Evolvium: STUDENT AUTOMATION PROJECT DETAILED DESIGN AND ANALYSIS DOCUMENT

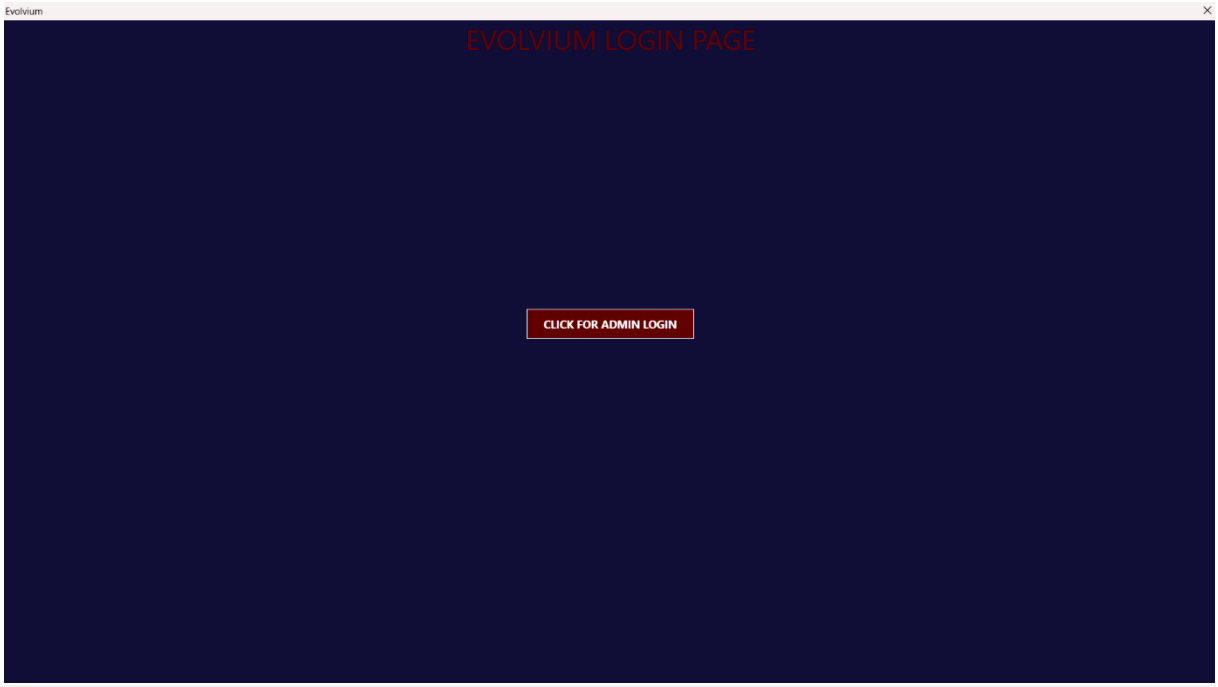
Contents

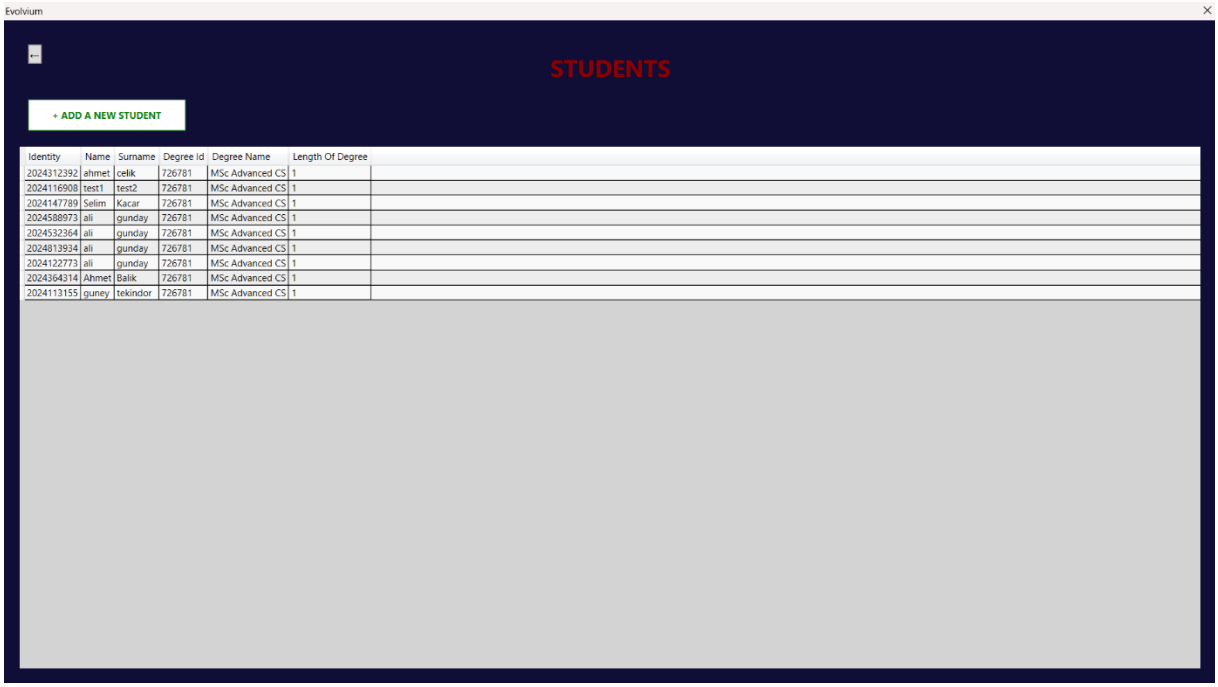
INTRODUCTION

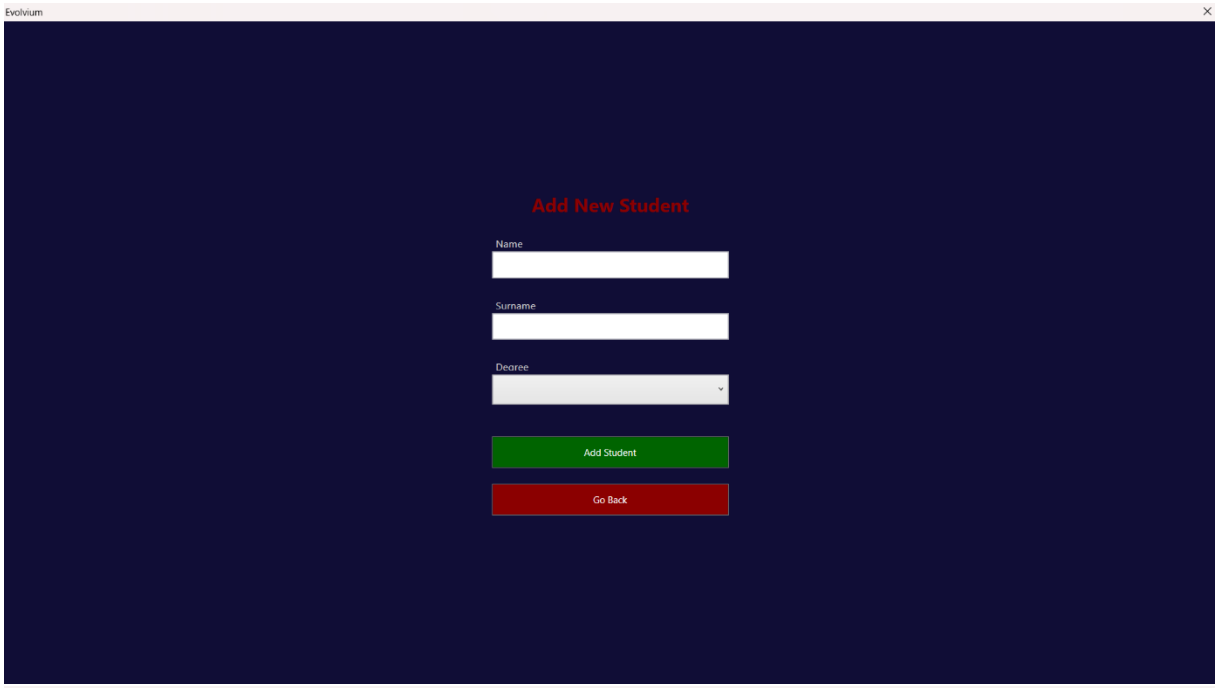
This project aims to follow students' progress, calculate grades automatically, and create reports for summaries. The project consists of seven screens Login, Dashboard, Student Management, Course Management, Module Management, Assessment Management, and Grades/Results Screens. This document was created to explain how the project is designed. It also shows the relationship between screens and their functionalities and explains the project's layers.

