Extracted from:

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

Jack Moffitt
and Fred Daoud

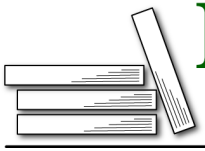
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

Jack Moffitt
Fred Daoud

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Bruce A. Tate (series editor)
Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93778-563-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2014

Day 1: Building Objects and Synchronizing Changes

In the first day of our CanJS journey, we'll learn how can.Construct helps with creating hierarchies with inheritance. This is useful in itself but is also important to know because the other parts of CanJS build on can.Construct.

Next, we'll discover can.Observe, an extremely useful part of CanJS that triggers events when changes occur. This helps with keeping components independent. We'll also see how can.Model makes it easy to keep data synchronized between client and server. Finally, we'll discuss view rendering and how using can.Observe objects in views is particularly useful because of live binding.

Let's begin by setting up the CanJS library and its supporting cast.

Hello, CanJS!

The first thing we'll do is look at a minimal setup for loading and using CanJS with jQuery. By running the following "Hello, World"–type of program on your own computer, you'll confirm that you've got things working properly and you'll be able to modify the files to experiment with your own code.

If you look at the book's sample code, under the canjs/public folder, you'll find that jQuery and CanJS are set up in the lib directory as follows:

```
+lib
 |+canjs.com-1.1.8
 | `-can.jquery.js
 | `-(other can.*.js files)
 `+jquery
   `-jquery-1.10.2.js
```

Now, look at the canjs/public/index-basic.html file. This is a regular HTML page that loads jQuery and CanJS using the standard <script> tags:

canjs/public/index-basic.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>CanJS Basic</title>
  </head>
  <body>
    <div id="result"></div>
  </body>
  <script src="lib/jquery/jquery-1.10.2.js"></script>
  <script src="lib/canjs.com-1.1.8/can.jquery.js"></script>
  <script src="index-basic.js"></script>
</html>
```

The last file, index-basic.js, is the place for you to write your code to experiment with CanJS. If you open the file in your favorite code editor, you'll see that it currently contains some simple code just to confirm that jQuery and CanJS are properly loaded:

canjs/public/index-basic.js

```
$(document).ready(function() {
  // Use can for CanJS
  var $result = $("#result");
  can.each(["One", "Two", "Three"], function(it) {
    $result.append(it).append(", ");
  });
  $result.append("Go CanJS!");
});
```

Open the index-basic.html file in your browser. If you see the result One, Two, Three, Go CanJS!, you know that both jQuery (from the use of $ in the preceding code) and CanJS (from the call to can.each) are working. That's all there is to it. Congratulations, you're ready to start exploring CanJS! You can edit index-basic.html and index-basic.js to experiment with your own code.

Now let's start exploring the different moving parts of CanJS, beginning with the foundational building block, can.Construct.

Constructing and Extending Objects

As you can see in Figure 2, *The core CanJS stack*, on page ?, many parts of CanJS inherit from can.Construct. Let's have a closer look.

JavaScript is a *class-free* language. It uses prototypal inheritance instead of classical inheritance. The nuances involved can be a challenge, especially if you are used to more conventional object-oriented languages. can.Construct removes the complexity by providing a factory that makes it painless to quickly build objects with common properties. can.Construct is essentially a function that you call to create hierarchies with single-parent inheritance. You can call parent functions and even override them in the children.

To use can.Construct, call its extend function with an object containing properties, and you get back a constructor function. Use new to create objects with those properties. Here's an example:

canjs/public/concepts/concepts-test.js

```
var Example = can.Construct.extend({
  count: 1,
  increment: function() {
    this.count++;
  }
});
```

```
var example = new Example();
example.increment(); // example.count is now 2
```

To pass parameters when creating new objects, define an init function:

canjs/public/concepts/concepts-test.js

```
var Example = can.Construct.extend({
  init: function(count) {
    this.count = count;
  }
});
var example = new Example(42); // example.count is 42
```

Here's the kicker: for inheritance, define a child by calling the parent's constructor function (without new), passing the child's properties. The child inherits from the parent; it can add new properties and override the parent's properties. A child can even call its parent's functions with _super:

canjs/public/concepts/concepts-test.js

```
var Parent = can.Construct.extend({
  init: function(count) {
    this.count = count;
  },
  increase: function() {
    this.count++;
  },
  read: function(prefix) {
    return prefix + " " + String(this.count);
  }
});

var Child = Parent({
  // Child inherits the init function

  // Override increase
  increase: function() {
    this.count += 10;
  },
  // Add new function: decrease
  decrease: function() {
    this.count--;
  },
  // Override read, but call parent's version
  read: function() {
    return this._super("Count is") + "!";
  }
});

var child = new Child(2); // calls Parent's init
child.increase(); // calls Child's increase
```

