

## Contents

#1 Project Formulation & Initial Requirements:	2
Project Description (Domain)	2
Why This Project?	2
Requirements (User Stories)	2
Delimitations	3
Potential Features Not Prioritized in the Initial Version:	3
#2 Web Service Design & Implementation	4
<b>HOW WAS WORKING WITH RESTFUL API AND UNDERLYING REASONS</b>	4
The flow of browsing initiative posts.	6
Adding Authentication and Authorization	8
Handling Relationships and Navigation Properties	9
Using RESTful Principles	9
<b>OVERVIEW OF ENDPOINTS:</b>	10
API Documentation overview	10
Auth	10
Categories	11
InitiativePost	12
Register	15
Users	15
<b>KEY REQUIREMENTS</b>	17
User registration and login	17
The ability to submit, view, edit, and delete product development initiatives	22
TO SUMBIT A NEW INITIATIVE	22
TO EDIT INITIATIVES	24
TO DELETE INITIATIVE	26
Communication between the Blazor frontend and the ASP.NET Core Web API	27
<b>AN OVERVIEW OF THE PAGES IN YOUR WEB APPLICATION.</b>	28
<b>ACCOUNT.RAZOR</b>	28
<b>BROWSE.RAZOR</b>	29
<b>EDITINITIATIVE.RAZOR</b>	29
<b>HOME.RAZOR</b>	30
<b>LOGIN.RAZOR</b>	30
<b>REGIISTER.RAZOR</b>	30
<b>WRITEINITIATIVE.RAZOR</b>	31

<b>HOW FRONTEND CONNECTS TO YOUR WEB SERVICE. ....</b>	<b>31</b>
IN THE CLASS ATELIERCONTEXT.CS .....	32
#5 Data Access: .....	32
.....	36
#6 Project Conclusion & Demonstration: .....	39

## #1 Project Formulation & Initial Requirements:

### Project Description (Domain)

The DND course project focuses on developing a platform that enables users to share, categorize, and evaluate product development initiatives. The platform will facilitate user registration, authentication (login/logout), and submission of initiatives on product development. Each initiative will include a descriptive title and detailed information about the proposed idea. Users will also have the ability to assign initiatives to specific functional areas of product development.

### Why This Project?

This project was designed as an ad hoc solution with the future internship in the Next Generation Platform within the product development team in mind. A significant part of the internship focuses on exploring how Artificial Intelligence can be integrated across various functional areas. To support this research, I developed an application that empowers users to contribute meaningful ideas, leveraging their expertise and experience to drive change. Recognizing that people are the company's greatest asset, this platform emphasizes the importance of capturing and amplifying their voices.

The project incorporates essential software development principles, including the implementation of RESTful APIs to ensure seamless communication between systems and using Blazor to provide an intuitive and engaging interface, fostering active participation and collaboration. Thus, the aim is to create a platform that is practical, user-driven, and scalable. The project also allows us to deal with real-world challenges like managing user-generated content and securing user data.

### Requirements (User Stories)

#### User Registration & Login

- As a new user, I want to create an account, so I can submit my initiatives.
- As an existing user, I want to log in securely using my username and password, so I can access my profile and submit or manage my initiatives.

#### Submit Product Development Initiatives

- As a logged-in user, I want to create a new initiative post where I can share my initiative, including details like title, content, and the area of product development.
- As a user, I want to edit or delete my previous initiatives, so I can manage and update my content.

### **View Product Development Initiatives**

- As a visitor or logged-in user, I want to view a list of all initiatives submitted by other users, so I can read about their initiatives.
- As a visitor, I want to browse initiatives by category, so I can easily find initiatives related to specific areas.

### **Initiative Viewing for Guests**

- As a visitor (without logging in), I want to browse and view initiatives, so I can read about initiatives even if I haven't created an account.

### **Role-Based Access (Admin vs Regular User)**

- As an admin, I want to moderate all submitted initiatives, so I can ensure inappropriate or irrelevant content is removed.
- As an admin, I want to delete or edit any user's initiative, so I can maintain the quality of the NextGenAtelier platform.

### **User Authentication & Security**

- As a user, I want my account to be protected with a secure password hashing mechanism, so my credentials are safe.

### **Initiative Pagination**

- As a user, I want to view the initiatives in a paginated format, so I can easily browse through multiple initiatives without overwhelming the interface.

## **Delimitations**

While the project concept includes a broad range of features and user stories, it's important to note that due to time constraints, resource limitations, and the complexity of some features, plus working solo, I may not be able to implement all the desired functionalities within the given project timeline.

The core focus of the project will be on the following required functionalities:

- User registration and login
- The ability to submit, view, edit, and delete product development initiatives
- Communication between the Blazor frontend and the ASP.NET Core Web API

## **Potential Features Not Prioritized in the Initial Version:**

- Advanced search and filtering of initiatives based on keywords.
- Profile management features beyond basic login and password change.

- Full mobile-responsive design or extensive UI/UX improvements.
- Uploading images or media within initiative posts (which could be considered as future enhancements).
- Extensive admin controls for moderating content beyond basic CRUD operations.

These additional features represent future extensions of the platform and would be considered if time and resources allow. However, for the initial phase, I will focus on meeting the primary technical requirements and delivering a working, secure, and user-friendly system.

## #2 Web Service Design & Implementation

### HOW WAS WORKING WITH RESTFUL API AND UNDERLYING REASONS

The core design purpose of incorporating a web service into the Atelier project lies in establishing a centralized system for managing data through a standardized interface (thus standardized communication) without navigating fragmented and disjointed data silos. Hence, owning a web service will ensure that all information—such as user-submitted initiatives are in a common or widely-accessible location. As a result, consistency will be ensured across the platform all stakeholders operate on the same data set. Moreover, a web service facilitates real-time updates that are instantly reflected. As the Atelier project evolves, the user base and the volume of data are expected to grow proportionally which can be supported mainly grace to a web service setting by exposing APIs that can be consumed by multiple front-end clients. In addition, the web service inherently provides mechanisms for robust security, ensuring that sensitive information, such as user credentials and proprietary initiatives, is protected. Features like JWT-based authentication and role-based access control (RBAC) allow for fine-grained user permissions, restricting access to authorized users only.

The RESTful architectural style was selected visionary to align with existing Siemens Gamesa infrastructure/digital ecosystem. The company's infrastructure includes modular platforms such as SAP for project management, TeamCenter for product lifecycle management, and SharePoint for document collaboration. These tools are designed to interact with external systems through standard protocols such as RESTful APIs or file-based data exchange. Atelier's API-first approach ensures seamless compatibility, enabling the export of initiative-related data and the import of updates or feedback from these platforms. This integration supports real-time synchronization, reducing redundancy and improving operational efficiency. Atelier's web service can export data in structured formats such as JSON or CSV, which are directly compatible with Power BI's data models. This capability enables live connections between Atelier and Power BI, allowing managers to visualize initiative details to generate insightful visuals. In the same context, by utilizing event-driven architecture, Atelier can trigger updates or actions in other systems. For example, when an initiative is created in Atelier, a webhook can automatically create a corresponding task in Siemens' SAP project management module. Similarly, in the future, the updates from Siemens' systems—such as changes in project status or resource allocation—can be pushed back into Atelier, ensuring all platforms remain synchronized. The integration feasibility is further strengthened by Atelier's adherence to modern security protocols, which

