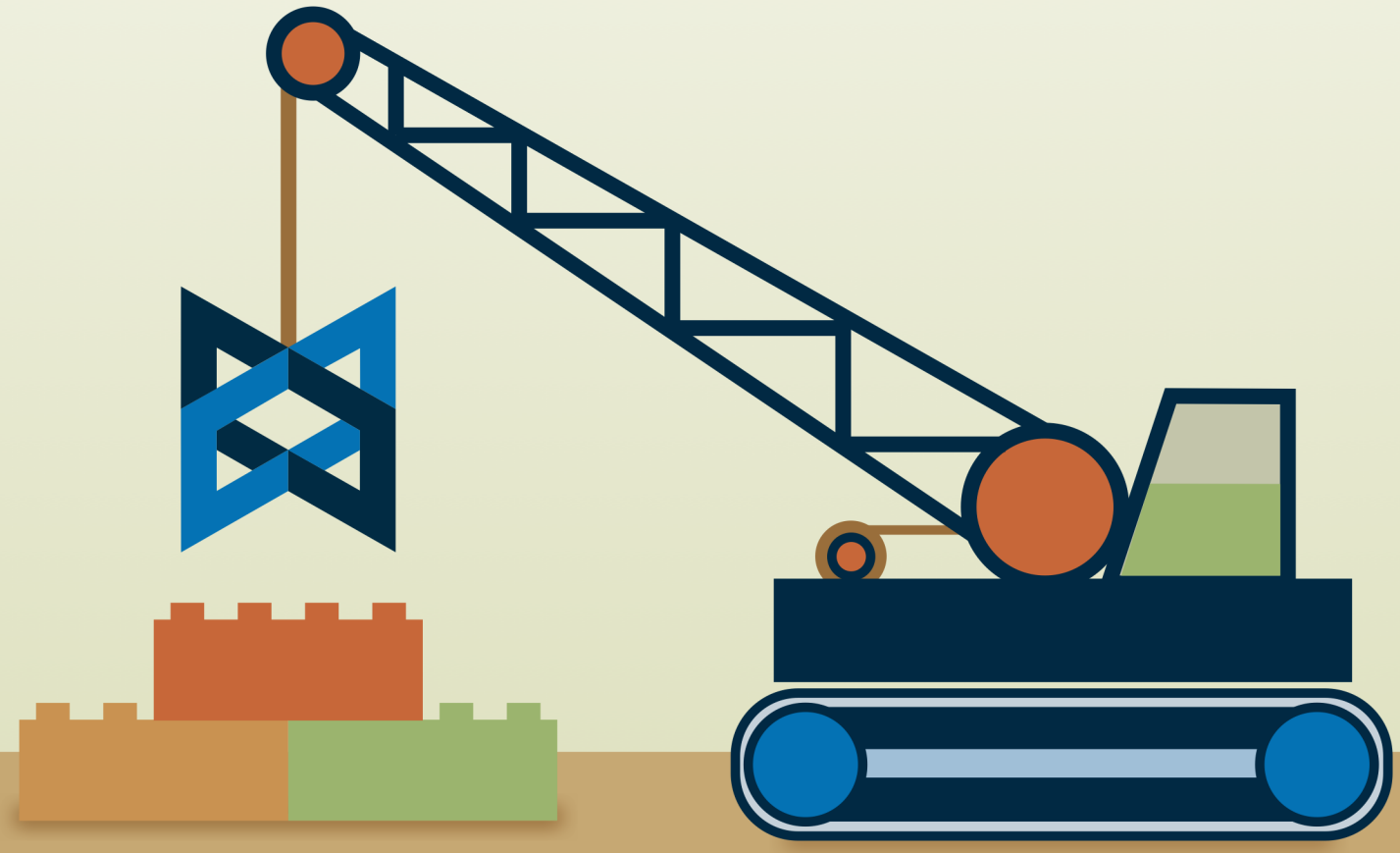
Covers Backbone.js v1.0

BUILDING BACKBONE PLUGINS



By Derick Bailey

Eliminate The Boilerplate In Backbone.js Apps

Building Backbone Plugins

Eliminate The Boilerplate In Backbone.js Apps

Derick Bailey and Jerome Gravel-Niquet

Tweet This Book!

