Final Report

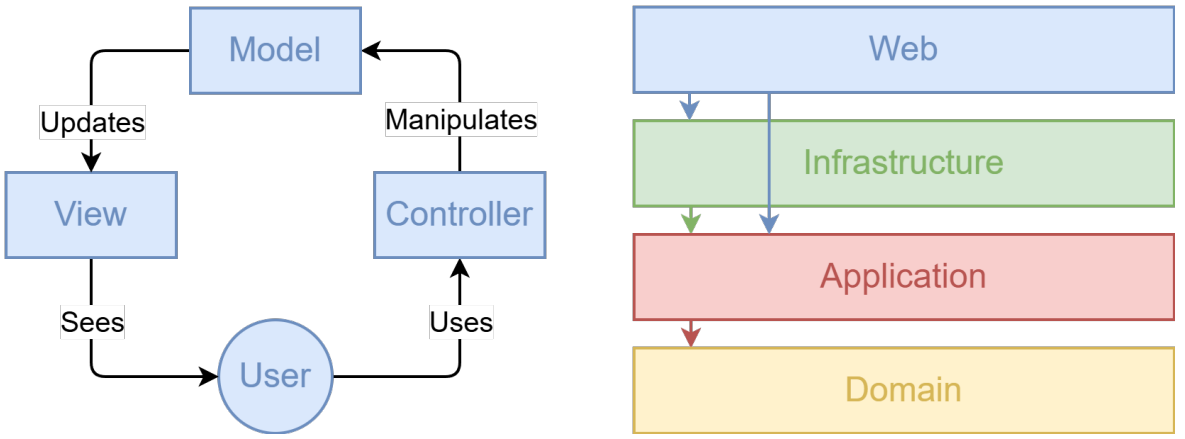
Summary

The goal of this project was to modernize and rebuild an outdated art portfolio website for my parents, who are artists. The new website utilizes modern frameworks such as .NET with C#, featuring full CRUD (Create, Read, Update, Delete) functionality to manage artworks efficiently. The website also includes a signup and login system to handle user authentication and authorization. Additionally, it has search and filter functionality to query the database for specific artists or artworks. By following best practices in programming, the new website is more maintainable and responsive, ensuring a better user experience across different devices. Throughout the project work I have gained insight into contemporary web development, including dynamic data handling and responsive design. The code for this project is available on GitHub [1].

Product Description

In this project, I used both the Model-View-Controller (MVC) architecture and Clean Architecture, see Figure 1 for a visual representation of these architecture patterns. Clean Architecture is a software design philosophy [2]. Its main goal is to create systems that are easy to maintain, test, and evolve. The architecture is organized into concentric layers, each with a specific responsibility. At the center are the Entities, which represent the core business logic and rules. Surrounding this layer are the Use Cases, which contain application-specific business rules. The next layer includes Interface Adapters, which convert data from the outer layers to a format usable by the inner layers. The outermost layer is the Frameworks and Drivers, which includes external tools, UI, databases, and web frameworks. This separation ensures that business logic is independent of external systems.

The MVC architecture is a design pattern that divides an application into three parts [3]: the Model, the View, and the Controller. The Model manages the application's data and business rules. The View is the user interface that displays this data. The Controller handles user input, updating the Model and deciding which View to display. By separating these components, MVC allows developers to work on each part independently, making the application easier to develop, test, and maintain. This architecture is used on the web layer of this project. The remainder of this section is an overarching description of the architecture layers in my project.



(a) Model-View-Controller

(b) Clean Architecture

Figure 1: Architecture patterns used in the project

The domain layer

This layer (**ArtPortfolio.Domain**) defines the core business entities and logic, which are used throughout the application. In this project the primary entities are the **ApplicationUser**, **Artist**, and **Artwork**. See Figure 2 for a visuall representation.

The ApplicationUser entity extends the identity user framework to leverage built-in authentication and authorization features provided by ASP.NET Core Identity. This extension allows for seamless integration of user management functionalities such as login, registration, and role management. It handles crucial security tasks like salting and hashing passwords to ensure they are stored securely. Besides these inherited attributes, the entity defines additional properties like the name and the date of creation. It also includes a foreign key linking to an artist if the user is associated with one, facilitating the management of artist-specific user accounts. In this project, users can be assigned one of two roles, namely Artist or Admin, each of which comes with different levels of permissions.

The Artist entity represents an individual artist, capturing essential details such as the first and last name, biography, email, website, date of birth, and optionally, a profile picture. This entity also maintains a collection of Artwork objects through a navigation property. A navigation property is a special type of property in Entity Framework that represents a relationship between entities; in this case, it allows access to all the artworks created by the artist.

The Artwork entity details individual pieces of art, including attributes like title, description, price, creation date, medium, dimensions, and an image URL. It includes a foreign key to link the artwork to an Artist, facilitating the relationship between artists and their works.

