Group Marketing

Front end

Coding Standards and Best Practices

Version 1.0

Approval and Version History

*List all changes to the Coding Standards, capturing any changes in the document*

| **Version** | **Date** | **Author** | **Reviewer** | **Description** |
| --- | --- | --- | --- | --- |
| 1.0 | 10 Jan 2021 | Mikul Gohil/Victor |  | Initial Draft |
| 1.0 | 17 January 2021 |  | Ashish Bansal |  |
|  |  |  |  |  |

Contents

[1. Introduction 5](#_Toc61772042)

[1.1 Goals 5](#_Toc61772043)

[1.2 Getting Started 6](#_Toc61772044)

[1.3 General Standards 6](#_Toc61772045)

[1.4 Deliverables 8](#_Toc61772046)

[2. HTML 8](#_Toc61772047)

[2.1 Goals for Markup 9](#_Toc61772048)

[2.2 Getting Started on Markup 9](#_Toc61772049)

[2.3 HTML Markup Best Practices 10](#_Toc61772050)

[2.4 Markup Deliverables 15](#_Toc61772051)

[2.5 Next Steps & HTML5 Resources 15](#_Toc61772052)

[3. CSS 16](#_Toc61772053)

[3.1 Goals for Effective CSS 16](#_Toc61772054)

[3.2 Getting Started with CSS 17](#_Toc61772055)

[3.3 CSS Best Practices 19](#_Toc61772056)

[3.4 CSS Deliverables 27](#_Toc61772057)

[3.5 Next Steps & CSS Resources 27](#_Toc61772058)

[4. JavaScript 28](#_Toc61772059)

[4.1 Goals 28](#_Toc61772060)

[4.2 Getting Started on JavaScript 29](#_Toc61772061)

[4.3 JavaScript Best Practices 31](#_Toc61772062)

[4.4 JavaScript Deliverables 37](#_Toc61772063)

[4.5 Next Steps & JavaScript Resources 38](#_Toc61772064)

[5. Responsive Web Design 39](#_Toc61772065)

[5.1 Goals of Responsive Web Design 39](#_Toc61772066)

[5.2 Getting Started with Mobile Development 40](#_Toc61772067)

[5.3 Responsive Design Best Practices 42](#_Toc61772068)

[6. ReactJS 49](#_Toc61772069)

[6.1 Keep components small and function-specific 49](#_Toc61772070)

[6.2 Reusability is important, so keep creation of new components to the minimum required 49](#_Toc61772071)

[6.3 Consolidate duplicate code – DRY your code 50](#_Toc61772072)

[6.4 Comment only where necessary 51](#_Toc61772073)

[6.5 Name the component after the function 51](#_Toc61772074)

[6.6 Use capitals for component names 52](#_Toc61772075)

[6.7 Mind the other naming conventions 52](#_Toc61772076)

[6.8 Separate stateful aspects from rendering 52](#_Toc61772077)

[6.9 Code should execute as expected and be testable 53](#_Toc61772078)

[6.10 All files related to any one component should be in a single folder 53](#_Toc61772079)

[6.11 Use tools like Bit 54](#_Toc61772080)

[6.12 Use snippet libraries 54](#_Toc61772081)

[6.13 Write tests for all code 54](#_Toc61772082)

[7. PWA 55](#_Toc61772083)

[7.1 Web app manifest file: manifest.json 55](#_Toc61772084)

[7.2 Service Worker 55](#_Toc61772085)

[7.3 HTTPS 56](#_Toc61772086)

[7.4 App icon 56](#_Toc61772087)

[8. SEO 56](#_Toc61772088)

[8.1 Principles of search engine optimization 56](#_Toc61772089)

[8.2 What’s wrong with optimizing single-page applications (SPA) for search engines? 57](#_Toc61772090)

[8.3 Solving the problem 58](#_Toc61772091)

[8.4 Isomorphic React apps 59](#_Toc61772092)

[8.5 Server-side rendering with Next.js 59](#_Toc61772093)

[8.6 The bottom line 60](#_Toc61772094)

[9. File Structure 60](#_Toc61772095)

1. Introduction

This document contains the guidelines and best practices for the front-end web development team at EmiratesNBD.

Each item here represents either:

1. A reminder to follow existing standards or industry conventions,
2. guidance on what constitutes professional patterns and organization, or
3. a decision we've made favoring one method over its alternatives.

What this document is not is a series of explanations as to how front-end technologies work; a basic familiarity is assumed. It also does not provide evaluations of the pros and cons of various alternatives unless there is common confusion about which option is best; when appropriate we pick what we consider to be the best solutions and present them. Issues that don't yet have a clear solution are considered flexible and may or may not be listed.

1.1 Goals

Our motivations in creating this document are to:

1. Foster code consistency across our projects.
2. Facilitate ease of maintenance.
3. Ensure we create professional quality Web sites.
4. Guide staff on-boarding or educate new developers.

This document is not intended to replace common sense, conventions requested by particular clients, teams, or prevent expressive or creative solutions to problems. Team or project-specific agreements or client requests will always supersede this document's content.

1.1.1 Professional Responsibility

We are experts in our field creating solutions for our clients and their audiences, not for ourselves. Every technology and code choice needs to be measured against the benefits to the project versus the cool factor or how trendy a particular solution may be.

Our industry is wrought with the flavor of the month, so please be deliberate.

Always remember that **just because you can does not mean you should**. Some solutions are not reliable, may not perform well, or may be difficult to maintain over time or add more code to. Always remember your code may not be the last added to a project in that particular feature area.

1.2 Getting Started

At the outset of the project it is essential to **properly understand the goals of the project** and **identify the specific deliverables** expected of the front-end team. Where your responsibilities begin and end should not be taken for granted or assumed.

It's important to understand how the development environment will work, what tools will be available, and what the differences between development, test, and production environments may ultimately be.

Finally, all project teams should get a reasonable understanding of the what client's **browser and device requirements** are. Make no assumptions as to the technology available either from the client or their audience.

1.2.1 Pillars of Front-end Development

Whenever possible, the front-end technology solutions produced shall adhere to industry best practices honoring as strict a separation of concerns as possible between:

* [Semantic](http://www.bbc.co.uk/guidelines/futuremedia/technical/semantic_markup.shtml) HyperText Markup Language (HTML) for structure
* Cascading Style Sheets (CSS) for presentation
* JavaScript (JS) for behavior and interaction

When at all possible, we strive for a [progressive enhancement](https://en.wikipedia.org/wiki/Progressive_enhancement) strategy.

1.3 General Standards

For any project:

* Consistency and conventions between team members is paramount.
* Solutions should be as simple and clear as possible.
* Solutions should serve a specific purpose.
* Clever code does not mean good code; readability is **critical**

A key hallmark of professional code includes a notion that while we are writing code that must reach a desired goal, we are also creating code that must be read and understood by others.

1.3.1 Code Consistency

Usage of the same patterns is critical between team members so as to never cause confusion.

It's worth establishing conventions at the project start or enabling automatic settings in the build or editor environments that might enforce particular rules.

1.3.2 Indentation

Please consistently indent, nest, include braces, quotes, and new lines so that code is clear and can be read easily. New code that is added should never deviate from existing formatting conventions or change the indent levels.

For all code languages, we recommend the **use soft tabs** comprised of four spaces per tab. Hitting the Tab key in your text editor should generate four space characters rather than one tab character. This results in our code appearing identical across platforms.

If **tab stops** are favored by a team, simply **maintain consistency** for a project and it's deliverables so developers can make adjustments to their editing environments a **single** time.

1.3.3 Readability

We encourage liberal use of whitespace, comments, and descriptive variable names as appropriate for writing easy-to-read code.

* There is no need to write code in an obfuscated or compressed way for the purpose of file-size savings.
* We will use automated server-side or other build processes to optimize files.
* This includes concatenating files, code minification, gzipping, and setting "Far Future Expires".

The ability for another developer to read the code is paramount above other concerns, especially if optimization can be handled another way.

1.3.4 Third-Party Libraries

Un-minified libraries and third-party scripts should be leveraged in local development environments for easier debugging if available. The code should be committed to source control in an unmodified state, or simply referenced in dependency management components that resolve the files in a build. The final products will be compressed with the rest of the source for delivery.

Likewise, third-party code and libraries should never be modified and their original source and the license must be documented and be appropriate for a project. Any changes to third party code must be agreed upon and must be for specific reasons. If changes are mandated by bug fixes then the appropriate upstream project should have the changes submitted (assuming the code is part of an open source repository).

Library code should be treated as an external dependency and should be considered something that may need to be wholesale updated or replaced at a later time.

Inclusion of any third-party code should be carefully considered and verified with the project team as the appropriate solution to a given problem. "Adding another plug-in" is not always the best solution. Finally, selection of third party libraries should be done carefully and not be out of alignment with the nature of the problem being addressed.

Note:

Please see the [JavaScript section](#_eshsxfa0big4) for more information about selection of third party code in a project.

To be blunt, use the right tool for the right job.

1.4 Deliverables

Quality deliverables are essential for professionals. Sloppy or messy deliverables are unprofessional and reflect poorly on the final product and the delivery team. Please remove legacy files, be certain the work is delivered in a clean file system, and in an orderly, logical structure that serves a clear purpose.

2. HTML

HTML markup defines the content of a document and gives it a rudimentary structure such as section dividers, headers, paragraphs, lists, menus, and forms.

2.1 Goals for Markup

Please follow conventions established for a given project so all team members can have the same expectations around document structure and markup.

Structural consistency is critical when talking about the types of pages being used on a site or in a Web app. The markup structure provides all the necessary hooks for scripting and behavior, so it's important that the appropriate hooks are in place.

A clear, clean, and concise HTML structure is also necessary for semantics, flexibility, and a **reliable deployment environment**. Do **not** deviate from established templates or patterns without architect approval.

Which markup is used does matter:

* Use the most meaningful yet minimal markup required to present the styles and interaction required
* Application-centric deliverables often have different types of requirements; please code accordingly
* Maintain a clear separation of concerns, avoid in-line styles and in-line JavaScript whenever possible
* Have reference implementations so that each team member knows what sorts of structures are appropriate, as well as where to add new code.
* Build pages as a **library of components**, in such a way that blocks of code can be broken up and reused when implemented.
* Be sure front-end code is compatible with destination environments and delivery platforms.

The flexible nature of HTML markup and how loosely browsers interpret markup sometimes lends itself to inconsistencies not always being discovered immediately. This belies the care necessary in crafting a document's structure and in following established patterns.

