# Purpose

The purpose of using the feature flag system is to always have a mergeable dev branch. When developing new features, each step of development needs to be reviewed and merged as often as possible to dev in order to avoid conflicts. However, unfinished features should not appear in production - this is why the feature flag system will allow the QA to only enable the unfinished feature in the staging environment, leaving production unchanged.

# Overview of the process

Whenever a new feature enters development, a macro is created on the board. The macro does **not** have it’s own PR, each subtask of the macro is instead **directly merged to dev**, even if unfinished. The developer must create a new feature flag in the staging Unleash UI with correct description, as well as include it in the code, making sure that:

* if the feature flag is **disabled**, the entire app **MUST** function as before, with no changes visible
* if the feature flag is **enabled**, then the new feature should be available instead

## Feature flag naming conventions

* For a flag to be accessible from the front-end, it needs to contain .web. in its identifier
* A flag that is supposed to be **long-term** and that can be enabled and disabled at any moment should have the following format:  
    
   ops.web.<name>
* A flag that is only required for a specific feature during its development should have the following format:  
    
   feat.web.<name>
* <name> should be a unique identifier for the flag, in case of feature flags it should describe the feature shortly, not the task; e.g. “feat.web.dialog-redesign” or “feat.web.new-team-effort-algorithm”

## For developers

When starting work on a feature whose development will take several tasks:

* create the subtasks on the board - each subtask should have a PR on Bitbucket whose target is the ‘dev’ branch
* create a new issue assigned to yourself on the board linked to the macro (blocked by the macro) with title “Cleanup after WMS-1234” - this task would remove the old implementation from the code and remove the feature flag
* create a feature flag on your local Unleash instance (accessible at <http://localhost:4242>) - for naming conventions look above, the description is optional and the default strategy is enough for local development
* create a feature flag on the staging Unleash instance (accessible at [https://staging.wms.gcp.lastingdynamics.net/\_\_unleash](https://staging.wms.gcp.lastingdynamics.net/__unleash/#/features)) - with the same name as previously but with description that helps the QA recognize it, e.g. by providing the task number (of the macro): “[WMS-1234] New dialog redesign”
* develop the feature in each of the subtasks and move to “Ready for Review” whenever a particular subtask is complete
* a subtask will also require QA approval, as well as code review - in such situation, mark the QA as reviewers and place a comment describing what exactly needs checking. Only in cases where something cannot be checked QA, create a comment that it’s not required
* after all subtasks are merged, the macro should be moved to “Ready for Review” where it will be checked by QA

## For quality assurance

* list of all feature flags is located at: [https://staging.wms.gcp.lastingdynamics.net/\_\_unleash](https://staging.wms.gcp.lastingdynamics.net/__unleash/#/features)
* whenever testing a subtask that is part of a feature macro, make sure that the new (unfinished) functionality is **only** available when the switch of the feature is **turned on** in Unleash
* when a feature is turned on, then it will be available **both on a single branch and on staging** - that’s why when performing regression tests for the next deploy, all the feature switches should be **turned off**
* when all subtasks of a macro are completed, the whole feature should be **tested on dev,** with the switch for that feature **turned on**
* if the feature is ready, create (or request the developer to do it) a new feature flag **with the same name** and with proper, informative description on the **production Unleash**: <https://app.roundrush.com/__unleash>
* the macro can be moved to “Merged”, and after the next deploy, the switch should be turned on, so that it appears on production
* in case the feature should be reverted from production, the switch can be turned off, it should return to the state before the macro was completed

# Development with flags

In order to make some parts of the code available only with a particular flag, there are two options available:

* useFeatureEnabled hook - it will return a boolean indicating whether a switch is turned on or off, e.g.  
    
   const enabled = useFeatureEnabled(‘feat.web.new-dialog’)
* <FeatureFlag /> component - it will only render the matching “arm”, e.g. it will render the “on” prop if the feature is enabled and “off” prop is the feature is disabled:  
    
   <FeatureFlag  
   name=”feat.web.new-dialog”  
   on={() => <NewDialog />}  
   off={() => <OldDialog />}  
   />  
    
  Both props accept a function that returns a React element, it will be called and the result of it rendered as long as the flag is either on or off.

## How to use

The component form (<FeatureFlag />) is the preferred way to change behaviour depending on the feature flag being enabled or disabled. The hook form is only allowed in certain situations where using the component is not possible, e.g. in logic that does not deal with component rendering.

The correct usage pattern of the component form is to only use it once inside a component being changed, and as high in the tree as required. Using it multiple times in the same component is an anti-pattern as it would lead to complicated conditional logic that cannot be easily understood and maintained.

**For any component that needs to be changed during development of a feature, wrap the entire body of it into a <FeatureFlag /> and place the old version in the “off” block, developing the new one in the “on” block.**

**The best way to do it - and required If you need to change hooks or any logic that cannot be placed in an inline function - is to split the component into two, e.g.:**

function OldImplementation({ … }: Props) {

const value = oldHook()

return ...

}

function NewImplementation({ … }: Props) {

const value = newHook()

return ...

}

function ChangedComponent(props: Props) {

return (

<FeatureFlag

name=”feat.web.changed-component”

off={() => <OldImplementation {...props} />}  
 on={() => <NewImplementation {...props} />}

/>

)

}

The form of the example above should be used most of the time when checking for feature flags.