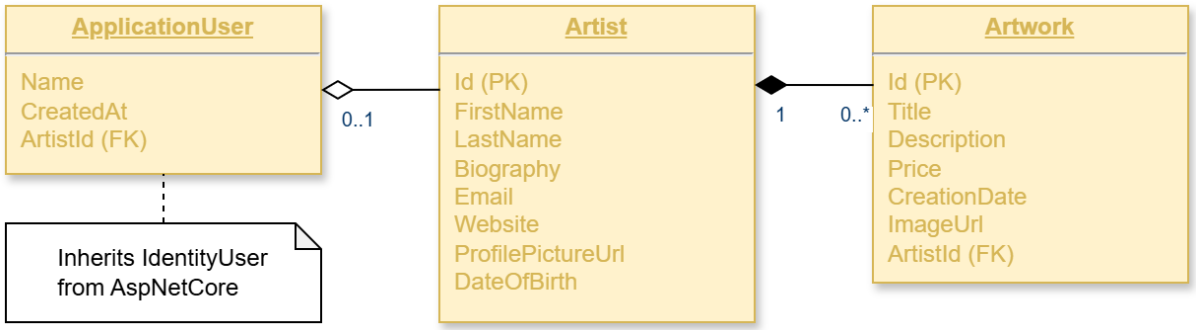


Figure 2: Entities defined in the domain layer

The infrastructure layer

This layer (**ArtPortfolio.Infrastructure**) implements the data access interfaces and manages data persistence, supporting the application layer's needs without exposing the core business logic to external systems. In this project, the data access layer utilizes SQL Server, a relational database management system developed by Microsoft, to handle the underlying data storage and operations.

The ApplicationDbContext class is the main point of interaction with the database, and it also defines the database relationships and rules using Entity Framework Core and a code-first approach. The code-first approach allows developers to define the database schema using C# classes, which are then used to generate the database tables automatically. This method facilitates easy management and versioning of the database schema.

Repository classes in this layer provide an abstraction over the data access layer, promoting a cleaner separation of concerns. The base Repository class uses generics, allowing it to handle various entity types through common data operations like adding, removing, and retrieving entities. More specialized repositories, such as ArtistRepository and ArtworkRepository, inherit from this base repository and implement additional methods specific to their respective entities. We use interfaces for these classes to facilitate dependency injection, which enhances testability and flexibility by allowing the injection of different implementations should the need arise. See the diagram in Figure 3 for a visual representation.

The UnitOfWork class encapsulates the repositories and ensures that multiple operations can be performed within a single transaction, maintaining data consistency and integrity. It acts as a common access point to the database, coordinating the work of multiple repositories using a single instance of ApplicationDbContext. While this class includes a repository for ApplicationUsers, the project mostly uses a separate instance called User that manages the current authenticated user and provides access to their identity and claims for authorization purposes, distinct from the data access handled by the UnitOfWork.

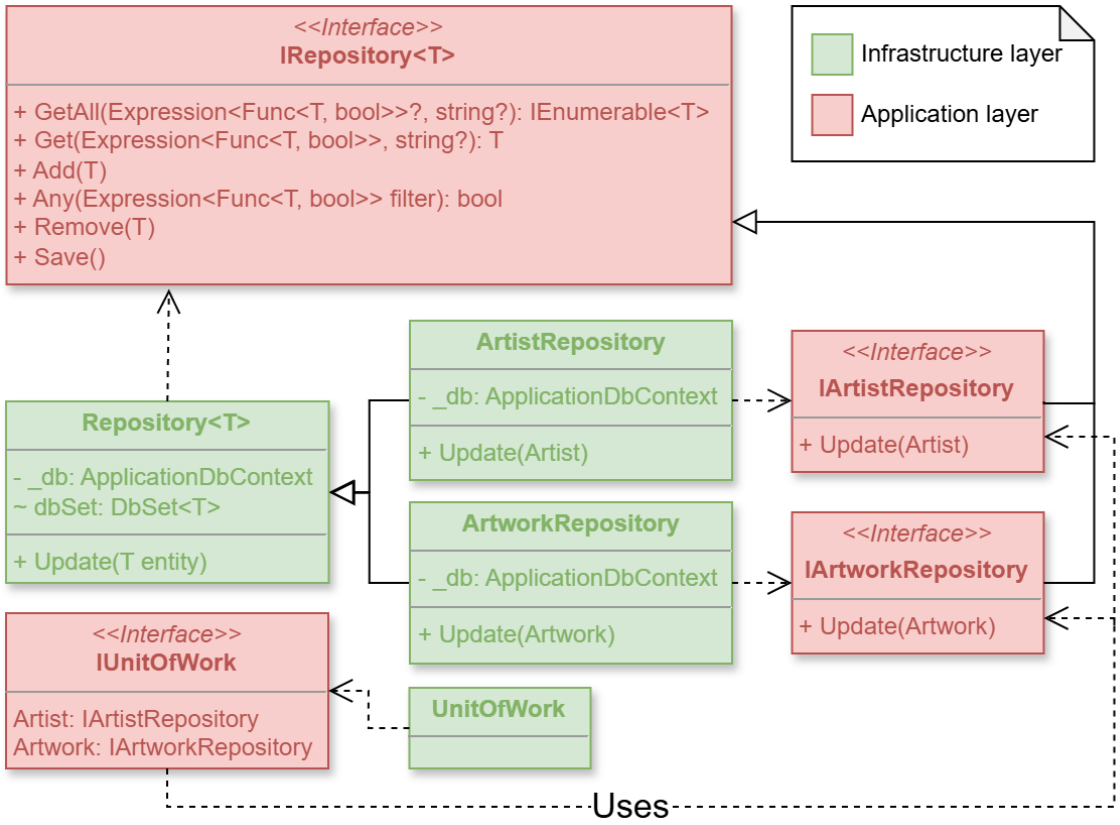


Figure 3: class diagram of the Unit of work and Repository classes

The application layer

This layer (**ArtPortfolio.Application**) acts as the bridge between the domain layer and the presentation layer. It contains business logic and service orchestration that directly interacts with the domain entities and repositories. In this project, it includes all interfaces, a Static Details (SD) class, and a predicate building class.

