O'REILLY®

SIGN IN TRY NOW

See everything available through O'Reilly Online Learning a

Search

Chapter 1. Why Isomorphic JavaScript?

Jason Strimpel and Maxime Najim

In 2010, Twitter released a new, rearchitected version of its site. This "#NewTwitter" pushed the UI rendering and logic to the JavaScript running in the user's browser. For its time, this architecture was groundbreaking. However, within two years, Twitter had released a re-rearchitected version of its site that moved the rendering back to the server. This allowed Twitter to drop the initial page load times to one-fifth of what they were previously. Twitter's move back to server-side rendering caused quite a stir in the JavaScript community. What its developers and many others soon realized was that client-side rendering has a very noticeable impact on performance.

NOTE

The biggest weakness in building client-side web apps is the expensive initial download of large JavaScript files. TCP (Transmission Control Protocol), the prevailing transport of the Internet, has a congestion control mechanism called *slow start*, which means data is sent in an incrementally growing number of segments. Ilya Grigorik, in his book *High Performance Browser Networking* (O'Reilly), explains how it takes "four roundtrips... and hundreds of milliseconds of latency, to reach 64 KB of throughput between the client and server." Clearly, the first few KB of data sent to the user are essential to a great user experience and page responsiveness.

The rise of client-side JavaScript applications that consist of no markup other than a `<script>` tag and an empty `<body>` has created a broken Web of slow initial page loads, hashbang (`#!`) URL hacks (more on that later), and poor crawlability for search engines. Isomorphic JavaScript is about fixing this brokenness by consolidating the code base that runs on the client and the server. It's about providing the best from two different architectures and creating applications that are easier to maintain and provide better user experiences.

Defining Isomorphic JavaScript

Isomorphic JavaScript applications are simply applications that share the same JavaScript code between the browser client and the web application server. Such applications are isomorphic in the sense that they take on equal (*iso*) form or shape (*morphosis*) regardless of which environment they are running on, be it the client or the server. Isomorphic JavaScript is the next evolutionary step in the advancement of JavaScript. But advancements in software development often seem like a pendulum, accelerating toward an equilibrium position but always oscillating,

swinging back and forth. If you've done software development for some time, you've likely seen design approaches come and go and come back again. It seems in some cases we're never able to find the right balance, a harmonious equilibrium between two opposite approaches.

This is most true with approaches to web application in the last two decades. We've seen the Web evolve from its humble roots of blue hyperlink text on a static page to rich user experiences that resemble full-blown native applications. This was made possible by a major swing in the web client–server model, moving rapidly from a fat-server, thin-client approach to a thin-server, fat-client approach. But this shift in approaches has created plenty of issues that we will discuss in greater detail later in this chapter. For now, suffice it to say there is a need for a harmonious equilibrium of a shared fat-client, fat-server approach. But in order to truly understand the significance of this equilibrium, we must take a step back and look at how web applications have evolved over the last few decades.

Evaluating Other Web Application Architecture Solutions

In order to understand why isomorphic JavaScript solutions came to be, we must first understand the climate from which the solutions arose. The first step is identifying the primary use case.

> **NOTE**
>
> Chapter 2 introduces two different types of isomorphic JavaScript application and examines their architectures. The primary type of isomorphic JavaScript that will be explored by this book is the ecommerce web application.

A Climate for Change

The creation of the World Wide Web is attributed to Tim Berners Lee, who, while working for a nuclear research company on a project known as "Enquire" experimented with the concept of hyperlinks. In 1989, Tim applied the concept of hyperlinks and put a proposal together for a centralized database that contained links to other documents. Over the course of time, this database has morphed into something much larger and has had a huge impact on our daily lives (e.g., through social media) and business (ecommerce). We are all teenagers stuck in a virtual mall. The variety of content and shopping options empowers us to make informed decisions and purchases. Businesses realize the plethora of choices we have as consumers, and are greatly concerned with ensuring that we can find and view their content and products, with the ultimate goal of achieving conversions (buying stuff)—so much so that there are search engine optimization (SEO) experts whose only job is to make content and products appear higher in search results. However, that is not where the battle for conversions ends. Once consumers can find the products, the pages must load quickly and be responsive to user interactions, or else the businesses might lose the consumers to competitors. This is where we, engineers, enter the picture, and we have our own set of concerns in addition to the business's concerns.

