Branching Plan

for

Group Testing Environment

v1.0

12/08/2022

Prepared by

Filipe Lagrenade,

Sanjay Williams,

Jia Son Pow, and

Isaac Ellis

Capstone Group I3

**Table of Contents:**

[**Branching Workflow**](#_355to1io25l9) **3**

[Options](#_rw2t9b90r5cb) 3

[Git Flow](#_ram51ng4149g) 3

[GitHub Flow](#_9kb4exn2ji33) 4

[GitLab Flow](#_toes283ikpjl) 4

[One Flow](#_nxizbkdy8vf7) 5

[Conclusion](#_4jj0s6jzvyo8) 6

[**Choosing a Branching Strategy**](#_hihlez8n86b8) **7**

[Options](#_ueg1hzcubas3) 7

[Release Branching](#_j9rurpiylqe1) 7

[Feature Branching](#_936llnvqhqen) 8

[Conclusion](#_5ounzqwup7k9) 9

[**Feature Branches**](#_4mktfpjar00b) **10**

[Overview](#_invlblq5rv8a) 10

[Creating Branches](#_z0j7lkjwhwr4) 10

[Merging Branches](#_nc31kp27upgf) 11

[Example Workflow](#_mrhjn05c7z3) 11

[**Merging**](#_l5wvdoi8cwvq) **12**

[Overview](#_q6wdwdtp8kdu) 12

[Recursive Merge](#_dtaskzqnc90e) 12

[Octopus Merge](#_wt52awmiqwc2) 13

[Fast-Forward Merge](#_y1rh7lu0ymgj) 13

[**Testing**](#_ntmyypltaxh) **14**

[5.1 Overview](#_9apl986jx0ne) 14

[5.2 Types of Tests](#_1t6rognzyhd4) 14

[5.2.1 Unit Testing](#_1xna30zijv33) 14

[5.2.2 Integration Testing](#_7xsysar1z94z) 14

[5.2.3 End-to-end Testing](#_502av0ok117y) 15

[5.3 API/Repository Integration Testing](#_m0d4e0iiu86g) 15

[5.4 Selenium End-to-end Testing](#_vso9tr7uprg9) 15

[5.5 PMD Testing (Java)](#_enatjnffdvxy) 16

[5.6 ESLint (JavaScript)](#_kclha6pybe0a) 16

# 

# Branching Workflow

Feature developments should be done in the feature branch instead of the main branch. This is to ensure that the main coding branch will not be affected by having multiple developers working together. It also allows developers to open pull requests which makes it easy for others to comment on each other’s work before having it integrated into the official project, thus creating an overall final product.

## Options

For this section, we will be looking at four different options that are widely available. These workflow options provide solutions that would be more suitable than one another for various scenarios that may occur during the development of a project.

### Git Flow

Introduced by Vincent Driessen in 2010.  
Essentially, it involves five different branch types that isolate work from the main branch. Under this model, developers create a feature branch and delay merging it to the main branch until the feature development is complete.

* + - 1. **Advantages**
* At any moment during the life cycle of a project, using this workflow method ensures that the state of the branches will stay clean.
* Branch naming follows a systematic pattern that allows for easy comprehension.
* This is ideal for when there is a need for multiple versions created for production.
  + - 1. **Disadvantages**
* Many branches with complicated rules, which can make git history unreadable.
* For released versions, there will be a heavy workload required for maintenance.
* Git Flow’s rigid structure and development path conflicts with the iterative approach of Agile methodology.

### GitHub Flow

A lightweight workflow option that involves no release branches. It was developed in 2011 by GitHub to provide a simpler alternative to Git Flow.

* + - 1. **Advantages**
* Encourages high speed feedback loops, making it possible for issues to be quickly identified and solved.
* Continuous Deployment is a must with GitHub Flow. This means that there will not be finished features waiting to be released, thus no time/resources will be wasted.
* Lower risk of technical debt, the emphasis on shallow structures and small changes make it easier in the long run to identify suboptimal pieces of code and thus largely reduce the risk of refactoring.   
    
    
  + - 1. **Disadvantages**
* Not as well-organized as Git Flow. The high speed of this workflow comes at the cost of ease of management for the development of the whole project.
* Master branch can become cluttered even more easily as it functions now as both the production and development branch. High level of attention is essential to maintain the branch.

