# Datenbanken und Web-Techniken Term Paper

*Final Project: Web File Sharing Service*

TECHNISCHE UNIVERSITÄT
IN DER KULTURHAUPTSTADT EUROPAS
CHEMNITZ

Name: Calvin Liusnando
Course of Study: Web Engineering
Matriculation Number: 708955

# Table of Contents

## Utilised Technologies / Frameworks

The programming aspect of the project can be divided into 3 sections: the database, the client-side (Frontend) , and the server-side (Backend). **PostgreSQL** and **PL/pgSQL** are used for the database implementation since it is readily available as one of the database service provided by the university. Furthermore, with relational database such as PostgreSQL, data integrity can be easily maintained in the database-level by using appropriate constraints. This would not be achievable if a No-SQL database is used instead.

For the client-side implementation, **ReactJS** is chosen as the framework (library) because of its flexibility and simplicity with the component-based approach. In addition, it has a rich ecosystem and tooling options which made developing the project easier. **Create React App (CRA)** with **TypeScript** is used to create the frontend project file, with **yarn** as the package manager. With CRA, React applications can be created and maintained easily as the project configurations are generated automatically.

**NodeJS** is used for the server-side section. This ensures language uniformity (JavaScript) between client and server, which make the communication between the two sides effortless. **ExpressJS** is used to create the server-side Application Programming Interfaces (APIs) quickly and easily, while **Fetch API** is used in the client-side to make requests to the server-side APIs.

Furthermore, the project uses a lot of other packages and dependencies. **Material UI,** an open-source React component library, is used for the user interface design of the project. **react-router-dom** is used to implement routing for the application. For connection between the server-side and the database, the package **node-postgres (pg)** is utilised. The package **request** is used to send external API calls to the blocklist web service. **request** is chosen over other tools due to its simplicity, especially since it follows redirects by default and cookies can be easily enabled for the requests made, which is essential for the cookie-based SAML authentication.

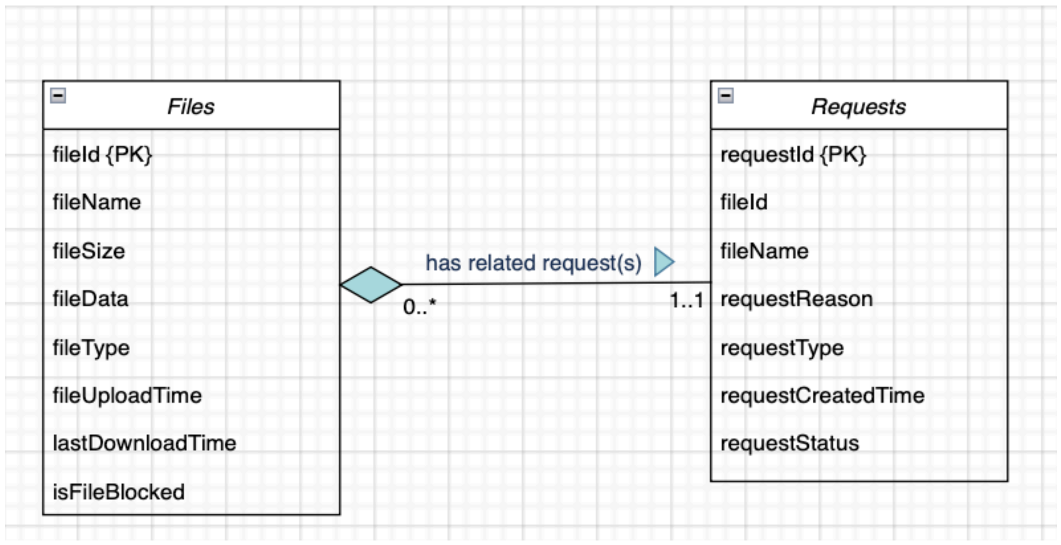
The module **crypto** is used to generate a SHA256 string for a file, **express-fileupload** is used to simplify the file uploading process, and **cheerio** is used to parse the HTML document response when sending requests for the authentication to the university's Web-Trust-Center. The modules/packages mentioned here are not exhaustive. To see the complete list of the packages and dependencies used in the project, please check the *package.json* and *yarn.lock* inside the immediate project directory and also inside the *client* folder.

Finally, to implement job scheduling for the database, a simple **shell script** is written, and the "job" is executed with **cron**, a command-line job scheduler. For more details on this, please refer to the section *6.5 Functionalities — Removal of Inactive files*.

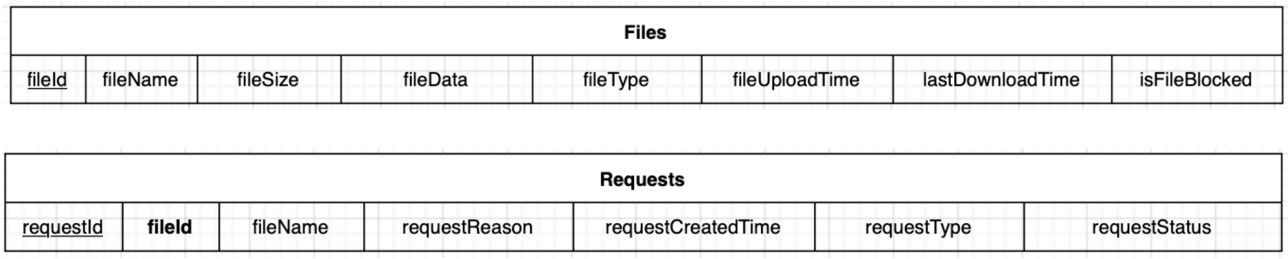
## Database Design and Implementation

The database design and implementation are kept as simple as possible for this project. Only two tables are used: *files* to store the uploaded files and *requests* to store the requests (blocking or unblocking) made for the “admin” to process. The UML diagram and the relational schema can be seen below.

UML



Relational Schema



| Files | | | | | | | |
|---|---|---|---|---|---|---|---|
| <u>fileId</u> | fileName | fileSize | fileData | fileType | fileUploadTime | lastDownloadTime | isFileBlocked |

| Requests | | | | | | |
|---|---|---|---|---|---|---|
| <u>requestId</u> | **fileId** | fileName | requestReason | requestCreatedTime | requestType | requestStatus |

<u>Notes on the database design</u>

- the table *Files* has a composite relation with the table *Requests,* since when a certain file is removed from *Files,* all of the entries related to that file would also be deleted in *Requests.* This is achieved with a BEFORE DELETE trigger function on files.
- *Files* uses a SHA256-like string corresponding to the file as *fileId*, while *Requests* uses the data type UUID as its primary key. The UUIDs are randomly generated in the server-side.

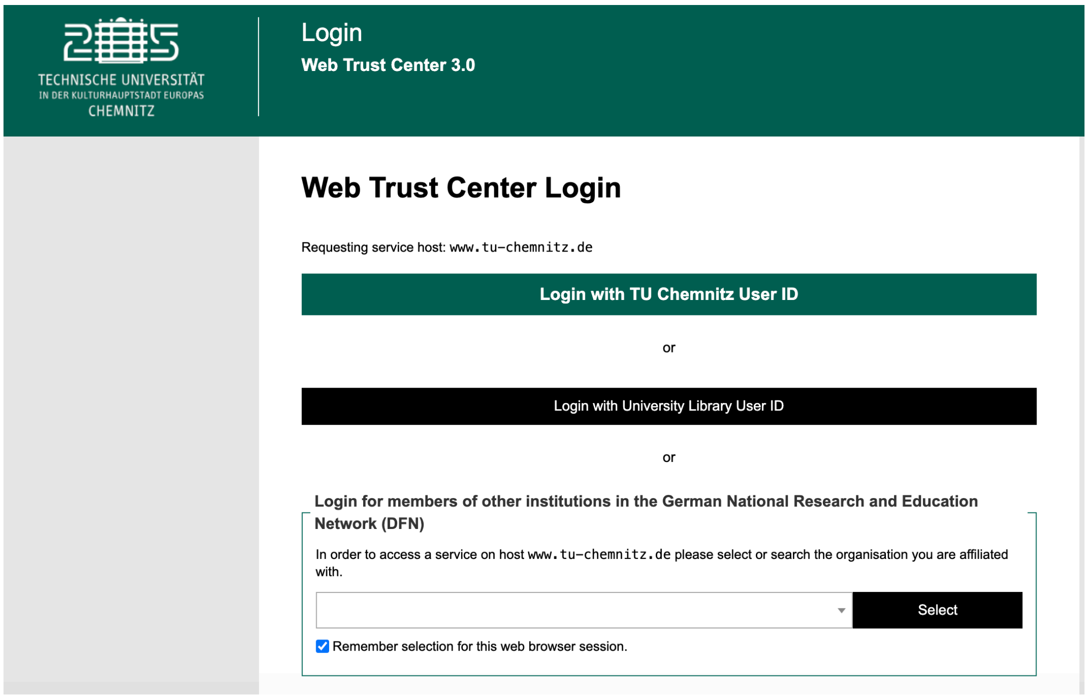
- *fileSize* is an integer with a value ranging from 0 to at most 10485760 (10 MiB). This file size validation is also done in the client-side of the project.
- *fileData* of *Files* uses the bytea data type, which allows the storage of binary strings inside the database. The format used is the default bytea Hex Format, which encodes binary data as 2 hexadecimal digits per byte.
- *fileUploadTime, lastDownloadTime* and *requestCreatedTime* use the timestamp data type (with GMT time zone), where the default value is the current database system timestamp value (CURRENT_TIMESTAMP) when the entry is inserted inside the table.
- *requestType* is an integer with two accepted values: 0 for a block-type request, and 1 for unblock-type request.
- *requestStatus* is a text with three possible values: 'Pending', 'Declined', and 'Accepted'. 'Pending' status indicates that the admin has not yet taken an action for that particular request. 'Declined' indicates that the request is rejected, and 'Accepted' means that the admin has agreed with the request made. **Only requests with 'Pending' status will be shown in the admin page.** For recording purposes, other requests are kept inside *Requests* as long as the related files are not deleted.
- **Note that a file can only have at most one request corresponding to said file**. The case where a file has both blocking and unblocking requests is not possible (and it does not make sense), and more than one requests with the same purpose would be redundant. Hence, before a request for a file is inserted into *Requests,* the old request related to that file, if there is any, would be deleted first.

