Getting Started with Twitter Flight

Build scalable, modular JavaScript applications with the Twitter Flight framework

Foreword by Dan Webb, Web Core Engineering Manager at Twitter

Tom Hamshere

[PACKT] open source*
PUBLISHING   community experience distilled

Getting Started with Twitter Flight

Build scalable, modular JavaScript applications
with the Twitter Flight framework

Tom Hamshere

[PACKT] open source *
PUBLISHING   community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Twitter Flight

Credits

Author
Tom Hamshere

Reviewers
Cameron Hunter
Katsuya Noguchi
Andrey Popp
Simon Smith
Veturi JV Subramanyeswari

Acquisition Editors
Sam Birch
Andrew Duckworth

Commissioning Editor
Subho Gupta

Technical Editors
Pratik More
Anusri Ramchandran

Project Coordinator
Amey Sawant

Proofreader
Jonathan Todd

Indexer
Hemangini Bari

Graphics
Yuvraj Mannari

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

Foreword

You might not be able to tell by using it but the frontend of twitter.com is incredibly complex. Back in 2011, this complexity was growing at an exponential rate as we added features. Our client- side code was essentially a proto MVC application, built long before frameworks such as Backbone.js and Ember.js had become popular, so was far from spaghetti code but, nevertheless, the multitude of relationships between these models, views, and controllers was almost impossible for one engineer to fully understand.

In order to make a change in one part of the code base, our engineers needed to understand every reference made to that piece of code in the application. If JavaScript was statically typed, we'd have been able to rely on that to help us, as it was all too easy to make a trivial update to one part of the application but later, much, much later, we discovered that this had broken a seemingly unrelated part of the code base. I'm sure you've been there, done that, and lost your mind more than once.

This was the situation that inspired us to create Flight.js. We wanted to be able to create the complex client-side interactions that power great web applications while retaining the simplicity of our code. Flight.js has massively improved the testability, robustness, and maintainability of many of our web applications and now, with the help of this book, we think you can, maybe, hopefully, save a little bit of sanity too.

Dan Webb
Web Core Engineering Manager at Twitter

About the Author

Tom Hamshere is based in North London and has been a frontend developer since 1998, working on a wide variety of projects. He has been a part of the Twitter UK flock since May 2012, focusing primarily on TweetDeck, an awesome JavaScript Twitter client. He was the first developer to implement Flight outside of `www.twitter.com`, and is responsible for a number of Flight-related open source projects, including jasmine-flight and flight-keyboard-shortcuts.

When not hacking away at the codeface, Tom enjoys gardening, cooking, skiing, and traveling.

> I'd like to thank the whole team at TweetDeck for their insights on JavaScript and application development, and the Flight team at Twitter HQ, not only for producing Flight (which is a pleasure to work with) but also for giving me their support and the opportunity to write this book.

About the Reviewers

Cameron Hunter is a developer hailing from Belfast, Northern Ireland. An advocate for the web as an open platform, he spends his time building products on Twitter's web team internally, and creating and contributing to open source projects externally. His work ranges across the frontend/backend scale from JavaScript and CSS to Scala and Java, the latter of which he worked with in his previous life at Amazon.

When away from glowing screens, Cameron dedicates time to pints of craft beer, climbing rocks, and geeking out on board games.

Katsuya Noguchi is a software engineer living in the San Francisco Bay area. After obtaining an MSc in Computer Science from Oxford, he started his career with Twitter, Inc., where he worked on a variety of projects, including internationalization, translation reputation system, user suggestion system, and tweet translation service. He now works at Gumroad, Inc. At Gumroad, he focuses on frontend operation, where he gained significant experience in Twitter Flight by refactoring entire frontend code to make it more maintainable, scalable, and testable. He also has a few open source contributions, such as activerecord-reputation-system and jQuery.bank.

Andrey Popp is a software hacker based in Moscow, Russia. While not hacking on a new computing architecture he is busy experimenting with human-computer interaction paradigms and contributing to open source software projects.

Simon Smith lives and works in London and has been a frontend developer for more than six years. When not drinking copious amounts of tea, he can be found obsessing over mobile first responsive design and JavaScript.

He has worked with companies large and small, most notably for the BBC where he led frontend development on a rebuild of the Radio Times website.

Simon writes about various frontend-related topics at `www.simonsmith.io`.

Veturi JV Subramanyeswari is currently working as a Solution Architect at a well-known software consulting MNC in India. Prior to joining this company, she served a few Indian MNCs, many startups, and R&D sectors in various roles such as programmer, tech lead, research assistant, and Architect. She has around 10 years of working experience in delivering a diverse variety of projects, utilizing the latest cutting-edge technologies in web/mobile areas covering media and entertainment, retail, publishing, healthcare, enterprise architecture, manufacturing, public sector, defense communication, gaming, and so on. She has reviewed other tech books including:

- Drupal Rules
- DevOps
- Twitter Bootstrap
- Salesforce CRM
- Drupal 7 Multi Sites Configuration
- Building Powerful and Robust Websites with Drupal 6
- Drupal 6 Module development
- PHP Team Development
- Drupal-6-site-blueprints
- Drupal 6 Attachment Views
- Drupal E-Commerce with Ubercart 2.x
- Drupal 7: First Look
- Drupal SEO

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

JavaScript development has come a long way in recent years, emerging from a miasma of inline scripts and a chain of callbacks to embrace application-level programming, through a variety of frameworks that have managed to corral its unruly nature, imposing structure where seemingly none existed.

However, these frameworks often seem to subtract from the experience of JavaScript, providing heavy APIs and requiring extensive boilerplate code.

Flight, with the same goal in mind, takes a different path. It builds on a structure that forms the heart of any web page or application: the DOM. Flight provides atomic components that are joined together as an infinite lattice, adding functionality without increasing complexity, and allowing for truly scalable applications.

In this book, I aim to provide a working knowledge of Flight to both seasoned application developers and novices coming from a traditional JavaScript development background.

I was able to write this book from a unique perspective, having been given the opportunity to work with Flight on a major JavaScript application (TweetDeck) before its initial open source release, and as a part of the Twitter organization, with direct access to those responsible for designing Flight, the Twitter Web Core team.

Their ongoing support, advice, and understanding of the intentions behind Flight has hopefully led this book to be more than just a technical manual, allowing me to dig deep into how to design applications that conform to Flight's ideals.

Thanks go to Dan Webb, Angus Kroll, Todd Kloots, and Kenneth Kufulk for their help and advice, and also to Sol Plant who was my partner in introducing Flight to TweetDeck; to Nicolas Gallagher for his work on the Yo Flight Generator; to Cameron Hunter, Katsuya Noguchi, Simon Smith, and Andrey Popp for helping to review the book; and to Emma Dingle for being there when the going got tough.

What this book covers

Chapter 1, *What is Flight?*, covers the basics of how Flight works and the problems it aims to solve.

Chapter 2, *The Advantages of Flight*, details Flight's advantages over other frameworks. This includes its shallow-learning curve, reliability, reusability, agnostic architecture, performance, and the idea of well-organized freedom.

Chapter 3, *Flight in the Wild*, captures Flight's use in the real world before and after its release, by providing examples of applications and open source projects using Flight.

Chapter 4, *Building a Flight Application*, explains how to scaffold a Flight application with the Yeoman Flight Generator and walks through the resulting application structure.

Chapter 5, *Components*, aims to provide an overview of what components are and how to build them.

Chapter 6, *UI Components*, provides examples on how to listen for browser events, how to access elements within a component, and the use of defaults and settings within components.

Chapter 7, *Data Components*, deals with how data components differ from UI components, how to create them, attach them to the DOM, and how to handle UI events and trigger data events.

Chapter 8, *Event Naming*, discusses the importance of good event names, what an event really is, and also provides an example naming convention.

Chapter 9, *Mixins*, focusses on what mixins are, when to use them, and how to create them. It also covers Advice, a mechanism used to override or extend existing methods.

Chapter 10, *Templating and Event Delegation*, discusses various templating approaches in Flight, providing examples of DOM node templating, client-side templating with Hogan, and server-side templating with Grunt. Also covered is how to use generated HTML to determine state and event-delegation for dynamic HTML.

Chapter 11, *Web Application Performance*, shows how Flight can be used to render pages efficiently and how to work around latency in Ajax requests.

Chapter 12, *Testing*, provides an example BDD test written in Jasmine, explains what to focus on while testing components, how to gain references to component instances, and how to extend existing test frameworks to handle components, mixins, and events.

Chapter 13, *Complexities of Flight Architecture*, deals with the problems with nesting components and how to better define components to avoiding nesting.

Appendix, *Flight API Reference*, gives an overview of the essential API methods including how to create components and mixins, how to use Advice, listen for and trigger events, and define default attributes.

What you need for this book

To follow the examples in this book, you should have a working computer and a text editor. All the software required for installing and running Flight is covered within the book.

Who this book is for

This book is for anyone who wants to build Flight components, mixins, and applications. No previous knowledge of Flight or other application frameworks is required. A good understanding of JavaScript is required and any knowledge of jQuery and AMD will help, though is not essential.

Conventions

In this book, you will find a number of styles of text that distinguishes between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Generally, data components only trigger a single data event, for example, `dataTask`, `dataTags`."

A block of code is set as follows:

```
define(function (require) {
  // import dependencies
  var defineComponent = require('flight/lib/component');
```

```
    // export component constructor
    return defineComponent(helloWorld);
    // component definition
    function helloWorld () {};
});
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
define(function (require) {
  var defineComponent = require('flight/lib/component');
  var withTextUtils = require('component/with_text_utils');
  // mixin other mixins
  var withLocalStorage = function() {
compose.mixin(this, [withTextUtils]);
  };
});
```

