Extracted from:

Dart 1 for Everyone

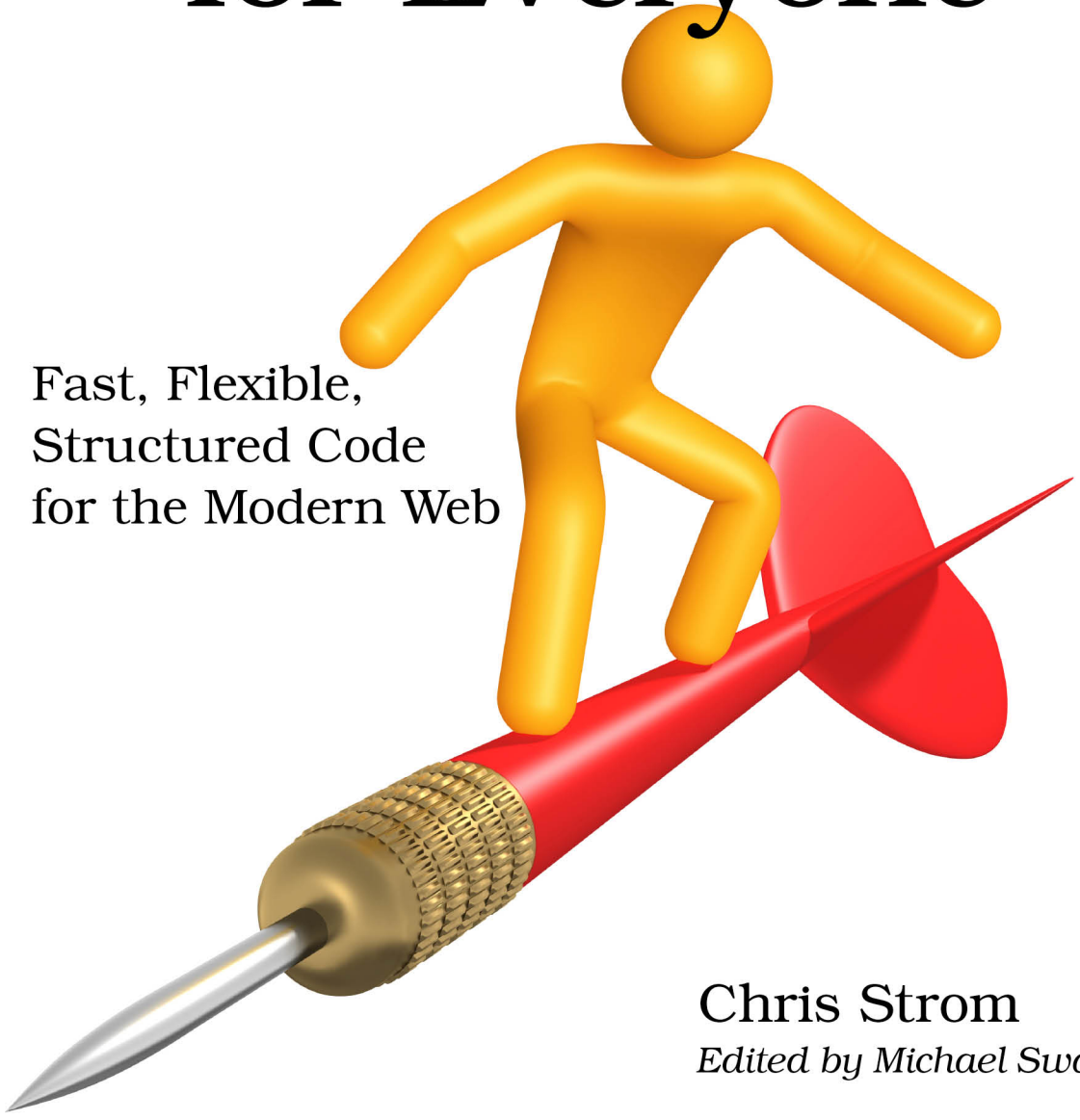
Fast, Flexible, Structured Code for the Modern Web

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Dart 1 for Everyone



Fast, Flexible, Structured Code for the Modern Web

Chris Strom

Edited by Michael Swaine

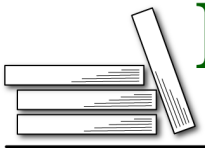
Dart 1 for Everyone

Fast, Flexible, Structured Code for the Modern Web

Chris Strom

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-25-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2014

CHAPTER 6

Project: MVC in Dart

In this chapter, you'll get your first real feel for what it means to write Dart code. Until now, our discussion has not strayed far from the familiar—or at least from what is similar to JavaScript.

