The Modern Web

The electronic computer, the Internet, and the World Wide Web are the greatest inventions of our time, and we are lucky to be working on the crest of the wave that the impact of these inventions is having on society.

Developers have typically worked in technology companies, or inside technology departments of other organizations, but as the earliest developers are now retiring, the latest generation is finding employment in many different areas. The term “digital industry” is often used to describe more traditional businesses that are waking up to the potential that technologists have paved the way for. For these organizations, technology is not just a cost center and a utility, but a core part of whatever business they’re in, even if the services or products they provide are not technological in nature.

It’s with the rise of these digital businesses that the nature of development has changed. If the products and services you provide aren’t technological, it no longer makes sense to align your development teams around technologies, but to instead have your development and delivery teams working alongside non-techies in these digital businesses. To some, “digital” may just be a buzzword, but for those who use it, the change it suggests is very real.

For developers working on these digital teams, deep technical knowledge is less important than knowledge of the business and organization, and speed of delivery becomes key. Getting something out there three weeks early can give a digital organization an edge over its competitors, and with communication being one of the greatest overheads in development, it becomes advantageous for a development team to be able to implement all parts of a new feature, or to fix a bug that impacts multiple components, without having to negotiate with another team to do so. The developers working on these teams are sometimes referred to as T-shaped, because their breadth of knowledge is just as important as any depth of technical specialty they may have. Similarly, when software fails, having a team operating the software that is separate from the team that builds it increases the communication overhead. Digital teams eliminate those communication overheads by adopting ways of working and a culture known as DevOps, which blends the lines between development and operational responsibility.

All of these factors have given rise to the “full stack” developer.

For some, the term full stack is a misnomer. It’s rare to see a developer who’s as comfortable writing x86 assembler as they are slinging Sass, but for the organizations these developers work for, the stack they care about is the one that gives them value. Everything below this—from the programming language to the OS to the physical hardware—is a utility. A development team in this kind of organization should be able to each part of the stack that matters. For some organizations, the stack does extend much further, but it’s only because at their scale, there is value over using off-the-shelf solutions for those lower levels, or it exists for legacy reasons. Many organizations still run their own data centers, for example, as the investment has already been made, or they have particular requirements (legal regulations, heightened security, or very largestorage), but for many others, running that part of the stack themselves offers no valueover buying it as a utility.

For these digital organizations, development teams are no longer full of programmers. The output that matters to these teams is not code; it is solutions to problems. Many of these problems tend to involve interaction with users, which has led to the rise of User Experience as a discipline, or may simply be process issues, where a dedicated business analyst can add value. The traditional roles of developer, QA, and project manager are all still there, but we now have a bunch of other people to help us.

The breadth of modern web development is staggering, and it’d be hard for all developers in a team to be equally good at all parts of it. In reality, there is no such thing as a full stack developer, but instead the full stack development team. You can make a full stack development team from individual specialists, but it will never be as effective of ones filled with “specializing generalists”. Work comes in peaks and troughs, and is never evenly distributed over all parts of the stack. The full stack development team should therefore be greater than the sum of its parts, and share traits with the concept of the High-Performing Team in management theory.

Teams of specialists are also less adaptable to change. LAMP is no longer as cool as it was, MEAN is the new thing, and new languages on the JVM are reviving the popularity of the Java stack, but specialized generalists who work on the principles and foundations rather than the details can adapt to these changes more easily than those who have deep knowledge of one thing.

The stack is dead; long live the stack.

The modern digital business sees tech stacks as a tactical, rather than strategic, decision. They want to be able to leverage the best utilities to help them fulfil that goal, rather than to build and double down on a technical platform that can be hard to change. New languages and techniques are appearing all the time, and a good full stack team should be empowered to choose the best tool for the job, and should be able to adopt new tools as they mature.

So, who is this book for? It's for junior developers and graduates who want to understand modern web development in a digital business. It's for engineers in traditionally structured organizations who are transitioning into this new digital world. It's for leaders of development teams who want to understand more about the work that they lead. It's for developers already on these full stack teams who want to refine their skills. It's not for people who don't know how to code, or who are starting out in their web development journey.

This book assumes a basic understand of web development techniques (i.e., you've probably built at least a web page, or an API), but won't go into details of implementation. The beauty of full stack development is that you get to choose your own parts of the stack, based on the best tool for the job, and in a rapidly changing environment, any concrete recommendations are likely to quickly date. What it does aim to teach you is the techniques that apply to modern development, and the pitfalls to avoid when building an application on today's web.

Rise of the Web

Interest in the Web has never been higher. New frameworks, better tooling, and theevolution of standards are emerging from everywhere, and it's hard to keep up.

