Aplicações Single-page: Caso de implementação com Backbone.js

Autor: Bruno Vitorino - 1402214

Introdução

Este artigo pretende fornecer uma pequena introdução ao conceito de aplicações Single-Page e um exemplo de implementação utilizando a framework Backbone.js

Aplicações Single-Page

O conceito de aplicação Single-Page é relativamente recente quando comparado com a história da computação e da Internet. Existem referências ao conceito que remontam ao ano de 2003, mas apenas nos anos mais recente, com a introdução do HTML5, é que o conceito se tornou bastante mais popular.

As aplicações Single-Page (SPA) são aplicações Web que carregam o seu conteúdo num único ficheiro HTML, fazendo posteriormente atualizações parciais à página de acordo com as interações do utilizador com a mesma. As SPAs utilizam a técnica AJAX e o HTML5 para criar uma experiência mais próxima do que o utilizador está habituado num aplicação não estilo Web (ou nativa), ou seja, sem constantes recarregamento de página em cada interação do utilizador. Este tipo de aplicação é particularmente difícil de implementar utilizando um tipo de abordagem tradicional de desenvolvimento Web, pois normalmente uma aplicação que forneça uma experiência avançada ao utilizador é constituída por vários estados intermédios (clique no botão X, Item do Menu X aberto, etc). A implementação de todos estes estados intermédios tornam o desenvolvimento da aplicação bastante mais complexa quando é escolhida uma abordagem de renderização de página no servidor, pois todos estes estados teriam de ser mapeados em URLs.

Ciclo de vida

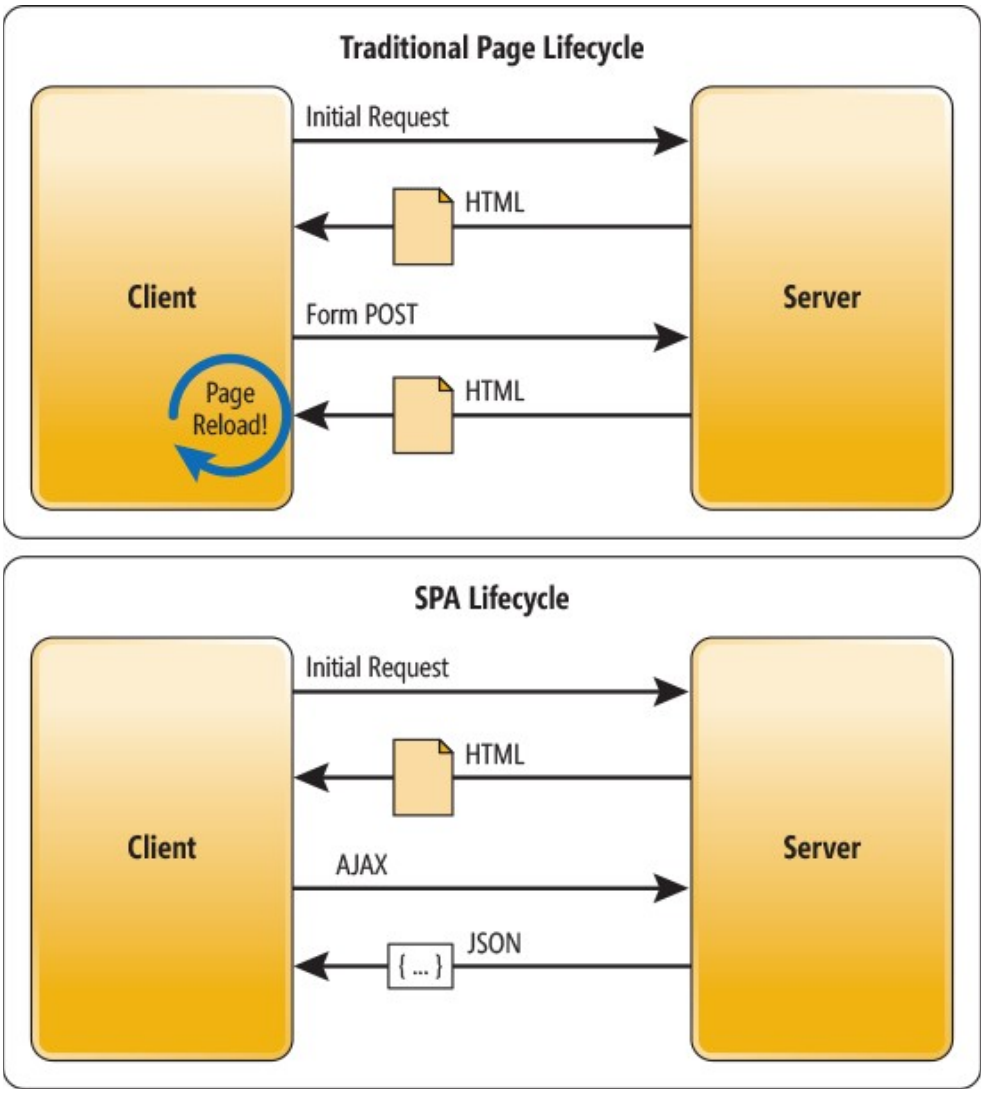
Numa aplicação Web tradicional, cada vez que a aplicação efetua um pedido ao servidor, o servidor renderiza uma nova página HTML, e isto por sua vez despoleta um refrescamento de página no browser.

