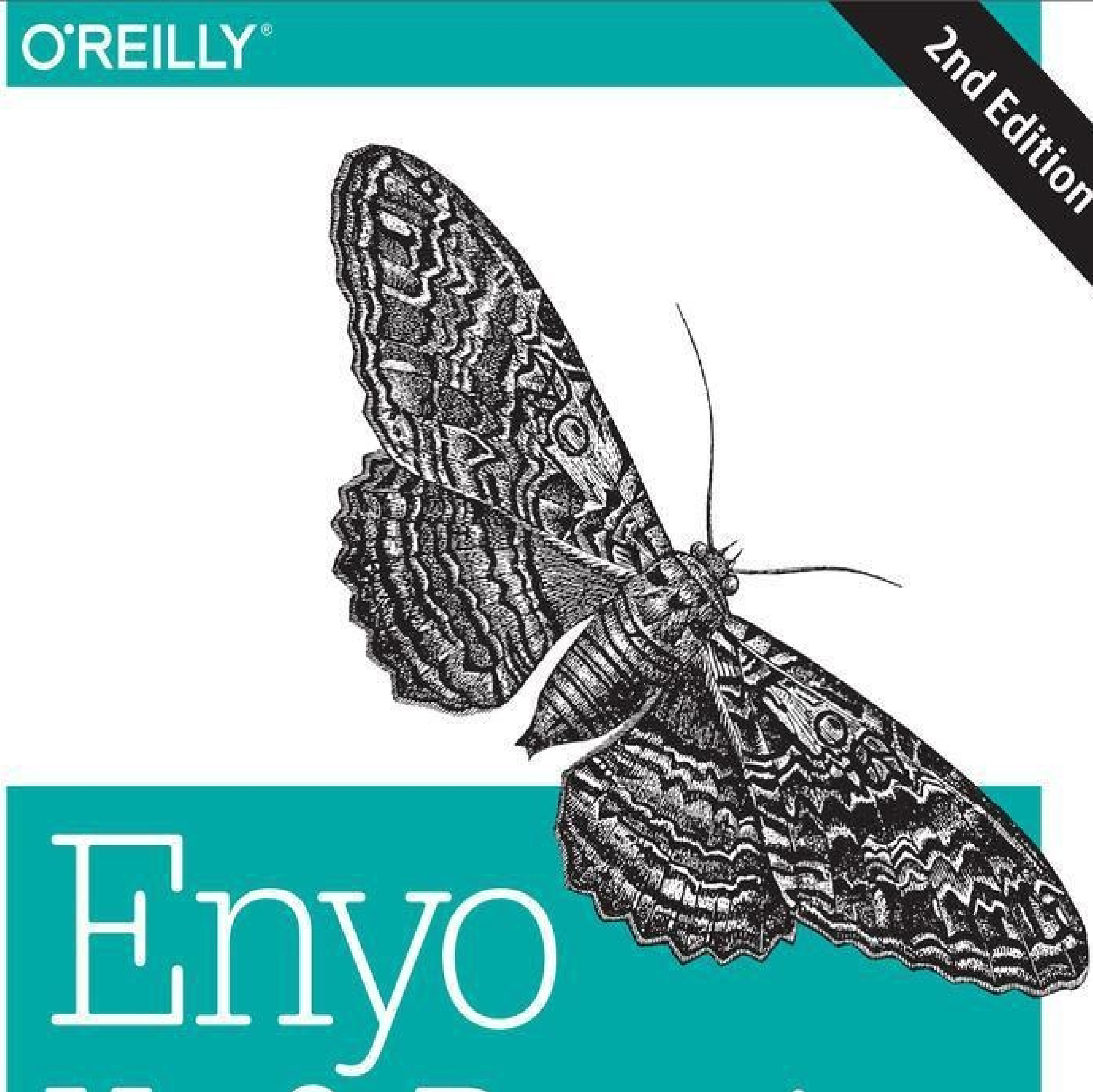


Enyo Up & Running

BUILD NATIVE-QUALITY CROSS-PLATFORM JAVASCRIPT APPS

Roy Sutton

www.allitebooks.com

Enyo: Up and Running

Roy Sutton

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

www.allitebooks.com

Preface

HTML5 technologies hold the promise of providing compelling user experiences through the web browser. The Web has evolved as a platform for delivering content to users regardless of the operating system their computers (or smartphones, tablets, and smart TVs) use. As users spend more time on the Web, they not only expect to receive content but also perform the actions of their daily lives. The Web is evolving from static pages to true web applications.

Enyo is a JavaScript framework designed to help developers create compelling interactive web applications (or apps). What makes Enyo special? Why should you be interested in it? I'll try to tackle those questions and, along the way, help you get productive in Enyo.

www.allitebooks.com

Where Did Enyo Come From?

Enyo grew out of the need to create applications for the HP TouchPad tablet. It was designed to be an easy-to-learn, high-performance framework that provided a pleasing and consistent user interface. As Enyo grew, HP realized that the technologies could be applied not only to tablets but also to the larger screens of desktops and the smaller screens of smartphones.

On January 25, 2012, HP announced they were going to release Enyo as an open source project under the Apache 2.0 license. Development moved to GitHub and the broader JavaScript community was invited to participate. Since that time, Enyo has matured and now offers robust tools for developing web apps on a wide variety of platforms. In March of 2013, LG Electronics acquired the webOS group from HP and the core Enyo team focused on adapting the framework for creating smart TV applications.

www.allitebooks.com

Core Beliefs

The Enyo team believes very strongly in the power of the open Web. To that end, Enyo embraces the following concepts:

- Enyo and its code are free to use, always.
- Enyo is open source — development takes place in the open and the community is encouraged to participate.
- Enyo is truly cross-platform — you should not have to choose between mobile and desktop, or between Chrome and Internet Explorer.
- Enyo is extensible.
- Enyo is built to manage complexity — Enyo promotes code reuse and encapsulation.
- Enyo is lightweight and fast — Enyo is optimized for mobile and its core is small.

www.allitebooks.com

What’s Enyo Good For?

Enyo is designed for creating apps. While a discussion of exactly what an app is could probably fill a book this size, when I say “apps” I’m referring to an interactive application that runs in a web browser (even if the browser itself may be transparent to the user).

This is to say Enyo is not designed for creating web pages. Enyo apps run in the browser and not on the server. This doesn’t mean Enyo cannot interact with data stored on servers; it certainly can. And it doesn’t mean that Enyo can’t be served to the browser by a web server; it can.

www.allitebooks.com

Who Is This Book For?

This book is written for web developers looking to learn new ways of developing applications or for programmers who are interested in learning web app design. It is not intended as an “introduction to programming” course. While designing with Enyo is easy, I expect some familiarity with HTML, CSS, or JavaScript.

www.allitebooks.com

Minimum Requirements

The absolute minimum requirement for working through the book is a web browser that is compatible with Enyo and access to the jsFiddle website. To get the most out of the book, I recommend a PC (Mac, Windows, or Linux), a code editor, and a modern web browser. A web server, such as a local installation of Apache or a hosting account, can be helpful for testing. Git and Node.js round out the tools needed for the full experience.

Information on setting up your environment to develop Enyo applications can be found in Appendix A. This book was based off Enyo version 2.5.1, though it should apply to later versions.

www.allitebooks.com

Typographic Conventions

The following conventions are used in this book:

Italic

Ital indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

`CW` is used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

`CWB` shows commands or other text that should be typed literally by the user.

`Constant width italic`

`CWI` shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This icon precedes a link to runnable code samples on jsFiddle.

TIP

This icon precedes a tip, suggestion, or note.

WARNING

This icon precedes a warning or clarification of a confusing point.

www.allitebooks.com

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Enyo: Up and Running*, 2nd Edition, by Roy Sutton (O'Reilly). Copyright 2015 Roy Sutton, 978-1-491-92120-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

www.allitebooks.com

Safari® Books Online

NOTE

> Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/enyo-upandrunning_2e.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

First and foremost I would like to thank my wife Connie Elliott and son Ian for their help and forbearance as I prepared the book you now hold (even if only virtually). Next, I would like to thank the team at O’Reilly, in particular Simon St. Laurent and Megan Blanchette, who really made the process of getting this book finished as painless as possible, Kara Ebrahim, who slew typos and wrangled words into their proper places, and Kristen Brown, who made the second edition painless. V. L Elliott played a special role in helping bring structure to the original thoughts for this book. Further, thanks to those readers (technical and otherwise) who helped reduce the number of errors herein. Special thanks to Ben Combee and Arthur Thornton for their efforts in spotting technical errors in the first edition. Many thanks to Jim Tang for keeping the official Enyo documentation in order and for spotting errors in the second edition. I’d also like to acknowledge the dozens of individuals at HP and LG (from Silicon Valley to France, Korea, and India) who have had a hand in developing, testing, supporting and evangelizing Enyo, and our numerous contributors from the broader Enyo community. There are far too many people to name here, but without their hard work and passion, Enyo would not be what it is today. Finally, special mention goes to the creators of Enyo, Scott Miles and Steve Orvell, without whom this book wouldn’t exist.

Content Updates

January 8, 2015

A lot has happened with Enyo since the first edition of the book. Most notable are the addition of data binding to the core (Bindings and Observers) and the new Moonstone smart TV UI library (Moonstone Controls). Other changes include the deprecation of published properties and the introduction of the new `set()` and `get()` methods (Properties), the switch from g11n to iLib for internationalization (Going Global), and the new `Application` object (Application). If you read the previous edition of this book, you will want to focus on the new chapter (Chapter 5) and then check individual chapters for changes.

Chapter 1. Light It Up

One of the best ways to get familiar with Enyo is to get a taste for what an Enyo app looks like. We’re going to do a little virtual time travel and fast-forward to a point just after you finish reading this book. We’re going to imagine you work for a software company that produces apps for customers.

A New Project

Your boss just came and told you that he needs a light implemented in JavaScript right away. He tells you that your company's best client needs to be able to embed a light app on their web page and it must work cross-platform. Fortunately, you've just finished reading this book and are excited to use Enyo on a project.

You decide to make a nice yellow circle and draw that on the screen:

```
enyo.ready(function() {
    enyo.kind({
        name: 'Light',
        style: 'width: 50px; height: 50px; border-radius: 50%;' +
            'background: yellow;'
    });

    new enyo.Application({ name: 'app', view: 'Light' });
});
```

TIP

Try it out: jsFiddle.

With Enyo, you don't (usually) have to worry about the HTML that makes up your app. Enyo creates it all for you. In this case, you've created a new kind (Enyo's building blocks are called kinds, you recall) called `Light` and you used a little CSS magic you found on the Web to draw a circle without the use of images or the canvas.

While using Enyo's `Application` component, you placed the new kind into the page's `body` element, causing Enyo to create the HTML. You recall that the `enyo.ready()` method executes code when the framework is fully loaded. You inspect the HTML for the circle using your favorite browser's debugging tool and see that Enyo created a `div` element for you and applied the style you supplied. Not bad for a few minutes' work.

Improvements

Now that you're feeling good about what you did, you check in the first version of the code to the company's source code management system. You know from past experience that sales will probably need the light in more colors than yellow. So, you decide to use Enyo's `property` feature to set the color when the kind is created:

```
enyo.kind({
    name: 'Light',
    color: 'yellow',
    style: 'width: 50px; height: 50px; border-radius: 50%;',
    create: function() {
        this.inherited(arguments);
        this.colorChanged();
    },
    colorChanged: function(oldValue) {
        this.applyStyle('background-color', this.color);
    }
});
```

TIP

Try it out: jsFiddle.

TIP

This code (and the following samples) does not include the `enyo.ready()` line and the instantiation of the `Application` kind, but you'll still need it. We'll only focus on the areas that changed.

You note that you've added a default color for the light, in case none is defined, and you've added a function that Enyo will call if anyone updates the light color after the kind has been created. You had to add some code to the `create()` function that Enyo calls on all components so that you can set the initial color. First, you test that you can set the color at create time by passing in a JavaScript object with the color value you want:

```
new enyo.Application({ name: 'app', view: { kind: 'Light', color: 'green' } });
```

Looks like that works as expected. Now you can test that you can set the color after creation:

```
var app = new enyo.Application({ name: 'app', view: Light });
app.set('view.color', 'blue');
```

TIP

Try it out: jsFiddle.

You remember that when you use `set()`, Enyo will automatically call the `colorChanged()` method for you when the color changes. Looks like that works well, too.

You check in the latest version of the code and shoot an e-mail off to your boss. Your latest change added a bit more code but you know that you'll be able to use that light component again and again, regardless of what color sales promises.

Curveball

Not long after you send off the e-mail, the phone rings. Your boss explains that sales finally let him know that the light they needed was actually a traffic light, with red on the top, yellow in the middle, and green on the bottom.

Fortunately, you've done the hard work. Getting the traffic light done should be a breeze now. You recall that Enyo supports composition, allowing you to make a new kind by combining together other kinds. Diving back into the code, you create a new `TrafficLight` kind:

```
enyo.kind({
    name: 'TrafficLight',
    components: [
        { name: 'stop', kind: 'Light', color: 'red' },
        { name: 'slow', kind: 'Light', color: 'yellow' },
        { name: 'go', kind: 'Light', color: 'green' }
    ]
});
```

TIP

Try it out: jsFiddle.

Not bad, if you do say so yourself. You reused the `Light` kind you created and you didn't have to copy all that code over and over. You push your changes up, shoot another e-mail off to your boss and wait for the phone to ring again.

QA on the Line

The next call is not, surprisingly, from your boss, but from the QA department. They did some testing with the lights and found that they don't turn off. They mention something about the specs for the light, saying that tapping the light should toggle it on and off. While wondering how they managed to get ahold of specs you'd never seen, you begin thinking about how you'll implement that. You quickly hang up after asking for a copy of the specs.

You remember that Enyo has an event system that allows you to respond to various events that occur. You can add a new property for the power state of the light and you can toggle it when you receive a tap event (an event you know is optimized to perform well on mobile devices with touch events). After thinking some more about the problem, you realize you don't really want to change your existing light kind. You remember that Enyo supports inheritance, allowing you to create a new light that has all the same behaviors as your existing light, plus the new behaviors you need:

```
enyo.kind({
    name: 'PoweredLight',
    kind: 'Light',
    powered: true,
    handlers: {
        'ontap': 'tapped'
    },
    create: function() {
        this.inherited(arguments);
        this.poweredChanged();
    },
    tapped: function(sender, event) {
        this.set('powered', !this.get('powered'));
    },
    poweredChanged: function(oldValue) {
        this.applyStyle('opacity', this.powered ? '1' : '0.2');
    }
});
```

TIP

Try it out: jsFiddle.

You made use of the `handlers` block to add the events you want to listen for and specified the name of the method you wanted to call. You recall that in Enyo, you use the name of the event instead of the event itself because Enyo will automatically bind the methods to each instance of your kind so it can access the methods and data of your kind's instance.

In your tap handler, you used the partner to the `set()` method, `get()`, to retrieve the current value of the `powered` property and toggle it. In the `poweredChanged()` function, you apply a little opacity to the light to give it a nice look when it's powered off (you can read a local property directly without `get()`). You update the `TrafficLight` kind, give it a quick test in the browser, and verify that everything looks good.

