Flux Architecture

Flux helps you to build data-rich applications that engage your users, and scale to meet every demand. It is a key part of the Facebook technology stack that serves billions of users every day.

This book will start by introducing the Flux pattern and help you get an understanding of what it is and how it works. After this, we'll build real-world React applications that highlight the power and simplicity of Flux in action. Finally, we look at the landscape of Flux and explore the Alt and Redux libraries, which make React and Flux developments easier.

Filled with fully-worked examples and code-first explanations, by the end of the book, you'll not only have a rock-solid understanding of the architecture, but will be ready to implement Flux architecture in anger.

Who this book is written for

Are you trying to use React, but are struggling to get your head around Flux? Maybe you're tired of Mv* spaghetti code at scale? Do you find yourself asking what the Flux?!

Flux Architecture will guide you through everything you need to understand the Flux pattern, and design and build powerful web applications that rely on the Flux architecture.

You don't need to know what Flux or ReactJS to read the book. But it is recommended that you have a good working knowledge of JavaScript.

What you will learn from this book

- Understand the Flux pattern and how it will impact your React applications
- Build real-world applications that rely on Flux
- Handle asynchronous actions in your application
- Implement immutable stores with Immutable.js
- Replace React.js with alternate View components such as jQuery and Handlebars
- Test and benchmark your Flux architecture using Jest—Facebook's enhancement of the Jasmine library


Adam Boduch

Flux Architecture

Learn to build powerful and scalable applications with Flux, the architecture that serves billions of Facebook users every day

Community Experience Distilled
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'What is Flux?'
- A synopsis of the book’s content
- More information on Flux Architecture
Adam Boduch has been involved with large-scale JavaScript development for nearly 10 years. Before moving to the front end, he worked on several large-scale cloud computing products using Python and Linux. No stranger to complexity, Adam has practical experience with real-world software systems and the scaling challenges they pose.

He is the author of several JavaScript books, including *JavaScript Concurrency*, and is passionate about innovative user experiences and high performance.
I love Backbone.js. It's an amazing little library that does so much with so little. It's also unopinionated—there are endless ways to do the same thing. This last point gives many Backbone.js programmers a headache. The freedom to implement things the way we see fit is great, until we start making those unavoidable consistency errors.

When I first started with Flux, I couldn't really see how such an architecture could help out a mere Backbone.js programmer. Eventually, I figured out two things. First, Flux is unopinionated where it matters—the implementation specifics. Two, Flux is very much like Backbone in the spirit of minimal moving parts that do one thing well.

As I started experimenting with Flux, I realized that Flux provides the missing architectural perspective that enables scalability. Where Backbone.js and other related technologies fall apart is when something goes wrong. In fact, these bugs can be so difficult that they're never actually fixed—the whole system is scarred with workarounds.

I decided to write this book in the hope that other programmers, from all walks of JavaScript, can experience the same level of enlightenment as I have working with this wonderful technology from Facebook.

**What this book covers**

*Chapter 1, What is Flux?,* gives an overview of what Flux is and why it was created.

*Chapter 2, Principles of Flux,* talks about the core concepts of Flux and the essential knowledge for building a Flux architecture.

*Chapter 3, Building a Skeleton Architecture,* walks through the steps involved in building a skeleton architecture before implementing application features.
Chapter 4, Creating Actions, shows how action creator functions are used to feed new data into the system while describing something that just happened.

Chapter 5, Asynchronous Actions, goes through examples of asynchronous action creator functions and how they fit within a Flux architecture.

Chapter 6, Changing Flux Store State, gives many detailed explanations and examples that illustrate how Flux stores work.

Chapter 7, Viewing Information, gives many detailed explanations and examples that illustrate how Flux views work.

Chapter 8, Information Lifecycle, talks about how information in a Flux architecture enters the system and how it ultimately exits the system.

Chapter 9, Immutable Stores, shows how immutability is a key architectural property of software architectures, such as Flux, where data flows in one direction.

Chapter 10, Implementing a Dispatcher, walks through the implementation of a dispatcher component, instead of using the Facebook reference implementation.

