Fear the walking web Docs&Demos

1: Présentation:

"**Fear the walking web - flesh and bones**" est une bibliothèque javascript pour vous aider a réaliser des applications HTML5 en permettant **une séparation nette entre le code de présentation HTML et les données de votre application**. Chaque fois que vous avez besoin d'afficher des données provenant d'objets javascript (provenant d'un service web, d'une base de données ou chargé via JSon…) FTW2 peut vous aider à l'implémenter de **façon simple et maintenable**.

Les concepts clés de FTW2 sont:

Déclaratif: les Data-binding sont renseignés **uniquement dans le code html** (a la façon de XAML, pour ceux qui connaissent), grâce à une syntaxe simple et comportant peu de mots clés. Les templates pour afficher des objets complets ou des tableaux (appelés ici "presenter" ou "item-presenter") sont aussi déclarés dans la page HTML.

Réutilise l'existant: permet de réutiliser au maximum les codes métiers existants sans avoir a (trop) le modifier,

Invisible: aucune API (ou presque), il suffit de lier le fichier javascript et ça marche! **Pas de d'API à apprendre, pas de fonctions à appeler**!

Stand-alone: FTW² est **VanilliaJS:** pas besoin de bibliothèque tierce pour fonctionner, et fonctionne sur la plus part des navigateurs (en-théorie, en-core en test;).

Cette bibliothèque vient en 2 versions:
***“fear-the-walking-web-flesh-and-bones.js”**: le moteur de liaison de données simple. Si vous voulez juste rajouter du binding a une page HTML.*

***“fear-the-walking-web-pandemie.js”**: un framework de gestion d’applications HTML5 complet avec:*
- navigation par pages,
- sauvegarde/restauration des états,
- composition de pages a partir de Fragments (pour les androids fans)
- possibilité de découper son application en plusieurs fichiers regroupant pages et fragments de facon logique (en cours)
- remoteStorage pour sauvegarder les etats des pages sur un serveur cloud (a faire…)
- un set d’UI (panel, dialog) pour créer des interfaces rapidement (à la façon d’un JQuery) (a compléter)
- et bien d’autres choses...

Cette version est pour l’instant en cours de développement, je l’ai inclue dans sa version 0.1 avec une application d’exemple pour vous faire une idée.

Merci d'avoir télécharger FTW²!

2: Version et Installation:

Version:

Actuellement, FTW² est en **version 0.3**, il reste encore pas mal de travaille a faire dessus, notamment d'optimisation de temps d'exécution ou d'utilisation de mémoire, mais les déclarations de binding, les arguments… sont finalisés, et je ferais mon possible pour que toute nouvelle version de "Flesh and Bones" reste **rétro-compatible.**

Installation:

Référencez simplement le fichier javascript dans votre code HTML:

```
<script src="fear-the-walking-web-flesh-and-bones.js"></script>
```

en adaptant l'attribut "src" pour correspondre à l'endroit ou vous avez mis FTW².

3: Data-context

Un **data-context** (aussi appelé Model dans le pattern MVVM) est un **simple objet javascript** de portée globale qui servira de conteneur pour les données a afficher dans l'UI et les opérations a effectuer dessus.

Pour créer un data-context pour votre application, (en général, se sera la première chose a faire), créez juste un objet javascript et **déclarez le dans la balise <body> de votre page HTML,**Le moteur de liaison de données saura alors ou aller chercher les informations dont il a besoin pour créer l'interface graphique.

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>Hello</title>
        <script src="fear-the-walking-web-flesh-and-bones.js"></script>
        <script >
        var MyContext = {
                //ici, les données de mon application
        };
```

```
    </script>
</head>
<body data-context="MyContext ">
    <!-- mon UI ici! -- >
</body>
```

Vous devez renseigner cet attribut, sinon un exception sera levée au lancement de la page.

4: un simple Data-binding:

la liaison de données (binding) dans FTW² est **déclarative**: on ne la renseigne que dans le fichier html sous la forme :

{binding paramètre1:valeur paramètre2:valeur etc...}.

Le framework se chargera de faire les liaisons entre votre contexte de données (ici, MyContext) et le ou les éléments HTML visés. Aucune modification des objets javascript n'est nécessaire, et **on peut lier autant de propriétés/attributs de l'élément html que voulu**.

Marquer un binding dans un élément HTML

On peut renseigner une liaison de données de plusieurs façon, et on peut ajouter autant de liaisons que l'on veut dans un élément HTML.

- **<a data-binded-href="{binding from:le_nom_de_ma_variable }">**:lie la variable à l'attribut href de l'anchor. Cette notation permet de lier autant d'attribut qu'on le souhaite, mais une seule liaison par attribut/propriété est autorisée.
- **{binding from:le_nom_de_ma_variable }**: un simple raccourcie d'écriture pour
- ****: permet de faire un binding sur une class CSS. Lors de la liaison, seule la partie {binding….} sera remplacée.

Les différents paramètres sont:

- **from**: Obligatoire, le nom de la propriété à lier *-idée: pouvoir mettre une URL pour mettre a jour le binding via ajax?*
- **to**: optionnel, si renseigné, le nom de la propriété de l'élément HTML/attribut a lier (reliquat d'ancienne version)
- **converter**: optionnel, si renseigné, une méthode globale pour convertir la donnée avant la liaison, avec les arguments suivant:
  - @param: value: la valeur a convertir
  - @param converter_params: paramètres optionnels (array)
- **converter_params**: optionnel, permet de passer des paramètres a la méthode de conversion.
- **fallback**: optionnel, si value==null, affichera le fallback a la place
- **alt**: optionnel, expérimental, permet de lier ce binding a une autre clé que celle de la propriété
- **forceAttr**: optionnel, doit être égal a true. Si renseigné, forcera la liaison sur l'attribut html plutôt que la propriété de l'objet

Exemple: binding d'une propriété simple:
simple_binding.html

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>Hello</title>
        <script src="fear-the-walking-web-flesh-and-bones.js"></script>
        <script >
        var MyContext = {
              app_name : "Ma super application",
        };
        </script>
</head>
<body data-context="MyContext ">
        <h1>{binding from:app_name}</h1>
</body>
```

On crée une propriété “app_name” contenant le nom de notre application comme n'importe quelle propriété javascript, et on demande l’affichage dans la balise <H1> en utilisant la notation de FTW² en indiquant le nom de la propriété que l’on veut afficher (paramètre FROM).

Pour récupérer la valeur de app_name dans votre code javascript, ou changer sa valeur, on fait tout simplement:

```
var v = MyContext.app_name; //recupere la valeur
MyContext.app_name = “Hello world!”; //modifie la valeur et met a jour l’UI
```

Propriétés invisibles:

Parfois, vous pouvez avoir besoin de créer une propriété juste pour un usage interne, sans avoir besoin de l’afficher dans l’UI, ou vous voulez pouvoir contrôler sa valeur en créant des getters et setters. Pour créer une propriété “invisible” pour FTW², il suffit de faire débuter le nom de la variable par un underscore (par exemple, _app_name), et elle sera superbement ignorée par le framework!

Si vous créer des getters/setters pour cette propriété, vous pourrez alors lier ces derniers a votre UI:

```
var MyContext = {

    your_name : “Stéphane”,

    _invisble_shadow = 12 //on la rend invisible pour FTW², on ne pourra pas l’utiliser dans l’UI

};

//on crée une property type getters setters pour _invisible_shadow, le nom de cette property -ici, hello- servira pour renseigner le paramètre from dans nos bindings
Object.defineProperty(MyContext,”hello”,{
    get: function(){return this._invisible_shadow + 1;},
    set: function(val){this._invisible_shadow = value;}
});
```

//désormais, on peut lier la nouvelle propriété “hello” dans nos binding!

…

<h1>{binding from:your_name }</h1>
<h2>vous avez: {binding from:hello} ans</h2>

Pour résumé:

Toutes les propriétés ‘simples’ de votre contexte de données sont référençable dans les bindings par leur nom.
Toutes les property type getter/setter de votre contexte de données sont référençable dans les bindings par leur nom
une propriété dont le nom commence par un underscore dans votre contexte de donnée est invisible et non référençable dans les bindings.

Exemple: convertir une donnée avant affichage:

Converter_params:

Un “converter” est une fonction de portée globale qui sert a transformer la valeur d’une propriété du contexte de données en autre chose au moment de l’afficher a l’écran.

Par exemple, une fonction qui localiserait un float pour l’affichage:

var MyContext = {

// la valeur a afficher dans l’UI

```
        ma_valeur : 13.879,
};
function localize(value, params){
        //remplace le point par une virgule
        if (value == null) return "";
        return value.toLocaleString ("fr-FR", {maximumFractionDigits:2, minimumFractionDigits:2});

}
```

Dans le html, on indique au binding d'utiliser le converter pour l'affichage;

```
<h2>localisée: {binding from:ma_valeur  converter:localize}</h2>
```

Un "converter" prend en argument **la valeur du binding (@arg value)**, et **un tableau d'arguments optionnels**. Il doit retourner une valeur (généralement sous forme de String) qui sera utilisée par le binding lors de l'affichage.

Pour passer des arguments optionnels a un converter, il faut renseigner le paramètre **converter_params** dans le html. Les valeurs possibles sont:

- **converter_params:valeur**: renseigne 1 argument optionnel.
- **converter_params:[liste,de,valeurs]**: renseigne des arguments optionnels. Ils seront passés sous forme de tableau au "converter".
- Si une valeur est une **string avec des espaces**, l'inclure dans des apostrophes
- Si on veut passer **la valeur d'une propriété du context**, on fait précéder le nom de la propriété par $. Si la propriété passée en paramètre change, le binding sera prévenu et se mettra a jour aussi.

Dans l'exemple précédent, pour contrôler le nombre de chiffres après la virgule, on transforme la fonction "localize":

```
function localize(value, after_dot_count){
        //remplace le point par une virgule
        if (value == null) return "";
       after = after_dot_count || 2; //on garde 2 par défaut
       return value.toLocaleString ("fr-FR", {maximumFractionDigits:after, minimumFractionDigits:after});
}
```

et on met a jour le binding:

```
<!-- ici, on affiche avec 2 chiffres après la virgule -->
<h2>localisée: {binding from:ma_valeur  converter:localize}</h2>
<!-- ici, on en demande 3 -->
<h2>localisée aussi: {binding from:ma_valeur  converter:localize converter_params:3}</h2>
```

On verra d'autres utilisation des converters (notamment couplés avec des classes CSS) dans l'application WoWS de démonstration en fin de documentation.

Exemple: gérer le cas NULL:

Si une de vos valeurs est null ou undefined, on peut demander a afficher une valeur par défaut en renseignant le paramètre **fallback**. Par exemple, on veut que si la propriété “your_name” du data-context est nulle, on affiche le message “Oups, vous êtes un inconnu…”, on modifiera alors le binding:

<h1>{binding from:your_name fallback:’Oups, vous êtes un inconnu…’ }</h1>

Si vous cliquez sur le lien “Fallback: met le nom a null!” dans la page d’exemple, vous devriez voir apparaître le message d’erreur.
Notez que si la valeur du fallback comporte des espaces, vous devez l’inclure dans des apostrophes.

