Shift Schedule Builder Desktop - Client Server

Yu-Chun Lin and Kuei-Ching Yang

December 6, 2023

1 Description

Shift Schedule Builder Desktop is an application designed to simplify the task of shift arrangement for employees. Its primary aim is to provide a user-friendly interface that allows users to input their shift requirements and configurations easily and generate a shift that satisfies these shift requirements as much as possible.

The user can set up some basic shift parameters such as the number of days and the number of employees. The application will generate an empty shift that satisfies the basic requirement. In addition to basic shift parameters, users are able to add some shift requirements. The application provides several common shift requirements. For example, users are able to set up the expected number of working days in the given range of the shift. The application will automatically generate a schedule for them based on the given data and constraints. Since the scheduling problem is a multi-objective optimization problem, the users have to specify the weight of each constraint. If the user values a shift requirement a lot, the user has to specify a bigger weight on that shift requirement.

The project is developed using Python, HTML, Javascript, and CSS.

1.1 UI Sketch

We developed the UI by using HTML, Javascript, and CSS for both desktop-based client and a web-based client. We employed **QWebEngineView** and **QWebEnginePage** to embed the HTML, Javascript, and CSS code in the desktop-based client. We opted to integrate the website into the desktop-based client primarily due to the website's more user-friendly and aesthetically pleasing design. Furthermore, there is no gap or learning curve for users transitioning from the web-based application to the desktop-based application. Both applications maintain a consistent user experience.

1.1.1 Login

In the beginning, the user has to log in. The login page will collect the username and password of the user to check and set up different users. The login page serves as a view and controller in the application. After collecting the user's information, it will send the information to the backend's database adapter to check if the user exists or not.

After receiving the user's information, the application will store the information by using Cookies.

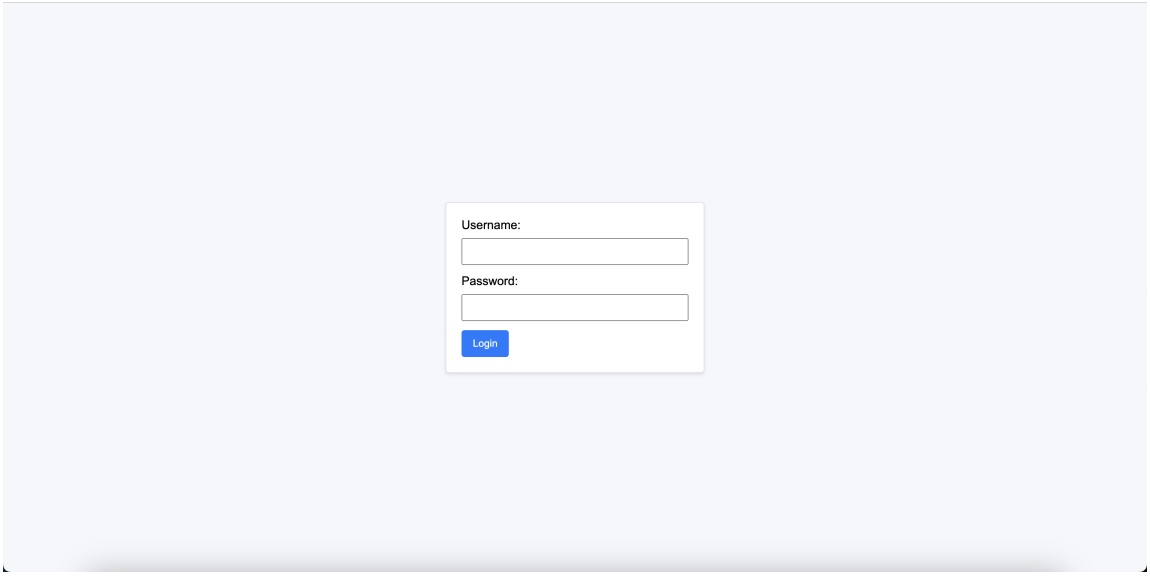

Figure 1: User Interface - Login Dialog

1.1.2 Main Page

The main page enables users to create multiple shifts by adding tabs and setting up the parameters. The main page is composed of several components. The UI components are shown in the figure below. The details of the design pattern will be elaborated in the architecture design section.

