# Upload Files in .Net Core Web Api

In many projects, we need to provide a possibility to our customers to upload different files in the app and to use them as well. So, this would be the topic for this blog post. We are going to upload files to the server (.NET Core Web API part) and then to use those files in our Angular client app.

For this post, we will stick to image files, but the logic is reusable for other file types as well.

We have created the starter project to work with through this blog post and it can be downloaded from (link). We strongly recommend you to download this project because it would be much easier to follow along with this blog post. In this project, we create a new user and show all the created users as well. Through this post, we are going to modify the create logic by adding an upload functionality and our new user will be created together with an image path related to it.

If you want to download our finished project, you can do that from (link).

This complete blog post is going to be divided into the following sections:

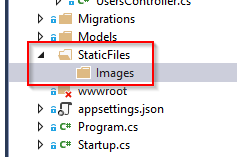
* Controller and Action Logic - .NET Core Part
* Upload File - Angular Part
* Using Uploaded File in Our Application

## Controller and Action Logic - .NET Core Part

After we have downloaded our starter project, we are going to open the UploadFilesServer project.

This project is created on top of the SQL database, so to create that database, we need to run the update-database command in a Package Manager. By doing this, our migrations will be executed and the database and table will be created.

The next step is to create a new folder StaticFiles and inside a new folder Images in the Solution Explorer window:



To continue, let’s create a new API Controller file in the Controllers folder and name it UploadController.

Let’s modify that file by adding a new action that will be responsible for the upload logic:

[HttpPost, DisableRequestSizeLimit]

public IActionResult Upload()

{

try

{

var file = Request.Form.Files[0];

var folderName = Path.Combine("StaticFiles", "Images");

var pathToSave = Path.Combine(Directory.GetCurrentDirectory(), folderName);

var dbPath = Path.Combine(folderName, file.FileName);

if (file.Length > 0)

{

var fileName = ContentDispositionHeaderValue.Parse(file.ContentDisposition).FileName.Trim('"');

var fullPath = Path.Combine(pathToSave, fileName);

using (var stream = new FileStream(fullPath, FileMode.Create))

{

file.CopyTo(stream);

}

return Ok(new { dbPath });

}

else

{

return BadRequest();

}

}

catch (Exception ex)

{

return StatusCode(500, "Internal server error");

}

}

We are using a POST action for the upload-related logic and disabling the request size limit as well. The logic inside this action is pretty straightforward. We extract the file from the request and providing the path where the file will be stored. Moreover, if the file is really there, we just take its name and provide a full path on the server to store file to and create a path to store in the database. This database path is going to be returned as a result of this action.

### Serving Static Files

Usually, all the files in the wwwroot folder are servable for the client applications. We provide that by adding app.UseStaticFiles() in the Startup class in the Configure method. Of course, our uploaded images will be stored in the StaticFiles folder, and due to that, we need to make it servable as well. To do that, let’s modify the Configure method in the Startup.cs class:

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseCors("CorsPolicy");

app.UseStaticFiles();

app.UseStaticFiles(new StaticFileOptions()

{

FileProvider = new PhysicalFileProvider(Path.Combine(Directory.GetCurrentDirectory(), @"StaticFiles")),

RequestPath = new PathString("/StaticFiles")

});

app.UseMvc();

}

And that is it. We have prepared our server side and it is time to jump right to the client side code.

## Upload File – Angular Part

Let’s open the UploadFilesClient project and take a look at the app component files. For the sake of simplicity, we have implemented all our logic inside the app component files.

So, the first thing we are going to do is to create a new Upload component in which we will handle all the upload-related logic:

ng g component upload --spec false

This will create three files in the upload folder, and we are going to modify the upload.component.ts file first:

import { Component, OnInit, Output, EventEmitter } from '@angular/core';

import { HttpEventType, HttpClient } from '@angular/common/http';

@Component({

selector: 'app-upload',

templateUrl: './upload.component.html',

styleUrls: ['./upload.component.css']

})

export class UploadComponent implements OnInit {

public progress: number;

public message: string;

@Output() public onUploadFinished = new EventEmitter();

constructor(private http: HttpClient) { }

ngOnInit() {

}

public uploadFile = (files) => {

if (files.length === 0) {

return;

}

let fileToUpload = <File>files[0];

const formData = new FormData();

formData.append('file', fileToUpload, fileToUpload.name);

this.http.post('https://localhost:5001/api/upload', formData, {reportProgress: true, observe: 'events'})

.subscribe(event => {

if (event.type === HttpEventType.UploadProgress)

this.progress = Math.round(100 \* event.loaded / event.total);

else if (event.type === HttpEventType.Response) {

this.message = 'Upload success.';

this.onUploadFinished.emit(event.body);

}

});

}

}

So, what are we doing here?

We create two variables to hold the message when upload action is finished and to show the upload progress. In the uploadFile function, we create a formData object and append our file that we want to update.