2.2 Getting Started on Markup

When crafting the HTML for a website, environment or technical constraints may impact the type of markup that can be used. Please discuss the final delivery environment in depth with technical leads and clients so that pages are not structured or styled in some way that is not effective for the project solution.

Discuss types of:

* Templates and types of pages.
* Which sections of pages (i.e. components) are reused or managed by software vs. by hand.
* Frameworks, CSS grid systems (custom or otherwise).
* Server-Side delivery platforms.

Note that it is vital to take into account how the site will ultimately be maintained and who will be doing that work.

2.3 HTML Markup Best Practices

As noted these guidelines are flexible for projects as long as consensus or need determines a particular path, consistency is what matters most.

2.3.1 Semantic Markup

HTML provides a number of [semantic constructs](http://www.bbc.co.uk/guidelines/futuremedia/technical/semantic_markup.shtml) that allow automated tools like search engines and screen readers to make sense of the document and to understand relationships between pieces of content. Use semantic markup whenever possible — that is to say use elements with specific meanings for specific purposes to convey the spirit of the markup.

A well-written HTML document will make appropriate use of these semantic elements and leave all responsibility for controlling the presentation of the document to the CSS style sheet.

2.3.2 HTML Standards and Browser Support

All markup will be written using the latest HTML5 markup specifications from the W3C, as implemented by browsers and devices that meet project requirements. When creating markup be sure that the target environments support the techniques being implemented, or that there is a fall-back plan.

2.3.3 Doctype

Always include a proper doctype to trigger standards mode. Omitting the doctype [triggers quirks mode](https://developer.mozilla.org/en-US/docs/Quirks_Mode_and_Standards_Mode) and should always be avoided. The HTML5 doctype is simple and easy to remember.

|  |
| --- |
| <!doctype html> |

2.3.4 Character Encoding

All markup should be delivered as UTF-8, since it has the best support for internationalization. The character encoding should be designated in both the HTTP header and the head of the document via a meta tag. If the server happens to omit the HTTP header, browsers can take a guess at the character encoding and begins parsing and rendering the markup in a particular way. If there are inconsistencies, the browser will re-parse and re-render, throwing away all that work and starting over if it encounters the meta tag and its guess was incorrect. As a best practice, we always put the meta tag as early in the <head> tag as early as possible — however server-settings are ideal.

|  |
| --- |
| <meta charset="UTF-8"> |

2.3.5 Optional and Self-closing Tags

While current standards designate certain closing elements and even document level elements as optional, use all open and closing elements nested in the correct ways to ensure maximum compatibility and clarity of document structure.

Generally speaking, self-closing XML (i.e. XHTML, XML) style tags are not necessary.

|  |
| --- |
| <!-- closing "/" is not necessary --> <img src="/logo.png" alt="ISOBAR">  <!-- include closing tags, however --> <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit:</p> <ul>  <li>Vero sunt veritatis magni sit odit,</li>  <li>voluptatum ratione suscipit.</li> </ul> |

Unusual markup (or indeed, invalid) can lead to bugs in page rendering, DOM interpretation, or even how styles are applied, so it should be avoided whenever possible.

2.3.6 Validation

[Valid markup](https://validator.w3.org/) is a goal but not a mandate. However, be aware validation can be an excellent starting place while debugging a Web page — especially if the problems are unusual.

If it becomes necessary, please have reasons for invalid markup — otherwise it is just sloppy code.

2.3.7 Indentation in HTML

Indent nested elements and tags with single indentation settings, whatever they may be, for each level in the hierarchy of the document.

|  |
| --- |
| <div>  <p>Lorem ipsumLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod.</p>  <ul>  <li>tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,</li>  <li>quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo  consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse</li>  <li>cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non</li>  <li>proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</li>  </ul> </div> |

2.3.8 HTML5 Elements

To provide additional semantic value to our documents, make use of HTML5 elements such as <header>, <article>, and <section> where appropriate. However, in cases where the HTML needs to be as backwards-compatible as possible, do not apply IDs or classes to them, since older browsers do not understand these elements by default and will not apply styling to them.

|  |
| --- |
| <header>  <div class="site-header">  ...  </div> </header> |

2.3.9 Attribute Values

Use quotes to surround all attribute values in HTML, despite quotes being optional in HTML5. This maintains consistency between attribute values that contain whitespace and those that don't.

<form class="registration module" action="/register" method="POST">

2.3.10 IDs vs. Classes

HTML elements can be identified by using the id and class attributes. An ID is a unique identifier for that particular element; no other element on the page should use the same ID.

This uniqueness allows <label> elements to associate themselves with a particular input and URLs to jump to a particular scroll position on a page.

Classes are not unique. The same class can be used on multiple elements within a page, and a single element can have more than one class, in a space delimited list.

|  |
| --- |
| <ul id="categories">  <li class="category">Jackets</li>  <li class="category specials">Accessories</li>  <li class="category">Shoes</li> </ul> |

When coming up with names for an ID or class, we use semantic names like "secondary-nav" or "primary-button" that describe what the element is, rather than names like "left-nav" or "blue-button" that describe what the element looks like, which can change over time. We also use lowercase names with hyphens separating words as opposed to camelCase or underscores.

2.3.11 Anchors & Links

All links should point to absolute or relative URLs with user-readable content. Do not link to XML or JSON resources that are designed to be Ajaxed by JavaScript instead of navigated to directly, and do not put JavaScript in an anchor's href attribute like javascript:loadPage(2);. This allows search engines to index the content, allows the user to open the links in a new tab or window, and means the links will still work when JavaScript is broken, disabled, or not supported. This will require that the back-end be able to return a full HTML page for each important content state (e.g. sorting a table column).

2.3.12 Paragraphs

Avoid using <br> tags to separate paragraphs or lines of text. Use <p> instead with proper opening and closing elements.

2.3.13 Definition Lists

Use definition lists to display a single record of name-value pairs, like a contact card.

2.3.14 Tables

Tables should not be used for page layout; only use them when you need **to display tabular data**. Tables provide an important semantic association (used mostly by screen readers for the sight-impaired) between row/column headers and their data, so use <table> rather than other elements when displaying multiple records of data.

The <caption> element is the recommended way to describe a table for both sighted and sight-impaired users, though this can also be done less semantically in the normal page text around the table. Use the <thead> and <tbody> elements to denote which row contains column headers so when a user prints the website and the table runs onto another page, browsers can display the <thead> on each page for easier readability. Remember to use the scope attribute on the <th> element to indicate whether the header applies to the row or column.

|  |
| --- |
| <table>  <caption>First two U.S. presidents</caption>  <thead>  <tr>  <th scope="col">Name</th>  <th scope="col">Took office</th>  <th scope="col">Party</th>  </tr>  </thead>  <tbody>  <tr>  <td>George Washington</td>  <td>April 30, 1789</td>  <td>n/a</td>  </tr>  <tr>  <td>John Adams</td>  <td>March 4, 1797</td>  <td>Federalist</td>  </tr>  </tbody> </table> |

2.3.15 Forms

For both semantic and functional reasons, we make full use of the <form> tag for all sections requiring user input. All form action attributes should point to URLs with user-readable content, so they will still work if the form is submitted by the user before JavaScript has loaded on a page, or if JavaScript is broken, disabled, or not supported. This will require that the back-end be able to return a full HTML page for form submission (e.g. registering a new user, editing the quantity in a shopping cart).

Do not nest the HTML form element tag.

2.3.16 Input Labels

All input fields should be associated with a <label> element. The for attribute of the <label> element should contain the ID of the corresponding input field. This means the input field will receive focus when a user clicks the label and also enables screen readers for sight-impaired users to read out an appropriate description of the input field.

|  |
| --- |
| <label for="home-address">Home Address</label> <input id="home-address" type="text"> |

2.4 Markup Deliverables

Typically HTML deliverables are incorporated into Content Management Systems or application delivery platforms as templates. A plan for incorporation of templates that leverage patterns created during the markup creation phase should be followed and matching types of pages to templates that were created, so that an association between the source markup and the destination markup can be maintained over time.

### 

2.5 Next Steps & HTML5 Resources

Considerations:

1. Site maintenance procedures
2. Browser testing strategies
3. How new features will be added
4. Where new features will be added
5. What the file system looks like for static site assets
6. If a CDN is involved
7. Naming conventions and organization of graphics and photography assets
8. If the "back-end implementation" of static HTML templates will require review by front-end team members

For current links and references, please see our Wiki on Github.

3. CSS

Cascading Style Sheets (CSS) is where the visual presentation and design rules for a website belong. Well-written CSS makes good use of its cascading nature - general styles are applied first, and those styles are overridden for more specific instances as necessary.

3.1 Goals for Effective CSS

CSS is an unusual language which can easily lead to code bloat, inconsistencies in design or clashing code techniques. It is easy to end up with CSS code that is so fragile it can cause site-wide regressions with small changes.

CSS should:

* Be easy to maintain.
* Follow clear enough patterns to understand.
* Offer a clear place for new styles going forwards.
* Not be a drag on page loading performance.
* Not include unused style rules.
* Address different devices, browser versions, and do as much as it can with as little code as possible.

When setting up the CSS for a site, always consider:

* What the default styles for HTML elements are going to be.
* Which styles are global styles versus specific one-off use-cases.
* Distinctions between code for layout and for content.
* How the code will evolve and grow.
* Potential impact of bug fixes on the overall site.
* Use of images as CSS background images vs. in-line HTML (content).

3.2 Getting Started with CSS

Planning for a CSS build isn't that different than any other software requirements for Web development. A solid foundation starting point is critical.

Before coding, you should always:

* Review the design.
* Plan around technical constraints.
* Identify how content will be managed.
* Be certain who will be able to modify the design directly via code.

Try to segment code in logical ways:

* Separate page grids and containers from the content.
* Create baseline components which may be extended by other styles.

Baseline components should rarely be modified directly.

More specifically, you may have:

* Core brand styles.
* Basic typography or default HTML rules.
* Site-wide styles.
* Distinct sections of the site.
* Micro-sites and landing pages.
* Components, widgets, or re-usable modules.

3.2.1 Tools

Sometimes a third party library is helpful — but please be certain to not include extra code for no reason other than personal enjoyment. Libraries or tools should be picked based on the advantages they provide.

Common types CSS-related tools might include:

* File concatenation
* Preprocessors
* Minifiers
* Post processors

These should be considered in the context of the rest of the site construction, back-end, and continuous integration processes. Discuss the options with the technical lead on the project.