### GitLab Flow

Developed by GitLab in 2014, this is a simpler alternative to GitFlow that combines feature-driven development and feature branching with issue tracking.   
  
It follows 11 rules:  
1. Use feature branches, no direct commits on master.  
2. Test all commits, not only ones on master.  
3. Run all the tests on all commits.  
4. Perform code reviews before merges into master not afterwards.  
5. Deployments are automatic, based on branches or tags.  
6. Tags are set by the user, not by CI.  
7. Releases are based on tags.  
8. Pushed commits are never rebased.  
9. Everyone starts from master and targets master.  
10. Fix bugs in master first and release branches second.   
11. Commit messages reflect intent.

* + - 1. **Advantages**
* Much cleaner git history compared to Git Flow.
* Ideal for when there is only a single version needed for production.
  + - 1. **Disadvantages**
* Higher in complexity compared to GitHub Flow. Can become even worse than Git Flow if multiple versions need to be maintained for production.

### One Flow

Developed by Adam Ruka in 2015 as an alternative to GitFlow. It focuses on having only one branch (hence the name) in order to reduce clutter/complexity.

### 

* + - 1. **Advantages**The biggest advantage that the developer of One Flow advocated for against the other workflow options is that One Flow provides readable git history (something Git Flow is severely flawed at) while following very similar workflow approaches to that of Git Flow.
      2. **Disadvantages**While OneFlow is pretty flexible, it’s not suitable for every project. OneFlow will be a bad fit in basically the same circumstances that GitFlow would be. There are 2 main reasons why that can be the case.

First, OneFlow is not recommended for projects with Continuous Development and Continuous Delivery. For example, take the Python programming language. It has two incompatible versions, 2 and 3. Both of them receive bug fixes and security patches – however, that doesn’t mean a new release of Python 3 is based on the commit of the latest release of Python 2. If the project needs to have multiple simultaneous yet incompatible release versions maintained, OneFlow will not be a good option. The main challenge is usually in the interactions between the versions and how to effectively share code between them, and OneFlow was not designed to be a solution to that problem.

Second, if the project has a high degree of automation then this workflow will most likely be too heavy. Parts of it might still be useful, but other elements (like the release process, for instance) would have to be heavily modified to make sense when releasing on such a very frequent schedule.

## Conclusion

The branching workflow chosen for the project is GitLab Flow.   
While Git Flow has many advantages that are useful, we feel that none of them applied to our project specifically. The project will not require multiple versions to be supported so using a workflow option designed for it like Git Flow should be unnecessary.  
GitLab Flow applies most well for our project as we will be working on having features added individually to the master branch,   
The short sprints and relatively simple features makes it a sensible choice to pick GitLab Flow as we will likely be branching and merging often to ensure a functional product at the end of each sprint.

## 

# Choosing a Branching Strategy

Choosing the right branching strategy for this project will be essential for the project's progress over the next few sprints. Having a solid and consistent branching strategy will help ensure efficient version control and mitigate any issues that may arise without a good strategy.

## Options

The main two options considered were the Release and Feature branching approaches. Both had their advantages and disadvantages which helped when choosing the right one for this project.

Note that while multiple branching strategies can be used simultaneously and can generally differ for different teams and sub-teams, an overall strategy was considered here due to the nature of the project and the short development cycle.

### Release Branching

The Release Branching approach is structured in such a way that the release is contained within its own branch.

* + - 1. **Advantages**
* Works well for projects that support multiple versions
* Allows for easy customisation for specific users
* Allows devs to focus on specific issues between releases
  + - 1. **Disadvantages**
* Can be hard to keep track when working with multiple releases
* Can inadvertently create more work for devs
* Extra processes are required for a clean progress
  + - 1. **Summary**

As the project, during development, will not require the support of multiple versions or patches, the first advantage listed is less crucial in this case. As there are specific builds for the product planned, the customisation for specific users is also not as necessary. Being able to focus on specific issues could be helpful; however, as there will not be any fully functional versions until the end of the development cycle, focussing on issues between versions is not so helpful.

As each item in the list of advantages either doesn’t affect our project or has little advantage in our case, this, along with the disadvantages, makes the Release Branching approach a less desirable option for this project.

### Feature Branching

The Feature Branching approach makes each new feature to be worked on its own branch. These are often coupled with feature flags to enable or disable the features within the app.

