AngularJS - Jour 4

0 - Programme

Au programme:

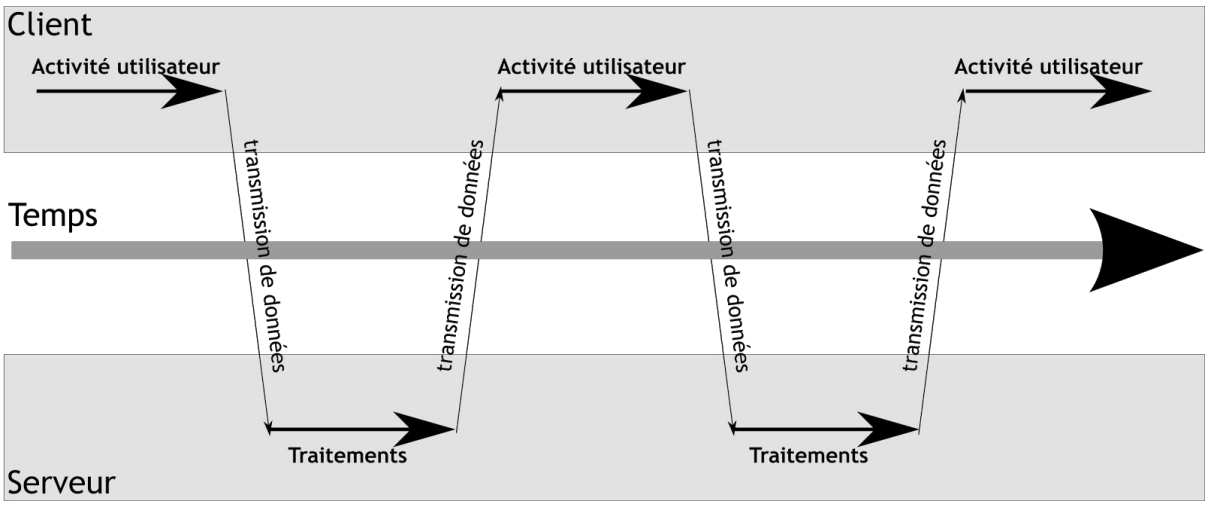
- Les promesses
- Les providers AngularJS
- La communication avec le serveur
- TP
- Élaboration d'un plan de TP personnalisé (20' / personne)