Be careful as these tools can also introduce unnecessary complexity unless used wisely.

3.2.2 Frameworks

Pre-built UI components or CSS frameworks can be beneficial, however just like any third party code please choose wisely and based on benefit of features and flexibility. Locking development into a library that unintentionally imposes limits is not good.

Some examples of third party frameworks might include:

* UI component or widget libraries (e.g. Foundation, Bootstrap, jQuery UI)
* Grid Systems
* Typography adjustments
* Normalizing code

3.2.3 Establishing Conventions and Development Strategies

Like other aspects of the code on a site, consistency is key. Areas of critical consistency include:

* Code formatting
* Naming conventions
* File and folder structure
* Examples or sample code
* How page components might be broken down or re-used

On a large site never develop using a single CSS style sheet, though a single file served for a page is best. To this end we often recommend the use of CSS preprocessors to break style sheets into smaller, better organized files, or the use of a build process to combine files for serving via HTTP.

3.2.4 Living Style Guides and Reference Implementations

One technique to consider is maintaining static HTML style reference implementations well into integration with server-side / back-end systems. These could be a series of templates or widgets that use the live styles being built. This helps reduce regressions that can happen across the board as the code for the site evolves. Continue to test these reference implementations as they will be the "source of record" for the styles created on the site. They also allow you to more easily distinguish the front-end bugs from the bugs potentially introduced by integration with a complex back-end.

These reference implementations can serve as a living style guide and broken components are easily spotted in testing over time.

Defining a solid style guide to be applied to tag names can significantly reduce the size of the CSS if that style guide is adhered to by both the design and development teams. It is recommended that a style guide is agreed upon at the beginning of a project, defined in HTML and then iterated on by both the design and development teams.

3.3 CSS Best Practices

What follows are some basic concepts for standardization of CSS code. Naturally, feel free to fork, update per project, and even issue pull requests for further discussion based upon experience.

3.3.1 Inclusion

Use the <link> tag to include all your style sheets in the <head> of the document. For optimal page performance, concatenate your CSS into as few files as possible and do not use the @import command to include other style sheets, as this will fire an additional HTTP request and block page rendering until its completion.

|  |
| --- |
| <link rel="stylesheet" type="text/css" href="main.css"> |

3.3.2 Formatting CSS

Basic rules for formatting CSS files:

* Use a new line for every selector and every declaration.
* Use a single space before the opening brace in a set of rules.
* Use lowercase for elements and shorthand hex values, e.g., #aaa.
* Hyphenate class selector names; avoid underscores and camelCase
* Quote attribute values in selectors
* Use one level of indentation for each declaration.
* The closing brace of declaration goes in the same column as the first character of the set of rules.
* Use a single blank line between sets of rules.

Inside sets of rules or style declarations:

* Add a single space between the property and value, for example:  
  prop: value; and not prop:value;.
* Use double quotes for quoted values
* Always include a semi-colon at the end of the last declaration.
* Use shorthand if you can, like:  
  padding: 15px 0; and not padding: 15px 0px 15px 0px;
* When allowed, use 0 without units.

Putting each selector on its own line and each property on its own line is great for readability and so version control systems can clearly show which parts have changed in a diff.

The attributes within a selector can be alphabetized for easy scanning and so that compression algorithms like gzip have a greater chance of finding repeatable patterns.

Some examples:

|  |
| --- |
| .content {  margin-left: -2%; }  .twitter-popular, .twitter-favorites, .twitter-feed {  float: left;  padding-left: 2%;  width: 33.33%;  padding: 15px 0; } |

**Do not indent child styles** underneath their parent styles; this is important for a number of reasons:

* We usually recommend indenting media queries, so this can cause confusion.
* Some CSS preprocessors heavily use indentation.
* Various levels of indentation hinders maintainability.
* HTML and CSS structure can change frequently over the course of a project, quickly rendering obsolete the parent-child relationship the indentation used to represent.

3.3.3 Specificity

Use the minimum specificity required to achieve the desired style. It can be difficult to quickly read and locate styles or even bugs with heavily nested styles in the CSS.

The ID is the most specific selector, since it can only match one element, and the class is a close second. Use those whenever possible rather than HTML tag names.

|  |
| --- |
| /\* BAD \*/ button#back-button { ... } .popular ul li a { ... } .popular > ul > li > a { ... }  /\* GOOD \*/ .back-button { ... } .popular-link { ... } .unpopular-link { ... } |

As a rule, CSS is most maintainable with the simplest selectors possible. Try applying a class to the element you want to target instead.

3.3.4 Do Not Use !important

Avoid using the !important keyword. Treat it like the nuclear option, only to be used in the most extreme of cases. This fundamentally destroys the specificity feature and can even break accessibility for some users.

There is usually another way to achieve the same goal without causing headaches for developers in the future who are either trying to debug a styling issue, or trying to use normal specificity to override a style for a particular element only to find that they can't.

3.3.5 ID Selectors

As noted above, use the lowest level of specificity necessary to get the desired results. This means the use of the ID selector should be minimized. Often creating a new class is preferable to using inheritance or additional specificity to target an element or elements.

ID selectors, if used, should be used mainly as access points for JavaScript or if a very particular use case surfaces. Styles and classes can be applied via the same element with a className.

3.3.6 Vendor Prefixes

When using vendor prefixed features, put the standardized rule at the end to ensure browsers optimize and use the standard if they recognize it.

For example:

|  |
| --- |
| .thing {  -webkit-transition: all 100ms;  transition: all 100ms; } |

3.3.7 Inline Styling

Do not hard code style information into your HTML markup directly, either with the style attribute that accepts CSS or with deprecated attributes such as align, border, or width. These are difficult to maintain and make it harder to track down what is causing an element to appear as it does.

3.3.8 Performance Caveats

In some cases for performance reasons it may be good practice to in-line critical styles in a style block in the document's head. This delivers these styles to the browser in the fastest method possible by preventing the need for an additional HTTP request. Fetching linked style sheets are a blocking operation on the rendering of a Web page in a browser in most cases. An enormous CSS file can mean a highly reduced time to first rendering because a browser may pause during loading of the page to download CSS which may not even be used on the first page.

With the above in mind it may be desirable to include the rules required to render the top portions of a page (i.e. "Above the fold") in advance of styles loaded after the rendering begins. Critical styles can be identified either manually or through the use of a tool. Non critical styles can then be asynchronously loaded, increasing the perceived page load speed.

3.3.9 Box Model

To simplify CSS authoring, we set the box-sizing attribute to border-box for all page elements. This enables us to use round numbers for width like 50% and then apply a padding or border to that same element without needing to

1. adjust the width accordingly using calc (since borders use pixels rather than percents) or
2. create an element inside it to take the padding and border. This is the only case where we use the inefficient universal selector (\*).

Example:

|  |
| --- |
| \* {  -moz-box-sizing: border-box;  -webkit-box-sizing: border-box;  box-sizing: border-box; } |

3.3.10 Coding Patterns

There are a number of popular design patterns for naming conventions on selectors, groupings or extensions of styles in CSS files. Sometimes these are of value and may be used on projects as long as the developers are on board and they are used consistently by the team.

Examples of pattern systems include BEM, SMACSS, Object Oriented CSS, Atomic design, and others.

The downsides to some of these systems are:

* Often rely on less obvious rules that may be difficult to follow.
* They may use syntax that may be objectionable to some developers.
* Some developers may find the syntax difficult to read.

For these reasons it is often best to go with the most simple, basic set of conventions possible, based on obvious patterns.

3.3.11 A Simple CSS Code Pattern

The following sections describe one simple approach, and as long as the types are defined on a project and the patterns are followed, then a clear meaning can be interpreted fairly quickly and easily reading through the CSS, HTML, and JavaScript.

Every site will have distinct requirements but some examples of things that can be standardized on a site build include:

* Global Defaults
* Page Level Rules (grids, site template types, and so on)
* Components
* Modifiers
* State
* JavaScript-only Rules
* Utilities

The following sections describe how some might work.

Global Defaults

For the purposes of discussion we could refer to the global defaults as the baseline HTML elements and their associated styles. Frequently you may wish to use a third party library which normalizes CSS behavior across browsers. Either way, it makes sense to keep these files in their own files.

Page Level Rules

Any site is liable to have a standard baseline set of grids and types of pages. These grids can be collected into their own distinct set of component files — using a broad definition of a global "thing" as a component.

Components

Components are a high level concept for organizing CSS files and rules. A component simply means a grouped set of rules pertaining to an object or set of related objects on a page.

Additionally, encapsulating these components into distinct files is a great option. Within this file, using a naming convention is a tremendous help here for code maintainability and readability.

Group styles under a simple name-space using a **prefix-suffix-modifier** type pattern such as:

|  |
| --- |
| /\* core component \*/ .component { ... }  /\* component elements \*/ .component-header { ... } .component-content { ... }  /\* component descendant \*/ .component-content-group { ... }  /\* component descendant element \*/ .component-content-group-header { ... } .component-content-group-imgs { ... } |

This type of pattern is easy to read, extend, and follow in the absence of something more sophisticated.

Additionally, generally components will be the only CSS class with distinct names that do not have prefixes before the core, root, or base name of the component (e.g. above we are using component).

Just to get developers thinking, these might be things like:

* navbar
* footer
* page-info
* article-date
* lead
* widgetfoo
* byline

Modifiers

If you need to extend an existing component then create distinct modifier classes with the prefix mod- to easily indicate that it is a modifier and not a complete style. Using the mod name prefix prevents confusing the class with a full class.

|  |
| --- |
| .mod-modifier-a { ... } .mod-modifier-b { ... } |

Then, when used in the HTML, the class stands out:

|  |
| --- |
| <div class="component-content mod-modifier-a">...</div> |

This is a strong technique because the modifier classes can stand on their own in the CSS. Further, they may also be altered via more complex rules:

|  |
| --- |
| .component-header.mod-modifier-a { ... } |

This is simplistic and easy to follow, understand, and expand upon.

##### 

3.3.12 State

A state for an element or component is presentation information for a given component. This may be a dynamic state set by JavaScript or a user interaction, but not always. It could also be a preset from the server or the results after a transaction. State modifiers are a great way for a distinct class to be provided to engineers unfamiliar with the design to be provided hooks for various things. This is slightly different than JS specific classes, however.

