

4

Anatomy of a React Application

In an application of any reasonable complexity, React plays a significant but limited role. React is a component rendering and composition system for views. This definition leaves out many facets of a complete application. In this chapter, we will explore complex web application design as three aspects. We will also identify the subcomponents of each aspect and develop a rationale for choosing particular tools to service each aspect.

In this chapter, we are going to cover the following:

- What is a Single Page Application (SPA)?
- Aspects of a SPA design
- Build systems
- CSS preprocessors
- Compiling modern JS syntax and JSX templates
- Front-end architecture components
- Application design

The goal of this chapter is to become familiar with the structure of web applications and the technologies involved. In the following chapters, 5 through 9, a full-featured multi-user blog application will be built. The preparation provided here will guide you not only with the configuration of the blog application, but will also introduce you to a few design procedures. The design procedures will define the components of the app and how they are interconnected. In the application design section of this chapter, an email application is used as an example in order to illustrate the design tasks for the first time. In *Chapter 5, Starting a React Application*, the same procedures are followed once more for the blog application. Then, in chapters 6-9, we'll flesh out the feature code for the multi-user blog application prototype.

What is a single-page application?

A **single-page application (SPA)** is a rich application—one that delivers the same features, functionality, and sophistication normally associated with a desktop or native application. In a SPA, only one main document, or page, is loaded into the browser. After the initial document load, other resources, such as scripts, stylesheets, data, and assets such as images, are loaded asynchronously, but the initially requested document does not change. In other words, throughout the lifecycle of the application, the content of the URL in front of the hash mark (#) typically does not change. As a result, the browser never requests any subsequent "page". The history API in modern browsers does allow changes to the URL before the hash without an entire page request, but most JavaScript frameworks and routing libraries use the portion after the hash exclusively for front-end routing. For simplicity, we'll operate using this clear separation of server-side and client-side routes.

http://example.com/app	#/primary View
SERVER ROUTE	CLIENT ROUTE

The portion of the URL before the hash mark is a server-bound (browser request) route.
The portion after the hash mark is the client-side route controlled by the SPA.

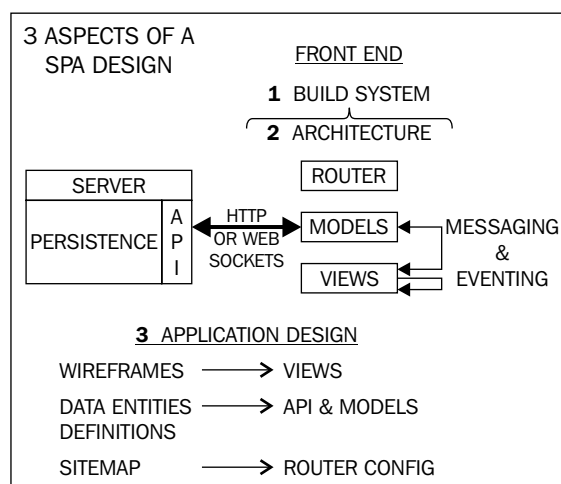
Navigation is handled on the front end through a router. The front-end router reacts to URL changes after the hash. This portion of the URL was historically used for contextual linking between headings within a web page via anchor tags referencing the hashed URLs, sometimes called jump links. When the application changes the contents of the URL after the hash, a view system morphs the DOM by composing and rendering different high-level views. The mapping between these URL changes and views is done via the router configuration. Any change before the hash belongs to the server route and would result in a new browser document request. By definition, a SPA exclusively drives navigation through front-end routes only.

At the highest level, a SPA still consists of a front end and back end. Historically, the back end performed most of the application logic and the front end merely handled the concern of presenting an interactive facade to the user in order to display data and gather input. With a SPA, the back end is typically just a persistence mechanism and an API whose design could still be largely defined by front-end modeling. In a SPA, the front end now handles all significant routing, DOM construction and composition, and view modeling. View models are projections or subsets of one or more of the models in the application or problem domain.