The common interfaces in the Application layer define the contracts for data access and manipulation. These interfaces outline the essential methods for interacting with domain entities, such as retrieving, adding, updating, and removing records. By defining these interfaces, the Application layer enforces a clear separation of concerns and supports dependency injection, making the application more modular, flexible, and testable. The ASP.NET Core MVC framework utilizes its own Dependency Injection (DI) container to manage and resolve dependencies.

The PredicateBuilder is a utility class within the Application layer that provides a set of static methods for dynamically constructing LINQ expressions. It simplifies the creation of complex query predicates by offering methods to create expressions that evaluate to true or false, and to combine predicates using logical operators like And, Or, and Not. This utility enhances the expressiveness and maintainability of business logic by allowing for the dynamic composition of query conditions. In the context of this project, it allows the filter predicate to be used when querying the database, and it opens up the possibility for extending the number of filter options further down the line without having to modify the previous code.

The SD class serves as a centralized repository for application-wide constants, such as roles and sorting options. By centralizing these constants, the SD class helps avoid the use of magic strings throughout the codebase, promoting consistency and reducing the risk of errors. This organization improves the maintainability of the application and ensures that key configuration values are easily accessible and modifiable.

The presentation layer

This layer (**ArtPortfolio.Web**) handles user input and presentation logic, interacting with the application layer to execute business rules and render the appropriate responses. In this project, the presentation layer is implemented as an ASP.NET MVC application.

Views in the project are implemented using CSHTML files that allows for server-side code execution directly within the HTML structure. This integration enables the seamless embedding of C# code within HTML, facilitating the generation of dynamic web pages based on the application's data and logic. The Razor syntax in CSHTML provides a more streamlined and efficient way to manage the dynamic aspects of the user interface, reducing the need for extensive client-side scripting. The project also utilizes component views and partial views. Partial views are used for the modal popup that contains artwork details, allowing for the reuse of view logic in multiple places without duplicating code. This makes it easier to manage and update the modal content separately from the rest of the view. The search bar is implemented as a view component, which is a self-contained unit that includes both the logic and the markup needed to render it. Unlike normal views, which are tied to specific controller actions, partial views and view components can be rendered within other views, providing modularity and improving the organization of the code. Partial views are lightweight and focused on rendering small pieces of content, while view components can encapsulate more complex logic and are designed for reuse across different parts of the application. Consistent elements across views, such as the navigation bar, are defined in the layout template, a shared structure that provides a common framework for all pages, ensuring a uniform look and feel while allowing individual views to inject their specific content. The styling of HTML elements in this project relies on Bootstrap 5 and CSS. See Figure 4 for an example of the responsive layout, and see the Appendix for some visual examples of the views in the project.

View models are utilized to populate the views in this project, serving as a tailored data structure designed specifically for the needs of the user interface. Unlike domain entity models, which represent the core business data and logic of the application, view models are crafted to fit the particular requirements of the view, often aggregating and formatting data in ways that the domain models do not. While alternative methods such as ViewData and ViewBag can be used to pass data from controllers to views—where ViewData is a dictionary and ViewBag provides a dynamic property container—these approaches are less structured and offer limited type safety. It is considered good practice to use view models as they provide strong typing, reduce coupling between the presentation and business logic layers, and enhance maintainability by ensuring a clear and reliable contract between the controller and the view. This structured approach helps manage changes more effectively and ensures that the data displayed is accurate and well-organized.

Controllers manage the underlying server-side logic and populate the view models. This project includes three main controllers responsible for managing artists, artworks, and user accounts respectively. These controllers have CRUD methods for their entity, usually with a GET request to present the view and POST request to push the changes to the database. However, there are instances where certain logic needs to be handled on the client-side. In this project, jQuery is used to simplify DOM manipulation and event handling. Furthermore, we rely on some third-party JavaScript libraries for specific tasks. Masonry [4], in conjunction with imagesLoaded [5], are used to create a masonry layout gallery for displaying artworks. This combination ensures that the layout is properly adjusted once all images are loaded, providing a visually appealing and organized presentation. Infinite-scroll is used to detect when the user reaches the end of the page and then proceeds to append new artworks (JSON objects from the controller) to the gallery, allowing for a seamless browsing experience without the need for manual pagination. A drawback to this is that this requires that the user have JavaScript enabled in the browser for content to be loaded and displayed correctly. Bootstrap does not support a dynamic masonry layout, and neither does CSS. An alternative approach could have involved using a simple grid layout and rescaling or cropping artworks to fit the available space. However, I felt that this would compromise

the visual integrity of the artworks, as it might distort their aspect ratios and detract from the overall presentation. By using Masonry, we preserve the original proportions of the artworks while creating a visually dynamic and engaging gallery experience.