```
child.decrease(); // calls Child's decrease
child.count; // returns 11
child.read(); // returns "Count is 11!"
```

Finally, when you call can.Construct with *two* parameters, the first parameter defines the static properties. The second parameter is the prototype (or instance) properties we've been using so far. Here's an example of that:

canjs/public/concepts/concepts-test.js

```
var Example = can.Construct.extend({
  staticCount: 0,
}, {
  protoCount: 0
});

var example1 = new Example();
var example2 = new Example();

example1.constructor.staticCount = 2;
example1.protoCount = 2;

Example.staticCount; // returns 2
example2.constructor.staticCount; // returns 2
example2.protoCount; // returns 0
```

Notice how static properties are accessed from an instance via its constructor property or are accessed directly from the constructor function itself, as in Example.staticCount.

Remember that if you pass only one parameter to can.Construct, you define *prototype* properties. If you need a constructor function with *only static* properties, be sure to pass *two* parameters, with an empty object as the second, like so:

canjs/public/concepts/concepts-test.js

```
var ExampleStatic = can.Construct.extend({
  staticCount: 4
}, {
});
```

Knowing how to define a construct with just static properties is useful because that's often all you need when creating models, as we'll soon see when we talk about can.Model.

Other core parts of CanJS, such as can.Observe, can.Model, and can.Control, build on can.Construct. Let's continue our CanJS tour with can.Observe.

Observing Attribute Changes

The core, the heart, the *soul* of a CanJS application is the observable object, or simply the "observe."

can.Observe provides the ability to observe attribute changes on an object. This is extremely important because it provides a great way to keep an application's components cleanly decoupled. As you build your application, blocks of code remain independent of each other and communicate via observable objects. You never end up with a tangled mess of unmaintainable code!

Imagine an online bookstore. You could have several user interface components on a page: a list of books, quantities remaining in the inventory, and the books in your cart with the total cost. When you add a book to your cart, each component needs to be refreshed to reflect recalculated values. As you can see in the following figure, if the "Add to cart" component refreshes the other components, you end up with direct references between components. This is bad because components are tied to other components instead of being independent.

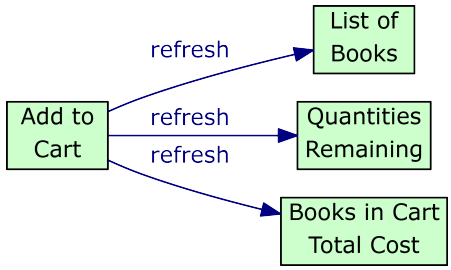


Figure 3—Direct references from one component to others

When using observes, the "Add to cart" component just needs to update an observe. The other components on the page *bind* to the observe, meaning that they listen for changes and update themselves to reflect the latest data. As shown in the next diagram, the "Add to cart" component is no longer tied to any other component. It just updates the observe, and components that listen for changes will be notified. "Add to cart" remains independent and doesn't need to know about the other components on the page.

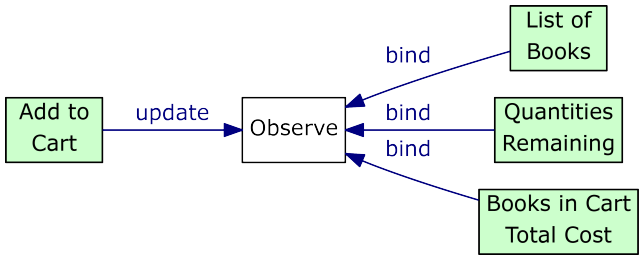


Figure 4—An observe keeps components decoupled

Here's how we create an observe and listen for changes on its attributes:

canjs/public/concepts/concepts-test.js

```
// Create an observe
var observe = new can.Observe({});

// Listen for changes on the "title" attribute
observe.bind("title", function(evt, newTitle, oldTitle) {
  console.log("title: newTitle=", newTitle, "oldTitle=", oldTitle);
});

// Set a value for the "title" attribute
observe.attr("title", "First");
// the console logs:
// title: newTitle= First oldTitle= undefined

// Set another value for the "title" attribute
observe.attr("title", "Second");
// the console logs:
// title: newTitle= Second oldTitle= First
```

As you can see, attributes are set with the attr method, with the attribute name and value as parameters. Supply only the attribute name to *read* the value, as in observe.attr("title").

You can also listen for changes on *any* attribute by binding to change:

canjs/public/concepts/concepts-test.js

```
observe.bind("change", function(evt, attr, how, newValue, oldValue) {
  console.log("change: attr=", attr, "how=", how,
    "newValue=", newValue, "oldValue=", oldValue);
});
observe.attr("title", "Third");
// change: attr= title how= set newValue= Third oldValue= Second
observe.removeAttr("title");
// change: attr= title how= remove newValue= undefined oldValue= Third
```

Note that if you actually have an attribute called `change`, you can also listen for changes with `bind("change", ...)`. However, your function gets called for *all* changes, so you would check `attr` to see if `change` is the attribute that was changed.

Finally, observe *lists* are similar to observes but are for lists of values. In this case, we can listen for values being added to or removed from the list:

canjs/public/concepts/concepts-test.js

