Builder Maintenance Guide

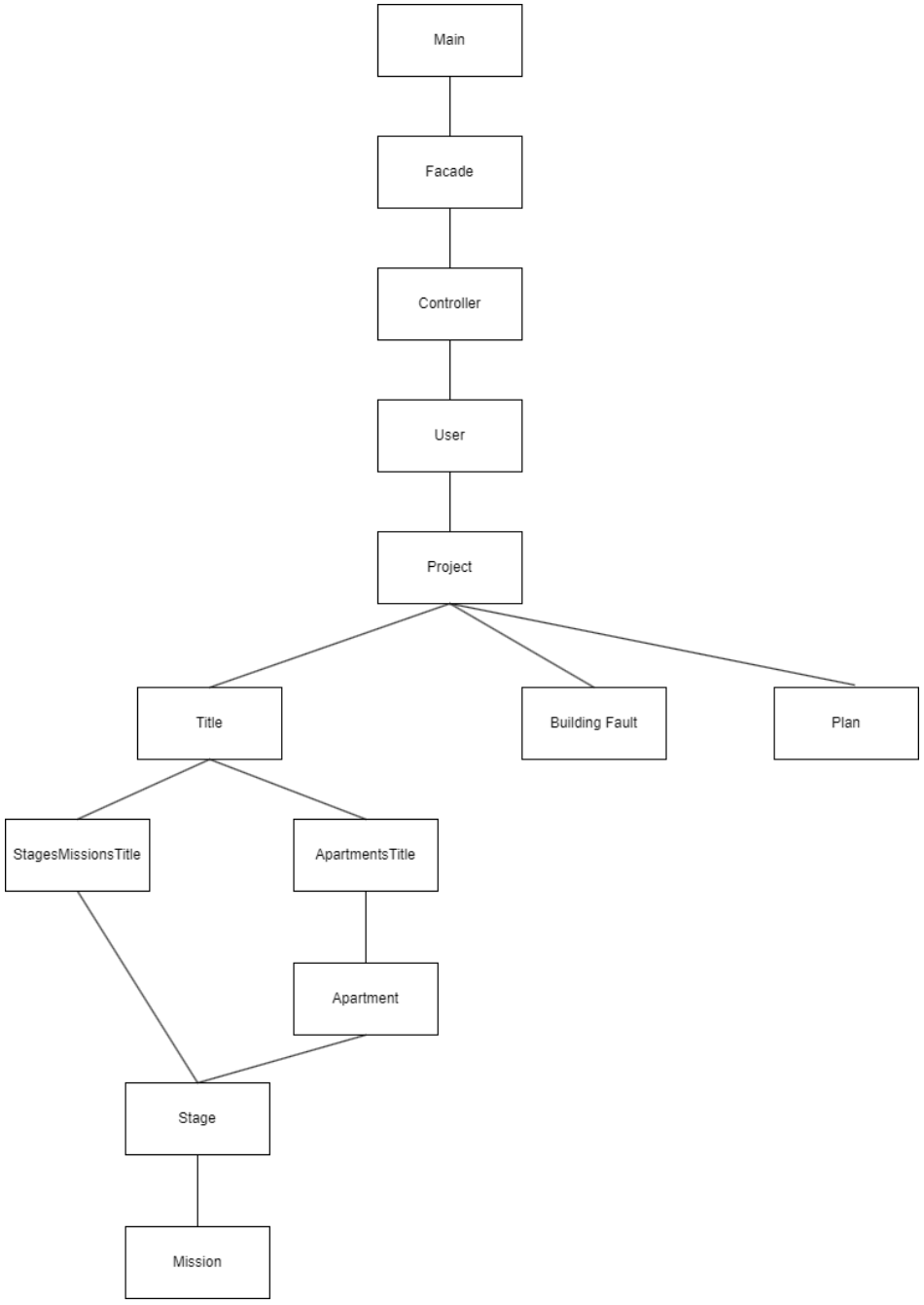
Table of Contents

Backend Maintenance

The backend for the application is written in python (version 3.7 or higher).
The code is running on an AWS server on an EC2 machine (request account details from project owner), and acting as a REST API for the mobile application
Backend repository link - https://github.com/odedgal67/BuilderBackend

Backend structure and usage

The backend has hierarchy to its objects and classes and the methods percolate down the hierarchy. Here is a basic class diagram demonstrating the hierarchy (more information can be found in ADD):



