Test Documentation

Release alpha

Test

December 20, 2015
# The Ten Commandments of Egoless Programming

1. Understand and accept that you will make mistakes. ........................................ 3
2. You are not your code. ......................................................................................... 3
3. No matter how much “karate” you know, someone else will always know more. . 3
4. Don’t rewrite code without consultation. ............................................................. 3
5. Treat people who know less than you with respect, deference, and patience. .... 3
6. The only constant in the world is change. ........................................................... 4
7. The only true authority stems from knowledge, not from position. .................... 4
8. Fight for what you believe, but gracefully accept defeat. ...................................... 4
9. Don’t be “the coder in the corner.” ...................................................................... 4
10. Critique code instead of people – be kind to the coder, not to the code. .......... 4

# Our Platforms

2.1 Django .............................................................................................................. 5
2.2 Vumi .................................................................................................................. 6
2.3 Logging .............................................................................................................. 6

# Tools we use

3.1 Slack .................................................................................................................. 7
3.2 Git ....................................................................................................................... 7
3.3 Git Flow ............................................................................................................. 7
3.4 Issues & Tickets ............................................................................................... 8
3.5 Sentry ............................................................................................................... 8
3.6 Puppet .............................................................................................................. 8
3.7 Sideloader ........................................................................................................ 9
3.8 Databases / data stores .................................................................................... 9
3.9 Django Applications ......................................................................................... 9
3.10 Translations .................................................................................................... 10
3.11 Graphite ........................................................................................................ 10
3.12 Front-end ....................................................................................................... 10

# Our project process

4.1 Our development process ................................................................................. 11
4.2 Example flow ..................................................................................................... 12
4.3 Our front-end development process ............................................................... 12
4.4 CSS .................................................................................................................. 13
4.5 JavaScript ........................................................................................................ 13
4.6 Contributing back ............................................................................................ 13
5 Things you need to know when starting a project

6 On boarding guide to git flow
   6.1 Overview
   6.2 Required readings
   6.3 Branches
   6.4 Environments
   6.5 Branch name conventions
   6.6 Pull Requests
   6.7 Merging via Pull Requests
   6.8 Commits guideline
   6.9 Init repository with main branches
   6.10 Feature branches flow
   6.11 Hotfix branches flow
   6.12 Release branches flow
   6.13 Use cases

7 Angular Style Guide
   7.1 Community Awesomeness and Credit
   7.2 See the Styles in a Sample App
   7.3 Single Responsibility
   7.4 IIFE
   7.5 Modules
   7.6 Controllers
   7.7 Services
   7.8 Factories
   7.9 Data Services
   7.10 Directives
   7.11 Resolving a Controller
   7.12 Manual Annotating for Dependency Injection
   7.13 Minification and Annotation
   7.14 Exception Handling
   7.15 Naming
   7.16 Application Structure LIFT Principle
   7.17 Application Structure
   7.18 Modularity
   7.19 Startup Logic
   7.20 Angular $ Wrapper Services
   7.21 Testing
   7.22 Animations
   7.23 Comments
   7.24 ESLint
   7.25 Constants
   7.26 Routing
   7.27 Task Automation
   7.28 Filters
   7.29 Angular docs
   7.30 Contributing
   7.31 License

8 Commit message guideline
   8.1 Important
Here are some bits & pieces of documentation to help you get up to speed with our ways of working, the tools we use, processes follow and the things we would be expecting from you when doing projects.

If anything is unclear or plain wrong please do notify us or better yet, submit a pull request to the repository!

Enjoy!
The Ten Commandments of Egoless Programming

From The Psychology of Computer Programming, written in 1971, here are The Ten Commandments of Egoless Programming:

1.1 **Understand and accept that you will make mistakes.**

The point is to find them early, before they make it into production. Fortunately, except for the few of us developing rocket guidance software at JPL, mistakes are rarely fatal in our industry. We can, and should, learn, laugh, and move on.

1.2 **You are not your code.**

Remember that the entire point of a review is to find problems, and problems will be found. Don’t take it personally when one is uncovered.

1.3 **No matter how much “karate” you know, someone else will always know more.**

Such an individual can teach you some new moves if you ask. Seek and accept input from others, especially when you think it’s not needed.

1.4 **Don’t rewrite code without consultation.**

There’s a fine line between “fixing code” and “rewriting code.” Know the difference, and pursue stylistic changes within the framework of a code review, not as a lone enforcer.

1.5 **Treat people who know less than you with respect, deference, and patience.**

Non-technical people who deal with developers on a regular basis almost universally hold the opinion that we are prima donnas at best and crybabies at worst. Don’t reinforce this stereotype with anger and impatience.
1.6 The only constant in the world is change.

Be open to it and accept it with a smile. Look at each change to your requirements, platform, or tool as a new challenge, rather than some serious inconvenience to be fought.

1.7 The only true authority stems from knowledge, not from position.

Knowledge engenders authority, and authority engenders respect — so if you want respect in an egoless environment, cultivate knowledge.

1.8 Fight for what you believe, but gracefully accept defeat.

Understand that sometimes your ideas will be overruled. Even if you are right, don’t take revenge or say “I told you so.” Never make your dearly departed idea a martyr or rallying cry.

1.9 Don’t be “the coder in the corner.”

Don’t be the person in the dark office emerging only for soda. The coder in the corner is out of sight, out of touch, and out of control. This person has no voice in an open, collaborative environment. Get involved in conversations, and be a participant in your office community.

1.10 Critique code instead of people – be kind to the coder, not to the code.

As much as possible, make all of your comments positive and oriented to improving the code. Relate comments to local standards, program specs, increased performance, etc.

For more on Frank Bush, read Frank Bush’s Contributions to the Computing Profession, compiled by his coworkers at TTU.

Copyright (c) 2015 Stephen Wyatt Bush. All rights reserved.
Our Platforms

Praekelt uses two main platforms for the bulk of engineering work:

1. Django
   We use Django for websites, mobi sites, responsive sites, mobi HTML5 apps.
2. Vumi
   We use our Vumi platform for SMS, USSD and other messaging protocols.

Use of any other platform must be approved by our engineering management, and may need to be hosted separately from our usual environments, so please get this sorted out prior to commencing development on a project.

2.1 Django

You should use the latest stable release of Django as of the start of the project unless otherwise specified. Pin that version in your requirements so that the project won’t break by accidently being deployed on a newer version without testing.

We strongly recommend you use our sample django-skeleton as a starting point for Django projects, as it has the layout and configurations we use, so that deployment will be smooth.

We deploy Django in the following stack:

- Ubuntu Server (current LTS release)
- haproxy for load balancing where appropriate
- nginx
- gunicorn
- supervisord
- postgresql

For development, you can simplify this, and for QA we won’t bother about haproxy but the rest of the stack will be required for QA so we recommend you keep your dev environment as close to this as you can.

Notes:

- We manage hostnames in nginx, because there may be multiple QA and live hostnames so don’t use Django’s ALLOWED_HOSTS.
- Make use of pip and virtualenv
• Avoid using CachedStaticFilesStorage, or generating CSS/JS automatically as this breaks load balanced environments.
• There are specific requirements for logging. Please see the Logging section for more information.

2.2 Vumi

Vumi is a scalable messaging engine which we use for SMS, USSD and other messaging protocols. Vumi Go is a hosted version of Vumi. Where Vumi gives you the tools to build large scale messaging applications, Vumi Go provides you with a working environment that is already integrated into numerous countries.

Apps can be written for Vumi Go, to power messaging campaigns or information systems. These apps can be written in Javascript to run in a sandboxed environment (which is our preferred option) or in Python.

See the Vumi Go documentation as well as the JS Sandbox toolkit documentation for writing apps.

2.3 Logging

All logging must happen to a standardized path, under /var/praekelt/logs. If your app has a large amount of files (celery, worker, and app, for instance), write each of these to a named path under /var/praekelt/logs/appname/foo.log, otherwise just /var/praekelt/logs/foo.log is sufficient.

Logs always end with the extension .log. .err is not valid. If you need to write out error logs, either use the name foo-error.log, or write your supervisord configuration with the redirect_stderr option.

The basis for this requirement is to ease debugging (hunting logs in 7 different directories is never fun, and causes issues when under time pressure), simplifies log rotation, and allows them to easily fit within our automatic log collection and indexing system.
Tools we use

The following are tools we use on a regular basis and which we expect you to either use or at least be very familiar with.

### 3.1 Slack

Slack is our team’s communication tool of choice. Join us on (http://edinenie.slack.com).
Various tools report into these channels and provide insight into what is going on.

### 3.2 Git

We use Git. If you work with us, you will use Git for revision control, and Bitbucket. There is no exception.

Provide us with your Bitbucket username and we will provide you with a repository to work on.

**Please** read What’s in a Good Commit? and this Commits guideline for a good introduction to effective use of version control commits.

**Avoid these:**

- Don’t commit merge conflicts. See the Pro Git book on merge conflicts
- Don’t commit snapshots. Only make one change per commit. **Read What’s in a Good Commit above.**
- Don’t commit large content files. Manage them in the CMS.

### 3.3 Git Flow

We use the Git Flow branching model as part of our development. It’s a convenient way to manage your branches. You are not required to use Git Flow but you are required to follow naming conventions it sets with regard to branch-names and prefixes.

Have a read through the blog post describing the general idea and follow the installation instructions in the repository to install it for your development platform of choice.

Unless you’ve explicitly been told otherwise, we require our team to review your code before landing it in the develop branch. Please provide pull requests for our review, the command line tool Hub (see below) is a convenient way of turning GitHub issues into pull-requests.
The pull-request requirement still remains when using Jira. You can still use Hub - however your Jira ticket’s status will not automatically change when the feature branch lands, so you will need to update this yourself.

Please follow On boarding guide to git flow.

For projects with issues tracked in Github issues, We use Hub to interface with GitHub’s API. It allows one to turn issues on GitHub into pull-requests. If that is done then once the pull-request is merged into the main branch the issue is automatically closed.

### 3.4 Issues & Tickets

For project work we use Jira. Only our core open-source platforms maintain their issues in the GitHub repository. You will be given an account to use which will have access to the relevant projects.

For development, if there is no ticket it does not exist. Make sure the work you are doing has a ticket and is being tracked. Stop working and protest immediately if people are treating your mailbox as their ticketing system. We’ve tried that, it does not work.

If a Jira project has a workflow, you need to update your tickets appropriately: New - &gt; Open - &gt; Fixed in dev (when pushed to github) - &gt; Deployed to QA

Our QA team will move the ticket to QA Passed, and our DevOps team will be responsible for the production deployment before the ticket is resolved.

If a ticket is QA Failed then it’s back into your section of the workflow.

A ticket should represent a solid piece of work you intend to do. Make an effort to keep the work you are trying to do in one ticket to no more than 16 hours.

Any estimate you make for actual work done beyond 16 hours is assumed to be

a) largely thumb-suck.

b) going to be very hard to review.

Make an effort to keep it to 16 hours or break it up unto multiple tickets each representing 16 hours of work.