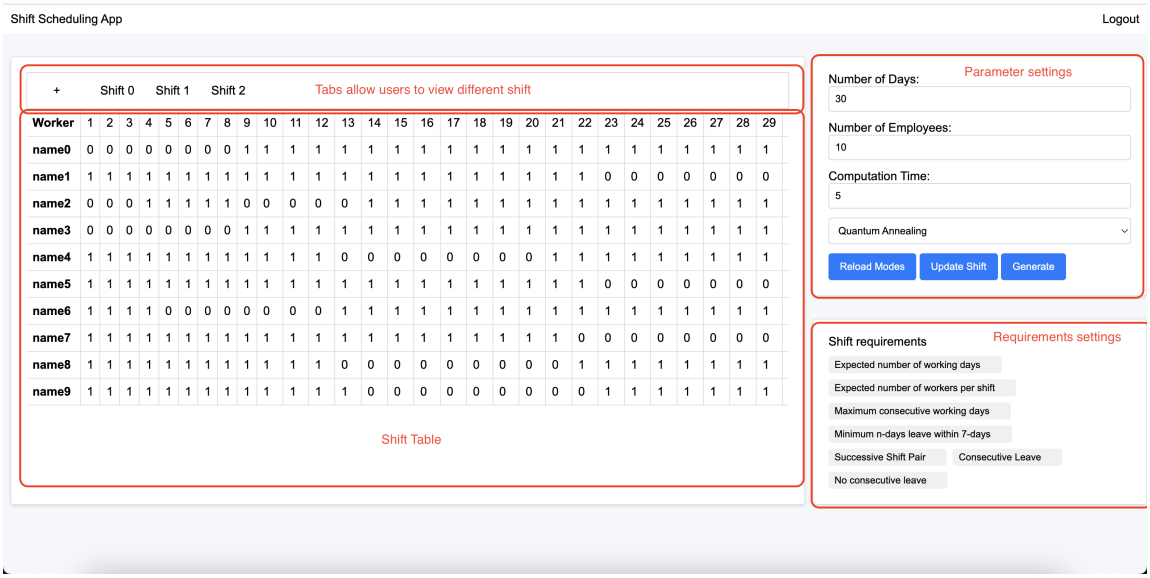

Figure 2: User Interface - Main Window

1.2 Main use cases

1.2.1 Login

Use case: Login

Actor: System user

Goal: To login to the system

Overview: When the users open the application, they have to sign in to the system. If they type their username and password correctly, they are able to sign in. However, if they type either username or password wrong, the system will pop up a window showing a message that tells the users they failed to sign in.

Actor action:

1. user: A user types username and password
2. user: The user clicks the login button
3. system: If both username and password are correct, then the system will allow the user to log in; otherwise, the system will pop up a message that tells the user that the given information is not correct.
4. user: If the user fails to login, the user may click the close button to close the application; otherwise, the system will close the login dialog window and show the application's user interface.

Alternative courses:

Steps 1 and 2: The user may click the close button to close the application.

1.2.2 Create a new shift

Use Case: Create a new shift

Actor: System user

Goal: To create a new empty shift

Overview: Before the application generates a shift for the user, the user has to create a new empty shift.

Actor action:

1. user: A user clicks the '+' button at the tab line.
2. system: The system create a new empty shift.
3. user: The user may set up shift parameters.

Alternative courses:

Steps 1 and 2: The user may click the close button to close the application.

1.2.3 Set up shift parameters

Use case: Set up shift parameters

Actor: System user

Goal: To set up shift parameters

Overview: Before the application generates a shift for the user, the user has to specify the shift parameters. The parameters include days, number of workers, computation time, and running mode.

Actor action:

1. user: A user types the number of days in the parameters form.
2. system: The system regenerates a shift based on a new parameter configuration.
3. user: The user types the number of workers in the parameter form.
4. system: The system regenerates a shift based on a new parameter configuration.
5. user: The user types the computation time in the parameter form.

Alternative courses:

Steps 1 and 2: The user may click the close button to close the application.

1.2.4 Sets up shift requirements

user case: Set up shift requirements

Actor: System user

Goal: To sets up shift requirements

Overview: Before the application generates a shift for the user, the user is able to give some shift requirements. The available shift requirements are the expected number of working days, the expected number of workers per shift, the successive shift pair, the consecutive 2 days off, no more than consecutive days off, the maximum number of consecutive shifts, and the minimum number of N days off within 7 days.

Actor action:

1. user: The user clicks one of the shift requirements.
2. system: The system pops up a dialog that shows the description of the parameter and ask the user to provide the parameter of the shift requirement.
3. user: The user provides the preferred parameters.
4. user: The user clicks the Add button to add the parameters.

Alternative courses

Steps 3 and 4: The user may click the close button to close the dialog.

1.2.5 Generate a shift

user case: Generate a shift requirements

Actor: System user

Goal: To generate a shift

Overview: After the user finishes setting up the parameters and requirements, the user is ready to generate a shift. The user can click the "Generate" button to start generating the shift.

Actor action:

1. user: The user clicks the Generate button to start generating a shift.
2. system: The Generate button becomes disabled and starts generating a shift for the user.
3. system: The progress bar will keep posted on the status of the task.
4. system: When finishing generating a shift, the system shows the shift on the user interface.
5. system: The Generate button will be unlocked so that users can re-run the task.

Alternative courses

Steps 3 and 4: The user may click the close button to close the dialog.

1.2.6 Save a shift

user case: Save a shift

Actor: System user

Goal: To save a shift

Overview: After acquiring a shift, the user is able to save his/her shift to the device.

Actor action:

1. user: The user clicks download button on the website.
2. system: The system directly download the file to user's device.
3. user: The user open the file at the default download directory.

1.2.7 Logout

user case: Logout

Actor: System user

Goal: To log out

Overview: Users can log out of the system or switch to a different user.

Actor action:

1. user: The user clicks logout button on the website at navigation bar area.
2. system: The system delete the Cookies and return to the login page.

2 Database design

The data models of the application are intricate. The database not only has to store the user's information but also the shift data. Since users are allowed to set up multiple shift requirements, the number of shift constraints will vary. Besides, the data size of the shift will vary based on the number of days and the number of workers. It is intuitive to use the NoSQL database to store and load the data; thus, we choose to use Redis and MongoDB to store the data. Redis is responsible for storing the user's data. The key is the combination of user's username and password. The data of the user stored in Redis is in JSON format. The document and the data model of the application are compatible. In MongoDB we create a collection to store shift data. The entity relationship diagram is shown below.