State rules will use the is- prefix.

|  |
| --- |
| .component-group.is-full { ... }  .component-group.is-expired { ... } |

Treating components' state as a modifier that is boolean (i.e. true or false) also:

* Semantically helps provide information about the content.
* Separates the code for state from default presentation.
* Removes the need to update corresponding states or components if the name of either changes.

This last point is important from a maintenance perspective.

It's best to try to restrict these state indicators as being restricted to a specific component.

3.3.13 JavaScript (JS) Prefix

The usage of a js- prefix is present in the markup but should never really appear in the CSS file itself. If the styles are being set, then use modifiers or state type classes.

|  |
| --- |
| <button class="component-button js-execute">...</button> |

The js-execute rule should not appear in the CSS file, but only in JS files they are tied to behavior. These are events, verbs, or action related, and are access points for JavaScript not a toggle or state changer. It's best to think of these classes as closer to and ID attribute in the HTML.

##### 

3.3.14 Utilities

A utility is a type of component modifier that is specifically designed to be used on more than one component type. If it was restricted to a single component, it would simply be a modifier.

Utilities will use the prefix of u- and should serve only the modifier purpose intended without side effects when applied to any component, or component descendant.

|  |
| --- |
| .u-warning {  } .u-scroll-infobox {  overflow-x: scroll;  width: auto; } |

#### 

3.3.15 The Mobile Web, Media Queries, Responsive Design

With the mobile Web taking off Media Queries are mandatory in CSS going forwards.

We discuss media queries in the [Mobile / Responsive](#_qkr9otba9gq7) section of this document.

3.4 CSS Deliverables

Please be aware of potential conflicts between the original development environment for CSS and an ultimate deployment to production systems, if continuous integration will allow the continued use of CSS preprocessors, or if there should a cross-platform development strategy.

Delivered CSS should be concatenated, minified, tested against browser bugs (e.g. MSIE selector count bugs) and extra files should be removed.

File naming conventions should be consistent and language or use-case specific files should be clear and not be easily confused with the global style CSS.

3.5 Next Steps & CSS Resources

This is just the tip of the iceberg where CSS is concerned.

* Browser Compatibility
* Media Queries
* Accessibility and CSS
* CSS pre-processors usage
* Internet Explorer, or browser-specific bugs
* Usage of CSS3 transitions, transforms, and more
* Vendor prefixes
* Color Management

For current links and references, please see our Wiki on Github.

4. JavaScript

JavaScript is where extra behaviors, features, and functionality not offered natively by Web browsers through CSS and HTML is created.

JavaScript has gained an enormous amount of attention in recent years due to more feature rich, faster browsers and server run-times such as Node.js. For the purposes, general discussion here focuses on client-side JavaScript development, with references to where it crosses over.

It is of note that many techniques identified these days as "HTML5" are actually enabled by the related JavaScript APIs.

4.1 Goals

Unless we are talking about a complex client-side Single Page Application (SPA), JavaScript should be used sparingly, and when a deliberate choice is made to not perform a task with other available technologies. The decision to add more scripts to a Web page should be made carefully. Even with a SPA, it is critical to make controlled choices so as to not include too much unorganized impossible to maintain code.

Any and all JavaScript code that's added to a Web page should be there if and only if it is needed for the page to achieve the desired ends or if there aren't any negative impacts with it there.

Included JavaScript should:

* Be included after careful consideration.
* Have the performance overhead and file size evaluated.
* Have a feature set that is understood and appropriate.
* Perform only the necessary tasks without needless overhead.
* Have maintainability carefully assessed.

While being:

* Fast, efficient, and perform well.
* Re-usable if possible.
* Not conflict with other code on a given page or sets of pages.
* Executed only when necessary on a given page or sets of pages.

Likewise, the absence or failure of the code should be carefully considered:

* What happens if for some reason this code is missing or does not run?
* What happens if the code triggers an error?

4.2 Getting Started on JavaScript

All too often a developer will solve a problem in a closed context and not consider the whole picture. "Add another plugin" is not always a good answer.

It is well worth considering if parts or all of the code being added can be useful elsewhere. Centralized code is excellent because it can be updated once and re-used everywhere.

For JavaScript, a Front-end developer should be thinking about:

* If there is code that does this task already?
* Code formatting rules, naming conventions, file locations, etc.
* Testing the code on various browsers and devices.
* If strings of text should be external for content management or translation.
* If code being added might be useful outside of the current problem.

Examples of possible things to centralize:

* Code that modifies the DOM
* Ajax, validation, or other libraries
* Query string parsing utilities, router-type code
* Tests for global conditions (e.g. window size, feature support, etc.)
* Page, window, or document level events (e.g. Ajax, resize, etc.)
* UI controls (e.g. spinners, modals, tabs, etc.)
* Date handling utilities
* Files with strings of text in a given language
* Finally, settings and configuration options (e.g. paths to services, debug flags, duration settings, minimum or maximum values, etc.) are common things to set in a centralized, distinct place.

Bottom line, please understand what the JavaScript does and how it does it if you are including third party code.

4.2.1 JavaScript Libraries, Frameworks, and Plugins

Recent years have seen a virtual explosion in new JavaScript libraries sometimes calling themselves "frameworks".

Libraries and frameworks can be useful, especially when the client-side is become more responsible for larger parts of applications and Web sites.

* **Libraries** are code you use within your structure, featuring code that is available for you call upon.
* **Frameworks** are code collections that serve specific purposes in a particular way, and call your code that is included following their patterns.

Either way, this is typically third party code that should be carefully considered when it is determined to be included in a project or not.

Selection of Third Party Code

Selection of a library or framework is never an easy task. Things that should be considered include:

* Technical Requirements for the project.
* Quality and maturity of code in question.
* Future support for the code.
* Staffing skill sets required to support the code.
* How tightly coupled to the layers of the application the code may be.
* How actively supported its open source community may be.
* Be tested against various devices and platform requirements to verify it works for the project.

Usage of Third Party Code

Third party code should be included as-is and:

* Treated as it may be updated (i.e. versions) at some point in the future.
* Should **never be modified** unless documented thoroughly for the project.
* As many conventions of its use followed in their recommended standard ways.
* Un-minified code should be included.
* Any required licenses should be included as specified by the library.
* Commercial code must be approved if necessary.

A team may decide to write wrapper code around the third party library and provide a more simple API for the code.

4.2.2 ECMAScript 6, ES6, ECMAScript 2015

Developers are encouraged to begin learning and using the [latest version of JavaScript, ES6](https://leanpub.com/understandinges6/read/). Please use appropriate transpilers and never release untested or unsupported code in the deliverables. Do **not** assume a feature [is supported](https://kangax.github.io/compat-table/es6/) in a browser.

4.3 JavaScript Best Practices

4.3.1 Inclusion of Code

Use external JavaScript files. Do NOT include JavaScript in-line in the page unless there is a good reason.

Use the <script> tag to include your JavaScript files at the bottom of your HTML document just before the closing </body> tag. For optimal page performance, concatenate your JavaScript into as few files as possible.

|  |
| --- |
| <script src="bundle.js"></script> |

* This should link to concatenated and minified, finalized JavaScript files.
* Enable source maps to assist with debugging and testing.

In development environments, this may point at a non-optimized file, however having techniques in place to toggle optimized files on and off is often beneficial.

A reference similar to this may need to include a build-specific file name based upon a hash or something along those lines for HTTP cache purposes.

Loading Files On Demand

On some sites it may be appropriate to load a single JavaScript file with all dependencies bundled together, or it may be more appropriate (such as in a very large SPA) to load files on demand, as they are needed, asynchronously.

4.3.2 Writing and Formatting JavaScript

The use of whitespace should follow long-standing English writing conventions, with blank lines between ideas and groups of code such as objects, functions, and new lines for new statements.

Formatting the language statements and patterns should follow these basics:

* **Open braces** are preceded by a single space.
* **Open braces** should appear on the same line as their preceding argument.
* **Close braces** should appear at the same indentation as the statement preceding the opening brace
* There should be no space characters between **parentheses** and their contents.
* Use **semicolons** and do not rely on automatic semicolon insertion.
* Each **comma** and **colon** (and semi-colons that don't end a line) should be followed by a single space.
* **Binary** and **ternary operators** should have a single space on each side.
* **Quoted values** should be in 'single quotes' so that double quotes may easily exist inside them.
* **Comment JavaScript** code thoroughly and consider using a pattern such as those described by [JSDocs](http://usejsdoc.org/) so that documentation may be generated automatically.
* Conditional statements go on a new line followed by the opening brace.
* Else/else go on the same line as the brace.
* Use type strict checks with === as opposed to == whenever possible.

|  |
| --- |
| for (var i = 0, len = arr.length; i < len; i++) {  var example = 1;  if (example === i) {  // we are looping  } else {  // this will never happen  } } |

To maximize readability without worrying about which boolean operators bind more tightly than others, each segment of a boolean expression should be enclosed in parentheses.

|  |
| --- |
| if ((allowUpdate) && ((user.isAdmin) || (user.role === item.owner))) {  // do something } |

4.3.3 Variable Declaration

To avoid confusion between global and local variables, we declare each variable on its own line with the var keyword. We do not use a single var keyword and then chain several variable declarations onto it separated by a comma.

|  |
| --- |
| var windowWidth; var windowHeight; var currentVal = $(this).val(); var min = parseInt($(this).attr('min'), 10); |

This has been [debated at length](http://benalman.com/news/2012/05/multiple-var-statements-javascript/) and is controversial, however we believe this is a better practice due to several technical reasons:

* Easier debugging with debuggers.
* Easier merges with version control and diff utilities.
* Most technical issues are resolved by 'use strict'.

4.3.4 Best Practices

* Avoid user-agent sniffing and rely on [feature detection](https://isobar-us.github.io/code-standards/#javascript_feature-detection) instead. Browser detection is dangerous and error-prone.
* Avoid using document.write.
* Only run scripts on a page that are needed for that page.
* Don't repeat yourself (i.e. keep your code [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself))
* Do not modify JavaScript core objects .prototype unless you really know what you're doing.
* Use method names that make sense, such as init() or setup() for code that starts things off. Be consistent on your project.

Variable Scope

Minimize the use of global or window level variables and name-spaces. Pollution of the global name-space is error prone and a bad practice.

If referencing a window or global level variable that isn't obvious, please comment as such or explicitly state it.

|  |
| --- |
| var window.thing = {}; |

Variable Names and Types

Always use meaningful variable names that can be read as words, not as silly abbreviations only you understand.