Engineering Concerns

As engineers, we have a number of concerns, but for the most part these concerns fall into the main categories of maintainability and efficiency. That is not to say that we do not consider business concerns when weighing technical decisions. As a matter of fact, good engineers do exactly the opposite: they find the optimal engineering solution by contemplating the short- and long-term pros and cons of each possibility within the context of the business problem at hand.

Available Architectures

Taking into account the primary business use case, an ecommerce application, we are going to examine a couple of different architectures

within the context of history. Before we take a look at the architectures, we should first identify some key acceptance criteria, so we can fairly evaluate the different architectures. In order of importance:

1. The application should be able to be indexed by search engines.
2. The application's first page load should be optimized—i.e., the *critical rendering path* should be part of the initial response.
3. The application should be responsive to user interactions (e.g., optimized page transitions).

> **NOTE**
>
> The critical rendering path is the content that is related to the primary action a user wants to take on the page. In the case of an ecommerce application it would be a product description. In the case of a news site it would be an article's content.

These business criteria will also be weighed against the primary engineering concerns, maintainability and efficiency, throughout the evaluation process.

Classic web application

As mentioned in the previous section, the Web was designed and created to share information. Since the premise of the World Wide Web was the work done for the Enquire project, it is no surprise that when the Web first started, web pages were simply multipage text documents that linked to other text documents. In the early 1990s, most of the Web was rendered as complete HTML pages. The mechanisms that supported (and continue to support) it are HTML, URIs, and HTTP. HTML (Hypertext Markup Language) is the specification for the markup that is translated into the

document object model by browsers when the markup is parsed. The URI (uniform resource identifier) is the name that identifies a resource; i.e., the name of the server that should respond to a request. HTTP (Hypertext Transfer Protocol) is the transport protocol that connects everything together. These three mechanisms power the Internet and shaped the architecture of the classic web application.

A classic web application is one in which all the markup—or, at a minimum, the critical rendering path markup—is rendered by the server using a server-side language such as PHP, Ruby, Java, and so on (Figure 1-1). Then JavaScript is initialized when the browser parses the document, enriching the user experience.

