Contents

1 Doubtfire Git Workflow

We follow a Forking workflow[1] when developing Doubtfire.

1.1 Table of Contents

1.2 About the Doubtfire Branch Structure

We try to keep two main branches at all times:

- `master` for production
- `develop` for current development, a branch off `master`

That way, we follow the workflow:

1. branch off `develop`, giving your branch one of the prefixes defined below,
2. make your changes in that branch,
3. merge your branch back into `develop`,
4. delete your branch to clean up

[1] See `https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow`

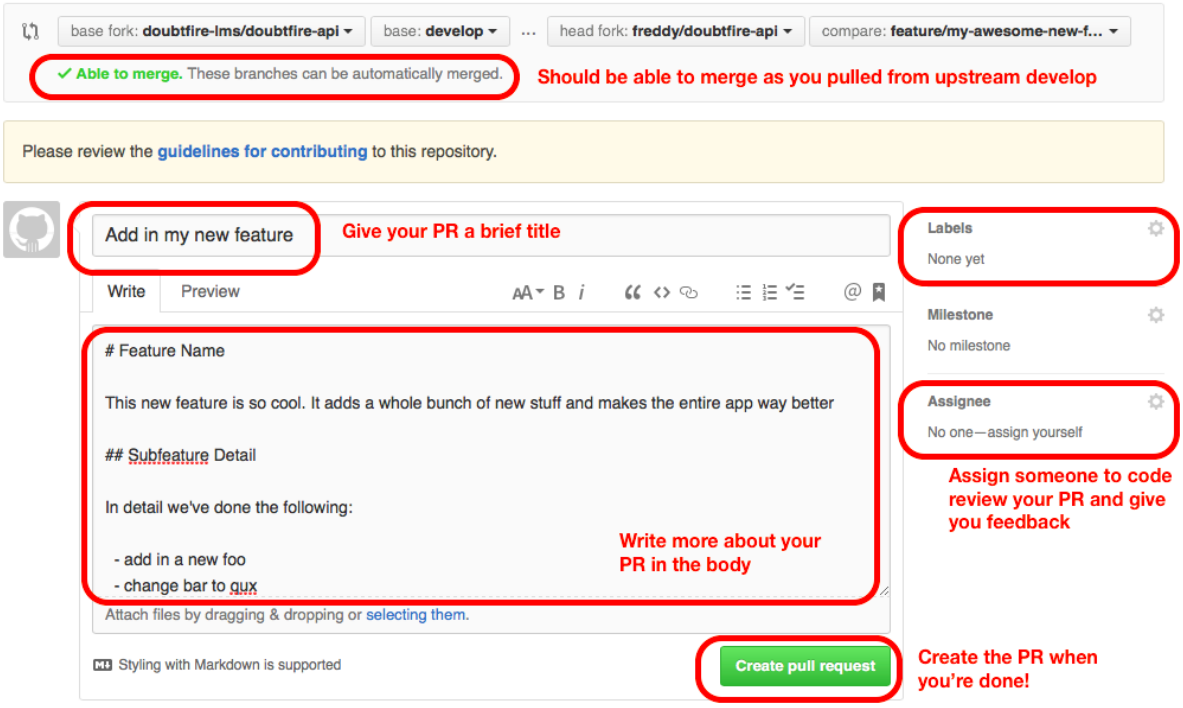

Figure 1: Feature Branches

In some cases, your branches may only consist of one or two commits. This is still okay as you can submit a pull request for code review back into `develop`.

You may want to branch again, e.g.:

```
* master
|\
| \
| |
| (b1) develop
| |\
| | (b2) feature/my-new-feature
| | |\
| | | (b3) test/unit-tests-for-new-feature
| | |/
| | (m1)
| |/
| (m2)
| |\
| | (b4) fix/broken-thing
| |/
| (m3)
| /|
|/ |
(m4)
|  |
|  |
*  *
```

Here, we:

1. branched off `master` to create our `develop` branch, at **`b1`**
2. branched off `develop` to create a new feature under the new branch `feature/my-new-feature`, at **`b2`**
3. branched off `feature/my-new-feature` to create some unit tests for that feature under `test/unit-tests-for-new-feature`, at **`b3`**

