

# USTOJ Final Report

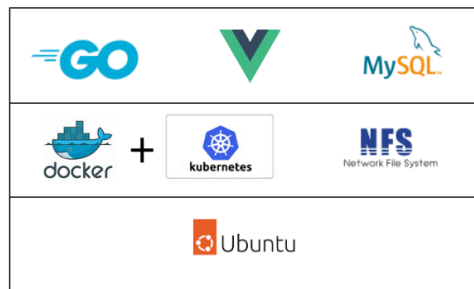
## Group 4

TANG Chaoyang (20722537), Lee Tsz Fung (20785905), PENG Shouheng (20799176), Huang Yangjun (20787329)

## Architecture

### Technology choice

USTOJ is a cloud native system based on Kubernetes. It includes front-end components and back-end components, which are developed by vue framework and golang language respectively. Both of these components support distributed transactions, so that multiple instances can run in the same cluster, and can be scaled in parallel according to the load intensity. The system uses NFS as a file system to save user data and allows all nodes to access target files depending on the specific user. For the OS system, we select Ubuntu. However, technically, the system supports all OSs that can run kubernetes on it, including centos. But we haven't tested the feasibility on the OS yet.



Though currently USTOJ uses a mysql database and an NFS file system. In subsequent updates, we will consider porting to distributed databases and distributed file systems.

### Code job

In a OJ system, A naive practice is to run submitting codes with multiple threads, which leads to several problems:

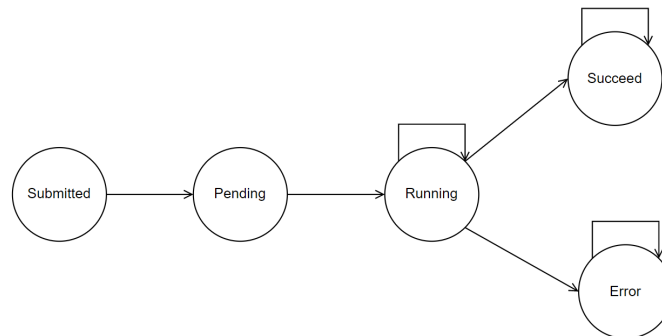
- Different users will use different languages and they have different libraries. Sometimes it is even necessary to deploy multiple environment versions for the same language. If we all submit code in one environment, these environments may affect each other and raise problems, which is not our expectation.
- If the user's code and USTOJ run in the same environment at the same time, they may encounter many security issues such as access rights control. It may also face security issues in the same memory space.

We hope to use a sandbox to run these codes. These sandboxes have the runtime environment of the corresponding language and version. After the operation is completed, the results can be left to release resources. Obviously, docker technology is well suited for this task.

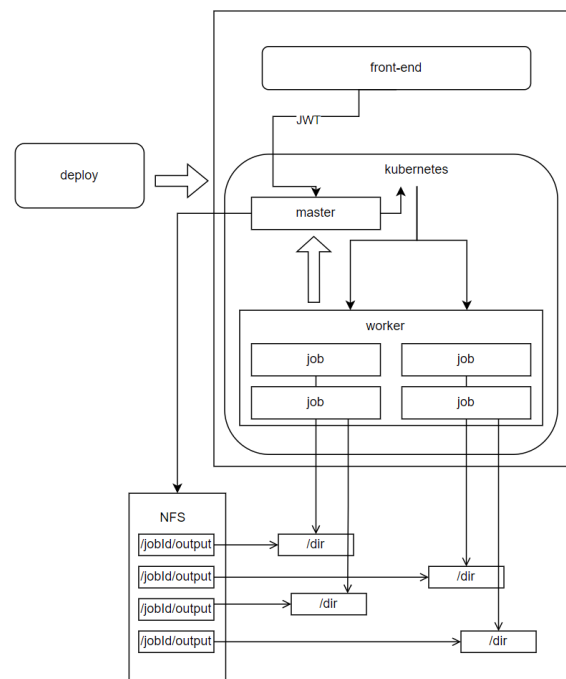
We regard each test code submitted by a user of the USTOJ system as a job, and our system will create this job on the cluster and return the running result to the user. Each job is actually a pod of K8S. There are many benefits of this design:

1. The environment is isolated, so that we no longer worry about permissions and security issues
2. We can dynamically schedule these tasks on the entire cluster, and there will be no situation where some machines shut down and cause errors in a large number of tasks
3. At the same time, with the help of kubernetes, we can easily limit the resources of these jobs, so as to avoid the situation that the user code unintentionally or maliciously occupies a large amount of resources and leads the cluster to run unhealthy.