1 - Les promesses

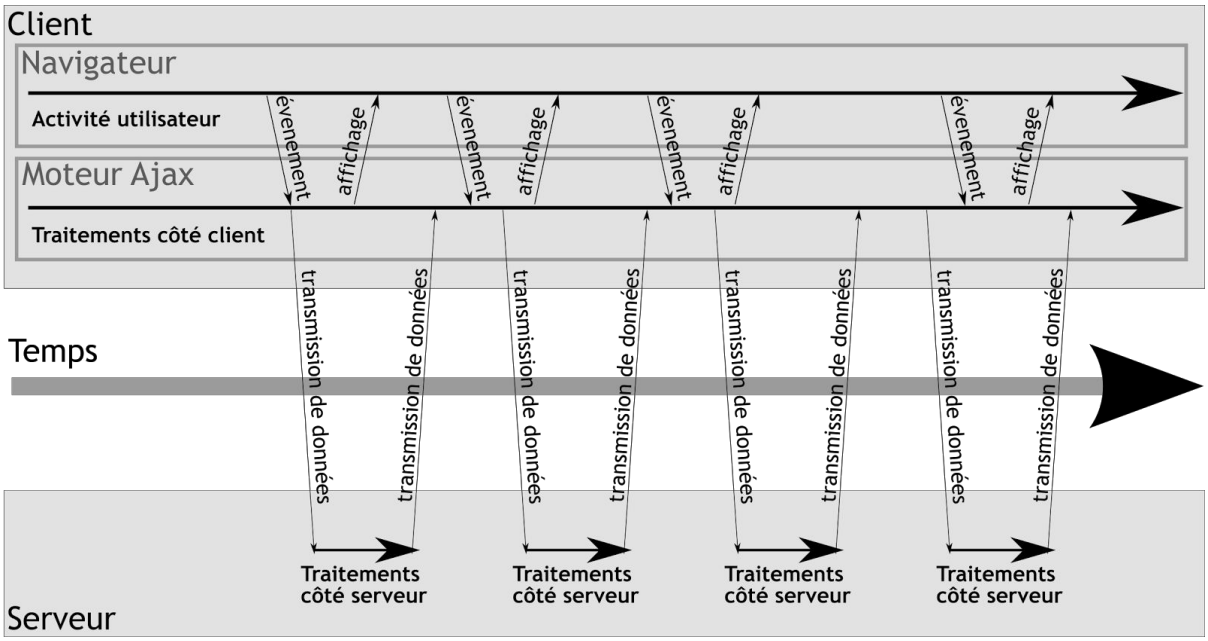
Kezako

Le JavaScript est un langage asynchrone, l'inverse du séquentielle.

Non asynchrone : Ancienne application avec un appel serveur à chaque activité utilisateur



Asynchrone : Single Page Application avec une utilisation d'AJAX



Les promesses permettent de lancer l'exécution d'un code asynchrone et de traiter son retour au même endroit.

L'asynchrone pour une Web App

Les callbacks

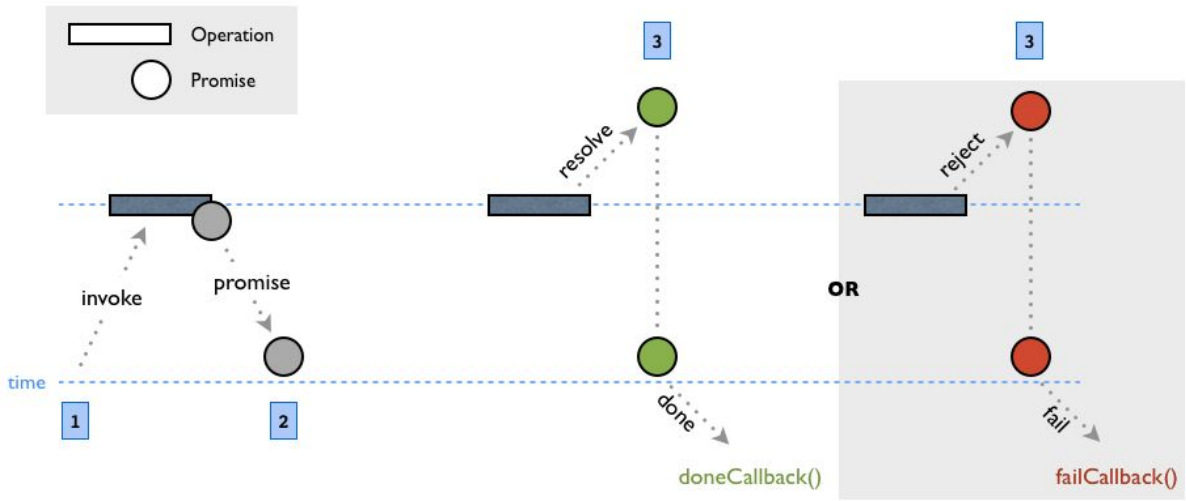
On peut gérer le code en asynchrone avec des callbacks.
Méthode utilisée jusqu'à récemment.
Peu lisible.

```
var myAsyncFunction = function (id, callback) {
  // Do some stuff
  setTimeout(function() {
    callback(); // Retour asynchrone
  }, 1000);
  return true; // Retour séquentielle
};
```

```
asncFunction1(function(err, result) {
    asncFunction2(function(err, result) {
        asncFunction3(function(err, result) {
            asncFunction4(function(err, result) {
                asncFunction5(function(err, result) {
                    // do something useful
                })
            })
        })
    })
})
```

Les promesses

Permets d'écrire en forme séquentielle comment traiter le résultat asynchrone.




```
function fetchData(id){
  return getDataFromServer(id)
          .then(transformData)
          .then(saveToIndexDB);
}
```

La gestion du résultat

Le code exécuté dans une promesses est protégé (try/catch).

