Summer Fellowship Report

on

Osdag on Web

Submitted by

Aaranyak Ghosh

under the guidance of

Prof. Siddhartha Ghosh
Department of Civil Engineering
IIT Bombay

and under the mentorship of

Mr. Nagesh Karmali
Project Sofware Engineer, IIT Bombay

Mr. Danish Ansari
Project Manager, IIT Bombay

November 12, 2024

Acknowledgment

Firstly, I would like to acknowledge my sincere gratitude towards those who have provided this opportunity, and to those who have enabled, and supported me to undertake this journey which has been a crucial learning experience that has left me with certain skills that I will require for my work in the future. Specifically, I would like to express gratitude to Mr. Danish Ansari, the project manager, and Mr. Nagesh Karmali, the team leader, who have provided the guidance and support required for me to understand how such a project would be organised, planned and executed. I would also like to thank the developers who I have worked with, with whom I was able to collaborate seamlessly on an otherwise complex and many-layered project.

I would also like to extend my gratitude to my school, for providing the time to take part in this intensive development project, and to the organisation FOSSEE, for creating the opportunities for my to take part in this project, and for incubating the power of open-source software in this world.

Contents

Chapter 1

Introduction – Why Osdag on Web

1.1 What is Osdag

Osdag is an open source software specifically created for steel design. It has been designed as a desktop application with an interactive user interface. The interface is used to design a wide variety of connections, members and systems and has the ability to display the model in a 3D view. The software can also export to CAD models which can be viewed on other software. This software is primarily developed using Python, and uses the library PythonOCC, to render CAD models of the designs.

Further details on Osdag can be found on `https://osdag.fossee.in`

1.2 Why build Osdag on Web

The original implementation of the software – the desktop application – had various issues, but of them, the most pressing concern was compatibility. The software was released as a set of Python scripts, which were then dependent on a very specific environment, and had a large set of dependencies which would conflict with another. On top of that, some of these dependencies had graphics issues with some systems, and the installer would be misinterpreted as a virus by others. As a result, the software could only be run on very specific systems. This was a major concern, as the software was intended for educational use, and should therefore be easy to install.

Considering this, it was decided that the best solution would be to skip the installation process completely and, and allow users to use the application from a browser. This can provide students easier access to the software, since a lengthy install process could deter users who want to try the application to see it it would suit their needs. It also opens up a new range of benefits, such as minimising the load on the user's computer, and ability to load projects from different computers.

1.3 Osdag on Web – A Short Summary

To minimise load on the server, the original plan was to create a web app that can run the required calculations on the browser itself. But since this would mean re-writing the entire Osdag application in JavaScript, we decided to start to a simpler method.

Since the Osdag software is written in Python, we could write our server-side software in Python, so that we could directly use the software to run calculations. We used a library named Django, which makes it easier to write server-side code in Python. Then, this processed data could be sent to the frontend to be displayed to the user. This framework would have a few components:

- The frontend, which is written in JavaScript
- The backend, built using the Django framework
- The Osdag software, running on the server
- And a bridge layer between the web backend, and the application.

The frontend uses THREE.js for rendering the CAD models on the web browser. The frontend sends requests, which are then parsed by the backend. The backend uses the bridge to communicate with the Osdag software, which then returns the data back to the backend. This backend will then generate a response, which can be understood by the frontend. The frontend is developed using React.

1.4 Osdag Workflow

The web version of Osdag needs to replicate the desktop version as closely as possible, so the **API** needs to be developed while keeping the workflow in mind. The Osdag workflow contains a series of steps. Some of these steps will be executed internally by the **Javascript interface**, while others will need to be sent to the **backend** for calculation. These are the steps of the Osdag workflow:

1. The user opens Osdag, and selects the module.
2. The user sets the module's properties and settings in the input values editor.
3. The software generates the module - which involves three steps.
   (a) Generate a CAD model, and display it in the 3D view.
   (b) Generate module information, and display it in the output values view.
   (c) Print logs in the log view.
4. The user now views the CAD model and output information, has two options:
   - Change input properties and regenerate module.
   - Generate design report.

The interface and the display is entirely handled by the **Javascript frontend**. Whenever any calculations need to be run (eg. generating design report), the frontend will send the information to the backend, which will use the bridge to run the calculations. The **Web Api** therefore needs to be able to respond to the following requests:

1. Get module output from input
2. Generate CAD model from input
3. Generate design report

The Osdag **API** needs to parse each of these requests separately, and not store any data in the backend - it needs to be stateless. Therefore, it needs to be able to generate the module from the input values every time. The module generation is handled by the **bridge**, which communicates with the **Osdag application**.

1.5 Objective and Scope of the Report

Therefore, the objective of the project is to replicate the features of the Osdag Application in a browser window. This involve would replicating the same user interface in javascript and creating an interface for the frontend to communicate with an instance of the Osdag Application running on the back-end. To simplify this process, the goals need to be limited to just developing the functionality for a single module.

In the next chapter, I describe my development process for the bridge, which is an interface which allows other parts of the software to communicate with an instance of the Osdag Application in a module-independent manner. In the third chapter, I explain my role in developing a Web API, which allows a frontend running on the client side to communicate with the back-end of the software, which runs on the server. In the fourth chapter, I describe some changes I made to the part of the GUI which loads the CAD model, which allows multiple users to use the software at a time.

Chapter 2

API/Osdag Bridge

This chapter primarily concerns the backend of the application, which will be running on a server. The backend is divided into three main parts: **a)** the web API **b)** the Osdag application, and **c)** the API/Osdag bridge.

The **web API** is located under the folder `osdag_web/osdag`. This parses incoming requests to the server, and then communicates with the **bridge**. this is located in the rather confusingly named `osdag_web/osdag_api`. The **bridge** can access the **Osdag application**, and use it to run the required calculations for various modules.

2.1 Bridge Layer Structure

When the **API** needs to run calculations for a specific module, it can request the implementation for that module. Each module implementation needs to have a set of functions, which the **API** can use to communicate with the **Osdag application**. A template for each of these implementations, written in python, is given below:

```
class ModuleApiType(_Protocol):
    def validate_input(self, input_values: Dict[str, Any]) -> None:
        """Validate type for all values in design dict. Raise error when
        ↪  invalid"""
        pass
    def get_required_keys(self) -> List[str]:
        pass
    def create_module(self) -> Any:
        """Create an instance of the module design class and set it up for use"""
        pass
    def create_from_input(self, input_values: Dict[str, Any]) -> Any:
        """Create an instance of the module design class from input values."""
        pass
    def generate_output(self, input_values: Dict[str, Any]) -> Dict[str, Any]:
        """Generate, format and return the input values from the given output
        ↪  values."""
        pass
    def create_cad_model(self, input_values: Dict[str, Any], section: str,
    ↪  session: str) -> str:
        """Generate the CAD model from input values as a BREP file. Return file
        ↪  path."""
```

