Build Native-Quality Cross-Platform JavaScript Apps

Enyo

Up and Running

O'REILLY®

Roy Sutton

Enyo: Up and Running

Learn how easy it is to design and build responsive, cross-platform apps with the Enyo open source JavaScript framework. With this hands-on book, Enyo contributor Roy Sutton shows you how to get started with the framework's core object-oriented features, including its modular design, reusable and extensible components, layout and widget libraries, easy-to-use deployment options, and support for HTML5 standards.

Before you know it, you'll be writing native-quality apps that work equally well on smartphones, tablets, and desktops. Are you familiar with HTML, CSS, or JavaScript? Then you're ready for Enyo.

- Start with a sample project to get the feel of Enyo right away
- Learn about Enyo's "kinds" (component building blocks), encapsulation, inheritance, and other core features
- Design compelling and responsive apps with Enyo's layout strategies and Layout library
- Use existing components to create new components
- Build unique user interfaces with the Onyx library and widget set
- Explore the community gallery to find and share reusable components
- Tackle debugging, performance tuning, and globalization
- Package your app for web, desktop, and mobile targets, using Bootplate, AppUp, and PhoneGap

Twitter: @oreillymedia
facebook.com/oreilly

US $14.99 CAN $15.99
ISBN: 978-1-449-34312-5
51499
9 781449 343125

O'REILLY®
oreilly.com

Enyo: Up and Running

Roy Sutton

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Enyo: Up and Running

by Roy Sutton

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: Kara Ebrahim
Proofreader: Kara Ebrahim
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January 2013: First Edition

Revision History for the First Edition:

2013-01-22 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449343125* for release details.

ISBN: 978-1-449-34312-5

[LSI]

Table of Contents

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 and tablets) 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.

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.

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.

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.

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.

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 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 2.1.1, though it should apply to later versions.

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.

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

This icon precedes a tip, suggestion, or note.

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

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* by Roy Sutton (O'Reilly). Copyright 2013 Roy Sutton, 978-1-449-34312-5."

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*.

Safari® Books Online

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 product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers 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 dozens 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://oreil.ly/enyo-upandrunning*.

To comment or ask technical questions about this book, send email to *bookques tions@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, and Kara Ebrahim, who slew typos and wrangled words into their proper places. 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. Finally, many thanks to the HP webOS team who not only helped make Enyo a reality but also provides one of the best work environments a person could hope for.

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.kind({
    name: "Light",
    style: "width: 50px; height: 50px; border-radius: 50%; background: yellow;"
});

new Light().renderInto(document.body);
```

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 `renderInto()` function, you placed the new kind into the page's `body` element, causing Enyo to create the HTML. You inspect the HTML for the circle using your favorite browser 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

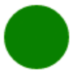
Now that you're feeling good about what you did, you check in the first version of the code to the company's source code management system. You know from past experience that sales will probably need the light in more colors than yellow. So, you decide to use Enyo's published properties feature to set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

```
enyo.kind({
    name: "Light",
    published: {
        "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);
    }
});
```

Try it out: jsFiddle.

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 kinds 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 Light({color: "green"}).renderInto(document.body);
```



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

```
var light = new Light({color: "green"}).renderInto(document.body);
light.setColor("blue");
```

Try it out: jsFiddle.

You remember that when you create properties, Enyo will automatically create a function for you to call to change the color. Looks like that works well, too.

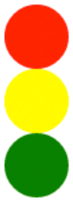
You check the latest version of the code and shoot an e-mail off to your boss. Your latest change added a bit more code but you know that you'll be able to use that light component again and again, regardless of what color sales promises.

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" }
    ]
});
```



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",
    published: {
        powered: true
    },
    handlers: {
        "ontap": "tapped"
    },
    create: function() {
        this.inherited(arguments);
        this.poweredChanged();
    },
    tapped: function(inSender, inEvent) {
        this.setPowered(!this.getPowered());
    },
    poweredChanged: function(oldValue) {
        this.applyStyle("opacity", this.powered ? "1" : "0.2");
    }
});
```

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 function 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 functions to each instance of your kind so it can access the functions and data of your kind's instance.

