Backbone.js

Data-driven javascript framework

Presentation

Part 1

Introduction

Backbone.js is:

- A data-driven JS framework
- Lightweight (< 10kb minified, including dependencies)
- Unobtrusive
- Easy to use with Rails
    - Data is manipulated through RESTful request.

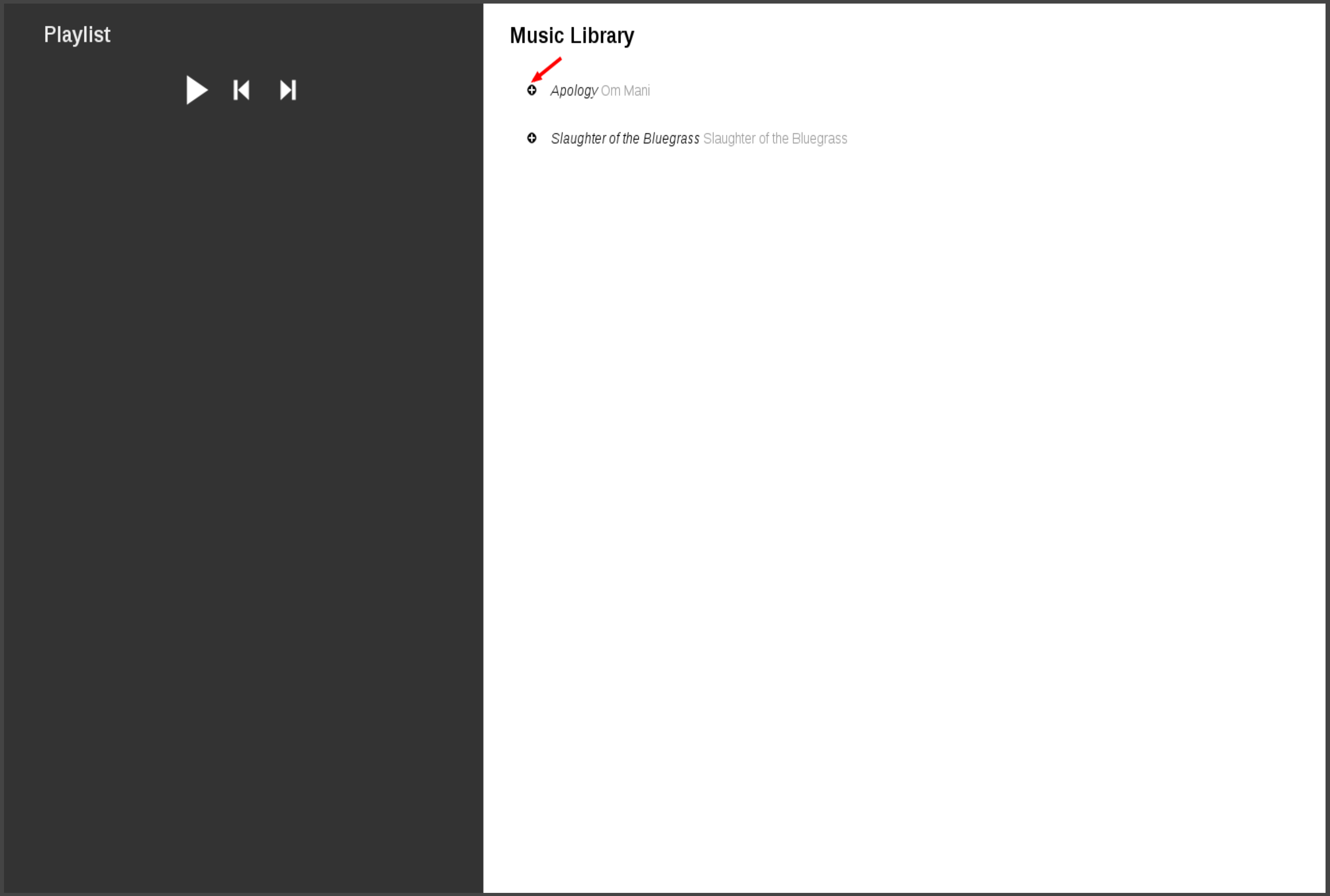
Data-driven, what does it mean?

That means an app with a better structure!

- No more jQuery code directly tied to the DOM.
- No more handling events caused by user input.