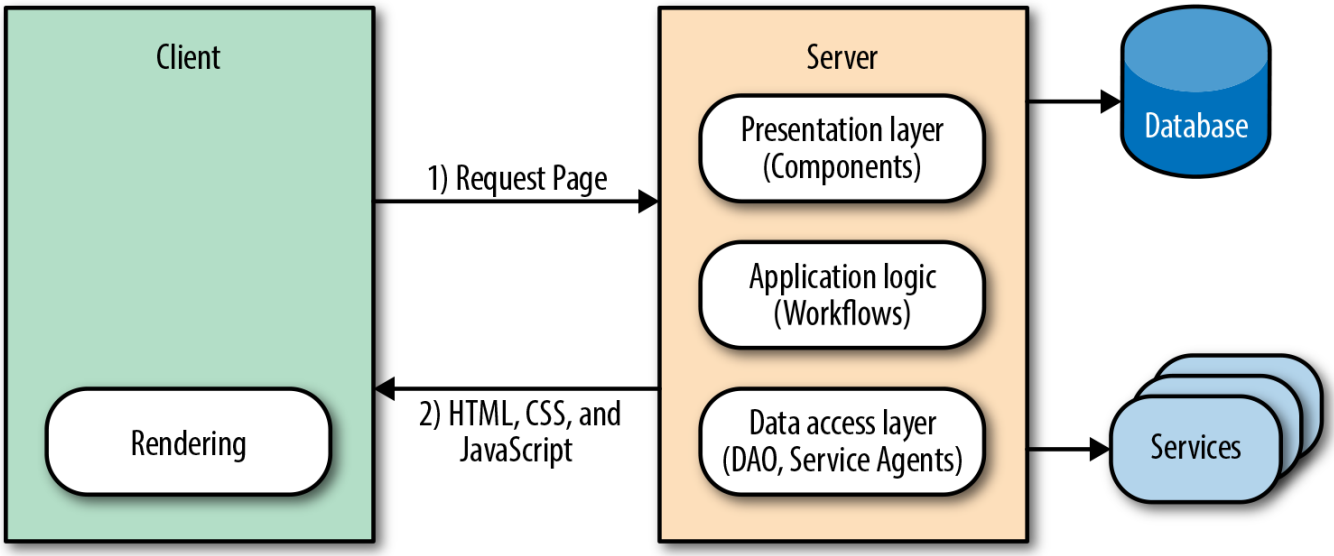

Figure 1-1. Classic web application flow

In a nutshell, that is the classic web application architecture. Let's see how it stacks up against our acceptance criteria and engineering concerns.

Firstly, it is easily indexed by search engines because all of the content is available when the crawlers traverse the application, so consumers can find the application's content. Secondly, the page load is optimized because the critical rendering path markup is rendered by the server, which improves the perceived rendering speed, so users are more likely not to bounce from the application. However, two out of three is as good as it gets for the classic web application.

NOTE

What do we mean by "perceived" rendering speed? In *High Performance Browser Networking*, Grigorik explains it this way as: "Time is measured objectively but perceived subjectively, and experiences can be engineered to improve perceived performance."

In the classic web application, navigation and transfer of data work as the Web was originally designed. The browser requests, receives, and parses a full document response when a user navigates to a new page or submits form data, even if only some of the page information has changed. This is extremely effective at meeting the first two criteria, but the setup and teardown of this full-page lifecycle are extremely costly, so it is a suboptimal solution in terms of responsiveness. Since we are privileged enough to live in the time of Ajax, we already know that there is a more efficient method than a full page reload—but it comes at a cost, which we will explore in the next section. However, before we transition to the next section we should take a look at Ajax within the context of the classic web application architecture.

The Ajax era

The `XMLHttpRequest` object is the spark that ignited the web platform fire. However, its integration into classic web applications has been less impressive. This was not due to the design or technology itself, but rather to the inexperience of those who integrated the technology into classic web applications. In most cases they were designers who began to specialize in the view layer. I myself was an administrative assistant turned designer and developer. I was abysmal at both. Needless to say, I wreaked havoc on my share of applications over the years (but I see this as my contribution to the evolution of a platform!). Unfortunately, all the applications I touched and all the other applications that those of us without the proper training and guidance touched suffered during this

evolutionary period. The applications suffered because processes were duplicated and concerns were muddled. A good example that highlights these issues is a related products carousel (Figure 1-2).