On verra ,dans l’application WoWS de démonstration en fin de documentation, l’utilisation du paramètre “alt”, ou le binding de plusieurs attributs/propriétés d’élément HTML dans une application real-world.

alt: réagir a d'autres clés de binding

EXPERIMENTAL. Parfois, il peut etre utile de 'lier' des données entre elles, ou de mettre a jour une donnée en fonction d'une autre. Tous les bindings ont un paramètre **alt:name1,name2,name3**... pour permettre cette liaison, ou name1,name2... correspondent aux nom des propriété du contexte courant de données auxquelles on veut que le binding réagisse en plus de la propriété indiquée dans le 'from'.

5: Data-binding d’objets et presenters:

exemples: presenters.html

Lorsqu'on veut lier un objet dans FTW², il faut auparavant créer un **presenter**, une sorte de template d'affichage pour les propriété de l’objet. Ceux ci se définissent directement dans le fichier HTML.

- Directement sous la balise <**BODY**>, on crée **un conteneur a presenters** (un div avec un **data-role=presenters**)
- Chaque childs de ce conteneur représente un presenter particulier. un presenter **doit avoir un id unique** et renseigner l'attribut **data-role=presenter**
- Chaque childs de ce presenter représente un **data-type** particulier, le firstChild étant le type par défaut, n'a pas besoin de préciser le type
- A l'intérieur d'un presenter, **le contexte de données change**. Il devient l'objet qui est lié , rendant accessible directement les propriété de l'objet pour le binding.

Les différents paramètres sont:

- **from**: Obligatoire, le nom de la propriété à lier
- **presenter**: Obligatoire, l’identifiant d’un élément HTML a utiliser pour afficher les informations de l’objet.
- **converter**: optionnel, si renseigné, une méthode globale pour convertir la donnée avant la liaison.
    - @param: value: la valeur a convertir
    - @param converter_params: paramètres optionnels
- **converter_params**: optionnel, permet de passer des paramètres a la méthode de conversion.
- **fallback**: optionnel, si value==null, id du div presenter a afficher a la place
- **alt**: optionnel, expérimental, permet de lier ce binding a une autre clé que celle de la propriété

- **deep**: optionnel, doit être false. Empêche de créer les liaisons de données pour les propriétés internes de l'objet. (ie, on peut afficher les propriétés de l'objet, mais on ne souhaite pas être notifier de leurs changement). Par défaut true= si une propriété interne de l'objet est modifié, le framework mettra a jour le binding.

Pour illustrer l'utilisation des "presenters", on va afficher des informations sur un 'client' (un nom, prénom, sexe) dans une page HTML.
On commence par créer en javascript un data-context et une classe Client pour regrouper nos informations:

```
function Client(n,p,s){
        //on initialise les informations de notre client...
        this.nom = n;
        this.prenom = p;
        this.sexe = s;
}
MyContext = {
        //on crée un client pour pouvoir tester les presenters….
        client1 : new Client("PONTEINS","Stéphane","M"),
        client2: new Client("PITT","Bebert","M")
};
```

Dans notre HTML, on renseigne le contexte de données dans la balise <BODY> et on ajoute notre template pour afficher les données de notre client et le binding pour l'afficher:

```
<body data-context="MyContext ">
        <!-- le conteneur pour les presenters -->
        <div data-role="presenters">
```

```
        <!-- le presenter pour afficher les données d'un client
habituellement, j'utilise un div, mais n'importe quel élément peut faire l'affaire, la seule restriction est qu'il ne peut pas être lui même
la cible d'un binding-- >

        <div id="client_presenter" data-role="presenter">
            <div>
                <!-- a l'interieur d'un presenter, le context de données change et devient l'objet a afficher (ici:
MyContext.client) -->
                <h1>Nom: {binding from:nom}</h1>
                <h2>Prénom: {binding from:prenom}</h2>
            </div>
        </div>
    </div>



    <!-- affiche le contenu de MyContext.client en utilisant le presenter dont
l'identifiant est "client_presenter" -- >
    <h3>Mon premier client:</h3>
    <div>{binding from:client1 presenter:client_presenter}</div>

    <h3>Mon autre client:</h3>

    <div>{binding from:client2 presenter:client_presenter}</div>

</body>
```

Data-types

Javascript n'étant pas fortement typé, une propriété peu contenir a peu près n'importe quoi. Pour refléter cela, les presenters peuvent définir des “data-types” pour adapter l'affichage aux données présentes. Il suffit, dans un presenter, d'ajouter des childs avec un attribut **data-type="type de la property"**.

Pour reprendre l’exemple de l’affichage des clients, on va créer une nouvelle classe BonClient qui hérite de Client:

```
function BonCLient (n,p,s,c){
        Client.call(this,n,p,s); //BonClient est un client, on reutilise les informations
        this.code_promo = c;
}
BonClient.prototype.__proto__ = new Client(); //pour la chaine de prototype


MyContext = {
        //on crée un client pour pouvoir tester les presenters….
        client1 : new Client(“PONTEINS”,”Stéphane”,”M”),
        client2: new BonClient(“PITT”,”Bebert”,”M”, 3246) // Bébert deviens mon bon client
};
```

Dans le presenter “client_presenter”, on rajoute le data-type pour BonClient afin d’afficher les informations en plus:

```
<body data-context="MyContext ">
```

```
<!-- le conteneur pour les presenters -->
<div data-role="presenters">
    <!-- le presenter pour afficher les données d'un client -->
    <div id="client_presenter" data-role="presenter">
        <div>
            <!-- a l'interieur d'un presenter, le context de données change et devient l'objet a afficher (ici: MyContext.client) -->
            <h1>Nom: {binding from:nom}</h1>
            <h2>Prénom: {binding from:prenom}</h2>
        </div>

        <div data-type="BonClient">
            <h1>Nom: {binding from:nom}</h1>
            <h2>Prénom: {binding from:prenom}</h2>
            <h2>Code: {binding from:code_promo}</h2>

        </div>

    </div>
</div>
<!-- affiche le contenu de MyContext.client en utilisant le presenter dont l'identifiant est "client_presenter" -- >
<h3>Mon premier client:</h3>
<div>{binding from:client1 presenter:client_presenter}</div>

<h3>Mon autre client:</h3>
```

{binding from:client2 presenter:client_presenter}

</body>

Note sur la création d'objet:

En javascript, on peut créer des objets de 2 façons:

implicitement avec la notation obj = { … };

explicitement avec le mot clé new et une fonction constructeur. ***Pour utiliser les data-types, vos objets doivent être crées avec une fonction constructeur et appel au mot clé NEW.***

Data-types et héritage:

FTW², lorsqu'il cherche à afficher un objet, va utiliser la chaîne de prototypes afin de déterminer le data-type le plus probable. Pour illustrer le fonctionnement, on va créer un nouveau type de Client, SuperClient, héritant de BonClient mais sans data-type dédié dans le presenter.

```
function SuperClient (n,p,s,c,i){
    BonClient.call(this,n,p,s,c); //permet de recuperer les initialisations des autres constructeurs
    this._infos = i; //une donnée très importante surement…. mais comme le nom commence par un underscore, il reste caché pour le binding
}

// Une petite note sur la notation du prototype:
//si on avait ecrit SuperClient.prototype = new BonClient(); simplement, l'objet aurait perdu l'information sur son constructeur (comportement normal de javascript il semble). En initialisant __proto__, les informations sur le constructeur reste et permettent de différencier les types de données;
SuperClient.prototype.__proto__ = new BonClient();
```

Dans le data-context global, on crée un nouveau client de type SuperClient, et on l'affiche avec un binding. FTW² après avoir recherché le type "SuperClient" sans succès, **remontera la chaîne des prototypes jusqu'à trouver un type qu'il connaît**...

Fallback et data-type:

Comme pour les bindings simples, il est possible de décider quoi afficher dans le cas ou l'objet à afficher est null ou undefined (en place du "unknown model!" affiché de base par le framework). Il y a 2 façon de gérer ce cas: ou **renseigner un identifiant de "presenter"** a utiliser dans ce cas avec le paramètre "fallback", ou **créer un data-type="fallback"** dans le "presenter".

fallback:

un "presenter de fallback" se crée comme n'importe quel autre "presenter" mais aura pour contexte de données le contexte "global" (Dans notre exemple, le contexte du "fallback" sera MyContext). Un "presenter" de fallback n'est pas associé à un type particulier et peut être utilisé par n'importe quel binding d'objet dans votre page.

```
MyContext = {

        message : "Désolé, pas d'infos sur ce client.", //le message a afficher si le client est null
        //on crée un client pour pouvoir tester les presenters….
        client1 : null,// ici, le premier client est indefini...
        client2: new BonClient("PITT","Bebert","M", 3246)
};
```

```
<div data-role="presenters">

    <!-- le presenter pour afficher un fallback-->
    <div id="fallback_presenter" data-role="presenter">
      <div>
        <!-- dans un fallback, le contexte de données est le contexte global! -->
        <h1>{binding from:message}</h1>
      </div>
    </div>

    <!-- le presenter pour afficher les données d'un client -->
    <div id="client_presenter" data-role="presenter">
        …
    </div>
</div>
```

Et on précise juste dans notre binding le paramètre "fallback" pour lui indiquer ce qu'il faut afficher:

```
<h3>Mon premier client:</h3>
<div>{binding from:client1 presenter:client_presenter fallback:fallback_presenter}</div>
```

data-type="fallback":

Une autre façon de gérer le cas null ou undefined est de créer un data-type dédié dans le "presenter". Dans ce cas, le contexte de données sera aussi le contexte global de données. C'est juste une question de préférence… Un data-type="fallback" est associé a un type de donnée (ou pour être exact, a un type de presenter) et ne peux donc pas être partagé par un autre type de presenter.

```
<div data-role="presenters">
    <!-- le presenter pour afficher les données d'un client -->
    <div id="client_presenter" data-role="presenter">
        <div>
            <!-- a l'interieur d'un presenter, le context de données change et devient l'objet a afficher (ici: MyContext.client) -->
            <h1>Nom: {binding from:nom}</h1>
            <h2>Prénom: {binding from:prenom}</h2>
        </div>

        <div data-type="BonClient">
            <h1>Nom: {binding from:nom}</h1>
            <h2>Prénom: {binding from:prenom}</h2>
            <h2>Code: {binding from:code_promo}</h2>

        </div>

        <div data-type="fallback">
            <!-- dans un fallback, le contexte de données est le contexte global! -->
            <h1>Erreur: {binding from:message}</h1>
        </div>
    </div>
</div>
```

Note: si vous définissez un data-type="fallback" ET renseignez le paramètre fallback dans le binding, le paramètre fallback sera l'affichage choisi en priorité!

6: Data-binding et arrays:

item_presenters.html

Un binding utilisant un 'item_presenter' sert a afficher une liste d'objets. Lorsqu'on veut lier un tableau (liste) dans FTW², il faut auparavant créer un **item_presenter**, une sorte de template d'affichage pour les propriétés de chaque item du tableau. Ceux ci se définissent directement dans le fichier HTML.