Any command-line input or output is written as follows:

```
mkdir flight-example && cd $_
yo flight example
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Press *cmd / Ctrl + Alt + J* to open **Console** in Chrome."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
What is Flight?

Flight is an open source, lightweight, component-based JavaScript framework, designed to create elegant JavaScript applications.

Who made Flight?

Flight was created at Twitter by Angus Croll, Dan Webb, and Kenneth Kufluk, along with other members of the Twitter Web Core team. A full list of the team involved in the development of the open source project is available on Flight's GitHub page at `https://github.com/flightjs/flight`.

Flight was created to provide a reliable, extensible framework for web applications, specifically to address performance and maintenance issues on the `twitter.com` website.

It is designed to obviate the need for a rigid model or view and the boilerplate that entails, allowing rapid development without sacrificing maintainability.

How does Flight work?

Flight piggybacks on DOM to provide application structure and utilizes DOM events to act as an interface between components. A simple API provides constructors for components and mixins and access to DOM nodes and events.
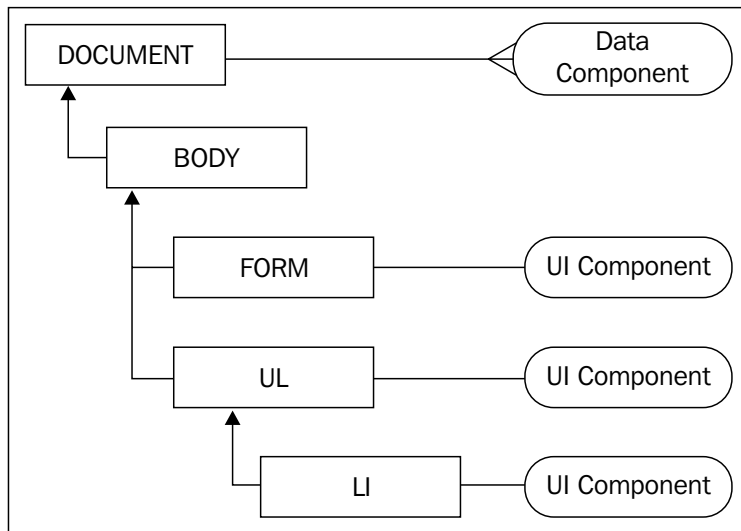
Event-driven interfaces

Component instances are attached to DOM nodes (known as the component's root node). Events triggered within a component are initiated on the component's root node and bubble up the tree using the standard HTML event model.

By default, a component listens to events on its root node. As all events received on that node must have come from either the component's root node or from nodes within its DOM subtree, it can be sure that the event is relevant to itself (that is, all events triggered on nodes within a particular form element would be relevant to the component controlling the form).

Components may listen to events on specific nodes within the component's DOM subtree, but outside of that, it can only listen to events on either the document or the window and has no knowledge of the rest of the DOM.

Scalable architecture

If a component was to trigger an event on its root node, components attached to nodes further down the DOM tree would not hear that event. Thus, if we were to attach a component to a specific input element (child) within a form (parent), the input would not receive events triggered on the form, making it difficult to keep the child informed.

The best solution to this problem is to rethink the whole **parent-child** concept. In Flight, components are not intended to instantiate other components. Application structure in Flight is almost entirely flat with no parent-child relationships.

No parent-child relationships

This may seem counterintuitive at first, but is in fact a major feature in making Flight scalable. By preventing parent-child structures, each component becomes truly independent and reusable. No component relies on another to do its job.

No spaghetti code

Whether through a parent-child relationship or through a direct use of an exposed API, referencing one module's methods from within another is a major cause of problems when refactoring large applications. It is all too easy to end up with spaghetti code, a complex web of interdependent modules, as seen in the following figure:
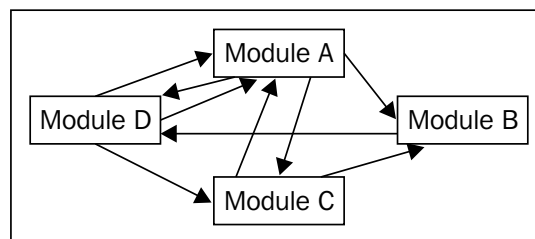
For example, a theoretical module named `TaskManager` may provide a method, `get`, to return a specific task. `TaskManager.get()` might make sense, but think about what happens when you change the input/output of the `get` method. How do you find every reference to it? How many other methods named `get` are there? Is it clear that a particular method belongs to the `TaskManager` module? This can become especially problematic when instances are assigned inconsistent variable names such as `taskManager`, `taskMan`, `tm`.

It is possible to obviate some of these problems with good method naming and a good IDE but these are by no means ideal solutions.

Flight offers an alternative.

Promoting reusability with well-defined interfaces

In Flight, each component has a clear, well-defined interface and is essentially unaware of the existence of the rest of the application. A component acts as an independent entity and will continue to work as intended if all other components on the site are removed.

These simple, well-defined interfaces can be exhaustively tested, ensuring that each component behaves in a predictable manner. This makes it possible to rapidly develop new features by reusing existing components and mixins, safe in the knowledge that they will behave as expected in their new environment.

The missing model

Generally, JavaScript frameworks employ a strict data model to facilitate reliable transfer of data from the data store to the UI. The data store, data processors, and the view are all dependent on knowledge of this rigid structure. In an attempt to create a neat separation of model and view, the two have, in fact, been inextricably intertwined.

Simplification

In Flight, there is no concept of a model or view. Instead, data components manage the interaction between the data store and the UI, listening for UI events, and producing data events containing only what the UI requires.

UI components in Flight can be thought of as behavioral templates. In the same way that a **Mustache** template renders HTML, UI components render behavior.

Reducing boilerplate

The data required to determine behavior comes in the form of simple object literals with no classes and no rigid structure. This results in considerably less boilerplate code than equivalent frameworks and a total separation of data from behavior.

The missing model is not a hardship but rather an ideal.

Summary

Flight offers a minimal API, uses the DOM to provide structure, and DOM events as an interface between components. It uses a flat structure to reduce complexity and create scalable applications. It avoids rigid model-view relationships, reducing the amount of boilerplate code required.

Flight gives developers the freedom to use a variety of best practice approaches in a well-organized environment.

In the next chapter, we will see why Twitter developed Flight and also take a look at why you may want to use it on a project.

2
The Advantages of Flight

The number of JavaScript frameworks available today can be overwhelming, and when it comes to choosing one on which to base your application, you need to make the right decision. So, why choose Flight?

This chapter details Flight's advantages over other frameworks. This includes its shallow-learning curve, reliability, reusability, agnostic architecture, performance, and the idea of well-organized freedom. At the end of the chapter, you can find some specific scenarios such as single-page apps and classic web pages.

Simplicity

First and foremost, Flight is simple. Most frameworks provide layouts, data models, and utilities that tend to produce big and confusing APIs. Learning curves are steep.

In contrast, you'll probably only use 10 Flight methods ever, and three of those are almost identical.

All components and mixins follow the same simple format. Once you've learned one, you've learned them all.

Take a look at the "Hello, World!" example component in *Chapter 5*, *Components*, to see what a simple component might look like.

Simplicity means fast ramp-up times for new developers who should be able to come to understand individual components quickly.

Efficient complexity management

In most frameworks, the complexity of the code increases almost exponentially with the number of features. Dependency diagrams often look like a set of trees, each with branches and roots intermingling to create a dense thicket of connections. A simple change in one place could easily create an unforeseen error in another or a chain of failures that could easily take down the whole application.

Flight applications are instead built up from reliable, reusable artifacts known as components. Each component knows nothing of the rest of the application, it simply listens for and triggers events. Components behave like cells in an organism. They have well-defined input and output, are exhaustively testable, and are loosely coupled.
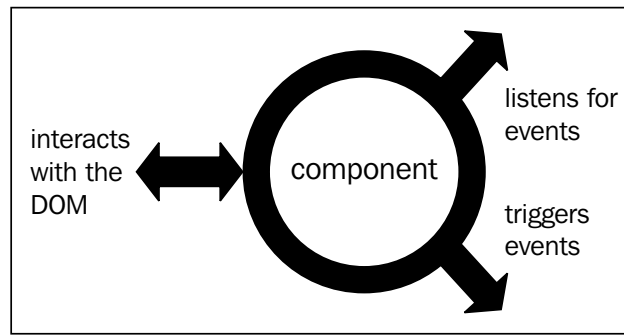
A component's cellular nature means that introducing more components has almost no effect on the overall complexity of the code, unlike traditional architectures. The structure remains flat, without any spaghetti code.
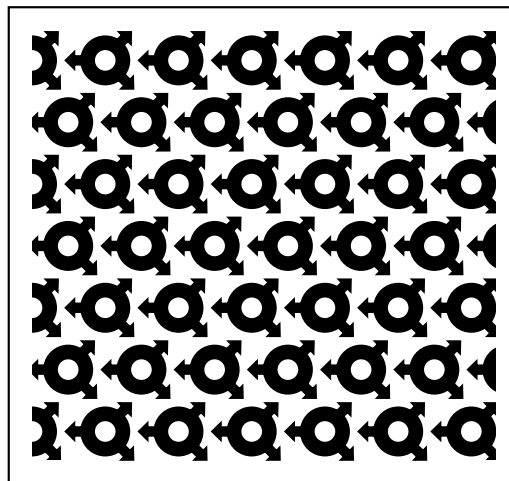
This is particularly important in large applications. Complexity management is important in any application, but when you're dealing with hundreds or thousands of components, you need to know that they aren't going to have unforeseen knock-on effects.

This flat, cellular structure also makes Flight well-suited to large projects with large or remote development teams. Each developer can work on an independent component, without first having to understand the architecture of the entire application.

Reusability

Flight components have well-defined interfaces and are loosely coupled, making it easy to reuse them within an application, and even across different applications.

This separates Flight from other frameworks such as **Backbone** or **AngularJS**, where functionality is buried inside layers of complexity and is usually impossible to extract.

Not only does this make it easier and faster to build complex applications in Flight but it also offers developers the opportunity to give back to the community.

There are already a lot of useful Flight components and mixins being open sourced. Try searching for "flight-" on **Bower** or **GitHub**, or check out the list at `http://flight-components.jit.su/`.

Twitter has already been taking advantage of this reusability factor within the company, sharing components such as **Typeahead** (Twitter's search autocomplete) between `Twitter.com` and **TweetDeck**, something which would have been unimaginable a year ago.

Agnostic architecture

Flight has agnostic architecture. For example, it doesn't matter which templating system is used, or even if one is used at all. Server-side, client-side, or plain old static HTML are all the same to Flight.

Flight does not impose a data model, so any method of data storage and processing can be used behind the scenes.

This gives the developer freedom to change all aspects of the stack without affecting the UI and the ability to introduce Flight to an existing application without conflict.

Improved Performance

Performance is about a lot more than how fast the code executes, or how efficient it is with memory. Time to first load is a very important factor. When a user loads a web page, the request must be processed, data must be gathered, and a response will be sent. The response is then rendered by the client. Server-side processing and data gathering is fast. Latency and interpretation makes rendering slow.

One of the largest factors in response and rendering speed is the sheer amount of code being sent over the wire. The more code required to render a page, the slower the rendering will be. Most modern JavaScript frameworks use deferred loading (for example, via **RequireJS**) to reduce the amount of code sent in the first response. However, all this code is needed to be able to render a page, because layout and templating systems only exist on the client.

Flight's architecture allows templates to be compiled and rendered on the server, so the first response is a fully-formed web page. Flight components can then be attached to existing DOM nodes and determine their state from the HTML, rather than having to request data over **XMLHttpRequest** (**XHR**) and generate the HTML themselves.

Performance, server-side rendering, and using the DOM to determine state are covered in more detail in *Chapter 11*, *Web Application Performance*.

Well-organized freedom

Back in the good old days of JavaScript development, it was all a bit of a free for all. Everyone had their own way of doing things. Code was idiosyncratic rather than idiomatic. In a lot of ways, this was a pain, that is, it was hard to onboard new developers and still harder to keep a codebase consistent and well-organized.

On the other hand, there was very little boilerplate code and it was possible to get things done without having to first read lengthy documentation on a big API.

jQuery built on this ideal, reduced the amount of boilerplate code required. It made JavaScript code easier to read and write, while not imposing any particular requirements in terms of architecture.

What jQuery failed to do (and was never intended to do) was provide an application-level structure. It remained all too easy for code to become a spaghetti mess of callbacks and anonymous functions.

Flight solves this problem by providing much needed structure while maintaining a simple, architecture-agnostic approach. Its API empowers developers, helping them to create elegant and well-ordered code, while retaining a sense of freedom.

Put simply, it's a joy to use.

Summary

The Flight API is small and should be familiar to any jQuery user, producing a shallow learning-curve. The atomic nature of components makes them reliable and reusable, and creates truly scalable applications, while its agnostic architecture allows developers to use any technology stack and even introduce Flight into existing applications.

In the next chapter, we take a look at how Flight is already being used to build applications and extended to create new frameworks.

3

Flight in the Wild

This chapter covers Flight's use in the real world before and after its release. It also provides examples of applications and open source projects using Flight in an attempt to show what is possible with this flexible framework.

Flight at Twitter

Flight is used by Twitter on two high-profile products: `twitter.com` and TweetDeck. Despite outward appearances, both these products are great examples of single-page applications. `twitter.com` gives the impression of a multipage site, but underneath that facade lies a JavaScript application, creating the appearance of pages while maintaining continuous data layer at the document level.

Flight was originally created to solve an interesting problem on `twitter.com`. The site had recently switched from a multipage, HTML-driven site to a single-page, JavaScript-driven application, and was suffering from a variety of problems as a result, including slow load times, poor accessibility, and high maintenance costs.

TweetDeck was much in the same boat. A rapid development process and a growing development team had led to a variety of coding styles being implemented, making the application hard to understand and develop on.

So, why was Flight the answer?

Better performance

Poor performance in single-page applications is a result of various factors including time to first load, template rendering, memory leaks, and plain old bad code.

Flight provided `twitter.com` with a fast, lightweight, efficient framework that allowed pages to load faster using server-side rendering, and by allowing components to be torn down cleanly between pages avoided hogging of memory.

A manageable codebase

With a development team as large as Twitter's, creating consistent, maintainable code is a must. With a well-organized code structure, well-tested interfaces, and consistent style, it becomes possible to rapidly onboard new developers, refactor existing code, and have everyone understand how things work.

Flight components can be exhaustively tested, have clearly defined interfaces, and are easy to read, making them ideal for the job. The lack of complex interdependencies between components creates a simple, reliable structure that is easy to maintain.

Quotes from developers

The following quotes are taken from various developers discovering Flight:

On refactoring

"…one of the great things about Flight: ease of refactoring. One piece of advice I can offer is to not worry too much about getting everything right the first time around.

We spent quite some time at the outset considering how … components would behave,… (but) we realised we needn't have bothered. It's desperately easy to alter a component to be a mixin, or the other way around. It's simple to change the way a component works internally because nothing else cares. It's dead simple to break up a data component into lots of little components because nothing is talking to it directly."

> *— TweetDeck Blog*

"We are a very small team and didn't want to spend too much time refactoring existing code. Before deciding to use Flight, we tried a few other alternatives. Flight ended up being the most appealing to us because 1) it is very lightweight, and 2) it is agnostic towards client-server communication. The latter property allowed us to refactor our JavaScript code without changing any server-side code, which would have taken us much longer."

> *— Gumroad Blog*

On Flight's component architecture

"Flight is also inherently modular; this leads to code getting very organized without any effort at all. It is also strongly in accordance with the DRY philosophy; Flight components can be attached to multiple DOM elements, Flight mixins can be added to multiple components, A single Flight component can have multiple mixins added to it."

— Ameya Karve, developer of Icarus

Open source Flight projects

A number of open source projects based on Flight have sprung up since its release. Presented here are a few that show Flight's flexibility and potential:

TodoMVC

The first question I get asked when talking about Flight is whether there's a **TodoMVC** implementation. In case you've not heard of it, TodoMVC provides examples of the same todo app written with various frameworks. Thanks to *Michal Kulkis*, one is available at:

`http://todomvc.com/dependency-examples/flight/`

Components for web applications

Developers have already created some useful, open source Flight components, ready for you to drop into your own projects. Here are a few examples from *Cameron Hunter*, *Andy Hume*, and myself:

- Manage device orientation with flight-orientation available at: `https://github.com/cameronhunter/flight-orientation`.

- Access the HTML5 geolocation API with flight-geolocation available at: `https://github.com/cameronhunter/flight-geolocation`.

- flight-storage uses various methods to store data including the Local Storage, HTML attributes, and cookies. It is also available at `https://github.com/cameronhunter/flight-storage`.

- Create click traps for modals with flight-click-trap which is available
  at: `https://github.com/ahume/flight-click-trap`.

- Manage keyboard shortcuts including sequences and combos with
  flight-keyboard-shortcuts. Have a look at:
  `https://github.com/tbrd/flight-keyboard-shortcuts`.

Extending Flight with two-way data binding

A number of recent JavaScript frameworks, including **Ember.js** and AngularJs, use two-way data binding. This refers to the binding of UI and data object together: making a change on the UI immediately updates the data, and vice versa.

There are various ways of achieving this in Flight. One early implementation is **Icarus** by *Ameya Karve*, an extension of the Flight framework that adds two-way binding functionality using **Laces.js**. Have a look at the following site and the quote that follows by *Ameya Karve*: `https://github.com/ameyakarve/Icarus`

> "…*The philosophy behind writing Icarus was to combine the best of both worlds. Icarus components are completely independent, and can communicate with one another, and with themselves using events. Icarus components also provide Models, which are similar to Models used in typical MV* frameworks…*"

Summary

Flight has already proven itself in large applications. The teams at TweetDeck and Twitter are both very positive about the experience of developing for Flight and their enthusiasm for it seems to be spreading.

Flight's flexibility offers developers the opportunity to easily hack on top of the framework as well as develop with it, feeding back into the community.

In the next chapter, we get down to the nitty-gritty of Flight: creating Flight applications.

4

Building a Flight Application

Deciding which framework you are going to use is only one step in designing an application. This chapter explains how to scaffold a Flight application with the **Yeoman** Flight generator and walks through the resulting application structure.

Scaffolding a Flight application with Yo

Yeoman is a workflow for modern web apps. It includes **Yo**, a system that allows a developer to write generators which can create scaffolds for applications and packages with a single command. Yo generators have been created for a wide variety of structures such as jQuery plugins, Node servers, and (of interest to us) Flight applications.

The Flight generator (created by Nicolas Gallagher) can scaffold out Flight applications, components, mixins, and standalone packages in a matter of seconds. It even sets up a test runner using **Karma** and **Jasmine Flight**. We'll look at testing Flight in detail in *Chapter 12*, *Testing*.

Installing Yo

Yo uses Node.js and the **Node Package Manager** (**NPM**) to operate. Follow the installation instructions at `http://nodejs.org/`. The Flight generator requires Bower, a JavaScript package manager.

> If you are working with other projects that depend on the older version of grunt, you can use the **Node Version Manager** (**NVM**) to install multiple versions of Node alongside one another.

You should now be able to install Yo, Bower, and the Flight generator using the following command:

```
npm install -g bower yo generator-flight
```

If you're unfamiliar with npm, the `-g` option indicates that these packages should be installed globally, so they'll be available from any directory. Once installed, you're ready to scaffold your application:

```
mkdir flight-example && cd $_
```

```
yo flight example
```

You will be asked if you want to use **Normalize** and Bootstrap. Answer `Yes` for both to get some useful CSS.

> Visit `http://necolas.github.io/normalize.css/` and `http://getbootstrap.com/` for more information.
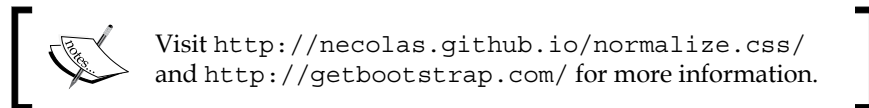
This will create an application structure under `flight-example` and install all the required dependencies (including Flight itself) with Bower.

Understanding the application structure

In the root of `flight-example`, you'll find a lot of configuration files, a `node_modules` directory, and a `test` directory; none of which you don't have to worry about, for now. In fact, all that's important right now is the `app` directory.

In there, you'll see some basic application files, such as 404, index, robots, and favicon, which should be familiar to you.

In addition, there will be a few directories:

- `bower_components` contain the application's Bower dependencies (including Flight and jQuery).
- `css`, `img`, and `js`, are also quite familiar. In the `js` folder, you'll find:
  - `main.js`, the application's initialization script (loaded from `index.html`).
  - `component`, which in time will contain your Flight components.
  - `page`, which holds `default.js`, which will be used to instantiate your components.

Running the application

Opening `index.html` in a browser should be enough; though, due to security issues, it may not be possible to do this in your browser without running a server. The simple `http-server` (which does exactly what it says on the tin) is available through npm.

