# Atelier ES6

Création d'un framework MVC en ES6 Décrire tout ce que nous allons faire Bien sûr nous ne verrons pas tout si nous n'arrivons pas au bout : pinguer moi ici : ph.charriere@gmail.com

Notre framework s'appelera "skeleton"

# ES6 - Partie 1

*Ou, tout ce dont vous avez besoin pour le 1er exercice.*

## Ce que nous attendions tous: les Classes

class Dog {  
 constructor (name="cookie") { /\* mot-clé constructor + valeurs par défaut \*/  
 this.name = name; /\* propriétés définies dans le constructeur \*/  
 }  
 wouaf () { /\* pas de mot-clé function \*/  
 console.log(this.name + ": wouaf! wouaf!");  
 }  
}  
  
let wolf = new Dog();  
wolf.wouaf()

## Bien sûr, on peut "hériter"

class Animal {  
 constructor(name) {  
 this.name = name;  
 }  
}  
  
class Dog extends Animal {  
 constructor (name="cookie") {  
 super(name) /\* on appelle le constructeur de la classe mère \*/  
 }  
 wouaf () {  
 console.log(this.name + ": wouaf! wouaf!");  
 }  
}  
  
let wolf = new Dog();  
wolf.wouaf()

## Import / Export: ou la modularisation facile

### Classe Animal dans Animal.js

class Animal {  
 constructor(name) {  
 this.name = name;  
 }  
}  
export default Animal;

### Classe Dog dans Dog.js

import Animal from './Animal'; /\* pas d'extension .js \*/  
  
class Dog extends Animal {  
 constructor (name="cookie") {  
 super(name) /\* on appelle le constructeur de la classe mère \*/  
 }  
 wouaf () {  
 console.log(this.name + ": wouaf! wouaf!");  
 }  
}  
export default Dog;

### Utilisation de Dog dans main.js

import Dog from './Dog'  
let wolf = new Dog();  
wolf.wouaf()

## Fat Arrow =>

/\* Avant \*/  
var sayHello = function(name) { return "hello " + name; }  
  
/\* Après \*/  
var sayHello = (name) => "hello " + name  
// ou var sayHello = (name) => { return "hello " + name; }  
sayHello("Bob Morane");

Remarques:

* pas "newable"
* pas d'objet arguments, à la place : "rest parameters"

var sayHello = (...people) => people.forEach((somebody) => console.log("Hello", somebody));  
  
sayHello("Bob Morane", "John Doe", "Jane Doe");

### Fat Arrow + Lexical this binding

La valeur de this est déterminée par l'endroit où se trouve la "Arrow function"

/\* Avant \*/  
function Animal(friends) {  
 this.friends = friends;  
 this.hello = function(friend) {  
 console.log("hello " + friend);  
 }  
 this.helloAll = function() {  
 this.friends.forEach(function(friend) {  
 this.hello(friend); /\* error \*/  
 });  
 }  
}  
  
var wolf = new Animal(["rox", "rookie"]);  
wolf.helloAll();  
  
  
/\* Correction : bind \*/  
function Animal(friends) {  
 this.friends = friends;  
 this.hello = function(friend) {  
 console.log("hello " + friend);  
 }  
 this.helloAll = function() {  
 this.friends.forEach(function(friend) {  
 this.hello(friend);  
 }.bind(this)); // ou var that = this  
 }  
}  
  
var wolf = new Animal(["rox", "rookie"]);  
wolf.helloAll();  
  
  
/\* Après \*/  
class Animal {  
 constructor (friends=[]) {  
 this.friends = friends;  
 }  
 hello(friend) { console.log("hello " + friend); }  
 helloAll() {  
 this.friends.forEach((friend) => this.hello(friend));  
 }  
}

**Remarque**: ne pas utiliser les fat arrows pour définir les méthodes d'un objet:

var bobMorane = {  
 friends: ["John Doe","Jane Doe"],  
 getFriends:()=>this.friends,  
 getAllFriends: function() { return this.friends; },  
 sayHelloToAll: function() { this.friends.forEach((friend)=>console.log("hello", friend)) }  
}  
  
console.log(bobMorane.friends, bobMorane.getFriends(), bobMorane.getAllFriends());  
// !!! -> bobMorane.getFriends() is undefined  
bobMorane.sayHelloToAll();