* Variable names should be camelCase.
* Objects, classes, and name-spaces should be TitleCase.
* Boolean values should be prefixed with is if at all possible.
* Cached jQuery objects can be prefixed with $.
* Use shorthand versions of empty Arrays and Objects.

|  |
| --- |
| // some examples var exampleValue = 'my example variable value'; var numberOfTimes = 3; // booleans var isThisWorking = true; var isNotWorking = 0; // cache a selector var $body = $('body'); // short hand objects and arrays var newObject = {}; var newArray = []; |

Settings, Constants

Put settings together in obvious places such as an Object literal space inside your module. Make settings that are possibly to be considered "constants" to be obvious -- some developers like to use ALLCAPS.

Feature Detection

Always test for the existence of a browser API, function, or object property before you use it, and make sure the user experience is still functional (to the extent possible) if it's not found. We rely on JavaScript-based feature detection rather than server-side device detection because it's more robust, easily maintained, and future-proof.

Limit Events — Use Event Delegation

It is always preferable to use fewer events being bound to objects on a page as possible. Too many events bound on a page can mean memory leaks or just an accumulation of handlers bound to DOM elements which becomes less and less efficient over time. Additionally, event delegation has the added benefit of persisting events over dynamic page updates when items are added or removed from the DOM.

With jQuery this is easy, simply use the on method with a selector:

|  |
| --- |
| $('body').on('click', 'a.scroller', function(){  // this only runs if the a.scroller is matched }); |

##### 

JavaScript Performance

One of the most costly operations a browser can perform is updating the DOM in the page via inefficient JavaScript techniques. The most important thing to know is that the more you do on a Web page with JavaScript, the more work is being done, the more memory and the bigger the footprint it can generate. Additionally, updating a complex DOM structure over and over in JavaScript can cause re-flow, repainting, and jank.

A book could be written on the subject, but here's a taste of various references:

* [Minimizing browser re-flow](https://developers.google.com/speed/articles/reflow?hl=en)
* [Repaints and Reflows, Manipulating the DOM Responsibly](http://blog.letitialew.com/post/30425074101/repaints-and-reflows-manipulating-the-dom)
* [Reflows & Repaints: Css Performance Making Your Javascript Slow?](http://www.stubbornella.org/content/2009/03/27/reflows-repaints-css-performance-making-your-javascript-slow/)
* [Rendering: repaint, reflow/relayout, restyle](http://www.phpied.com/rendering-repaint-reflowrelayout-restyle/)
* [Speed Up Your JavaScript (part 4)](http://www.nczonline.net/blog/2009/02/03/speed-up-your-javascript-part-4/)
* [Memory Management and Performance](https://gist.github.com/dypsilon/4252079)
* [Writing Fast, Memory-Efficient JavaScript](http://www.smashingmagazine.com/2012/11/writing-fast-memory-efficient-javascript/)
* [Front-end developer essentials – 5 tips for efficient jQuery](http://www.punkchip.com/javascript-efficiency/)

4.3.5 Basic JavaScript Architecture

Today it is common for the JavaScript code on a site to be a vast collection of "Modules" brought together by build scripts, dependency tools, or even manually by the developer (not really recommended).

Smaller sites can get away with more simple structures, but for longer term, JavaScript-heavy code bases the following rule is critical:

* For a complex site, never use a single JavaScript file for development, unless it is tiny and serves a very targeted purpose.

This is so that the code is maintainable and scalable. Smaller files are easier to debug, swap in and out, and blocks of code should serve as small a purpose as possible ([single responsibility principle](http://blog.codinghorror.com/curlys-law-do-one-thing/)).

In most simple terms, most sites benefit from a basic structure similar to:

* Global site-wide JavaScript
* Specific modules for specific sections of the site
* Specific modules used for specific purposes / features
* Available vendor libraries

With this in mind, for strict control over the code base it's best to consider:

* What is the **central entry point**, or the central point of execution? This is to say, what kicks off the JavaScript? This may be a simple jQuery document.ready() or some other mechanism to run the site's code, such as a router.

Having explicit control over the page life-cycle is preferable to having a dozen jQuery document.ready() statements all competing for the first chance to execute on a page.

Effectively the application core, it should kick off the rest of the code to run. Typically this has module-management baked in in some way.

Additional considerations:

* How are the modules going to communicate with each other?
* How tightly coupled are the modules in the code base?
* How much JavaScript code would need to be updated if/when the HTML / CSS changes on the project?
* Can individual parts call as few libraries or plugins as indirectly as possible, to facilitate changes later?
* Does the server need to provide the scripts dynamic values for JavaScript?

About JavaScript Modules

The term "module" in JavaScript has probably been over-used. It can refer to specific patterns used by specific tools and frameworks, or simple blocks of code following some typical [JavaScript design patterns](http://www.slideshare.net/stoyan/javascript-patterns).

These days options for JavaScript modules include some of the following.

Vanilla JavaScript:

* [Object literal notation](http://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript)
* The [Revealing Module Pattern](http://addyosmani.com/resources/essentialjsdesignpatterns/book/#revealingmodulepatternjavascript)

Or, a **common standard**, used by many dependency tools:

* [AMD modules](http://requirejs.org/docs/whyamd.html) (most commonly used by [require.js](http://requirejs.org/))
* CommonJS modules (most common used by node.js and browserify)
* [ES6 modules](https://leanpub.com/understandinges6/read/#leanpub-auto-modules)

**Frameworks** will have their own unique set ups.

4.4 JavaScript Deliverables

The most obvious fact is you will need to provide working files that are error-free and will work in a variety of scenarios. We can't assume that clients will always leave scripts and pages as we leave them, though we can provide direction as to how scripts should be used.

4.4.1 Understanding the Code's Place in the Project

* Understand where your code will live vs. any code introduced in a destination environment
* Understand if the code will need to coexist with other code.

4.4.2 Clean, Clear, Organized, Readable Code

* As bug free as possible.
* Always remember we write code for other developers, not for a runtime or a browser.
* Remove code that is no longer used. Remove excess "noise" or distractions from source code such as large commented out blocks of unused code. Source control can solve problems like this.
* Have console and debugging statements be removed or a plan in place for suppression during the build or deployment.

Delivery of a flat folder full of JavaScript files is not advised.

|  |
| --- |
| ├── \_assets/ │ ├── js/ │ │ ├── app.js │ │ ├── tools.js │ │ ├── ... │ │ ├── vendor │ │ │ ├── source01.js │ │ │ ├── source02.js │ │ │ ├── ... │ │ ├── views │ │ │ ├── view01.js │ │ │ ├── view02.js │ │ │ ├── ... |

4.5 Next Steps & JavaScript Resources

There is an enormous volume of JavaScript reference material out in the wild today. We hope to add more at some point but here are various topics worth following up on:

* Debugging JavaScript
* [Learning more about ES6](https://github.com/addyosmani/es6-equivalents-in-es5)
* [JavaScript Design Patterns](http://addyosmani.com/resources/essentialjsdesignpatterns)
* Unit Testing JavaScript code
* [Node.js](https://nodejs.org/)
* Compare [JS Frameworks, Frameworks, and more Frameworks](http://todomvc.com/)
* [Baseline For Front End Developers](http://rmurphey.com/blog/2012/04/12/a-baseline-for-front-end-developers/)
* [JavaScript Style Guides and Beautifiers](http://addyosmani.com/blog/javascript-style-guides-and-beautifiers/)
* [Douglas Crockford's JS Code Conventions](http://javascript.crockford.com/code.html)
* [Maintainable JavaScript Book](http://shop.oreilly.com/product/0636920025245.do)
* [Maintainable JavaScript Presentation](http://www.slideshare.net/nzakas/maintainable-javascript-2012)
* [Large Scale JavaScript Applications](https://speakerdeck.com/addyosmani/large-scale-javascript-application-architecture)

For more current links and references, please see [our Wiki on Github](https://github.com/isobar-idev/code-standards/wiki/Useful-Links-and-Resources).

5. Responsive Web Design

Responsive Web Design (RWD) is the term used for the practice of creating page layouts and user experiences that work on a variety of devices and screen sizes.

With the ever expanding mobile landscape and the evolution of the Internet of Things (IoT), the idea of a "standard" screen size has fallen by the wayside in favor of the rise of a device-agnostic approach. It is less and less common to launch two versions of a site, one for desktop and one for mobile.

To this end a series of techniques have been put together for pages to adjust based off browsers' current specs (e.g. width, height, pixel density, orientation, etc).

The techniques are referred to as Responsive Web Design (RWD) and it is another technique in the toolbox of progressive enhancement and adaptive web design.

5.1 Goals of Responsive Web Design

The goals of RWD are:

* Offer an optimized user experience (UX),
* regardless of the user's screen size or features supported on their device.

This is true whether it is a stadium jumbotron or the screen of a watch — and of course everything between.

Despite this lofty ideal, on Web projects the term "responsive web design" has generally been used in reference to an optimized experience for a set of target devices, usually:

* A particular set of cell (smart) phones,
* various tablets,
* or desktop computer monitors (depending upon the target audience).

Typically reasonable effort is applied to accommodate the ideal UX for devices with screen sizes that fall outside of and in between.

Overall, critical content and features on a site should be:

* Adjustable to different types of user interaction (e.g. click, swipe, pinch)
* Realistically accessible based on the capabilities of different devices.

If meeting desktop browsers' feature sets and dealing with different platforms was difficult before, RWD introduces an almost infinite ecosystem of hardware and software that Web pages need to work on. The level of effort does often increase on projects featuring RWD.

5.2 Getting Started with Mobile Development

As the name of RWD implies, it all starts with a flexible design that features components having mutable characteristics based on the available screen canvas and varied types of user interactions. Careful consideration needs to be taken when innovating to derive a design that can respond to various screens.

For a given project, always consider:

* The extent of the use cases for each feature on the site.
* The target audience and likelihood they may be using an alternative device to conduct certain activities on the site.
* If all target devices will support all the technologies required.
* That the UX will **not** be 100% the same across all devices, browsers and screens — nor should it be!
* What happens to the design when the screen gets smaller and larger than the static canvas size it is being designed on.
* For placement of content and decorative elements, are there patterns or rules that can describe where it falls and adjust with screen sizes?
* Will some components be better suited as vector graphics (SVG or fonts) instead of raster files to allow for distortion free scaling?
* Will assets need to be produced in different formats for different devices (e.g. Flash video vs. HTML5 video, lower resolution artwork vs. high resolution artwork, smaller file sizes vs. larger sizes).
* If a device offers a superior UX for some types of interaction (e.g. native date-pickers vs. traditional browser controls, or swiping instead of clicking on dots).
* What happens when a feature is not supported by a device, or if a feature were to fail in a given device?
* What if a device went off-line during usage of a feature?
* The best ways to detect support for various features (e.g. Modernizr, etc.).