In your tap handler, you used the partner to the "set" method of published properties, "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 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 for 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 function 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",
    published: {
        powered: true
    },
    events: {
        "onStateChanged" : ""
    },
    handlers: {
        "ontap": "tapped"
    },
    create: function() {
        this.inherited(arguments);
        this.poweredChanged();
    },
    tapped: function(inSender, inEvent) {
        this.setPowered(!this.getPowered());
    },
    poweredChanged: function(oldValue) {
        this.applyStyle("opacity", this.powered ? "1" : "0.2");
        this.doStateChanged({ "powered" : this.powered });
    }
});
```

Try it out: jsFiddle.

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 `inSender` parameter to check to see which light sent the event and you can use the `inEvent` 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(inSender, inEvent) {
            enyo.log(inSender.name + " powered " + (inEvent.powered ? "on" : "off")
                + " at " + new Date());
        }
    });
```

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.

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 combining simple pieces creates complex apps. 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 and you will see them in later chapters.

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 that will be reused and lowercase names for instances of kinds.

`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 functions 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 can use the `renderInto()` function.

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

Published Properties

Kinds can declare published properties (for example, the `color` and `powered` properties from Chapter 1). These properties automatically get "getter" and "setter" functions as well as a mechanism for tracking changes to the properties. This is ideal for enforcing acceptable types and values. To declare published properties on a kind, include a section called `published` in the kind declaration:

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

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

The default setter function does not call the changed function if the value to be set is the same as the current value.

As mentioned, Enyo automatically implements the getter and setter functions for you. The default functions simply get or set the value of the property and the setter calls the changed function. You can override this behavior by implementing the functions yourself. In this way you can perform validation or calculations. If you override the setter, be sure to call your changed function, if appropriate.

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. Many of those values are indeed 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 the constructor:

```
enyo.kind({
    name: "MyKind",
    published: { myValue: 3 },
    create: function() {
        this.inherited(arguments);
        this.myValueChanged();
    },
    myValueChanged: function(oldValue) {
        // Some processing
    }
});
```

You should only specify simple values (strings, numbers, booleans, etc.) for the default values of published properties and member variables. Using arrays and objects can lead to strange problems. See "Instance Constructors" (page 15) for a method to initialize complex values.

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 subscribe to events that they're interested in and receive data. Events are declared like this:

```
enyo.kind({
    name: "MyKind",
    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 function whose name is prefixed with "do". Enyo creates the "do" helper function for us and it takes care of checking that the event has been subscribed to. The first parameter passed to the "do" function, 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: "MyKind", onMyEvent: "answered" }],
    answered: function(inSender, inEvent) {
        alert("The answer is: " + inEvent.answer);
        return(true);
    }
});
```

Try it out: jsFiddle.

The inSender parameter is the kind that last bubbled the event (which may be different from the kind that originated the event). The inEvent parameter contains the data that was sent from the event. In the case of DOM events, this object contains a dispatch Target property that points to the Enyo control 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 inEvent.preventDefault().

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() function 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(inSender, inEvent) {
        this.$.display.setContent("Got it!");
    }
});

enyo.kind({
    name: "App",
    components: [
        { kind: "Signaller" },
        { kind: "Receiver" }
    ]
});

new App().renderInto(document.body);
```

Try it out: jsFiddle.

Like regular events, signal names are 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. 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.

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 published 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. Enyo does not have any mechanisms to enforce that separation, however code comments and documentation can serve to help other users of your kinds understand what is and isn't available to outside kinds. Enyo includes a documentation viewer that can process JavaScript files and pull out tagged documentation. For more information on this facility, read Documenting Code on the Enyo wiki.

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 functions from that existing kind. All kinds inherit from at least one other kind. The ultimate ancestor for 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: "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 override a function on a derived kind and wish to call the same named method on the parent, use the `inherited()` function. You may recall that we did this for the `create` function in the `Light` kind. You must always pass `arguments` as the parameter to the `inherited()` function.

Advanced Kinds

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

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);
    }
```