# Authentication to Web-Trust-Center

One of the challenging aspects of the task is to implement an authentication to the TU Chemnitz's Web-Trust Center (WTC), which is necessary before communication with the distributed block service can be carried out. For some background information, WTC uses a cookie-based SAML (Security Assertion Markup Language) protocol and Shibboleth, a web-based single sign-on (SSO) infrastructure for the authentication process. With this procedure, there is no logout and instead the SAML session cookie is used to determine whether the user is currently authenticated or not.

For this project, the authentication process is done manually by sending appropriate HTTP requests to the related URLs. The goal is to emulate the login process that is usually done in the browser with the server, which in this project, is NodeJS. The step-by-step procedures is as follows:

1. When the node server starts, as the session cookie is empty, the authentication process is started by sending a GET request to the URL of the distributed blocklist service ("https://www.tu-chemnitz.de/informatik/DVS/blocklist/"). As the server is not yet authenticated, it will get redirected to the first WTC login URL, see the picture below.



Login
Web Trust Center 3.0

Web Trust Center Login

Requesting service host: www.tu-chemnitz.de

Login with TU Chemnitz User ID

or

Login with University Library User ID

or

Login for members of other institutions in the German National Research and Education Network (DFN)

In order to access a service on host www.tu-chemnitz.de please select or search the organisation you are affiliated with.

Select

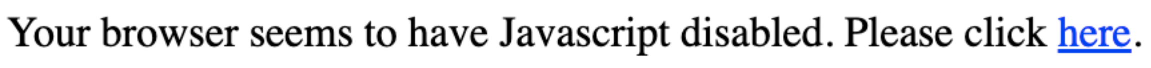
☑ Remember selection for this web browser session.

2. Now, the server need to emulate what is equivalent to clicking the "Login with TU Chemnitz User ID" button in the browser. This is done by sending a GET request to a specific URL: "https://www.tu-chemnitz.de/Shibboleth.sso/Login?" + *the string after Login%3F from the first WTC login URL* + "&entityID=https://wtc.tu-chemnitz.de/shibboleth". Note that the string after *Login%3F* would be the SAMDLS

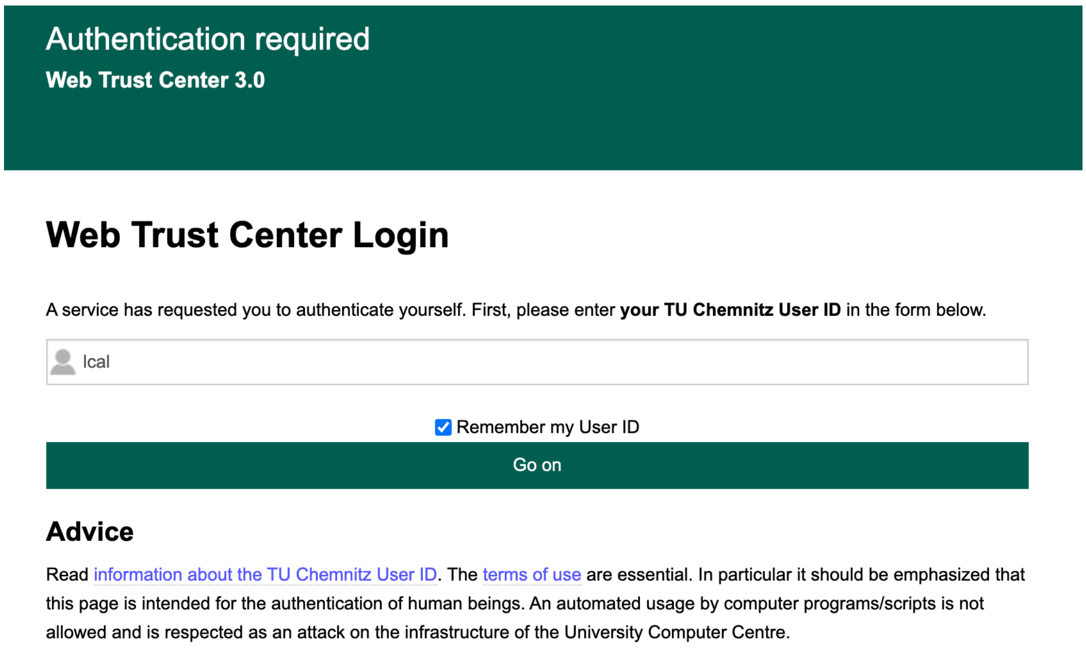
(SessionInitiator) flag and the *target* value, which is the URL to return the user after successful authentication, which in this case is the URL of the distributed blocklist service. The string (after decoded from URL-encoded format) for example would be something like this:

**SAMLDS=1&target=ss%3Amem%3A5c1cc7c3a0cdb6ae4f5f7a8db8c13e11f067554f3cc820e4770bc7d2000f535e**

3. As the authentication is done in the server-side, javascript is disabled (cannot be rendered) on the requests made. Sending a GET request to the URL in the previous step would lead the server to a page with this response body:


Your browser seems to have Javascript disabled. Please click here.

4. Now, from the previous response body, retrieve the href value and send another GET request to that URL, and the server would "reach" the page below. Get the value of the *AuthState*, which is easily retrievable by splitting the current URL. Finally, send a POST request to the next login URL, which has the format of: https://wtc.tu-chemnitz.de/krb/module.php/TUC/username.php?AuthState={authStateValue}. The POST request submits a form data with the values of *username* (your TU Chemnitz User ID) and *AuthState.*


Authentication required
Web Trust Center 3.0

Web Trust Center Login

A service has requested you to authenticate yourself. First, please enter **your TU Chemnitz User ID** in the form below.

lcal

☑ Remember my User ID

Go on

Advice

Read information about the TU Chemnitz User ID. The terms of use are essential. In particular it should be emphasized that this page is intended for the authentication of human beings. An automated usage by computer programs/scripts is not allowed and is respected as an attack on the infrastructure of the University Computer Centre.

5. Send another POST request with the form data consisting of *username, password,* and *AuthState* to an URL that is very similar as the previous one (replace the "TUC/username.php" to "core/loginuserpass.php", other parts of the URL are exactly the same) . From the response body of this request, retrieve the *SAMLResponse* and the *RelayState* values. These values are inside the <input type="hidden"> tags. *SAMLResponse* contains an encrpyted Assertion, which is the XML document

containing user authorization that are transferred from the identity provider to the service provider. *RelayState* is used to identify the destination URL the user will access after successfully signing in with SSO.

6. Finally, send a POST request to https://www.tu-chemnitz.de/Shibboleth.sso/SAML2/POST with the form values of *SAMLResponse* and *RelayState.* After this POST request, the server is successfully authenticated, and at long last, it can start communicating with the distributed blocklist service.

* Note that this implementation assumes that the server **will not** run continuously until the session time expires, which is the case for this project. Of course, the implementation could be adjusted to solve this hypothetical problem. Instead of doing the authentication process when the server starts, do the authentication process only if the server gets redirected when trying to communicate with the blocklist service. This can be implemented by wrapping the authentication inside a function and checking the response body of each request made.