Three aspects of a SPA design

A SPA can be conceived in three parts: a **build system**, front-end **architecture components**, and application design. Application design is called out specifically because it is instrumental in defining the server API as well as moving forward with the actual implementation. The artifacts that result from the application design process bridge the gap between the various needed parts, and the interesting use of those parts in an actual implementation. We'll begin by identifying the various parts and end by exploring a few design procedures. This will carry us nicely into the next chapter, where we will begin to implement an actual application.

The following diagram illustrates the three major aspects of a SPA design and their relationships:



In the diagram, you can see a clear association between the core front-end models and the server API, as well as the somewhat nebulous relationship between the application design and everything else. Stay with me on the application design aspect. It will generate some meaningful artifacts and make both the relationships in the anatomy diagram and the implementation of our application clearer.

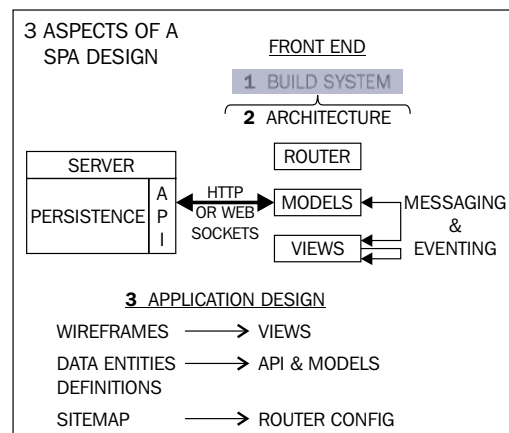
As we explore these aspects and their details, we'll also address how there are lots of options for each detail, offer some thoughts on trade-offs for each option, and ultimately describe a choice with some justification. There are so many ways to address each problem that, sooner or later, you just need to choose and move forward. How you make such choices should take into account the problem you are trying to solve, the other components you have already chosen, and obvious advantages each choice offers your particular application. Try to avoid naïve comparison analysis that is rampant on the Web (X versus Y) and focus on what each option actually **does**, which qualities of it **appeal to you personally** and how those qualities **apply to the problem at hand**. Popularity of an option should be considered in two regards:

- The ease of finding information on the option (documentation quality, help forums, thoughtful examples in blogs, etc)
- General acceptance of that option within the community of practitioners using the components you've already chosen

In other words, if you've already chosen React for its strengths at component composition and rendering performance, then you should probably lean a bit into other tools that are trending in the React community in order to have better support.

Build systems

Building a SPA (as opposed to an older style web application) means that many application concerns have migrated to the front end, making the client-side responsibilities necessarily more complex. Also, the nature of modern web development lends itself to an endless buffet of tools aimed at making HTML, JavaScript, and CSS more manageable.



The front-end build system

Here are some of the types of tools for managing code complexity:

- Module and code packaging/delivery systems
- CSS preprocessors
- Next-generation JavaScript syntax (ES6 and beyond)
- Templates and other syntax processing (needed for JSX)

In terms of managing complexity, the first item in this list is paramount. Being able to organize portions of code, inject it into other portions, and efficiently deliver it to the browser is essential for making web applications.

Choosing a build system

The build system will tie all of the builders and preprocessors together into a manageable pipeline of transformations for development and, ultimately, deployment.

For small experiments, using the in-browser JSX compiler is great. You can also use the ES6 (ECMAScript 6) Harmony syntax if you use `<script type="text/jsx;harmony=true">`. This feature was added in React 0.11.

For serious work, though, it is recommended that you use a build system for JSX compilation, CSS preprocessing (such as LESS and SASS), as well as for bundling your application into payloads of an efficiently small number of files. There are many solutions for this, and it seems that there's a new one every couple of months. An early and enduring favorite is **Grunt**, in which build tasks are specified in code but resemble a large configuration file. When streams caught on in the node community, a stream pipelining build task system was created called **Gulp**.