No caso de uma aplicação Single-Page, depois do primeiro carregamento da página, todas as restantes interações ocorrem através de pedidos AJAX. Estes pedidos respondem com

informação, geralmente em formato JSON. Em seguida, esta informação é utilizada para atualizar a página dinamicamente sem efetuar o refrescamento da página.

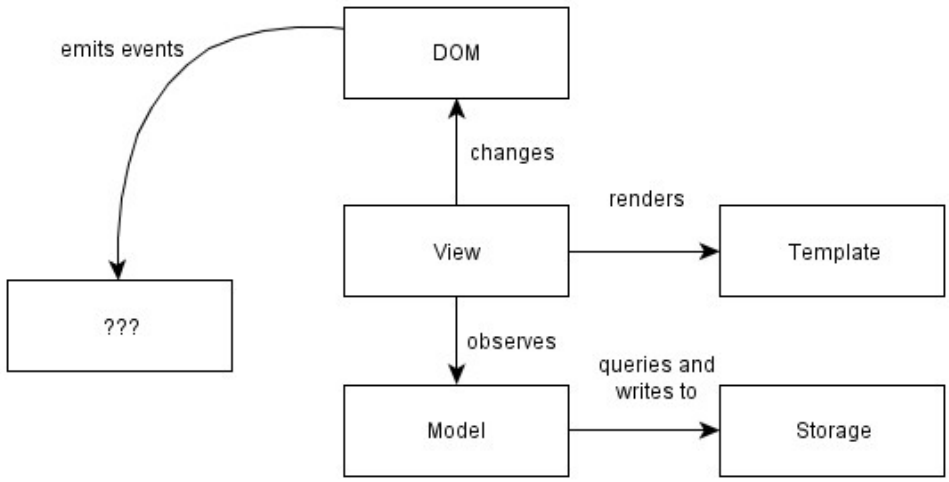
Um dos benefícios das SPA é que torna por regra as aplicações mais fluidas e *responsivas*, evitando assim o processo por vezes maçador de recarregamento demorado de uma página completa. Outro benefício, talvez menos evidente, é o da completa separação da camada de apresentação (HTML markup) da camada da lógica aplicacional (pedidos AJAX e respostas em formato JSON).

Numa aplicação SPA pura, todas as interações com a IU ocorrem do lado do cliente através de Javascript e CSS. Depois do carregamento inicial, o servidor funciona puramente como camada de serviço. Assim, o cliente necessita apenas de conhecer quais pedidos HTTP que necessita efetuar. Tornando assim o cliente completamente agnóstico em relação à implementação efetuada no servidor.



Arquitetura da uma SPA

Uma Aplicação Single-Page terá em geral a seguinte estrutura:



Esta arquitetura acenta fundamentalmente nos seguintes conceitos:

Elementos DOM são apenas escritos Nenhuma informação deve ser lida dos elementos DOM. Estes devem apenas servir como fonte de interação com o utilizador. A informação referente ao estado da aplicação não deve ser guardada no DOM, pois torna-se difícil de gerir a partir do momento em que é necessário mostrar o mesmo bloco de informação em vários locais da aplicação.

