Approach

Learn about the guiding principles, strategies, and techniques used to build and maintain Bootstrap so you can more easily customize and extend it yourself.

While the getting started pages provide an introductory tour of the project and what it offers, this document focuses on *why* we do the things we do in Bootstrap. It explains our philosophy to building on the web so that others can learn from us, contribute with us, and help us improve.

See something that doesn't sound right, or perhaps could be done better? Open an issue—we'd love to discuss it with you.

Summary

We'll dive into each of these more throughout, but at a high level, here's what guides our approach.

- Components should be responsive and mobile-first
- Components should be built with a base class and extended via modifier classes
- Component states should obey a common z-index scale
- Whenever possible, prefer a HTML and CSS implementation over JavaScript
- Whenever possible, use utilities over custom styles
- Whenever possible, avoid enforcing strict HTML requirements (children selectors)

Responsive

Bootstrap's responsive styles are built to be responsive, an approach that's often referred to as *mobile-first*. We use this term in our docs and largely agree with it, but at times it can be too broad. While not every component *must* be entirely responsive in Bootstrap, this responsive approach is about reducing CSS overrides by pushing you to add styles as the viewport becomes larger.

Across Bootstrap, you'll see this most clearly in our media queries. In most cases, we use `min-width` queries that begin to apply at a specific breakpoint and carry up through the higher breakpoints. For example, a `.d-none` applies from `min-width: 0` to infinity. On the other hand, a `.d-md-none` applies from the medium breakpoint and up.

At times we'll use `max-width` when a component's inherent complexity requires it. At times, these overrides are functionally and mentally clearer to implement and support than rewriting core functionality from our components. We strive to limit this approach, but will use it from time to time.

Classes

Aside from our Reboot, a cross-browser normalization stylesheet, all our styles aim to use classes as selectors. This means steering clear of type selectors (e.g., `input[type="text"]`) and extraneous parent classes (e.g., `.parent .child`) that make styles too specific to easily override.

As such, components should be built with a base class that houses common, not-to-be overridden property-value pairs. For example, `.btn` and `.btn-primary`. We use `.btn` for all the common styles like `display`, `padding`, and `border-width`. We then use modifiers like `.btn-primary` to add the color, background-color, border-color, etc.

Modifier classes should only be used when there are multiple properties or values to be changed across multiple variants. Modifiers are not always necessary, so be sure you're actually saving lines of code and preventing unnecessary overrides when creating them. Good examples of modifiers are our theme color classes and size variants.

z-index scales

There are two `z-index` scales in Bootstrap—elements within a component and overlay components.

Component elements

- Some components in Bootstrap are built with overlapping elements to prevent double borders without modifying the `border` property. For example, button groups, input groups, and pagination.
- These components share a standard `z-index` scale of `0` through `3`.
- `0` is default (initial), `1` is `:hover`, `2` is `:active`/`.active`, and `3` is `:focus`.
- This approach matches our expectations of highest user priority. If an element is focused, it's in view and at the user's attention. Active elements are second highest because they indicate state. Hover is third highest because it indicates user intent, but nearly *anything* can be hovered.

Overlay components

Bootstrap includes several components that function as an overlay of some kind. This includes, in order of highest `z-index`, dropdowns, fixed and sticky navbars, modals, tooltips, and popovers. These components have their own `z-index` scale that begins at `1000`. This starting number was chosen arbitrarily and serves as a small buffer between our styles and your project's custom styles.

Each overlay component increases its `z-index` value slightly in such a way that common UI principles allow user focused or hovered elements to remain in view at all times. For example, a modal is document blocking (e.g., you cannot take any other action save for the modal's action), so we put that above our navbars.

Learn more about this in our `z-index` layout page.

HTML and CSS over JS

Whenever possible, we prefer to write HTML and CSS over JavaScript. In general, HTML and CSS are more prolific and accessible to more people of all different experience levels. HTML and CSS are also faster in your browser than JavaScript, and your browser generally provides a great deal of functionality for you.

This principle is our first-class JavaScript API using `data` attributes. You don't need to write nearly any JavaScript to use our JavaScript plugins; instead, write HTML. Read more about this in our JavaScript overview page.

Lastly, our styles build on the fundamental behaviors of common web elements. Whenever possible, we prefer to use what the browser provides. For example, you can put a `.btn` class on nearly any element, but most elements don't provide any semantic value or browser functionality. So instead, we use `<button>`s and `<a>`s.

The same goes for more complex components. While we *could* write our own form validation plugin to add classes to a parent element based on an input's state, thereby allowing us to style the text say red, we prefer using the `:valid`/`:invalid` pseudo-elements every browser provides us.

Utilities

Utility classes—formerly helpers in Bootstrap 3—are a powerful ally in combatting CSS bloat and poor page performance. A utility class is typically a single, immutable property-value pairing expressed as a class (e.g., `.d-block` represents `display: block;`). Their primary appeal is speed of use while writing HTML and limiting the amount of custom CSS you have to write.

Specifically regarding custom CSS, utilities can help combat increasing file size by reducing your most commonly repeated property-value pairs into single classes. This can have a dramatic effect at scale in your projects.

Flexible HTML

While not always possible, we strive to avoid being overly dogmatic in our HTML requirements for components. Thus, we focus on single classes in our CSS selectors and try to avoid immediate children selectors (`>`). This gives you more flexibility in your implementation and helps keep our CSS simpler and less specific.