### 3.5 Sentry

We have a dedicated Sentry instance for our projects. You are expected to configure your application to make use of this for error reporting.

You will be given access to your Sentry project and access tokens to will be made available for you to configure your application’s client with.

### 3.6 Puppet

We try and automate as much as possible, this includes our hosting environment. You will need to give us your SSH key so we can provision a machine for your project. Generally you will be given access to a machine that is to be used for QA. Since our DevOps team do the production deployments, and you will get access to production error reports via Sentry, you won’t get access to production without a valid need for troubleshooting, and then it will be without sudo access.

These machines are provisioned using Puppet_. You will not get access to our puppet repository. If you need specific software installed on your machine that it was not provisioned with then please ask for it to be added. Do not install it
yourself without notifying us. This would break our assumption that every machine can be provisioned from scratch with puppet.

If the machine you’ve been working on needs to be rebuilt and you’ve made changes that are not in puppet then it’ll be provisioned without those changes.

3.7 Sideloader

Our DevOps team automate deploys using Sideloader, our tool that creates deb packages from github repos. To enable a repo for this deploy automation, create a .deploy.yaml file in your repository, listing dependencies and scripts.

We then use puppet to install the debs whenever a new one is published. Ask our DevOps team for help with Sideloader, and to set up the puppet automation to install the debs.

We can optionally set up a post commit hook to deploy any changes that are pushed to the develop branch, to QA - if you’re feeling lucky...

See Sideloader help for more info (requires login via github).

3.8 Databases / data stores

We use the following services to store our data. Not all projects will use all of them but generally a number of these will be involved.

1. PostgreSQL
2. Riak
3. Memcached
4. Redis
5. Neo4J

These will be made available to you on a per project basis. Puppet ensures that each of these are backed up.

3.9 Django Applications

For Django applications, some applications are mandatory:

1. Sentry for application reporting.
2. South for managing database schema changes.
3. Nose for running tests.
5. Memcached for caching.

We strongly recommend you use our sample django-skeleton as a starting point for Django projects, as it has some of these already included.
3.10 Translations

We use Gettext or translations in shell scripts, applications and web pages. Read more about Gettext along with some examples on Wikipedia: [http://en.wikipedia.org/wiki/Gettext](http://en.wikipedia.org/wiki/Gettext)

In Django, Gettext is used by default for translations, utilizing ugettext_lazy for models.py and ugettext in other places. We like {% trans %} and {% blocktrans %} tags and enforce these for our open source products.

3.11 Graphite

We use Graphite for the majority of our metric publishing for dashboards. If appropriate, you will be given details for the Graphite server and how metrics are to be published to it.

3.12 Front-end

Sass CSS pre-processor so that we can take advantage of things that CSS doesn’t have yet, or doesn’t do properly: variables; nesting (used sparingly); CSS partials / includes; media queries used more like element queries; mixins.

JavaScript task runners like Grunt and Gulp, with lots of plugins. These handle code linting, image minification, processing Sass into CSS, concatenation and minification of CSS and JS, and running tests. –>
Our project process

The lifecycle of our projects is typically as follows:

1. We produce a Scope of Work for a project, which might not have all the technical details, but should be comprehensive enough to list all the features so that you can quote on the project’s development. Wireframes may also be provided as part of the scope for your CE (Cost Estimate).

2. We work on a fixed cost basis for a fixed scope. If the scope changes, we ask you for a costing for the delta or new work.

3. The authorisation to proceed with work consists of a Purchase Order, without which you cannot invoice us - so never start work without the PO.

4. Development commences - see below. **If you don’t have a github repo by this point, please bug us until we provide it - please do not use your own repo.**

5. We provide you with one QA server, with the same OS setup that we’ll use in production - for all the projects you do for us, unless a project has special needs which justify its own QA server. **Please bug us for a QA URL for this project to be pointed to your QA server.** It must be on our domain for client UAT. Please note that QA may need sample or real data to be populated. Often, the QA data gets migrated to the production site when finally deploying that, so please ensure that dummy data can be cleaned up, and use CMS credentials on QA that are suitable for production.

6. You are responsible for deploying your code to this QA server, so that you can support the fixing of bugs found during our QA testing. You should **always** deploy to QA from the github repo, to avoid any side effects of uncommitted code.

7. We’ll deploy to production so that we can support it - see below.

### 4.1 Our development process

The process involved in how we work is fairly straight forward and we expect you to follow this as well.

1. We use Git Flow’s convention with regard to branch names.

2. All work requires a ticket with a unique number or name.

3. Work happens in a feature branch. Feature branches names are composed of the ticket / issue number along with a one-line description of the issue.

4. Write tests for the new features you are developing.

5. Make one change per commit, and follow What’s in a Good Commit?
6. Put the ticket number in the commit message. For github issues in particular, see Closing issues via commit messages

7. Your schema changes are expected to be handled by a schema migration script.

8. When work in a feature branch is ready for review then we create a pull-request.

9. All collaborators on the GitHub repository are notified of the pull-request and will start the process of reviewing the changes.

10. Any issues, concerns or changes raised or recommended are expected to be attended to. Once done please notify the reviewers of the changes and ask for the changes to be re-reviewed.

11. Once all the changes are approved and one or more of the collaborators has left a :+1: in the pull-request’s comments it can be merged into the main branch and is ready for a deploy.

For your code to be ready for review we have the following expectations:

1. It is to be pep8 compliant and pyflakes is not raising any issues.

2. It is to have tests. Unless otherwise agreed, 90% test coverage is required. See Coverage

3. The tests have to pass.

4. There are no commented lines of code.

5. There is adequate amount of documentation.

### 4.2 Example flow

$ virtualenv ve
$ source ve/bin/activate
(ve)$ git flow feature start issue-1-update-documentation
(ve)$ git flow feature publish issue-1-update-documentation
..// hack hack hack // ..
(ve)$ nosetests
............
Ran 13 tests in 0.194s
OK
(ve)$ git push
(ve)$ hub pull-request -b develop -i 1
https://github.com/praekelt/some-repository/pulls/1
..// review, update, re-eview, update, re-review ... +1 // ..
(ve)$ git flow feature finish issue-1-update-documentation
..// changes merged to develop by git flow // ..
(ve)$ git push

### 4.3 Our front-end development process

We build web sites so that people can access them quickly and easily, regardless of the device they’re using, the type of connection they are on, or any disabilities they have. That means we build things with [Progressive Enhancement](#).

A basic, functional, experience is delivered to everyone; JavaScript is not required for any key functionality. We then do feature tests in JavaScript to load in additional CSS and JS enhancements that we’ve determined the browser can handle. Different browsers will be served different experience; they will be consistent and may be quite similar, but will not be identical.
We also care about front-end performance and accessibility, so we run regular performance and accessibility audits.

4.4 CSS

We keep all our CSS in a few, minified, external CSS files. We don’t use inline style blocks or write inline CSS.

We write our CSS like SMACSS. CSS is organised into: Base; Layout; lots of Modules; States. We keep nesting shallow, and never use IDs for styling.

We make sites Responsive by default as a Future Friendly measure.

We prefer to move into HTML, CSS, and JS sooner rather than later and build Front-end Style Guides (something like Pattern Lab) that evolve into pages and templates.

4.5 JavaScript

We write unobtrusive, js-hinted, JS. We keep all our JS in a few, minified, external JS files. We don’t use inline script blocks or write inline JS. We only include jQuery when really necessary, preferring vanilla JavaScript code and micro-frameworks.

We Cut the Mustard to serve less capable browsers just the core, lighter and faster, experience, rather than send them lots of code they will struggle to run.

4.6 Contributing back

Many of our components in github are open source. In the course of using them, you might find improvements are necessary or possible. We like having your contributions!

Please submit a pull request for our review. Although we don’t recommend it, if you can’t wait for our review and merge, you will need to fork that project on github and submit your changes to us as soon as the pressure is off. Please do create the pull request then.

4.7 Production Deployments

Our DevOps team are responsible for all production deployments. This enables us to support the live sites and systems after hours, and ensure that infrastructural requirements like backups and monitoring are standardised.

Please note that production deployments need to be booked with the DevOps team by the appropriate Praekelt project manager, and that we deploy on Mondays through Thursdays.
Things you need to know when starting a project

Sometimes there’s a rush to get a project started. To spare yourself future trouble here’s a checklist of things you need to have before starting any work.

1. You need to have been issued a purchase order.
2. You need to have been given a Scope of Work describing the thing you are to be building.
3. You need to have agreed to the timelines, estimates and deliverables described in the Scope of Work. If there are any ambiguities in any of the wording they need to be resolved before you start.
4. You need to have a clear picture of which stats need to be collected for the project and how those are to be stored to enable the people wanting access to those stats do the type of analysis they need to do. This differs per project so make sure you take the time to do this properly.
6.1 Overview

Most important thing you should take from this guide is an understanding that git, Github and gitflow is only a tools to achieve your/project goals! Goal of this guide is to explain gitflow, show pros and cons of this approach, and how to prevent possible issues.

6.1.1 Meaning

So the very important thing is to understand main concepts:

- feature - is a complete peace of functionality, it could be a new feature, hotfix, doc, test, refactoring, etc...
- branches - should serve to separate your work by type of goals you want to achieve or to react on issues in production, like hotfixes, with corresponding flows and actions
- pull request - is designed to share the knowledge of what you done with team, to take in count code review comments as a source of knowledge sharing, and last and not least is to check that you did not forgot to add anything :)

6.1.2 Pros and cons

Gitflow is a way to develop things in a team and has it strong and weak parts.