The **bridge** includes a function, to return the implementation of the module from the given name. This way, the **API** can be written in a *module-independent* manner, while the different implementations for the modules can be handled by the bridge. The **API** can use the bridge like this:

```python
"""Some example code to illustrate how the API would communicate with the
↪ bridge"""
import osdag_api # Import bridge library, mistakenly named osdag_api

def handle_some_request(module_name: str, input_values: Dict[str, Any]) -> None:
    """
        A function to illustrate how the bridge is used
         -> module_name is a string that specifies the module being used,
            such as Plate Connection
         -> input_values is a dictionary of properties to create the module
    """
    module = osdag_api.get_module_api(module_name) # Returns the module
    ↪ implementation
    log_string = "" # Log file
    try: # In case the input values are formatted incorrectly, this will return
    ↪ an error
        module.validate_input(input_values) # Checks if input values are valid
    except: OsdagApiException as e: # In case they were formatted incorrectly
        print(e) # Display the error in the console
        log_string += e + "\n" # Add the error to the log file
    except: Exception as e: # If something else went wrong
        print("Internal Error:", e) # Display message in console
        log_string += "Internal Error: " + e + "\n" # Add the error to the logs
    else: # Nothing went wrong
        output_values, logs = module.generate_output(input_values) # Generate the
        ↪ output for this module
        print(output_values) # Display the output values in the console
    finally: # After everything is done
        print(logs) # Print the logs to the console

handle_some_request("Fin Plate Connection", input_values) # Run that function
```

The above program shows a hypothetical example of how the **bridge** is used, in this case to generate output values. In the following sections, the implementation of each of these functions is explained.

2.2 Module Creation

Currently, only one module has been properly implemented - the *FinPlateConnection*. The bridge for this implementation imports the **osdag library**, and creates an instance of the module, so that it can use that object to do the calculations for it. The following code creates the module:

```python
import typing # Type names
from typing import Dict, Any, List # Use these type names
# The following import will log a lot of data, and we don't want to print that,
↪ so the stdout will be temporarily redirected
```

```python
old_stdout = sys.stdout # Backup old stdout
sys.stdout = open(os.devnull, "w") # Redirect stdout
from design_type.connection.fin_plate_connection import FinPlateConnection #
↪  Import the module class from the osdag application
sys.stdout = old_stdout # Reset stdout

def create_module() -> FinPlateConnection:
    """Create an instance of the fin plate connection module design class and set
    ↪  it up for use"""
    module = FinPlateConnection() # Create an instance of the FinPlateConnection
    module.set_osdaglogger(None) # We won't need the logger for now.
    return module
```

To set the input values of the module, we can call the set_input_values function on the module. The following function creates a module, and sets its input values:

```python
def create_from_input(input_values: Dict[str, Any]) -> FinPlateConnection:
    """Create an instance of the fin plate connection module design class from
    ↪  input values."""
    validate_input(input_values) # Checks if input values is correctly formatted,
    ↪  throws error if they aren't
    module = create_module() # Create module instance. (shown above)
    module.set_input_values(input_values) # Set the input values on the module
    ↪  instance.
    return module
```

The function first calls the the validate_input function, which checks if all the values are formatted correctly. The implementation details of this function are outlined in the following section.

2.3 Input Data Validation

For the input values to be sent to the **Osdag application** to use for calculating the details of the module, the text needs to be formatted in a specific way. Since this data will be coming from API requests, we have no assurance that it is correctly formatted, and to prevent any errors on the server-side, we need to check this data and not send it to the **Osdag application**.

At first, we need to check if the dictionary contains all the required values. For this we can just check to see if the dictionary contains the given keys. The function for doing that is given below:

```python
def contains_keys(data: dict, keys: List[str]) -> Optional[Tuple[str]]:
    """Check whether dictionary contains all given keys."""
    missing = []
    for key in keys:
        if key not in data.keys():
            missing.append(key)
    if missing != []:
        return tuple(missing)
```

But some values in these might be incorrectly formatted. This could cause conversion errors in the **Osdag application**.

Therefore, we need to have a check on the data to see if the values are of correct types. By default, Python is not a typed language, so we need to write these checks ourselves. First, we need to check the type of single variables, such as whether something is a string, or an integer, or a float. Some variables are sent as string representations of numbers. So we need to check if the strings can be converted into numbers. The following code, in `osdag_api/utils.py` contains the functions to validate types of strings, and what they can be converted to.

```python
def int_able(value: str) -> bool:
    """Check if str can be converted to int."""
    try:
        int(value)
    except:
        return False
    return True

def float_able(value: str) -> bool:
    """Check if str can be converted to float."""
    try:
        float(value)
    except:
        return False
    return True

def is_yes_or_no(value: Any) -> bool:
    """Checks if value is 'Yes' or 'No'."""
    if not isinstance(value, str):
        return False
    if not (value == "Yes" or value == "No"):
        return False
    return True
```

We also need to validate arrays of elements, to check what they contain. This is a complex process, so there are a few functions for this. Firstly, a function to check if all the elements in an array are of a specific type, and secondly, a function that checks each element in an array and validates it.

```python
def contains_keys(data: dict, keys: List[str]) -> Optional[Tuple[str]]:
    """Check whether dictionary contains all given keys."""
    missing = []
    for key in keys:
        if key not in data.keys():
            missing.append(key)
    if missing != []:
        return tuple(missing)

def custom_list_validation(iterable: Iterable, validation: Callable[[Any], bool])
↪   -> bool:
    """Validate all items in the list using a custom function."""
    for item in iterable:
        if not validation(item):
            print(item)
            return False
    return True
```

The second function can be used to check, for example, if an array contains only strings that can be converted into ints. To do this, we can pass our test function into the second argument, and it will be run for each element of the array.

There are five types of elements in the input values, two of which are arrays. These are:

1. `string elements`
2. `string representation of integers`
3. `string representation of floats`
4. `arrays that contain string representation of integers`
5. `arrays that contain string representation of floats`

There are functions to test for each of these. If they fail, they raise an `InvalidInputTypeError`. The other error that can be raised is a `MissingKeyError`, which is raised when a key is missing from the input values. These are the functions to validate each of these types, in the file `osdag_api/validation_utils.py`.

```
from osdag_api.utils import contains_keys, custom_list_validation, float_able,
↪  int_able, is_yes_or_no, validate_list_type
from osdag_api.errors import MissingKeyError, InvalidInputTypeError
import typing
from typing import Dict, Any, List

def validate_string(key: str, value: Any) -> None:
    """Check if value is a string. If not, raise error."""
    if not isinstance(value, str): # Check if value is a string.
            raise InvalidInputTypeError(key, "str") # If not, raise error.

def validate_num(key: str, value: Any, is_float: bool) -> None:
    """Check if value is a string that can be converted to a number. If not,
    ↪  raise error."""
    if is_float: # If value can be a float.
        checker = float_able # Function for conversion checking
        type = "float" # Type in error
    else:
        checker = int_able
        type = "int"
    if (not isinstance(value, str) # Check if value is a string.
            or not checker(value)): # Check if value can be converted to
            ↪  int/float.
        raise InvalidInputTypeError(key, "str where str can be converted to " +
        ↪  type) # If any of these conditions fail, raise error.

def validate_arr(key: str, value: Any, is_float: bool) -> None:
    """Check if value is a list where all items can be converted to numbers."""
    if is_float: # If value can be a float.
        checker = float_able # Function for conversion checking
        type = "float" # Type in error
    else:
        checker = int_able
        type = "int"
    if (not isinstance(value, list) # Check if value is a list.
            or not validate_list_type(value, str) # Check if all items in value
            ↪  are str.
            or not custom_list_validation(value, checker)): # Check if all items
            ↪  in value can be converted to float.
        raise InvalidInputTypeError(key, "non empty List[str] where all items can
        ↪  be converted to float") # If any of these conditions fail, raise
        ↪  error.
```