ANALYSIS OF PROJECT

User Interface Design

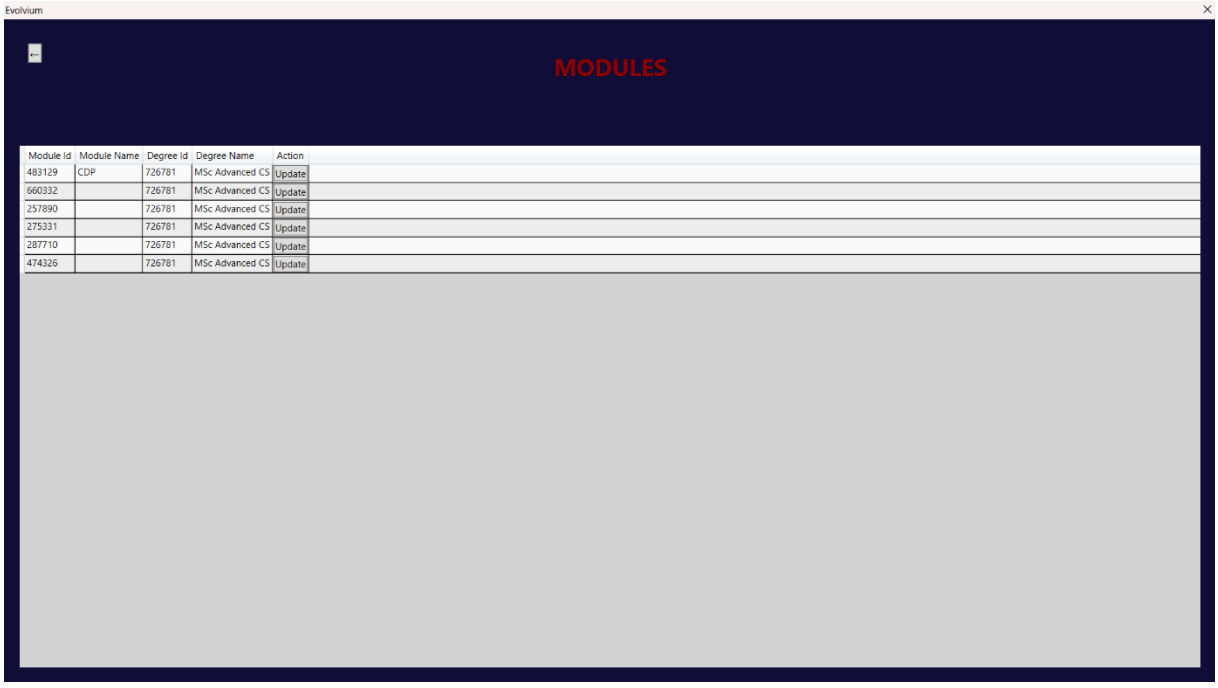


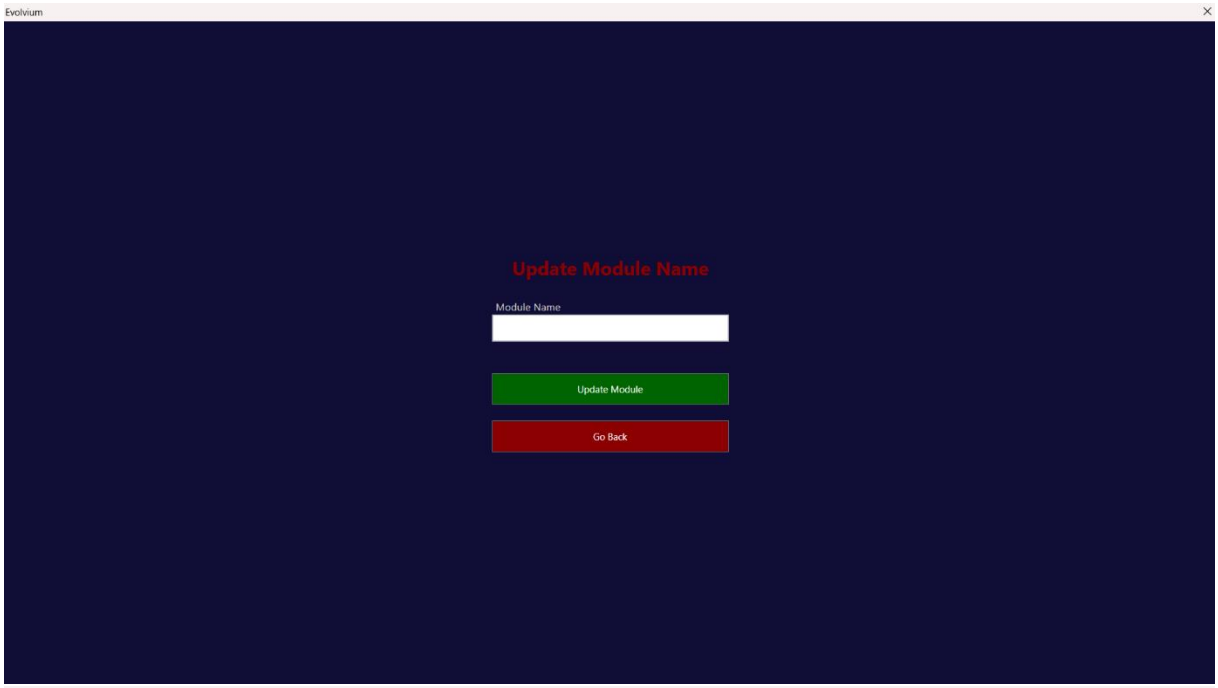


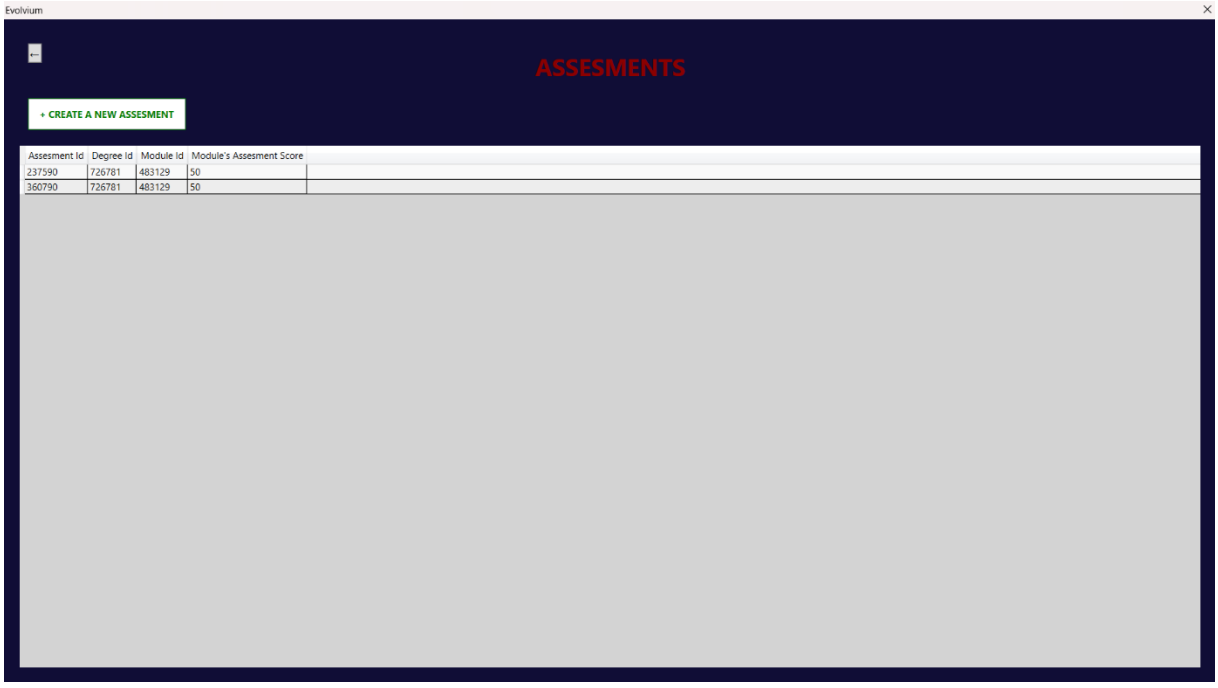

Evolvium
Add New Student
Name
Surname
Degree
Add Student
Go Back

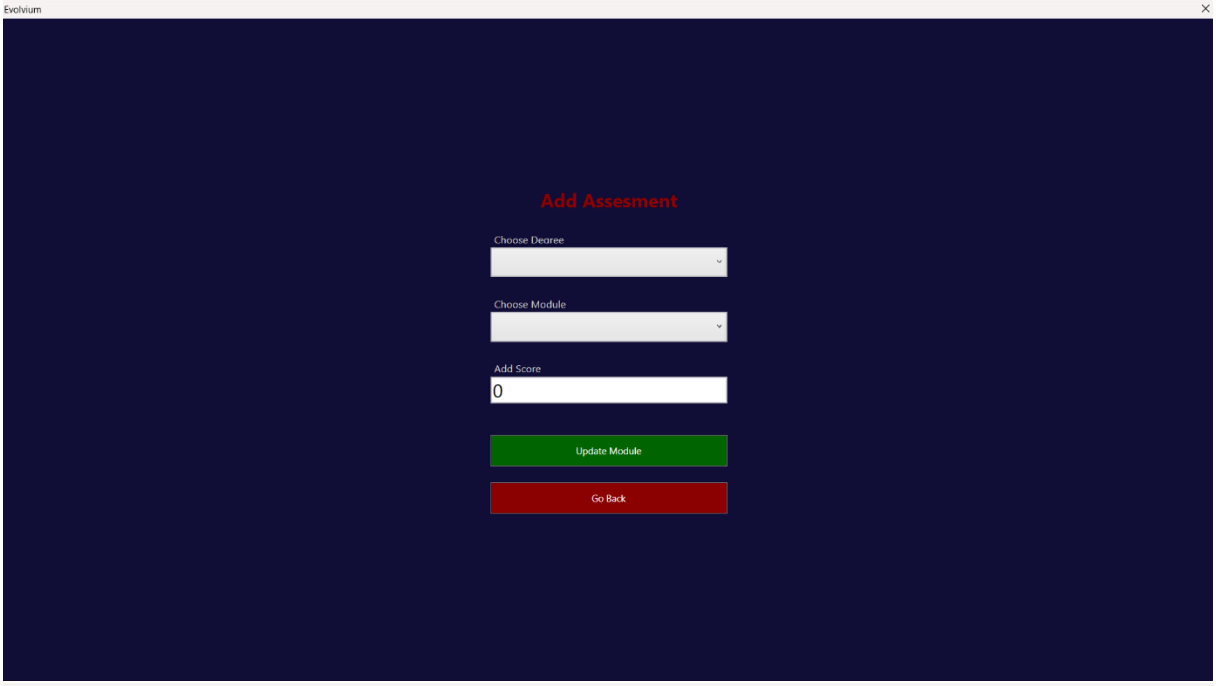

Evolvium
DEGREES
+ ADD A NEW DEGREE
Number
Name
Length Of Degree
Action
726781
MSc Advanced CS
1
Update