Typically, a module system is paired with a build system in order to manage dependencies. **CommonJS** and **Asynchronous Module Definition (AMD)** are the prominent modularity strategies. AMD is valued for its natural disposition toward asynchronous loading, and CommonJS is valued for its familiar looking syntax and use in NodeJS.

Alongside build tools are scaffolding tools that whip your filesystem structure into a ready-to-code state. These tools also download requisite libraries and perform configuration by asking you to answer a series of yes or no questions. One very popular scaffolding tool is **Yeoman**, which handles all these scaffolding tasks. While Gulp is used for general task running and sequencing, a Gulp-related answer to the scaffolding aspect exists called **Slush**. As previously stated, these types of tools (and JS frameworks for that matter) are released on a continual basis. Developing an aptitude for identifying the trade-offs and merits of new tools without getting overwhelmed or too complacent with a particular set is an essential skill. Shiny new ones are always on the horizon but, whichever you choose, you should spend enough time with them to complete a nontrivial project in order to know where their true power lies. Doing this will hone your technical judgment and help you to develop a personal style.

Within this swirl of options, there's a very versatile tool that the React JS community seems to have gravitated toward called **Webpack**. Webpack is substantial. It has a pluggable interface and a project build specification mechanism, which can intelligently split your code into chunks for efficient delivery to the browser. It uses a streaming pipeline style similar to Gulp but, unlike Gulp, which uses standard NodeJS imperative streaming code, Webpack configuration defines a build pipeline using a more succinct syntax. It also carries the burden of code chunk dependency calculation and dynamic loading. Further, it supports both CommonJS and AMD style dependency syntax. Chunk specification in Webpack is defined by syntax that closely resembles AMD-style **dependency injection (DI)** syntax. Webpack also has a server component which is used to dynamically generate necessary code chunks and hot load them into the browser environment during development. It's quite a full-featured and impressive tool.

In summary, a complete build system and pipeline includes a scaffolding aspect to kick-start your project organization, the means to optimize a dependency tree, a modularization component to express the dependency hierarchy, any precompilation processes (CSS preprocessors and ES6 transpilers), code minification steps for packaging, and active reload mechanisms for development. Take the time to try the latest tools, but for pragmatic purposes, the path of least resistance is to use what your community is using. For instance, you may find it easier to get help if you use Webpack while exploring React. So, that's what we'll use going forward. We are going to leave some of the details of Webpack aside for personal exploration, but we will examine an interesting project configuration that uses many features of Webpack for our example project.

Module systems

There are two dominant module system styles in the JavaScript community: AMD and CommonJS.

CommonJS

CommonJS is traditional in that a single assignment statement is used to specify a portion of code to be imported into the target code. The statement looks like this: `var someModule = require('path/to/module');` A key point to this structure is that, for the module referenced by the variable to be immediately used in a following statement, the module assignment must block until the module code can be loaded. So, it had better be loaded or your JS code would have to pause! Though, a more advanced parser, which understands both JavaScript and CommonJS syntax, can look ahead to require statements and build an intelligent dependency graph for preloading and concatenation. CommonJS syntax is the natively supported module system in NodeJS. It is also the style used in the JavaScript core language going forward from ES6.



Here's a quick note about CommonJS with Webpack. Typically, the way you call for a CommonJS module is by assigning the result of a `require` invocation to a `var` statement. This means that the `var` statement could potentially pollute the function scope in which it was defined, as `var` statements do. Webpack, though, wraps the module within another function scope. In fact, it rewrites your `require` invocation to a Webpack one that can intelligently load application chunks. So, your `var` statements within a Webpack module become effectively isolated. This is a nice extra isolation that brings CommonJS within Webpack closer to the isolation and DI style of AMD.