We need to check these for each of the keys. The function shown below first checks if all the required keys are there, and if they are not, throws a `MissingKeyError`. After that, it goes through each of the values, and runs the validation tests on them. If any of them fail, it throws a `InvalidInputTypeError`. The following code shows the validation function implementation for the *FinPlateConnection* module:

```
from osdag_api.utils import contains_keys, custom_list_validation, float_able,
↪  int_able, is_yes_or_no, validate_list_type
from osdag_api.errors import MissingKeyError, InvalidInputTypeError
from osdag_api.validation_utils import validate_arr, validate_num,
↪  validate_string

def validate_input_new(input_values: Dict[str, Any]) -> None:
    """Validate type for all values in design dict. Raise error when invalid"""

    # Check if all required keys exist
    required_keys = get_required_keys()
    missing_keys = contains_keys(input_values, required_keys) # Check if
    ↪  input_values contains all required keys.
    if missing_keys != None: # If keys are missing.
        raise MissingKeyError(missing_keys[0]) # Raise error for the first
        ↪  missing key.

    # Validate key types using loops.

    # Validate all strings.
    str_keys = ["Bolt.Bolt_Hole_Type", # List of all parameters that are strings
                "Bolt.TensionType",
                "Bolt.Type",
                "Bolt.Connectivity",
                "Bolt.Connector_Material",
                "Design.Design_Method",
                "Detailing.Edge_type",
                "Material",
                "Member.Supported_Section.Designation",
                "Member.Supported_Section.Material",
                "Member.Supporting_Section.Designation",
                "Member.Supporting_Section.Material",
                "Module",
                "Weld.Fab"]
    for key in str_keys: # Loop through all keys.
        validate_string(key) # Check if key is a string. If not, raise error.

    # Validate for keys that are numbers
    num_keys = [("Bolt.Slip_Factor", True) # List of all parameters that are
    ↪  numbers (key, is_float)
                ("Detailing.Gap", False),
                ("Load.Axial", False),
                ("Load.Shear", False),
                ("Weld.Material_Grade_OverWrite", False)]
    for key in num_keys: # Loop through all keys.
        validate_num(key[0], key[1]) # Check if key is a number. If not, raise
        ↪  error.

    # Validate for keys that are arrays
    arr_keys = [("Bolt.Diameter", False), # List of all parameters that can be
    ↪  converted to numbers (key, is_float)
                ("Bolt.Grade", True),
                ("Connector.Plate.Thickness_List", False)]
    for key in arr_keys:
        validate_arr(key[0], key[1]) # Check if key is a list where all items can
        ↪  be converted to numbers. If not, raise error.
```

Once all the input values are validated, only then can they be sent to the **Osdag application** to run calculations, such as the getting the output values, as shown below.

2.4 Output Generation

The output calculation is done entirely by the **Osdag application**. To execute these, we need to create an instance of our desired module, and use it to run the calculation. The bridge includes a program, which can ask this module to run the calculations, and then format the output parameters in a way which can be sent in an HTTP request and parsed by the client. The following program truns these calculations and returns it in an easily readable format

```
def generate_ouptut(input_values: Dict[str, Any]) -> Dict[str, Any]:
    """
    Generate, format and return the input values from the given output values.
    Output format (json): {
        "Bolt.Pitch":
            "key": "Bolt.Pitch",
            "label": "Pitch Distance (mm)"
            "value": 40
        }
    }
    """
    output = {} # Dictionary for formatted values
    module = create_from_input(input_values) # Create the module, and input the
    ↪  given values in them
    # Different output values are generated separately.
    raw_output_text = module.output_values(True) # Generate output values in
    ↪  unformatted form.
    raw_output_spacing = module.spacing(True) # Output values for spacing
    raw_output_capacities = module.capacities(True) # Output values for
    ↪  capacities
    raw_output = raw_output_capacities + raw_output_spacing + raw_output_text #
    ↪  Combine all of them, to get the whole list of output values
    os.system("clear") # Will print a lot of unnessecarry stuff to the console,
    ↪  clear it.
    # Loop over all the text values and add them to ouptut dictionary.
    for param in raw_output: # For each output value
        if param[2] == "TextBox": # If the parameter is a text output,
            key = param[0] # A unique id for each type of output value
            label = param[1] # Human-readable string for the same
            value = param[3] # The actual output value (as a string)
            output[key] = {
                "key": key,
                "label": label,
                "value": value
            } # Set label, key and value in output
    return output
```

The output values were originally meant to be displayed in a tab, with buttons for more information. These values are formatted in a way such that the original GUI could be created directly from them. The program above, first generates each type of output value (including the ones inside the buttons), and starts extracting the important pieces of information from them. After that, it puts them inside a dictionary, which can be used as a JSON representation by the server to put in the HTTP response.

2.5 CAD Model Generation

The CAD model generation was a difficult part to execute. The original version of Osdag used a library known as PythonOCC to render the CAD model to a 3d view display. The modlel would be created, and then directly accessed by the GUI to be rendered. This is a problem, since the model is stored as a Python object, which would be very difficult to send across a network, and be parsed in javascript. This was also one of the backend tasks that need to be most closely integrated with the Frontend, since the CAD model type needs to be supported by THREE.js, the library that is being used to render 3d models on the frontend.

Helpfully though, there was a utility in the Osdag application, which allows a CAD model to be exported to be saved in a 3d model file. This allowed to export it into formats such as BREP, STL, IGS or STEP. This feature needed a few steps to be set up though, and didn't have a single function in the module to execute it. Another problem was that this function was written in a module-dependent manner, and many modules had different setup steps. This is why we needed to create separate implementations for each module. A 3d model is saved to a file (by the **Osdag application**), read by the **API**, sent to the frontend, and then deleted.

