# ASP.NET Core MVC: File Upload, Download, and Viewing Files

This guide provides a comprehensive, step-by-step approach to implementing file upload, download, and viewing functionalities in an ASP.NET Core MVC application. It explains each part of the code in detail and provides instructions on how to integrate it into your project.

## Prerequisites

1. Visual Studio with ASP.NET Core support.

2. A SQL Server database for storing file information.

3. Entity Framework Core for data access.

## 1. Set Up the Database

### 1.1 Create a FileEntity Model

The `FileEntity` model represents the file data that will be stored in your database. It contains properties like `Id` (Primary Key), `FileName` (name of the file), `ContentType` (MIME type of the file), and `Data` (binary data of the file).

public class FileEntity  
{  
 public int Id { get; set; } // The unique identifier for each file (Primary Key).  
 public string FileName { get; set; } // The name of the uploaded file.  
 public string ContentType { get; set; } // The MIME type (e.g., 'application/pdf') of the uploaded file.  
 public byte[] Data { get; set; } // The binary data of the uploaded file.  
}

This model will allow us to store files in the database by converting the file to a byte array.

### 1.2 Configure the DbContext

In this step, we configure the `DbContext` to manage our file data in the database. We need to include the `FileEntity` model as a `DbSet<FileEntity>` property.

public class AppDbContext : DbContext  
{  
 public DbSet<FileEntity> Files { get; set; } // The Files DbSet is used to interact with the 'Files' table in the database.  
  
 public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }  
}

The `DbContext` class is a key component of Entity Framework Core that enables us to interact with the database.

## 2. Implement File Upload

The following steps explain how to implement file upload functionality. This allows users to upload files from their local system to the server.

### 2.1 Create the Upload View

We begin by creating a Razor view (`Upload.cshtml`) that provides a file input field for users to select a file and upload it.

<form asp-action="UploadFile" asp-controller="File" method="post" enctype="multipart/form-data">  
 <div class="form-group">  
 <label for="fileInput">Select File</label>  
 <input type="file" name="uploadedFile" id="fileInput" class="form-control" />  
 </div>  
 <button type="submit" class="btn btn-primary">Upload</button>  
</form>

This form sends the file data to the `UploadFile` action method of the `FileController`. The `enctype='multipart/form-data'` attribute allows the form to send binary data (files).

### 2.2 Implement the Upload Action

Next, we create an action in the `FileController` to handle the uploaded file. The file is converted into a byte array and saved to the database.

[HttpPost("UploadFile")]  
public async Task<IActionResult> UploadFile(IFormFile uploadedFile)  
{  
 if (uploadedFile == null || uploadedFile.Length == 0)  
 {  
 ModelState.AddModelError("", "Please select a file to upload.");  
 return View("Upload");  
 }  
  
 using (var memoryStream = new System.IO.MemoryStream())  
 {  
 await uploadedFile.CopyToAsync(memoryStream);  
 var fileData = memoryStream.ToArray();  
  
 var fileEntity = new FileEntity  
 {  
 FileName = uploadedFile.FileName,  
 ContentType = uploadedFile.ContentType,  
 Data = fileData  
 };  
  
 \_context.Files.Add(fileEntity);  
 await \_context.SaveChangesAsync();  
 }  
  
 return RedirectToAction("Index");  
}

The `UploadFile` method handles the file upload. If no file is selected, it adds an error. Then, it converts the file content into a byte array and saves the file in the database.

## 3. Implement File Download

The following steps explain how to implement file download functionality.

### 3.1 Add the Download Action

The `DownloadFile` method retrieves the file from the database by its ID and sends it to the client as a download.

[HttpGet("DownloadFile/{id}")]  
public async Task<IActionResult> DownloadFile(int id)  
{  
 var file = await \_context.Files.FirstOrDefaultAsync(f => f.Id == id);  
 if (file == null)  
 {  
 return NotFound();  
 }  
  
 return File(file.Data, file.ContentType, file.FileName);  
}

The `DownloadFile` action fetches the file's binary data from the database and serves it with the appropriate MIME type. It allows the user to download the file by name.

## 4. Display Files in a List

Now we will show how to list all files from the database and provide download links.

### 4.1 Add an Index Action

The `Index` action retrieves all files from the database and passes them to the view.

[HttpGet("Files")]  
public async Task<IActionResult> Index()  
{  
 var files = await \_context.Files.ToListAsync();  
 return View(files);  
}

### 4.2 Create the Index View

In the `Index.cshtml` view, we display the list of files with download links.

<h2>Available Files</h2>  
<table class="table table-striped">  
 <thead>  
 <tr>  
 <th>File Name</th>  
 <th>Content Type</th>  
 <th>Action</th>  
 </tr>  
 </thead>  
 <tbody>  
 @foreach (var file in Model)  
 {  
 <tr>  
 <td>@file.FileName</td>  
 <td>@file.ContentType</td>  
 <td>  
 <a class="btn btn-primary" href="@Url.Action("DownloadFile", "File", new { id = file.Id })">Download</a>  
 </td>  
 </tr>  
 }  
 </tbody>  
</table>

This Razor view displays a table listing the available files, their content types, and a button to download each file.