

Sommaire :

1.	Introduction	2
2.	Fonctionnement.	3
3.	Environnement technique.....	5
4.	Description technique des fonctions et classes (ou comment expliquer pourquoi 1+1 = 3)	6
4.1.	Couche javaScript :	6
4.1.1.	jsCore.js :	6
4.1.2.	jsAjaxCore.js :	7
a.	createXHR :	7
b.	SendPage :	7
c.	getReponse :	7
d.	setReponse :	7
e.	addHeader :	8
f.	generateUrlParams :	8
4.1.3.	jsPersonnages.js :	8
a.	getPersonnage(personnageID) :	8
b.	showPersonnage(json_infosPersonnage,json_listeComics) :	8
c.	getListPersonnages(params) :	9
5.	Améliorations et optimisations	10

1. Introduction

Cette application est un simple exercice visant à interroger l'API (Application Programming Interface) Marvel, la franchise ayant publiées d'innombrables comics produit des films et autres films d'animations (dessins animés).

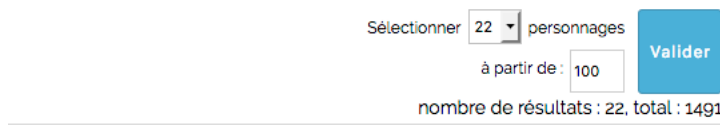
2. Fonctionnement.

L'application comporte une page d'accueil et d'une page où figure la liste des personnages de comics.



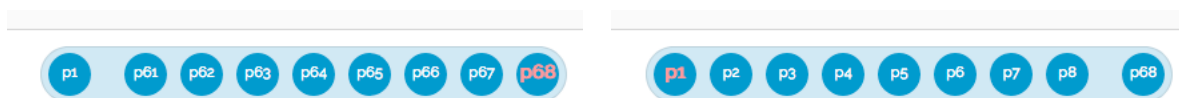
Par défaut, la liste affiche 22 personnages à partir du centième. En revanche, il est possible de changer ces paramètres pour afficher ce que l'on souhaite.

Pour cela, on saisit le nombre de personnage que l'on souhaite afficher, et à partir de combien de personnages et de cliquer sur valider.



Le nombre de résultat et le nombre total de personnage est également affiché.

Il est possible de naviguer dans les pages de résultat via les numéros situés en bas de page :



La règle de navigation affiche les 8 pages autour de la page courante (en orange) et « glisse » en fonction de la page courante affichée. Un bouton d'accès rapide à la dernière page et à la première page sont disponibles, en fonction de la position de la règle de pagination.

Il est possible de filtrer les personnages :

- En fonction de leur description : Seulement ceux qui ont une description
- En fonction de si ils ont leur propre comic: Seulement si ils ont leur propre comic.



Liste des personnages

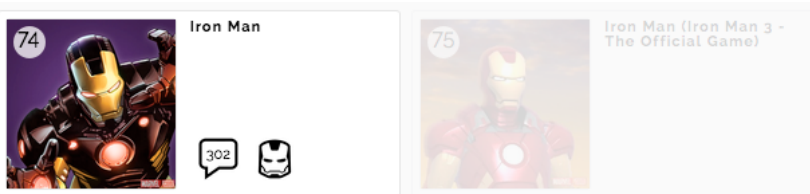
Filtrer les personnage

Seulement ceux qui ont au moins un comic

Seulement ceux qui ont une description

Le filtrage est exclusif, c'est à dire que c'est soit l'un, soit l'autre.

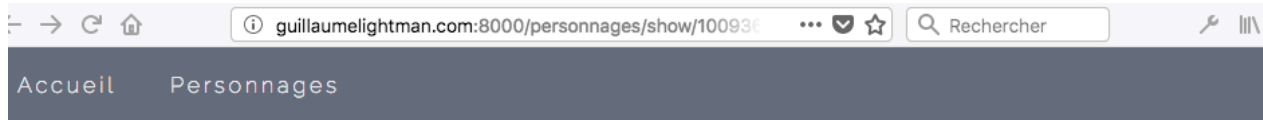
La vignette descriptive du personnage comporte sa photo, son nom et des icones indiquant si il a une description  et / ou a son comic . La bulle de la description indique la longueur (en caractères) de la description.



Lorsque l'on clique sur un personnage, sa fiche descriptive apparaît.