Pros:

- separation of concepts - each branch has clear purpose and flow how to work with it
- confidence - at any given point of time master and release branches are stable and ready for production
- isolation - you develop independent and isolated feature, so you can focus on problem
- continuous processes - continuous deployment and testing for feature branches, continuous integration for master branch
- code review - pull request allows to increase code quality and knowledge rotation in team, and as a consequence reduce “bus factor”
Cons:

- “scary merge” - well illustrated problem by Fowler, to avoid it keep semantic changes as separate branches (explained later)
- “afraid of refactoring” - semantic changes (refactoring, renaming) leads to “scary merges”, so try to do it when code targeted by refactoring is not changed by any other branches
- dependencies (libraries) update - makes Pull Requests view polluted by library files and in general useless, so keep this updates as a separate branches

6.2 Required readings

Take attention to Feature Branching with Pull Requests

1. Try github
2. Atlassian git tutorial (except Migrate to Git from SVN)
3. Feature branching classic by Martin Fowler
4. Closest to ours branching model by Vincent Driessen
5. Vincent Driessen git-flow cheatsheet (ru)

6.2.1 Supporting readings

1. Semantic Versioning 2.0.0
2. Git cheat sheet
3. Github help: user collaborating

6.3 Branches

- master - always stable and release ready branch
- development - default branch, contains latest features and fixes, on which developers should orient
- feature-* - branches for feature development and dependencies update
- release-* - branches for release (production) version
- hotfix-* - branches for release fixes
- refactor-* - branches for semantic changes

6.4 Environments

- master - production
- development - staging
- release-* - for history reasons or supporting several versions at production
- feature-* - continuous deployment to test environment
• **hotfix-** - development (local)

## 6.5 Branch name conventions

Branch names should contain only lowercase letters, numbers and hyphens, regexp: `/[a-z0-9\-]+/`

**Important**: why not with slashes, like in git flow by Vincent Driessen? - because of continuous deployment to corresponding subdomain, for example: `feature-ui-auth` will be deployed to `ui-auth.project.com`, this is why we should use only symbols allowed for DNS names

## 6.6 Pull Requests

- For each pull request it should be an issue (except documentation)
- **Note**: issue should answer on: what you should do and why?
- Pull Request must have a description, which should explain what was done and how it affects project
- **Note**: Pull request description should answer to: what you did and how it affects project
- Fixed issue has to be mentioned at the end of Pull Request Name: “`[fixes #123]`” (same for commit names)
- Pull Requests should be small and focused (see “Commits guideline”)
- All pull request should satisfy Definition of Done(DoD) and Acceptance Criteria used by project’s team
- Commits in pull request should follow to corresponding code conventions

**Important**: keep semantic changes(refactoring, renaming) and library updates as a separate pull requests

## 6.7 Merging via Pull Requests

This flow will be used almost by any branch flow, so read it carefully

1. Create PR from current branch to target branch
   - **Note**: current and target branches depends on what you are doing, see branch flows below
   - **Note**: except hotfix branches, target branch is development
   - assign it to yourself
   - add corresponding labels

1. When you done with changes in branch:
   - **Optional**: you can reset your commits and arrange them in meaningful commits with interactive `git rebase -i`
   - Rebase your branch on top of target branch, usually development
     ```
     # checkout current-branch
     git checkout current-branch
     # fetch latest changes
     git fetch origin --prune
     # rebase on top of development branch
     git rebase origin/development
     ```
• In case of merge conflicts - solve them :)  
  – While rebasing  
    # run merge tool  
    git mergetool  
    # clean up after merge  
    find -name ``*.orig'' -delete  
    # continue rebasing  
    git rebase --continue  
    # repeat this steps until all merging conflicts solved  
  – Abort rebasing and use merge  
    # abort rebase  
    git rebase --abort  
    # merge latest development into your branch  
    git merge origin/development

• Link to PR should be sent to your team chat with code review request (to make things move faster)  

  Sample: “Please review my Pull Request: https://github.com/org/project/pull/1”  

  1. If there are some code review notes which should be applied, apply them and repeat from step 2  
  2. When Pull Request was approved by at least 2 team members  
    • Note: you can use :+1: as an approvement  
    • Note: reviewer, who gave second approvement, can send a note to PR author, to make things go faster  
    • When pull request got 2 approvals it should be merged by code author  

### 6.8 Commits guideline

**Important:**  
• commits without task id (link for asana) are not allowed (except docs and libs version udpates)  
• keep application version bump as a separate commits

Make the changes to the code and tests and then commit to your branch. Be sure to follow the commit message conventions.

Commit message summaries must follow this basic format:  

**Tag:** Message [Action #1234]  

The Tag is one of the following:  
• Fix - for a bug fix.  
• Update - for a backwards-compatible enhancement.  
• Breaking - for a backwards-incompatible enhancement.  
• Docs - changes to documentation only.  
• Build - changes to build process only.  
• New - implemented a new feature.  
• Upgrade - for a dependency upgrade.

**Message** is short description of what have been done **Action** - depends on used task tracking, but in general case:
- **refs** - link commit to task
- **fixes** - closes task, moving it to testing state **Important**: use **fixes** only in Pull Request name

The message summary should be a one-sentence description of the change. The issue number should be mentioned at the end. If the commit doesn’t completely fix the issue, then use [refs #1234] instead of [fixes #1234].

Here are some good commit message summary examples:

**Build**: Update Travis to only test Node 0.10 [refs #734]
**Fix**: Semi rule incorrectly flagging extra semicolon [fixes #840]
**Upgrade**: Esprima to 1.2, switch to using Esprima comment attachment [fixes #730]

The commit message format is important because these messages are used to create a changelog for each release. The tag and issue number help to create more consistent and useful changelogs.

### 6.9 Init repository with main branches

master branch should be in place for any repository by default.

Create development branch and make it as default for GitHub.

```
git checkout -b development origin/master
# switched to a new branch `development`
git push origin development
```

At GitHub:

- Open repository **Settings** -> **Options** -> **Settings** section
- Set development as default branch (it will ease Pull Request creation in future)

### 6.10 Feature branches flow

- **feature** branch should be started from development and merged back.
- should not change version (version will be changed while creating release)

1. Create **feature** branch from development head

```
# update your local repository with latest changes
git fetch origin --prune
# create feature branch from development head
git checkout -b feature-you-will-do origin/development
```

1. Create a Pull Request when feature implementation done (see DoD ...comming)

2. Go through Pull Request review process

3. Cleanup local repository

```
# origin/feature will be removed by Github
git fetch origin --prune
git branch -d feature-you-will-do
```

1. If automatic PR merging not available, merge development to feature branch locally and start from step 3
git fetch origin --prune
git pull origin
git checkout feature-you-will-do
git merge development
# if automatic merging of Pull Request is not available
# it means that there are merge conflicts and we need to solve them locally
# remove trash after merge
find -name `*.orig` -delete
# merge commit will be auto-generated, no need for commit message
git commit
git push origin feature-you-will-do

1. Cleanup branches

   git fetch origin --prune
   git checkout development
   git pull origin development
   # delete remote branch
   git push origin :feature-you-will-do
   # delete local branch
   git branch -d feature-you-will-do

6.11 Hotfix branches flow

- should be used only to fix bugs at production
- **hotfix** branch should be started from master and must be merged back to master and development
- **hotfix** branch may be started from release branch and than must be merged back to release, master and development branches
- **Important**: **hotfix** branch name contains full version number, sample: **hotfix-1.2.2** and **not release-1.2**
- **Important**: each **hotfix** should increase PATCH version of project

   1. Create **hotfix** branch from master (or release)

      git fetch origin --prune
      # create hot fix brach
      git checkout -b hotfix-0.1.2 origin/master

      1. Bump application version and make a separate commit for it

         # bump application PATCH version
         ./bumpApplicationVersion.sh `0.1.2`
         git commit -am `Bumped version number to 0.1.2`

      1. Fix an issue, commit your changes and push to repository

         git commit -am ```Fix: terrible production bug fixed [refs #123]```
         git push origin hotfix-0.1.2

      1. Go through merging process via Pull Request to master

         • **Note**: in case of emergency, approvement and code review may be done later
         • **Note**: if **hotfix** was started from release branch it must be merged to release branch it was started
1. When pull request closed, merge hotfix branch to development and master (if hotfix was started from release branch)

2. Cleanup branches

```bash
git fetch origin --prune
Git checkout development
git pull origin development
# delete remote branch
git push origin :hotfix-0.1.2
# delete local branch
git branch -d hotfix-0.1.2
```

### 6.12 Release branches flow

- should be used when all features required by next release already in development branch
- any release branch should start from development branch
- starting release branch unblocks development for new features development
- merging release branch to master automatically means new release to production, and could be used as an event for continuous integration
- **Important**: all features planed for next release should not be merged in development until current release branch created
- **Important**: release branch name contains only MAJOR and MINOR version number, sample: release-1.2 and not release-1.2.3. PATCH version of project has to be increased only by hotfix branches

1. Create release branch from development head

```bash
# update local version of development to latest state
git checkout origin development
git pull origin development

# create new release branch
git checkout -b release-1.2 development

# bump project version
./bump-project-version.sh 1.2

# keep version bump commit as separate one
git commit -am "Bumped version number to 1.2"

# push your changes to Github
git push origin release-1.2
```

1. Deploy to staging, test on real data, test on appliance to Definition of Done, Acceptance Criteria, Code conventions, etc...
2. When release fully tested and ready for production, start Pull Request to master branch

- **Note**: if you have Continuous Integration, closing Pull Request will start deploy to production, so be ready for fun and choose proper time (not friday evening!)

1. Finish release
• Note: could be merge conflicts

# get latest master state
git checkout master
git pull origin master
# add release tag to merge commit
git tag -a 1.2
# push tag to repository
git push origin master --tags

# merge release back to development
git checkout development
git pull origin development
git merge master
git push origin development --tags

1. Delete release branch and use tags

6.13 Use cases

6.13.1 Remove all local branches

    git branch -l | egrep -v `\^\*\|master\|development` | xargs -n 1 git branch -D
Chapter 7

Angular Style Guide

Opinionated Angular style guide for teams by @valorkin

If you are looking for an opinionated style guide for syntax, conventions, and structuring Angular applications, then step right in. These styles are based on my and Valor Software developers development experience with Angular, presentations, and working in teams.

The purpose of this style guide is to provide guidance on building Angular applications by showing the conventions I use and, more importantly, why I choose them.

7.1 Community Awesomeness and Credit

Fill free to check guidelines used as a basis for this guideline:

- Todd’s guidelines
- John Papa’s guidelines

7.2 See the Styles in a Sample App

Coming soon

7.3 Single Responsibility

7.3.1 Rule of 1

- Define 1 component per file.

The following example defines the app module and its dependencies, defines a controller, and defines a factory all in the same file.

```javascript
/* avoid */
angular
  .module('app', ['ngRoute'])
  .controller('SomeController', SomeController)
  .factory('someFactory', someFactory);

function SomeController() {

```
function someFactory() {}

The same components are now separated into their own files.

```javascript
/* recommended */

// app.module.js
angular
.module(`app', ['ngRoute']);
/* recommended */

// someController.js
angular
 .module(`app')
 .controller(`SomeController', [
  function SomeController() {}
]);
/* recommended */

// someFactory.js
angular
 .module(`app')
 .factory(`someFactory', [
  function someFactory() {}
]);
```

### 7.4 IIFE

#### 7.4.1 Good idea:

- To avoid polluting the global scope with our function declarations that get passed into Angular, ensure build tasks wrap the concatenated files inside an IIFE

#### 7.4.2 But:

- I see wrapping each component in IIFE hard and not necessary. Just keep all required code inside of function passed to Angular.

```javascript
/* avoid */

// someFactory.js
angular
.module(`app')
.factory(`someFactory', [
  function someFactory() {
    this.onClick = sayHello;
  }
]);

function sayHello() {}
```
/* recommended */

// someFactory.js
angular
  .module('app')
  .factory('someFactory', [
    function someFactory() {
      this.onClick = sayHello;

      function sayHello() {} 
    }
  ]);

Original: [Style Y010]

### 7.5 Modules

#### 7.5.1 Avoid Naming Collisions

- Use unique naming conventions with separators for sub-modules.

  **Why?** Unique names help avoid module name collisions. Separators help define modules and their submodule hierarchy. For example `app` may be your root module while `app.dashboard` and `app.users` may be modules that are used as dependencies of `app`.

