Canon File Uploader

Documentation

2023r.

Author: Konrad Lempart

1. Front End side

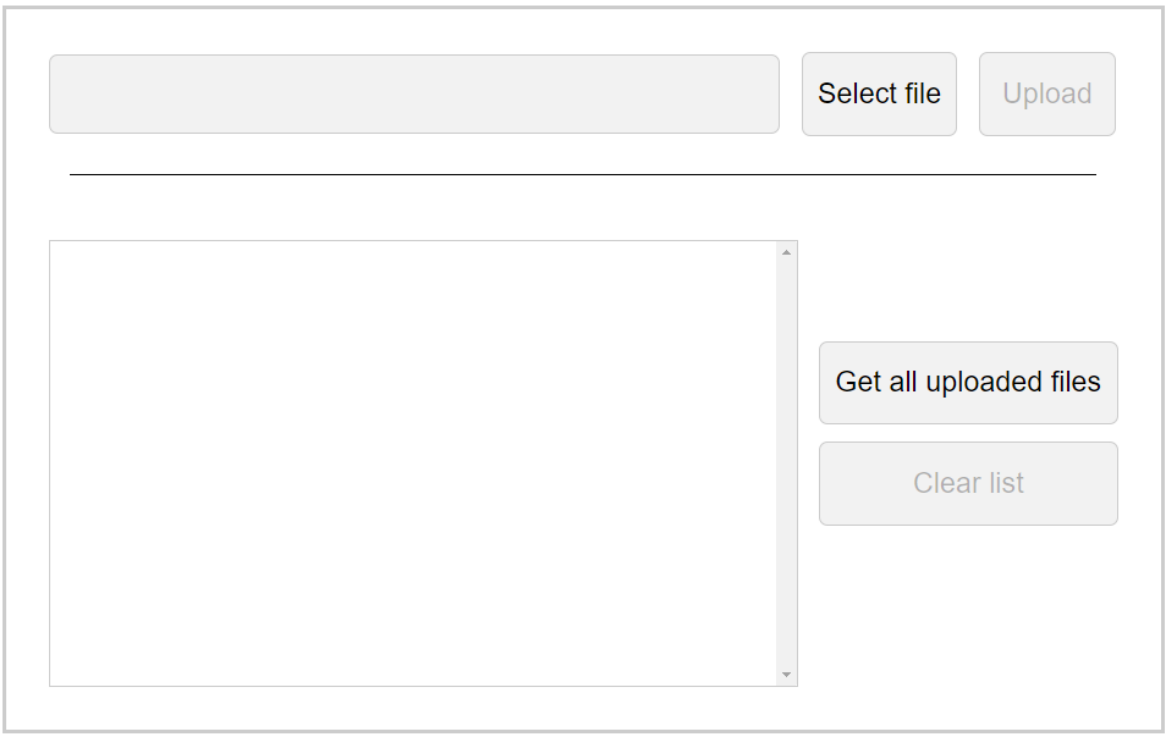
The application simulates adding files to the database. It consists of several components that communicate with each other and a service that sends queries to the backend project (CanonFileAPI).