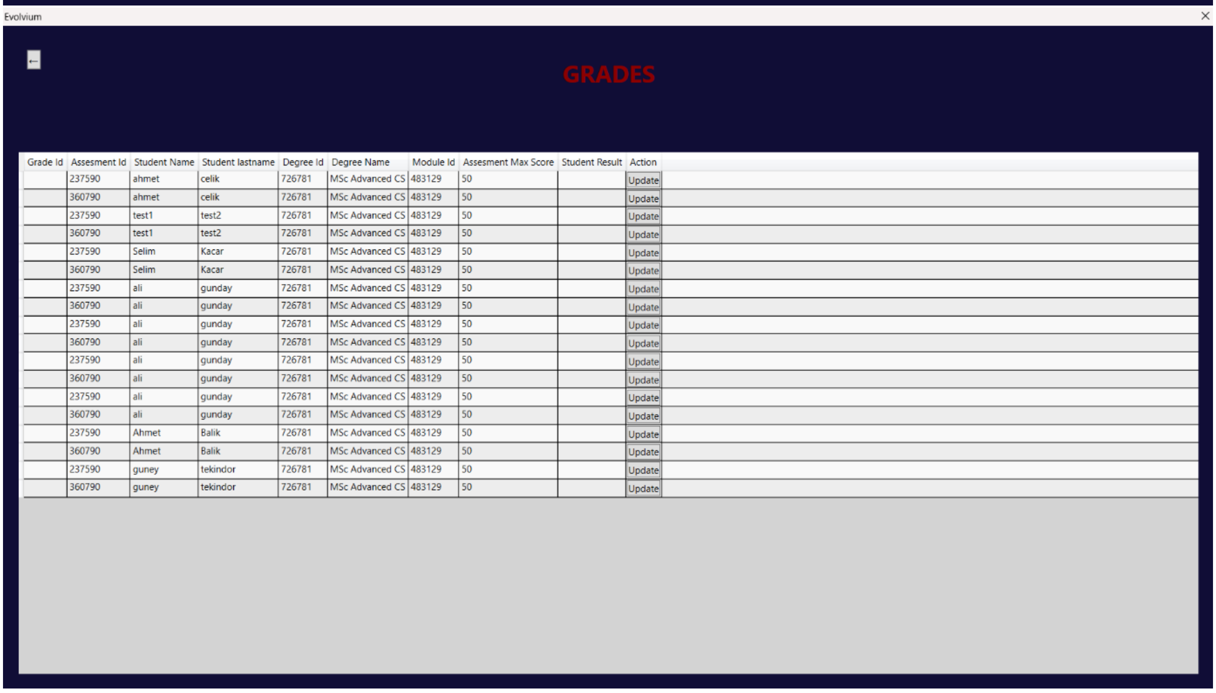

Evolvium
Add New Degree
Degree Name
Length Of Degree
Add Degree
Go Back


Evolvium
MODULES
Module Id
Module Name
Degree Id
Degree Name
Action
483129
CDP
726781
MSc Advanced CS
Update
660332
726781
MSc Advanced CS
Update
257890
726781
MSc Advanced CS
Update
275331
726781
MSc Advanced CS
Update
287710
726781
MSc Advanced CS
Update
474326
726781
MSc Advanced CS
Update