On a 4 retours possible :

- resolve
- reject
- error
- notify

```
function fetchData(id){
  return getDataFromServer(id)
          .then(transformData, handleError)
```

```
            .then(saveToIndexDB);
}
```

Clôturer une promesse

Pour terminer une promesse et donner une réponse, on a deux possibilités: reject(msg) ou resolve(data).
reject(msg) refuse la promesse avec un message en string
resolve(data) valide la promesse avec la data (object, string, etc.)

La gestion des erreurs

Dans le second paramètre de la méthode then.

```
fetchData(1)
  .then(function(result){

  }, function(error){
    // exceptions dans transformData ou saveToIndexDB
  });
```

2 - Les providers AngularJS

Kezako

Il existe 6 providers angularJS:

Provider	**Singleton**	**Instantiable**	**Configurable**
Constant	Yes	Yes	No
Value	Yes	No	No
Service	Yes	No	No
Factory	Yes	Yes	No
Decorator	Yes	No?	No
Provider	Yes	Yes	Yes

Constant

Une **constante** ne peut être modifié (même pas un décorateur).

```
angular.module('app', []);
app.constant('API_URL', 'http://api.twitter.com');
app.controller('MyController', function (API_URL) {
  console.log(API_URL === 'http://api.twitter.com');
});
```

Value

Une **value** est simplement une valeur injectable (number, function, string, etc.).

```
angular.module('app', []);
app.value('myApiUrl', 'http://api.twitter.com');
app.controller('MyController', function (myApiUrl) {
  console.log(myApiUrl === 'http://api.twitter.com';
});
```

Service

C’est un singleton (donc n’existe qu’une seule fois dans l’application).
Service applique un new de la function du service.

```
angular.module('app', []);
app.service('userService', function($http) {
  this.getUser = function () {
    return $http.get('url.com/users');
  };
});
app.controller('MyController', function (userService) {
  var self = this;
  userService.getUser().then(function(res) {
    self.users = res;
  });
});
```

Factory

Identique à service hormis le fait qu’AngularJS n’instancie pas la factory

```
angular.module('app', []);
app.factory('userService', function ($http) {
  var _privateUser = [];
  return {
    getUser: function () {
      return $http.get('url.com/users').then(function(res) {
        _privateUser = res.data;
        return _privateUser;
      });
    }
});
app.controller('MyController', function (userService) {
  var self = this;
  userService.getUser().then(function(res) {
    self.users = res;
  });
});
```

Decorator

Les décorateurs permettent de modifier ou encapsuler d’autres providers (sauf les constants).

```
angular.module('app', []);
app.value('apiUrl', "api.com");
app.config(function ($provide) {
  $provide.decorator('apiUrl', function ($delegate) {
    return $delegate + '/v2';
  });
});
app.controller('MyController', function (apiUrl) {
  console.log(apiUrl, 'api.com/v2');
});
```

Provider

C’est la plus compliqué des providers.
Permet d’avoir une phase de configuration (un **factory** configurable).

```
angular.module('app', []);
app.provider('apiUrl', function () {
  var version;
  return {
    setVersion: function (version) {
      version = version;
    },
    $get: function () {
      return url = "api.com/" + version;
    }
});
app.config(function (apiUrlProvider) {
  apiUrlProvider.setVersion('v2');
});
app.controller('MyController', function (apiUrl) {
  console.log(apiUrl, 'api.com/v2');
});
```

3 - La communication avec le serveur

Kezako

AJAX est le coeur des Single Page Applications.
Une application client à besoin d'aller chercher des informations sur un serveur.

