poste client.
- Bindings sur des fonctions sans stocker le résultat
- Le modèle de données est un vrai modèle objet. C'est un view model qui peut se différencier des entités côté serveurs. Ce n'est pas que le JSON envoyé par le serveur. On peut y mettre des fonctions. Pour les objets importants de l'application, se recréer un vrai modèle objet ;
- Service dans le scope ou une façade : évite de faire du passe plat au niveau contrôleur. Pas si sale que cela.

Structure d'une grosse application

- Structure par type ? (ex : contrôleurs avec les contrôleurs). Pas assez pertinent pour une appli Angular. Seulement au plus bas niveau. Nécessite tout d'abord un découpage fonctionnel.

Exemple :

```
front/
  profiles/
```

  repositories/
back/
  statistiques/
communs/

- Bonne pratique : 1 fichier JS = 1 module. Permet d'éviter l'ordre de dépendance des fichiers. Facilite l'écriture des tests

Exemple :

Front/
  front.js
  profiles/
    profiles.js
    profiles-controller.js
    profiles-service.js
    profiles-direcive.js

Les modules sont calqués sur l'arborescence des répertoires.
Possibilité de subdiviser encore plus.

- Dans la même structure arborescente, on peut mettre templates, tests, images … Plus simple pour les dév même si en Java les conventions sont différentes. Possibilité de déplacer / copier tout un module d'un seul coup.
- Routes : dans chaque branche
    - Ajouter des données à la définition de la route
    - Récupérables dans $route.curent : permet de stocker des infos comme le fil d'ariane
- Un service constant() (publié comme service + provider) utilisable dans config() pour définir le chemin des templates.

Contrôleurs

- Le Contrôleur a pour unique rôle d'initialiser le Scope. Pas de traitement sur les données dans les contrôleurs. Il fournit le lien entre services et scope.
- Les Contrôleurs doivent rester légers. Sinon ingérable.
- Pour éviter d'avoir de trop gros contrôleurs, on peut créer des contrôleurs locaux au niveau de la vue, et en particulier sur les structures répétées :
    - <li ng-repeat="xxx" ng-controller="yyy"/>
- Syntaxe « controller as » préconisée par Google
    - Le contrôleur ne publie rien dans le scope. C'est le contrôleur qui est mis directement dans le scope.
    - Intérêts :
        - mieux pour l'héritage entre les scopes (évite les problèmes)
        - plus explicite dans les templates
        - mais plus verbeux

Services

- Très importants car tout le code métier et une bonne partie du code de présentation sont codés dedans
- Chaque fonctionnalité doit être isolée dans un service
    - Rien qu'une fonctionnalité
    - Mais dans son intégralité : pas de code métier dans les templates et les contrôleurs
        - quantity(row) > 100 : mauvaise pratique ➔ orderSrv.isAlert(row)
- Les services permettent de conserver des données et proposent des traitements. Un service est un singleton. Les données propres à une vue sont renvoyés par les services. Possibilité d'utiliser un service de cache.
- Les services asynchrones doivent renvoyer des promesses
- Découper les services en couches. Regarder les services Angular (ex : $route utilise $url, $ressource utilise $http)
- Erreurs et notifications :
    - Un intercepteur $http
    - Surcharge du service $exceptionHandler
    - Un service pour les erreurs
    - Un service de log serveur
    - Un service de notification aux utilisateurs (ne sert pas exclusivement aux erreurs)
- Comment gérer l'utilisateur connecté ?
    - Créer un service user contenant les données concernant les utilisateurs
        - Ce service peut être injecter dans les contrôleurs et les services
    - Publier le service dans le scope afin de pouvoir l'utiliser dans les templates

Promesses

- Pratique pour les cas simples. API indispensable pour les appels asynchrones
- Fonctionne aussi bien avec du code synchrone et asynchrone
- Une seule méthode then() à connaitre
- Le secret est de ne travailler qu'avec des promesses, jamais les résultats
- En cas d'échec de l'opération asynchrone, renvoyer une promesse en erreur ou lever une exception

Filtres

- Ne jamais modifier les données en entrée. Le filtre doit toujours créer de nouvelles données.
- Eviter de prendre des noms de propriétés, mais parser / évaluer des expressions
    - users | orderBy :'profile.name' :false
    - fait en 2 lignes : service $parser
- Ne pas dupliquer le code des filtres. Garder le filtre dans une variable. Exemple :
    - Item in filtered = (list | filter :search)
    - Filtered.length

Directives

- Privilégier les bindings aux manipulations de DOM. Limiter au maximum l'utilisation de jQuery.
- S'appuyer sur le HTML plutôt que de le remplacer (syndrome JSF). Ajouter une directive au html existant plutôt que créer une directive qui génère du HTML
- Attributs HTML = interface de la directive. Bien les définir
- Pas d'héritage implicite (données du scope parent).
- Utilisation d'un contrôleur de directives pour établir un lien entre des directives parents / enfants
- Expression dans les attributs.
- Utiliser des scopes isolés afin de ne plus utiliser le nom du paramètre
- On peut utiliser des directives sur plusieurs éléments (syntaxe –start et –end). Valable pour ses propres directives. Permet de recevoir un ensemble d'éléments dans le code de la directive.

- Eléments HTML standard : directives input, form, script plutôt que de rechercher les éléments dans la page. Permet de déclencher des évènements. Pratique dans certains cas d'usage (ex : colorer des éléments)

Tests

- Il faut maintenir le code. Soigner le code du test. Eviter le copier / coller.
- Avec Jasmine, on peut imbriquer un describe pour factoriser dans un beforeEach
- Tests E2E avec Protactor : factoriser des ElementFinder et des fonctions

Conclusion

- Faire simple (ce qui reste compliqué)
    - Partir du HTML
    - Bien connaitre JS afin de profiter de sa souplesse. Se former.
    - Les bonnes pratiques de programmation objet restent valables avec Angular

Angular propose tout ce qui faut.