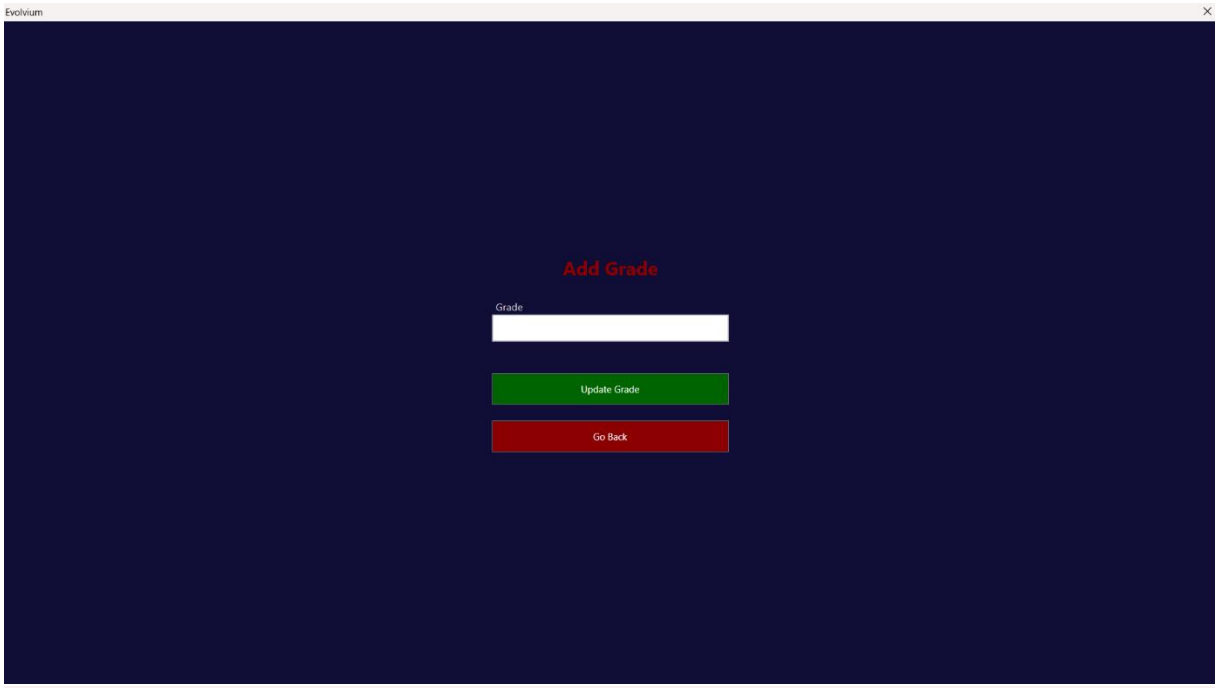

Evolvium
Update Module Name
Module Name
Update Module
Go Back

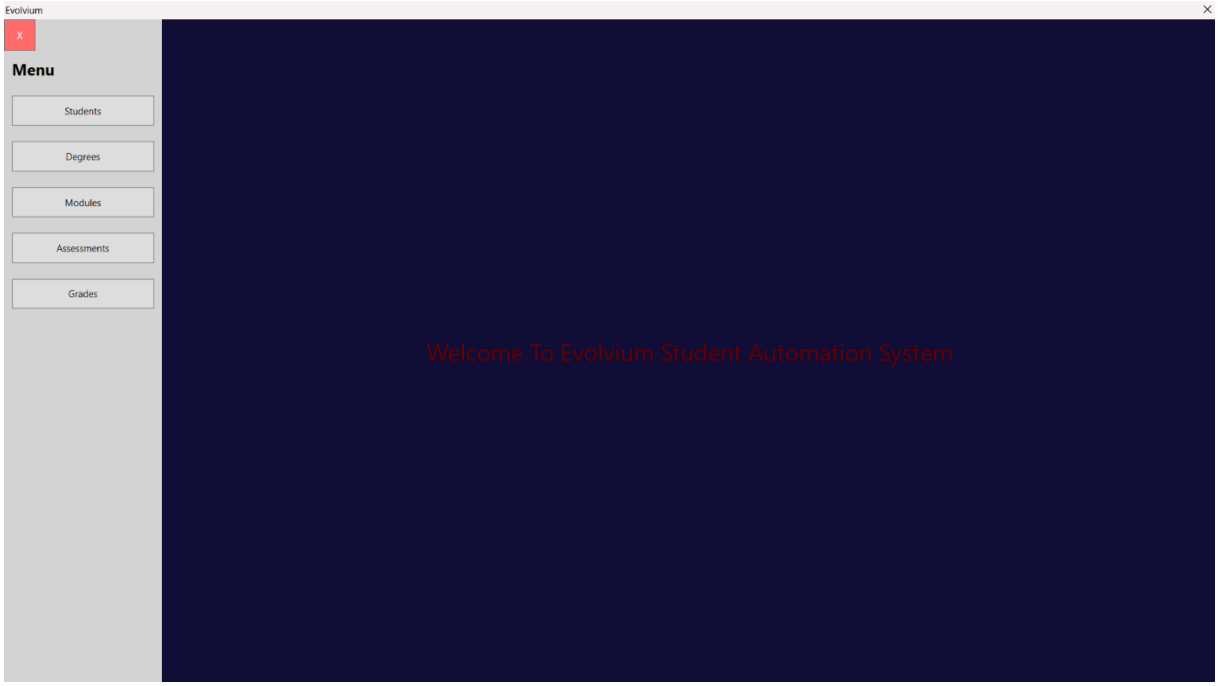

Evolvium
ASSESMENTS
+ CREATE A NEW ASSESMENT
Assesment Id
Degree Id
Module Id
Module's Assesment Score
237590
726781
483129
50
360790
726781
483129
50