- Directement sous la balise <**BODY**>, dans **le conteneur a presenters** (un div avec un **data-role=presenters**)
- Chaque childs de ce conteneur représente un presenter particulier. un presenter **doit avoir un id unique** et renseigner l'attribut **data-role=presenter.** (un item_presenter n'est en fait qu'un presenter tout à fait normal)
- Chaque childs de cet item_presenter/presenter représente un **data-type** particulier, le firstChild étant le type par défaut, n'a pas besoin de préciser le type. Le data-type correspond **au type des items** du tableau/liste qui est lié.
- A l'intérieur d'un item_presenter, **le contexte de données change**. Il devient tour à tour chaque item du tableau qui est lié , rendant accessible directement leurs propriétés pour le binding.

Les différents paramètres sont:

- **from**: Obligatoire, le nom de la propriété à lier
- **item_presenter**: Obligatoire, l'identifiant d'un div a utiliser pour afficher les informations des éléments de l'array.
- **converter**: optionnel, si renseigné, une méthode globale pour convertir la donnée avant la liaison.
    - @param: value: la valeur a convertir

    - @param converter_params: paramètres optionnels
- **converter_params**: optionnel, permet de passer des paramètres a la méthode de conversion.
- **fallback**: optionnel, si value==null, id du div presenter a afficher a la place
- **empty**: optionnel, si value.length == 0, id du div a afficher a la place
- **alt**: optionnel, expérimental, permet de lier ce binding a une autre clé que celle de la propriété
- **deep**: par défaut True, indique si doit créer des liaisons pour les éléments de l'array

Pour continuer avec nos clients, on va cette fois créer une liste de clients et demander l'affichage par un item-presenter. On crée dans le contexte de données global une propriété clients et on l'initialise.

```
MyContext = {
        clients : [ new Client("JOE","joe","M"),
                        new Client("MALLOW","Marc","M"),
                        new Client("DALTON","M'a","F")]
};
```

Comme pour les objets "normaux", on définit un "presenter" pour les items du tableau:

```
<div data-role="presenters">
        <!-- le presenter pour afficher les données du tableau -- >
        <div id="client_item_presenter" data-role="presenter">
        <--ici, on n'utilise pas un div mais un LI puisqu'on veut afficher dans un UL-->
                <li>
                        <span>Nom: {binding from:nom}</span> 
                        <span>Prénom: {binding from:prenom}</span>
                </li>
        </div>
```

```
</div>
```

Et dans la page HTML, on demande a afficher la liste par un binding:

```
<ul>{binding from:clients item_presenter:client_item_presenter}</ul>
```

Pour chaque élément du tableau “clients”, le framework va rechercher le data-type correspondant à l’item courant dans le presenter “client_item_presenter”. Une fois trouvé, il renseigne les bindings présents avec les données de l’item courant,(ie: clients[0], clients[1]...) puis rajoute a l’élément <UL> le code HTML généré pour l’item. (j'espère être clair…). Ainsi de suite pour tous les éléments du tableau.

Modification du tableau et mise à jour de l’UI:

Les méthodes supportées par le framework pour la manipulation des tableaux, cad qui mettront automatiquement à jour l’UI, sont:

Array.push: ajoute un ou plusieurs éléments à la fin du tableau
Array.pop: supprime le dernier élément du tableau
Array.unshift: ajoute au début du tableau
Array.shift: supprime en début de tableau
Array.splice: supprime et ajoute à un index

Note: ce sont les méthodes de manipulation les plus utilisées, mais on pourra toujours en rajouter au besoin

Le cas de l’opérateur [ ]:

Malheureusement, je n’ai pas trouvé, avec ECMAScript 5, une façon de mettre à jour l’UI automatiquement lors de l’assignation d’une valeur dans un tableau en utilisant l'opérateur [ ]. ECMAScript 6, avec la création des Proxy, semblerai permettre de résoudre le problème… En attendant son adoption par tous les navigateurs, j’ai **rajouté aux objets de type Array une méthode set(item, index)** qui permet de modifier un élément d’un tableau par son index et de mettre à jour l’UI.

Data-type et valeur NULL:

Définir un item_presenter est exactement la même chose que définir un presenter normal. Vous pouvez aussi utiliser des data-types avec! La gestion du fallback diffère légèrement par rapport aux objets: **le paramètre fallback** dans le binding sert a **gérer le cas ou la propriété(le tableau) est nulle**, le **data-type="fallback"** sert lui a gérer le cas ou **un item du tableau serait null**!

Data-type 2: Lier un tableau de valeurs simples:

imaginons qu'on définisse une propriété dont la valeur serait un tableau de valeurs simples (cad de primitives: number, string…) :

ma_prop : ["une","liste",2,"valeurs"];,

Comme c'est un tableau, on devra utiliser un "presenter" pour l'affichage. Or, dans un "presenter", le contexte de données change, il deviendra donc tour a tour "une", puis "liste", puis 2, puis... Dans ce cas, ce qui nous intéresse n'est pas d'afficher une propriété du contexte (de String ou Number), mais le contexte lui même! Le paramètre **from** accepte une valeur particulière: **$this**. Lorsqu'il rencontre cette valeur, cela indique au moteur de binding d'afficher une représentation du contexte sous forme de String.

```
<div data-role="presenters">
  <div data-role="presenter" id="monPresenter">
    <li>{binding from:$this}</li>
```

```
    <div>
```

Notez que même avec des types simples, on peut définir des data-types pour gérer les différents cas séparément:

```
<div data-role="presenters">
    <div data-role="presenter" id="monPresenter">
        <li><span>{binding from:$this}</span></li>
        <li data-type="Number"><span>un nombre: {binding from:$this}</span></li>
    <div>
<div>
```

Paramètre deep:

Par défaut, **tout objet affiché par un binding** va subir un traitement pour permettre de mettre à jour l'UI **lors de la modification d'une de ses propriétés internes.** (Par exemple, si vous afficher un objet Client par un "presenter" puis que vous en modifier le nom par myContext.monclient.nom = "nouveau nom", le binding associé a la propriété "nom" de l'instance "monclient" de la classe "Client" lancera une mise à jour de l'UI pour le refléter). **Lors de l'affichage d'un tableau, le même processus à lieu pour tous les items du tableau** (sauf les types primitifs), ce qui peut devenir lourd, surtout dans le cas ou on veut juste afficher une liste non modifiable (pour permettre à l'utilisateur d'en sélectionner un item, par exemple).

On peut alors passer le paramètre **deep:false** au binding pour lui indiquer de **considérer le tableau (ou l'objet) comme une propriété primitive** (comme un String ou un Number), et donc de ne pas prendre en compte les modifications de ses propriétés internes.

7: Commandes:

commands.html

Celui ci permet de lier **une méthode du contexte de données global** a un évènement javascript quelconque. Pour lier un event a une commande, la notation est:

```
<tagname data-bind-eventName = "{binding command:nom_methode command_params:param}">
```

Les différents paramètres, pour un binding de commande, sont:

- **command**: Obligatoire, le nom de la méthode du contexte de donnée global à lier a l'event. Cette méthode prend en arguments:
  - @param evt: l'objet event généré
  - @param args: paramètre optionnel (tableau si plusieurs)
- **command_params**: optionnel, des paramètres pour la commande (même fonctionnement que les paramètres pour converter)
- **converter**: optionnel, si renseigné, une méthode globale pour déterminer le nom de la méthode avant la liaison.
  - @param: value: la valeur a convertir
  - @param converter_params: paramètres optionnels
- **converter_params**: optionnel, permet de passer des paramètres a la méthode de conversion.

L'eventName correspond au **nom de l'event javascript** auquel la commande souhaite réagir (par exemple, pour réagir à un click, on écrira **data-bind-click = {binding….}**)

Pour donner un exemple de fonctionnement des commandes, on va reprendre (encore) nos clients… On va afficher une liste de clients, avec **dans l'item_presenter une commande pour pouvoir supprimer ce client**, et dans la page, **une commande pour pouvoir en ajouter un nouveau** (toujours en dur…).

javascript:

```
function Client (n,p,s){
        //on ajoute un identifiant unique au client (comme s'il provenait d'une base de données par exemple)
      //generateUUID() est une méthode définie dans le framework qui fait ce que son nom indique...
          this.id = generateUUID();
          this.nom = n;
          this.prenom = p;
          this.sexe=s;
        };
        myContext = {
              //une liste de clients renseignée par défaut
                  clients : [ new Client("MANSON","Marylin","???"),
                                    new Client("MALLOW","Marc","M"),
                                     new Client("DALTON","M'a","F")],

        };

        //On défini les méthodes pour les commandes
        //paramètres: evt: l'objet event généré (ici ClickEvent), et les paramètres optionnels
        //comme je ne défini qu'un seul paramètre -id- je le  récupère directement,
        //si j'en avait défini plusieurs, on aurait eu un tableau
        myContext.delete_client = function(evt, id){
              //recherche l'index du client a supprimer
              for (i=0;i<this.clients.length;i++){
                    //si l'id est le bon
                    if (this.clients[i].id == id){
```

```
                    //supprime le du tableau
                    this.clients.splice(i,1);
                    break;
                }
            }
        }

        myContext.add_client = function(evt){
            //ajoute un nouveau client à la liste, en dur pour l'instant...
            //le fait d'appeller push sur mon tableau suffit a mettre a jour l'UI!!!
            this.clients.push(new Client("NOUVEAU","Client","F"));
        };
```

html:

```
<body data-context="myContext">
    <div data-role="presenters">
        <!-- le presenter pour afficher les données du tableau -->
        <div id="client_item_presenter" data-role="presenter">
        <!--ici, on n'utilise pas un div mais un LI puisqu'on veut afficher dans un UL-->

            <li>
                <span>Nom: {binding from:nom}</span> 
                <span>Prénom: {binding from:prenom}</span> 
            <!-- On cree une commande qui prend en parametre optionnel l'identifiant du client pour pouvoir le retrouver -->
            <a href="#" data-binded-click="{binding command:delete_client command_params:$id}">Supprimer</a>
            </li>
```

```
            </div>

        </div>
        <h3>Une simple liste de clients:</h3>
        <ul>{binding from:clients item_presenter:client_item_presenter}</ul>
       <-- ici, une commande simple sans paramètres -->
        <a href="#" data-binded-click="{binding command:add_client}">Ajouter un client</a>


    </body>
```

8: Input binding: récupérer les informations de l'utilisateur:

2way.html

Maintenant qu'on sait afficher des données provenant d'objets javascript, et interagir avec eux via des commandes, on va voir comment **récupérer des informations entrées par l'utilisateur grâce aux bindings mode 2way**.

notation: **data-binded-value="{binding from:... mode:2way event:...}"**

Les différents paramètres sont:

- **from**: Obligatoire, le nom de la propriété à lier
- **converter**: optionnel, si renseigné, une méthode globale pour convertir la donnée avant la liaison.
    - @param: value: la valeur a convertir
    - @param converter_params: paramètres optionnels
- **converter_params**: optionnel, permet de passer des paramètres a la méthode de conversion.
- **fallback**: optionnel, si value==null, affichera le fallback a la place (placeholder)
- **alt**: optionnel, expérimental, permet de lier ce binding a une autre clé que celle de la propriété
- **mode=2way: obligatoire, permet de noter l'input**
- **event**: par défaut **blur**, event sur lequel réagir pour prendre en compte les changements

Pour qu'un binding mode 2way puisse fonctionner, il doit se faire sur **un élément HTML possédant une propriété contenant la valeur de l'input** (en règle générale VALUE) et **générant des events lors de sa modification**. (Ce qui est le cas pour les élément <INPUT> par exemple).