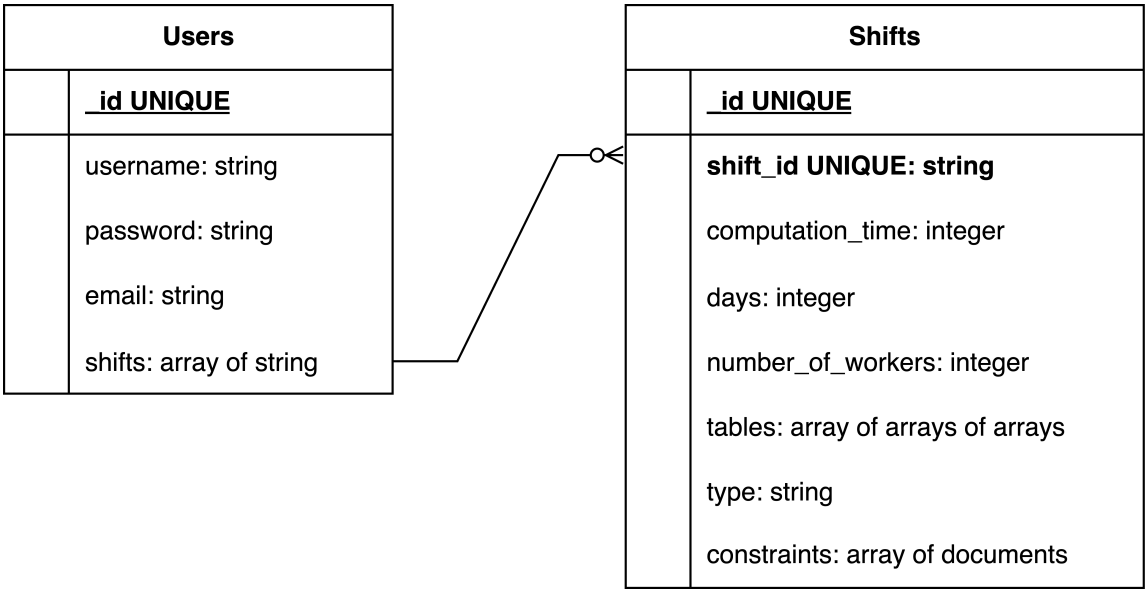

Figure 3: Entity Relationship Diagram

2.1 Users

In Redis, the database stores the basic information of a user such as username, password, email, and shifts. Here is an example of a entry.

```
{
    username: 'admin',
    password: 'password',
    email: 'email@gmail.com',
    shifts: [
      '66d74521-6947-4cd3-b2e9-d201140893bb',
      '0926ea12-8907-43e7-a2cb-340043eb7c12',
      '9223cea4-69f8-4dab-8832-4b48c952d91d',
      '0750a390-7f19-4341-9aa2-f41ccfc2c0f9'
    ]
}
```

The description of those fields is as follows.
1. username: The username of the user.
2. password: The password of the user.
3. email: The email of the user.
4. shifts: A nested document that stores the shift is owned by the user. In this example, the user owns 4 shifts. The shifts ID will be used to load the shifts in the Shifts collection.

2.2 Shifts

The collection stores shifts' information. The shifts' information includes shift_id, parameters, and the content of the shift. Here is a snippet of a document in the collection.

```
[
  {
    _id: ObjectId("6519d6a87b2c1ef5a957ae5c"),
    shift_id: '594f91af-cade-47f3-aeff-a0d461285e1a',
    computation_time: 30,
    constraints: [
      {
        name: 'expected_number_of_working_days',
        parameters: { weight: '10', ewd: '20', days_off_index: [] }
      },
      {
        name: 'expected_number_of_workers_per_shift',
        parameters: { weight: '10', enwps: '8', days_off_index: [] }
      },
      {
        name: 'consecutive_2_days_leave',
        parameters: { weight: '10', days_off_index: [] }
      },
      {
        name: 'minimum_n_days_leave_within_7_days',
        parameters: { weight: '10', mndlw7d: '2', days_off_index: [] }
      }
    ],
    days: 30,
    name_list: [
      'name1', 'name2',
      'name3', 'name4',
      'name5', 'name6',
      'name7', 'name8',
      'name9', 'name10'
    ],
    number_of_workers: 10,
    tables: [
      [
        [
          1, 1, 1, 1, 0, 0, 1, 1, 1,
          1, 1, 0, 0, 1, 1, 1, 1, 1,
```

```
          1, 1, 0, 0, 0, 0, 0, 1, 1,
          1, 1, 1
        ],
        ...
        [
          1, 1, 0, 0, 1, 1, 1, 1, 1,
          1, 0, 0, 1, 1, 1, 1, 1, 0,
          0, 1, 1, 1, 1, 1, 1, 1, 1,
          0, 0, 0
        ]
      ],
      [
       [
          1, 1, 1, 1, 0, 0, 1, 1, 1,
          1, 1, 0, 0, 1, 1, 1, 1, 1,
          1, 1, 0, 0, 0, 0, 0, 1, 1,
          1, 1, 1
        ],
        ...
        [
          1, 1, 0, 0, 1, 1, 1, 1, 1,
          1, 0, 0, 1, 1, 1, 1, 1, 0,
          0, 1, 1, 1, 1, 1, 1, 1, 1,
          0, 0, 1
        ]

      ]
    ],
    type: 'DAU'
  }
]
```

1. _id: A unique object id that is automatically generated by the database.
2. shift_id: The id of the shift. The user entity will also store **shift_id** in the **shifts** field.
3. computation_time: The computation time of the shift.
4. constraints: The shift requirements. The field is a nested document that stores the name of the requirement and parameters.
5. days: The number of days
6. name_list : The name list.
7. number_of_workers: The number of workers in the shift.
8. tables: Tables are a 3-D array that can store multiple shifts. Each shift is a 2-D array.
9. type: The type of optimizer that solves the shift scheduling optimization problem. The type field will either be 'SA' or 'DAU'. SA means the optimization algorithm is a simulated annealing algorithm, and DAU means the problem is solved by using the digital annealing unit.