Data-driven workflow sample (1/4):

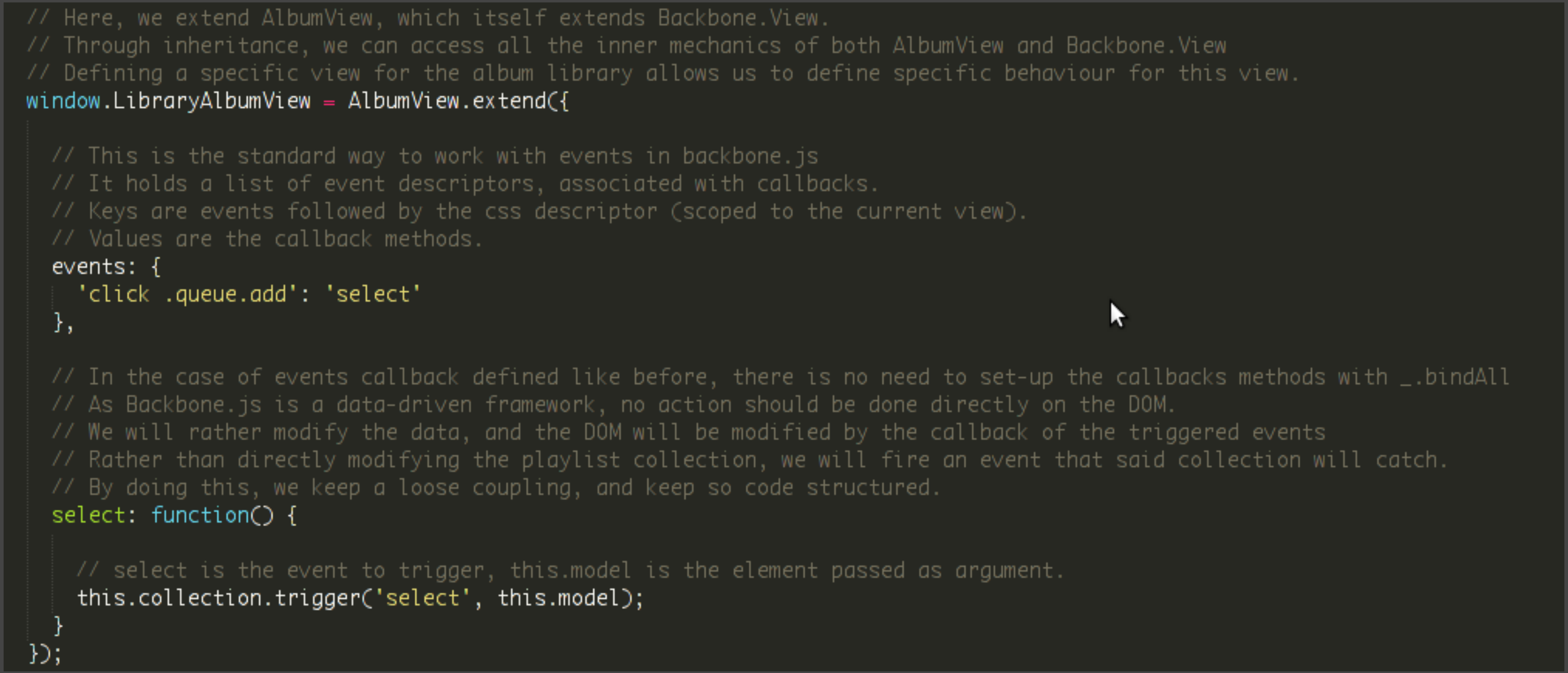
User input:
Click to add an event to the playlist.