```
var observe = new can.Observe.List([42, 44, 46]);
observe.bind("add", function(evt, newValues, index) {
  console.log("add: newValues=", newValues, "index=", index);
});
observe.bind("remove", function(evt, oldValues, index) {
  console.log("remove: oldValues=", oldValues, "index=", index);
});

observe.push(48);
// add: newValues= [48] index= 3
observe.splice(1, 2);
// remove: oldValues= [44, 46] index= 1
```

We'll see more of how we can make observes work for us throughout the rest of the chapter.

Building a CanJS Bookmarking Application Front End

For the rest of the chapter, we're going to build a single-page JavaScript front end for the bookmark server application that we created in Chapter 1, *Sinatra*, on page ?. We had created a rudimentary user interface with Mustache templates, but now we'll build something snazzier that dynamically refreshes without doing full-page reloads. CanJS is a client-side-only framework that works best with a REST and JSON interface (although a REST/JSON server is not required). As such, it's a perfect match for our Sinatra server. The following screenshot shows what our application will look like when we finish.

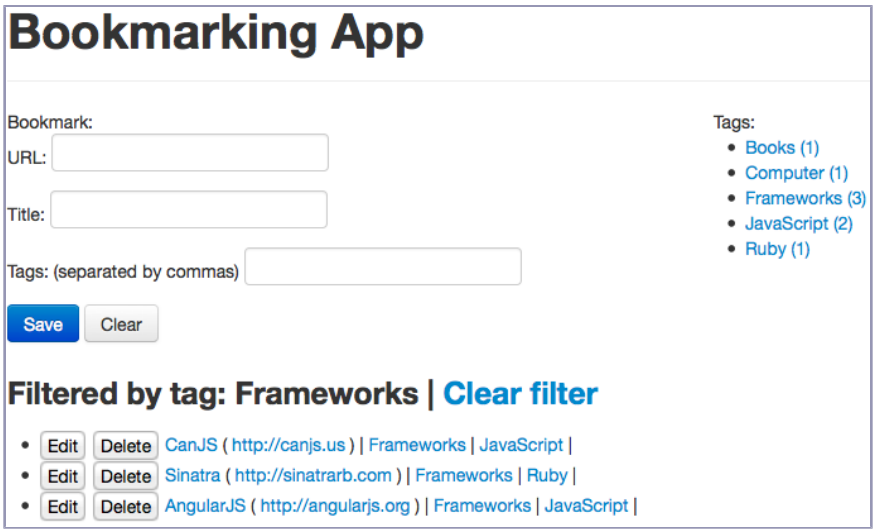


Figure 5—The complete CanJS bookmarking application

We'll begin building the application by creating a model for bookmarks that talks to our server.

Connecting Models to the Server

The first thing we'd like to do is retrieve the list of bookmarks from the server. We also want to think about keeping that data synchronized with the server. We could hook up listeners that send Ajax requests when attributes change and handle server responses to update attributes.

Or we could just use `can.Model`.

`can.Model` builds on `can.Observe` and adds a way to specify how you want your data to be retrieved from and sent to your server. Here is a `Bookmark` model that talks to our Sinatra bookmark server:

canjs/public/app/base/app.js

```
var Bookmark = can.Model.extend({
  findAll: "GET /bookmarks",
  create: "POST /bookmarks",
  update: "PUT /bookmarks/{id}",
  destroy: "DELETE /bookmarks/{id}",
}, {
});
```

We've defined the API that our server provides for creating, reading, updating, and deleting bookmarks. CanJS automatically converts the `findAll`, `create`, `update`,

and destroy properties into functions that issue Ajax requests and handle responses using the strings that we've associated with those properties. The first part of the string is the request method (GET, POST, PUT, DELETE). The second part is the request URI. Any part between {} in the URI is replaced by the object's corresponding attribute. For example, to update an existing Bookmark object, CanJS first calls bookmark.attr("id"), places the value at the {id} part of the URI, and finally issues a PUT request.

Models also handle server responses. For GET requests, CanJS parses the response as JSON and creates model objects with the same attributes. For PUT and POST requests, the model object's attributes are sent to the server in the request body. Any attributes that the server returns are updated on the model object. In particular, in response to a POST request, the server should return the id of the newly created object and any other new or changed attributes. It does not need to return attributes that have not changed. For a DELETE request, the model just checks that the server returned a success response code.

There is one more REST method that CanJS automatically handles in a can.Model object. We haven't defined it here because we won't need it for our bookmarking application. Can you guess what it is?

If you answered findOne, reward yourself with a trip to the refrigerator! Indeed, that method is for getting a *single* model object from the server according to its ID. For our bookmark model, that would be findOne: "GET /bookmarks/{id}".

To use these methods, we can call findAll and findOne as functions on the Bookmark object and receive the bookmarks or single bookmark via a callback function:

canjs/public/app/base/app.js