The E-mail

Just after you commit the latest changes, you receive a copy of the specs from QA. Looks like you've got everything covered except for a logging feature. The specs call for a log to be maintained of which light was activated or deactivated and the time of the event. Events, huh? Sounds like it's time to revisit Enyo events. You recall from your training that Enyo allows kinds to create their own events, to which other kinds can subscribe.

You quickly add a new event to the `PoweredLight` kind called `onStateChanged`. You know that Enyo automatically creates a method called `doStateChanged()` that you can call to send the event to a subscriber. You quickly add the relevant code:

```
enyo.kind({
    name: 'PoweredLight',
    kind: 'Light',
    powered: true,
    events: {
        'onStateChanged' : ''
    },
    handlers: {
        'ontap': 'tapped'
    },
    create: function() {
        this.inherited(arguments);
        this.poweredChanged();
    },
    tapped: function(sender, event) {
        this.set('powered', !this.get('powered'));
    },
    poweredChanged: function(oldValue) {
        this.applyStyle('opacity', this.powered ? '1' : '0.2');
        this.doStateChanged({ powered : this.powered });
    }
});
```

Now you just need to subscribe to the event in the `TrafficLight` kind. You could, of course, subscribe to `onStateChanged` in each `Light` definition, but you remember that the `handlers` block lets you subscribe to events a kind receives regardless of which child originates them. You know you can use the `sender` parameter to check to see which light sent the event and you can use the `event` parameter to access the object sent by the light:

```
enyo.kind({
    name: 'TrafficLight',
    handlers: {
        'onStateChanged': 'logStateChanged'
    },
    components: [
        { name: 'stop', kind: 'PoweredLight', color: 'red' },
        { name: 'slow', kind: 'PoweredLight', color: 'yellow' },
        { name: 'go', kind: 'PoweredLight', color: 'green' }
    ],
    logStateChanged: function(sender, event) {
        enyo.log(sender.name + ' powered ' + (event.powered ? 'on' : 'off')
            + ' at ' + new Date());
    }
});
```

TIP

Try it out: jsFiddle.

A quick logging function and a `handlers` block later and things are starting to look finished. After the code has been checked in and QA has signed off, you can relax and start planning that vacation — as if that will happen.

www.allitebooks.com

Summary

We’ve just worked through a simple Enyo application and explored several of the concepts that make using Enyo productive. We saw how easy it is to quickly prototype an application and how Enyo kept the code maintainable and potentially reusable. With this foundation, we’ll be able to explore the deeper concepts of Enyo in the coming chapters.

Chapter 2. Core Concepts

Introduction

In this chapter, we’ll cover the core concepts of Enyo that we only touched on in the last chapter. You will be able to write powerful apps after absorbing the information in just this chapter. We’ll go over the concepts one by one and illustrate each with code you can run in your browser.

One of the driving ideas behind Enyo is that you can combine simple pieces to create more complex ones. Enyo introduces four concepts to assist you: kinds, encapsulation, components, and layout. We’ll cover components and layout more thoroughly in Chapter 3 and Chapter 4, respectively.

Kinds

Enyo is an object-oriented framework. It is true that every JavaScript application regardless of framework (or lack thereof) contains objects. However, Enyo's core features provide a layer on top of JavaScript that makes it easier to express object-oriented concepts such as inheritance and encapsulation.

In Enyo, kinds are the building blocks that make up apps. The widgets that appear on screen are instances of kinds, as are the objects that perform *Ajax* requests. Kinds are not strictly for making visual components. Basically, kinds provide a template from which the actual objects that make up your app are generated.

Be Kind

One of the simplest possible declarations for a kind is:

```
enyo.kind({ name: 'MyKind' });
```

NAMES

Kinds don't even need names. Enyo will automatically assign unique names, though you won't know what they are. Anonymous kinds are often used in Enyo apps. You saw one in Chapter 1 when the color of the light was set to green in the view declaration.

Top-level kinds (those declared outside of other kinds) automatically get a global object created with that name (for example, `Light` in the previous chapter). It is possible to put kinds into a *namespace* by separating name parts with periods. For example, using `name: myApp.Light` will result in a `myApp` object with a `Light` member. Namespaces provide a good mechanism for preventing naming conflicts with your apps, particularly when using reusable components.

As a convention, we use uppercase names for kind definitions and lowercase names for instances of kinds (those kinds declared in the `components` block).

`enyo.kind()` is a "factory" for creating new kinds. In this case, we get a new object that inherits from the Enyo control kind, `enyo.Control`. `Control` is the base component for objects that will render when placed on a web page.

When creating kinds, you pass in an object that defines the starting state of the kind as well as any methods it will need. For example, control kinds have a content property:

```
enyo.kind({ name: 'MyKind', content: 'Hello World!' });
```

As you saw in Chapter 1, when rendered onto a page this code will create a `div` tag with the content placed in it. To render this into a body on a web page, you specify it as the view of an `Application`.

We can add behaviors to our kind by adding methods (for example, the tap handling method we added to the `Light` kind). As you may recall, we referenced the method name in the `handlers` block using a string. We use strings so Enyo can bind our methods as kinds are created.

Encapsulation

Encapsulation is a fancy computer science term that refers to restricting outside objects' access to an object's internal features through providing an interface for interacting with the data contained in the object. JavaScript does not have very many ways to prohibit access to an object's data and methods from outside, so Enyo promotes encapsulation by giving programmers various tools and conventions.

By convention, Enyo kinds should have no dependencies on their parent or sibling kinds and they should not rely on implementation details of their children. While it is certainly possible to create Enyo kinds that violate these rules, Enyo provides several mechanisms to make that unnecessary. Those mechanisms include properties and events.

By being aware of encapsulation, Enyo programmers can tap in to the benefits of code reuse, easy testing, and drop-in components.

Properties

Kinds can declare properties (for example, the `color` and `powered` properties from Chapter 1). The property system allows for some very powerful features, such as two-way data binding and notification for changes to values. We'll discuss the basic features of properties first and then dive into the more advanced features of bindings and observers.

Basic Properties

Properties are accessed using the `get()` and `set()` methods defined on all kinds. In addition, there is a mechanism for tracking changes to properties. Properties don't need to even be declared on a kind, though you should at least document their presence so that users of your kinds (including yourself) will know (remember) that they exist.

```
enyo.kind({
    name: 'MyKind',
    myValue: 3
});
```

As you can see, you also specify a default value for a property. Within `MyKind` you can read the property directly using `this.myValue`. When you are accessing `myValue` externally (e.g., from a parent control), you should use the `get()` or `set()` methods. Whenever the value is modified using the setter, Enyo will automatically call a "changed" method. In this case, the changed method is `myValueChanged()`. When called, the changed method will be passed the previous value of the property as an argument.

WARNING

The `set()` method does not call the changed method if the value to be set is the same as the current value. You can, however, override this behavior by passing a *truthy* value as a third argument to `set()`.

If you look back to our earlier discussion on kinds you may have noticed that we passed in some values for properties when we were declaring our kinds. Those values set the initial contents of those properties. Enyo does *not* call the changed method during construction. If you have special processing that needs to occur, you should call the changed method directly within `create()`:

```
enyo.kind({
    name: 'MyKind',
    myValue: 3,
    create: function() {
```

```
        this.inherited(arguments);
        this.myValueChanged();
    },
    myValueChanged: function(oldValue) {
        // Some processing
    }
});
```

If you want to tie the value of a property to another property within your kind (such as the `content` of a control), you can use a binding, which is triggered during construction. We'll cover bindings in the next section.

WARNING

> You should only specify simple values (strings, numbers, booleans, etc.) for the default values of properties and member variables. Using arrays and objects can lead to strange problems. See Instance Constructors for a method to initialize complex values.

Bindings and Observers

The `set()` method makes it possible to set up bindings that tie the value of two properties together. You can even monitor properties on other kinds. Bindings really stand out when it comes to associating data with the contents of controls. We'll cover the use of bindings with data-driven applications in Chapter 5.

At their simplest, bindings create a one-way association between two properties. The following example creates a property called `copy` that will be updated any time the value of `original` changes:

```
enyo.kind({
    name: 'ShadowKind',
    original: 3,
    copy: null,
    bindings: [
        { from: 'original', to: 'copy' }
    ]
});
```

We could have accomplished the same thing using an `originalChanged()` method — however, it would have taken more code and we would not be able to monitor the value of properties declared on components declared within our kind. Further, we would have to set up a `copyChanged()` method if we wanted to create a two-way connection. Using bindings, we can do this with one simple change:

```
enyo.kind({
    name: 'ShadowKind',
    original: 3,
    copy: null,
    bindings: [
        { from: 'original', to: 'copy', oneWay: false }
    ]
});
```

Bindings have even more power, including the ability to transform values when they are triggered. We'll cover transformations in Chapter 5.

Observers, like bindings, monitor properties for changes. When an observer detects a value change, it invokes the method specified in the observer declaration. We can rewrite the earlier changed example as follows:

```
enyo.kind({
    name: 'MyKind',
    myValue: 3,
    observers: [
        { path: 'myValue', method: 'myValueUpdated' }
```

```
        ],
        myValueUpdated: function(oldValue, newValue) {
            // Some processing
        }
    });
```

Note that we did not need to override `create()` to invoke `myValueUpdated()` because bindings and observers will be triggered during initialization.

TIP

> With bindings and observers, the path to a property is a string and is relative to `this`. Binding to a nested component's property (see Chapter 3) can be accomplished like so: *`$.component.value`*.

Events

If properties provide a way for parent kinds to communicate with their children, then events provide a way for kinds to communicate with their parents. Enyo events give kinds a way to be notified when something they're interested in occurs. Events can include data relevant to the event. Events are declared like this:

```
enyo.kind({
    name: 'Eventer',
    handlers: { ontap: 'myTap' },
    events: { onMyEvent: '' },
    content: 'Click for the answer',
    myTap: function() {
        this.doMyEvent({ answer: 42 });
    }
});
```

Event names are always prefixed with "on" and are always invoked by calling a method whose name is prefixed with "do". Enyo creates the "do" helper method for us and it takes care of checking that the event has been subscribed to. The first parameter passed to the "do" method, if present, is passed to the subscriber. Any data to be passed with the event must be wrapped in an object.

Subscribing is easy:

```
enyo.kind({
    name: 'Subscriber',
    components: [{ kind: 'Eventer', onMyEvent: 'answered' }],
    answered: function(sender, event) {
        alert('The answer is: ' + event.answer);
        return(true);
    }
});
```

TIP

> Try it out: jsFiddle.

The `sender` parameter is the kind that last bubbled the event (which may be different from the kind that originated the event). The `event` parameter contains the data that was sent from the event. The object will always have at least one member, `originator`, which is the Enyo component that started the event.

When responding to an event, you should return a *truthy* value to indicate that the event has been handled. Otherwise, Enyo will keep searching through the sender's ancestors for other event handlers. If you need to prevent the default action for DOM events, use `event.preventDefault()`.

TIP

> Enyo kinds cannot subscribe to their own events, including DOM events, using the `onXxx` syntax. If you need to

subscribe to an event that originates on the kind, you can use the `handlers` block, as we did for the previous `tap` event.

Advanced Events

The standard events described previously are bubbling events, meaning that they only go up the app hierarchy from the object that originated them through the object's parent. Sometimes it's necessary to send events out to other objects, regardless of where they are located. While it might be possible to send an event up to a shared common parent and then call back down to the target, this is far from clean. Enyo provides a method called signals to handle this circumstance.

To send a signal, call the `send()` method on the `enyo.Signals` object. To subscribe to a signal, include a `Signals` kind in your `components` block and subscribe to the signal you want to listen to in the kind declaration. The following example shows how to use signals:

```
enyo.kind({
    name: 'Signaller',
    components: [
        { kind: 'Button', content: 'Click', ontap: 'sendit' }
    ],
    sendit: function() {
        enyo.Signals.send('onButtonSignal');
    }
});

enyo.kind({
    name: 'Receiver',
    components: [
        { name: 'display', content: 'Waiting…' },
        { kind: 'Signals', onButtonSignal: 'update' }
    ],
    update: function(sender, event) {
        this.set('$.display.content', 'Got it!');
    }
});
```

TIP

Try it out: jsFiddle.

Like regular events, signals have names prefixed with "on". Unlike events, signals are broadcast to all subscribers. You cannot prevent other subscribers from receiving signals by passing back a truthy return from the signal handler. Multiple signals can be subscribed to using a single `Signals` instance.

Signals should be used sparingly. If you begin to rely on signals for passing information back and forth between objects, you run the risk of breaking the encapsulation Enyo tries to help you reinforce. It might be better to use a shared model to hold the data. We'll discuss models in Chapter 5.

TIP

Enyo uses the signals mechanism for processing DOM events that do not target a specific control, such as `onbeforeunload` and `onkeypress`.

Final Thoughts on Encapsulation

While properties and events go a long way towards helping you create robust applications, they are not always enough. Most kinds will have methods they need to expose (an API, if you will) and methods they wish to keep private. While Enyo does not have any mechanisms to enforce that separation, code comments and documentation can serve to

help other users of your kinds understand what is and isn’t available to outside kinds.

Inheritance

Enyo provides an easy method for deriving new kinds from existing kinds. This process is called *inheritance*. When you derive a kind from an existing kind, it inherits the properties, events, and methods from that existing kind. All kinds inherit from at least one other kind. The ultimate ancestor for nearly all Enyo kinds is `enyo.Object`. Usually, however, kinds derive from `enyo.Component` or `enyo.Control`.

To specify the parent kind, set the `kind` property during creation:

```
enyo.kind({
    name: "InheritedKind",
    kind: "enyo.Control"
});
```

As mentioned, if you don't specify the kind, Enyo will automatically determine the kind for you. In most cases, this will be `Control`. An example of an instance where Enyo will pick a different kind is when creating menu items for an Onyx `Menu` kind. By default, components created within a `Menu` will be of kind `MenuItem`. If you want to specify the kind for child components in your own components, set the `defaultKind` property.

If you override a method on a derived kind and wish to call the same named method on the parent, use the `inherited()` method. You may recall that we did this for the `create()` method in the `Light` kind. You must always pass `arguments` as the parameter to the `inherited()` method.

www.allitebooks.com

Advanced Kinds

Enyo provides two additional features for declaring kinds, which are most often used when creating reusable kinds: instance constructors and statics.

Instance Constructors

For some kinds, initialization must take place when an instance of that kind is created. One particular use case is defining array properties. If you were to declare an array member in a kind definition then all instances would be initialized with the last value set to the array. This is unlikely to be the behavior you wanted. When declaring a constructor, be sure to call the `inherited()` method so that any parent objects can perform their initialization as well. The following is a sample constructor:

```
constructor: function() {
    this.instanceArray = [];
    this.inherited(arguments);
}
```

TIP

It’s worth noting that `constructor()` is available for all kinds. The `create()` method used in many examples is only available for descendants of `enyo.Component`.