```
npm install –g http-server
cd app
http-server –c-1 –p 8000
open http://localhost:8000
```

And behold! Your application lives! Here's what it should look like in Chrome with the console open (press *cmd / Ctrl + Alt + J* to open **Console** in Chrome).
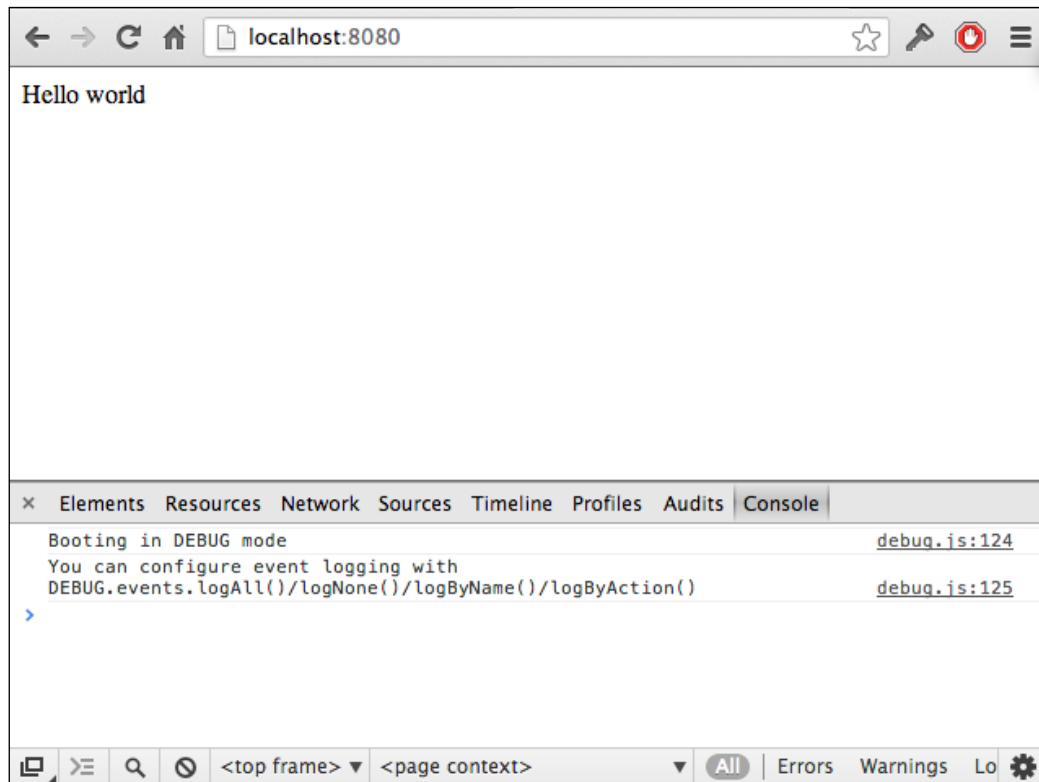
You're now ready to flesh out the application scaffold. Feel free to skip ahead to the next chapter, or read on for information on creating your own Flight applications.

Creating custom applications

It may be the case at some point that you want to create a Flight application which does not conform to the structure generated by the Flight generator. This will almost always be true, if you're introducing Flight to a pre-existing application.

To create your own unique structure, you will have to install Flight manually.

There are a number of ways to do this, but the recommended method is Bower. Have a look at `http://bower.io` for installation instructions. Once Bower is installed, Flight and its dependencies can be installed from the command line with a single command:

```
bower install flight
```

Alternatively, you can download Flight from GitHub, or import it as a git submodule. If you download the files, you will also have to manually install the dependencies such as jQuery, **es5-shim**, and RequireJS.

Using Flight without a module loader

It is possible to use Flight as a standalone package without RequireJS. Instructions on how to do this are available in the Flight docs on GitHub.

> Have a look at the full Flight documentation at
> `http://flightjs.github.io/` for more information.

Troubleshooting

As you work through the examples, you are no doubt going to make mistakes, whether through spelling errors, syntax errors, or misunderstandings. The first step in mitigating this would be to get hold of a good editor that provides syntax highlighting and autocomplete.

Debugging

Flight provides some useful tools to aid in debugging your code. As Flight is event-driven, a logger is provided that reports on when events are bound, unbound, and triggered. Logs are output to the browser console (for example, Chrome Dev Tools).

In the previous application generated, DEBUG is turned on and logs all events. As you work through the example, you may find that you want to change the logging level. You can do so with the following commands:

```
DEBUG.events.logAll(); //log everything
```

```
DEBUG.events.logByAction('trigger'); //only log event triggers
```

```
DEBUG.events.logByName('click'); //only log events named 'click' -
  accepts * as wildcard
```

```
DEBUG.events.logNone(); //log nothing
```

Once your code is ready for production, you can turn the DEBUG mode off entirely by removing the following line from app/js/main.js:

```
debug.enable(true);
```

Summary

The Yeoman Flight generator is a great starting point for any Flight project. Flight is also available in various forms, if you want to roll your own or introduce Flight to an existing application.

In the next chapter, you will start to flesh out the example application by creating your first Flight component.

5

Components

Flight is a component-based framework. This chapter aims to provide an overview of components and how to build them. By the end of this chapter, you will have created a component that will, when instantiated, print "Hello, World!" to the browser console.

What is a component?

A component is a constructor function. It declares properties and methods that describe its behavior.

Flight provides each component with a set of utilities, such as event handling and DOM node selection. These make up most of Flight's API, and are covered in subsequent chapters.

A component constructor is used to create instances of the component. Each instance is attached to a DOM node.

A component instance interacts with DOM through the node it is attached to, and with other component instances through events.

Components do not inherit from other components. Functionality can be extended and shared through mixins.

Component types

Components are generally broken up in to two types: **User Interface** (**UI**) and **Data**, although the distinction between the two is purely conceptual; all components are essentially the same.

All components, both Data and UI, are attached to a specific DOM node. They are then able to trigger and handle events, interrogate and alter the DOM on their root node and its subtree.

The difference lies in their behavior: only UI components should modify the DOM or listen to user-initiated events, while Data components manage data storage, requests, and manipulation.

This separation is not a hard-and-fast rule, and you should feel free to build components in any style you wish, though keeping the separation helps keep your components small, well-defined, and reusable. The examples in this book attempt to maintain a clear separation between UI and Data.

Mixins

Mixins are an alternative to the classic object-based inheritance. It is useful to think of them as a form of multiple inheritance, which allows components to share a set of common functionalities.

A mixin has the same structure as a component, and methods provided by the mixin are available as first-class citizen component instances.

Creating your first component

Every programming tutorial needs its "hello world" moment. Here, you are going to create a component that writes "Hello, World!" to the console when the component is initialized.

All Flight components are AMD modules. Create a new JavaScript file, `app/js/component/hello_world.js`, and add a basic AMD module using the `CommonJS` format:

```
define(function (require) {
  // import dependencies
  var defineComponent = require('flight/lib/component');

  // export component constructor
  return defineComponent(helloWorld);

  // component definition
  function helloWorld () {};
});
```

The `flight/lib/component` module exports a method, `defineComponent`, which mixes Flight's component utilities into a component definition and returns a component constructor function. This is the first step in creating any component.

Attaching components to the DOM

All component constructors have a single exposed method, `attachTo`, which attaches an instance of the component to a DOM node.

> In the example application, `RequireJS` loads `main.js`, which in turn loads an AMD module, `/app/js/page/default.js`. Pages are a special type of component, neither UI nor data. Their sole purpose is to instantiate other components. In a single-page application such as this one, `default.js` will probably be the only page.

Open the default page component, `app/js/page/default.js`. You'll need to require the Hello World component you just created in the module dependencies section:

```
/**
 * Module dependencies
 */
var HelloWorld = require('component/hello_world);
```

> The RequireJS configuration in `main.js` specifies the path component as `app/js/component`. Thus the module loaded will be `app/js/component/hello_world.js`. For more information on paths, see the RequireJS documentation.

Then attach the component to the document DOM node in the `initialize` section:

```
/**
 * Module function
 */

function initialize() {
  HelloWorld.attachTo(document);
}
```

> The `attachTo` method accepts HTML elements, CSS selectors, or jQuery collections as its first parameters. In the latter two cases, it is possible that multiple DOM nodes may be represented; an instance of the component will be attached to every node. For example, `HelloWorld.attachTo('div')` would attach an instance of Hello World to every `DIV` element on the page. The console message would appear once for every instance of the component.

Performing actions on component initialization

One of the methods mixed into components by `defineComponent` is `initialize`. However, this method is used internally by Flight in the `attachTo` process and should not be called directly, nor should it be overridden. Flight instead provides a mechanism for doing something after another method is called.

The `after` method accepts two arguments: a method name and a callback. The callback will be executed immediately after the named method is called.

Within the component definition, the current context (`this`) always refers to the component instance. Utilities mixed in by `defineComponent` are first-class citizens of the component and can thus be accessed via the `this` keyword:

```
define(function (require) {
// import dependencies
  var defineComponent = require('flight/lib/component');

  // export component constructor
  return defineComponent(taskManager);

  // component definition
  function taskManager () {
// execute some code after initialization
this.after('initialize', function () {
console.log('Hello, world!');
});
    };
});
```

Now, whenever an instance of Hello World is attached to the DOM, the
`after:initialize` callback will be executed and `Hello, World!` will be
written to the console. Go ahead and reload the application now to see it
in action. Here's what it will looks like in Chrome:
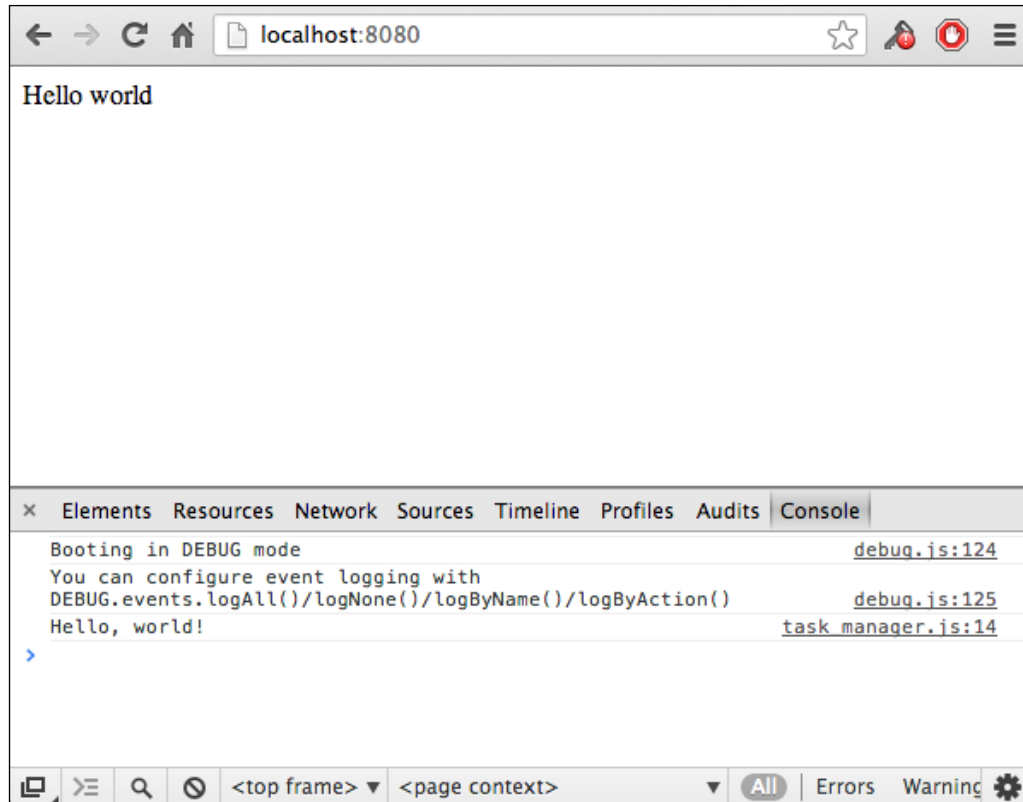
> The `after` method is one of the three methods (`before`,
> `after`, and `around`) which together are known as Advice,
> which is covered in more detail in *Chapter 9, Mixins*.

Summary

Components are constructors, defined using the AMD `CommonJS` format. They are
used by pages to attach component instances to the DOM. Each instance has access
to a set of core Flight functionalities including `after`, which allows callback hooks
to be added to other methods.

6
UI Components

This chapter provides examples on how to listen for browser events, how to access elements within a component, and the use of defaults and settings within components. By the end of the chapter, you will have created a component that handles form submit events and triggers a custom event containing sanitized input.

Attaching components to existing HTML

In the previous chapter, you learned how to attach components to the document. However, most UI components are attached not to the document but to elements such as forms, buttons, and lists.

What with this app being a task manager, it helps if a user can create tasks. Go ahead and replace the `Hello World` text in `index.html` with an `add task` form. This form has a single input (the task description) and a **Submit** button.