Sa fiche comporte :

- Sa photo
- So nom
- Sa description (si présente, sinon, il est affiché « aucune description n'a été fournie pour ce personnage »).
- Le nombre de comics dans lesquels apparaît ce personnage
- Ses trois premiers comics

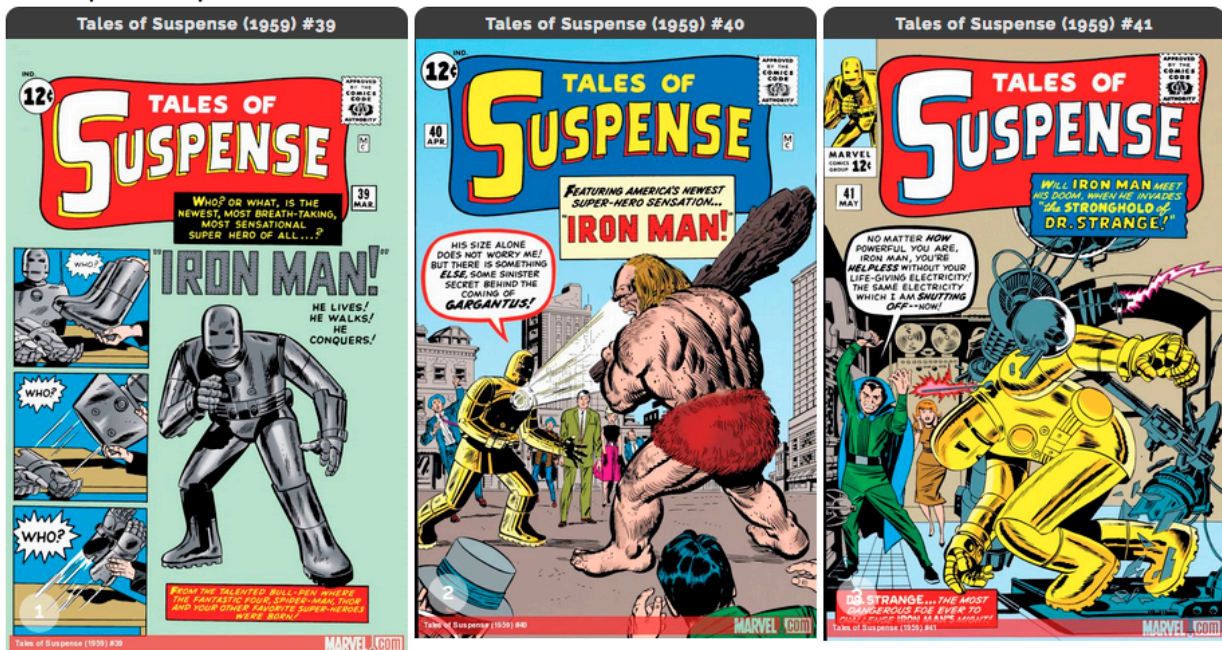


Iron Man

Wounded, captured and forced to build a weapon by his enemies, billionaire industrialist Tony Stark instead created an advanced suit of armor to save his life and escape captivity. Now with a new outlook on life, Tony uses his money and intelligence to make the world a safer, better place as Iron Man.

Ce personnage a dans **2394** comics de Marvel.

Ses trois premières parutions :



On notera, tout de même, qu'à l'époque, une BD coûtait 12 cents(\$)... :oD

3. Environnement technique.

L'application est hébergée sur un serveur Linux Debian 9.

Elle s'appuie sur le framework Symfony 4. En ce qui concerne le côté client, l'application est développée en Vanilla JS, HTML5 et CSS3.

Les sources sont disponibles sur GitHub à l'adresse <https://github.com/lightman69fr/marvel> et le suivi de projet s'effectue via un tableau Trello.

L'application répond aux exigences du pattern MVC.

La connexion et la récupération des données s'effectue par le biais de l'API Marvel, en exécutant des requêtes dites « Server Side », paramétrées.

Les données reçues sont au format JSON, directement exploitables par JavaScript ou tout autre langage pouvant lire du JSON.

Les interactions entre le navigateur (JavaScript) et le serveur (php Symfony) s'opère grâce à des requêtes asynchrones Ajax.

Le développement de la couche JavaScript s'appuie sur le principe du JavaScript Orienté Objet (JSOO).

