

Knockout.jsの マイ運用方針



はじめに

Knockout.jsは難しい。

チュートリアルを少し読みかじったばかりの知識を片手に、Knockout.jsをプロジェクトに導入すると痛い目を見てしまう。(ちなみに、痛い目を見たのは私です。)

Knockout.jsを各プロジェクトで使ってきた経験をもとに、これがベストだと言える、運用方針を考えてみた。

この運用方針を取り入れることで、小規模から中規模、大規模までカバーできるよう、Knockout.jsを運用できるのではないかと思う。

前提

- エンタープライズWebアプリを作成することを前提として考える。
- サーバーサイドレンダリングは、必要最低限の画面情報のみとする。
- サーバーサイド、クライアントサイドのデータはjson形式とする。
- データの取得・検索・保存などは、全てAjaxを利用することにする。
- マスタデータによる動的な画面情報の構築(selectタグなど)が必要な場合も、Ajax・jsonデータを利用することにする。

※上記は、フルAjaxを適用する場合の前提となるが、機能の一部でAjaxを使う場合においても、本方針を適用する。
ただし、Knockout.jsから受けられるメリットが半減することに注意すること。

双方向バインディングを必ず使う

Knockout.jsでは、片方向バインディングと双方向バインディングの何れかを選択することができる。

結論から言うと、双方向バインディング (observable) を必ず利用することにする。

何故、双方向バインディングか？

双方向では、ViewModelを更新するとUIが更新され、UIを更新するとViewModelが更新される。まとめると、双方向にデータが同期されるため、Knockout.jsを利用するメリットを最大限に享受できる。

Mappingプラグインを必ず使う

ViewModelの各プロパティに双方向バインディング (observable) を適用するのは非常に面倒。よって、Mappingプラグインを必ず使うこと。

<http://kojs.sukobuto.com/docs/plugins-mapping>

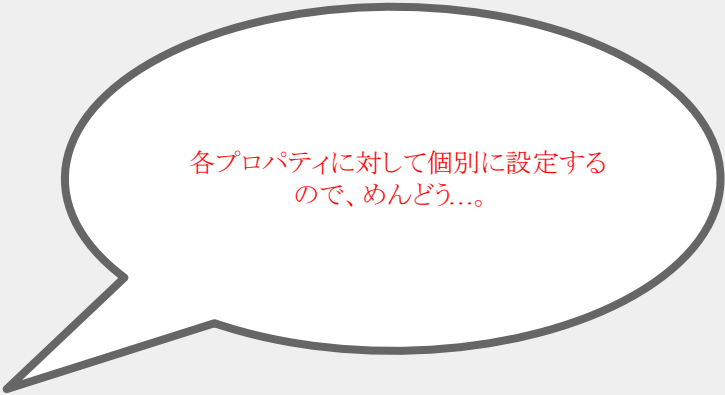
Mappingプラグインを必ず使う

•通常

// 以下は、Ajaxで取得するという想定

```
var myViewModel = {  
  personName: 'ボブ',  
  personAge: 123  
};
```

```
var myViewModelObs = {  
  personName: ko.observable(myViewModel.personName),  
  personAge: ko.observable(myViewModel.personAge)  
};
```



各プロパティに対して個別に設定する
ので、めんどろ...

Mappingプラグインを必ず使う

- Mappingプラグイン

// 以下は、Ajaxで取得するという想定

```
var myViewModel = {  
  personName: 'ボブ',  
  personAge: 123  
};
```

// 以下のユーティリティ関数にjsonデータを入れ込むことで、observableなデータオブジェクトに変換することができる

```
var myViewModelObs = ko.mapping.fromJS(myViewModel);
```



一行で変換可能。便利！

Knockout.jsにおけるイベントは使用禁止

DOMに対してのイベントハンドリングは、jQueryの `on / off` を使用する。jQueryを使用することで、特定のグループ(一つまたは一つ以上)に対するイベント関数を登録できるが、Knockout.jsでは、単体のDOMに対して個別にイベント関数のバインドを適用しないといけないため、不便に感じることもある。

よって、イベントハンドリングは、jQueryで一元化する。

Knockout.jsのViewModelのライフサイクル

本運用を適用した場合の、ViewModelのライフサイクル

1. AjaxでデータロードしてViewModelに変換(マッピングプラグイン)
2. ↓↓ ここからViewModelが生まれる
3. ユーザーの入力に伴い、UIに与えた更新がViewModelに同期される
4. 何らかのイベントが発生して、javascript処理が実行される。その際に ViewModelなどが編集されることで、UIも同時に更新される
5. サーバーに HTTP送信する際に、FormデータをserializeしてUIデータを抽出する
`$("#form").serialize()`
6. ↑↑ ここでViewModelを破棄。(後続処理でページ遷移などが発生する)

Knockout.jsのViewModelのライフサイクル

(詳細) 本運用を適用した場合の、ViewModelのライフサイクル

1. AjaxでデータロードしてViewModelに変換(マッピングプラグイン)
2. ↓↓ ここからViewModelが生まれる

```
// 以下は、Ajaxで取得するという想定  
var myViewModel = { ... };
```

```
// 以下のユーティリティ関数にjsonデータを入れ込むことで、observableなデータオブジェクトに変換することができる  
var myViewModelObs = ko.mapping.fromJS(myViewModel);
```

Knockout.jsのViewModelのライフサイクル

(詳細) 本運用を適用した場合の、ViewModelのライフサイクル

3. ユーザーの入力に伴い、UIに与えた更新がViewModelに同期される

テキストボックス

Text Box|

```
<input type="text" name="" value="" data-bind="value: text" />
```



```
// テキストボックスの内容がViewModelのtextに反映されていることを確認してみる  
console.log(myViewModelObs.text());
```

Knockout.jsのViewModelのライフサイクル

(詳細) 本運用を適用した場合の、ViewModelのライフサイクル

4. 何らかのイベントが発生して、javascript処理が実行される。その際に ViewModelなどが編集されることで、UIも同時に更新される

```
// フォーカスアウト時にイベントをハンドリング
$("input[name=text]").on("blur", function() {
    // テキストボックスの内容をViewModel経由で更新する
    myViewModelObs.value("Text Box Change!");
});
```



テキストボックス

Text Box Change!

```
<input type="text" name="text" value="" data-bind="value: text" />
```

Knockout.jsのViewModelのライフサイクル

(詳細) 本運用を適用した場合の、ViewModelのライフサイクル

5. サーバーに HTTP送信する際に、FormデータをserializeしてUIデータを抽出する
6. ↑↑ ここでViewModelを破棄。(後続処理でページ遷移などが発生する)

```
// フォームの入力内容を取得  
var formData = $("#form").serialize();
```