

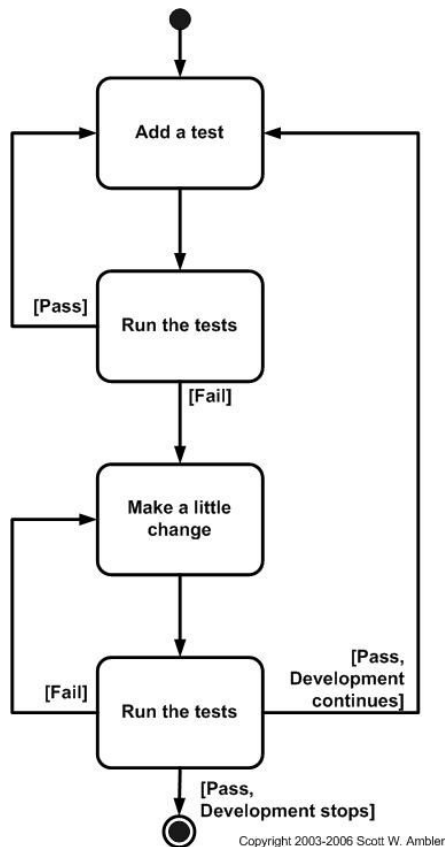
# SQE Strategy

## Feature-driven

We have already defined several user stories. We will map each user story into one (or more) system feature(s) and then map each feature into a git branch.

## Tests

The overall strategy is very similar to TDD or Test Driven Development, so the overall workflow is as follows



And therefore, for each chunk of development, we create tests for it and only keep on developing when the tests pass, otherwise we will continue to adjust code until it passes the tests.

One of the biggest advantages of using TDD is that we are making small increments when writing code, and therefore issues are easier to fix, since they only address a small chunk of code.

### Three Laws of TDD

- You may not write production code until you have written a failing unit test.

- You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- You may not write more production code than is sufficient to pass the currently failing test.

The tests on **Spring Boot** will be done using **JUnit**, **Hamcrest** and **Mockups**.  
For the frontend the tests will be made with **Selenium**

To merge a pull request all tests must pass.

## Git Workflow

Never merge directly, always make pull requests and identify at least one person to check (review) that pull request before merging the PR. (All repositories are configured to not accept a single person merge)

**New feature branch:** For each new feature create a branch following the standard: feature/<feature\_name>.

**New Issue branch:** For each fix create a branch following the standard: fix/<fix-name>.

**Closing issues/tasks** can be done by writing in commit message: "this closes #<issue\_nr>"

## Development workflow

We decided to adopt a standard Git Flow, which resides on the following:

1. Map each user story into one (or more) system feature(s).
2. Map each feature into a git branch.
3. Classify the feature as done when the expected system behavior is accomplished.
4. Merge the feature branch into the master/production branch.

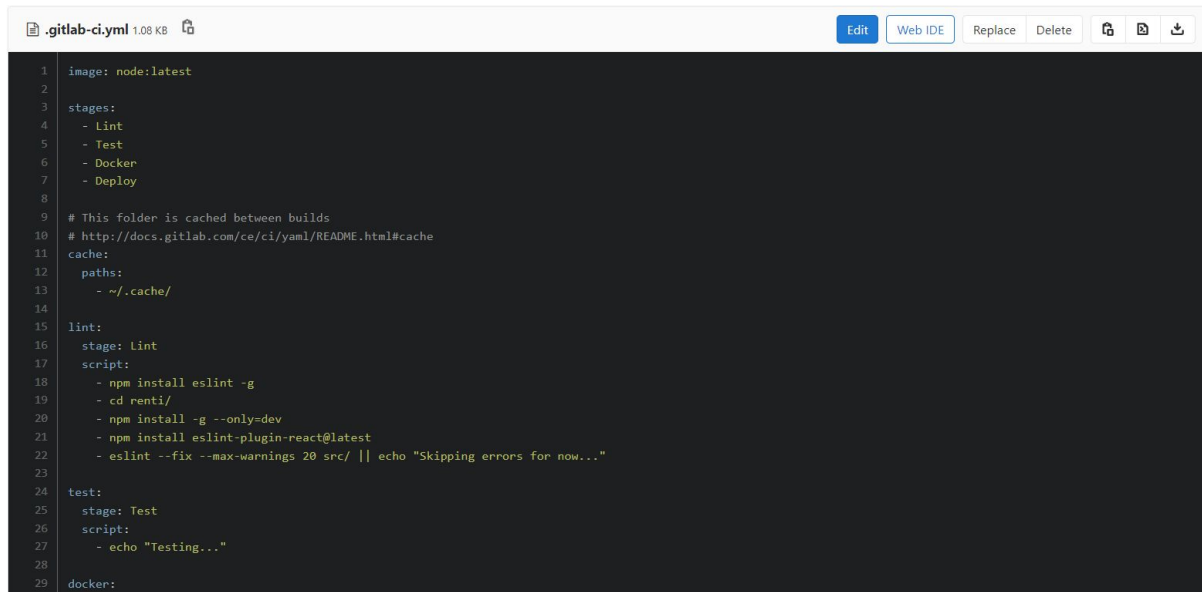
## Code analysis

We will probably use **SonarQube**

Gitlab also has a feature that integrates Static Code analysis into the Gitlab CI pipeline

# CI/CD

For continuous integration we'll use the gitlab ci pipeline. For each commit pushed into the repository the pipeline will be triggered

A screenshot of a web editor displaying a GitLab CI pipeline configuration file named .gitlab-ci.yml. The file is 1.08 KB in size. The editor has a dark theme and includes buttons for 'Edit', 'Web IDE', 'Replace', 'Delete', and download icons. The configuration defines a pipeline with stages: lint, test, and docker. The lint stage uses the node:latest image and includes a script to install and run ESLint. The test stage also uses node:latest and includes a script to echo 'Testing...'. The docker stage is currently empty.

```
1 image: node:latest
2
3 stages:
4   - lint
5   - test
6   - docker
7   - Deploy
8
9 # This folder is cached between builds
10 # http://docs.gitlab.com/ce/ci/yaml/README.html#cache
11 cache:
12   paths:
13     - ~/.cache/
14
15 lint:
16   stage: lint
17   script:
18     - npm install eslint -g
19     - cd renti/
20     - npm install -g --only=dev
21     - npm install eslint-plugin-react@latest
22     - eslint --fix --max-warnings 20 src/ || echo "Skipping errors for now..."
23
24 test:
25   stage: Test
26   script:
27     - echo "Testing..."
28
29 docker:
```

## Docker containers

The software elements are packaged into docker containers, used both to run the services locally, and in deployment. In the Gitlab pipeline, this docker images are automatically build and uploaded to a private container registry, making them accessible for our group and for the deployment environment.