```
<form class="js-add-task form-inline">
    <label class="control-label">Add a task</label>
    <div class="controls">
        <input type="text" placeholder="description"
          class="js-add-task-description" />
        <input type="submit" class="js-add-task-submit btn btn-
          primary" />
    </div>
</form>
```

> Classes using the `js-` prefix are intended for use as JavaScript hooks and do not have any CSS associated with them. For clarity, `js-` classes are always the first class in a class attribute.

To handle form submissions, we will create a component (Add Task) and attach it to the form element.

Create a new component file, `/app/js/component/add_task.js`, and add the following code:

```
define(function (require) {
  var defineComponent = require('flight/lib/component');

  return defineComponent(addTask);

   function addTask () {
       // initialize
       this.after('initialize', function () {
           console.log('Initializing Add Task form');
  });
    };
});
```

All UI components follow the same format, so this should already be starting to look familiar. You can skip the console log in the initialization method if you like—it's only there to aid in debugging.

`AddTask` then needs to be imported in default.js and attached to the DOM. You can replace the `HelloWorld` references with `AddTask`, as shown in the following code:

```
var AddTask = require('component/add_task');
…
function initialize() {
AddTask.attachTo(document);
}
```

This will attach `AddTask` to the document. However, we want to attach it to the form element itself. To do this, we can pass a CSS selector to the `attachTo` method, as shown in the following code:

```
function initialize() {
  AddTask.attachTo('.js-add-task');
}
```

Now, we need to set up Add Task to listen for submit events.

Listening for browser events

When the user submits the form, a submit event will be fired in the browser. The Add Task component needs to listen for that event.

Attaching event handlers

Flight's .on() method attaches event handlers to DOM nodes. It is very similar to jQuery's method of the same name, though has some differences in syntax. As with jQuery's .on() method, it also has the matching methods off and trigger. However, the Flight versions also provide a little extra magic. Event listeners attached with Flight's on method are automatically removed when the component is torn down, so you don't have to remember to do it yourself. Less boilerplate makes for cleaner code.

Events are attached to a component's root node, unless otherwise specified. Go ahead and add an event handler for the submit event in Add Task.

```
this.after('initialize', function () {
    this.on('submit', this.handleSubmit);
});
```

You haven't yet created the handleSubmit method, so if you reload the app now, you'll see Flight throwing an error in the console as it is being asked to attach a handler that doesn't exist.

```
Uncaught Error: Unable to bind to 'submit' because the given callback is not
a function or an object                                        component.js:135
```

Defining event handlers

Event handlers (and all other component methods) are declared within the component definition as first-class citizens of the object:

```
define(function (require) {
  var defineComponent = require('flight/lib/component');
  return defineComponent(addTask);

  function addTask () {

    // declare handleSubmit method
    this.handleSubmit = function (event) {
    event.preventDefault();
    };
```

```
      // initialize
      this.after('initialize', function () {
        this.on('submit', this.handleSubmit);
      });

   }
});
```

As with standard jQuery events, the first parameter passed to the handler is a `jQuery.Event` object.

The next step is to get the description from the text input element.

Finding DOM nodes

A component's root node (the node it was attached to) can be accessed from within the component using `this.$node`.
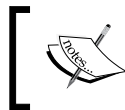
> The $ prefix is used throughout the example application to indicate that the value of a variable or attribute is a jQuery collection.

Although it is possible to use jQuery methods to find elements within `this.$node`, Flight provides a neater method, `this.select`, which uses CSS selectors defined as an attribute on the component.

`this.select` requires the use of named CSS selectors, which we will define using default attributes.

Setting default attributes

`defaultAttrs` is used to define default attributes for a component; to define its default configuration. These attributes can then be referenced from within a component definition, as shown in the following code:

```
function helloWorld () {
this.defaultAttrs({
  welcomeText: 'Hello, world!',
  backgroundColor: 'blue'
});
this.after('initialize', function () {
  console.log(this.attr.welcomeText);
```

```
        this.$node.style('background-color', this.attr.backgroundColor);
    });
});
```

Attributes can be used to store any serializable data but also have one special use case in Flight: defining named CSS selectors.

Later, we will use `defaultAttrs` to define a CSS selector, which in turn will be used to locate the text input field.

> It's best to keep `defaultAttrs` at the top of a component definition for the sake of clarity.

Using attributes to select nodes

Once a selector has been declared as an attribute, `this.select` can be used to find nodes within the component root matching the selector (essentially the equivalent of `this.$node.find(selector)`). In the following example, `$descriptionInput` would contain any nodes within the component root with the class name `js-add-task-description`.

```
this.defaultAttrs({
    descriptionSelector: '.js-add-task-description'
});
var $descriptionInput = this.select('descriptionSelector');
```

To see this in action, update Add Task to use `defaultAttrs` to declare the CSS selector for the input element, and select to get a reference to it in `handleSubmit`, then retrieve the value.

```
define(function (require) {
  var defineComponent = require('flight/lib/component');
  return defineComponent(addTask);

function addTask () {
        this.defaultAttrs({
            descriptionSelector: '.js-add-task-description'
        });

        this.handleSubmit = function (event) {
            // don't actually submit the form
            event.preventDefault();
```

```
            // get the input element
            var $description = this.select('descriptionSelector');
            var description = $description.val();

            // trim whitespace
            description = $.trim(description);
      console.log(description);
   };

        this.after('initialize', function () {
            // listen for submit events
            this.on('submit', this.handleSubmit);
        });

     };
  });
```

Reload your page (`http://localhost:8080`), type `test` in the input field, and hit **Submit**. You should see the following output in the console:

```
Booting in DEBUG mode                                            debug.js:124
You can configure event logging with
DEBUG.events.logAll()/logNone()/logByName()/logByAction()        debug.js:125
<- on [submit] 'form.js-add-task form-inline' addTask            logger.js:66
<- on [submit] function () { [native code] } addTask             logger.js:66
test                                                             add_task.js:20
>
```

Triggering custom events in Flight

Now that `handleSubmit` has the event description, it needs to tell the rest of the application that the user wants to add a task with that description. To do this, it will trigger a custom event named `uiAddTask`. See *Chapter 8, Event Naming*, for details on the event name.

As with `on` and `off`, Flight wraps jQuery's trigger in its own `trigger` method.

```
    this.trigger([selector, ] eventName [, data]);
```

`trigger` accepts up to three arguments. Of these, only the event name is required, though in this case we will provide data as well (the description).

Go ahead and add the following code in place of the console log in Add Task.

```
this.trigger('uiAddTask', {
  task: {
    description: description
  }
});
```

Triggering events on specific elements

You may provide an optional selector to trigger. This can be a CSS selector string, a DOM element, or a jQuery collection. The selector is optional and if no selector is provided, the event will be triggered on the root node of the component.

If the selector represents multiple DOM nodes, a unique event will be triggered on each node.

> CSS selector strings may not behave as you expect when triggering events. The selector is applied to the document root and not to the component node. To trigger an event on a specific node, use `select` to find the node and pass the result to `trigger`, as shown in the following code:
>
> ```
> this.trigger(this.select('addTaskSelector'),
> 'resetForm');
> ```

Event names

Event naming is a very important aspect of Flight. It helps a great deal if event names are descriptive and it is a good idea to lean toward longer rather than shorter event names. An event name is generally used in two or three places, so there's no harm in making them long, whereas the shorter the name the more likely conflicts will occur, and the less obvious the nature of the event will be.

Event naming is covered in detail in *Chapter 8*, *Event Naming*.

Event data

`data` is the event payload. It is optional, but if used, must always be a serializable object. Flight will reject any attempt to use non-serializable objects, such as methods, strings, object instances, or DOM nodes, causing runtime errors or invisible failures.

You can structure your data however you wish, but it is generally a good idea to be as clear as possible, even if this means creating what feels like unnecessary structure.

In the case of the task description, it is possible to make data an object that directly represents a task with a single attribute, description:

```
{
  description: 'buy lightbulbs'
}
```

The event handler would then look like the following:

```
this.handleAddTaskAction = function(event, task) {
  var description = task.description;
  // do something
}
```

This might seem efficient until it becomes necessary to pass some data that is not part of the task, such as a request ID or source ID (to help determine what fired the event). It is not part of a task as such, so the data structure changes:

```
{
  task: {
    description: 'fit lightbulbs'
  },
  sourceId: 'addTask'
}
```

All the event handlers for this event would then have to be altered to handle the new data structure.

It is not possible to obviate all such refactoring, but good planning can make a huge difference.

Modifying the DOM

The last thing this component needs to do is update the Add Task form so the user knows their request is being acted upon. This can be done easily by disabling the form input and **Submit** button, as shown in the following code:

```
this.defaultAttrs({
  submitSelector: '.js-add-task-submit',
  descriptionSelector: '.js-add-task-description'
});
```

```
this.handleSubmit = function(event, data) {
  var $submit = this.select('submitSelector');

  $description.attr('disabled', true);
$submit.attr('disabled', true);
});
```

Currently, the form will always remain disabled. When we start getting responses back from the data layer (which we have yet to build), we'll come back to this and re-enable the form controls.

Summary

A UI component's scope is limited to the DOM node it was attached to and its children.

A UI component:

- Listens for browser events
- Triggers custom events
- Interrogates the DOM
- Modifies the DOM

Components use custom events delivering simple, serializable data objects to communicate with other components.

The next chapter discusses how to process and store data using data components.

7
Data Components

Data components are generally an interface between the UI and storage layers. This chapter explains how data components differ from UI components, how to create them, attach them to the DOM, and how to handle UI events and trigger data events.

What is a data component?

Flight data components are much the same as UI components. The core difference is that data components don't interact with the DOM. Instead, they listen for UI events such as requests for data, or user-initiated actions that require data processing.

This separation of interests allows multiple UI components to utilize the same data components. It also means that if a change is made to how data is stored or processed, it is only done in one place and has no effect on the UI.

For example, the `addTask` component triggers a UI action event, `uiAddTask`, which provides a task description. It is totally agnostic as to what the rest of the app does with that information. It doesn't matter which other components handle that event, or even if anyone does anything with it at all.

A data component could, in response to a UI event, make an asynchronous HTTP request, store the data in local storage, place it in memory, or just ignore it; it's all the same to the UI.

Attaching data components

All data components should be attached to the document. This allows them to receive UI events from the entire application. This also means that data events will be triggered on the document. Data components are agnostic as to the document structure.

Data components are instantiated at the page level. UI components should never be aware of which data components they are interacting with.

Naming data events

As with UI events, the naming of data events is very important. Events define a component's interface with the rest of the application. Event names should be clear and descriptive.

Generally, data events describe their payload: `dataTask`, `dataTasks`, `dataSettings`, and so on.

Occasionally, it may be necessary to trigger an event that describes an interesting moment in the data process. For example, to help an application display a spinner while content is loading, a data component might trigger an event such as `dataWaitingForAsynchronousResponse` and `dataReceivedAsynchronousResponse` to allow the UI to show and hide the spinner.

Creating a data component

Create a new file, `/app/js/component/task_data.js`, and add the basic component code:

```
define(function () {
  var defineComponent = require('flight/lib/component');

  // return a constructor for this component
  return defineComponent(taskData);

  // component definition
  function taskData () {
    // component methods go here
  }
});
```

This code should look very familiar by now. It is a good idea to use this base as a component template so you can add new components more easily. How you do this depends on your preference and **Integrated Development Environment** (**IDE**), but the easiest way is probably to create a new file, / `app/js/component/component_template.js`, and just copy and paste as required.

This component needs to be attached to the document during page initialization. In `default.js`, import `TaskData` as a dependency and attach it to the DOM using the `attachTo` method.

`default.js` should now look something like this:

```
define(function (require) {

  'use strict';

  /**
    * Module dependencies
    */

  var AddTask = require('component/add_task');
  var TaskData = require('component/task_data');

  /**
    * Module exports
    */

  return initialize;

  /**
    * Module function
    */

  function initialize() {
    AddTask.attachTo('.js-add-task');
    TaskData.attachTo(document);
  }

});
```

Listening for UI events

This is a good time to think about the events `TaskData` component will consume and produce.

`TaskData` currently only needs to do one thing, handle the addition of new tasks. However, it is obvious that it will also need to manage task completion and request for task data from the UI (both for single tasks and all tasks).

These requirements correspond to four UI events:

- `uiAddTask`
- `uiTaskCompleted`
- `uiNeedsTask`
- `uiNeedsTasks`

Let's go ahead and set up event listeners and handlers for each of these events. In `TaskData`, add event listeners for the UI events to `after:initialize`, and create dummy event handlers:

```
this.handleAddTask = function(e, data) {};
this.handleNeedsTask = function(e, data) {};
this.handleNeedsTasks = function(e, data) {};
this.handleTaskCompleted = function(e, data) {};

this.after('initialize', function() {
  this.on('uiAddTask', this.handleAddTask);
  this.on('uiNeedsTask', this.handleNeedsTask);
  this.on('uiNeedsTasks', this.handleNeedsTasks);
  this.on('uiTaskCompleted', this.handleTaskCompleted);
});
```

Event handlers

Event handlers in Flight accept two parameters. As seen in `TaskData`, the first parameter is a jQuery event. The second is the data object passed to the `trigger` method.

The data passed with the `uiAddTask` event looks like this:

```
{
  task: {
    description: 'Make tea.'
  }
}
```

`TaskData` needs to store this data in a way that will be able to retrieve it easily later. To do this, each task will be assigned a randomly generated ID and stored in a hash. First, we need to initialize the hash in `after:initialize`:

```
this.after('initialize', function() {
  this.tasks = {};
```

Then, in `handleAddTask`, generate the unique ID and store the task:

```
this.handleAddTask = function(e, data) {
  // generate ID and store on task object
  data.task.id = Date.now();
  // store task
  this.tasks[data.task.id] = data.task;
}
```

Triggering data events

Once the task is stored, `handleAddTask` can trigger an event to announce that the process was successful:

```
this.handleAddTask = function(e, data) {
  data.task.id = Date.now();
  this.tasks[data.task.id] = data.task;

  // trigger event
  this.trigger('dataTaskAdded, {
    task: data.task
});
}
```

Great! You can now create and store tasks. The task data is only stored in memory at the moment, so it will be lost when you reload the app, but we shall do it for the time being. We shall look at storing the data more permanently in *Chapter 9*, *Mixins*.

Completing the task_data component

It should be quite obvious now how the other handlers are going to work. They're included here for the sake of completion, but before you look at the code, first think about how you would implement them.

Also ask yourself, "What else might this component need to do?"

handleNeedsTask

Retrieve a single task corresponding to `data.taskId` and fire the `dataTask` event.

It is worth considering what will happen if the requested task is not available. Usually the most useful action is to trigger an error event, to let other components know that an error occurred. In some cases, it may be acceptable to swallow the error.

```
this.handleNeedsTask = function(e, data) {
  var task = this.tasks[data.taskId];
  if (task) {
    this.trigger('dataTask', {
      task: task
    });
  }
};
```

handleNeedsTasks

Retrieve all tasks and fire the `dataTasks` event.

It's worth considering in what format you would expect the task list to be. An array? A hash? Does `dataTasks` need to provide all information about all tasks, or a subset?

```
this.handleNeedsTasks = function() {
// convert this.tasks to an array to allow easy iteration in UI //
components
  var tasks = Object.keys(this.tasks).map(function(key) {
    return this.tasks[key];
  }, this);

  // trigger data event
  this.trigger('dataTasks', {
    tasks: tasks
  });
};
```

handleTaskCompleted

Handle the `uiTaskCompleted` event, set `task.completed` and `trigger` `dataTaskCompleted`.

This method could result in two possible errors. First, the task may not exist. Secondly, the task may already be completed. It's worth considering whether to create separate error events for each eventuality, or to have a single error event with a different payload. The first option may be clearer, but the second requires less boilerplate to maintain.