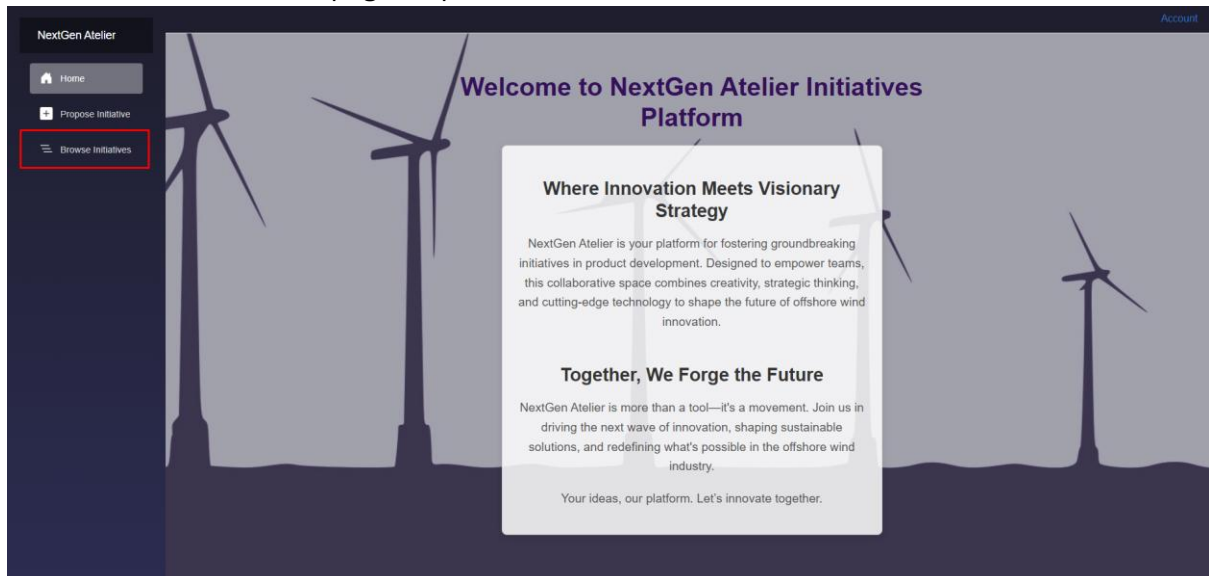
align with Siemens Gamesa's stringent requirements. Atelier's use of OAuth 2.0 for secure authentication ensures that only authorized users and systems can access its APIs, while TLS 1.3 encryption protects data during transmission. These features provide robust safeguards for sensitive product development data and proprietary initiatives, ensuring that integration does not compromise security or violate compliance standards. The scalability and incremental integration of Atelier's architecture are driven by its RESTful API design. Provide a modular resource-based structure, enabling specific features (e.g., retrieving or managing initiatives) to be integrated independently and expanded over time. Their stateless nature supports horizontal scalability, allowing the platform to handle growing traffic efficiently through load balancing and distributed processing. To conclude in a generalist way: the RESTful API design ensures that Atelier can evolve alongside Siemens Gamesa's needs, supporting phased integration while maintaining high performance, security, and adaptability for future system expansions.

Besides the reasons that has been discovered in the implementations , the original reason for the the choice of using RESTful APIs over other alternatives like SOAP, GraphQL, WebSockets, or RPC-based services was likely influenced by practicality and alignment with the project's requirements. REST is lightweight, easy to implement, and widely adopted, making it ideal for a platform that primarily involves CRUD operations. Its ability to handle stateless communication and support JSON format seamlessly integrates initially decided frontend (Blazor).

In retrospect, while considering whether other types of web services could offer competitive advantages ( making abstraction of the course requirements) - the conclusion is cREST stands out as the optimal solution, particularly when considering the nature of Atelier's core operations and potential future growth. For instance, GraphQL offers distinct advantages, particularly in scenarios involving highly nested data or selective data retrieval. However, Atelier's operational scope—focusing on creating, retrieving, updating, and deleting initiatives—does not necessitate such advanced querying capabilities. REST's ability to expose straightforward endpoints for these operations suffices, without introducing the schema design, versioning, and query optimization complexities inherent to GraphQL. On the other hand SOAP, while robust and highly secure, is overly complex for a project like Atelier. SOAP's reliance on strict XML formatting, WSDL-based service contracts, and verbose messaging would introduce unnecessary overhead for a platform focused on CRUD operations and straightforward data exchange. Additionally, Atelier's functionality—primarily request-response interactions—does not benefit from real-time data streaming. Implementing WebSockets in this context would add architectural complexity and resource overhead without delivering tangible value. REST's stateless communication model is far better suited for handling discrete user requests efficiently, aligning with Atelier's current use case. In conclusion the adoption of RESTful API was both a visionary and strategic decision.

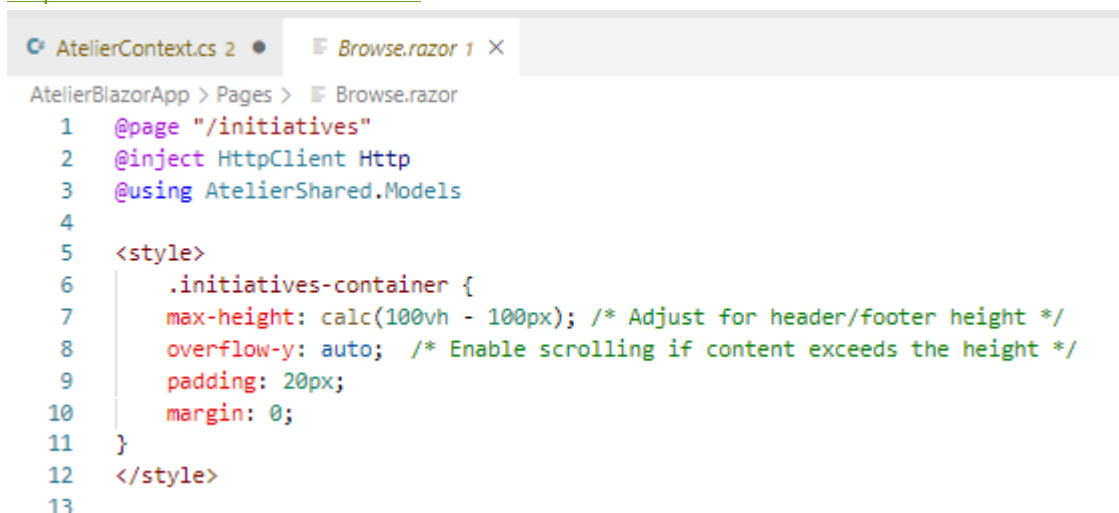
## The flow of browsing initiative posts.

When we are on the main page we press browse initiatives.



```
<div class="nav-item px-3">
  <NavLink class="nav-link" href="initiatives">
    <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Browse Initiatives
  </NavLink>
</div>
```

We are taken to /initiatives endpoint. This is the Browse.razor page which is loaded when <https://localhost:7106/initiatives> is called.



The method LoadInitiativesAsync() is responsible for calling the correct api endpoint that loads all the initiatives registered in the database. In this case <https://localhost:5062/api/InitiativePosts> is called.

```

@code {
    5 references
    private List<InitiativePost> Initiatives = new();

    1 reference
    protected override async Task OnInitializedAsync()
    {
        await LoadInitiativesAsync();
    }

    2 references
    private async Task LoadInitiativesAsync()
    {
        try
        {
            Initiatives = await Http.GetFromJsonAsync<List<InitiativePost>>("https://localhost:5062/api/InitiativePosts");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Failed to load initiatives: {ex.Message}");
            // Optional: Show a message to the user
        }
    }
}

```

The InitiativePostController.cs (class from the AtelierAPI directory) is currently responsible for handling all HTTP requests related to the InitiativePost entity, including creating, retrieving, updating, and deleting posts. It directly interacts with the AtelierContext database context to perform operations like fetching initiatives, saving new posts, and modifying existing ones. The BlazorApp method LoadInitiativesAsync() calls the GetInitiativesPosts() method from the controller through a http request to api/InitiativePosts

```

// GET: api/InitiativePosts
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<InitiativePost>>> GetInitiativePosts()
{
    try
    {
        // Include related Category navigation property
        var initiatives = await _context.InitiativePosts
            .Include(i => i.Category) // Load Category details
            .ToListAsync();

        return Ok(initiatives);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error fetching initiatives: {ex.Message}");
        return StatusCode(500, "An error occurred while fetching initiatives.");
    }
}

```

The \_context is a field created on the top of the controller and is an instance of the AtelierContext class. Currently, the controller directly accesses the database through \_context.InitiativePosts. This points to:

```

7 references
public DbSet<InitiativePost> InitiativePosts { get; set; }

```

, which corresponds to the

following visual representation:

Table: InitiativePosts						
	<u>Id</u>	AuthorId	CategoryId	Content	DatePublished	Title
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	1	3	Implement automated robotic systems for precision cutting and assembly ...	2024-12-06 11:08:04.7691788	Improve Blade Manufacturing Efficiency
2	5	4	4	Implement recycling programs in schools to reduce waste and educate...	2024-12-06 07:27:58.1707823	Recycling in Schools
3	6	4	3	Leverage artificial intelligence to optimize production lines and ...	2024-12-06 07:28:31.402357	AI for Manufacturing
4	7	4	2	Design and develop innovative prototypes for renewable energy ...	2024-12-06 07:28:48.5960692	Renewable Energy Prototypes
5	8	4	1	Provide a platform for students to develop and pitch their innovative ...	2024-12-06 07:29:03.2881859	Idea Incubator for Students
6	9	4	3	Develop eco-friendly packaging materials to reduce plastic waste i...	2024-12-06 07:29:19.4929967	Sustainable Packaging
7	15	8	4	This initiative focuses on developing highly efficient and ...	2024-12-06 11:00:22.3486289	Innovative Solar Panel Design
8	16	4	10	This initiative aims to develop an AI-powered system to assist ...	2024-12-06 11:09:02.3571038	Automated Compliance Monitoring System

This design works functionally but results in a tightly coupled structure where the controller handles both HTTP-specific logic and business logic, making it harder to maintain, test, and scale as the project grows.