Chapter 11, Alternative View Components, shows how view technologies other than React can be used within a Flux architecture.

Chapter 12, Leveraging Flux Libraries, gives an overview of two popular Flux libraries—Alt.js and Redux.

Chapter 13, Testing and Performance, talks about testing components from within the context of a Flux architecture and discusses performance testing your architecture.

Chapter 14, Flux and the Software Development Life Cycle, discusses the impact Flux has on the rest of the software stack and how to package Flux features.
What is Flux?

Flux is supposed to be this great new way of building complex user interfaces that scale well. At least that's the general messaging around Flux, if you're only skimming the Internet literature. But, how do we define this great new way of building user interfaces? What makes it superior to other more established frontend architectures?

The aim of this chapter is to cut through the sales bullet points and explicitly spell out what Flux is, and what it isn't, by looking at the patterns that Flux provides. And since Flux isn't a software package in the traditional sense, we'll go over the conceptual problems that we're trying to solve with Flux.

Finally, we'll close the chapter by walking through the core components found in any Flux architecture, and we'll install the Flux npm package and write a hello world Flux application right away. Let's get started.

Flux is a set of patterns

We should probably get the harsh reality out of the way first—Flux is not a software package. It's a set of architectural patterns for us to follow. While this might sound disappointing to some, don't despair—there's good reasons for not implementing yet another framework. Throughout the course of this book, we'll see the value of Flux existing as a set of patterns instead of a de facto implementation. For now, we'll go over some of the high-level architectural patterns put in place by Flux.
**Data entry points**

With traditional approaches to building frontend architectures, we don't put much thought into how data enters the system. We might entertain the idea of data entry points, but not in any detail. For example, with MVC (Model View Controller) architectures, the controller is supposed control the flow of data. And for the most part, it does exactly that. On the other hand, the controller is really just about controlling what happens after it already has the data. How does the controller get data in the first place? Consider the following illustration:

At first glance, there's nothing wrong with this picture. The data-flow, represented by the arrows, is easy to follow. But where does the data originate? For example, the view can create new data and pass it to the controller, in response to a user event. A controller can create new data and pass it to another controller, depending on the composition of our controller hierarchy. What about the controller in question—can it create data itself and then use it?

In a diagram such as this one, these questions don't have much virtue. But, if we're trying to scale an architecture to have hundreds of these components, the points at which data enters the system become very important. Since Flux is used to build architectures that scale, it considers data entry points an important architectural pattern.
Managing state
State is one of those realities we need to cope with in frontend development. Unfortunately, we can't compose our entire application of pure functions with no side-effects for two reasons. First, our code needs to interact with the DOM interface, in one way or another. This is how the user sees changes in the UI. Second, we don't store all our application data in the DOM (at least we shouldn't do this). As time passes and the user interacts with the application, this data will change.

There's no cut-and-dry approach to managing state in a web application, but there are several ways to limit the amount of state changes that can happen, and enforce how they happen. For example, pure functions don't change the state of anything, they can only create new data. Here's an example of what this looks like:

As you can see, there's no side-effects with pure functions because no data changes state as a result of calling them. So why is this a desirable trait, if state changes are inevitable? The idea is to enforce where state changes happen. For example, perhaps we only allow certain types of components to change the state of our application data. This way, we can rule out several sources as the cause of a state change.

Flux is big on controlling where state changes happen. Later on in the chapter, we'll see how Flux stores manage state changes. What's important about how Flux manages state is that it's handled at an architectural layer. Contrast this with an approach that lays out a set of rules that say which component types are allowed to mutate application data – things get confusing. With Flux, there's less room for guessing where state changes take place.

Keeping updates synchronous
Complimentary to data entry points is the notion of update synchronicity. That is, in addition to managing where the state changes originate from, we have to manage the ordering of these changes relative to other things. If the data entry points are the what of our data, then synchronously applying state changes across all the data in our system is the when.
What is Flux?