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

Statics

Enyo supports declaring functions that are defined on the kind constructor. These functions are accessed by the kind name rather than from a particular instance of the kind. Statics are often used for utility functions that do not require an instance and for variables 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,
        getCount: function() {
            return(this.count);
        }
    }
});
```

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
- Published properties, events, and handlers
- Kind variables
- Classes and styles
- Components

- Public functions
- Protected functions
- 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, I introduced 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, I'll focus on two important kinds that Enyo provides: controls and components. I'll also touch on some of the other kinds that you'll need to flesh out your app.

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 base app kind, will be based upon `Component` or one of its descendants.

Composition

Composition is a powerful feature that lets us focus on breaking down the functionality of our 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.

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

Component Functions

Components introduce `create()` and `destroy()` methods to asisst with the component's lifecycle. These functions can be overridden by kinds that derive from `Component` to provide extra functionality, such as allocating and deallocating resources. We previously used the `create()` function to initialize published properties on our `Light` kind. 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({});
    }
});
```

Try it out: jsFiddle.

We used the `destroy()` function to ensure that we cleaned up the timer we allocated in the `create()` function. You may also notice that we introduced a new function: `enyo.bind()`. In all our previous event handlers, Enyo handles making sure the context of the event handlers is 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 functions 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()`. A component can be removed from an instance of a component by calling its `destroy()` method. It is also possible to destroy all owned components by calling `destroyComponents()`. The following example shows creating a component dynamically:

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

new DynamicSample().renderInto(document.body);
```

Try it out: jsFiddle.

New controls are not rendered until requested. Call the `render()` function 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 library expands upon those basic controls and provides 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 our 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"`).

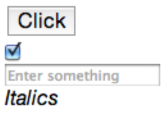
Core Controls

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(inSender, inEvent) {
        // React to taps
    },
    changed: function(inSender, inEvent) {
        // React to changes
    }
});

new ControlSample().renderInto(document.body);
```



Try it out: jsFiddle.

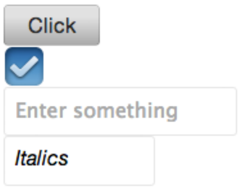
You will note that the controls themselves are unstyled, appearing with the browser's default style. We'll see how the Onyx versions of these controls compare next. You may also note that some controls use the `content` property to set the content of the control. The exceptions to this rule are the text field controls: `Input`, `TextArea`, and `RichText`. These controls use the `value` property to get and set the text content.

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(inSender, inEvent) {
        // React to taps
    },
    changed: function(inSender, inEvent) {
        // React to changes
    }
});
```



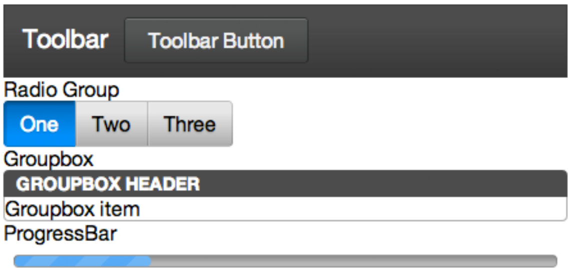
Try it out: jsFiddle.

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 `Drawer`, `ProgressBar`, `TabPanel`, and `TimePicker`, among others. Here's a sample of some of the new Onyx controls that show 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(inSender, inEvent) {
        // React to radio button activation change
    }
});

new OnyxSample().renderInto(document.body);
```



Try it out: jsFiddle.