## let versus var

### var

var bob = {  
 firstName:"Bob", lastName:"Morane"  
}  
  
var bob = { foo:"foo"}

### let : c'est plus propre

let bob = {  
 firstName:"Bob", lastName:"Morane"  
}  
  
let bob = { foo:"foo"} /\* Duplicate declaration, bob \*/

## Tips: Getter & Setter (pas spécifique à ES6)

let bob = {}  
  
Object.defineProperty(bob, "Name", {  
 get: function (){  
 console.log("get value:", this.name) /\* ! nous avons Name et name \*/  
 return this.name;  
 },  
 set: function (value) {  
 console.log("set value to:", value)  
 this.name=value;  
 }  
});  
  
bob.Name = "BOB MORANE";  
console.log(bob.Name);

## Exécuter du code ES6: Préparation de index.html

<script src="node\_modules/traceur/bin/traceur.js"></script>  
  
<script>  
 traceur.options.experimental = true;  
</script>  
  
<script>  
 System.import('js/main').catch(function (e) {console.error(e);});  
</script>

**PS**: il y a d'autres méthodes, mais pour apprendre, c'est la plus simple

# ES6 - Partie 1 : Exercice 1: Models & Collections

## Le minimum pour commencer

### Côté serveur

* on fonctionne en "mode http", il nous faut donc un serveur http, j'ai pris node + express
* j'utilise NeDb pour "simuler" une base de données, c'est un MongoDb-like avec du fichier plat
* j'ai besoin de npm

#### package.json

{  
 "name": "es6",  
 "description": "es6",  
 "version": "0.0.0",  
 "dependencies": {  
 "body-parser": "1.0.2",  
 "express": "4.1.x",  
 "nedb": "0.10.5"  
 }  
}

Et je vous ai préparé un ficihier app.js avec les API REST qui vont bien pour faire du CRUD

### Côté client

Notre webapp sera dans le répertoire /public

Elle sera composée de:

* index.html dans /public, déjà préparée pour vous
* main.js dans public/js qui contiendra le code principal de notre application (essentiellement des tests qui sont déjà **codés**)
* Model.js et Collection.js dans public/js/skeleton : **vous allez devoir les coder** pour que les tests fonctionnent
* Human.js et Humans.js dans public/js/app/models : **vous allez devoir les coder** pour que les tests fonctionnent

Là aussi j'ai besoin de npm pour installer les dépendances :

* **Traceur**, qui va donc nous permettre d'exécuter du code ES6 (nous allons faire de la "transpilation online")
* **QUnit**, pour faire nos tests unitaires

#### package.json

{  
 "name": "es6",  
 "description": "es6 formation",  
 "author": "@k33g\_org",  
 "license": "MIT",  
 "dependencies": {  
 "traceur": "0.0.65"  
 },  
 "devDependencies": {  
 "qunitjs": "^1.15.0"  
 }  
}

## Exercice

Créer nos 1ers modèles & collections :

* public/js/skeleton/Model.js
* public/js/skeleton/Collection.js
* public/js/app/models/Human.js
* public/js/app/models/Humans.js

**Remarque**: les spécifications sont décrites dans les fichiers

cd 01-models  
node app.js  
http://localhost:3000

# Correction

## Model.js

/\*--- model ---\*/  
  
/\* === Spécifications ===  
une classe Model  
  
Paramètres du constructeur:  
  
- fields, valeur par défaut {}, contiendra les "champs" du model, ex: {firstName:"Bob", lastName:"Morane"}  
- observers, valeur par défaut []  
  
Propriétés:  
  
- fields : initialisé par le paramètre correspondant du constructeur  
- observers : initialisé par le paramètre correspondant du constructeur  
  
Méthodes:  
  
- addObserver (observer)  
- notifyObservers (context)  
- get (fieldName), va lire la valeur d'un champ dans fields  
- set (fieldName, value), va modifier la valeur d'un champ dans fields  
- toString (), retourne une représentation json de fields  
  
un observer est juste un objet avec une méthode update  
donc notifyObservers execute la méthode update de tous les observers avec context en paramètre  
  
\*/  
  
class Model {  
 constructor (fields={}, observers=[]) {  
 this.fields = fields;  
 this.observers = observers;  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
  
 get (fieldName) {  
 return this.fields[fieldName];  
 }  
  
 set (fieldName, value) {  
 this.fields[fieldName] = value;  
 return this;  
 }  
  
 toString () {  
 return JSON.stringify(this.fields)  
 }  
  
}  
  
export default Model;