I have tried to perform validation on both client and server side to provide good user experience while still maintaining data integrity. ASP.NET provides "Tag Helpers" to seamlessly integrate client-side validation into the Razor views. By applying Tag Helpers like **asp-for** and **asp-validation-for** to form fields, I enabled automatic generation of HTML attributes and validation messages that enforce client-side validation rules. On the server side, I utilized **ModelState.IsValid** in the controller actions to ensure that the submitted data adheres to the validation attributes defined in the model. **ModelState** is a key component in ASP.NET MVC that tracks the state of the data as it flows through the application. It captures any validation errors that occur when the data is bound to model properties during form submission. By checking **ModelState.IsValid**, I can verify whether the incoming data meets all the required validation rules, such as data types, required fields, and custom validation logic.

However, the server-side logic does not automatically provide feedback to the user for the server-side validation. To address this, I used the TempData container to send a Toastr message to the user indicating whether the action was successful or not. Toastr is a JavaScript library used to create non-blocking, elegant notifications that appear on the web page, often used for displaying success, error, or informational messages to users. TempData is a storage mechanism that is used to store data that needs to persist across a single request, often for the purpose of redirect scenarios. Unlike ViewData or ViewBag, which are used to pass data from controller to view, TempData is well-suited for temporary data that is required only for the duration of the redirect. It works by storing the data in the session until it is read, at which point it is automatically removed. To display feedback messages, I included a partial view in the template file. This partial view is rendered whenever TempData contains a "success" or "error" key, allowing us to display Toastr notifications to inform the user of the outcome of their action.

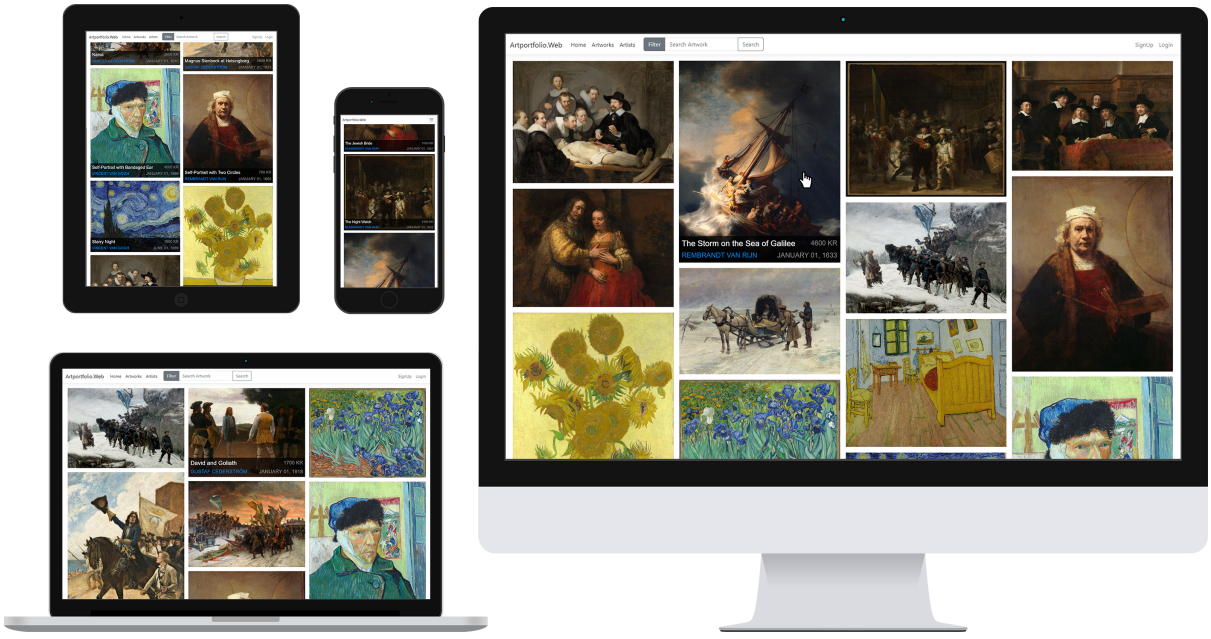


Figure 4: Responsive layout for different devices

Product Quality

In my opinion, the final product is stable and performs well under various conditions. However, there are known deficiencies such as limited scalability for very high traffic and some UI elements that could be further refined for better user experience and security.

Currently, only one image is being served regardless of screen size, which can impact performance. Future updates could address some of this issue by using **srcset** and **sizes** alongside **¡img¿** to give fine-grained control over which images are served for different screen sizes, thereby reducing load times and improving performance. The drawback of this method is that it requires managing multiple versions of the same image, which can be time-consuming and complex. To mitigate this, I could try configuring tools like **ImageSharp** [6] or **ImageResizer** [7] to automatically generate different image sizes as part of the build process.

While the code is secure from SQL injection attacks, as we interact with the database through Entity Framework using LINQ, it is still possible to inject HTML into fields such as the title or description of artwork entities. When content is added to a Razor view, it is usually done through the Model. The "Model" consists of classes that represent the application's data structure and provide a way to pass data from the controller to the view. Razor automatically escapes any potentially harmful code when rendering content from the Model to ensure the page is secure from injection attacks. However, in the index view of the Artwork controller, content is delivered through JSON strings, which are not automatically escaped. This creates a risk of Cross-Site Scripting (XSS) attacks, where an attacker could inject malicious scripts into the page. These scripts can execute in the context of the victim's browser and perform harmful actions such as stealing sensitive information (e.g., cookies, session tokens), hijacking user sessions, or redirecting users to malicious sites. With additional development time, this issue could be mitigated by implementing proper sanitization and encoding techniques when handling data uploads and when populating content using JSON.