#### 7.5.2 Definitions (aka Setters)

- Declare modules without a variable using the setter syntax.

  **Why?** With 1 component per file, there is rarely a need to introduce a variable for the module.

/* avoid */
var app = angular.module('app', [ 
  'ngAnimate',
  'ngRoute',
  'app.shared',
  'app.dashboard' 
]);

Instead use the simple setter syntax.

/* recommended */
angular
  .module('app', [ 
    'ngAnimate',
    'ngRoute',
    'app.shared',
    'app.dashboard' 
  ]);;

#### 7.5.3 Getters

- When using a module, avoid using a variable and instead use chaining with the getter syntax.
Why?: This produces more readable code and avoids variable collisions or leaks.

```javascript
/* avoid */
var app = angular.module(`app');
app.controller(`SomeController', [
    function SomeController() { }
]);

/* recommended */
angular
    .module(`app')
    .controller(`SomeController', [
        function SomeController() { }
    ]);"}

7.5.4 Setting vs Getting

- Only set once and get for all other instances.

Why?: A module should only be created once, then retrieved from that point and after.

- Use `angular.module(`app', []);` to set a module.
- Use `angular.module(`app');` to get a module.

7.5.5 Named vs Anonymous Functions

- Use named functions instead of passing an anonymous function in as a callback.

Why?: This produces more readable code, is much easier to debug, and reduces the amount of nested callback code.

```javascript
/* avoid */
angular
    .module(`app')
    .controller(`Dashboard', function() { })
    .factory(`logger', function() { });

/* recommended */

// dashboard.js
angular
    .module(`app')
    .controller(`Dashboard', [ function Dashboard() { } ]);

// logger.js
angular
    .module(`app')
    .factory(`logger', [ function logger() { } ]);"}
7.6 Controllers

7.6.1 controllerAs View Syntax

Important: never use ngController directive

- Use the controllerAs syntax over the classic controller with $scope syntax.

Why?: Controllers are constructed, “newed” up, and provide a single new instance, and the controllerAs syntax is closer to that of a JavaScript constructor than the classic $scope syntax.

Why?: It promotes the use of binding to a “dotted” object in the View (e.g. customer.name instead of name), which is more contextual, easier to read, and avoids any reference issues that may occur without “dotting”.

Why?: Helps avoid using $parent calls in Views with nested controllers.

<!-- avoid -->
<div ng-controller='Customer'>
  {{ name }}
</div>

<!-- recommended - but avoid 'ngController' directive usage -->
<div ng-controller='Customer as customer'>
  {{ customer.name }}
</div>

7.6.2 controllerAs Controller Syntax

- Use the controllerAs syntax over the classic controller with $scope syntax.

- The controllerAs syntax uses this inside controllers which gets bound to $scope.

Why?: controllerAs is syntactic sugar over $scope. You can still bind to the View and still access $scope methods.

Why?: Helps avoid the temptation of using $scope methods inside a controller when it may otherwise be better to avoid them or move them to a factory. Consider using $scope in a factory, or if in a controller just when needed. For example when publishing and subscribing events using $emit, $broadcast, or $on consider moving these uses to a factory and invoke from the controller.

/* avoid */
function Customer($scope) {
  $scope.name = {};
  $scope.sendMessage = function() { };
}

/* recommended - but see next section */
function Customer() {
  this.name = {};
  this.sendMessage = function() { };
}

7.6.3 controllerAs with self

- Use a capture variable for this when using the controllerAs syntax. Choose a variable name such as self to keep this usage consistent.
or: such as *vm*, which stands for ViewModel.

*Why?:* The *this* keyword is contextual and when used within a function inside a controller may change its context. Capturing the context of *this* avoids encountering this problem.

```javascript
/* avoid */
function Customer() {
    this.name = {};
    this.sendMessage = function() { }
}

/* recommended */
function Customer() {
    var self = this;
    self.name = {};
    self.sendMessage = function() { }
}
```

*Note: When creating watches in a controller using `controller as`, you can watch the *ctrl.* member using the following syntax. (Create watches with caution as they add more load to the digest cycle.)*

```javascript
<input ng-model=''someCtrl.title''/>

function SomeController($scope, $log) {
    var self = this;
    self.title = 'Some Title';

    $scope.$watch(`someCtrl.title', function(current, original) {
        $log.info(`someCtrl.title was %s', original);
        $log.info(`someCtrl.title is now %s', current);
    });
}
```

### 7.6.4 Bindable Members Up Top

- Place bindable members at the top of the controller, and not spread through the controller code.

- Proposed order:
  - bind default values
  - bind event handlers
  - initiate controller
  - event handlers declarations

*Why?:* Placing bindable members at the top makes it easy to read and helps you instantly identify which members of the controller can be bound and used in the View.

*Why?:* Setting anonymous functions in-line can be easy, but when those functions are more than 1 line of code they can reduce the readability. Defining the functions below the bindable members (the functions will be hoisted) moves the implementation details down, keeps the bindable members up top, and makes it easier to read.

```javascript
/* avoid */
function Sessions() {
    var self = this;

    self.gotoSession = function() {
```
self.refresh = function() {
    /* ... */
};
self.search = function() {
    /* ... */
};
self.sessions = [];
self.title = 'Sessions';
}

function SessionsController() {
    var self = this;

    // bind default values
    self.sessions = [];
    self.title = 'Sessions';

    // bind event handlers
    self.gotoSession = gotoSession;
    self.refresh = refresh;
    self.search = search;

    // initiate controller
    activate();

    // event handler declarations
    function gotoSession() {
        /* */
    }

    function refresh() {
        /* */
    }

    function search() {
        /* */
    }
}

Note: If the function is a 1 liner consider keeping it right up top, as long as readability is not affected.

/* avoid */
function SessionsController(data) {
    var self = this;

    self.sessions = [];
    self.title = 'Sessions';

    self.gotoSession = gotoSession;
    self.refresh = function() {
        /**
        * lines
```javascript
self.search = search;
/

function SessionsController(dataservice) {

    var self = this;
    self.sessions = [];
    self.title = 'Sessions';

    self.gotoSession = gotoSession;
    self.refresh = dataservice.refresh; // 1 liner is OK
    self.search = search;
}

7.6.5 Function Declarations to Hide Implementation Details

- Use function declarations to hide implementation details. Keep your bindable members up top. When you need to bind a function in a controller, point it to a function declaration that appears later in the file. This is tied directly to the section Bindable Members Up Top. For more details see this post.

  *Why?:* Placing bindable members at the top makes it easy to read and helps you instantly identify which members of the controller can be bound and used in the View. (Same as above.)

  *Why?:* Placing the implementation details of a function later in the file moves that complexity out of view so you can see the important stuff up top.

  *Why?:* Function declaration are hoisted so there are no concerns over using a function before it is defined (as there would be with function expressions).

  *Why?:* You never have to worry with function declarations that moving `var a` before `var b` will break your code because `a` depends on `b`.

  *Why?:* Order is critical with function expressions

    /**
    * avoid
    * Using function expressions.
    */

    function Avengers(dataservice, logger) {
        var self = this;
        self.avengers = [];
        self.title = 'Avengers';

        var activate = function() {
            return getAvengers().then(function() {
                logger.info(`Activated Avengers View`);
            });
        }

        var getAvengers = function() {
        }
    }
```

32 Chapter 7. Angular Style Guide
return dataservice.getAvengers().then(function(data) {
    self.avengers = data;
    return self.avengers;
});
}

self.getAvengers = getAvengers;

activate();

Notice that the important stuff is scattered in the preceding example. In the example below, notice that the important stuff is up top. For example, the members bound to the controller such as `self.avengers` and `self.title`. The implementation details are down below. This is just easier to read.

/*
 * recommend
 * Using function declarations
 * and bindable members up top.
 */
function Avengers(dataservice, logger) {
    var self = this;
    self.avengers = [];
    self.title = 'Avengers';

    self.getAvengers = getAvengers;

    activate();

    function activate() {
        return getAvengers().then(function() {
            logger.info('Activated Avengers View');
        });
    }

    function getAvengers() {
        return dataservice.getAvengers().then(function(data) {
            self.avengers = data;
            return self.avengers;
        });
    }
}

7.6.6 Defer Controller Logic to Services

- Defer logic in a controller by delegating to services and factories.

  * Why?: Logic may be reused by multiple controllers when placed within a service and exposed via a function.
  * Why?: Logic in a service can more easily be isolated in a unit test, while the calling logic in the controller can be easily mocked.
  * Why?: Removes dependencies and hides implementation details from the controller.
  * Why?: Keeps the controller slim, trim, and focused.
/* avoid */
function Order($http, $q, config, userInfo) {
  var self = this;
  self.checkCredit = checkCredit;
  self.isCreditOk;
  self.total = 0;

  function checkCredit() {
    var settings = {};
    // Get the credit service base URL from config
    // Set credit service required headers
    // Prepare URL query string or data object with request data
    // Add user-identifying info so service gets the right credit limit for this user.
    // Use JSONP for this browser if it doesn't support CORS
    return $http.get(settings)
      .then(function(data) {
        // Unpack JSON data in the response object
        // to find maxRemainingAmount
        self.isCreditOk = self.total <= maxRemainingAmount
      })
      .catch(function(error) {
        // Interpret error
        // Cope w/ timeout? retry? try alternate service?
        // Re-reject with appropriate error for a user to see
      });
  }

/* recommended */
function Order(creditService) {
  var self = this;
  self.isCreditOk;
  self.total = 0;

  self.checkCredit = checkCredit;

  function checkCredit() {
    return creditService.isOrderTotalOk(self.total)
      .then(function(isOk) { self.isCreditOk = isOk; })
      .catch(showServiceError);
  }
}

7.6.7 Keep Controllers Focused

- Define a controller for a view, and try not to reuse the controller for other views. Instead, move reusable logic to factories and keep the controller simple and focused on its view.

  Why?: Reusing controllers with several views is brittle and good end to end (e2e) test coverage is required to ensure stability across large applications.
7.6.8 Assigning Controllers

- When a controller must be paired with a view and either component may be re-used by other controllers or views, define controllers along with their routes.

Why?: Pairing the controller in the route allows different routes to invoke different pairs of controllers and views.

```javascript
/* avoid - when using with a route and dynamic pairing is desired */

// route-config.js
angular
  .module('app')
  .config([
    '$routeProvider',
    function config($routeProvider) {
      $routeProvider
        .when('/avengers', {
          templateUrl: 'avengers.html'
        });
    }
  ]);  // avengers.html - avoid -->
</div>

/* recommended */

// route-config.js
angular
  .module('app')
  .config([
    '$routeProvider',
    function config($routeProvider) {
      $routeProvider
        .when('/avengers', {
          templateUrl: 'avengers.html',
          controller: 'AvengersController',
          controllerAs: 'avengersCtrl'
        });
    }
  ]);  // avengers.html -->
</div>
```

7.7 Services

7.7.1 Singletons

- Services are instantiated with the `new` keyword, use `this` for public methods and variables. Since these are so similar to factories, use a factory instead for consistency.