Une fonction mère (classe) est créée dans un seul fichier. De fait, chaque classe possède son propre fichier. Par convention, le fichier de classe est nommé ainsi : « jsNomDeLaClasse.js ».

L'utilisation de cette méthode permet une excellente modularité. Ainsi, chaque classe peut être considérée comme un module, s'appuyant sur un « Core » pour les fonctions primaires.

Le Core correspond au fichier « jsCore.js ». Ce fichier compote l'ensemble des fonctions nécessaires au fonctionnement des modules. Il doit donc être inclus en premier, lors de l'insertion dans la page HTML.

4. Description technique des fonctions et classes (ou comment expliquer pourquoi $1+1 = 3$)

4.1. Couche javascript :

4.1.1. jsCore.js :

Ce fichier contient l'ensemble des fonctions nécessaires au bon fonctionnement de la couche client de l'application. Il s'agit du cœur de l'application. On y retrouve, entre autre, des fonctions « raccourcis » (pour en écrire un peu moins), des fonctions de cross-browsing (pour assurer au maximum la compatibilité entre les navigateurs) et des fonctions communes pouvant être utilisées de façon indépendantes.

Si l'on doit utiliser un système d'initialisation, ce sera ici qu'il faudra le déclarer.

Note : la version optimisée du fichier est : jsCore.min.js

4.1.2. jsAjaxCore.js :

C'est la classe Ajax. Elle a en charge l'envoi de contenu Asynchrone vers un serveur Web (Apache, serveur Symfony...).

Elle doit être instanciée de la manière suivante :

```
var instanceAjax = new AjaxHome(params);
```

Le constructeur nécessite un paramètre au format objet :

```
var params =  
{  
  methode : 'string' // GET - POST  
  page : 'string' // page à appeler pour la requête  
  data : {nom : valeur} // liste des paramètres au format objet  
  headers : [{nomHeader : valeurHeader}] // tableau de headers supplémentaires  
  useCallback : function (){} // fonction à appeler lorsque la requête a abouti (statut 200)  
  withLoader : function (){} // fonction à appeler lorsque l'on souhaite insérer un loader  
};
```

Les paramètres obligatoires sont :

- methode
- page

L'absence d'au moins un de ces deux paramètres affichera un message indiquant qu'il faut renseigner ces paramètres

Si le paramètre `useCallback` n'est pas précisé, la requête sera exécutée, la réponse sera reçue mais ne pourra pas être récupérée.

L'utilisation de ce paramètre est donc le seul moyen d'exploiter la réponse reçue de la requête.

La déclaration de `withLoader` avec une fonction permet d'activer automatiquement l'utilisation d'un loader, le temps que les données soient récupérées.

Descriptif des méthodes :

a. *createXHR* :

Permet de créer une requête Ajax.

- Arguments : aucun.
- Valeurs de retour : objet XMLHttpRequest

b. *sendPage* :

Méthode d'envoi de la requête.

- Arguments : Aucun.
- Paramètres requis :
 - Page
 - methode
- Valeurs de retour :
 - la réponse issue de la requête, accessible depuis le callback.

c. *getResponse* :

getter permettant de récupérer la réponse dans la classe.

d. *setResponse* :

setter permettant d'enregistrer la réponse dans la classe.

- Arguments : La réponse.

e. *addHeader* :

Méthode permettant d'ajouter des headers supplémentaires à la requête.

- Arguments : le header au format objet : {nomHeader : valeurHeader}
- Valeur de retour : aucun.

f. *generateUrlParams* :

Méthode permettant de transformer l'objet contenant les paramètres à transmettre (par la magie du code) en chaîne de paramètres texte valide pour la fonction send de XMLHttpRequest.

- Arguments : objet, issu du paramètre « data ».
- Valeur de retour : chaîne de caractère de type :
nomparam1=param1&nomparam_n=param_n

4.1.3. jsPersonnages.js :

Classe permettant de récupérer et afficher les données des personnages.

Elle doit être instanciée de la manière suivante :

```
var instancePersonnage = new Personnages(params);
```

Les paramètres sont optionnels. Si ceux-ci ne sont pas précisés, la classe fonctionnera en mode « liste » et n'affichera que la liste des personnages.

Pour paramétrer la classe, il faut transmettre au constructeur les paramètres suivants :

```
var params =
{
  mode          : 'liste',
  paramsListe   : objet Liste des personnages.
}

var params=
{
  mode          : 'showPers'
  persIdD       : 'string' // ID du personnage
}
```

Descriptif des méthodes :

a. *getPersonnage(personnageID)* :

Méthode de récupération des informations sur le personnage auprès de l'API Marvel.