Use of Knowledge

This project leveraged knowledge from previous courses such as web development, database management, and software engineering principles. It was a boon to have knowledge about the coding language C# and .NET in general, which provided a solid foundation for the backend development. However, I was less familiar with frontend technologies, so these tasks took longer to find solutions to and implement effectively. I also used non academic sources for information. The ASP.NET documentation was an good repository of information, often including practical code snippets and explanations. I also took inspiration from the code of Bhrugen Patel, a well-known software developer and instructor specializing in .NET technologies, particularly for the infrastructure layer.

I was surprised by how good Large Language Models (LLMs) can be as a pair programmer or "rubber duck." LLMs, in my case OpenAI's ChatGPT, assist in various development stages by providing instant feedback, suggesting code improvements, and offering solutions to problems. They can act as an ever-available pair programmer, helping debug code, explain concepts, and explore alternative approaches. Additionally, using ChatGPT as a "rubber duck" helped to clarify thoughts and identifing issues by explaining problems to an attentive listener, even if virtual. However, I also noticed some cases of incorrect or suboptimal solutions. So while LLMs are valuable, I think they should be used judiciously alongside other resources.

Successes

This section highlights some of the key successes achieved throughout the project. It reflects on the accomplishments that contributed to the development process, as well as the personal satisfaction and learning experiences gained during the project.

Technical Successes

I successfully implemented CRUD functionality, enabling efficient management of artworks. Additionally, I achieved a responsive design that ensures a seamless user experience across various devices. I am especially happy with the index views of the Artwork and Home controllers, which ultimately turned out to be both visually pleasing and technically robust. By adhering to Clean Architecture and MVC principles, I ensured that the website is maintainable and well-structured.

Project Work

I thoroughly enjoyed the process of transforming the website into a functional platform. It felt like a creative endeavor, much like building with Lego, where each piece comes together to form a cohesive whole. There was a deep satisfaction in tackling complex technical challenges and witnessing the tangible results of my efforts come to life. Unlike most of my courses, which focus less on user experience, this project offered a refreshing opportunity to engage with the frontend and design aspects in a meaningful way.

Problems

This section outlines the challenges encountered during the project and the strategies employed to address them. I encountered somme issues while integrating Colcade [8], a newer and more lightweight alternative to Masonry, for creating masonry layouts. Colcade is one-eighth the size of Masonry, does not require the imagesLoaded library when working with images, and is more responsive in browsers. However, I was unable to get it to work with the infinite scroll functionality, where new artworks are appended as you reach the end of the page. This resulted in uneven distribution of artworks across columns, leaving gaps in the layout. Consequently, I decided to continue using Masonry for its reliability and compatibility with infinite scroll functionality.

During the development of the application's filtering functionality, I encountered issues with using the LINQKit library due to compatibility problems with Entity Framework's eager loading. As a result, I explored alternative approaches to dynamically build query predicates without relying on third-party libraries. After conducting research, I discovered a well-established solution proposed by Joseph Albahari. He is known for his contributions to the .NET community, including the co-authorship of "C# in a Nutshell" [9], introduced the PredicateBuilder class in his article "Dynamic LINQ Queries with Expression Trees". This class enables the dynamic composition of LINQ query predicates in a flexible and efficient manner. I decided to implement Albahari's PredicateBuilder class in the project, as it offers a straightforward and reusable method for constructing complex queries. The class allows for chaining multiple conditions using logical operators such as And, Or, and Not, making it an ideal replacement for LINQKit in this context. By incorporating this solution, I was able to maintain the desired functionality of the application's filtering system without the complications introduced by third-party dependencies.

In my initial implementation, I experimented with using custom policies to enforce rules governing which users were authorized to perform specific CRUD operations. This approach involved defining authorization policies and applying them through methods within the controller. However, I ultimately decided against this approach. Instead, I opted to use the standard [Authorize] attribute, which verifies if the user has signed in. I combined this with private methods to handle the authorization logic directly within the controller. This alternative approach provided greater flexibility and simplicity in managing access control across different operations, such as offering users feedback on the outcome of their actions using Toastr messages.

Work Plan

I have achieved the overarching goal of creating the website for displaying and managing artworks. This project has been a rewarding and enjoyable experience, allowing me to apply and enhance my skills in web development and application design. The website successfully meets the initial objectives, providing a functional and user-friendly platform for artwork management.

One deviation from the original time plan was the decision not to implement SEO improvements on the website. However, I deemed it more beneficial to polish the existing functionality and ensure a stable and smooth user experience rather than introducing new content and features so late in the course. This decision allowed me to focus on refining the core aspects of the website, resulting in a more robust and reliable application.

I also decided against publishing the site with the content from the old website, even though this was considered as a potential option in the initial project plan. There are two primary reasons behind this

decision. First, the infinite-scroll [10] library requires a commercial license. To avoid licensing issues, I would need to revert to my own scroller implementation, which lacks many of the features provided by their version. Second, publishing the site would require implementing a scraper to retrieve content from the old website. While I successfully tested the feasibility of fetching images and titles, extracting more complex information like artwork descriptions, prices, and dimensions proved challenging. These details are embedded as plain text within the same HTML element, making it difficult to accurately extract and input them into the database. Given the large number of images and the potential for human error, manually extracting this information did not seem feasible within the remaining time of the course.