2.3 Sample Code

Since we use the NoSQL database, we are only able to provide the sample code that we used to insert, search, and update the document. The following code snippets are parts of the class DataAdapter.

2.3.1 Search user

```
redis_client = redis.StrictRedis(
                    host='redis-12297.c302.asia-northeast1-1.gce.cloud.redislabs.com',
                    port=12297,
                    password=''
                )
key = "eugenepassword"
json_data = redis_client.execute_command('JSON.GET', key)
user_data = json.loads(json_data)
print(json.dumps(user_data, indent=4))
```

The result is

```
{
    "username": "eugene",
    "password": "password",
    "email": "email@gmail.com",
    "shifts": [
        "6885d765-1b7c-680f-f583-9011e385a991",
        "e31f03eb-a7e6-580a-e602-21e8fb22ce20",
        "533cdbce-1386-5971-ec73-fff22325ca81"
    ]
}
```

2.3.2 Update user's shifts

The following code snippet updates the user's shifts by using the update_one function.

```
user_collection = self.db.Users
user_collection.update_one(
{"username": user.getUsername()},
{"$set": {"shifts": user.getShifts()}}
)
```

2.3.3 Save a shift

The following code snippet uses update_one to save shifts, but we set upsert=True for the case that the shift doesn't originally exist. In that case, it will insert a document.

```
collection = self.db.Shifts
inserted_data = shift.getShiftConfiguration()
inserted_data['table'] = tables
collection.update_one(
{"shift_id" : shift.getShiftId()},
```

```
{"$set": inserted_data},
upsert=True
)
```

2.3.4 Load a shift

The following code snippet will find a shift based on given shift_id.

```
shift = self.db.Shifts.find_one({"shift_id": shift_id})
```

3 Architectural design

The scope of the application includes the user interface, server, database, and solvers. The user interface collects the user's preference for the shift. The server components process the user's requests to load/save the data and handle the computation. The database stores and loads the users' information and shifts. Solvers are used to solve the optimization problem to generate the shift that satisfies the requirements as much as possible. After acquiring the solution from Fujitsu's server or the local solver at the server, Solvers will return the shift back to view/controller to show on the user interface. In this version, solvers use a web socket to do the communication. This allows the server to communicate the client. After the View/controller receives the shift data, it will send the data back to the server to store the data in the database.

In this version, the server-side architecture is designed as a microservices architecture. An API gateway is responsible for managing fundamental requests, including tasks such as login, loading/saving shifts, and querying available services for clients. Additionally, the API gateway oversees the registration of microservices. When a microservice is activated, it sends a registration request to the microservices gateway. Subsequently, when a client sends a request to inquire about the currently available services, the gateway initiates a health check by sending requests to each microservice to ensure their operational status.

The following section will give details on the architecture design for each component.

3.1 Server

We developed a tiny framework that enables us to flexibly extend the function of the server. In this project, the API gateway, and two microservices are developed under this framework. Each part is able to establish a HTTP server by initializing **HttpServer** and a websocket server by initializing **WebsocketServer**. Each server can register several routes for clients to operate.

Clients' requests are handled by each route. The route will response the data directly through the established socket connection based on the protocol. For example, a microservice that handles the simulated annealing solver is shown in Fig. 5

Here is a snippet of code that shows how to use this tiny framework.

```
class TestRoute(Route):

    def handle(self):
        data = {
            "message": "successful"
        }
```