Notation:

data-binded-nom_de_la_propriete = {binding from:... mode:2way}

1: un simple exemple de binding: créer/éditer un client

Comme on ne change pas une équipe qui gagne, reprenons notre liste de client et modifions le code pour intégrer un formulaire pour pouvoir ajouter/mettre à jour les informations.

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title>Fear the walking web! Demos</title>
```

```
  <script src="../../fear-the-walking-web-flesh-and-bones.js"></script>
  <script>
function Client (n,p,s){
        //on ajoute un identifiant unique au client (comme s'il provenait d'une base de données par exemple)
        //generateUUID() est une methode définie dans le framework qui fait ce que son nom indique...
        this.id = generateUUID();
        this.nom = n;
        this.prenom = p;
        this.sexe=s;
};
myContext = {
        //une liste de clients renseignée par défaut
        clients : [ new Client("MANSON","Marylin","U"),new Client("MALLOW","Marc","M"), new Client("DALTON","M'a","F")],
        //le client sélectionné à éditer
        selected : null,

};
myContext.delete_client = function(evt, id){
        //recherche l'index du client a supprimer
        for (i=0;i<this.clients.length;i++){
                //si l'id est le bon
                if (this.clients[i].id == id){
                        //supprime le du tableau
                        this.clients.splice(i,1);
                        break;
                }
        }
```

```
}
myContext.add_client = function(evt){
	//ajoute un nouveau client à la liste
	cl = new Client("NOUVEAU","Client","F");
	this.clients.push(cl);
	//sélectionne le pour édition
	this.selected = cl;
};

myContext.edit_client = function(evt, id){
	//comme pour le delete, recherche le client à editer
	for (i=0;i<this.clients.length;i++){
		//si l'id est le bon
		if (this.clients[i].id == id){
			//Trouvé
			this.selected = this.clients[i];
			break;
		}
	}
};



	//un converter pour determiner la classe CSS a appliquer lors de la selection d'un client
function is_selected (value, params){
```

```
        if (value){

            //recupere l'index de l'objet
            for (i=0;i<myContext.clients.length;i++){
                if (myContext.clients[i].id == value.id){
                    //génère la règle css a appliquer
                    index = i+1;//nth-child commence à 1!
                    style=".list li:nth-child("+index+"){background:#E3E3E3;}";
                    return style;
                }
            }

        }
        return " ";//si pas de sélection, renvoie vide
    }

</script>

</head>
<body data-context="myContext">
    <div data-role="presenters">


        <!-- le presenter pour afficher les données du tableau -->
        <div id="client_item_presenter" data-role="presenter">
        <!--ici, on n'utilise pas un div mais un LI puisqu'on veut afficher dans un UL-->
```

```
                    <li>
                        <!-- on crée un lien avec 2 bindings: un sur l'event clic et un sur le contenu du lien... -->
                        <a href="#" data-binded-click="{binding command:edit_client command_params:$id}">Nom: {binding
from:nom}</a> 
                        <span>Prénom: {binding from:prenom}</span> 
                        <!-- On cree une commande qui prend en parametre optionnel l'identifiant du client pour pouvoir le retrouver -->
                        <a href="#" data-binded-click="{binding command:delete_client command_params:$id}">Supprimer</a>

                    </li>
                </div>



                <!-- un presenter utilisé comme formulaire -->
                <div id="client_formulaire" data-role="presenter">
                    <div>
                        <!--on cree un binding 2way vers myContext.selected.nom a partir de la propriété VALUE de l'input, et on demande la mise a
jour de l'UI par l'event input (lors de la frappe) -->
                        <input type="text" data-binded-value="{binding from:nom mode:2way fallback:'Le nom du client' event:input}"></input>
                        <!--on cree un binding 2way vers myContext.selected.prenom, et on demande la mise a jour de l'UI par l'event blur (par defaut)
-->
                        <input type="text" data-binded-value="{binding from:prenom mode:2way fallback:'Le prenom du client'}"></input>

                        <!-- binding avec un radiogroup: travaille a partir de la propriété CHECKED, c'est un peu la seule facon de faire...-->
                        <input type="radio" value="M" data-binded-checked="{binding from:sexe mode:2way}" name="sexe">Masculin</input>
                        <input type="radio" value="F" data-binded-checked="{binding from:sexe mode:2way}"name="sexe" >Feminin</input>
                        <input type="radio" value="U" data-binded-checked="{binding from:sexe mode:2way}" name="sexe" >Inconnu</input>
```

```
            </div>


            <div data-type="fallback">
                <!-- si le client est null, un simple message???? -->
                <h3>Selectionnez un client pour l'editer ou cliquez sur "Ajouter un nouveau client"</h3>
            </div>
        </div>
</div>
```

```
<!-- on tente le coup: generer un style par binding?
Dommage qu'il faille le déclarer dans le body pour prendre en compte les bindings....-->
<style>{binding from:selected converter:is_selected}</style>


<h3>Une simple liste de clients:</h3>
<ul class="list">{binding from:clients item_presenter:client_item_presenter}</ul>

<!-- Un presenter pour le client selectionné -->
<div>{binding from:selected presenter:client_formulaire}</div>
<a href="#" data-binded-click="{binding command:add_client}">Ajouter un client</a>
```

```
  </body>
</html>
```

2: quelques exemples d'input binding:

2.1: input type="text, url,....."

Le binding 2way permet de lier la propriété **VALUE**, l'event par défaut est **CHANGE**, pour spécifier une modification à chaque caractère, on passera le paramètre **EVENT:INPUT**

```
<input type="text" data-binded-value="{binding from:ma_prop
                                       mode:2way
                                       fallback:'valeur du placeholder'}"></input>
```

2.2 input type="radio"

Le fonctionnement des radiogroups est un peu particulier… Par défaut, la propriété "intéressante" du radiogroup est **VALUE** (la valeur du radiobutton sélectionné), mais c'est la propriété **CHECKED** qui permet de savoir quel radiobutton est actuellement sélectionné!

Par défaut, l'event utilisé est le **CLICK**.

*Note: les paramètres **CONVERTER** et **CONVERTER_PARAMS** sont utilisés dans le framework pour déterminer la valeur du radiogroup. C'est mieux de ne pas les utiliser dans vos binding sur radiogroup. (En fait, le converter est une méthode (nommée dans le framework check_radio_value, vers le début du fichier) qui prend en arguments la valeur de la propriété du data-context, et en argument optionnel la valeur du radiobutton cliqué. Elle revoie TRUE si les 2 valeurs sont identiques)*

```
<input type="radio" value="M" data-binded-checked="{binding from:sexe mode:2way}" name="sexe">Masculin</input>
```

2.3 input type=”checkbox”

Comme pour les radiobuttons, les checkboxs doivent lier la propriété **CHECKED** pour connaître leur état. Pour une checkbox, **la propriété VALUE est ignorée**, et le binding renverra **true/false** suivant l’état de la checkbox.

```
<input id="chk" type="checkbox" data-binded-checked="{binding from:types mode:2way}">Check me!</input>
```

2.4 input type=”text” list=”...”

Nouveau en HTML5, mais fonctionne de la même façon que les inputs text…. L’event pris en compte par défaut est **INPUT** (les changements sont pris en compte a chaque modification du texte)

```
<input type="text" data-binded-value="{binding from:possible mode:2way fallback:fr}" list="myList"></input>
<datalist id="myList">
<option value=”fr”>Bonjour</option>
<option value=”es”>Ola</option>
<option value=”en”>Hello</option>

</datalist>
```

Générer une datalist pour l'input:

Comme pour n'importe quel élément HTML, on peut créer un binding sur une datalist pour afficher une liste de choix dynamiquement.

1 On crée dans notre data-context un tableau contenant les informations de la liste a afficher:

//on crée une liste d'objets contenant les propriétés value: la valeur de l'option et label: ce qu'on affichera. C'est juste pour l'exemple, n'importe quel objet fait l'affaire.

possibles : [{value:"fr",label:"Bonjour"},{value:"en",label:"Hello"},{value:"es",label:"Ola"}]

2 On crée un presenter pour pouvoir afficher les informations des objets de la liste:

```
<div id="datalist_item_presenter" data-role="presenter">
        <option value="{binding from:value}">{binding from:label fallback:'--'}</option>
 </div>
```

3 et dans la page, on indique les bindings: comme on veut afficher un tableau, on utilise un item_presenter

```
<input type="text" data-binded-value="{binding from:possible mode:2way fallback:fr}" list="myList"></input>
<datalist id="myList">{binding from:possibles item_presenter:datalist_item_presenter}    </datalist>
```

2.5 select

L'utilisation du select se fait comme pour un input type="text". L'event par défaut est **CHANGE** et on peut aussi faire un binding sur les options à afficher dans le select…

<select data-binded-value="**{binding from:possible mode:2way}**">**{binding from:possibles item_presenter:datalist_item_presenter}**</select>

3: vérification des inputs et getter/setters et formulaires:

La validation des entrées utilisateur dans FTW² se fait via l'utilisation des **property type getter/setter javascript et les formulaires**. On 'cache' la propriété à vérifier par une nouvelle property, et dans le setter, on fait la validation. **Si elle échoue, on lance une Error** avec un message qui sera affiché à l'utilisateur. **Pour que s'affiche les messages d'erreurs, il faut créer un formulaire <FORM>** et placer les champs inputs,select… à l'intérieur.

Exemple: création d'un formulaire "client" et validation des données:

La classe représentant notre client:

```
function Client (id, n,p,s){
        this.id = id; //identifiant unique du client

        //mes propriétés cachées: rappel: une propriété commençant par un
        // underscore n'est pas visible par le framework
        this._Nom = n;
        this._Prenom = p;
        this._Sexe = s;

        //les property type getter/setter pour les validations
        // ce seront ces property qui seront visibles par le binding
```

```
Object.defineProperty(this,"nom",{
        get: function(){return this._Nom;},
        set: function(v){
                //le code de validation de la valeur, par exemple
                //n'autorise pas les valeurs nulles
                if(v == null || v == undefined || v.trim()=="") {
                        //envoie un message a l'utilisateur
                        throw new Error ("Le Nom doit etre renseigné");
                }
                 //sinon, fait les modifications
                this._Nom = v;                        }
});

Object.defineProperty(this,"prenom",{
        get: function(){return this._Prenom;},
        set: function(v){
                //n'autorise pas les valeurs nulles
                if(v == null || v == undefined || v.trim()=="") {
                        //envoie un message a l'utilisateur
                        throw new Error ("Le prenom doit etre renseigné aussi!");
                }
                this._Prenom = v;                        }
});
Object.defineProperty(this,"sexe",{
        get: function(){return this._Sexe;},
        set: function(v){
```

