

NIT 3004 - IT Capstone Project 2
Semester 2, Block 4

FINAL REPORT

CASTING WEB APPLICATION -

ROOMCAST



Group Members:

Aleksandar Ivanov - s8090291

Nathan Goncalves - s8081735

Lecturer:

Dr Gongqi Lin

Contents

Contents.....	2
1. Introduction.....	4
1.1 Functional Requirements.....	6
1.2 Non-Functional Requirements.....	6
2. Architecture Overview.....	7
2.1 System Design.....	7
2.2 Technologies used.....	9
2.3 Modules and Key Functions.....	10
2.3.1 Authentication and User Management Module.....	10
2.3.1.1 Model - ApplicationUser.cs.....	11
2.3.1.2 View Model - LoginViewModel.cs.....	11
2.3.1.3 Controller - AccountController.cs.....	12
2.3.1.4 Logic and Flow.....	13
2.3.2 Media Upload Module.....	14
2.3.2.1 Model MediaFile.cs.....	14
2.3.2.2 View Upload.cshtml.....	14
2.3.2.3 Controller MediaFilesController.cs.....	15
2.3.2.4 Logic and Flow.....	16
2.3.3 Media Edit Module.....	17
2.3.3.1 Model MediaFile.cs.....	17
2.3.3.2 View Edit.cshtml.....	17
2.3.3.3 Controller MediaFilesController.cs.....	18
2.3.3.4 Logic and Flow.....	18
2.3.4 Album creation and Management Module.....	19
2.3.4.1 Model Album.cs and AlbumFile.cs.....	20
2.3.4.2 View Create.cshtml and Details.cshtml.....	21
2.3.4.3 Controller AlbumController.cs.....	22
2.3.4.4 Logic and Flow.....	22
2.3.5 Screen Module.....	23
2.3.6 Assigning Album and Media Files to Screens.....	24
2.3.6.1 Models Screen.cs , ScreenMediaAssignment.cs and AlbumScreenAssignment.cs.....	24
2.3.6.2 Views AssignToScreen.cshtml and AssignMedia.cshtml.....	24
2.3.6.3 Controller AlbumsController.cs and ScreenController.cs.....	26
2.3.6.4 Logic and Flow.....	27
2.3.7 Electron Display Client App.....	28
2.3.7.1 Key Code main.js.....	28

2.3.7.2 Logic and Flow.....	29
2.4 Database.....	29
3. Implementation Strategy.....	31
3.1 Key Programming Concepts.....	31
3.2 Key Data Structures Used.....	32
3.3 External APIs and Libraries.....	32
4. System Configuration and Setup.....	33
4.1 Configuration files.....	33
4.2 Deployment Process.....	33
4.2.1 Server Side.....	33
4.2.2 Client Side.....	35
5. Testing and Validation.....	36
5.1 Testing Strategies.....	36
5.1.1 Manual Functional Testing.....	37
5.1.2 Black-Box Testing.....	37
5.1.3 Integration Testing (Manual).....	37
5.1.4 UI/UX Testing.....	38
5.1.5 Error-Case Testing.....	38
Server:.....	38
Client:.....	39
5.2 Testing Tools.....	39
5.2.1 Browser Developer Tools.....	39
5.2.2 EF Core Logging.....	39
5.2.3 Electron Debug Console.....	40
5.3 Validation Criteria.....	40
5.3.1 Authentication Module.....	40
5.3.2 Media Upload Module.....	40
5.3.3 Media Editing Module.....	41
5.3.4 Albums Module.....	41
5.3.5 Screens & Casting Module.....	41
5.3.6 Electron Display Client.....	42
5.3.7 Deployment & Startup.....	42
6. Challenges and Solutions.....	42
7 Future Enhancements.....	44
7.1 Database Scalability.....	45
7.2 Real-Time Casting Using SignalR WebSockets.....	45
Benefits:.....	45
7.2 Support for More Media Types.....	45
7.3 User Roles & Permissions.....	46

7.4 Cloud Deployment Option.....	46
7.5 Automated Installer (MSI/EXE).....	46
8. Conclusion.....	47

1. Introduction

This purpose of this document is to provide an in-depth technical overview of the **RoomCast** system, detailing its design, development, and deployment processes. This document serves as a comprehensive reference for developers, evaluators, and technical stakeholders by outlining the core architectural decisions, programming methodologies, and technologies used to implement the system.

We will highlight the structural and functional components of the project, including backend logic, database schema, user interface design, and integration mechanisms between the web application and the client display system. It explains how each part of the system contributes to the seamless management and presentation of media content across multiple screens in real time. By presenting a detailed account of the coding standards, frameworks, and implementation steps, the document ensures that the RoomCast platform can be easily maintained, scaled, and enhanced in the future. It also demonstrates adherence to modern software engineering principles, including modular architecture, security-driven design, and user-centric usability.

RoomCast is a **multi-screen media management system** developed using **ASP.NET Core MVC**, **Entity Framework Core**, and **SQLite**. The application enables users to upload, categorize, edit, and display various types of media - including **images, videos, and documents** across one or more connected screens or smart displays.

The primary goal of the project is to provide an efficient and intuitive digital signage solution for environments such as educational institutions, business offices, event venues and fairs. Traditional digital signage systems often rely on manual screen updates, USB transfers, or third-party subscription services that limit flexibility and scalability. **RoomCast** addresses these challenges by offering a **centralized, web-based platform** where administrators can control what content appears on each display in real time.

The project was conceived to solve three key problems:

1. **Lack of centralized management:** Existing systems require individual screen updates or separate logins, creating inefficiency. **Limited content control:** Many platforms restrict supported media types or real-time customization options. **Accessibility and affordability:** Commercial solutions are often expensive or locked behind proprietary hardware and software.

RoomCast overcomes these issues by offering:

- A **secure login and registration system** using ASP.NET Core Identity.
- A **robust media management module** supporting file upload, preview, and metadata storage.
- **Album creation and organization**, allowing media grouping for specific campaigns or themes.
- A **screen assignment system** where administrators can assign single files or entire albums to different displays.
- A **client application** (built using Electron/Node.js) that displays assigned media content on target screens without requiring manual browser control.

This application is worth building because it bridges the gap between **enterprise-level digital signage** and **affordable open solutions**, providing an adaptable tool for small to medium-sized organizations. Its modular architecture and lightweight database make it ideal for deployment on local servers, mini-PCs, or embedded systems such as Raspberry Pi — enabling RoomCast to function as a scalable, flexible, and cost-effective signage ecosystem.

1.1 Functional Requirements

Initially in the project proposal document we have listed 5 main functions on this project, as the project was in the developing process it came to a need to implement few more functions that are critical for the project to achieve its purpose.

1. User Authentication - Login and Registration, Password hashing and validation.
2. Media Upload - Uploading images, videos and document files with metadata
3. Media Edit - Modifying file names, replace content or delete media
4. Album Creation and Management - Group multiple media files into albums
5. Screen Management - Registering multiple screens in the system
6. Media Assignment to Screens - Specific media can be assigned to one or more screens
7. Casting - Client app that retrieves and display assigned media on target screens.

1.2 Non-Functional Requirements

-
1. Users must create a strong password that meets complexity requirements. Passwords must be securely hashed and stored in the database to protect user data.
 2. The media upload system must support concurrent uploads of files (documents, images, videos) with a maximum size of 50 MB and a response time of 3 seconds for validation.
 3. It should ensure 99.9% availability, provide user-friendly error messages for failed uploads.
 4. The system should be compatible with Google Chrome and Microsoft Edge and maintain a modular architecture for future updates.
 5. Display user-friendly error messages for unsupported file formats or network issues, with suggestions for resolution (e.g., “Please select a supported format: MP4, AVI, JPEG”).
 6. Server-side validation should detect unsupported formats before display, with errors logged for monitoring and system diagnostics.
 7. User actions for media control (e.g., play, pause, stop, loop) must execute within 1 second to ensure real-time responsiveness on each selected screen. User can only see all files uploaded by the user himself
 8. User experience: creating an error or successful message after editing.
 9. No loss of uploaded files or metadata during operations.
 10. Users should receive immediate feedback for all actions.
 11. Critical user actions (upload, edit, delete, assign) must be traceable.

2. Architecture Overview

2.1 System Design

The RoomCast system follows a multi-tier, client–server architecture based on the Model–View–Controller (MVC) design pattern. This approach ensures modularity, scalability, and separation of concerns between the user interface, business logic, and data management components.