Let's think about why this matters for a moment. In a system where data is updated asynchronously, we have to account for race conditions. Race conditions can be problematic because one piece of data can depend on another, and if they're updated in the wrong order, we see cascading problems, from one component to another. Take a look at this diagram, which illustrates this problem:

![Diagram](image1.png)

When something is asynchronous, we have no control over when that something changes state. So, all we can do is wait for the asynchronous updates to happen, and then go through our data and make sure all of our data dependencies are satisfied. Without tools that automatically handle these dependencies for us, we end up writing a lot of state-checking code.

Flux addresses this problem by ensuring that the updates that take place across our data stores are synchronous. This means that the scenario illustrated in the preceding diagram isn't possible. Here's a better visualization of how Flux handles the data synchronization issues that are typical of JavaScript applications today:

![Diagram](image2.png)
Information architecture

It's easy to forget that we work in information technology and that we should be building technology around information. In recent times, however, we seem to have moved in the other direction, where we're forced to think about implementation before we think about information. More often than not, the data exposed by the sources used by our application doesn't have what the user needs. It's up to our JavaScript to turn this raw data into something consumable by the user. This is our information architecture.

Does this mean that Flux is used to design information architectures as opposed to a software architecture? This isn't the case at all. In fact, Flux components are realized as true software components that perform actual computations. The trick is that Flux patterns enable us to think about information architecture as a first-class design consideration. Rather than having to sift through all sorts of components and their implementation concerns, we can make sure that we're getting the right information to the user.

Once our information architecture takes shape, the larger architecture of our application follows, as a natural extension to the information we're trying to communicate to our users. Producing information from data is the difficult part. We have to distill many sources of data into not only information, but information that's also of value to the user. Getting this wrong is a huge risk for any project. When we get it right, we can then move on to the specific application components, like the state of a button widget, and so on.

Flux architectures keep data transformations confined to their stores. A store is an information factory—raw data goes in and new information comes out. Stores control how data enters the system, the synchronicity of state changes, and they define how the state changes. When we go into more depth on stores as we progress through the book, we'll see how they're the pillars of our information architecture.

Flux isn't another framework

Now that we've explored some of the high-level patterns of Flux, it's time to revisit the question: what is Flux again? Well, it is just a set of architectural patterns we can apply to our frontend JavaScript applications. Flux scales well because it puts information first. Information is the most difficult aspect of software to scale; Flux tackles information architecture head on.

So, why aren't Flux patterns implemented as a framework? This way, Flux would have a canonical implementation for everyone to use; and like any other large scale open source project, the code would improve over time as the project matures.
What is Flux?

The main problem is that Flux operates at an architectural level. It's used to address information problems that prevent a given application from scaling to meet user demand. If Facebook decided to release Flux as yet another JavaScript framework, it would likely have the same types of implementation issues that plague other frameworks out there. For example, if some component in a framework isn't implemented in a way that best suits the project we're working on, then it's not so easy to implement a better alternative, without hacking the framework to bits.

What's nice about Flux is that Facebook decided to leave the implementation options on the table. They do provide a few Flux component implementations, but these are reference implementations. They're functional, but the idea is that they're a starting point for us to understand the mechanics of how things such as dispatchers are expected to work. We're free to implement the same Flux architectural pattern as we see it.

Flux isn't a framework. Does this mean we have to implement everything ourselves? No, we do not. In fact, developers are implementing Flux libraries and releasing them as open source projects. Some Flux libraries stick more closely to the Flux patterns than others. These implementations are opinionated, and there's nothing wrong with using them if they're a good fit for what we're building. The Flux patterns aim to solve generic conceptual problems with JavaScript development, so you'll learn what they are before diving into Flux implementation discussions.

Flux solves conceptual problems

If Flux is simply a collection of architectural patterns instead of a software framework, what sort of problems does it solve? In this section, we'll look at some of the conceptual problems that Flux addresses from an architectural perspective. These include unidirectional data-flow, traceability, consistency, component layering, and loosely coupled components. Each of these conceptual problems pose a degree of risk to our software, in particular the ability to scale it. Flux helps us get out in front of these issues as we're building the software.