1.1 Used technologies

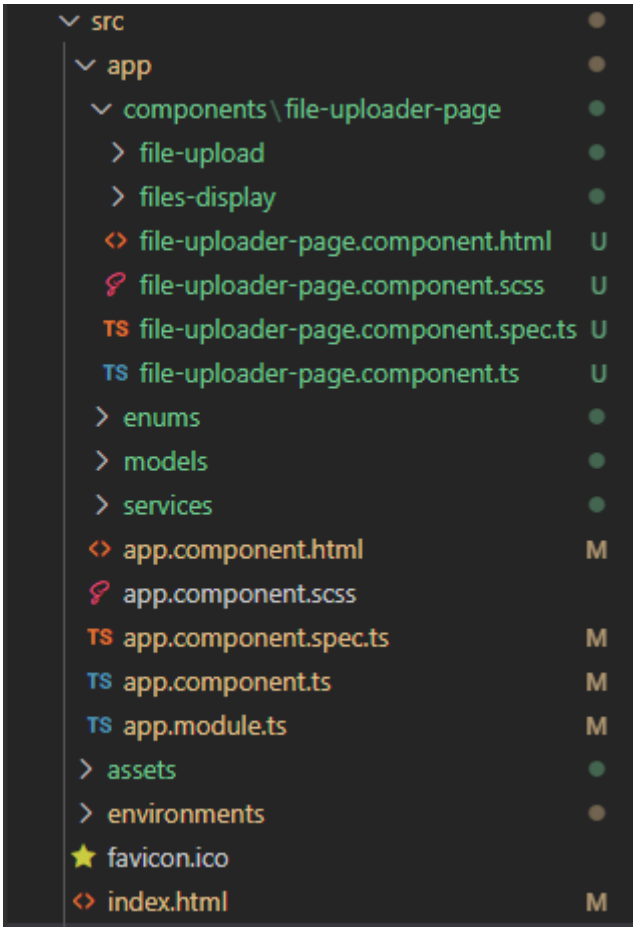
Angular framework version 14.3.0 was used. Project also consists of several Angular Material components. Whole application was created in Visual Studio Code.

1.2 Project structure

The application view is presented in the screenshot below:



Project was split into multiple parts:

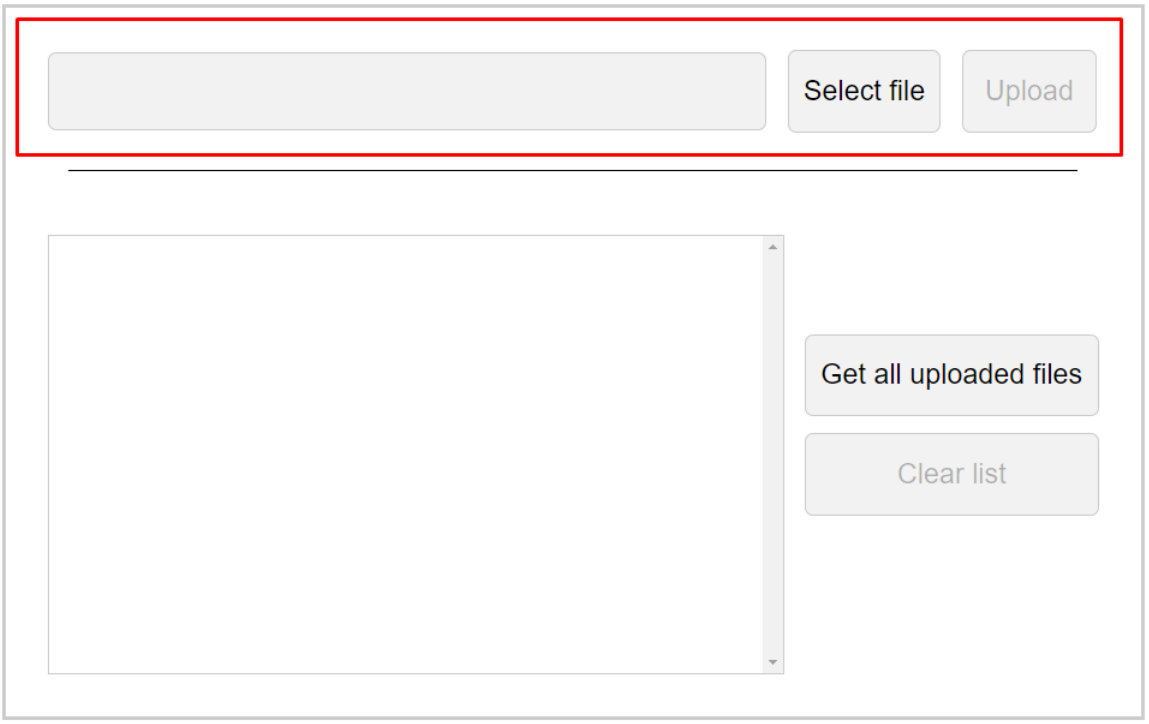


All of the code is localized in the **/src** directory. The **/app** directory inside of it consists of **components**, **enums**, **models**, and **services** folders. **App.component** is the main component of the project.