We'll take the very simple comic book collection application from Chapter 1, *Project: Your First Dart Application*, on page ? and convert it to an MVC design pattern. Since this will be client based, it won't be Model-View-Controller. Rather, it will be Model-Collection-View, plus a Router, similar to Backbone.js.

We'll start by implementing collections of objects in Dart and then describe the objects themselves. Once we have the foundation in place, we'll take a look at views and templates.

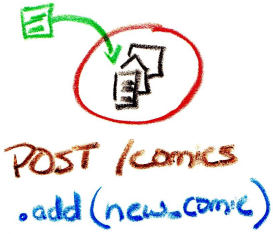
This is another "project" chapter, so we'll gloss over some Dart details to focus on writing code.

MVC in Dart

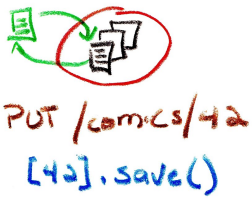
The foundation of our Hipster MVC library (of course that's the name) will be collections of objects, not the objects themselves. The collection maps nicely onto REST-like web services, resulting in a clean API for adding, deleting, and updating records.

Hearkening back to the first chapter, our comics collection can be retrieved via an HTTP GET of /comics. In Hipster MVC parlance, we will call that a fetch().

With REST-like resources, we can also refer to /comics as the URL root because it serves as the root for all operations on the collection of individual records. This is shown in the following sketches.



Adding a new comic to the collection is an HTTP POST operation on /comics. And, in hipsterese, that is an add().



To update a comic book with new information, we use HTTP's PUT, supplying the ID in the subpath of the URL: PUT /comics/42. From Hipster MVC's perspective, we retrieve the record, update it, and save it with save().

Lastly, to remove a record from the collection, we use the destroy() method. This will result in an HTTP DELETE. Again we use the collection URL including the ID.

Let's get started writing that code.

Hipster Collections

Recall from *Project: Your First Dart Application* that our comics.dart looks something like this:

your_first_dart_app/web/scripts/skel.dart

```
import 'dart:html';
import 'dart:convert';
main() {
  // Do stuff here
}
```

We replaced the // Do stuff here comment with code that retrieves the comic book collection from /comics and displays it on the web page. In MVC, the collection object retrieves the records, and a view object displays the contents of the collection.

mvc/web/scripts/comics.dart

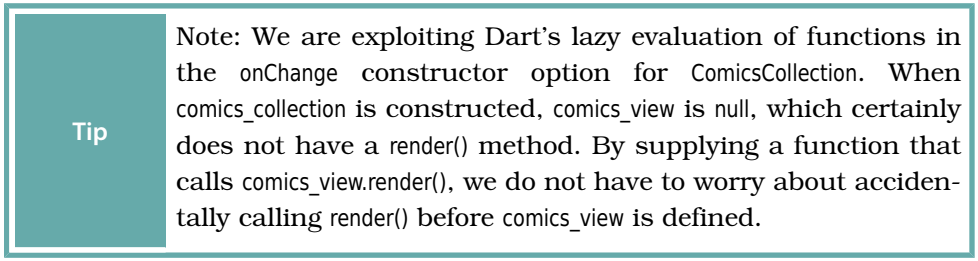
```
import 'dart:html';
import 'dart:convert';
import 'dart:collection';
main() {
  var comics_view, comics_collection;
```

```
  comics_collection = new ComicsCollection(
    onChange: ()=> comics_view.render()
  );
  comics_view = new ComicsView(
    el: document.query('#comics-list'),
    collection: comics_collection
  );
  comics_collection.fetch();

}
```

This is a first pass at MVC, not a final product. Already, it is quite promising. The ComicsCollection class needs very little construction—just a callback that re-renders the collection view when the collection changes. Similarly, ComicsView needs only an element on the page to which it can attach itself and, of course, a reference to the collection that it will display.

With both the collection and view constructed, we fetch the collection from the REST-like back-end server. Once the response comes back, the collection will be populated, resulting in a change. This change will invoke our callback, which will update the view. That is fairly tidy, which is the benefit of using an MVC pattern, after all.



