

Riga Technical University

Third Practical Exercise : Full Stack Web Development

Telecommunication Software

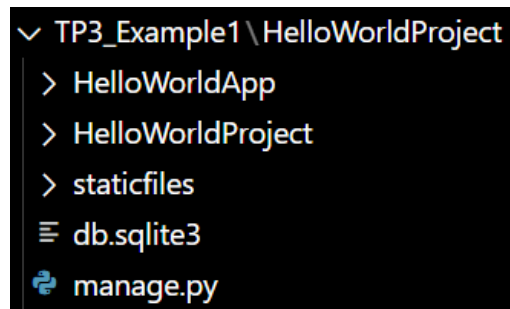
Baptiste CUIGNEZ
2023/2024

Table of Contents

I)	Structure of code and the folders.....	2
a)	‘HelloWorldApp’ folder	2
b)	‘HelloWorldProject’ folder	4
c)	‘staticfiles’ folder	5
d)	Proper operations files.....	5
II)	Example 1	6
III)	Example 2	9
IV)	Example 3	14
V)	Example 4	18
a)	Bad Request response.....	18
b)	Not Found Response	18
c)	Internal Server Error	19
d)	JSON response	19
e)	Streaming response	20
f)	Files uploading	21
g)	Downloading files	23
VI)	Conclusion	25

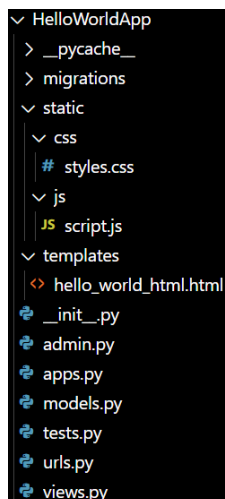
l) Structure of code and the folders

To simplify the code, all the examples are called 'HelloWorldProject'. They are just in different folders. There are 3 mains folders : 'HelloWorldApp', 'HelloWorldProject' and 'staticfiles'. There is also 2 files for the proper operation of our website : 'db.sqlite3' and 'manage.py'



a) 'HelloWorldApp' folder

In this folder we can find other folder and files containing our website backend and frontend.



In this folder we can find :

- '__pycache__' folder : contain all our compiled website and all Python module associated to our website. This folder only contains Binary files for Python (.pyc).
- 'migration' folder : where the framework stores instructions on how to modify the database schema in response to changes in your models. It

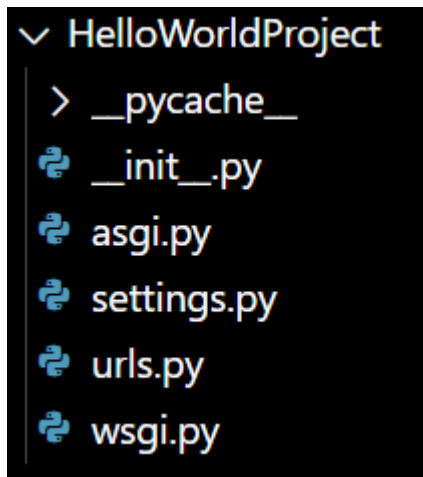
plays a key role in the versioning and evolution of your database structure.

- 'static' folder : where all the statics files our server will use when it will runtime. These files will not change during all the runtime. It mainly includes the JavaScript files and the CSS files.
- 'template' folder : where the HTML files our website will use is stored.
- '__init__.py' file : in this file we execute all the initialisations of all the modules we use for our website when we start our website.
- 'admin.py' file : in this file we configure how the Admin interface will be for our website (ourwebsite.com/admin).
- 'apps.py' file : in this file we can configure options about our website.
- 'models.py' file : in this file we define how our website's database will be : for example how to store some files we can upload on our website.
- 'tests.py' file : in this file we can design some tests for our website.
- 'urls.py' file : we store the URLs patterns of our website. It's here that we link our functions and the URL associated. This file is crucial for defining the structure of our web application, mapping URLs to views, and creating a logical organization for the project.
- 'views.py' file : in this file we can find all our function for the website (all the backend of the website). Views are a fundamental part of our project. It's responsible for handling user requests, processing data, and returning appropriate responses. They connect the URL patterns defined in urls.py to the actual logic that drives our web application.

All these files and folders are the kernel of our website. They contain the backend of our project (database management, function, ...) and also all the frontend (HTML page, CSS visualization and JS scripts).

b) 'HelloWorldProject' folder

In this folder we can find more further back option and configuration of our website. If the 'HelloWorldApp' folder was the engine of the webpage the 'HelloWorldProject' will be engine of how the webpage is manage on the server.



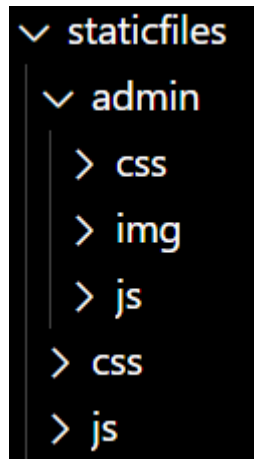
In this folder we can find:

- '__pycache__' folder : as in the app, this folder contains all the binaries files of the server.
- '__init.py__' file : contain all the initialisation of the server.
- 'asgi.py' file : contains all the configuration for the well working of all asynchronous features of our webpage.
- 'settings.py' file : contains the mains setting of our server and how manage the staticsfiles database.
- 'urls.py' file : contains the link between all the URLs of our webpage and the URL manage by the server. In this file we include the urls.py file of our webpage.
- 'wsgi.py' file : this file provides us the web server gateways interface.

