LO_5 notes

For 5.1, I introduced multiple review techniques and applied one of them to the code and showed the results.

For 5.2, I described how to implement a CI pipeline in general and gave an example how a CI pipeline could be built using GitLab.

For 5.3, I explained the embedding of testing in the pipeline, and gave examples of the aspects that can be automated by using the CI pipeline.

For 5.4, I listed multiple issues the proposed CI pipeline may identify and provided examples of how these issues might be identified.

LO 5

5.1 Identify and apply review criteria to selected parts of the code and identify issues in the code

Identify appropriate review techniques:

Peer Review: This is when other developers review each other's code and provide feedback. This can be done through formal inspection or informal walkthroughs.

Automated Code Review: This is when code is automatically analysed using tools such as linting and static analysis to identify potential issues.

Code Review Checklists: This is when a checklist of specific items to review is used, such as best practices, coding conventions, and security vulnerabilities.

Test-Driven Development: This is a development approach where test cases are written before the code, and the code is written to pass the tests.

Code Reviews using Git: This is when code reviews are done using a Git tool, such as GitHub or GitLab, where changes are reviewed and discussed before being merged into the main branch.

To achieve better results in the code review process, it is important to have a culture of collaboration and open communication within the team. This can be achieved by encouraging developers to share their knowledge, providing feedback, and discussing the issues found during the review process and continuously improve the codebase.

Code review report: user.js

1. Are meaningful variable and function names used?

Yes, the variable and function names used in the code are meaningful and easy to understand.

2. Have all data errors been considered, and tests written for them?

One potential issue is that the program does not check for duplicate email addresses before inserting a new user, which could lead to multiple users having the same email. This can be resolved by adding a check for duplicate emails before creating the new user.

Another potential issue is that the program does not check if a user exists when getting a user's info, which could lead to unexpected behaviour if an error occurs.

Also, the code doesn't check for injection attacks.

3. Are all exceptions explicitly handled?

Exceptions are handled in the code and appropriate error messages are returned.

4. Are types used consistently?

Yes, the code uses types consistently, which helps to maintain consistency and readability.

5. Is the code properly formatted?

Yes, the code is properly formatted and easy to read.

Additionally, providing more comments in the code would help future developers understand the codebase better.

Code review report: order.js

1. Are meaningful variable and function names used?

Yes, the variable and function names are meaningful and easy to understand.

2. Have all data errors been considered, and tests written for them?

The code did not check is the user exists when updating and deleting an order.

3. Are all exceptions explicitly handled?

Exceptions are handled and logged, and appropriate error messages are returned to the client.

4. Are types used consistently?

Yes, the code is consistent in its use of types and follows the conventions of the language.

5. Is the code properly formatted?

The code is properly formatted and easy to read.

Additionally, providing more comments in the code would help future developers understand the codebase better.

5.2 Construct an appropriate CI pipeline for the software

CI pipeline

- Code Commit: This is the first stage. The pipeline would be triggered by a code commit to the repository.
- Build: The second stage of the pipeline would involve building the code. This would include tasks such as compiling the code, packaging it into a deployable format, and generating any necessary documentation.
- Test: The third stage of the pipeline would involve testing the code. This would include unit testing, integration testing, and system testing. This stage would ensure that the code is functional, reliable, and meets the requirements of the end-users.
- Code Linting: A linting step would be performed to ensure that the code adheres to the coding standards and conventions.
- Deployment: The final stage of the pipeline would involve deploying the code to a production environment. This would include tasks such as configuring the environment, deploying the code, and monitoring the application.

Continuous Integration: After the code is deployed to the staging environment, the pipeline would be set up to monitor the staging environment and automatically deploy new changes to the production environment as they are committed and pass through the pipeline successfully.

Additionally, to ensure that the pipeline is efficient and reliable, the pipeline would be integrated with other tools such as monitoring, logging, and alerting. This would enable the pipeline to detect and respond to failures and issues as soon as they occur, reducing downtime and increasing the reliability of the application.