Meanwhile, in order to realize distributed transactions, we designed the state machine of the job, so that each state is idempotent and changed in a one-way direction. Depending on our design, multiple instances can be modified at the same time without the need to design asynchronous locks to avoid the problem of simultaneous modification.



## System structure



The system architecture diagram is shown in the figure above. As the image presents, the USTOJ system is deployed on a K8S cluster, and the machines in the cluster will be divided into two roles: master and worker. The master is responsible for providing the upstream-oriented API interface, submitting the code job written by the user to K8S for running, and collecting the running status. At the same time, the master also serves as the master node of NFS and the node of the mysql database. Worker is responsible for running these code job tasks. Workers will mount the NFS export provided by the master. When the Worker runs the code jobs, it will mount a specific directory to a job pod, so

that the information left by the running job can be collected. At the same time, this directory is the only channel connecting the cluster and the sandbox. Therefore, even if it is maliciously attacked, it will not have other effects on the system. Frontend can run on any machine, and of course it can run on the same K8S cluster.

Finally, each component is independent of each other, and they are coordinated through a unified configuration. So in order to better deploy this system, we use Ansible to configure and deploy this system, we build an Ansible project to handle this job.

## Frontend

In this section, we are introducing some detailed information about our project's frontend and give some examples of the web pages.

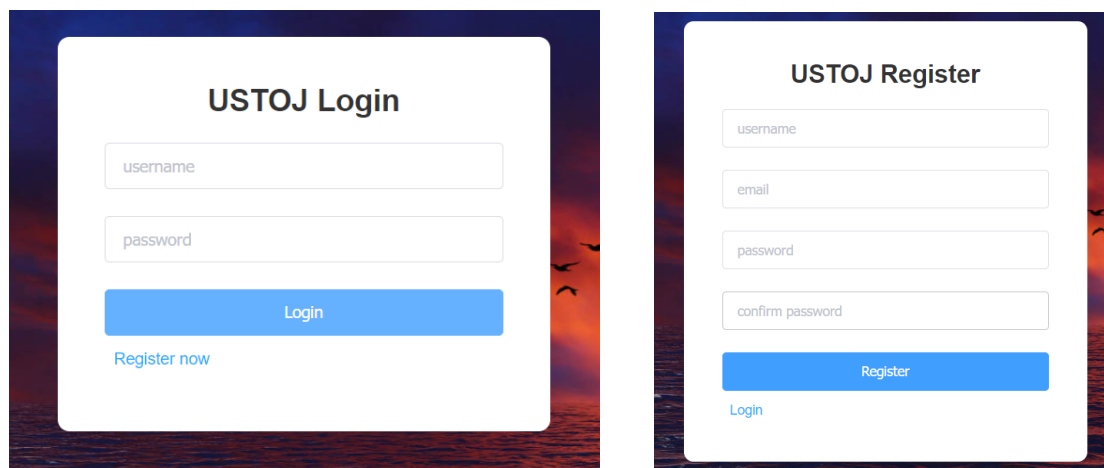
### Technical Details

In the Architecture section, it is mentioned that the UI of the USTOJ is developed using the Vue framework. The Vue is a progressive, incrementally-adoptable JavaScript framework for building UI on the web. We also use a Vue based component library named Element UI to construct the web page in our project. Element UI is built upon the principles of consistency, tangible feedback, efficiency and controllability. Vue and Element are powerful toolkits and that's why we choose them to construct the web pages.

Inside the project, we create a file named "config.json" where the backend address and frontend port are recorded. When launching the frontend, the system will load it. All network interactions between frontend and backend are HTTP-based using JWT authorization. The requests are handled by the "axios" library in JavaScript. All requests will be sent to the backend address recorded in the "config.json".