Functions and Properties

Controls have a number of methods and properties that focus on their special role of interacting with the DOM. These methods include `rendered()`, `hasNode()`, and a number of methods for manipulating the DOM.

The first, `rendered()`, is a method that 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, we have used the `renderInto` function to tell Enyo to create the nodes associated with the kinds in our examples. As always, be sure to call the `inherited()` function within `rendered()`.

The second important method is `hasNode()`. This function 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 function is most useful when you are creating new controls that will need to manipulate the DOM or for 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()` functions:

```
enyo.kind({
    name: "Svg",
    published: {
        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" />';
        }
    }
});

new Svg({
    svg: "http://upload.wikimedia.org/wikipedia/commons/8/84/Example.svg"
}).renderInto(document.body);
```

Try it out: jsFiddle.

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 `Animator`, `Ajax`, and `JsonpRequest`.

`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.$.Expander.setContent("Presto");
        this.$.Animator.play();
    },
    expand: function(inSender, inEvent) {
        this.$.Expander.applyStyle("width", Math.floor(inSender.value) + "px");
    },
    done: function() {
        this.$.Expander.setContent("Change-O");
    }
});

new Expando().renderInto(document.body);
```

Try it out: jsFiddle.

`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(inSender, inResponse) {
        var output = "";
        for(i = 0; i < inResponse.length; i++) {
            output += inResponse[i].name + "<br />";
        }
        this.$.repos.setContent(output);
    }
});

new AjaxSample().renderInto(document.body);
```

Try it out: jsFiddle.

In this sample we used the GitHub API to fetch the list of repositories owned by enyojs, the user account that owns the Enyo repository. In the button's tap handler we created an Ajax object, populated it with the appropriate API url, and set the callback function for a successful response. We could have passed additional parameters for the service when we called the `go()` function. In general, we would have trapped error responses by calling `ajax.error()` with a context and error handling function.

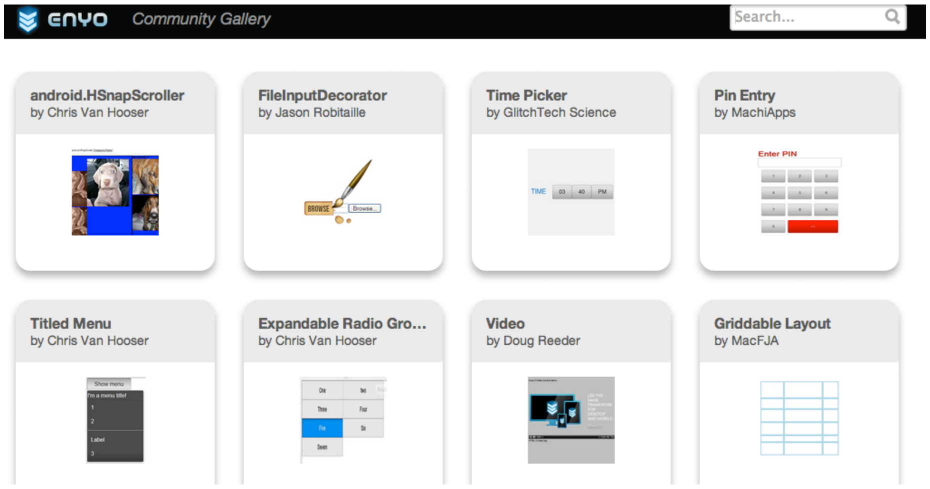
The Ajax object performs its request asynchronously so the call to `go()` does not actually cause the request to start. The request is not initiated until after the `fetch()` function returns.

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

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.

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 has to offer and learned a bit about using them. We also covered some non-visual objects Enyo provides. We'll take Enyo to the next level by exploring how to arrange these controls in the next chapter.

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.

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 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.

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).

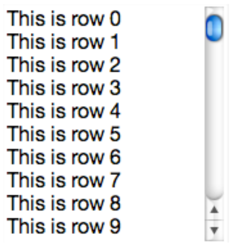
For more information on scrollers, visit the scroller documentation page.