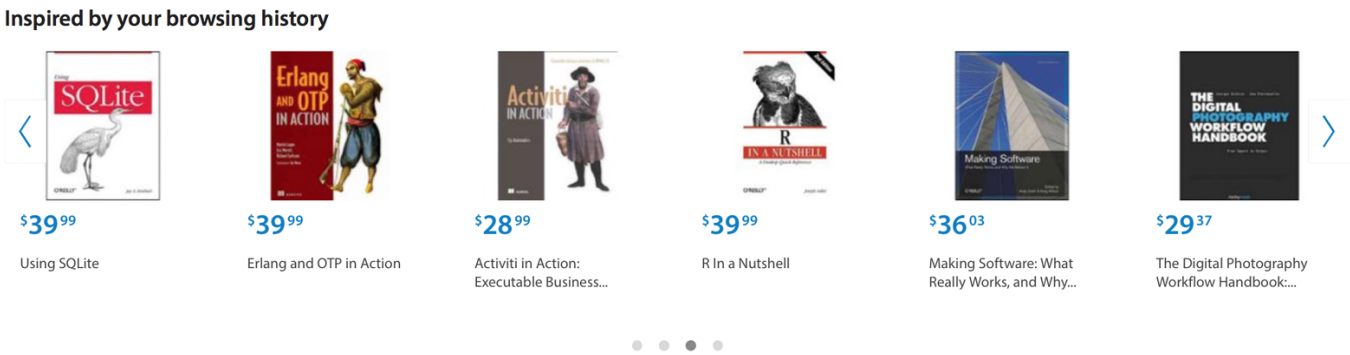

Figure 1-2. Example of a product carousel

A (related) products carousel paginates through products. Sometimes all the products are preloaded, and in other cases there are too many to preload. In those cases a network request is made to paginate to the next set of products. Refreshing the entire page is extremely inefficient, so the typical solution is to use Ajax to fetch the product page sets when paginating. The next optimization would be to only get the data required to render the page set, which would require duplicating templates, models, assets, and rendering on the client (Figure 1-3). This also necessitates more unit tests. This is a very simple example, but if you take the concept and extrapolate it over a large application, it makes the application difficult to follow and maintain—one cannot easily derive how an application ended up in a given state. Additionally, the duplication is a waste of resources and it opens up an application to the possibility of bugs being introduced across two UI code bases when a feature is added or modified.

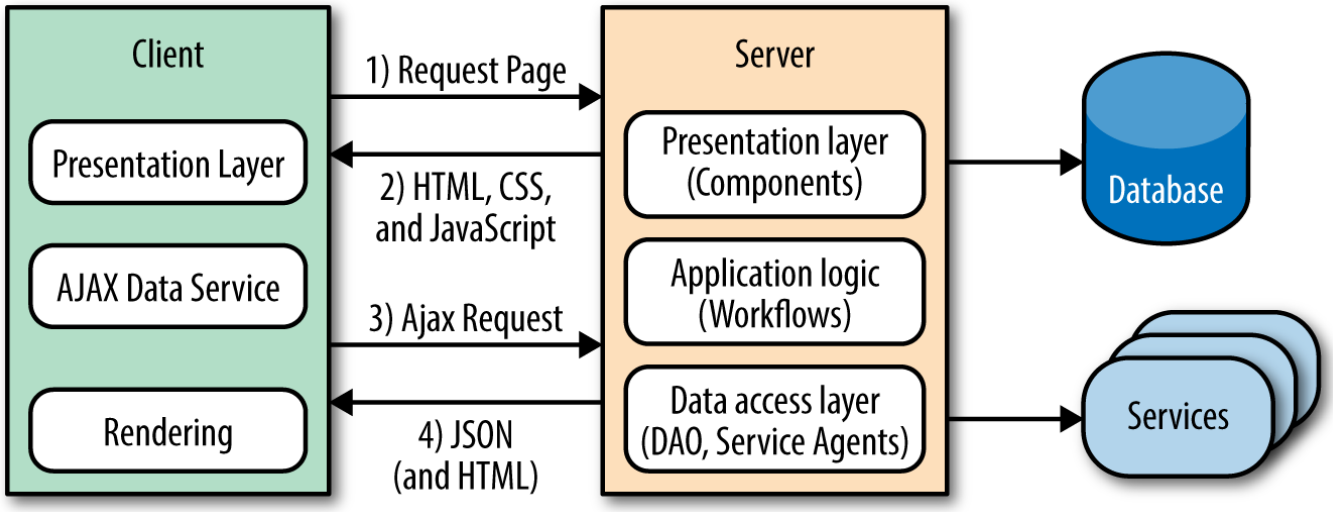

Figure 1-3. Classic web application with Ajax flow

This division and replication of the UI/View layer, enabled by Ajax and coupled with the best of intentions, is what turned seemingly well-constructed applications into brittle, regression-prone piles of rubble and frustrated numerous engineers. Fortunately, frustrated engineers are usually the most innovative. It was this frustration-fueled innovation, combined with solid engineering skills, that led us to the next application architecture.