Note: All Angular services are singletons. This means that there is only one instance of a given service per injector.
Note: Ng Source service vs factory

```javascript
// service
angular
  .module(`app')
  .service(`logger', [
    function logger() {
      this.logError = function(msg) {
        /* */
      }
    }
  ]);

// factory
/*/ avoid - returning raw object, see later */
angular
  .module(`app')
  .factory(`logger', [
    function logger() {
      return {
        logError: function(msg) {
          /* */
        }
      }
    }
  ]);
```

### 7.8 Factories

#### 7.8.1 Single Responsibility

- Factories should have a single responsibility, that is encapsulated by its context. Once a factory begins to exceed that singular purpose, a new factory should be created.

#### 7.8.2 Singletons

- Factories are singletons and return an object that contains the members of the service.
  
  Note: All Angular services are singletons.

#### 7.8.3 Accessible Members Up Top

- Expose the callable members of the service (its interface) at the top, using a technique derived from the Revealing Module Pattern.

  Why?: Placing the callable members at the top makes it easy to read and helps you instantly identify which members of the service can be called and must be unit tested (and/or mocked).

  Why?: This is especially helpful when the file gets longer as it helps avoid the need to scroll to see what is exposed.
Why?: Setting functions as you go can be easy, but when those functions are more than 1 line of code they can reduce the readability and cause more scrolling. Defining the callable interface via the returned service moves the implementation details down, keeps the callable interface up top, and makes it easier to read.

```javascript
/* avoid */
function dataService() {
    var someValue = `
    function save() {
        /* */
    }
    function validate() {
        /* */
    }

    return {
        save: save,
        someValue: someValue,
        validate: validate
    };
}

/* recommended */
function dataService() {
    var someValue = `
    var service = {
        save: save,
        someValue: someValue,
        validate: validate
    };
    return service;

    //////////////

    function save() {
        /* */
    }

    function validate() {
        /* */
    }
}
```

This way bindings are mirrored across the host object, primitive values cannot update alone using the revealing module pattern.

### 7.8.4 Function Declarations to Hide Implementation Details

- Use function declarations to hide implementation details. Keep your accessible members of the factory up top. Point those to function declarations that appears later in the file. For more details see this post.

  Why?: Placing accessible members at the top makes it easy to read and helps you instantly identify which functions of the factory you can access externally.

  Why?: Placing the implementation details of a function later in the file moves that complexity out of view so you can see the important stuff up top.
Why?: Function declaration are hoisted so there are no concerns over using a function before it is defined (as there would be with function expressions).

Why?: You never have to worry with function declarations that moving `var a` before `var b` will break your code because `a` depends on `b`.

Why?: Order is critical with function expressions

```javascript
/**
 * avoid
 * Using function expressions
 */
function dataservice($http, $location, $q, exception, logger) {
    var isPrimed = false;
    var primePromise;

    var getAvengers = function() {
        // implementation details go here
    };

    var getAvengerCount = function() {
        // implementation details go here
    };

    var getAvengersCast = function() {
        // implementation details go here
    };

    var prime = function() {
        // implementation details go here
    };

    var ready = function(nextPromises) {
        // implementation details go here
    };

    var service = {
        getAvengersCast: getAvengersCast,
        getAvengerCount: getAvengerCount,
        getAvengers: getAvengers,
        ready: ready
    };

    return service;
}

/**
 * recommended
 * Using function declarations
 * and accessible members up top.
 */
function dataservice($http, $location, $q, exception, logger) {
    var isPrimed = false;
    var primePromise;

    function DataService() {}()
```
DataService.prototype = {
    getAvengersCast: getAvengersCast,
    getAvengerCount: getAvengerCount,
    getAvengers: getAvengers,
    ready: ready
};

return new DataService();

 ///////////////////

function getAvengers() {
    // implementation details go here
}

function getAvengerCount() {
    // implementation details go here
}

function getAvengersCast() {
    // implementation details go here
}

function prime() {
    // implementation details go here
}

function ready(nextPromises) {
    // implementation details go here
}

7.9 Data Services

7.9.1 Separate Data Calls

- Refactor logic for making data operations and interacting with data to a factory. Make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.

  Why?: The controller’s responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the controller be simpler and more focused on the view.

  Why?: This makes it easier to test (mock or real) the data calls when testing a controller that uses a data service.

  Why?: Data service implementation may have very specific code to handle the data repository. This may include headers, how to talk to the data, or other services such as $http. Separating the logic into a data service encapsulates this logic in a single place hiding the implementation from the outside consumers (perhaps a controller), also making it easier to change the implementation.

/* recommended */

// dataservice factory
angular
  .module('app.core')
  .factory('dataservice', [
    '$http', 'logger',
    function dataservice($http, logger) {
      function AvengeresRepository() {} 
      AvengeresRepository.prototype = {
        getAvengers: getAvengers
      };
      return new AvengeresRepository() {} 
    }]
    )

    angular
    .module('app.avengers')
    .controller('Avengers', [
      'dataservice', 'logger',
      function Avengers(dataservice, logger) {
        var self = this;
        self.avengers = [];
        activate();
        function activate() {
          return getAvengers(function() {
            logger.info('Activated Avengers View');
          });
        }
        function getAvengers(cb) {
          return dataservice
            .getAvengers(function(err, data) {
              self.avengers = data;
              return cb(err, data);
            });
        }
    });

Note: The data service is called from consumers, such as a controller, hiding the implementation. /* recommended */

// controller calling the dataservice factory
angular
  .module('app.avengers')
  .controller('Avengers', [
    'dataservice', 'logger',
    function Avengers(dataservice, logger) {
      var self = this;
      self.avengers = [];
      activate();
      function activate() {
        return getAvengers(function() {
          logger.info('Activated Avengers View');
        });
      }
      function getAvengers(cb) {
        return dataservice
          .getAvengers(function(err, data) {
            self.avengers = data;
            return cb(err, data);
          });
      }
    });
7.9.2 NEVER Return a Promise from Data Calls

- When calling a data service that returns a promise such as $http, never return a promise in your calling function.
- Always use callbacks with applying of node.js signature convention function(err, data)
  Why?: It will be really hard to debug promise chains
- Use promises only to integrate with events outside of angular $digest

7.10 Directives

7.10.1 Limit 1 Per File

- Create one directive per file. Name the file for the directive.
  Why?: It is easy to mash all the directives in one file, but difficult to then break those out so some are shared across apps, some across modules, some just for one module.
  Why?: One directive per file is easy to maintain.
  Note: “Best Practice: Directives should clean up after themselves. You can use element.on('$destroy', ...) or scope.$on('$destroy', ...) to run a cleanup function when the directive is removed” ... from the Angular documentation

```javascript
/* avoid */
/* directives.js */

angular
  .module('app.widgets')

  /* order directive that is specific to the order module */
  .directive('orderCalendarRange', orderCalendarRange)

  /* sales directive that can be used anywhere across the sales app */
  .directive('salesCustomerInfo', salesCustomerInfo)

function orderCalendarRange() {
  /* implementation details */
}

function salesCustomerInfo() {
  /* implementation details */
}

/* recommended */
/* calendarRange.directive.js */
```
* @desc order directive that is specific to the order module at a company named Acme
* @example <div acme-order-calendar-range></div>
* /

```javascript
angular
  .module(`sales.order')
  .directive(`acmeOrderCalendarRange', [  
    function orderCalendarRange() {  
      /* implementation details */
    }
  ]);

/**
 * @desc sales directive that can be used anywhere across the sales app at a company named Acme
 * @example <div acme-sales-customer-info></div>
 * /

angular
  .module(`sales.widgets')
  .directive(`acmeSalesCustomerInfo', [  
    function salesCustomerInfo() {  
      /* implementation details */
    }
  ]);
```

Note: There are many naming options for directives, especially since they can be used in narrow or wide scopes. Choose a file name distinct and clear. Some examples are below, but see the [Naming](#naming) section for more recommendations.

### 7.10.2 Manipulate DOM in a Directive

- When manipulating the DOM directly, use a directive. If alternative ways can be used such as using CSS to set styles or the animation services, Angular templating, `ngShow` or `ngHide`, then use those instead. For example, if the directive simply hides and shows, use `ngHide/ngShow`.

  *Why?*: DOM manipulation can be difficult to test, debug, and there are often better ways (e.g. CSS, animations, templates)

### 7.10.3 Provide a Unique Directive Prefix

- Provide a short, unique and descriptive directive prefix such as `acmeSalesCustomerInfo` which would be declared in HTML as `acme-sales-customer-info`.

  *Why?*: The unique short prefix identifies the directive’s context and origin. For example a prefix of `cc-` may indicate that the directive is part of a CodeCamper app while `acme-` may indicate a directive for the Acme company.

  *Note*: Avoid `ng-` as these are reserved for Angular directives. Research widely used directives to avoid naming conflicts, such as `ion-` for the Ionic Framework.

### 7.10.4 Restrict to Elements and Attributes

- When creating a directive that makes sense as a stand-alone element, allow restrict E (custom element) and optionally restrict A (custom attribute). Generally, if it could be its own control, E is appropriate. Generally,
guideline is allow EA but lean towards implementing as an element when it’s stand-alone and as an attribute when it enhances its existing DOM element.

Why?: It makes sense.

Why?: While we can allow the directive to be used as a class, if the directive is truly acting as an element it makes more sense as an element or at least as an attribute.

Note: EA is the default for Angular 1.3+

<!-- avoid -->
<div class='my-calendar-range'></div>
/* avoid */
angular
    .module('app.widgets')
    .directive('myCalendarRange', [
    function myCalendarRange() {
        var directive = {
            link: link,
            templateUrl: '/template/is/located/here.html',
            restrict: 'C'
        };
        return directive;

        function link(scope, element, attrs) {
            /* */
        }
    }]);
<!-- recommended -->
<my-calendar-range></my-calendar-range>
<div my-calendar-range></div>
/* recommended */
angular
    .module('app.widgets')
    .directive('myCalendarRange', [
    function myCalendarRange() {
        var directive = {
            link: link,
            templateUrl: '/template/is/located/here.html',
            restrict: 'EA'
        };
        return directive;

        function link(scope, element, attrs) {
            /* */
        }
    }]);

7.10.5 Directives and ControllerAs

- Use controller as syntax with a directive to be consistent with using controller as with view and controller pairings.
Why?: It makes sense and it’s not difficult.

Note: The directive below demonstrates some of the ways you can use scope inside of link and directive controllers, using controllerAs. I in-lined the template just to keep it all in one place.

Note: Regarding dependency injection, see *Manually Identify Dependencies*.