Statics

Enyo supports declaring methods that are defined on the kind constructor. These methods are accessed by the kind name rather than from a particular instance of the kind. Statics are often used for utility methods that do not require an instance and for properties that should be shared among all instances, such as a count of the number of instances created. The following kind implements an instance counter and shows off both statics and constructors:

```
enyo.kind({
    name: ‘InstanceCounter’,
    constructor: function() {
        InstanceCounter.count += 1;
        this.inherited(arguments);
    },
    statics: {
        count: 0,
        currentCount: function() {
            return(this.count);
        }
    }
});
```

TIP

Try it out: jsFiddle.

STRUCTURE OF A KIND

It’s good to be consistent when declaring kinds. It helps you and others who may need to read your code later to know where to look for important information about a kind. In general, kinds should be declared in the following order:

- Name of the kind
- Parent kind
- Properties, events, and handlers

- Kind variables
- Classes and styles
- Components
- Bindings and observers
- Public methods
- Protected and private methods
- Static members

Summary

We have now explored the core features of Enyo. You should now understand the object oriented features that allow for creating robust and reliable apps. We'll build upon this knowledge in the next chapters by exploring the additional libraries and features that make up the Enyo framework.

Chapter 3. Components, Controls, and Other Objects

In Chapter 2, we covered kinds and inheritance. It should come as no surprise that Enyo makes good use of those features by providing a rich hierarchy of kinds you can use and build upon in your apps. In this chapter, we'll focus on two important kinds that Enyo provides: `Component` and `Control`. We'll also touch on some of the other kinds that you'll need to flesh out your apps.

Components

Components introduce one of the most-used features of Enyo apps: the ability to create kinds composed of other kinds. This ability to compose new components from other components is one of the key features that encapsulation allows. Most kinds you'll use, including the `Application` kind, will be based upon `Component` or one of its descendants.

Composition

Composition is a powerful feature that lets you focus on breaking down your app into discrete pieces and then combine those pieces together into a unified app. We used this feature in Chapter 1 when we built a traffic light out of three individual lights. Each descendant of `Component` has a `components` block that takes an array of component definitions.

For example, in Advanced Events, the `Receiver` kind has a `Control` named `display`:

```
enyo.kind({
    name: 'Receiver',
    components: [
        { name: 'display', content: 'Waiting…' },
        { kind: 'Signals', onButtonSignal: 'update' }
    ],
    …
```

Methods within `Receiver` can access `display` through `this.$.display`. For `set()` and `get()`, the path would be *`$.display.propertyName`*. Enyo stores references to all owned components in the `$` object. Components without explicit names (such as `Signals` in the previous example) are given unique names and added to `$`.

TIP

Every component declared within a kind will be owned by the kind, even if nested within multiple `components` blocks.

Many of the components that Enyo supplies were designed as containers for other components. We'll cover many of these kinds in Chapter 4. Some, such as `Button`, weren't intended to contain other components.

Component Methods

Components introduce `create()` and `destroy()` methods to assist with the component's lifecycle. These methods can be overridden by kinds that derive from `Component` to provide extra functionality, such as allocating and deallocating resources. We previously used the `create()` method when we wanted to invoke the `myValueChanged()` method. We can use this feature to create a simple heartbeat object:

```
enyo.kind({
    name: 'Heartbeat',
    events: {
        onBeat: ''
    },
    create: function() {
        this.inherited(arguments);
        this.timer = window.setInterval(enyo.bind(this, 'beat'), 1000);
    },
    destroy: function() {
        if(this.timer !== undefined) {
            window.clearInterval(this.timer);
        }
        this.inherited(arguments);
```

```
    },
    beat: function() {
        this.doBeat({});
    }
});
```

TIP

Try it out: jsFiddle.

We used the `destroy()` method to ensure that we cleaned up the timer we allocated in the `create()` method. You may also notice that we introduced a new method: `enyo.bind()`. In all our previous event handlers, Enyo made sure the context of the event handlers was set correctly. We'll need to take care of that ourselves when subscribing directly to non-Enyo events. For more information on binding and why it's necessary, please see this article on Binding Scope in JavaScript.

Dynamic Components

Up to this point we've always created components when a kind is being instantiated. It is also possible to create and destroy components dynamically. Components have a number of methods for interacting with their owned components. You can use `createComponent()` to create an individual component or create a number of components at once using `createComponents()`. To remove a component from its owner, call the component's `destroy()` method. It is also possible to destroy all owned components by calling `destroyComponents()`. The following example shows how to create a component dynamically:

```
enyo.kind({
    name: 'DynamicSample',
    components: [
        { kind: 'Button', content: 'Click', ontap: 'tapped' }
    ],
    tapped: function(sender, event) {
        this.createComponent({ content: 'A new component' });
        this.render();
        return true;
    }
});
```

TIP

Try it out: jsFiddle.

New controls are not rendered until requested. Call the `render()` method on a control to ensure that it and its children are rendered to the DOM.

Controls

`Control`, a descendant of `Component`, is the kind responsible for providing the user interface to your apps. A large part of what makes an app an app is the user interface. The Enyo core provides wrappers around the most basic type of controls found natively in browsers. The Onyx and Moonstone libraries expand upon those basic controls and provide the more specialized elements expected in modern apps.

Controls are important because they map to DOM nodes. They introduce a number of properties and methods that will be important for your apps. By default, controls render into a `div` element. You can override this behavior by specifying the `tag` property when defining the control (e.g., `tag: ‘span’`).

Core Controls

The core visual controls in Enyo are wrappers around the basic elements you can create directly with HTML. Of course, because they’re Enyo controls, they’ll have properties and events defined that make them easy to use within your apps. The core controls include: `Button`, `Checkbox`, `Image`, `Input`, `RichText`, `Select`, and `TextArea`.

The following code sample creates a simple app with several controls:

```
enyo.kind({
    name: ‘ControlSample’,
    components: [
        { kind: ‘Button’, content: ‘Click’, ontap: ‘tapped’ },
        { tag: ‘br’},
        { kind: ‘Checkbox’, checked: true, onchange: ‘changed’ },
        { tag: ‘br’},
        { kind: ‘Input’, placeholder: ‘Enter something’, onchange: ‘changed’ },
        { tag: ‘br’},
        { kind: ‘RichText’, value: ‘<i>Italics</i>’, onchange: ‘changed’ }
    ],
    tapped: function(sender, event) {
        // React to taps
    },
    changed: function(sender, event) {
        // React to changes
    }
});
```

TIP

Try it out: jsFiddle.

You will note that the controls themselves are unstyled, appearing with the browser’s default style. In Onyx Controls, we’ll see how the Onyx versions of these controls compare to the base versions. You may also note that some controls use the `content` property to set the content of the control. The exceptions to this rule are the text field controls: `Input`, `TextArea`, and `RichText`. These controls use the `value` property to get and set the text content. In these samples we use simple `br` tags to arrange the controls. In an actual app, you’ll want to use CSS or the layout controls described in the next chapter.

TIP

By default, most Enyo controls escape any HTML in their `content` or `value` properties. This is to prevent the inadvertent injection of JavaScript from unsafe sources. If you want to use HTML in the contents, set the `allowHtml` property to `true`. By default, `RichText` allows HTML content.

Onyx Controls

The Onyx library (an optional piece of Enyo) includes professionally designed widgets. These controls expand upon the basic set available in the Enyo core. The Onyx controls that correspond to the core controls use the same interface as those core controls:

```
enyo.kind({
    name: ‘ControlSample’,
    components: [
        { kind: ‘onyx.Button’, content: ‘Click’, ontap: ‘tapped’ },
        { tag: ‘br’},
        { kind: ‘onyx.Checkbox’, checked: true, onchange: ‘changed’ },
        { tag: ‘br’},
        { kind: ‘onyx.InputDecorator’, components: [
            { kind: ‘onyx.Input’, placeholder: ‘Enter something’,
              onchange: ‘changed’ }
        ]},
        { tag: ‘br’},
        { kind: ‘onyx.InputDecorator’, components: [
            { kind: ‘onyx.RichText’, value: ‘<i>Italics</i>’,
              onchange: ‘changed’ }
        ]}
    ],
    tapped: function(sender, event) {
        // React to taps
    },
    changed: function(sender, event) {
        // React to changes
    }
});
```

TIP

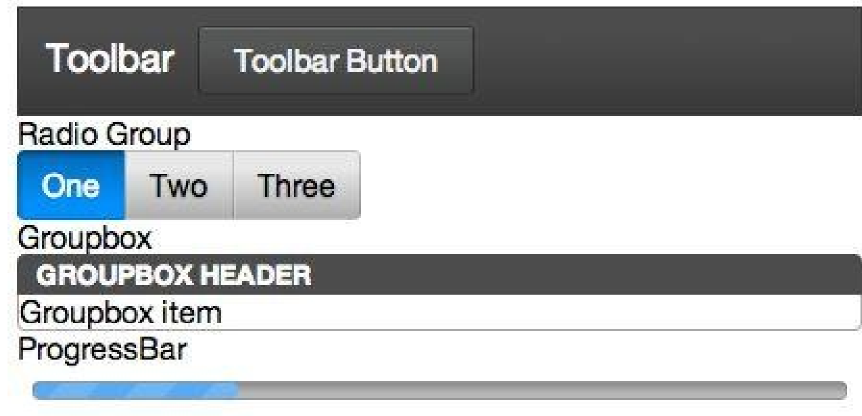
Try it out: jsFiddle.

As you can see, the Onyx widgets are much more pleasing to look at. With Onyx, we wrapped the text input controls in an `InputDecorator`. This is a control that allows for additional styling and should be used for all Onyx input controls.

The Onyx library also provides a number of new controls, including `Groupbox`, `ProgressBar`, `Toolbar` and `TimePicker`, among others. Here’s a sample of some of the new Onyx controls that shows off their important properties and events:

```
enyo.kind({
    name: ‘OnyxSample’,
    components: [
        { kind: ‘onyx.Toolbar’, components: [
            { content: ‘Toolbar’ },
            { kind: ‘onyx.Button’, content: ‘Toolbar Button’ }
        ]},
        { content: ‘Radio Group’ },
        { kind: ‘onyx.RadioGroup’, onActivate: ‘activated’, components: [
            { content: ‘One’, active: true },
```

```
            { content: 'Two' },
            { content: 'Three' }
        ]},
        { content: 'Groupbox' },
        { kind: 'onyx.Groupbox', components: [
            { kind: 'onyx.GroupboxHeader', content: 'Groupbox Header' },
            { content: 'Groupbox item' }
        ]},
        { content: 'ProgressBar' },
        { kind: 'onyx.ProgressBar', progress: 25 }
    ],
    activated: function(sender, event) {
        // React to radio button activation change
    }
});
```



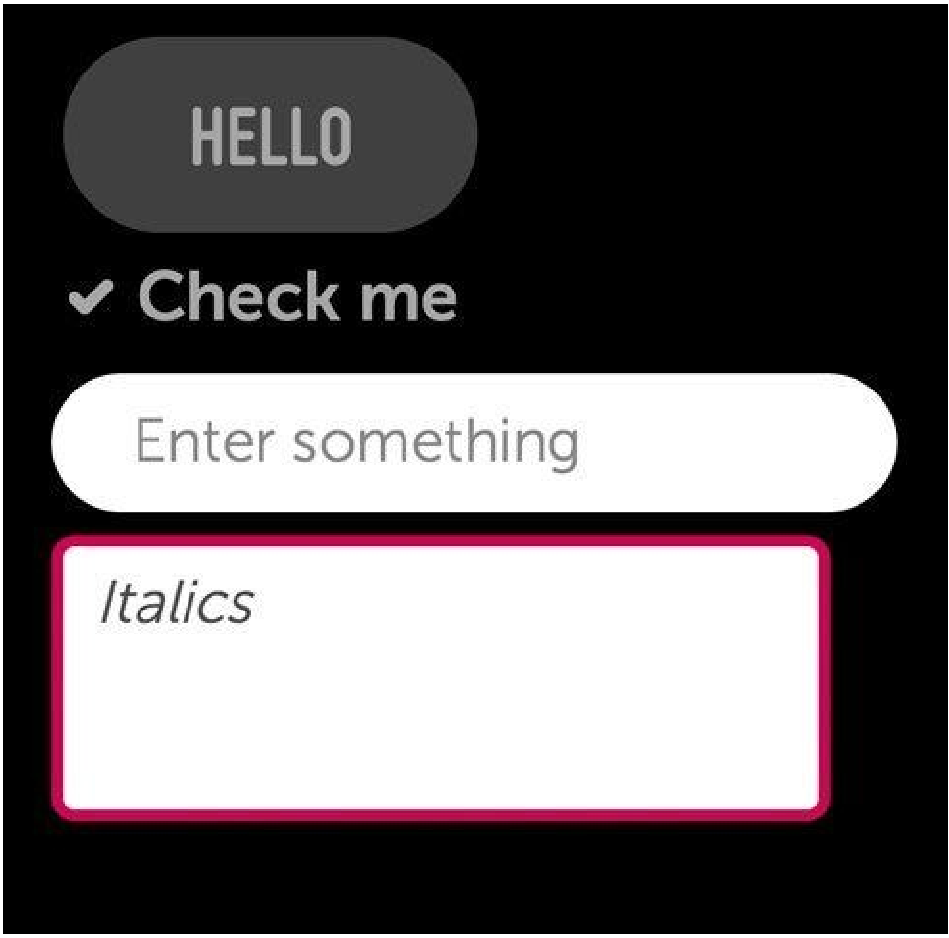
TIP

Try it out: jsFiddle.