Every component consists of **.html** file (responsible for displayed components arrangement), **.scss** file (responsible for displayed components' styles), **.ts** file (responsible for the component business logic) and **.spec.ts** (unit tests for the component). Unit tests were created based on specific component logic.

1.2.1 File upload component

Part of the application that **File upload component** is responsible for is presented below:



Html code:

```
<div class="container">
    <mat-card class="selected-file-display overflow-span">
        <mat-card-content class="card-content ">
            <span class="overflow-span">{{ selectedFile }}</span>
            <button *ngIf="selectedFile" class="clear-button" (click)="clearSelectedFile()">
                <mat-icon>clear</mat-icon>
            </button>
        </mat-card-content>
    </mat-card>

    <button mat-button class="button" (click)="fileInput.click()">Select file</button>
    <input type="file" #fileInput (change)="handleFileSelect()" style="display: none">
    <button mat-button class="button" [disabled]="isUploadDisabled()" (click)="uploadSelectedFile()">Upload</button>
</div>
```

On the left side there is a mat-card that displays **selectedFile**. SelectedFile is a name of the file that can be selected by **Select file** button. After pressing it, a file explorer opens and the file can be chosen. When file is chosen, the **handleFileSelect()** function is called:

```
handleFileSelect() {
  const file = this.fileInput.nativeElement.files[0];

  if (file) {
    this.selectedFile = file.name;
  }

  this.fileInput.nativeElement.value = null;
}
```

It takes a file name from the selected file and saves it as a **selectedFile**.

Upload button is disabled as long as there is no selected file. If the file is selected and the upload button is clicked, an **uploadSelectedFile()** function is called:

```
uploadSelectedFile() {
  if (!this.selectedFile || this.isFileAlreadyAdded()) {
    return;
  }

  const newFile: IFile = {
    name: this.selectedFile
  }

  this.filesService.addFile(newFile).subscribe(
    () => {
      this.files.push(newFile);
      this.filesService.setFiles(this.files)
    }
  );
}
```