## Collection.js

/\*--- collection ---\*/  
  
/\* === Spécifications ===  
 une classe Collection  
  
 Paramètres du constructeur:  
  
 - model : ce sera le type de la collection (une classe qui héritera de Model), pas de valeur par défaut  
 - models : un tableau, contiendra les instances de modèles, valeur par défaut : []  
 - observers, valeur par défaut []  
  
 Propriétés:  
  
 - model : initialisé par le paramètre correspondant du constructeur  
 - models : initialisé par le paramètre correspondant du constructeur  
 - observers : initialisé par le paramètre correspondant du constructeur  
  
 Méthodes:  
  
 - addObserver (observer)  
 - notifyObservers (context)  
 - toString (), retourne une représentation json de la propriété models  
 - add (model), ajoute un model à models et notifie les observers avec un "contexte" égal à {event: "add", model: model}  
 - each (callbck) : parcourir les models et exécuter callbck pour chacun (et passer le modèle en paramètre à callbck)  
 - filter (callbck) : retourner un tableau de modèle filtré selon callbck  
 - size () : retourner le nombre de modèles dans la collection  
   
 un observer est juste un objet avec une méthode update  
 donc notifyObservers execute la méthode update de tous les observers avec context en paramètre  
  
 \*/  
  
  
class Collection {  
 constructor (model, models = [], observers = []) {  
 this.model = model;  
 this.models = models  
 this.observers = observers;  
 }  
  
 toString () {  
 return JSON.stringify(this.models);  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
  
 add (model) {  
 this.models.push(model);  
 this.notifyObservers({event: "add", model: model});  
 return this;  
 }  
  
 each (callbck) {  
 this.models.forEach(callbck)  
 }  
  
 filter (callbck) {  
 return this.models.filter(callbck)  
 }  
  
 size () { return this.models.length; }  
  
}  
  
export default Collection;

## Human.js

/\* === Spécifications ===  
 une classe Human qui hérite de Model  
   
 Paramètres du constructeur:  
  
 - fields, valeur par défaut {firstName:"John", lastName:"Doe"},   
   
 Propriétés:  
   
 - initialiser la propriété fields de la classe mère avec le paramètre du constructeur  
  
 Getters & Setters pour :  
  
 - firstName -> set or get of this.fields.firstName  
 - lastName -> set or get of this.fields.lastName  
   
 Méthodes:  
  
 - sans objet  
  
 \*/  
import Model from '../../skeleton/Model';  
  
class Human extends Model {  
 constructor (fields = {firstName:"John", lastName:"Doe"}) {  
 //superclass's constructor invocation  
 super(fields);  
  
 //Getters and Setters  
 Object.defineProperty(this, "firstName", {  
 get: function (){ return this.fields["firstName"]} ,  
 set: function (value) { this.fields["firstName"]=value; }  
 });  
  
 Object.defineProperty(this, "lastName", {  
 get: function (){ return this.fields["lastName"]} ,  
 set: function (value) { this.fields["lastName"]=value; }  
 });  
  
 }  
}  
  
export default Human;

## Humans.js

/\* === Spécifications ===  
une classe Humans qui hérite de Collection  
  
Paramètres du constructeur:  
  
- humans, un tableau de modèles  
  
Propriétés:  
  
- initialiser la propriété model de la classe mère avec le type Human  
- initialiser la propriété models de la classe mère avec le paramètre humans du constructeur  
  
Méthodes:  
  
- sans objet  
\*/  
  
import Collection from '../../skeleton/Collection';  
import Human from './Human';  
  
class Humans extends Collection{  
  
 constructor (humans) {  
 super(Human,humans);  
 }  
}  
  
export default Humans;

# ES6 - Partie 2

## Les Promises!

let doSomeThing = new Promise((resolve, reject) => {  
  
 // faites quelque chose (asynchrone)  
  
 let allisfine = true; // essayez avec false  
   
 if (allisfine) {  
 resolve("Hello World!");  
 }  
 else {  
 reject(Error("Ouch"));  
 }  
});  
  
doSomeThing  
 .then((data) => { console.log(data); })  
 .catch((err) => { console.log(err); });