Moonstone Controls

Another UI library available for use with Enyo is the Moonstone library. Moonstone was designed specifically for use on smart TVs. The use cases for smart TVs are very different from those for touch-based devices. Moonstone is a lot more styled than Onyx and includes many more components. Here are the basic controls we showed previously, rendered with the Moonstone styling:

```
enyo.kind({
    name: 'View',
    classes: 'moon',
    components: [
        { kind: 'moon.Button', content: 'Hello', ontap: 'tapped' },
        { kind: 'moon.CheckboxItem', checked: true, content: 'Check me',
            onchange: 'changed' },
        { kind: 'moon.InputDecorator', components: [
            { kind: 'moon.Input', placeholder: 'Enter something',
                onchange: 'changed' }
        ] },
        { kind: 'moon.InputDecorator', components: [
            { kind: 'moon.RichText', value: '<i>Italics</i>',
                onchange: 'changed' }
        ] }
    ],
    tapped: function(sender, event) {
        // React to taps
    },
    changed: function(sender, event) {
        // React to changes
    }
});
```



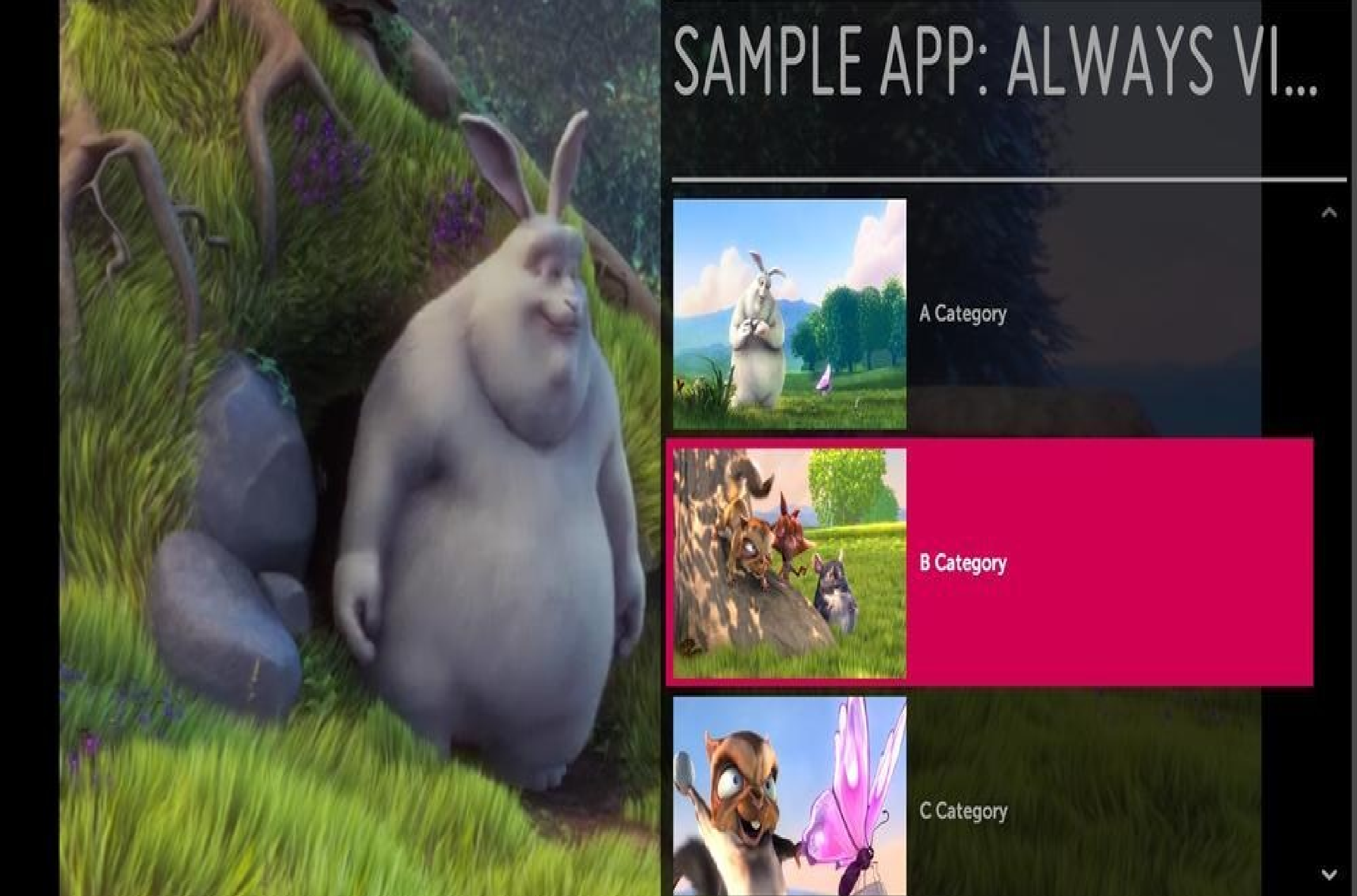
TIP

Try it out: jsFiddle.

Moonstone components are designed to work with the Enyo Spotlight library, which supports both five-way navigation (up, down, left, right, select) and cursor selection. If you mouse over the components, you will see them highlight to indicate they have focus. You can also switch to using the arrow keys on your keyboard to navigate among the components. Spotlight is a topic unto itself and we won't cover it here. If you're interested in learning more about it, see the Spotlight documentation.

There are too many Moonstone controls to get into here, so here's a screenshot of the Moonstone *Always Viewing VideoPlayer* sample:

www.allitebooks.com



For more information on Moonstone, see the “Building TV Applications” section of the Enyo docs site.

Methods and Properties

Controls have a number of methods and properties that focus on their special role in interacting with the DOM. These methods include `rendered()`, `hasNode()`, and a number of others for manipulating the DOM. The important properties include `classes` and `style`, which we’ll cover in Chapter 6.

The first method, `rendered()`, can be overridden to perform processing that only takes place when the DOM node associated with the control is available. By default, controls are not rendered into the DOM until they are required. In our samples, the `Application` kind takes care of rendering its view at startup. As always, be sure to call the `inherited()` method within `rendered()`.

The second important method, `hasNode()`, allows us to test whether the DOM node for the control exists and to retrieve it, if available. `hasNode()` will return `null` if no node is available. This is most useful when you are creating new controls that will need to manipulate the DOM, or when you want to wrap a widget from another UI library.

The following example shows a naive way to implement a scalable vector graphic (SVG) container object. The only purpose is to show off the `rendered()` and `hasNode()` methods:

```
enyo.kind({
    name: ‘Svg’,
    svg: ”,
    rendered: function() {
        this.inherited(arguments);
        this.svgChanged();
        // Can only call when we have a node
```

```
    },
    svgChanged: function() {
        var node = this.hasNode();
        if(node !== null) {
            node.innerHTML = '<embed src="' + this.svg +
                '" type="image/svg+xml" />';
        }
    }
});
```

TIP

Try it out: jsFiddle.

There are many additional methods and properties available on `Control`. Please see the API documentation for details.

Other Important Objects

Not all functionality in an app is provided by visible elements. For many apps, there is processing that must be done in the background. Enyo provides a number of objects that handle such processing. These objects include `Application`, `Router`, `Animator`, `Ajax`, and `JsonpRequest`. See Chapter 5 for more information on other important non-visual objects.

Application

One important component that we have not discussed yet is the `Application` component. `Application` is a type of controller that takes care of rendering the app. Each of the samples we've looked at uses `Application`. In general, an app will derive a new kind based on `Application` and specify the startup view.

The `view` property can contain either the name of a `Control` to render or a kind definition. When rendered, the `view` property will contain the instance of the view that was created. By default, the view is rendered as soon as the `Application` object is created. If you need to do some processing before the view is rendered, set the `renderOnStart` property to `false` and then call `render()` when ready.

The `Application` component is also a good place to keep track of data shared among various controls. All controls will have an `app` property that contains the instance of `Application`. This property can be used to bind to shared models and collections.

Router

Enyo has a `Router` component that handles routing. Routing, for our purposes, is the process of setting application state through the use of the URL. Specifically, `Router` uses the URL location hash to store information about the state of the app. The hash allows apps to maintain state between page loads and respond to the back button in the browser. Changing the location hash does not force a page reload.

The router works by monitoring and updating the location hash. Routers have routes, which are patterns for the data in the location hash and specify methods to be invoked. The following is an example router definition:

```
enyo.kind({
    name: 'Routing',
    components: [
        { name: 'router', kind: 'Router', routes: [
            { path: 'user/:userid', handler: 'routeUser' },
            { path: 'about', handler: 'routeAbout' },
            { path: 'home', handler: 'home', default: true }
        ]}
    ],
    routeUser: function(userID) {
        // Display user profile
    },
    routeAbout: function() {
        // Show about screen
    },
    home: function() {
        // Default route if no other path matches
    }
});
```

In the route declarations, any portions of the route that are prefixed with a colon (`:`) are converted into arguments to the handler method. By default, the router will trigger an

update when it is created. In our example, the location hash will be tested against the paths, and if none match, `home()` will be executed (because it's the default). The `trigger()` method is used to trigger the routing and, optionally, update the location hash. The following command updates the location hash and triggers the home action:

```
this.$.router.trigger({ location: 'home', change: true });
```

Routing is a powerful way to centralize app state changes and pairs very well with `Application`. For example, route handlers can set the active view, apply models and collections (see Chapter 5), or change the active panel (see Panels).

Animator

`Animator` is a component that provides for simple animations by sending periodic events over a specified duration. Each event sends a value that iterates over a range during the animation time. The following example shows how you could use `Animator` to change the width of a `div`:

```
enyo.kind({
    name: 'Expando',
    components: [
        { name: 'expander', content: 'Presto',
          style:
          'width: 100px; background-color: lightblue; text-align: center;' },
        { name: 'animator', kind: 'Animator', duration: 1500, startValue: 100,
          endValue: 300, onStep: 'expand', onEnd: 'done' },
        { kind: 'Button', content: 'Start', ontap: 'startAnimator' },
    ],
    startAnimator: function() {
        this.set('$.expander.content', 'Presto');
        this.$.animator.play();
    },
    expand: function(sender, event) {
        this.$.expander.applyStyle('width', Math.floor(sender.value) + 'px');
    },
    done: function() {
        this.set('$.expander.content', 'Change-o');
    }
});
```

TIP

Try it out: jsFiddle.

Enyo also has some kinds for dealing with sprite animation. Find out more from this blog post.

Ajax and JsonpRequest

`Ajax` and `JsonpRequest` are both objects that facilitate performing web requests. It is worth noting that they are objects and not components. Because they are not components, they cannot be included in the `components` block of a kind definition. We can write a simple example to show how to fetch some data from a web service:

```
enyo.kind({
    name: 'AjaxSample',
    components: [
        { kind: 'Button', content: 'Fetch Repositories', ontap: 'fetch' },
        { name: 'repos', content: 'Not loaded…', allowHtml: true }
    ],
    fetch: function() {
        var ajax = new enyo.Ajax({
            url: 'https://api.github.com/users/enyojs/repos'
        });
        ajax.go();
        ajax.response(this, 'gotResponse');
```

```
    },
    gotResponse: function(sender, inResponse) {
        var i, output = ”;
        for(i = 0; i < inResponse.length; i++) {
            output += inResponse[i].name + ‘<br />’;
        }
        this.set(‘$.repos.content’, output);
    }
});
```

TIP

Try it out: jsFiddle.

In this sample we use the GitHub API to fetch the list of the Enyo repositories. In the button’s tap handler, we create an Ajax object, populate it with the appropriate API URL, and set the callback method for a successful response. We could have passed additional parameters for the service when we called the `go()` method. In general, we would trap error responses by calling `ajax.error()` with a context and error handling method.

TIP

The Ajax object performs its request asynchronously, so the call to `go()` does not actually cause the request to start. The request is not initiated until after the `fetch()` method returns.

A general discussion of when and how to use Ajax and JSON-P are outside the scope of this book.

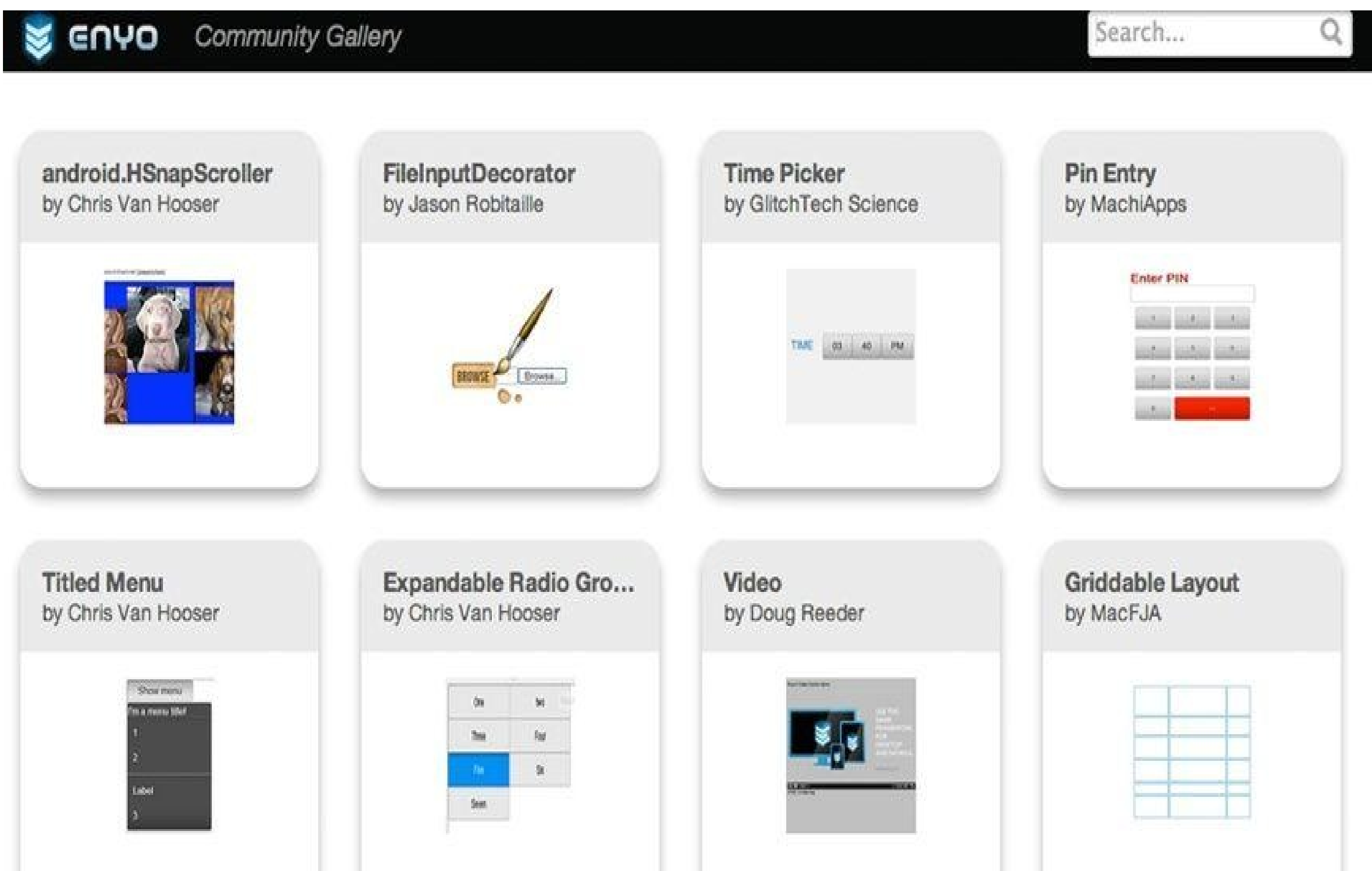
WARNING

By default, Enyo adds a random query string onto Ajax requests to prevent aggressive browser caching. This can interfere with some web services. To disable this feature, add `cacheBust: false` to the Ajax configuration object.

We’ll touch more on using web data sources in Chapter 5.

Community Gallery

The Enyo developers decided to keep the core of Enyo very simple. The additional libraries supplied with Enyo are also similarly focused. No framework can provide all the possible components that users will need. Fortunately, all the features of Enyo that we've discussed up to this point mean that it's very easy to create reusable components. The developers have created a community gallery to make it easy to find and share these reusable components. The gallery includes a variety of components that can be easily dropped in to your apps.



Hopefully you will feel motivated to create new components and share them with the community.

Summary

In this chapter, we explored components and the visual controls that Enyo developers use to make beautiful apps. We explored the various widgets that Onyx and Moonstone have to offer and learned a bit about using them. We also covered some non-visual objects Enyo provides. In the next chapter, we'll take Enyo to the next level by exploring how to arrange controls.