In GitLab:

- Connect GitLab project to Git repository.
- Create a .gitlab-ci.yml file in the root of your project. This file will define the CI pipeline and the jobs that need to be run.
- In the .gitlab-ci.yml file, specify the stages of the pipeline and the jobs that should be run in each stage. For example, you can have a build stage, a test stage and a deploy stage.
- In each job, specify the script that should be run. For example, in the build job, you can specify the script to run npm install and npm build.
- In the test job, specify the script to run the test suite, such as npm test.
- In the deploy job, specify the script to deploy the application to a staging or production environment.
- Commit and push the changes to your Git repository.
- GitLab will automatically detect the .gitlab-ci.yml file and run the pipeline for each commit and pull request.

5.3 Automate some aspects of the testing

Incorporating testing as part of the continuous integration and continuous delivery (CI/CD) process is called embedding testing in the pipeline. This means that tests are automatically executed at various phases of the development process. The tests are integrated into the pipeline and run automatically, allowing for early bug identification, and guaranteeing that the code works as intended before it is deployed to production. The pipeline will also track test results and provide reports, which can be used to monitor code quality and highlight areas for improvement. By incorporating testing into the pipeline, teams may guarantee that the system is resilient, dependable, and secure before it is distributed to end users.

To automate testing in GitLab, we can utilize GitLab CI/CD pipeline to run our tests at different stages of the development process.

First, we can set up a pipeline that runs our unit and integration tests after every push to the repository. This can be done by creating a .gitlab-ci.yml file in the root of the project that specifies the test script and any necessary dependencies.

Then, we would want to set up a suite of unit and integration tests using Jest. These tests should be written to cover all the key functionality of the system, including user registration, authentication, and order placement. The tests should also cover edge cases and exception handling, to ensure the system is robust and can handle unexpected input.

Next, we can set up a pipeline that runs our performance tests on a schedule or after certain events, such as a new release. This can be done by using GitLab's built-in scheduling feature or by using webhooks to trigger the pipeline. We can also use Artillery.

To ensure the required level of testing for a CI environment, we can set up a pipeline that runs our tests on different environments such as different versions of Node.js and different Operating Systems.

Using GitLab's built-in code coverage feature or Istanbul, we can track the code coverage of our tests to ensure that we are achieving our desired coverage levels.

Finally, generating a report is needed.

5.4 Demonstrate the CI pipeline functions as expected

"Here you should indicate the kinds of issues your proposed CI pipeline would identify and provide some examples of how you propose these issues would be identified."

(1) Code errors

During the build stage, the pipeline would check for any syntax or semantic errors every time a code commit is made to the repository, such as missing semicolons or undeclared variables. If there is any error on the code, the pipeline would identify it.

For example, if a developer commits new code with error, the pipeline will fail at the build stage, and the developer would be notified of the issue, and fix it before it gets deployed.

(2) Test failures

The pipeline would run unit and integration tests in the testing stage. If one or more tests failed, developers will notice it easily and fix the code.

For example, if a developer makes a change in the codebase that causes a unit test to fail, the pipeline will fail at the Unit and Integration Testing stage and the developer would be notified of the failure. The developer would then need to fix the issue and re-run the pipeline.

(3) Performance issues

During the performance testing stage, the pipeline would check if the system is able to handle the expected load and if there are any bottlenecks or issues with the performance of the system.

If the performance testing stage detects that the API is not able to handle the expected number of concurrent users, the pipeline would fail, and the developer would need to optimize the code to improve performance. This could occur if the code developed has a big time-complexity and a developer performs an expensive operation.

(4) Test coverage issues

The pipeline would use a code coverage tool such as Istanbul to check that the codebase has adequate test coverage. Any areas of the codebase that are not covered by tests would be identified and flagged.

(5) Code style issues

During the code linting stage, the pipeline would check if the code adhered to the coding standards and conventions, such as variable naming and indentation.

For example, a developer may forget to use proper indentation.

(6) Deployment errors

During the deployment stage, the pipeline would check if the code can be deployed without any errors and if the system is running smoothly after the deployment.

If the pipeline fails at this stage, the reason could be the environment is not configured properly.