```
                    //n'autorise pas les valeurs autres que M,F et U
                    if(v == null || v == undefined || v.trim()=="") {
                        //envoie un message a l'utilisateur
                        throw new Error ("Le Sexe doit etre renseigné");
                    }
                    if (v=='M' || v=='F' || v=='U'){
                        this._Sexe = v;
                    }
                    else throw new Error ("Le sexe renseigné est inconnu!");

                }
            });
        };
```

Le contexte de données et les méthodes d'interactions:

```
myContext = {
            //une liste de clients renseignée par défaut
        clients : [ new Client(generateUUID(),"MANSON","Marylin","U"),
                new Client(generateUUID(),"MALLOW","Marc","M"),
                new Client(generateUUID(),"DALTON","M'a","F")],
            //le client selectionné à éditer
            selected : null,
};
//command: suppression d'un client
myContext.delete_client = function(evt, id){
            //recherche l'index du client a supprimer
```

```
            for (i=0;i<this.clients.length;i++){
                if (this.clients[i].id == id){
                    this.clients.splice(i,1);
                    break;
                }
            }
        }
//command: cree un nouveau client sans informations (ie: sans id, toutes les données a
// undefined)
        myContext.add_client = function(evt){
            cl = new Client();
            //sélectionne le pour édition
            this.selected = cl;
        };
//command: sélectionne un client pour l'édition
        myContext.edit_client = function(evt, id){
            for (i=0;i<this.clients.length;i++){
                if (this.clients[i].id == id){
                    this.selected = this.clients[i];
                    break;
                }
            }
         };

         //command: lors de la modification ou création d'un client,
        // la méthode OnSubmit a appeler
        //params: evt: l'objet event généré par la commande
```

```
//              params: dans ce cas, null
 myContext.action_submit = function(evt, params){

        evt.preventDefault();//ne pas oublier ça....
        id = this.selected.id;
       //si le client a déjà un id, met a jour, sinon, enregistre un nouveau
        if (id){
              //une simple mise a jour des infos
              console.log("mise a jour des données...");
        } else {

              //crée un id pour ce client
              this.selected.id=generateUUID();
              //ajoute a la liste
              this.clients.push(this.selected);
              //remet selected a null
              this.selected = null;
        }

        //pour éviter le rechargement de la page
        return false;
};


 //un converter pour déterminer la classe CSS a appliquer
// permet de surligner en gris la ligne du client sélectionné dans
// la liste des clients
```

```
function is_selected (value, params){

        if (value){

                //recupere l'index de l'objet
                for (i=0;i<myContext.clients.length;i++){
                        if (myContext.clients[i].id == value.id){
                                //génère la règle css a appliquer
                                index = i+1;//nth-child commence à 1!
                                style=".list li:nth-child("+index+"){background:#E3E3E3;}";
                                return style;
                        }
                }

        }
        return " ";//si pas de sélection, renvoie vide
};

//un converter pour afficher ou non le bouton submit (ne le fait apparaître
// que pour la création d'un client)
//si id existe, renvois 'block', sinon "none" comme valeur du display
function got_id (v, params){
        if( v == undefined ) return 'display:block;';
        else return 'display:none;';
};
```

Dans le HTML, on modifie le presenter “client_formulaire” pour ajouter notre formulaire.

```
<!-- un presenter utilisé comme formulaire -->
        <div id="client_formulaire" data-role="presenter">

            <form  data-binded-submit="{binding command:action_submit}">
                <!--on cree un binding 2way vers myContext.selected.nom, et on demande la mise a jour de l'UI par l'event input -->
                <input type="text"
                data-binded-value="{binding from:nom mode:2way fallback:'Le nom du client' event:input}" required></input><br/>

                 <!--on cree un binding 2way vers myContext.selected.prenom, et on demande la mise a jour de l'UI par l'event blur (par defaut)
-->
                <input type="text"
                        data-binded-value="{binding from:prenom mode:2way fallback:'Le prenom du client'}" required></input><br/>

                <!-- binding avec un radiogroup utilise l'event click pour recuperer les changements, c'est un peu la seule facon de faire...-->
                <input type="radio" value="M" data-binded-checked="{binding from:sexe mode:2way}" name="sexe">Masculin</input>
                <input type="radio" value="F" data-binded-checked="{binding from:sexe mode:2way}"name="sexe" >Feminin</input>
                <!-- ici, indique un fallback a true pour séléctionner cette radio par défaut -->
                <input type="radio" value="U"
                        data-binded-checked="{binding from:sexe mode:2way fallback:true}" name="sexe">Inconnu</input><br/>

                <!-- on ajoute un submit au cas ou... -->
                <input type="submit" data-binded-style="{binding from:id converter:got_id forceAttr:true}" value="Enregistrer" ></input>
            </form>
```

```
                <div data-type="fallback">
                    <!-- si le client est null, un simple message???? -->
                    <h3>Sélectionnez un client pour l'éditer ou cliquez sur "Ajouter un nouveau client"</h3>

                </div>
            </div>
```

Et pour finir, l'affichage de la page:

```
<!-- on tente le coup: generer un style?
      Dommage qu'il faille le déclarer dans le body pour prendre en compte les bindings....-->
<style>{binding from:selected converter:is_selected}</style>


      <h3>Une simple liste de clients:</h3>
        <!-- les clients déjà dans la liste -->
      <ul class="list">{binding from:clients item_presenter:client_item_presenter}</ul>
        <!-- la place de mon formulaire: si myContext.selected == null, affichera le message du data-type="fallback", sinon, affichera le formulaire.
-->
      <div>{binding from:selected presenter:client_formulaire}</div>
    <a href="#" data-binded-click="{binding command:add_client}">Créer un nouveau client...</a>
```

ReportValidity() et les navigateurs ne le supportant pas:

Pour gérer l'affichage des messages d'erreurs, FTW² repose sur les méthodes setCustomValidity() et reportValidity() de HTML5. Étonnamment, ma version de Firefox (43.0.1) ne semble pas disposer d'une implémentation pour le reportValidity et n'affiche donc pas les bubbles box d'informations! Un workaround simple consiste a créer à la fin du formulaire un **<input type="submit" style="display:none;">.** Lors de la modification de la valeur d'un champs du formulaire, si il y a une erreur, le framework simulera un click sur ce bouton et affichera les messages pendant 2 secondes.

Note: Pour Flesh&Bones, je ne voulais pas rajouter de CSS ou modifier le HTML de la page, seulement utiliser le javascript. Pour PANDEMIE, le système changera peut être en interne pour utiliser une validation et affichage des messages d'erreurs fonctionnant avec tous les navigateurs de la même façon…

9: FEAR THE WALKING WEB-FLESH&BONES:

Comme un exemple est plus explicite qu'une doc, je propose de développer une application complète avec FTW². Vous pouvez voir *l'application finie ici*.

Pour la récupération des données, j'ai choisi le service web Microsoft, qui a l'avantage de renvoyer les données au format JSON, ce qui simplifie la vie (contrairement au CSV de Yahoo).

the Wolf of WallStreet

Cette application servira a visionner les cours boursiers d'entreprises. La page affichera une zone d'infos sur l'entreprise (le nom, la place boursière, la valeur actuelle et l'évolution), un graphique SVG pour afficher le cours (sur 1 jour, 1 semaine, 1 mois, 1 année ou 5 ans), et une zone d'informations générales (volume, close, plage, amplitude...).

On pourra rechercher des actions en entrant les premières lettres de leurs nom dans une zone de texte, et l'application proposera des entreprises possibles.

Screenshots de l'application

Mise en place du projet

Créez un dossier wows sur votre serveur (j'utilise une version de ***WampServer*** installée en locale, dans ce cas, 'C:/wamp/www/wows'). Dans ce dossier, **on ajoutera le fichier fear-the-walking-web-flesh-and-bones.js** , puis on créera 3 fichiers pour notre application:

- **index.html**: contiendra le code html de notre application
- **wows.js**: contiendra le code javascript nécessaire pour son fonctionnement
- **wows.css**: contiendra les styles pour les affichages

Création d'un data-context pour l'application

La première chose a faire, dans FTW², est de créer un **Data-context** pour l'application. Cela consiste en un plain object javascript de portée globale que l'on défini dans le fichier wows.js. Celui ci servira de conteneur à toutes les propriétés que l'on voudra afficher dans l'application.

fichier: wows.js

```
wowsContext = {
   app_name: "Wolf of WallStreet"
}
```

Maintenant, on passe au fichier html: on défini le charset et **on importe les scripts et css** dans le head (l'ordre des imports n'a pas d'importance), on crée le body, **on renseigne le data-context** pour l'application et **on place notre conteneur à presenters** (on en aura besoin).

fichier index.html

```
<!DOCTYPE html>
<head>
```

```
  <meta charset="utf-8">
  <title>Wolf Of WallStreet</title>
    <link rel="stylesheet" href="wows.css"/>
        <script src="fear-the-walking-web-flesh-and-bones.js"></script>
        <script src="wows.js"></script>
</head>
<body data-context="wowsContext">
        <div data-role=”presenters”>
        </div>
        <div id=”recherche”>
                <!-- la zone de recherche des cotations -->
        </div>
        <div id=”stock”>
                <!-- la zone d’affichage des informations sur le stock -->
        </div>
</body>
```

Récupération des données entreprises par le webservice bing.finance:

Pour pouvoir récupérer les informations d'un stock (cotation), **il faut fournir au webservice le code unique de l'entreprise** (ressemblant à quelque chose comme "160.1.ALO.PAR" pour Alstom, bourse de Paris). Heureusement, BING fourni aussi un service qui, à partir du nom (ou début du nom) d'une entreprise, va renvoyer les noms et codes des entreprises possibles. Un exemple d'appel au webservice de résolution de nom serait:

http://finance.services.appex.bing.com/Market.svc/MTAutocomplete?q=alsto&locale=FR-FR&count=5

Les paramètres sont:

q: ce qu'on recherche (query)

locale: la langue de la réponse (ici, FR-FR)

count: le nombre de réponses maximum désirées (ici, 5)

Le webservice renvoie un objet JSON avec 2 propriétés : **COUNT**: le nombre de réponses renvoyées et **DATA**: un tableau contenant les réponses sous forme d'Objets. Parmi les propriétés de ces objets, on s'intéressera uniquement à:

FullInstrument: le code unique de l'entreprise (par exemple: 160.1.ALO.PAR)

LocaleName: le nom de l'entreprise (par exemple "ALSTOM")

AC040: le nom de la place boursière (par exemple CAC40)

Dans notre <DIV id=”recherche”>, on va rajouter une zone de saisie, qui a chaque caractère tapé enverra une requête pour récupérer les codes et affichera les résultats dans une liste. Cliquer sur un élément de la liste permettra de sélectionner l’entreprise à afficher.

HTML:

Dans la page html, on place la zone de recherche <INPUT> et une liste <UL> pour afficher les résultats de la recherche.