Apology Om Mani

Slaughter of the Bluegrass Slaughter of the Bluegrass

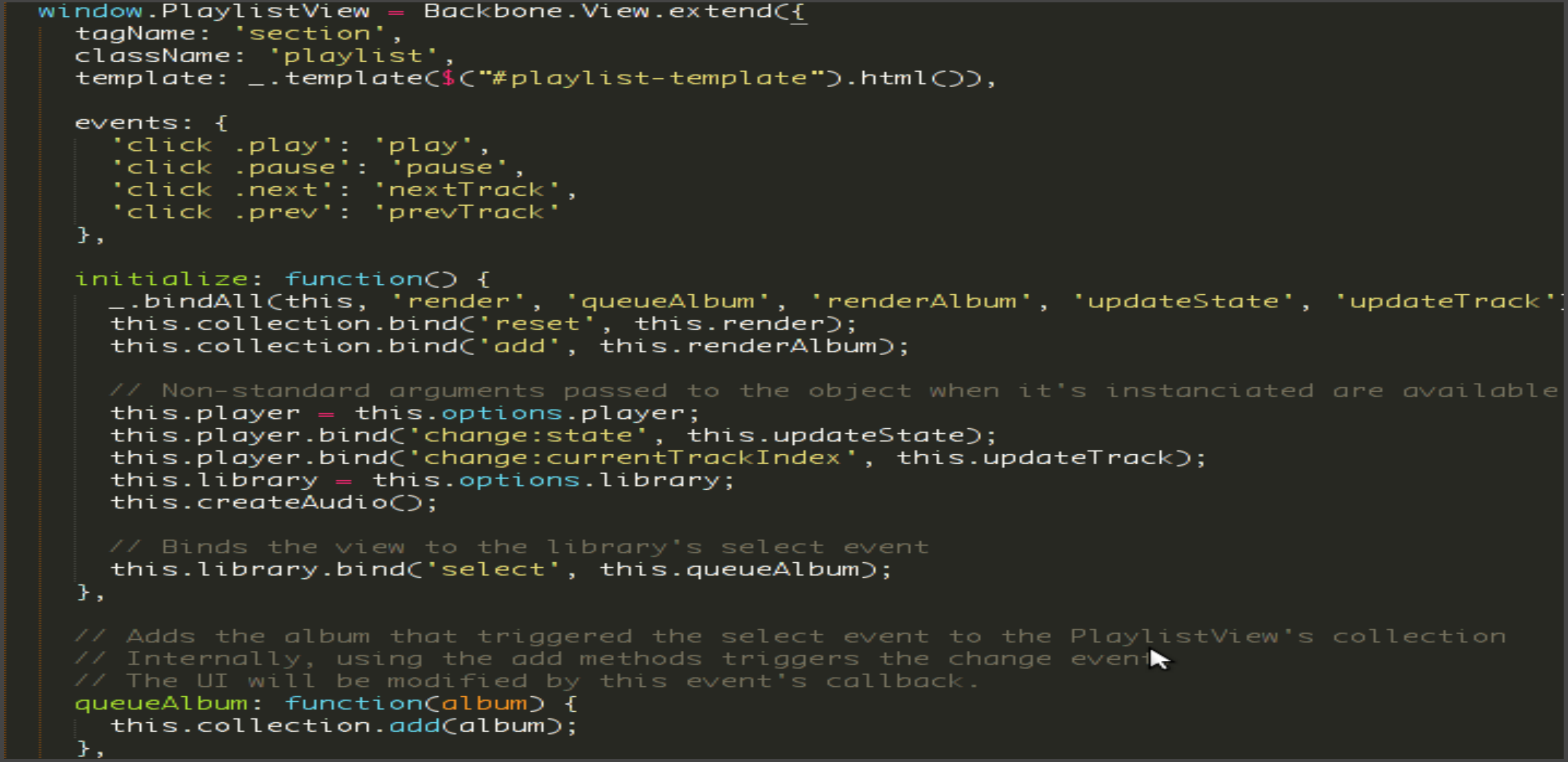
Data-driven workflow sample (2/4):



```
// Here, we extend AlbumView, which itself extends Backbone.View.
// Through inheritance, we can access all the inner mechanics of both AlbumView and Backbone.View
// Defining a specific view for the album library allows us to define specific behaviour for this view.
window.LibraryAlbumView = AlbumView.extend({

  // This is the standard way to work with events in backbone.js
  // It holds a list of event descriptors, associated with callbacks.
  // Keys are events followed by the css descriptor (scoped to the current view).
  // Values are the callback methods.
  events: {
    'click .queue.add': 'select'
  },

  // In the case of events callback defined like before, there is no need to set-up the callbacks methods with _.bindAll
  // As Backbone.js is a data-driven framework, no action should be done directly on the DOM.
  // We will rather modify the data, and the DOM will be modified by the callback of the triggered events
  // Rather than directly modifying the playlist collection, we will fire an event that said collection will catch.
  // By doing this, we keep a loose coupling, and keep so code structured.
  select: function() {

    // select is the event to trigger, this.model is the element passed as argument.
    this.collection.trigger('select', this.model);
  }
});
```

The click event does only one thing: trigger a custom 'select' event.

