**Independent Project 03**

**Test Plan and Cases of Collaborative Online judge**

**Ip Chan**

**City University of Seattle**

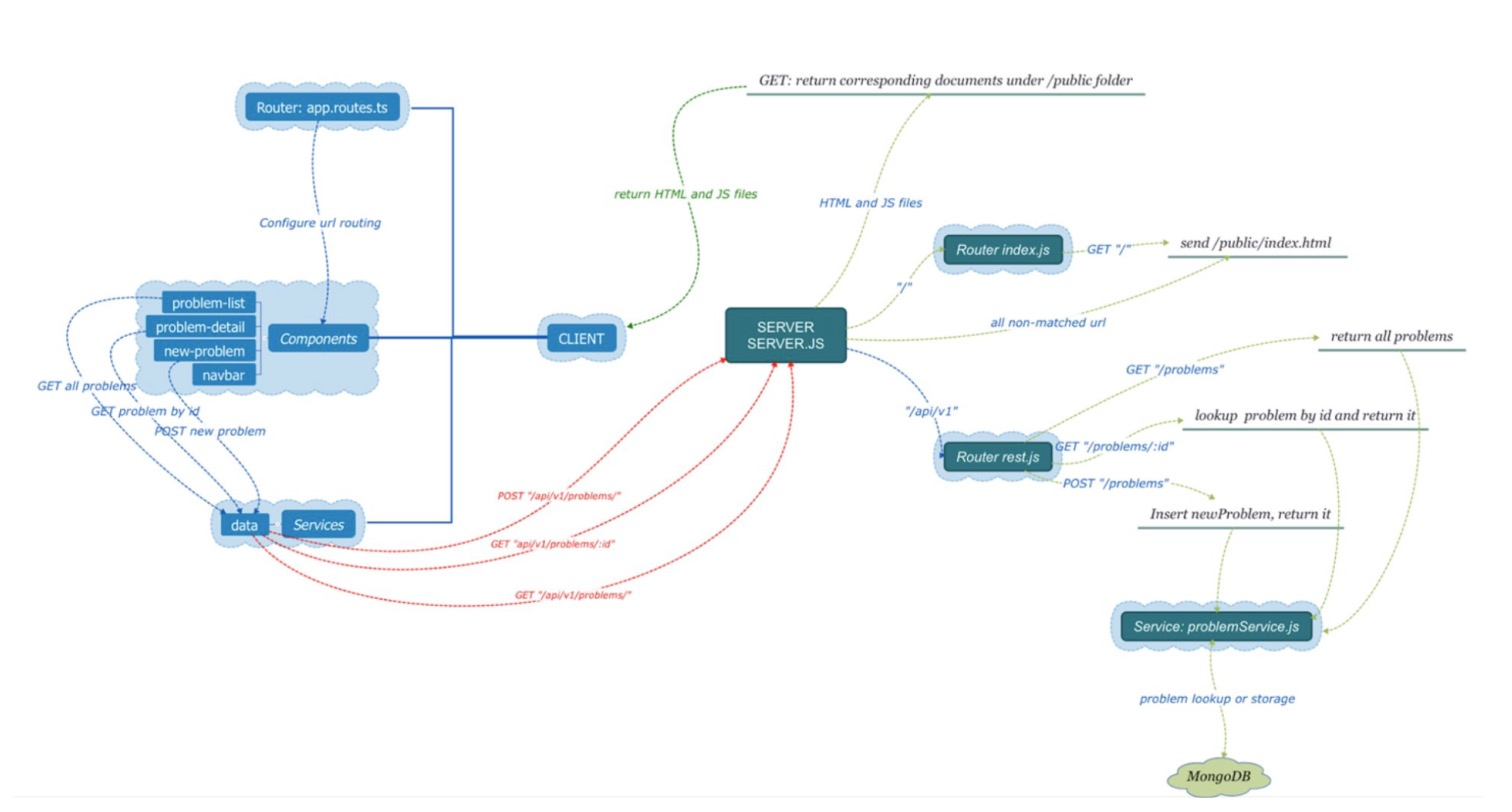
[chanip@cityuniversity.edu](mailto:chanip@cityuniversity.edu)