Please help Derick Bailey and Jerome Gravel-Niquet by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm leveling up my #Backbone skills with the #BackbonePlugins e-book! http://backboneplugins.com

The suggested hashtag for this book is #BackbonePlugins.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#BackbonePlugins

Contents

Chapter 12: Building Backbone.localStorage **1**
Storing data on the client 1
Overriding `Backbone.sync` properly 3
Syncing the data 3
Universal Module Definition 5
Lessons Learned 7

Chapter 12: Building Backbone.localStorage

Lots of real-world apps need to persist data in some form of storage. This is normally achieved by creating a server which acts as an interface between a database and your client. Sadly, this adds a ton of dependencies and complexity. People are less likely to contribute to your project if it's difficult to setup.

When I built Backbone Todos, I didn't want to add any dependencies of the sort. But how would I be able to store data without a dependency on a database system?

For years now, browsers expose a nifty `localStorage` object, which does exactly what the name implies: it stores data locally. More specifically, in a file on your hard drive, a file which your browser knows how to access and parse.

The answer, then, is to use `localStorage` as the data store. While it is a dependency, it is not one that has to be installed. Your browser already has it built in.

Storing data on the client

Choosing to use `localStorage` made sense at the time. Although other solutions like IndexedDB and Web SQL were more powerful, they were less widespread.

`localStorage`'s API can be a pain to use if you're building something complex. You're better off wrapping it with syntactic sugar. For instance, this little `save` method:

```
1  save: function() {
2    this.localStorage().setItem(this.name, this.records.join(","));
3  }
```

In this case, `this.records` holds a simple array of all the documents managed by Backbone.localStorage.

Generating IDs

For some client-only apps, instances might/will not have IDs. Some people use Backbone.localStorage for caching data temporarily (and so, they might have IDs) and others use it as their primary storage strategy (and so, don't have IDs unless they manually generate ones for their models.) But for data storage and retrieval, it is necessary to generate an ID for every object.

Backbone.localStorage needs to support both cases. The former is easy, use whatever ID is supplied. The other case, generating an ID, can be achieved with this snippet of code:

```
// Generate four random hex digits.
function S4() {
  return (((1+Math.random())*0x10000)|0).toString(16).substring(1);
};

// Generate a pseudo-GUID by concatenating random hexadecimal.
function guid() {
  return (S4()+S4()+"-"+S4()+"-"+S4()+"-"+S4()+"-"+S4()+S4()+S4());
};
```

What are `S4()` and `guid()`?

The function `S4` generates a random number, converts it to a string (resembling *"298c"*) and then `guid` assembles them to create a complex string which will almost assuredly be unique (enough for our needs.) The final form looks like *"4df366d6-26d3-dab8-8018-ca6252b252a7"*

Then in the `create` method of the plugin, we use the generated ID:

```
if (!model.id) {
  model.id = guid();
  model.set(model.idAttribute, model.id);
}
```

`localStorage` gotchas

`localStorage` is pretty good when it comes to storing simple key/value data. You come across a few issues when trying to store full data representation of complex models.

Size limit

The default size limit on data you can store in `localStorage` can vary wildly. It may be set by the user, a website may request a user to allow for more storage and, by default, it's set at 5 MB (may vary depending on the browser.)

Once you hit that limit, you need to handle a "QUOTA_EXCEEDED_ERR". The way Backbone.localStorage handles that is by throwing it back at the app through the `options.error` callback.

Performance

`localStorage` is not very fast. It saves data to disk. As much as possible, you want reduce the number and weight of your writes.

In first versions of Backbone.localStorage, everything used to be held in a single, monolithic, key inside `localStorage`. It wasn't a great idea.

Later on, when that performance issue hit, a refactor occurred to store data more or less like a normal database. The plugin now saves an "index" of all the IDs for a collection in a key and then has one key for each record.

Retrieving the data is still pretty fast (even if it has to go through multiple keys.) The real difference, when it comes to writing data to `localStorage`, is about a 66x performance gain.