**Voir cet article**: <http://www.html5rocks.com/en/tutorials/es6/promises/>

## Interpolations de chaînes

### Template strings

let firstName = "Bob", lastName = "Morane";  
console.log(`Hello I'm ${firstName} ${lastName}`); // Hello I'm Bob Morane

### Multiline strings

let firstName = "Bob", lastName = "Morane";  
console.log(`  
Hello I'm   
 ${firstName}   
 ${lastName}  
`  
);   
/\*  
Hello I'm   
 Bob   
 Morane  
\*/

### Tagged template strings

let upper = (strings, ...values) => {  
 console.log(strings); // ["Hello I'm ", " ", "", raw: Array[3]]  
 console.log(values); // ["Bob", "Morane"]   
 let result = "";  
 for(var i = 0; i < strings.length; i++) {  
 result = result + strings[i];  
 if (i < values.length) {  
 result = result + values[i];  
 }  
 }  
 return result.toUpperCase();  
}  
let firstName = "Bob", lastName = "Morane";  
console.log(upper `Hello I'm ${firstName} ${lastName}`)  
/\*  
HELLO I'M BOB MORANE  
\*/

# ES6 - Partie 2 : Exercice 2: On parle avec le serveur

## Objectifs:

Donner la possibilité aux modèles et aux collections d'échanger des informations de persistence avec le serveur:

* compléter Request.js, La classe Request nous permettra de faire des requêtes ajax
* compléter Model.js
* compléter Collection.js
* Modifier Human.js
* Modifier Humans.js

## Exercice

**Remarque**: les spécifications sont décrites dans les fichiers

Stoppez l'exercice 1, puis:

cd ..  
cd 02-models-sync  
node app.js  
http://localhost:3000

# Correction

## Request.js

/\*--- Request ---\*/  
/\* === Spécifications ===  
  
Ajouter 4 méthodes à la classe Request:  
  
- get() retourne une promise via sendRequest() avec la propriété method = "GET"  
- post(jsonData) retourne une promise via sendRequest() avec la propriété method = "POST" et la propriété data prend la valeur jsonData  
- put(jsonData) retourne une promise via sendRequest() avec la propriété method = "PUT" et la propriété data prend la valeur jsonData  
- delete() retourne une promise via sendRequest() avec la propriété method = "DELETE"  
  
\*/  
class Request {  
  
 constructor (url = "/") {  
 this.request = new XMLHttpRequest();  
 this.url = url;  
 this.method = null;  
 this.data = null;  
 }  
  
 sendRequest () { /\*json only\*/  
  
 return new Promise((resolve, reject) => {  
 this.request.open(this.method, this.url);  
 this.request.onload = () => {  
 // If the request was successful  
 if (this.request.status === 200) {  
 resolve(JSON.parse(this.request.response)); // JSON response  
 } else { /\* oups \*/  
 reject(Error(this.request.statusText));  
 }  
 }  
 // Handle network errors  
 this.request.onerror = function() {  
 reject(Error("Network Error"));  
 };  
  
 this.request.setRequestHeader("Content-Type", "application/json");  
 this.request.send(this.method === undefined ? null : JSON.stringify(this.data));  
 });  
 }  
  
 get () {  
 this.method = "GET";  
 this.data = {};  
 return this.sendRequest();  
 }  
  
 post (jsonData) {  
 this.method = "POST";  
 this.data = jsonData;  
 return this.sendRequest();  
 }  
  
 put (jsonData) {  
 this.method = "PUT";  
 this.data = jsonData;  
 return this.sendRequest();  
 }  
  
 delete () {  
 this.method = "DELETE";  
 this.data = {};  
 return this.sendRequest();  
 }  
}  
  
export default Request;

## Model.js

/\*--- model ---\*/  
  
/\* === Spécifications ===  
  
Remarques:   
  
- j'ai ajouté une propriété url à la classe Model,  
 Vous pouvez vérifier que Human.js a été modifié pour en tenir compte  
- lorsque je vais créer un modèle et le persister côté serveur,   
 un identifiant unique lui est affecté par la base de donnée  
 et le modèle "gagne" un nouveau champ \_id renseigné par la base:  
 j'ai donc créé une méthode : id() { return this.get("\_id");}  
 pour pouvoir récupérer la valeur de l'id du modèle  
  
Donc, complétez la classe en lui ajoutant:  
  