To improve this structure, the `InitiativePostController` can be refactored to delegate the business logic and database interactions to a dedicated `InitiativeService` class. This would allow the controller to focus solely on HTTP request handling, such as validating inputs, managing authentication, and constructing appropriate HTTP responses. The service class would encapsulate all business logic, making the codebase more modular, testable, and scalable, while also promoting the principles of separation of concerns and single responsibility. This change would enhance maintainability and adaptability, especially as the application grows or becomes complex. Subsequently, this change can be applied to all the controllers in the current state.

## Adding Authentication and Authorization

Some endpoints required authentication, ensuring only authorized users could perform certain actions. For example, only authenticated users could create or delete their own initiatives. This was achieved by adding the `[Authorize]` annotation, ensuring only the authorized users can access this endpoint.

```
// POST: api/InitiativePost
[HttpPost]
[Authorize]
0 references
public async Task<ActionResult<InitiativePost>> PostInitiativePost(InitiativePost initiativePostRequest)
{
    // Extract User ID from the token using the correct claim type
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

    if (string.IsNullOrEmpty(userId))
    {
        return Unauthorized(new
        {
            message = "User ID not found in token.",
            claims = User.Claims.Select(c => new { c.Type, c.Value }).ToList()
        });
    }
    . . .
}
```



## Handling Relationships and Navigation Properties

The API leveraged EF Core's capabilities to include navigation properties, such as the Category associated with an initiative post. This ensured that related data was available in responses without requiring separate calls.

```
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<InitiativePost>>> GetInitiativePosts()
{
    try
    {
        // Include related Category navigation property
        var initiatives = await _context.InitiativePosts
            .Include(i => i.Category) // Load Category details
            .ToListAsync();

        return Ok(initiatives);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error fetching initiatives: {ex.Message}");
        return StatusCode(500, "An error occurred while fetching initiatives.");
    }
}
```

## Using RESTful Principles

I adhered to RESTful principles by:

- **Using proper HTTP methods:**
  - GET for retrieving resources
  - POST for creating resources
  - PUT for updating resources
  - DELETE for removing resources
- **Returning appropriate HTTP status codes:**
  - 200 OK for successful requests
  - 201 Created for resource creation
  - 404 Not Found for missing resources
  - 401 Unauthorized for authentication failures

## Conclusion

By following RESTful principles and leveraging the ASP.NET Core framework, I built a web API that is structured, scalable, and adheres to best practices. The API supports secure operations,

relational data handling, and comprehensive error management, ensuring a robust foundation for client applications.

## OVERVIEW OF ENDPOINTS:

One of the core features of adopting the architectural style of a RESTful API is accessing resources using URLs allowing identification of a specific and structured resource from the platform spectrum of existing web addresses that subsequently establishing clear communication between client and server. RESTful APIs are stateless, meaning every request from the client to the server must include all the necessary information to process the request. Endpoints help achieve this by clearly defining the resource and the required data.

The repository is structured into three main projects: AtelierAPI, AtelierBlazorApp, and AtelierShared. The AtelierAPI project contains controllers that define the API endpoints for Auth, Categories, InitiativePosts, and Users.

Auth		^
POST	/api/Auth/register	▼
POST	/api/Auth/login	▼
Categories		^
GET	/api/Categories	▼
InitiativePost		^
GET	/api/InitiativePosts	▼
POST	/api/InitiativePosts	▼
GET	/api/InitiativePosts/{id}	▼
DELETE	/api/InitiativePosts/{id}	▼
PUT	/api/InitiativePosts/{id}	▼
GET	/api/InitiativePosts/user	▼
Register		^
POST	/api/Register	▼
Users		^
GET	/api/Users	▼
POST	/api/Users	▼
GET	/api/Users/{id}	▼
PUT	/api/Users/{id}	▼
DELETE	/api/Users/{id}	▼

## API Documentation overview

### Auth

**POST /api/Auth/register:** Allows a user to register.

POST /api/Auth/register

Parameters Try it out Reset

No parameters

Request body application/json

Example Value | Schema

```
{
  "username": "ggg",
  "password": "string",
  "email": "string",
  "role": "string"
}
```

Responses

Curl

```
curl -X 'POST' \
  'https://localhost:5062/api/Auth/register' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "username": "ggg",
    "password": "string",
    "email": "string",
    "role": "string"
  }'
```

Request URL

https://localhost:5062/api/Auth/register

Server response

Code	Details
200	<p>Response body</p> <p>User registered successfully.</p> <p>Response headers</p> <p>content-type: text/plain; charset=utf-8 date: Sun, 05 Jan 2025 21:35:40 GMT server: Mestrel</p>

Responses

Code	Description	Links
200	Success	No links

**POST /api/Auth/login:** Authenticates a user and provides a token.

POST /api/Auth/login

Parameters Try it out

No parameters

Request body application/json

Example Value | Schema

```
{
  "username": "string",
  "password": "string"
}
```

Responses

Code	Description	Links
200	Success	No links

## Categories

**GET /api/Categories:** Retrieves a list of available categories.

### Categories

GET /api/Categories

Parameters

No parameters

Responses

Curl

```
curl -X 'GET' \
  'https://localhost:5062/api/Categories' \
  -H 'accept: text/plain'
```

Request URL

https://localhost:5062/api/Categories

Server response

Code Details

200

Response body

```
{
  "category": "Sustainability and ESG (Environmental, Social, Governance)",
  "id": 0,
  "category": "Installation and Deployment",
  "id": 1,
  "category": "Operational Efficiency",
  "id": 2,
  "category": "Cost Optimization",
  "id": 3,
  "category": "Safety and Risk Management",
  "id": 4,
  "category": "Regulatory Compliance",
  "id": 5,
  "category": "Customer-Focused Initiatives",
  "id": 6,
  "category": "Digitalization and Data Management"
}
```

Response headers

```
content-type: application/json; charset=utf-8
date: Sun, 05 Jan 2025 21:46:02 GMT
server: Kestrel
```

Responses

Code	Description	Links
200	Success	No links

## InitiativePost

**GET /api/InitiativePosts:** Retrieves all initiative posts.

### InitiativePost

GET /api/InitiativePosts

Parameters

No parameters

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept header.

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "content": "string",
  "authorId": 0,
  "datePublished": "2025-01-05T22:24:12.749Z",
  "categoryId": 0,
  "category": {
    "id": 0,
    "category": "string"
  }
}
```

**POST /api/InitiativePosts:** Creates a new initiative post.

POST /api/InitiativePosts

Parameters

No parameters

Request body

application/json

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "content": "string",
  "authorId": 0,
  "datePublished": "2025-01-05T22:24:39.448Z",
  "categoryId": 0,
  "category": {
    "id": 0,
    "category": "string"
  }
}
```

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept header.

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "content": "string",
  "authorId": 0,
  "datePublished": "2025-01-05T22:24:39.448Z",
  "categoryId": 0,
  "category": {
    "id": 0,
    "category": "string"
  }
}
```

**GET /api/InitiativePosts/{id}**: Retrieves a specific initiative post by its ID.

GET /api/InitiativePosts/{id}

Parameters

Cancel

Name	Description
id * required integer(int32) (path)	44

Execute Clear

Responses

Curl

```
curl -X GET \
  'https://localhost:5062/api/InitiativePosts/44' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:5062/api/InitiativePosts/44
```

Server response

Code	Details
404 Undocumented	Error: response status is 404  Response body <pre>{   "type": "https://tools.ietf.org/html/rfc7159#section-15.5.1",   "title": "Not Found",   "status": 404,   "errorCode": "00-0983defc6d0586a2fedc225959efc33-a56a053383fa390-00" }</pre> Response headers <pre>content-type: application/problem+json; charset=utf-8 date: Sun, 05 Jan 2025 22:25:30 GMT server: Kestrel</pre>

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept header.

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "content": "string",
  "authorId": 0,
  "datePublished": "2025-01-05T22:25:30.429Z",
  "categoryId": 0,
  "category": {
    "id": 0,
    "category": "string"
  }
}
```

**DELETE /api/InitiativePosts/{id}**: Deletes a specific initiative post by its ID.

DELETE

/api/InitiativePosts/{id}

Try it out

Parameters

Name	Description
id <small>required</small>	
integer(\$int32)	id
(path)	

Responses

Code	Description	Links
200	Success	No links

**PUT /api/InitiativePosts/{id}**: Updates a specific initiative post by its ID.

PUT

/api/InitiativePosts/{id}

Try it out

Parameters

Name	Description
id <small>required</small>	
integer(\$int32)	id
(path)	