Grade Id	Assesment Id	Student Name	Student lastname	Degree Id	Degree Name	Module Id	Assesment Max Score	Student Result	Action
	237590	ahmet	celik	726781	MSc Advanced CS	483129	50		Update
	360790	ahmet	celik	726781	MSc Advanced CS	483129	50		Update
	237590	test1	test2	726781	MSc Advanced CS	483129	50		Update
	360790	test1	test2	726781	MSc Advanced CS	483129	50		Update
	237590	Selim	Kacar	726781	MSc Advanced CS	483129	50		Update
	360790	Selim	Kacar	726781	MSc Advanced CS	483129	50		Update
	237590	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	360790	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	237590	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	360790	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	237590	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	360790	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	237590	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	360790	ali	gunday	726781	MSc Advanced CS	483129	50		Update
	237590	Ahmet	Balik	726781	MSc Advanced CS	483129	50		Update
	360790	Ahmet	Balik	726781	MSc Advanced CS	483129	50		Update
	237590	guney	tekindor	726781	MSc Advanced CS	483129	50		Update
	360790	guney	tekindor	726781	MSc Advanced CS	483129	50		Update





Business Logic

1. Navigation Logic

The application contains a NavigationService class, which enables smooth navigation between the different views (pages).

Implementation Details:

The NavigationService identifies the pages dynamically by their names. It constructs a fully qualified type name based on the name of the page that was passed and resolves it by using reflection.

A NavigateTo method is used to move to a specific page. This method allows passing an optional parameter that the target page can consume.

Dependency: NavigationService must have an instance of a Page to work with the WPF navigation system. Example Use Case In the ModulesViewModel, upon execution of the EditCommand, the method NavigationService.NavigateTo should be called, in order to move to the ModulesForm page with an associated parameter, for example. ModuleId to edit. 2. ViewModel and Command Logic

The ModulesViewModel is the class responsible for managing the data and interactions for the "Modules" page. It encapsulates business logic for fetching, editing, and displaying module data.

2. Commands:

LoadModulesCommand: Loads the module data asynchronously. It fetches data from a REST API (http://localhost:5218/api/Modules) into an observable collection called Modules.

EditCommand: This command will invoke some navigation logic when the intention is to edit a given module. It will navigate the user to the relevant page by using NavigationService, passing the ID as a parameter.

Async Data Loading:

LoadModulesAsync ensures that data of the modules is fetched without blocking. The method is fail-friendly; in case of an exception, it will show error messages through a MessageBox.

Pulled modules are added to the Modules observable collection to ensure real-time UI updates.

Example Workflow:

On page load, the LoadModulesCommand will be fired automatically to load the list of modules into the UI.

The user will select a module to edit. The EditCommand fires, navigating to the ModulesForm page with the ID of the selected module.

3. Data Handling Logic

Use of HttpClient to connect with REST API to pull data on modules.

Base Configuration

The HttpClient is provided with a base address, http://localhost:5218/, to simplify relative URI handling when making API calls.

Error Handling:

LoadModulesAsync data-fetching logic is wrapped in a try-catch block for better error handling, whether due to API or connectivity issues.

In case an error occurs, it shows a friendly message via MessageBox.Show.

Example API Call:

The application performs a GET request against http://localhost:5218/api/Modules.

It deserializes the response into modules and adds them dynamically to the UI.

4. Parameter Passing and Dynamic Page Binding

Business logic also provides the ability to dynamically pass parameters between pages with the purpose of maintaining the data consistency or sharing context if needed.

In cases of navigation to ModulesForm, the parameter argument in the NavigateTo method holds the ModuleId to point at which module is being edited.

This is received on the receiving page in its initialization and accordingly changes its state, say, pre-populating form fields.

5. Dependency Management Without Constructors

In order to simplify the dependency management and avoid constructor-based initialization issues, the project will integrate a ServiceProvider for dependency injection. This will ensure that no constructor injection is directly used but instead each component, for example, ModulesViewModel, NavigationService is created and maintained by the service container.

Example Dependency Injection Setup:

ModulesViewModel is registered in the service container as a transient service, meaning a new instance will be created for every request.

NavigationService is registered as a singleton, meaning there is only one instance of it in the application.

6. Command Triggering and UI Interaction

Commands such as LoadModulesCommand and EditCommand are directly bound to UI actions, for example, button clicks or page initialization. These commands encapsulate business logic, ensuring a clean separation of concerns.

Example of Edit Workflow:

User clicks the "Edit" button for a module.

The EditCommand is executed, which triggers navigation to the ModulesForm page with the data of the selected module.

API

The API section of the document focuses on the integration between the application and the REST API used to manage the modules data. This API serves as the backbone for data retrieval and manipulation, allowing the application to remain dynamic and up-to-date. The business logic in the application relies heavily on the API for fetching data, ensuring that the UI reflects the latest state of the underlying data source.

The API that is used in the application is hosted locally at http://localhost:5218/, and its endpoint /api/Modules handles all module-related operations. While the current implementation focuses primarily on retrieving data, the API supports CRUD operations to manage modules effectively.

In the application, an instance of the HttpClient class is used to communicate with the API. The BaseAddress property of HttpClient is set to the root URL of the API so that relative URI handling could be much easier. The application sends a GET against the /api/Modules endpoint to fetch module data. The response is expected as JSON, which is to be deserialized into a list of Module objects. These objects are then added to the Modules collection that is bound to the UI, which displays the list of modules.

To ensure a robust user experience, API calls are wrapped in error handling. If the API is unreachable or returns an error, the application catches the exception and displays a friendly error message to the user. This prevents the app from crashing and lets the user know what's going on, so they can do something to fix it, such as checking their internet connection or contacting support.

The integration of APIs has been done in an asynchronous way using the async and await keywords in C#. By making this design, any calls to APIs will not lock down the main thread but be responsive even when it has to fetch data, and it takes time. Finally, the method LoadModulesAsync encapsulates this whole API interaction, fetch data, and update the UI seamlessly. This is fired both on page initialization as well as through a command, which allows the user to refresh the data anytime if needed.

