Documentation checklist

The purpose of this document is to provide a framework for analyzing your project and infrastructure in order to capture all revelant documentation. Some sections may not apply in your case. Or, there may be something very unique about your environment that is not covered here. Use this as a guide and a starting point.

Documentation is only useful if it can be easily found, and if it is kept up to date. High-level documentation should exist in one central spot, not tucked away inside one particular project or repository. One useful approach is to create a dedicated repo called `infrastructure` in GitHub, and use it for those high-level docs. If you chose to use the Wiki functionality in Github, include a prominent link in the repo's `README` explaining that is where the documentation can be found.

> *Even if you only have one project/repo currently, it's still a good idea to keep the high-level documentation in a separate, central place. That way, when you add a second repo, you don't have to shuffle things around.*

Keeping documentation up to date requires process and discipline. Speak with the team and any stakeholders about the need to keep documentation up to date. If anyone sets up a new service, changes service configuration, or decommissions a service, they should also take responsibility to include documentation. One benefit of using Github for documentation is that you can use your normal code review process to make sure documentation is accurate and up to date.

Action items

- [ ] Decide on a location for high-level documentation. Create an `infrastructure` repo, if necessary.
- [ ] Communicate with the team and stakeholders about how to find and contribute to documentation.
- [ ] If you have a code review process or checklist, add documentation review to that process.

Credentials

This document will refer to many different services that require login credentials. Do not add the credentials to your documentation. You should be able to freely share this documentation with your team. If you add credentials directly to the document, you would only be able to share it with those that should have full access to every documented service.

Instead, you should use this section of the document to explain where credentials can be found. Then, you can use other access control mechanisms to decide how to share credentials when necessary. For example, popular password managers have team functions or shared vaults, where you can invite specific individuals to have access to specific credentials.

Credential sharing should be considered a last resort, or as a disaster recovery scenario. Most services have functionality to manage multiple users on a team. Instead of sharing one common account, leverage those user management features. It may require you to pay for a higher tier of service, but this is usually worth the added security and functionality.

Consider AWS as an example: No one should ever need to share AWS credentials for an account. See our AWS Account Setup guide for instructions on how to invite other developers, even contractors outside your organization, to your AWS account. That particular setup guide assumes inviting a trusted developer who can manage everything, but you could alter the Role permissions to allow access to only specific services.

That being said, there is one root AWS account which has certain access which cannot be fully delegated. It is critical to make sure this root AWS account is shared with more than one person in your organization. Losing access to that can be extremely difficult to recover from. This is the disaster recovery scenario mentioned above.

The other scenario where credential sharing may be necessary are for secrets like API keys or for services that don't have any sort of user management capability. In these cases, there is no other choice but to share a single credential, but this should represent a very small percentage of your credentials.

It may require some research and team communication to identify all these credentials. As you start documenting it, you may identify credentials managed by a team member or accounts owned by a third party (perhaps an outside contractor).

Action items

- [ ] Set up a mechanism for sharing credentials (for example, 1Password).
- [ ] Capture all existing credentials into that central secure store.
- [ ] Interview the team to identify "one-off" services that were set up without your knowledge.
- [ ] Migrate "one-off" or third-party-owned services to be under your control.
- [ ] Invite at least one highly-trusted person to share all credentials for disaster recovery purposes.
- [ ] Invite team members to share credentials for secrets or services without user management.
- [ ] Review all previous use of shared credentials to identify services with user management features and migrate to user accounts.
- [ ] Reset any passwords previously shared that are now migrated to individual user accounts.
- [ ] Create a Credentials section in your high-level documentation.
- [ ] Document the mechanism for sharing credentials.
- [ ] Document the primary and disaster recovery contacts that will manage shared credentials.

Infrastructure

The goal of this section is to capture a high-level list of all the infrastructure services in use across your organization. This can be especially helpful as the organization grows and additional projects are built and launched. There will be much more detailed project-specific information in each repo, but here you can see a quick list of what is already in use.

Start by listing out every resource you currently use. You can use the credentials gathered earlier as a starting point. It may also be helfpul to review credit card statements, project environment files, and any existing documentation to find additional services that were overlooked. You will likely find yourself coming back and adding in things here as you build out each project's infrastructure documentation.

For each service, include a simple description of how it's used. For example, instead of just listing AWS, which has hundreds of services, you could list "AWS - servers, file storage, transactional email". It is also helpful to mention if a particular service requires use of a shared credential or secret.