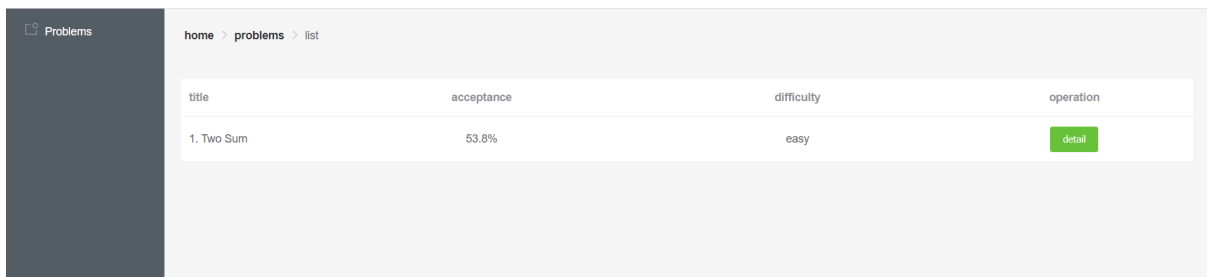
### Sample web pages

#### User management



The image displays two side-by-side screenshots of the USTOJ web application's user management interface. The left screenshot shows the 'USTOJ Login' page, which features a white card with a dark blue header. It contains two input fields for 'username' and 'password', a blue 'Login' button, and a blue link 'Register now' below the button. The right screenshot shows the 'USTOJ Register' page, also with a white card and dark blue header. It includes input fields for 'username', 'email', 'password', and 'confirm password', a blue 'Register' button, and a blue link 'Login' below the button. Both pages are set against a background image of a sunset over water with birds in flight.

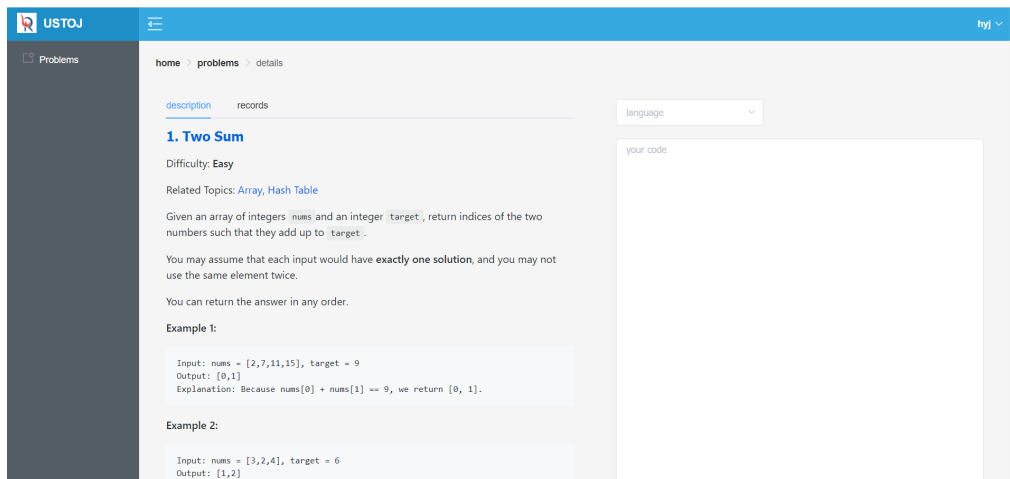
These are the login and register pages. If a user does not have an account, he can register by inputting username, email, and his password. Then, he can input the username and password to login the system.



The image shows the 'Problems' page on the USTOJ website. It features a sidebar with a 'Problems' link and a main content area with a breadcrumb 'home > problems > list'. Below the breadcrumb is a table listing problems. The first problem is '1. Two Sum' with an acceptance rate of 53.8% and a difficulty of 'easy'. A green 'detail' button is located to the right of the 'easy' difficulty label.

title	acceptance	difficulty	operation
1. Two Sum	53.8%	easy	<a href="#">detail</a>

After logging in, we can see the problem list in USTOJ. We can see the title, the acceptance rate, and the difficulty of each problem. We then click the “detail” button to get more information.