While the current scope of API interaction is limited to fetching module data, the architecture is flexible enough to accommodate additional endpoints in the future. For instance, endpoints responsible for creating, updating, or deleting modules could easily be integrated by extending the HttpClient configuration and adding new methods in the business logic layer. This ensures modularity within the design, which will enable the application to evolve with the API.

In conclusion, the API is a critical component of the application, providing the necessary data for the core functionalities. The seamless integration, coupled with error handling and asynchronous programming, ensures a smooth user experience while maintaining flexibility for future enhancements.

Data Storage

This is discussed in the "Data Storage" section of the document and discusses how the application manages or stores data, particularly its approach to storing, fetching, and displaying module data. The application does not make use of a local database or any form of in-app persistence but is based on an API as regards dynamic fetching of module data in runtime. This approach shifts the responsibility of data storage and management to the server, ensuring the application will always work with current, precise data.

The application is designed to work with an ObservableCollection<Module> on the client to temporarily store the module data while the application is running. Such a collection is the perfect intermediary between API-provided data and the UI representation of it. It's ObservableCollection that will turn out useful because it is able to refresh the UI automatically if one or more items of it were changed, and the applications will reflect all the changes automatically without manual refreshing.

It abstracts the storage of data on the server side from the application's business logic. Module data is made available through the /api/Modules endpoint and is presumably stored in a backend database. The database technology used, such as SQL Server, PostgreSQL, or MongoDB, is not directly relevant to the client application, as the REST API encapsulates these details. This separation of concerns allows the application to focus solely on the fetching and displaying of data without managing complex database operations itself.

The temporary nature of data storage in the application aligns with its design philosophy. Since the application fetches data on demand and does not store it locally, the list of modules is reloaded each time the application is launched or the user triggers a refresh. This ensures that the application always displays up-to-date data, avoiding potential inconsistencies that could arise from outdated local storage.

However, this approach does have implications. For example, the app will require an active network connection to fetch data about modules because it is dependent on an API. If this API is down or the user is without internet access, there is no way for the application to pull or even show module data. It is for this reason that error-catching mechanisms are in place to present such issues to the user and ensure that even with bad scenarios, the UX is clear and responsive.

In summary, the application follows a very lightweight data storage model, leveraging temporary in-memory collections on the client side and deferring persistent storage to the server via the API. This is a simplification for the client application and allows for flexibility and scalability because changing the backend data model or storage technology does not necessitate changes in the application itself.

LOGICAL DESIGN

The "Logical Design" section outlines the structural and conceptual framework that defines how the application functions internally and interacts with external components. This design focuses on the logical arrangement of components, their responsibilities, and the flow of data and commands within the application to ensure a cohesive and maintainable architecture.

The core of the application's logical design is based on the MVVM architectural pattern, which keeps concerns separate and code clean. The Model layer represents the data structures and entities, such as Module objects, which are used to encapsulate the module-related information retrieved from the backend API. These entities are designed to be lightweight and easy to work with, serving as the building blocks for the application's data.

The ViewModel layer sits between the Model and the View. The ModulesViewModel class in this layer is crucial to contain the business logic of the application and the state of the UI. It retrieves data from the API regarding modules and caches them in an ObservableCollection bound to the UI components in the View. This design ensures that any changes in the data are automatically reflected in the UI, maintaining a reactive and dynamic user experience. Furthermore, the ViewModel exposes commands, such as LoadModulesCommand and EditCommand, which enable user interactions like loading modules or navigating to an editing screen. These commands encapsulate the logic required for these actions, ensuring that the View remains free of direct logic and focuses solely on presentation.

The View layer consists of the user interface elements defined in XAML. The Views are bound to the ViewModel through data binding, which allows the UI to display and interact with data without being tightly coupled to the underlying logic. This separation ensures that the UI can be easily modified or replaced without altering the application's core functionality.

Another important aspect of the logical design is navigation, which is handled by a custom implementation of the NavigationService. This service encapsulates the navigation logic, enabling the ViewModel to initiate navigation actions without knowing the details of the UI or the navigation framework being used. Decoupling navigation from the ViewModel enhances reusability and testability.

The logical flow of the application starts with the ModulesViewModel fetching data from the backend API using an HttpClient. Once retrieved, this data is populated into an ObservableCollection which the View will automatically render. User interactions, such as selecting a module for editing, fire the commands in the ViewModel which then call the appropriate methods, for example, navigating to another page via the NavigationService.

It gives the components a well-defined role in this logical separation, allowing better maintainability and scalability of the application. Boundaries are clear between the Model, View, and ViewModel, which helps to facilitate code reuse and allows each layer to be developed, tested, and modified independently. All in all, the logical design is robust, follows the best practices, and serves as a good base for further enhancements.

STATE INVARIANTS

The "State Invariants" section of this document will discuss the rules and conditions that define and maintain the consistent state of the application throughout its life cycle. State invariants are crucial in ensuring that the application works predictably and with no errors, whatever the input or operation done by the user. These invariants ensure the internal data and logic of the application are valid, coherent, and maintain the intended functionality.

In this application, the state invariants are related to the relations between the View Model, Model, and View, and the synchronization of data fetched from the backend API. One of the main state invariants is that the Modules collection in ModulesViewModel should always exactly correspond to the data fetched from the API. This means that any addition, removal, or updating of this collection should occur only after the data has been validated and processed to avoid inconsistency. For example, in loading modules from the API, the application ensures that only valid and complete data objects are added to the collection to protect the UI from displaying incomplete or erroneous information.

Another important invariant is related to navigation. NavigationService guarantees that navigation will be performed only when valid parameters are given. For instance, navigation to the "ModulesForm" page must have a valid moduleId so that it can be sure which module is being edited. In this way, this invariant prevents any invalid or null parameters from disturbing the course of navigation or causing run-time errors. The application does not let this rule break in order to keep it predictable.