Main

The component is responsible for the REST API. It is implemented with the Flask library. Each method has its own route added to the server's URL, and is called by sending an HTTP request with a JSON payload containing all the required data (required method arguments).
Let's review an example method from main.py:

```python
@app.route("/set_urgency", methods=["POST"])
@require_login
def handle_request_set_urgency():
    print("\n\nSet Urgency request received")

    # Parse JSON payload from the request
    data = request.get_json()
    print(f"data : {data}")

    # Call the facade method
    try:
        facade.set_urgency(
            data["project_id"],
            data["building_fault_id"],
            data["new_urgency"],
            data["username"],
        )
        return jsonify({"result": "success"})
    except KnownServerException as e:
        print(f"[set_urgency] : raised exception {str(e)}")
        return jsonify({"error": str(e)}), ERROR_CODE
```

Method name convention in "handle_request_<method name>".

The method has 2 decorators:
app.route - A Flask decorator to define the method's route and type (GET, POST, etc,,,)
require_login - A decorator that validates that the user calling this method is logged in.

Getting payload data - This can be done by using python's library "request" (request.get_json) as seen in the example above.

After we have successfully retrieved the data from the HTTP request we can call the relevant facade method. Arguments are retrieved by using data["<argument name>"] or data.get(<argument name>, <default value if not exists>). Please note that the facade call is contained in a try-except to catch exceptions thrown by the facade and not crash the server. The facade will throw exceptions even with correct usage, such as permissions errors for example.

After we received a result from the facade we can send it back to the user calling the server by returning a json with the result's data or just a "success" message if the facade has no return value (as seen in this example).

Error Codes:
When returning to the user, we can return a tuple, where the second part of the tuple is an error code.
200 - OK
400 - Expected error from backend
401 - Refresh user login token
404 - File not found
500 - Unexpected backend exception

Facade

The component is a facade to simplify and hide the implementation from the API. It holds the Controller component and redirects requests to it. Return values from the facade to the server will never be “real” backend objects but DTO containing restricted data, for example when registering a user we would like to return the registered user to the server, but we wouldn’t like to share the password field, so we create a DTO without the password field to return to the user.

Here is an example:

```
def register(self, username: str, password: str, name: str):
    # return user
    return self.controller.register(username, password, name).to_json()
```

Controller

The component holds all the users in the system. Each request from the server is accompanied with the user’s username. With that username we can find the relevant user object and pass the responsibility down the hierarchy. Note that the DTO conversion is also a responsibility of this component, so the facade will not have to do any logic and can remain simple and abstract.

File System Controller

All of the files uploaded by the users are saved locally on the server machine, this was done to allow for easier querying, and data size scalability. The server allows you to query any file by accessing the url of the server concatenated with the file’s name to allow access from any platform. The controller class has 1 main method that adds a file to the file system:

```python
def add_file(self, data, file_type: str):
    new_file_name = str(uuid.uuid1()) + "." + file_type
    try:
        blob = BytesIO(data)
        if not os.path.isdir(self.directory):
            os.mkdir(self.directory)
        file_path = os.path.join(self.directory, new_file_name)
        with open(file_path, 'wb') as f:
            f.write(blob.getvalue())
        return new_file_name
    except Exception as e:
        traceback.print_exc()
        raise Exception(f"Error processing the file: {new_file_name}\n{str(e)}")
```

Data is a bytes stream and file type is the type of the data in string. In order to allow more file types simply add methods similar to these:

```python
def add_image(self, data, original_file_name: str):
    allowed_image_type = ['jpeg', 'png', 'bmp']
    return self.__add_file_of_type(data, original_file_name, allowed_image_type)

4 usages
def add_doc(self, data, original_file_name: str):
    allowed_docs_types = ['pdf']
    return self.__add_file_of_type(data, original_file_name, allowed_docs_types)
```

User

The User object holds the username, name, hashed password (we immediately hash and store the hashed password for privacy), logged in, projects and projects permissions.
The project field is a dictionary where the key is the UUID (unique ID) of the project and the value is the project object. Only projects that are assigned to a user can be found in the projects field, which restrices the user's access to projects he is not assigned to.
The project permission field is also a dictionary. The key is the project UUID and the value is an Abstract Permission. Every action done in the user will validate the user's permission using this dictionary.
Let's review an example for the "add stage":

```python
def add_stage(self, project_id: UUID, title_id: int, stage_name: str, apartment_number: int = None):
    project: Project = self.get_project(project_id)
    project_permission: AbstractPermission = self.get_project_permission(project_id)
    return project_permission.add_stage(project, title_id, stage_name, apartment_number)
```

As we can see we first get the project object using the UUID. Then we get the project permission object using the UUID.
We pass the responsibility to the project permission object, and pass the project object as an argument to the function.