Request body

application/json

Example Value

Schema

```
{
  "id": 0,
  "title": "string",
  "content": "string",
  "authorId": 0,
  "datePublished": "2025-01-05T22:27:49.561Z",
  "categoryId": 0,
  "category": {
    "id": 0,
    "category": "string"
  }
}
```

Responses

Code	Description	Links
200	Success	No links

**GET /api/InitiativePosts/user**: Retrieves posts associated with a specific user.

GET

/api/InitiativePosts/user

Cancel

Parameters

No parameters

Execute

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept header:

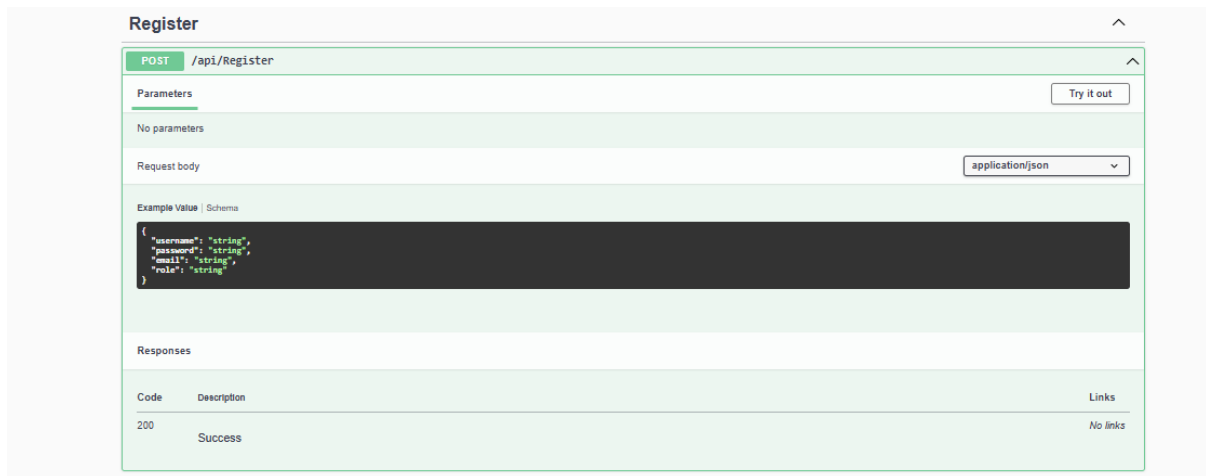
Example Value

Schema

```
{
  "id": 0,
  "title": "string",
  "content": "string",
  "authorId": 0,
  "datePublished": "2025-01-05T22:28:27.021Z",
  "categoryId": 0,
  "category": {
    "id": 0,
    "category": "string"
  }
}
```

## Register

**POST /api/Register:** An additional endpoint for user registration



The image shows the Swagger UI for the **Register** endpoint. The endpoint is **POST /api/Register**. It has no parameters. The request body is set to **application/json**. An example value is shown in a dark box: 

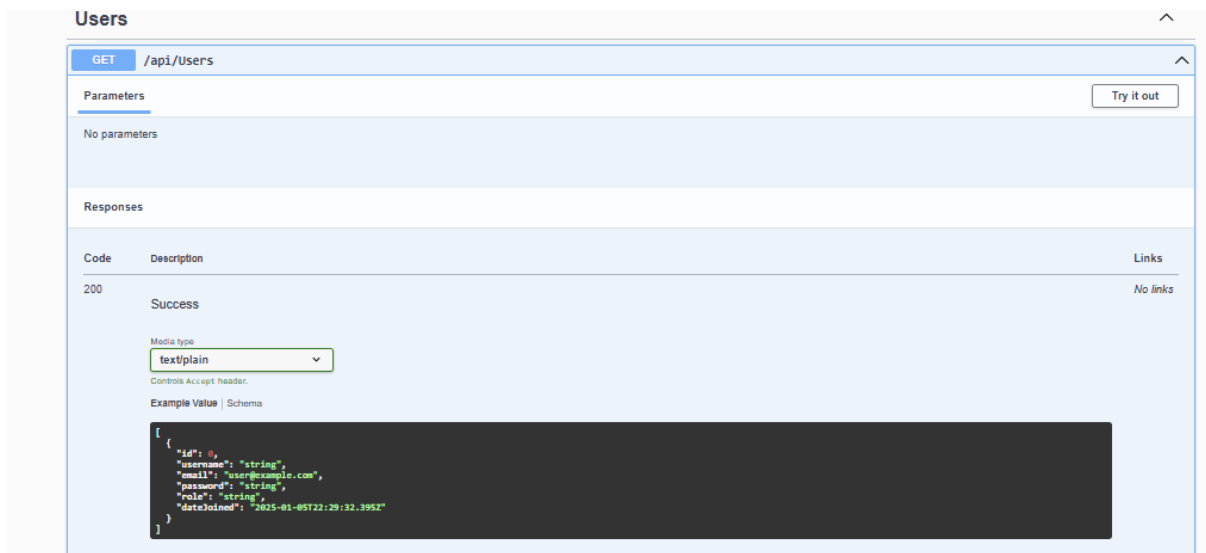
```
{  "username": "string",  "password": "string",  "email": "string",  "role": "string"}
```

. The response table shows a **200** status code with the description **Success** and no links.

Code	Description	Links
200	Success	No links

## Users

**GET /api/Users:** Retrieves a list of all users.



The image shows the Swagger UI for the **Users** endpoint. The endpoint is **GET /api/Users**. It has no parameters. The response table shows a **200** status code with the description **Success** and no links. Below the response table, there is a media type dropdown set to **text/plain** and an example value shown in a dark box: 

```
{  "id": 0,  "username": "string",  "email": "user@example.com",  "password": "string",  "role": "string",  "dateJoined": "2025-01-05T22:29:32.395Z"}
```

Code	Description	Links
200	Success	No links

**POST /api/Users:** Creates a new user.

POST

/api/Users

Try it out

Parameters

No parameters

Request body

application/json

Example Value | Schema

```
{  "id": 0,  "username": "string",  "email": "user@example.com",  "password": "string",  "role": "string",  "dateJoined": "2025-01-05T22:29:58.779Z"}
```

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept header:

Example Value | Schema

```
{  "id": 0,  "username": "string",  "email": "user@example.com",  "password": "string",  "role": "string",  "dateJoined": "2025-01-05T22:29:58.781Z"}
```

**GET /api/Users/{id}**: Retrieves a specific user by ID.

GET

/api/Users/{id}

Try it out

Parameters

Name	Description
id * required	id

integer(int32)  
(path)

Responses

Code	Description	Links
200	Success	No links

Media type

text/plain

Controls Accept header:

Example Value | Schema

```
{  "id": 0,  "username": "string",  "email": "user@example.com",  "password": "string",  "role": "string",  "dateJoined": "2025-01-05T22:38:52.551Z"}
```

**PUT /api/Users/{id}**: Updates a specific user's details.



PUT /api/Users/{id}

Try it out

Parameters

Name	Description
id * required integer(int32) (path)	<input type="text" value="id"/>

Request body

application/json

Example Value | Schema

```
{
  "id": 0,
  "username": "string",
  "email": "user@example.com",
  "password": "string",
  "role": "string",
  "dateJoined": "2025-01-05T22:31:12.069Z"
}
```

Responses

Code	Description	Links
200	Success	No links

**DELETE /api/Users/{id}**: Deletes a specific user by ID.

DELETE /api/Users/{id}

Try it out

Parameters

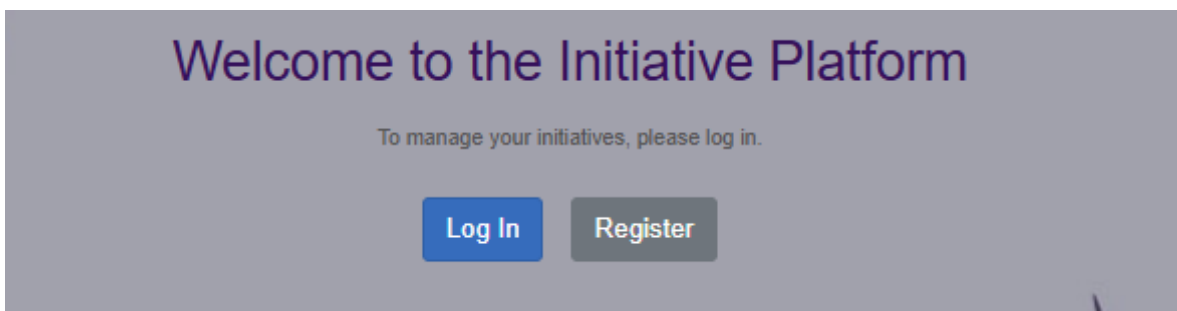
Name	Description
id * required integer(int32) (path)	<input type="text" value="id"/>