The ASP.NET Core MVC framework provides the foundation for the RoomCast web application. This pattern divides the application into three interconnected components:

- ❖ **Model Layer:** Represents the data and business logic of the system, including entities such as MediaFile, Album, Screen, and ScreenMediaAssignment.
The models are implemented using **Entity Framework Core (EF Core)** with relationships such as one-to-many, many-to-one and many-to-many.
- ❖ **View Layer:** Handles the user interface and presentation logic.
Built using **Razor Pages**, **Bootstrap 5**, and partial views to render responsive UI components such as media galleries, upload forms, and preview modals.
- ❖ **Controller Layer:** Acts as an intermediary between the View and Model layers.
Controllers such as MediaFilesController, AlbumsController, and ScreensController process HTTP requests, execute business logic, interact with the database, and return appropriate views or JSON responses.

The overall system is modular and consists of two primary components:

1. RoomCast Web Application (Server-side):
 - Built with ASP.NET Core 8 MVC.
 - Manages user authentication, file uploads, database operations, album creation, and media assignments.
 - Hosts a lightweight REST API that allows external clients (e.g., the RoomCast Display App) to fetch assigned media dynamically.
2. RoomCast Display Client (Client-side):
 - A desktop application built using Electron.js and Node.js.
 - Connects to the RoomCast API to retrieve and display assigned media (images, videos, PDFs, etc.) on target screens in real time.
 - Operates in kiosk mode for full-screen playback, suitable for smart TVs, mini-PCs, or Raspberry Pi devices.

2.2 Technologies used

Visual Studio Code - Visual Studio Code served as the primary Integrated Development Environment (IDE) for the entire RoomCast development lifecycle. It was selected for its lightweight architecture, cross-platform compatibility, and powerful extension ecosystem, which made it ideal for working on both the ASP.NET Core MVC backend and the Electron.js client within a single, unified workspace.

C# - C# serves as the backbone of the RoomCast web application. It handles the core logic within the MVC architecture, such as model definitions, data validation, and business rules. Its strong typing, LINQ support, and compatibility with the .NET ecosystem make it ideal for large-scale, maintainable projects.

JavaScript - JavaScript powers client-side interactivity, asynchronous AJAX requests, and instant updates to the interface. It plays a crucial role in enabling dynamic media previews, live screen assignments, and content refresh without page reloads.

Node.js / Electron.js - The Electron based RoomCast Client uses Node.js as its runtime environment to fetch and render assigned media files. Electron combines Chromium and Node.js to provide a desktop-like environment for full-screen playback, ensuring the media content is displayed accurately and reliably on different display screens or kiosks.

HTML5 / Razor Syntax - The Razor engine integrates server-side logic directly within the HTML structure, allowing dynamic content rendering. This enables interactive and data-driven views, where user actions (like uploading media or creating albums) reflect immediately on the page without full reloads.

CSS / Bootstrap 5 - The frontend interface is built using Bootstrap 5 to achieve responsiveness, visual consistency, and compatibility with multiple device resolutions. CSS classes ensure clean layouts for media previews, forms, modals, and navigation bars, improving user experience.

.NET 8 (ASP.NET Core MVC) - ASP.NET Core provides the foundation of the system's server-side logic. Its MVC framework enforces a clean separation between data, presentation, and control flow, enabling modular development. The use of .NET 8 ensures modern performance benefits, built-in security mechanisms, and long-term support from Microsoft.

Entity Framework Core (EF Core) - EF Core serves as the ORM (Object Relational Mapper), translating database entities into strongly-typed C# objects. It simplifies database operations such as CRUD (Create, Read, Update, Delete), schema migrations, and relationship mapping between models like MediaFile, Album, and Screen.

ASP.NET Core Identity - This framework handles user registration, authentication, and authorization. It provides secure password hashing, role management and cookie-based session handling, ensuring that only authorized users can upload, edit, or assign media.

SQLite- A lightweight, self-contained SQL database engine, SQLite was selected for its simplicity, portability, and zero-configuration nature. It allows the RoomCast system to store media metadata, user information, and screen assignments without the need for complex database setups.

Git and GitHub - Used for version control and collaboration among the group members. Git tracks code changes, while GitHub serves as a remote repository for backup and coordination between team members.

2.3 Modules and Key Functions

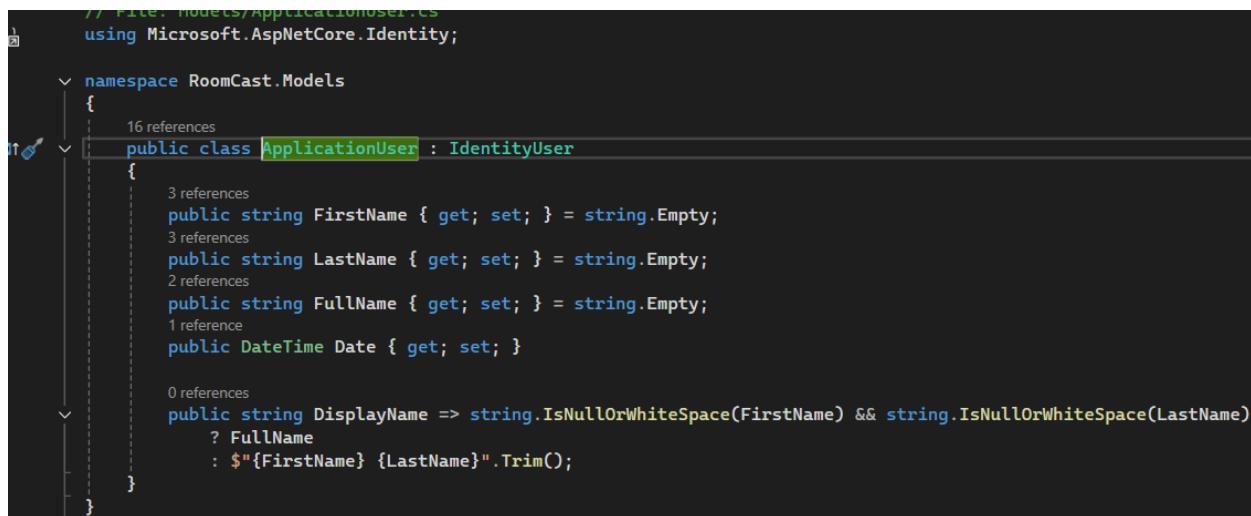
The RoomCast system is structured into several core modules that collectively deliver a seamless media management and screen display experience. Each module is designed following separation of concerns, ensuring maintainability, testability, and scalability. This section describes the core modules, their responsibilities, and the key programming logic used to implement them.

2.3.1 Authentication and User Management Module

This module handles user registration, login, logout and access control using ASP.NET Core Identity. It ensures that only authenticated users can access sensitive operations such as uploading, editing, or assigning media. Users are stored in the Identity database tables (AspNetUsers, AspNetRoles, etc.), and each user is actually assigned as admin. The web app doesn't have role based control.

2.3.1.1 Model - ApplicationUser.cs

This class inherits from `IdentityUser` and may include additional fields such as `FullName`, `DateRegistered`, etc., extending the default ASP.NET Core Identity system. Defines the structure of user data stored in the database.



```
// FILE: Models/ApplicationUser.cs
using Microsoft.AspNetCore.Identity;

namespace RoomCast.Models
{
    16 references
    public class ApplicationUser : IdentityUser
    {
        3 references
        public string FirstName { get; set; } = string.Empty;
        3 references
        public string LastName { get; set; } = string.Empty;
        2 references
        public string FullName { get; set; } = string.Empty;
        1 reference
        public DateTime Date { get; set; }

        0 references
        public string DisplayName => string.IsNullOrWhiteSpace(FirstName) && string.IsNullOrWhiteSpace(LastName)
            ? FullName
            : $"{FirstName} {LastName}".Trim();
    }
}
```

Figure 1 - Model ApplicationUser.cs with its entities

2.3.1.2 View Model - LoginViewModel.cs

Collects login data from user through the login form. ViewModels are used to decouple user input from the database structure.

```
using System.ComponentModel.DataAnnotations;

namespace RoomCast.Models.ViewModels
{
    3 references
    public class LoginViewModel
    {
        [Required, EmailAddress]
        4 references
        public string Email { get; set; } = string.Empty;

        [Required]
        [DataType(DataType.Password)]
        4 references
        public string Password { get; set; } = string.Empty;
        3 references
        public bool RememberMe { get; set; }

    }
}
```