Single-page web application

Everything moves in cycles. When the Web began it was a thin client, and likely the inspiration for the Sun Microsystems NetWork Terminal (NeWT). But by 2011 web applications had started to eschew the thin client model, and transition to a fat client model like their operating system counterparts had done long ago. The monolith had surfaced. It was the dawn of the single-page application (SPA) architecture.

The SPA eliminates the issues that plague classic web applications by shifting the responsibility of rendering entirely to the client. This model separates application logic from data retrieval, consolidates UI code to a single language and runtime, and significantly reduces the impact on the servers (Figure 1-4).

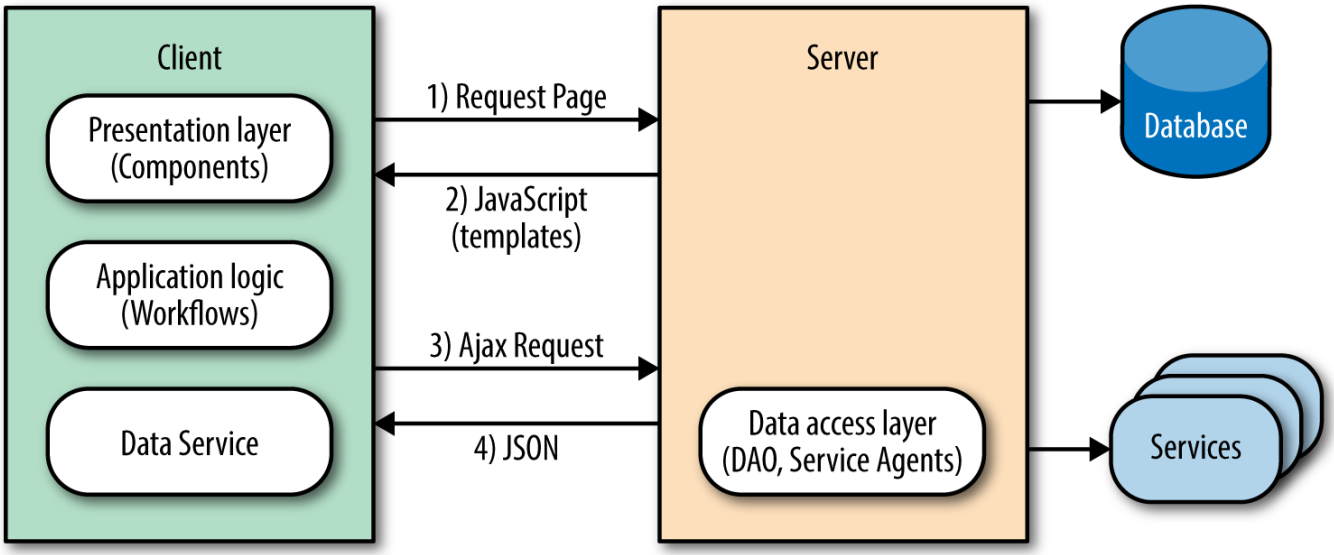

Figure 1-4. Single-page application flow

It accomplishes this reduction because the server sends a payload of assets, JavaScript, and templates to the client. From there, the client takes over only fetching the data it needs to render pages/views. This behavior significantly improves the rendering of pages because it does not require the overhead of fetching and parsing an entire document when a user requests a new page or submits data. In addition to the performance gains, this model also solves the engineering concerns that Ajax introduced to the classic web application.

Going back to the product carousel example, the first page of the (related) products carousel was rendered by the application server. Upon pagination, subsequent requests were then rendered by the client. The blurring of the lines of responsibility and duplication of efforts evidenced here are the primary problems of the classic web application in the modern web platform. These issues do not exist in an SPA.