# Functionalities

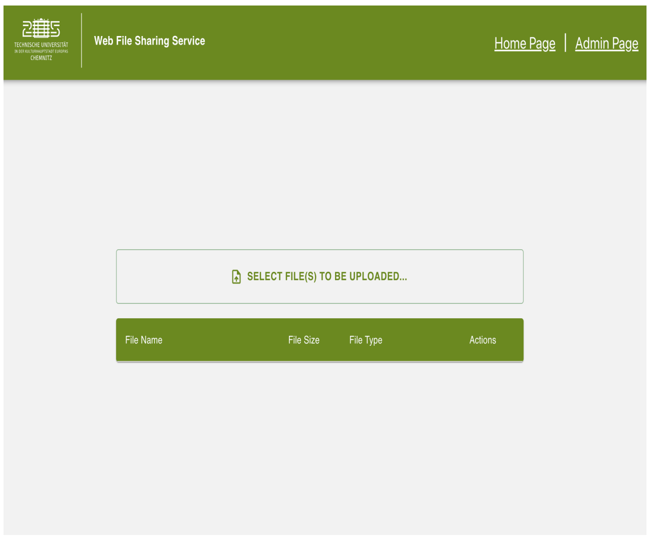
## Overview

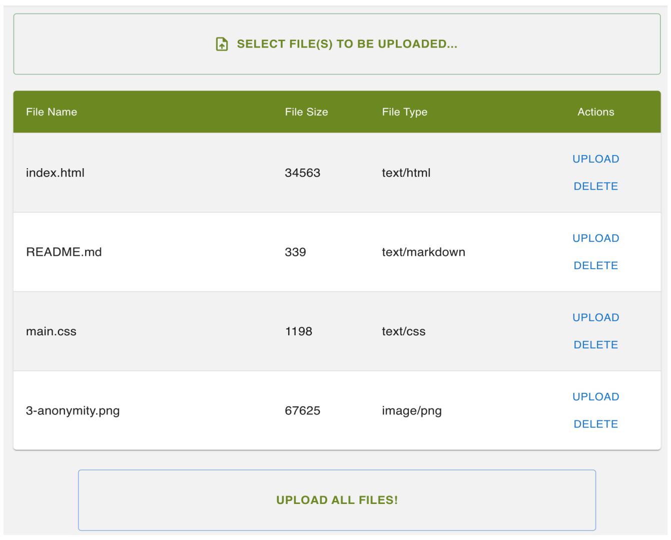
All the required functionalities/constraints for individual work as specified in the project task are implemented. In addition, the option to upload multiple files at once is also supported. The details on how these features are implemented will be explained in the sub-section below.

## File Upload

In the client-side, specifically the home page, there is a large button that user can click, and inside the button, the <input> HTML element with type="file" is used to let user choose one or more files from their device to be uploaded.

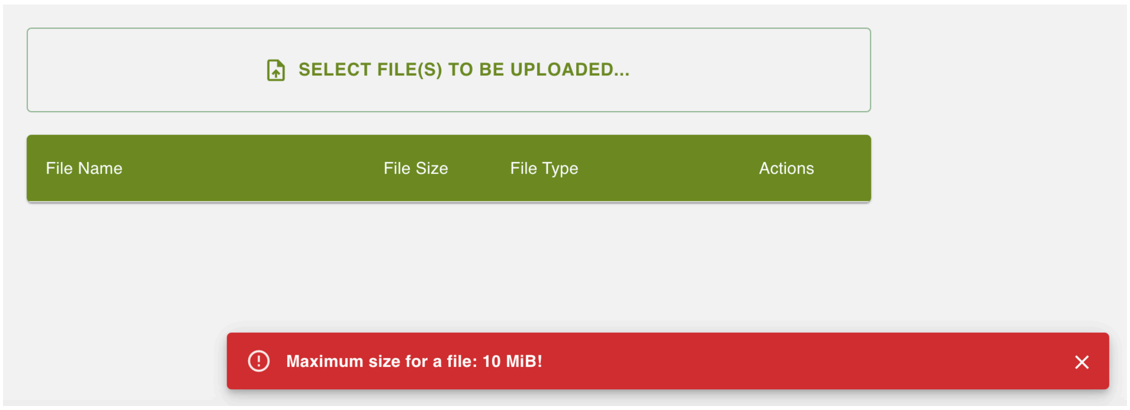
For each of the chosen file, the metadata of the file (file name, file size, and file type), will be displayed, and user has the option to upload all files simultaneously or upload or delete a file from the table.





| File Name | File Size | File Type | Actions |
|---|---|---|---|
| index.html | 34563 | text/html | UPLOAD DELETE |
| README.md | 339 | text/markdown | UPLOAD DELETE |
| main.css | 1198 | text/css | UPLOAD DELETE |
| 3-anonymity.png | 67625 | image/png | UPLOAD DELETE |

If user tries to upload a file larger than 10 MiB, an error alert message would be displayed and the file would be deleted immediately.

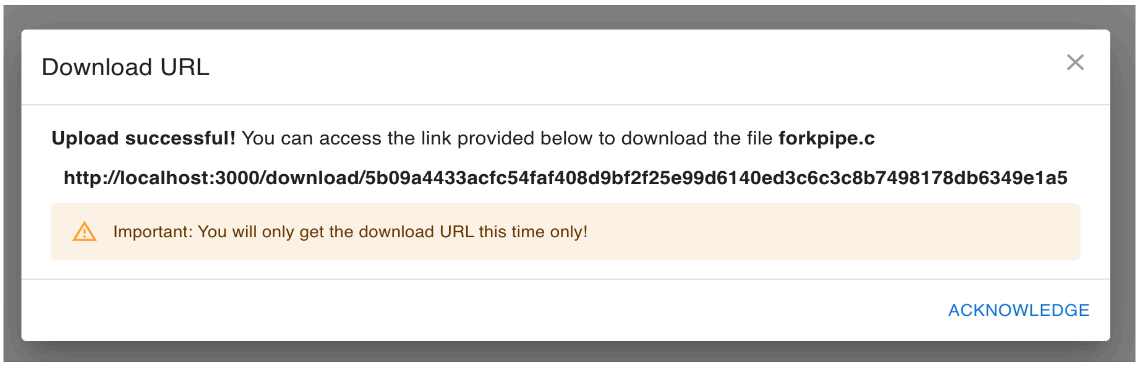


Maximum size for a file: 10 MiB!

When user clicks on the *Upload* or *Upload All Files* button, the respective API would be called, and the file(s) data would be send to the server-side. Note that there are two separate APIs for the case of a single file upload and multiple files upload (for details, refer to the *7. API Documentation* section).

Inside the server, for each file, the corresponding SHA256 string is generated, and a GET request based on this hash would be made to the blocklist service to check whether this file should be blocked or not based on the response status code as specified in the website. Then, the server store the file data intto the database. Note that in case there is an identical file stored in the database (a file with the same SHA256 string), the old file would be first deleted, and then the new file is inserted. This ensures that the *fileUploadTime* and *lastDownloadTime* of this particular file get updated appropriately.

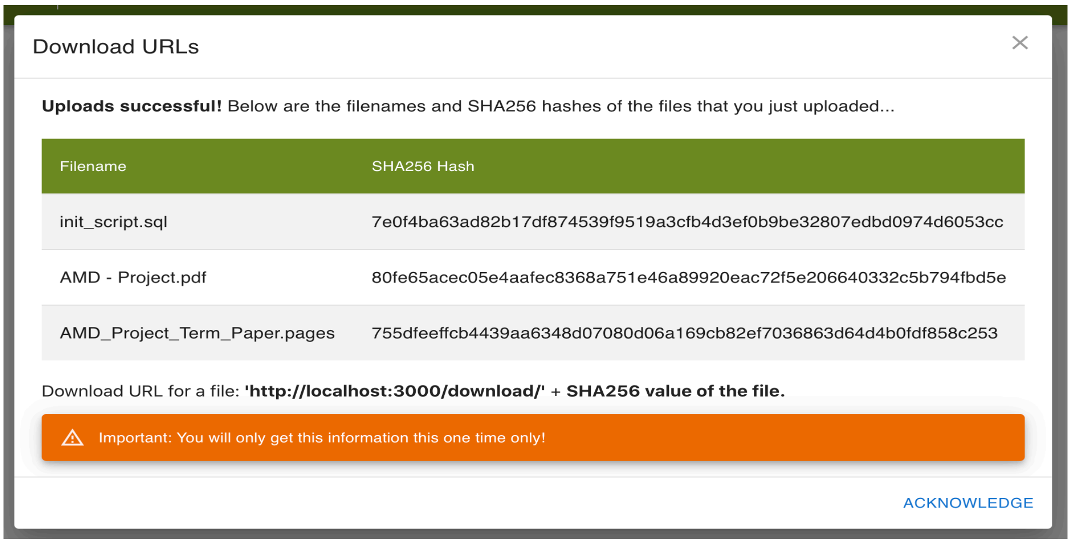
Finally, the server will send back a response to the client in JSON format, containing the file name and the hash value of the file, which corresponds to the download URL of that file. This information will be shown in the client-side to notify the user that the file(s) is/are successfully uploaded. In case of error(s) encountered, the server would send a JSON response containing an error message to the client instead, and the client would handle the error accordingly. Note that there are some differences on how the information is shown between the download URL(s) for a single file upload and multiple files upload.


Download URL

**Upload successful!** You can access the link provided below to download the file **forkpipe.c**

**http://localhost:3000/download/5b09a4433acfc54faf408d9bf2f25e99d6140ed3c6c3c8b7498178db6349e1a5**

⚠ Important: You will only get the download URL this time only!