Action items

- [ ] Document all domains owned, which registrar and DNS is used
- [ ] Document all cloud providers in use (AWS, Digital Ocean, Google Cloud Platform, Azure, etc)
- [ ] Document any managed hosting providers (WP Engine, Squarespace, etc)
- [ ] Document any transactional or marketing email providers
- [ ] Document any billing providers (Stripe, Paddle, Recurly, etc)

- [ ] Document any third-party APIs in use, whether they are free or paid
- [ ] Document any provisioning tools in use (Forge, Envoyer, etc)
- [ ] Document any developer-focused services (bug trackers, CI pipelines, source code hosting, Docker image hosting, etc)

Cross-project workflows

If you have projects across multiple repos, and those interact with each other, it can be helpful to provide a high-level flowchart of how they interact. For example, if you have one repo for a mobile app, another for an API backend, and a micro-service for authentication and identity, you could describe how these projects interact, and how data flows from one to the other. This becomes even more critical if you have a microservice architecture with many independent, but interconnected, services.

As with the infrastructure, this centralized documentation is meant to be just a high-level overview. The details of each individual microservice would be within that project's repository.

Action items

- [ ] Identify any interactions between your projects/repositories.
- [ ] Document a high level flowchart to visualize these interactions. Mermaid is a handy way to do this, and is supported natively in Github.

Project documentation

Each of your projects requires its own specific documentation as well. There will be some minor overlap with the high-level organizational documentation, but that's okay. Each set of documentation serves its own purpose.

The project's `README` is a great place to put this documentation. Depending on the size and scope of your documentation, you may also choose to put some of it in the Github wiki. Even if you choose to use the wiki, make sure your project has a `README` with the most common pieces of documentation, and then include links to the wiki for the rest.

Project purpose In a few sentences, explain the basic purpose of this repo. For example, you could describe what domain the site is hosted at, who uses the site, and its importance to the business. This should definitely be at the top of the `README`.

Project history Things change over the life of a project, and it can be very useful to capture that information in the documentation. For example, if your project has an API, and there is both a v1 and v2 API, describe why that's the case. Who uses v1 versus v2? Are there plans to sunset v1 at a certain date?

In addition to major version changes like this, there could be other historical details that would be useful to share. Did you switch billing providers? Did you change from a subscription model to a pay-per-use model? Is there some old functionality that is deprecated? One helpful thing to consider: if I were coming new into this project today, what sorts of things would be useful for me to know about the history of the project? Capture those in the documentation.

Technical documentation This is probably the single most useful section of the documentation for someone new joining the project. There should be a step by step guide on how to get a local development environment set up. A lot of this could be captured automatically in a Docker config, or similar tooling, but even if you don't currently use those tools, there should be a reproducible way to get everything set up for local development.

Once the environment is running locally, you'll want to know common scripts and tasks for day-to-day development. Is there a particular coding standard enforced? Explain how to check it. What are the various test suites in use? Explain how to run them. Do you enforce rules via static analysis? Capture all of these handy tools in this section.

If you have developer-focused tools setup, highlight what those are and how they can be used. For example, is Telescope enabled? Is there a particular logging strategy in place? Does one set of Docker config have xdebug enabled, but another doesn't? Those are the sorts of things that should be explained in the documentation.

Process documentation Each team has their own process for how work is assigned, reviewed, and deployed to production. Use this section to document that process. Some questions that should be answered: Do you create new branches for each feature? Is there a particular naming convention for those feature branches? What branch should they be based off? When a developer is done with a feature, what is the process for code review?

This section is also a good spot to explain how any automated integration process is executed, and what sorts of things it checks. The CI configuration is likely already inside the repo, but a high-level explanation here can be useful. For very detailed CI notes, you can include more code comments in the CI configuration directly.

Closely related to CI is deployment. When is a feature deemed ready to deploy? Who makes that decision? How do you get code deployed into the various environments (QA, staging, production)? Whether deployments are automated or not, make sure to document the exact steps needed.

Action items

- [ ] Add a "Purpose" section to the README
- [ ] Add a "History" section (either README or wiki)
- [ ] Write up exact steps to get your project running locally on a new computer
- [ ] Document any important scripts for common development tasks
- [ ] Document how to run the various test suites (phpunit, javascript, etc)
- [ ] Document common troubleshooting or debugging scenarios
- [ ] Describe the typical workflow for working on a feature or bugfix
- [ ] Document your CI system
- [ ] Documment how to deploy into each of your environments