* + - 1. **Advantages**
* Allows devs to independently work on specific features of the app
* Helps devs more easily segment their work
  + - 1. **Disadvantages**
* Feature branches are open for the entire development process of the feature, meaning some sections can be worked on independently for extended periods of time.
* Only works if devs branch and merge often
* Extra processes are required for a clean progress
  + - 1. **Summary**

Due to the short development cycle of the project, the ability to work on specific features of the application would be a huge advantage in our case. The easy segmenting of workloads will prove useful in the next few sprints; especially due to the small team size.

Some of the disadvantages will likely not apply to our project.

For instance, the problem of having feature branches open for long periods of time will likely not be a problem in our case due to the relatively simplistic nature of each feature to be included in the product. This will also help to be alleviated by ensuring that we branch and merge often. This will likely be the case anyway due to the relatively short sprint lengths.

## Conclusion

The branching strategy chosen for the project is Feature Branching.

While Release Branching has many advantages that would be useful for certain projects, none of them directly applied to our specific project. Release branching can also be used alongside other strategies such as Feature branching; however, as the project won’t require multiple versions to be supported and the team is small, having multiple strategies is not necessary.

Feature branching applies most well to our project as the product we are working on fits well with having features worked on individually to the master branch.

The short sprints and relatively simple features help this strategy fit even better as we will likely be branching and merging often to ensure a functional product at the end of each sprint.