Note: Note that the directive’s controller is outside the directive’s closure. This style eliminates issues where the injection gets created as unreachable code after a `return`.

```html
<div my-example max="77"></div>
```

```javascript
// my-example-directive.js
angular
  .module(`app')
  .directive(`myExample', [function myExample() {
    var directive = {
      restrict: `EA',
      templateUrl: `app/feature/example.directive.html',
      scope: {
        max: `='
      },
      link: linkFunc,
      controller: ExampleController,
      controllerAs: `ctrl',
      bindToController: true // because the scope is isolated
   );
    return directive;
    function linkFunc(scope, el, attr, ctrl) {
      console.log(`LINK: scope.min = %s *** should be undefined', scope.min);
      console.log(`LINK: scope.max = %s *** should be undefined', scope.max);
      console.log(`LINK: scope.self.min = %s', scope.self.min);
      console.log(`LINK: scope.self.max = %s', scope.self.max);
    }
  }]);
// example-controller.js
angular
  .module(`app')
  .controller(`ExampleController', [`
    $scope',
    function ExampleController($scope) {
      // Injecting $scope just for comparison
      var self = this;

      self.min = 3;

      console.log(`CTRL: $scope.self.min = %s', $scope.self.min);
      console.log(`CTRL: $scope.self.max = %s', $scope.self.max);
      console.log(`CTRL: self.min = %s', self.min);
      console.log(`CTRL: self.max = %s', self.max);
    }
  })
```
<!-- example.directive.html -->
<div>hello world</div>
<div>max={{ctrl.max}}<input ng-model=''ctrl.max''/></div>
<div>min={{ctrl.min}}<input ng-model=''ctrl.min''/></div>

Note: You can also name the controller when you inject it into the link function and access its properties.

// Alternative to above example
function linkFunc(scope, el, attr, self) {
    console.log(`LINK: scope.min = %s *** should be undefined', scope.min);
    console.log(`LINK: scope.max = %s *** should be undefined', scope.max);
    console.log(`LINK: self.min = %s', self.min);
    console.log(`LINK: self.max = %s', self.max);
}

• Use bindToController = true when using controller as syntax with a directive when you want to bind the outer scope to the directive’s controller’s scope.

Why?: It makes it easy to bind outer scope to the directive’s controller scope.

Note: bindToController was introduced in Angular 1.3.0.

<!-- example.directive.html -->
<div my-example max='77'></div>

// my-example-directive.js
angular
 .module(`app')
 .directive(`myExample', [
     function myExample() {
         var directive = {
             restrict: `EA',
             templateUrl: `app/feature/example.directive.html',
             scope: {
                 max: `='
             },
             controller: ExampleController,
             controllerAs: `ctrl',
             bindToController: true
         };

         return directive;
     }
 ]);

// example-controller.js
angular
 .module(`app')
 .controller(`ExampleController', [
     function ExampleController() {
         var self = this;
         self.min = 3;
         console.log(`CTRL: self.min = %s', self.min);
         console.log(`CTRL: self.max = %s', self.max);
     }
 ]);

<!-- example.directive.html -->
<div>hello world</div>

7.10. Directives
7.11 Resolving a Controller

7.11.1 Controller Activation

• Resolve start-up logic for a controller in an `activate` function.

  *Why?* Placing start-up logic in a consistent place in the controller makes it easier to locate, more consistent to test, and helps avoid spreading out the activation logic across the controller.

  *Why?* The controller `activate` makes it convenient to re-use the logic for a refresh for the controller/View, keeps the logic together, gets the user to the View faster, makes animations easy on the `ng-view` or `ui-view`, and feels snappier to the user.

  *Note:* If you need to conditionally cancel the route before you start use the controller, use a `canActive` approach introduced in `angular 1.4`.

```javascript
/* avoid */
function Avengers(dataservice) {
  var self = this;
  self.avengers = [];
  self.title = `Avengers';

  dataservice.getAvengers().then(function(data) {
    self.avengers = data;
    return self.avengers;
  });
}

/* recommended */
function Avengers(dataservice) {
  var self = this;
  self.avengers = [];
  self.title = `Avengers';

  activate();

  /////////// 

  function activate() {
    return dataservice.getAvengers(function(err, data) {
      self.avengers = data;
    });
  }
}
```
7.11.2 Route Resolve Promises

7.12 Manual Annotating for Dependency Injection

7.12.1 UnSafe from Minification

- Avoid using the shortcut syntax of declaring dependencies without using a minification-safe approach.

  Why?: The parameters to the component (e.g. controller, factory, etc) will be converted to mangled variables. For example, common and dataservice may become a or b and not be found by Angular.

  /* avoid - not minification-safe*/
  angular
  .module('app')
  .controller('Dashboard', Dashboard);

  function Dashboard(common, dataservice) {
    // This code may produce mangled variables when minified and thus cause runtime errors.
  }

  /* avoid - not minification-safe*/
  angular.module('app').controller('Dashboard', d);function d(a, b) {

7.12.2 Manually Identify Dependencies

- Use the inline array annotation (preferred)
- Always use ngStrictDi with ngApp to avoid unexpected behaviour after minification

  Why?: This safeguards your dependencies from being vulnerable to minification issues when parameters may be mangled. For example, common and dataservice may become a or b and not be found by Angular.

  /* avoid */
  angular
  .module('app')
  .controller('Dashboard', Dashboard);

  Dashboard.$inject = ['$location', '$routeParams', 'common', 'dataservice'];

  function Dashboard($location, $routeParams, common, dataservice) {
    // /* recommended */
    angular
    .module('app')
    .controller('Dashboard', ['
      $location', '$routeParams', 'common', 'dataservice',
      function Dashboard($location, $routeParams, common, dataservice) {}
    ]);}

7.13 Minification and Annotation

TODO
7.14 Exception Handling

7.14.1 decorators

- Use a decorator, at config time using the `$provide` service, on the `$exceptionHandler` service to perform custom actions when exceptions occur.

**Why?** Provides a consistent way to handle uncaught Angular exceptions for development-time or run-time.

Note: Another option is to override the service instead of using a decorator. This is a fine option, but if you want to keep the default behavior and extend it a decorator is recommended.

```javascript
/* recommended */
angular
  .module(`blocks.exception')
  .config([`
    `$provide',
    function exceptionConfig($provide) {
      $provide.decorator(`$exceptionHandler', `[`$delegate', `toastr',
        function extendExceptionHandler($delegate, toastr) {
          return function(exception, cause) {
            $delegate(exception, cause);
            var errorData = {
              exception: exception,
              cause: cause
            };
            /**
             * Could add the error
             * to a service's collection,
             * add errors to $rootScope,
             * log errors to remote web server,
             * or log locally.
             * Or throw hard. It is entirely up to you.
             * throw exception;
             */
            toastr.error(exception.msg, errorData);
          }
        }
    ]);
  ]);`
}
```

7.14.2 Exception Catchers

- Create a factory that exposes an interface to catch and gracefully handle exceptions.

**Why?** Provides a consistent way to catch exceptions that may be thrown in your code (e.g. during XHR calls or promise failures).

Note: The exception catcher is good for catching and reacting to specific exceptions from calls that you know may throw one. For example, when making an XHR call to retrieve data from a remote web service and you want to catch any exceptions from that service and react uniquely.

```javascript
/* recommended */
```
angular
    .module('blocks.exception')
    .factory('exception', [
        'logger',
        function exception(logger) {
            function ExceptionLogger() {}
            ExceptionLogger.prototype = {
                catcher: catcher
            };
            return new ExceptionLogger();

            function catcher(message) {
                return function(reason) {
                    logger.error(message, reason);
                };
            }
        }
    ]);

7.14.3 Route Errors

- Handle and log all routing errors using $routeChangeError.

  Why?: Provides a consistent way to handle all routing errors.
  Why?: Potentially provides a better user experience if a routing error occurs and you route them to a friendly screen with more details or recovery options.

  /* recommended */
  var handlingRouteChangeError = false;

  function handleRoutingErrors() {
      /**
       * Route cancellation:
       * On routing error, go to the dashboard.
       * Provide an exit clause if it tries to do it twice.
       */
      $rootScope.$on('$routeChangeError',
          function(event, current, previous, rejection) {
              if (handlingRouteChangeError) { return; }
              handlingRouteChangeError = true;
              var destination = (current && (current.title ||
                  current.name || current.loadedTemplateUrl)) ||
                  'unknown target';
              var msg = `Error routing to ` + destination + `. ` +
                  (rejection.msg || '');

              /**
               * Optionally log using a custom service or $log.
               * (Don't forget to inject custom service)
               */
              logger.warning(msg, [current]);

              /**
               * On routing error, go to another route/state.
               */
      });
7.15 Naming

7.15.1 Naming Guidelines

- Use consistent names for all components following a pattern that describes the component’s feature then its type. My recommended pattern is feature.type.js. There are 2 names for most assets:
  - the file name (avengers.controller.js)
  - the registered component name with Angular (AvengersController)

*Why?:* Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.

*Why?:* The naming conventions should simply help you find your code faster and make it easier to understand.

7.15.2 Feature File Names

- Use consistent names for all components following a pattern that describes the component’s feature then its type. My recommended pattern is feature.type.js.

*Why?:* Provides a consistent way to quickly identify components.

*Why?:* Provides pattern matching for any automated tasks.

```javascript
/**
 * common options
 */

// Controllers
avengers.controller.js
avengersController.js

// Services/Factories
logger.service.js
loggerService.js

/**
 * recommended
 */

// controllers
avengers.controller.js
avengers.controller.spec.js

// services/factories
logger.service.js
```
// constants
constants.js

// module definition
avengers.module.js

// routes
avengers.routes.js
avengers.routes.spec.js

// configuration
avengers.config.js

// directives
avenger-profile.directive.js
avenger-profile.directive.spec.js

### 7.15.3 Test File Names

- Name test specifications similar to the component they test with a suffix of `spec`.  
  *Why?: Provides a consistent way to quickly identify components.*
  
  *Why?: Provides pattern matching for `karma` or other test runners.*
  
  ```
  /**
   * recommended
   */
  avengers.controller.spec.js
  logger.service.spec.js
  avengers.routes.spec.js
  avenger-profile.directive.spec.js
  ```

### 7.15.4 Controller Names

- Use consistent names for all controllers named after their feature. Use UpperCamelCase for controllers, as they are constructors.  
  *Why?: Provides a consistent way to quickly identify and reference controllers.*
  
  *Why?: UpperCamelCase is conventional for identifying object that can be instantiated using a constructor.*
  
  ```
  /**
   * recommended
   */
  
  // hero-avengers.controller.js
  angular
    .module
      .controller('HeroAvengersController', [ 
        function HeroAvengersController() { }
      ]);
7.15.5 Controller Name Suffix

- Append the controller name with the suffix `Controller`
- Don’t use `Ctrl`, it is good only for tutorials

*Why?* The `Controller` suffix is more commonly used and is more explicitly descriptive.

```javascript
/**
 * recommended
 */

// avengers.controller.js
angular
  .module
    .controller('AvengersController', [
      function AvengersController() { }
    ]);
```

7.15.6 Factory Names

- Use consistent names for all factories named after their feature. Use camel-casing for services and factories. Avoid prefixing factories and services with `$`.
- Append the factory name with the suffix `Service`

*Why?* Provides a consistent way to quickly identify and reference factories.

*Why?* Avoids name collisions with built-in factories and services that use the `$` prefix.