Figure 2 - View Model - LoginViewModel.cs

2.3.1.3 Controller - AccountController.cs

The AccountController.cs authenticates the user based on submitted email and password. If successful redirects the user to specified return URL.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Login(LoginViewModel model, string? returnUrl = null)
{
    if (!ModelState.IsValid)
        return View(model);

    var result = await _signInManager.PasswordSignInAsync(
        model.Email,
        model.Password,
        model.RememberMe,
        lockoutOnFailure: true
    );
    if (result.Succeeded)
    {
        if (!string.IsNullOrEmpty(returnUrl) && Url.IsLocalUrl(returnUrl))
            return Redirect(returnUrl);
        return RedirectToAction("Index", "Home");
    }
    if (result.IsLockedOut)
        ModelState.AddModelError("", "Account locked. Please try again later.");
    else
        ModelState.AddModelError("", "Invalid login attempt. Please try again.");
    return View(model);
}
```

Figure 3 - Controller - AccountController.cs

2.3.1.4 Logic and Flow

User submits login form. Controller checks if the model is valid (client/server validation). If valid, PasswordSignInAsync attempts authentication. If successful, user is redirected to the protected area. If failed, an error message is shown.

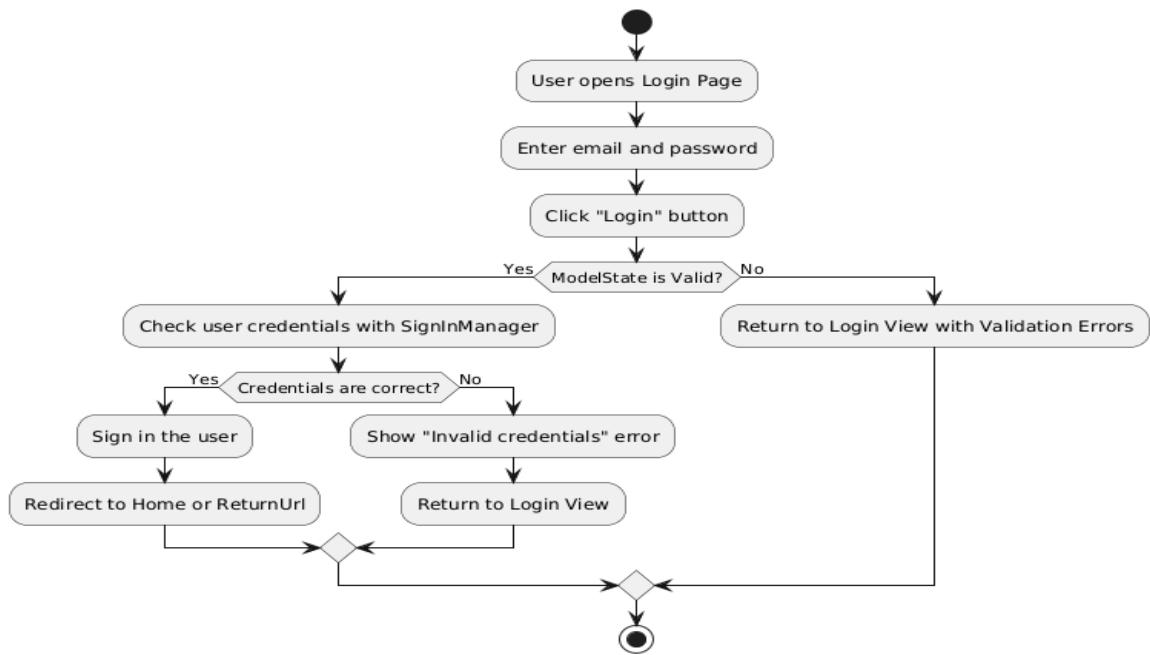


Figure 4 - Flow Chart of User Authentication Module

Parameters and Values

LoginViewModel model: Contains user input for email, password, and remember me.

string? returnUrl: Optional redirection path after login.

On success: RedirectToLocal(returnUrl) → returns a redirect to Home/Index or another secure area.

On failure: returns the login view with error.

[ValidateAntiForgeryToken] protects against CSRF attacks.

_signInManager is injected via dependency injection and manages identity operations.

ASP.NET Identity also handles cookies and tokens under the hood.

2.3.2 Media Upload Module

The Upload module is a core future of the RoomCast platform that allows authenticated users to upload various media files such as images, videos and documents to the system. These uploaded files are stored in the file system and registered in the database with metadata like title, description, file type and tags. Uploaded media can later be previewed, grouped into albums, editor and assigned to specific screens. This module ensures file validation, file path generation and secure storage, providing a streamlined user experience for managing digital content.

2.3.2.1 Model MediaFile.cs

The [MediaFile.cs](#) model defines the structure of uploaded content stored in the database. Each media file has a unique ID, title, description, file path, file type, timestamp and uploader identity.

```
[Required]
[MaxLength(200)]
26 references
public string Title { get; set; } = string.Empty;

[Required]
[MaxLength(20)]
29 references
public string FileType { get; set; } = string.Empty; // Document, Picture, Video

[Required]
[MaxLength(20)]
12 references
public string FileFormat { get; set; } = string.Empty; // .docx, .pdf, .jpg, etc.

[MaxLength(255)]
```

Figure 5 - MediaFile.cs with its few entities shown

2.3.2.2 View Upload.cshtml

The Upload view provides a form for users to select and upload a file. It also collects metadata such as the title and description. The form uses multipart/form-data encoding to handle file transmission.

```
<form asp-action="Upload" method="post" enctype="multipart/form-data" class="space-y-6">
    <div asp-validation-summary="ModelOnly" class="validation-summary rounded-lg border border-red-200 bg-red-50 p-4 t
```

Figure 6 - Code snippet shows the form using multipart/form-data encoding to handle file transmission.

2.3.2.3 Controller MediaFilesController.cs

The controller handles file validation, processing and saving. It ensures that files are stored securely and uniquely to prevent collisions.

```
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Upload(FileUploadViewModel model)
{
    PopulateAllowedExtensions(model);

    if (!ModelState.IsValid)
    {
        return View(model);
    }

    if (model.File == null || model.File.Length == 0)
    {
        ModelState.AddModelError(nameof(model.File), "Please choose a file to upload.");
        return View(model);
    }

    if (model.File.Length > MaxFileSizeBytes)
    {
        ModelState.AddModelError(nameof(model.File), $"File exceeds the {MaxFileSizeBytes / (1024 * 1024)}MB limit.");
        return View(model);
    }

    var extension = Path.GetExtension(model.File.FileName).ToLowerInvariant();

    var normalizedType = GetFileTypeForExtension(extension);
    if (normalizedType == null)
    {
        ModelState.AddModelError(nameof(model.File), $"Unsupported file format. Allowed formats: {FormatAllowedExtensions}");
        return View(model);
    }