Download URLs

**Uploads successful!** Below are the filenames and SHA256 hashes of the files that you just uploaded...

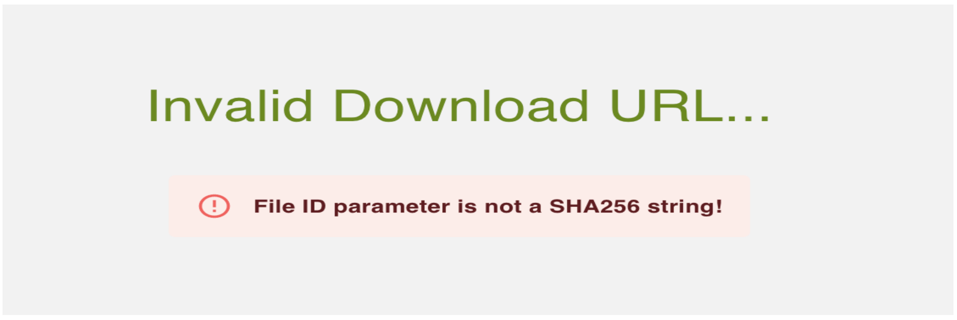
| Filename | SHA256 Hash |
|---|---|
| init_script.sql | 7e0f4ba63ad82b17df874539f9519a3cfb4d3ef0b9be32807edbd0974d6053cc |
| AMD - Project.pdf | 80fe65acec05e4aafec8368a751e46a89920eac72f5e206640332c5b794fbd5e |
| AMD_Project_Term_Paper.pages | 755dfeeffcb4439aa6348d07080d06a169cb82ef7036863d64d4b0fdf858c253 |

Download URL for a file: **'http://localhost:3000/download/'** + **SHA256 value of the file.**

⚠ Important: You will only get this information this one time only!

## File Download

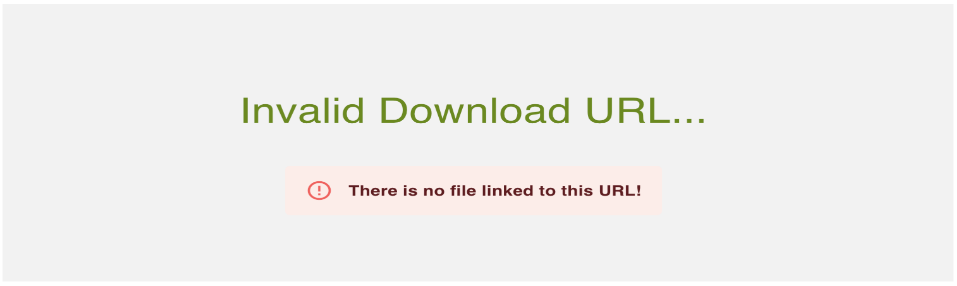
To download a file, user needs to first have an access to the download URL of that particular file. The URL is in the format of **HOST/download/:id,** in which the :id parameter is the SHA256 string of that particular file. If :id is not in the correct format, the "error message" below would be displayed, and no API call to the server would be made.



Invalid Download URL...

File ID parameter is not a SHA256 string!

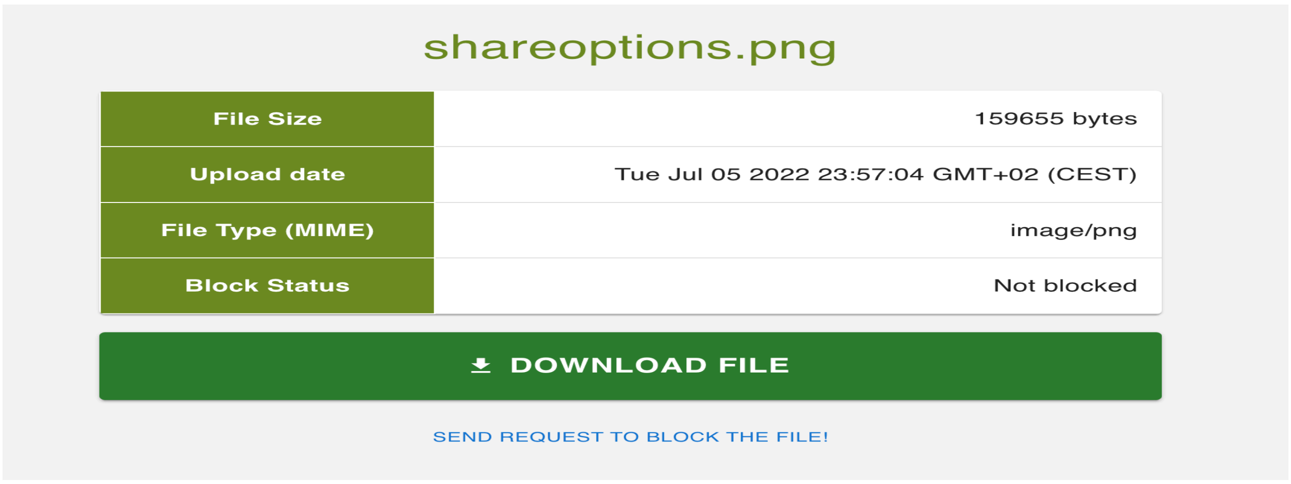
On the other hand, if :id is a valid SHA256 string, an API would be called to try to get the information of the corresponding file. If the database does not have a file with the same hash value, the message below would be shown.



Invalid Download URL...

There is no file linked to this URL!

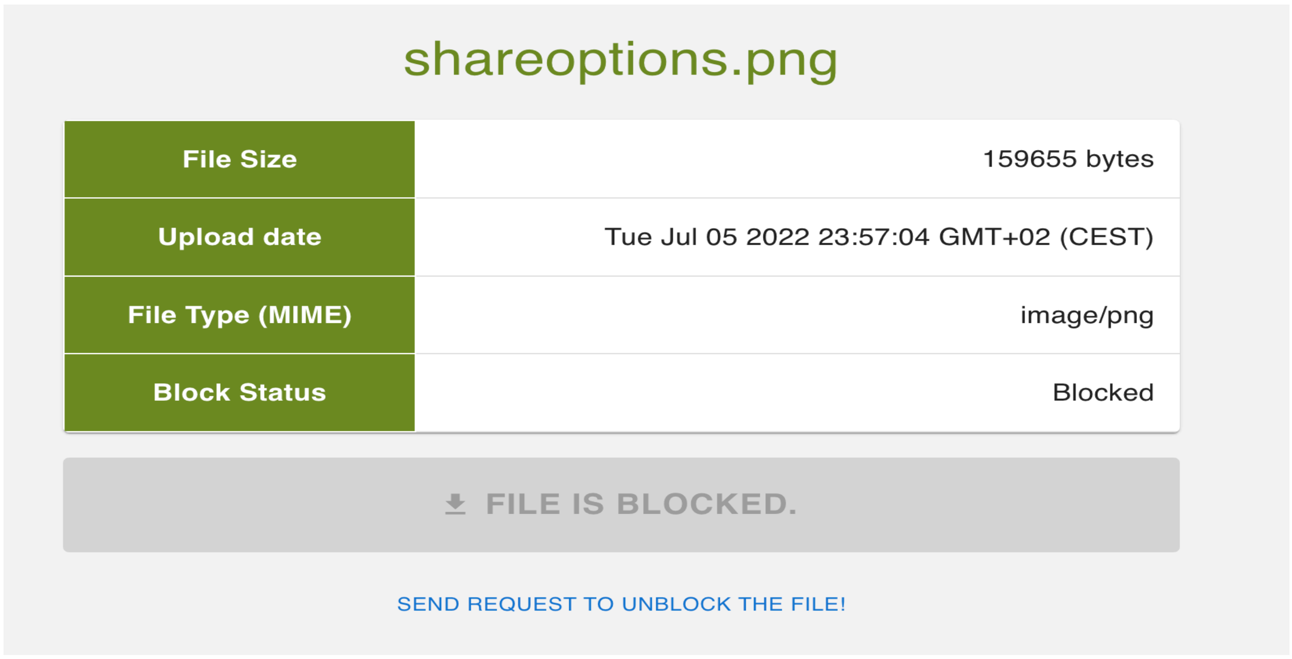
Finally, if there is a file with the SHA256 ID that corresponds with :id, the data of this particular file would be sent from the database via the server-side to the client-side. The file information (file name, file size, upload date, file type, and the block status) would be displayed to the user, and if the file is unblocked, user can click on the *Download* button to download the file.



shareoptions.png

| | |
|---|---|
| File Size | 159655 bytes |
| Upload date | Tue Jul 05 2022 23:57:04 GMT+02 (CEST) |
| File Type (MIME) | image/png |
| Block Status | Not blocked |

However, if the file is blocked, the button would be disabled and it would be impossible to download the file.