Originally, an STL file would be generated by the function, since that supported by default by the THREE.js renderer. The code for generating the STL file looks like this:

```
from cad.common_logic import CommonDesignLogic # For generating the CAD model
from osdag_api.modules.fin_plate_connection import * # Easily generating a module
from OCC.Core.StlAPI import StlAPI_Writer # Converting to STL
```

```
# The following imports are used for creating a dummy GUI so that the CAD model
↪  can be generated
from OCC.Display.backend import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
# Important general functions
from Common import *

def create_stl(input_values: Dict[str, Any], file_path: str) -> :
    """Creates an STL file from the input values of the cad model"""
    app = QApplication(["-v"]) # Create a dummy application for the CAD Model
    ↪  Generater to believe that there is a GUI.

    fp = create_from_input(input_values) # Create the module (shown in previous
    ↪  sections)

    # The CAD model generater needs a display to generate an STL - the following
    ↪  lines of code are for that
    used_backend = load_backend(None) # Checking if the 3d viewer can be used
    if 'qt' in used_backend: # If the GUI needs to be created
        from OCC.Display.qtDisplay import qtViewer3d # Import the 3d viewer
        ↪  library
        QtCore, QtGui, QtWidgets, QtOpenGL = get_qt_modules() # The modules used
        ↪  to generate the 3d viewer
    dummyview = qtViewer3d() # Dummy 3d viewer
    dummyview.InitDriver() # Start it up
    display = dummyview._display # Get the 3d viewer's display

    # Generate the 3d model, so that it can be exported
    commonLogic = CommonDesignLogic(display, '', fp.module, fp.mainmodule) # The
    ↪  object that creates the 3d model
    commonLogic.display = display # Give it a display to work with
    commonLogic.call_3DModel(True, fp) # Set the module of the 3d model generator
    commonLogic.component = "Model" # The component that will be generated (The
    ↪  whole model, in this case)
    fuse_model = commonLogic.create2Dcad() # Create the whole 3d model from all
    ↪  the objects, that have been generated.
    os.system("clear") # The above code will output a lot of logs in the console
    ↪  - clear it.

    # Convert it to an STL
    stl_writer = StlAPI_Writer() # The object that converts to STL
    stl_writer.SetASCIIMode(False)
    stl_writer.Write(fuse_model, file_path) # Export the model to an STL
```

But there was a problem with using the STL model in the 3d viewer. Even though it was easy to send to the **frontend**, and easy to display, there were constant glitches while viewing the 3d model, and strange artefacts, which would be a problem for people trying to inspect it. None of the other formats (BREP/STEP/IGS) had supported loaders in the THREE.js library. The other format, that was supported by THREE.js, was an OBJ file.

After some searching, we came across a tool, named Freecad Utils[1] This tool

[1] I'm sorry I could not find a reference for this.

allows the software to convert a BREP file to an OBJ file using a script. This means, we could just generate the BREP file directly, and the **API** could handle the conversion. The following is the code to generate the BREP file.

The first function figures out the type of connectivity for the model generator to use. This function can be used for all the *Shear Connection* type modules, so it is stored in the file `osdag_api/modules/shear_connection_common.py`. The second function generates the CAD model.

```
from cad.common_logic import CommonDesignLogic
from OCC.Display.backend import *
from Common import *
def setup_for_cad(cdl: CommonDesignLogic, module_class):
    """Sets up the CommonLogicObjct before generating CAD model"""
    cdl.module_class = module_class # The python class of the module that we will
    ↪  be generating the CAD model for (in our case FinPlateConnection)
    module_object = module_class() # Instantiate that module class
    cdl.loc = module_object.connectivity # Set the connectivity of the module in
    ↪  the design logic object.
    if cdl.loc == CONN_CWBW: # If connection type is 'Column Web-Beam Web'.
        cdl.connectivityObj = cdl.create3DColWebBeamWeb() # I guess it creates
        ↪  the connection object.
    if cdl.loc == CONN_CFBW: # If connection type is 'Column Flange-Beam Web'.
        cdl.connectivityObj = cdl.create3DColFlangeBeamWeb() # Create the other
        ↪  type of connection object
    else: # If it is none of them,
        cdl.connectivityObj =cdl.create3DBeamWebBeamWeb() # I guess it creates
        ↪  the last type of connection.
```

Figures out the connectivity object used for generating the CAD model

```
import os # Clearing the terminal
import typing # Type names
from typing import Dict, Any, List # Use these type names
# The following import will log a lot of data, and we don't want to print that,
↪  so the stdout will be temporarily redirected
old_stdout = sys.stdout # Backup old stdout
sys.stdout = open(os.devnull, "w") # Redirect stdout
from design_type.connection.fin_plate_connection import FinPlateConnection #
↪  Import the module class from the osdag application
from OCC.Core import BRepTools # Used for generating a BREP model
import osdag_api.modules.shear_connection_common as scc # The above function
sys.stdout = old_stdout # Reset stdout

def create_cad_model(input_values: Dict[str, Any], section: str, session: str) ->
↪  str:
    """Generate the CAD model from input values as a BREP file. Return file
    ↪  path."""
    if section not in ("Model", "Beam", "Column", "Plate"): # The code can
    ↪  generate three sections of the model, or the whole model.
        raise InvalidInputTypeError("section", "'Model', 'Beam', 'Column' or
        ↪  'Plate'") # Raise an error if it is not one of them
    module = create_from_input(input_values) # Create module from input.
    cld = CommonDesignLogic(None, '', module.module, module.mainmodule) # Object
    ↪  that will create the CAD model.
    scc.setup_for_cad(cld, module) # Set up the cad model generator for this type
    ↪  of connection
    cld.component = section # Specify the section of the model that will be
    ↪  generated.
    model = cld.create2Dcad() # Generate CAD Model.
    os.system("clear") # Clear the console
    file_name = session + "_" + section + ".brep" # Generate a temporary
    ↪  filename, from the unique session id.
    file_path = "file_storage/cad_models/" + file_name # Put it in the correct
    ↪  folder for temporary files
    BRepTools.breptools.Write(model, file_path) # Generate CAD Model
    return file_path
```

CAD model generation implementation for the FinPlateConnection module.

The above code generates a 3d model using the **Osdag Application**'s 3d model generation capabilities, and saves it to a temporary file unique for each user. This is read by the **Osdag Api**, and converted to an OBJ. Both those files are immediatly deleted, and the OBJ data is sent accross the network by the server as an HTTP response, to the client. The client can then display the OBJ 3d model in the viewer using THREE.js

2.6 Design Report Generation

The last feature, that the **bridge** is responsible for handling is in fact the last step in the Osdag workflow - generating a design report. The design report is a PDF, which contains all the information about the module in it. This report is generated by the **Osdag Application** and saved as a PDF file. For this, we could use a similar approach as the CAD model, by creating a temporary file, reading it, and then deleting it. The data can be sent as an HTTP response.

