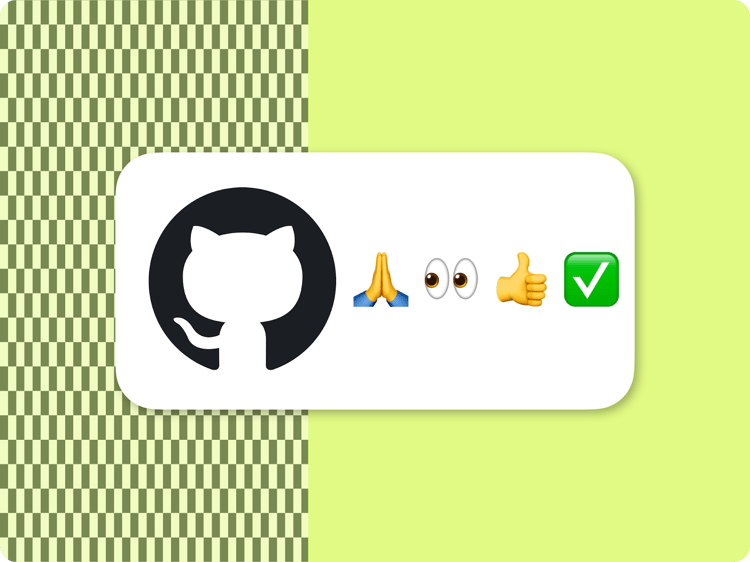
| **Section Metadata** | |
| --- | --- |
| style | content |



**Development Collaboration and Good Practices**

Working with a large number of development teams across many projects and organizations, we found that it is useful to collect some of our insights. Some of those are related to AEM, but the majority are related to general purpose frontend development or are just general guidelines on how to collaborate in a team of developers.

You may read some of those items and think that it is generally understood as common sense amongst developers. We agree, and that’s a great sign that you are ready to work in a collaborative way on AEM projects together with other developers.

At this point this is just a collection of lessons learned from our engagements on a growing set of projects.

**GitHub**

**Keep the repository public**

Given that all project code is sent to the client-side and therefore accessible in everyone's browser, we recommend keeping the repository as public. Adobe has a longstanding history in the open-source community and generally advocates for code to be public unless there is a compelling reason to make it private.

Maintaining a public repository offers several benefits of open source projects like community code reviews, and developers are able to share knowledge and code, in turn fostering innovation and collaboration. Public repositories also make it quick and easy for Adobe and other developers to create free pull requests to help improve your project and automatic [PSI checks](https://www.aem.live/developer/keeping-it-100) (on the feature branches) are provided by Adobe, which are not enabled by default for private repositories.

If you still have a strong need to keep your repository private, be aware that certain features are only available in paid plans for private repositories, such as branch protection rules and limited CI/CD minutes. For more details, please see [GitHub’s pricing page](https://github.com/pricing).

**Create pull requests**

If you work on a project with multiple developers it is rarely a good idea to push directly to main. When your project is in production code changes that are merged or pushed to main often means that they are released to production. Protecting your main branch is a good mechanism to ensure that people don’t push to main by accident, which is especially advisable with a site that is in production.

**Pull request etiquette**

If you open a pull request, make sure that you include a URL to a page (or a number of links to pages) on your branch, where the reviewer can see your code in action. If you are updating code of an existing block, make sure to include the link that features the block you are updating as the reviewer may not know where this block is in use to test its functionality.

Keep the scope of your Pull Request to what’s in the title/description of the PR.

For work still in progress, opening a Draft Pull Request helps prevent wasting people's time with reviewing code still in flux as well as accidental merging.

**Linting**

The standard boilerplate setup runs the linting tools eslint (airbnb-base) and stylelint for each change in a Pull Request. Do not submit a Pull Request with linting errors for review.

Changing the linting configuration is not recommended unless you have a really good reason. Personal preference is not a good reason. Changing linting rules makes it difficult to reuse code from the AEM boilerplate, [block collection](/developer/block-collection) or other open source AEM projects. Arguing if something is a good reason to change the linting rules is usually a lot more effort than just categorically saying no.

**PageSpeed Insights**

The AEM Code Sync GitHub app runs [Google PageSpeed Insights](https://pagespeed.web.dev/) for each change in a pull request to assess the impact on Web Performance, Accessibility, SEO and Best Practices. Do not submit a pull request for review that doesn’t have a green Lighthouse Score for mobile and desktop, [ideally 100 for both](/developer/keeping-it-100).

**Reviews**

It is good practice to have your code reviewed by the maintainer or main developer of the project you are working on. This can be encouraged by setting up branch protection on main to require pull requests with at least 1 approval before any code can be merged. You can still allow project administrators to bypass branch protection settings for emergencies.