Modelos como a única fonte da verdade Em lugar de guardar informação nos elementos DOM ou em objetos aleatórios não estruturados, esta deve ser guardada num grupo de modelos existentes em memória, estruturados de forma a representar o estado e informação da aplicação.

As Views observam as modificações que ocorrem nos Models Em vez de controlar manualmente as modificações que ocorrem nos modelos, deve existir um mecanismo de emissão de eventos aquando a ocorrência de modificações. Desta forma, quando várias views dependem do mesmo modelo, cada view é informada das alterações no modelo e as mesmas sabem como se regenerar para refletir as alterações.

*Módulos dissociados que expõem interfaces de comunicação** A ideia é criar pequenos subsistemas que não são interdependentes, em vez de usar objetos globais.

Minimizar código que depende do DOM Todo o código que depende do DOM necessita de ser testado para assegurar compatibilidade cross-browser. As incompatibilidades cross-browser são um realidade e são muito difíceis de evitar. Se escrevemos código que minimiza a dependência do DOM, isolamos e minimizamos a quantidade de código que deve ser testado tendo esta situação em conta.

Backbone.js

Introdução

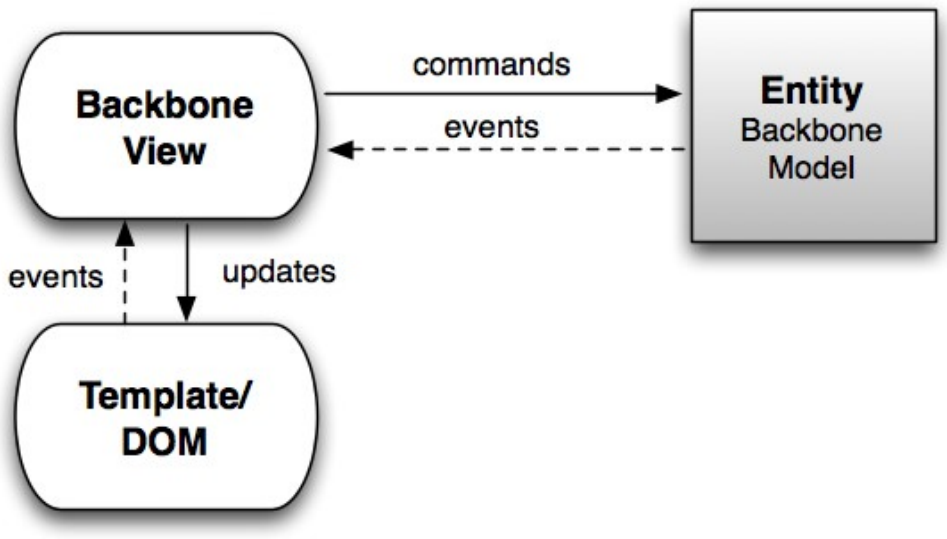
O Backbone.js é uma biblioteca Javascript baseada no paradigma MVP (Model-View-Presenter). E foi fundamentalmente concebida para o desenvolvimento de Aplicações Single-Page.

O paradigma MVP é utilizado maioritariamente na construção de interfaces com o utilizador, e define 3 camadas:

- **Model** Define a estrutura de um objeto;
- **View** É responsável por definir como os objetos (modelos) são apresentados e por dirigir as interações com os utilizadores até ao Presenter, para que ele lide com os objetos;
- **Presenter** Faz a ponte entre os objetos (modelos) e as Views. Obtém informação de repositórios (web service, etc) e formata essa informação para ser renderizada na view.

Embora o Backbone.js seja baseado no padrão MVP, a nomenclatura utilizada para a definição dos seus conceitos fundamentais têm apenas em parte relação direta com a nomenclatura do padrão.

A figura seguinte apresenta a arquitectura de uma aplicação *Backbone.js"*:



Os próximos pontos apresentam os elementos basilares do Backbone.js.

Backbone.Model

Os modelos guardam os dados da aplicação e são responsáveis pela sincronização dos mesmos com serviços REST. Um modelo pode definir por defeito os seus atributos e emite eventos existem alterações em algum dos seus atributos.

Backbone.Collection

As coleções gerem um grupo de modelos e são responsáveis pela sincronização dos mesmos com serviços REST. Uma coleção fornece funcionalidades básica de pesquisa nos modelos associados a si, e emite eventos quando qualquer dos seus modelos é adicionado, removido, editado ou ordenado.

