Coding Guidelines:-

Common

- Follow the Angular Styleguide
- Every component should have its own folder. Any directive and/or service related to the component should be placed in the same folder as the component.
- Unsubscribe Custom Observables
- Every component should have its own module. This brings in a clear separation of code.
- Every object should have a well-defined type. Usage of the 'any' type for an object is allowed only in cases where the object is coming from a non-ES6 library.
- Each class/interface/factory/adapter/mediator/etc. should be in a separate file
- Avoid side-effects
- Variables should be initialized nearest to first usage
- Use const whenever possible
- Don’t use a public accessor within classes
- All methods/properties/objects that are not meant to be exposed should have a private accessor
- All private methods/properties/objects names should start with _ (i.e. private readonly _myProperty: string;)
- Don’t use camelCase/PascalCase for file names. File names should be lowercased with a '-' (split character).
- Don't have any unused variables – this one catches tons of bugs!
- Add a space after keywords, for example if (condition) { ... }
- Always use === instead of ==. But obj == null is allowed to check null || undefined.
- Infix operators must be spaced.
- Keep else statements on the same line as their curly braces.
- For multi-line if statements, use curly braces.
- Multiple blank lines are not allowed.
- For the ternary operator in a multi-line setting, place ? and : on their own lines.
- Single quotes (') are preferred unless escaping.
  const location = env.development
  ? 'localhost'
  : 'www.api.com'
- Add spaces inside single line blocks.
- Dot should be on the same line as property.
- See the full list of guidelines here.

Class items order

The following is the preferred items order within a class, struct, or abstract class:

- Constant Fields
- Fields
- Constructors
- Events
- Enums
- Interfaces (interface implementations)
- Properties
- Indexers
- Methods
- Structs
- Classes

Within each of these groups, order by the following access:

- public
- internal
- protected internal
- protected
- private

Within each of the access groups, order by static, then non-static.

Within each of the static/non-static groups of fields, order by readonly, then non-readonly.

Array

Annotate arrays as 'foos: Foo[]' instead of 'foos: Array<Foo>'.

Comments

Use JSDoc style comments for functions, interfaces, enums, and classes. Whitespace is mandatory after the opening construction (i.e. Good '/** my comment', Bad: '/**mycomment').

CSS

- Use BEM
  - http://getbem.com/
  - https://zellwk.com/blog/css-architecture-1/
  - https://spaceninja.com/2018/09/17/what-is-modular-css/
  - every tag should have its own class: avoid overcomplicating markup
  - **THERE ARE NO GRANDCHILDREN IN BEM**
- Usage of ~styles.scss is **not allowed** in any component
- Every CSS class should be separated by an empty line
- Never rely on tags in CSS: add a CSS class to a tag if you need to change its style

- When measurement includes 0, it should not contain a measurement unit (i.e. Good: 'padding: 0;', Bad: 'padding: 0px;')
- Every component should have its own SCSS file
- Use **ViewEncapsulation** as none so it will be easier to change this component's styles from the parent component
    - Be advised that in this case you shouldn't use class names (i.e. header, button, table, etc.) that are too common as this will create issues in the future
- Import only needed styles in the component's SCSS file. Never import the styles.css file.
- Use SASS features:
    - Variables
    - Reference symbol '&'
    - Partials and '@import' directive
    - Interpolation
    - Lists and '@each' directive
    - Control directives

HTML

- No usage of **FlexLayout** is allowed in templates
- No usage of **Bootstrap** is allowed
- No usage of **!important** is allowed
- All tags should be aligned. There shouldn't be any additional whitespaces (more than one), and there shouldn't be any additional empty lines.
- Event methods for controls (i.e. '(click)', '(focusout)', etc.) should contain only one method invocation. Constructions like this '(click)="next > 1 && next() && dropDatabase()"' are not allowed.

Images

- Images should be located in ./assets/*. When referring to an image from your code it is mandatory to use the absolute URI path
- Image names should be lowercased with the '_' character
- Image names should be self-explanatory

Commit message guide-

Commit messages are very important during contribution and play a crucial role if any feature or bug needs to be traced/tracked.

Basic rules

- Commit related features
    - For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other developers to understand the changes and roll them back if something goes wrong.
- Don't commit half-done work
- Test your code before you commit
- Write good commit messages; for further guidance and samples, see the Formatting rules and Pull request and commit summary format sections below.
- Use branches
- Agree on a workflow
    - Git lets you pick from a lot of different workflows: long-running branches, topic branches, merge or rebase, and git-flow to name a few. Which one you choose depends on specific factors unique to your project and team. However you choose to work, make sure that you agree on a common workflow that everyone follows.
    - An example of a commonly used workflow in Chubb Studio is the Git Feature Branch Workflow. The core idea with this workflow is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the codebase. This also means that the main branch will never contain broken code, which is highly beneficial for continuous integration environments.