# Introduction

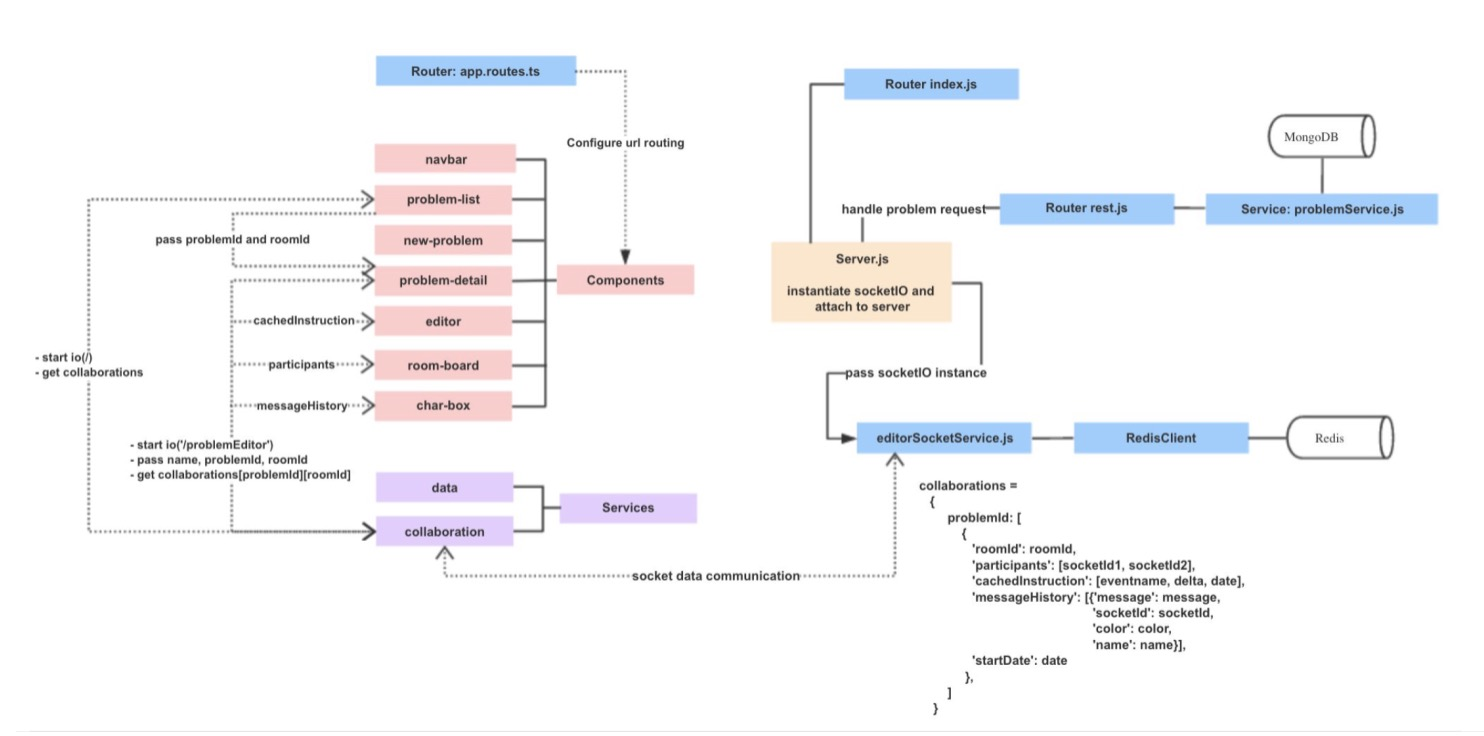
In this paper I will discuss how I would design a collaborative online judge system. For this week I have created a test plan including test cases for the system I designed and mentioned in the previous paper. I will explain the process model to be used for the development of the system. The test plan will be developed to be consistent with the process model and include an overview of the approach to testing including the project test philosophy. Besides, this test plan also include needed unit, systems, validation and verification, and acceptance testing.

**System Description**: This collaborative-online-judge-system is a full-stack project with an Angular based front end, NodeJS based restful API, and docker based code execution engine. I created multiple components in the front end to allow users to review and add problems and provide solution. The data is passed to the restful API, which in turn saves to MongoDB. Multiple users can also collaborate on problems real-time using websocket.