- une méthode save():  
 si this.id() == undefined, c'est une création (POST)  
 sinon c'est une mise à jour (PUT)  
 !!!: save() retourne une promise (grâce à Request)  
 ce qui nous permettra d'écrire quelque chose comme ceci:  
  
 let Olivia = new Human({firstName:"Olivia", lastName:"Dunham"});  
 Olivia.save().then((data) => {  
 console.log("Olivia", data);  
 }).catch((err) => {});  
   
 Remarque : côté node c'est ceci qui est appelé:  
 app.post("/models\_url", function(req, res) { ... });  
   
 ou (dans le cas d'une mise à jour)  
   
 app.put("/models\_url/:id", function(req, res) { ... });   
   
 !!!: Pensez à notifier les observers du modèle:  
   
 - quand c'est une création: {event: "create", model: this}   
 - quand c'est une mise à jour: {event: "update", model: this}  
  
- une méthode fetch(id) qui permet de retrouver un modèle par son identifiant (GET)  
 si le paramètre id n'est pas utilisé, utiliser this.id()  
 fetch(id) retourne une une promise (grâce à Request)  
  
 Remarque : côté node c'est ceci qui est appelé:  
 app.get("/models\_url/:id", function(req, res) { ... });  
  
 !!!: Pensez à notifier les observers du modèle:  
   
 - {event: "fetch", model: this}  
  
  
- une méthode delete(id) qui permet de supprimer un modèle par son identifiant (DELETE)  
 si le paramètre id n'est pas utilisé, utiliser this.id()  
 delete(id) retourne une une promise (grâce à Request)  
  
 Remarque : côté node c'est ceci qui est appelé:  
 app.delete("/models\_url/:id", function(req, res) { ... });  
  
 !!!: Pensez à notifier les observers du modèle:  
   
 - {event: "delete", model: this}  
  
 \*/  
  
import Request from './Request';  
  
class Model {  
 constructor (fields={}, url="/", observers=[]) {  
 this.fields = fields;  
 this.url = url;  
 this.observers = observers;  
 }  
  
 get (fieldName) {  
 return this.fields[fieldName];  
 }  
  
 set (fieldName, value) {  
 this.fields[fieldName] = value;  
 return this;  
 }  
  
 toString () {  
 return JSON.stringify(this.fields)  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
  
 /\*--- sync ---\*/  
 id() { return this.get("\_id");}  
  
 save () {  
 return new Promise((resolve, reject) => {  
 if (this.id() == undefined) {  
 // create (insert)  
 new Request(this.url).post(this.fields)  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "create", model: this});  
 resolve(data);  
 })  
 .catch((error) => reject(error))  
 } else {  
 // update  
 new Request(`${this.url}/${this.id()}`).put(this.fields)  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "update", model: this});  
 resolve(data);  
 })  
 .catch((error) => reject(error))  
 }  
 });  
  
 }  
  
 fetch (id) {  
  
 return new Promise((resolve, reject) => {  
 if (id == undefined) {  
 new Request(`${this.url}/${this.id()}`).get()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "fetch", model: this});  
 resolve(data)  
 })  
 .catch((error) => reject(error))  
 } else {  
 new Request(`${this.url}/${id}`).get()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "fetch", model: this});  
 resolve(data)  
 })  
 .catch((error) => reject(error))  
 }  
 });  
  
 }  
  
 delete (id) {  
 return new Promise((resolve, reject) => {  
 if (id == undefined) {  
 new Request(`${this.url}/${this.id()}`).delete()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "delete", model: this});  
 resolve(data)  
 })  
 .catch((error)=>reject(error))  
 } else {  
 new Request(`${this.url}/${id}`).delete()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "delete", model: this});  
 resolve(data)  
 })  
 .catch((error)=>reject(error))  
 }  
 });  
 }  
  
}  
  
export default Model;

## Collection.js

/\*--- collection ---\*/  
/\* === Spécifications ===  
  
Remarques:  
  
- j'ai ajouté une propriété url à la classe Collection,  
Vous pouvez vérifier que Humans.js a été modifié pour en tenir compte  
  
Donc, complétez la classe en lui ajoutant:  
  
- une méthode fetch() qui permet de retrouver tous les modèles   
 fetch() retourne une promise (grâce à Request)  
  
Remarque : côté node c'est ceci qui est appelé:  
app.get("/models\_url", function(req, res) { ... });  
  