Responses

Code	Description	Links
200	Success	No links

## #3 Web Application:

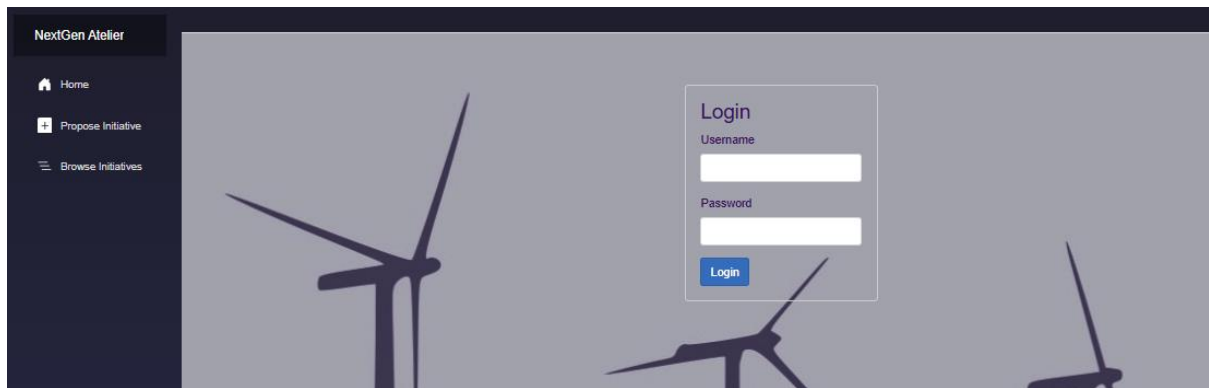
### KEY REQUIREMENTS



The core focus of the project is built upon the following requirements:

User registration and login

//the endpoint of user authentication



The POST `/api/Auth/Login` action, marked with `[HttpPost("login")]` and `[AllowAnonymous]`, allows unauthenticated access to this endpoint. It accepts a `LoginRequest` object containing the username and password from the request body. The method retrieves the user from the database using `_context.Users.SingleOrDefaultAsync` by matching the provided username. The password verification is performed using `BCrypt.Net.BCrypt.Verify`, which compares the hashed password stored in the database with the one provided by the user. If the username is not found or the password does not match, the method returns an `Unauthorized` response with a generic "Invalid credentials" message, which is a good practice to avoid revealing unnecessary information.

```
// POST: api/Auth/Login
[HttpPost("login")]
[AllowAnonymous]
0 references
public async Task<IActionResult> Login([FromBody] LoginRequest login)
{
    var user = await _context.Users.SingleOrDefaultAsync(u => u.Username == login.Username);

    if (user == null || !BCrypt.Net.BCrypt.Verify(login.Password, user.Password))
    {
        return Unauthorized("Invalid credentials.");
    }

    var token = GenerateToken(user);
    return Ok(new { token });
}
```

When authentication is successful, a token is generated using the `GenerateToken` method and returned in the response. The code demonstrates good use of password hashing with `BCrypt`, but improvements could be made by adding rate limiting to mitigate brute force attacks and logging failed login attempts to monitor suspicious activity.

```

<div class="login-container">
  <h3>Login</h3>
  <EditForm Model="@loginRequest" OnValidSubmit="HandleLogin">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div class="mb-3">
      <label for="username" class="form-label">Username</label>
      <InputText id="username" class="form-control" @bind-Value="loginRequest.Username" />
    </div>

    <div class="mb-3">
      <label for="password" class="form-label">Password</label>
      <InputText id="password" class="form-control" @bind-Value="loginRequest.Password" type="password" />
    </div>

    <button type="submit" class="btn btn-primary">Login</button>
  </EditForm>
</div>
42 @code {
    5 references
    private LoginRequest loginRequest = new LoginRequest();
43
44
    2 references
    private async Task HandleLogin()
    {
45
46
47         try
48         {
49             var response = await Http.PostAsJsonAsync("https://localhost:5062/api/Auth/login", loginRequest);
50             if (response.IsSuccessStatusCode)
51             {
52                 var result = await response.Content.ReadFromJsonAsync<LoginResponse>();
53                 Console.WriteLine("Login successful!");
54                 Console.WriteLine($"JWT Token: {result?.Token}");
55
56                 // Store JWT token in local storage
57                 await JSRuntime.InvokeVoidAsync("localStorage.setItem", "authToken", result?.Token);
58
59                 // Redirect to main page
60                 Navigation.NavigateTo("/");
61             }
62             else
63             {
64                 Console.WriteLine("Login failed.");
65             }
66         }
67         catch (Exception ex)
68         {
69             Console.WriteLine($"Error during login: {ex.Message}");
70         }
71     }
72
73     private class LoginResponse
74     {
75         public string? Token { get; set; }
76     }
77 }
78

```

This Blazor component implements a login page that interacts with a backend API endpoint (/api/Auth/login) to authenticate users. The EditForm binds to a LoginRequest model, capturing the Username and Password input, with validation provided by DataAnnotationsValidator and ValidationSummary. Upon submission, the HandleLogin method sends the login credentials to the backend using Http.PostAsJsonAsync. If the response is successful, it retrieves a JWT token from the server's response, stores it in the browser's local storage via IJSRuntime, and redirects the user to the main page using NavigationManager. The component also includes error handling for login failures or exceptions, logging the results to the console for debugging.

## //endpoint for registration

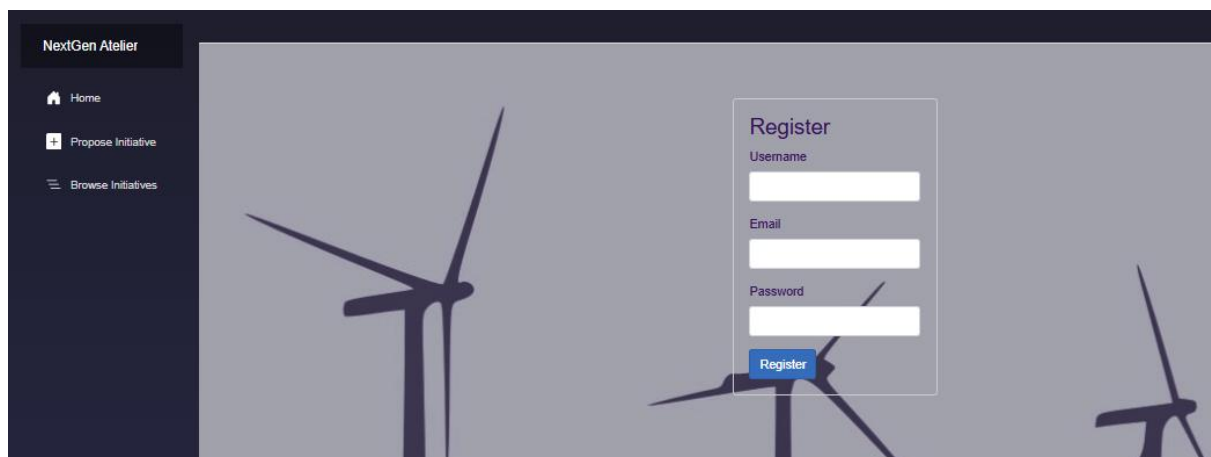
```
// POST: api/Auth/Register
[HttpPost("register")]
[AllowAnonymous]
0 references
public async Task<IActionResult> Register([FromBody] RegisterRequest request)
{
    // Validate the request
    if (string.IsNullOrEmpty(request.Username) || string.IsNullOrEmpty(request.Password)
        || string.IsNullOrEmpty(request.Email))
    {
        return BadRequest("Username and Password are required.");
    }

    // Check if the username already exists
    if (await _context.Users.AnyAsync(u => u.Username == request.Username))
    {
        return Conflict("Username already exists.");
    }

    // Create the new user
    var hashedPassword = BCrypt.Net.BCrypt.HashPassword(request.Password);
    var user = new AtelierShared.Models.User
    {
        Username = request.Username,
        Password = BCrypt.Net.BCrypt.HashPassword(request.Password),
        Email = request.Email,
        Role = "User", // Default role
        DateJoined = DateTime.Now
    };

    _context.Users.Add(user);
    await _context.SaveChangesAsync();
}
```

The POST /api/Auth/Register action, decorated with [HttpPost("register")] and [AllowAnonymous], allows unauthenticated users to register a new account. The Register method accepts a RegisterRequest object from the request body, containing details such as Username, Password, and Email.