The primary job of the **bridge** here is to add the report metadata, in case it has not been provided and fill unused values with default values. These metadata values are then put in the design report, in the heading.

```
def generate_report(input_values: Dict[str, Any], metadata: Dict[str, Any],
↪  report_id: str) -> str:
    """Generate the design report from the input values as a PDF file. Return
    ↪  file path."""
    metadata_profile = { # List of all the required metadata (key: default value)
        "CompanyName": "Your Company",
        "CompanyLogo": "",
        "Group/TeamName": "Your Team",
        "Designer": "You",
    }
    metadata_other = { # List of all the required metadata (key: default value)
        "ProjectTitle": "Fin Plate Connection",
        "Subtitle": "",
        "JobNumber": "1",
        "AdditionalComments": "No Comments",
        "Client": "Someone Else",
    }
    if "Profile_Summary" in metadata.keys(): # Check if profile metadata is
    ↪  provided
        profile = {} # Create a new dict for the profile metadata
        for key in metadata_profile.keys(): # Go through all the keys in given
        ↪  profile metadata
            if key in metadata["ProfileSummary"].keys(): # if key exists
                profile[key] = metadata["ProfileSummary"][key] # use value
            else: # Otherwise, stick to default value.
                profile[key] = metadata_profile[key]
    else: # Otherwise, stick to default metadata
        profile = metadata_profile
    file_path = "file_storage/design_reports/" + report_id # Generate a temporary
    ↪  filename from the session id
    metadata_final = {"ProfileSummary": profile, "filename": file_path,
    ↪  "does_design_exist": True, "logger_messages": ""} # Add the metadata in
    ↪  the parameters
    # Add other metadata
    for key in metadata_other.keys(): # Go through all metadata keys
        if key in metadata.keys(): # if key exists
            metadata_final[key] = metadata[key] # Use key
        else: # if not, use default value
            metadata_final[key] = metadata_other[key] # Use default value
    module = create_from_input(input_values) # Create module from input
    module.save_design(metadata_final) # Create design report
    return file_path
```

The file is stored as a temporary file `file_storage/design_reports/<session-id>.pdf`. This is loaded and deleted by the **Osdag API**, and then sent to the user as a downloadable file. The user can then save it on their computer.

Chapter 3

Web API

Although most of my work was focused on programming the **bridge** between the **Osdag API**, and the **Osdag Application**, I had originally written part of the **Osdag API**, which later lead to the development of the actual one. The original API was written using Django, a python library used for building web apps. The later version of the API, not built by me, was built upon how this one was built, even reusing some of the functions from it. I later worked to upgrade the CAD Model API, to such a way where it can be used on multiple devices at the same time.

3.1 Web API Creation

The older version of the **Web API**, used a way of maintaining data on the backend, which I had named the sessions protocol. This protocol was a way of recognising different users on different devices. The protocol would create a temporary id, when a user opens a module, and use it to collect saved data from the database, such as the input parameters when retrieving CAD model. This session protocol was partly retained in the newer version, but discarded for most of the part. Other implementations, such as the CAD model API, were relatively unchanged.

3.1.1 Session Protocol

```python
from django.shortcuts import render, redirect
from django.utils.html import escape, urlencode
from django.http import HttpResponse, HttpRequest
from django.views import View
from osdag.models import Design
from django.utils.crypto import get_random_string
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator
from osdag_api import developed_modules
import typing


@method_decorator(csrf_exempt, name='dispatch')
class CreateSession(View):
    """
```

```
        Create a session in database and set session cookie.
            Create Session API (class CreateSession(View)):
                Accepts POST requests..
                Accepts content-type/form-data.
                Request body must include module id.
                Creates a session object in db and returns session id as cookie.
    """
    def post(self,request: HttpRequest) -> HttpResponse:
        module_id = request.POST.get("module_id") # Type of Osdag Module
        if module_id == None or module_id == '': # Error Checking: If module id
        ↪  provided.
            return HttpResponse("Error: Please specify module id", status=400) #
            ↪  Returns error response.
        if request.COOKIES.get("design_session") is not None: # Error Checking:
        ↪  Already editing design.
            return HttpResponse("Error: Already editing module", status=400) #
            ↪  Returns error response.
        if module_id not in developed_modules: # Error Checking: Does module api
        ↪  exist
            return HttpResponse("Error: This module has not been developed yet",
            ↪  status=501) # Return error response.
        response = HttpResponse(status=201) # Statuscode 201 - Successfully
        ↪  created object.
        cookie_id = get_random_string(length=32) # Session Id - random string.
        response.set_cookie("design_session", cookie_id) # Set session id cookie.
        try: # Try creating session.
            session = Design(cookie_id = cookie_id, module_id = module_id) #
            ↪  Create design session object in db
            session.save()
        except Exception as e: # Error Checking: While saving design.
            return HttpResponse("Inernal Server Error: " + repr(e), status=500) #
            ↪  Return error response.
        response = HttpResponse(status=201) # Statuscode 201 - Successfully
        ↪  created object.
        response.set_cookie("design_session", cookie_id) # Set session id cookie
        return response

@method_decorator(csrf_exempt, name='dispatch')
class DeleteSession(View):
    """
        Delete session cookie and session data in db.
            Delete Session API (class CreateSession(View)):
                Accepts POST requests.
                Requires no POST data.
                Requires design_session cookie.
                Deletes session object in db and deletes session id cookie.
    """
    def post(self,request: HttpRequest) -> HttpResponse:
        cookie_id = request.COOKIES.get("design_session") # Get design session
        ↪  id.
        if cookie_id == None or cookie_id == '': # Error Checking: If design
        ↪  session id provided.
            return HttpResponse("Error: Please open module", status=400) #
            ↪  Returns error response.
        if not Design.objects.filter(cookie_id=cookie_id).exists(): # Error
        ↪  Checking: If design session exists.
```

```
            return HttpResponse("Error: This design session does not exist",
             ↪  status=404) # Return error response.
        try: # Try deleting session.
            design_session = Design.objects.get(cookie_id=cookie_id) # Design
             ↪  session object in db.
            design_session.delete()
        except Exception as e: # Error Checking: While saving design.
            return HttpResponse("Inernal Server Error: " + repr(e), status=500) #
             ↪  Return error response.
        response = HttpResponse(status=200) # Status code 200 - Successfully
         ↪  deleted .
        response.delete_cookie("design_session")
        return response
```

Above is the code for handling the session protocol. These function create a session and delete a session.

The first function handles a POST request. When sent, it accepts data which needs to contain name of the module being used. A design session object is created, and stored in the database. This can be looked up by the session id, and contains information such as the input parameters. The response sends a cookie, which is stored on the user's device. This cookie contains the session id, and is sent with following requests, so that the Design Sesion object can be looked up when required.

The second function also handles a post request. This one simply checks for the cookie containing the session id. Once it gets the session id, it locates the design session object, and deletes it.