With all these folders and files we can well manage our server and our webpages on the server.

c) 'staticfiles' folder

In this file we will find all the static files used while our server is running. We will find mainly 3 folders. One for the admin page. With all the admin pages html codes, CSS, JS and eventually the pictures displayed on the admin pages. One for the CSS files of our webpages and one for the JS of the webpages.



d) Proper operations files

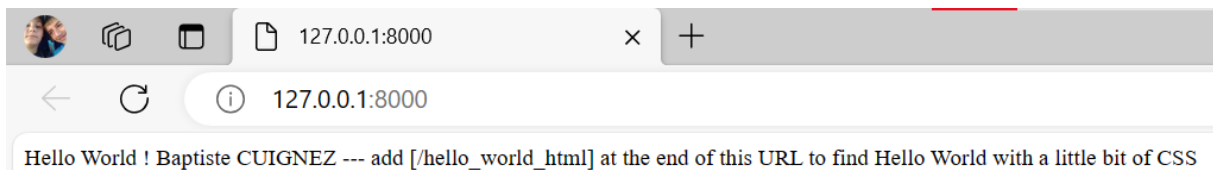
We will find two additional files in the tree structure of our project : db.sqlite3 which is the database of our website : if we have some messages to store, it will be in there. We can also find the 'manage.py' file, we use it to manage our server : make migrations, update the database, launch the server...

II) Example 1

In the first example we have to create a simple webpage with HTTP response. For that, we only have to create a function with the built-in Django function called 'HttpResponse'. This function only return us a single string to our page. We have to write this function in the 'views.py' file and in the 'urls.py' file we have to link this function to the URL

```
def hello_world(request):  
    return HttpResponse('Hello World ! Baptiste CUIGNEZ--- add [/hello_world_html],at the end of this URL to find
```

```
from django.urls import path  
from .views import *  
  
urlpatterns = [  
    path('', hello_world, name='hello_world'),
```



Because we want something more beautiful for the user we can add some CSS on our webpage. For that we will use the function 'render' of Django. This function will display an HTML page when it is called.

```
def hello_world_html(request):  
    return render(request, 'hello_world_html.html')
```

This function allow us to link an HTML page to our response and moreover we can add some CSS. We can also add the link to the 'urls.py' to add another branch to our website.

```

from django.urls import path
from .views import *
urlpatterns = [
    path('', hello_world, name='hello_world'),
    path('hello_world_html/', hello_world_html, name='hello_world_html'),
]

```

```

{% load static %}

<DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Hello, World!</title>
    <link rel="stylesheet" type="text/css" href="{% static 'css/styles.css' %}">
</head>
<body>
    <div class="container">
        <h1>Hello, World!</h1>
        <p>Baptiste CUIGNEZ</p>
        <button id="counterButton">Do not click me!</button>
        <p id="count">The button is currently under maintenance</p>
    </div>
    <script src="{% static 'js/scripts.js' %}"></script>
</body>
</html>

```

```

document.addEventListener('DOMContentLoaded', function () {

    // Counter functionality
    var counterButton = document.getElementById('counterButton');
    var countElement = document.getElementById('count');
    var counter = 0;

    counterButton.addEventListener('click', function () {
        counter++;
        countElement.innerText = counter;
    });

    console.log('Hello, World! This is JavaScript.');
```

```

body {
    font-family: 'Arial', sans-serif;
    background-color: #f0f0f0;
    margin: 0;
    padding: 0;
}

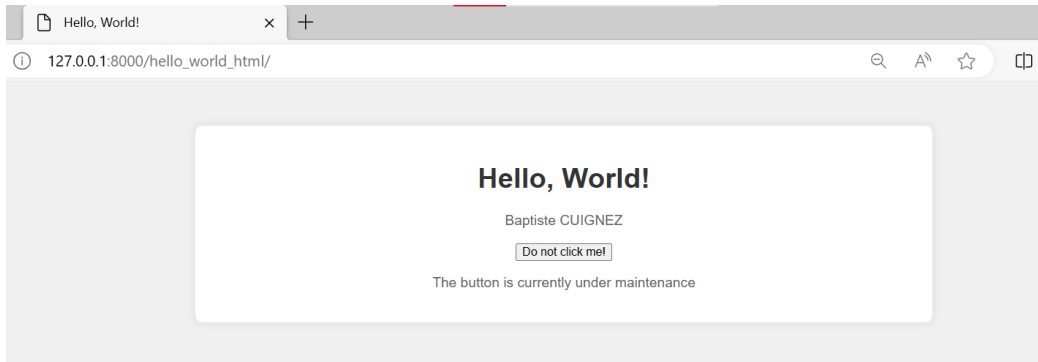
.container {
    max-width: 800px;
    margin: 50px auto;
    text-align: center;
    background-color: #ffffff;
    padding: 20px;
    border-radius: 8px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1 {
    color: #333;
}

p {
    color: #666;
}

```

The first screenshot, show use the 'urls.py' file where we linked our function to the URL. The three following screenshots show the HTML, CSS and JS code. In the HTML code, we have to add '{% load static %}' to tell our compiler that the HTML code and the CSS/JS code are not on the same folder. With this line the compiler knows that it have to find the 'static' folder to find the CSS and the JS associated with the HTML.