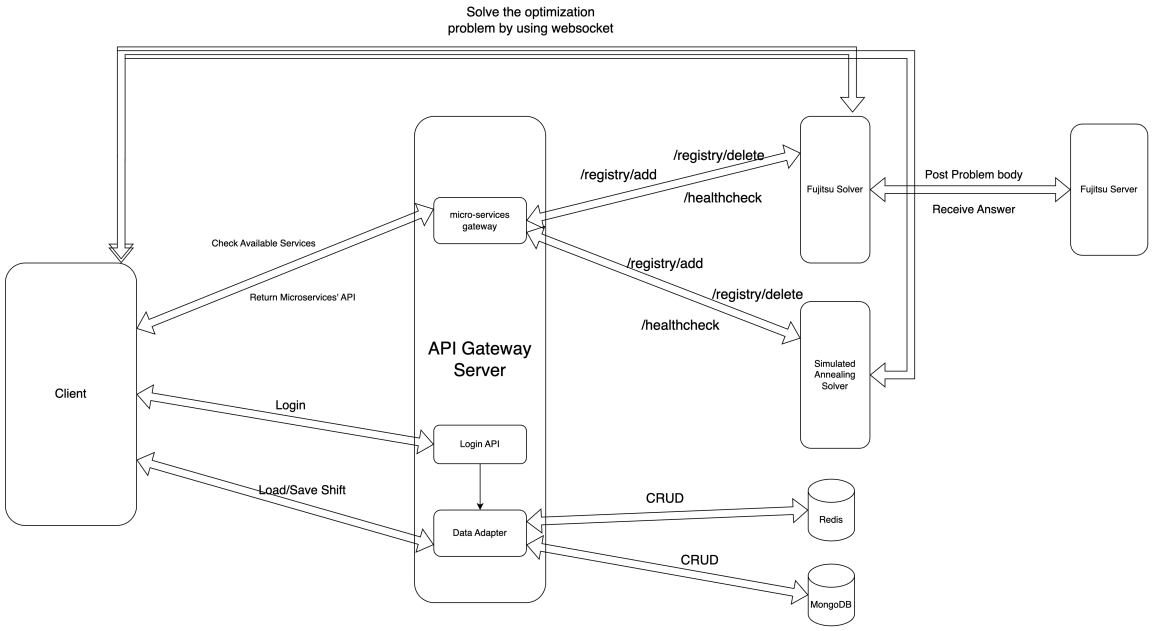

Figure 4: System Architecture

```
        self.response.send(200, json.dumps(data), content_type="application/json")


class EchoWebsocketRoute(WebSocketRoute):

    def handle(self):
        while True:
            received_message = self._recv()
            print("Received Text: ", received_message)
            if received_message is None:
                break
            else:
                self.response.send(received_message)

if __name__ == '__main__':
    server = ProtocolTypeRouter({
        'http': HttpServer(routes=[(r'/', TestRoute)]),
        'websocket': WebSocketServer(routes=[(r'/chat', EchoWebsocketRoute)])
    })
    server.run()
```

We just need to inherit the Route type override the handle function, and register the Route type to the corresponding server, and specify the URI. This enables us to extend the function of the server easily. Fig. 6 details the UML of the server side.

3.1.1 RemoteSolver

RemoteSolver is a class that establishes the web socket connection between UI and the server. It also processes the message sent from the server side and connects itself and the

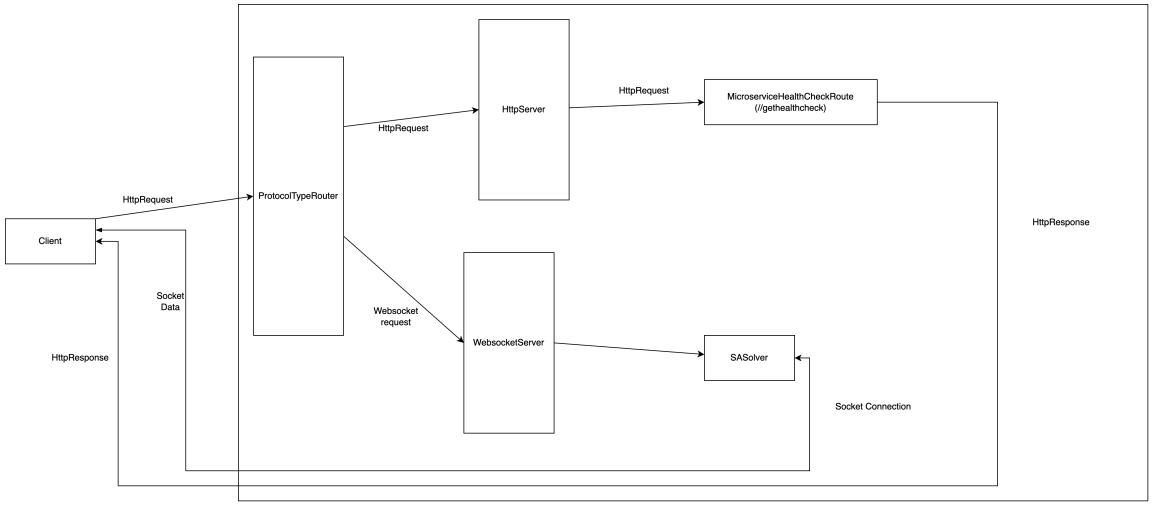

Figure 5: The architecture that uses this tiny framework to create a micro-service.

progress bar to update the status.

We wrote two remote solvers, RemoteDAUSolver and RemoteSASolver. The corresponding APIs are ws://localhost:8888/dau and ws://localhost:8888/sa. Both solvers inherit from RemoteSolver. When the function solve is called, it will use a web socket to send the optimization problem to the server. Here is a simple example of the data. The data format in the communication is in the JSON format.

```
{
    "type": "SA",
    "days": 30,
    "number_of_workers": 10,
    "computation_time": 10000,
    "reserved_leave": [],
    "constraints": [
        {
            "name": "expected_number_of_working_days",
            "parameters": {
                "weight": "10",
                "ewd": "20"
            }
        },
        {
            "name": "expected_number_of_workers_per_shift",
            "parameters": {
                "weight": "8",
                "enwps": "8"
            }
        }
    ],
    "content": [
        {
            "name": "name1",
```