In this project, I implemented a web-based collaborative code editor which supports multiple user editing simultaneously (ACE, Socket.io, Redis); I also Designed and developed a single-page web application for coding problems (Angular2, Auth0, Node.js, MongoDB); Besides, a user-code executor service was built which can build and execute user’s code (Docker, Flask); Finally, system throughput was improved by decoupling services using RESTful API and loading balancing by Nginx (REST API, Nginx).



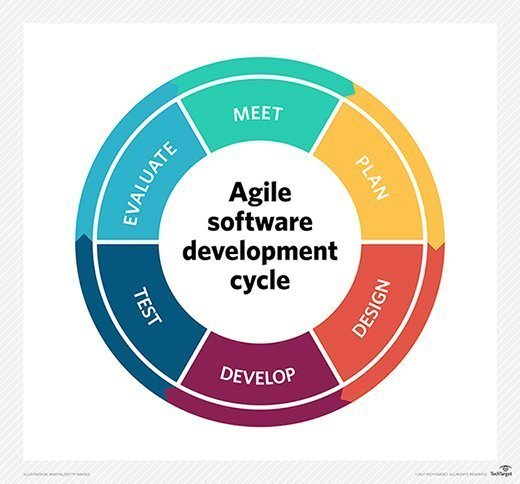
*Figure 1. Diagram for oj-client and oj-server*



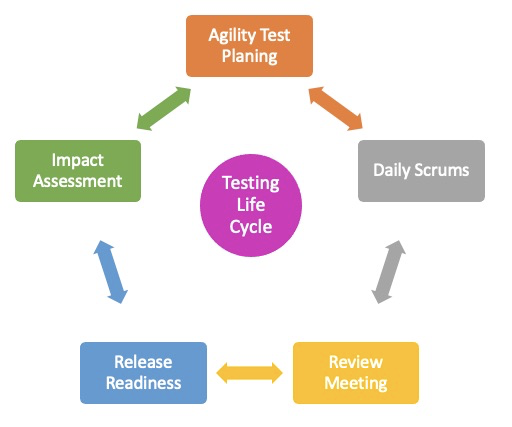
*Figure 1.1. Socket IO communications*

​​

**Test Plan:** In this project, I will focus on using the Agile development process. The Agile software development cycle can be broken down into six steps: meet, plan, Design, Develop, Test, Evaluate. The following shows this test planning cycle which including Planning, Daily scrums, review meeting, release readiness, impact assessment.



*Figure 1.2 Agile software development cycle.*

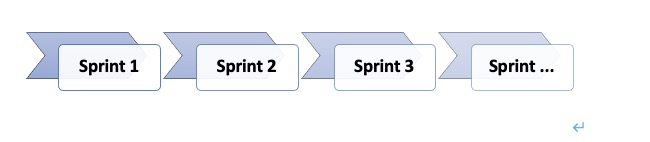
****

*Figure 1.3 Testing life cycle .*

**Test Content:** Each scrum sprint has the same process — sprint planning, daily scrums, a sprint review, and a sprint retrospective. Sprints are developmental cycles that repeat until your project is complete. Requirements are developed, tested, integrated, and approved within each sprint.

In this project, the test content includes developer requirements documents and technical reports and software requirements specifications. Three main levels: Unit test, functional test, Integration or system testing. The development version is available in GitHub for users to verify requirements and test plans:

* **Github link** : <https://github.com/chanipsophie/CollabrativeOnlineJudge>



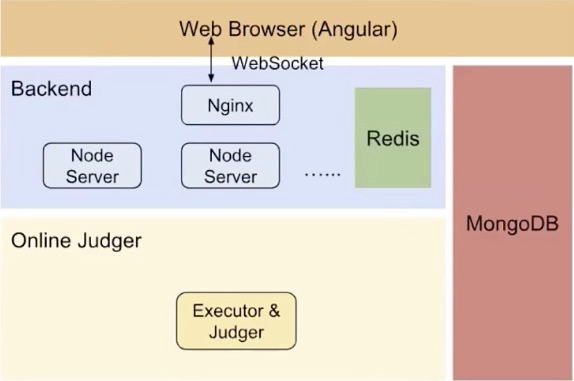
*Figure 1.4 Testing life cycle and Sprints.*

# Concrete Test Cases

**Test Methods**:

· **Unit testing**: A unit test, as Agile teams understand the term, is a short program fragment written and maintained by the developers on the product team, which exercises some narrow part of the product's source code and checks the results.