The render of this webpage seems more simple for a user to see and to use.

With this first example we learn how to send an simple HTTP response and how to display an HTML page with CSS and JS associated. We also learn to link the different URLs with different responses.

III) Example 2

In this example we have to create a improve Hello world page with HTML page and we have to use an MVT (Model View Template) to design a pattern. A MVT is simply a web page where the user interact with the server. For example an MVT webpage can be simply a page where when the user click on a button the page will display a new message. For my example I tried to make a webpage which will display a bunch of messages store into a database. Moreover I will like this page to another where we can add messages to the database et so add the number of messages in the first webpage.

In order to do that we will use mainly 4 files into our folders. We first have to code a 'model.py' file to manage our database. In this file we will imagine how our data will be stored.

```
from django.db import models

class Message(models.Model):
    text = models.TextField()

    def __str__(self):
        return self.text
```

For us it will only store the text we have to display in our webpage. Then we have to create a 'form.py' file. This file allows us to upload data from our website to the server (our computer for now).

```
from django import forms
from .models import Message

class MessageForm(forms.ModelForm):
    class Meta:
        model = Message
        fields = ['text']
```

In this form we let the compiler knows that we will have to upload text and we have to store it as described in the 'model.py' file. Now that we have our database and the way to upload to our database we can create the views of our website.

```
from django.shortcuts import render, redirect
from .models import Message
from .forms import MessageForm

def message_list(request):
    messages = Message.objects.all()
    return render(request, 'message_list.html', {'messages': messages})

def add_message(request):
    if request.method == 'POST':
        form = MessageForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('message_list')
    else:
        form = MessageForm()

    return render(request, 'add_message.html', {'form': form})
```

For that we have two views : one for the display of the messages stored in the database and the other for adding messages to the database. In the render parameters we have to specify that for the first one we want to display the messages and for the second we want to use the form. We can now link our views to the URLs of our website. For that we just have to modify the 'urls.py' file as following.

```
from django.urls import path
from .views import message_list, add_message

urlpatterns = [
    path('', message_list, name='message_list'),
    path('add/', add_message, name='add_message'),
]
```

Now that our views are linked, we can create an HTML page for each views. For that we will make simple pages with only one button going to the Add message page or going back to the display message page.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Add Message</title>
</head>
<body>

  <h1>Add a Message</h1>

  <form method="post" action="">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Submit</button>
  </form>

  <a href="{% url 'message_list' %}">Back to Message List</a>

</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Message List</title>
</head>
<body>

  <h1>Message List</h1>

  {% for message in messages %}
  |   <p>{{ message.text }}</p>
  {% endfor %}

  <a href="{% url 'add_message' %}">Add a Message</a>

</body>
</html>
```

Now that our website is finished we have to create the database on our server. For that we just have to execute in our terminal the function 'manage.py makemigrations'. This command will create all parameters needed to well run the server (database shape, server parameters) but this modifications are not applied to the server. There are just stored, ready to be applied with the command 'manage.py migrate'. Now we have a new file in our folder : the 'db.sqlite3', this is our database. Our database look like this :

name
HelloWorldApp_message
auth_group
auth_group_permissions
auth_permission
auth_user
auth_user_groups
auth_user_user_permissions
django_admin_log
django_content_type
django_migrations
django_session
sqlite_sequence

This is obtain with the SQL command 'SELECT tbl_name as name FROM sqlite_master WHERE type='table' Order by tbl_name ASC' Our database contain 12 tables in total. The table we create earlier to store the messages is 'HelloWorldApp_message' and contain for now only 2 messages which are my tests (obtain with the command 'SELECT * FROM HelloWorldApp_message;'):

id	text
1	First Test Message
2	It's working :)

Now we can launch our server and look our webpage.

Message List

First Test Message

It's working :)

[Add a Message](#)

The webpage is displaying the two messages in our database. And when we click on the 'Add a Message' link we have a page to add a message :

Add a Message

Text:

This is the example for the report

Submit

[Back to Message List](#)

When we click on the Submit button and then we go back on our main page we will see the new message displayed. And this is now how our database look like:

id	text
1	First Test Message
2	It's working :)
3	This is the example for the report

The new message is well stored in it.

With this second example we learn how to create an interaction between the user and the server using database, models and form.

IV) Example 3

The third example aim to create an instant-message like website where we can add as input the name of the sender and the receiver and the text to send. And another function where we can see the last 20 messages receive by someone.

For that I've design 6 webpages within 4 URLs : one for the welcome page when we lunch the server, one for the choice (send message or view message), two for the sending process (one to send the message and one to agree the sending) and two for the viewing process (one to select the recipient and on to view the messages)

To store the message we have to design a model for our database as seen in the previous example. For the storage the of the message we will store 4 information : the sender, the recipient, the content ant the timestamp to well displays the last 20 messages.