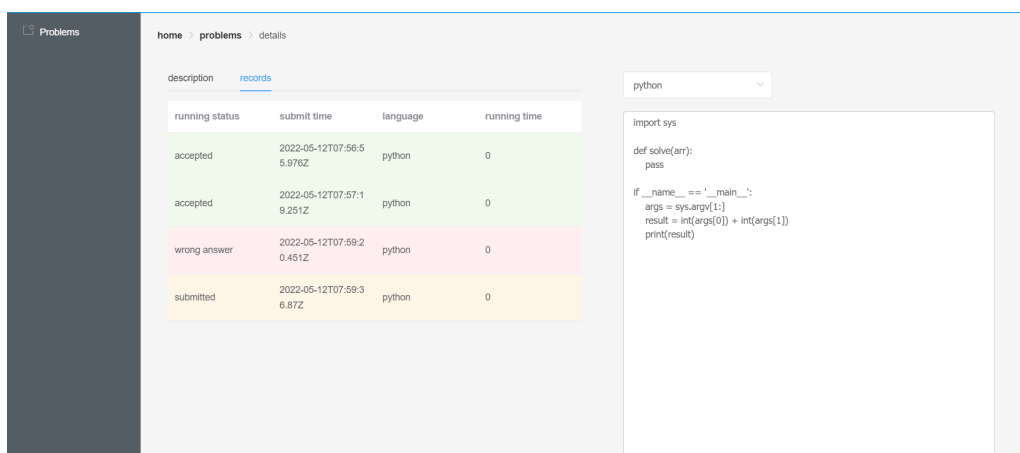
The image shows the 'details' page for the '1. Two Sum' problem. The left sidebar contains the problem title '1. Two Sum', its difficulty 'Easy', related topics 'Array, Hash Table', and a description. The right side of the page has a 'language' dropdown menu and a 'your code' text area for submitting a solution.

**1. Two Sum**  
 Difficulty: Easy  
 Related Topics: [Array](#), [Hash Table](#)  
 Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.  
 You may assume that each input would have **exactly one solution**, and you may not use the same element twice.  
 You can return the answer in any order.

**Example 1:**  
 Input: `nums = [2,7,11,15], target = 9`  
 Output: `[0,1]`  
 Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

**Example 2:**  
 Input: `nums = [3,2,4], target = 6`  
 Output: `[1,2]`

In the problem details page, we can see the problem description on the left and code area on the right. We select the python as the programming language, type in our code and then submit the code.



The image shows the 'details' page for the '1. Two Sum' problem after a submission. The left sidebar now shows the 'records' tab, which displays a table of submission records. The right side of the page shows the 'python' language selected in the dropdown menu and a code editor containing a Python solution.

running status	submit time	language	running time
accepted	2022-05-12T07:56:5 6.976Z	python	0
accepted	2022-05-12T07:57:1 9.251Z	python	0
wrong answer	2022-05-12T07:59:2 0.451Z	python	0
submitted	2022-05-12T07:59:3 6.87Z	python	0

```
import sys

def solve(arr):
    pass

if __name__ == '__main__':
    args = sys.argv[1:]
    result = int(args[0]) + int(args[1])
    print(result)
```

After we submit the code, the left navigation bar will switch to the submitting records. In the record list, we can see the submitted status. The front end will refresh the page every 5 seconds.