Data flow direction

We're creating an information architecture to support the feature-rich application that will ultimately sit on top of this architecture. Data flows into the system and will eventually reach an endpoint, terminating the flow. It's what happens in between the entry point and the termination point that determines the data-flow within a Flux architecture. This is illustrated here:
Data flow is a useful abstraction, because it's easy to visualize data as it enters the system and moves from one point to another. Eventually, the flow stops. But before it does, several side-effects happen along the way. It's that middle block in the preceding diagram that's concerning, because we don't know exactly how the data-flow reached the end.

Let's say that our architecture doesn't pose any restrictions on data flow. Any component is allowed to pass data to any other component, regardless of where that component lives. Let's try to visualize this setup:

As you can see, our system has clearly defined entry and exit points for our data. This is good because it means that we can confidently say that the data-flows through our system. The problem with this picture is with how the data-flows between the components of the system. There's no direction, or rather, it's multidirectional. This isn't a good thing.
What is Flux?

Flux is a unidirectional data flow architecture. This means that the preceding component layout isn't possible. The question is—why does this matter? At times, it might seem convenient to be able to pass data around in any direction, that is, from any component to any other component. This in and of itself isn't the issue—passing data alone doesn't break our architecture. However, when data moves around our system in more than one direction, there's more opportunity for components to fall out of sync with one another. This simply means that if data doesn't always move in the same direction, there's always the possibility of ordering bugs.

Flux enforces the direction of data-flows, and thus eliminates the possibility of components updating themselves in an order that breaks the system. No matter what data has just entered the system, it'll always flow through the system in the same order as any other data, as illustrated here:

![Data Flow Diagram]

**Predictable root cause**

With data entering our system and flowing through our components in one direction, we can more easily trace any effect to it's cause. In contrast, when a component sends data to any other component residing in any architectural layer, it's a lot more difficult to figure how the data reached its destination. Why does this matter? Debuggers are sophisticated enough that we can easily traverse any level of complexity during runtime. The problem with this notion is that it presumes we only need to trace what's happening in our code for the purposes of debugging.
Flux architectures have inherently predictable data-flows. This is important for a number of design activities and not just debugging. Programmers working on Flux applications will begin to intuitively sense what's going to happen. Anticipation is key, because it lets us avoid design dead-ends before we hit them. When the cause and effect are easy to tease out, we can spend more time focusing on building application features—the things the customers care about.

**Consistent notifications**

The direction in which we pass data from component to component in Flux architectures should be consistent. In terms of consistency, we also need to think about the mechanism used to move data around our system.

For example, publish/subscribe (pub/sub) is a popular mechanism used for inter-component communication. What's neat about this approach is that our components can communicate with one another, and yet we're able to maintain a level of decoupling. In fact, this is fairly common in frontend development because component communication is largely driven by user events. These events can be thought of as fire-and-forget. Any other components that want to respond to these events in some way, need to take it upon themselves to subscribe to the particular event.

While pub/sub does have some nice properties, it also poses architectural challenges, in particular scaling complexities. For example, let's say that we've just added several new components for a new feature. Well, in which order do these components receive update messages relative to pre-existing components? Do they get notified after all the pre-existing components? Should they come first? This presents a data dependency scaling issue.

The other challenge with pub-sub is that the events that get published are often fine-grained to the point where we'll want to subscribe and later unsubscribe from the notifications. This leads to consistency challenges because trying to code lifecycle changes when there's a large number of components in the system is difficult and presents opportunities for missed events.
What is Flux?

The idea with Flux is to sidestep the issue by maintaining a static inter-component messaging infrastructure that issues notifications to every component. In other words, programmers don't get to pick and choose the events their components will subscribe to. Instead, they have to figure out which of the events that are dispatched to them are relevant, ignoring the rest. Here's a visualization of how Flux dispatches events to components:

![flux-diagram](image)

The Flux dispatcher sends the event to every component; there's no getting around this. Instead of trying to fiddle with the messaging infrastructure, which is difficult to scale, we implement logic within the component to determine whether or not the message is of interest. It's also within the component that we can declare dependencies on other components, which helps influence the ordering of messages. We'll cover this in much more detail in later chapters.

