Open Government Products - Best Practices

Development

Node.js Developer Setup

This covers various aspects of setting up an environment for development purposes

Linting

See Linting

Running things on a dev machine

When you're developing, you want to see your changes reflect automatically. Various packages can automatically watch for changes and restart your build and run steps.

- nodemon to restart your server
- webpack --watch to build static assets
- http-server - to host static assets
- concurrently - to run multiple processes in a single command via npm, eg webpack --watch and nodemon

Linting

Code should be at the very least consistently formatted. This section covers how to enforce linting across deve environments as well as within source control

Setup

IDE/Editor

Most IDEs/text editors have some way to enable automatic linting of the files you're editing. Here's a list of editors + plugins that the team are using:

- Sublime Text + SublimeLinter
    - Provide sublime text setup for linting here
- VS Code + ESLint + Document This

TODO: Flesh out with more examples

Git Hooks

A recommended way to enforce linting in a codebase is to set up git hooks to lint the codebase. This will help to ensure that the git history is tidy and that all pull requests meet the code style/formatting standards, preventing code reviews from being cluttered with basic comments on code style/formatting.

Recommended hooks: pre-commit or pre-push

Sample setup: beeline-frontend

1. Install the pre-commit npm package to set up the git pre-commit hook
2. Configure pre-commit to run an npm script in package.json (see here)
3. Install the lint-staged npm package to only run linter against staged files
4. Configure lint-staged to only look for *.js files (optionally automatically git add) (see here)

CI

Linting should also be done as part of the CI process. A common setup in the team is to use Travis CI. A sample Travis YAML file can be found here.

JavaScript

eslint config

The team has a standard eslint config that every project should use as a base config before extending it where necessary. It's available as an npm package - instructions to install and use can be found here.

An example of how to use this can be seen here. Pay special attention to .eslintrc.json and how to extend the base config to suit individual project needs (e.g. unique global variables for angular projects).

TODO:

- Add addtional eslint plugins which can provide useful static analysis:
    - eslint-plugin-node and eslint-plugin-import (untested)
        - Makes sure you require available files and modules
    - eslint-plugin-promise
        - Makes sure promises are used correctly

TODO:

- Add links to good tutorials (for onboarding)

Using GitHub

Source control is vital to collaboration between developers. This sets out a few guidelines worth observing when coordinating changes in GitHub

Branching strategy

- Maintain a mainline and release branch, both protected
- Changes in mainline branch can be moved to the release branch through one of the following:
    - Raising a PR merging mainline changes into release
      Recommended - the release branch clearly indicates what changes the releases have, making rollbacks easier and less error-prone to execute.
    - git fetch && git checkout <mainline-name> && git pull && git push origin <mainline-name>:<release-name>
      Discouraged - while it leads to a cleaner commit tree (since both branches end up on the same sequence of commits), switching between releases is more prone to human error (though this can be mitigated using release tags), and requires a fair understanding of git and using git via the command line. That said, this facilitates more precise control over what is released, at the commit level.
- Feature branches should hang off and merge into the mainline branch
    - Favour rebasing over merging to update feature branches
    - Keep feature branches **small**. This makes the PR easier to understand, and also makes it easier to rebase the feature branch on mainline should the need arise
    - If implementing a feature will take too many commits or too much change, you might be undertaking too much work in one go. If so, split the work into more manageable chunks
- Hotfixes and trivial changes can be squashed as a single commit onto mainline

See Atlassian's article on Gitflow Workflow, on which our strategy is loosely based on

Branch naming

- Mainline and release branches can have the following respective names:
    - master and release
    - master and production
    - develop and master
    - Any pair of names that conveys the sense that one branch is more stable than the other
- Branch names should be kebab-case (ie, small caps only, hyphen as word separator)
- Optionally, branch names can be prefixed by a category name followed by /,
    - eg. feature/more-cowbell or hotfix/intermittent-hackfix

Writing commits

See How to Write a Git Commit

Writing PRs and Comments

Make use of the following features:

- Closing issues via Pull Requests
- Referencing commits, issues and other repos
- Referencing repository code
- GitHub Flavored Markdown
- Marking duplicates