Brad Frost's [Responsive Design Patterns](https://bradfrost.github.io/this-is-responsive/patterns.html) and [Responsive Design Resources](https://bradfrost.github.io/this-is-responsive/resources.html) are two great resources that continue to grow due to user contributions.

5.2.1 To Use A Pre-built Grid Or Not?

Sometimes CSS grid frameworks are a good place to start ... but sometimes not.

They must match the design in terms of flexibility.

It's critical a developer learn the basics of how grid systems

While using a Framework isn't always the way to go, there is much to be learned from examining [how they work and how they are put together](https://bradfrost.github.io/this-is-responsive/resources.html#frameworks).

5.2.2 Progressive Enhancement

Since mobile phones and tablets are frequently the lower end in terms of capabilities, it is recommended to start with building the mobile experience first, and gradually add features. With this in mind, RWD could be considered a type of [Progressive Enhancement](https://www.filamentgroup.com/dwpe/), whereby users with basic devices can access basic content and features, however care is taken to layer on more sophisticated features for more powerful devices and desktop users.

With these techniques, users only get what their device or browser can handle, and feature detection can be used to add more features when appropriate — without breaking on less capable devices.

5.2.3 Setting Target Device Requirements

For a given project, check Web server logs, current industry statistics and trends, and consult with a client on their audience and proposed use cases when determining a baseline set of devices to build the site and test for.

What types of users are there? Is an audience:

* Working on their commute from a bus or train?
* Paying bills or their taxes on the go?
* Accessioning financial information with their clients on the go, over lunch?
* Entering sales figures remotely?
* Enjoying witty content or editorials?
* Comfortably browsing for fun at home?

It is near impossible to test on every version of every platform, particularly with something like Android where fragmentation of the feature set is totally unpredictable. Comparison of platforms and statistics regarding particular releases of particular versions of an operating system may need to be considered.

5.3 Responsive Design Best Practices

RWD is frequently said to be achieved through the use of:

* Percentage-based grids,
* flexible images that scale,
* and CSS media queries.

These are the core ideas behind RWD, though other techniques are often employed as the term grows in popularity and the use cases evolve.

Like all programming and creative processes there is flexibility in how you go about achieving these, so long as there is consistency within your team.

Some core guidelines:

* Always design and develop the **mobile UX first** (thus start with small defaults and scale upwards; please see Mobile First)
* Set the **baseline Media Queries as a team** for various types of devices, and try to stick to them as much as possible.
* **Add additional Media Queries** — **as necessary** — for less than ideal experiences at the sizes in between, as appropriate for different types of content and components.
* **Build for speed** (load time and interaction) and the minimum viable product for slower connection speeds and less able devices.
* Be careful of **images**. Do not render a 5GB image at 100px by 100px. This does not change the bandwidth needed to download the file! Similarly if you are hiding an image with display: none, visibility: hidden or similar, the image may still be downloaded.
* Test on **real hardware**. Actual testing on actual devices cannot be substituted with resizing a browser window, or even an "emulation" mode offered by a desktop browser.
* A **mobile browser** is not a small desktop browser. Mobile versions of desktop browsers are frequently woefully different (or a totally different product) than their desktop counterparts, so do not assume that just because it works on Chrome desktop that it will work on an Android device.

Initial work, and first pass tests of media queries, may be performed in desktop browsers by resizing the screen or using a browser's developer tools that may feature an emulation mode (e.g. Chrome DevTools).

However, this is never a substitution for actual testing on actual devices with various Operating System versions and

5.3.1 Media Queries and CSS Breakpoints

Added as part of the CSS3 specification, **media queries** consist of a media type and at least one expression of a media feature (such as height, width and orientation) that describe the conditions under which a set of CSS rules apply.

For example:

|  |
| --- |
| @media screen (min-width: 300px) and (max-width: 800px) {  .some-selector{  /\* CSS declarations go here \*/  } } |

These widths are referred to as breakpoints as they are the point at which layout behavior breaks with the design at the prior size. This is typically screen size, and sometimes other conditions such as pixel density or even screen orientation (e.g. landscape or portrait; although this isn't that useful).

These are applied with CSS Media Queries test the device or browsers current feature set for various conditions and feature support. The most common example is screen size.

The golden rule is:

* Don’t specify vendor- or device-specific widths

Let the content and the design dictate the breakpoints.

* For instance, targeting an ipad in portrait or an ipad landscape.
* Popular devices will come and go and even within devices the specifications will change over time.
* Avoid orientation/resolution based specifications in favor of width based ones (remember you are **not** targeting devices!).

[Media Query Resources](https://bradfrost.github.io/this-is-responsive/resources.html#media-queries)

[A Device Agnostic Approach to Responsive Web Design](http://www.smashingmagazine.com/2012/03/device-agnostic-approach-to-responsive-web-design/) is a great read on responsive techniques that do not favor one device over another, and are driven by content elements.

If you are placing breakpoints every 50-100px you are doing something wrong. While there is not a correct number of breakpoints there must be a balance between the control of the design and a manageable code base.

While working with media queries here are a few things to consider:

* All modern browsers support CSS3 Media Queries (including browsers as far back as the stock Android 2.1 browser)
* IE8 and below do not support Media Queries.
* [Respond.js](https://github.com/scottjehl/Respond) (and similar JavaScript libraries) can be used to enable support for Media Queries in these older browsers (if needed).
* Attempting to polyfill Media Query behavior in older browsers is usually not advisable. This is because the JS needed will cause an additional performance hit to the already slow JavaScript and rendering engine.

Instead, you are better off forcing older browsers to experience the site through a desktop experience (optionally tailored specifically for these older browsers).

* This can be done either through feature detection (with the likes of Modernizr) or with IE conditional statements and optionally including additional CSS style sheets.

5.3.2 Mobile First

The "Mobile First" techniques consider the lowest common denominator first, which is likely to be your mobile devices due to bandwidth limitations, loading times (not just screen size!), and even CPU speed of the devices.

Key considerations and techniques include:

* Starting small, and work upwards. Use min-width, **not** max-width breakpoint definitions).
* Begin with global content styles that apply across all breakpoints.
* Next, add the styles that are seen below the smallest break point -- remember that using the min-width approach means that the "first breakpoint" will not be for small screens (like mobile size screens) but rather for larger ones (like mobile landscape or phablet size screens).
* From here add additional styles at successive breakpoints using min-width media queries leveraging the CSS cascade to progressively add additional styles.
* As the queries increase in minimum sizes, add the markup and styles necessary to lay components out differently on larger screens.
* Examine site features for mobile-specific JavaScript events (e.g. touch, pointer, vs. mouse) and interactions (e.g. swipe).

Generally speaking smaller screen designs are a bit less complex (fewer images, effects, etc). Thus as the screens get larger the complexity tends to increase, conveniently lending to an additive approach to styling, very much in line with a mobile first pattern.

An example of working upwards in a CSS file might be:

|  |
| --- |
| /\* GENERAL STYLES \*/ .example {...}  /\* SECTION SPECIFIC STYLES - aimed at the smallest devices\*/ .hero {...}  @media(min-width: 600px) {  .hero {  /\* just the new stuff here, no need to be repetitive... \*/   }  }  @media(min-width: 800px) {  .hero {  /\* just the new stuff here, no need to be repetitive... \*/   }  }  @media(min-width: 1400px) {  .hero {  /\* might need larger font sizes at largest screen sizes \*/   }  } |

Note: the breakpoints specified above are **not** intended to be recommended breakpoints, but rather just illustrative of this general concept.

Another way of thinking about a mobile first pattern (from a design and development standpoint) is to:

* Innovate for your smallest reasonable target device.
* Then start "sizing your window up" until the user experience or the design degrades.
* Add in a breakpoint and make some design modifications to fix the degradation.
* Resume sizing up your view port until you reach a similar breakdown in design and repeat this exercise.

Recurse on this until you reach your largest reasonable target device - this is the mobile first pattern.

5.3.3 Speed is a feature

People expect page load time to be as fast if not faster on their mobile phones in comparison to a desktop experience.

* Try to keep your Web site's foot print as small as possible (this is a general best practice).
* Start with global styles that apply across all breakpoints
* Optionally in-line these vital styles for faster initial render time
* Consider every HTTP request
* The request itself may end up taking as much if not more time then the transfer of the data and could block downstream actions or more vital downloads.
* Concatenate, gzip and minified your production CSS/JS where possible as this will lower page load size and time
* Load the least amount of JavaScript that is needed.
* Wherever possible include script files at the end of your HTML document just before the </body> tag.

Advanced CSS3 techniques are easy to implement, but when you start combining them, rendering and scrolling performance can be affected.

Be careful of images!

* Use a responsive image pattern that starts by serving a mobile optimized (and thus smaller file size) image first.

5.3.4 Responsive Images

Currently there is little to no native support for responsive images.

The proposal with the most traction appears to be the <picture> element and related srcset and sizes attributes.

You can read about this element and the evolution and standard at [ResponsiveImages.org](http://responsiveimages.org/)

At this time the best application of the <picture> is with the [Picturefill polyfill](https://scottjehl.github.io/picturefill/).

It can be cumbersome to set up, so take care and be certain the benefits are worth it on a given site.

Some basic rules to follow:

* **Always** optimize your images using a tool such as Adobe Photoshop to assure you have the ideal image size and right amount of lossy compression.
* Run all images though a lossless compressor like [Compressor.io](https://compressor.io/), [Smush.it](https://smush.it/) or [ImageOptim](https://imageoptim.com/) (which has related tools also available as build-time scripts).

You can also consider “Compressive Images”, which are basically higher-resolution jpegs compressed at a higher percentage rate:

* [Compressive Images](http://filamentgroup.com/lab/rwd_img_compression/)
* [Retina Revolution](http://blog.netvlies.nl/design-interactie/retina-revolution/)
* [Compressive Image Tests](http://www.vanseodesign.com/web-design/compressive-image-tests/)

However, there are drawbacks:

* The browser tends to use significantly more memory when storing and resizing these higher resolution images than scaling them to fit a container

What’s important to realize is that this is changing faster than we can develop. What you do now will be obsolete very soon, so be sure to stay on top of current trends.