description		records		python	
running status	submit time	language	running time		
accepted	2022-05-12T07:56:55.976Z	python	0		
accepted	2022-05-12T07:57:19.281Z	python	0		
wrong answer	2022-05-12T07:59:20.481Z	python	0		
accepted	2022-05-12T07:59:36.87Z	python	0		

```

import sys

def solve(arr):
    pass

if __name__ == '__main__':
    args = sys.argv[1:]
    result = int(args[0]) + int(args[1])
    print(result)

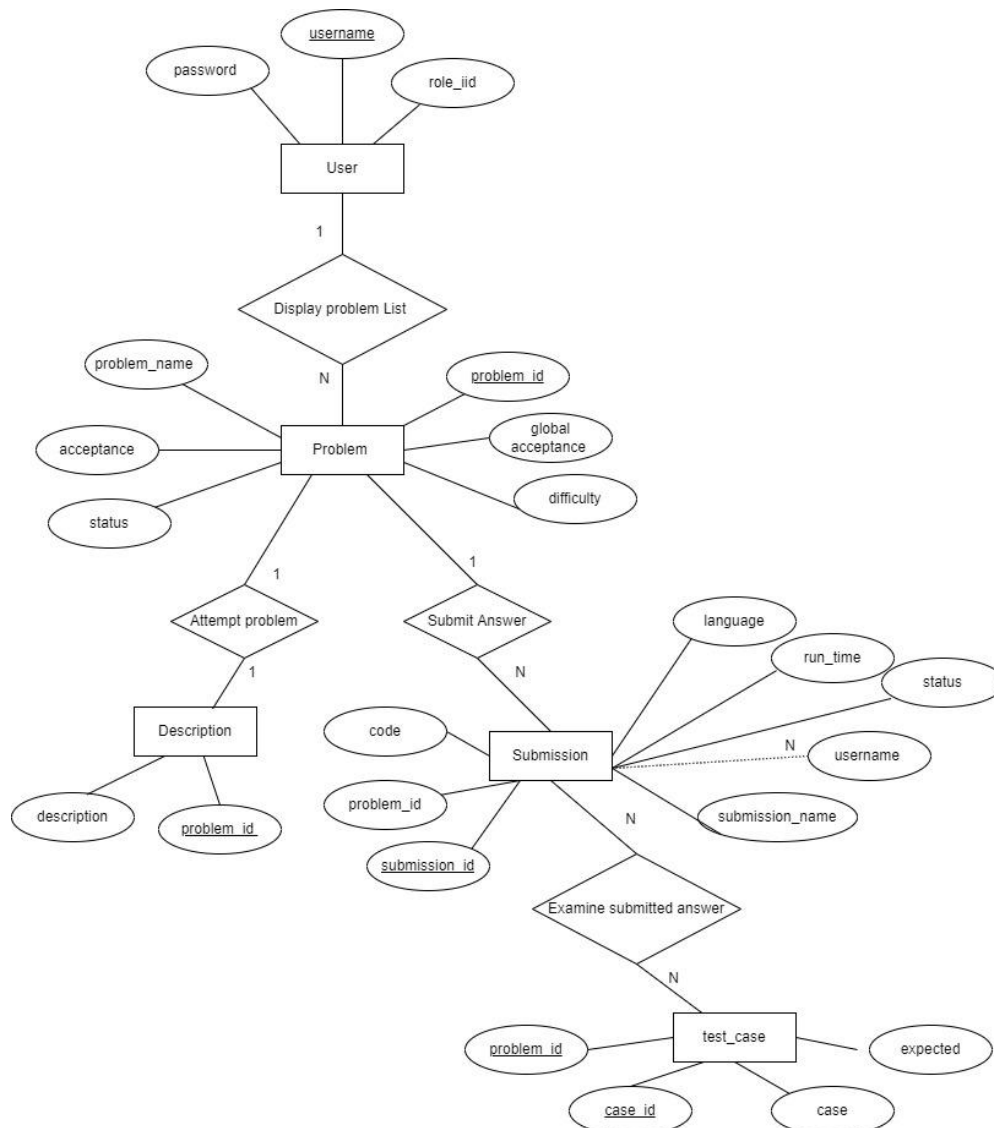
```

About 5 seconds later, we can see the code is accepted by the system. That's the example of the major process in USTOJ.

## Backend

### Model Design

#### ER-diagram



In order to implement the above functional requirements, there is an ER diagram designed to describe the data requirements for our system. There are five entity type sets involved in our system which are User, Problem, Description, Submission, and test\_case.

As the above ER diagram captures the different used collection of data and their relationship in the whole online judgment process, it is easily transformed into a relational database schema.