Chapter 4. Layout

In this chapter we’ll explore how to enhance the appearance of Enyo apps by using various layout strategies to place controls where we want them. By combining the knowledge gained in the previous chapters with the layout tools in this chapter, you’ll have most of the knowledge you need to create compelling apps using Enyo. We’ll explore each of the layout tools using examples you can run in your browser.

As with visual controls, Enyo provides both core layout strategies and an optional library called Layout. The core strategies provide the “simpler” approach to layout while the Layout library provides some more advanced features. The Onyx library also provides a layout tool in the form of the `Drawer` component and the Moonstone library has a number of enhanced layout controls.

Responsive Design

Before we begin talking about layout strategies we should discuss *responsive design*. Responsive design means that an app or web page changes its appearance (or functionality) depending upon the device or display size it is used on. It's important to consider how your app will look on different displays when designing a cross-platform app. Responsive web design is a topic that probably deserves a book of its own. You are encouraged to research the associated tools and techniques on the Web. Many of those same tools are used both within Enyo and by Enyo app developers. In particular, CSS media queries are often used in Enyo apps. We'll discuss the tools that Enyo makes available for designing responsive apps, but you may need to supplement these tools in certain circumstances.

Core Layout Features

Enyo provides two useful mechanisms for layout in the core: scrollers and repeaters. The `Scroller` kind implements a section of the display that is scrollable by the user while the `Repeater` kind is useful for making repeating rows of items. There are also data-aware controls for list and grid layout that we will cover in Chapter 5.

Scrollers

One of the bigger challenges in a mobile app is presenting a scrolling area of information that would otherwise be too big to fit. While many solutions exist, their cross-platform performance varies greatly. The Enyo team has spent a considerable amount of time analyzing performance issues and bugs across various browsers to produce the `Scroller` component.

Scrollers require very little configuration but do have some settings you can control. The `vertical` and `horizontal` properties default to automatically allow scrolling if the content of the scroller exceeds its size. Setting either to `'hidden'` disables scrolling in that direction while setting either to `'scroll'` causes scroll thumbs to appear (if enabled) even if content otherwise fits. The `touch` property controls whether desktop browsers will also use a touch-based scrolling strategy (instead of thumb scrollers).

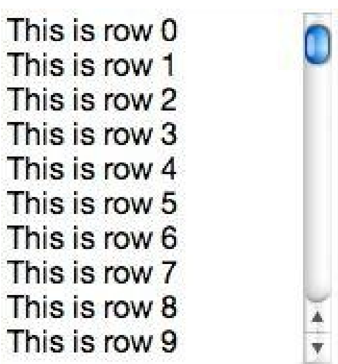
For more information on scrollers, visit the scroller documentation page.

Repeaters

Another challenge is to display a list of repeating rows of information. The `Repeater` component is designed to allow for the easy creation of small lists (up to 100 or so items) of consistently formatted data. A repeater works by sending an event each time it needs data for a row. The method that subscribes to this event fills in the data required by that row as it is rendered. The following sample shows a repeater that lists the numbers 0 through 99:

```
enyo.kind({
    name: 'RepeaterSample',
    kind: 'Scroller',
    components: [{
        kind: 'Repeater',
        count: 100,
        components: [{ name: 'text' }],
        onSetupItem: 'setupItem',
        ontap: 'tapped'
    }],
    setupItem: function(sender, event) {
        var item = event.item;
        item.set('$.text.content', 'This is row ' + event.index);
        return(true);
    },
    tapped: function(sender, event) {
        enyo.log(event.index);
    }
});
```

www.allitebooks.com



TIP

Try it out: jsFiddle.

You'll notice that we placed the `Repeater` into a `Scroller`. As the contents would (likely) be too large to fit onto your screen, we needed the scroller to allow all the content to be viewable. The `components` block of the `Repeater` is the template for each row and can hold practically any component, though it is important to note that fittables (see Fittable) cannot be used inside a repeater.

Also of note is the fact that each time we respond to the `onSetupItem` event, we reference the component(s) in the `components` block directly off the item passed in through the event. The repeater takes care of instantiating new versions of the components for each row. If you need to update a specific row in a repeater, you should call the `renderRow()` method and pass in the index of that row.

TIP

To redraw the whole repeater, such as when the underlying data has changed, set a new value for the `count` property. It is a good idea to pass a *truthy* value for the third parameter to `set()` in the case where only the data but not the number of records has changed (e.g., `this.set('$.repeater.count', 100, true);`). Alternately, you can call the `build()` method to redraw the list.

Layout Library Features

The modular Layout library includes several kinds for arranging controls. Three of the kinds we'll discuss are `Fittable`, `List`, and `Panels`. Visit the Enyo docs website to find out more information on the Layout library and the kinds not covered here.

Fittable

One aspect of layout that Enyo makes easier is designing elements that fill the size of a given space. Enyo provides two layout kinds, `FittableColumnsLayout` and `FittableRowsLayout`, to accomplish this. Fittable layouts allow for a set of components to be arranged such that one (and only one) component expands to fill the space available while the others retain their fixed size. `FittableColumnsLayout` arranges components horizontally while `FittableRowsLayout` arranges them vertically. To specify the child component that will expand to fit the space available, set the `fit` property to `true`.

To apply the fittable style to controls, set the `layoutKind` property. To make it easier to use, the Layout library includes two controls with the layout already applied: `FittableColumns` and `FittableRows`. Fittables can be arranged within each other, as the following code sample shows:

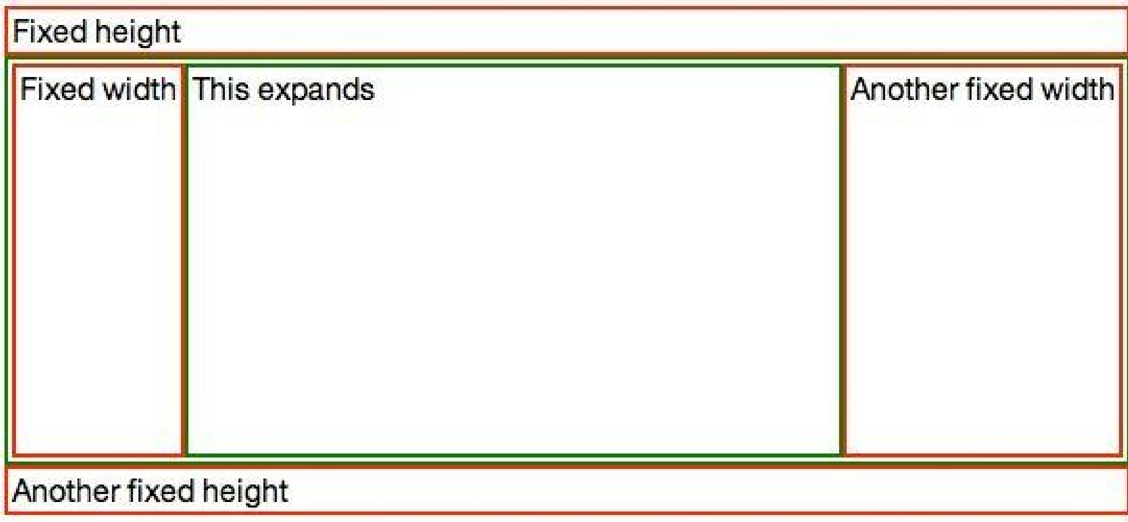
```
enyo.kind({
    name: 'Columns',
    kind: 'FittableColumns',
    components: [
        { content: 'Fixed width', classes: 'dont' },
        { content: 'This expands', fit: true, classes: 'do' },
        { content: 'Another fixed width', classes: 'dont' }
    ]
});

enyo.kind({
    name: 'FittableSample',
    layoutKind: 'FittableRowsLayout',
    components: [
        { content: 'Fixed height', classes: 'dont' },
        { kind: 'Columns', fit: true, classes: 'do' },
        { content: 'Another fixed height', classes: 'dont' }
    ]
});
```

TIP

Try it out: jsFiddle.

In the previous sample, we used both styles of applying a fittable layout, using a `layoutKind` for the row layout and using the `FittableColumns` for the column layout. We applied a simple CSS style that added colored borders to the expanding regions. If you resize the browser window, you'll see that the area in the middle will expand while the areas above and to the sides have fixed heights and widths, respectively.



TIP

Fittables only relayout their child controls in response to a resize event. If you need to relayout the controls because of changes in the sizes of components, call the `resize()` method on the fittable component.

While fittables provide an easy way to create specific layouts, they should not be overused. Reflows are performed in JavaScript and too many nested fittables can affect app performance.

Lists

Earlier we covered repeaters, which display a small number of repeating items. The `List` component serves a similar purpose but allows for a practically unlimited number of items. Lists include a built-in scroller and support the concept of selected items (including multiple selected items). Lists use a *flyweight* pattern to reduce the number of DOM elements that get created and, therefore, speed up performance on mobile browsers.

All this performance doesn't come without downsides, though. Because list items are rendered on the fly it is difficult to have interactive components within them. It is recommended that only simple controls and images be used within lists:

```
enyo.kind({
    name: 'ListSample',
    kind: 'List',
    count: 10000,
    handlers: {
        onSetupItem: 'setupItem',
        ontap: 'tapped'
    },
    components: [{ name: 'text' }],
    setupItem: function(sender, event) {
        this.set('$.text.content', 'This is row ' + event.index);
        return(true);
    },
    tapped: function(sender, event) {
        enyo.log(event.index);
    }
});
```

TIP

Try it out: jsFiddle.

In both this example and the `Repeater` example, we knew the number of items to display and set the `count` property when creating them. Often, you won't know how many items to display while writing your app. In that case, leave the `count` property undefined and

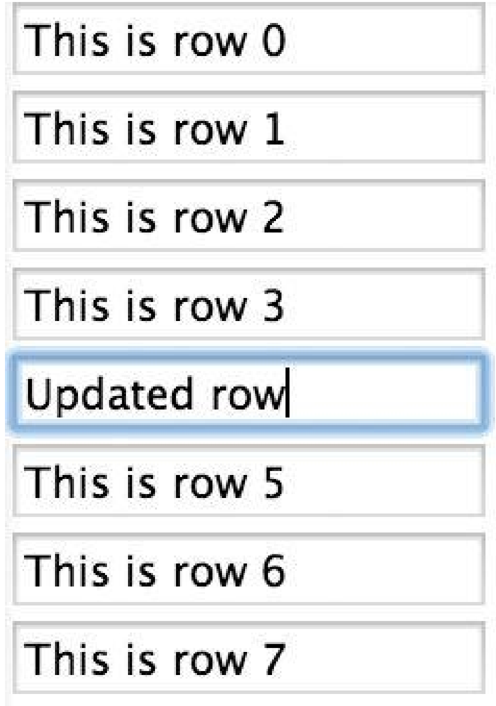
`set()` it once you have received the data. Once set, the `List` will render itself. If the underlying data changes, call `refresh()` to redraw at the current position or `reset()` to redraw from the start.

In order to make a `List` row interactive, you must first use the `prepareRow()` method. Then, a call to `performOnRow()` can be used to act on the row. Finally, `lockRow()` should be called to return the row to its non-interactive state. Let's modify the tap handler from the last example to show how to add an interactive element to a row:

```
enyo.kind({
    name: 'ListSample',
    kind: 'List',
    count: 1000,
    items: [],
    handlers: {
        onSetupItem: 'setupItem'
    },
    components: [
        { name: 'text', kind: 'Input', ontap: 'tapped',
           onchange: 'changed', onblur: 'blur' }
    ],
    create: function() {
        this.inherited(arguments);
        for(var i = 0; i < this.count; i++) {
            this.items[i] = 'This is row ' + i;
        }
    },
    setupItem: function(sender, event) {
        this.$.text.setValue(this.items[event.index]);
        return(true);
    },
    tapped: function(sender, event) {
        this.prepareRow(event.index);
        this.set('$.text.value', this.items[event.index]);
        this.$.text.focus();
        return(true);
    },
    changed: function(sender, event) {
        this.items[event.index] = sender.getValue();
    },
    blur: function(sender, event) {
        this.lockRow();
    }
});
```

TIP

Try it out: jsFiddle.



In this version, we detect a user tapping into a row and then lock that row so that we can make the `Input` editable. If we did not prepare the row, then the input control would not be properly associated with the row being edited and our changes would not be preserved. We look for the `onblur` event so we can call `lockRow()` to put the list back into non-interactive mode.

TIP

This sample isn't complete, as there are ways to move out of fields without triggering the blur event correctly. A better way to handle this kind of situation would be to use `Popup` to open a dialog on top of the list in response to a click on the row.

`List` and `Repeater` have data-aware versions that are easier to work with. We'll cover data-aware components in Chapter 5.

Panels

`Panels` are one the most flexible layout tools Enyo has to offer. Panels give you the ability to have multiple sections of content that can appear or disappear as needed. You can even control how the panels arrange themselves on the screen by using the `arrangerKind` property. The various arrangers allow for panels that collapse or fade as moved, or that are arranged into a carousel or even a grid.

`Panels` have an `index` property that indicates the active panel. Although the various arrangers can present more than one panel on the screen at a time and all such visible panels can be interactive, the active panel is important. You can easily transition the active panel by using the `previous()` and `next()` methods, or detect when a user has moved to a new panel (e.g., by swiping) by listening for the `onTransitionFinish` event.