AMD

AMD (Asynchronous Module Definition) is a specification in which module code is encapsulated into the invocation of a function named `define`. The `define` signature is comprised of the following parameters:

- First, an optional module name (the module is referred to by filename if this parameter is omitted)
- Next, an optional list (array) of dependencies by name or filesystem location
- Finally, and most importantly, the definition of the module itself as an **IIFE (immediately invoking function expression)**

The IIFE (also the module definition function) signature contains positional parameters that directly map in arity (number of parameters) and parameter order to the dependencies specified in the previous define parameter, the dependency array. This allows assignment and symbol renaming upon invocation of the target module, our IIFE. Another way to say this is that the dependencies you specify in the dependency array parameter will directly be mapped to the function parameters of the module being defined, allowing you to name dependencies whatever you want within the module. This syntax lends itself to a more natural-feeling asynchronous loading pattern for JavaScript. It is considered more natural for asynchronous loading for two reasons.

First, it is a common pattern in JavaScript to make the final parameter to a function signature a callback for continuation of execution. It is also generally expected that the callee will invoke the function supplied in the final parameter asynchronously. This is the callback pattern for asynchronous JS.

Second, the define invocation itself establishes a registry of modules and their dependencies. This lends an AMD system to all sorts of optimization opportunities where modules can be loaded and executed more efficiently. For instance, as define invocations occur, the AMD system could begin loading the modules in turn. Logical branches within code can call define for subsequent on-demand loading. Here's another example of a possible optimization: the AMD system could lazily load modules only when required and fetch them from the server. A final reason AMD may be considered "natural" is that the syntax not only resembles a historically familiar pattern used in JavaScript, but also closely mirrors other DI systems.

Our module choice

I personally bought into the syntactical style and more "natural" asynchronicity of AMD during the duration of the great module debate. However, in the following chapters, we'll use CommonJS for pragmatic reasons. ES6 uses this style and has a lot of other great features that we want to use. The moment of judgment on this debate has largely passed as NodeJS and JavaScript natively support CommonJS going forward. Finally, there are a lot of tools, including Webpack, that take the burden of asynchronous loading and packaging of code for the wire off of the developer, even when using the CommonJS syntax. As such, further mentions of JavaScript modules will loosely imply the ES6 CommonJS style. It is the path of least resistance.

CSS preprocessors

CSS preprocessors are a fantastic way to organize CSS. They help to simplify complicated selectors, handle vendor prefixing, establish variables for reuse in layout measurements and color calculations, and even roll up image references into the stylesheet, eliminating extra HTTP requests. If you are a serious web developer, you need to get comfortable with CSS preprocessors.

The two prominent options are **LESS** and **SASS**. While many CSS gurus seem to advocate SASS, we are going to use LESS. This is not a stance on preprocessors in general, but this book is about React and we are already using Node for tooling because of Webpack. LESS syntax, while perhaps a bit less powerful, more closely resembles actual CSS and runs a bit more easily just with JavaScript (specifically node), while SASS requires Ruby or a Node bridge to native bindings. So, the barrier to entry, both cognitively and environmentally, is lower overall for our project using LESS.

Compiling the modern JS syntax and JSX templates

ECMAScript 6 (ES6), a long overdue update to the specification upon which JavaScript is based, has many useful features and additional convenient syntax. It heralded a future of frequent updates to the language, so you should get familiar with what it has to offer.

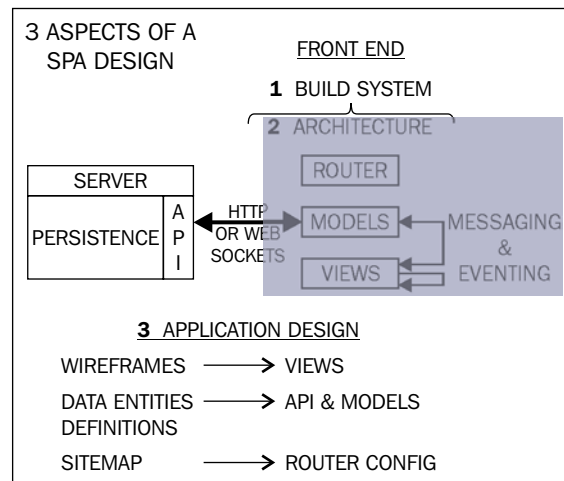
If we are going to inhabit React-land, we should use JSX. It's a very convenient way to compose React components, albeit a leaky abstraction of HTML (really XML). It looks like HTML, but it's really just shorthand for JS. Remembering this mantra at all times can keep you out of trouble: JSX is a dialect of JavaScript, not HTML.