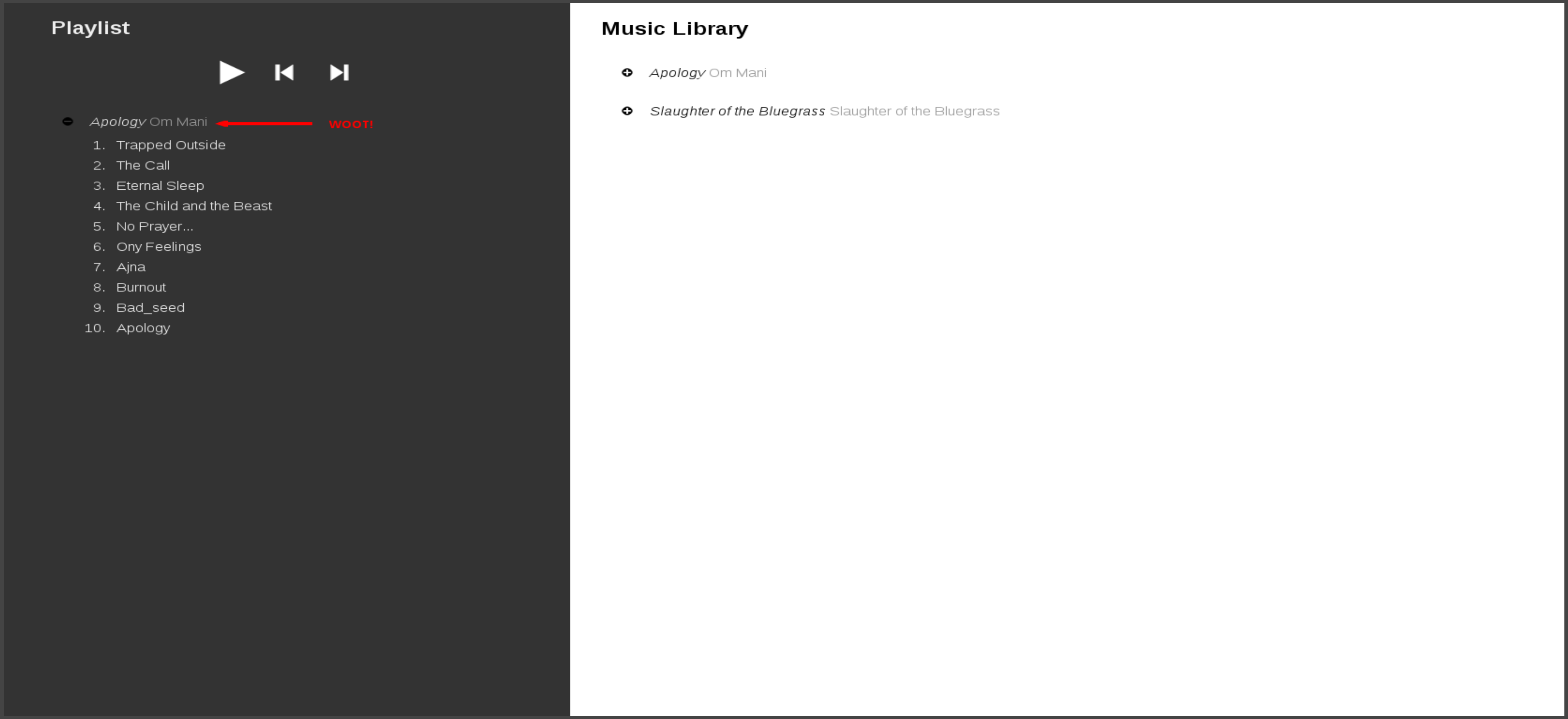
Data-drive workflow sample (3/4):



```
window.PlaylistView = Backbone.View.extend({
  tagName: 'section',
  className: 'playlist',
  template: _.template($("#playlist-template").html()),

  events: {
    'click .play': 'play',
    'click .pause': 'pause',
    'click .next': 'nextTrack',
    'click .prev': 'prevTrack'
  },

  initialize: function() {
    _.bindAll(this, 'render', 'queueAlbum', 'renderAlbum', 'updateState', 'updateTrack'
    this.collection.bind('reset', this.render);
    this.collection.bind('add', this.renderAlbum);

    // Non-standard arguments passed to the object when it's instanciated are available
    this.player = this.options.player;
    this.player.bind('change:state', this.updateState);
    this.player.bind('change:currentTrackIndex', this.updateTrack);
    this.library = this.options.library;
    this.createAudio();

    // Binds the view to the library's select event
    this.library.bind('select', this.queueAlbum);
  },

  // Adds the album that triggered the select event to the PlaylistView's collection
  // Internally, using the add methods triggers the change event
  // The UI will be modified by this event's callback.
  queueAlbum: function(album) {
    this.collection.add(album);
  },
```

The playlistView catches the 'select' event, and queues the album into its own Album collection.
The collection.add method then triggers the renderAlbum method. The album's view is rendered, and appended to the DOM.

Data-driven workflow sample (4/4):


Playlist

Apology Om Mani

WOOT!

1. Trapped Outside
2. The Call
3. Eternal Sleep
4. The Child and the Beast
5. No Prayer...
6. Ony Feelings
7. Ajna
8. Burnout
9. Bad_seed
10. Apology

Music Library

Apology Om Mani

Slaughter of the Bluegrass Slaughter of the Bluegrass

Finally, the DOM is modified, in response to data modification. The code is clean and re-usable.