Backbone.View

As Views ligam os modelos às suas representações num documento HTML (DOM). Elas renderizam a informação do modelo para a estrutura DOM, e também capturam a interação do utilizador com o DOM, redirecionando essas interações para o modelo.

Backbone.Route

Fornece métodos para criar rotas para as "páginas" no client-side da aplicação, e permite ligá-las a ações e eventos. A biblioteca faz uso da nova API do HTML5: History. De uma forma simplificada, podemos descrever um Router como o componente que permite a definição de URLs dentro da página para permitir a navegação utilizando as funcionalidades de retroceder e avançar do browser. Sendo possível assim colmatar uma lacuna existente.

Caso prático

Visão global

O objetivo desta secção é fornecer um exemplo prático de uma aplicação Single-Page que comunica com uma interface REST para alimentar a aplicação com dados obtidos de uma base de dados.

O exemplo é um site de anúncios online, em que o utilizador pode publicar e consultar anúncios publicamente. É de salientar que a aplicação não possui sistema de autenticação de utilizadores.

No sistema de ficheiro vamos proceder à organização do nosso código da seguinte forma:

- anúncios
  - api
  - web
    - css
    - images
    - js
      - collections
      - lib

- models
- views

Setup inicial

Antes de tudo, teremos de fazer o download da biblioteca *Backbone.js*, esta pode ser obtida no seguinte link:

http://backbonejs.org/backbone.js
A biblioteca *Backbone.js* tem como dependências as bibliotecas *jQuery* e *underscore.js*.
http://code.jquery.com/jquery-2.1.4.min.js
http://underscorejs.org/underscore-min.js
Depois do download destas duas bibliotecas, as mesmas devem ser colocadas na diretoria `web/js/lib` .

> *Nota:* O download destas bibliotecas pode ser feito diretamente do site oficial das mesmas, ou podemos utilizar um gestor de dependências como o Bower para obter estas bibliotecas e mantê-las atualizadas.

Para estilizar a nossa aplicação vamos usar uma pequena framework css *skeleton*.
https://github.com/dhg/Skeleton/releases/download/2.0.4/Skeleton-2.0.4.zip

Funcionalidades

A nossa aplicação terá duas funcionalidades fundamentais:

1. Adicionar novos anúncios;
2. Consultar a lista de anúncios;

Models

Para este exemplo apenas necessitamos de definir um modelo. Este modelo corresponde ao anúncio que vai ser publicado e consultado pelos utilizadores da aplicação.

```
// web/js/models/Ad.js
var app = app || {};

app.Ad = Backbone.Model.extend({
    defaults: {
        title: 'anonymous',
        body: 'No content',
        author: 'anonymous@nowhere.net',
        publicationTime: new Date()
```

```
    }
});
```

- **Linha 2** utilizamos uma técnica conhecida do javascript para definir um *namespace* para a aplicação. A partir da **linha 4** procedemos à definição do modelo, utilizando a função `Backbone.Model.extend()` para implementar o nosso modelo e manter as funcionalidades fornecidas pela biblioteca *Backbone.js*. Com a propriedade `defaults` (**linha 5**) definimos as propriedades do objeto e os seus valores por defeito. O modelo vai ter os seguintes campos:
  - title: O titulo do anúncio
  - body: O conteúdo do anúncio
  - author: O autor do anúncio
  - publicationTime: A data e hora de publicação do anúncio.

Collections

Como referido anteriormente, uma coleção é responsável por gerir um grupo de modelos e por fazer a sincronização destes com a interface REST.

```
// web/js/collections/AdList.js
var app = app || {};

app.AdList = Backbone.Collection.extend({
    model: app.Ad,

    url: "/api/ads"
});
```

Na **linha 2** repetimos o que foi feito para o modelo e fazemos a inicialização do namespace no caso deste ainda não existir. Nas linhas seguintes *extendemos* a coleção fornecida pelo *Backbone.js* e inicializamos o nosso objeto com o modelo pelo qual a coleção vai ser responsável e também o URL em que a interface REST vai estar disponível.