A quick example of how to use `Panels` will help explain. In this example, we'll set up a layout that can have up to three panels, depending on the available width. As the available width shrinks, the number of panels visible will also shrink, until only one remains:

```
enyo.kind({
    name: 'PanelsSample',
    kind: 'Panels',
    arrangerKind: 'CollapsingArranger',
```

```
        classes: 'panels-sample',
        narrowFit: false,
        handlers: {
            onTransitionFinish: 'transitioned'
        },
        components: [
            { name: 'panel1', style: 'background-color: blue' },
            { name: 'panel2', style: 'background-color: grey' },
            { name: 'panel3', style: 'background-color: green' }
        ],
        transitioned: function() {
            this.log(this.index);
        }
    });
```

In order to achieve the sizing, we'll use a little CSS and some *media queries* to size the panels appropriately:

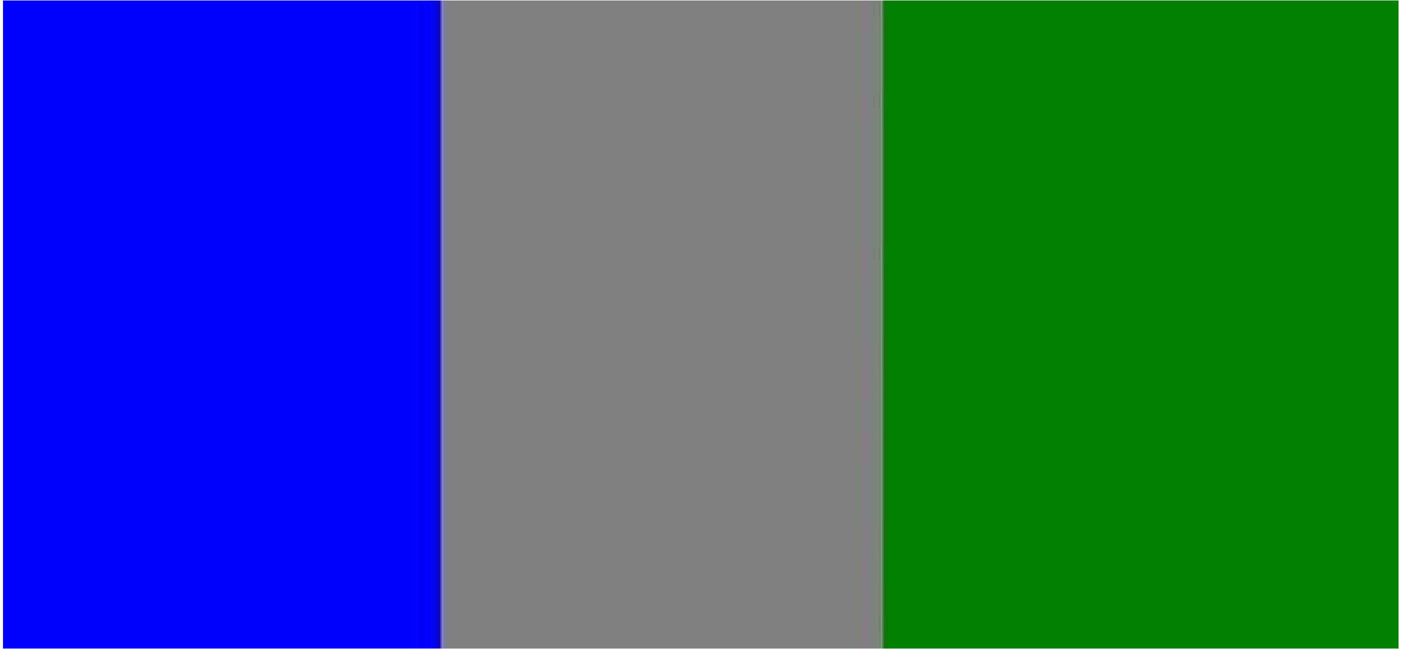
```
.panels-sample > * {
    width: 200px;
}

@media all and (max-width: 500px) {
    .panels-sample > * {
        min-width: 200px;
        max-width: 100%;
        width: 50%;
    }
}

@media all and (max-width: 300px) {
    .panels-sample > * {
        min-width: 100%;
        max-width: 100%;
    }
}
```

TIP

Try it out: jsFiddle.



For this sample, we set the `narrowFit` property to `false`. By default, the individual panels in a `CollapsingArranger` panel will fill the available space when the screen size is below 800px. We overrode the default in order to use 200px as the minimum width of a panel. The CSS we used detects when the screen gets below 500px and we limit each panel to half the space. Then, when the screen gets below 300px, we cause the panels to take up all the space. The user can still swipe left and right to reveal panels that aren't currently

visible.

We have only touched on the power of the `Panels` component. You should check out the Panels documentation for more ideas on how to use them.

Summary

You are now well on your way to producing beautiful apps that perform well on mobile and desktop platforms. We looked at some techniques for designing responsive apps that make the best use of a user’s display size. In the next chapter we’ll learn how to create data-driven applications using Enyo.

Chapter 5. Writing Data-Driven Applications

Enyo provides first-class support for creating rich, data-driven applications. Along with the data binding and observer features we touched on briefly in Chapter 2, there are models, collections, data-driven controls, and ways to synchronize data with remote data sources. In this chapter we'll explore these concepts and components.

Models

The bindings in Enyo work with any `Object`, which makes it easy to associate the data from one component to another. Sometimes, however, the data that needs binding doesn't live neatly within any one component in the app. To handle such situations, Enyo has `Model`. `Model`, which is not actually an `Object` but does support `get()` and `set()`, is designed to wrap plain-old JavaScript objects and make the data available for binding. The following illustrates the creation of a simple model:

```
var restaurant = new enyo.Model({
    name: 'Orenchi',
    cuisine: 'Japanese',
    specialty: 'ramen'
});
```

You can derive from `Model` to create new model types and specify default attributes and values:

```
enyo.kind({
    name: 'RestaurantModel',
    kind: 'enyo.Model',
    attributes: {
        name: 'unknown',
        cuisine: 'unknown',
        specialty: 'unknown',
        rating: 0
    }
});
```

Whenever a `RestaurantModel` is instantiated, the defaults will be applied to any properties whose values are not explicitly defined:

```
var mcd = new RestaurantModel({ name: 'McDonalds' });

mcd.get('specialty');
// returns 'unknown'
```

TIP

Try it out: jsFiddle.

TIP

In this sample and some that follow, there is an interactive console that allows you to experiment with the code. You can type JavaScript statements into the gray box and run them to see what happens. Try creating new models or changing some of the values around and see what happens. The console is based on a lightly modified version of JS Console by Remy Sharp.

In addition to defaults, you can add methods, computed properties (discussed later in this chapter), observers, and bindings to models. For example, to track how often the name of a restaurant has changed, you can add a `nameChanged()` method:

```
enyo.kind({
    name: 'RestaurantModel',
    kind: 'enyo.Model',
    attributes: {
        name: 'unknown',
        cuisine: 'unknown',
        specialty: 'unknown',
        rating: 0
    },
    nameChanged: function(was, is) {
        if(is) {
            this.changeCount = this.changeCount ?
                this.changeCount++ : 1;
        }
```

www.allitebooks.com

```
    }
});
```

TIP

Try it out: jsFiddle.

The previous code checks to ensure that the name is being set to a new value and, if so, increments the count (unless it was undefined, in which case it is assigned a value of 1).

TIP

The `nameChanged` method is not invoked during model creation, as the example shows. Also, note that the `changeCount` property is not fetchable using `get()` because it was not declared in the `attributes` block. When calling `get()` or `set()` on a model, you are interacting with properties on the `attributes` member, not the model itself. Always use `get()` and `set()` when working with model properties.

It is very easy to design components that work with models. Let's create a component to view our restaurant model:

```
enyo.kind({
    name: 'RestaurantView',
    components: [
        { name: 'name' },
        { name: 'cuisine' },
        { name: 'specialty' },
        { name: 'rating' }
    ],
    bindings: [
        { from: 'model.name', to: '$.name.content' },
        { from: 'model.cuisine', to: '$.cuisine.content' },
        { from: 'model.specialty', to: '$.specialty.content' },
        { from: 'model.rating', to: '$.rating.content' }
    ]
});
```

TIP

Try it out: jsFiddle.

The `RestaurantView` component uses bindings to map the fields from its `model` property to the appropriate controls. Whenever a new model is assigned or one of the properties in the assigned model changes, the contents of the control will be updated. To assign the model to the view, set the `model` property during creation, use `set()`, or bind the `model` property to a model stored elsewhere.

Collections

While `Model` wraps a single object, `Collection` is a `Component` that wraps arrays of objects. A collection can be initialized with an array, in which case each object in the array is upgraded to a model as it's added:

```
var fruits = new enyo.Collection([
    { name: 'apple' },
    { name: 'cherry' },
    { name: 'banana' }
]);
```

Individual models can be retrieved with the `at()` method. Models can be added by calling the `add()` method and passing in an object, a model, or an array of either:

```
fruits.at(0).get('name');
// returns "apple"

fruits.add({ name: 'rambutan' });
fruits.at(fruits.length-1).get('name');
// returns "rambutan"
```

TIP

Try it out: jsFiddle.

In many ways, collections behave like arrays. They have a `length` property that reflects the number of items in the collection. They also support various `Array` methods like `find()` and `forEach()`. For a comprehensive list of methods supported by `Collection`, see the full API documentation.

Like `Model`, `Collection` can be subclassed to specify additional configuration and methods. For example, you can specify a default model to be used when objects are added to the array:

```
enyo.kind({
    name: 'RestaurantCollection',
    kind: 'enyo.Collection',
    model: 'RestaurantModel'
});
```

Collections are very powerful when they are teamed up with data-aware components. We'll explore those later in this chapter.

Computed Properties

Applications often need to alter or combine data before it can be used. For example, it is convenient to combine a person’s first and last names into a full name. Enyo provides computed properties to centralize this work instead of requiring you to write repetitive code. Computed properties can be used on any `Object` or `Model` and work just like other properties.

Let’s adjust the `RestaurantModel` to have a property that contains the rating expressed in number of stars:

```
enyo.kind({
    name: ‘RestaurantModel’,
    kind: ‘enyo.Model’,
    attributes: {
        name: ‘unknown’,
        cuisine: ‘unknown’,
        specialty: ‘unknown’,
        rating: 0
    },
    computed: [
        { method: ‘starRating’, path: ‘rating’ }
    ],
    starRating: function() {
        var rating = this.get(‘rating’);
        return rating + ‘ star’ + ((rating == 1) ? ” : ‘s’);
    }
});
var rest = new RestaurantModel({
    name: ‘The French Laundry’,
    rating: 5
});

rest.get(‘starRating’);
// returns “5 stars”
```

TIP

Try it out: jsFiddle.

A computed property requires a method with the same name to compute the value. The `path` in the declaration refers to the property (or properties) it is dependent upon, much like observers. The `path` can be either a string (if there is only one property) or an array of strings. If one of its `path` properties changes, a computed property is recalculated the next time it is needed.

Data-Aware Components

Using bindings, any component can be linked with data from another component or with a model. When dealing with collections, you need controls that know how to render the contents. Unsurprisingly, these components deal with displaying data in lists or tables. The core data-aware components include `DataList`, `DataRepeater`, and `DataGridList`. The Moonstone library includes some additional data-aware components.

Each of these components looks for a `collection` property that will contain the data to be rendered. Let's implement a `DataRepeater` that can display the collection of restaurants we created earlier:

```
enyo.kind({
    name: 'RestaurantRepeater',
    kind: 'enyo.DataRepeater',
    components: [{
        components: [
            { name: 'name' },
            { name: 'cuisine' },
            { name: 'specialty' },
            { name: 'rating' }
        ],
        bindings: [
            { from: 'model.name', to: '$.name.content' },
            { from: 'model.cuisine', to: '$.cuisine.content' },
            { from: 'model.specialty', to: '$.specialty.content' },
            { from: 'model.rating', to: '$.rating.content' }
        ]
    }]
});
```

TIP

Try it out: jsFiddle.

As with `Repeater`, the `components` block is the template for each row. The `bindings` section in the preceding code references a `model` property, which is automatically set from the collection for each row that needs to be rendered. This allows for a simple mapping from the properties of the model to the components in the `DataRepeater`.

We can simplify the previous code by reusing our `RestaurantView` component:

```
enyo.kind({
    name: 'RestaurantRepeater',
    kind: 'enyo.DataRepeater',
    components: [{ kind: 'RestaurantView' }]
});
```

TIP

Try it out: jsFiddle.

Some of the benefits of using data-aware components over their non-data-aware versions include automatic updates when any of the underlying models change, built-in support for selection, and simpler binding of data to the components. You can find out more about the data-aware components in the API viewer.

Fetching Remote Data

It's a rare app these days that doesn't interact with data stored somewhere in the cloud or locally in the browser. Enyo uses the concept of data sources to work with persistent data. There are three data sources included with Enyo: `AjaxSource`, `JsonpSource`, and `LocalStorageSource`. Apps can use or extend these to fetch and commit data.

TIP

The Ajax and JSONP sources are intended to be extended by app developers. They will work as-is in cases where the server interaction is very simple.

Let's revisit the sample from Chapter 3 where we fetched the list of repos from GitHub. We'll update that sample to use a collection, a source, and a `DataRepeater`:

```
enyo.ready(function() {
    enyo.AjaxSource.create({ name: 'ajax' });
    var collection = new enyo.Collection({
        source: 'ajax',
        url: 'https://api.github.com/users/enyojs/repos'
    });

    enyo.kind({
        name: 'RepoView',
        kind: 'DataRepeater',
        collection: collection,
        components: [{
            components: [{ name: 'repoName' }],
            bindings: [
                { from: 'model.name', to: '$.repoName.content' }
            ]
        }]
    });

    new enyo.Application({ name: 'app', view: 'RepoView' });

    collection.fetch();
});
```

TIP

Try it out: jsFiddle.

In the preceding code, we created a new instance of `AjaxSource` (the Ajax source component) and assigned it to a new collection. We then assigned the collection to the `collection` attribute of the `DataRepeater`. Finally, we called the `fetch()` method on the collection to get the list of repositories from GitHub.

In addition to the `fetch()` method, models and collections support `commit()` and `destroy()`. All three methods can take an optional parameter hash that affects the way the source treats the data. If not supplied, the options will be taken from the model or collection's `options` property. In this way, you can override the default options for a specific method call.

WARNING

`fetch()`, `commit()`, and `destroy()` require that the model or collection not be in an error state. You must call the `clearError()` method on a model after an error occurs. You should define an error handler either in the `options` hash for the collection or when passing the options to those methods.

TIP

Enyo has a feature that will attempt to consolidate models to reduce memory usage and avoid out-of-sync data. For example, if two collections use the same model, they will share any instances of models that have the same

`primaryKey` (by default `'id'`).

Putting It All Together

To get a feel for what a full Enyo application is like, take a look at a restaurant list app online. This app implements several of the features we’ve covered in previous chapters, including the Onyx UI library, `Router`, Collections, and a collection-aware list. The app also uses local storage to persist the restaurants between loads. You can view the source on GitHub.

Summary

In this chapter, we touched on just a few of the features Enyo provides for creating data-driven applications. We covered models, the basic building blocks of data-driven applications, and collections. We discussed computed properties and how to use them. Lastly, we covered how to fetch data from remote sources. There are even more features that we didn’t get to explain, including collection filters and relational data. With all these rich features, it is easy to create data-driven applications with Enyo.

Chapter 6. Fit and Finish

In the preceding chapters we laid down the foundations you need to create Enyo apps. In this chapter, we’ll explore some of the pieces necessary to make those apps more memorable. We’ll cover how to style your apps, how to tune them to perform well on less powerful platforms, how to prepare them for translation to other languages, and how to troubleshoot bugs that inevitably arise. As always, we’ll explore these concepts through interactive samples.

Styling

Enyo provides some very nice looking controls with the Onyx and Moonstone libraries. However, an app can set itself apart from others by having a unique user interface. Fortunately, it's very easy to change the look of controls. We'll explore several ways to accomplish that.

Styles and Classes

All Enyo controls have two properties to aid in styling: `style` and `classes`. These two properties correspond to an HTML element's `style` and `class` attributes. The `style` property can be used to apply a specific style to a single control. To work with the `classes` property, you must add CSS classes to a style sheet. In general, it is better to use `classes` in an app for two reasons: first, components are more reusable if styling is not embedded within them; second, using CSS classes allows you to keep the styling in a single, centralized location.

Enyo provides `applyStyle()` to update an individual style and `addStyles()` to add styles onto the existing styles of a control. We used the `applyStyle()` function in the traffic light sample at the start of the book. Passing a `null` as the second parameter to `applyStyle()` removes the style. For updating classes, Enyo provides `addClass()`, `removeClass()`, and `addRemoveClass()`.

WARNING

It might seem like calling `set()` with the `style` and `classes` properties would be a good way to update a control. However, doing so completely replaces the styles and classes of the control. Use `set()` carefully with these properties.