```
from django.db import models

class Message(models.Model):
    sender = models.CharField(max_length=50, default="DefaultSender")
    recipient = models.CharField(max_length=50, default='DefaultValue')
    content = models.TextField()
    timestamp = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"{self.sender} to {self.recipient}: {self.content}"
```

We had to add a 'DefaultSender' and 'DefaultValue' for the sender and the recipient, otherwise the compiler would not create the model and the associated database.

Because we have 4 URLs, we have to create 4 associated views. In some views we will displays two webpages. The two first views are the welcome page and the redirect page. In the first one when we click on the link we will be redirected to the second one and in the second one the two links provided redirect the user into the view page or the send page.

```
def hello_world_html(request):
    return render(request, 'hello_world.html')

def messages_home(request):
    return render(request, 'messages/messages_home.html')
```

This is my WEBPAGE about the messages exchanges in Django

[Join Messages](#)

Messages Home

Choose an option:

- [Send Message](#)
- [View Messages](#)

These screenshots represent the welcome page and the choice page displayed.

For the send page we create it own views with the two pages : the form one and the success one. We can join this page by clicking the first link on the choice page.

```
def submit_message(request):
    if request.method == 'POST':
        sender = request.POST['sender']
        recipient = request.POST['recipient']
        content = request.POST['content']

        Message.objects.create(sender=sender, recipient=recipient, content=content)

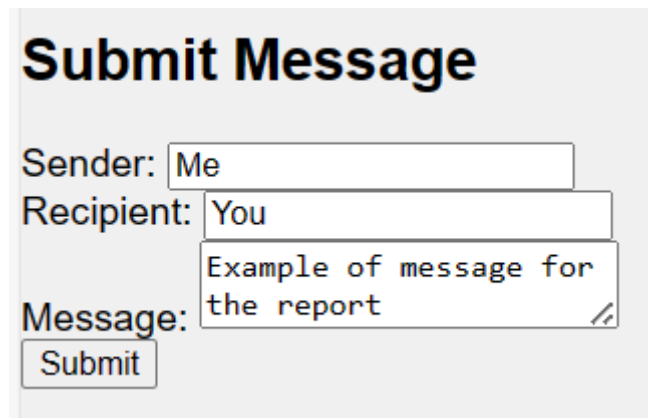
        return render(request, 'messages/submit_success.html')

    return render(request, 'messages/submit_message.html')
```

```
<body>
    <h2>Submit Message</h2>
    <form method="post" action="{% url 'submit_message' %}">
        {% csrf_token %}
        Sender: <input type="text" name="sender" required><br>
        Recipient: <input type="text" name="recipient" required><br>
        Message: <textarea name="content" required></textarea><br>
        <button type="submit">Submit</button>
    </form>
</body>
```

With this view we create grab the information send by the HTML page (within the form diamonds) and we associate the POST request to the object that we will store into our database with the 'Message.object.create' method. When the POST request is received and well process by our function the browser will

display the Success page (with the same URL). This page contain a button to go back to viewing page or the choice page.



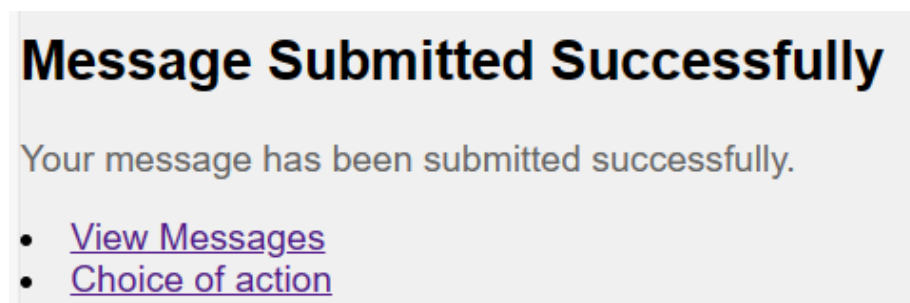
Submit Message

Sender:

Recipient:

Message:

The send page



Message Submitted Successfully

Your message has been submitted successfully.

- [View Messages](#)
- [Choice of action](#)

The success page

Now that we have a sending page we can create a viewing page. For that we have a function doing that in our 'views.py' file.

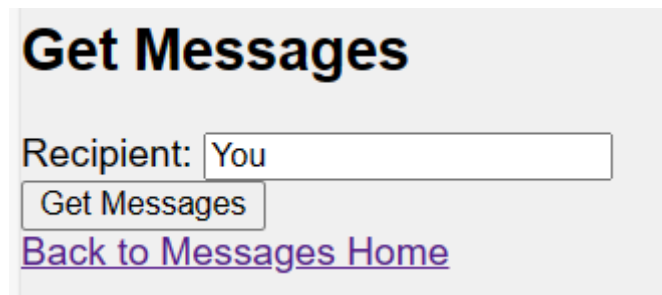
```
def get_messages(request):
    if request.method == 'POST':
        recipient = request.POST['recipient']
        messages = Message.objects.filter(recipient=recipient).order_by('-timestamp')[:20]
        return render(request, 'messages/message_list.html', {'messages': messages})