3.1.2 Input Values API

```
from django.shortcuts import render, redirect
from django.utils.html import escape, urlencode
from django.http import HttpResponse, HttpRequest
from django.views import View
from osdag.models import Design
from django.utils.crypto import get_random_string
from django.views.decorators.csrf import csrf_exempt
from django.utils.decorators import method_decorator
from osdag_api import developed_modules, get_module_api
from osdag_api.errors import OsdagApiException
import typing
import json

@method_decorator(csrf_exempt, name='dispatch')
class InputValues(View):
    """
        Update input values in database.
        InputValues API (class InputValues(View)):
            Accepts POST requests.
            Accepts content_type application/json
            Request must provide session cookie id.
    """
    def post(self, request: HttpRequest):
        cookie_id = request.COOKIES.get("design_session") # Get design session
         ↪  id.
```

```
if cookie_id == None or cookie_id == '': # Error Checking: If design
↪  session id provided.
    return HttpResponse("Error: Please open module", status=400) #
    ↪  Returns error response.
if not Design.objects.filter(cookie_id=cookie_id).exists(): # Error
↪  Checking: If design session exists.
    return HttpResponse("Error: This design session does not exist",
    ↪  status=404) # Return error response.
if not request.content_type == "application/json": # Error checking: If
↪  content/type is not json.
    return HttpResponse("Error: Content type has to be text/json",
    ↪  status=400) # Return error response.
try: # Error checking while loading body.
    body_unicode = request.body.decode('utf-8')
    input_data = json.loads(body_unicode)
except Exception as e:
    return HttpResponse("Error: Internal server error: " + repr(e),
    ↪  status=500) # Return error response
try: # Error checking while getting module api
    design_session = Design.objects.get(cookie_id=cookie_id) # Get the
    ↪  design session from the db.
    module_api = get_module_api(design_session.module_id) # Get the
    ↪  module api using the module id.
except Exception as e:
    return HttpResponse("Error: Internal server error: " + repr(e),
    ↪  status=500) # Return error response
try:
    module_api.validate_input(input_data) # Check if input data is valid.
except OsdagApiException as e: # Catch input validation error.
    return HttpResponse("Error: " + repr(e), status=400) # Return error
    ↪  response
except Exception as e: # Catch other error.
    return HttpResponse("Error: Internal server error: " + repr(e),
    ↪  status=500) # Return error response
try: # Error checking while saving input values
    json_data = json.dumps(input_data) # Convert dict to json string

    ↪  Design.objects.filter(cookie_id=cookie_id).update(input_values=json_data)
    ↪  # Sets the input values in the design session object in the db.
    Design.objects.filter(cookie_id=cookie_id).update(current_state=True)
    ↪  # The input values have been set
except Exception as e: # Handle other errors
    return HttpResponse("Error: Internal server error: " + repr(e),
    ↪  status=500) # Return error response
response = HttpResponse(status=200) # Status code 200 - Success!
return response
```

The above function handles setting input parameters. The data from the client is meant to be a dictionary, containing all the parameters. First, these parameters are validated by the **bridge**. Then, the session_id cookie is used to locate the correct design session in the databas, and then the input parameters are set in the design session. The return value contains no data

Most of the code in the above section is used for error checking. Since all the data comes from an HTTP request, we have no assurance that it is formatted correctly. In case any of the data is formatted such that it causes an error, the server code may shut down, and require manual restarting. That is why any of code handling HTTP request data or using the **Osdag application** is checked for errors, which are handled and returned in the request.

3.1.3 Output Values API

```
# Imports same as above

@method_decorator(csrf_exempt, name='dispatch')
class OutputValues(View):
    """
        Update input values in database.
            Output Values API (class OutputValues(View)):
                Accepts GET requests.
                Returns content_type application/json
                Request must provide session cookie id.
                Data Format: (json): {
                    "Bolt.Pitch":
                        "key": "Bolt.Pitch",
                        "label": "Pitch Distance (mm)"
                        "value": 40
                    }
                }
    """
    def get(self, request: HttpRequest):
        cookie_id = request.COOKIES.get("design_session") # Get design session
        ↪  id.
        if cookie_id == None or cookie_id == '': # Error Checking: If design
        ↪  session id provided.
            return HttpResponse("Error: Please open module", status=400) #
            ↪  Returns error response.
        if not Design.objects.filter(cookie_id=cookie_id).exists(): # Error
        ↪  Checking: If design session exists.
            return HttpResponse("Error: This design session does not exist",
            ↪  status=404) # Return error response.
        try: # Error checking while loading input data
            design_session = Design.objects.get(cookie_id=cookie_id) # Get
            ↪  session object from db.
            module_api = get_module_api(design_session.module_id) # Get module
            ↪  api
            if not design_session.current_state: # Error Checking: If input data
            ↪  not entered.
                return HttpResponse("Error: Please enter input data first",
                ↪  status=409) # Return error response.
            input_values = json.loads(design_session.input_values) # Load input
            ↪  data into dictionary.
        except Exception as e:
            return HttpResponse("Error: Internal server error: " + repr(e),
            ↪  status=500) # Return error response.
        try: # Error checking while calculating output data.
            output_values = module_api.generate_ouptut(input_values)
        except Exception as e:
            return HttpResponse("Error: Internal server error: " + repr(e),
            ↪  status=500) # Return error response.
        try: # Error checking while formatting output as json.
```

```
            output_data = json.dumps(output_values)
        except Exception as e:
            return HttpResponse("Error: Internal server error: " + repr(e),
            ↪  status=500) # Return error response.
        response = HttpResponse(status=200)
        response["content-type"] = "application/json"
        response.write(output_data)
        return response
```

The above code is a function that handles a GET request. This request is supposed to return the output parameters, once the input parameters are entered. First, it checks for the design session cookie, and looks up the entry in the database. Once it finds it, it uses the input values in the entry, and passes that to the **bridge**. The **API**, formats the data, and returns it as the body of the HTTP response.

3.2 Cad Model Generation

I had originally developed a version of the CAD model API, which could return a BREP file generated by the software and send it to the client. This would use the a temporary folder to store the file, and delete it after reading it. The following is the code for executing that.

```
@method_decorator(csrf_exempt, name='dispatch')
class CADGeneration(View):
    """
        Update input values in database.
            CAD Model API (class CADGeneration(View)):
                Accepts GET requests.
                Returns BREP file as content_type text/plain.
                Request must provide session cookie id.
    """
    def get(self, request: HttpRequest):
        cookie_id = request.COOKIES.get("design_session") # Get design session
        ↪  id.
        if cookie_id == None or cookie_id == '': # Error Checking: If design
        ↪  session id provided.
            return HttpResponse("Error: Please open module", status=400) #
            ↪  Returns error response.
        if not Design.objects.filter(cookie_id=cookie_id).exists(): # Error
        ↪  Checking: If design session exists.
            return HttpResponse("Error: This design session does not exist",
            ↪  status=404) # Return error response.
        try: # Error checking while loading input data
            design_session = Design.objects.get(cookie_id=cookie_id) # Get
            ↪  session object from db.
            module_api = get_module_api(design_session.module_id) # Get module
            ↪  api
            if not design_session.current_state: # Error Checking: If input data
            ↪  not entered.
                return HttpResponse("Error: Please enter input data first",
                ↪  status=409) # Return error response.
            input_values = json.loads(design_session.input_values) # Load input
            ↪  data into dictionary.
        except Exception as e:
            return HttpResponse("Error: Internal server error: " + repr(e),
            ↪  status=500) # Return error response.
        section = "Model" # Section of model to generate (default full model).
        if request.GET.get("section") != None: # If section is specified,
            section = request.GET["section"] # Set section
        try: # Error checking while Generating BREP File.
            path = module_api.create_cad_model(input_values, section, cookie_id)
            ↪  # Generate CAD Model.
        except OsdagApiException as e: # If section does no exist
            return HttpResponse(repr(e), status=400) # Return error response.
        except Exception as e:
            return HttpResponse("Error: Internal server error: " + repr(e),
            ↪  status=500) # Return error response.
        with open(path, 'r') as f: # Open CAD file
            cad_model = f.read() # Read CAD file Data
        os.remove(path) # Delete CAD File
        response = HttpResponse(status=200)
        response["content-type"] = "text/plain"
        response.write(cad_model)
        return response
```