Overriding Onyx Styles

Each Onyx control includes one or more classes. It is possible to override some (or all) of the default styling by overriding those styles in your CSS file. One simple way to discover the class names to override is to use your browser's inspector to see what classes are applied to a particular control. You can then use those classes to override the way that control looks everywhere in your app. The following image shows the Chrome inspector output of the Onyx sample from Chapter 3:

```
▼<div id="controlSample" class="enyo-fit enyo-clip">
    <button class="enyo-tool-decorator onyx-button enyo-unselectable" id="controlSample_button">Click</button>
    <br id="controlSample_control">
    <div class="enyo-checkbox onyx-checkbox" id="controlSample_checkbox" type="checkbox" checked="checked"></div>
    <br id="controlSample_control2">
  ▶<label class="enyo-tool-decorator onyx-input-decorator" id="controlSample_inputDecorator">…</label>
    <br id="controlSample_control3">
  ▶<label class="enyo-tool-decorator onyx-input-decorator" id="controlSample_inputDecorator2">…</label>
  </div>
```

The Onyx button has, among its classes, `onyx-button`. If we want to override the styling on all the buttons in our app without having to manually add a class to each one, we could write our own CSS rule for `onyx-button`:

```
.onyx-button {
    background-color: cyan;
}
```

www.allitebooks.com

TIP

Try it out: jsFiddle.

In general, you will need to use a CSS selector that is more specific than the styles in the Enyo CSS. One method is to add a base class to your view component and then use that in combination with your CSS selector. In the Onyx sample, we cannot override the background color of the actual input control without a more specific selector:

```
.myapp .onyx-input {
    background-color: tomato;
}
```

TIP

Try it out: jsFiddle.

In general, it’s better to style the input decorator rather than the input itself.

Less Is More

You could, of course, simply go into the Onyx library directory and directly edit the CSS file. Knowing that app developers would want to do this, the Enyo developers provide Less files for generating the CSS used by Onyx (and Moonstone as well). Less provides a programmatic approach to creating CSS while keeping most of the flavor of CSS. In order to compile Less you will need to have Node.js installed, and it helps to be working on a Bootplate project (see Appendix A).

TIP

Less can be used “live” in a browser. The debug build of Bootplate projects loads a JavaScript library that processes Less files in the browser. Because of the additional processing needed, it isn’t recommended to use that method with deployed code. You can disable Less by commenting out the line in *debug.html* that loads *less.js*.

Less files can be found in the *css* directory of the Onyx library, along with a previously compiled CSS file. Of particular interest is the *onyx-variables.less* file, which contains some common settings used throughout the library. Here’s a sample from that file:

```
/* Background Colors */
/* -------------*/
@onyx-background: #EAEAEA;
@onyx-light-background: #CACACA;
@onyx-dark-background: #555656;
@onyx-selected-background: #C4E3FE;
```

```
@onyx-button-background: #E1E1E1;
```

By overriding a particular variable, we can affect a wide range of Onyx styles. If you want to create your own overrides, you can modify your app as follows:

- In *source/package.js*, change `$lib/onyx` to `$lib/onyx/source`.
- In *source/style/package.js*, uncomment the reference to *Theme.less*.
- Edit *source/style/Theme.less* and place your overrides into the places indicated.

To change all Onyx buttons to lime green, you could place the following where variable overrides go:

```
@onyx-button-background: lime;
```

For more information on Less, see the Less website. For more information on theming Enyo, visit the Enyo UI Theming page.

Moonstone provides Less files as well and the override process is very similar.

Performance Tuning

With all of the styling options available to you it can be very tempting to pull out all the stops and add drop shadows, rounded corners, and all sorts of bells and whistles to your app. You need to be careful, though. While Enyo enables you to make native quality apps with HTML5 and CSS, you need to test the performance on mobile devices and older browsers (such as Internet Explorer 8), if you target them.

In desktop environments you can expect very good performance regardless of the CSS tricks you use. In the mobile world, where there's less processing power and less memory, things can get bogged down very quickly. Particular performance hogs include the aforementioned drop shadows and rounded corners. Other offenders include computed gradients, overlarge images, and long-running JavaScript. It's very important that you test how your app performs on the least capable system you're targeting. You may need to disable some features by using `enyo.platform` to detect the platform you are running on.

One of the most important factors in how an app is perceived is its responsiveness. It is very important that when a user taps on buttons, there is visual feedback that something happened. If you attempt to perform a long running calculation in a tap handler, the user will not see the button respond properly to the tap. Attempt to return as quickly as possible from event handlers and perform the calculations in response to a timer or animation frame request. Enyo includes an `Async` object for performing asynchronous actions.

TIP

> With mobile devices, the simpler the HTML and CSS, the faster the performance. It can be tempting to create every single object that your app might use. However, placing all those components into the DOM, even if they're hidden, affects performance. Create only the objects you need and get rid of those you don't need anymore.

Lastly, with installable apps you should be careful about loading remote resources. Mobile users may not always have an Internet connection and, when they do, it may be slow. If your app depends on particular images, package them with your app. If your resources change over time, use caching techniques.

A full discussion of performance tuning is outside the scope of this book. For more information on some of the pitfalls, you can read HTML5 Techniques for Optimizing Mobile Performance and other sites.

Debugging

So far we've painted a rosy picture of life with Enyo. Of course, sometimes things don't go so well. Fortunately, you have a number of tools at your disposal to figure out what went wrong. First and foremost, because Enyo is truly cross-platform, many problems can be detected and fixed by running your apps in a desktop browser. All the modern browsers have JavaScript debuggers available that make it very easy to see errors and even inspect the state of the DOM. In general, problems come in two varieties: code issues and layout issues.

> **TIP**
>
> One of the most common errors in Enyo apps is forgetting to call `this.inherited(arguments)` when overriding methods on a parent object. This occurs most often with the `create()` and `render()` methods but can also happen with others. Leaving out this call can cause components to render incorrectly or not appear at all. Another common error is failing to return a *truthy* value from event handlers and having the event handled by more than one component. This can be especially bad in the case of nested `List` components.

Layout Issues

We covered some of the great layout features that Enyo offers in Chapter 4. However, even with these features at your disposal, things can go wrong. One common problem app developers experience occurs when they fail to provide a height to `List` or `Scroller` components. Without a height, these elements will end up invisible. Providing a height or placing the component in a fittable layout can solve that issue. Fittables themselves can also cause problems. Forgetting to assign `fit: true` to one and only one of the components of a fittable component can lead to rendering issues.

Sometimes things just end up in the wrong place or have the wrong style. A quick way to see what has happened is to use the DOM inspector in your browser to see what styles have been applied to the elements in question. Sometimes, CSS precedence can be the source of problems. Use the DOM inspector to check whether a component rendered at all or if it's merely hidden. Other times, it can help to add `!important` to a style.

Layout is a complex topic. Some links that might help include: "Four simple techniques to quickly debug and fix your CSS code in almost any browser" and "Diagnose and fix layout problems" (IE-specific, but the same concepts exist in other browsers). Also check out "CSS In Your Pocket - Mobile CSS Tips From The Trenches" by Angelina Fabbro for a great introduction to CSS debugging tools and techniques.

Code Issues

Bugs are unavoidable. Fortunately, there are lots of ways to squash them. One of the best ways to detect code errors is to keep the JavaScript console open while testing Enyo apps. If there's an error or typo in your code, it can cause strange problems. Seeing errors as they occur really helps in trapping the problem.

Enyo apps can also write to the JavaScript console with the `info()`, `warn()`, and `error()` methods on the `enyo` object. In addition, every Enyo kind can call `this.log()` to send output that includes the kind's name and the name of the method that generated the log message.

Sometimes a passive approach to debugging a problem isn't enough. In these cases you

can set breakpoints in your code and step through methods that are misbehaving. A handy trick that is supported by the major browsers is putting the `debugger` command into code you want to inspect. When the browser reaches that code, it will stop and allow you to inspect the state of the app. Just remember to remove that command before publishing your app!

Once you've identified a place in the code that you want to inspect, it's easy to see all the components belonging to a component. `this.$` will contain a hash of all the owned components. Also, `enyo.$` contains a hash of all named components in your app. You can easily walk through these by inspecting deeper and deeper into a kind.

TIP

When inspecting an element in the DOM you can use its *id* (e.g., "panelsSample_panel1") to find the reference within the `enyo.$` hash. To make things even simpler, Firefox, Chrome, and Safari all set `$0` to the last element inspected in the DOM. To find the associated Enyo control simply use `enyo.$[$0.id]`.

JSFIDDLE DEBUGGING

There's no doubt that jsFiddle is a great environment for quickly testing ideas. It does have some drawbacks, though. Among them is that debugging can be a little more difficult and there aren't a lot of options for working with complex apps. jsFiddle runs your code in an *iframe*. Because the code you are executing gets reloaded into the iframe each time you click Run, it can be a little tricky keeping breakpoints in line.

Using the `debugger` command mentioned previously can be helpful, as can using the Inspector window that shows you active source files. The source for your app will be launched from *fiddle.jshell.net* and will be in-line with the HTML source for the page. If your app does not appear to be working at all, make sure you have selected the correct framework:

If you have the wrong settings, you may see the following error message:

```
Uncaught TypeError: Cannot read property 'className' of undefined
```

If you run into an issue with a specific version of Enyo, you can try using the *nightly* build, which contains the most recent code submissions to Enyo. Conversely, if you run into a problem using the nightly build, try using the last released version. Released versions are tested more thoroughly, while nightly builds have the most recent fixes and features.

When using the JavaScript console, unless you use a breakpoint, you will be in the wrong context to execute commands to inspect the state of Enyo. Both Chrome and Safari support selecting the context through a drop-down list.

Going Global

Now that you've produced a beautiful (and bug-free) app, you'll want to share it with the world. Enyo provides a wrapper for the open source internationalization library iLib. This library provides facilities for substituting translated strings as well as formatting names, dates, and other data based upon a user's locale. The name of the enyo wrapper is enyo-ilib.

The library will attempt to figure out the locale based on cues from the browser. The current locale can be retrieved by calling `ilib.getLocale()`. In cases where the locale can't be determined, you can explicitly create a locale by calling `new ilib.LocaleInfo()` with the desired locale.

WARNING

The enyo-ilib library is not included if you are using a standard (non-Moonstone) Bootplate-based setup (see Appendix A). To enable the library in a Bootplate setup, execute the command: `git submodule add https://github.com/enyojs/enyo-ilib.git lib/enyo-ilib` and then add the library to your *package.js* with the line `'$lib/enyo-ilib'`.

Globalization Basics

In its most basic form, iLib handles string substitutions. Substitutions are performed using the `$L()` global function. At run time, the `$L()` function searches for an appropriate translation file for the user's locale (or the locale you set manually) and then attempts to locate the string that was passed in as the first argument. If a match is found, the translated string is used. If not, the original string is used.

Translation files should be placed in the *resources* directory of your app. Each locale translation should be in its own JSON file. Translation files are named *strings.json* and are stored in directories named after the language and optional subdirectories for the country code and variant. For example, a Canadian English translation file would be found in *resources/en/CA/strings.json*. Such a file might look like this:

```
{
    "Click": "Click, eh?"
}
```

Names, Dates, and Measures

App developers can add some extra polish to their apps by correctly formatting information for the user's region. The iLib library includes the ability to format names, phone numbers, dates, times, and more. For more information on the options supported by these formatters, please refer to the localization documentation page. The following example shows some of the basic routines for formatting these items:

```
enyo.kind({
    name: 'ILibSample',
    components: [
        { name: 'date' },
        { name: 'number' }
    ],
    create: function() {
        this.inherited(arguments);
        var dateFmt = new ilib.DateFmt({ length: 'short' });
        this.$.date.set('content', dateFmt.format(new Date()));
        var numFmt = new ilib.NumFmt({maxFractionDigits: 1});
        this.$.number.set('content', numFmt.format('86753.09'));
    }
```

```
});
```

TIP

Try it out: jsFiddle.

Two Onyx components are locale-aware: `DatePicker` and `TimePicker`. If enyo-ilib is loaded, they will use the current locale to format their contents. If the library isn’t loaded, then they will default to the US format. All of the number and date components in Moonstone are locale-aware.

There is a lot more power in iLib. You can find out more at the Enyo localization documentation page and the iLib documentation page.

Summary

You’ve now picked up some more tools for creating beautiful and functional Enyo apps and you know what to do when things go wrong. If you get stuck, there are many good resources available to you, including the Enyo forums and the Enyo IRC channel.

Chapter 7. Deploying

So now you’re a budding Enyo developer looking to deploy your app to all the supported platforms. The only question you have is: how? In this chapter we’ll explore the tools and techniques you’ll need to structure your apps and deploy them to various targets. At this point, we’ll need to set up a “real” development environment, since we can’t deploy apps by directing users to a page on jsFiddle.

For this chapter you should follow the Bootplate environment setup guide in Appendix A. Bootplate is a ready-to-use template that includes tools for easy deployment. Even if you ultimately choose a different structure for your app, you may still be able to apply some of these tools.

Bootplate App Structure

It's worth taking a few moments to discuss how an Enyo Bootplate app is structured. Until now, all of our samples have been run on jsFiddle and have consisted of, at most, three files. We haven't had to concern ourselves with where the Enyo framework files are coming from or how to add assets. Here's the Bootplate directory structure:

```
debug.html
index.html
assets/
build/
deploy/
enyo/
lib/
source/
tools/
```

There are some additional files included but these are the most important. *debug.html* is the file to load during debugging. It includes non-minified versions of Enyo and the libraries. *index.html* will load a deployed version of the app that has been minified (compressed). If a non-minified build of the application is not available, *index.html* will redirect to *debug.html*. The *assets* directory is a place to store images, fonts and other assets your app requires. The *build* directory contains the minified source of your app, including Enyo and the libraries. These files are loaded by *index.html*. The *deploy* directory contains a ready-to-deploy version of your application. These directories (*deploy* and *build*) are generated by the deployment scripts located in the *tools* directory. The *enyo* directory contains the framework source for Enyo, while the *lib* directory contains the various Enyo libraries needed for the app

The most interesting directory is the *source* directory. Within it will live the files you create to breathe life into your app. The *source* directory structure looks like this:

```
app.js
package.js
data/
    data.js
    package.js
style/
    main.less
    package.js
    Theme.less
views/
    package.js
    views.js
```

app.js contains the source for the `Application` object that forms the base of the app. This file can be used to set the initial view and instantiate any global models. *package.js* is part of a system that tells Enyo what source and stylesheets to load. Inside *package.js* is a call to `enyo.depends()`. Each line in the call adds either a file or directory to the app. Each directory specified should have its own *package.js*. This system makes it so that reusable components can be easily added to a project simply by adding the directory and including it in *package.js*.

The *data* directory is useful for defining models and collections needed by the app. The *style* directory contains *main.less*, which is where the app's CSS is specified. The *Theme.less* file allows for overriding the UI library's styles and is discussed in Chapter 6.

views contains source for the UI of your application. This is where you will define the

controls and components that make up your application. A simple app may only need to modify *views.js*, while a more complex app will have many files or directories.