    return render(request, 'messages/get_messages.html')
```

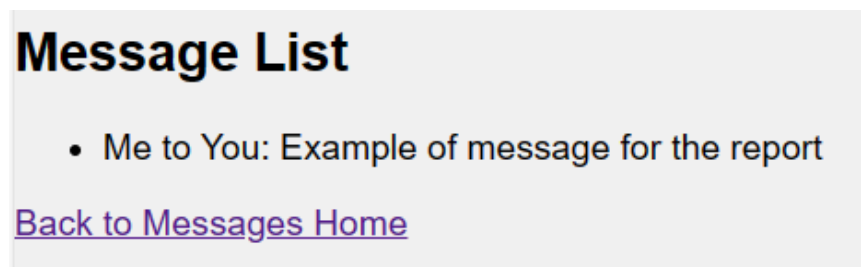
```
<h2>Get Messages</h2>
<form method="post" action="{% url 'get_messages' %}">
    {% csrf_token %}
    Recipient: <input type="text" name="recipient" required><br>
    <button type="submit">Get Messages</button>
</form>
```

As in the previous view, we create here a form on our HTML page linked to our backend function in Python. Then when we get the POST request we process to collect all the message associated to the recipient chosen. After that we display

the list message page with the parameter messages (containing all the object message receive by the recipient chosen.

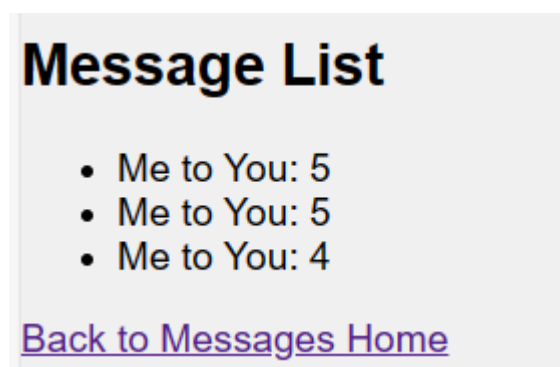


The Get page



The message list page

For the a fast test, I set the limit of message at 3 and there is the last 3 messages on the web page displayed for the user 'You' and all the messages received by 'You' according to our database :



id	recipient	sender	timestamp	content
1	You	Me	2024-01-12 19:14:14.455525	Example of message for the report
2	You	Me	2024-01-12 19:23:53.040537	1
3	You	Me	2024-01-12 19:24:03.734284	2
4	You	Me	2024-01-12 19:24:10.777823	3
5	You	Me	2024-01-12 19:24:56.171752	4
6	You	Me	2024-01-12 19:25:03.850844	5
7	You	Me	2024-01-12 19:25:17.374572	5

In this example we learn how to create more than one webpage display on the same URL and more importantly we learn how to handle POST with more than one parameters.

V) Example 4

Now that we have explored some of the more common responses in Django we can try to explore it deeper and try some new features. For that I've done 7 different responses in this final example.

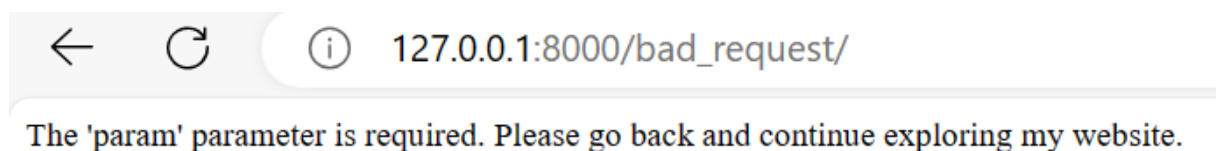
a) Bad Request response

This HTTP response is a basic one. For example we can have this response when we didn't fill a form correctly and we have a missing parameter for example. IN Django we can raise an exception and send an 'HttpResponseBadRequest' with a specific message, so the browser knows that there is a problem with it previous request and the user knows where is the problem.

```
def my_bad_request_view(request):
    if not request.GET.get('param'):
        return HttpResponseBadRequest("The 'param' parameter is required. Please go back and continue exploring my website.")

    return render(request, 'my_bad_request_view.html')
```

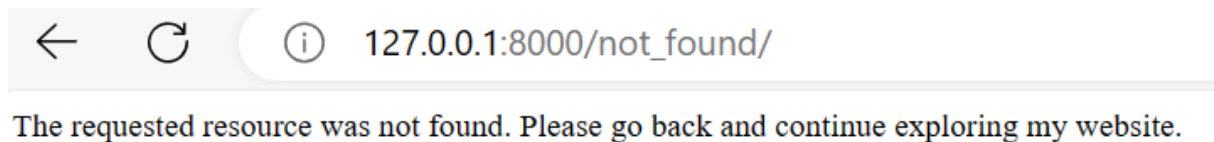
When we test if the parameter is well given and it is not, we just send a bad request response to our webpage. This response is displayed as a simple HTTP response, without HTML.



b) Not Found Response

This response is sent when the server can process a request because the item requested is not found in the database. For example we need to access the item with the id number 45 and we only have 5 items (id's between 0 and 4) so the server will send back a 'NotFoundResponse'.