Views

As views são provavelmente um dos elementos mais importantes da nossa aplicação. São elas que são responsáveis pela geração dos elementos DOM associados por exemplo a um modelo, pelo processamento dos eventos desencadeados pelos elementos DOM e pela ligação com as coleções e modelos da nossa aplicação. No nosso caso concreto, vamos necessitar de duas views: uma para o modelo definido anteriormente `Ad` e outra para a coleção de modelos `AdList` . De uma forma geral, sempre que temos uma coleção de modelos, é comum definir uma view para o modelo e outra para a coleção, deste forma, cada uma das views sabe como se renderizar a si mesma, e a coleção delega a

responsabilidade de renderização dos modelos pelos quais é responsável aos próprios modelos.

Ad

```
// web/js/views/Ad.js
var app = app || {};

app.AdView = Backbone.View.extend({
    tagName: 'div',
    className: 'book',

    template: _.template($('#adTemplate').html()),

    render: function () { this.
        $el.html(this.template(this.model.toJSON()));

        return this;
    }
});
```

Como podemos observar neste bloco de código, a definição de uma view não é muito diferente do que vimos anteriormente para os modelos e coleções, no entanto já temos algumas particularidades. Iremos por partes:

- **Linha 4:** Uso da função `Backbone.View.extend()` para definição de uma view.
- **Linha 5:** À propriedade `tagName` deve ser atribuída um nome de uma tag HTML/XHTML, e indica que aquando da renderização da view, os elementos DOM gerados serão incluídos como elementos filho do elemento atribuído. Neste caso concreto, eles vão ser gerados dentro de um elemento `div` .
- **Linha 6:** À propriedade `className` é atribuída a class css que vai ser aplicada ao elemento definido na propriedade `tagName` . Esta propriedade é opcional.

- **Linha 8:** A propriedade `template` é opcional mas útil, pois iremos utilizar o template engine fornecido pela biblioteca *underscore.js* para pré-compilar templates em HTML para formato Javascript e depois renderizar elementos HTML baseados em informação que será injetada quando necessário.

  Para mais informação, consultar http://underscorejs.org/#template. `_.template()` invoca a função de compilação de HTML para javascript.

  `$('#adTemplate').html()` utiliza a biblioteca jQuery para obter o elemento DOM utiliza um seletor para obter o elemento HTML que contem o nosso template (ver mais na secção seguinte - Templates).

- **Linha 10:** Definição da função `render` . Esta função é, como o nome indica, a função responsável pela renderização da view, e deve fundamentalmente definir o conteúdo do

elemento definido pelo `tagName`. Para isso devemos atribuir este conteúdo à propriedade `this.$el`.

O `this.$el` é uma propriedade especial das views que corresponde ao elemento HTML da view.

- **Linha 11:** Atribuição do resultado da renderização do template com os dados do modelo. Para renderizar o template é utilizada a propriedade anteriormente definida. Para obter os dados do modelo utilizamos `this.model.toJSON()`. É importante referir que a propriedade `template` é de facto uma função que corresponde à conversão de HTML para um objeto Javascript como referido anteriormente.
- **Linha 13:** É fundamental retornar o próprio objeto (`this`) no final desta função.

AdList

A definição de uma view para um coleção é praticamente igual à de um modelo, no entanto no nosso exemplo, existem algumas funcionalidades adicionais que devem ser implementadas e que a tornam um pouco mais complexa e extensa. Vamos então por partes:

Inicialização

```
// web/js/views/AdList.js
var app = app || {};

app.AdListView = Backbone.View.extend({
    el: '#adList',

    initialize: function () { this.collection
        = new app.AdList();
        this.collection.fetch({reset: true});
        this.render();
    }
});
```

- **Linha 4:** Tal como na view anterior, criamos a view para a coleção utilizando a função `Backbone.View.extend()`.
- **Linha 5:** Define o elemento ao qual vai ser atribuído esta view.
- **Linha 7:** Definimos a função `initialize`. Esta função é fundamental para inicializar o objeto.
- **Linha 8:** Criamos uma propriedade do objeto que contem a nossa coleção de modelos.
- **Linha 9:** Chama a função `fetch()` da coleção que desencadeia um pedido HTTP à nossa API REST que irá obter a lista de anúncios. O objeto passado como parâmetro à função (`{reset: true}`), indica que queremos substituir os elementos existentes na propriedade `collection` pelos objetos recebidos da API REST.
- **Linha 10:** Efetuamos a renderização da view.