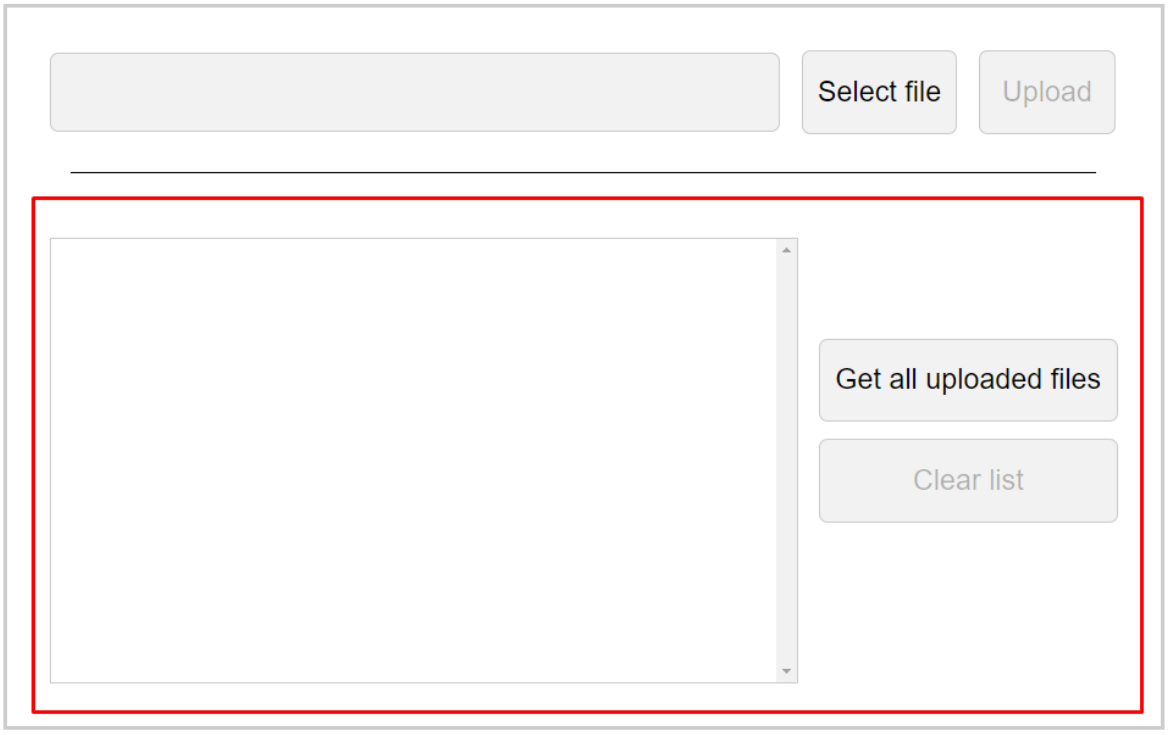
It checks if a file is already added and a selected file exists. After that, it uses the **addFile()** function of the **FileUploadService** instance provided in the constructor. After the method is used, it adds the new file to the currently stored files.

If a file is selected, the clear button is displayed:

After pressing it, the **selectedFile** is cleared and no longer displayed.

1.2.2 Files display component

Part of the application that **Files display component** is responsible for is presented below:

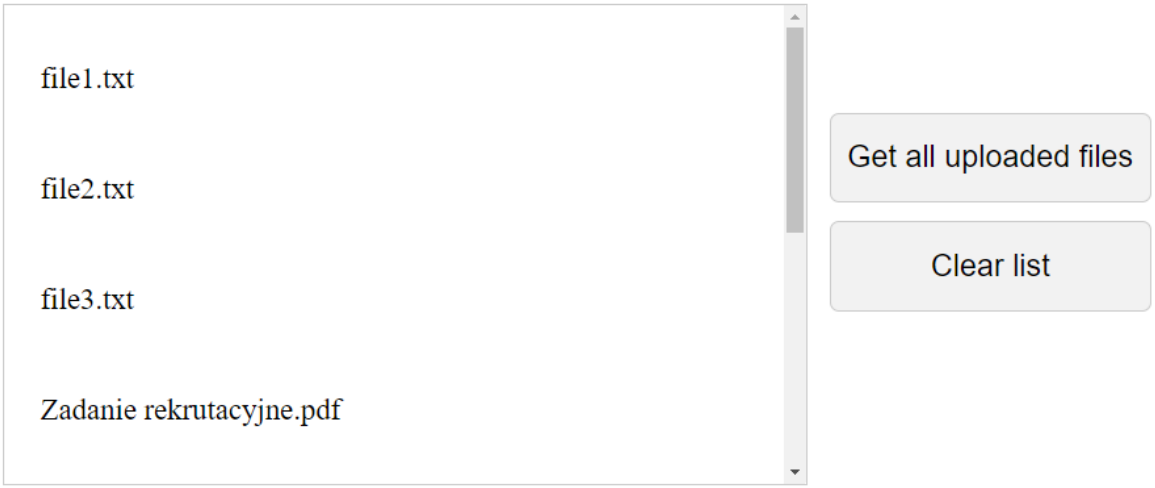


Html code:

```
<div class="container">
    <mat-list class="list">
        <mat-list-item  class="mat-list-item-content" *ngFor="let file of files">
        {{ file?.name }}
        </mat-list-item>
    </mat-list>
    <div class="button-container">
        <button mat-button class="button" (click)="getAllUploadedFiles()">Get all uploaded files</button>
        <button mat-button class="button" [disabled]="isClearDisabled()" (click)="clearList()">Clear list</button>
    </div>
</div>
```

On the left part of the component, there is a mat-list that displays the files inside of the **files** field.