Overriding `Backbone.sync` properly

Taken from the Backbone documentation (emphasis mine): > `Backbone.sync` is the function that Backbone calls every time it attempts to read or save a model to the server. By default, it uses jQuery.ajax to make a RESTful JSON request and returns a jqXHR. **You can override it in order to use a different persistence strategy, such as WebSockets, XML transport, or Local Storage**.

People often override `Backbone.sync` to accommodate custom headers/params needed to use with their server.

Overriding is too destructive for a proper Backbone plugin. In Backbone.localStorage, we check for a property on the model or collection (quite simply `localStorage`) and then override the `Backbone.sync` function with a small function which checks for that property and calls either the old `Backbone.sync` or a new pimped one. Like so:

```
Backbone.ajaxSync = Backbone.sync;

Backbone.getSyncMethod = function(model) {
  if(model.localStorage || (model.collection && model.collection.localStorage)) {
    return Backbone.localSync;
  }

  return Backbone.ajaxSync;
};

// Override 'Backbone.sync' to default to localSync,
// the original 'Backbone.sync' is still available in 'Backbone.ajaxSync'
Backbone.sync = function(method, model, options) {
  return Backbone.getSyncMethod(model).apply(this, [method, model, options]);
};
```

Syncing the data

Since we're going through the database directly, instead of going through a server, we don't most of what the current `Backbone.sync` method does.

Let's take a closer look at the replacement function.

Sync methods

```
Backbone.sync = function(method, model, options) {
```

The first argument passed to the function is a method for syncing the data. It can be either one of those: `create`, `read`, `update` or `delete`.

Normally, Backbone translate those to HTTP methods and then passes that to `$.ajax` which calls a endpoint defined by a collection or model's `url` property.

We need none of that here and so the "server logic" for handling data is directly in the plugin

```
switch (method) {
  case "read":
    resp = model.id != undefined ? store.find(model) : store.findAll();
    break;
  case "create":
    resp = store.create(model);
    break;
  case "update":
    resp = store.update(model);
    break;
  case "delete":
    resp = store.destroy(model);
    break;
}
```

Granted, a `switch` isn't the fastest way to do this, but it's not slow enough to sacrifice the legibility it provides.

Faking the response

Once data is gathered, we need to call `success` or `error` callbacks. These are usually "wrapped" by Backbone to handle operating on collections/models **and** responding to manually specified callbacks (for instance: providing the `success` and/or `error` keys in the options for a `fetch`, `create`, etc. method call.)

The `success` Callback

```
if (resp) {
  if (options && options.success) {
    if (Backbone.VERSION === "0.9.10") {
      options.success(model, resp, options);
    } else {
      options.success(resp);
    }
  }
  if (syncDfd) {
    syncDfd.resolve(resp);
  }
}
```

Mostly, this piece of code calls the `success` option with the data returned from storage.

For a while now, Backbone has been using `$.Deferred` and so it also needs to be handle. Those "promises" need to be `resolved` in the case of success and `rejected` in case of failure.

Funky Backbone fact

In Backbone 0.9.10, `Backbone.sync` called the `success` callback with arguments in a different order.

Going from `success(response)` to `success(model, response, options)`.

This was then reverted in the next stable release, Backbone 1.0. Therefore any plugin dealing with `sync` had to have a conditional for 0.9.10 specifically.

Eventually, this little "shim" will be removed from `Backbone.localStorage`.

The `error` Callback

Remember the `localStorage` woes? Due to its not-quite-yet-supported-everywhere-the-same-way nature, error handling is a must.

The crux of the replaced `sync` method is actually wrapped inside a `try {} catch (error) {}` just for that purpose.

The process then goes on doing its thing and if an error occurred earlier, it'll simply call `options.error` and the promise's `reject` method.

