Criterion C

On a high level overview, the website I designed is a dynamic web application that uses tags and title to organise pdf documents in a database, which, specifically for my client, are psychology papers that students will use, but because the application is highly dynamic, it can also be used for other types of paper. I will give a general overview of technical skills and algorithm. I

Most of the processing is done in the server backend, with some light processing mostly responsible for the UI in the front end. All the users using the website share the same database.

List of Technical Skills

- Using multiple frameworks and languages.
- Using Django Models
- Using cloud storage (S3)
- Using JS to dynamically update the page content
- Concurrent programming with JS Promise
- Multithreading in the server to increase responsiveness
- Using presigned URL
- Securely handle API secrets in an open source application
- Graceful multiple fall back to increase responsiveness

List of Algorithm Used

- Concurrency to improve responsiveness
- Memorisation for optimisation
- Primitive Search algorithm

Part 1: UI Design and Dynamic Web Layout

The UI was designed with Semantic UI, rendered with the help of Django Template language with some custom JS script for the dynamic elements.

Example 1: Using Django template language to implement loose coupling. Because all pages share the same menu items and UI library, I have put them in one HTML and allows subsequent pages to extend on that to reduce the code size, reducing the potential sources of bugs. The HTML comments illustrate the technical skills.

In `codestudy/template/codestudy/base.html`:

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
        ...
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/semantic-ui@2.4.2/dist/
        ↪ semantic.min.css"> <!--Semantic UI libray-->
    <meta name="google-signin-client_id"
         content="330093561618-bc6pprilkfqi5sc6oibm376psnfe9tub.apps.googleusercontent
             ↪ .com"> <!--Log in with Google library-->
    <link rel="shortcut␣icon" type="image/png" href="{%␣static␣'favicon.ico'␣%}"/> <!
        ↪ --The favicon to be displayed-->
    <title>{% block title %}Code Study{% endblock %}</title>
</head>
<body>
<div class="ui␣menu">
        ...
    {% if user.can_edit %} <!--Example of Dynamic UI. The options only show if the
        ↪ user has premission to edit the database-->
        <a class="item" id="menu-add-paper" href="{%␣url␣'codestudy:add-paper'␣%}">Add
            ↪ Paper</a>
```

```
        <a class="item" id="menu-edit-tags" href="{% url 'codestudy:edit-tags' %}">Edit
            ↪ Tags</a>
    {% endif %}
    {% for tag_class in tag_classes %} <!--Example of Dynamic UI. The number of items
        ↪ in the menu expands as the user adds more tag class-->
        <div id="menu-{{ tag_class.pk }}" class="ui dropdown item">
            {{ tag_class.name }}
            <i class="dropdown icon"></i>
            <div class="menu">
                {% for tag in tag_class.tag_set.all %}
                    <a class="item" href="{% url 'codestudy:browse' tag_class.name tag.
                        ↪ name %}">{{ tag.name }}</a>
                {% endfor %}
            </div>
        </div>
    {% endfor %}
    <a class="item" id="menu-all-papers" href="{% url 'codestudy:all-papers' %}">All
        ↪ Papers</a>
        ...
</div>
    ...
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></
    ↪ script>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta1/dist/js/bootstrap.
    ↪ bundle.min.js"
        integrity="sha384-ygbV9kiqUc6oa4msXn9868pTtWMgiQaeYH7/
            ↪ t7LECLbyPA2x65Kgf80OJFdroafW"
        crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/semantic-ui@2.4.2/dist/semantic.min.js"></
    ↪ script>
<script src="https://apis.google.com/js/api:client.js"></script>
<script>
        ...
</script>
{% block js %} <!--Example of lose coupling in the Django template langauge. The block
    ↪  tag allows me to add more javascript based on the base-->
{% endblock %}
</body>
</html>
```

Example 2: Even though there are multiple ways to search for a paper, they all share the same HTML base template, except with different data.

In codestudy/templates/codestudy:

```
{% extends "codestudy/base.html" %}
{% load codestudy_tags %}
{% block title %}
    {{ page_title }} | Code Study
{% endblock %}
{% block content %}
    <div class="ui container">
        {% if not papers %} <!--Example of responsive UI design. If there is nothing,
            ↪ the UI displays a message affirmation so the page would not seem to not
            ↪ have loaded-->
            <div class="ui text container">
                <div class="ui message">
                    <div class="header">Nothing to Show</div>
                    <p>
```