The method starts by validating the input to ensure that the Username, Password, and Email fields are not null or whitespace. If validation fails, it returns a BadRequest response with an appropriate error message. Next, it checks if the username already exists in the database using \_context.Users.AnyAsync. If the username is found, the method returns a Conflict response, indicating that the username is already taken.

If the input is valid and the username is available, the code hashes the provided password using BCrypt.Net.BCrypt.HashPassword to ensure secure storage of passwords. It then creates a new User object, populating fields such as Username, Password, Email, Role (defaulting to "User"), and DateJoined (set to the current date and time). The new user is added to the database context, and SaveChangesAsync is called to persist the changes to the database.

This Blazor component implements a user registration page that integrates with a backend API endpoint (/api/Auth/Register). The EditForm binds to a RegisterRequest model, capturing user input for Username, Email, and Password, with validation provided by DataAnnotationsValidator and ValidationSummary. Upon form submission, the HandleRegister method sends an HTTP POST request to the backend with the registration data using Http.PostAsJsonAsync. If the registration is successful (response.IsSuccessStatusCode), the user is redirected to the login page (/Auth/login) via NavigationManager. Otherwise, errors are logged to the console. The component also includes error handling for exceptions during the registration process and features a responsive, styled form to enhance user experience.

```

17 <div class="register-container">
18     <h3>Register</h3>
19     <EditForm Model="@registerRequest" OnValidSubmit="HandleRegister">
20         <DataAnnotationsValidator />
21         <ValidationSummary />
22
23         <div class="mb-3">
24             <label for="username" class="form-label">Username</label>
25             <InputText id="username" class="form-control" @bind-Value="registerRequest.Username" />
26         </div>
27
28         <div class="mb-3">
29             <label for="email" class="form-label">Email</label>
30             <InputText id="email" class="form-control" @bind-Value="registerRequest.Email" />
31         </div>
32
33         <div class="mb-3">
34             <label for="password" class="form-label">Password</label>
35             <InputText id="password" class="form-control" @bind-Value="registerRequest.Password" type="password" />
36         </div>
37
38         <button type="submit" class="btn btn-primary">Register</button>
39     </EditForm>
40 </div>
@code {
    6 references
    private RegisterRequest registerRequest = new RegisterRequest();

    2 references
    private async Task HandleRegister()
    {
        try
        {
            var response = await Http.PostAsJsonAsync("https://localhost:5062/api/Auth/Register", registerRequest);
            if (response.IsSuccessStatusCode)
            {
                Console.WriteLine("User registered successfully!");
                Navigation.NavigateTo("/Auth/login");
            }
            else
            {
                Console.WriteLine($"Failed to register user: {response.StatusCode}");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error during registration: {ex.Message}");
        }
    }
}

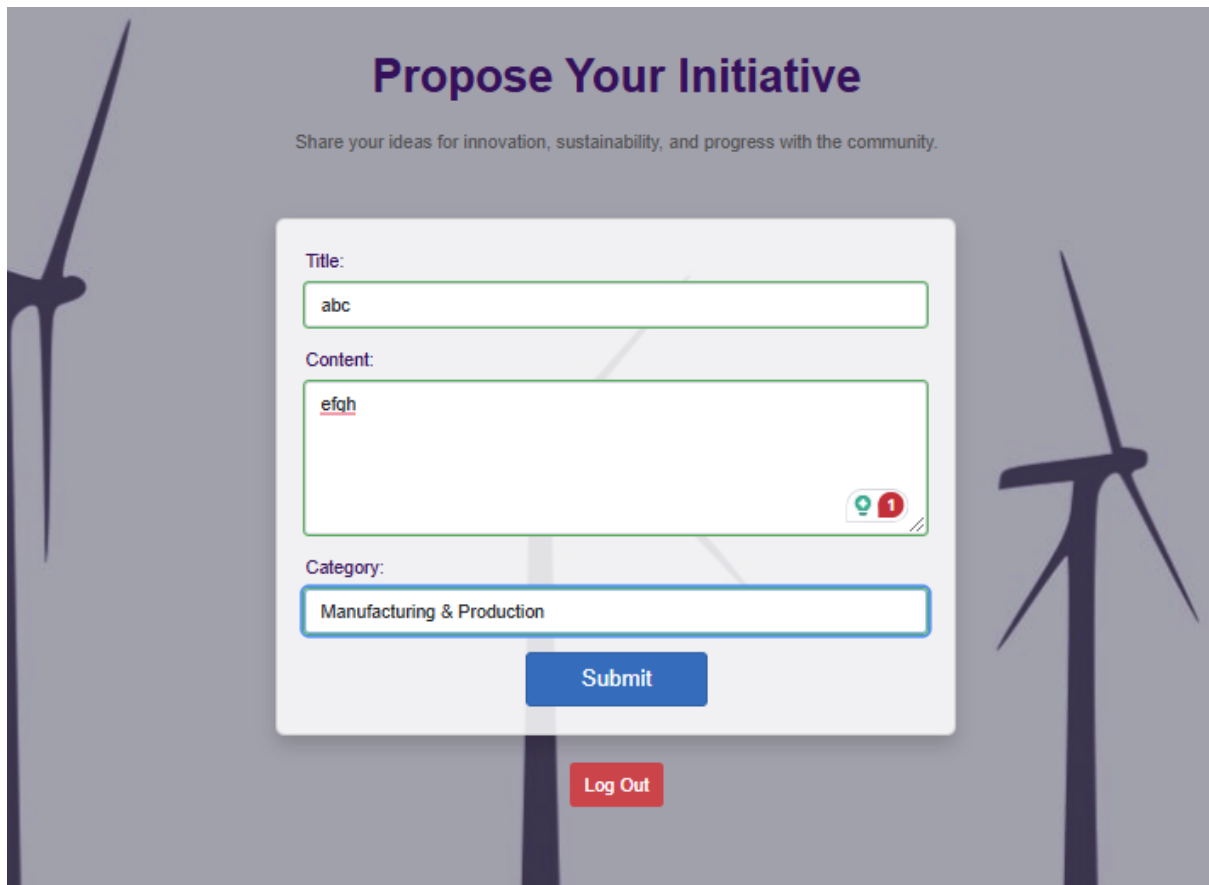
```

**class System.String**  
Represents text as a sequence of UTF-16 code units.

The ability to submit, view, edit, and delete product development initiatives

## TO SUMBIT A NEW INITIATIVE

The user interface, where authenticated users can fill out a form with fields for the initiative's title, content, and category. The form uses validation to ensure all required fields are completed before submission.



The image shows a web form titled "Propose Your Initiative" with the subtitle "Share your ideas for innovation, sustainability, and progress with the community." The form is set against a background of wind turbines. It contains three input fields: "Title:" with the value "abc", "Content:" with the value "efgh", and "Category:" with the value "Manufacturing & Production". A blue "Submit" button is located below the form, and a red "Log Out" button is positioned below the "Submit" button. A small notification icon with a red circle and the number "1" is visible in the bottom right corner of the content field.

On the frontend, the SubmitInitiative method processes the user input, decodes the JWT token to retrieve the user's ID, and sends a POST request with the initiative data to the API.

```

div class="container my-5">
    @if (!isLoggedIn)
    {
        <div class="home-content">
            <h1>You need to log in or create an account to write an initiative.</h1>
            <button class="btn btn-primary m-2" @onclick="NavigateToLogin">Login</button>
            <button class="btn btn-secondary m-2" @onclick="NavigateToRegister">Register</button>
        </div>
    }
    else
    {
        <div class="home-content">
            <h1><b>Propose Your Initiative</b></h1>
            <p>Share your ideas for innovation, sustainability, and progress with the community.</p>
        </div>

        <EditForm Model="@initiativePost" OnValidSubmit="SubmitInitiative" class="card shadow p-4">
            <DataAnnotationsValidator />
            <ValidationSummary class="text-danger mb-3" />

            <div class="mb-3">
                <label for="title" class="form-label">Title:</label>
                <input type="text" id="title" @bind="initiativePost.Title" class="form-control" placeholder="Enter the title of your initiative" />
            </div>

            <div class="mb-3">
                <label for="content" class="form-label">Content:</label>
                <input type="text" id="content" @bind="initiativePost.Content" class="form-control" rows="5" placeholder="Describe your initiative here">
            </div>

            <div class="mb-3">
                <label for="category" class="form-label">Category:</label>
                <input type="text" id="category" @bind="initiativePost.CategoryId" class="form-control">
                <option value="0">-- Select a Category --</option>
                @foreach (var category in Categories)
                {
                    <option value="@category.Id">@category.Category</option>
                }
            </div>

            <div class="text-center">
                <button type="submit" class="btn btn-primary btn-lg px-5">Submit</button>
            </div>
        </EditForm>
    }