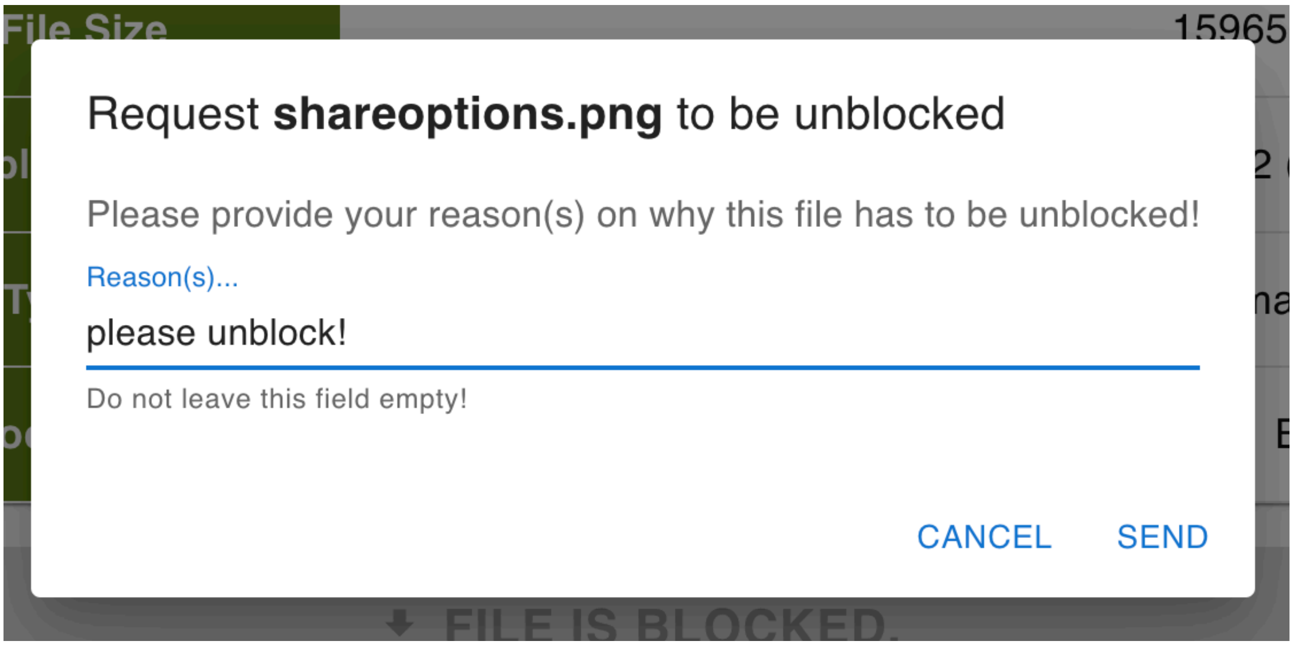
shareoptions.png

|  |  |
|---|---|
| File Size | 159655 bytes |
| Upload date | Tue Jul 05 2022 23:57:04 GMT+02 (CEST) |
| File Type (MIME) | image/png |
| Block Status | Blocked |

To implement the download functionality, the Blob (binary large object) is utilized. The *fileData* which initially is stored in the *bytea* data class is automatically converted to Buffer class by NodeJS, and in the client-side this Buffer is converted into ArrayBuffer of Uint8Array, which is used to create the Blob of the file. When the download button is clicked, a <link> element with href attribute to the blob object URL and with a download attribute is created, and the HTMLElement method *click* is called on the <link> element start the download process. After the file is downloaded, <link> is removed from the document, and the object URL of the file is released so that the browser does not keep reference to that file anymore (less memory used). Finally, the client-side will make another API call to the server-side so that the last download date of this file is updated in the database.

**Blocking / Unblocking a File**

As shown in the pictures of the previous sub-section, user can create a blocking or unblocking request for a file to the admin by clicking on the button in the bottom of the page. After clicking on that button, a dialog with text field would be displayed, requiring user to write the reason for blocking or unblocking the file.



Request **shareoptions.png** to be unblocked

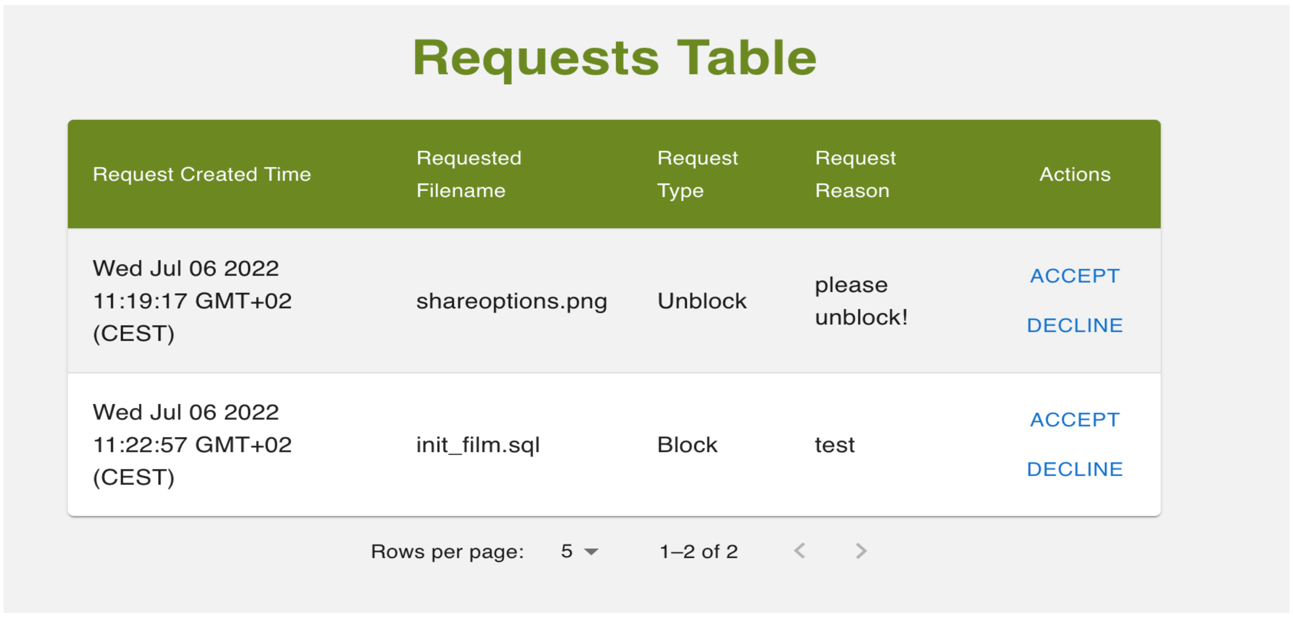
Please provide your reason(s) on why this file has to be unblocked!

Reason(s)...

please unblock!

Do not leave this field empty!

If user clicks *Send,* the API to create this request will be called, and in the server-side this request will be inserted into the table *Requests* in the database. In the admin page, the list of pending requests is shown, and for each request, the admin can choose to either accept or decline.



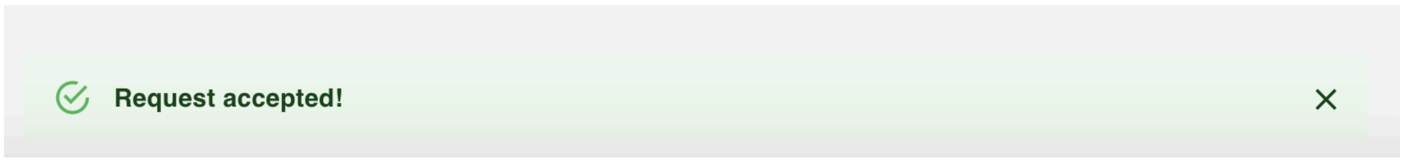
Requests Table

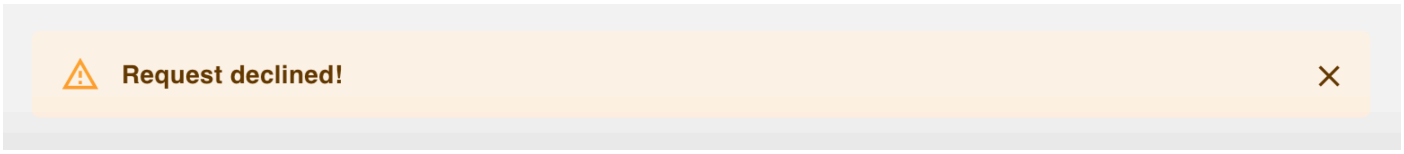
| Request Created Time | Requested Filename | Request Type | Request Reason | Actions |
|---|---|---|---|---|
| Wed Jul 06 2022 11:19:17 GMT+02 (CEST) | shareoptions.png | Unblock | please unblock! | ACCEPT DECLINE |
| Wed Jul 06 2022 11:22:57 GMT+02 (CEST) | init_film.sql | Block | test | ACCEPT DECLINE |

Rows per page: 5 1–2 of 2

When the *Accept* or *Decline* button is clicked, the API for processing a request will be called. In case of *Decline,* the process is simpler. In the server-side, a query to the database will be made to update the value of *requestStatus* to be *'Declined'*. On the other hand, if an *Accept* button is clicked, the server would first communicate with the blocklist service. It would send a PUT request if the request type is block, and send a DELETE request if the request type is unblock. Depending on the response status code from the blocklist service, the server might send a query to the database to update the *isFileBlocked* value of the corresponding file. Another query would be made to update the value of *requestStatus* of the request to be *'Accepted'.*

Finally, an alert indicating whether the request is accepted or declined will be shown in the client-side, and the related request will also be deleted from the Requests Table.