```
                    There is nothing in the database that matches your search.
                </p>
            </div>
        </div>
    {% endif %}
    <div class="ui three column stackable grid">
        {% for paper in papers %}
                        ... Layout for the paper. Omitted for concision.
                        <!--Exampl of lose coupling. Depending on the paper
                            ↪ variable passed in from Python script, the document
                            ↪ will display different results-->
        {% endfor %}
    </div>
</div>
{% endblock %}
{% block js %}
    <script>
            ...
        function like(pk) {
                    <!--Example of Dynamic web design. When the bookmark button is
                        ↪ clicked, the JS immediately sends a request to the server to
                        ↪  update the status. Notice that the user is not specified in
                        ↪  the request because that is automatically delivered by a
                        ↪ browser cookie for security purposes (because the cookie is
                        ↪ signed upon login verification).-->
            const likeBtn = $('#like-' + pk);
            const xhr = new XMLHttpRequest();
            xhr.open('GET', '{% url 'codestudy:bookmark' %}?pk=' + pk);
            xhr.send();
            if (likeBtn.hasClass('outline')) {
                likeBtn.removeClass('outline');
            } else {
                likeBtn.addClass('outline');
            }
        }
    </script>
{% endblock %}
```

The templates in combination with a helper function `get_base_context(request)` in `codestudy/views.py` allows the dynamic menu bar to be displayed across all pages without much effort.

In `codestudy/views.py`:

```
def get_base_context(request):
    return {
        'user': get_user(request),
                # get_user is a helper function that returns the User object.
                    ↪ Implementation of it will be addressed in the next section.
        'tag_classes': TagClass.objects.all()
    }
```

Besides the dynamic rendering on the server side, the client also uses dynamic rendering implemented using JS. One prime example is in `codestudy/templates/codestudy/edit-tags.html`:

```
...
function htmlToElement(html) {
    let template = document.createElement('template');
    html = html.trim(); // Never return a text node of whitespace as the result
    template.innerHTML = html;
```

```
    return template.content;
}

function addTag(tagClassPk, tagClassName) {
    const newTagInput = $('#new_tag_input_' + tagClassPk);
    const tagName = newTagInput.val();
    const tagPk = uuid.v4();
    changeLog.newTags.push({
        pk: tagPk,
        name: tagName,
        tagClass: tagClassName
    });
    newTagInput.val('');
    $('#new_tags_' + tagClassPk).before(htmlToElement(`
        <tr id="tag_${tagPk}">
            <td>${tagName}</td>
            <td class="right aligned" onclick="deleteTag('${tagPk}')">
                <button class="ui icon button"><i class="trash icon"></i></button>
            </td>
        </tr>
    `))
}

function deleteTag(tagPk) {
    const tagUi = $(`#tag_${tagPk}`);
    const tagDeleteButton = tagUi.find('td > button');
    if (tagDeleteButton.hasClass('negative')) {
        tagDeleteButton.removeClass('negative');
        changeLog.deletedTags.delete(tagPk);
    } else {
        tagDeleteButton.addClass('negative');
        changeLog.deletedTags.add(tagPk);
    }
    tagDeleteButton.removeClass('active')
}

function addTagClass() {
    const newTagClassInput = $('#new_tag_class_input');
    const tagClassName = newTagClassInput.val();
    if (tagClassName === '') {
        const TagClassError = $('#tag_class_error');
        TagClassError.removeClass('hidden');
        setTimeout(() => TagClassError.addClass('hidden'), 2000);
        return;
    }
    const tagClassPk = uuid.v4();
    changeLog.newTagClasses.push({
        pk: tagClassPk,
        name: tagClassName,
    });
    newTagClassInput.val('');
    $('#new_tag_classes').before(htmlToElement(`
        <div id="tag_class_${tagClassPk}" class="ui stackable grid">
            <div class="twelve wide column">
                <h1>${tagClassName}</h1>
            </div>
            <div class="four wide column right aligned">
                <button class="ui icon fluid button" onclick="deleteTagClass('${
```