!!!: Pensez à notifier les observers de la collection:  
  
- {{event: "fetch", models:models}  
\*/  
  
import Request from './Request';  
  
class Collection {  
 constructor (model, url="/", models = [], observers = []) {  
 this.model = model;  
 this.models = models  
 this.observers = observers;  
 this.url = url;  
 }  
  
 toString () {  
 return JSON.stringify(this.models);  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
  
 add (model) {  
 this.models.push(model);  
 this.notifyObservers({event: "add", model: model});  
 return this;  
 }  
  
 each (callbck) {  
 this.models.forEach(callbck)  
 }  
  
 filter (callbck) {  
 return this.models.filter(callbck)  
 }  
  
 size () { return this.models.length; }  
  
 /\*--- sync ---\*/  
  
 fetch () {  
 return new Promise((resolve, reject) => {  
 new Request(this.url).get().then((models) => {  
 this.models = []; /\* empty list \*/  
  
 models.forEach((fields) => {  
 this.add(new this.model(fields));  
 });  
  
 this.notifyObservers({event: "fetch", models:models});  
 resolve(models);  
 })  
 .catch((error) => reject(error))  
 });  
  
 }  
  
}  
  
export default Collection;

# ES6 - Partie 3

## Mixin d'objets

let tonyStark = {  
 firstName:"Tony", lastName:"Stark"  
};  
let armorAbilities = {  
 fly:() => console.log("I'm flying")  
};  
Object.assign(tonyStark, armorAbilities);  
  
tonyStark.fly(); // I'm flying

## Array.from

Exemple:

Avant pour parcourir comme un tableau le résultat d'un document.querySelectorAll, il fallait d'abord transformer ce résultat en tableau :

var items = [].slice.apply(document.querySelectorAll("li"));  
// ou var items = Array.prototype.slice.apply(document.querySelectorAll("li"));  
items.forEach(function(item) { ... });

Maintenant :

`javascript Àrray.from(document.querySelectorAll("li")).forEach((item) => {})

# ES6 - Partie 3 : Exercice 3: View Model

## Objectifs:

* créer un mini jQuery (mini mini) que l'on utilisera de cette façon: $q(selector), complétez js/skeleton/selector.js
* créer une classe ViewModel, complétez js/skeleton/ViewModel.js
* créer une classe HumansList héritant de ViewModel déstinée à afficher une liste de modèles Humans, complétez js/app/viewModels/HumansList.js

## Exercice

**Remarque**: les spécifications sont décrites dans les fichiers

Stoppez l'exercice 2, puis:

cd ..  
cd 03-views-models  
node app.js  
http://localhost:3000

# Correction

## selector.js

/\* === Spécifications ===  
  
écrire une lambda qui prend en paramètre un sélecteur (selector) de DOM  
qui va exécuter un `document.querySelectorAll(selector)`  
et qui retourne directement un élément du DOM si il n'y a qu'un seul résultat  
ou un tableau d'élements du DOM si il y a plusieurs résultats  
  
dans le cas de plusieurs éléments, ajouter à la valeur de retour:  
une méthode first() qui retourne le 1er élément  
une méthode last() qui retourne le dernier élément  
  
\*/  
  
export default (selector) => {  
 var nodes = Array.from(document.querySelectorAll(selector));  
 if (nodes.length == 1) { nodes = nodes[0]; } else {  
 Object.assign(nodes, {  
 first () { return this[0]; },  
 last () { return this[this.length-1]; }  
 })  
 }  
 return nodes;  
}

## ViewModel.js

/\*--- viewModel ---\*/  
/\* === Spécifications ===  
 Ecrire une classe ViewModel qui soit un observer  
 Elle sera abonnée aux modifications des modèles ou des collections  
 Quand son abonnement est déclenché, cela déclenche sa méthode render et lui passe le contexte en paramètre  
  
 Paramètres du constructeur:  
  
 Le constructeur prendra en paramètre un objet `options` avec pour valeur par défaut {}  
 les membres d'options deviendront les propriétés de l'instance de ViewModel:  
  
 let myViewModel = new MyViewModel({ // MyViewModel est une instance de ViewModel  
 model: bob, // ou collection: humans,  
 element: $q("#mylist")  
 });  
  
 si model existe, alors ajouter l'instance de ViewModel aux observers du modèle  
  
 si collection existe, alors ajouter l'instance de ViewModel aux observers de la collection  
  