```javascript
/**
 * recommended
 */

// logger.service.js
angular
  .module
    .factory('loggerService', [
      function loggerService() { }
    ]);
```

7.15.7 Directive Component Names

- Use consistent names for all directives using camel-case. Use a short prefix to describe the area that the directives belong (some example are company prefix or project prefix).

*Why?* Provides a consistent way to quickly identify and reference components.

```javascript
/**
 * recommended
 */

// avenger-profile.directive.js
angular
  .module
```
7.15.8 Modules

- When there are multiple modules, the main module file is named `app.module.js` while other dependent modules are named after what they represent. For example, an admin module is named `admin.module.js`. The respective registered module names would be `app` and `admin`.

  Why?: Provides consistency for multiple module apps, and for expanding to large applications.

  Why?: Provides easy way to use task automation to load all module definitions first, then all other angular files (for bundling).

7.15.9 Configuration

- Separate configuration for a module into its own file named after the module. A configuration file for the main `app` module is named `app.config.js` (or simply `config.js`). A configuration for a module named `admin.module.js` is named `admin.config.js`.

  Why?: Separates configuration from module definition, components, and active code.

  Why?: Provides an identifiable place to set configuration for a module.

7.15.10 Routes

- Separate route configuration into its own file. Examples might be `app.route.js` for the main module and `admin.route.js` for the `admin` module. Even in smaller apps I prefer this separation from the rest of the configuration.

7.16 Application Structure LIFT Principle

7.16.1 LIFT

- Structure your app such that you can locate your code quickly, identify the code at a glance, keep the flattest structure you can, and try to stay DRY. The structure should follow these 4 basic guidelines.

  Why LIFT?: Provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. Another way to check your app structure is to ask yourself: How quickly can you open and work in all of the related files for a feature?

When I find my structure is not feeling comfortable, I go back and revisit these LIFT guidelines

1. Locating our code is easy
2. Identify code at a glance
3. Flat structure as long as we can
4. Try to stay DRY (Don't Repeat Yourself) or T-DRY
7.16.2 Locate

- Make locating your code intuitive, simple and fast.

*Why?* I find this to be super important for a project. If the team cannot find the files they need to work on quickly, they will not be able to work as efficiently as possible, and the structure needs to change. You may not know the file name or where its related files are, so putting them in the most intuitive locations and near each other saves a ton of time. A descriptive folder structure can help with this.

```plaintext
/public
/assets
/components
/avengers
/blocks
    /exception
    /logger
/core
/dashboard
/data
/layout
/shared
/widgets
/content
/libs <!-- bower_components -->
index.html
.bower.json
```

7.16.3 Identify

- When you look at a file you should instantly know what it contains and represents.

*Why?* You spend less time hunting and pecking for code, and become more efficient. If this means you want longer file names, then so be it. Be descriptive with file names and keeping the contents of the file to exactly 1 component. Avoid files with multiple controllers, multiple services, or a mixture. There are deviations of the 1 per file rule when I have a set of very small features that are all related to each other, they are still easily identifiable.

7.16.4 Flat

- Keep a flat folder structure as long as possible. When you get to 7+ files, begin considering separation.

*Why?* Nobody wants to search 7 levels of folders to find a file. Think about menus on web sites … anything deeper than 2 should take serious consideration. In a folder structure there is no hard and fast number rule, but when a folder has 7-10 files, that may be time to create subfolders. Base it on your comfort level. Use a flatter structure until there is an obvious value (to help the rest of LIFT) in creating a new folder.

7.16.5 T-DRY (Try to Stick to DRY)

- Be DRY, but don’t go nuts and sacrifice readability.

*Why?* Being DRY is important, but not crucial if it sacrifices the others in LIFT, which is why I call it T-DRY. I don’t want to type session-view.html for a view because, well, it’s obviously a view. If it is not obvious or by convention, then I name it.
7.17 Application Structure

7.17.1 Overall Guidelines

- Have a near term view of implementation and a long term vision. In other words, start small and but keep in mind on where the app is heading down the road. All of the app’s code goes in a root folder named `components`. All content is 1 feature per file. Each controller, service, module, view is in its own file. All 3rd party vendor scripts are stored in another root folder and not in the `components` folder. I didn’t write them and I don’t want them cluttering my app (`bower_components`, `scripts`, `libs`).

Note: Find more details and reasoning behind the structure at this original post on application structure.

Note: Small and good article about components oriented structure

7.17.2 Layout

- Place components that define the overall layout of the application in a folder named `layout`. These may include a shell view and controller may act as the container for the app, navigation, menus, content areas, and other regions.

*Why?:* Organizes all layout in a single place re-used throughout the application.

7.17.3 Folders-by-Feature Structure

- Create folders named for the feature they represent. When a folder grows to contain more than 7 files, start to consider creating a folder for them. Your threshold may be different, so adjust as needed.

*Why?:* A developer can locate the code, identify what each file represents at a glance, the structure is flat as can be, and there is no repetitive nor redundant names.

*Why?:* The LIFT guidelines are all covered.

*Why?:* Helps reduce the app from becoming cluttered through organizing the content and keeping them aligned with the LIFT guidelines.

*Why?:* When there are a lot of files (10+) locating them is easier with a consistent folder structures and more difficult in flat structures.

```*/
 * recommended
 */

app/
  app.module.js
  app.config.js
components/
  layouts/
    shell/
      shell.html
      shell.controller.js
  topnav/
    topnav.html
    topnav.controller.js
people/
  people.routes.js
attendees/
   attendees.html
   attendees.controller.js
speakers/
   speakers.html
   speakers.controller.js
speaker-detail/
   speaker-detail.html
   speaker-detail.controller.js
shared/
   user-profile/
      user-profile.directive.js
      user-profile.directive.html
calendar/
   calendar.directive.js
   calendar.directive.html
sessions/
   sessions.html
   sessions.controller.js
   sessions.routes.js
session-detail/
   session-detail.html
   session-detail.controller.js

Note: **Do not use structuring using folders-by-type.** This requires moving to multiple folders when working on a feature and gets unwieldy quickly as the app grows to 5, 10 or 25+ views and controllers (and other features), which makes it more difficult than folder-by-feature to locate files.

```*/
* avoid
* Alternative folders-by-type.
* I recommend `\`folders-by-feature``', instead.
*/
```

app/
   app.module.js
   app.config.js
   app.routes.js
controllers/
   attendees.js
   session-detail.js
   sessions.js
   shell.js
   speakers.js
   speaker-detail.js
topnav.js
directives/
   calendar.directive.js
   calendar.directive.html
   user-profile.directive.js
   user-profile.directive.html
services/
   dataservice.js
   localstorage.js
   logger.js
7.18 Modularity

7.18.1 Many Small, Self Contained Modules

- Create small modules that encapsulate one responsibility.

  Why?: Modular applications make it easy to plug and go as they allow the development teams to build vertical slices of the applications and roll out incrementally. This means we can plug in new features as we develop them.

7.18.2 Create an App Module

- Create an application root module whose role is pull together all of the modules and features of your application. Name this for your application.

  Why?: Angular encourages modularity and separation patterns. Creating an application root module whose role is to tie your other modules together provides a very straightforward way to add or remove modules from your application.

7.18.3 Keep the App Module Thin

- Only put logic for pulling together the app in the application module. Leave features in their own modules.

  Why?: Adding additional roles to the application root to get remote data, display views, or other logic not related to pulling the app together muddies the app module and make both sets of features harder to reuse or turn off.

  Why?: The app module becomes a manifest that describes which modules help define the application.

7.18.4 Feature Areas are Modules

- Create modules that represent feature areas, such as layout, reusable and shared services, dashboards, and app specific features (e.g. customers, admin, sales).

  Why?: Self contained modules can be added to the application with little or no friction.

  Why?: Sprints or iterations can focus on feature areas and turn them on at the end of the sprint or iteration.

  Why?: Separating feature areas into modules makes it easier to test the modules in isolation and reuse code.
7.18.5 Reusable Blocks are Modules

- Create modules that represent reusable application blocks for common services such as exception handling, logging, diagnostics, security, and local data stashing.

Why?: These types of features are needed in many applications, so by keeping them separated in their own modules they can be application generic and be reused across applications.

7.18.6 Module Dependencies

- The application root module depends on the app specific feature modules and any shared or reusable modules.

Why?: The main app module contains a quickly identifiable manifest of the application’s features.

Why?: Each feature area contains a manifest of what it depends on, so it can be pulled in as a dependency in other applications and still work.

Why?: Intra-App features such as shared data services become easy to locate and share from within `app.core` (choose your favorite name for this module).

Note: This is a strategy for consistency. There are many good options here. Choose one that is consistent, follows Angular’s dependency rules, and is easy to maintain and scale.

My structures vary slightly between projects but they all follow these guidelines for structure and modularity. The implementation may vary depending on the features and the team. In other words, don’t get hung up on an exact like-for-like structure but do justify your structure using consistency, maintainability, and efficiency in mind.

In a small app, you can also consider putting all the shared dependencies in the app module where the feature modules have no direct dependencies. This makes it easier to maintain the smaller application, but makes it harder to reuse modules outside of this application.

7.19 Startup Logic

7.19.1 Configuration

- Inject code into module configuration that must be configured before running the angular app. Ideal candidates include providers and constants.

Why?: This makes it easier to have a less places for configuration.

```javascript
angular
.module('app')
.config([
  'routerHelperProvider', 'exceptionHandlerProvider', 'toastr',
  function configure (routerHelperProvider, exceptionHandlerProvider, toastr) {
    exceptionHandlerProvider.configure(config.appErrorPrefix);
    configureStateHelper();
    toastr.options.timeOut = 4000;
    toastr.options.positionClass = 'toast-bottom-right';

    ///////////////
    function configureStateHelper() {
      routerHelperProvider.configure({
```
7.19.2 Run Blocks

- Any code that needs to run when an application starts should be declared in a factory, exposed via a function, and injected into the run block.

Why?: Code directly in a run block can be difficult to test. Placing in a factory makes it easier to abstract and mock.

```javascript
angular
    .module(`app')
    .run([`
        `authenticator', `translator',
        function runBlock(authenticator, translator) {
            authenticator.initialize();
            translator.initialize();
        }
    ]);
```

7.20 Angular $ Wrapper Services

7.20.1 $document and $window

- Use $document and $window instead of document and window.

Why?: These services are wrapped by Angular and more easily testable than using document and window in tests. This helps you avoid having to mock document and window yourself.

7.20.2 $timeout and $interval

- Use $timeout and $interval instead of setTimeout and setInterval.

Why?: These services are wrapped by Angular and more easily testable and handle Angular’s digest cycle thus keeping data binding in sync.

7.21 Testing

Unit testing helps maintain clean code, as such I included some of my recommendations for unit testing foundations with links for more information.