```
def my_not_found_view(request):
    if True:
        return HttpResponseNotFound("The requested resource was not found. Please go back and continue exploring my website.")
```

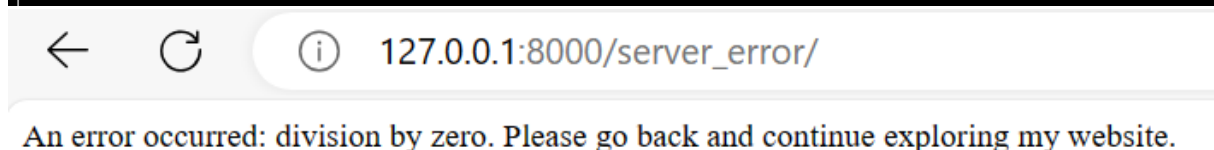


For the example we forced our server to send back a Not Found Response with the 'if True:' to see how the server will manage this kind of response. We can see that the response is a simple HTTP string send back to the browser and user.

c) Internal Server Error

This kind of response is when the server occur an error while it process the user's request. With this kind of response we can notify the user what error occur during the processing of the request. In this view we divide by 0 and that create an error for the server. Then it send back a response to the browser telling that we have a division by 0 so the request cannot be well processed.

```
def my_server_error_view(request):  
    try:  
        result = 1 / 0  
    except Exception as e:  
        return HttpResponseRedirect("An error occurred: {}. Please go back and continue exploring my website.".format(str(e)))
```



We can see that for each error, we have a feed back in the server terminal.

```
Bad Request: /bad_request/  
[13/Jan/2024 13:29:52] "GET /bad_request/ HTTP/1.1" 400 84  
Not Found: /not_found/  
[13/Jan/2024 14:22:34] "GET /not_found/ HTTP/1.1" 404 87  
Internal Server Error: /server_error/  
[13/Jan/2024 14:25:42] "GET /server_error/ HTTP/1.1" 500 86
```

Each error have a different color with the error code and another code to precise the error

d) JSON response

Sometimes when we have to send data we can send them via a JSON string. JSON is a standard in the data transmission world.

```
def my_json_view(request):
    data = {'This is': 'a test', 'Name': 'CUIGNEZ Baptiste', 'Message': 'Please go back and continue exploring my website'}
    return JsonResponse(data)
```

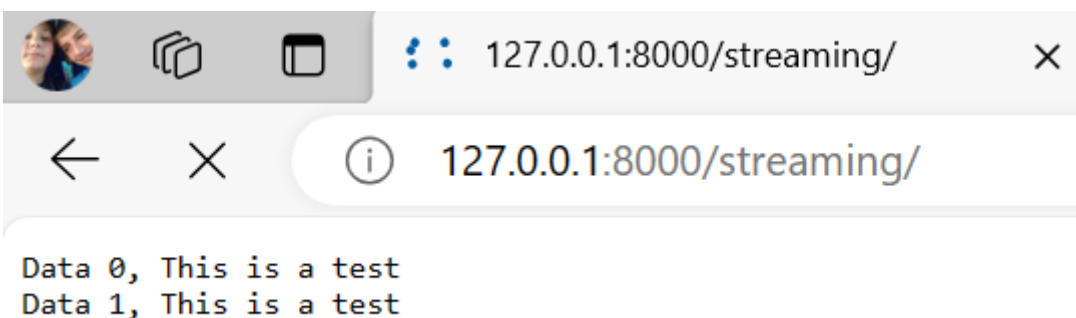


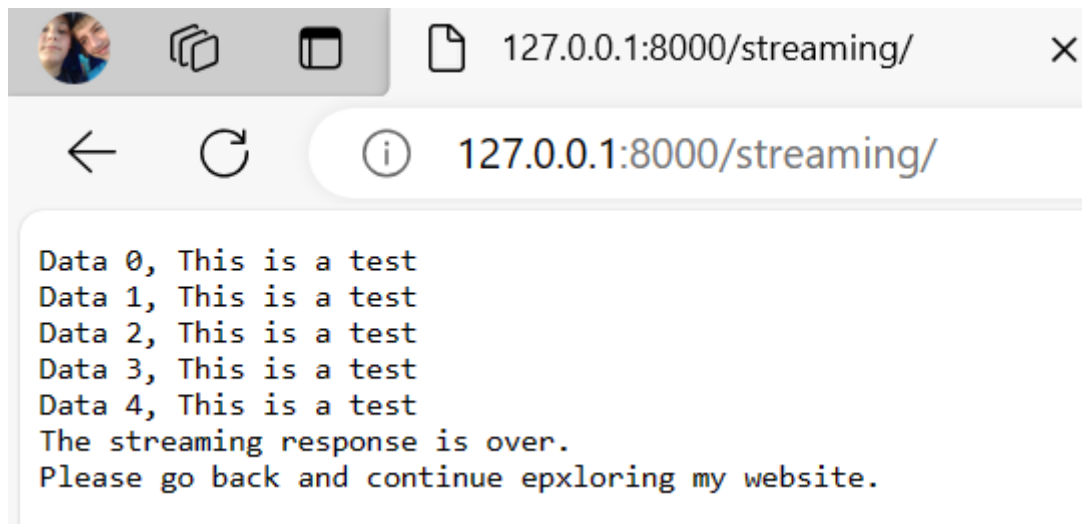
Our browser understand that it's a JSON response so it displays it as a JSON with all the characteristics.

e) Streaming response

Sometimes we need to transfer data flow without reloading the webpage. For that we can use the 'StreamingHttpResponse' in Django. This function allows us to put delay in our webpage. In our function we print 5 lines of data with a second between them.