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 function 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(inSender, inEvent) {
        var item = inEvent.item;
        item.$.text.setContent("This is row " + inEvent.index);
        return(true);
    },
    tapped: function(inSender, inEvent) {
        enyo.log(inEvent.index);
    }
});

new RepeaterSample().renderInto(document.body);
```


This is row 0
This is row 1
This is row 2
This is row 3
This is row 4
This is row 5
This is row 6
This is row 7
This is row 8
This is row 9

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 is the template for each row and can hold practically any component, though it is important to note that fittables 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()` function and pass in the index of that row.

To redraw the whole list, set a new value for the `count` property. Unlike most other properties, calling `setCount()` with the same value as before will trigger a new redraw. This is to handle the case of data sets that may contain the same number of items even though the data within those items has changed.

Layout Library Features

The modular Layout library includes several methods for arranging data within our apps. Three of the kinds we'll discuss are `Fittable`, `List`, and `Panels`. Visit the Enyo wiki online to find out more information on the Layout library and the kinds we didn't cover here.

Fittable

One area 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 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` published property to `true`.

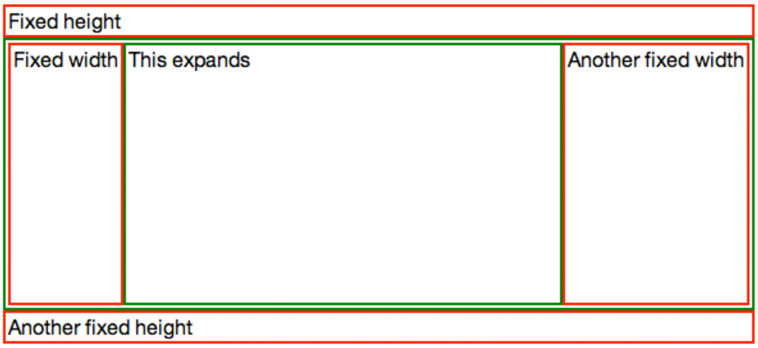
To apply the fittable style to controls, set the `layoutKind` property. To make it easier to use, the Layout library includes two controls with the layout already applied: `FittableColumns` and `FittableRows`. Fittables can be arranged within each other, as the following code sample shows:

```
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" }
    ]
});

new FittableSample().renderInto(document.body);
```

Try it out: jsFiddle.



In the previous sample, we used both styles of applying a fittable layout, using a `layout Kind` 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.

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 `resized()` function on the fittable component.

While fittables provide an easy solution to creating 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 allowed us to 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 simple controls and images be used within lists:

```
enyo.kind({
    name: "ListSample",
    kind: "List",
    count: 10000,
    handlers: {
        onSetupItem: "setupItem",
        tap: "tapped"
    },
    components: [{ name: "text" }],
    setupItem: function(inSender, inEvent) {
        this.$.text.setContent("This is row " + inEvent.index);
        return(true);
    },
    tapped: function(inSender, inEvent) {
        enyo.log(inEvent.index);
    }
});

new ListSample().renderInto(document.body);
```

Try it out: jsFiddle.

In both this example and the `Repeater` example we knew the number of items we wanted to display so we 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 call `setCount()` once you have received the data. Once set, the `List` will render itself.

In order to make a `List` row interactive, you must first use the `prepareRow` function. 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 we used to see how we might modify 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(inSender, inEvent) {
        this.$.text.setValue(this.items[inEvent.index]);
```

```
            return(true);
        },
        tapped: function(inSender, inEvent) {
            this.prepareRow(inEvent.index);
            this.$.text.setValue(this.items[inEvent.index]);
            this.$.text.focus();
            return(true);
        },
        changed: function(inSender, inEvent) {
            this.items[inEvent.index] = inSender.getValue();
        },
        blur: function(inSender, inEvent) {
            this.lockRow();
        }
    });

    new ListSample().renderInto(document.body);
```

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 woud 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.