7.21.1 Write Tests with Stories

- Write a set of tests for every story. Start with an empty test and fill them in as you write the code for the story.
Why?: Writing the test descriptions helps clearly define what your story will do, will not do, and how you can measure success.

```javascript
it(`should have Avengers controller', function() {
    // TODO
});

it(`should find 1 Avenger when filtered by name', function() {
    // TODO
});

it(`should have 10 Avengers', function() {
    // TODO (mock data?)
});

it(`should return Avengers via XHR', function() {
    // TODO ($httpBackend?)
});

// and so on
```

### 7.21.2 Testing Library

- Use Mocha for unit testing.
  
  Note: When using Mocha, also consider choosing an assert library such as Chai.

### 7.21.3 Test Runner

- Use Karma as a test runner.
  
  Why?: Karma is easy to configure to run once or automatically when you change your code.
  
  Why?: Karma hooks into your Continuous Integration process easily on its own or through Grunt or Gulp.
  
  Why?: Some IDE’s are beginning to integrate with Karma, such as WebStorm
  
  Why?: Karma works well with task automation leaders such as Gulp (with gulp-karma).

### 7.21.4 Stubbing and Spying

- Use Sinon for stubbing and spying.
  
  Why?: Sinon works well with both Jasmine and Mocha and extends the stubbing and spying features they offer.
  
  Why?: Sinon makes it easier to toggle between Jasmine and Mocha, if you want to try both.
  
  Why?: Sinon has descriptive messages when tests fail the assertions.

### 7.21.5 Headless Browser

- Use PhantomJS to run your tests on a server.
  
  Why?: PhantomJS is a headless browser that helps run your tests without needing a “visual” browser. So you do not have to install Chrome, Safari, IE, or other browsers on your server.
Note: You should still test on all browsers in your environment, as appropriate for your target audience.

### 7.21.6 Code Analysis

- Run ESLint on your tests.

  *Why*: Tests are code. ESLint can help identify code quality issues that may cause the test to work improperly.

### 7.21.7 Alleviate Globals for ESLint Rules on Tests

- Relax the rules on your test code to allow for common globals such as `describe` and `expect`. Relax the rules for expressions, as Mocha uses these.

  *Why*: Your tests are code and require the same attention and code quality rules as all of your production code. However, global variables used by the testing framework. Add the following to your JSHint Options file.

    ```
    `jasmine'': true,
    `mocha'': true
    ```

### 7.21.8 Organizing Tests

- Place unit test files (specs) side-by-side with your client code. Place specs that cover server integration or test multiple components in a separate `tests` folder.

  *Why*: Unit tests have a direct correlation to a specific component and file in source code.

  *Why*: It is easier to keep them up to date since they are always in sight. When coding whether you do TDD or test during development or test after development, the specs are side-by-side and never out of sight nor mind, and thus more likely to be maintained which also helps maintain code coverage.

  *Why*: When you update source code it is easier to go update the tests at the same time.

  *Why*: Placing them side-by-side makes it easy to find them and easy to move them with the source code if you move the source.

  *Why*: Having the spec nearby makes it easier for the source code reader to learn how the component is supposed to be used and to discover its known limitations.

  *Why*: Separating specs so they are not in a distributed build is easy with grunt or gulp.

    `/src/client/app/customers/customer-detail.controller.js`
    `/customer-detail.controller.spec.js`
    `/customers.controller.spec.js`
    `/customers.controller-detail.spec.js`
    `/customers.module.js`
    `/customers.route.js`
    `/customers.route.spec.js`

### 7.22 Animations

#### 7.22.1 Usage

- Use subtle animations with Angular to transition between states for views and primary visual elements. Include the `ngAnimate` module. The 3 keys are subtle, smooth, seamless.
Why?: Subtle animations can improve User Experience when used appropriately.
Why?: Subtle animations can improve perceived performance as views transition.

### 7.22.2 Sub Second

- Use short durations for animations. I generally start with 300ms and adjust until appropriate.
  
  Why?: Long animations can have the reverse effect on User Experience and perceived performance by giving
  the appearance of a slow application.

### 7.22.3 animate.css

- Use animate.css for conventional animations.
  
  Why?: The animations that animate.css provides are fast, smooth, and easy to add to your application.
  
  Why?: Provides consistency in your animations.
  
  Why?: animate.css is widely used and tested.
  
  Note: See this great post by Matias Niemelä on Angular animations

### 7.23 Comments

#### 7.23.1 jsDoc

- If planning to produce documentation, use jsDoc syntax to document function names, description, params and
  returns. Use @namespace and @memberOf to match your app structure.
  
  Why?: You can generate (and regenerate) documentation from your code, instead of writing it from scratch.
  
  Why?: Provides consistency using a common industry tool.

```javascript
/**
 * Logger Factory
 * @namespace Factories
 */
(function() {
    angular
        .module('app')
        .factory('logger', logger);

    /**
     * @namespace Logger
     * @desc Application wide logger
     * @memberOf Factories
     */
    function logger($log) {
        var service = {
            logError: logError
        };
        return service;
    }

    /////////
/**
 * @name logError
 * @desc Logs errors
 * @param {String} msg Message to log
 * @returns {String}
 * @memberOf Factories.Logger
 */
function logError(msg) {
    var loggedMsg = `Error: ` + msg;
    $log.error(loggedMsg);
    return loggedMsg;
}

7.24 ESLint

7.24.1 Use an options file

- Use ESLint for linting and checking your coding styles of your JavaScript and be sure to customize the ESLint options file and include in source control. See the ESLint docs for details on the options.

Why?: Provides a first alert prior to committing any code to source control.

Why?: Provides consistency across your team.

Latest eslint config you can find here

7.25 Constants

7.25.1 Vendor Globals

- Create an Angular Constant for vendor libraries’ global variables.

Why?: Provides a way to inject vendor libraries that otherwise are globals. This improves code testability by allowing you to more easily know what the dependencies of your components are (avoids leaky abstractions). It also allows you to mock these dependencies, where it makes sense.

// constants.js

/* eslint-env toastr, moment */
angular
    .module(`app.core'
    .constant(`toastr', toastr)
    .constant(`moment', moment);

- Use constants for values that do not change and do not come from another service. When constants are used only for a module that may be reused in multiple applications, place constants in a file per module named after the module. Until this is required, keep constants in the main module in a constants.js file.

Why?: A value that may change, even infrequently, should be retrieved from a service so you do not have to change the source code. For example, a url for a data service could be placed in a constants but a better place would be to load it from a web service.
**Why?**: Constants can be injected into any angular component, including providers.

**Why?**: When an application is separated into modules that may be reused in other applications, each stand-alone module should be able to operate on its own including any dependent constants.

```javascript
// Constants used by the entire app
angular
  .module('app.core')
  .constant('moment', moment);

// Constants used only by the sales module
angular
  .module('app.sales')
  .constant('events', {
    ORDER_CREATED: 'event_order_created',
    INVENTORY_DEPLETED: 'event_inventory_depleted'
  });
```

- Download the [WebStorm Angular file templates and snippets](assets/webstorm-angular-file-template.settings.jar?raw=true)
- Open WebStorm and go to the `File` menu
- Choose the `Import Settings` menu option
- Select the file and click `OK`
- In a JavaScript file type these commands followed by a `TAB`:

```
```

```javascript
ng-c // creates an Angular controller
ng-f // creates an Angular factory
ng-m // creates an Angular module
```

-->  

## 7.26 Routing

Client-side routing is important for creating a navigation flow between views and composing views that are made of many smaller templates and directives.

- Use the [AngularUI Router](https://angular-ui.github.io/ui-router/) for client-side routing.
  
  **Why?**: UI Router offers all the features of the Angular router plus a few additional ones including nested routes and states.

  **Why?**: The syntax is quite similar to the Angular router and is easy to migrate to UI Router.

- Define routes for views in the module where they exist. Each module should contain the routes for the views in the module.

  **Why?**: Each module should be able to stand on its own.

  **Why?**: When removing a module or adding a module, the app will only contain routes that point to existing views.

  **Why?**: This makes it easy to enable or disable portions of an application without concern over orphaned routes.
7.27 Task Automation

Use **Gulp** for creating automated tasks.

- Use task automation to list module definition files `*.module.js` before all other application JavaScript files.
  
  **Why?** Angular needs the module definitions to be registered before they are used.

  **Why?** Naming modules with a specific pattern such as `*.module.js` makes it easy to grab them with a glob and list them first.

  ```javascript
  var clientApp = `./src/client/app/';

  // Always grab module files first
  var files = [
    clientApp + `**/*.module.js',
    clientApp + `**/*.js'
  ];
  ```

7.28 Filters

- Avoid using filters for scanning all properties of a complex object graph. Use filters for select properties.

  **Why?** Filters can easily be abused and negatively effect performance if not used wisely, for example when a filter hits a large and deep object graph.

7.29 Angular docs

For anything else, API reference, check the Angular documentation.

7.30 Contributing

Open an issue first to discuss potential changes/additions. If you have questions with the guide, feel free to leave them as issues in the repository. If you find a typo, create a pull request. The idea is to keep the content up to date and use github’s native feature to help tell the story with issues and PR’s, which are all searchable via google. Why? Because odds are if you have a question, someone else does too! You can learn more here at about how to contribute.

*By contributing to this repository you are agreeing to make your content available subject to the license of this repository.*

7.30.1 Process

1. Discuss the changes in a GitHub issue.
2. Open a Pull Request, reference the issue, and explain the change and why it adds value.
3. The Pull Request will be evaluated and either merged or declined.

7.31 License

*tl;dr: Use this guide. Attributions are appreciated.*
7.31.1 Copyright

Copyright (c) 2015 valorkin Copyright (c) 2014-2015 John Papa

7.31.2 (The MIT License)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ‘Software’), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ‘AS IS’, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
8.1 Important

- commits without task id (link for asana) are not allowed

Make the changes to the code and tests and then commit to your branch. Be sure to follow the commit message conventions.

Commit message summaries must follow this basic format:

Tag: Message [Action #1234]

The Tag is one of the following:

- Fix - for a bug fix.
- Update - for a backwards-compatible enhancement.
- Breaking - for a backwards-incompatible enhancement.
- Docs - changes to documentation only.
- Build - changes to build process only.
- New - implemented a new feature.
- Upgrade - for a dependency upgrade.

Message is short description of what have been done Action - depends on used task tracking, but in general case:

- refs or ref - link commit to task
- fixes or fix - closes task, moving it to testing state

The message summary should be a one-sentence description of the change. The issue number should be mentioned at the end. If the commit doesn’t completely fix the issue, then use [refs #1234] instead of [fixes #1234].

Here are some good commit message summary examples:

Build: Update Travis to only test Node 0.10 [refs #734]
Fix: Semi rule incorrectly flagging extra semicolon [fixes #840]
Upgrade: Esprima to 1.2, switch to using Esprima comment attachment [fixes #730]

The commit message format is important because these messages are used to create a changelog for each release. The tag and issue number help to create more consistent and useful changelogs.

Copyright (c) 2013 Nicholas C. Zakas. All rights reserved.