Comparison with a pure jQuery web-app

Backbone.js:

- Data is organised into models and collection.
- User input affects the data.
- Data modification is the source of events.
- Event handlers modify the DOM.

jQuery:

- Data is tied to the DOM.
- User input affects the DOM directly.

Components of a Backbone.js application

Backbone.js implements four main classes:

1. Models
2. Views
3. Routers
4. Collections

Backbone models

Backbone models are:

- Classes, with attributes, methods, etc...
- Data-containers
- Events are fired when an attribute changes.
- If an attribute is an array of elements, only the change of the root of the attribute will trigger an event.
    - If it doesn't seem right, maybe you should think of changing the attribute to a Backbone collection.

Backbone collections

Backbone collections are:

- Collection of models
- Resemble ActiveRecord's collections
- Populated with data from a remote source
    - Ruby on Rails app, anyone?
- Can be manipulated with handy methods from Underscore.js, such as:
    - Each
    - Pluck
    - Map
    - Etc...

Backbone Views

Backbone views are where the magic happens:

- Views are rendered using a template system
- Views can register to events, such as:
  - Model's attribute change
  - Collection population
  - etc...
- Trigger events on user input for other views to react.

Backbone Routers

Backbone routers (formerly known as Controllers) are:

- The way a Backbone app's state is saved/restored.
- Initialize the application in a pre-determined state.
- Handle page navigation and history.

Do's and Don't's

Do:

- Modify the DOM in reaction to data manipulation.
- Modify data on user input.
- Re-use Views and collections (different data, same model).
- Use inheritance in a smart way.

Don't:

- Modify the DOM on user input.

ToDo app Walk-through

Part 2

Application specifications

Simple ToDo app:

- Create tasks
- Delete tasks
- Update tasks
- Flag tasks for deletion
- Delete all flagged tasks
- Count the number of tasks
- Data persistence (through localStorage)

Application components

Models:
- ToDo: Will hold the information about a specific task

Collections:
- TodoList: Will hold a set of tasks

Routers:
- BackboneTodo: Will initialize the application on page load

Views:
- Page: Will be the main container of the application
- ToDo: Will display a single task

Step 1: Placeholders and structure

In this step, we:

- Create the necessary directory structure
- Write just the code necessary to initialize the app:
  - The page view
  - A really basic router
  - A file to instantiate and initialize the router on page load.

Step 2: The Todo model

In this step, we add a Todo model that must:

- Extend Backbone.Model
- Not be flagged as 'done' by default
- Have a helper method to check their state (current/done)
- Have a method to toggle between states
- Have a method to update themselves

Step 3: The Todo view

Ok, now that we have a Model, let's write the corresponding View that:□

- Has enough code to display a task properly
- Erase itself if the corresponding model is deleted
- Provides a way to edit the content of the task
- Provides a way to flag a task for deletion

Step 4: The TodoList collection

Last step before real interactivity (no more console, yay!):
Create a collection to manage a set of tasks.
This collection must:

- Manage models of the Todo class
- Have a method that returns the number of items flagged for deletion
- Have a method that returns the number of remaining items
- Have a method that purges the collection from items to delete.

Step 5: Binding the view to the logic

In this step, we will:

- Make the views listen to user-induced events
- Modify data in reaction to these events
- Update the DOM in reaction to the data change
- Add statistics at the bottom of the view

Step 6: Data persistence

All of this is good, but as it is, the application is quite useless! There is no way (yet) to make data persist when reloading the page.

For this example, we will use the localStorage features of web-browser:

- Access with Backbone.js' localStorage extension
- Save, update and delete models to/from the localStorage on events
- Reload previously stored tasks on page load

Conclusion

Part 3

What is left to try?

Done:

- Create a sample music player.
- Populate Data through a RESTful request.
- Manipulate Data, and through events, update the DOM.
- Save data into localStorage.

ToDo:

- Save the modifications to model objects remotely.
- Integrate more closely with a web-app.
- Read more litterature!

Remember

Data is the center of your app. Every action should either trigger or be triggered by data-manipulation.

Sources

Backbone.js website:
http://documentcloud.github.com/backbone/

Peepcode screencasts:
http://peepcode.com/products/backbone-js
http://peepcode.com/products/backbone-ii

Inspiration for the sample app:
http://documentcloud.github.com/backbone/examples/todos/index.html

This presentation:
https://github.com/elhu/Presentation-Backbone.js

Any questions?

Now is the time!

Demo repositories:
github.com/elhu/test-backbone
github.com/elhu/backbone-todo