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 `arranger Kind` 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` introduce the concept of an 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 an important concept. You can easily transition the active panel by using the `previous` and `next` function, 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: "FittableRows",
    components: [
        { kind: "Panels", fit: true, arrangerKind: "CollapsingArranger",
          classes: "panels-sample", narrowFit: false,
          components: [
              { name: "panel1", style: "background-color: blue" },
              { name: "panel2", style: "background-color: grey" },
              { name: "panel3", style: "background-color: green" }
          ]
        }
    ]
});

new PanelsSample().renderInto(document.body);
```

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%;
    }
}
```

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 changed that default so we could 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 panels left and right to reveal the other panels.

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 should now be able to start producing beautiful and high performance apps that run on mobile and desktop platforms. We explored some of the features necessary to design responsive apps that make the best use of a user's display size. In the next chapters we'll learn about the tools Enyo developers can use to polish, package, and deploy their applications.

CHAPTER 5
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 diagnose things when bugs occur. As always, we'll explore these concepts through interactive samples.

Styling

Enyo provides some very nice looking controls with the Onyx library. 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 the Onyx widget set. We'll explore several ways to accomplish that.

Styles and Classes

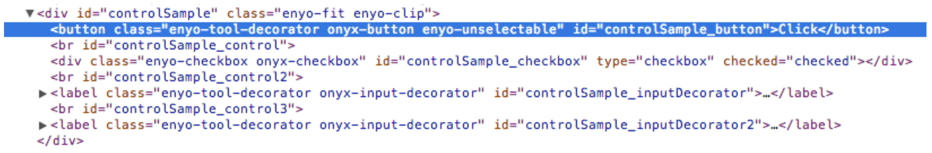
All Enyo controls have two published properties to aid in styling: `style` and `classes`. These two properties correspond to an HTML element's `style` and `class` attributes. The `style` property can be used to apply a specific style to a single control. To work with the `classes` property, you must add CSS classes to a style sheet. In general, it is better to use `classes` in an app for two reasons: a component is more reusable if a style is not embedded within it and by using a CSS class you can modify the style from a single place.

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 upating classes, Enyo provides `addClass()`, `removeClass()`, and `addRemoveClass()`.

It might seem like `setStyle()` and `setClasses()` would be good methods to call to update the style on a control, however, these two functions completely replace the styles and classes of the control.

Overriding Onyx Styles

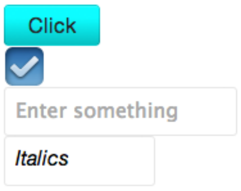
Each Onyx control includes one or more classes. It is possible to override some (or all) of the default styling by overriding those styles in your CSS file. One simple way to discover the class names to override is to use your browser's inspector to see what classes are applied to a particular control. 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`:

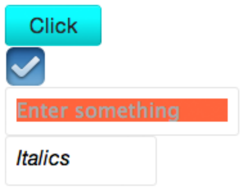
```
.onyx-button {
    background-color: cyan;
}
```