4. merged those unit tests back into `feature/my-new-feature`, at **`m1`**
5. merged the new feature back into `develop`, at **`m2`**
6. found a new bug in the feature later on, so branched off `develop` into `fix/broken-thing`, at **`b4`**
7. after we fixed our bug, we merged `fix/broken-thing` back into `develop`, at **`m3`**
8. decide we're ready to release, so merge `develop` into `master`, at **`m4`**

Note that along the way **we're deleting branches after we don't need them**. This helps us keep *short-lived* branches that don't go *stale* after months of inactivity, and prevents us from forgetting about open branches. The only branch we kept open was `develop`, which we can always branch off for new, un-released changes again.

Ideally, any changes that are merged into `master` have been **code-reviewed** before they were merged into `develop`. **You should always code review before merging back into `develop`**. You can do this by performing a Pull Request, where the reviewer can see the changes you want to merge in to `develop`.

1.3 Getting started with the Forking Workflow

1.3.1 1. Forking and Cloning the repository

1.3.1.1 Fork the Repo

To get a copy of a Doubtfire repositories on your user account, you will need to fork it *for each repository*:

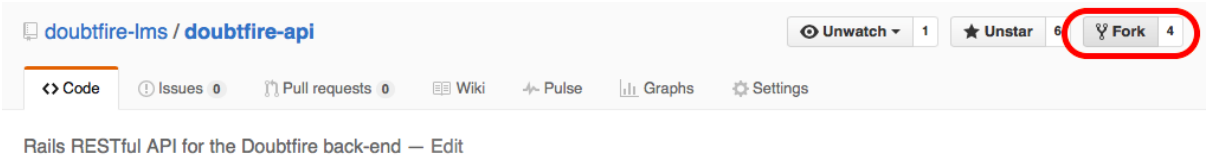

Figure 2: Fork the repo

1.3.1.2 Clone the Fork

You can then clone the repositories you have forked to your machine. To do so, navigate to your forked repositories and copy the clone URL:

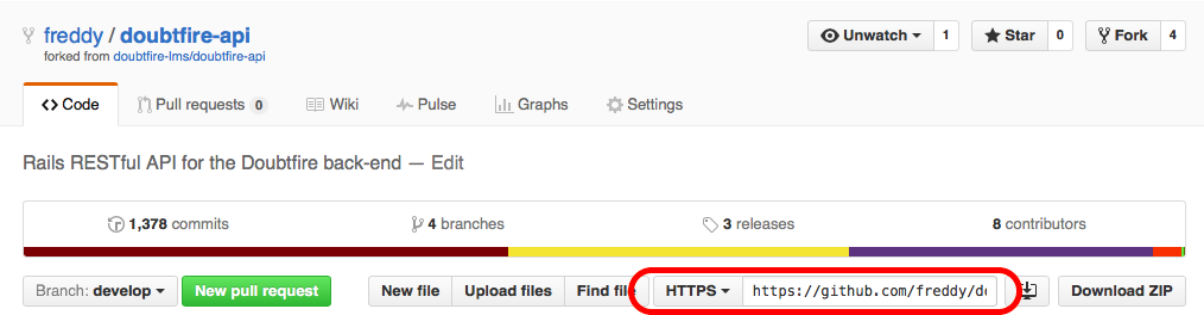

Figure 3: Copy the clone URL

Navigate to your `projects` or `repo` folder, and make a `doubtfire` folder. Then clone using the URLs you copied above:

```
$ cd ~/repos
$ mkdir doubtfire
$ cd doubtfire
$ git clone https://github.com/{username}/doubtfire-api.git
$ git clone https://github.com/{username}/doubtfire-web.git
```

1.3.1.3 Set up your upstream to `doubtfire-lms`

By default, git tracks your remote forked repository (the repository you cloned). This remote is called `origin`.