    var title = model.Title?.Trim();
    if (string.IsNullOrWhiteSpace(title))
    {
        title = Path.GetFileNameWithoutExtension(model.File.FileName);
    }
```

Figure 7 - Part of the code in the MediaFilesController

2.3.2.4 Logic and Flow

The upload logic follows a structured sequence

The user navigates to upload page. They fill in the title and optionally a description. A file is selected and submitted through the form. The controller validates the file existence and size. A unique filename is generated using GUID. The file is saved in the database. The user is redirect to the “My Files” view for confirmation. The logic ensures that duplicate files dont overwrite each other and that metadata is always consistent.

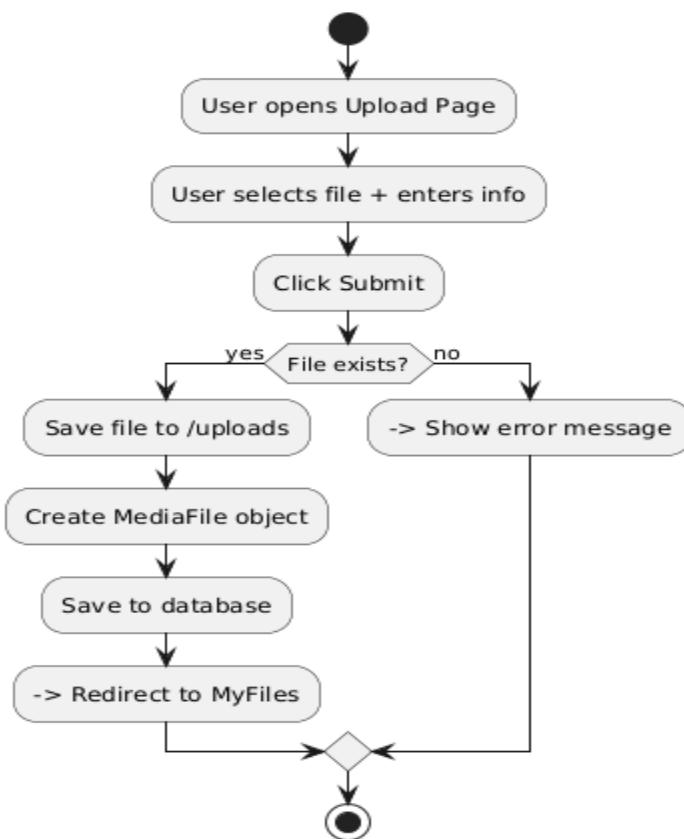


Figure 8 - Flow Chart of Media Upload Module

Parameters and Return Values

IFormFile file: The file to be uploaded (image, video, PDF)

string Title: Title input by the user

string Description: Optional description input by the user

Task<ActionResult>: On success, redirects to MyFiles view; otherwise, re-renders the form.

2.3.3 Media Edit Module

The Edit Media module enables users to modify existing media files after they've been uploaded. This includes changing titles and descriptions or replacing the file itself. It ensures flexibility for correcting mistakes, improving file versions or updating media content for display or album purposes. This module operates securely by validating input and verifying ownership.

2.3.3.1 Model MediaFile.cs

This model is reused as same as in Upload Module. Refer to **Figure 5**

2.3.3.2 View Edit.cshtml

The view shows a form to modify the file's metadata or replace the file.

```
<form method="post" asp-action="SaveText" class="space-y-5">
    @Html.AntiForgeryToken()
    <input type="hidden" asp-for=" fileId" />
    <input type="hidden" asp-for=" title" />
    <div>
        <label asp-for="Content" class="mb-2 block text-sm font-semibold text-slate-700"></label>
        <textarea asp-for="Content"
            rows="24"
            class="w-full rounded-xl border border-slate-200 bg-white px-4 py-3 text-sm font-mono leading-relaxed
            spellcheck="false"
            data-text-editor></textarea>
        <span asp-validation-for="Content" class="mt-2 block text-sm text-red-600"></span>
        <div class="mt-2 flex items-center justify-between text-xs text-slate-500">
            <span>Content type: @Model.ContentType</span>
            <span data-text-editor-count=0 characters</span>
        ...</div>
    </div>
</form>
```

Figure 9 - Code snippet for text editing

```
<form method="post"
    asp-action="SaveTrimmedVideo"
    class="space-y-6"
    data-trim-form>
    @Html.AntiForgeryToken()
    <input type="hidden" name=" fileId" value="@Model.FileId" />
    <input type="hidden"
        name="StartSeconds"
        value="@trimStart.ToString("0.###", CultureInfo.InvariantCulture)"
        data-trim-start-hidden />
    <input type="hidden"
        name="EndSeconds"
        value="@trimEnd.ToString("0.###", CultureInfo.InvariantCulture)"
        data-trim-end-hidden />
    <input type="hidden"
        name="SourceDurationSeconds"
        value="@effectiveDuration.ToString("0.###", CultureInfo.InvariantCulture)"
        data-trim-duration-hidden />
</form>
```

Figure 10 - Code snippet for editing videos

2.3.3.3 Controller MediaFilesController.cs

This action handles updates to title, description and optionally the file.

```
[HttpGet]  
1 reference  
public async Task<IActionResult> EditText(int id)  
{  
    var user = await _userManager.GetUserAsync(User);  
    if (user == null)  
    {  
        return Challenge();  
    }  
  
    var mediaFile = await _context.MediaFiles  
        .Where(m => m.FileId == id && m.UserId == user.Id)  
        .FirstOrDefaultAsync();
```

Figure 11 - Code snippet from the MediaFiles controller for text editing

2.3.3.4 Logic and Flow

User navigates to My Files page. The system retrieves the file by FileId. The user modifies the title/description or uploads a new version. The existing database record is updated. User redirected back to My Files.

The logic maintains the original file's identity (ID) while allowing flexible updates

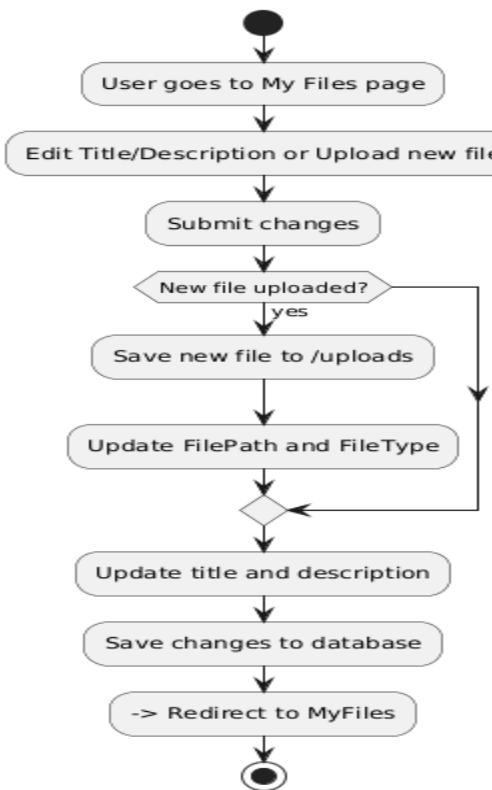


Figure 12 - Flow of events for Edit

Parameters and Return Values

int FileId: ID of the file to update.

string Title: New or updated title.

string Description: New or updated description.

IFormFile NewFile: (Optional) new file to replace existing media.

Task<ActionResult>: Redirects to MyFiles if successful; otherwise returns appropriate error view.

2.3.4 Album creation and Management Module

The Album Creation and Management module allows users to group multiple media files into albums for better organization and batch display. Albums can be named, timestamped, and updated with new media. This feature is crucial for categorizing files (e.g., by topic, session, or time) and managing media in bulk when assigning to screens.

Albums support adding/removing files, viewing content, deleting albums, and assigning them to screens.

2.3.4.1 Model Album.cs and AlbumFile.cs

Album.cs (Figure 13) represents the album metadata, while AlbumFile.cs (Figure 14) represents the many-to-many relationship between an album and its media files.

```
public class Album
{
    [Key]
    14 references
    public int AlbumId { get; set; }

    [Required]
    [StringLength(100)]
    13 references
    public string AlbumName { get; set; } = string.Empty;
    5 references
    public DateTime Timestamp { get; set; } = DateTime.UtcNow;
    4 references
    public string UserId { get; set; } = string.Empty;
    [ForeignKey("UserId")]
    0 references
    public ApplicationUser? User { get; set; }
    7 references
    public ICollection<AlbumFile> AlbumFiles { get; set; } = new List<AlbumFile>();
}
```

Figure 13 - Class [Album.cs](#)

```
public class AlbumFile
{
    [Key]
    0 references
    public int AlbumFileDialog { get; set; }

    [ForeignKey(nameof(Album))]
    2 references
    public int AlbumId { get; set; }

    [ForeignKey(nameof(MediaFile))]
    2 references
    public int FileId { get; set; }
    1 reference
    public Album Album { get; set; } = null!;
    5 references
    public MediaFile MediaFile { get; set; } = null!;
}
```

Figure 14 - Class [AlbumFile.cs](#)

2.3.4.2 View Create.cshtml and Details.cshtml

Create.cshtml lets user create and define album name.

Details.cshtml shows album metadata and the included media files with options to add and remove files.

```
model RoomCast.Models.Album
 @{
    ViewData["Title"] = "Create Album";
}



<div>
        <h2 class="text-3xl font-semibold text-slate-900">Create Album</h2>
        <p class="text-sm text-slate-500">Give your album a descriptive name to make it easy to find later</p>
    </div>

    <form asp-action="Create" method="post" class="space-y-5">
        @Html.AntiForgeryToken()
        <div asp-validation-summary="All" class="text-sm text-red-600"></div>
        <div class="space-y-2">
            <label asp-for="AlbumName" class="text-sm font-semibold text-slate-700"></label>
            <input asp-for="AlbumName" class="w-full rounded-lg border border-slate-300 bg-white px-4 py-2 text-sm text-slate-700" type="text"/>
            <span asp-validation-for="AlbumName" class="text-sm text-red-600"></span>
        </div>

        <button type="submit" class="inline-flex items-center justify-center rounded-md bg-emerald-600 px-4 py-2 text-white font-semibold">Save</button>
    </form>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}


```

Figure 15 - Code for creating Albums

2.3.4.3 Controller AlbumController.cs

The controller handles the creation, display and media linking to albums.