```
                ↪ tagClassPk}')"><i class="trash icon"></i></button>
            </div>
        </div>
        <table class="ui left aligned table">
            <tbody>
                <tr id="new_tags_${tagClassPk}">
                    <td>
                        <div class="ui input">
                            <input id="new_tag_input_${tagClassPk}" type="text"
                                ↪ placeholder="New tag...">
                        </div>
                    </td>
                    <td class="right aligned">
                        <button class="ui icon button" onclick="addTag('${tagClassPk}', '
                            ↪ ${tagClassName}')"><i class="plus icon"></i></button>
                    </td>
                </tr>
            </tbody>
        </table>
    `));
}
...
```

Importantly, the webpage tracks and uploads only the changes to the database instead of the entire data The `addTag` and `addClass` functions adds the element to the webpage with the relevant click handler. Admittedly, using a string to render html element is not the most efficient and elegant approach. I should create reusable elements using frameworks like React, but judging this is the only page that requires such dynamic rendering, the improved efficiency would not justify the cost.

Another example of dynamic rendering is in selecting the ways to upload. The user can either choose to upload a link or a PDF by selecting from the dropdown menu. The JS element will then change the requirement validation of the form accordingly.

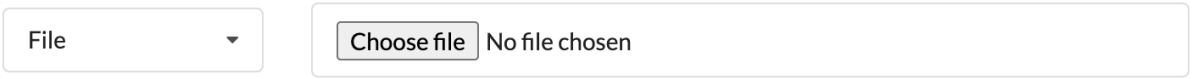

Figure 1: Upload with File

Figure 2: Upload with URL

In `codestudy/templates/codestudy/add-paper.html`:

```
...
uploadOptionsSelect
    .dropdown({
        onChange: function (value, text, $selectedItem) {
            uploadOption = Number(value);
            if (uploadOption === uploadOptionsEnum.file) {
                            // Change the element displayed on the screen
                pdf.removeAttr('hidden');
                link.attr('hidden', 'hidden');
                            // Update form validation standard
                form.form({
                    fields:
```

```
                    {
                        title: 'empty',
                        description: 'empty',
                        pdf: 'empty',
                    }
            });
        } else if (uploadOption === uploadOptionsEnum.link) {
                        // Change the element displayed on the screen
            pdf.attr('hidden', 'hidden');
            link.removeAttr('hidden');
                        // Update form validation standard
            form.form({
                fields:
                    {
                        title: 'empty',
                        description: 'empty',
                        link: 'empty',
                    }
            });
        } else {
                        // Assertion to notify a potential source of bug
            console.assert(false, "Upload Option does not match any known type");
        }
    }
})
;
...
```

Part 2: Storage and Database

The storage was implemented in AWS S3, and database uses PostgresSQL build in on Heroku. This illustrates my technical ability of using cloud database.

There are primarily three parts that enabled the cloud storage — provisioning and configuring an S3 bucket, integrating the storage into Django models, and optimisation by allowing the user to directly upload into the s3 bucket without having to go through the server.

The bucket is called `codestudy` — the name of the application and an IAM user is created in AWS with the following policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "s3:ListBucket",
            "Resource": "arn:aws:s3:::codestudy"
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:DeleteObject"
            ],
            "Resource": "arn:aws:s3:::codestudy/*"
        }
```

```
    ]
}
```

Because the browser needs to upload paper into the database, I have to configure it so that it allows for cross-origin request that is normally blocked for security reasons. It access from the `codestudy` url and `localhost` for development purposes.

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "PUT",
            "POST",
            "HEAD",
            "DELETE"
        ],
        "AllowedOrigins": [
            "https://codestudy.herokuapp.com",
            "http://localhost:8000"
        ],
        "ExposeHeaders": []
    },
    {
        "AllowedHeaders": [],
        "AllowedMethods": [
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": []
    }
]
```

The S3 database requires API keys and secrete, but because I intend to have the application open source, I cannot direly put the keys in plain text in the source code. Therefore, I have put them in the environmental variable with a file `.env` which is untracked by git and separately uploaded onto heroku.

In `main/settings.py`:

```
import dotenv

dotenv_file = os.path.join(BASE_DIR, ".env")
if os.path.isfile(dotenv_file):
    dotenv.load_dotenv(dotenv_file)
...
AWS_ACCESS_KEY_ID = os.getenv('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.getenv('AWS_SECRET_ACCESS_KEY')
```

In `.env`:

```
AWS_ACCESS_KEY_ID=...
AWS_SECRET_ACCESS_KEY=...
```

These demonstrate my technical ability to use cloud storage effectively.

The integration in the Django model is achieved through using the (buggy) `django-storage` library. Because the storage library does not support the Hong Kong server, I have to use a lower level library `boto3` to sign the object urls.

In `main/storage_backends.py`:

```
from abc import ABC
from storages.backends.s3boto3 import S3Boto3Storage
from utils import generate_presigned_url


class MediaStorage(S3Boto3Storage, ABC):
    file_overwrite = False

    def url(self, name, parameters=None, expire=None, http_method=None):
        # Not sure what http_method will give, so default to https in all cases (there's
            ↪ not many reasons to use http).
        if expire is None:
            expire = 3600
        response = generate_presigned_url(name, 600)
        return response
```

In `utils.py`:

```
from django.conf import settings
import boto3
import botocore.client

s3_client = boto3.client('s3',
                         aws_access_key_id=settings.AWS_ACCESS_KEY_ID,
                         aws_secret_access_key=settings.AWS_SECRET_ACCESS_KEY,
                         config=botocore.client.Config(s3={'addressing_style': 'virtual'
                             ↪ }, signature_version='s3v4'),
                         region_name=settings.AWS_S3_REGION_NAME)


def generate_presigned_url(object_key, expiration):
    presigned_url = s3_client.generate_presigned_url('get_object',
                                                     Params={'Bucket': settings.
                                                         ↪ AWS_STORAGE_BUCKET_NAME,
                                                             'Key': object_key},
                                                     ExpiresIn=expiration)
    # presigned_url = fix_hk_s3_url(presigned_url)
    return presigned_url
```

In the previous code example, I have overridden the default general url function to use a lower level function provided by `boto3` and wrapped in `utils.py`.

JS in browser allows hte user to upload to s3 directly, greatly imporving the speed of the website.

In `codestudy/templates/codestudy/add-paper.html`:

```
...
<button type="button" class="ui␣fluid␣large␣teal␣button" id="fake-submit" onclick="fs
    ↪ ()">Add</button>
<!-- fs stands for fake submit, btw -->
...
<script>
    ...
    async function fs() {
        const submit = $('#submit');
```

```
        if (uploaded || !form.form('is valid') || uploadOption === uploadOptionsEnum.
            ↪ link) {
            submit.click();
            return;
        }
        $('#fake-submit').css('display', 'none');
        bar.css('display', 'block');
        const s3Data = await getS3Data(pdf.val().replace(/C:\\fakepath\\/i, ''));
        let fileName = await uploadFile(pdf.prop('files')[0], s3Data);
        $('#pdf-key').val(fileName);
        pdf.val(null);
        form.form({});
        uploaded = true;
        submit.click();
    }

    /** This gets the presigned data from the server in order for the browser to
        ↪ upload, and returns a Promise
     * of the data.
     * @param {string} fileName The file name of the file to be uploaded. Used as part
         ↪  of the presigned signature.
     * @return {Promise<Object>} The presigned S3 data.
     */
    function getS3Data(fileName) {
        return new Promise((resolve, reject) => {
            const url = "{% url 'codestudy:presign-s3' %}?" + $.param({file_name:
                ↪ fileName});
            $.get(
                url,
                data => resolve(data)
            )
        });
    }

    /**
     * Upload the file given file to the S3 bucket.
     * @param {File} file The file to be uploaded.
     * @param {Object} s3Data The presigned S3 data.
     * @return {Promise<string>} The object key in the S3 bucket.
     */
    function uploadFile(file, s3Data) {
        return new Promise(resolve => {
            postData = ... // adds the post data as required by s3 specifications.
            xhr.onreadystatechange = () => {
                if (xhr.readyState === 4 && (xhr.status === 200 || xhr.status === 204))
                    ↪ {
                    resolve(s3Data.fields.key);
                }
            };
            xhr.send(postData);
        });

    }
</script>
```

In particular in `getS3Data(fileName)`, I have demonstrated my ability to code using concurrency in JS through Promise. I have also demonstrated to dynamically communicate between the front-end and the back-end.

Part 3: User Login, Permissions & Security

Logging in is achieved through Sign in with Google. I have decided not to use username and password because that leads to higher security vulnerability as now I have to handle hashing, salting and storing the user data safely myself. Sign in with Google is easier for use as well.

In `codestudy/templates/base.html`:

```
<script>
    function logout() {
        const xhr = new XMLHttpRequest();
        xhr.open('GET', '{% url 'codestudy:logout' %}');
        xhr.onload = () => {
            location.reload();
        };
        xhr.send();
    }

    function sendIdToken(idToken) {
        var xhr = new XMLHttpRequest();
        xhr.open('POST', "{% url 'codestudy:login' %}");
        xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
        xhr.onload = function () {
            const response = JSON.parse(xhr.responseText);
            location.reload();
        };
        xhr.send('id-token=' + idToken + '&csrfmiddlewaretoken=' + '{{ csrf_token }}');