You will then need to set up a new remote to track to the `doubfire-lms` owned repository. This will be useful when you need to get the latest changes other developers have contributed to the `doubtfire-lms` repo, but you do not yet have those changes in your forked repo. Call this remote `upstream`:

```
$ cd ~/repos/doubtfire/doubtfire-api
$ git remote add upstream https://github.com/doubtfire-lms/doubtfire-api.git
$ cd ~/repos/doubtfire/doubtfire-web
$ git remote add upstream https://github.com/doubtfire-lms/doubtfire-web.git
```

1.3.1.4 Ensure you have your author credentials set up

You should ensure your git user config are set and set to the email address you use with GitHub:

```
$ git config --global user.email "my-github-email@gmail.com"
$ git config --global user.name "Freddy Smith"
```

1.3.1.5 Use a rebase pull

We also want to avoid having merge commits whenever you pull from `upstream`. It is useful to pull from upstream using the `--rebase` switch, as this avoids an unnecessary merge commit when pulling if there are conflicts.

To fix this, always pull with `--rebase` (unless otherwise specified—see the `--ff` switch needed in Step 3):

```
$ git pull upstream develop --rebase
```

or alternatively, make a rebase pull as your default setting:

```
$ git config --global pull.rebase true
```

1.3.2 2. Writing your new changes

As per the branching structure, you need to branch off of `develop` to a new branch that will have your code changes in it. When branching, **be sure you are using a branch prefix**:

```
$ cd ~/repos/doubtfire/doubtfire-api
$ git checkout -b feature/my-awesome-new-feature
```

You can now begin making your changes. Commit along the way, **being sure to conform to the commit message guidelines**, on this branch and push to your fork:

```
$ git status

On branch feature/my-awesome-new-feature
Your branch is up-to-date with 'origin/feature/my-awesome-new-feature'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
  modified:   src/file-that-changed.js
  modified:   src/another-file-that-changed.js

$ git add src/file-that-changed.js src/another-file-that-changed.js
$ git commit

[feature/my-awesome-new-feature 7f35016] DOCS: Add new documentation about git
 2 files changed, 10 insertions(+), 15 deletions(-)

$ git push -u origin feature/my-awesome-new-feature
```

Note you only need to add the `-u` flag on an initial commit for a new branch.

1.3.3 3. Prepare for a Pull Request

Note, while it is advised you perform this step, it you can skip it and move straight to the Pull Request step. If the branch cannot be automatically merged, then you should run through these steps.

When you are done with your changes, you need to pull any changes from `develop` from the `upstream` repository. This essentially means "get me anything that has changed on the `doubtfire-lms` repository that I don't yet have".

To do this, pull any changes (if any) from the `upstream` repository's `develop` branch into your local `develop` branch:

```
$ git checkout feature/my-awesome-new-feature
$ git pull --ff upstream develop
```

If there are merge conflicts, you can resolve them now. Follow GitHub's guide[2] for resolving merge conflicts.

We can now update your `origin` repository's `my-awesome-new-feature` on GitHub such that it will include the changes from `upstream`:

```
$ git push origin feature/my-awesome-new-feature
```

[2] See `https://help.github.com/articles/resolving-a-merge-conflict-from-the-command-line`

1.3.4 4. Submitting a Pull Request (PR) to the upstream repository

Once you have pushed your changes to your fork, and have ensured nothing has broken, you can then submit a pull request for code review to Doubtfire.

To submit a pull request, go to the relevant Doubtfire LMS Repo and click "New Pull Request":