In an SPA there is a clear line of separation between the responsibilities of the server and client. The API server responds to data requests, the application server supplies the static resources, and the client runs the show. In the case of the products carousel, an empty document that contains a payload of JavaScript and template resources would be sent by the application server to the browser. The client application would then

initialize in the browser and request the data required to render the view that contains the products carousel. After receiving the data, the client application would render the first set of items for the carousel. Upon pagination, the data fetching and rendering lifecycle would repeat, following the same code path. This is an outstanding engineering solution. Unfortunately, it is not always the best user experience.

In an SPA the initial page load can appear extremely sluggish to the end users, because they have to wait for the data to be fetched before the page can be rendered. So instead of seeing content immediately when the pages loads, they get an animated loading indicator, at best. A common approach to mitigate this delayed rendering is to serve the data for the initial page. However, this requires application server logic, so it begins to blur the lines of responsibility once again and adds another layer of code to maintain.

The next issue SPAs face is both a user experience and a business issue. They are not SEO-friendly by default, which means that users will not be able to find an application's content by searching. The problem stems from the fact that SPAs leverage the hash fragment for routing. Before we examine why this impacts SEO, let's take a look at the mechanics of SPA routing.

SPAs rely on the fragment to map faux URI paths to a route handler that renders a view in response. For example, in a classic web application an "about us" page URI might look like *http://domain.com/about*, but in an SPA it would look like *http://domain.com/#about*. The SPA uses a hash mark and a fragment identifier at the end of the URL. The reason the SPA router uses the fragment is because the browser does not make a network request when the fragment changes, unlike when there are changes to the URI. This is important because the whole premise of the SPA is that it only requests the data required to render a view/page, as opposed to fetching and parsing a new document for each page.

SPA fragments are not SEO-compatible because hash fragments are never sent to the server as part of the HTTP request (per the specification). As far as a web crawler is concerned, *http://domain.com/#about* and *http://domain.com/#faqs* are the same page. Fortunately, Google has implemented a work around to provide SEO support for fragments: the hashbang (*#!*).

> **NOTE**
>
> Most SPA libraries now support the History API, and recently Google crawlers have gotten better at indexing JavaScript applications—previously, JavaScript was not even executed by web crawlers.

The basic premise is to replace the # in an SPA fragment's route with a *#!*, so *http://domain.com/#about* would become *http://domain.com/#!about*. This allows the Google crawler to identify content to be indexed from simple anchors.

> **NOTE**
>
> An anchor tag is used to create links to the content within the body of a document.

The crawler then transforms the links into fully qualified URI versions, so *http://domain.com/#!about* becomes *http://domain.com/?query&_escaped_fragment=about*. At that point it is the responsibility of the server that hosts the SPA to serve a snapshot of the HTML that represents *http://domain.com/#!about* to the crawler in response to the URI *http://domain.com/?query&_escaped_fragment=about*. Figure 1-5 illustrates this process.