The modern Web is constantly growing, more devices than ever can access it and interact with it, and techniques are constantly growing to address this. The addition of media queries to CSS allowed web page styling to target screens with specific characteristics, rather than have the page look the same on every device. This small adjustment fundamentally changed how sites were designed, giving birth to a brand-new set of techniques for building websites known as Responsive Web Design, in which the design of a single web page should respond to the capabilities of the device it is being rendered on. However, CSS is still missing some of the features needed to make responsive web design a success—the only information we can reliably use is the screen size and pixel density, whereas there are many other characteristics that would influence design if they were available, such as interaction methods (mouse, touch-screen, speech, five-point remote control, etc.) So the Web is still a constrained format. Regardless of these constraints, the Web's distribution method has seen the replacement of the traditional desktop app in many cases.

In the early days of computing, users interacted with applications on servers via terminals, or thin clients. This was powerful, as it meant the application was only running in one place, and everyone was using the same one. The rise of personal computing and more power on individual desktops enabled a move to fat clients, where more and more of the business logic ran in programs distributed to the end users' desktops. This caused problems with distributing those applications to all the people that needed them as well as maintaining different versions on individual devices. It was a slow way of doing things. There were many benefits to computers becoming smaller and more portable, allowing them to appear in people's homes and not just in large connected facilities, but an always-connected world was still a long way away.

With the rise of the Web, we can now benefit from both worlds: servers distribute software to clients at the point of request, rather than the software being preloaded onto machines. At first, we lost the ability to use an app without a connection to a server, but the rise in mobile connectivity made this less of a problem, usually by considering an internet connection as the primary state, then using various controlled modes of failure or degradation to handle being disconnected (such as error messages, or being temporarily offline). More recently, the Web has adopted better support for working offline, using features such as service workers that allow apps to continue working while disconnected.

For a long time, the standards of the Web and the performance of browsers were far behind that of a native application. Microsoft and others introduced tools such as ActiveX, Java applets, and Flash to work around this, but each of these approaches hadits own challenges. It was Microsoft that kick-started the rise of what would eventually allow web standards to be used to produce applications that would replace those desktop ones, with the introduction of the `XMLHttpRequest` API (XHR) to JavaScript.

Now, web standards have matured to the point of meeting many needs of rich applications, and development of the Web is happening at such a pace that any missing functionality will quickly appear. The Web has become a tool for distributing these applications, and web technologies are even replacing parts of native apps, with web apps packaged for native distribution mechanisms.

Mobile Web

Just as web applications are replacing many desktop applications, native apps on mobile platforms have gone the other way. Most major mobile platforms now have a built-in app distribution mechanism. Desktop Linux distributions had this for years as well, but Linux on the desktop has never had the breakthrough that many enthusiasts hoped for. Apple and Microsoft, on the other hand, now bring app stores to their desktop OS's as an extension of their mobile stores.

This "app store" distribution mechanism has two key advantages over the Web that desktop applications never had: an effective market with a mechanism for charging a fee to obtain the app, and the ability to live on the launcher for that device to gain headspace for the user. The mechanism of charging fees is also one of the weaknesses of the distribution model, as the owner of the store can include unfair terms or limit certain apps from the store, which has led some app publishers to bypass the store. Websites and web applications can be added to launchers too, but this process is often clunky by comparison and rarely used. Of course, mobile applications, like desktop apps before them, still have some run-time performance gains, as well a higher level of access to the device and its hardware. But the mobile market is more fragmented than the desktop market, and in order to maximize reach, "cross-platform" toolkits are often used to build native apps to target multiple devices. Many recent examples of these are actually just wrappers for web technologies. For many organizations, web remains the default distribution mechanism, unless there are specific requirements that only native apps and app marketplaces can fulfill, or the perceived mindshare that a native app seems tohave over a web app is too tempting to ignore.

The State of HTML

HTML is the language of the Web. To build a web site, at some point HTML will be involved. The HTML standard doesn't just cover the tags you write on a page (the markup), but also specifies the Document Object Model (how you manipulate web pages from JavaScript) and CSS too.

And HTML is a little bit complicated right now. The World Wide Web Consortium (W3C) originally specified the HTML standard, and were happy with the HTML4 standards. Focus then moved to XHTML, a subtly different form of HTML based on the XML standard. XML, the Extensible Markup Language, offered a large degree of power—for example, other XML documents could be embedded inside an HTML document, but as XHTML was famously never supported by Internet Explorer, this power was never realized. There were other issues with XHTML too: a standards- compliant XML parser refuses to render anything if there's an issue with the XML, which hindered migration from the much laxer HTML4 standards.