Renderização

Para a renderização desta view temos de ter 2 coisas em consideração. A renderização da view em si - da coleção -, e a dos seus modelos. Portanto teremos de definir uma função que renderiza a view e que por sua vez executa outra função que renderiza um modelo.

```
// web/js/views/AdList.js
var app = app || {};

app.AdListView = Backbone.View.extend({

    (...)

    render: function () {
        this.collection.each(function (item) {
            this.renderAd(item);
        }, this);
    },

    renderAd: function (item) {
        var adView = new app.AdView({
            model: item
        }); this.

        $el.append(adView.render().el);
    }

});
```

- **Linha 8-12:** `render` é a função que renderiza a coleção. Esta itera sobre a sua `collection` e para cada elemento chama a respetiva função de renderização para o elemento - `renderAd` .
- **Linha 14-17:** `renderAd` é a função que inicializa uma nova view para o elemento da coleção passado por parâmetro.
- **Linha 19:** Junta ao parâmetro `$el` o resultado da renderização da nova view para o elemento usando o método `append` . **Atenção:** é essencial passar ao método `append` o valor de `adView.render().el` .

Eventos

A nossa aplicação tem duas funcionalidades associadas a esta view.

- Publicar um anúncio
- Pesquisar anúncios

Vamos começar por implementar a funcionalidade de publicação de anúncios.

Publicação de anúncios

Em primeiro lugar vamos adicionar a função que recolhe informação de uma formulário, constrói um objeto baseado no modelo `Ad`, e o adiciona à nossa coleção `AdList`.

```
// Event Handlers
addAd: function (e) {
    e.preventDefault();

    var formData = {};

    $("#addAd").children("input").each(function (i, el) {
        var inputValue = $(el).val();
        if ( inputValue != "") {
            formData[el.name] = inputValue;
        }
    });

    // Add default publication time
    formData['publicationTime'] = new Date();

    this.collection.create(formData);

    // This clears the form for next insertion
    this.clearFormData();
}

clearFormData: function () {
    $("#addAd").children("input").each(function(i, el) {
        $(el).val('');
    });
}
```

- **Linha 4:** Inicializamos um objeto que vai conter a informação do nosso formulário.
- **Linha 5-11:** Iteramos sobre todos os elementos do tipo `input` e obtemos o seu valor. Há que notar que o elemento `addAd` vai ser definido mais à frente quando abordamos os templates.
- **Linha 14:** Definimos o valor da data e hora de publicação do anúncio.
- **Linha 16:** Adicionamos o novo objeto à coleção invocando o método `create` fornecido pelo *Backbone*.
- **Linha 19:** Chamamos a função que efetua a limpeza dos campos formulário.
- **Linha 22-26:** Declaramos a função que efetua a limpeza dos campos do formulário.

A nossa interface com o utilizador vai ter o botão que nos vai permitir submeter o formulário com o conteúdo do novo anúncio. Para isso é necessário indicar na nossa view quais os objetos que emitem eventos e quais a função que vão lidar respetivamente com estes eventos. Além disso, visto que vamos alterar a nossa coleção de modelos, será necessário também lidar com estas alterações na nossa view, caso contrário haverá um desfasamento entre a informação na coleção e o que é de facto apresentado no ecrã.

```
// web/js/views/AdList.js
var app = app || {};

app.AdListView = Backbone.View.extend({
    (...)

    initialize: function (initialAds) {
        (...)

        // The events the object is listening to
        this.listenTo(this.collection, 'add', this.renderAd);
        this.listenTo(this.collection, 'reset', this.render);
    },

    events: {
        "click #btnAdd": "addAd",
    },

    (...)
});
```

- **Linha 11-12:** Indicamos à nossa view que deve estar à escuta dos eventos `add` e `reset` da nossa coleção `this.collection`. No caso de `add` devemos renderizar o novo objeto. No caso de `reset` toda a lista de anúncios deve ser novamente renderizada.
- **Linha 15-17:** A propriedade `events` é, de forma simplificada, uma lista de chave-valor, em que a chave indica o evento e o elemento que emite este evento, e o valor define qual a função que deve lidar com este evento.