```
// | Add Media (POST)
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> AddMedia(int albumId, int fileId)
{
    var exists = await _context.AlbumFiles
        .AnyAsync(x => x.AlbumId == albumId && x.FileId == fileId);

    if (!exists)
    {
        _context.AlbumFiles.Add(new AlbumFile
        {
            AlbumId = albumId,
            FileId = fileId
        });
        await _context.SaveChangesAsync();
    }

    TempData["Message"] = "Media added to album successfully.";
    return RedirectToAction(nameof(Details), new { id = albumId });
}

// ☑ Album Details
[HttpGet]
1 reference
public async Task<ActionResult> Details(int id)
```

Figure 16 - Code snippet with the CRUD operations that the controller handles it. The focus here is on adding media to the album

2.3.4.4 Logic and Flow

Use navigates to Album page to create new album. System stores album metadata in Albums table. User is redirected to the created album with its ID. From there user can assign media to the album, view all media added to the album, remove media and assign the album to a screen. Or can delete album.

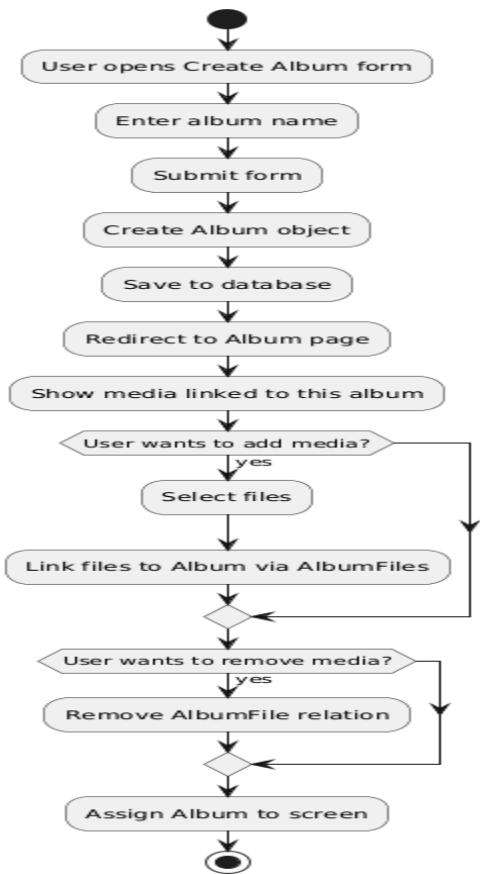


Figure 17 - Flow of events for creating album and management

Parameters and Return Values

string Name: Album name input from user.

Guid id: Album ID used to fetch or update a specific album.

Task<IActionResult>: View with album data or redirection to detail/edit pages.

2.3.5 Screen Module

This module is implemented so the user can create screens, assign media files or albums.

Created screen with specific name and description is later pushed to the client app(this app will be already programmed or installed on the preferred real screen like TVs or Billboards for casting). Displaying the content on the screen is the last and final stage of the purpose of this web application.

2.3.6 Assigning Album and Media Files to Screens

This module enables users to assign either complete albums or individual media files (images, videos, documents) to specific screens, allowing for flexible and organized content display. The functionality is divided into two parts:

- **Assign Albums to Screens:** Link an entire pre-created album to one or more digital signage screens. Ideal for grouped thematic or scheduled content.
- **Assign Media Files Individually:** Enables ad-hoc media casting where the user selects specific media files without bundling them in albums.

This dual approach supports both structured and spontaneous content distribution, suitable for campuses, businesses, events, and classrooms.

2.3.6.1 Models Screen.cs , ScreenMediaAssignment.cs and AlbumScreenAssignment.cs

Screen.cs class contains properties such ScreenId, Name, Location and Description

ScreenMediaAssignment.cs contains properties such Id, ScreenId (foreign key) from [Screen.cs](#) class, Screen, MediaField and MediaFile

AlbumScreenAssignment contains Id (foreign key to [Album.cs](#)), AlbumID, Album and ScreenID

2.3.6.2 Views AssignToScreen.cshtml and AssignMedia.cshtml

Assign Album Figure 18

Assign Media Figure 19

```

@model RoomCast.Models.ViewModels.AlbumAssignmentViewModel

@if
    ViewData["Title"] = "Assign Album to Screen";
    Layout = "_Layout";
}



<h2 class="text-2xl font-semibold mb-6 text-slate-800">Assign Album to Screen</h2>

    <form asp-action="AssignToScreen" method="post">
        @Html.AntiForgeryToken()

        <input type="hidden" asp-for="AlbumId" />

        <div class="mb-4">
            <label asp-for="SelectedScreenId" class="block text-sm font-medium text-gray-700">Select Screen</label>
            <select asp-for="SelectedScreenId" class="form-select mt-1 block w-full border border-gray-300">
                <option value="">-- Choose a screen --</option>
                @foreach (var screen in Model.Screens)
                {
                    <option value="@screen.ScreenId">@screen.Name</option>
                }
            </select>
            <span asp-validation-for="SelectedScreenId" class="text-sm text-red-600"></span>
        </div>

        <div class="flex items-center justify-between">
            <a asp-action="Index" class="text-sm text-gray-600 hover:underline">← Back to Albums</a>
            <button type="submit"
                    class="bg-indigo-600 hover:bg-indigo-500 text-white px-4 py-2 rounded-md shadow-sm">
                Assign Album
            </button>
        </div>
    </form>
</div>


```

Figure 18

```

@using RoomCast.ViewModels
@model ScreenAssignmentViewModel

@if
    ViewData["Title"] = "Assign Media";
    Layout = "_Layout";
}



<h2 class="text-2xl font-bold mb-4 text-slate-800">
        Assign Media Files to
        <span class="text-indigo-600 font-semibold">@Model.ScreenName</span>
    </h2>

    <form asp-action="AssignMedia" method="post">
        @Html.AntiForgeryToken()
        <input type="hidden" asp-for="ScreenId" />

        <div class="grid grid-cols-1 sm:grid-cols-2 gap-4 mb-6">
            @foreach (var media in Model.AllMediaFiles)
            {
                <div class="flex items-center space-x-2 bg-gray-50 p-2 rounded">
                    <input type="checkbox"
                           name="SelectedMediaIds"
                           value="@media.FileId"
                           id="media-@media.FileId"
                           class="h-4 w-4 text-indigo-600 border-gray-300 rounded">
                    @(Model.SelectedMediaIds.Contains(media.FileId) ? "checked" : "") />
                    <label for="media-@media.FileId" class="text-slate-700">
                        @media.Title (@media.FileType)
                    </label>
                </div>
            }
        </div>
    </form>


```

Figure 19

2.3.6.3 Controller AlbumsController.cs and ScreenController.cs

Figure 20 - Assignment Albums to Screen below

```
// Assign to Screen (POST)
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> AssignToScreen(AlbumAssignmentViewModel model)
{
    var exists = await _context.AlbumScreenAssignments
        .AnyAsync(x => x.AlbumId == model.AlbumId && x.ScreenId == model.SelectedScreenId);

    if (!exists)
    {
        var assignment = new AlbumScreenAssignment
        {
            AlbumId = model.AlbumId,
            ScreenId = model.SelectedScreenId
        };

        _context.AlbumScreenAssignments.Add(assignment);
        await _context.SaveChangesAsync();
    }

    TempData["Message"] = "Album assigned to screen successfully.";
    return RedirectToAction(nameof(Index));
}
```

Figure 20

Assigning individual Media Files to screen for casting is implemented in [ScreenController.cs](#).

Figure 21 below

```
// POST: Screens/AssignMedia
[HttpPost]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> AssignMedia(ScreenAssignmentViewModel model)
{
    if (!ModelState.IsValid)
    {
        model.AllMediaFiles = await _context.MediaFiles.ToListAsync();
        return View(model);
    }

    var existingAssignments = _context.ScreenMediaAssignments
        .Where(x => x.ScreenId == model.ScreenId);
    _context.ScreenMediaAssignments.RemoveRange(existingAssignments);

    if (model.SelectedMediaIds != null && model.SelectedMediaIds.Any())
    {
        foreach (var fileId in model.SelectedMediaIds)
        {
            _context.ScreenMediaAssignments.Add(new ScreenMediaAssignment
            {
                ScreenId = model.ScreenId,
                MediaFileDialog = fileId
            });
        }
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Figure 21

2.3.6.4 Logic and Flow

User navigates to screen details. Chooses between assigning albums or media. For albums: selects album - submit form - controller saves assignment. For media: Selects a file - submits form - controller saves it. Database updates track assignments by timestamp. UI updates reflect linked content.