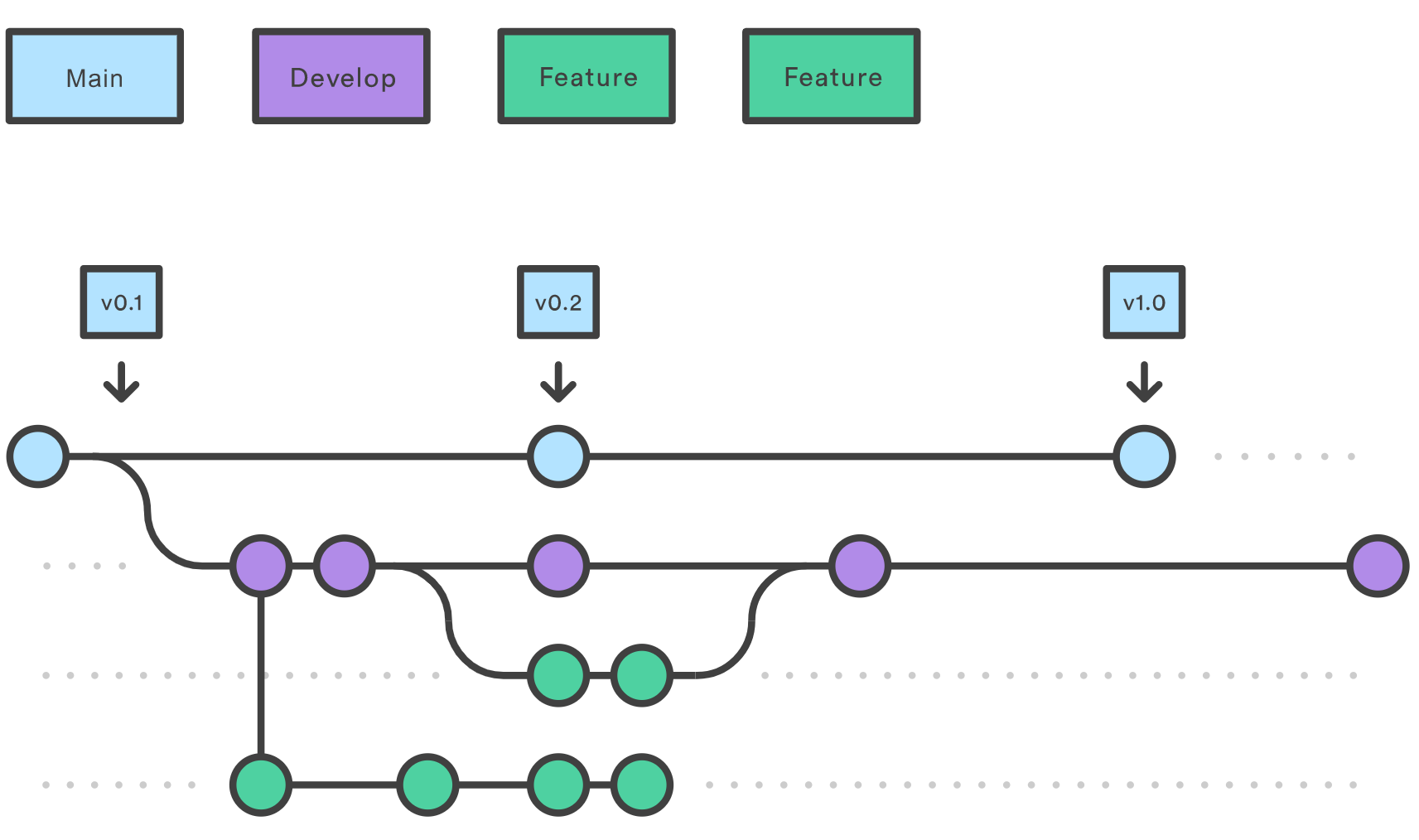
# Feature Branches

## Overview

Each new feature exists in its own branch, which, when complete, can be pushed to the central repository. In the case of feature branches, they branch off the ‘develop’ branch rather than the ‘main’ branch. The same applies to merging; the feature branch merges back into the ‘develop’ branch rather than the main branch.

An example of feature branch usage is shown here:

(acquired from: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>)



## Creating Branches

Using git-flow extensions, feature branches can be created using:

*git flow feature start feature\_branch*

Or otherwise, using:

*git checkout develop*

*git checkout -b feature\_branch*

## Merging Branches

Using git-flow extensions, feature branches can be merged back into the develop branch using:

*git flow feature finish feature\_branch*

Or otherwise, using:

*git checkout develop*

*git merge feature\_branch*

## Example Workflow

1. **Person A begins a new feature.**

* Person A requests a new branch off the ‘develop’ branch using:

*git checkout -b person-A-feature main*

1. **Person A has a break.**

* Person A, before going on a break, pushes the feature branch back to the central repository using:

*git push -u origin person-A-feature*

1. **Person A finishes their feature.**

* Once complete, Person A can merge into ‘develop’ after filing a pull request to let the team know they’re done, while first making sure the central repository has their latest commits using:

*git push*

1. **Person B receives the pull request.**

* After conversing with Person A, Person B suggests that there should be a change made to *person-A-feature*

1. **Person A makes the changes using the same process as before and publishes their feature to the ‘develop’ branch.**

* Either Person A or B can merge the feature into ‘develop’ using:  
  *git checkout develop*

*git pull*

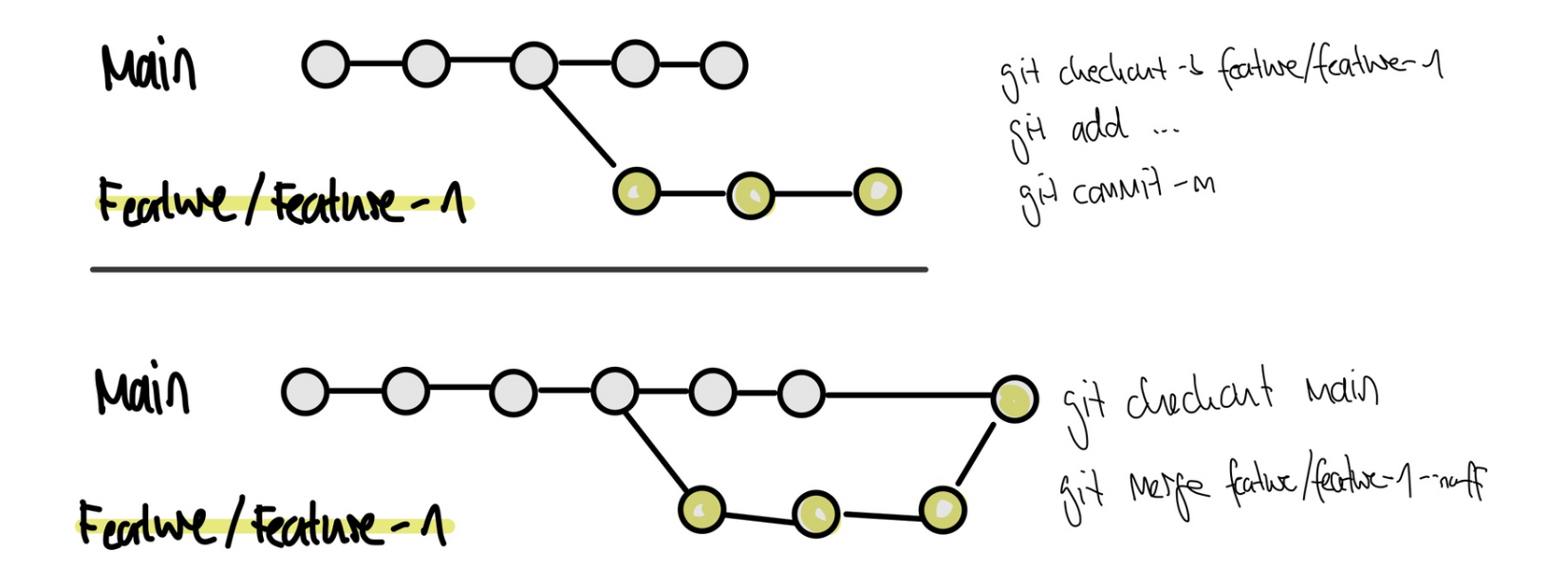
*git pull origin person-A-feature  
git push*