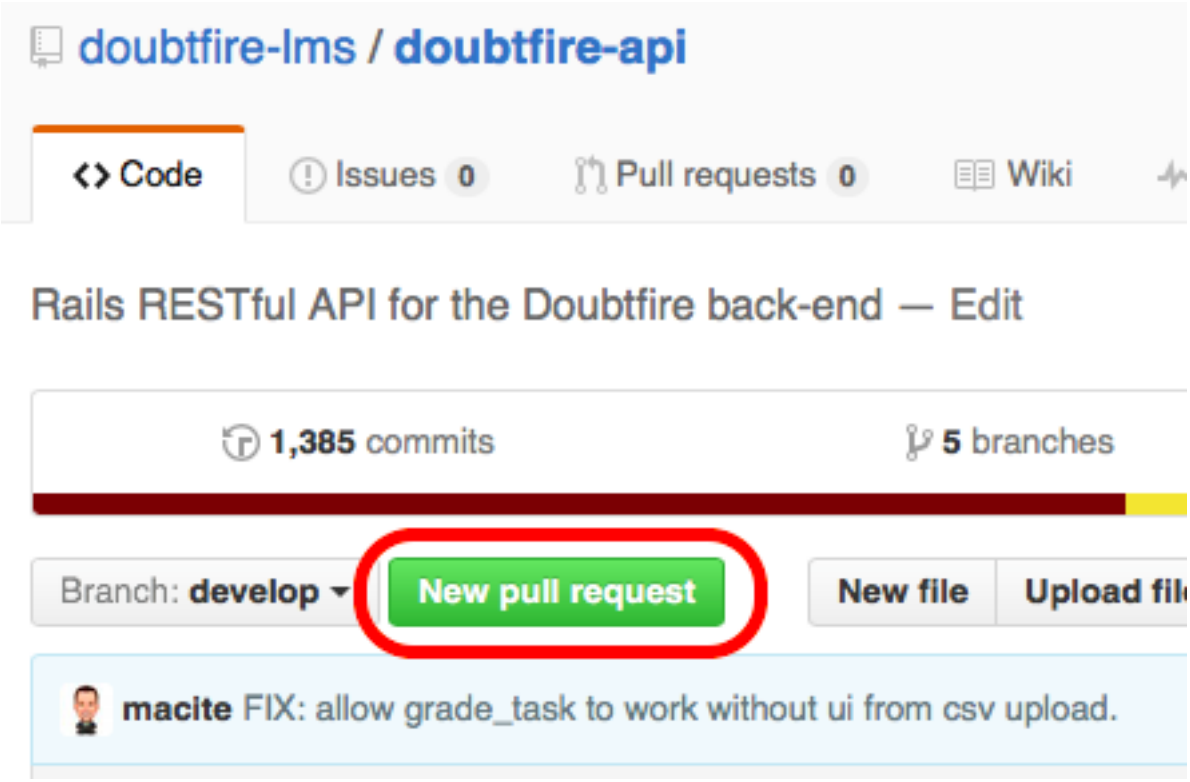

Figure 4: New PR

Ensure that the **Head Fork** is set to your forked repository and on your feature branch. If you cannot see your repository, try clicking the "Compare across forks" link.

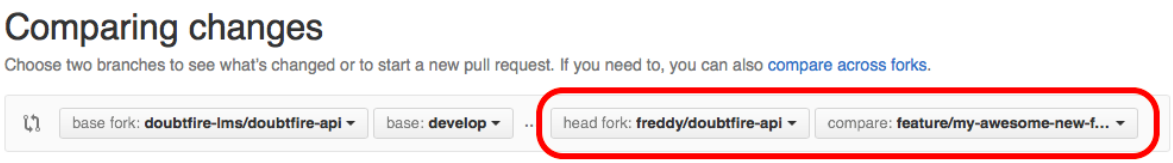

Figure 5: Compare forks

You can then begin writing the pull request. Be sure you are **Able to Merge**, otherwise **try repeating an upstream pull of develop into your feature branch, as per the previous step**.