```
<div id="recherche">
    <!-- une zone de recherche, met a jour la propriété a chaque caractère entré -->
    <input id='searchinput' type="text"
        data-binded-value="{binding from:search_txt mode:2way fallback:'search for stock' event:input}"/>
    <!-- les réponses: un binding de tableau, utilise le presenter suggestionItem pour afficher les éléments du tableau, et ne fait pas de binding sur les objets du tableau (deep=false), le tableau change a chaque requete -->
    <ul>{binding from:suggestList item_presenter:suggestionItem deep:false}</ul>
</div>
```

Puis on crée les presenters nécessaires pour afficher les informations d’une entreprise. On ne s'intéressera qu’au code, nom et place boursière.

```
<div data-role="presenters">
    <div data-role="presenter" id='suggestionItem'>
            <!-- pour notre tableau on crée un presenter type LI et on place une commande dessus qui prend en paramètre le code unique de l'entreprise -->
        <li class="suggest" data-binded-click="{binding command:selectNewStock command_params:$FullInstrument}" >
         <!--  un binding tout simple pour le nom de l'entreprise -->
          <h2 class="stck_place">{binding from:FriendlyName fallback:'no data'}</h2>
          <!-- idem pour la place boursière -->
           <h5 class="stck_place">{binding from:AC040 fallback:'no data'}</h5>
        </li>
</div>
```

et un peu de mise en style pour la forme…

CSS:

```
html{
        color:#181D20;
```

```
        text-shadow: 1px 2px #d3d3d3;
        font-family: Arial, Helvetica, sans-serif;
}
/* le div recherche */
#recherche{
        display:table;
        width:20%;
        float: left;
}
/* la liste de résultats */
#recherche ul{
        display:table-row;
        list-style:none;
        padding-left:0;
}
/* les éléments de la liste */
```

```
#recherche ul li {
        text-align:center;
        padding-top:20px;
        cursor:pointer;
}
/* zone de saisie */
#recherche input {
  padding-top:30px;
  margin-bottom:30px;
  font-size:1.3em;
  display:table-row;
}
/* le look: simplement une barre grise en bottom
note: dans chrome, la zone de texte en focus aura un border ajouter, je ne vois pas
comment le supprimer…. */
#recherche ul li ,
```

```
#recherche input {
  margin:0;
  vertical-align:bottom;
  border: none;
  border-bottom: solid 2px #c9c9c9;
  transition: border 0.3s;
  font-family: sans-serif;
  width:100%;
}
#recherche ul li:hover ,
#recherche ul li.hover ,
#recherche input.focus ,
#recherche input:focus {
  border-bottom: solid 2px #45B2FB;
}
/* styles pour le nom de l'entreprise et la place boursière */
```

```
.stck_name {
        font-size:1.3em;
        margin:2px;
        padding:0px;
}
.stck_place {
        font-style:italic;
        font-size:1.0em;
        margin:2px;
        margin-bottom: 5px;
}
```

Coté Javascript, on commence par créer 2 propriétés dans le contexte de données:

_search_text: pour la zone de saisie, comme on veut a chaque modification de sa valeur lancer une requête AJAX, on la rend invisible (le underscore devant!) et on lui adjoint une property type getter/setter pour le binding. Dans le setter, on lancera les requêtes ajax.

suggestList: pour enregistrer la liste des réponses du webservice de résolution de nom.

La variable **xhr_delay** sert juste à éviter d'envoyer trop de requête au serveur en mettant un délais entre la modification de **_search_text** et l'envoie réel de la requête.

fichier: wows.js

```
xhr_delay = null;//pour éviter de flooder le serveur de requêtes


wowsContext = {
   app_name: "Wolf of WallStreet",
   _search_txt : null, //la recherche d'action
   suggestList : null, //les suggestions obtenues
   selectNewStock : function (evt, id){
      //la commande lors du click sur un <LI>, récupére en argument le code (id) de l'entreprise selectionnée
  },

};
```

on défini la property 'bindable' et a chaque modification, on lance une requête XHR pour avoir les codes des entreprises.

```
Object.defineProperty(wowsContext,"search_txt",{
    get: function(){return this._search_txt;}, //renvoie simplement la valeur


    set: function(value){


        if (value == null || value.trim() == "") throw "Recherche invalide: merci d'entrer quelque chose a rechercher...";
        this._search_txt = value;
        //charge les reponses via le webservice
         if (xhr_delay != null) clearTimeout(xhr_delay);//annule la requête précédente
                //crée l'URL pour la requête
                var url = "http://finance.services.appex.bing.com/Market.svc/MTAutocomplete?q="
                +search
                +"&locale=FR-FR&count=5";


                xhr_delay = setTimeout(function(){
                        clearInterval(xhr_delay);
```

```
                xhr_delay = null;//fin de l'attente
                //juste une méthode pour me simplifier l'envoie de requête ajax.
                //prends en paramètre l'url du webservice, le nom de la méthode a appeler en cas de réussite de l'appel,
                // les noms des méthodes a appeler en cas d'erreur ou de timeout...
                __load_async_json (url,"show_suggestions",error_loading,timeout_error);
            }, 300);//attends 300ms avant de lancer la requête, au cas ou
            //ajoute des lettres
    }
});

//récupère les données du webservice et affiche dans la liste
wowsContext.show_suggestions = function(data){
        if (datas.count>0){
            this.suggestList = datas.data; //enregistre directement les objets crées par JSON, normalement, on devrait créer nos
            //propre type de données...
        } else {
```

```
            this.suggestList = [];//renvoie une liste vide, pas de réponses….
        }
};

//gestion du xhr
var xhr_timeout = 4000;//4 secondes avant annulation de la requete XHR

//charge les données d'une URL via XHR
//param url: l'url ou se connecter
//param handler: le nom de la méthode a appeller si le chargement a reussi
//param error_handler: nom de la méthode a appeller si erreur
//param timeout: nom de la méthode a appeller si timeout
function __load_async_json( url, handler, error_handler, timeout){
  var xhr = new XMLHttpRequest();

  xhr.onload = function(e){
```

```
        if(xhr.status !== 200) {
                //erreur
                if (error_handler) error_handler(xhr.responseText);
                else alert("error loading JSON file");
                return;
        }
        json_datas = xhr.response;
        datas = JSON.parse(json_datas);

        //juste pour pouvoir avoir un 'this' qui veut dire quelquechose....
        wowsContext[handler](datas);
}
if (error_handler){
  xhr.onerror = error_handler;
}
if(timeout){
```

```
        xhr.timeout = xhr_timeout;
        xhr.ontimeout = timeout;
    }

    xhr.open('GET', url);
    xhr.send();
};
//une méthode basique pour afficher une erreur dans la console
error_loading = function(err){
        console.log("erreur: "+err);
};
//une méthode basique pour afficher un timeout dans la console
timeout_error = function(err){
        console.log("timeout: "+err);
};
```

Gestion des erreurs:

Lorsqu'on fait une recherche d'entreprise, on peut rencontrer 2 types d'erreurs: une erreur de connection ou de timeout (pour simplifier le code, on ne les distinguera pas), et une erreur de type "pas de réponses trouvées" si le aucune entreprise correspond à la recherche.

Erreur de connection/timeout:

En place d'un "console.log" pas très user-friendly, on va rajouter dans le presenter **'suggestionItem'** un data-type pour gérer les objet **Error,** un simple élément de liste qui demandera a afficher la propriété **message**:

```
<div data-role="presenter" id='suggestionItem'>
    ...
    <li data-type="Error">
        <h3 class="stck_name">{binding from:$message}</h3>
    </li>
</div>
```

En javascript, on modifie les méthodes error_loading et timeout_error pour modifier la liste de résultats en fonction:

```
//une méthode basique pour afficher une erreur dans la console

error_loading = function(err){

    wowsContext.suggestList = [err];

};
```

```
//une méthode basique pour afficher un timeout dans la console
timeout_error = function(err){
    wowsContext.suggestList = [err];
};
```

Erreur type “pas de réponses”:

Pour gérer le cas ou aucune réponses n’a été trouvée, on va créer un nouveau presenter “**noSuggestions**” pour afficher un message d’erreur à l’utilisateur et renseigner le paramètre **EMPTY** du binding. *Rappel: dans un item_presenter (affichage de liste), le paramètre fallback sert a gérer le cas ou la propriété est NULL, le paramètre empty sert dans le cas ou la liste est VIDE (length==0)*

```
<div data-role="presenters">
    …
    <div data-role="presenter" id='noSuggestions'>
        <li><h3  class="stck_name">Aucune réponses pour la recherche...</h3></li>
    </div>
</div>
```

et on modifie le binding de la liste:

{binding from:suggestList item_presenter:suggestionItem deep:false **empty:noSuggestions**}

Récupérer les informations de cotation d'une entreprise:

Pour récupérer les informations de cotation, on va lancer une requête vers le webservice BING dédié. Un exemple d'URL est:

http://finance.services.appex.bing.com/Market.svc/M-TodayEquityV4?rtSymbols=160.1.ALS.PAR&chartSymbols=160.1.ALS.PAR&chartType=1Y&lang=FR-FR&localizeFor=FR-FR

Les paramètres sont:

rtSymbols: le ou les codes d'entreprises dont on veut récupérer les informations (ici, on en récupérera qu'une seule)

chartSymbols: le ou les codes d'entreprises dont on veut récupérer les cotations

chartType: la durée de cotation voulue (de 1jour à 5 ans)

lang & **localizeFor**: la langue de réponse désirée.

On obtient en retour un objet JSON avec les propriétés suivantes:

FrNm: nom de l'entreprise (en francais)

FlIns: code unique de l'entreprise

EExNm: place boursière

market: place boursièrer (le nom)

Lp: valeur de la cotation

Ch: gain de la cotation (du jour?)

Chp: gain de la cotation (pourcentage)

Yl, Yh: amplitude sur 1 an

Op: valeur a l'ouverture

Pp; valeur a la fermeture précédente

V: volume d'échage

avgV: volume d'échange moyen

Mc: capitalisation de l'entreprise

Pe: gain depuis l'introduction en bourse.

ChartDatas: tableaux de valeurs de la cotation

Note: il y a beaucoup d'autres informations renvoyées par le webservice, mais dans ce tutoriel, on se limitera à ceux la

Création des objets et récupérations des informations.

Cette fois, nous allons créer 2 classes pour empaqueter les informations sur les actions: StockData pour les informations de base (nom, code, market…) et une classe Extra pour des informations annexes (volume, capitalisation…).

NOTE: la classe Extra est un peu tiré par les cheveux, j'avoue, c'est juste pour illustrer le fait qu'on peut inclure un presenter dans un presenter, ou un item_presenter dans un presenter.