**Shared branches**

It is good practice to consider branches for individual Pull Requests as private to the developer who created the branch. Do not just push into other developers’ branches without having been invited to do so. There are situations where people are collaborating on Pull Requests but it should be an explicit agreement.

**Merging pull requests (deploying to production)**

Merge your own pull requests only. If the person who opened the Pull Request has the ability to merge their own Pull Request, the author of the code is the ideal person to merge. There are situations where the author specifically states that this can and should be merged by someone else, and in those cases a maintainer (main developer) of the project should feel free to merge a Pull Request.

Even if a Pull Request is approved, you should always check with the author of the Pull Request if they are ready to merge.

The AEM Code Sync GitHub app will automatically deploy changes merged into the main branch to production.

**(Scaled) Trunk-based development**

For AEM projects with their built-in devops and CI/CD, we recommend following the [scaled trunk-based development](https://trunkbaseddevelopment.com/) model. This means you merge small pull requests into production often, but the quality assurance & review efforts are limited to small change sets. Nobody wants to review and test large pull requests, and long-lived branches with lots of changes tend to be difficult (and dangerous) to merge.

**Dependency management**

Make sure your dependencies are kept up to date. Even if you don't add more dependencies, it is good practice to keep the minimal out of the box dependencies (linting) up to date. Feel free to use tools like [Renovate](https://www.mend.io/renovate/).

**CSS**

**CSS selector block isolation**

AEM blocks most often operate as components collaboratively in the same DOM / CSSOM. This means that you should write your CSS selectors in a block .css in a way that isolates your CSS from impacting layout of elements outside your block. The easiest way to do this is by making sure that every CSS selector in the .css of a block only applies to the block.

**Cascade in CSS**

Use your CSS classnames wisely. Some CSS classes and variables are used across different blocks, and others are not expected to be used outside your block. Prefixing classes and variables that are private to your block with the block name is good practice. Conversely, if there are CSS classes and CSS context that should be inherited (often those can be authored) those classes and variables should not be prefixed.

**CSS indentation and property order**

Outside of a CSS refactoring PR, don’t change sequencing on properties or the indentation across the CSS files you touch in a functional PR. Every developer has different preferences on sort order of properties or indentation. Make sure the diff that you see in your PR is isolated to the changes you actually want reviewed before submitting it.

**CSS selectors complexity**

Don’t let your CSS Selectors become complex and unreadable. Often it is better to decorate additional CSS classes/Elements onto your DOM and write readable CSS instead. Complex CSS selectors also often are harder to maintain and more brittle than the equivalents in JS.

**CSS naming**

Naming your classes simple and intuitive is helpful for other developers. Avoid namespacing unless it is necessary within the scope of a project. There is often no need to specify the type or the origin (e.g. the name of your design system) of a CSS variable that is to be used across the entire project.

**Leverage ARIA attributes for styling**

In many situations you will add ARIA Attributes for accessibility. Since those have well defined semantics like expanded or hidden that are understood by all developers, in most cases there is really no need to come up with additional classes in your vocabulary that have unknown semantics.

**Mobile first**

Generally Web projects should be developed “Mobile First”. This means that your CSS without media query should render a mobile site. Add media queries to extend the layout for tablet and desktop.

**Breakpoints**

Generally use 600px, 900px and 1200px as your breakpoints, all min-width. Don’t mix min-width and max-width. Only use different breakpoints in exceptional cases where you have good understanding why that’s needed.

**Less, Sass, PostCSS, Tailwind and friends.**

If you are working in the context of a bigger organization, make sure that you don’t introduce a dependency to any CSS preprocessor or framework of your personal preference without getting the buy-in from the entire team and organization. As there are a lot of differing personal preferences in this area, it makes code hard to maintain if every project or every block inside a project uses different technologies.  
The simplest solution is to rely on the growing CSS feature set which is supported by the browsers.

**Modern CSS features**

Make sure the features you are using are well supported by evergreen browsers. Depending on the features more or less pervasive support may be acceptable.

**JavaScript**

**Frameworks**

On most web sites, frameworks are overkill for simple layout problems outside of application-like functionality. Frameworks often introduce web performance issues (Lighthouse and Core Web Vitals), particularly if they are in the pathway of the LCP or introduce TBT, while trying to address trivial problems. Keep simple things simple.  
If you are using Javascript Frameworks make sure that you don’t introduce a dependency to any JS Framework or library of your personal preference without getting the buy-in from the entire team and organization. As there are a lot of personal preferences, it makes code hard to maintain if every project or every block inside a project uses different technologies.  
The simplest solution is to rely on the growing feature set supported by browsers.