Request accepted!


Request declined!

## Removal of Inactive Files

One of the tasks states that inactive files (files which are not downloaded for certain days) are to be removed by the system. This means that the database has to be inspected and sanitized periodically. To achieve this, a job scheduler needs to be created, such that an execution of stored procedure and/or SQL statements to update the table(s) in the database can be done automatically at a certain time recurrently. In this project, the job is executed daily at 10AM, where file(s) which have not been downloaded for the **last 7 days** would be deleted from the database.

Since PostgreSQL does not have a built-in job scheduler, an external tool is needed to run the task. Unfortunately, it seems like it is not possible to install extension such as pgAgent or pg_cron in the PostgreSQL database provided by the university. Hence, the system software **cron** is used, in which a shell script that would run a simple SQL script on execution is the task to be done daily. The credentials of the database is stored in the *env* file, which you can change the data if you want to run the project with your own database.

For reference, the cron command is written as follows:

```
0 10 * * * ~/Desktop/dbw-final-project && sh ./cronjob.sh >> ~/Desktop/
dbw-final-project/cron.txt 2>&1
```

**`0 10 * * *`** indicates that the job will be executed daily at 10AM.
**`~/Desktop/dbw-final-project`** is the directory where the shell script is located.
**`&& sh ./cronjob.sh`** runs the shell script
**`>> ~/Desktop/dbw-final-project/cron.txt`** indicates the location in where the output of the cronjob will be written
**`2>&1`** is added to disable cron emails.

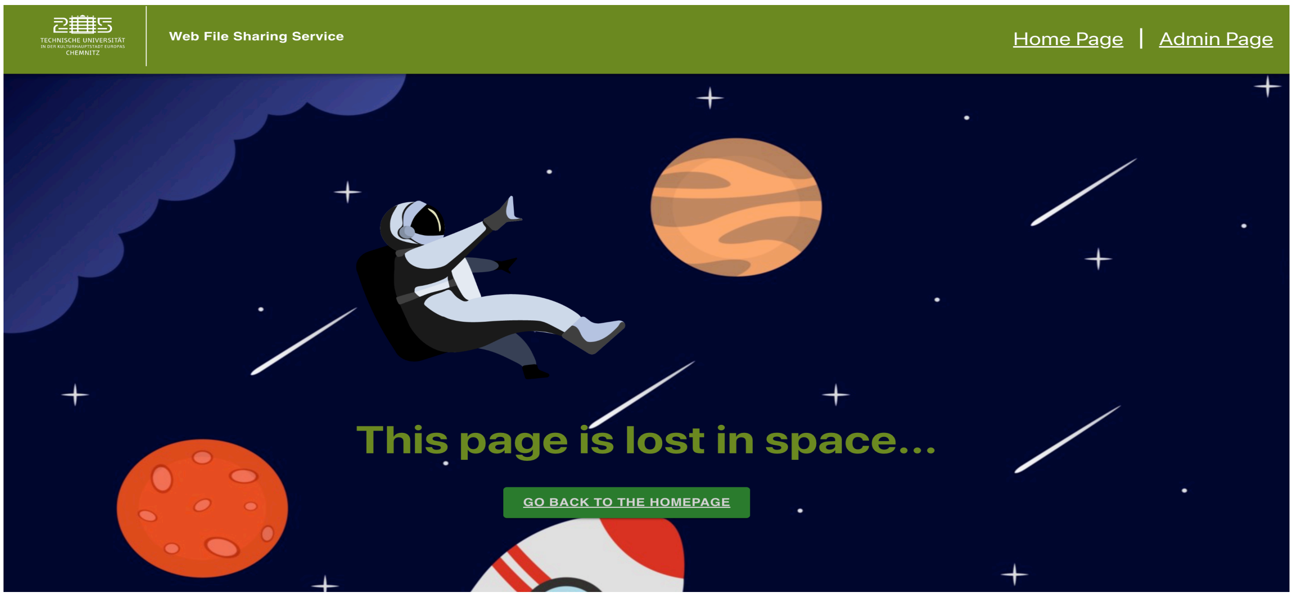
Note that *psql* has to be installed to run the cronjob. In addition, if you want to run the cronjob in your own PC, change the first argument of the last line (line 9) of the cronjob.sh file to the path where your psql is located.

# Frontend: Routes and Responsive Web Design

This section will briefly discuss the frontend aspect of the project, specifically the routing for each page and also the responsive web design aspect of the pages.

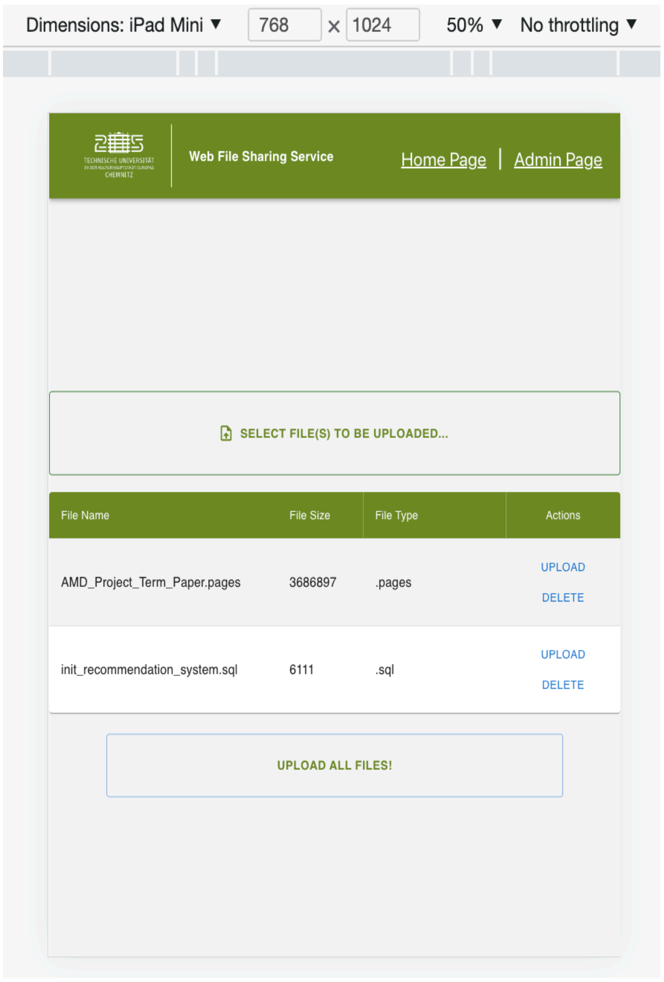
The project has 4 pages in total:

| Path | Description |
|---|---|
| / or /home | the home page where user can upload file(s). |
| /admin | the admin page where the requests made can be processed. |
| /download/:id | the download page of a file with id={id}. The information of the file is displayed and user can download the file here. |
| * | for all other paths, no found page will be displayed. See the picture below. |


TECHNISCHE UNIVERSITÄT IN DER KULTURHAUPTSTADT EUROPAS CHEMNITZ
Web File Sharing Service
Home Page | Admin Page
This page is lost in space...
GO BACK TO THE HOMEPAGE

The aspect of responsive web design (RWD) is maintained with the help of Material UI, specifically with the <Grid> component. <Grid> uses CSS's flexible box layout module, and this responsive layout adapts to screen size and viewport of the browser. In addition, the font size, width, and height of all elements and components are set with relative units (%) or with viewport units (vh, vw, vmin, vmax). With viewport units, the components/elements are resized depending on the device display size, which ensures that the UI is consistent across different viewports.