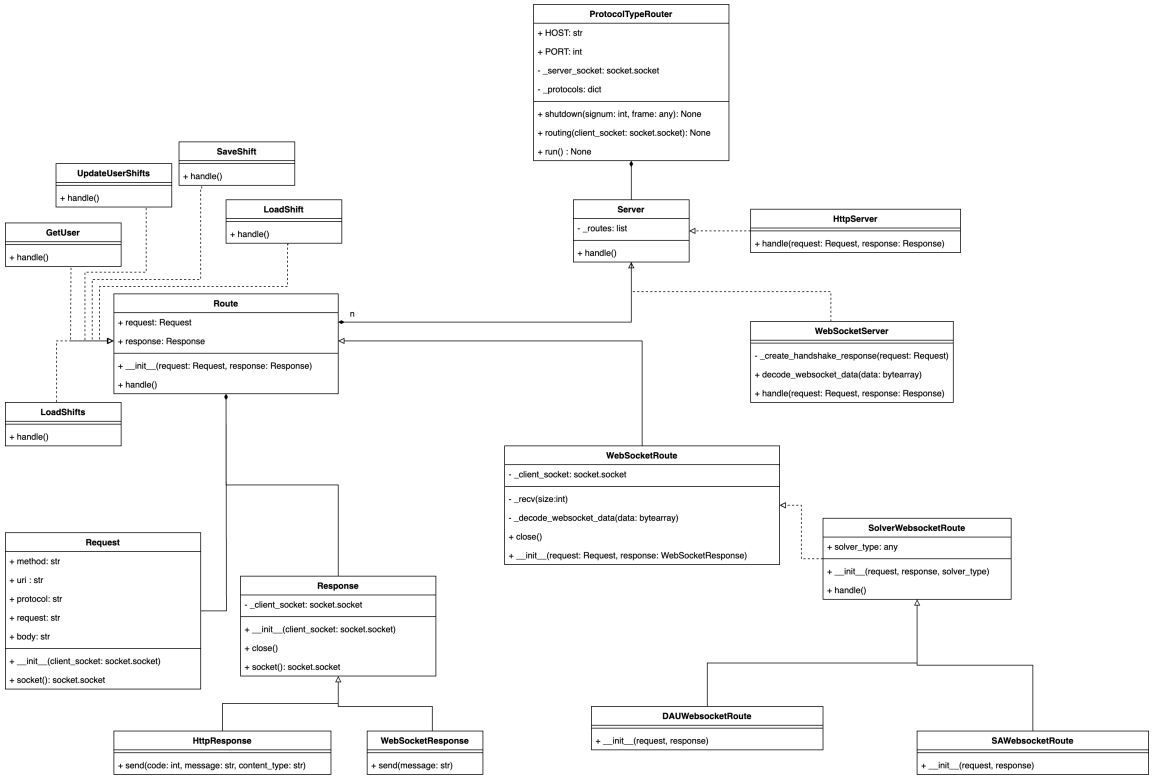

Figure 6: The UML diagram of the server

```
            "shift_array": ["0","0","1","1","0","0","1","1","1","1","1","0","0",
            "0","1","1","0","1","1","1","1","1","0","1","1","1","1","1","1","1"
            ]
        },
        {
            "name": "name2",
            "shift_array": [...]
        },
        ...
    ]
}
```

On the server side, we defined a class SolverWebsocketRoute which inherits from WebSocketRoute. The class will process the request and invoke the corresponding computation service to solve the optimization problem. We also defined two classes that inherit from SolverWebsockeRoute, which are DAUWebsocketRoute and SAWebsocketRoute. Both give the computation service to the base class. Both DAUWebsocketRoute and SAWebsocketRoute are used in two different micro-services. During the computation, the server will keep UI posted on the state of the computation. For example, when the problem is compiled. It will use a web socket to send

`{"message": "status", "status" : "compiling"}`.

The user interface can update the state of the computation to the user.

3.1.2 Data Server

The data server uses HTTP protocol to communicate with the user interface. There are 5 APIs written in the server. Each API is handled by a Route class which inherits from the class Route and overrides the function handle.

Get the user

API Endpoints: /user

Description: Through /user URI, the server accepts the post method. The client posts a username and password to the server. The server will use the function getUser defined in the DataAdapter to get the user owned by the collection User. Then, the server will return the user in JSON.

Method: POST

Example Request Data:
```
{
    "username":"admin",
    "password":"password"
}
```

Example Response Data:
```
{
    "username": "admin",
    "password": "password",
    "email": "admin@gmail.com",
    "shifts": [
        "10a67b34-dede-4a99-97ef-8f0b76927947"
    ]
}
```

Update user shifts

API Endpoints: /updateusershifts

Description: Through /updateusershifts URI, the client uses the post method to send the user's information to the server. The server will use the function updateUserShifts defined in the DataAdapter to update the shift_id array which is owned by the user and stored in the collection User.

Method: POST

Example Request Data:
```
{
        "username": "admin",
        "password": "pasword",
        "email": "admin@gmail.com",
        "shifts": [
            "10a67b34-dede-4a99-97ef-8f0b76927947",
            "0fc9f879-b236-40c4-a6f6-b5f3d5c7ddab"
        ]
}
```

Example Response Data: `no response data`

Save a shift

API Endpoints: /saveshift

Description: Through /saveshift URI, the client uses the post method to send the user's information and the shift's information to the server. The server will use the function saveShift defined in the DataAdapter to save the parameters owned by the shift and stored in the collection Shifts.

Method: POST

Example Request Data: {

```
        "shift_id": "10a67b34-dede-4a99-97ef-8f0b76927947",
        "computation_time": 10000,
        "constraints": [],
        "days": 30,
        "name_list": [ "name1", "name2", "name3", "name4", "name5",
            "name6", "name7", "name8", "name9",  "name10" ],
        "number_of_workers": 10,
        "reserved_leave": [],
        "tables": [
    [ [ 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1,
     1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 4 1, 1, 1 ], ...
    [ 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
    1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0 ] ] ],
        "type": "DAU",
    }
```

Example Response Data: `no response data`

Load a shift

API Endpoints: /loadshift

Description:

Method: POST

Example Request Data: {

```
        "shift_id":"10a67b34-dede-4a99-97ef-8f0b76927947"
    }
```

Example Response Data: {

```
        "shift_id": "10a67b34-dede-4a99-97ef-8f0b76927947",
        "computation_time": 10000,
        "constraints": [],
        "days": 30,
        "name_list": [ "name1", "name2", "name3", "name4", "name5",
            "name6", "name7", "name8", "name9",  "name10" ],
        "number_of_workers": 10,
        "reserved_leave": [],
        "tables": [
    [ [ 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1,
     1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 4 1, 1, 1 ], ...
    [ 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
    1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0 ] ] ],
        "type": "DAU",
```

```
}
```

Load shifts

API Endpoints: /loadshifts

Description: Through /loadshifts URI, the server accepts the post method. The client posts the user's information to the server to load all shifts for the user. The server will use the function loadShifts defined in the DataAdapter to get all shifts owned by the user. The function loadShifts will return a list of Shift instances. Finally, the server will convert each shift in the list to a dictionary variable and store the data in a list. Then, return the data to the client.