Final Time Report

In this section, I provide a weekly breakdown of the time spent on various tasks throughout the course. The table below, see Table 1, outlines the specific activities completed during each week and the corresponding hours dedicated to these tasks. The total time recorded serves as an estimate, acknowledging that the productive time allocations may vary due to factors such as different level of focus between sessions (some of the work was performed after the weekly reports, during the weekends). However, this summary offers a clear estimation of the time invested in the different phases of the course.

Week	Goal	Status	Time Spent
24	Project Setup and Initial Planning	Completed	Unkown
25	Environment Setup and Initial (Backend and Frontend)	Completed	40 hours
26	Basic CRUD Operations (Backend and Frontend)	Completed	42 hours
27	User Authentication (Backend and Frontend)	Completed	35 hours
28	Enhancing CRUD Operations and User Authentication	Completed	40 hours
29	Search and Filter Functionality (Backend and Frontend)	Completed	40 hours
30	Responsive Design and UI Enhancements	Completed	43 hours
31	Search Engine Optimization	Not Completed	40 hours
32	Final Testing and Buffer Week	Completed	38 hours
33	Final Report	Completed	27 hours

Table 1: Weekly Goals, Outcomes, and Time Spent

References

[1] V. Forsman, *Art portfolio website*, Accessed: 2024-08-02, 2024. [Online]. Available: `https://github.com/GunGoat/ArtPortfolio`.

[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[3] A. Leff and J. T. Rayfield, "Web-application development using the model/view/controller design pattern," in *Proceedings fifth ieee international enterprise distributed object computing conference*, IEEE, 2001, pp. 118–127.

[4] D. DeSandro, *Masonry v4.2.2*, Accessed: 2024-08-02, 2018. [Online]. Available: `https://github.com/desandro/masonry`.

[5] D. Desandro, *Imagesloaded v4.1.4*, Accessed: 2024-08-02, 2024. [Online]. Available: `https://github.com/desandro/imagesloaded`.

[6] J. Jackson-South, *Imagesharp*, Accessed: 2024-08-09, 2024. [Online]. Available: `https://github.com/SixLabors/ImageSharp`.

[7] L. River, *Imageresizer*, Accessed: 2024-08-09, 2024. [Online]. Available: `https://github.com/imazen/resizer`.

[8] D. DeSandro, *Colcade*, GitHub repository, 2024. [Online]. Available: `https://github.com/desandro/colcade`.

[9] J. Albahari, *C# 12 in a Nutshell*. O'Reilly Media, Incorporated, 2023.

[10] Metafizzy, *Infinite-scroll v4.0.1*, Accessed: 2024-08-02, 2024. [Online]. Available: `https://github.com/metafizzy/infinite-scroll`.

A Appendix A: Additional Figures

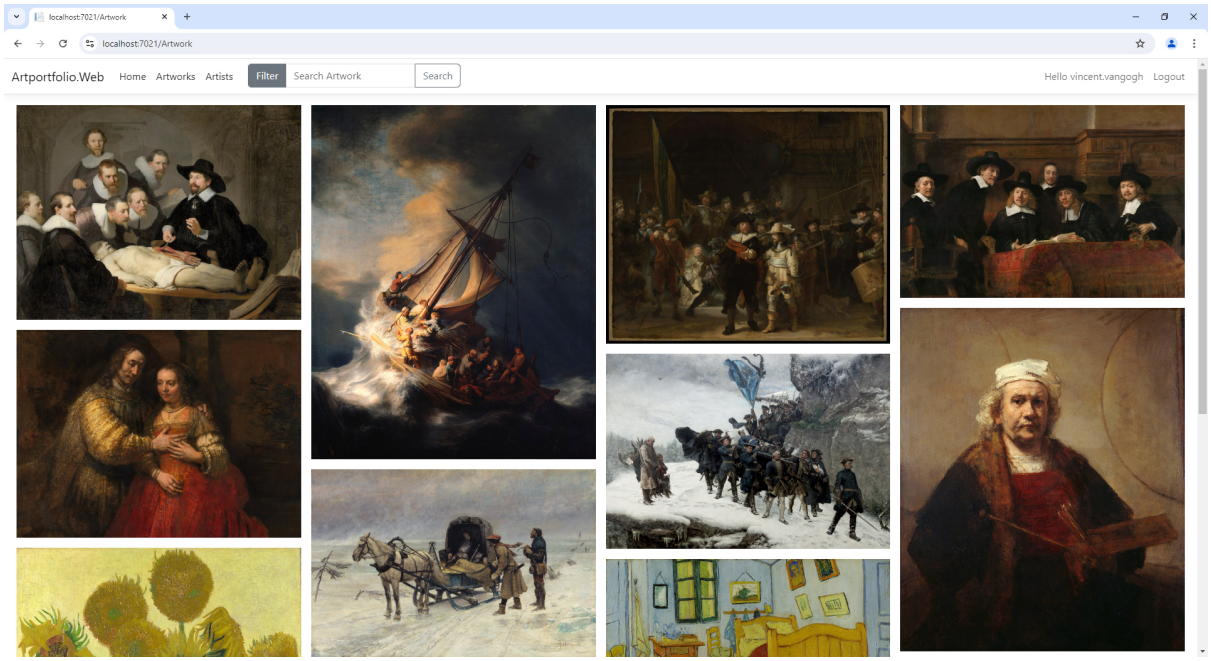


Figure 5: Artwork index view, where artworks from all artist are displayed. the navigation bar has options for filtering and sorting artworks.