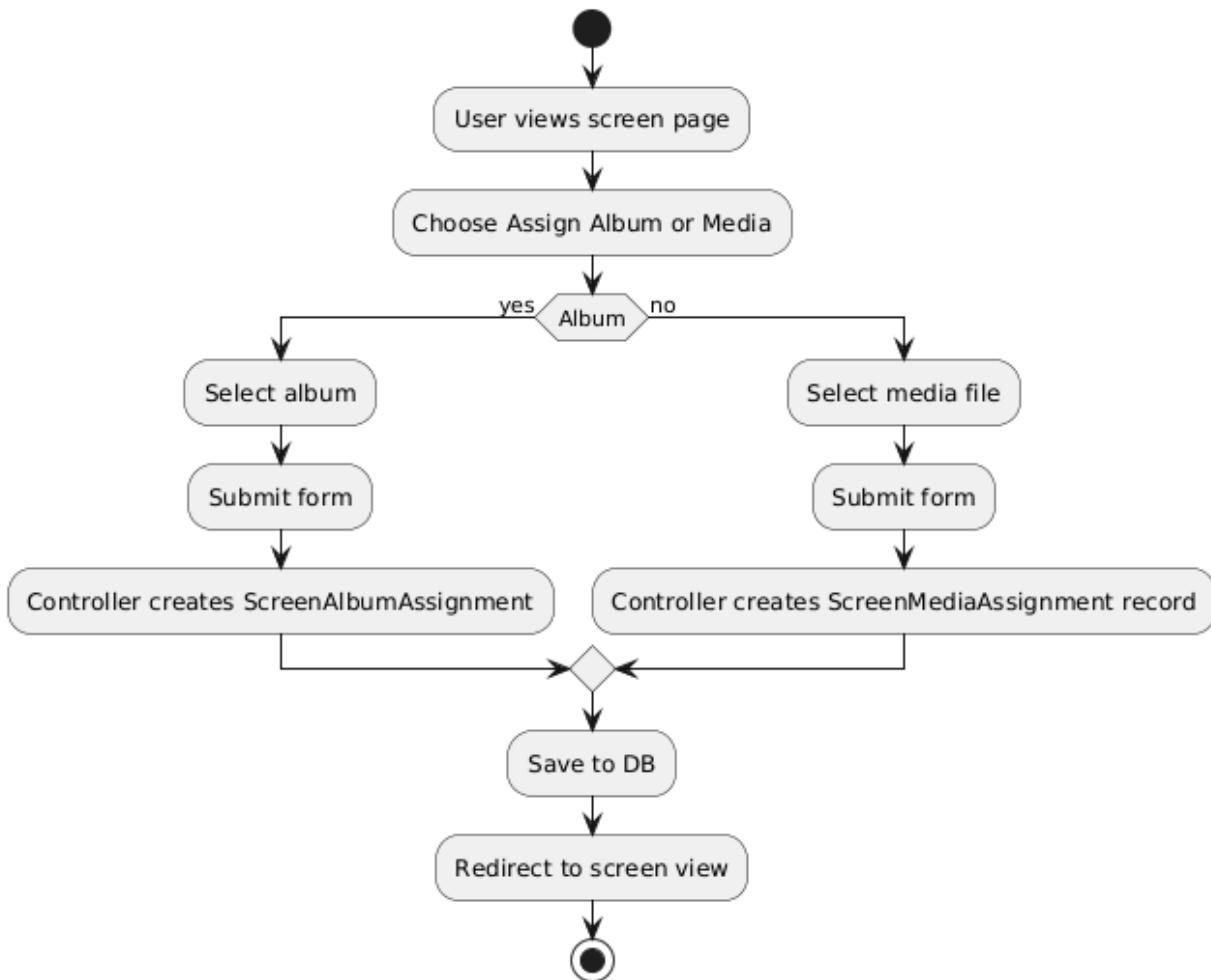


Figure 22 - Flow of events for assigning media or album to screen

Parameters and Return Values

ScreenId: ID of the target screen.

AlbumId: ID of the selected album.

SelectedMediaIds: List of media file IDs selected for assignment.

RedirectToAction("Details"): Reloads the screen view after successful update.

2.3.7 Electron Display Client App

This module is responsible for rendering and continuously displaying assigned content (albums or media) to a screen in full-screen kiosk mode. It acts as a receiver client, typically running on a smart TV, Raspberry Pi, or mini-PC, and auto-loads a remote ASP.NET Core page that serves the assigned screen content.

2.3.7.1 Key Code [main.js](#)

The key function reads the screen URL from a local configuration file called client-config.json. It is used by the Electron Display Client to know which Screen page to load (e.g., /Display/Screen/TV1).

```
// Read config (screen URL) from client-config.json
function getScreenUrl() {
  try {
    const configPath = path.join(__dirname, 'client-config.json');
    const raw = fs.readFileSync(configPath, 'utf-8');
    const config = JSON.parse(raw);

    if (!config.screenUrl || typeof config.screenUrl !== 'string') {
      throw new Error('screenUrl is missing in client-config.json');
    }

    return config.screenUrl;
  } catch (err) {
    console.error('Failed to read client-config.json:', err.message);
    return null;
  }
}
```

Figure 23 - Key code for reading the screen

2.3.7.2 Logic and Flow

The Electron Display Client App follows a continuous, event driven loop designed to always show the correct content assigned to a specific screen. Its logic is optimised for reliability, auto-recovery, and real-time updates. The workflow consists of four main stages: **initialisation**, **content loading**, **periodic refreshing**, and **auto-recovery**.

The Electron runtime boots and loads main.js. The application reads configuration (screen ID, base URL) from config.json, then the system creates a **Window** in kiosk mode (no frames, no menu, full screen and lastly the window loads the Screen Display endpoint from the ASP.NET Core app (<http://127.0.0.1:7058/Display/Screen/{screenId}>)

Electron app is designed to stay running even if connection is lost, [ASP.NET](#) server is down and when URL is down

2.4 Database

The database module serves as the core foundation for storing, managing, and retrieving all application data in RoomCast. This includes users, media files, albums, screens, assignments, and audit logs. The schema is designed to support flexible relationships, ensure data integrity, and provide performance for real time content casting. For the purpose of this project at this stage we used **Entity Framework Core** with **SQLite** as the database provider, the application maintains strong referential integrity while supporting scalable content management and assignment operations. File app.cd is database file, the file from the project structure where contains the actual database with all the tables

Database entities and Relationships:

Users: Stores authentication credentials and profile details.

MediaFiles: Contains metadata and paths for uploaded files.

Albums: Represents grouped media files.

AlbumFiles: Junction table linking albums and media files (many-to-many).

Screens: Represents digital displays/screens.

ScreenMediaAssignments: Tracks individual media assigned to screens.

ScreenAlbumAssignments: Tracks entire albums assigned to screens.

All the key functions in the RoomCast project are connected with the database. User uploads the file (previously registered successfully with its Credentials, email and password stored in the database) and uploads a file that is stored in MediaFiles. When an Album is created it is stored in Albums. Media assigned to albums via AlbumFiles. Screens are registered in Screens. Users assign media or album to screen saved in MediaScreenAssignment or ScreenAlbumAssignments.

With executing all the functions in fact the user has used all the key functions that the project can offer.

user_id	file_type	file_format	tags	timestamp	file_name
1	1	jpg.	Nef302	22:35:17 08-08-2025	MyPhoto
2	1	mp4.	NEF302	22:40:12 08-08-2025	MyVideo

Figure 23 - Example of Upload table that stores media files in the database

EF Core Migrations are used to scaffold and update the database schema:

```
dotnet ef migrations add InitialCreate
```

```
dotnet ef database update
```

3. Implementation Strategy

Our development process followed an **Agile iterative methodology**, broken into sprints aligned with project milestones such as feature scoping, wireframing, database design, module (functions) implementation, testing, and final deployment. Regular team meetings and progress logs were maintained to ensure steady collaboration, planning, and delivery across all major features.