Simple architectural layers

Layers can be a great way to organize an architecture of components. For one thing, it's an obvious way to categorize the various components that make up our application. For another thing, layers serve as a means to put constraints around communication paths. This latter point is especially relevant to Flux architectures since it's important that data flow in one direction. It's much easier to apply constraints to layers than it is to individual components. Here is an illustration of Flux layers:

![flux-layers-diagram](image)
This diagram isn't intended to capture the entire data flow of a Flux architecture, just how data-flows between the main three layers. It also doesn't give any detail about what's in the layers. Don't worry, the next section gives introductory explanations of the types of Flux components, and the communication that happens between the layers is the focus of this book.

As you can see, the data-flows from one layer to the next, in one direction. Flux only has a few layers, and as our applications scale in terms of component counts, the layer counts remains fixed. This puts a cap on the complexity involved with adding new features to an already large application. In addition to constraining the layer count and the data-flow direction, Flux architectures are strict about which layers are actually allowed to communicate with one another.

For example, the action layer could communicate with the view layer, and we would still be moving in one direction. We would still have the layers that Flux expects. However, skipping a layer like this is prohibited. By ensuring that layers only communicate with the layer directly beneath it, we can rule out bugs introduced by doing something out-of-order.

**Loosely coupled rendering**

One decision made by the Flux designers that stands out is that Flux architectures don't care how UI elements are rendered. That is to say, the view layer is loosely coupled to the rest of the architecture. There are good reasons for this.

Flux is an information architecture first, and a software architecture second. We start with the former and graduate toward the latter. The challenge with view technology is that it can exert a negative influence on the rest of the architecture. For example, one view has a particular way of interacting with the DOM. Then, if we've already decided on this technology, we'll end up letting it influence the way our information architecture is structured. This isn't necessarily a bad thing, but it can lead to us making concessions about the information we ultimately display to our users.

What we should really be thinking about is the information itself and how this information changes over time. What actions are involved that bring about these changes? How is one piece of data dependent on another piece of data? Flux naturally removes itself from the browser technology constraints of the day so that we can focus on the information first. It's easy to plug views into our information architecture as it evolves into a software product.
What is Flux?

**Flux components**

In this section, we'll begin our journey into the concepts of Flux. These concepts are the essential ingredients used in formulating a Flux architecture. While there's no detailed specifications for how these components should be implemented, they nevertheless lay the foundation of our implementation. This is a high-level introduction to the components we'll be implementing throughout this book.

**Action**

Actions are the *verbs* of the system. In fact, it's helpful if we derive the name of an action directly from a sentence. These sentences are typically statements of functionality – something we want the application to do. Here are some examples:

- Fetch the session
- Navigate to the settings page
- Filter the user list
- Toggle the visibility of the details section

These are simple capabilities of the application, and when we implement them as part of a Flux architecture, actions are the starting point. These human-readable action statements often require other new components elsewhere in the system, but the first step is always an action.

So, what exactly is a Flux action? At it's simplest, an action is nothing more than a string—a name that helps identify the purpose of the action. More typically, actions consist of a *name* and a *payload*. Don't worry about the payload specifics just yet—as far as actions are concerned, they're just opaque pieces of data being delivered into the system. Put differently, actions are like mail parcels. The entry point into our Flux system doesn't care about the internals of the parcel, only that they get to where they need to go. Here's an illustration of actions entering a Flux system:

![Action Diagram](image-url)

---

[12]
This diagram might give the impression that actions are external to Flux, when in fact they're an integral part of the system. The reason this perspective is valuable is because it forces us to think about actions as the only means to deliver new data into the system.