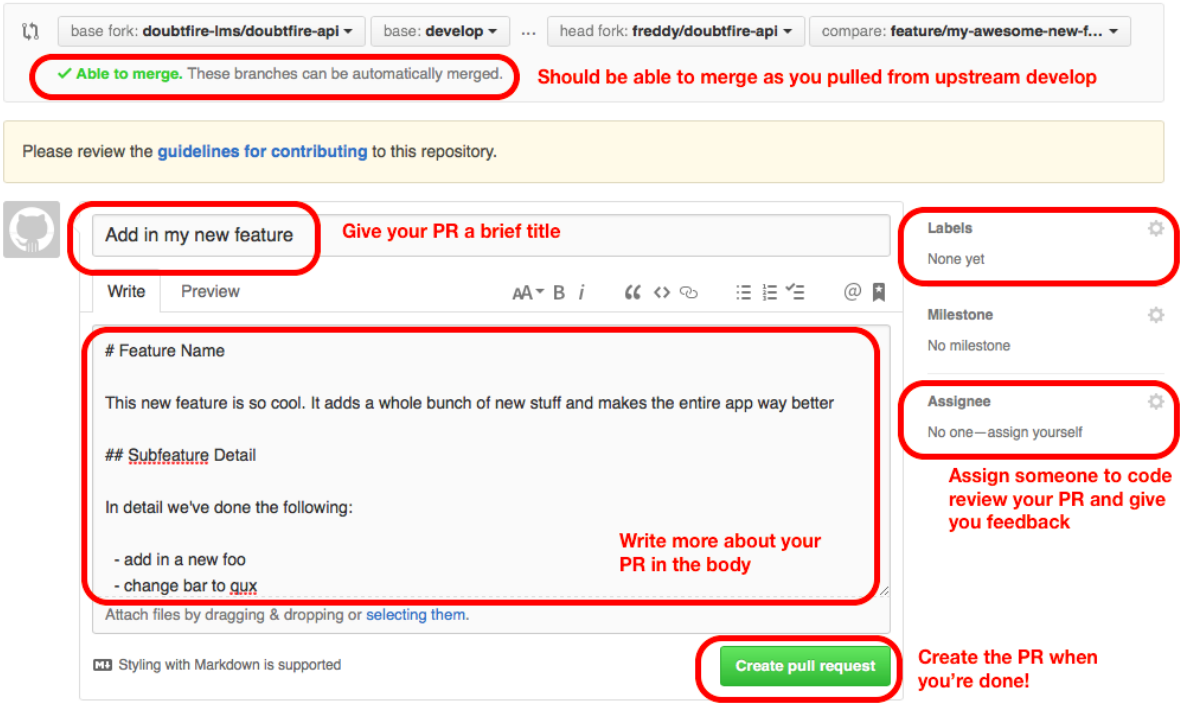

Figure 6: Writing a Pull Request

With your PR body, be descriptive. GitHub may automatically add a commit summary in the body. If fixing a problem, include a description of the problem you're trying to fix and why this PR fixes it. When you are done, assign a code reviewer and add a tag (if applicable) and create the pull request!

If your code is ok, it will be merged into `develop`, (and eventually `master`, meaning your code will go live - woohoo :tada:)

If not, the reviewer will give you suggestions and feedback for you to fix your code.

STOP! Continue to the next step once your Pull Request is approved and merged into the `doubtfire-lms`'s `develop` branch.

1.3.5 5. Cleaning Up

Once your pull request is approved, your code changes are finalised, and merged you will want to delete your old feature branch so you don't get lots of old branches on your repository.

Following from the example above, we would delete `feature/my-awesome-new-feature` as it has been merged into `develop`. We first delete the branch locally:

```
$ git branch -D feature/my-awesome-new-feature
```

Then remove it from your fork on GitHub:

```
$ git push origin --delete feature/my-awesome-new-feature
```

Then ensure you are git is no longer tracking the deleted branch from `origin` by running a fetch prune:

```
$ git fetch --prune
```

As your changes have been merged into `upstream`'s `develop` branch, pull from `upstream` and you can grab those changes into your local repository:

```
$ git checkout develop
$ git pull upstream develop
```

Then push those changes up into your `origin`'s `develop` so that it is synced with `upstream`'s `develop`:

```
$ git push origin upstream
```

1.3.6 Workflow Summary

Step 1. Set up for new feature branch:

```
$ git checkout develop                  # make sure you are on develop
$ git pull --rebase upstream develop    # sync your local develop with upstream's develop
$ git checkout -b my-new-branch         # create your new feature branch
```

Step 2. Make changes, and repeat until you are done:

```
$ git add ... ; git commit ; git push     # make changes, commit, and push to origin
```

Step 3. Submit a pull request, and **if unable to merge**:

```
$ git pull --ff upstream develop          # merge upstream's develop in your feature branch
$ git add ... ; git commit                # resolve merge conflicts and commit
$ git push origin                         # push your merge conflict resolution to origin
```

Step 4. Only when the pull request has been **approved and merged**, clean up:

```
$ git checkout develop                    # make sure you are back on develop
$ git branch -D my-new-branch             # delete the feature branch locally
$ git push --delete my-new-branch         # delete the feature branch on origin
$ git fetch origin --prune                # make sure you no longer track the deleted branch
$ git pull --rebase upstream develop      # pull the merged changes from develop
$ git push origin develop                 # push to origin to sync origin with develop
```

1.4 Branch Prefixes

When branching, try to prefix your branch with one of the following prefixes shown in Table 1.