Project Permission

The module has 4 classes. AbstractPermission which is an abstract class (interface) and 3 classes implementing it: WorkManagerPermission, ProjectManagerPermission, ContractorPermission.

By using this design we can enforce the permissions for each user and its project specific permission. Notice that every action a WorkManager can do, can also be done by ProjectManager and Contractor and every action a ProjectManager can do, can be done by the Contractor. This is possible because the classes inherit from each other using this logic: WorkManager < ProjectManager < Contractor

When we want to allow an action for a specific permission we will pass the responsibility to the project object received as an argument. When we want to deny permission we can simply raise a PermissionError.

Let's review an example:

We would like to restrict "remove mission" action for WorkManager but allow it to ProjectManager.

First we define the method in the interface (abstract class AbstractPermission):

```
def remove_mission(
    self, project, title_id, stage_id, mission_id, apartment_number: int = None
):
    pass
```

Then we implement the permission error under WorkManager:

```
def remove_mission(
    self, project: Project, title_id, stage_id, mission_id, apartment_number: int = None
):
    raise UserPermissionError
```

Then we implement the functionality under ProjectManager:

```
def remove_mission(
    self, project: Project, title_id, stage_id, mission_id, apartment_number: int = None
):
    return project.remove_mission(title_id, stage_id, mission_id, apartment_number)
```

Title

There are two types of titles: TitleMissionStages and TitleApartments. TitleMissionStages is a title that aggregates missions by stages, TitleApartments aggregates missions by stages and apartments. Each title class implements managing stages and missions differently.

Project

This component holds titles, plans and building faults.
It is responsible for forwarding general requests to the correct object and managing its own fields. Each project has 4 default titles, and must have building faults and plans (which is why they are fields).

Database

The backend implements a JSON database for all the objects using MongoDB.

Each persistence object has a method “to_json” which turns the object into a dict (which can be easily translated into JSON string). This method is used when we want to persist a full object to the database.

Each persistence object has a static method called “load_<class name>” which loads and creates a full object from the database using a JSON string.

In order to load all the objects when the server is loading for the first time, we use the “load_database” static method in the Controller module. This will call all the relevant loading methods and build all the objects stored in the database, according to the JSON string read from the database.

To insert and update the database we have 2 main ways:

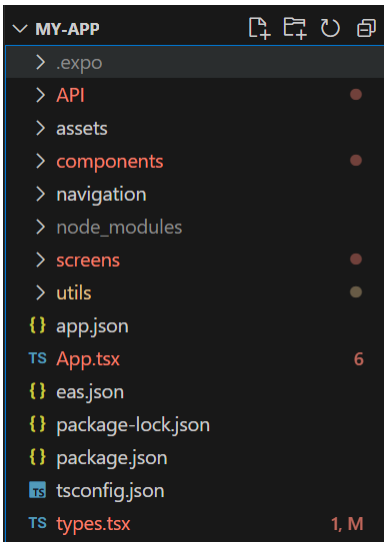
1. persist_user method which updates the database in changes in the user object.
2. update_project_methods decorator which decorates every project method that changes the state of the project with an update to the database.

Transactions in our application are relatively simple (1 field updates), therefore transaction management rollbacks are not addressed.

The database can be set locally or remotely. It is currently set remotely on a MongoDB Atlas free server (request user’s information from project owner).

Frontend Maintenance

File hierarchy:



- App.tx - holds the root component.
- API - directory that has all implementation regarding api interaction.
- Components - has all sub components child nodes in screens.
- screens - has all screen components registered in the application navigator.
- utils - utility functions.
- types.tsx - has hall type definitions used in the projects.