**Golden Flux Rule**: If it's not an action, it can't happen.

**Dispatcher**

The dispatcher in a Flux architecture is responsible for distributing actions to the store components (we'll talk about stores next). A dispatcher is actually kind of like a broker—if actions want to deliver new data to a store, they have to talk to the broker, so it can figure out the best way to deliver them. Think about a message broker in a system like RabbitMQ. It's the central hub where everything is sent before it's actually delivered. Here is a diagram depicting a Flux dispatcher receiving actions and dispatching them to stores:

The earlier section of this chapter—"simple architectural layers"—didn't have an explicit layer for dispatchers. That was intentional. In a Flux application, there's only one dispatcher. It can be thought of more as a pseudo layer than an explicit layer. We know the dispatcher is there, but it's not essential to this level of abstraction. What we're concerned about at an architectural level is making sure that when a given action is dispatched, we know that it's going to make its way to every store in the system.

Having said that, the dispatcher's role is critical to how Flux works. It's the place where store callback functions are registered and it's how data dependencies are handled. Stores tell the dispatcher about other stores that it depends on, and it's up to the dispatcher to make sure these dependencies are properly handled.
What is Flux?

Golden Flux Rule: The dispatcher is the ultimate arbiter of data dependencies.

Store

Stores are where state is kept in a Flux application. Typically, this means the application data that's sent to the frontend from the API. However, Flux stores take this a step further and explicitly model the state of the entire application. If this sounds confusing or like a generally bad idea, don't worry—we'll clear this up as we make our way through subsequent chapters. For now, just know that stores are where state that matters can be found. Other Flux components don't have state—they have implicit state at the code level, but we're not interested in this, from an architectural point of view.

Actions are the delivery mechanism for new data entering the system. The term *new data* doesn't imply that we're simply appending it to some collection in a store. All data entering the system is new in the sense that it hasn't been dispatched as an action yet—it could in fact result in a store changing state. Let's look at a visualization of an action that results in a store changing state:

![Visualization of an action that results in a store changing state]

The key aspect of how stores change state is that there's no external logic that determines a state change should happen. It's the store, and only the store, that makes this decision and then carries out the state transformation. This is all tightly encapsulated within the store. This means that when we need to reason about particular information, we need not look any further than the stores. They're their own boss—they're self-employed.

Golden Flux Rule: Stores are where state lives, and only stores themselves can change this state.
The last Flux component we're going to look at in this section is the view, and it technically isn't even a part of Flux. At the same time, views are obviously a critical part of our application. Views are almost universally understood as the part of our architecture that's responsible for displaying data to the user—it's the last stop as data-flows through our information architecture. For example, in MVC architectures, views take model data and display it. In this sense, views in a Flux-based application aren't all that different from MVC views. Where they differ markedly is with regard to handling events. Let's take a look at the following diagram:

Here we can see the contrasting responsibilities of a Flux view, compared with a view component found in your typical MVC architecture. The two view types have similar types of data flowing into them—application data used to render the component and events (often user input). What's different between the two types of view is what flows out of them.

The typical view doesn't really have any constraints in how its event handler functions communicate with other components. For example, in response to a user clicking a button, the view could directly invoke behavior on a controller, change the state of a model, or it might query the state of another view. On the other hand, the Flux view can only dispatch new actions. This keeps our single entry point into the system intact and consistent with other mechanisms that want to change the state of our store data. In other words, an API response updates state in the exact same way as a user clicking a button does.

Given that views should be restricted in terms of how data-flows out of them (besides DOM updates) in a Flux architecture, you would think that views should be an actual Flux component. This would make sense insofar as making actions the only possible option for views. However, there's also no reason we can't enforce this now, with the benefit being that Flux remains entirely focused on creating information architectures.
What is Flux?

Keep in mind, however, that Flux is still in its infancy. There's no doubt going to be external influences as more people start adopting Flux. Maybe Flux will have something to say about views in the future. Until then, views exist outside of Flux but are constrained by the unidirectional nature of Flux.

**Golden Flux Rule:** The only way data-flows out of a view is by dispatching an action.

## Installing the Flux package

We'll close the first chapter by getting our feet wet with some code, because everyone needs a hello world application under their belt. We'll also get some of our boilerplate code setup tasks out of the way too, since we'll be using a similar setup throughout the book.

We'll skip going over Node + NPM installation since it's sufficiently covered in great detail all over the Internet. We'll assume Node is installed and ready to go from this point forward.