```
this.handleTaskCompleted = function(event, data) {
  var task = this.tasks[data.taskId];
  if (task) {
    task.completed = true;
    this.trigger('dataTaskCompleted', {
      task: task
    });
  }
};
```

Error handling

Each of these examples could be improved by triggering error events, if the task doesn't exist in `this.tasks`:

```
if (task) {
…
} else {
  this.trigger('dataTaskError', {
    error: 'Task does not exist',
    request: {
      event: event,
      data: data
    }
  });
}
```

Handling data events

Now that the `dataTaskAdded` event is being fired, tasks can be added to a task list in the UI. For this, we'll need a task list in `index.html` and a UI component, `TaskList`, to manage the adding and removing of tasks. Go ahead and add a header and a list to `index.html`, under the `js-add-task` form:

```
</form>

<h2>Tasks</h2>
      <ul class="js-task-list">
      </ul>
```

Next, create a new UI component, `public/app/ui/task_list.js` (using the component template you created earlier in the chapter), and add a handler for the `dataTaskAdded` event:

```
define(function(require) {
  var defineComponent = require('flight/lib/component');
```

```
   return defineComponent(taskList);
   function taskList () {

     this.handleTaskAdded = function(event, data) {
       // render task list item
       var html = '<li>' + data.task.description + '</li>';
       // append to this.$node. It is assumed this will be a UL
       this.$node.append(html);
     };

     this.after('initialize', function () {
       // attach event listeners
       this.on(document, 'dataTaskAdded', this.handleTaskAdded);
     });
   };
     return defineComponent(taskList);
});
```

Remember, all data events are triggered on the document, so that's where this listener needs to be attached.

You'll then need to attach this component to the DOM in `ui/default_page`:

```
var TaskList = require('components/task_list');
…
TaskList.attachTo('.js-task-list');
```

Also, you can now re-enable the `AddTask` form, as the action completed successfully. In `AddTask` (`app/js/component/add_task.js`), add an event listener and handler for `dataTaskAdded` and re-enable the form:

```
this.handleTaskAdded = function (event, data) {
  this.select('submitSelector').attr('disabled', false);
  this.select('descriptionSelector').attr('disabled', false);
};
…
this.on('dataTaskAdded', this.handleTaskAdded);
```

Summary

Most data components are, as with this example, very straightforward. They define clear interfaces with well-named events and rarely do much beyond simple **Create Read Update Delete** (**CRUD**) tasks.

The key to this simplicity is good planning. By outlining which events a component will send and receive before coding starts, clarity can easily be maintained.

8

Event Naming

This chapter discusses the importance of good event names, what an event really is, and provides an example of naming convention.

The importance of event names

Events define the interface between components. Poor event naming can lead to confusing interfaces, resulting in a wealth of interesting problems. By setting out some good naming conventions early in the design process, it should be possible to mitigate issues such as event loops and duplicate names.

While debugging, clear event names can be an enormous aid.

Events are not instructions

Events are created in a void. The component creating the event should not be seen to be aware of the rest of the application, thus events should not appear as instructions to other components.

For example, it may be tempting to create an event such as `uiAddTask`, an instruction to some other part of the application to create a task. However, if the `add_task` component is unaware of the existence of the rest of the application, this instruction makes no sense.

Instead, this event could be named `uiTaskAdded`, or `uiUserAddedTask`. It is a description of an event that has occurred, not of one that is going to happen.

This might seem like a pointless semantics argument, but, as an application becomes more complex, the particulars of event naming can make the difference between a well-structured, easy-to-maintain application and a tangled web of confusion.

Suggested naming conventions

The conventions presented here are suggestions only, and you are, of course, free to use any convention you like. Event names can be any valid string; though it is always best to make them human-readable, by avoiding use of jargon, abbreviations, and acronyms.

- **Data request**: This is a request from the UI for data, for example, `uiNeedsTask`, `uiNeedsAllTasks`. Data requests are handled by data components and generally result in data events or interesting moments.

- **UI event**: This is an event produced by the UI, for example, `uiTaskAdded`, `uiTaskDeleted`. UI events are handled both in the UI and in data. Other UI components may well be interested in the fact that the user just tried to add a task (messaging, task list, and so on) and the related data response or error.

  It is tempting to try separate events into user-initiated actions and actions performed by UI components. However, it is hard to see where to draw the line between the two, as all actions are at some point a result of a user action, however indirect.

- **Data event**: This is an event that contains data. The data prefix is generally followed by a description of the payload. Generally, data components only trigger a single data event, for example, `dataTask`, `dataTags`.

- **Data error**: This is an event resulting from a data processing or request error response such as a 404 or 501. In these cases, a data error event will be triggered instead of a data event, for example, `dataTaskError`, `dataTagsError`. It's good practice to include both the original request (the event data payload from a data request or UI event) and the response from the server.

- **Interesting moment**: Any process has a variety of interesting moments. For example, in an animation, interesting moments occur when the animation starts, when it stops, and at each step in the process. These are interesting because they may affect other parts of the application, for example, `uiTransitionStart`, `uiTransitionEnd`.

  Another example would be when the UI performs an action as a result of a UI action (for example, `uiShowTaskList`). The interesting moments that result from this might be `uiShowingTaskList` and `uiTaskListShown`.

  Interesting moments happen in data, too, for example, `dataTaskCreated`, `dataTaskDeleted`, or perhaps `dataWaitingForAsynchronousResponse`, and `dataReceivedAsynchronousResponse`.

Summary

Naming events is entirely up to you. The previous examples do not cover every eventuality. However, it does help to maintain consistent event naming within an application.

Event names should be descriptive of the event that occurred. If this means creating long names, so be it. The more descriptive the name is, the easier it is to understand what caused the event and what data is associated with it.

The next chapter discusses mixins, an alternative to class-based inheritance, which are used to share functionality between components.

9
Mixins

Mixins are Flight's method of sharing functionality among components. A mixin extends a component's behavior directly—methods declared in the mixin can be used in a component as if they were its own.

This chapter explains what mixins are, when to use them, and how to create them. It also covers **Advice**, a mechanism used to override or extend existing methods.

What are mixins?

Mixins are an alternative to classic class-based inheritance. In the class-based inheritance, functionality is added to a base class by extending that class to create a new class. For example, both frogs and fleas are subclasses of animal.

Class-based inheritance does have its share of problems. Many animals have similar functionality; for example, both frogs and fleas can jump, while their siblings newts and beetles do not.

Thus, in the world of classical inheritance, the jump functionality would have to be written independently for both frogs and fleas, as well as most other jumping animals, as they have no common jumping ancestor.

Mixins allow the jump function to be written once and then shared among any component that requires it.

When to use mixins

Mixins should be employed whenever you need to share functionality between components. Basic utility methods such as string prettification or more complex features, such as search field behavior or API utilities, can be extracted to mixins and shared among components.

How do mixins work?

Methods defined in mixins act as first-class citizens in a component instance. For example, a `withStringUtils` mixin might declare a `prettyString` method. Any component that mixes in with `withStringUtils` will have access to `this.prettyString`.

Creating mixins

A mixin definition is given in the following example. As you can see, the only difference between mixins and components is that mixins don't use `defineComponent`.

```
define(function() {
  var withJumping = function () {
    this.jump = function(howFar) {
      // perform jump
    });
  };

  return withJumping;
});
```

Using mixins

A component uses a mixin by passing the mixin as an argument to `defineComponent`. The mixin's methods are immediately accessible from within the component. For example, if the path for the mixin defined previously is `component/with_example_mixin.js`, it would be imported in a component as shown in the following code:

```
define(function() {
  var defineComponent = require('flight/lib/component');
var withJumping = require('component/with_jumping');

return defineComponent(frog, withJumping);
  function frog() {
    after('initialize', function(){
      this.on('hasSeenCat', this.jump);
});
  };
});
```

Note that the methods declared in mixins will override similarly named methods declared in a component, which imports that mixin.

Mixins can declare their own `defaultAttrs`, but unlike with methods, mixins cannot override `defaultAttrs` set on the component. In fact, Flight will throw an error if you attempt to do this.

Components can mix in any number of mixins. It's not rare to see component definitions like this:

```
defineComponent(taskList, withTemplating, withSelectAll,
  withDraggable, …);
```

Mixin priority

The order in which mixins appear as arguments to `defineComponent` is important. As stated previously, methods declared in a mixin will override methods already defined on the component. This includes methods defined in other mixins.

In the previous case, a method declared in `withDraggable` would override similarly named methods in `withSelectAll`, `withTemplating`, and `taskList`.

This also affects `defaultAttrs`. Attempting to declare the same `attr` as one which has already been declared in a mixin to the left will result in an error.

It's interesting to note that `defineComponent` only accepts mixins as arguments— `taskList` is as much a mixin as `withTemplating`.

Creating your first mixin

Currently, your Task Manager only stores data in memory. Fine for testing, but not much good in an actual app. To fix this, you're going to build a data mixin, which uses the browser's local storage API.
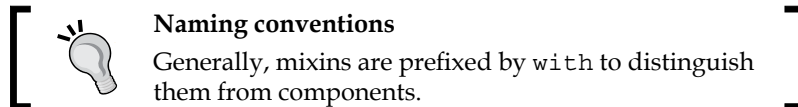
Local storage is a very simple key/value structure. As with cookies, local storage values must be serializable, so task objects will need to be converted to strings before they can be stored.

> Many older browsers don't support local storage, or it may be disabled. Check if your browser supports local storage, and that it is enabled or this example will not work.

Create a new file, `app/js/component/with_storage.js`. All mixins have the same basic structure, so this may be a good time to create a mixin template.

You may at some point want to break out your mixins into a separate directory to reduce clutter, but for now the component directory will do just fine.

> **Naming conventions**
> Generally, mixins are prefixed by `with` to distinguish them from components.

This mixin needs to provide two methods, `read` and `write`. The following code shows these methods being defined in the `withStorage` mixin:

```
define(function () {
  return withStorage;
  function withStorage () {
    this.write = function(key) {
var serializedValue = JSON.stringify(value);
      localStorage.set(key, value);

    };

    this.read = function(key, value) {
var value;
  var serializedValue = localStorage.get(key);
if (serializedValue !== undefined) {
value = JSON.parse(serializedValue);
} else {
    value = serializedValue;
}
return value;
    };
  };
});
```

Mixing storage into taskData

This mixin can now be used by any component, giving it access to `localStorage`. You can mix it into the `taskData` component, as shown in the following code:

```
define(function (require) {
  var defineComponent = require('flight/lib/component');
  var withStorage = require('component/with_storage');
```

```
    return defineComponent(taskData, withStorage);
    function taskData () {
        …
});
});
```

Once mixed in, the read and write methods are available to the component as
first-class citizens and can be used when creating tasks:

```
this.defaultAttrs({
  taskStorageKey: 'tasks'
});

this.handleAddTask = function(e, data) {
  data.task.id = _.uniqueId('task');
this.tasks[data.task.id] = data.task;
this.write(this.attr.taskStorageKey, this.tasks);

// trigger event
this.trigger('dataTaskAdded', {
  task: data.task
});
}

this.handleNeedsTasks = function() {
  this.tasks = this.read(this.attr.taskStorageKey);
  this.trigger('dataTasks', {
tasks: this.tasks
});
});
```

To minimize data storage access, tasks retrieved from localStorage can be stored
locally at initialization:

```
this.after('initialize', function () {
  this.tasks = this.read(this.attr.taskStorageKey) || {};
  …
});
```

Initializing the task list from storage

Now that tasks are being stored in local storage, the Task List can show store tasks at startup.

In Task List a handler needs to be added for the `dataTasks` event to write out the full Task List. This uses the same rendering code as `this.handleTaskCreated`, so that can be extracted to a new method, `this.addTask`.

```
this.addTask = function(task) {
  // render task list item
var html = '<li>' + task.description + '</li>';
// append to task item container
this.$node.append(html);
};
this.handleTaskCreated = function(event, data) {
this.addTask(data.task);
};
this.handleTasks = function(event, data) {
  data.tasks.forEach(this.addTask, this);
};
```

Note that `this` needs to be passed to `forEach` to maintain context.

Extending existing methods

With traditional, class-based inheritance, methods on the parent class can usually be executed by calling the super method from within the overriding method.

However, as we have seen, declaring a method in a mixin completely overrides any existing method with the same name.

To achieve behavior similar to super, Flight provides Advice.

We have already seen `this.after` being used to execute a method after a component instance has been initialized (for example, after `this.intialize` has been executed). `after` is one of three methods, `before`, `after`, and `around`, provided by Flight's Advice API.

Advice can be used on any method, not just `initialize`.

```
this.after('jump', function() {
// prepare for landing
});
```

before and after

Before works exactly the same way as `after`, except the function provided is executed before the named method is called. For example, look at the following code:

```
this.before('addTask, function() {
  console.log('this will happen before addTask is called');
});
this.after('addTask, function() {
  console.log('this will happen after addTask is called');
});
```

`before` and `after` both accept two arguments: a method name (for example, `addTask`) and a callback function.

The callback is called with the same arguments that were passed to the named method. In this example, `before` is used to log all calls to the `trigger` method:

```
this.before('trigger', function(selector, eventName, data) {
  console.log(selector, name, data);
});
```

around

`around` works slightly differently: named method can be called from within the callback function (this is similar to calling `super()` in class-based languages such as Java). If the named method is not called, the callback replaces the method rather than wrapping it.

This allows the callback to modify arguments or add logic to decide whether to execute the method at all.

The first argument passed to the callback is the named method. The following example could be used to wrap a component's content with some pop-up chrome.

```
this.around('render', function(originalRender, params) {
  var $wrapper = $('<div class="popup" />');
  var $content = this.originalRender(params);
  return $wrapper.append($content);
});
```

Advice priority for components and mixins

If multiple mixins and their parent component have advice for the same named method, the execution order is dictated by the order provided in the call to `defineComponent`. For example, in the following component definition, `after('initialize')` callbacks would be executed first for task component, then `withLocalStorage`, then `withCookies`, and lastly `withMemoryStore`.