- Paramètres : ID du personnage
- Valeurs de retour : objet contenant les informations du personnage (nom/description/image/comics...)

b. *showPersonnage(json_infosPersonnage,json_listeComics)* :

Méthode permettant d'afficher le personnage et ses informations.

- Paramètres :
 - json_infosPersonnage

```
{
  data
  {
    results
    {
      id          : id du personnage,
      name        : nom du personnage,
      description : description du personnage,
      thumbnail
      {
        path      : chemin de l'image,
        extension : extension du fichier
      }
    }
  }
}
```



```

    }
}

• json_listeComics
{
    data
    {
        total : nombre total de comics
        results
        [
            comicDatas
            {
                id : id du comic,
                title : titre du comic,
                description : description du comic,
                images[0] : image du comic,
                dates : dates du comic
            }
        ]
    }
}

```

- Valeurs de retour : données au format JSON.stringify.

c. *getListPersonnages(params)* :

Cette méthode permet de récupérer les données de la liste des personnages.

- Paramètres :
 - Cette méthode peut être paramétrée de plusieurs façons :
 - via le formulaire de sélection (à partir de / nombre de)
 - `var selectLimit = parseInt(getID('selectLimit').value);`
 - `var pageCourante = parseInt(getID('pageCourante').value);`
 - via le paramètre « params »


```

params
{
    selectFrom : 100,
    nbByPage   : 22
}
                            
```
 - sans aucun paramètre, avec les valeurs par défaut déclarées.

5. Améliorations et optimisations

Comme on a pu le voir, l'application est dépendante des temps de réponse de l'API Marvel qui peut s'avérer, parfois, lente.

L'ajout d'un loader pour faire patienter l'utilisateur ne remplace pas une expertise en ce qui concerne la performance globale de l'application.

La première idée, concernant l'augmentation de performances, consiste à rapatrier sur le serveur où est hébergé l'application l'ensemble des données nécessaires à l'application. Cela nécessite d'avoir conçu son propre modèle de données.

Il existe néanmoins quelques contraintes :

- Il faut développer le modèle de données
- Il est nécessaire de mettre en place un script de synchronisation afin d'assurer une cohérence entre les données locales et les données distantes, qui se chargera de récupérer, à intervalles réguliers, les données qui sont nécessaires à l'application.

Cette stratégie est utilisée dans de gros systèmes de données, où pour éviter de saturer le serveur primaire, des vues sont mises à disposition, à intervalles réguliers.

Les applications clientes utilisent ces vues pour travailler.

Cependant, cette technique comporte plusieurs avantages :

- Compte tenu que les données sont en local, l'accès n'est plus dépendant de la connexion internet, ni de l'état du serveur distant.
- La vitesse de récupération des données, surtout si elles sont volumineuses, ne sera plus dépendante de la vitesse de la connexion, ni de l'état du serveur distant.
- Les données étant organisées selon notre propre modèle de données, les requêtes SQL nécessaires au rapatriement seront bien plus optimisées et seront en adéquation avec les besoins de l'application. Il y a aussi un impact évident sur le volume des données récupérées : Comme on récupère seulement ce dont on a besoin, aucune donnée inutile n'est rapatriée. On économise en volume de données, mais aussi en développement, car le tri s'opère au niveau SQL, et non plus au niveau du code.
- Compte tenu que les données seront directement accessibles via une base de données (MySQL, Oracle...), ces dernières seront directement accessibles depuis la couche serveur de l'application. Cela fera économiser du code et une couche applicative pour la récupération des données.

Compte tenu de la nature de l'application, on peut également se poser la question de la pertinence d'utiliser Symfony, quand une simple page web d'appel aurait suffi.

En effet, compte tenu que dans l'état actuel de l'application, aucune donnée n'est enregistrée sur une base de données. De plus, la récupération des données s'effectue via une requête XMLHttpRequest et les données reçues sont au format json.

Donc on peut totalement s'affranchir du PHP et développer l'application totalement au niveau client.

On pourra alors développer l'application soit en Vanilla JS, soit en utilisant un framework du type

Angular. En procédant ainsi, on pourra s'affranchir de couches applicatives de conversions de données d'un format à un autre