Important context hooks

ProjectContext and UserContext: almost every action requires the user to identify which user he is and which project is modifying, to avoid prop drilling these parameters to each child node, we use these contexts. Usage example:

Important reused components

- StagesTable: this component has a misleading title, designed initially to hold all the stages under some title, this component also used for missions list, apartment stages

```
onPress: () => {
  handleLoadFromExcel(getProject().id, getUser().id).then(() =>
    setModalVisible(false)
  )
const { getProject } = useContext(ProjectContext);
const { getUser } = useContext(UserContext);
```

  list and building faults list. Props:
  - stages- any object to be represented in the list, and has the following attributes: name, status, id and title.
  - Buttonhandler: function to be called when a listen item is clicked.

    - addStageHandler: function to be called when the user is trying to add a new listed item.
    - allow_change_status: bool deciding if this table allows managing statuses.
    - onDelete: function to be called when the user wants to delete a certain listed item.
    - onEditName: function to be called when the user wants to rename a certain listed item.
    - onChangeStatus: if allow_change_status is true, it is a function to be called when the user wants to change the status of a certain listed item.
- GetTextModal: this component is a modal that can be used to get some text input from the user.
- StatusRectangle: clickable status to be used for managing or presenting status of something.
- Background: wraps most page nodes to apply the same background.

API calls

In our application we have a restful API, on the client side we use the library Axios. To simplify our implementation process we created 2 main utility classes - Responses and PostWrappers:

- PostWrappers: this is a generic class with one implemented method: send_request which is responsible for sending a data request to a specific url on the server and returning a parsed object response, using abstract method get_response_class. This class is also responsible for refreshing id tokens when the timer runs out, using the latest authentication used by the user. This class is implemented per return type from the server, using the Response class I will cover next. Implementation example:

```
export class PostWrapperMission extends PostWrapper<Mission> {
  get_response_class(data: any): Response<Mission> {
    return new MissionResponse(data);
  }
}
```

- Response: after implementing any new PostWrapper you need to implement the relevant TypeResponse to parse the result, example:

```
export class MissionResponse extends Response<Mission> {
  get_result(): Mission {
    let output: Mission = {
      id: this.result.id,
      name: this.result.name,
      green_building: this.result.green_building,
      completion_date: this.result.completion_date,
      completing_user: this.result.completing_user,
      comment: this.result.comment,
      status: status_mapping[this.result.status],
      proof_link: this.result.proof,
      document_link: this.result.tekken,
      plan_link: this.result.plan_link,
    };
    return output;
  }
}
```

With these 2 utility classes we can now easily implement new API calls by instantiating the relevant PostWrapper<T> (depending on the result value we expect from the server) and calling send_request with the proper url and parameters:

```
add_mission(
  project_id: string,
  stage_id: string,
  title: Title,
  mission_name: string,
  username: string,
  apartment_number?: number
): Promise<Mission> {
  return new PostWrapperMission().send_request(this.get_url("add_mission"), {
    project_id: project_id,
    apartment_number: apartment_number,
    stage_id: stage_id,
    title_id: title,
    mission_name: mission_name,
    username: username,
  });
}
```

Deploying the System

Backend

The backend repository uses poetry to manage all of its packages. In order to run the server first make sure you have poetry installed using ‘pip list’, to install simply run ‘pip install poetry’. After installing poetry, to install all packages simply run ‘poetry lock’ and then ‘poetry install’ and it should proceed to installing all relevant packages (don’t worry poetry deploys a local virtual environment).
Before running the server, check-out Config.py to see you will run your desired configuration. Last but not least to run the server simply run ‘python main.py’, notice that it will ask you for permissions since it runs on port 80, and on unix based machines ‘sudo’ might be needed.

Database

In our system we use MongoDB through python’s package pymongo. In order to use the system with the database simply update your hosting server credentials on the config file in the server repository.
To install and run the database server locally, simply follow the official instructions here: https://www.mongodb.com/docs/manual/tutorial/install-mongodb-on-windows/

Frontend

After cloning the repository, navigate to the directory my-app where the package.json file sits. Then, simply run ‘npm install’ to install all relevant dependencies. To set your desired server url you want to work with (local or remote) set the url in the api_bridge.tsx file.
Now the application is ready to be bundled by your preferred bundler, or alternatively, use expo to run the app. To use expo you’ll need to install the expo app on your mobile device from the google play store and then in the my-app directory run ‘npx expo start’. This will present a barcode you can scan with the expo app to quickly bundle and deploy the app on your phone. NOTICE: expo raises a server available on your local network! so make sure both your mobile device and the hosting computer are on the same network!