Similarly, there are specific invariants for Commands in the ViewModel, such as LoadModulesCommand and EditCommand: these one should be executed only if an application gets into a state where such operations are allowed. For instance, the LoadModulesCommand shouldn't be run in parallel with another operation of data loading to avoid data corruption or performance degradation. Similarly, the EditCommand can only be executed if the parameter passed to it is not null, ensuring that edit operations are only initiated for valid module entries.

The state invariants also extend to error handling and user notifications. For example, if the API fails to provide the requested data, the application should handle this gracefully and show an appropriate error message to the user, not leaving the application in some undefined or partially loaded state. This invariant ensures the user experience remains clear and informative, even in the face of unexpected issues.

Following these state invariants keeps the application consistent and predictable. These invariants are guardrails that, by following them, allow operations and transitions within an application to occur in expected and predictable ways. Not only does this make for a more stable application, but debugging and future development will also be easier because the rules of how the state should change are explicitly stated.

DATA DESIGN

The "Data Design" section focuses on the structured organization and representation of the application's data to ensure efficiency, accuracy, and maintainability. This section will explain how the data is modeled, stored, and manipulated within the application to meet its functional requirements.

Data design in this application begins with the definition of the primary data entity-the Module object. Each module will represent an individual unit within the greater system, complete with properties that are essential to its identification and interaction. This could include attributes such as ModuleId, Name, Description, and other fields required for displaying module details or performing operations like editing or navigation. The Module class encapsulates this data, ensuring consistency in its representation throughout the application.

The next layer of data design involves the ModulesViewModel, which acts as a bridge between the backend data and the user interface. This ViewModel will hold an ObservableCollection of Module objects; this is the dynamic, real-time representation of the modules fetched from the API. An ObservableCollection is utilized here because it automatically updates the UI any time elements are added, removed, or updated in the collection. This design choice reinforces the reactive nature of the application and provides users with a seamless and responsive experience.

Data design also accounts for how information is retrieved and stored. Data is fetched from the backend API via an HTTP client that is configured to communicate with a specific base URL. The application makes sure that whatever data is received from the API is validated before being integrated into the Modules collection. This step is very important, so that the integrity of data is maintained and there is no generation of errors related to incomplete or malformed responses. In this way, the application keeps its word and shows the user valid meaningful information.

Furthermore, the design includes the use of commands like LoadModulesCommand, which enables the treatment of data-loading operations in an asynchronous manner. This is not only good for performance in that the UI does not freeze, but also it promotes a clean separation of concerns: the ViewModel is responsible for orchestrating data retrieval and manipulation, while the API acts as the source of truth regarding module information.

Other than that, error handling is another important feature in the data design. If there is some problem when retrieving data-for instance, due to an unsuccessful API request-the application will

handle this error by providing a notification to the user and keeping the state of the application consistent. The rationale behind this design choice was centered around the user experience, ensuring the application didn't become unstable because of unexpected issues with data.

In summary, the application data design is based on a robust foundation that provides data integrity and usability throughout. It balances functionality and reliability by structuring data entities clearly, maintaining a reactive data layer, and enforcing rigorous error handling. Such forethought in the design of data not only makes it easier to develop and maintain the application but also provides users with a reliable experience.

TOOLS

The "Tools" section describes the software, frameworks, and libraries that are most important and used during the creation of the application. In fact, each of these tools plays a very vital role in making this application functional, efficient, and maintainable. This section explores the rationale behind choosing these tools and how they contribute toward the overall success of the project.

The.NET framework will play a major role in this application since it forms the backbone on which the application stands.NET presents a perfect and flexible environment for the implementation of applications with unique built-in support for MVVM, data binding, and solid API integration. Its in-depth libraries and tools help developers go through the process easily by allowing different layers to pass information easily among them. By using.NET, the project will get the benefit of a mature and stable platform that gives scalability and performance.

The UI and presentation aspects of the application will be handled by XAML. XAML allows for a declarative way of doing UI, which makes creating rich and dynamic interfaces relatively easy. Its tight integration with.NET and the MVVM pattern makes for a clear separation of concerns, which in turn makes it easier to manage the interaction between the visual components and the underlying logic. XAML is flexible enough to ensure that the interface can be adapted or extended without significant disruption to the overall structure of the application.

For handling the communication of data, HttpClient is used. It is crucial for consuming RESTful APIs. This client is efficient but also highly configurable, allowing a fine grain of control over what exactly goes forth and comes back from client to server using it. Using HttpClient ensures that an application is able to fetch and manipulate data reliably, a bridge between a user interface and a back-end.

To handle navigation within the app, there is a custom service called NavigationService. Although this is not an external tool, it is an important part of the architecture, which should enhance modularity and flexibility in the application's navigation logic. It works seamlessly with the MVVM structure to ensure that decisions about navigation are encapsulated within one reusable component. This design choice eliminates redundancy and enhances maintainability.

Visual Studio used in this project is an integrated development environment that will provide the workspace for writing, debugging, and testing the code. Some important features of Visual Studio are powerful debugging, IntelliSense, and Git integration, which contribute a lot to speeding up the development while keeping quality at its best. Its interface allows easy collaboration and project management.

Version control is handled with Git, a critical tool for tracking changes, collaborating with team members, and maintaining a reliable history of the project's evolution. It ensures the development process is organized, and any probable issues arising can be easily resolved by reverting to previous versions when needed. By integrating Git with platforms like GitHub or Azure DevOps, the project has a very robust workflow for collaborative development and deployment.

These tools are selected with care in order to optimize the development process for the best product. Each tool is complementary to the others, making the environment cohesive in which the application's logic, design, and data communication will work together. These tools not only streamline the development process but also provide a solid foundation for future growth and enhancement of the application.