5.3.5 Vector Graphics (SVG)

When working with an audience on unknown screen sizes and resolutions, as is the pretense in RWD, having graphics that can scale without degrading is a very appealing prospect. There are several vector implementation options.

Web fonts: Fonts are vectors. With the exposure of custom web fonts as part of the CSS level 2 specification, many developers have turned to custom font packages as a solution for icons and simple vectors.

* Pros: this option allows for easy control of vector color, size and usage
* Cons: all of the vectors single color and must be grouped others in one file.

SVG: SVG is an XML syntax for describing vector shapes.

* Pros: vectors can be manipulated by CSS and/or JavaScript and allows for complex filters, animations and transitions.
* Cons: SVG is only supported by IE9+

[Grumpicon](http://www.filamentgroup.com/lab/grumpicon-workflow.html) is a script that will detect SVG support and server the appropriate CSS (or needed fall back CSS code to assure compatibility.

5.3.6 Other Categories

Virtually everything can change when screen sizes change:

* Grids
* Typography
* Accessible forms / validation messages
* Main site navigation
* Header or Footer content
* In-page navigation, such as tabbed navigation, or accordions
* Images
* Data tables
* Interactions and animations
* Advertising types
* Performance metrics

More thoughts and ideas: [Multi-device Layout Patterns](http://www.lukew.com/ff/entry.asp?1514)

Navigation Changes

Large horizontal navigation and mega-drop downs are not always practical on smaller screens. Frequently this means off-canvas navigation, collapsing menus into select boxes, or other approaches such as a full-screen overlay.

[Progressive and Responsive Navigation](http://www.smashingmagazine.com/2012/02/progressive-and-responsive-navigation/) is a great example of modifying the types of menu based on screen size.

[Responsive Navigation Patterns](http://bradfrost.com/blog/web/responsive-nav-patterns/) and [Complex Navigation Patterns](http://bradfrost.com/blog/web/complex-navigation-patterns-for-responsive-design/)

How navigation changes per screen size will depend entirely on the requirements and design for the project.

Responsive Data Tables

Displaying data tables in a responsive way on Mobile sites is a challenge. There are only so many good ways to address this.

* Simply make a wrapper around the table and scroll horizontally.
* Have CSS rules which collapse the tables.
* Sometimes JavaScript might be used to hide/show certain less critical columns, however this does add overhead.

6. ReactJS

6.1 Keep components small and function-specific

As we all know, with React, it’s possible to have huge components that execute a number of tasks. But a better way to design components is to keep them small, so that one component corresponds to one function. Ideally, a single component should render a specific bit of your page or modify a particular behavior. There are many advantages to this:

* Function-specific components can be standalone, which makes [testing](https://www.codeinwp.com/blog/best-cross-browser-testing-tools/) and maintenance easier.
* Each small component can be reused across multiple projects.
* Components executing general functions can be made available to the community.
* With smaller components, it’s easier to implement performance optimizations.
* It’s easier to update smaller components.
* Bigger components have to perform harder and may be difficult to maintain.

The balance between creating one concise component and creating multiple function-specific components can vary from organization to organization. After all, you can have as many components as you want, and recombine them in any way you want to achieve the same end result.

6.2 Reusability is important, so keep creation of new components to the minimum required

By sticking to the rule of one function = one component, you can improve the reusability of components. What this means is that you should skip trying to build a new component for a function if there already exists a component for that function.

By reusing components across your project or across any number of projects, not only will you achieve consistency, you’ll also be contributing to the community.

On the other hand, if any component becomes huge, unwieldy and difficult to maintain, it’s better to break it up into as many smaller components as required.

For example, you can even go further and create a Button component that can handle icons. Then, each time you need a button, you’ll have a component to use. Making it more modular will allow you to cover many cases with the same piece of code. You have to aim somewhere in the middle. Your components should be abstract enough, but shouldn’t be overly complex.

|  |
| --- |
| class IconButton extends React.Component {   [...]   render() {     return (       <button onClick={this.props.onClick()}>         <i class={this.props.iconClass}></i>       </button>     );   } } |

## 

6.3 Consolidate duplicate code – DRY your code

A common rule for all code is to keep it as brief and concise as possible.

It’s no different here too, since React best practices also instruct you to keep code brief and precise. One way to do this is to avoid duplication – Don’t Repeat Yourself (DRY).

You can achieve this by scrutinizing the code for patterns and similarities. If you find any, it’s possible you’re repeating code and there’s scope to eliminate duplication. Most likely, a bit of rewriting can make it more concise.

This relies heavily on the reusability principle in React. Let’s say you want to add multiple buttons that contain icons, instead of adding the markup for each button, you can simply use the IconButton component that we shown above. You could even go further by mapping everything into an array.

|  |
| --- |
| const buttons = ['facebook', 'twitter', 'youtube'];  return (   <div>     {       buttons.map( (button) => {         return (           <IconButton             onClick={doStuff( button )}             iconClass={button}           />         );       } )     }   </div> ); |

6.4 Comment only where necessary

Attach comments to code only where necessary. This is not only in keeping with React best practices, it also serves two purposes at the same time:

* It’ll keep code visually clutter free.
* You’ll avoid a potential conflict between comment and code, if you happen to alter the code at some later point in time.

6.5 Name the component after the function

It’s a good idea to name a component after the function that it executes so that it’s easily recognizable.

For example, ProductTable – it conveys instantly what the component does. On the other hand, if you name the component based on the need for the code, it can confuse you at a future point of time.

Another example, it’s preferable to name a component Avatar so that it can be used anywhere – for authors, users or in comments. Instead, if we name the component AuthorAvatar in the context of its usage, we’d be limiting the utility of that component.

Besides, naming a component after the function makes it more useful to the community as it’s more likely to be discovered.

6.6 Use capitals for component names

If, like most folks, you’re using JSX (a JavaScript extension), the names of the components you create need to begin with uppercase letters. For instance, you’ll name components as SelectButton instead of selectbutton, or Menu instead of menu. We do this so that JSX can identify them differently from default HTML tags.

Earlier React versions maintained a list of all built-in names to differentiate them from custom names. But as the list needed constant updating, that was scrapped and capitals became the norm.

In case JSX is not your language of choice, you can use lowercase letters. However, this may reduce the reusability of components beyond your project.

6.7 Mind the other naming conventions

When working with React, you are generally using JSX (JavaScript Extension) files. Any component that you create for React should therefore be named in Pascal case, or upper camel case. This translates to names without spaces and the capitalizing the first letter of every word.

If you want to create a function that submits a form, you should name it SubmitForm in upper camel case, rather than submitForm, submit\_form, or submit\_form. Upper camel case is more commonly called Pascal case. Here is a further [list of examples](https://techterms.com/definition/pascalcase) of variable and function names in Pascal case.

6.8 Separate stateful aspects from rendering

Components in [React](https://www.codeinwp.com/blog/react-ui-component-libraries-frameworks/) can be stateful or stateless. Stateful components store information about the component’s state and provide the necessary context. In contrast, stateless components have no memory and cannot give context to other parts of the UI. They only receive props (inputs) from parent component and return you JSX elements. They are scalable and reusable, and similar to pure function in JavaScript.

One of React best practices is to keep your stateful data-loading logic separate from your rendering stateless logic. It’s better to have one stateful component to load data and another stateless component to display that data. This reduces the complexity of the components.

The later React versions v16.8 have a new feature – React Hooks, which write stateful function-related components. This may eventually eliminate the need for class-based components.

For example, your app is fetching some data on mount. What you want to do is manage the data in the main component and pass the complex [render task](https://www.codeinwp.com/blog/free-react-native-templates/) to a sub-component as props.

|  |
| --- |
| import RenderTable from './RenderTable';  class Table extends Component {   state = { loading: true };    render() {     const { loading, tableData } = this.state;     return loading ? <Loading/> : <RenderTable data={tableData}/>;   }    componentDidMount() {     fetchTableData().then( tableData => {       this.setState( { loading: false, tableData } );     } );   } } |

## 

6.9 Code should execute as expected and be testable

Really, this rule needs no explanation. The code you write should behave as expected, and be testable easily and quickly. It’s a good practice to name your test files identical to the source files with a .test suffix. It’ll then be easy to find the test files.

You can use JEST to test your React code.

6.10 All files related to any one component should be in a single folder

Keep all files relating to any one component in a single folder, including styling files.

If there’s any small component used by a particular component only, it makes sense to keep these smaller components all together within that component folder. The hierarchy will then be easy to understand – large components have their own folder and all their smaller parts are split into sub-folders. This way, you can easily extract code to any other project or even modify the code whenever you want.

For instance, for the Form component, all pieces such as CSS files, icons, images, tests and any other sub-components relating to *Form* should all reside in the same folder. If you name files sensibly, and keep related files together logically, you’ll not have any difficulty finding them later.

6.11 Use tools like Bit

One of React best practices that helps to organize all your React components is the use of tools like [Bit](https://bit.dev/).

These tools help to maintain and reuse code. Beyond that, it helps code to become discoverable, and promotes team collaboration in building components. Also, code can be synced across projects.

6.12 Use snippet libraries

Code snippets help you to keep up with the best and most recent syntax. They also help to keep your code relatively bug free, so this is one of the React best practices that you should not miss out on.

There are many snippet libraries that you can use, like, ES7 React, Redux, JS Snippets, etc.

6.13 Write tests for all code

In any programming language, adequate testing ensures that any new code added to your project integrates well with existing code and does not break existing functionality. It is a good idea to write tests for any new component that you create. As a good practice, you should create a \_\_Test\_\_ directory within your component’s directory to house all relevant tests.

You can broadly divide tests in React into two parts: testing the functionality of components using a React app, and tests on your complete application once it renders in the browser. You can use [cross browser testing tools](https://www.codeinwp.com/blog/best-cross-browser-testing-tools/) for tests in the latter category.

For the former, you can use a JavaScript test runner, [Jest](https://jestjs.io/) to emulate the HTML DOM using jsdom to test React components. While a completely accurate test is only possible in a browser on a real device, Jest provides a good approximation of the real testing environment during the development phase of your project.