1.5 Writing Commit Messages

Parts of this section have been adapted from Chris Beam's post, How to Write Good Commit Messages[3].

When writing commits, try to follow this guide as described in this subsection.

1.5.1 Prefix your commit subject line with a tag

Each one of your commit messages should be prefixed with one of the following shown in Table 2

1.5.2 Formatting your message

Capitalise your commit messages and do not end the subject line with a period

```
FIX: Change the behaviour of the logging system
```

[3]See `http://chris.beams.io/posts/git-commit/`

Table 1: Branch prefixes

Prefix	Description	Example
`feature/`	New feature was added	`feature/add-learning-outcome-alignment`
`fix/`	A bug was fixed	`fix/crash-when-code-submission-finished`
`enhance/`	Improvement to existing feature, but not visual enhancement (See `LOOKS`)	`enhance/allow-code-files-to-be-submitted`
`looks/`	UI Refinement, but not functional change (See `ENHANCE`)	`looks/rebrand-ui-for-version-2-marketing`
`quality/`	Refactoring of existing code	`quality/make-code-convention-consistent`
`doc/`	Documentation-related changes	`doc/add-new-api-documentation`
`config/`	Project configuration changes	`config/add-framework-x-to-project`
`speed/`	Performance-related improvements	`speed/new-algorithm-to-process-foo`
`test/`	Test addition or enhancement	`test/unit-tests-for-new-feature-x`

Table 2: Commit tagging guide

Tag	Description	Example
`NEW`	New feature was added	**NEW**: Add unit outcome alignment tab
`FIX`	A bug was fixed	**FIX**: Amend typo throwing error
`ENHANCE`	Improvement to existing feature, but not visual enhancement (See `LOOKS`)	**ENHANCE**: Calculate time between classes to show on timetable
`LOOKS`	UI Refinement, but not functional change (See `ENHANCE`)	**LOOKS**: Make plagiarism tab consistent with other tabs
`QUALITY`	Refactoring of existing code	**QUALITY**: Make directives in consistent format with eachother
`DOC`	Documentation-related changes	**DOC**: Write guide on writing commit messages
`CONFIG`	Project configuration changes	**CONFIG**: Add new scheme for UI automation testing
`SPEED`	Performance-related improvements	**SPEED**: Reduce time needed to batch process PDF submissions
`TEST`	Test addition or enhancement	**TEST**: Add unit tests for tutorial administration

and not

```
fix: change the behaviour of the logging system.
```

1.5.3 Use the imperative mood in your commit subject line

Write your commits in the imperative mood and not the indicative mood

- "Fix a bug" and **not** "Fix*ed* a bug"
- "Change the behaviour of Y" and **not** "*Changed* the behaviour of Y"
- "Add new API methods" and **not** "Sweet new API methods"

A properly formed git commit subject line should always be able to complete the following sentence:

> If applied, this commit will **your subject line here**
>
> If applied, this commit will **fix a bug**
>
> If applied, this commit will **change the behaviour of Y**

and not

> If applied, this commit will **sweet new API methods**

1.5.4 Subject and body lines

Write a commit subject, and explain that commit on a new line (if need be):

```
FIX: Derezz the master control program

MCP turned out to be evil and had become intent on world domination.
This commit throws Tron's disc into MCP (causing its deresolution)
and turns it back into a chess game.
```

Keep the subject line (top line) concise; keep it **within 50 characters**.

Use the body (lines after the top line) to explain why and what and *not* how; keep it **within 72 characters**.

1.5.4.1 But how can I write new lines if I'm using `git commit -m "Message"`?

Don't use the `-m` switch. Use a text editor to write your commit message instead.