Method: POST

Example Request Data:

```
{
        "username": "admin",
        "password": "pasword",
        "email": "admin@gmail.com",
        "shifts": [
            "10a67b34-dede-4a99-97ef-8f0b76927947",
            "0fc9f879-b236-40c4-a6f6-b5f3d5c7ddab"
        ]
}
```

Example Response Data:

```
  "shift_list":[
{
    "shift_id": "10a67b34-dede-4a99-97ef-8f0b76927947",
    "computation_time": 10000,
    "constraints": [],
    "days": 30,
    "name_list": [ "name1", "name2", "name3", "name4", "name5",
        "name6", "name7", "name8", "name9",  "name10" ],
    "number_of_workers": 10,
    "reserved_leave": [],
    "tables": [
[ [ 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1,
 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 4 1, 1, 1 ], ...
[ 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0 ] ] ],
    "type": "DAU",
},
{
    "shift_id": "0fc9f879-b236-40c4-a6f6-b5f3d5c7ddab ",
    "computation_time": 10000,
    "constraints": [{
                    "name": "expected_number_of_workers_per_shift",
                    "parameters": {
                        "weight": "20",
                        "enwps": "10",
                        "days_off_index": []
```

```
                    },
                {
                    "name": "no_consecutive_leave",
                    "parameters": {
                        "weight": "10",
                        "days_off_index": []
                    }
                }
                }
    ],
        "days": 30,
        "name_list": [ "name1", "name2", "name3", "name4", "name5",
            "name6", "name7", "name8", "name9",  "name10" ],
        "number_of_workers": 10,
        "reserved_leave": [],
        "tables": [
    [ [ 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1,
     1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 4 1, 1, 1 ], ...
    [ 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1,
    1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0 ] ] ],
        "type": "SA",
    }
      ]
    }
```


Download a shift

API Endpoints: /download/[shift_id]

Description: Through /download/ URI, the server accepts the get method. The client request the server with the shift's information to the server to download the shift. The server will use the function loadShift defined in the DataAdapter to get the shift owned by the user. The function loadShift will return a Shift instance. Finally, the server will convert the shift to a csv file and return the data to the client.

Method: GET

Example Request Data: No need to provide the data

Example Response Data: A CSV file

3.2 Microservices

Request for available services

API Endpoints: /gethealthcheck

Description: This API is opened to clients to check the availability of the service. The API will return the current available services and service's url. This enables clients to access micro services through those urls.

Method: GET

Example Request Data: No need to provide the data

Example Response Data:

```
[
    {
        "service": "SA",
        "status": "ON",
        "url": "ws://localhost:8890/sa",
        "healthcheck": "http://localhost:8890/healthcheck"
    },
    {
        "service": "DAU",
        "status": "ON",
        "url": "ws://localhost:8889/dau",
        "healthcheck": "http://localhost:8889/healthcheck"
    }
]
```

Micro-service's health check

API Endpoints: /healthcheck

Description: The API will return the current status of the service. This API is available to each micro-service. Once the client request a health check, the main API gateway will send the request to each micro-service to check if the service is healthy or not.

Method: GET

Example Request Data: No need to provide the data

Example Response Data:

```
{
    "status": "healthy"
}
```

Microservices registration

API Endpoints: /registry/add

Description: The API is opened to the microservices. Once the microservice is opened, it needs to use this API to register in the server.

Method: POST

Example Request Data:

```
{
    "healthcheck": "http://localhost:8889/healthcheck",
    "service": "DAU",
    'url' : "ws://localhost:8889/dau"
}
```

Example Response Data:

```
{
    "status": "success"
}
```

Microservices delete

API Endpoints: /registry/delete

Description: The API is opened to the microservices. Once the microservice is closed, it needs to use this API to drop the microservice in the server.

Method: POST

Example Request Data:

```
{
    "service": "DAU",
}
```

Example Response Data:

```
{
    "status": "success"
}
```

3.3 Database

We design two models, User and Shift, and class DataAccess to manipulate the data in the Application. Class User corresponds to a document in the collection Users and class Shift corresponds to a document in the collection Shifts. Please refer to Fig. 8. for the UML of the data adapter class. Class DataAccess is an interface that will be implemented by both DataAdapter and RemoteDataAdapter.

3.3.1 DataAdapter

The DataAdapter will establish a connection to the MongoDB database. We can load and save the document into the collections. The following describes the function defined in the class DataAdapter.

getUser: We can get the user's information by providing the username and password to the function. The function will use username and password as a key to search and load the user's data from Redis database. If the function successfully obtain the data, it will generate an instance of class User and return the object; otherwise, return None.

updateUserShifts: We can update the user's information by providing the document in the collection Users to the function. The function will search and update the document in the collection of Users with the given data.

saveShift: We can update the shift's information by providing the document in the collections Users and Shifts to the function. The function will search and update the document in the collection Shifts with the given data.

loadShift: We can get the shift's information by providing the shift_id to the function. The function will search the document in the collection Shifts with the given data. If the function successfully gets a document from the collection, it will generate an instance of class Shift and return the object.

loadShifts: We can get the several shifts' information by providing the document in the collection Users to the function. The function will search the document in the collection Shifts with the given data. The function will get several documents from the collection and generate a list to store the documents and return the object.