Throughout the development, we relied on Git version control, Visual Studio Code for coding, and .Net CLI for running migrations, scaffolding, and testing builds. SQLite was used for local development to simplify setup and reduce dependencies, and we maintained code modularity using the MVC pattern.

3.1 Key Programming Concepts

Model-View-Controller (MVC): The entire RoomCast platform was structured using the ASP.NET Core MVC framework. Each feature like User Authentication, Media Upload, Editing, Album Management, Screen Assignment was modularized into Models, Views, and Controllers for better separation of concerns. Models manages the data and logic, Views handles the user interface from where can be worked on and tested separately and the Controllers as intermediary that routes user requests to the Models and selects a the required View to render the response. We decided to go with this concept because the code is separated and makes it more organized maintainable and easier to develop by separating concerns.

Repository Pattern (Internalized via EF Core): Though we didn't explicitly create repositories, EF Core's DbContext served a similar purpose, managing object-to-database mapping with LINQ and lambda expressions.

Dependency Injection : ASP.NET Core's built-in DI container was used throughout the app. For example, services like IMediaFilePreviewBuilder were injected into controllers to centralize preview generation logic and keep code testable and maintainable.

3.2 Key Data Structures Used

Entity Models (Classes): Core data structures such as User, MediaFile, Album, Screen, and the linking tables like ScreenAlbumAssignment and AlbumMediaFile were modeled as C# classes with properties and relationships. These directly translated to relational tables in SQLite.

Collections and Lists: Most many-to-many or one-to-many relationships used `ICollection<T>` in the model definitions to represent foreign key collections. Example: Screen holds `ICollection<ScreenAlbumAssignment>`.

ViewModels: Custom view models like MediaFilePreviewViewModel, ScreenAssignmentViewModel, and TextFileDialogViewModel allowed combining multiple data models and form elements in a single page view.

All modules were tightly integrated via routing and navigation. For example:

- Uploads linked directly to the media preview module.
- Edited files were automatically saved and redirected to the file listing.
- Albums were created from existing uploads.
- Screens displayed album or file previews depending on assignment.

Each controller action interacted with the database via ApplicationDbContext, and page transitions were controlled using RedirectToAction or PartialViewResult depending on flow.

3.3 External APIs and Libraries

Electron (for Client App): A Node.js-based desktop wrapper was developed to load the /Screens/ScreenDisplay/{id} page in fullscreen kiosk mode using Electron.

FontAwesome & Bootstrap: For UI icons and responsiveness.

Entity Framework Core: For data access, migrations, and schema updates.

4. System Configuration and Setup

4.1 Configuration files

RoomCast system uses several configuration files essential for the system to operate.

A. `appsettings.json` (Server Configuration)

Used by ASP.NET Core to configure:

- Database connection string
- Identity settings
- File storage paths
- Authentication keys

B. `client-config.json` (Electron Display Client)

This file tells the Electron client which screen to load.

Everything else that is required is mentioned earlier in this document.

4.2 Deployment Process

This section explains how to deploy both the server and the client in a production-like environment.

The developer team has already run `dotnet publish -c Release` (or a self-contained publish) and provided a publish folder containing `RoomCast.exe`, `app.db`, and supporting files.

4.2.1 Server Side

Step 1 - User to have the Server Folder to the Computer, like

- From USB drive
- Network share
- Downloaded .zip

Step 2 - Copy the entire folder to a specific location on the computer

(eg. C:\Programs\RoomCastServer\RoomCast.exe and all the other files that RoomCastServer contains. Also when project runs even for first time if there is no database it creates database app.db (at this point of development using SQLite)).

Step 3 - To make the server behave like a normal Windows program(double click on desktop to run), a small VBScript file will be used. In the server folder (e.g. C:\RoomCastServer\), right-click and choose: **New → Text Document** → name it StartRoomCast.vbs.

Open StartRoomCast.vbs in Notepad and paste:

```
Set shell = CreateObject("WScript.Shell")  
  
' Start the RoomCast server silently (no console window)  
  
shell.Run "RoomCast.exe", 0, False  
  
' Wait 5 seconds for the server to start  
  
WScript.Sleep 5000  
  
' Open the admin login page in Google Chrome  
  
shell.Run """C:\Program Files\Google\Chrome\Application\chrome.exe"""  
http://127.0.0.1:7058/Account/Login"
```

Save the file and close Notepad.

What this does:

- RoomCast.exe is started in the background (console hidden).
- After 5 seconds, Chrome automatically opens the login page.
- The user does not see any terminal window—only the browser UI.

Step 4 - Create a Desktop Shortcut to the Server

In File Explorer, right-click StartRoomCast.vbs → **Send to** → **Desktop (create shortcut)**.

Go to the Desktop, rename the shortcut to:

RoomCast Server

(Optional) Change the icon:

Right-click the shortcut → Properties → Change Icon

After all these steps the user only needs to Double Click the icon on the Desktop named

RoomCast Server. After a few seconds the app will open in the default Browser landing on Login Page and continue to operate. To fully close the app , the user will need to open Task Manager (CTRL+ALT+DEL) and find RoomCast.exe and click End Task.

4.2.2 Client Side

In this section we explain how to install and start the **RoomCast Display Client** on a machine connected to a TV or monitor (for example: a mini-PC, laptop, or dedicated display box).

The developers team will also have to have already run npm install and npm run build, and provided the **built client folder** containing RoomCast Display.exe, client-config.json, and other files under dist\win-unpacked\.

Step 1 - User will be provided with RoomCast Display build. User have to copy and extract this file on the target machine for casting. The folder contains all the files and codes needed to run the display client.

Step 2 - In C:\RoomCastDisplay_TV1\, open **client-config.json** in Notepad.

Set the **screenUrl** to the correct screen display URL of the RoomCast server, for example:

```
{  
  "screenUrl": "http://127.0.0.1:7058/Display/Screen/TV1"  
}
```

TV1 can be replaced with the actual screen name configured in the system.

If the server runs on another machine, replace 127.0.0.1 with the server's IP, for example:

```
{  
  "screenUrl": "http://192.168.0.10:7058/Display/Screen/TV1"  
}
```

Save and close the file.

Step 3 - Creating a Desktop Shortcut for the Display Client

To make it easy for non-technical users to start the display: Right-click RoomCast Display.exe → **Send to → Desktop (create shortcut)**. Go to Desktop and rename the shortcut to: RoomCast Display – TV1. This clearly indicates that this shortcut controls Screen TV1.

Step 4 - Ensure the TV is on and the correct input is selected.

On the computer, double-click **RoomCast Display – TV1**.

The display app opens in full-screen and starts showing whatever content is assigned to that screen in the RoomCast admin panel.

5. Testing and Validation

This chapter outlines the testing methodologies, tools, and validation criteria used to ensure the correctness, reliability, and usability of the RoomCast system. Since RoomCast consists of two independent applications — the **ASP.NET Core Server** and the **Electron Display Client** — testing was performed across both platforms to validate end-to-end functionality.

5.1 Testing Strategies

Multiple testing strategies were used throughout the development lifecycle to validate the major components of the system: authentication, media handling, screen assignment, album management, and client display rendering.

5.1.1 Manual Functional Testing

This was the **primary testing approach** used due to the rapid and iterative nature of the project.

Each key feature was manually tested using realistic scenarios, including:

- Creating accounts (registration)
- Logging in and out
- Uploading images, videos, and PDFs
- Editing media files (video, image, text, document replacement)
- Creating albums and adding/removing media
- Assigning albums/media to screens
- Starting and stopping casting
- Verifying correct content shown on the Electron Client
- Checking automatic refresh and reconnection behaviour

Testing was performed during and after development of each module.

5.1.2 Black-Box Testing

Black-box testing was applied to ensure that the system behaves correctly from the user's point of view without examining internal code.

For example:

- Uploading a video → correct preview generated → correct database entry created
- Assigning an album → Electron screen correctly switches content
- Stopping casting → display shows “No content assigned”

This verified outward behavior and user experience.