Browser makers, most famously Opera and Mozilla, disagreed with this move to XML, and were frustrated by the slow approach of a standards body, so they took matters into their own hand and put together the WHATWG (Web Hypertext Application Technology Working Group) which developed it's own variation of the HTML standard. They proposed this standard —HTML5—to the W3C, and it was adopted, but the W3C and WHATWG definitions of HTML5 have drifted. The WHATWG defines HTML5 as a living standard, which means there will never be a definitive specification of HTML5 to implement.

Sounds like a nightmare, right? Fortunately, most browser makers have now reacted to this living standard and moved to a much more frequent release model, and users have followed, so the time between a new feature in HTML being accepted and it becoming available in browsers is getting shorter all the time. The biggest exception is Apple, which still links versions of Safari and Mobile Safari to OS X and iOS releases (this is particularly bad on iOS, where users are restricted from installing alternate runtimes), and some Android vendors who bundle their own browser instead of Google's Chrome.

Fortunately, there is a lot of tooling to help you here. More often than not, you can write HTML, CSS, or JavaScript using the latest techniques, and there are tools in the front end that will transform your code in a backwards compatible version that can handle the old browsers for you. Fortunately, with HTML markup, most browsers will treat unrecognized tags as generic `<div>` tags, and they can be still be styled as such (although some older browsers require a polyfill, such as the famous `html5shiv.js`). Additionally, there are many JavaScript libraries, known as "polyfills," that exist to provide a pure JavaScript version of any new features of the JavaScript language, allowingthem to be used on older browsers. These are covered in detail in the Front End chapter.

With two differing standards, it can also be hard to know which documentation to turn to. In reality, what really matters is what the browsers actually do. Fortunately, thedays of the browser wars—where different vendors would implement the same idea
in different ways—are over, and there is much more collaboration between browser developers in order to make the standards work. Browsers also indicate a non-standard feature with a "vendor prefix," where the JavaScript API or CSS attribute is prefixed with the name of the browser maker to indicate that it is a browser-specific feature (such

features should be used with caution as part of a progressive enhancement approach, or avoided entirely). The Mozilla Developer Network is a fantastically thorough resource, and the brilliant website caniuse.com will tell you how well supported a particular feature is, and any quirks it may have.

Many sites being built today will have to run on a vast number of browsers, all with different versions and different levels of support, but there are also many that don't have to. If you're building an internal app for a company, and that organization uses Internet Explorer on Windows, it's very tempting to build and test only in that browser, but things could suddenly change— for example, the sudden introduction of tablets to an organization. Targeting a known runtime can make your life easier, since you'll have less to test, and you won't have to support very old/broken devices, but don't lull yourself into a false sense of security—write to the standards, and verify this by testing it in a different browser. Doing this upfront can save you and your organization a lot of pain down the line.

Sometimes, you'll find a feature you want to use that isn't supported by every browser you want to target. In cases like this, a technique known as progressive enhancement is very useful. Progressive enhancement is a technique where you deliver a basic version of some functionality to a device, and then test if the device can support a more advanced alternative, then enable the enhanced alternative if the test is successful. Progressive enhancement is a useful technique to solve many different problems, and is covered in much more detail in the Front End chapter.

Applications vs. Web Sites

With the rise of the Web being used as a delivery platform for what once would have been a desktop application, the line between what once would have been called a "web site" and what is now called a "web app" has blurred. Web sites were generally content-based sites—they existed to communicate information, often with some interactivity involved, but at their heart, they were about information. Many frameworks have risen to help build web apps, and these frameworks are often used to build web sites too, even though that's not what they're best suited for. Angular and Backbone are examples of these types of frameworks, and are optimized for when data needs to be modified by the users of these apps by providing abstractions for these operations. These abstractions are often not helpful for content-based websites. Before embarking on a project, you have to decide whether you’re making a web site or a web app. Many projects are likely to include both—for example, a catalogue a user browses works best as a site, but inventory management tools work best as an application.

You could consider a website as a subset of the functionality of a fully-fledged web application, but by identifying it as purely a website, it becomes easier to use a set of more constrained tools that will often give you a better result, as well as let you reach that result in less time. Web app toolkits and frameworks may appear powerful, but often break an important fundamental of the Web - that the Web is made up of a series of linked pages. If content on your site needs to be linked to directly from a URL, either through sharing the URL on social media, or from a search engine, then considering it a website, and avoiding web app toolkits will allow you to benefit from many fundamentalfeatures of the Web (such as shareable URLs, as well as caching and archiving).