In regard to language features, browsers don't really support a particular version of JavaScript. Adoption of features is more fluid. Browser updates roll out chunks or bursts of the latest features. Having many moving targets is annoying for a developer, but transpilers come to the rescue! It turns out, we can just write using our favorite features. Luckily, the hardworking people of the web development community have managed to make implementations of new features in runtime environments that don't explicitly support them via **transcompilation** (aka source-to-source compilers) and **polyfills** (runtime supported code that fills in missing features). A couple of past popular transpilers were Google Traceur compiler and 6to5. 6to5 was renamed to **Babel** because the maintainers wanted to be more forward looking than just compiling ES6 to ES5. Babel will support ES6 and beyond as new features are ratified. It's also generally easy to use and includes JSX support! One wonderful tool lets us use the latest and greatest of JavaScript and do JSX compilation.

Front-end architecture components

A user interface is all about views. The primary views tend to map directly to user goals. For instance, in an email application, you read and send emails. So, your primary views could be "inbox" and "create email". In most applications, defining the primary views typically consists of taking the system nouns (document, email, order, user, post, etc) and making a view for each associated verb, usually "find" and "create/edit".

The following diagram highlights the front-end architecture aspect of an SPA design:



Front-end architecture components

In a React app, your front-end architecture components are:

- The router
- Models
- Views (Layout and CSS)
- View models (compositions of system models)
- View controllers (logic and rules in the view)
- Messaging and eventing mechanisms

The front-end router

As mentioned at the beginning of this chapter, the front-end router is a piece of software that reacts to changes in the URL after the hash mark. The result of the routing is a transition between major views. This changes the user's context or workflow. An obvious choice for a front-end router in a React application is **React Router**. It handles all of the standard functions needed from a router. Router functions include the ability to map routes to views or compositions of views as well as to break apart the hashed part of the route into positional parameters and query parameters (everything after the "?" in the URL). The query parameters are packaged neatly into props for the components targeted for render by the router.

Front-end models

In the front end, models are often correlated one-to-one with the data entities of the application at large. Early on, there wasn't a particular solution for this concern targeted for React applications, although Backbone models were probably most often used for this purpose. Now there are several solutions specifically targeted at React applications.

Facebook, the creators and general maintainers of React, have devised a modeling pattern they call **Flux**. Flux embraces a one-way data flow model and works by supplying three types of entities: **stores**, **actions**, and the **dispatcher**. Actions are just verbs in the system, commands. When an action occurs, the dispatcher routes it to interested listeners, the stores. Stores are essentially bags of data that can be queried via actions and emit store change events which views respond to by updating internal state and re-rendering if needed.

Another model solution targeted at React with growing popularity is **Reflux**. Reflux is very similar to Flux but omits the dispatcher. Reflux was crafted with the notion that the dispatcher isn't particularly useful and is just extra work. Instead, in Reflux, actions are commands but are merely named commands that can be published or subscribed to directly. So, stores listen to actions directly instead of being mapped through the dispatcher and, just as in the Flux architecture, generate store events that can be listened to by views. Because of this simplification in Reflux, we'll use it in our prototype application in the application building chapters. Reflux also supplies some useful mixins for our views listening to stores. These mixins will automatically tie the payload of the store event emission to a state variable in our views and call `setState` whenever there is a store change.