· **Integration and system testing**: System testing is a testing level in which tests are performed to know if a complete build aligns with functional and nonfunctional requirements made for it. Integration testing is a testing stage where two or more software units are joined and tested simultaneously.

****

*Figure 2.1 High Level Architecture of Collaborative Online Judge*

*Figure 2.1 shows the high level architecture of the Collaborative Online Judge. Now it is time to discuss the concrete test cases for each component of the service.*

## Client-side Web Browser

### Unit Testing

Client-side web browser has mainly five components: user profile, editor, new-problem, problem-detail, and problem-list. Before running any unit test, there should be a mocked backend service fired up with some mocked coding problems.

* **Problem-list**
  + From the mocked DB, it should show all the mocked problems in a list form, including the name, status, difficulty, acceptance rate, etc. All of these should match the data in the mocked backend service.
  + The total number of problems should match the number of problems in the mocked backend service.
* **Problem-detail**
  + For each problem in the mocked backend service, it should show exactly the same details as the testing result. These details include the problem title (name), description, difficulty, etc.
* **New-problem**
  + When CRUD problems, it should return the expected result from the mocked DB. For example, when a new problem is created, the mocked backend service should return “OK”.
* **Editor**
  + The coding editor with a specified language should appear
* **User Profile**
  + We should see a mocked user’s profile data from the testing result, which should match the one from the mocked backend service. Profile data includes name, email address, number of completed problems, number of failed problems, number of uncompleted problems, etc.

### Integration Testing

All of the above unit test cases should be added into the integration tests, using a real database and real backend service. Here we use MongoDB to store problems.

## Backend Service

### Unit Testing

Backend service should provide a list of APIs, to, for example, CRUD problems, submit problems to the judges, and so forth. This time, we will need a mocked DB before we run any unit test cases.

For CRUD problems, it essentially should be similar to the unit testing in the client web browser. The only difference is that the backend service should interact with the mocked DB to fetch mocked data.

### Integration Testing

All of the unit test cases should be added into the integration tests, using a real database. Here we use MongoDB to store problems.

In addition, we should test web sockets to ensure the communication between multiple users editing the same coding problem behaves as expected.

## Online Judger

### Unit Testing

Online judges should run a few unit test cases. In this case, we just need one or two mocked coding problems, with different support coding languages.

* For each supported coding language, online judges should successfully compile, run against multiple tests, and return expected results. Except results include pass if all tests for that code submission succeeded, or fail if one or more tests failed.

### Integration Testing

Online judges should interact with Backend service and database to run a few real coding problems. Integration test cases should be similar to those from the unit test cases; i.e. test should pass if all tests for that code submission succeeded, or fail if one or more tests failed.

## Other Types of Tests

Other than the unit tests and integration tests, there are a few other important test types which should be included during the deployment. In a typical CI/CD deployment, unit tests and integration tests are first executed, followed by the other types of tests listed below.

* **End-to-end tests**. This type of test is also called “Test on demand” or TOD. It should be run during the deployment phase. Dedicated servers and databases should be used for this type of test. Typical way to perform such a test is to write an external software to call specific HTTP requests (as a client), and check if the responses are expected.
* **Canary tests**. TODs are end-to-end tests written by an org's developers, as a result, there might be some bias or corner cases that they have missed. Many companies use third party tools to perform the so-called “canary tests”. It is as if real customers or users are using their services. One popular such tool is from Cedexis, which acts as the real customers for CDN providers. They make requests to major CDN providers and get certain results, both functional and non-functional .
* **Regression tests**. Sometimes a code change has been deployed, the whole software will go back, in terms of functionalities or performance. For functionalities, we should be able to catch such “regression” through the testing strategies discussed before. For performance regressions, we typically need a new type of testing called “regression tests”. It can be implemented in the form of load testing, to make sure any new code changes won’t make the performance worse. In our case, we should run regression testing (in the form of load testing) by making thousands of HTTP requests to fetch the problem list, problem details, and so forth. We should also make thousands of code submissions to load test the performance of online judges.