Communiquer avec $http

Ce service AngularJS peut être injecter en dépendance dans un contrôleur.
Il facilite la communication avec le serveur en HTTP (XMLHttpRequest ou JSONP).
Il est compatible avec tout les verbe HTTP : GET, POST, HEAD, DELETE, PUT, JSONP
Ce service utilise les Promises pour faciliter l'asynchrone.

```
$http.get('url/someUrl')
  .success(function(data, status, headers, config) {  })
  .error(function(data, status, headers, config) {  })

$http.post('/someUrl', {msg: 'Message sent to the server!'})
  .success(function(data, status, headers, config) {  })
  .error(function(data, status, headers, config) {  })

// Disponible : $http.get, $http.head, $http.post, $http.put, $http.delete,
$http.jsonp, $http.patch
```

Aller plus loin

https://docs.angularjs.org/api/ng/service/$http

Communiquer avec une API de type REST avec $resource

REST: http://goo.gl/FvfGMV
Pour résumer, REST permet de suive une convention de nommage des routes pour accéder aux ressources. Ce qui nous permet de faire des méthodes génériques de dialogue avec le serveur.
On utilise les méthodes suivantes : Get, Save, Query, Remove et Delete.
Le service $resource permets de facilement les mettre en place.

```
angular.factory("userService", function($resource) {
  return $resource('/users/:userId', {userId: '@id'});
```

```
});
```

1. Service **/users** est appelé
2. **:userId** sera remplacé par la valeur passé lors de l'appel de la méthode Get
3. @id permet de savoir quel champs mettra à jour le userId

Utilisation

Récupere une liste (query)

Méthode statique, un tableau d'objet :

```
$scope.users = [
  {userObject1},
  {userObject2},
  {userObject3}
];
```

Méthode dynamique avec la factory userService :

```
$scope.users = userService.query();
```

Récupérer une entrée (get)

```
$scope.user = userService.get({userId: 5});
```

Créer une entrée (save)

```
var user = new userService();
user.name = 'my name';
user.$save();
```

Modifier une entrée

Si une API RESTful, il faut définir une méthode *update* qui utilise la méthode PUT.

```
angular.factory("userService", function($resource) {
  return $resource('/users/:userId', {userId: '@id'}, {'update', {method: 'PUT'}
});
});
//
$scope.user = userService.get({userId: 5});
$scope.user.name = 'new name';
$scope.user.$udapte();
```

4 - TP

Reprendre le TP précedent (Jour 3).

Ecrire un service User

- Créer un service *UserService* qui a pour rôle de gérer le model User.
  Ce service doit comporter:
  - Une méthode pour récupérer la collection d'utilisateur
  - Une méthode pour récupérer un utilisateur par son ID
  - Une méthode pour ajouter un utilisateur
  - Une méthode pour supprimer un utilisateur
  - Une méthode pour connaître le nombre d'utilisateur

Mettre à jours l'application

- Mettre à jour les contrôleurs pour utiliser les données venant du service
    - Route /user/:id
    - Route /user

Utiliser les données d'un serveur

- Créer une constante contenant l'URL de l'api https://randomuser.me/api/
- Mettre à jour le service *UserService* pour utiliser **$http** et l'API à la place de la collection d'utilisateur existantes.
- Faire un appel à la route http://api.randomuser.me/?results=2000 en **GET**

Architecture composant

Le but de cette architecture est de ranger tout les fichiers par fonctionnalités en composant. Il faut donc ranger les différents composants dans différents modules

projet/
- index.html
- **vendors**/
--- angular.js
--- jquery.js
--- etc.
- **app**/
--- app.js ⇒ *angular.module('myApp', ['myApp.home', 'myApp.user'])*
--- **components**/
----- **home**/
------- home.js (controller) ⇒ *angular.module('myApp.home', [])*
------- home.html (template)
------- homeFilter.js
------- etc.
----- **user**/
------- user.js (controller) ⇒ *angular.module('myApp.user', [])*
------- user.html (template)
------- userService.js (récupération données serveur)
------- userDirective.js