```
def my_streaming_view(request):
    def stream_response():
        for i in range(5):
            time.sleep(1)
            yield f"Data {i}, This is a test\n"
        time.sleep(1)
        yield "The streaming response is over.\nPlease go back and continue exploring my website."
    return StreamingHttpResponse(stream_response(), content_type="text/plain")
```





We can see that the page is loading while we print the data and when all the data are printed the page is not loading anymore.

f) Files uploading

We previously seen that we can send a message to our database. Now we can send files to our database. For that we have to create a model in our database and to create a form to standardize the upload.

```
class UploadedFile(models.Model):
    description = models.CharField(max_length=255)
    file = models.FileField(upload_to='uploads/')

    def __str__(self):
        return self.description

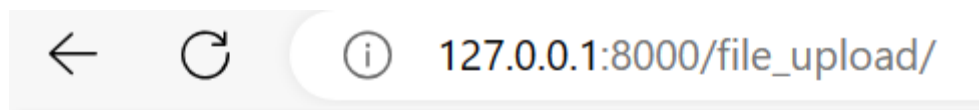
    @classmethod
    def get_all_files(cls):
        return cls.objects.all()
```

In this model we specify where the file will be stored and its name. We also add a function to get all the files for our next webpage.

```
def file_upload(request):
    if request.method == 'POST':
        form = FileUploadForm(request.POST, request.FILES)
        if form.is_valid():
            description = form.cleaned_data['description']
            file = form.cleaned_data['file']
            UploadedFile.objects.create(description=description, file=file)
            return redirect('success/')
        else:
            form = FileUploadForm()

    return render(request, 'requests/my_file_upload_view.html', {'form': form})
```

In our function we check that the form is well filled then we stored the file in our database then we redirect the user to another page to tell him that the upload was a success.

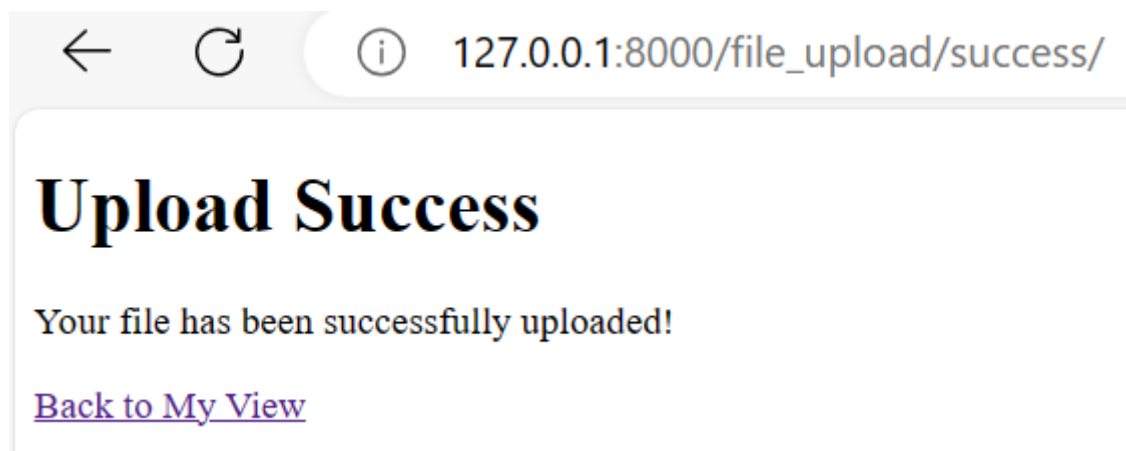


File Upload

Description:

Choose file: RTU-vizuala-id...eika-0001.png

[Back to My View](#)



After we uploaded the RTU logo in our server it will appear in the folder we decided. Here it is 'uploads' :



g) Downloading files

Now that we can upload a file it will be great if we can download them. For that we just have to create a page where we can display all the files we have in our database with the possibility of downloading them by clicking on a link. We have 2 views for that : one for the display and one for the downloading.

```
def list_uploaded_files(request):
    files = UploadedFile.get_all_files()
    return render(request, 'requests/my_list_uploaded_file_view.html', {'files': files})

def my_file_download_view(request, file_id):
    file_instance = get_object_or_404(UploadedFile, pk=file_id)

    file_path = file_instance.file.path

    response = FileResponse(open(file_path, 'rb'))

    response['Content-Disposition'] = 'attachment; filename="{0}"'.format(file_instance.file.name)

    return response
```

The first view is used when we are on the display page and the second view is used when we click on the link, the HTML page interacts with this view to allow the user to download the file.