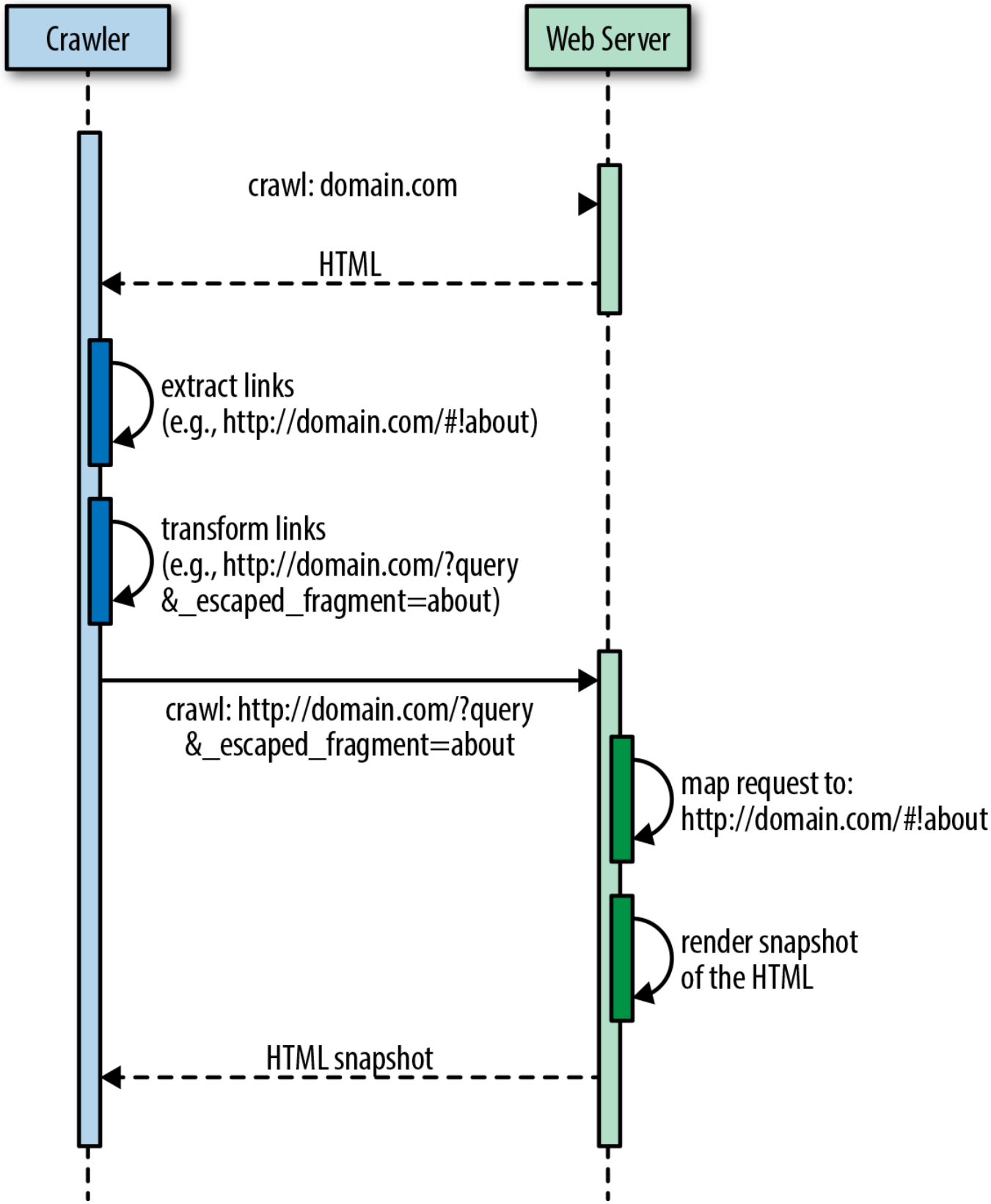

Figure 1-5. Crawler flow to index an SPA URI

At this point, the value proposition of the SPA begins to decline even more. From an engineering perspective, one is left with two options:

1. Spin up the server with a headless browser, such as PhantomJS, to run the SPA on the server to handle crawler requests.

2. Outsource the problem to a third-party provider, such as BromBone.

Both potential SEO fixes come at a cost, and this is in addition to the suboptimal first page rendering mentioned earlier. Fortunately, engineers love to solve problems. So just as the SPA was an improvement over the classic web application, so was born the next architecture, isomorphic JavaScript.

Isomorphic JavaScript applications

Isomorphic JavaScript applications are the perfect union of the classic web application and single-page application architectures. They offer:

- SEO support using fully qualified URIs by default—no more *#!* workaround required—via the History API, and graceful fallbacks to server rendering for clients that don't support the History API when navigating.

- Distributed rendering of the SPA model for subsequent page requests for clients that support the History API. This approach also lessens server loads.

- A single code base for the UI with a common rendering lifecycle. No duplication of efforts or blurring of the lines. This reduces the UI development costs, lowers bug counts, and allows you to ship features faster.