This code would save the file, and then read it and delete it instantly. The data would get sent as the HTTP response body. Later on in the newer version of the **Osdag API**, this method was abandoned. The **API** would first generate the 3d model, and then save it in the directory. A tool named Freecad Utils would then be called by the **API** to convert the BREP file to an OBJ file. The OBJ file would then be saved in a directory on the server. The **Backend** and the **Frontend** would be hosted on the same server, the frontend could access that file, and render it.

There were a number of problems with this, other than the fact that this is not a good way of transferring files from a backend to a frontend. Ideally, a backend and a frontend of a software should be independent from each other, and communicate with each other by a single channel. This would result in a major problem, that if multiple users opened different models on their computer, it would result in conflicts between which model was being shown on the frontend. To prevent this, I decided to rewrite how the backend and the frontend would handle this – below is the change I made to the backend:

```
# The code shown below is under the class CADGeneration(View)

time.sleep(3)
with open(output_dir, 'r') as data_file: # Open the file that contains the OBJ
↪  file
    obj_data = data_file.read() # Read OBJ data from the file
os.system("rm {}".format(output_dir)) # Delete Brep file
os.system("rm {}".format(path_to_file)) # Delete OBJ file
response = HttpResponse(obj_data, status=201) # Pack the obj data into the HTTP
↪  response body
response["content-type"] = "text/plain" # Pretend it's plain text.
return response
```

To load in data from this format, the frontend would also need to be adapted to reading in the data. The next chapter is regarding the how the frontend was needed to be changed to adapt to that.

Chapter 4

Frontend Work

Once the way CAD models were handled on the **backend** was changed, the **frontend** needed to be updated accordingly. Since the module object cannot be loaded from a file anymore, I needed to figure out a way of loading it from the request. This meant changing multiple parts of the CAD model rendering code.

This also meant working using a library known as react, which I had never used before. React allows one to build web apps, which has it's own state management system. While working on the **frontend**, I added a set of new features to the code, which would improve the CAD model viewing experience, and all of this resulted in an updated version of the CAD model renderer. This included:

- Multiple users being able to view different modules in parallel.
- Different colours for different sections in of the model, as in the original.
- Displaying the module loading state to the user, and notifying loading errors.

The majority of the code of the updated CAD model renderer is located in the file `osdagclient/src/assets/components/shearConnection/cad_renderer_updater.jsx`. This file contains various functions. Each of these functions play an important role in rendering the final result.

Firstly, we need to load the CAD model from the **backend**. We can send a request to the API to achieve this, as shown below.

```
const BASE_URL = 'http://127.0.0.1:8000/';
async function load_cad_model(section) { /* Async functions wait for a request to
↪  be finished before executing the rest of the code */
    /* Loads the CAD model, and it on a view section */
    var response = await fetch(`${BASE_URL}design/cad?section=${section}`, { /*
    ↪  Request the correct section of the model /
        method: 'GET', /* Get request */
        mode: 'cors',
        credentials: 'include'
    }); /* Send a request to the API to get the CAD model */
    var cad_data = undefined; /* Will return undefined if something goes wrong */
    if (response.status == 201) { /* If CAD model successfully generated */
        cad_data = await response.text();/* Get OBJ data in string format */
    } return cad_data; /* Return the response */
}
```

To render different sections of the model in different colous, we need to load them as different objects. Since the **API** has the ability to send a specific section of the model in the response, we can use this to request each section separately. A problem I encountered here, was that each of the sections were rotated differently. For each type of connection, the rotations needed to be set manually. A little bit of state management was required in this situation. The following code is in the `osdagclient/public/src/shearConnection/FinePlate.jsx` file, under the function `FinePlate()`. That function mostly not written by me, but I had changed parts of it to work with the updated CAD model renderer.

```
import { useContext, useEffect, useState } from 'react'; /* State management */
import { init_cad_data, load_cad_model, RenderCadModel } from
↪  './cad_renderer_updated'


function FinePlate() {
    /* Handles the interface for the FinPlateConnection Module */

    // ... A lot of code here that is ommited for clarity ...

    /* The variables defined below are for handling the state of the CAD model:
     *  -> To prevent the rotation from changing every time the connection type
     *     is changed in the input parameters, it needs to be fixed once the CAD
     *     model is loaded.
     *  -> The other state managements are for handling the loading state, which
     *     can be in four possible states:
            -> Model is not yet loaded.
            -> Model is being loaded.
            -> Model has been loaded and can be viewed.
            -> Error in  loading model.
     * The other variables handle that state.
    */
    const [loadingCadModel, setLoadingCadModel] = useState(false); /* Will be
    ↪  true when the CAD model is being loaded, otherwise false. */
    const [currentType, setCurrentType] = useState("Column Flange-Beam-Web"); /*
    ↪  Fixes the current connection type, to set the rotation. */
    const [cadFail, setCadFail] = useState(false); /* Is true if the CAD model
    ↪  loading fails, otherwise false. */

    // ... Some other code here ...'

    const handleSubmit = async () => { /* Most of this I did not write */
        // ...
        createDesign(param)
        init_cad_data(setLoadingCadModel, setCurrentType, selectedOption,
        ↪  setCadFail); /* Download the CAD model (Added by me) */
        setDisplayOutput(true)
    }
}
```

The function `init_cad_data` downloads the correct parts of the CAD model. It is located in the file `cad_model_updated.jsx`:

```
import React, {useMemo, useState} from "react"; /* State management */
```

```
// Model Sections
var plate_data = undefined;
var beam_data = undefined;
var column_data = undefined;

async function init_cad_data(setLoadingCadModel, setCurrentType, selectedOption,
↪  setCadFail) {
    /* Download all CAD models */
    setCadFail(false); /* Not yet failed */
    setLoadingCadModel(true); /* Downloading CAD data. */
    plate_data = await load_cad_model('Plate'); /* Load the "Plate" section */
    if (plate_data == undefined) { /* Check if data has arrived */
        setCadFail(true); /* Set the failed flag */
        return; /* No need to continue */
    }
    column_data = await load_cad_model('Column'); /* Load the "Column" section */
    if (column_data == undefined) { /* Errror checking */
        setCadFail(true);
        return;
    }
    beam_data = await load_cad_model('Beam');
    if (column_data == undefined) {
        setCadFail(true); /* Set Failiure Message */
        return;
    }
    setCurrentType(selectedOption); /* Set the rotation value for the CAD model
    ↪  */
    setLoadingCadModel(false); /* Done downloading cad data */
}
```