**Build tool chain**

Differing build tool chains from project to project make it hard for new developers to be onboarded and often introduce additional complexity. Make sure that you don’t introduce a dependency of your personal preference without getting the buy-in from the entire team and organization.  
The simplest solution is to keep the entire project build-less.

**Modern JavaScript features**

Make sure the features you are using are well supported by evergreen browsers. Depending on the features more or less pervasive support may be acceptable. While AEM can be used with any browser, aem.js has a dependency on browsers that support dynamic import(). Any feature that is supported by the set of browsers that support dynamic import() should be considered safe. Technically, of course, older browsers (e.g., IE) can be supported by AEM projects, but those require heavy customization.

Not all features have the same consequences if a browser doesn’t support them, some may be cosmetic and others may stop the site from working. A common example is “optional chaining”. If a browser doesn’t support “optional chaining”, a single usage can have fatal consequences for the entire page.

**Loading 3rd party libraries**

Don’t add 3rd party libraries to your <head> via head.html as they will be in the critical path of loading content above the fold and will often be loaded when they are not needed. Add the dependencies where needed via loadScript() to the specific block that has the corresponding requirement.  
In case of larger 3rd party libraries, you may even want to consider using an IntersectionObserver to make sure you only load them when the block depending on it is actually being scrolled into view.

**AEM library (aem.js)**

The [AEM library](https://github.com/adobe/aem-lib) (usually called aem.js in a boilerplate project) is currently not minified and obfuscated to make debugging easier. We discourage making changes to it on a project basis and instead recommend project specific extensions to be kept outside of the library. We welcome Pull Requests via GitHub, if you would like to propose changes or bug fixes that are universally applicable.

**<head>**

The <head> that is delivered from the server as part of the HTML markup should remain minimal and free of marketing technology like Adobe Web SDK, Google Tag Manager or other 3rd party scripts due to performance impacts. Adding inline scripts or styles to <head> is also not advisable for performance and code management reasons.

**Minification**

Typically this is additional complexity without much benefit unless you have really large JS/CSS files which again would be an anti-pattern. With Edge Delivery, the way the code is structured around blocks, the files should be usually small in size and minifying should not make much of a difference. To be sure, you can compare the [Lighthouse Score](https://pagespeed.web.dev/) pre/post minification.  
Minification makes code slightly harder to debug and you'd need sourcemaps. Also minifying requires an additional build step which can potentially slow down site development work. So you’d want to go there only if there is a tangible benefit of this additional complexity

**Content first**

**Start your development with content**

Before writing a line of code, create your (sample) content in a Word or Google Doc (or spreadsheet). Make sure that it feels good for authoring and share it with people on your team that have experience supporting authors. It requires support experience to understand what content structures are easy for authors to understand and recreate. Once you have settled on a content structure that contains all elements you need for your block, and have it reviewed you can get started developing your CSS and JS code.

**Use drafts**

The content lifecycle is very different from the code life cycle. If you are proposing changes to an existing content structure in your code or come up with a new block, don’t just make those changes on the page you are working on. Copy the page into your /drafts/<yourname>/ folder and make changes there.  
Once your code changes are merged to main , you can have authors copy or merge your content with the content outside of your /drafts/ folder.

**Backwards compatibility of new features**

Especially in production environments it is important to keep your anticipated changes to the content structure backwards compatible with existing content. Ideally, code that is being merged should not have an impact on the website or require refactoring the content. Only when new content is put in place through a preview and publish cycle, the new functionality becomes available. This of course doesn’t apply to things like design changes across the existing content or functional bug fixes.

**Use content for “static” resources**

Generally, it is not a good idea to commit binaries into your GitHub repo. Even text-based static resources, for example HTML files or SVGs should only be put into GitHub in exceptional cases. A good reason to add an SVG to your git repo is if it is referenced from code. Don’t commit anything that is related to the content authoring process, or could be a part of an authoring process. There are some exceptions (usually related to legacy and non-browser clients) that require a certain set of fixed resources that cannot be produced and manipulated dynamically by AEM, but in general if you find a large set of static resources (e.g. images, etc.) or an HTML file in a PR it is most likely not a good practice.

**Use content for strings/literals**

Strings that are displayed to end users and could possibly be translated or changed at some point should always be authorable and sourced from content (eg. [placeholders](/docs/placeholders) or other spreadsheets or documents). If you have a literal string that is displayed as text to the visitor of your website via javascript or css code, it is good practice to replace it with a reference to content.

| **Metadata** | |
| --- | --- |
| Title | Development Collaboration and Good Practices |
| Image | A white rectangular object with a cat face and hands  AI-generated content may be incorrect. |
| Description | Working with a large number of development teams across many projects and organizations, we found that it is useful to collect some of our insights. ... |
| Template | guides |
| Category | Build |