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 app 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;
}
```



Try it out: jsFiddle.

In general, you'd actually style the input decorator rather than the input.

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 that Onyx uses. 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).

LESS can be used "live" in a browser. The debug build of Bootplate projects can load 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. To enable this, uncomment the line in *debug.html* that includes the less libary (e.g., "…less-1.3.0e.min.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/package.js*, uncomment the reference to *Theme.less*.
- Edit *source/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.

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 app perception is 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.

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. Sometimes things don't always 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 on 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.

One of the most common errors in Enyo apps is forgetting to call `this.inherited(arguments)` when overriding functions on a parent object. This occurs most often with the `create()` and `render()` functions but can also happen with others. Forgetting to make this call can cause components to not appear at all or can cause them to render incorrectly. Another common error is forgetting 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. Even with these features, things can turn out wrong. One common problem app developers experience is failing to provide a height to `List` or `Scroller` components. Without a height, these elements will end up invisible. Providing a height or including them within a fittable layout can solve that issue. Fittables themselves can cause problems. Forgetting to assign `fit: true` to one and only one of the components of a fittable component can lead to rendering issues as well.

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 applied to the elements in question. Sometimes, an issue such as CSS precedence caused the problem. It can also help to see if a component rendered at all or if it's hidden somehow. Other times, it can help to add `!important` to a style.

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()` functions 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 function that issued the log message.

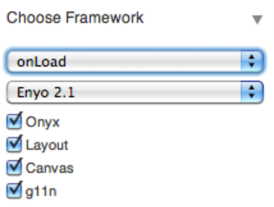
Sometimes a passive approach to debugging a problem isn't enough. In these cases you can set breakpoints in your code and step through functions that are misbehaving. A handy trick that is supported by the major browsers is putting the `debugger` command into code you want to inspect. When the browser 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.

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, check the code wrap settings at the top to make sure you have selected `onLoad` and the correct framework:



Choose Framework

onLoad

Enyo 2.1

[✓] Onyx
[✓] Layout
[✓] Canvas
[✓] g11n

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

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

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 modular library to handle some of the globalization issues you'll face. The library is called `g11n`, which is a shorthand way to write "globalization" (g, followed by eleven letters, followed by n). This library provides facilities for substituting translated strings as well as formatting names, dates, and other measures based upon a user's locale. It even supports loading CSS based upon locale.

The globalization library will attempt to figure out the locale based on cues from the browser. The current locale can be retrieved by calling `enyo.g11n.currentLocale()`. In cases where the locale can't be determined, you can explicitly set it using `enyo.g11n.setLocale()`.

The g11n library is not included if you are using a Bootplate-based setup (see Appendix A). To enable the library in a Bootplate setup, execute the command: `git submodule add` *`https://github.com/enyojs/g11n.git`* `lib/g11n` and then add the g11n library to your *package.js* with the line `$lib/g11n`.

Globalization Basics

In its most basic form, the globalization library 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 *assets/resources* directory of your app. Each translation should be in its own JSON file. Translation files are named starting with language and optionally adding country code and variant. For example, a Canadian English translation file would be named *en_ca.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 globalization library includes modules to format names, phone numbers, dates, times, and more. For more information on the countries supported by these functions, please refer to the localization documentation page. The name, phone, and address modules included with g11n are not loaded by default. The following example shows some of the basic routines for formatting these items:

```
enyo.kind({
    name: "G11nSample",
    components: [
        { name: "date" },
        { name: "number" }
    ],
    create: function() {
        this.inherited(arguments);
        var dateFmt = new enyo.g11n.DateFmt({ date: "short" });
        this.$.date.setContent(dateFmt.format(new Date()));
        var numFmt = new enyo.g11n.NumberFmt({ fractionDigits: 1 });
        this.$.number.setContent(numFmt.format("86753.09"));
    }
});