The next action is to send a post request and let’s pay attention to it. Besides the url and body properties, we have another JSON object which states that we want to track changes of our http request progress. As long as the upload is in progress, we will update the progress variable and show that percentage on the screen, but as soon as the upload is finished, we are going to write a message on the screen and emit the new event.

This event contains the body of our response, which is nothing more than the database path of our uploaded file. We need that path to show the uploaded image with other user details.

The files with small size will be instantly uploaded so, we will see 100% progress as soon as we choose our file. But for the larger files, the progress bar will update its values for sure.

Let’s modify the update.component.html file now:

<div class="row" style="margin-bottom:15px;">

<div class="col-md-3">

<input type="file" #file placeholder="Choose file" (change)="uploadFile(file.files)" style="display:none;">

<button type="button" class="btn btn-success" (click)="file.click()">Upload File</button>

</div>

<div class="col-md-4">

<span class="upload" \*ngIf="progress > 0">

{{progress}}%

</span>

<span class="upload" \*ngIf="message">

{{message}}

</span>

</div>

</div>

This logic is pretty straightforward except to the part where we hide the actual upload control and use its reference (#file) to invoke its click event with the button which looks much better. We could have styled the upload control as well, but this is the better way, at least from our side.

Finally, let’s modify the upload.component.css file:

.upload{

font-weight:bold;

color:#28a745;

margin-left: 15px;

line-height: 36px;

}

And add a selector from the update component to the app.component.html file:

<app-upload></app-upload>

<div class="row">

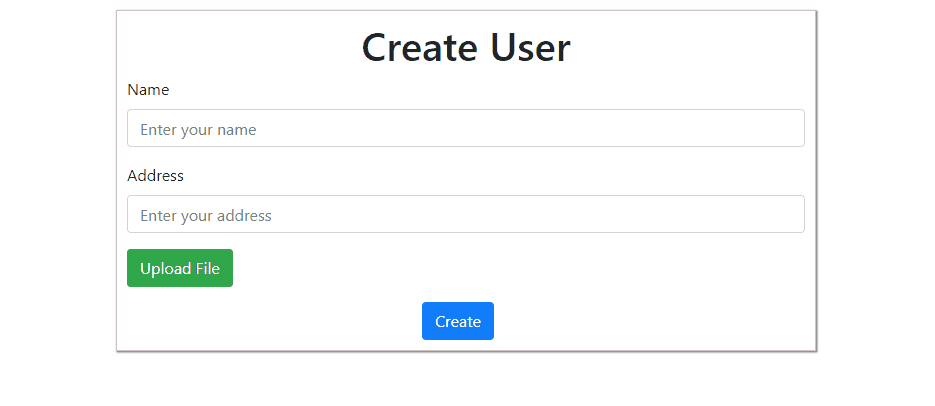
<div class="offset-md-5 col-md-2">

<button type="button" class="btn btn-primary" (click)="onCreate()">Create </button>

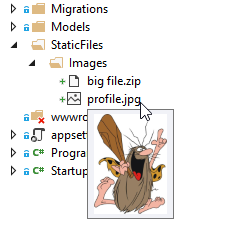
</div>

</div>

Excellent. We can now inspect our result:



We can check our StaticFiles/Images folder as well, to be sure that the file is really uploaded:



## Using Uploaded File in Our Application

As soon as we press the Create button on our form, we are going to see our newly created user. But its profile picture won’t be rendered. So, let’s fix that.

First, we need to react on the onUploadFinished event from the update component, and to do that let’s modify the app.component.html file:

<app-upload (onUploadFinished)="uploadFinished($event)"></app-upload>

This change forces us to modify the app.component.ts file as well.

First, let’s add an additional property in that file:

public response: {dbPath: ''};

Then let’s add the uploadFinished function to populate this property:

public uploadFinished = (event) => {

this.response = event;

}

With this modification, we have the response object in which we can find a path to be saved in the database.

Lastly, we have to modify the user object in the onCreate function in the same file:

this.user = {

name: this.name,

address: this.address,

imgPath: this.response.dbPath

}

Great job. Now we know the image file path related to the created user, so let’s use that knowledge to render that picture next to other user details.

To do that, let’s change a table inside the app.component.html file:

<table class="table table-striped">

<thead>

<tr>

<th scope="col">Image</th>

<th scope="col">Name</th>

<th scope="col">Address</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let user of users">

<td><img [src]="createImgPath(user.imgPath)" alt="profile picture" style="width:60px; height:60px;"></td>

<td>{{user.name}}</td>

<td>{{user.address}}</td>

</tr>

</tbody>

</table>

And let’s modify the app.component.ts file by adding the createImgPath function:

public createImgPath = (serverPath: string) => {

return `https://localhost:5001/${serverPath}`;

}

Our result should be as follows:

## 

## Conclusion