The following table is the DB schema in the implementation data model of DBMS

### **User**

<u>username</u>	password	role_id
-----------------	----------	---------

For the user entity set, the user's name (username) is unique and necessary to identify each user. The role id is to distinguish different groups of users. It is developed for future extension.

After logging in, the user can see a problem list that contains all the problems with simple information. Therefore, there is a one-to-many relationship between the user and the problem.

### **Problem**

<u>problem_id</u>	problem_name	status	difficulty	acceptance	global_acceptance
-------------------	--------------	--------	------------	------------	-------------------

For the problem entity set, the problem id of the problem is unique and necessary, we can use it as the primary key. Each problem has a problem id, problem, acceptance, status, global acceptance, acceptance, and difficulty. The attribute problem\_name represents the name of the problem. Acceptance represents the passing rate of use for this problem. Global Acceptance represents the passing rate of the overall user. Difficulty indicates how difficult the problem is.

### **Description**

<u>problem_id</u>	description
-------------------	-------------

For the description entity set, the problem id of the problem is unique and necessary, we can use it as the primary key. The attribute description represents the description of the specific problem.

When the user clicks the “detail” button to try to attempt the problem, a detailed description of the chosen problem will be provided to the user. Therefore, there is a one-to-one relationship between problem and description.

### **Submission**

<u>submission_id</u>	submission_name	problem_id	usrname	code	language	status	run_time
----------------------	-----------------	------------	---------	------	----------	--------	----------

For the submission entity set, the submission id of the problem is unique and necessary, we can use it as the primary key. The attribute code represents the answer created by the user. Language is the programming language that users use to answer problems. Each user may submit the answer to a specific problem many times to the system for online judgment so the same user name allows multiple occurrences of the same problem. Each submission has a submission id, problem id, code, submission

name, status, user name, run time, and language. There is a one-to-many relationship between problem and submission.

### Test\_case

<u>problem_id</u>	<u>case_id</u>	case	expected
-------------------	----------------	------	----------

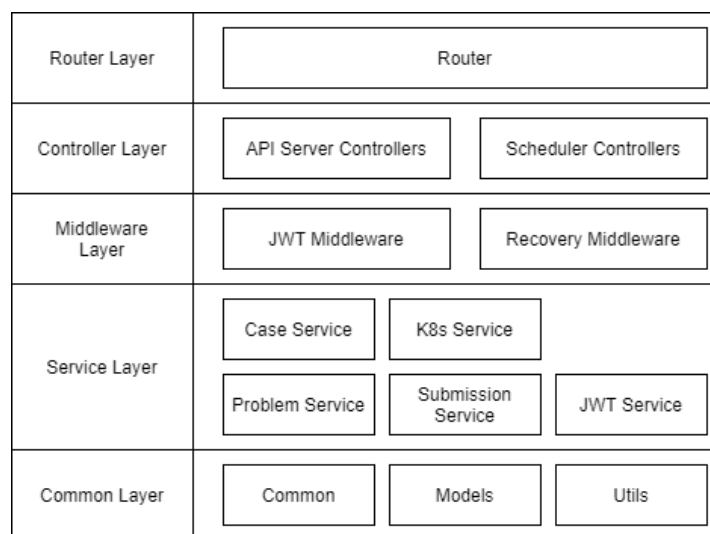
For the test case entity set, problem id and case id is used as the primary key to identify different test cases of different problems. The attribute case represents the testing input of the problem for examining the code quality of the submitted answer. Expected represents the expected result of the test case. Each test case has a problem id, case id, case, and expected.

After the user finishes and submits the code answer, the system can receive the submitted answer. During the “submitting records” status, the system will use several test cases of problems to examine the code quality of the submitted answers in Kubernetes. As a result, there is a many-to-many relationship between submission and test cases. At the end of examining, the system will update the status and run time of submission.

## Backend Implement

### Overall Structure

As the following figure shows, there are basically five layers in the backend, namely common layer, service layer, middleware layer, controller layer, and router layer.



The common layer contains some common and public components, defining models for data, providing database connection pool, providing logging framework, providing necessary utils, etc.