```
function Extra (){
    //les informations annexes pour la cotation...
    this.open = null;//valeur a l'ouverture
  this.close = null;//valeur a la cloture precedente
  this.volume = null;//volume d'echange
  this.avgVolume=null;//volume d'echange moyen
  this.capitalisation = null;//capitalisation boursiere
  this.gain = null;//gain de l'action depuis sa mise en bourse

  //permet d'initialiser un objet Extra a partir des informations d'un objet JSON récupérer via le webservice
  this.copyFromJSON = function(infos){
```

```
        this.open = infos.Op;
        this.close = infos.Pp;
        this.volume = infos.V;
        this.avgVolume=infos.avgV;
        this.capitalisation = infos.Mc;
        this.gain = infos.Pe;
    }
}
function StockData (){
        //les informations sur la cotation
        this.name = null;
        this.code = null;
        this.place = null;
        this.market = null;
        this.value = null;
        this.gain = null;
```

```
this.pgain = null;
this.amplitude = null;
//les informations annexes
this.extra = new Extra();

this.copyFromJSON = function(data){
        //initialise un objet StockData avec les données récupérées via le webservice
        this.name = data.FrNm;
        this.code = data.FlIns;
        this.place = data.EExNm;
        this.market = data.market;
        this.value = data.Lp;
        this.pgain = data.Chp;
        this.gain = data.Ch;
        this.amplitude = [data.Yl,data.Yh];
```

```
            this.extra.copyFromJSON(data);
        }
}
```

Dans le contexte de données, on crée une nouvelle propriété “stock” qui représentera la cotation sélectionnée par l’utilisateur.

```
wowsContext = {
    app_name: "Wolf of WallStreet",
    _search_txt:null, // le nom de l'entreprise a rechercher
    suggestList:null, //les resultats de la recherche
    stock : new StockData(),//le stock a afficher
};
```

Puis on modifie la méthode “wowsContext.**selectNewStock**” pour lancer une requête xhr et récupérer les informations de cotations. Si elle réussie, la méthode “wowsContext.**copy_stock_details**” sera appellée pour mettre a jour l’affichage:

```
wowsContext.selectNewStock = function(evt, id){
        if (id != undefined){
```

```
            this.stock_infos = null;
        url = "http://finance.services.appex.bing.com/Market.svc/M-TodayEquityV4?rtSymbols="
        +id
        +"&chartSymbols="+id
        +"&chartType=1Y&lang=FR-FR&localizeFor=FR-FR";
          //chargement des données
          __load_async_json (url,"copy_stock_details" );
        }
}
//recupere et affiche les infos du stock
wowsContext.copy_stock_details = function(data){
        obj = data.Rtd[0];//les résultats se trouvent ici!
        this.stock.copyFromJSON(obj);//copie les données dans notre objet perso
}
```

Affichage des informations:

On va maintenant modifier le <DIV id=”stock”> pour qu’il affiche les informations chargées dans notre propriété wowsContext.stock :

```
<div id="stock">
    <div>{binding from:stock presenter:stock_presenter}</div>
</div>
```

et créer le presenter “**stock_presenter**” qui sera chargé d’afficher les propriétés d’un objet de type **StockData**:

```
<div data-role="presenter" id="stock_presenter">
        <div>
            <div id=”infos”>
                <div class="inline">
                    <h2 class="stck_name">{binding from:name fallback:--}</h2><br/>
                    <h4 class="stck_place" >{binding from:place fallback:'no data'}</h4>
                    <h4 class="stck_place" >({binding from:market fallback:'--'})</h4>
                </div>
```

```
            <div id="values" >
                        <h4 class="stck_place">
                                    {binding from:gain fallback:'--'}
                          </h4> 
                         <h4 class="stck_place">{binding from:gain fallback:'--'}%</h4>
                          <h1 class="stck_value">{binding from:value fallback:'--'}</h1>
                 </div>
        </div>
          <div id=”graph”>
                 <svg>
                 </svg>
          </div>
        <!-- ici, on inclue un presenter pour afficher la propriété “extra” -->
        <div id="extra">{binding from:extra presenter:extra_presenter}</div>
</div>
```

```
    </div>
```

Et donc, on en profite pour créer aussi le presenter pour le type **Extra**:

```
<div data-role="presenter" id="extra_presenter">
            <ul id="st_infs" class="center_horizontal">
                <li><span class="label" >Ouverture</span>
                    <span class="datas">{binding from:open fallback:--,--}</span>
                </li>
                <li>
                    <span class="label">Cloture precedente</span>
                    <span class="datas">{binding from:close fallback:--,--}</span>
                </li>
                <li>
                    <span class="label" >Volume</span>
                    <span class="datas">{binding from:volume fallback:--,--}</span>
                </li>
                <li>
```

```
                    <span class="label" >Volume moyen</span>
                    <span class="datas">{binding from:avgVolume fallback:--,--}</span>
                </li>
                <li>
                    <span class="label" >Capitalisation (Mrds)</span>
                    <span class="datas">{binding from:capitalisation fallback:--,--}</span>
                </li>
                <li>
                    <span class="label" >Prix/Gain (%)</span>
                    <span class="datas     ">{binding from:gain fallback:--,--}</span>
                </li>
            </ul>
        </div>
```

Pour finir, un peu de style pour l'affichage:

wows.css:

```
h1, h2, h3, h4, h5{
        line-height: 1em;
        margin: 0;
        display: block;
}

#stock_infos{
        float:right;
        width:80%;
}
#extra >div{
        width:75%;
}
#extra ul{
        display:table;
        list-style:none;
```

```
        margin:2%;
        margin-top:10px;
        width:90%;
}
#extra ul >li{
        display:table-row;
}
#extra ul >li:last-child>span {
        border-bottom:none !important;
}
.label{
        font-size: 0.8em;
        font-style: italic;
        color: #a6a6a6;
        display: table-cell;
        border-bottom: 1px dashed #e3e3e3;
```

```
        padding-bottom: 10px;
        padding-top:10px;
}
.datas {
        display: table-cell;
        font-size: 1.1em;
        font-weight: bolder;
        text-align: right;
        border-bottom: 1px dashed #e3e3e3;

}
.stck_name{
    line-height: 30px;
    margin-right: 10px;
    display:inline-block;
}
```

```
.inline {display: inline-block;}
#values{float:right;}
#infos{padding-left:10px;
    padding-right:10px;}
```

améliorer les affichages: signToColor, Localize:

Pour rendre les affichages un peu plus lisibles, **on va localiser les différentes valeurs**, c.a.d. transformer un "2.3456" en une version francisée avec 2 chiffres après la virgule: "2,35". Dans le fichier wows.js, **on crée une méthode de conversion "localize"** qui prends en paramètres la valeur du binding, et un extra "after_dot_count" qui, s'il est renseigné, permet de préciser le nombre de chiffres après la virgule:

```
function localize(value, after_dot_count){
  //remplace le point par une virgule
  if (value == null) return "";
  after = after_dot_count || 2;
  return  value.toLocaleString("fr-FR",{maximumFractionDigits:after, minimumFractionDigits:after});
```

```
}
```

Et, à chaque valeur numérique, on renseigne le converter pour avoir l'affichage en français, par exemple,le binding des informations de gains/valeur actuelle deviendrait:

```
<div id="values" >
        <h4 class="stck_place">
            {binding from:gain converter:localize  fallback:'--'}
          </h4> 
         <h4 class="stck_place">{binding from:pgain  converter:localize fallback:'--'}%</h4>
          <h1 class="stck_value">
            <!-- ici, on demande 3 chiffres après la virgule! -->
            {binding from:value  converter:localize convereter_params:3 fallback:'--'}
          </h1>
</div>
```

De même, **on va modifier la couleur d'affichage des gains/valeurs suivant le signe du gain**, rouge si la valeur est négative, vert si elle est positive. On commence par créer 2 nouvelles classes CSS:

```
.negatif{color:red;}
.positif{color:green;}
```

Puis une méthode de conversion qui prendra en valeur le gain de l'action et renverra le nom de la classe CSS à utiliser suivant le signe:

```
function signToColor(value){
    if(value < 0) return "negatif";
    else return "positif";
}
```

Et on crée de nouveaux bindings, cette fois sur l'attribut "class" de nos éléments HTML (qu'on renomme **data-binded-class**):

```
<div id="values" >
        <h4 data-binded-class="stck_place {binding from:gain converter:signToColor}">
            {binding from:gain converter:localize  fallback:'--'}
        </h4> 
        <h4  data-binded-class="stck_place {binding from:gain converter:signToColor}">
```

```
                {binding from:pgain  converter:localize fallback:'--'}%
            </h4>
            <h1  data-binded-class="stck_value {binding from:gain converter:signToColor}">
                <!-- ici, on demande 3 chiffres après la virgule! -->
                {binding from:value  converter:localize convereter_params:3 fallback:'--'}
            </h1>
        </div>
```

SVG, affichage des informations sous forme de graphiques:

Cette fois ci, on va travailler avec les données de ChartDatas pour afficher dans un graphique SVG l'évolution de la valeur de la cotation dans le temps. **On ajoute une propriété "charts"** dans notre objet "**StockData**" et on l'initialise avec les autres dans la méthode "copyFromJSON":

```
function StockData (){
    ...
    this.charts = null;
    this.copyFromJSON = function(data, charts){
```

```
            ...
            this.charts = charts;//les valeurs dans le temps
        }
    }
```

On modifie aussi la méthode **wowsContext.copy_stock_details** pour passer les données des graphique à la méthode copyFromJSON:

```
    wowsContext.copy_stock_details = function(data){
        obj = data.Rtd[0];
        this.stock.copyFromJSON(obj, data.Charts[0].Series);
    }
```

Et on demande a afficher un graphique SVG dans notre page HTML:

- dans le tag <SVG>, **on ajoute un tag <POLYGON>** qui représentera notre graphique
- **on crée le binding sur l'attribut "POINTS"** (qu'on renomme "data-binded-points") et on force le binding sur l'attribut html plutôt que la propriété de l'objet SVGElement. (pour laisser au soin du navigateur les conversions string/SVGPoints)
- on indique d'**utiliser un converter "linearise"** en passant en paramètre l'amplitude de la courbe.

```
<div id="graph">
    <svg class="graph center_horizontal" id="svg_graph" viewBox="0 0 100 100" preserveAspectRatio="none">
        <polygon fill="#8FD1FD" data-binded-points="{binding from:chart converter:linearise converter_params:$amplitude forceAttr:true}"></polygon>
        <!-- on dessine juste des axes tout simplement -->
        <line id="axey" x1="0" y1="0" x2="0" y2="100%" style="stroke:#a6a6a6;stroke-width:0.2"></line>
        <line id="axex" x1="0" y1="100%" x2="100%" y2="100%" style="stroke:#a6a6a6;stroke-width:0.1"> </line>
    </svg>
</div>
```

Converter pour les données du chart:

```
function linearise(value, plage){
    //linearise le tableau de données pour dessiner les courbes
    //comme svg travaille avec des objets particuliers
    //on passe par les attributs et on laisse faire le navigateur
```

```
//le converter est appelé avant toute chose, vous devez vérifier si la valeur est null ou undefined
// et agir en conséquence
    if (value==null || value == undefined) return “”;
    total = value.length;
    if (plage == null || plage[0]==null || plage[1]==null) plage=[0,100];
    step_x = (100 / total);
    step_y = 100 /(plage[1]-plage[0])

    dts = "";
    for (i=0; i < total; i++){
        //converti en %
        //chaque objet de value possède une propriété P qui contient la valeur
        v = step_y * (plage[1]-value[i].P );
        x = step_x * i;
        dts+= x+","+v+" ";
    }
```

```
        return  "0,100 "+dts+"100,100";
}
```

Comme d’habitude, on style le graphique...