Don’t be too constrained by the directories Enyo provides. Feel free to add more; just be sure to modify *package.js* to include all of your files and directories.

Web Targets

One of the simplest ways to deploy Enyo is to host it on a server and serve the apps embedded into a web page. Although all our examples have shown rendering Enyo objects into the document body, it is possible to render them into any element on the page (by setting the `renderTarget` property of the `Application` kind to the id of the element). For web deployment, simply copy the Enyo library and app source code up to a directory on your server and include them in your HTML source.

Bootplate makes this process easy by including a Grunt deploy task. For more on using Grunt with Bootplate see Using Grunt. If you are not using Grunt, Bootplate includes a deploy script that packages all the files and minimizes the source. For Windows, this script is called *deploy.bat*; for Mac and Linux, it's called *deploy.sh*.

Deploying speeds up loading and combines everything into the *deploy* directory. Once deployed, simply transfer the files from the *deploy* directory to your destination (e.g., a web host). Keep in mind, though, that deployed, minified code is much tougher to debug than unminified code.

Desktop Targets

JavaScript apps might not seem like the best choice when targeting the desktop; however, many of the features that make it great for creating Web apps also make it good for creating desktop apps. Particularly with Node.js for communicating with the system, a browser engine for displaying a user interface, and, of course, Enyo to simplify writing the app, you can quickly bring up a cross-platform desktop app without having to learn the nuances of each platform.

Two projects have brought together the Chromium browser with Node.js: Atom Shell and node-webkit. To demonstrate how a desktop JavaScript app looks, we'll use Atom Shell along with the default Bootplate app. First, download the appropriate version of Atom Shell for your system from the releases page and unzip it to a directory on your computer.

Create a new directory to house the Atom Shell project. Copy the sample *package.json* and *main.js* files from the Atom Shell quick-start guide into the directory. You can modify either file to your liking.

Next, execute the deploy script of a Bootplate project to create the minified project files. Copy the files from the *deploy* directory into your Atom Shell project directory. You can now test the app by executing the Atom Shell binary (follow the directions in the quick-start guide). To create a distributable app, follow the directions in the application distribution guide.

Enyo Bootplate App

Hello World

Tap me

A completed project directory is available on GitHub.

Smart Devices

One of the most interesting places for apps these days is on smartphones, tablets, smart TVs and other such devices. Enyo is perfectly suited to this environment. Enyo itself doesn't provide any kinds that give direct access to the hardware components of these devices, and not all device features have an HTML-standard method for access. However, Cordova, an Apache open source project, handles direct access to these features on many devices and provides a method for creating natively installable apps.

Enyo supports Cordova events natively and has a library called enyo-cordova available on GitHub. For more information on Cordova support with Enyo, please see Making Use of Cordova's Native Functions.

There are two ways to create apps using Cordova: by using the online PhoneGap Build tool or by downloading the Cordova library. We'll look briefly at both options.

PhoneGap Build

One of the simplest ways to get started with deploying mobile apps is to use Adobe PhoneGap Build. PhoneGap Build is a web-based tool for packaging cross-platform JavaScript apps. Among other things, it allows you to create installable apps for multiple targets quickly and easily.

PhoneGap requires that its JavaScript library be loaded in *index.html*. To do this, add the PhoneGap `script` tag just before the line that loads the Enyo source, as follows:

```
…
        <!– js –>
        <script src="phonegap.js"></script>
        <script src="build/enyo.js" charset="utf-8"></script>
…
```

After registering for a PhoneGap build account, you can pull projects directly from GitHub or, for private (as opposed to public, open source) projects, upload a *.zip* file. It is very easy to zip the contents of the *deploy* directory and upload it to PhoneGap Build. For some platforms, you will need to supply developer credentials before you have an installable app. Additionally, you'll want to set up app icons and other metadata needed by the various mobile stores.

If you want to test PhoneGap build, use the following repository link to create a test package: https://github.com/Enyo-UpAndRunning/phonegap-build-sample.git.

If you use Bootplate and a GitHub-based PhoneGap build, you will either need to have a separate repository for the deployed files (as in the preceding sample) or you will need to commit your deployed source along with your app and use a *.pgbomit* file to omit all the unminified source from being included in your final application.

Local Cordova Builds

PhoneGap Build is easy to use but it doesn't give you a lot of flexibility. Installing Cordova locally gives you much finer-grained control, as well as access to the build tools available on your platform of choice. In general, you will want to start with a shell app appropriate for the platform you wish to deploy on and then copy the deploy files to the *www* directory. Be careful to ensure that you load *cordova.js*, or your app may not work

correctly.

The Enyo Yeoman generator includes an option to create a full Cordova project, including the Cordova command-line tools. To create a new Cordova project, use the following command:

```
yo enyo —cordova myProject
```

For more information on getting started with Cordova, visit the Cordova site.

webOS Smart TVs

LG has made it easy to deploy Enyo applications to webOS Smart TVs. Included in the SDK (see Appendix A) are command-line tools for creating, packaging, testing, and deploying apps. The `ares-package` command uses the Bootplate deploy script to minify an app and then it creates a deployable package. The following command packages the app in the current directory:

```
ares-package .
```

To install the app to the emulator or a TV, use the `ares-install` tool. To deploy a sample app created by the `ares-generate` tool to the emulator, issue the following command:

```
ares-install —device emulator com.example.sample_0.0.1_all.ipk
```

For more information about developing for webOS TVs, visit the webOS TV for developers site.

Summary

You should now be familiar with some of the ways to package and deploy Enyo apps. Using this knowledge, you can deploy your apps on various platforms and know that your apps will work.

Chapter 8. Conclusion

By now you should be up and running with Enyo. You've seen the major features and dabbled with many of the minor ones. Enyo, while remaining small, fast, and focused, has a lot of power and there is still more to learn. I encourage you to go out and interact with the Enyo community through the Enyo forums and the #enyojs freenode.net IRC channel (*irc://chat.freenode.net/enyojs*). You'll find me there under the handle Roy__.

Enyo is an active project and there are always new features and updates being worked on. Follow Enyo on Twitter and read the official Enyo blog for the latest news and events.

Finally, I encourage you to share your thoughts on this book with me. I intend for this book to also be an active project that attempts to keep pace with the changes to Enyo. Keep up with the latest updates, errata, and more at this book's O'Reilly page.

Now, get out there and start using Enyo. Who knows? Your boss may come to your desk and ask you to produce a fantastic cross-platform app…

Appendix A. Setting Up a Development Environment

At some point, you’ll need to set up a copy of Enyo on your local computer or a server, if only to package up the applications you’ve developed. We’ll cover a few methods of setting up Enyo and discuss the prerequisites for each.

Prerequisites

Two basic tools are used by Enyo, which you may need to install: Node.js and Git. Let's look at why they are needed and where to get them.

Node.js

Node.js is a platform for running JavaScript outside of a browser. It allows JavaScript to be used as a general purpose scripting language. Node is available for Windows, Linux, and Mac OS X. Visit the Node.js download page to download the appropriate version of Node for your system.

> **TO NODE OR NOT TO NODE**
>
> There are several features of Enyo that rely upon Node.js. In Enyo, Node is used for minimizing Enyo source, packaging apps derived from Bootplate, and compiling Less files into CSS. It is also a requirement for the Enyo Yeoman generator. If you plan to release an Enyo app, you will need to install Node. If you just want to play around with Enyo and you don't mind running the non-minimized, debug version of Enyo, you don't need Node.

Git

Git is a distributed source-code management tool. It allows software developers to keep versioned copies of their source code. It is also the tool the Enyo team uses for Enyo development and the tool required to work with GitHub, an online source code repository that hosts the Enyo source.

Git is not required to use the basic parts of Enyo. You'll want to install Git if any of the following is true:

1. You want to keep up with the latest developments with Enyo.
2. You want to contribute to Enyo.
3. You want to use a system that makes it easy to keep past versions of your source code.

GitHub has instructions for setting up Git. The basic installation installs a command-line client. There are also GUI clients available for all the major platforms.

Installing Enyo

There are two general methods for installing Enyo, and one method specific to developing webOS Smart TV apps. The easiest way to make Enyo apps is to start with Bootplate. Bootplate includes all the scaffolding you'll need to debug and deploy an app. We'll cover Bootplate and the other methods for installing Enyo.

Bootplate

Bootplate is a scaffold upon which to build an Enyo app. It includes tools that allow you to easily debug your app in a browser and then deploy a minified version of Enyo with your app. It also provides an easy-to-use structure for your app. There are two versions of Bootplate available: Onyx Bootplate (for mobile and desktop apps) and Moonstone Bootplate (for smart TV apps).

There are three ways to install Bootplate: use the Yeoman generator, download a zipped archive from the Enyo site, or clone the archive from GitHub.

The simplest method is to download the zip archive from the Get Enyo page. As of this writing, the latest version is 2.5.1. After downloading, simply unzip the archive.

The next easiest method is to use the Enyo Yeoman generator. After installing Node on your computer, install the generator using the following command:

```
npm install -g generator-enyo
```

Once installed, a new bootplate can be generated by executing the following command:

```
yo enyo MyProject
```

In this command, *MyProject* is a directory name. This method will create an Onyx Bootplate. To create a Moonstone Bootplate, use the following:

```
yo enyo -m=moonstone
```

For more information on the generator and its options, see the Bootplate guide.

The last method is cloning Bootplate from GitHub. Use the following command to download the Onyx Bootplate:

```
git clone —recursive https://github.com/enyojs/bootplate.git
```

To clone Moonstone Bootplate from GitHub:

```
git clone —recursive https://github.com/enyojs/bootplate-moonstone.git
```

Full Source

You can also download the full source-code tree for Enyo from GitHub. To set up Enyo, you will need to clone the Enyo repository and then create a *lib* directory within the directory that contains the cloned repo. Inside the *lib* directory, clone the Enyo libraries you need for your application. The following diagram shows the directory structure and the Git repos:

```
enyo/          git@github.com:enyojs/enyo.git
lib/           (mkdir the lib folder)
   onyx/       git@github.com:enyojs/onyx.git
   layout/     git@github.com:enyojs/layout.git
   …
```

Using the webOS Developer Tools

Enyo is the primary method for developing smart TV applications for LG webOS Smart TVs. To make it easier for developers to get started, LG has prepared a Software Development Toolkit (SDK). This SDK includes a TV Emulator and command-line tools. One of the command-line tools, `ares-generate`, can be used to generate app templates based on the Moonstone version of Bootplate. To create a new application template in the directory *sampleProj*, issue the following command:

```
ares-generate sampleProj
```

To see the list of available templates, use:

```
ares-generate -l
```

Other command-line tools are discussed in webOS Smart TVs.

TIP

> The version of Enyo that is included with the SDK may not be the latest. You can use more recent versions of Enyo with the TV and the SDK. Use one of the other methods to install Enyo and add any needed files to the project.

Using Bootplate

Bootplate gives you a head start in creating your app by providing a ready-to-use structure for your app. Among other things, it provides a *source* directory that contains your app's controller (*app.js*), a *views* directory that contains your views, and a *style* directory to house CSS. You can modify the included *package.js* to add additional source files and directories.

Bootplate provides scripts to create a ready-to-deploy version of your app, including a minified version of Enyo. For general testing, you will load *debug.html* into your browser. When you have created a production version, you can load it by opening *index.html*. To produce a deployable production version of your app, issue the following command:

```
tools/deploy.sh (tools\deploy.bat on Windows)
```

When all the source has been combined and minified, it will be placed into the *deploy* directory. The contents of that directory can be copied up to a web server or packaged with one of the various packaging tools. For more about the layout of Bootplate, see Bootplate App Structure.

WARNING

When testing Enyo apps by loading a file directly into the browser (as opposed to serving it from a web server), you can run into security restrictions in the browser, particularly when attempting to perform requests to load resources. Some browsers allow you to override those security restrictions. For best results, test your app by serving it with a web browser (such as Apache) or using the node-based server included with Bootplate (see Using Grunt).

Using Grunt

Bootplate includes an easy-to-use script for deploying Enyo apps. This script uses Grunt (a node package) to execute tasks. If you have installed Enyo using the Yeoman generator, Grunt is set up and ready to use. If not, you will need to initialize the dependencies and install the Grunt command-line tool. Execute the following commands from the Bootplate directory to initialize Grunt:

```
npm install -g grunt-cli
npm install
```

Once initialized, a minified version of the app can be created by issuing the command:

```
grunt
```

The script (*Gruntfile.js*) also includes tasks to start a simple HTTP server, check the app source for warnings using JSHint, and remove deployed files. The commands, in order, are:

```
grunt serve
grunt jshint
grunt clean
```

About the Author

Roy Sutton is a member of the HP webOS Developer Relations team and a contributor to the Enyo project. He has been a mobile developer for longer than the term has existed.

Colophon

The animal on the cover of *Enyo: Up and Running* is the rustic sphinx moth (*Manduca rustica*).

The cover image is from Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.

Enyo: Up and Running

Roy Sutton

Editor

Meg Foley

Revision History

2015-01-08	First release

Copyright © 2015

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safaribooksonline.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Enyo: Up and Running*, the image of the rustic sphinx moth, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

O'Reilly Media

1005 Gravenstein Highway North

Sebastopol, CA 95472

2015-01-08T07:42:42-08:00

Enyo: Up and Running

Table of Contents

Preface
Where Did Enyo Come From?
Core Beliefs
What’s Enyo Good For?
Who Is This Book For?
Minimum Requirements
Typographic Conventions
Using Code Examples
Safari® Books Online
How to Contact Us
Acknowledgments
Content Updates
January 8, 2015
1. Light It Up
A New Project
Improvements
Curveball
QA on the Line
The E-mail
Summary
2. Core Concepts
Introduction
Kinds
Be Kind
Encapsulation
Properties
Basic Properties
Bindings and Observers
Events
Advanced Events
Final Thoughts on Encapsulation
Inheritance
Advanced Kinds
Instance Constructors
Statics
Summary
3. Components, Controls, and Other Objects
Components

Composition
Component Methods
Dynamic Components
Controls
Core Controls
Onyx Controls
Moonstone Controls
Methods and Properties
Other Important Objects
Application
Router
Animator
Ajax and JsonpRequest
Community Gallery
Summary
4. Layout
Responsive Design
Core Layout Features
Scrollers
Repeaters
Layout Library Features
Fittable
Lists
Panels
Summary
5. Writing Data-Driven Applications
Models
Collections
Computed Properties
Data-Aware Components
Fetching Remote Data
Putting It All Together
Summary
6. Fit and Finish
Styling
Styles and Classes
Overriding Onyx Styles
Less Is More
Performance Tuning
Debugging
Layout Issues

Code Issues
Going Global
Globalization Basics
Names, Dates, and Measures
Summary
7. Deploying
Bootplate App Structure
Web Targets
Desktop Targets
Smart Devices
PhoneGap Build
Local Cordova Builds
webOS Smart TVs
Summary
8. Conclusion
A. Setting Up a Development Environment
Prerequisites
Node.js
Git
Installing Enyo
Bootplate
Full Source
Using the webOS Developer Tools
Using Bootplate
Using Grunt
About the Author
Colophon
Copyright