```
defineComponent(task, withLocalStorage, withCookies,
  withMemoryStore);
```

> **Be careful with before('initialize')**
>
> When a component is initialized, it is attached to the DOM. Attempting to do work before initialize will likely result in hard-to-trace bugs. In particular, avoid using on, `off` and trigger within `before:initialize`

Mixing mixins into mixins

Mixins can extend other mixins through Flight's compose utility. Compose is used by `defineComponent` to merge a component with its mixins. It can also be accessed from within a mixin, as shown in the following code:

```
define(function (require) {
  var defineComponent = require('flight/lib/component');
  var withTextUtils = require('component/with_text_utils');
  // mixin other mixins
  var withLocalStorage = function() {
compose.mixin(this, [withTextUtils]);
  };
});
```

> Thus, when a component mixes in `withLocalStorage`, `withMemoryStore`, and `withCookies` come along for the ride.

Summary

Mixins are a powerful alternative to class-based inheritance. They allow any functionality to be re-used in any component.

Methods declared in mixins override existing methods. Advice allows mixins and components to extend and conditionally override existing methods.

The next chapter discusses templates, including various methods of managing templates in Flight, how to create a templating mixin, and how to handle events in dynamically generated HTML.

10

Templating and Event Delegation

This chapter discusses various templating approaches in Flight, providing examples of **DOM node templating**, **client-side templating** with **Hogan**, and **server-side templating** with **Grunt**. Also covered is how to use generated HTML to determine state and event-delegation for dynamic HTML.

In the Task List component, new list items are created using string concatenation. In this simple case, this technique is relatively easy to read, but is not a scalable solution. In the following examples, tables are used to provide a more complex problem.

There are various ways of solving this: generating template objects from existing DOM nodes, constructing HTML templates within components and using mixins and precompiled templates.

Generating template objects from DOM nodes

In the `index.html` file, add a list with a single, empty list item. When the `task_list` component is attached to the DOM it can create a clone of this node and use it as a template. To achieve this, the table row needs to be hidden by default, so it doesn't show on first load.

```
<ul class="js-task-list">
    <li class="js-task-item hide"></li>
  </ul>
```

Note that the `hide` class is provided by `bootstrap.css`. You may want to use `style="display:none"` instead.

Then, in Task List, create the list item template and use it when rendering tasks. Note that the Task List component is intended to be attached to the `ul` element, `.js-task-list`.

```
this.defaultAttrs({
  taskItemSelector: '.js-task-item',
…});
this.addTask(task) {
  …
  // render task list item
  var $taskItem = this.$taskItemTemplate.clone();
  $taskItem.text(task.description);
  this.$node.append($taskItem);
};
this.after('initialize', function() {
  // get template
  this.$taskItemTemplate =
  this.select('taskItemSelector').clone();
  this.select('taskItemSelector').remove();
  …
});
```

This technique has the advantage that no HTML is rendered by JavaScript, which for small templates does make it easy to read. However, in larger templates the amount of code required to populate the template grows quickly, and the code itself becomes tightly bound to the template.

Templating systems such as Mustache, Hogan, and Handlebars can be employed for creating a more scalable solution.

Constructing templates in components

The example application uses Hogan to compile and render Mustache templates. For more information on Mustache, check the docs at `http://mustache.github.com/mustache.5.html`.

Hogan

Hogan is an open source project by Twitter that reimplements Mustache. It is highly optimized for performance (100 percent + faster), adds template inheritance and provides Hulk, a utility designed to pre-compile and render templates on the server. Find out more at `http://twitter.github.com/hogan.js`.

First, you'll need to download Hogan. The easiest way to do this is to use bower:

```
$ bower install hogan
```

You'll then need to import the Hogan compiler and templater. Add the following lines to the head of `index.html`:

```
<script src="bower_components/hogan/lib/template.js"></script>
<script src="bower_components/hogan/lib/compiler.js"></script>
```

This adds Hogan to the global namespace so it can be used in any component. In the following example, Hogan is used to construct HTML. Note that the plain text template must be compiled before it can be rendered. The compiled template is an instance of `Hogan.template`.

```
this.addTask = function(task) {
    // render task list item
    var html = this.taskItemTemplate.render(task);
    // append to task item container
    this.select('taskContainerSelector').append(html);
};
this.after('initialize', function () {
    // compile template
    this.taskItemTemplate =
Hogan.compile('<li>{{description}}</li>');
…
```

Constructing templates in components is clear and readable, and creates a strong connection between the HTML and the component. However, it does create an interesting problem.

What happens when the same template needs to be used in other components? They have no access to the compiled template in Task List, so would have to compile their own version of the template.

Compiling templates is an expensive process. This component only compiles the template once, but other components intended to be instantiated multiple times would need to compile again for every instance.

This problem can be solved by precompiling templates on the server. In Flight, the compiled templates would then be loaded and rendered by a mixin.

Creating a templating mixin

UI mixins work exactly like data mixins—see *Chapter 9, Mixins*. This mixin is simply a store for templates. The first step is to extract the template compilation to a mixin:

```
define(function () {
return function withTemplate () {
    this.before('initialize', function() {
      this.taskItemTemplate = Hogan.compile('<li>{{description}}</
li>');
    });
  };
  });
```

By calling `Hogan.compile` before initializing, rather than after, it can be ensured that the template is available to components and other mixins in the initialize phase.

This mixin could be further improved by making it agnostic to the templating system being used. Rather than exposing the compiled templates, it could expose a render method that would accept a template name. Although it would still be using Hogan internally, the components using it would be entirely unaware of what was going on behind the scenes. For example, look at the following code:

```
define(function () {
return withTemplate;

    function withTemplate () {

        this.defaultAttrs({
            taskItemTemplate: '<li>{{description}}</li>'
        });

        this.render = function (templateName, renderOptions) {
            if (this.templates[templateName]) {
                return this.templates[templateName].
render(renderOptions);
            }
        };

        this.before('initialize', function() {
            this.templates = {};
            this.templates.taskItem =
  Hogan.compile(this.attr.taskItemTemplate);
```

```
        });
    };
});
```

Then, in the component:

```
var html = this.render('taskItem', task);
```

Server-side compilation

Template compilation can be a costly process, slowing down rendering. Server-side compilation can reduce load on the client. Templates are precompiled into JavaScript on the server during the build process, or through an automated task. The resulting JavaScript file is then loaded as with any other JavaScript file, making the compiled templates available immediately.

A JavaScript environment is required on the server to compile the templates, but this is beyond the scope of this book. If you're familiar with grunt, check out grunt-contrib-hogan (`https://github.com/vanetix/grunt-contrib-hogan`), which allows you to generate compiled templates as part of your build step, or as a watch task. Alternatively, you can use Hogan's build tool, Hulk.

Either way, you'll end up with JavaScript files (for example, `templates.js`), which will contain the compiled versions of all your templates. You'll need to import this as a dependency to your templating mixin, or add it to the global namespace.

Adding templates as a dependency in `ui/with_template`:

```
define(function () {
  var templates = require('templates'); // path to templates.js
  return withTemplate;
   function withTemplate () {
        this.render = function (templateName, renderOptions) {
            // check template exists
            if (templates[templateName]) {
                // return rendered template
                return templates[templateName].render(renderOptions);
            }
        };
    };
});
```

Using HTML to determine state

If you do choose to render HTML on the server, your components will be able to read the generated HTML and determine state from it, that is, rather than making additional data requests with Ajax, they can read it directly from the generated HTML. This doesn't change much about the way code is written, but it is worth considering when designing HTML and components.

There are four ways to retrieve data from the HTML:

- **Structure**: The structure of the HTML can be used to determine state. The presence or absence of elements can be used to build up a representation of the available data.

- **Class names**: Class names such as `is-selected` and `has-tags` can be used to determine the initial state of a component.

- **Data attributes**: Data attributes can be read using jQuery or `HTMLElement.attr`. For example, look at the following code:

  ```
  var taskId = this.$node.data('task-id');
  ```

- **Content**: Lastly (and perhaps most obviously), state can be read from the text content of an element. For example, look at the following code:

  ```
  var description = $node.text();
  ```

Working with dynamic HTML – event delegation

When rendering templates in the client, new HTML nodes are inserted into the DOM. Using `this.on` as previously shown, it would be necessary either to attach new event listeners to elements after they have been inserted, or attach an event listener on the component root and determine the target element programmatically.

A third possibility is to use event delegation, whereby an event listener is attached to the root node but is only triggered if the target of the event matches a selector.

In jQuery, `on` allows the specification of a selector, and event listeners will only be called if the source element matches the selector.

Flight's `this.on` provides the same functionality, though the syntax is a little different. To listen for delegated events, the selector must first be declared in `this.attr` (or via `defaultAttrs`). The named selector is then referenced in `this.on`.

In the following example, `handleItemClick` would only be called if the user clicks on an element with class `.js-item`.

```
defaultAttrs({
  itemSelector: '.js-item'
});
this.after('initialize', function () {
this.on('click', {
  itemSelector: this.handleItemClick
});
});
```

This allows multiple delegated event listeners to be attached in a single statement:

```
this.on('click', {
  taskItemSelector: this.handleTaskItemClick,
  taskCompleteSelector: this.handleTaskCompleteClick
});
```

As mentioned previously, another common pattern that achieves the same goal is to listen for clicks on the parent and then determine the action based on the attributes of the target element. This could be achieved by adding a data-action attribute to the element, as shown in the following screenshot:

```
<li class="js-task- item" data-id="1234">
  <input type="checkbox" data-action="complete" />
   Make tea
</li>
```

When the event handler is called, it can interrogate the event target to determine the required action or task ID:

```
this.handleClick = function (event) {
  var action = $(event.target).closest('[data-
  action]').data('action');
  var id = $(event.target).closest('.js-task-item').data('id');
  switch (action) {
  case 'complete':
    // do something
  }
});
this.after('initialize', function() {
  this.on('click', this.handleClick);
});
```

Both patterns are equally effective and have their places in applications.

Adding delegated events to task_list

Whichever method you use, you should now be able to generate a task list from a template, so it's time to get on and do something with the list. In this section, you can see event delegation in action as we tackle the task of marking an item as complete.

Completing a task

The first thing to consider is which events the component will be creating. `data/task` listens for `uiTaskCompleted`, expecting data in the format:

```
{
  taskId: 'task id'
}
```

So, when an item is to be completed, `task_list` will need to know the ID of the selected item. The easiest way to do this is to add a `data-id` attribute to the table row. Event delegation requires a selector, so add a `js-` class to the table row. If you haven't already, add the checkbox too, as shown in the following code:

```
<li class="js-task-list-item" data-id="{{id}}">
<input type="checkbox" data-action="complete" />
{{description}}
</li>
```

Next, add a named selector to `defaultAttrs`, an `onclick` listener in `after:initialize` and an event handler method:

```
this.defaultAttrs({
  itemSelector: '.js-task-list-item'
});

this.handleItemClick = function (event) {
  // get target item from event
  var $item = $(event.target).closest(this.attr.itemSelector);
  var id = $item.data('id');
  // get action from target element
  var action = $(event.target).closest([data-
  action]).data('action');
  if (action === 'complete') {
  this.trigger('uiTaskCompleted', {
    taskId: id
});
}
};
```

```
this.after('initialize', function() {
  this.on('click', {
    itemSelector: this.handleItemClick
  });
});
```

Note that the examples provided don't allow you to mark a completed task as incomplete. This can be solved in a few different ways.

One way would be to pass the state of the checkbox along with the `taskId` with `uiTaskCompleted`, as shown in the following code:

```
this.trigger('uiTaskCompleted', {
  taskId: id,
  completed: false
});
```

The state would then have to be handled correctly in data/task.

Another might be to create a separate event, such as `uiTaskIncomplete`.

Summary

Flight's agnostic architecture allows you to use whichever templating system you see fit, and abstract away the particulars of the templating system used.

Thanks to this agnosticism, the same templates can be compiled and rendered on the server as in the client, increasing performance by reducing the load on the client and the number of round trips made to the server.

`this.on` provides an event delegation syntax to handle events on dynamically-generated DOM nodes.

In the next chapter, we will look in detail at how Flight aids in developing high-performance applications for high-latency environments.

11
Web Application Performance

Web applications suffer from poor performance for a variety of reasons. When we consider performance, we tend to think first of poor code and memory leaks, but in a web application, the most significant issue is generally how long it takes to load and interpret the code required to render a page.

This chapter explains how Flight can be used to render pages efficiently and how to work around latency in Ajax requests.

Reducing time to page load

Time to page load, the point at which a user can actually start to interact with a page, is the most important factor in website performance. When a user first visits a website, they are presented with a blank page by the browser while it downloads and interprets HTML, CSS, JavaScript, and images.

Two key factors in time to page load are page weight and the number of files being loaded (though the latter may be mitigated by using SPDY, an alternative to HTTP developed by Google).

The common techniques to reduce these factors are compression and concatenation. The aspect that often gets overlooked is the sheer amount of script being sent over the wire. With MV frameworks, the entire application has to be downloaded and interpreted before any real work can begin, as the view is dependent on the model.

Deferred loading

By using **Asynchronous Module Definition** (**AMD**) frameworks such as RequireJS to defer loading of JavaScript until after page load, the amount of JavaScript present at first load can be reduced down to the framework and nothing more. A minimalistic page is sent first, using empty HTML elements to build a structure.

The asynchronous modules are then loaded and start communicating with the server using **XMLHttpRequest** (**XHR**) to retrieve data, building up the page piece by piece.

Server-side rendering

Rendering entire pages or chunks of HTML on the server would seem to go against all other advice in reducing page weight. Providing more HTML, not less, sounds rather counterintuitive.

However, the time to first load isn't just about page weight, it's about how long it takes a browser to interpret the code.

Server-side rendering presents an entire page ready for rendering without any JavaScript other than a single, tiny file to kick off further loading of modules. Page load happens in an instant.

Using the DOM to determine state

Flight components are attached to DOM nodes. Once attached, a component can interrogate the state of the DOM to determine application state using classes such as `is-active`, `has-filters`, `is-own-tweet`. This allows the user to start interacting with that component without waiting for further data requests to be made.

Using request type to determine response

Once an application has loaded, interaction needs to be as fast as possible. Ideally, this means no more page loads. However, it also needs to be accessible. Clicking on a link should, even if no JavaScript has been loaded, perform the correct action.

Flight provides the option of progressive enhancement by allowing the same templates to be rendered on the client or on the server.

The server, on receipt of a request, can determine whether that request was sent over XHR (Ajax), or as a standard HTTP request. In the case of an XHR request, it should send an XHR response, some **JavaScript Object Notation** (**JSON**) data, or an HTML snippet rendered on the server. In the case of an HTTP request, it should send a fully formed page.

This is very useful when implementing `pushState`, the same URL works either for an entire page or an in-app XHR request, making managing history easy.

Perceived performance

A user's opinion of a web app is often greatly swayed not just by how long it takes to process requests but also by the time between a request being made and something happening. Web apps are generally asynchronous in nature, using XHR to communicate with a server and then waiting for a response. It's all the time spent waiting for a response that gives the impression of poor performance.

Many apps show spinners to indicate to the user that the app is waiting for a response. This ensures the user is aware that something is happening, but also tends to mean that the user sits there waiting for the response to be received and processed rather than just getting on with things.

An alternative is to embrace the idea of asking for forgiveness rather than for permission and just go ahead and pretend that the request has already been completed. In the rare case of a communications or server error, an error message can be displayed and the changes made to the UI reversed.

This results in a non-blocking interface, giving the impression of superior performance.

Applying perceived performance in Flight

In the example application, a list of tasks needs to be displayed below the Add Task form. Whenever there are edits or deletes, this list should be updated immediately, without waiting for a response from the server.

It is also possible to add tasks immediately, though if the user then edits or deletes the task before a response is received from the server, the server and application states could diverge.

Summary

A Flight component can render itself or rely on server-side rendering, allowing a developer to optimize an application's performance by choosing when and how to render HTML, serving up the minimum required data at every point.

By listening for events and acting immediately rather than waiting for a response, an application's perceived performance can be improved dramatically.

The next chapter discusses how to test Flight components and mixins by extending an existing test framework.

12
Testing

This chapter provides an example BDD test, which is written in Jasmine, explains what to focus on while testing components, how to gain references to component instances, and how to extend existing test frameworks to handle components, mixins, and events.

What does a test look like?

Which framework you use to test Flight is of course entirely up to you. Currently, the most popular options are Jasmine and Mocha, both of which are BDD (Behavior-Driven Development) frameworks.

Here's an example Jasmine test using the jasmine-flight extensions (we'll look at these in more detail later):