```

2 references

```
private async Task SubmitInitiative()
```

```

try
{
    var token = await AuthService.GetTokenAsync();

    if (string.IsNullOrEmpty(token))
    {
        Console.WriteLine("No token found. Redirecting to login.");
        Navigation.NavigateTo("/Auth/login");
        return;
    }

    // Decode the token to extract the UserId
    var handler = new System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler();
    var jwtToken = handler.ReadJwtToken(token);
    var userIdClaim = jwtToken.Claims.FirstOrDefault(c => c.Type == JwtRegisteredClaimNames.Sub);

    if (userIdClaim == null)
    {
        Console.WriteLine("User ID claim not found in token.");
        return;
    }

    var userId = int.Parse(userIdClaim.Value);
    initiativePost.AuthorId = userId;

    // Prepare the payload
    var payload = new
    {
        Title = initiativePost.Title,
        Content = initiativePost.Content,
        AuthorId = initiativePost.AuthorId, //backend ignores it
        CategoryId = initiativePost.CategoryId,
        Category = new
        {
            Id = initiativePost.CategoryId, // Include the Category object with Id
            Name = Categories.FirstOrDefault(c => c.Id == initiativePost.CategoryId)?.Category // Include the name
        }
    };
}

```

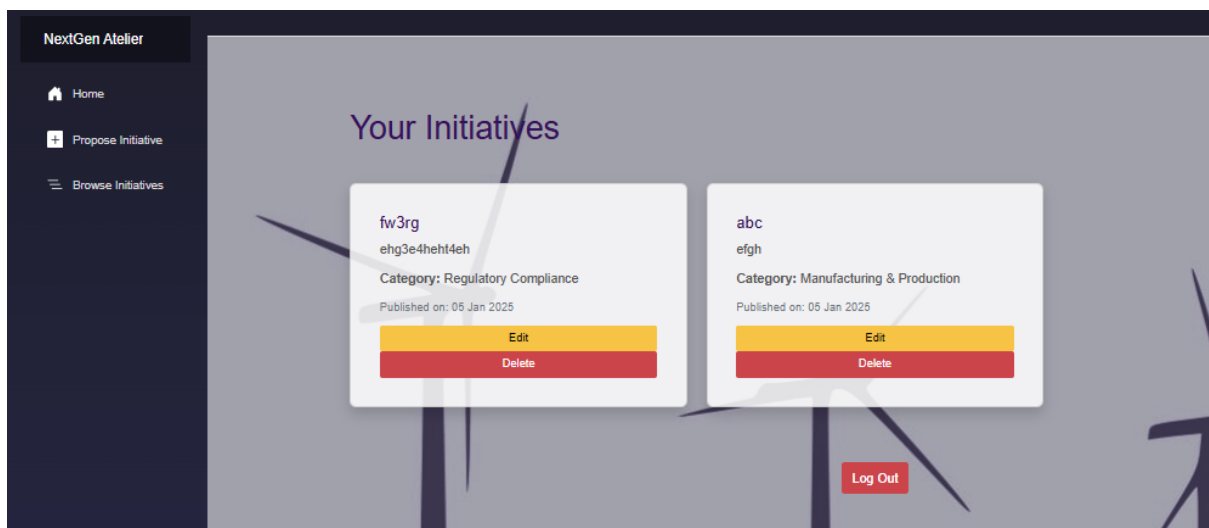
On the backend, the `PostInitiativePost` method in the controller validates the token, extracts the user ID, and saves the initiative to the database.

```
// POST: api/InitiativePost
[HttpPost]
[Authorize]
0 references
public async Task<ActionResult<InitiativePost>> PostInitiativePost(InitiativePost initiativePostRequest)
{
    // Extract User ID from the token using the correct claim type
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);

    if (string.IsNullOrEmpty(userId))
    {
        return Unauthorized(new
        {
            message = "User ID not found in token.",
            claims = User.Claims.Select(c => new { c.Type, c.Value }).ToList()
        });
    }

    Console.WriteLine($"Extracted User ID: {userId}");

    if (initiativePostRequest.Category == null || initiativePostRequest.Category.Id <= 0)
    {
        throw new ArgumentException("Invalid or missing Category.");
    }
}
```



## TO EDIT INITIATIVES

The first screenshot illustrates the user interface, which displays a list of initiatives submitted by the user, each accompanied by "Edit" and "Delete" options. Selecting the "Edit" button navigates the user to a dedicated form pre-populated with the initiative's existing details, allowing for updates to the title, content, and category. This interface ensures an intuitive and user-friendly experience for managing initiatives.

The second screenshot provides a detailed view of the code implementation for editing initiatives. The frontend logic includes the `LoadInitiativeToEdit` method, which sends a GET request to retrieve the specific initiative by its ID and pre-fills the form with its data. Upon submission, the updated data is sent to the backend via an HTTP PUT request.



The backend logic is implemented in the `UpdateInitiative` method, which enforces security by verifying that the user is authenticated and authorized to edit the initiative. The method ensures that the `AuthorId` of the initiative matches the user ID extracted from the JWT token, restricting edits to the initiative's creator. Once verified, the database is updated with the new information, and appropriate responses are returned to the client in case of success or errors.

This seamless integration between the frontend and backend ensures secure, efficient, and user-friendly management of initiatives, meeting high professional standards for functionality and security. Let me know if you'd like further refinement or additional details.

```
2 references
private async Task LoadInitiativeToEdit()
{
    try
    {
        var token = await AuthService.GetTokenAsync();
        var request = new HttpRequestMessage(HttpMethod.Get, $"https://localhost:5062/api/InitiativePosts/{Id}");
        request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);

        var response = await Http.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            initiativePost = await response.Content.ReadFromJsonAsync<InitiativePost>();
        }
        else
        {
            Console.WriteLine($"Failed to load initiative: {response.StatusCode}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error loading initiative: {ex.Message}");
    }
}
```

```

[HttpPut("{id}")]
[Authorize]
0 references
public async Task<IActionResult> UpdateInitiative(int id, [FromBody] InitiativePost updatedInitiative)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var initiative = await _context.InitiativePosts.FindAsync(id);

    if (initiative == null)
    {
        return NotFound();
    }

    if (initiative.AuthorId != int.Parse(userId))
    {
        return Forbid("You can only edit your own initiatives.");
    }

    initiative.Title = updatedInitiative.Title;
    initiative.Content = updatedInitiative.Content;
    initiative.CategoryId = updatedInitiative.CategoryId;

    try
    {
        await _context.SaveChangesAsync();
        return NoContent();
    }
    catch (DbUpdateException ex)
    {
        Console.WriteLine($"Database update error: {ex.Message}");
        return StatusCode(500, "Failed to update initiative.");
    }
}

```

## TO DELETE INITIATIVE

The first screenshot demonstrates the frontend logic for deleting an initiative. The `DeleteInitiative` method sends an HTTP DELETE request to the backend API, including the initiative's ID and the user's authentication token for security. Upon a successful response, the initiative is removed from the user's list of initiatives on the client side. If the deletion fails, an appropriate error message is logged for debugging purposes. This ensures a smooth and responsive user experience while maintaining security and error handling.

The second screenshot highlights the backend logic implemented in the `DeleteInitiative` method of the API controller. This method begins by extracting the user's ID from the JWT token and retrieving the initiative by its ID from the database. The method ensures that only the creator of the initiative can delete it by verifying that the `AuthorId` matches the user ID from the token. If the validation passes, the initiative is removed from the database, and a `NoContent` response is returned to indicate successful deletion. Proper error handling is implemented to manage cases where the initiative is not found or the user is unauthorized.

```

255 | < references
256 | private async Task DeleteInitiative(int id)
257 | {
258 |     try
259 |     {
260 |         var token = await AuthService.GetTokenAsync();
261 |         var request = new HttpRequestMessage(HttpMethod.Delete, $"https://localhost:5062/api/InitiativePosts/{id}");
262 |         request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);
263 |
264 |         var response = await Http.SendAsync(request);
265 |
266 |         if (response.IsSuccessStatusCode)
267 |         {
268 |             UserInitiatives = UserInitiatives.Where(i => i.Id != id).ToList();
269 |         }
270 |         else
271 |         {
272 |             Console.WriteLine($"Failed to delete initiative: {response.StatusCode}");
273 |         }
274 |     }
275 |     catch (Exception ex)
276 |     {
277 |         Console.WriteLine($"Error deleting initiative: {ex.Message}");
278 |     }
279 | }
280 |
[HttpDelete("{id}")]
[Authorize]
0 references
public async Task<IActionResult> DeleteInitiative(int id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    var initiative = await _context.InitiativePosts.FindAsync(id);

    if (initiative == null)
    {
        return NotFound();
    }

    if (initiative.AuthorId != int.Parse(userId))
    {
        return Forbid("You can only delete your own initiatives.");
    }

    _context.InitiativePosts.Remove(initiative);
    await _context.SaveChangesAsync();

    return NoContent();
}

```

## Communication between the Blazor frontend and the ASP.NET Core Web API

The communication between the Blazor frontend and the ASP.NET Core Web API in this project is implemented through a robust and efficient integration using HTTP client services. The Blazor frontend leverages the built-in HttpClient service to send HTTP requests and handle responses from the Web API endpoints. This setup ensures seamless data exchange between the client and server, enabling functionalities such as submitting, editing, and deleting initiatives.