3.4 Solvers(Algorithms)

The shift scheduling problem is modeled by the quadratic unconstrained binary optimization (QUBO) problem. 0 represents a day off and 1 represents a working day. Once the user adds shift requirements, they implicitly formulate their QUBO problem. The QUBO problem is compiled in the user's computer and solved by either the digital annealer or the simulated annealing algorithm on user's device. After the device finishes computation, the solution is then post-processed to generate the final schedule. Digital annealer is a computation service provided by Fujitsu Ltd. Digital annealer is a quantum-inspired device that is good at solving optimization problems.

In this prototype, the application offers limited shift requirements for users to choose from. The available shift requirements are listed and explained in detail below. The source code of the mathematic constraints is written in this file constraints.py.

Each constraint in the project represents a class. The constraint inherits a base class which is defined as follows:

```
class ConstraintFunction(object):

    def __init__(self, X:Array, **kwargs):
        self._X = X
        self._weight = float(kwargs['weight'])

    def hamiltonian(self) -> Model:
        pass

    def weighted_hamiltonian(self) -> Model:
        return self._weight * self.hamiltonian()

    def evaluate(self, table) -> dict:
        pass
```

In the project, there are two solvers available. One is a digital annealer, and the other is a simulated annealing solver. The UML diagram is shown in Fig. 7.

3.4.1 The Expected Number of Working Days

This requirement could be added to define the number of working days in the shift range. Workers in the shift are expected to have equal working days. The mathematical formulation for N workers and D days shift is as follow:

$$H = \sum_{i}^{N} \left(\sum_{j}^{D} x_{ij} - \alpha \right)^2 ,$$

where α is the expected number of working days, and H is the hamiltonian.

3.4.2 The Expected Number of Workers per Shift

The following constraint defines the expected number of workers in each shift. This constraint is common and used to balance the workforce and the workload each day. The constraint is modeled by summing up the variables on each shift and minus the expected

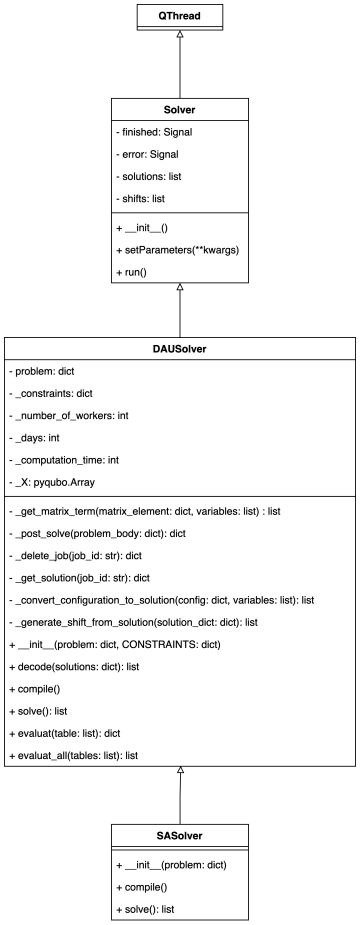

Figure 7: UML of solvers

number of workers and, finally, double themselves and make them quadratic. The mathematical formulation is as follows:(The situation is the same as the one used above)

$$H = \sum_{j}^{D} \left(\sum_{i}^{N} -\beta \right)^{2},$$

where β is the expected number of workers each shift.

3.4.3 Successive Shift Pair

This constraint is placed to make each worker has at least two consecutive shift. Here's the situation we don't want. The concept is shown below in regular expression form $((0|1)^*0|0^*)1(0^*|0(0|1)^*)$

To be simple, the mathematical formulation separates the boundary situations, $(0|1)^*01$ and $10(0|1)^*$, and the general situation, $(0|1)^*010(0|1)^*$.

3.4.4 Consecutive 2 days Leave

This is a soft constraint for days off preference. When this constraint is employed, the algorithm would try to arrange the days off together.

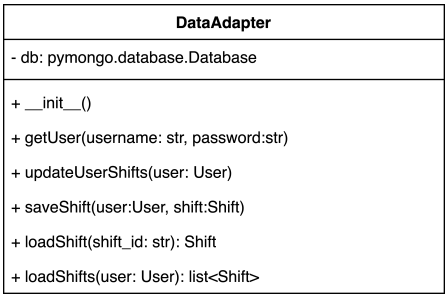

Figure 8: Data Adapter UML

The mathematical formulation is as follows:

$$H = \sum_{i}^{N} \sum_{j}^{D-1} (1 - x_{i,j} * x_{i,j+1})$$

.

3.4.5 No Consecutive 2 Days Leave

This constraint is an opposite version of the above constraint. The mathematical formulation is as follows:

$$H = \sum_{i}^{N} \sum_{j}^{D-1} [(1 - x_{i,j}) * (1 - x_{i,j+1})]$$

.

3.4.6 The Maximum Consecutive Shifts

The constraint set up a limit on the maximum number of consecutive shifts.

This constraint could be modeled by using the supplementary variables. However, this would result in an increase in the problem scale. This technique would probably double or triple the number of original variables. Moreover, the solver, Fujitsu's digital annealer, doesn't know the meaning of each variable after compiling the solution into the accepted data; thus, after the annealing process, it would be possible to find a better solution.

The digital annealer computation service allows users to post the inequality.

The inequalities would be an array and append the following inequality.

$$\sum_{j}^{j+\gamma+1} x_{ij} \leq \gamma, \ \forall i \in [1, N], \ j \in [1, D - \gamma - 1]$$

where γ is the maximum consecutive shifts

3.4.7 Minimum N-days leave within 7 days

The constraint is for employees' days off welfare. The employees working in graveyard shifts need to have days off in a range.

4 Runnable Prototype with GUI and Video

https://youtu.be/q6wpCm87sV8