Once that function is called, the CAD model can be rendered in the view The function for rendering the CAD model is given below:

```
import { OBJLoader } from 'three/examples/jsm/loaders/OBJLoader.js'
import loading_icon from '../../assets/loading_icon.gif'
import { Canvas } from '@react-three/fiber'
import { Euler } from 'three';

// The properties given below contai the flags from the FinePlate() function

function RenderCadModel(properties) {
    /* Renders the model and creates the viewport */
    if (!properties.cad_fail) { /* If the CAD rendering has failed */
        if (properties.render_boolean) { /* If the model is to be rendered */
            if (!properties.loadingState) { /* If the model has loaded */
                let rotation = new Euler(Math.PI / -2,0,0); /* The first kind of
                ↪  rotation */
                if (properties.type == "Column Web-Beam-Web") rotation = new
                ↪  Euler(0,Math.PI / -2,0); /* New rotation for this type  */
                else if (properties.type == "Beam-Beam") rotation = new
                ↪  Euler(Math.PI / 2,0,0); /* Another rotation for this type */
                if (plate_data  && beam_data && column_data ) { /* If all the
                ↪  models have beeen loaded */
                    var plate_3d = load_three_mesh(plate_data, "#a89732", 0.01,
                    ↪  rotation); /* Create a mesh object for the plate */
                    var column_3d = load_three_mesh(column_data, "#32a840", 0.01,
                    ↪  rotation); /* Ditto for column */
                    var beam_3d = load_three_mesh(beam_data, "#3268a8", 0.01,
                    ↪   rotation); /* Ditto for beam */

                    return ( /* What to put in the viewport */
                        <div>
                            <CadModelViewport> /* Creates a THREE.js viewport */
                                {plate_3d} /* Put all the models inside of it */
                                {beam_3d}
                                {column_3d}
                            </CadModelViewport>
                        </div>
                    )
                }
                else { /* If for some reason any of the sections failed to load
                ↪   */
                    return error_background; /* Show a background with an error
                    ↪   message */
                }
            }
            else { /* If the model is still loading */
                return loading_background; /* Show the background with the
                ↪   loading icon */
            }
        } else { /* Nothing on the screen */
            return empty_image_background; /* Show the empty background */
        }
    } else { /* Cad model generation failed */
        return error_background;
    }
}
```

The OBJ data needed to be loaded from a string, which was loaded from the request. This had been a problem for a while, which was why the model was loaded from a file, until now. After some searching, I found a function which can load OBJ data into a THREE.js object from a string. I used it to write a function which can handle the conversion named `load_three_mesh`.This function converts the OBJ data into a THREE.js readable object, which can be rendered in the viewport:

```
function load_three_mesh(data, colour, scale, rotation) {
    /* Converts the obj into a three.js geometry file */
    const loader = new OBJLoader();
    let object = loader.parse(data); /* Parse the OBJ string into an object */
    let geometry = object_to_mesh(object); /* Convert the three.js object to
    ↪   geometry */
    let mesh_object = ( /* Three.js Mesh object with colour */
        <mesh geometry={geometry} /* Set the mesh geometry */ scale={scale}/*
        ↪   Object Scale */ rotation={rotation}>
            <meshPhysicalMaterial attach = "material" color={colour}
            ↪   metalness={0.25} roughness={0.1} opacity={2.0} transparent =
            ↪   {true} transmission={0.99} clearcoat={1.0}
            ↪   clearcoatRoughness={0.25}/>
        </mesh>
    )
    return mesh_object;
}
```

For completeness's sake, I would like to show functions for generating the viewport during the various states of viewing/loading the CAD model.

```
const empty_image_background = (
    <div style={{ maxwidth: '740px', height: '600px', border: '1px solid black'
    ↪  }}>
      {<img src={cad_background} alt="Demo" height='100%' width='100%' />}
    </div>
); /* Display nothin g */

const loading_background = (
    <div style={{ maxwidth: '740px', height: '600px', border: '1px solid black'
    ↪  }}>
      {<img src={cad_background} alt="Demo" height='100%' width='100%'
      ↪  style={{position: "relative"}}/>}
      {<img src={loading_icon} alt="Loading..." height='100px' width='100px'
      ↪  style={{position: 'relative', top: -350, left: 350}}/>}
    </div>
); /* Display loading icon */

const error_background = (
    <div style={{ maxwidth: '740px', height: '600px', border: '1px solid black'
    ↪  }}>
    {<img src={cad_background} alt="Demo" height='100%' width='100%'
    ↪  style={{position: "relative"}}/>}
    <h1 style={{color: "red", position: 'relative', top: -350, left: 120}}>Error
    ↪  in generating CAD model.</h1>
  </div>
); /* Display error message */
const fail_message = "CAD model generation failed";
function CadModelViewport(properties) {
    /* Returns a CAD model viewport to place objects in */
    return ( /* Source - FinePlate.jsx */
        <div style={{ maxwidth: '740px', height: '600px', border: '1px solid
        ↪  black', backgroundImage: `url(${cad_background})` }}>
            <Canvas gl={{ antialias: true }} camera={{ aspect: 1, fov: 1500,
            ↪  position: [10, 10, 10] }} >
                <group name='scene'>
                    <axesHelper args={[200]}/>
                    {properties.children}
                    <OrbitControls />
                </group>
            </Canvas>
        </div>
    );
}
```

Chapter 5

Conclusions

5.1 A Summary of my Work

The Osdag Application, originally designed as a desktop application, had two major flaws - the software was only compatible with specific system specifications, and it required a long complex installation process. Therefore we decided to migrate it as a web application, which can run on a browser, therefore eliminating the need for installation. In the beginning, I searched through the source code trying to understand which functions in the original code was used to run the module calculations. After finding those functions, testing them, I started writing a small library which would work as a wrapper-layer over the application so that other parts of the program could easily communicate with it. This wrapper layer would work as a simple interface with various functions implemented separately for each module but providing the same functionality for the API. After finishing a limited version of the bridge, I started working on a simple version of the API, which was partially modified and used in the final version. Then my work progressed into the CAD model creation part of the project, involving trying various file formats, and a little work in improving the CAD model loading in the front-end.

My primary contribution to the development was the structure of the bridge, and how the API would communicate with the bridge. I had tried to develop the bridge in such a manner such that it could be extended and new functionality could be added without changing the original software or the API.

5.2 Suggestions for Further Development

I would suggest two developments which would be required so that development of the software can progress in a more streamlined manner. Firstly, it would make sense to correct the "osdag_api" folder to "osdag_bridge" and "osdag" to "osdag_api", since these namings accurately describe what these folders are used for

Secondly I would suggest that the Osdag API calls be standardised as to whether the design sessions protocol is to be used or discarded, or if used for certain specific situations, it should be referred to as something else.