If you are using the command line to write your commits, it is useful to set your git editor to make writing a commit body easier. You can use the following command to set your editor to `nano`, `emacs`, `vim`, `atom`.

```
$ git config --global core.editor nano
$ git config --global core.editor emacs
$ git config --global core.editor vim
$ git config --global core.editor "atom --wait"
```

If you want to use Sublime Text as your editor, follow this guide[4].

If you are not using the command line for git, you probably should be[5].

2 Contributing To Web

Please read through this document before contributing to Doubtfire.

Before continuing, **please read the contributing document[6] of the API**, as this outlines the Git workflow you should be following.

2.1 Coding Guidelines

For extendability and maintenance purposes, following these guidelines:

- Name a directive with it's role in mind (i.e., as a **Agent Noun**[7]) to give a small summary as to what the directive *does*:
- when *viewing* a project or task, the directive is project-**viewer** and task-**viewer**
- when *assessing* task submissions, the directive is task-submission-**assessor**
- when *editing* a unit's tutorials, the directive is unit-tutorial-**editor**

[4] See `https://help.github.com/articles/associating-text-editors-with-git/#using-sublime-text-as-your-editor`

[5] See `http://try.github.io`

[6] See `https://github.com/doubtfire-lms/doubtfire-api/blob/develop/CONTRIBUTING.md`

[7] See `https://en.wikipedia.org/wiki/Agent_noun`

- Name directives that show lots of data in one directive in a table a list: e.g.: `unit-student-list`, `group-member-list`, `project-top-task-list`
- Name directives with a series of steps to perform a goal a `wizard`, e.g.: `project-portfolio-wizard`, `new-user-wizard`
- Always name modals in Pascal Case `SomeModal` and create them as a factory/controller pair CoffeeScript file which can then be easily created on the fly:

```
# foo/modals/create-foo-modal.coffee
angular.module('doubfire.foo.modals.create-foo-modal', [])

#
# Prompts the user to create a Foo using a bar and qux variable
#
.factory('CreateFooModal', ($modal) ->
  CreateFooModal = {}

  CreateFooModal.show = (bar, qux) ->
    $modal.open
      templateUrl: 'foo/modals/create-foo-modal.tpl.html'
      controller: 'CreateFooModalCtrl'
      resolve:
        bar: -> bar
        qux: -> qux

  CreateFooModal
)

.controller('CreateFooModalCtrl', ($scope, bar, qux) ->
  # Does stuff with bar and qux to create a foo
  $scope.bar = bar
  $scope.qux = qux
)

# foo/states/foo-view/foo-view.coffee
```

```
# ...
.controller('FooViewCtrl', ($scope, CreateFooModal) ->
  # ...
  $createNewFoo = ->
    CreateFooModal.show($scope.bar, $scope.qux)
)
```

- Always name non-anonymous controllers with a `Ctrl` suffix
- Case correctly:
- `directiveName` should be camelCase - refer to this Angular documentation[8]
- `ServiceName`, `ControllerNameCtrl` should be in PascalCase
- Regardless of abbreviations, stick to these conventions (e.g., `pdfPanelViewer` directive works, but `PDFPanelViewer` won't work as it needs to be camelCase)
- Place modals and states in a `modals` and `states` folder under the root. All else can be in their own folders unless they are of a related concept (see the `project-portfolio-wizard` folder under `project`, `stats` under `tasks` and `units`)
- The name of a module should follow the directory structure of where it has been placed (i.e., in the above example, the template file was at `foo/modals/create-foo-modal.tpl.html`, the CoffeeScript file was at `foo/modals/create-foo-modal.coffee`, and thus the module is `doubtfire.foo.modals.create`
- Try to give a brief summary of what the directive, state or factory does. E.g., the comment in the example above for `CreateFooModal` is sufficient.
- Try to abstract as much code inside a model class as possible. At present a lot of this code is in a model's service, and it should be moved into the model's resource definition as much as possible:

```
Unit.addTutorial tutorialData
```

instead of:

```
unitService.addTutorial unit, tutorialData
```

[8] See `https://docs.angularjs.org/guide/directive#normalization`