```

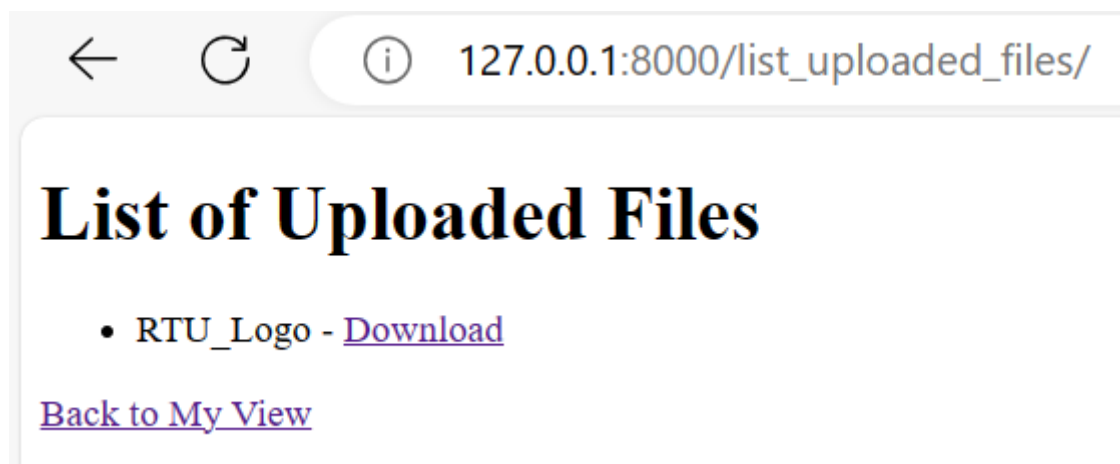
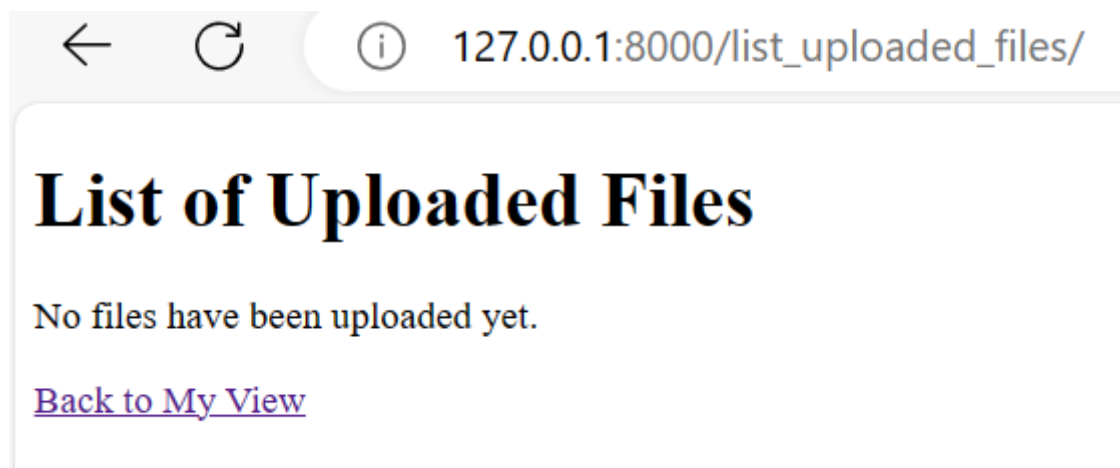
<body>

  <h1>List of Uploaded Files</h1>

  {% if files %}
    <ul>
      {% for file in files %}
        <li>
          {{ file.description }} -
          <a href="{% url 'file_download_view' file.id %}">Download</a>
        </li>
      {% endfor %}
    </ul>
  {% else %}
    <p>No files have been uploaded yet.</p>
  {% endif %}
  <a href="{% url 'homepage' %}">Back to My View</a>

```

When we have no files to display, the user knows it and when we have files to display he knows it too.



The screenshot shows a web browser window with the address bar at `127.0.0.1:8000/list_uploaded_files/`. The page title is "List of Uploaded Files". It contains a list item "RTU_Logo - Download" with a "Download" link. Below this is a "Back to My View" link. A download bar is visible on the right, showing the file "uploads_RTU-vizuala-identitate-Teika-0001.png" with a size of 126 Ko. Below the browser window, a terminal window displays the log entry: `[13/Jan/2024 14:52:43] "GET /file/1/ HTTP/1.1" 200 128556`. Below the terminal, a table shows the file details:

uploads_RTU-vizuala-identitate-Teika-0001.png	13/01/2024 14:52	Fichier PNG	126 Ko
---	------------------	-------------	--------

When we click on the Download link we have the file on our 'Download' folder on our computer and the server keep a log of what file was downloaded.

VI) Conclusion

We learn that with Django we can do all the website, starting with simple HTTP response to a more sophisticated website with more backend and more front end. We can manage a complex database using a lot function. We can also manage the different error which can occur during the process of the server.