new G11nSample().renderInto(document.body);
```

Try it out: jsFiddle.

Two Onyx components are locale-aware: `DatePicker` and `TimePicker`. If the globalization library 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.

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 6

Deploying

So now you're a budding Enyo developer looking to deploy your app to all the platforms supported by Enyo. 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. We'll now have to set up a "real" development environment since we're not likely to deploy our apps by directing users to a jsFiddle page.

For this chapter I'll assume you've followed the Bootplate environment setup guide in Appendix A. You will be able to apply some of these tools to other setups.

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. For web deployment, simply copy the Enyo library and app source code up to a directory on your server and include them into your HTML source.

Bootplate makes this process easy by including a `deploy` script that packages all the files and minimizes the source. This process speeds up loading and combines everything into a single directory. Once packaged, deploying is as simple as transferring the files to your host. Keep in mind that deployed code is much tougher to debug than debug code.

Desktop Targets

If you want to create an installable Windows app, then Intel's AppUp encapsulator makes it easy. Visit their site, fill out the fields in the "Make your app" tab, and then upload a *.zip* file of the contents of your deployment directory. In moments you will have an installable file that can be included in Intel's AppUp app center.

On the “Advanced (APIs etc)” tab you will find information on APIs that AppUp makes available to your app. Using these, you can extend your app to take advantage of native features and in-app purchases through the AppUp center.

Mobile Targets

By far the most interesting place for apps these days is on smartphones and tablets. Enyo is perfectly suited to this environment. Enyo 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. PhoneGap, which is based on Apache’s open source Cordova project, handles direct access to these features and provides a method for creating natively installable apps.

Enyo has a PhoneGap support package available in the *extra* repository at GitHub. For more information on Cordova support with Enyo, please see PhoneGap Native Functions on the Enyo wiki.

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.

After registering, 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/bootplate* 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.

Local PhoneGap Builds

PhoneGap Build is easy to use but it doesn’t give you a lot of flexibility. Installing PhoneGap locally gives you much finer-grained control and access to using 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.

For more information on getting started with PhoneGap, visit the Getting Started Guides page.

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 the various mobile and desktop platforms and know that your apps will work.

CHAPTER 7
Conclusion

By now you should be up and running with Enyo. You've seen the major features of Enyo 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 (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. At the time of this writing, a data-binding framework was being developed and is available in a *beta* form on GitHub. 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're going to need to set up a local or server copy of Enyo, 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

There are a couple of tools that Enyo makes use of that, depending on your needs, will be required. These tools include Node.js and Git. We'll cover why you'll need those and where to get them.

Node.js

Node.js is a platform for running JavaScript outside of a browser. It can 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 for compiling LESS files into CSS. 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 for software developers to keep revisioned 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 you wish to be able to keep up with the latest developments with Enyo, wish to contribute to Enyo, or you want to use a system that makes it easy to keep past revisions 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 several methods for installing Enyo. One of the easiest ways 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 other methods for installing Enyo.

Bootplate

Bootplate is a scaffold upon which to build an Enyo app. It includes tools that allow you easily debug your app in a browser and then deploy a minified version of Enyo with your app. There are two ways to install Bootplate: by downloading a zipped archive from the Enyo site and by cloning 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 2.1.1. After downloading, simply unzip the archive.

To download Bootplate from GitHub, perform the following steps:

```
git clone https://github.com/enyojs/bootplate.git
cd bootplate
git submodule update --init
```

Full Source

You can 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 in the same directory. 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
   ...
```

Enyo npm Module

Another method to quickly create Enyo apps is to use the Enyo npm module. npm is a package manager for Node.js. npm is installed with the latest versions of Node. To install the Enyo package, issue the following command:

```
npm install -g enyo
```

Once the Enyo npm module is installed you will have a new command-line tool that will create new Enyo apps for you. It also includes a debug server built with Node. To initialize a Bootplate project, use the following command within an empty directory:

```
enyo init:bootplate
```

You will be prompted to answer several questions about your new app. Once complete, you will have a project ready to test. To start the test server, use:

```
enyo debug
```

This creates a server on port 8000. You can test your app by navigating your browser to *http://127.0.0.1:8000/debug.html*.

Using Bootplate

Bootplate gives you a head start in creating your app by providing a ready-to-use structure for your app. It provides a *source* directory that contains *App.js* and *App.css*, which you can edit with your own source. 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 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/bootplate* directory. The contents of that directory can be copied up to a web server or packaged with one of the various packaging tools (see Chapter 6).

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 debug server built into the npm module.

About the Author

Roy Sutton is the Open webOS Community Manager for 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. He lives with his wife and son in Northern Virginia and you can find him on Twitter as @Pre101.

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.