The service layer contains the implementation of all service logic that the controller layer may want to use.

The middleware layer mainly handles JWT (Json Web Token) authentication and error recovery for user requests.

The controller layer contains two parts: API server controllers and scheduler controllers. API server controllers are responsible for handling user requests from the router layer, while scheduler controllers

are responsible for retrieving available submissions and submitting them to kubernetes for execution, and checking whether the outputs are the same as we expected.

The router layer is responsible for processing user requests and routing them to corresponding controllers for processing.

The backend architecture is like a microservice architecture. There are two microservices: “API Server” and “Scheduler”.

## API Server Design

The API server is responsible for handling user requests from the frontend. It can be divided into three parts: authentication, problem-related processing, and submission-related processing.

For validating the user identity during the whole online judgment process, we use the JSON Web Tokens (JWT) to provide authorization. Once the user is logged in, every subsequent request will contain the JWT which allows the user to access the routes, services, and resources allowed by the token.

The JWT consists of three parts which are header, payload, and signature. The first two parts are the JSON object. The header contains the necessary information for verifying the signature. The payload includes the claims which are statements about the entity (user). The header and payload and a secret key will be encoded for generating the signature. The signature can be used to check whether the message was changed along the way.

The reason for us to use JWT in the user management mechanism is that it is suitable for the situation of multiple servers because the token is not necessary to store in the session in the server's database.

As we chose the gin this website framework to create our application, we used jwt-go this library to provide JWT generation and authorization service. There are several steps to generate the JWT.

First, when the user log in to the application, the user name and password will be checked whether the same user name can be found in the database and the password is correct.

Second, the custom claims have declared to define the JWT. The standard claim of JWT is a part of the custom claim. After all required information is inside the custom claim. We choose the HMAC-SHA 256 as a signing method to generate a token.

Then, it will use the secret key with a token to generate the signing string.

Finally, the signed token will return to the client and the user will use it to access the protected router, resource, etc.

We created a middleware that is used to allow the validated user only to use the online judgment functions. When a user wants to access some functions of the system, he needs to pass the token validation of the router. The token is in the header of http so the server can use `GetHeader("Authorization")` function to obtain the token from the client side. Next, the server will parse and validate the token. The secret key will be returned for validation if the signature of the token is correct. Finally, the validation of the router is passed and the user can access the allowed function.

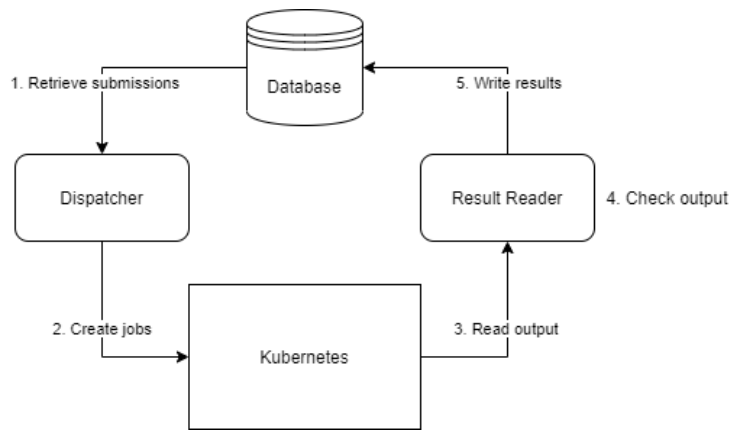
Problem-related processing is mainly about handling queries about problems. The users may want to do a range query among available problems or query the details of a specific problem.

As for submission-related processing, it handles users' submissions, stores them in the database, and handles user queries for their submitted results.



## Scheduler Design

In the scheduler module, there are two sub-modules: the dispatcher and the result reader. The dispatcher is responsible for retrieving available submissions and corresponding information and submitting them to kubernetes for execution. The result reader would check the job status in Kubernetes. Once there is a job completed, it would read the output and check with the expected output to define whether the result should be “accepted” or “wrong answer”. The following figure shows the whole workflow.