Universal Module Definition

UMD[1] is "[...] patterns for JavaScript modules that work everywhere"

[1] https://github.com/umdjs/umd

In a nutshell, it's boilerplate code to support the various module loading standards out there. AMD, CommonJS or plain old globals. In Backbone.localStorage, we've gone the way of supporting all possibilities, with this little piece of code:

```
(function (root, factory) {
   if (typeof exports === 'object' && root.require) {
     module.exports = factory(require("underscore"), require("backbone"));
   } else if (typeof define === "function" && define.amd) {
      // AMD. Register as an anonymous module.
      define(["underscore","backbone"], function(_, Backbone) {
        // Use global variables if the locals are undefined.
        return factory(_ || root._, Backbone || root.Backbone);
      });
   } else {
      // RequireJS isn't being used. Assume underscore and backbone are loaded in\
 <script> tags
      factory(_, Backbone);
   }
}(this, function(_, Backbone) {
```

Let's dig a little deeper in this code.

The function takes the global context (`root`) as a first argument and a `factory` as the second.

The first one, the global context, is either `window` in the browser or `global` in node.js.

The second is our main code. All its dependencies are defined as arguments. In our case, we need Underscore.js _ and Backbone.js `Backbone`.

Within the UMD, we first detect CommonJS support by checking for the existence of an `exports` object and a function `require`.

```
if (typeof exports === 'object' && root.require) {
  module.exports = factory(require("underscore"), require("backbone"));
}
```

If it is present, assign the result of our factory to `module.exports`.

In the case CommonJS is not present, we check for AMD support. A similar approach is used by evaluating if `define` is a function and if it complies to AMD standards.

```
if (typeof define === "function" && define.amd) {
  // AMD. Register as an anonymous module.
  define(["underscore","backbone"], function(_, Backbone) {
    // Use global variables if the locals are undefined.
    return factory(_ || root._, Backbone || root.Backbone);
  });
}
```

A bit more complex since it might need to load libraries asynchronously, we use the `define` method to "define" our module. It takes an array of dependencies and then a function with those dependencies resolved as arguments.

Often, libraries don't support AMD. This is very much the case for Underscore and Backbone (Underscore used to support it way back when.) Therefore, if they're undefined within our function, we use the global context's `Backbone` and `_`.

Finally, we assume Underscore and Backbone are already loaded in the global context if no other module loading strategy is being used.

```
} else {
  // RequireJS isn't being used. Assume underscore and backbone are loaded in <sc\
ript> tags
  factory(_, Backbone);
}
```

This whole module loading thing is fairly important for plugins and libraries in general. It's an easy fix to avoid people having to modify your code for this single purpose. In turn, people are more likely to use a pristine version of your library which reduces the difficulty of upgrading.

Furthermore, it's generally a good practice. Your code will be more contained, less likely to spread in the global context.

Lessons Learned

Backbone's ability to have parts of it replaced or removed speak s to the flexibility of this library. When one or more parts don't work the way you want, you are free to find ways to make it work. Having the ability to modify and/or replace the sync mechanism is a great example of this.

Be Careful When Replacing Things

Just because you can replace something, doesn't mean you should. Sometimes it is better to augment what was there and wrap it in your own code so that you can use the original behavior or the new behavior as needed. This is often referred to as a decorator or proxy pattern, and can help you add features and functionality without altering existing code.

Widespread Adoption Trumps Power

Software development is a series of trade-offs and choices, drawbacks and benefits that have to be weighed against each other. There will be times when you have to make a choice that you don't necessarily like. You may have to support an old browser because a client requires it, or you may have to use `localStorage` instead of IndexDB because you need better support across multiple browsers.

Understanding the context and constraints in which your application will run is important when making decisions. In the case of the Backbone Todos sample application, browser support with fewer external dependencies was more important than using something wide spread or more powerful.

Convenience Comes At A Price

Having something available when you need it, and available across many browsers doesn't always mean it's the best thing out there. The `localStorage` in browsers has limitations including speed and storage size. Be sure you are weighing the pros and cons of convenience and availability against the actual needs of your system.

Reproduce The Complete Behavior, Not Just The API

It's easy to think that reproducing the API of an object or method is good enough. But the API for the object/method is only the surface area of an iceberg under the sea. Before you can replace something, you need to understand the hidden behaviors and expectations of code that uses what you are about to replace.

In the case of `Backbone.sync`, replacing the `sync` method with another method of the same API is not enough. The new implementation needs to ensure the behavior of calling `success` and/or `error` are maintained so that calling code will continue to work as expected.