# Merging

## Overview

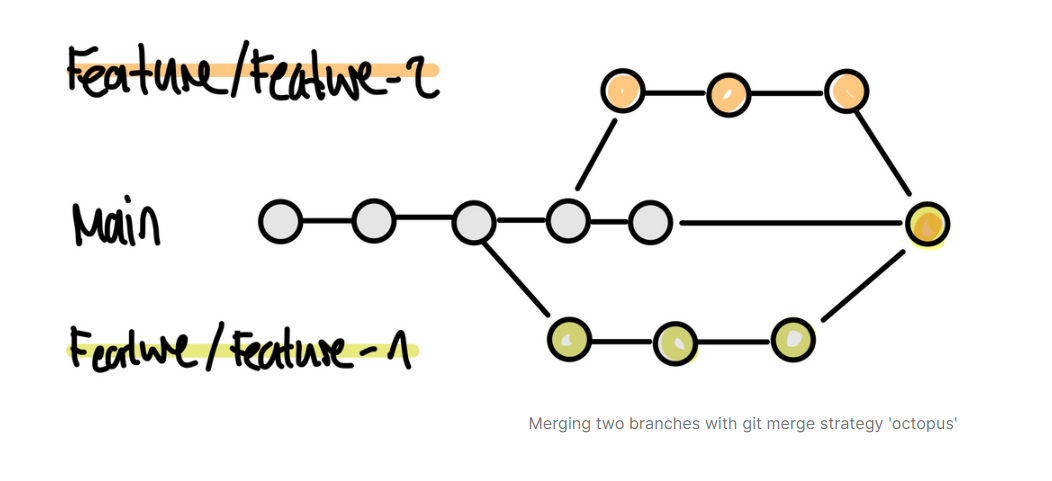
Different merge strategies get used for different applications. The main two types of merging in this case is octopus merge and recursive merge. Generally for this project, recursive merge will get used the most often. Fast-forward merge may also be used on the odd occasion where there has been no changes to the master branch.

### Recursive Merge

Recursive Merge is the default git merge strategy. It will be used to merge a single feature branch into the main branch. As we are using a feature branching strategy, recursive merge will be used the most often as every time a feature is completed it will get merged into master. It is called by using *git merge --no-ff.*

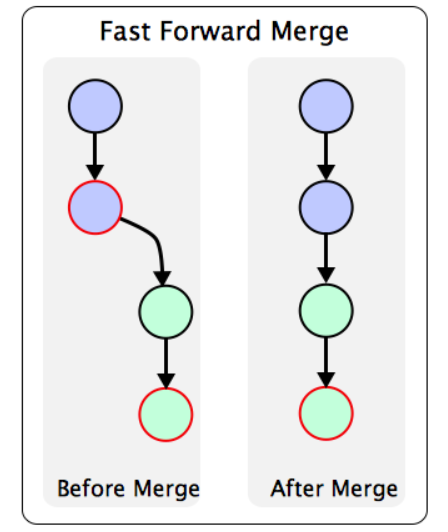
*A visual example of recursive merge (Acquired from:* [*https://www.workingsoftware.dev/which-git-merge-strategy-is-appropriate-for-our-team/*](https://www.workingsoftware.dev/which-git-merge-strategy-is-appropriate-for-our-team/) *)*