Formatting rules

- Lowercase, short (50 chars or less) summary
- Write your commit message in the imperative: "fix bug" and not "fixed bug" or "fixes bug"
- Always leave the second line blank

Commit types

- feat: a new feature or a feature enhancement
- fix: a bug fix
- docs: changes to documentation
- style: formatting, such as missing semicolons, etc.; no code change
- refactor: refactoring production code
- test: adding test(s), refactoring test; no production code change
- build: changes that affect the build system or external dependencies (example scopes: gulp, broccoli, npm)
- ci: changes to continuous integration configuration files and scripts
- chore: updating build tasks, such as packaging manager configs, etc.; no production code change
- perf: a code change that improves performance; performance refactoring

Pull request and commit summary format

<commit type>(<scope>): <commit summary>

i.e. feat(DCRUX-555): add dialog component, fix(DCRUX-555): fix close button in dialog component

When a change is being added/modified for a specific country, please indicate the country code in square brackets ([]) before the <commit summary>. For instance, if we are adding policy issuance as a feature in France, the commit message should look like the following:

feat(SCF-3429): [***fr***] add policy issuance through S6

fr above indicates the country code for France.

For more examples

Branch name format

/ i.e. DCRUX-555/feature, DCRUX-555/fix, etc.

Commit emoji key

Type	Emoji	code
feat - enhancement	✨	`:sparkles:`
feat - new feature	🚀	`:rocket:`
fix	🐛	`:bug:`
docs	📚	`:books:`
style	🎨	`:art:`
refactor	🔨	`:hammer:`
test	🚨	`:rotating_light:`
build	📦	`:package:`
ci	👷	`:construction_worker:`
chore	🔧	`:wrench:`
perf	⚡	`:zap:`

Deployment guidelines (how to deploy code)

the following environments available:

Environment	Branch Name	Applicable to
SIT	sit	All repositories
SIT Sandbox	sandbox-sit	eCommerce microsite
UAT	uat	All repositories
UAT Sandbox	sandbox-uat	eCommerce microsite
Prod	master	All repositories
Prod Sandbox	sandbox-master	eCommerce microsite

All deployment jobs are part of the studio-core job in Jenkins.

If you do not have access to the build jobs in Jenkins, please reach out to Anand Subramanian/Shabeer Mothi.

Deployments to SIT should be triggered by a release engineer. Read along to understand how to trigger deployments to the appropriate environments.

Deployments

It's a good idea to check out the dev branch and make changes to flo configurations all at once and push it as one commit. Performing a commit for every flo config change is not recommended.

Deploy to SIT

- Update/increment the build_tool > version in flo_sit.yml. Most often you will increment it to the next number in the sequence; it's always a good idea to double check it with the build number of the job in Jenkins. It's important for the version number to be the build number of the job that will be triggered as part of the deployment (the same build number will be used while deploying to UAT).
- Update/increment the build_number in flo_uat.yml with the same version as provided in the build_tool > version in flo_sit.yml.
- Commit flo_sit.yml & flo_uat.yml.
- Merge dev to sit.
- Validate that the job for the appropriate project is triggered in Jenkins.

Deploy to UAT

Deployments to UAT do not happen on a day-to-day basis. Deployments to UAT will be done only at the end of the Sprint. The release engineer is responsible for deploying to UAT as well.

Once all the deployments to SIT are done and validated, in order to trigger a release to UAT, do the following:

- Merge SIT to v1.0.x branch.
- Merge SIT to UAT (you need not change the version in flo_uat.yml as we would have already changed it while releasing to SIT).
- Get approval for the release to UAT from Anand Subramanian/Shabeer Mothi.
- Once approved, the UAT release job for the appropriate project should kick off in Jenkins.

Deploy to SIT Sandbox

Deployments to the SIT Sandbox are only applicable for an eCommerce microsite. The process for triggering a deployment to the Sandbox is the same as deploying to SIT/UAT. However, the branch to merge the changes is different and is prefixed with 'sandbox-'.

- Update/increment the build_tool > version in flo_sandbox-sit.yml. Most often you will increment it to the next number in the sequence; it's always a good idea to double check it with the build number of the job in Jenkins. It's important for the version number to be the build number of the job which will be triggered as part of the deployment (the same build number will be used while deploying to UAT Sandbox).
- Update/increment the build_number in flo_sandbox-uat.yml with the same version as provided in the build_tool > version in flo_sandbox-sit.yml.
- Commit flo_sandbox-sit.yml & flo_sandbox-uat.yml.
- Merge dev to sandbox-sit.
- Validate that the job for the appropriate project is triggered in Jenkins.

Deploy to UAT Sandbox

Deployments to the UAT Sandbox are only applicable to an eCommerce microsite. The process to trigger a deployment to the Sandbox is the same as deploying to SIT/UAT. However, the branch to merge the changes is different and is prefixed with 'sandbox-'.

Deployments to the UAT Sandbox do not happen on a day-to-day basis. Deployments to the UAT Sandbox will be done only at the end of the Sprint. The release engineer is responsible for deploying to the UAT Sandbox as well.

Once all the deployments to the SIT Sandbox are done and validated, in order to trigger a release to the UAT Sandbox, do the following:

- Merge sandbox-sit to sandbox-uat (you do not need to change the version in flo_sandbox-uat.yml as we would have already changed it while releasing to SIT Sandbox).
- Get approval for the release to UAT from Anand Subramanian/Shabeer Mothi.

- Once approved, the UAT Sandbox release job for the appropriate project should kick off in Jenkins.