Versão final

A versão final da nossa view deverá ser a seguinte:

```
// web/js/views/AdList.js
var app = app || {};

app.AdListView = Backbone.View.extend({
    el: '#adList',

    initialize: function (initialAds) {
        this.collection = new app.AdList();
        this.collection.fetch({reset: true});
        this.render();

        // The events the object is listening to
        this.listenTo(this.collection, 'add', this.renderAd);
        this.listenTo(this.collection, 'reset', this.render);
    },

    events: {
        "click #btnAdd": "addAd"
    },
```

```
    render: function () {
        this.collection.each(function (item) {
            this.renderAd(item);
        }, this);
    },

    renderAd: function (item) {
        var adView = new app.AdView({
            model: item
        });

        this.$el.append(adView.render().el);
    },

    // Event Handlers
    addAd: function (e) {
        e.preventDefault();

        var formData = {};

        $("#addAd").children("input").each(function (i, el) {
            var inputValue = $(el).val();
            if ( inputValue != "") {
                formData[el.name] = inputValue;
            }
        });

        // Add default publication time
        formData['publicationTime'] = new Date();

        //this.collection.add(new app.Ad(formData));
        this.collection.create(formData);

        // This clears form for next insertion
        this.clearFormData();
    },

    clearFormData: function () {
        $("#addAd").children("input").each(function(i, el) {
            $(el).val('');
        });
    }
});
```

Templates

Uma parte muito importante da nossa aplicação é a definição de templates. A biblioteca Backbone facilita-nos esta tarefa porque esta depende de outra biblioteca, o *underscore.js* que disponibiliza uma solução *micro-templating*. Um template, neste caso, corresponde a pequenos pedaços de código HTML definidos no nosso documento HTML principal, que são

lidos e populados com nova informação (renderização). Na nossa aplicação vamos apenas definir um template que serve para mostrar na nossa interface um anúncio.

web/index.html

```
<!-- Our underscore templates -->
<script type="text/template" id="adTemplate">
    <ul>
        <li><%= title %></li>
        <li><%= body %></li>
        <li><%= author %></li>
        <li><%= publicationTime %></li>
    </ul>
</script>
```

É de notar que um template pode ser definido de várias formas. O que é importante é que este tenha um `id` que possa servir de identificação única para um seletor jQuery, e que o browser não saiba como renderizar este elemento no ecrã. Neste caso, é definido no interior de um elemento `script` com tipo MIME `text/template` que não é reconhecido pelo browser, desta forma não haverá o perigo do conteúdo do template seja renderizado involuntariamente. Para mais informação sobre templates em *underscore.js*, nomeadamente sobre as tags `<%= %>` e `<% %>` consultar a documentação oficial em http://underscorejs.org/#template

Index.html e inicialização da aplicação

Nesta secção vamos, por assim dizer, colar as pontas de tudo o que fizemos anteriormente.

Vamos em primeiro lugar adicionar um ficheiro em que faremos a inicialização da nossa aplicação. A inicialização é muito simples, apenas temos de usar a função (neste caso o atalho) da função `onDocumentReady` do jQuery para inicializar a nossa view principal na **linha 5**: `AdListView`.

```
// web/js/app.js
var app = app || {};

$(function() {
    new app.AdListView();
});
```

Em seguida vamos efetivamente criar a página que é apresentada ao utilizador aquando do seu acesso à nossa aplicação.

web/index.html

```
<! DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Ads4every1</title>

    <link rel="stylesheet" href="css/normalize.css"/>
    <link rel="stylesheet" href="css/skeleton.css"/>
    <link rel="stylesheet" href="css/ads4every1.css"/>
</head>
<body>

<div id="adList" class="container">

    <div id="ads" class="row">
        <form action="#" id="addAd" class="twelve columns">
            <label for="txtTitle">Title: </label><input type="text" id="txtTitle"
            <label for="txtBody">Content: </label><input type="text" id="txtBody"
            <label for="txtAuthor">Author: </label><input type="text" id="txtAuthor
            <button id="btnAdd">Add</button>
        </form>
    </div>

</div>
<!-- Our underscore templates -->
<script type="text/template" id="adTemplate">
    <ul>
        <li><%= title %></li>
        <li><%= body %></li>
        <li><%= author %></li>
        <li><%= publicationTime %></li>
    </ul>
</script>

<!-- Our libraries -->
<script type="application/javascript" src="js/lib/jquery.js"></script>
<script type="application/javascript" src="js/lib/underscore.js"></script>
<script type="application/javascript" src="js/lib/backbone.js"></script>

<!-- Our backbone scripts -->
<script type="application/javascript" src="js/models/Ad.js"></script>
<script type="application/javascript" src="js/collections/AdList.js"></script>
<script type="application/javascript" src="js/views/Ad.js"></script>
<script type="application/javascript" src="js/views/AdList.js"></script>

<script type="application/javascript" src="js/app.js"></script>

</body>
</html>
```

