# Style Guide Suggestion for Scobees

## Git Style

- A clean git helps to keep track of your work and make it transparent to others involved.
- There are at least three factors that need to be discussed:
    - Branching
    - Commit messages
    - Commit style (squashing, merging)

### Git Branching

**Keep track of what is released and what is in development for what purpose.**

- The following document suggests a branching model that is still state of the art if you want to keep your production system stable: https://nvie.com/posts/a-successful-git-branching-model/
- In short, these are the branches you will need:
    - **feature branches:** This is where you develop new features; all branches are prefixed with *feature/*
    - **develop branch:** Finished feature branches are merged into *develop*; this is where most of your commits take place, current unstable development branch
    - **release branch:** This is where you prepare your current development state for release; no new features, only bug fixes on the way to your next planned release are committed here; remember to merge back into *develop*, if you commit bugfixes
    - **master branch:** This is where you keep your production code, add tags to keep track of your release versions; merge only from *release* or *hotfix*
    - **hotfix branches:** If there is a bug in your production code, create a branch prefixed with *hotfix/* to eliminate the bug; remember to merge it into *master* and *develop* when finished

### Git Commit Messages

**Help your team to keep track of what you have committed to the project.**

- Commit messages should be understandable without looking into the code; in fact, even your project maintainers (non-developers) should be able to understand, what you have done.
- This blog post suggests a clean and easily understandable way of writing commit messages: https://chris.beams.io/posts/git-commit/
- In short, the main points are:
    - Use subject line and message body (separated by a blank line)
    - Keep the subject line short
    - Capitalize the subject line
    - Use the imperative mood in the subject line
        - The subject line should complete the imaginary sentence “If applied, this commit will …”
    - Provide additional information in the message body

- In addition to these points, it may be useful to categorize your commits by adding prefixes to your subject line
    - “New:” New feature added
    - “Fix:” Something was fixed
    - “Change:” Behavior changed
    - “Migration:” Changes in data structure were necessary
- If you have a ticket system, refer to the related issue in your message body.

### Git Commit Style

**Squash commit feature branches to keep your other branches clean and their history short.**

- The afore mentioned rules for commit messages are not written into stone but should be respected for develop, release and hotfix branches.
- Feature branches take a special part, because I suggest merging finished feature branches into develop in a single commit.
- Doing it this way, you have to write only one elaborate commit message instead of thinking about it every time you push something into the repository.
- The other big advantage is, the develop branch stays clean and clear all the time, as you don’t have every single work-in-progress commit showing up in your commit history after merging.
- You can do whatever you want inside your feature branch, if you finalize it with one clean pull request into develop.
- Go thoroughly through every single changed file to avoid committing unnecessary changes and obsolete code.

## Code Style

### Documentation

**Help new team members and the “future you” to understand what you have done and why.**

- Every component should have a comment section that explains what it does.
- Every method should have a comment section that explains what it does, which parameters it takes and what it returns.
- Add inline comments wherever necessary; e. g. to elaborate what fields in your component are used for.
- The main README should explain the structure of your project, you can add more README files to your module folders as your project and complexity grow.
- Use proper Markdown syntax when writing your READMEs.
- If you copy code from Stack Overflow or another source, it means you had to google to find a solution to your problem and the solution might not always be obvious. Add the link to the solution as a comment.

### Naming Conventions

**Standardize how you name fields, translation keys, methods, components.**

- Development is a lot easier, if you don’t have to guess whether the field name is school_id, idSchool or schoolId. Not to mention there could even be a *shcoolId* somewhere.
- Suggested naming convention (in every case: use descriptive identifiers):
    - Method names: camel case

    - Field names: camel case, add Id as suffix if needed
    - Interface and class names: pascal case
    - Translation keys: kebab case, avoid slashes, upper case letters and underscores
    - CSS classes: kebab case

## CSS Style Classes

**Define context-based classes, not style-based classes.**

- You don't need classes like "text-center" or "align-center" as they are not better than the use of inline styles.
- Use classes like "card-header" or "card-grid" instead and define your styles context-based.
- Maybe even define classes locally inside your components' style sheets if they only apply to a single component.

**Use class hierarchy instead of the important keyword. In general, avoid "important".**

- CSS classes depend on the order in which they are defined. A second definition of a class overwrites the first one.
- Keep track of this when you define your styles or themes and avoid overwriting classes with *important*.
- If you make extensive use of the *important* keyword when overwriting classes from your UI framework (e. g. Angular Material), take a step back and try to figure out if there are other ways of theming the UI or if you should look for another framework that better suits your needs.

**Use CSS custom properties for better theming.**

- CSS supports custom properties which make it a lot easier to add theming to your style sheet and allow runtime changes (big advantage over SCSS variables).
- https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties

**Use SCSS hierarchy wherever useful.**

- If you have nested styles, make use of SCSS' scoping by curly braces and avoid CSS like notation.
- This will give a clean structure to your style sheet.

**Use imports to keep your files short.**

- If you still have a lot of classes that need to be defined globally, split your style sheet into multiple files and import them into your main file in the correct order.
- Define a meaningful structure while doing so.

## Folder and Project Structure

**Define a folder structure that fits your needs and stick to it.**

- There are a lot of different approaches to structure angular projects.
- Whatever way you decide for, restructure your code to fit the rules and respect the rules with every new feature you develop.
- If, at a time, you find out that the structure you decided for, doesn't fit, adjust the rules and restructure your project in a way that fits the new rules.

- Here are some approaches:
    - https://itnext.io/choosing-a-highly-scalable-folder-structure-in-angular-d987de65ec7
    - https://stackoverflow.com/questions/52933476/angular-project-structure-best-practice

## Keep Your Code Updated

**Angular provides new releases on a regular basis. Don't run out of service.**

- The angular team provides long term service for 12 months for every major version.
- As the web and browsers evolve quickly and the migration guide gets longer with every new release, you should try not to run out of service and plan upgrading angular on a regular basis.

## Refactoring

**Refactoring on a regular basis is necessary to keep your code clean.**

- Mistakes happen.
    - There may not always be enough time to add comments for every component and every method.
    - Old code needs to be cleaned up from time to time.
- If you refactor, check for all aspects described in your style guide.
- Create a feature branch for refactoring.
- Make extensive use of the find-and-replace function of your preferred IDE.
- Keep track of what you've done and describe it in your commit message(s).