## Illustrations



Dimensions: iPad Mini ▼ 768 × 1024 50% ▼ No throttling ▼

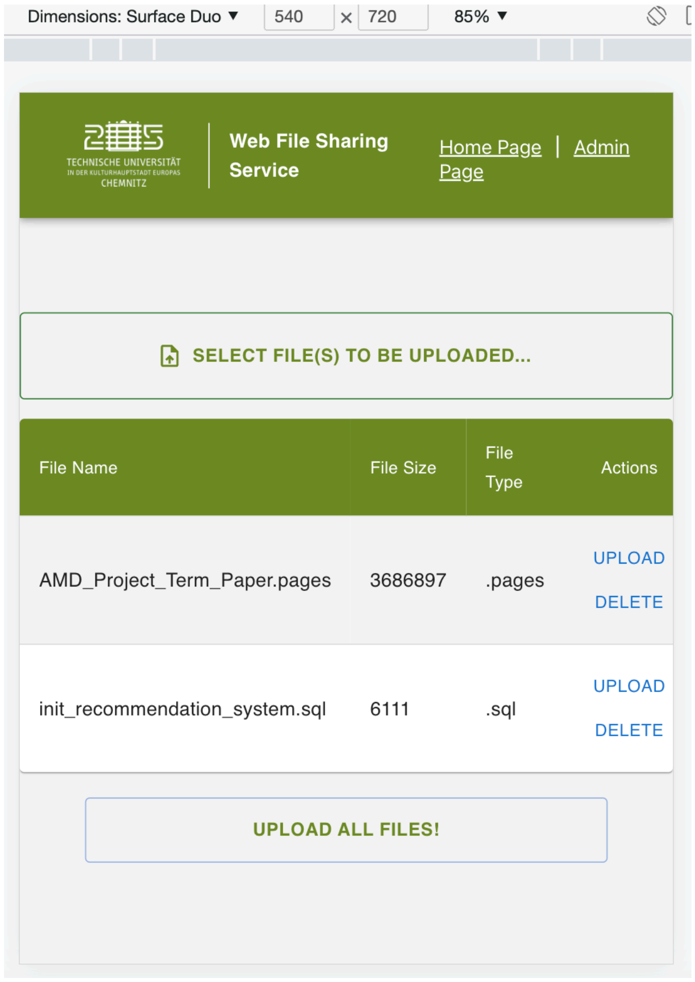
Web File Sharing Service

Home Page | Admin Page

SELECT FILE(S) TO BE UPLOADED...

| File Name | File Size | File Type | Actions |
|---|---|---|---|
| AMD_Project_Term_Paper.pages | 3686897 | .pages | UPLOAD DELETE |
| init_recommendation_system.sql | 6111 | .sql | UPLOAD DELETE |

UPLOAD ALL FILES!



Dimensions: Surface Duo ▼ 540 × 720 85% ▼

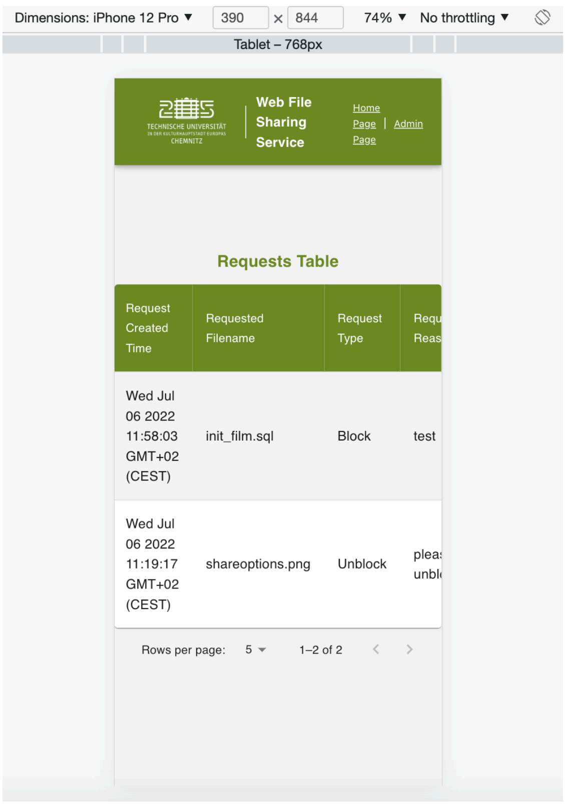
Web File Sharing Service

Home Page | Admin Page

SELECT FILE(S) TO BE UPLOADED...

| File Name | File Size | File Type | Actions |
|---|---|---|---|
| AMD_Project_Term_Paper.pages | 3686897 | .pages | UPLOAD DELETE |
| init_recommendation_system.sql | 6111 | .sql | UPLOAD DELETE |

UPLOAD ALL FILES!



Dimensions: iPhone 12 Pro ▼ 390 × 844 74% ▼ No throttling ▼

Tablet – 768px

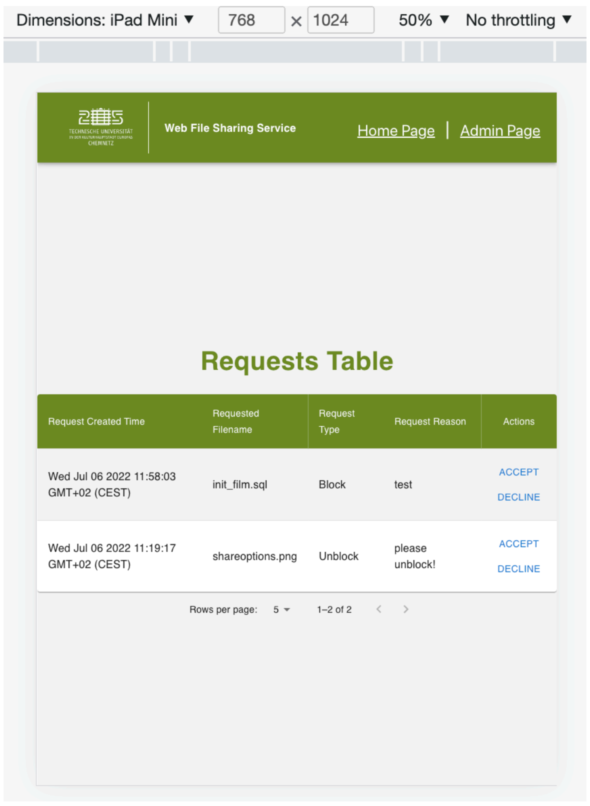
Web File Sharing Service

Home Page | Admin Page

Requests Table

| Request Created Time | Requested Filename | Request Type | Requ Reas |
|---|---|---|---|
| Wed Jul 06 2022 11:58:03 GMT+02 (CEST) | init_film.sql | Block | test |
| Wed Jul 06 2022 11:19:17 GMT+02 (CEST) | shareoptions.png | Unblock | pleas unbl |

Rows per page: 5 ▾ 1–2 of 2 ‹ ›



Dimensions: iPad Mini ▼ 768 × 1024 50% ▼ No throttling ▼

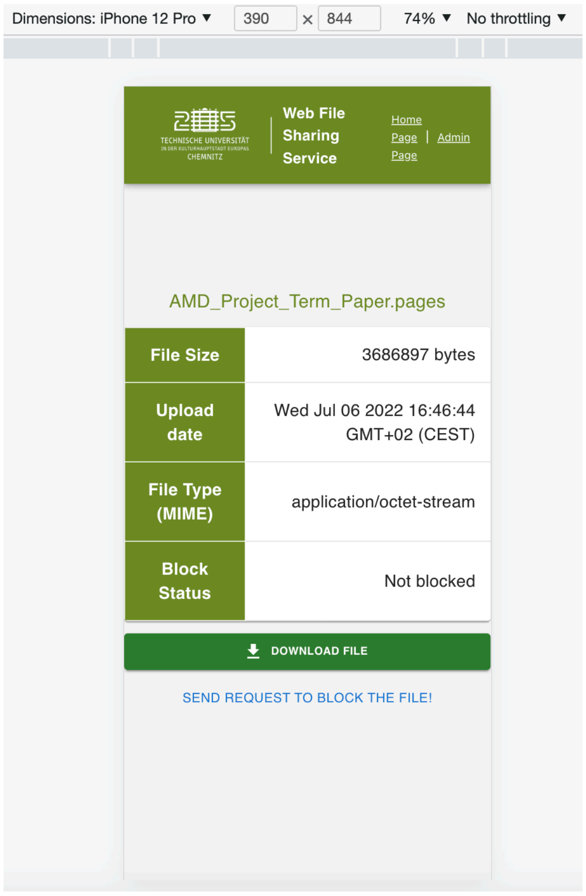
Web File Sharing Service

Home Page | Admin Page

Requests Table

| Request Created Time | Requested Filename | Request Type | Request Reason | Actions |
|---|---|---|---|---|
| Wed Jul 06 2022 11:58:03 GMT+02 (CEST) | init_film.sql | Block | test | ACCEPT DECLINE |
| Wed Jul 06 2022 11:19:17 GMT+02 (CEST) | shareoptions.png | Unblock | please unblock! | ACCEPT DECLINE |