## Automatic Deployment

As it is mentioned in “Architecture” section, we need a component to link these independent components through a unified configuration. This component will automatically setup the basic environment, including k8s setup and NFS setup, for running the USTOJ system from a blank Ubuntu system. After that, it will automatically generate configs for each component and make sure these configs are related so that they will work together healthily. Then it should deploy the components into the target k8s cluster.

We use Ansible to handle these complexities, which can manage machines quickly and in parallel, and describe infrastructure in a language that is both machine and human friendly compared to shell scripts. The coolest thing is that its operation is idempotent, so we don’t worry to deploy the system even if it is cracked in the middle of deployment.

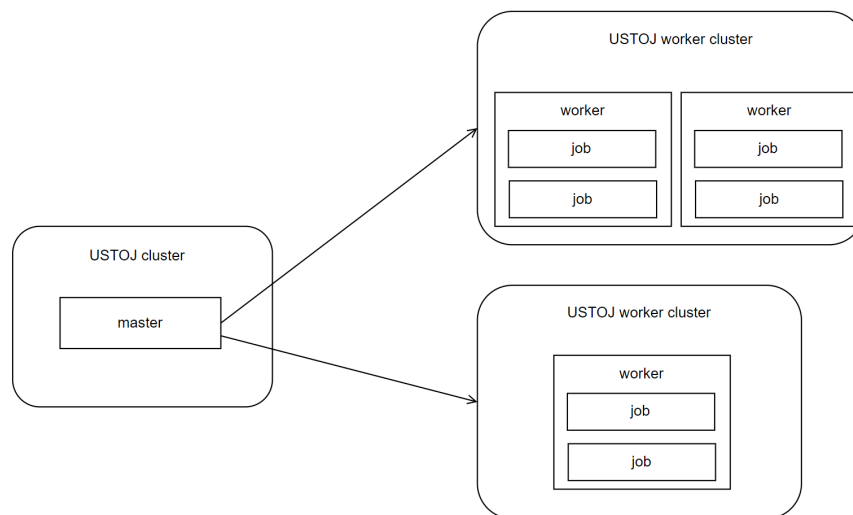
We set up the following roles for Ansibles to deploy USTOJ:

- **k8s\_master**: The machine to run k8s master node, we remove the master taint so that it can run our application.
- **k8s\_worker**: The machine to run k8s worker node.
- **mysql\_server**: The machine to run mysql. The database file is persistent on the local file system so that the system will retain the data after reboot.
- **nfs\_server**: The machine plays the NFS exporter role, and provides NFS service.
- **ustoj\_frontend**: The machine to run `ustoj_frontend`. This machine is not required to be in the same cluster of USTOJ.
- **ustoj\_master**: The machine to run USTOJ master components, including api-server and scheduler.
- **ustoj\_worker**: The machine to run USTOJ code job. K8S will allocate jobs on these machines.

## Future work

### Multiple clusters handling

During the development of USTOJ system, we realize that our system is more powerful than we expect. It has the potential to handle multiple Kubernetes clusters. This is an expansion worth looking forward to because sometimes we have more than one k8s cluster or our machines are used for another applications in advance. This update can give us a stronger ability to deal with more code submitted by users.



### Distributed file system and database

As it is mentioned in the “Architecture” section, now we use Mysql database and NFS for sharing data. But these two choices contain a potential risk, if the database node or the NFS node is shutdown for any reason, our system will stop working. In another words, there is no high availability on database and file sharing system.

Therefore, it will be better if we can choose some distributed database and file systems so that the system will still be running even if some of the machines in the cluster are shutdown. The HBase from Hadoop ecosystem is a good choice for distributed database. And HDFS or Ceph is also a potential choice for file system.

Luckily, our system is easy to accept these updating by updating the ORM module and update some file related modules of the project.

### Use Helm for better configuring

The deployment is using Ansible to render the template for configmap, deployment, service applied for our system. However, considering that our application is a K8S application, we can use Helm to deploy our application and better upgrade the cluster we have deployed. The deployment will be more human friendly while applying Helm. It becomes as simple as deploying the Helm application, instead of going through complicated rendering steps and ugly scripts to get the final configuration.