Views, view models, and view controllers

These entities are the ones that will be taken care of by React. The view is an instance of a React component that was specified via `React.createClass`. Logic within that component definition is effectively the view controller. The internal component state or a portion of that state could be considered the view models in a React application. View models in our blog app will be the portions of internal state set by our Reflux stores via the convenience of the Reflux mixins. View models (component state) in the app may also be a composition of more than one store, if necessary.

Messaging and eventing

With everything in our app componentized, we'll need a way to coordinate reactions to changing data, user interactions, and possibly scheduled events. In React, a parent component can communicate to a child component via props. The child can react to prop changes using the `componentWillReceiveProps` lifecycle method. However, our data stores aren't part of this component hierarchy, and it's nice to have a general communication bus for communications besides those within the view hierarchy. For this, we can use actions in Reflux. Actions are merely publish-subscribe eventing mechanisms. In addition, actions can be specified as asynchronous and supply a promise interface. This is sufficient for just about any SPA. As a bonus, we can go to a single place, our action definitions object, to see a list of all the verbs in the system and whether or not they are async. Nice!

Other utility needs

At this point, we have everything related to view management and front-end communication covered but an important piece is still missing: the bold double-headed arrow in the diagram pointing between front-end models (now stores) and the server API. For this, we really just need a simple AJAX library. This is usually baked into frameworks and other toolkit libraries such as jQuery and Backbone, but such libraries also come with a lot of things that we don't really need. Superagent is a great example of a full-featured AJAX library. Its clean, chainable, interface handles different REST verbs and HTTP headers and supplies a promise. Consider React to be in the spirit of the Unix philosophy, which espouses the virtue of "doing only one thing, and doing it well". React does views, only views, and does them well. Likewise Superagent does HTTP requests, only HTTP requests, and it does them well.

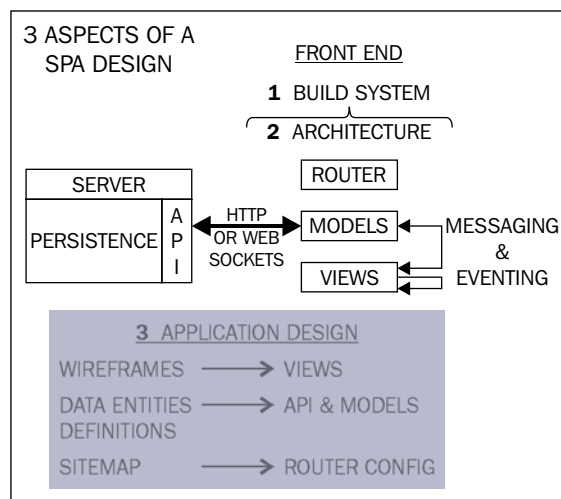
The last item of note in service of our blog application is some sort of rich text editor. After all, editing plain unformatted text would make a pure, albeit rather dull blogging experience. There are lots of options here. Among the ones I considered were Hallo and Quill. Both were simpler than TinyMCE or Aloha Editor, which I also evaluated. After some thought, Quill was chosen because unlike Hallo, which operates as a jQuery UI plugin, Quill does not require any additional libraries to operate.

The application design

First, a disclaimer: I am not a trained designer, but like most, I have some assumptions and notions in the department. Here are some tips that don't just apply to visual and interaction design, but also to software design in general. Think about the fewest number of things that identify what you are building. For starters, eschew all the features swirling in your brain. To begin a plan, focus is required. It will pay off when we start coding. Try to sum up the application using one word. For our impending blog app "posts" is an apt choice. For a lunar lander game "physics" comes to mind. For an adventure game, maybe that one word is "story". If you understand the number one thing that your app is supposed to get right and let it guide you throughout design and development, then it will at least do that one thing well.

In the next chapter, the following design tasks will be repeated for the blog application, which will be built over the course of chapters 5 through 9. In that application, the focus is on posts and people (bloggers). As such, the design should initially address text. A lot of color and other flourishes are probably okay, but in the spirit of simplicity, use of space and typography will be our primary concerns. So, we'll focus on placement, workflow (find posts, read posts, and follow an interesting author), and making things clear and easy to read.