    }

    gapi.load('auth2', function () {
        // Retrieve the singleton for the GoogleAuth library and set up the client.
        const auth2 = gapi.auth2.init({
            client_id: '330093561618-bc6pprilkfqi5sc6oibm376psnfe9tub.apps.
                ↪ googleusercontent.com',
            cookiepolicy: 'single_host_origin',
            // Request scopes in addition to 'profile' and 'email'
            //scope: 'additional_scope'
        });
        const element = document.getElementById('loginBtn');
        auth2.attachClickHandler(element, {},
            function (googleUser) {
                const id_token = googleUser.getAuthResponse().id_token;
                sendIdToken(id_token)
            }, function (error) {
                console.error(error);
            });
    });
</script>
```

In `codestudy/views.py`:

```
...
def login(request):
    """
    This is a Django handler function that verifies the login details according to the
        ↪ token gained by signing in with
    Google. Designed to be interacted programmatically
    :param request: The Django Request object
    :return: The login status in JSON format
```

```
    """
    if request.method == 'POST':
        id_token = request.POST['id-token']
        try:
            # Specify the CLIENT_ID of the app that accesses the backend:
            id_info = google.oauth2.id_token.verify_oauth2_token(id_token,
                                                                 google.auth.transport.requests
                                                                     ↪ .Request(),
                                                                 settings.G_CLIENT_ID)

            user = User.objects.get_or_create(pk=id_info['sub'])[0]
            user.email = id_info['email']
            user.name = id_info['name']
            user.given_name = id_info['given_name']
            user.family_name = id_info['family_name']
            user.save()
            request.session['sub'] = user.pk
            success = True
        except ValueError:
            # Invalid token
            success = False
    else:
        raise Http404()
    return JsonResponse({
        'success': success
    })
...
def logout(request):
    """
    This is a Django handler function that logs the current user out. Designed to be
        ↪ accessed programmatically.
    :param request:
    :return:
    """
    try:
        del request.session['sub']
        success = True
    except KeyError:
        success = False
    return JsonResponse({'success': success})
...
```

The logout function visits the log out page, which clears the This demonstrates my technical ability to integrate existing framework into my own app. The login function verifies the login details. For security, the data is accepted through post because the django post function requires a Cross-site request forgery token, which prevents arbitrary access to the database. In addition, I followed the documentation and used the id-token, which is signed with the secrete key of the Google API. This demonstrates my ability to handle user authentication well.

In order to prevent unauthorized access, not only is the corresponding url hidden from view, but direct access to the urls will return a 404 page as if the pages don't exist.

In `codestudy/views.py`:

```
def add_paper(request):
    if get_user(request) and get_user(request).can_edit:
        ...
    else:
        raise Http404()
```

Part 4: PDF processing

I need to complete two tasks with the PDF: extract the first page as a PNG image, and extract the plain text from the PDF to be used by the search engine, which is using the algorithm of optimization by memorization. Thankfully I was able to find external high-level libraries: `pdf2image` and `textract`. All I needed to do was to wrap the library so that I can integrate it easily into the website.

In `codestudy/pdf_processor.py`:

```
import pdf2image
import os
from django.contrib.staticfiles.storage import staticfiles_storage
import re
import logging
import uuid
import textract
import string


def pdf_to_png_and_save(paper):
    """
    Takes a paper with the pdf file, capture the first page as a png and saves it to
        ↪ the paper object.
    :param paper: The Paper object to be processed.
    :return: None
    """
    # noinspection PyBroadException
    try:
        paper.pdf.seek(0, 0)
        pdf_byte = paper.pdf.read()
        image = pdf2image.convert_from_bytes(pdf_byte, last_page=1, dpi=100)[0]
        png_name = str(uuid.uuid4())
        image.save(png_name, 'PNG')
        with open(png_name, 'rb') as f:
            paper.png.save(os.path.basename(paper.pdf.name).replace('.pdf', '') + '.png
                ↪ ', f)
        os.remove(png_name)
        paper.pdf.close()
    except:
        logging.exception('exception␣occurred␣converting␣pdf␣to␣png')
        with open(staticfiles_storage.path('failed/failed.png'), 'rb') as f:
            paper.png.save('failed.png', f)


def get_text(paper):
    """
    Extract the text content in the pdf, turn all characters to lowercase and replace
        ↪ each continuous whitespace with
    one space.
    :param paper: The Paper object to be processed.
    :return: The plain text in the paper.
    """
    id1 = str(uuid.uuid4())
    _RE_COMBINE_WHITESPACE = re.compile(r"\s+")
    paper.pdf.seek(0, 0)
    with open(id1, 'wb') as of:
        of.write(paper.pdf.read())
    paper.pdf.close()
    text = textract.process(id1, extension='pdf').decode('utf-8')
    os.remove(id1)
```