```
defineComponent('task_list', function () {
it ('should trigger uiNeedsTasks on initialization', function () {
  spyOnEvent(document, 'uiNeedsTasks');
  setupComponent();
  expect('uiNeedsTasks').toHaveBeenTriggeredOn(document);
});
});
```

Testing the interface

All Flight components interact with one another through events, and UI components interact with the DOM. The event interface defines a component's behavior and is the only aspect of its functionality that needs to be tested.

For example, in `task_data` there is no point in testing the results of `handleTaskCreated` directly. The component's interface is the `uiAddTask` event, not the handler (which is private). Thus, what needs to be tested is how the component reacts to a `uiAddTask` event.

How well a test framework manages the testing of events will be a big factor in deciding which one to employ.

> Twitter has released jasmine-flight, an open source project available at `http://github.com/flightjs/jasmine-flight`. jasmine-flight includes all the Jasmine extensions and event utilities in this chapter, as well as an example test runner, so if you wish, you can skip most of the code in this chapter and just use jasmine-flight.
>
> If you created your project with the Yo Flight Generator, jasmine-flight will already be installed in bower_components. Otherwise you can install it with bower: `$ bower install jasmine-flight`.

Obtaining a reference to a component instance

Although the event interface is all that needs testing, it remains essential to be able to modify the internal state of components so that responses can be predicted. Flight makes this seemingly impossible, as no reference to a component is provided.

Fortunately, this isn't entirely true. Although `Component.attachTo` does not return a reference to the component, new `Component()` does.

> This method for creating components should never be used in an application code. By keeping a reference to a component in this manner, the core tenets of the Flight architecture are broken, as it becomes possible to reach in and call component methods directly.

In the case of `task_data`, a test would need to import the component and attach an instance to an empty node. Once created, the internal methods and attributes of the component are exposed.

```
define(function() {
  var TaskData = require('component/task_data');
  var $node = $('<div></div>');
  // create a new instance of the component, attaching it to
```

```
// a floating node
var component = new TaskData($node);
// set up the component state
component.tasks = {
  1: {
    id: 1,
    description: 'Make tea'
  }
};
var result;
// expected data
var expected = {
  requestId: 100,
  task: {
    id: 1,
    description: 'Make tea'
  }
};
// listen for events and store data
$node.on('dataTask', function(event, data) {
  result = data;
});
// trigger the uiNeedsTask event
$node.trigger('uiNeedsTask', {
  taskId: 1,
  requestId: 100
});
// assert that result matches expected. How this is done depends
// on the testing framework used
assertEqual('dataTask should contain correct task & request id',
  result, expected);
});
```

Instantiating standalone mixins

Mixins provide an additional challenge; as to behave as intended they must be mixed into a component. The answer is relatively straightforward: create an anonymous component, mix the mixin into that, and then create an instance.

When dealing with mixins, the interface is, in fact, a method, not an event, so it may be necessary to call a method on the component instance, as shown in the following code:

```
define(function() {
  var withStorage = require('component/with_storage');
  var $node = $('<div></div>');
```

```
    var component = defineComponent(function () {}, withStorage);
    // mock local storage
// set response
var result = component.read('tasks');
});
```

Triggering browser events

Most UI components react to browser events such as `click` and `mouseover`. These can be simulated easily using jQuery. An event in jQuery is an instance of the `jQuery.Event` object. Passing an event to jQuery's `trigger` method will cause that event to be fired on the document. For example, look at the following code:

```
var clickEvent = new jQuery.Event('click');
component.$node.trigger(clickEvent);
```

It's also possible to simulate specific keypresses in the same way:

```
var keypressEvent = new jQuery.Event('keypress');
keypressEvent.which = 27; // escape
component.$node.trigger(keypressEvent);
```

Allowing for refactoring

When refactoring components, it's needless to say that tests are extremely helpful. However, they can also be a hindrance.

For example, the `task_list` component relies on two selectors, `.js-task-list-item-container` and `.js-task-list-item`. During a refactoring, if it becomes necessary to rename these, any tests that use them in fixtures or in assertions would fail.

In this case, the failure doesn't indicate that there is something wrong with the component, rather that there is something wrong with the test.

This can be solved permanently by using component options to override the component's `defaultAttrs`:

```
var $fixture = $('<div><div class="js-task-list-item-
  container"></div></div>');
TaskList.attachTo($fixture, {
  taskContainerSelector: '.js-task-list-item-container',
  itemSelector: '.js-task-list-item'
});
```

Now, if the selector is changed, the test will continue to work as expected.

Testing teardown

Sometimes it may be necessary to check whether a component has been successfully torn down. When a component is torn down, it is removed from the registry. Flight's registry provides a method, `findInstanceInfo`, which when provided with a component instance will return that instance if it is in the registry, or null if it is not.

```
define(function () {
  var registry = require('flight/lib/registry');
  var TaskList = require('component/task_list');
  var taskList = new TaskList(document);
  console.log(registry.findInstanceInfo(taskList)); // will log
    instance
  taskList.teardown();
  console.log(registry.findInstanceInfo(taskList)); // will log
    'null'  });
```

Testing component instantiation

In the case where a component is instantiated inside a loop or conditional statement, it may be necessary to determine if a specific component has been instantiated on a specific node.

It's easy enough to check whether a component has been instantiated on a node, using `registry.findInstanceInfoByNode`. However, this doesn't help in determining what kind of component has been attached.

To determine whether the instance found on a node is the correct type or not, we can use `registry.getComponentInfo` to find all the instances of a specific component, then compare them against the instances returned by `findInstantInfoByNode`:

```
define(function () {
var registry = require('flight/lib/registry');
  var TaskList = require('component/task_list');
  var taskList = new TaskList(document);
  var nodeInstances = registry.findInstanceInfoByNode(document);
  var componentInstances =
    registry.getComponentInfo(taskList).instances;
  var isIntanceOnNode = nodeInstances.some(function(nodeInstance)
  {
```

```
        return componentInstances.some(function(componentInstance) {
          return componentInstance === nodeInstance;
    };
      });
      });
```

Extending Jasmine for Flight

Jasmine is a Behavior-Driven Development test framework. Tests are built up from simple statements such as the following statements:

- `task_list` should fire a `uiTaskSelected` event when a user clicks on a task
- `task_list` should listen for `dataTaskDeleted` events and remove the specified task from the task list

This style is very well suited to Flight's architecture. Jasmine also has excellent asynchronous features, making it ideal for an event-based architecture.

Jasmine and AMD

Jasmine specs can be loaded as AMD modules, but this immediately causes a problem. The spec then needs to load a component file inside the `describe` method call, which is itself inside a require callback. This breaks the AMD dependency chain, so the tests will load and execute before the component file has been loaded. For example, check `task_list_spec.js` in the following code:

```
describe('ui/task_list', function() {
  require('/app/ui/task_list', function(TaskList) {
    // this describe will not be run as part of the test suite,
    // as the callback containing it will not be called until
// after the test suite has executed
    describe('… some test', function() {
      …
    });
  });
  });
```

Look at the following code for the test runner:

```
require('task_list_spec', function() {
  jasmine.getEnv().execute();
});
```

A neat solution to this is built into Jasmine: the asynchronous `waitsFor`.

```
describe('ui/task_list', function () {
  beforeEach(function () {
    this.component = null;
    this.$node = $('<div></div>');
    require(['app/ui/task_list'], function (TaskList) {
      this.component = new TaskList(this.$node);
    }.bind(this));
    waitsFor(function() {
      this.Component !== null;
    });
  });
  describe('should do something', function () {
    …
  });
});
```

Of course, including this at the top of every spec would be a pain. It's possible to extract this out to a wrapper method. This method (`describeComponent`) loads the component definition file and stores the component constructor, as shown in the following code:

```
var describeComponent = function (description, specDefinitions) {
    return jasmine.getEnv().describeComponent(description,
  specDefinitions);
};

jasmine.Env.prototype.describeComponent = function (componentPath,
  specDefinitions) {
  describe(componentPath, function () {
    beforeEach(function() {
      this.Component = null;
    var dependencies = [componentPath, 'imports/flight/registry'];
      requirejs(dependencies, function(Component, registry) {
        registry.reset();
      this.Component = Component;
});
waitsFor(function() {
    // when this returns true, Component is available and
// tests can be run
  this.Component !== null;
});
});
```

```
afterEach(function() {
  this.component && this.component.teardown();
});
specDefinitions.apply(this);
  });
};
```

Now that the spec has access to the component constructor, it can create a component instance. A new component instance needs to be created before each test. The setupComponent method creates a component and attaches it to a DOM node. HTML fixtures can be created by passing an HTML string or jQuery object as the first argument. Essentially, it works the same way as Component.attachTo.

```
var setupComponent = function (optionsOrHtml, options) {
    jasmine.getEnv().currentSpec.setupComponent(optionsOrHtml,
      options);
};

jasmine.Spec.prototype.setupComponent = function (node, options) {
  // node can be a selector, an html string or a jQuery object.
  if (typeof node === "string") {
    this.$node = $(node);
  } else if (node instanceof jQuery) {
    this.$node = node;
  } else {
    // both node and options are optional. If node is
    // found to be an object but not a jQuery object
    // it can be assumed it is an options object
    options = node;
  }
  if (!this.$node) {
    this.$node = $('<div id="fixture"></div>');
  }
  $('body').append(this.$node);
  this.component = new Component(this.$node, options || {});
});
```

Then in the spec:

```
describeComponent('app/ui/task_list', function() {
  beforeEach(function() {
    setupComponent({
      itemSelector: '.js-task-list-item',
```

```
        taskListContainerSelector: '.js-task-item-container'
      });
    });
    describe('Should do something', function() {
      // this.$node is the component's root node
      // this.component is an instance of the component
    });
  });
```

Event assertions

Out of the box, Jasmine does not include event-based assertions. Flight tests will often be of the form assert that event X was fired on node Y with data Z, but it is a pain to have to set up listeners and callbacks to test whether events were triggered correctly (as seen in the first code example in this chapter).

Luckily, help is at hand in the form of jasmine-jquery, an extension for Jasmine that provides event-based assertions and a lot more.

> Jasmine and jasmine-jquery are both available on GitHub at the following links:
> - `http://pivotal.github.com/jasmine/`
> - `https://github.com/velesin/jasmine-jquery`

The first example now becomes something much more readable.

```
describeComponent('app/ui/task_list', function() {
  beforeEach(function() {
    setupComponent();
    this.component.tasks = {
    1: {
      id: 1,
      description: 'Make tea'
    }
  };
  });
  describe('should handle uiNeedsTask',function() {
    it('and trigger dataTask', function() {
      var expected = {
    requestId: 100,
    task: {
      id: 1,
      description: 'Make tea'
    }
  };
```