7. PWA

Technically speaking, PWA is a website with a manifest. Conceptually, there’s much more! It’s the way (the only way) to save the website on the home screen of a mobile device without being listed in the app store of the platform. Not to mention it’s for free! Bearing that in mind, let’s see what’s the necessary minimum of 4 elements that a PWA should cover to fulfill basic requirements and practices, and let’s outline a checklist:

**7.1 Web app manifest file: manifest.json**

Web manifest file is a JSON container of meta-data regarding your website, that you put at the root of your web app. It gathers all the information that’s needed to configure the appearance of a PWA. The most crucial details are – icons, theme color, name, short name and display as standalone (as a PWA). Thanks to that, a device knows how a PWA will look like after installation or adding to the home screen, and how it will behave after launching.

The JSON file contains really basic data and is very simple. For example, it may look like this:

|  |
| --- |
| {   "short\_name": "Maps",   "name": "Google Maps",   "icons": [     {       "src": "/images/icons-192.png",       "type": "image/png",       "sizes": "192x192"     },     {       "src": "/images/icons-512.png",       "type": "image/png",       "sizes": "512x512"     }   ],   "start\_url": "/maps/?source=pwa",   "background\_color": "#3367D6",   "display": "standalone",   "scope": "/maps/",   "theme\_color": "#3367D6" } |

**7.2 Service Worker**

A service worker is in fact a JavaScript file, sitting between the browser and the server. It is a major component of a PWA that you put at the root of the web application, as it handles requests and sends responses in the background on behalf of both the browser and the server. The idea behind that is to provide offline capabilities and access to PWA’s features by caching single web pages of your app. Thanks to a service worker, users don’t have to wait until the data loads when they open an app next time. It also enables the implementation of push notifications.

**7.3 HTTPS**

It is a requirement of a service worker to serve over HTTPS, so it’s a must for a PWA to include SSL certification on a webserver. The most popular (not to say easy and cost-free) method is via Let’s Encrypt.

**7.4 App icon**

There’s no app without an icon, and a PWA is not an exemption here. Not to mention a manifest should have that covered. The icon will be used to launch a PWA from the home screen. The easiest way around this would be a single 512 x 512 px icon, but a PWA can have a vast variety of differently sized icons.

8. SEO

8.1 Principles of search engine optimization

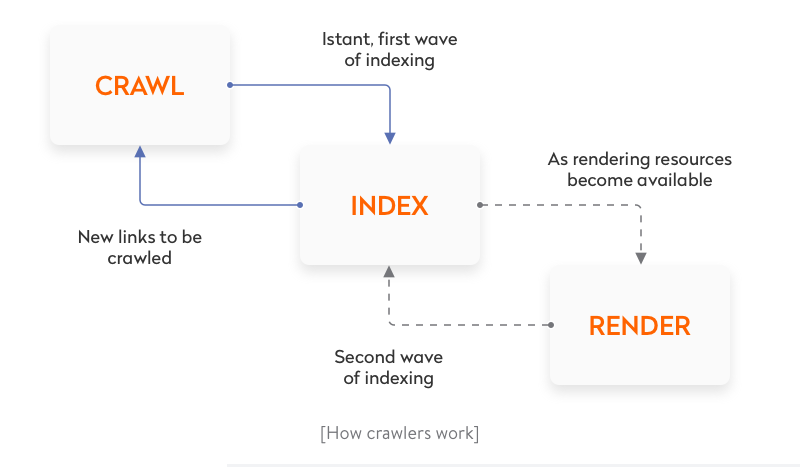
Search engine optimization (SEO) is the practice of increasing the quality and quantity of web traffic to a website by increasing its visibility on search engines.

The main task of SEO is to help a web app rank in search engines (Google, Bing) when the target audience searches for information by a certain keyword. Most SEO efforts focus on optimizing apps for Google. After all, as of 2020 Google has a 92.16% search engine market share according to StatCounter.

According to Backlinko, 75% of all user clicks go to the top three websites in search engine results, while websites from the second results page and beyond get only 0.78% of clicks. That’s why digital businesses are furiously fighting to get on this first page of search results.

For business owners, it’s crucial to think of an SEO optimization strategy from the very beginning of web app development to find an effective technology stack.

To determine a website’s ranking in search results, search engines use web crawlers. A web crawler is a bot that regularly visits web pages and analyzes them according to specific criteria set by the search engine. Each search engine has its own crawler, and Google’s crawler is called the Googlebot.



The Googlebot explores pages link by link, gathers information on website freshness, content uniqueness, number of backlinks, etc., downloads HTML and CSS files, and sends all of this data to Google servers. Then it’s analyzed and indexed by a system called Caffeine. This is a fully automatic process, so it’s vital to ensure that crawlers correctly understand the website content. And here’s where the problem appears.

8.2 What’s wrong with optimizing single-page applications (SPA) for search engines?

Typically, single-page apps load pages on the client side: at first, the page is an empty container; then JavaScript pushes content to this container. The simplest HTML document for a React app might look as follows:

|  |
| --- |
| <html lang="en"> <head>   <title>React App</title> </head>   <body>     <noscript>You need to enable JavaScript to run this app.</noscript>     <div id="root"></div>     <script src="/static/js/bundle.js"></script>   </body> </html> |

As you can see, there’s nothing except the <div> tag and an external script. Single-page applications need a browser to run a script, and only after the script is run will the content be dynamically loaded to the web page. So when a crawler visits the website, it sees an empty page without content. Hence, the page cannot be indexed.

In autumn 2015, [Google announced](https://webmasters.googleblog.com/2015/10/deprecating-our-ajax-crawling-scheme.html) that their bots would also inspect the JavaScript and CSS of web pages, so they would be able to render and understand web pages like browsers do. That was great news, but there are still problems with SEO optimization:

* Long delays

If the content on a page updates frequently, crawlers should regularly revisit the page. This can cause problems, since reindexing may only be done a week later after the content is updated, as Google Chrome developer Paul Kinlan[reports on Twitter](https://mobile.twitter.com/Paul_Kinlan/status/1039852756113080320).

This happens because the Google Web Rendering Service (WRS) enters the game. After a bot has downloaded HTML, CSS, and JavaScript files, the WRS runs the JavaScript code, fetches data from APIs, and only after that sends the data to Google’s servers.

* Limited crawling budget

The crawl budget is the maximum number of pages on your website that a crawler can process in a certain period of time. Once that time is up, the bot leaves the site no matter how many pages it’s downloaded (whether that’s 26, 3, or 0). If each page takes too long to load because of running scripts, the bot will simply leave your website before indexing it.

Talking about other search engines, Yahoo’s and Bing’s crawlers will still see an empty page instead of dynamically loaded content. So getting your React-based SPA to rank at the top on these search engines is a will-o'-the-wisp.

You should think of how to solve this problem on the stage of designing app architecture.

8.3 Solving the problem

There are a couple of ways to make a React app SEO-friendly: by creating an isomorphic React app

8.4 Isomorphic React apps

In plain English, an isomorphic JavaScript application (or in our case, an isomorphic React application) can run on both the client side and the server side.

Thanks to isomorphic JavaScript, you can run the React app and capture the rendered HTML file that’s normally rendered by the browser. This HTML file can then be served to everyone who requests the site (including Googlebot).

On the client side, the app can use this HTML as a base and continue operating on it in the browser as if it had been rendered by the browser. When needed, additional data is added using JavaScript, as an isomorphic app is still dynamic.

An isomorphic app defines whether the client is able to run scripts or not. When JavaScript is turned off, the code is rendered on the server, so a browser or bot gets all meta tags and content in HTML and CSS.

When JavaScript is on, only the first page is rendered on the server, so the browser gets HTML, CSS, and JavaScript files. Then JavaScript starts running and the rest of the content is loaded dynamically. Thanks to this, the first screen is displayed faster, the app is compatible with older browsers, and user interactions are smoother in contrast to when websites are rendered on the client side.

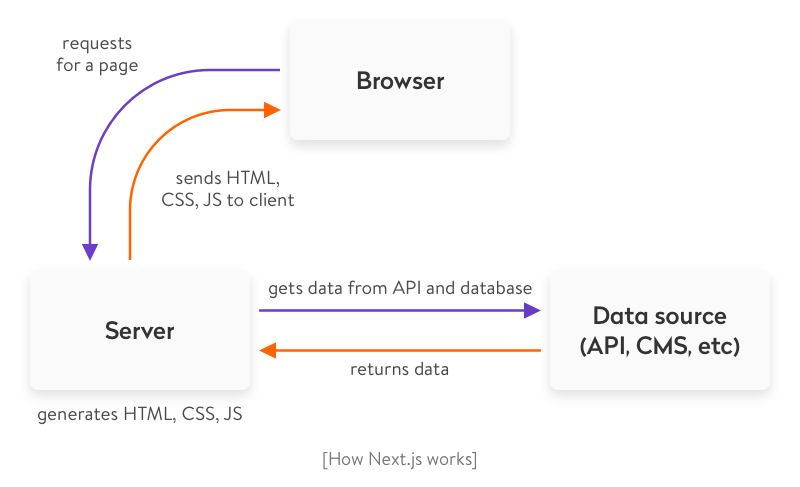
Building an isomorphic app can be really[time-consuming](https://reactjsnews.com/isomorphic-react-in-real-life). Luckily, there are frameworks that facilitate this process. The two most popular solutions for SEO are Next.js.

[Next.js](https://nextjs.org/) is a framework that helps you create React apps that are generated on the server side quickly and without hassle. It also allows for automatic code splitting and hot code reloading. Next.js can do full-fledged server-side rendering, meaning HTML is generated for each request right when the request is made.

8.5 Server-side rendering with Next.js

The Next.js rendering algorithm looks as follows:

* The Next.js server, running on Node.js, receives a request and matches it with a certain page (a React component) using a URL address.
* The page can request data from an API or database, and the server will wait for this data.
* The Next.js app generates HTML and CSS based on the received data and existing React components.
* The server sends a response with HTML, CSS, and JavaScript.



8.6 The bottom line

Single-page React applications offer exceptional performance, seamless interactions close to those of native applications, a lighter server payload, and ease of web development.

Challenges with SEO shouldn’t be a reason for you to avoid using the React library. Instead, you can use the above-mentioned solutions to fight this issue. Moreover, search engine crawlers are getting smarter every year, so in the future, SEO optimization may no longer be a pitfall of using React.

9. File Structure

my-app

├── assets

       ├── images

       ├── scripts

              └── vendors

       └── styles

              └── vendors

└── src

    ├── components

    ├── constants

    ├── models

    ├── routes

    ├── services

    ├── views

    ├── utilities

    ├── index.css

    ├── index.js

    └── serviceWorker.js