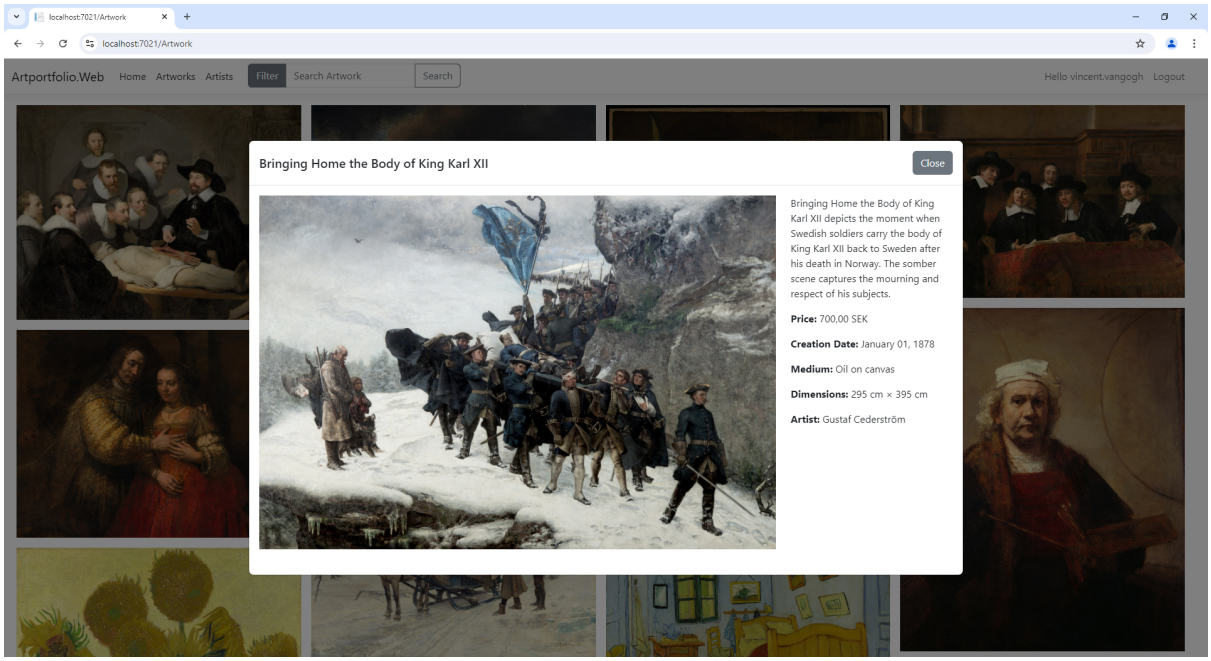


Figure 6: Artwork detail modal popup view. Admins and owner of the artwork also get the option to edit or delete the artwork from this view.

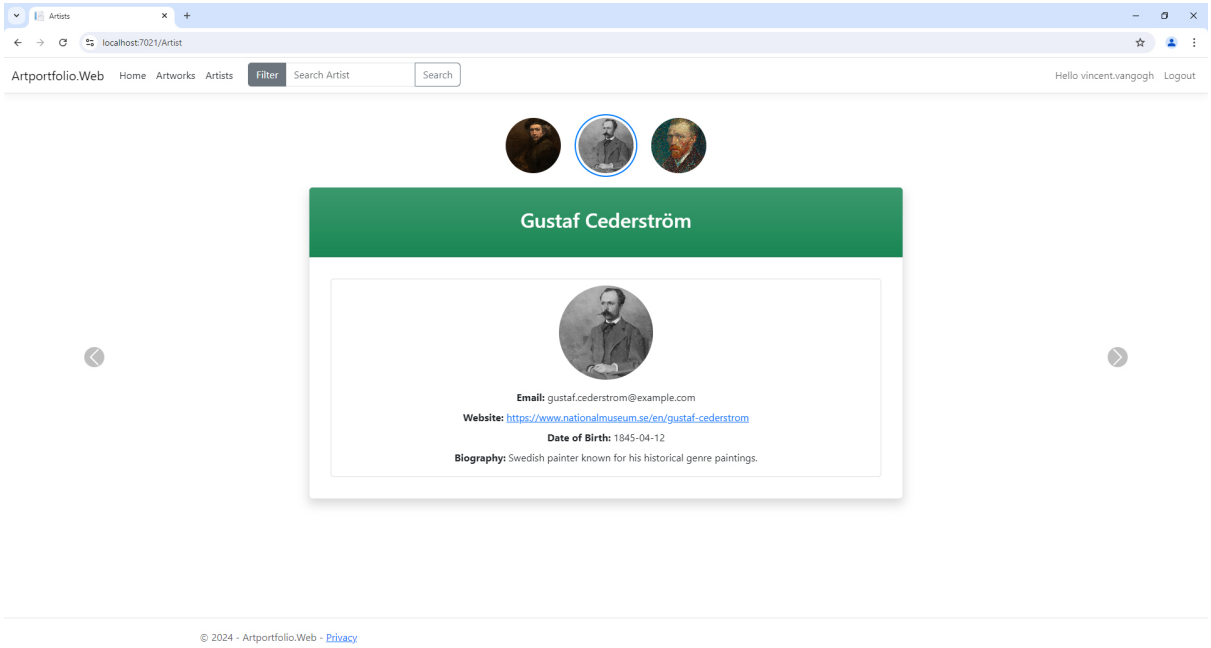


Figure 7: Artist index carousel view, where all artist are displayed. the navigation bar has options for filtering and sorting artists. Navigation between artist is done by using the preview button above, or the arrows at the sides.