Rows per page: 5 ▾ 1–2 of 2 ‹ ›


Dimensions: iPhone 12 Pro ▾ 390 × 844 74% ▾ No throttling ▾

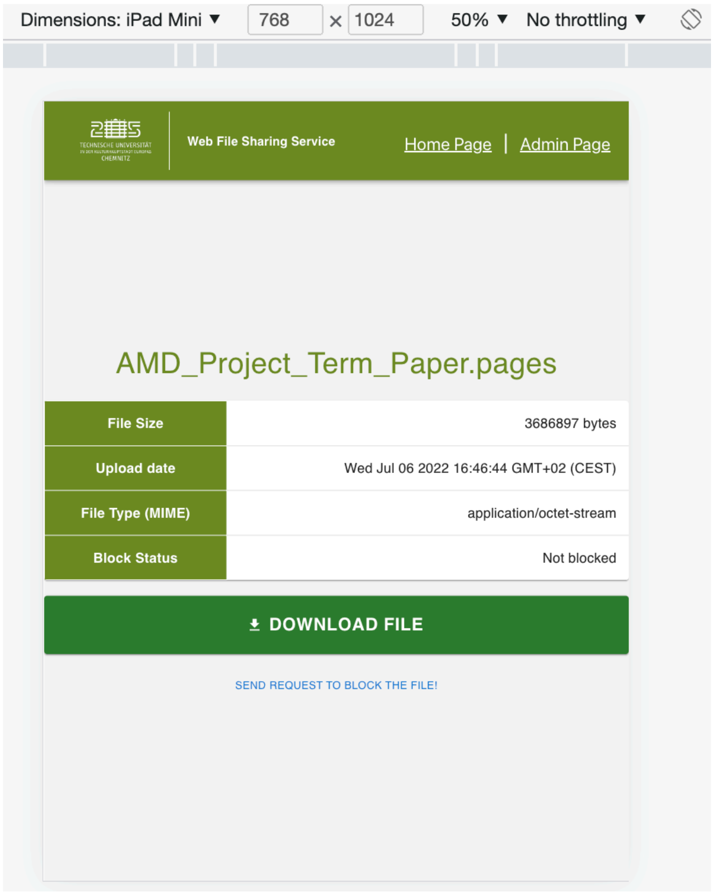
Web File Sharing Service | Home Page | Admin Page

AMD_Project_Term_Paper.pages

| | |
|---|---|
| File Size | 3686897 bytes |
| Upload date | Wed Jul 06 2022 16:46:44 GMT+02 (CEST) |
| File Type (MIME) | application/octet-stream |
| Block Status | Not blocked |

DOWNLOAD FILE

SEND REQUEST TO BLOCK THE FILE!


Dimensions: iPad Mini ▾ 768 × 1024 50% ▾ No throttling ▾

Web File Sharing Service | Home Page | Admin Page

AMD_Project_Term_Paper.pages

| | |
|---|---|
| File Size | 3686897 bytes |
| Upload date | Wed Jul 06 2022 16:46:44 GMT+02 (CEST) |
| File Type (MIME) | application/octet-stream |
| Block Status | Not blocked |

DOWNLOAD FILE

SEND REQUEST TO BLOCK THE FILE!

Note that when the device width is too small to fully show the table, which is the case with most of mobile devices, the table would become scrollable instead.

# APIs Documentation

**Overview**

In total, there are 7 APIs created and used for the project. When needed, the request body is constructed with **FormData**, which provides an easy way to create a set of key-value pairs representing form fields and their values. The values of the form fields are either File object or string. All of the API responses (successful or error) are in the JSON format.

**API details**

1. Upload a file
   - Path: '/uploadFile'
   - Description: upload a single file into the database.
   - Method: POST
   - Query parameter: -
   - Request body: { file: File }. Example of a File object received in the server:

     ```
     {
       name: 'script.sql',
       data: <Buffer 2d 2d 55 …>,
       size: 18000,
       encoding: '7bit',
       tempFilePath: '',
       truncated: false,
       mimetype: 'application/octet-stream',
       md5: '80244398ff24f461df03facf032c17cc',
     }
     ```
   - Response body example (successful):

     ```
     {
       hashValue: "3fdd6d5c6fc54d5f6e4d7acdfaa9657506ab3c2962e433595a0373f779a1c7f0",
       fileName: "test.txt"
     }
     ```
   - Successful status code: 201
   - Response body (error): { errMsg: 'Internal server error!' }
   - Error code: 500
   - WTC authentication: required

2. Upload multiple files
   - Path: '/uploadFiles'
   - Description: upload multiple files into the database.
   - Method: POST
   - Query parameter: -
   - Request body: { file_0: File, file_1: File, …, file_files.length: File }
   - Response body example (successful):

     [

       { hashValue: "3fdd6d5c6fc54d5f6e4d7acdfaa9657506ab3c2962e433595a0373f779a1c7f0", fileName: "test.txt" },

       …,

       { hashValue: "7e0f4ba63ad82b17df874539f9519a3cfb4d3ef0b9be32807edbd0974d6053cc", fileName: "test.png" }

     ], where array length = the number of files uploaded.
   - Successful status code: 201
   - Response body (error): { errMsg: 'Internal server error!' }
   - Error code: 500
   - WTC authentication: required

3. Update *lastDownloadDate* of a file
   - Path: '/lastDownloadDate'
   - Description: update the *lastDownloadDate* of a newly downloaded file
   - Method: PUT
   - Query parameter: id: string (required)
   - Request body: -
   - Response body example (successful): {}
   - Successful status code: 200
   - Response body (error): { errMsg: 'Internal server error!' }
   - Error code: 500
   - WTC authentication: not required

4. Get the file information
   - Path: '/getFileInfo'
   - Description: get the file information of a file with the corresponding id
   - Method: GET

- Query parameter: id: string (required)
- Request body: -
- Response body example (successful):

  ```
  {
     file_id: “133eb8df6203b34069aa83f8d493dae9d3bc5a0ec0aef76ec408f2e3f7cdb069”,
     file_name: “test.sql”,
     file_size: 35000,
     file_data: { type: ‘Buffer’, data: Array(35000) },
     file_upload_time: “2022-07-06T15:34:33.846Z”,
     last_download_time: “2022-07-06T15:34:33.846Z”,
     file_type: ‘application/octet-stream’,
     is_file_blocked: false
  }
  ```
- Successful status code: 200
- Response body (error): { errMsg: ‘Internal server error!’ }
- Error code: 500
- WTC authentication: required

5. Create a request
   - Path: ‘/createRequest’
   - Description: create a request to block or unblock a file, which is to be processed by the admin
   - Method: POST
   - Query parameter: -
   - Request body (example):

     ```
     {
        fileId: ‘133eb8df6203b34069aa83f8d493dae9d3bc5a0ec0aef76ec408f2e3f7cdb069’,
        fileName: ‘init.sql’,
        requestType: ‘1’,
        reason: ‘test123’
     }
     ```
   - Response body example (successful): {}
   - Successful status code: 201
   - Response body (error): { errMsg: ‘Internal server error!’ }
   - Error code: 500

- WTC authentication: not required

6. Get all requests
    - Path: '/getAllRequests'
    - Description: get all of the pending requests to be shown in the admin page.
    - Method: GET
    - Query parameter: -
    - Request body: -
    - Response body example (successful):

```
[
  ……,
  {
    file_id: “133eb8df6203b34069aa83f8d493dae9d3bc5a0ec0aef76ec408f2e3f7cdb069”,
    file_name: “init.sql”,
    request_created_time: “2022-07-06T16:00:14.840Z”,
    request_id: “b337ff99-acab-473f-acba-39e39561986b”,
    request_reason: “please unblock”,
    request_status: “Pending”,
    request_type: 1
  },
  …..
]
```

    - Successful status code: 200
    - Response body (error): { errMsg: ‘Internal server error!’ }
    - Error code: 500
    - WTC authentication: not required

7. Process a request
    - Path: '/processRequest'
    - Description: process (accept or decline) a request with the corresponding id
    - Method: PUT
    - Query parameter: id: string (required)
    - Request body (example):

```
{
   fileId: ‘133eb8df6203b34069aa83f8d493dae9d3bc5a0ec0aef76ec408f2e3f7cdb066’,
   requestType: '1',
   decision: 'acc'
}
```

*** *decision* is a string with two possible values: 'acc' to indicate that the request is accepted and 'dec' to indicate the request is declined. ***

- Response body example (successful): {}
- Successful status code: 200
- Response body (error): { errMsg: 'Internal server error!' }
- Error code: 500
- WTC authentication: required

*Note that *WTC authentication: required* means that the API will make an external request to the blocklist service.

# References

[1]. Documentations for used technologies

PostgreSQL: https://www.postgresql.org/docs/
yarn: https://yarnpkg.com/
ReactJS: https://reactjs.org/
Create React App: https://create-react-app.dev/
TypeScript: https://www.typescriptlang.org/
NodeJS: https://nodejs.org/
ExpressJS: http://expressjs.com/
FetchAPI: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
Material UI: https://mui.com/
react-router-dom: https://github.com/remix-run/react-router
node-postgres (pg): https://node-postgres.com/
request: https://github.com/request/request
crypto: https://nodejs.org/api/crypto.html
express-fileupload: https://github.com/richardgirges/express-fileupload
cheerio: https://cheerio.js.org/
crontab: https://man7.org/linux/man-pages/man5/crontab.5.html
shell script: https://www.shellscript.sh/

[2]. Information on SAML authentication:

- http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html
- https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language

[3]. Information on FormData
https://developer.mozilla.org/en-US/docs/Web/API/FormData

[4]. The project uses TU Chemnitz's logo which is taken from here:
https://www.tu-chemnitz.de/tu/pressestelle/cd/vorlagen.html

[5]. Additionally, some other images are also used: these images are open-source files found from Google Images.