The following diagram highlights the application design aspect of an SPA design:



The application design aspect.

The diagram shows how a few simple design procedures translate directly into implementations in our front-end application architecture. With a bit of upfront planning, we can save a lot of time. You don't have to have an eye for design to follow a process which will make ongoing code decisions a lot easier. In the next chapter, before we actually start writing the code, we'll make a few lists and diagrams. Right now, as a preview, let's explore these design procedures, what the output of each looks like, and how it will serve us when we actually begin to write code.

The following design procedures use an email application as an example. We'll repeat these procedures for the blog application in the next chapter.

Creating wireframes







This is a screen designing process. Sometimes it's good to just start sketching. Start drawing an interface that captures the placement of items and core user interactions in the application. There are a lot of tools that can be used for wireframing, but I prefer to dive in with a pen and paper. Color, fonts, and other final touches aren't necessary here. The aim is twofold:





- **Information hierarchy:** Spatial relationships and placement of data on screen
- **User interactions:** Navigation and user workflow (menus, links, and so on)

For the information hierarchy, think about the user goal in each main view, and how they would use each main component in sequence within each workflow. This will help you choose the correct size and placement in order to guide their eyes through their tasks linearly.

For interactions, stick with precedent. A chief principle of usability is "don't make me think". It turns out that *Don't Make Me Think* is also a very concise book on usability by Steve Krug, and is recommended reading for anyone making interfaces on the Web. Try to put primary and secondary navigation in a typical area. If you have primary action buttons, such as confirm buttons, give them a bit more visual weight and put them near the right edge of the screen (closer to a thumb on a mobile device).

The following diagram shows a wireframe for a familiar type of web application, an email web app:

FIND AND READ EMAIL				
 NEW EMAIL		<input type="text" value="Search"/> 		LOGO
FOLDERS		< FOLDER NAME OR SEARCH TERMS >		
▼ Inbox _____ _____ ► Sent Junk	 FROM NAME unread subject	SUBJECT _____ _____ _____	FROM DATE/TIME _____ _____ _____	
	 from name read subject			
	...			
CONTACTS				
 _____  _____			➔ FORWARD ↩ REPLY	
< CURRENT DATE/TIME > ☀ 76°				

CREATE NEW EMAIL				
 NEW EMAIL		<input type="text" value="Search"/> 		LOGO
FOLDERS		NEW MESSAGE		
▼ Inbox _____ _____ ► Sent Junk	Subject <input type="text"/>			
	To <input type="text"/>			
	Message _____ _____ _____			
CONTACTS				
 _____  _____			<input type="button" value="SAVE"/> <input type="button" value="SEND"/>	
8:38 PM 6/14/2015 ☀ 76°				

In the email application wireframe example, you can see how using a simple format such as a pen and paper can establish focus on component placement, relative size, and user workflow without the distractions of color, font choice, specific graphics, and the like. For your applications, you'll want to make a handful of these to capture the primary views that service main user goals. For email, this could be as few as two: "find mail" and "create new mail".

Examining the wireframes also forecasts reusable components. In the example image these would be: the header, contacts control, icon button, email list, and so on.

Main data entities and the API

Make a list of the main data entities in your app. This list should be quite short. One of the entities is probably the main focus of the app. For example, in an email app, the most important data item by far is the email! Alongside that, you'd probably have contacts and folders. If this list starts getting long, perhaps you are getting a little too detailed. To get an initial list, think about what objects a user would expect to see on a main page or screen of your app if you were to ask them conversationally.