```
    # Replace all white spaces with space:
    # https://stackoverflow.com/questions/2077897/substitute-multiple-whitespace-with-
        ↪ single-whitespace-in-python
    # Strip punctuation: https://stackoverflow.com/questions/265960/best-way-to-strip-
        ↪ punctuation-from-a-string
    return _RE_COMBINE_WHITESPACE.sub("␣", text).lower().translate(str.maketrans('', '
        ↪ ', string.punctuation))
```

In the `pdf_to_png_and_save` function, I used fall backs and uses a `filed.png` file to be stored in the database, demonstrating my ability to **gracefully fall back to increase responsiveness**. I have also used lower level API such as `.seek(0,0)` to copy the file from Django file field onto the hard drive where `pdf2image` can access.

In the `get_text` function, not only did I extract the text, but also did preprocessing to replace each continuous span of white spaces with a single space to aid with finding the Levenshtein distance later during the search.

Part 5: The Search Engine

The search engine is quite a naive one, for the pseudocode of the algorithm, please see the criterion B document.

Part 6: Validation, Error handling and Fallbacks

Because of the unpredictability of the user input of a web app, when the website fails, it should give as much helpful information to the user as possible and the crash should affect as little part of the program as possible.

The first action is to validate the user input in any form.

In `codestudy/templates/add-paper.html`:

```
<script>
    const uploadOptionsSelect = $('#upload-option');
    const uploadOptionsEnum = Object.freeze({
        'file': 1,
        'link': 2,
    });
    let uploadOption = uploadOptionsEnum.file;
    uploadOptionsSelect
        .dropdown({
            onChange: function (value, text, $selectedItem) {
                uploadOption = Number(value);
                if (uploadOption === uploadOptionsEnum.file) {
                    pdf.removeAttr('hidden');
                    link.attr('hidden', 'hidden');
                    form.form({
                        fields:
                            {
                                title: 'empty',
                                description: 'empty',
                                pdf: 'empty',
                            }
                    });
                } else if (uploadOption === uploadOptionsEnum.link) {
                    pdf.attr('hidden', 'hidden');
                    link.removeAttr('hidden');
                    form.form({
                        fields:
                            {
```

```
                        title: 'empty',
                        description: 'empty',
                        link: 'empty',
                    }
                });
            } else {
                console.assert(false, "Upload Option does not match any known type");
            }
        }
    })
;
uploadOptionsSelect.dropdown('set selected', uploadOptionsEnum.file);
```

This JS code in combination with that from Semantic UI verifies the input are not empty. Note that it is not necessary for me to sanitizes the input because Django automatically and by default escape strings so that code injection is impossible to my best knowledge.

Because all the papers are rendered in one page using the Django template language, if a single rendering raises an exception, the entire page will fail to render. So I have added a few precaution to prevent that.

In `codestudy/models.py`:

```
class Paper(models.Model):
    ...

    png = models.FileField(default='failed.png')
    pdf = models.FileField(default='failed.pdf')
```

In case that fails,

in `codestudy/templates/codestudy/results.html`:

```
...
{% if paper.png %}
    <img src="{{ paper.png.url }}" alt="{{ paper.title }}">
{% endif %}
...
{% if paper.link %}
    <a class="header" href="{{ paper.link }}">{{ paper.title }}</a>
{% elif paper.pdf %}
    <a class="header" href="{{ paper.pdf.url }}">{{ paper.title }}</a>
{% endif %}
```

In the above code, the template language checks if the files exists before trying to get the url from the model so that it would not show an exception. I have tested this by raising exceptions in various places to see if it crashes more than it should.

This demonstrate my technical ability to **handler errors gracefully**.

In addition, the 500 error also implement multiple level of fall backs.

In `codestudy/views.py`:

```
def handler500(request):
    try:
        context = get_base_context(request)
        context.update({
            'message': {
                'title': '500 Server Error',
                'description': 'Something went wrong. Please try again later.'
            }
        })
```

```
        return render(request, 'codestudy/base.html', context=context, status=500)
    except:
        return render(request, 'codestudy/500.html', status=500)
```

This function is hooked to the server error handler in Django. The first thing it tires is to have some dynamic element such as the header bar, if it still fails (which sometimes happens if the database is corrupt), it would return a static 500 page with all the links hardcoded.