 Méthodes:  
  
 - html(code) : modifie la propriété innerHTML de la propriété element de la classe  
 - render(context) : vide  
 - update(context) : c'est un observer, la méthode appelle render  
  
 \*/  
  
class ViewModel { /\* it's an observer \*/  
  
 constructor (options={}) {  
 /\*  
 options: {model,collection,element}  
 \*/  
 Object.assign(this, options);  
  
 if (options.model) {  
 this.model.addObserver(this)  
 }  
 if (options.collection) {  
 this.collection.addObserver(this)  
 }  
 }  
  
 html (code) {  
 this.element.innerHTML = code;  
 }  
  
 render (context) {  
 // afficher des information  
 }  
  
 // c'est un observer  
 update (context) {  
 this.render(context);  
 }  
  
}  
  
export default ViewModel;

## HumansList.js

/\* === Spécifications ===  
  
 écrire une classe HumansList qui hérite de ViewModel  
  
 Paramètres du constructeur:  
  
 - une collection de modèles Human (humansCollection)  
  
 Penser à passer au constructeur de la classe mère cette collection,  
 ainsi que l'élément du DOM qui portera les informations à afficher:  
  
 dans notre page index.html nous avons ceci:  
  
 <div id="humans-list"></div>  
  
 donc la propriété element de la vue sera : $q("#humans-list")  
  
 Méthodes de HumansList:  
  
 - template(collection) : retournera le code html de la liste à afficher à partir de la collection des modèles  
 <ul>  
 <li>Bob Morane</li>  
 <li>John Doe</li>  
 <li>etc. ...</li>  
 </ul>  
  
 - render(context) qui change le code html de la propriété element si l'évènement "fetch" de la collection est déclenché  
  
 \*/  
  
import HumanModel from '../models/Human';  
import HumansCollection from '../models/Humans';  
  
import ViewModel from '../../skeleton/ViewModel';  
import $q from '../../skeleton/selector';  
  
class HumansList extends ViewModel {  
  
 constructor (humansCollection) {  
  
 super({  
 collection: humansCollection,  
 element: $q("#humans-list")  
 });  
  
 }  
  
 template (humans) {  
 return `  
 <ul>${  
 humans.models.map(  
 (human) => `<li>${human.firstName}, ${human.lastName}</li>`  
 ).join("")  
 }</ul>  
 `;  
 }  
  
 render (context) {  
 if (context.event == "fetch") {  
 this.html(this.template(this.collection));  
 }  
 }  
  
}  
  
export default HumansList;

# ES6 - Partie 4

## Map

let map = new Map();  
map.set("one",{firstName:"John", lastName:"Doe"});  
map.set("two",{firstName:"Jane", lastName:"Doe"});  
  
console.log(map.has("one")); // true  
console.log(map.get("one")); // Object {firstName: "John", lastName: "Doe"}  
console.log(map.size); // 2  
  
for (let key of map.keys()) {  
 console.log("Key: %s", key);  
}  
/\*  
 Key: one  
 Key: two  
\*/  
  
for (let value of map.values()) {  
 console.log("Value: %s %s", value.firstName, value.lastName);  
}  
/\*  
 Value: John Doe  
 Value: Jane Doe  
\*/  
  
for (let item of map) {  
 console.log("Key: %s, Value: %s", item[0], item[1].firstName, item[1].lastName);  
}  
/\*  
 Key: one, Value: John Doe  
 Key: two, Value: Jane Doe  
\*/  
  
/\* Et aussi : \*/  
  
let myOtherMap = new Map([  
 ["one",{firstName:"John", lastName:"Doe"}],  
 ["two",{firstName:"Jane", lastName:"Doe"}],  
 ["three",{firstName:"Bob", lastName:"Morane"}]  
]);  
  
myOtherMap.delete("three")  
  
myOtherMap.forEach((item)=>{  
 console.log(item)  
})  
/\*  
 Object {firstName: "John", lastName: "Doe"}  
 Object {firstName: "Jane", lastName: "Doe"}  
\*/

## Remarque

Vos classes peuvent hériter des types javascript

# ES6 - Partie 4 : Exercice 4: Le routeur

## Objectifs:

Créer un routeur qui "écoutera" le navigateur :