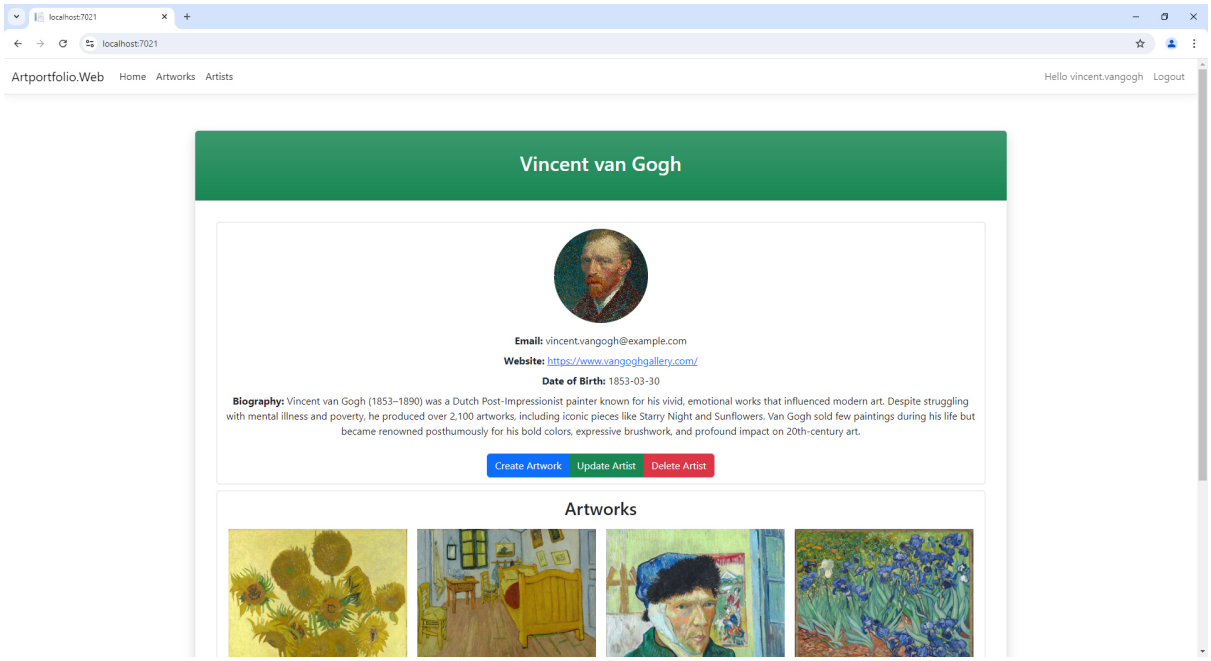


Figure 8: Home index view, the artist manages their artwork and profile from here.

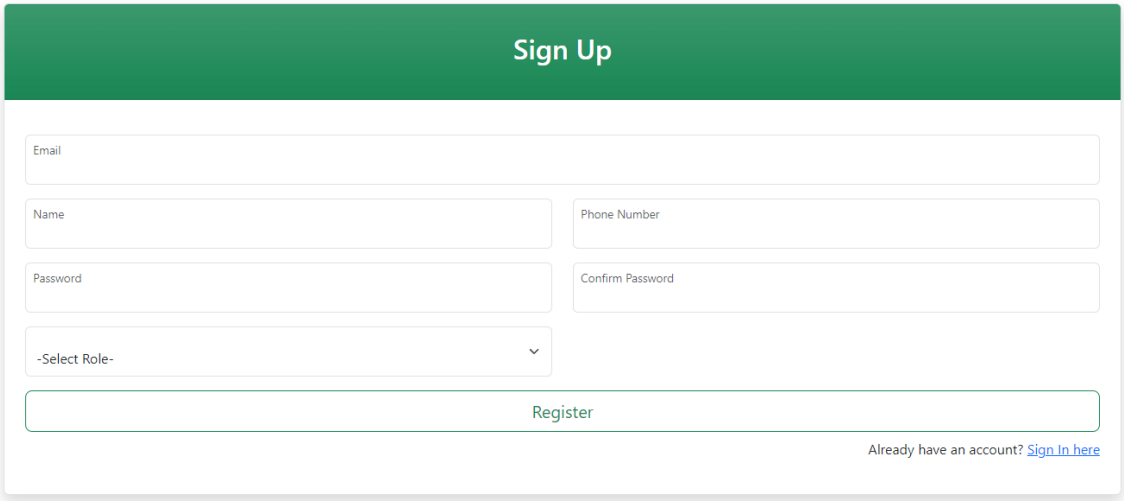


Figure 9: Sign up modal, each field has its own client side and server side validation feedback. Same principle for the login version.