```
Bookmark.findAll({}, function(bookmarks) {
});

Bookmark.findOne({id:42}, function(bookmark) {
});
```

Notice the first argument to findAll and findOne is an object with any parameters to be sent along with the request. For findOne, we specified the id parameter.

For create and update, a call to save() on a model instance automatically issues a POST or PUT request. The absence of an id attribute on the model instance makes it a *new* object, the presence of an id makes it an *existing* object, and CanJS calls create or update, respectively. Finally, calling destroy() on a model instance issues the DELETE request with the object's id.

For lists of models, can.Model.List is the same as can.Observe.List with one additional feature: can.Model.List automatically removes an object from the list after you call the destroy() function.

We're able to retrieve bookmarks from the server. How do we display them?

Rendering Views

CanJS renders views with Mustache or EJS. We'll be using Mustache. Remember that we looked at the Mustache syntax in *Mustache*, on page ?.

Here is a Mustache template that renders the list of bookmarks, each with an Edit and a Delete button:

canjs/public/app/base/bookmark_list.mustache

```
<ul>
  {{#bookmarks}}
    <li>
      <button class="edit">Edit</button>
      <button class="delete">Delete</button>
       <a href="{{url}}">{{title}}</a>
     ( <a href="{{url}}">{{url}}</a> )
    </li>
  {{/bookmarks}}
</ul>
```

To render the template, call the can.view function with the URI of the template file without the .mustache extension. Notice that the file is located at public/app/base/bookmark_list.mustache. Remember that Sinatra serves files from the public directory. The URI to use for the template is /app/base/bookmark_list.

The other parameter to pass to can.view is the model to use in the template. Putting it all together, we have this:

canjs/public/app/base/app-test.js

```
// a list of bookmarks, as we would receive from the server
var bookmarks = [
  {url:"http://one.com", title:"One"},
  {url:"http://two.com", title:"Two"}
];
var viewModel = {bookmarks:bookmarks};
var element = $("#target");

// Render view by calling can.view
element.html(can.view("/app/base/bookmark_list", viewModel));

// can.view is implicitly called
element.html("/app/base/bookmark_list", viewModel);
```

Notice that we called can.view and added the result to the page element with jQuery's html function. You can also just call the html function directly with the path to the template and the view model. CanJS implicitly calls can.view for you. This also works for other jQuery methods, such as append, prepend, after, and so on.

Live Binding

When the objects that are passed to the view are observes, CanJS automatically does *live binding* so that an attribute change on the observe updates the view. This also works for *lists* of observes; add or remove an object, and the list view is automatically refreshed. Let's see an example:

canjs/public/app/base/app-test.js

```
// 'bookmarks' is now a list of observes
var bookmarks = new can.Observe.List([
  {url:"http://one.com", title:"One"},
  {url:"http://two.com", title:"Two"}
]);
var viewModel = {bookmarks:bookmarks};
$("#target").html("/app/base/bookmark_list", viewModel);

// The view automatically refreshes to display these changes
bookmarks[0].attr("title", "Uno");
bookmarks.push({url:"http://three.com", title:"Three"});
```

With live binding, you can focus on working with *models* and let CanJS take care of keeping views up-to-date. Because can.Model extends can.Observe, the objects you get back from Ajax calls such as findAll are observes. You can use them directly as view models for your templates and benefit from live binding.

What We Learned on Day 1

Today we looked at some of the key parts of CanJS: can.Construct, can.Observe, can.Model, and can.view. We learned how to create object hierarchies and use inheritance. We discovered that using observes keeps different parts of an application independent from one another. We saw how to create models that synchronize with the server and how to display them with views. One more essential component of CanJS is can.Control, which we'll tackle tomorrow.

We discussed the core tenets of CanJS: keep components separate, and use observes to propagate changes between components and models to synchronize data with the server. Finally, we learned how to take advantage of live binding by using observes in views so that we don't have to do manual refreshes.

Day 1 Self-Study

Find:

- The main CanJS forum (Hint: CanJS is a child of JavaScriptMVC.)
- The CanJS API documentation
- The CanJS implementation of TodoMVC

Do:

- Set up a simple CanJS page with an observe. Change the observe in your browser's JavaScript console and see the view update itself automatically.
- Experiment with CanJS on jsFiddle with the CanJS jQuery Template.[8]

8. http://jsfiddle.net/donejs/qYdwR/