The first NPM package we'll need installed is Webpack. This is an advanced module bundler that's well suited for modern JavaScript applications, including Flux-based applications. We'll want to install this package globally so that the `webpack` command gets installed on our system:

```bash
npm install webpack -g
```

With Webpack in place, we can build each of the code examples that ship with this book. However, our project does require a couple of local NPM packages, and these can be installed as follows:

```bash
npm install flux babel-core babel-loader babel-preset-es2015 --save-dev
```

The `--save-dev` option adds these development dependencies to our file, if one exists. This is just to get started—it isn't necessary to manually install these packages to run the code examples in this book. The examples you've downloaded already come with a `package.json`, so to install the local dependencies, simply run the following from within the same directory as the `package.json` file:

```bash
npm install
```
Now the `webpack` command can be used to build the example. This is the only example in the first chapter, so it's easy to navigate to within a terminal window and run the `webpack` command, which builds the `main-bundle.js` file. Alternatively, if you plan on playing with the code, which is obviously encouraged, try running `webpack --watch`. This latter form of the command will monitor for file changes to the files used in the build, and run the build whenever they change.

This is indeed a simple hello world to get us off to a running start, in preparation for the remainder of the book. We've taken care of all the boilerplate setup tasks by installing Webpack and its supporting modules. Let's take a look at the code now. We'll start by looking at the markup that's used.

```html
<!doctype html>
<html>
<head>
  <title>Hello Flux</title>
  <script src="main-bundle.js" defer></script>
</head>
<body>
</body>
</html>
```

Not a lot to it is there? There isn't even content within the `body` tag. The important part is the `main-bundle.js` script—this is the code that's built for us by Webpack. Let's take a look at this code now:

```javascript
// Imports the "flux" module.
import * as flux from 'flux';

// Creates a new dispatcher instance. "Dispatcher" is
// the only useful construct found in the "flux" module.
const dispatcher = new flux.Dispatcher();

// Registers a callback function, invoked every time
// an action is dispatched.
dispatcher.register((e) => {
  var p;

  // Determines how to respond to the action. In this case,
  // we're simply creating new content using the "payload"
  // property. The "type" property determines how we create
  // the content.
  switch (e.type) {
    case 'hello':
      p = document.createElement('p');
```
What is Flux?

```javascript
const dispatcher = new Flux.Dispatcher();

const p = document.createElement('p');
p.textContent = e.payload;
document.body.appendChild(p);
break;
case 'world':
p = document.createElement('p');
p.textContent = `${e.payload}!`;
p.style.fontWeight = 'bold';
document.body.appendChild(p);
break;
default:
    break;
}
});

// Dispatches a "hello" action.
dispatcher.dispatch({
    type: 'hello',
    payload: 'Hello'
});

// Dispatches a "world" action.
dispatcher.dispatch({
    type: 'world',
    payload: 'World'
});
```

As you can see, there's not much to this hello world Flux application. In fact, the only Flux-specific component this code creates is a dispatcher. It then dispatches a couple of actions and the handler function that's registered to the store processes the actions.

Don't worry that there's no stores or views in this example. The idea is that we've got the basic Flux NPM package installed and ready to go.
Summary

This chapter introduced you to Flux. Specifically, we looked at both what Flux is and what it isn't. Flux is a set of architectural patterns that, when applied to our JavaScript application, help with getting the data-flow aspect of our architecture right. Flux isn't yet another framework used for solving specific implementation challenges, be it browser quirks or performance gains—there's a multitude of tools already available for these purposes. Perhaps the most important defining aspect of Flux are the conceptual problems it solves—things like unidirectional data flow. This is a major reason that there's no de facto Flux implementation.

We wrapped the chapter up by walking through the setup of our build components used throughout the book. To test that the packages are all in place, we created a very basic hello world Flux application.

Now that we have a handle on what Flux is, it's time for us to look at why Flux is the way it is. In the following chapter, we'll take a more detailed look at the principles that drive the design of Flux applications.
Where to buy this book
You can buy Flux Architecture from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.