Next, you can define an API for data persistence by writing out each entity and the standard list of verbs for data augmentation: **C.R.U.D.** (**create**, **read**, **update**, and **delete**). In the case of a RESTful API for a web application, the verbs would be **POST**, **GET**, **PUT**, and **DELETE**, respectively. Here's an example for an email application:

Entity name	Data members	Operations
Mails	<ul style="list-style-type: none">• mail uid• folder uid• from email• from name• to• subject• body• new	<ul style="list-style-type: none">• create• read• delete
Folders	<ul style="list-style-type: none">• folder uid• folder name	<ul style="list-style-type: none">• create• read• update• delete
Contacts	<ul style="list-style-type: none">• contact uid• contact name• contact email• contact photo	<ul style="list-style-type: none">• create• read• update• delete

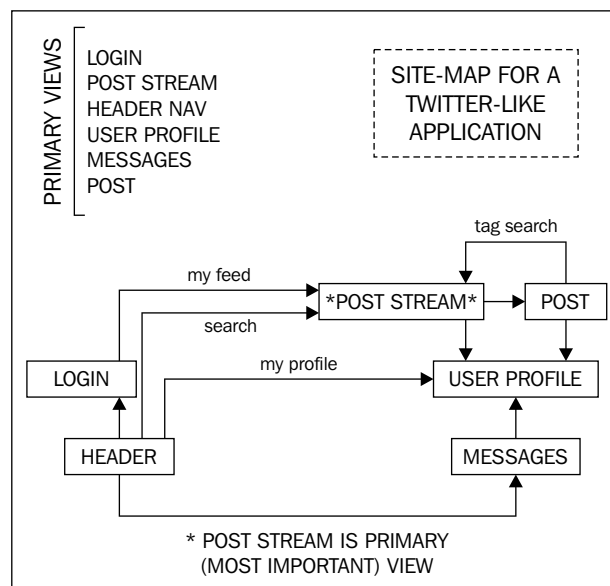
Most main entities will have the full complement of data operations. Often, they will have a read operation to get one instance and one or more read operations to get a collection of instances. Also, expect every main data entity to have a unique identifier (uid).

Main views, site map, and routes

Identifying the main views in the app is simple. They are usually just the user goals in regard to the main subject of the app. If you decide to wireframe most of your app, you've probably already identified all of these views. In the email app, two obvious main views could be a list of emails (inbox) and "create/edit email". You'll probably want to add the primary navigation view (often a header) to this list even though it's likely embedded in many or all of the main views. It will help when making the sitemap.

Once you have the main views figured out, you can make a sitemap. Sitemap is a bit of a misnomer here. After all, this isn't a website; it's an application! Still, this term is somewhat apt since it's also used as the name of a real artifact that can be used to index your application for **Search Engine Optimization (SEO)**.

Make a box for each main view and draw lines for user navigation between them. Don't be surprised if your sitemap looks like a spider web instead of a tree. This is the difference between a sitemap for an application, which deals with views, and one for a website, which deals with a hierarchy of pages. The following is an example of a sitemap for a Twitter-like application:



A wireframe for a twitter-like application

Finally, we've reached the goal of this exercise that will allow us to begin coding in earnest. Armed with the main views and the sitemap, you can now easily produce your router configuration. Each main view will be able to be linked and bookmarked. This final step will be especially useful when we design the blog application in the next chapter. The sitemap will allow us to scaffold our application workflows using React Router before filling in the details for the view logic and data management.

Summary

After reviewing a list of application aspects and some of the tools, it's apparent that programming applications for the web browser has become quite complex! Of course, if your application is very small and limited to only a single workflow or two, you may like to omit some of these tools. Although, it's best to become comfortable with all of them. They are the tools of your trade. Indeed, the reason for reaching for a tool like React is because it reduces complexity and affords power and performance for very complex interactive applications such as Facebook, the impetus for React's existence.

Finally, it's important to understand that just knowing a list of problems and possible tools isn't enough to effectively compose them into something complete. Some forethought and a design procedure can greatly improve both the coding process and the end result. A planning process is as important a part of an application developer's repertoire as their ability to decompose the parts of the app and choose software.

In the next chapter, we'll reiterate some of the tool choices here and repeat the design procedures for a multi-user blog application. The artifacts produced by this process will then be used to guide the construction of the app.