5.1.3 Integration Testing (Manual)

RoomCast required interaction between multiple components:

-
- ASP.NET Core Server
 - EF Core Database (SQLite)
 - Display Client (Electron App)

Integration testing ensured all modules worked together:

Examples:

- Changing assignment on the server correctly updates the Electron Client
- Editing a video file updates the preview service
- Assigning a PDF correctly renders inside the client

5.1.4 UI/UX Testing

User interface testing ensured:

- Layout consistency across all views
- Buttons and forms work correctly
- Feedback messages appear when expected
- Screens resize correctly
- Electron Display Client maintains full-screen view

5.1.5 Error-Case Testing

Negative scenario testing ensured the system handled errors safely:

Server:

- Invalid login
- Unsupported file formats
- Large file uploads
- Deleting media used by an album
- Accessing screen URLs without assignment

Client:

- Server offline
- Wrong screen URL
- No network connection
- Assignment changed during playback

Client correctly showed “Cannot reach server...” and auto-retried until available.

5.2 Testing Tools

Although no automated test frameworks were implemented (such as NUnit or xUnit), several tools were used during manual and integration testing.

5.2.1 Browser Developer Tools

Used for:

- Console errors
- Network request monitoring
- Checking preview rendering
- JavaScript debugging

Browsers used:

- Google Chrome
- Microsoft Edge

5.2.2 EF Core Logging

Console and in-app logs verified:

- Database creation
- SQL queries
- Migration generation

5.2.3 Electron Debug Console

When running the client in debug mode:

```
npm start
```

Electron provided:

- Connection error logs
- Rendering errors
- URL load failures
- Auto-refresh attempts

5.3 Validation Criteria

To ensure that each feature met the project requirements, validation criteria were defined for each key function.

Below are the main criteria used:

5.3.1 Authentication Module

A feature was considered valid if:

- User can successfully register
- User can log in with valid credentials
- Invalid credentials show clear error messages
- Logout successfully clears session
- Login redirect works properly

5.3.2 Media Upload Module

A feature was accepted when:

- File uploads without crashing

-
- Unsupported formats show an error message
 - File appears in “My Files”
 - Correct preview displayed (image/video/PDF/text)
 - Metadata saved correctly

5.3.3 Media Editing Module

Validated when:

- Replacing files updates previews
- Editing text files shows correct output
- Media file changes reflected immediately
- Invalid inputs rejected
- Database updates without errors

5.3.4 Albums Module

Validated when:

- Creating an album works
- Adding media is reflected instantly
- Removing media updates the album UI
- Album preview loads all media correctly
- No orphan records in the database

5.3.5 Screens & Casting Module

Validated when:

- Assigning content updates the screen list
- Starting/stopping casting triggers the correct UI
- Electron Client receives updates via refresh

-
- Screen URL loads correct content
 - Album slideshows run smoothly
 - PDFs/videos/images render correctly

5.3.6 Electron Display Client

Validated when:

- Correct screen URL loads
- Full-screen mode works
- Lost connection is detected
- Automatically reconnects when server comes back
- No crashing or freezing
- Switching assignments updates content without restart

5.3.7 Deployment & Startup

Validated when:

- Server starts via .vbs silently
Chrome opens automatically
- Client starts without errors
Shortcuts function properly
- System works without needing .NET or VS Code (self-contained option)

6. Challenges and Solutions

Throughout the development of RoomCast, several technical challenges were encountered across backend functionality, media handling, casting logic, and the Electron display client. This

section outlines the major issues faced, why they occurred, and how they were successfully resolved.

Guid vs Int Mismatch Errors

Problem

The system originally mixed identity types:

- Screens used Guid as primary key
- Assignments expected int
- EF Core threw binding and mapping errors

Solution

All related models were unified to use **Guid** keys:

```
public Guid ScreenId { get; set; }
```

```
public Screen Screen { get; set; }
```

EF Core was updated and migrations were regenerated.

This eliminated compile-time and runtime errors.

Display Client Showing “Cannot Reach Server”

Problem

The Electron app often displayed “Cannot reach server...” because:

- The Screens routes weren’t implemented
- The server wasn’t starting properly
- Wrong screenUrl in client-config.json

Solution

- Added CheckStatus API
- Rewrote main.js to retry every few seconds
- Improved error messages
- Ensured client-config.json was validated before startup

Result: The client auto-connects and auto-recovers.

Display Window Not Full-Screen or Escapable by User

Problem

The Electron client wasn't fully locked down:

- Users could resize window
- Users could exit full-screen
- Looked unprofessional on TV

Solution

BrowserWindow settings updated:

fullscreen: true,

frame: false,

autoHideMenuBar: true

Result: A real kiosk-mode digital signage display.

Running the Server Without .NET Installed

Problem

Target machines (admin computers) often did **not** have:

- .NET SDK
- .NET Runtime
- Visual Studio Code

Server couldn't start.

Solution

Published a **Self-Contained Deployment**:

dotnet publish -c Release -r win-x64 --self-contained true

This bundles the entire .NET runtime into the .exe.

Result: Server runs on ANY Windows PC.

7 Future Enhancements

Although RoomCast is fully functional and meets the requirements of a digital signage and media-casting system, several improvements can strengthen the platform further in terms of

performance, usability, automation, and scalability. These enhancements can be incorporated in future development phases or as part of ongoing maintenance.

7.1 Database Scalability

SQLite is excellent for local and small installations, but for larger deployments:

Upgrade to a scalable DBMS:

- SQL Server (Most Likely)
- PostgreSQL
- MySQL
- Azure SQL

This supports higher concurrency and faster queries.

7.2 Real-Time Casting Using SignalR WebSockets

Currently, the Electron Display Client periodically polls the server (every 8–10 seconds) to detect assignment changes.

A future improvement is to replace polling with **push-based updates** using SignalR.

Benefits:

- Immediate content change on screens
Less server load
- Smoother user experience

7.2 Support for More Media Types

Future improvements may include:

- Office documents (PowerPoint, Word)

-
- Interactive HTML content
 - Streaming URLs (YouTube, HLS streams)

This expands use cases beyond static media files.

7.3 User Roles & Permissions

Implement fine-grained access control:

- Admin
- Content Manager
- Viewer
- Screen Operator

Each role would have specific permissions (uploading, editing, assigning, deleting).

7.4 Cloud Deployment Option

Deploy RoomCast as a **cloud SaaS** solution using:

- Azure App Service
- AWS Elastic Beanstalk
- DigitalOcean App Platform

This allows multi-site deployments and remote management.

7.5 Automated Installer (MSI/EXE)

Package the entire system with a professional installer that:

- Installs server
- Installs display client
- Creates shortcuts

-
- Installs startup services
 - Verifies screen URLs
- Simplifies deployment for non-technical users.

As RoomCast grows, it must handle more screens, more content, and higher traffic. All these enhancement are most likely to be done in future development. Or if the project comes in different package Basic (like it is now), Pro and Premium.

8. Conclusion

The RoomCast project successfully implemented a fully functional digital signage and media-casting platform that enables administrators to upload, edit, organize, and broadcast multimedia content to multiple screens. Throughout the development process, several key programming concepts and architectural decisions were utilized to ensure that the system is reliable, modular, scalable, and user-friendly.

The backend was built using **ASP.NET Core MVC (.NET 9)** and integrated with **Entity Framework Core** and **SQLite**, enabling efficient data storage for users, media files, albums, and screen assignments. Core programming tasks included designing data models, implementing CRUD operations, managing authentication with Identity, handling file uploads and previews, and building REST-style controller actions for casting operations. These components formed the foundation of the RoomCast Server.

On the frontend, Razor views, Bootstrap UI elements, and custom JavaScript enhancements were used to create clean, intuitive pages for media management, album creation, editing tools, and screen control. Additional attention was given to user experience by ensuring previews, validation messages, and dynamic updates rendered correctly and consistently.

A major milestone in the project was the integration of the **Electron Display Client**, a standalone application designed for TVs and display devices. This client application communicates with the server, loads assigned content, refreshes automatically, runs in kiosk mode, and gracefully handles loss of connection. Packaging the client using electron-builder

enabled true standalone functionality, allowing RoomCast to run on devices without Node.js or developer tools.

Deployment and usability were also prioritized. The server was packaged into a self-contained executable, launched through a `.vbs` script for a professional, silent startup, and provided to users through easy desktop shortcuts. Likewise, the Display Client was configured with screen-specific URLs and set up to auto-launch on system startup, ensuring seamless operation in real display environments.

Overall, the combination of backend logic, database design, UI development, casting mechanisms, and deployment strategies demonstrates strong technical implementation skills. These programming components were crucial in achieving the project's primary goal: delivering a practical digital signage system capable of managing and displaying multimedia content across multiple screens in real time.