Vamos então decompor este ficheiro por secções de forma a compreender melhor a sua construção.

```
<head lang="en">
    <meta charset="UTF-8">
    <title>Ads4every1</title>

    <link rel="stylesheet" href="css/normalize.css"/>
    <link rel="stylesheet" href="css/skeleton.css"/>
    <link rel="stylesheet" href="css/ads4every1.css"/>
</head>
```

No elemento `head` do documento vamos definir o título da página e as folhas de estilo utilizadas pela aplicação.

```
<div id="adList" class="container">

    <div id="ads" class="row">
        <form action="#" id="addAd" class="twelve columns">
            <label for="txtTitle">Title: </label><input type="text" id="txtTitle"
            <label for="txtBody">Content: </label><input type="text" id="txtBody"
            <label for="txtAuthor">Author: </label><input type="text" id="txtAuthor
            <button id="btnAdd">Add</button>
        </form>
    </div>

</div>
```

Este é o elemento `adList` que é utilizado pela view `AdListView`, onde é definido o formulário na **linha 4** que é utilizado como fonte de informação para a adição de novos anúncios.

Inclusão do template:

```
<!-- Our underscore templates -->
<script type="text/template" id="adTemplate">
    <ul>
        <li><%= title %></li>
        <li><%= body %></li>
        <li><%= author %></li>
        <li><%= publicationTime %></li>
    </ul>
</script>
```

Inclusão das dependências da nossa aplicação:

```
<script type="application/javascript" src="js/lib/jquery.js"></script>
<script type="application/javascript" src="js/lib/underscore.js"></script>
<script type="application/javascript" src="js/lib/backbone.js"></script>
```

E finalmente os ficheiro criados para o funcionamento da nossa aplicação:

```
<script type="application/javascript" src="js/models/Ad.js"></script>
<script type="application/javascript" src="js/collections/AdList.js"></script>
<script type="application/javascript" src="js/views/Ad.js"></script>
<script type="application/javascript" src="js/views/AdList.js"></script>

<script type="application/javascript" src="js/app.js"></script>
```

É importante assinalar que a sequência de inclusão das dependências é fundamental para o funcionamento da aplicação, devido aos mecanismos inerentes ao processamento de Javascript no browsers. Mas no caso dos ficheiros Javascript da aplicação isto não é fundamental, pois estamos a utilizar a função do jQuery que apenas inicia a aplicação aquando da finalização do carregamento de todo o conteúdo da página.

Integração com API REST

O código necessário para a implementação da API REST está disponível no repositório github [https://github.com/ohvitorino/backbone-adverts/tree/master/api].

Conclusão

No início do artigo passamos em revista o metodologia de desenvolvimento de aplicações WEB Single-page, o ciclo de vida e a arquitetura típica de uma aplicação deste género. No final de artigo temos uma aplicação de Single-Page implementada dando uso à biblioteca *Backbone.js*, que nos permite adicionar anúncios e consultar todos os existentes. Vimos os principais componentes de uma aplicação MVP, como implementá-los e adicionar funcionalidade à nossa aplicação.

Recursos consultados

http://backbonejs.org/ (consultada em 21/06/2015)
https://msdn.microsoft.com/en-us/magazine/dn463786.aspx (consultada em 10/06/2015)
https:// developer.mozilla.org/en-US/docs/ Inner-browsing_extending_the_browser_navigation_paradigm (consultada em 10/06/2015)
http://singlepageappbook.com (consultada em 10/06/2015)
http:// victorsavkin.com/post/59496656297/building-large-backbone-applications (consultada em 10/06/2015)
https:// github.com/jashkenas/ backbone/wiki/Backbone%2C-The-Primer (consultada em 10/06/2015)
http://underscorejs.org (consultada em 21/06/2015)
Osmani, A, Developing Backbone.js Applications, O'Reilly, 2013
Potel, Mike, MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java, Taligent, Inc, 1996