* click sur un lien (met à jour window.location.hash)
* saisie d'url (met à jour window.location.hash)
* bouton back (met à jour window.location.hash)

Ce routeur contiendra des routes (couple url, traitement), si l'url d'une route = window.location.hash, alors le traitement correspondant est déclenché

## Exercice

**Remarque**: les spécifications sont décrites dans les fichiers

Stoppez l'exercice 3, puis:

cd ..  
cd 04-router  
node app.js  
http://localhost:3000

# Correction

## Router.js

/\* === Spécifications ===  
  
Le routeur a plusieurs routes  
  
Routes : une url ds le navigateur que l'on associe à un traitement (ex afficher une vue)  
Du coup les fonctionnalités de ma webapp sont bookmarkables  
  
Ma classe Router est une Map  
  
Donc si je veux créer un routeur et rajouter des routes :  
  
 let router = new Router();  
  
 router.set("humans", (args) => {  
 // faire quelque chose  
 });  
  
 router.set("animals", (args) => {  
 // faire quelque chose  
 });  
  
 router.set("/", (args) => {  
 // faire quelque chose  
 });  
  
  
A ma classe Router  
  
---------------------------------------  
je rajoute une méthode match(uri)  
---------------------------------------  
qui servira à vérifier les urls saisies ds le navigateur  
ou les liens cliqués et déclencher les méthodes associées  
  
uri peut prendre les types de valeurs suivantes:  
  
 #/humans  
 #/humans/1234  
  
 etc...  
  
 match(uri) va  
  
 - "retraiter" uri : enlever le #/ donc:  
  
 #/humans devient humans  
 #/humans/1234 devient humans/1234  
  
 - spliter uri : on sépare tous les éléments entre les "/"  
 - on obtient un tableau que l'on filtre pour ne garder que les éléments dont la taille > 0  
  
 - le 1er élément du tableau devient la clé à rechercher dans la map  
 - les éléments restants représentent les paramètres que l'on passera à la méthode correspondant à la clé  
  
 Par exemple si je saisie #/humans/bob/morane deviendra  
  
 ["humans", "bob", "morane"]  
  
 et j'irais chercher dans mon instance de Router la clé "humans" qui va me retourner une méthode  
 à laquelle je passerais (si elle existe) les paramètres ["bob", "morane"] (donc sous forme d'un tableau)  
 et je l'exécuterais  
  
 (allez voir main.js pour l'exemple)  
  
---------------------------------------  
je rajoute une méthode listen()  
---------------------------------------  
  
Qui va une 1ère fois au lancement vérifier l'url pour déterminer s'il y a un traitement à lancer  
  
 la méthode match du routeur sera appelé avec window.location.hash en paramètre  
  
 window.location.hash : retourne la partie de l'url qui correspnd à "anchor" : http://localhost:3000/#/humans donnera #/humans  
  
Cette 1ère vérification est utile si on a bookmarké une fonctionnalité de l'application  
  
Ensuite on va s'abonner à l'évènement "onpopstate" (déclenché lorsque l'utilisateur "navigue", ultilise le bouton back, etc...)  
  
et à chaque fois que l'évènement "onpopstate" sera déclenché, la méthode match du routeur sera appelé avec window.location.hash en paramètre  
  
  
\*/  
 class Router extends Map {  
  
 constructor () {  
 super();  
 this.set("/",(args)=>{});  
 }  
  
 match (uri) {  
   
 // on retraite l'uri: enlever les #/  
 uri = uri.replace("#\/","");  
  
 // splitter uri avec "/" et ne garder que les éléments non vides  
 let uriParts = uri.split("/").filter((part)=>part.length>0);  
  
 // clé à chercher  
 let route = uriParts[0];  
 // paramètres à passer à la méthode  
 let params = uriParts.slice(1);  
  
 // récupérer la méthode  
 let method = this.get(route);  
   
 // exécuter la méthode  
 if (method) { method(params) } else {  
 this.get("/")(params)  
 }  
 }  
  
 listen () {  
 // une fois le routeur en mode écoute  
 // lui faire vérifier une 1ère fois l'url pour déterminer quoi faire  
 this.match(window.location.hash);  
  
 /\* s'abonner à onpopstate \*/  
 window.onpopstate = (event) => {  
 this.match(window.location.hash);  
 };  
 }  
  
}  
  
export default Router