Get all uploaded files button calls the **getAllUploadedFiles()** function that uses filesService's **getAllFiles()** method. It asks the backend API for all of the stored data and sets received data as a **files** field:



Clear list button is disabled as long as there are no files inside of the **files** field. After clicking it, the **clearList()** function is called that sets empty array to the **files** field:

```
clearList(): void {
  this.files = [];
  this.filesService.setFiles(this.files);
}
```

1.2.3 File uploader page component

File uploader page component is a wrapper for the application view. It consists of a **File Upload** component, a **divider** and a **Files Display** component.

Html code:

```
<main class="page-container">
    <app-file-upload></app-file-upload>
    <mat-divider></mat-divider>
    <app-files-display></app-files-display>
</main>
```

It sets general rules for application style.

1.2.4 File upload service

File upload service is a service that components use for **files** data storage and for calling the backend API. It consists of **setFiles()** function that sets the files passed as a parameter to the files field:

```
setFiles(files: IFile[]): void{
  this.files = files;
  this.onFilesChange.next(files);
}
```

Also it provides **getAllFiles()** and **addFile()** functions that use **httpClient** to call specific HTTP methods:

```
getAllFiles(): Observable<IFile[]> {
  return this.http.get<IFile[]>(this.apiUrl)
}

addFile(file:IFile): Observable<any> {
  const headers = { 'content-type': 'application/json'}

  return this.http.post(this.apiUrl, file, {'headers':headers});
}
```

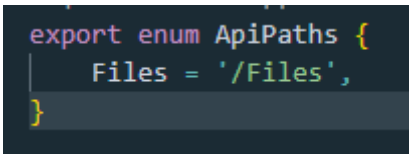
the apiUrl is passed to the service as a combination of environment variable and enum:

```
public apiUrl:string = environment.baseUrl + ApiPaths.Files;
```

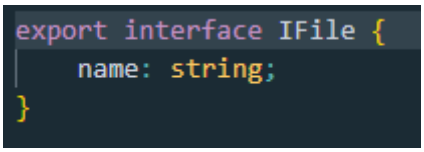
The **environment.baseUrl** variable is stored inside of the **environment.ts** file. **ApiPaths.Files** is an enum inside of the **app/enums** directory.

1.2.5 Enums and models

Project consists of an **ApiPaths** enum inside of **/enums** directory:

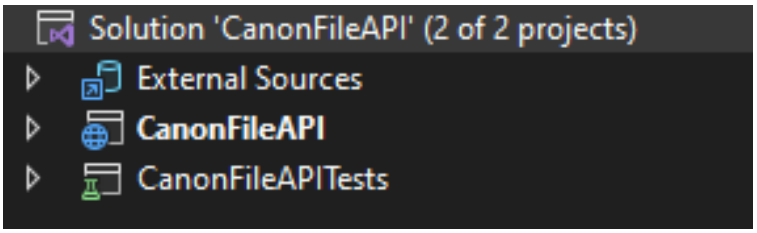


Application uses **IFile** data model from **/models** directory:



2. Back End Side

Backend side of the application was created in .NET 7 technology. It consists of two projects - CanonFileAPI and CanonFileAPITests:

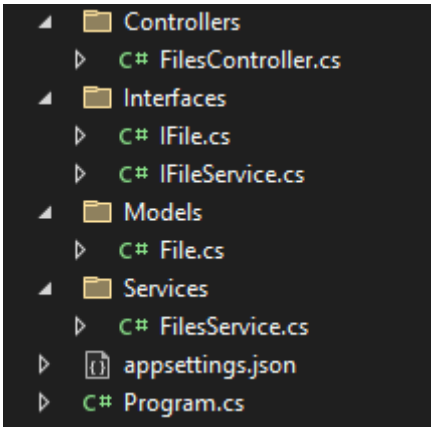


2.1 Used technologies

The CanonFileAPI and CanonFileAPITests projects were created with the .NET 7 framework. API uses OpenApi (swagger) NuGet package. For the testing purposes, NUnit and Moq packages were used. The whole application was created in Visual Studio 2022. For the API testing, Postman application was used.