```
.graph{
        width: 80%;
        height: 450px;
}
.center_horizontal{
        position: relative;
        left: 50%;
        transform: translate(-50%,0);
}
```

Note Renommer POINTS en DATA-BINDED-POINTS:

Il est vivement conseillé, quand on travaille avec le SVG, de renommer les attributs que vous allez "binder" avec le préfixe "data-binded-". Certains navigateurs (Edge, pour ne pas le nommer), lorsqu'il rencontre un attribut SVG invalide, semble tout simplement annuler le rendu et merci, au revoir. Heureusement, les attributs inconnus (et préfixés par data) ne semblent pas poser problèmes…

Des repères visuels pour les valeurs:

On va rajouter au graphique des repères visuels à 20%, 40%, 60% et 80% de la hauteur du graphique pour rendre la lecture plus facile. A chaque cran, on dessine une ligne horizontale en pointillée et en bout, la valeur que represente la ligne (ainsi, si les données sont comprises entre 100 et 200, on affichera les lignes correspondantes à 120, 140, 160 et 180).

On rajoute dans le SVG un binding sur "**amplitude**" (les valeurs min et max sur 52 semaines), on utilise un converter "toReferences" qui renverra à partir des informations d'amplitudes un tableau de 4 valeurs, et un item_presenter (puisque tableau) pour afficher les lignes.

```
<div id="graph">
    <svg …>
        <polygon …>
```

```
        <!-- g designe un groupe en svg, c'est un conteneur, un peu comme un div -->
        <g>{binding from:amplitude  item_presenter:stckref converter:toReferences fallback:nosvgdatas}</g>
    </svg>
```

On crée le presenter **stckref** et **nosvgdatas** dans le cas ou on n'a pas de données. Comme le SVG utilise des SVGElement et pas des HTMLElement, **on crée nos presenters avec une balise <SVG>** et pas <DIV> (sinon, il n'y aurait pas de rendu), et **on binde les attributs** y1, y2, etc… **en les préfixant par "data-binded" et en précisant forceAttr:true** pour éviter d'irriter les navigateurs tatillons.

```
<div data-role="presenters">
    …
    <svg data-role="presenter" id="stckref">
        <g>
            <line   x1="0" data-binded-y1="{binding from:i forceAttr:true fallback:'0'}"
              x2="100" data-binded-y2="{binding from:i forceAttr:true fallback:'0'}"
             style="stroke:#2A3138;stroke-width:0.2;" stroke-dasharray="1,1"></line>
```

```
                <text  x="99%" data-binded-y="{binding from:i forceAttr:true fallback:'0'}"
                        text-anchor="end" font-size="2.3" fill="#a6a6a6"
                        stroke="#a6a6a6;stroke-width:0.1;">{binding from:v  converter:localize fallback:'0'}</text>
            </g>
        </svg>
        <svg data-role="presenter" id="nosvgdatas">
            <text  x="50%" y="50%" text-anchor="middle" font-size="4" fill="#a6a6a6"
                    stroke="#a6a6a6;stroke-width:0.1;">no datas</text>
        </svg>
```

Ne reste plus qu'a créer la méthode de conversion d'amplitude:

```
function toReferences (value){
    //renvois les valeurs a 20, 40, 60 et 80% de la hauteur, suivant l'amplitude a 52 semaines (ie: valeurs
    // a 0%=Yl et 100%=Yh
    //sous la forme: i: le pourcentage de la hauteur, v: la valeur a cette hauteur
```

```
    if (value == null) return null;

    total = value[1]-value[0];
    min = value[0]
    //les 4 indexs qui nous interresse...

    v= [ {i:'80',v:min+(total*0.2)},{i:'60',v:min+(total*0.4)},{i:'40',v:min+(total*0.6)},{i:'20',v:min+(total*0.8)} ];

    return v;
}
```

Choisir la durée d'affichage:

Pour en finir avec l'application de démonstration, **on va donner la possibilité à l'utilisateur de choisir la durée du chart**. Par défaut, on propose 1 an, on va permettre de choisir d'afficher l'évolution pour la journée, la semaine, le mois, l'année ou 5 ans.

On va créer un radiogroup sous le graphique, lié à une propriété (disons: time) du contexte de données, à chaque click sur un bouton, on rechargera les informations de cotation.

On commence par créer une nouvelle propriété dans notre data-context, comme on veut agir lors de sa modification, on crée donc une propriété 'invisible'(précédée par un underscore: ici **_time**) et on lui adjoint un setter (dont le nom servira pour les bindings, ici **time**)

```
wowsContext = {
        …
        _time = "1Y", //par défaut, vaut 1 an
}


Object.defineProperty(wowsContext,"time",{
    get:function(){
        return this._time;
    },
    set:function(value){
        this._time=value;
        //recharge les données de cotation seulement si on en affiche une...
        if(this.stock.code!=null) this.selectNewStock(null, this.stock.code);
```

```
    }
})
```

Ensuite, on modifie la méthode **selectNewStock** pour prendre en compte notre nouveau paramètre **_time**:

```
wowsContext.selectNewStock = function(evt, id){
        if (id != undefined){
                url = "http://finance.services.appex.bing.com/Market.svc/M-TodayEquityV4?rtSymbols="
        +id
        +"&chartSymbols="+id
        +"&chartType=”+this.time+”&lang=FR-FR&localizeFor=FR-FR";
          //chargement des données
          __load_async_json (url,"copy_stock_details", error_loading, timeout_error );
        }
}
```

Dans notre HTML, sous le presenter pour notre objet stock, on rajoute notre radiogroup et on crée les bindings 2way:

```
<div id="stock">
        <div>{binding from:stock presenter:stock_presenter}</div>
```

```
<!-- menu pour les jours -->
<div class="ui-rdgroup-horizontal center_horizontal">
        <input type="radio" data-binded-checked="{binding from:time mode:2way}" value="1D" name="time" id="time_j"/>
        <label for="time_j">Jour</label>
        <input type="radio" data-binded-checked="{binding from:time mode:2way}" value="5D" name="time" id="time_s"/>
        <label for="time_s">Semaine</label>
       <input type="radio" data-binded-checked="{binding from:time mode:2way}" value="1M" name="time"  id="time_m"/>
        <label for="time_m">Mois</label>
       <input type="radio" data-binded-checked="{binding from:time mode:2way}" name="time" value="1Y" id="time_a"/>
        <label for="time_a">Année</label>
        <input type="radio" data-binded-checked="{binding from:time mode:2way}" name="time" value="5Y" id="time_5a"/>
        <label for="time_5a">5 ans</label>
    </div>
</div>
```

Et un peu de style pour finir:

```
.ui-rdgroup-horizontal {
        display:inline-block;
        width: 80%;
        text-align: center;
}

.ui-rdgroup-horizontal > input[type=radio]{
        display:none;
}
.ui-rdgroup-horizontal >input[type=radio] + label{

        background-color:inherit;
        font-size: 1.1em;
        color: inherit;
        text-align:center;
        margin-right: 10px;
```

```
    white-space: nowrap;
}
.ui-rdgroup-horizontal >label {cursor:pointer;}
.ui-rdgroup-horizontal >label:last-child{
    margin-right: 0px !important;
}
.ui-rdgroup-horizontal >input[type=radio]:checked + label{
    color:#45B2FB;
}
```

Loading widget:

Pour faire patienter l'utilisateur lors du chargement des informations de cotation, on va mettre en place un widget de chargement (il n'est pas de moi, je l'ai trouvé sur le net mais impossible de me rappeler où…) uniquement CSS.

wows.css

```
/* animation css de chargement*/
```

```
.nospinner {display:none;}
.spinner {
  width: 40px;
  height: 40px;

  z-index: 100;
  background-color: #45B2FB;
 display: block;
  margin: 100px auto;
  -webkit-animation: sk-rotateplane 1.2s infinite ease-in-out;
  animation: sk-rotateplane 1.2s infinite ease-in-out;

position: absolute;
  top:20%;
  left:50%;
}
```

```
@-webkit-keyframes sk-rotateplane {
  0% { -webkit-transform: perspective(120px) }
  50% { -webkit-transform: perspective(120px) rotateY(180deg) }
  100% { -webkit-transform: perspective(120px) rotateY(180deg)  rotateX(180deg) }
}

@keyframes sk-rotateplane {
  0% {
    transform: perspective(120px) rotateX(0deg) rotateY(0deg);
    -webkit-transform: perspective(120px) rotateX(0deg) rotateY(0deg)
  } 50% {
    transform: perspective(120px) rotateX(-180.1deg) rotateY(0deg);
    -webkit-transform: perspective(120px) rotateX(-180.1deg) rotateY(0deg)
  } 100% {
    transform: perspective(120px) rotateX(-180deg) rotateY(-179.9deg);
    -webkit-transform: perspective(120px) rotateX(-180deg) rotateY(-179.9deg);
```

```
  }
}
```

Pour gérer l'affichage ou pas du widget, on va créer dans le contexte de données global (wowsContext) une nouvelle propriété du type boolean nommée "**loading**", initialisée par défaut à false. Cette propriété servira à déterminer la classe à appliquer au <DIV> qui contiendra notre widget (**spinner** si est en train de charger(**true**), **no_spinner** si ne fait rien(**false**))...

```
wowsContext = {
        …
        loading: false,
};
```

...on modifie le code de **selectNewStock** pour la passer à true et indiquer le début de chargement)...

```
wowsContext.selectNewStock = function(evt, id){
        if (id != undefined){
                //indique le lancemenet du chargement
                this.loading = true;
        url = "http://finance.services.appex.bing.com/Market.svc/M-TodayEquityV4?rtSymbols="
        +id
```

```
        +"&chartSymbols="+id
        +"&chartType="+this._time+"&lang=FR-FR&localizeFor=FR-FR";
          //chargement des données, je passe par un setTimeout pour laisser le temps
         // au widget d'apparaître, le webservice étant assez réactif
          setTimeout(function(){__load_async_json (url,"copy_stock_details", error_loading, timeout_error )},1000);
       }
}
```

...et la méthode **copy_stock_details** où on la remettra à false pour indiquer la fin du chargement:

```
wowsContext.copy_stock_details = function(data){
       obj = data.Rtd[0];
       this.stock.copyFromJSON(obj, data.Charts[0].Series);

       //fin de chargement
       this.loading = false;
}
```

Pour récupérer le nom de la classe à appliquer en fonction de “loading”, on crée une méthode de conversion **is_loading**:

```
function is_loading(value){
        return value ? "spinner" : "no_spinner";
}
```

Il ne reste plus qu’à créer mon widget dans la page HTML et appliquer le binding:

```
…
<div id="stock">
                <div data-binded-class="{binding from:loading converter:is_loading fallback:spinner}"></div>
                <div>{binding from:stock presenter:stock_presenter}</div>
                <!-- menu pour les jours -->
                 …
```