Observant readers may have noticed that we have a new import: dart:collection. As the name suggests, this library adds lots of nifty collection-related code. Since we are writing an MVC collection, that will come in quite handy, starting with the class definition:

```
class ComicsCollection extends IterableBase {

List models = [];
Iterator get iterator => models.iterator;

  // ...
}
```

> **Tip**
>
> Note: When writing scripts or initial implementations of libraries, we can easily forgo typing information. When writing libraries that we hope others will use, it is a must. To be clear, it is possible to write reusable code without the type information, but it is tantamount to being a bad Dart citizen.

Extending IterableBase and defining iterator are a cheap way to get collection-like behavior in a class. We will discuss Dart's object-oriented approach in Chapter 7, *Classes and Objects*, on page ?, but the intent of this class is already fairly self-evident. Our ComicsCollection is going to extend another class that knows different ways to iterate over a collection of things. In this case, IterableBase can iterate with methods like forEach(), map(), reduce(), and many others. All that IterableBase needs is an iterator, which our list of models provides via the List class.

"Getter" methods like iterator are methods that are invoked without the trailing parentheses. Instead of invoking it as collection.iterator(), it would be simply collection.iterator. Getters, and their counterpart setters, can be quite useful as you'll see later.

Our ComicsCollection knows how to collect model objects, but we are still missing two requirements for an MVC collection. The first is the ability to communicate when changes occur (the views need a way to know when to update). Also, this would not be a REST-like collection without some create, read, update, and delete (CRUD) methods.

At this stage in our MVC solution, our collection will communicate change via a simple callback method. Have no fear, we will lose the callbacks in Chapter 8, *Events and Streams*, on page ?. But, for now, our class's constructor will accept a callback function, assigning it to the local onChange variable:

```
class ComicsCollection extends IterableBase {
  // ...

var onChange = (){};

// Constructor method
ComicsCollection({this.onChange});

  // ...
}
```

Here, we already see a glimpse of the extraordinary power of Dart constructors in the ComicsCollection() constructor methods. First, constructors are easy—they are a method with the same name as the class. Second, they do not require

a method body. Lastly, they take the optional arguments from Chapter 3, *Functional Programming in Dart, on page ?* a step further—prefixed with this, optional arguments are assigned directly to object instance variables.

In other words, instantiating the object as new ComicsCollection(onChange: (){ print('Awesome sauce here!) }); will print "Awesome sauce here!" to the console whenever changes are made to the collection. That is an amazing lines-of-code savings over not only JavaScript, but also over most established languages. It gets even better, but we'll leave that for *Classes and Objects*. For now, let's get back to building our ComicsCollection.

Now that we have our collection behaving like a collection and capable of communicating change, let's make it behave like an Ajax-backed object. For discussion purposes, we will not go into complete CRUD but will focus on fetching the objects from the back-end data store, creating new objects in the data store and deleting them.

We already know from *Project: Your First Dart Application* how to fetch data over Ajax in Dart. In this case, when the data has loaded, we call the private _handleOnLoad() method.

```
void fetch() {
  var req = new HttpRequest();
  req.onLoad.listen((event) {
    var list = JSON.decode(req.responseText);
    _handleOnLoad(list);
  });
  req.open('get', url);
  req.send();
}
```

Instead of populating a UI element as we did in our first application, we need to behave in a more frameworky fashion. That is, we build the internal collection and notify interested parties when changes to the collection occur.

```
_handleOnLoad(list) {
  list.forEach((attrs) {
    var new_model = new ComicBook(attrs, collection: this);
    models.add(new_model);
    onChange();
  });
}
```

For each set of model attributes, we create a new model object, set the model's collection property to our current collection, and add the model to the collection's models list. Once that's complete, we invoke the onChange() callback method, telling interested parties that a change has occurred.

The model does not strictly need to know about the collection (in fact, it should not communicate directly with the collection). We assign it here so that the model can reuse the collection's URL for finding, creating, and updating back-end objects. The model will communicate with the collection via callbacks just as we have done with `onChange()` here.

We still need the ability to create new comic books in our collection. Most of the heavy lifting will be done by the `ComicBook` model. In the collection we create a new model and tell it to save itself. Upon successful save, we add it to the internal list of comic books and notify interested parties via `onChange()`.

```
class ComicsCollection extends IterableBase {
  // ...
  create(attrs) {
    var new_model = new ComicBook(attrs, collection: this);
    new_model.save((event) {
      models.add(new_model);
      onChange();
    });
  }
  // ...
}
```

Of course, we have not even introduced the model base class yet, so let's get that out of the way next.