2.2 Project structure

Project was split into few parts:



Inside the **Controllers** directory, there is **FilesController** API controller that is responsible for handling API calls. FilesController uses **FilesService** service as the business logic. Both structures take use of **File.cs** data model from the **Models**

directory. FilesService implements **IFileService** interface. The whole API is configured inside of the **Program.cs** file.

2.2.1 FilesController

FilesController consists of two endpoints that handle HTTP GET and HTTP POST methods:

```
// GET: api/<FilesController>
[HttpGet]
1 reference | 1/1 passing
public IEnumerable<IFile> Get()
{
    return this.filesService.GetFiles();
}

// POST api/<FilesController>
[HttpPost]
4 references | 4/4 passing
public IActionResult Post([FromBody] Models.File file)
{
    if (file == null || String.IsNullOrEmpty(file.Name))
    {
        return BadRequest("File or its properties are missing");
    }

    try
    {
        this.filesService.AddFile(file);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }

    return this.Ok();
}
}
```

Get() method handles **GET api/Files** calls and responds with **filesService.GetFiles()** method.

Post() method handles **POST api/Files** calls. It takes the Models.File object as a parameter, checks for the data correctness and tries to add a file using **filesService.AddFile()** method. If the operation is successful, it responds with **OK (200).** If not, it sends back a **BadRequest(400)** response with a proper message.

2.2.2 FilesService

FilesService consists of **files** data, **GetFiles()** method and **AddFile()** method:

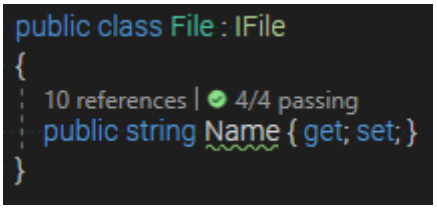
```
public class FilesService : IFileService
{
    public IList<IFile> files;

    0 references
    public FilesService()
    {
        this.files = new List<IFile>();
    }

    3 references | 1/1 passing
    public IList<IFile> GetFiles()
    {
        return this.files;
    }

    3 references | 1/1 passing
    public void AddFile(IFile file)
    {
        if (this.files.Any(x => x.Name == file.Name))
        {
            throw new ArgumentException("File already exists");
        }

        if (file.Name.IndexOfAny(Path.GetInvalidFileNameChars()) > -1)
        {
            throw new ArgumentException("Incorrect file name");
        }

        this.files.Add(file);
    }
}
```

GetFiles() method returns currently stored **files** data.

AddFile() takes a **file** as an argument, checks for possible duplicates, checks for invalid name characters and adds a file to the stored **files** data.

2.2.3 Models and interfaces

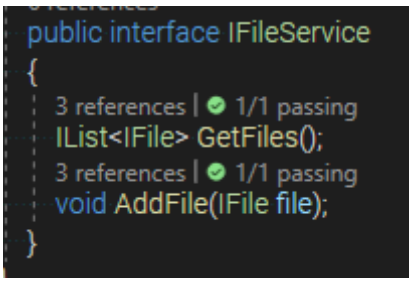
Project uses **File** data model that implements **IFile** interface:



File has a **Name** property that has a type of string.

IFile interface has a **Name** property:

FilesService implements **IFileService** interface:



It consists of a **GetFiles()** method that returns **IList<IFile>** collection and an **AddFile()** method that takes a **file** as a parameter.