```
spyOnEvent(document, 'dataTask');
this.$node.trigger('uiNeedsTask', {
  taskId: 1,
  requestId: 100
});
expect('dataTask').toHaveBeenTriggeredOnAndWith(do
  cument, expected);
    });
  });
});
```

Testing whether methods have been called

Although it may be tempting to write a test that checks if a specific method was called, or whether the called method is actually of any interest or not. Why it was called and what it did is much more interesting.

Often, the reason for testing whether a method was called is that the effect it has is otherwise untestable. It doesn't have clear output. If this is the case, then the method is a poor one. Change the way it works or (even better) don't write it in the first place.

This advice may seem facetious but is intended with all seriousness. BDD stands for Behavioral-Driven Development; the Driven is an important part. What it means is that you should write your tests first and let the tests drive the development.

Writing the tests after the fact is how you end up writing tests on methods lacking clear output.

The first step for writing any spec is figuring out what the behavior of a component should be. Which events is it going to consume? Which events is it going to trigger?

Summary

When testing Flight code, it is important to test component interfaces rather than specific methods.

A reference to a component can be gained by using `new` on the component constructor, allowing the component state to be altered.

Flight components and mixins can be effectively tested by extending existing JavaScript test frameworks to manage Flight's idiosyncrasies.

The following chapter discusses the peculiarities of Flight application architecture.

13
Complexities of Flight Architecture

Flight application designs should be quite straightforward, but complexities often arise due to a misunderstanding of what components are and when mixins should be used. This chapter discusses the problems with nesting components, and how to design solutions to complex problems while avoiding nested components.

The danger of nested components

It is often tempting to create nested components, where a component attaches another component to its DOM tree. Although Flight does not specifically prevent nesting, it does not lend itself well to it.

Teardown

When tearing down a component, the components it has instantiated are not automatically torn down with it, so it becomes necessary to do this manually. As the parent has no reference to the child, it must implement trigger events on its child components' root nodes, requesting that they teardown themselves.

This is, surprisingly, tough to manage and just feels wrong. A first thought may be that perhaps Flight should do this automatically, but there's a little more to it than that.

Atomic components

One of Flight's core tenets is the atomic nature of components. They can (and should) be developed and tested in isolation, allowing one component to instantiate another's results in a dependency between the two. They are inextricably intertwined. If at any point in time a refactor is required, or one of the components needs to be used in a different environment, this dependency can be difficult to work around.

Testing

Testing becomes challenging when components instantiate other components, because the second component must be instantiated as well.

It is tough to test whether the child component is successfully instantiated, and if more fixtures and mocks are required. This means more boilerplate, more complexity, and reduced reliability.

Creating a flat component structure

We have already seen an example of flat structure in the Task Manager application. The task list component manages all the tasks in the list, rather than creating a separate component for each task.

However, in more complex components it can be hard to see exactly how to create flat structures.

It helps to have a good understanding of when to use a mixin and when to use a component. This tends to be more obvious with data components, so in this chapter we will only concern ourselves with the UI.

Mixins versus components

Let's look at a real-world example, TweetDeck's search. NB: TweetDeck is undergoing continuous development, so these examples may not reflect the current state of the application.

At the top of the navigation bar is the **Search** input box as shown in the following screenshot:

When the **Search** input has focus, TweetDeck displays a list of recent searches using typeahead.
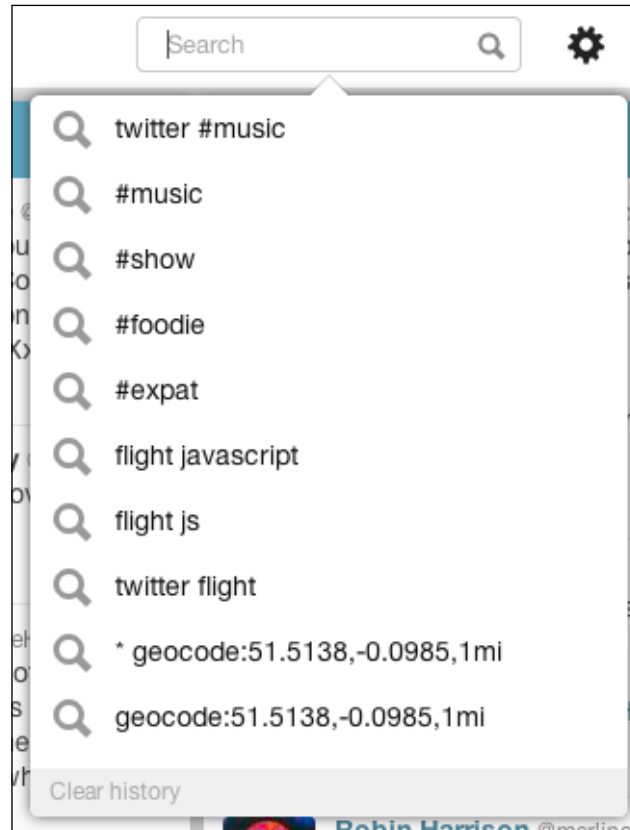
Selecting an item from typeahead or entering a search term and pressing *return* will show search results in place of typeahead.

And then selecting the **Users** button will switch to the user search results:
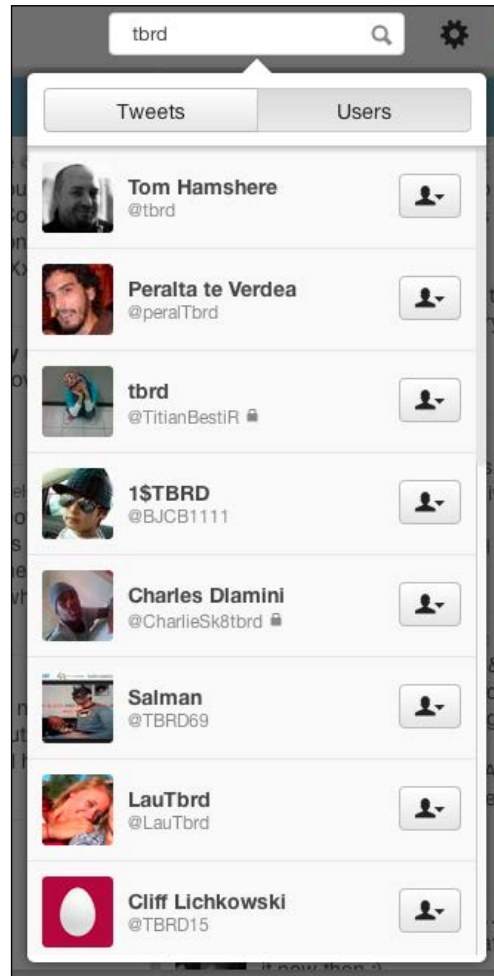
Working with components

There are four states that suggest four components:

- search
- typeahead
- search_results
- user_search_results

search would, depending on the current state, instantiate either typeahead or search_results. search_results would then create user_search_results when required.

However, there is a fair bit of complexity in this structure. For a start, to avoid conflicts in event handling, only one of typeahead, search_results, and user_search_results can exist at any one time, so for transition between them one needs to be torn down and the other instantiated. When an item is selected from typeahead, it needs to tear itself down and trigger an event to be handled by search which then instantiates search_results and triggers a search event.

In addition, when the user closes search_results, search needs to know how to use typeahead instead. Of course, if a search comes in from any other route, it needs to instantiate search_results. Unless search_results or user_search_results is already active, in which case it needs to tear that down but make sure it doesn't get confused and instantiate typeahead, or try to teardown search_results twice.

With all the events flying back and forth between these four components, it becomes tough to figure out exactly what is happening, or even what should be happening.

As you can see with this example, nested components cause complex event chains to arise quickly, and must be carefully thought through to prevent event loops and to ensure the correct state is maintained.

Of course, there is a better way…

Working with mixins

A much simpler structure would be to declare search_input, typeahead, search_results, and user_search_results as mixins to an all-encompassing search component.

```
defineComponent(search, search_input, typeahead, search_results,
  user_search_results);
```

This allows each aspect of the search (typeahead, search results, and so on) to be re-used elsewhere in the application. Some complexity remains in determining which of typeahead, search_results, and user_search_results is currently active, but this could be managed using Booleans on the component attributes:

```
function search() {
  defaultAttrs({
    isTypeaheadActive: true,
    isSearchResultsActive: false,
```

```
      isUserSearchResultsActive: false
  });
  }
```

The `search` component now includes all functionalities related to the search. It knows which aspect of the search is supposed to be active, and can call methods on the mixins directly.

Summary

Components are atomic. They should not depend on, or even assume the existence of, any other component. This precludes the nesting of components.

It is important to consider reusability when creating components and mixins. If part of a component should be re-usable, consider extracting it to a mixin.

Nested components are not atomic and therefore are less re-usable and tougher to test.

Flight API Reference

The following sections provide an overview of the essential API methods, including how to create components and mixins, how to use Advice, listen for and trigger events, and define default attributes.

> The complete Twitter API documentation can be found at
> `https://github.com/flightjs/flight`.

Components

There are three aspects to the component API: component and mixin definition, component instantiation, and base component methods. The base component methods are available as first-class citizens from within component and mixins definitions.

Component definition

Components are AMD modules and should return a defined component. This is an example component file:

```
// AMD module definition
define(function () {
  // require defineComponent
  var defineComponent = require('flight/lib/component');
  // export defined component
  return defineComponent(myComponent);
  // component definition
```

```
    function myComponent () {
      // internal methods
      this.foo = function () {
        // do something
      };
    };
  });
```

Mixin definition

Mixins are AMD modules exactly like components, only they return a component definition, rather than a defined component, so don't require `defineComponent`:

```
  // AMD module definition
  define(function () {
    return myMixin;
    // mixin definition
    function myMixin () {
      // internal methods
      this.bar = function () {
        // do something
      };
    };
  });
```

Using mixins

Mixins can be mixed into components, as shown in the following code:

```
  // AMD module definition
  define(function () {
    // require defineComponent
    var defineComponent = require('flight/lib/component');
    // require mixins
    var myMixin = require('component/my_mixin');
    // export defined component, adding mixins
    return defineComponent(myComponent, myMixin);
    // component definition
    function myComponent () {
      // internal methods
      this.foo = function () {
        // components have access to mixin methods
        // as first-class citizens
```

```
      this.bar();
    };
  };
});
```

Mixins can also be mixed into other mixins using `compose.mixin` within the mixin definition. For example, look at the following code:

```
// AMD module definition
define(function () {
  // require Compose
  var compose = require('flight/lib/compose');
  // require another mixin
  var someOtherMixin = require('component/some_other_mixin');

  return myMixin;
  // mixin definition
  function myMixin () {
    compose.mixin(this, [someOtherMixin]);
    // internal methods
    this.bar = function () {
      // do something
    };
  };
});
```

Instantiating components

The following code example shows a component (`component/my_component.js`) being required and then instantiated in three different ways:

```
// require component constructor
var MyComponent = require('component/my_component');
// attach component instance to an HTMLElement
MyComponent.attachTo(document);
// attach component instance to all elements matching a css
  selector
MyComponent.attachTo('.modal');
// attach component instance to a jQuery object
var $div = $('<div />');
MyComponent.attachTo($div);
// pass options to component instance
MyComponent.attachTo($div, {
  defaultText: 'Hello, World!'
});
```

Methods available on a component instance

The `defineComponent` method mixes in Flight's core functionality into a component definition. The methods in this section are available to any component within the function definition.

Component methods can be broken up into five sections: Advice, Attributes, DOM selection, Events, and Teardown.

Advice

Advice allows component and mixin methods to be extended or overridden without clobbering the original method using `before`, `after`, and `around`. Advice is also used to initialize a component.

before

Execute a method before the named method is executed:

```
this.before(methodName, callback);
```

after

Execute a method after the named method is executed:

```
this.after(methodName, callback);
```

around

Execute a method before the named method is executed, and optionally call the named method:

```
this.around(methodName, function (originalMethod) {
  // maybe call
originalMethod();
});
```

Initialization

Use `after` to initialize a component.

```
// component definition
function myComponent () {
  this.after('initialize', function () {
    // do initialization
    // this will be called when a component is
```

```
// attached to a DOM node
  });
}
```

defaultAttrs

Default attributes can be used to configure a component or mixin:

```
// component definition
function myComponent () {
this.defaultAttrs({
defaultText: "Hello, handsome!"
});
});
```

Default attributes can be overridden when instantiating a component:

```
var MyComponent = require('component/my_component');
MyComponent.attachTo(document, {
  defaultText: "Hello, World!"
});
```

select

Use a selector defined in `defaultAttrs` to gain a reference to a jQuery collection containing elements within the component that match the selector:

```
this.defaultAttrs({
  listItemSelector: '.list-item'
});
this.getListItems = function () {
var $listItems = this.select(attrKey);
return $listItems;
};
```

Events

Flight components should attach event listeners and trigger events through Flight's event methods:

on

Attach an event listener to the component root:

```
this.on([selector], eventName, callback);
```

In the `this` example, an event listener is attached to the component's root node:

```
this.handleTaskData = function (event, data) {
  // do something
};
this.after('initialize', function () {
this.on('dataTask', this.handleTaskData);
});
```

To attach an event listener to a specific element, a CSS selector, DOM element, or jQuery collection can be passed as the first argument to `on`, as seen in the following examples:

```
this.on('.list-item', 'click', this.handleListItemClick);
this.on($('.list-item'), 'click', this.handleListItemClick);
this.on(document, 'dataTask', this.handleTaskData);
```

To handle dynamic creation of elements, use the following syntax to provide event delegation. `attributeKey` needs to be defined in `defaultAttrs`.

```
this.on(eventName, {
  attributeKey: callback
});
```

In this example, `handleListItemClick` will be called when the user clicks on an element matching `.list-item`.

```
this.defaultAttrs({
  listItemSelector: '.list-item'
});
this.handleListItemClick = function (event) {
  // do something
});
this.after('initialize', function () {
this.on('click' {
    listItemSelector: this.handleListItemClick
});
});
```

off

Remove an event listener:

```
this.off([selector, ] eventName [, callback]);
```

In this example, all listeners listening for `dataTask` events will be removed.

```
this.off('dataTask');
```

trigger

Trigger an event:

```
this.trigger([selector, ] eventName [, data]);
```

This example shows a `dataTask` event being triggered on the component's root node with a data payload.

```
this.trigger('dataTask', {
  description: 'Make tea'
});
```

teardown

`teardown()` destroys the component instance and removes all event handlers declared with `this.on`.

```
this.teardown();
```

Using Flight's registry

It is possible to import Flight's registry module to provide access to advanced features around component management. The registry keeps track of which component is attached to which node. The following are a few useful methods:

findInstanceInfoByNode

Returns an array of all component instances for a given node:

```
registry.findInstanceInfoByNode(node)
```

findInstanceInfo

Return a component instance if it exists in the registry, otherwise null:

```
registry.findInstanceInfo(componentInstance)
```

allInstances

This is an array of all components in the registry. Here's an example using `allInstances` to tear down all components:

```
registry.allInstances.forEach(function(i) {
 registry.allInstances[i].instance.teardown();
});
```

Index

Thank you for buying
Getting Started with Twitter Flight

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

## About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
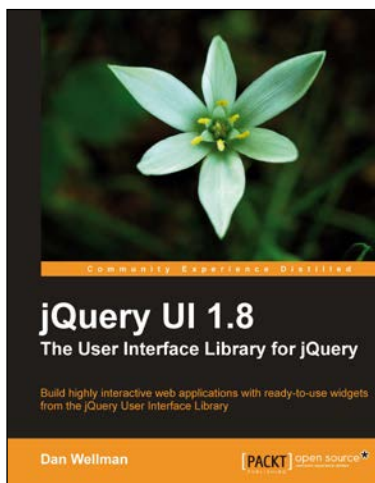
jQuery UI 1.8: The User Interface Library for jQuery

ISBN: 978-1-849516-52-5   Paperback: 424 pages

Build highly interactive web applications with ready-to-use widgets from the jQuery User Interface Library

1. Packed with examples and clear explanations of how to easily design elegant and powerful frontend interfaces for your web applications

2. A section covering the widget factory including an in-depth example on how to build a custom jQuery UI widget

3. Updated code with significant changes and fixes to the previous edition

jQuery UI Themes Beginner's Guide

ISBN: 978-1-849510-44-8   Paperback: 268 pages

Create new themes for your jQuery site with this step-by-step guide

1. Learn the details of the jQuery UI theme framework by example

2. No prior knowledge of jQuery UI or theming frameworks is necessary

3. The CSS structure is explained in an easy-to-understand and approachable way

Please check **www.PacktPub.com** for information on our titles

Twitter Bootstrap Web Development How-To

ISBN: 978-1-849518-82-6   Paperback: 68 pages

A hands-on introduction to building websites with Twitter Bootstrap's powerful front-end development framework

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results

2. Conquer responsive website layout with Bootstrap's flexible grid system

3. Leverage carefully-built CSS styles for typography, buttons, tables, forms, and more
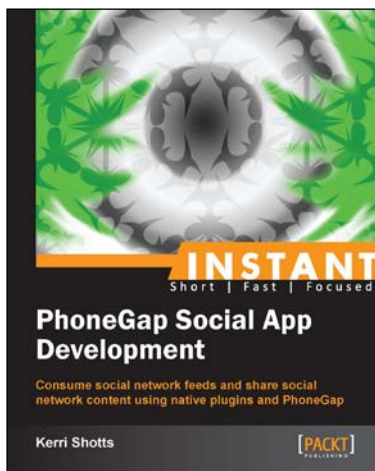
Instant PhoneGap Social App Development

ISBN: 978-1-849696-28-9   Paperback: 78 pages

Consume social network feeds and share social network content using native plugins and PhoneGap

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results

2. Learn how to consume content using Twitter's JSON API

3. Learn how to use the Twitter Web Intents to share content on the Twitter social network

Please check **www.PacktPub.com** for information on our titles