- Optimized page load by rendering the first page on the server. No more waiting for network calls and displaying loading indicators before the first page renders.

- A single JavaScript stack, which means that the UI application code can be maintained by frontend engineers versus frontend and backend engineers. Clear separation of concerns and responsibilities means that experts contribute code only to their respective areas.

The isomorphic JavaScript architecture meets all three of the key acceptance criteria outlined at the beginning of this section. Isomorphic JavaScript applications are easily indexed by all search engines, have an optimized page load, and have optimized page transitions (in modern browsers that support the History API; this gracefully degrades in legacy browsers with no impact on application architecture).

Caveat: When Not to Go Isomorphic

Companies like Yahoo!, Facebook, Netflix, and Airbnb, to name a few, have embraced isomorphic JavaScript. However, isomorphic JavaScript architecture might suit some applications more than others. As we'll explore in this book, isomorphic JavaScript apps require additional architectural considerations and implementation complexity. For single-page applications that are not performance-critical and do not have SEO requirements (like applications behind a login), isomorphic JavaScript might seem like more trouble than it's worth.

Likewise, many companies and organizations might not be in a situation in which they are prepared to operate and maintain a JavaScript execution engine on the server side. For example, Java-, Ruby-, Python-, or PHP-heavy shops might lack the know-how for monitoring and troubleshooting JavaScript application servers (e.g., Node.js) in production. In such cases, isomorphic JavaScript might present an additional operational cost that is not easily taken on.

NOTE

Node.js provides a remarkable server-side JavaScript runtime. For servers using Java, Ruby, Python, or PHP, there are two main alternative options: 1) run a Node.js process alongside the normal server as a local or remote "render service," or 2) use an embedded JavaScript runtime (e.g., Nashorn, which comes packaged with Java 8). However, there are clear downsides to these approaches. Running Node.js as a render service adds an additional overhead cost of serializing data over a communication socket. Likewise, using an embedded JavaScript runtime in languages that are traditionally not optimized for JavaScript execution can offer additional performance challenges (although this may improve over time).

If your project or company does not require what isomorphic JavaScript architecture offers (as outlined in this chapter), then by all means, use the right tool for the job. However, when server-side rendering is no longer optional and initial page load performance and search engine optimization do become concerns for you, don't worry; this book will be right here, waiting for you.

Summary

In this chapter we defined isomorphic JavaScript applications—applications that share the same JavaScript code for both the browser client and the web application server—and identified the primary type of isomorphic JavaScript app that we'll be covering in this book: the ecommerce web application. We then took a stroll back through history and saw how other architectures evolved, weighing the architectures against the key acceptance criteria of SEO support, optimized first page load, and optimized page transitions. We saw that the architectures that preceded isomorphic JavaScript did not meet all of these acceptance

criteria. We ended the chapter with the merging of two architectures, the classic web application and the single-page application, which resulted in the isomorphic JavaScript architecture.

Get *Building Isomorphic JavaScript Apps* now with O'Reilly online learning.

O'Reilly members experience live online training, plus books, videos, and digital content from 200+ publishers.

START YOUR FREE TRIAL

UPCOMING CONFERENCES

Artificial Intelligence Conference

Open Source Software Conference

Software Architecture Conference

Strata Data Conference

TensorFlow World

Velocity Conference

THE O'REILLY APPROACH

Our Company

Teach/Speak/Write

Careers

Community Partners

SOLUTIONS

For Teams

For Enterprise

For Individuals

For Government

For Education

SUPPORT

Customer Service

Contact Us

Privacy Policy

DOWNLOAD THE O'REILLY APP

Take O'Reilly online learning with you and learn anywhere, anytime on your phone or tablet. Download the app today and:

- Get unlimited access to books, videos, and live training
- Never lose your place—all your devices are synced
- Learn during your commute with online and offline access

O'REILLY®

© 2020, O'Reilly Media, Inc. All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.

Terms of Service • Privacy Policy • Editorial Independence