### Octopus Merge

Octopus merge may be used when two or more features need to be merged into the main branch at the same time. It will bundle the branches together. This is called by using *git merge -s octopus branch1 branch2.*

*A visual example of octopus merge (Acquired from:* [*https://www.workingsoftware.dev/which-git-merge-strategy-is-appropriate-for-our-team/*](https://www.workingsoftware.dev/which-git-merge-strategy-is-appropriate-for-our-team/) *)*

### Fast-Forward Merge

Fast Forward merge will be used in very select occasions where there have been no changes made to the main branch since the feature branch was made. This is called by using *git merge* *branch1.* 

*A visual example of fast forward merge (Acquired from:* [*https://www.bogotobogo.com/DevOps/SCM/Git/Git\_GitHub\_Fast-Forward\_Merge.php*](https://www.bogotobogo.com/DevOps/SCM/Git/Git_GitHub_Fast-Forward_Merge.php)

# Testing

### 5.1 Overview

It's important to make the distinction between manual and automated tests. Manual testing is done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling. This is very expensive and time consuming as it requires someone to set up an environment and execute the tests themselves, and it can be prone to human error as the tester might make typos or omit steps in the test script.

Automated tests, on the other hand, are performed by a machine that executes a test script written in advance. These tests can differ in level of complexity, from checking a single method in a class to making sure that performing a sequence of complex actions in the UI leads to the same results. It's much more reliable than manual tests but the quality of your automated tests depends on how well your test scripts have been written. Automated testing is a key component of [**continuous integration**](https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration) and [**continuous delivery**](https://www.atlassian.com/continuous-delivery/principles/pipeline) as new features are added to the application, though there is value in doing manual testing.

## 5.2 Types of Tests

### 

### 5.2.1 Unit Testing

Unit tests are very low level and close to the source of an application. They consist of testing individual methods and functions from the classes, components, or modules used by your software. Unit tests are generally quite cheap to automate and can run very quickly by a continuous integration server.

### 5.2.2 Integration Testing

Integration tests verify that different modules or services within an application work well together. For example, it can be a test of the interaction with the database or making sure that microservices work together as expected. These types of tests are more expensive to run as multiple parts of the application are required to be up and running.

### 5.2.3 End-to-end Testing

End-to-end testing copies a user’s behaviour with the software in a complete application environment. It verifies that various user flows work as expected and can be as simple as loading a web page or logging in or much more complex scenarios like verifying email notifications and online payments. End-to-end tests are very useful, but they are expensive and time-consuming to perform. On top of that, they are of very high maintenance when automated. It is recommended to have a few key end-to-end tests and rely more on lower level types of testing (unit and integration tests) to be able to quickly identify breaking changes.

## 

## 5.3 API/Repository Integration Testing

API Integration testing is the testing of a set of application programming interfaces that connects the APIs directly and as part of an integration test to determine if they meet expectations for functionality, reliability, performance, and security.

## 5.4 Selenium End-to-end Testing

For end-to-end testing, we have decided on Selenium as it is free of charge and boasts a large user community. [Selenium](https://www.leapwork.com/blog/whitepaper-selenium-automation) consists of Selenium Webdriver, their primary web-based automation tool, Selenium IDE, a record-and-playback tool, and Selenium Grid, a parallel testing tool. Combined, these tools allow you to automate anything that goes on in the browser, using a code-based approach. Selenium provides Docker images out of the box to test with one or several browsers. The images spawn a Selenium server and a browser underneath. It can work with different browsers. This will help our project as we will be trying to integrate 2 different clients (Student and UC) for functionality.

## 5.5 PMD Testing (Java)

PMD is an extensible cross-language static code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It includes CPD, the copy-paste-detector. It is especially useful for detecting possible bugs and bad programming practices. We are using it as it comes for free in the Maven build tool package and commonly used for JavaScript applications.

## 5.6 ESLint (JavaScript)

Code standards allow developers in a team to create code that consistently follows best practices.To ensure that these practices are enforced, our team will need to install one of many static code analysis tools. These tools come as plugins for IDEs and once installed can be configured to detect and enforce selected rules. For our project, we have picked ESLint, one extremely popular static code analysis tool for JavaScript.

## 

## 