The HttpClient service is utilized throughout the Blazor components to interact with the Web API. For example, when a user submits a new initiative, the SubmitInitiative method constructs an

HTTP POST request to the endpoint `/api/InitiativePosts`. This request includes the initiative's data in the body and the user's authentication token in the headers for secure communication. Similarly, other components such as `LoadInitiativeToEdit` and `DeleteInitiative` use GET, PUT, and DELETE requests to interact with the corresponding API endpoints, ensuring full CRUD functionality.

Consistency in data exchange is maintained through shared data models, commonly known as Data Transfer Objects (DTOs). Models like `InitiativePost` and `CategoryModel` are defined in a shared library and used across both the frontend and backend. This shared approach ensures that the structure of data remains uniform, simplifying validation and data binding. For example, the `InitiativePost` model, which includes properties such as `Title`, `Content`, `CategoryId`, and `AuthorId`, is used in Blazor forms for user input and in the API for processing and storing data in the database.

To enhance responsiveness, the Blazor components rely heavily on asynchronous operations when interacting with the API. Methods such as `LoadCategoriesAsync` and `SubmitInitiative` ensure non-blocking calls, allowing the user interface to remain responsive while awaiting data from the server. For instance, the `LoadCategoriesAsync` method asynchronously fetches a list of categories from the API, ensuring a smooth user experience even when the data retrieval takes time.

On the backend, the ASP.NET Core Web API controllers handle the incoming requests and perform operations on the database. Endpoints like `PostInitiativePost`, `UpdateInitiative`, and `DeleteInitiative` are secured using JWT-based authentication, ensuring that only authenticated users can perform these actions. The frontend includes the user's JWT token in the Authorization header of each HTTP request, and the backend validates the token to identify the user and authorize their actions.

In conclusion, the communication between the Blazor frontend and the ASP.NET Core Web API is built on a foundation of secure HTTP services, shared data models, and asynchronous operations. This integration ensures a consistent and responsive user experience while maintaining robust security and reliable data handling.

## AN OVERVIEW OF THE PAGES IN YOUR WEB APPLICATION.

### ACCOUNT.RAZOR

The **Account Page** serves as a centralized dashboard for users to manage their initiatives on the platform. It dynamically adapts based on the user's authentication status. If the user is not logged in, the page displays a welcoming prompt with options to log in or register, ensuring a smooth onboarding experience. For authenticated users, the page provides a list of their submitted initiatives, each displayed in an organized card layout showing key details such as the title, content, category, and publication date. The page offers intuitive options to edit or delete each initiative, with inline editing forms and secure communication with the backend API for updates and deletions. Additionally, users can log out directly from the page. Behind the scenes, the page leverages asynchronous API calls to fetch user-specific initiatives and categories, ensuring a responsive and seamless user experience. This page exemplifies user-centric design, enabling efficient initiative management in a secure and intuitive environment.

## BROWSE.RAZOR

This page is designed to display a collection of initiatives in a visually organized card layout, where each card presents key details such as the initiative's title, content, category, and publication date.

The layout dynamically adapts based on the availability of data. If no initiatives are present, a user-friendly message informs users that there are no available initiatives. While the initiatives are being loaded from the backend, a loading message ensures users understand the system is actively fetching data.

The page employs asynchronous API calls to retrieve the list of initiatives from the backend using the `LoadInitiativesAsync` method. This ensures that the interface remains responsive and interactive while data is being fetched. The initiatives are fetched via a GET request to the API endpoint `/api/InitiativePosts`, leveraging Blazor's built-in `HttpClient` and `GetFromJsonAsync` method for efficient data handling.

The clean and structured design, combined with responsive functionality, makes this page an intuitive and accessible platform for users to explore all published initiatives. It effectively balances functionality and user experience, ensuring seamless browsing of the platform's content.

## EDITINITIATIVE.RAZOR

The Edit Initiative Page provides a secure and intuitive interface for users to update their submitted initiatives. This page ensures that only authenticated and authorized users can access and modify their initiatives, safeguarding the integrity of the platform. Upon loading, the page checks the user's authentication status using the `AuthService.IsUserLoggedIn` method. If the user is not logged in, they are redirected to the `/account` page, where they can log in or register. This authentication layer prevents unauthorized access and ensures a personalized editing experience.

Once authenticated, the page dynamically loads the initiative details and available categories. The initiative is retrieved using the unique ID passed as a route parameter, and the data is fetched securely with a GET request to the API endpoint `/api/InitiativePosts/{Id}`. The `LoadInitiativeToEdit` method pre-populates the form with the initiative's existing title, content, and category, allowing users to make changes easily. Additionally, the `LoadCategoriesAsync` method fetches the list of categories, ensuring the dropdown list is always up-to-date.

The initiative editing form is designed with user experience in mind. It uses Blazor's `EditForm` component with `DataAnnotationsValidator` to enforce validation for required fields. This ensures that users cannot submit incomplete or invalid data. The form includes input fields for the title, content, and category, with a clear layout and descriptive labels to guide users through the editing process.

When the user submits the changes, the `SubmitEditedInitiative` method sends a PUT request to the API endpoint, including the updated initiative data and the user's authentication token. This secure communication guarantees that only the initiative owner can update the data. Upon successful submission, the user is redirected to the `/account` page, where they can review their updated initiatives.

The page also includes a logout button, allowing users to securely log out and navigate back to the login page. This ensures a seamless and secure experience for managing session-based access. Overall, the Edit Initiative Page is designed to provide a user-friendly and secure environment for updating initiatives while maintaining the platform's data integrity and responsiveness.

## HOME.RAZOR

The Home Page introduces the NextGen Atelier Initiatives Platform as a hub for innovation and collaboration in offshore wind product development. It highlights the platform's mission to empower teams with creativity, strategy, and cutting-edge technology to drive sustainable solutions. Through inspiring messaging, it invites users to join a movement that shapes the future of offshore wind innovation.

## LOGIN.RAZOR

The Login Page provides a simple and secure interface for users to authenticate and gain access to the platform. It features a clean design with a login form that includes fields for the username and password. The form is validated using Blazor's `EditForm` component with `DataAnnotationsValidator`, ensuring only valid credentials are submitted.

Upon submission, the `HandleLogin` method sends the login credentials to the backend API (`/api/Auth/login`) via an HTTP POST request. If the login is successful, the returned JWT token is stored in the browser's local storage using the `IJSRuntime` service, enabling secure and persistent authentication. The user is then redirected to the home page (`/`) to access the platform's features.

This page is an essential entry point for authenticated access, combining simplicity, security, and usability to streamline the login process.

## REGISTER.RAZOR

The **Register Page** provides a straightforward and user-friendly interface for new users to create an account on the platform. The page features a registration form that includes fields for the username, email, and password. Input validation is handled by Blazor's

EditForm component with DataAnnotationsValidator, ensuring that the data entered by users meets the required criteria.

When the form is submitted, the HandleRegister method sends the registration details to the backend API (/api/Auth/Register) using an HTTP POST request. If the registration is successful, a confirmation message is logged, and the user is redirected to the login page (/Auth/login), allowing them to log in with their newly created credentials.

This page ensures a seamless onboarding experience for new users, combining validation, security, and usability to streamline the registration process.

## WRITEINITIATIVE.RAZOR

The **Write Initiative Page** allows authenticated users to share their ideas for innovation and sustainability through a user-friendly and secure interface. Upon loading, the page checks the user's authentication status, redirecting unauthenticated users to the login or registration page to ensure only authorized users can access this feature. It dynamically fetches a list of categories from the backend API, allowing users to classify their initiatives accurately. The form, designed with validation mechanisms, includes fields for the title, content, and category, ensuring the information submitted is complete and valid. When the form is submitted, the page securely processes the initiative by decoding the user's authentication token to retrieve their ID and sends the data to the backend via an HTTP POST request. Successful submissions redirect users to the initiatives page, while robust error handling ensures a smooth experience even in case of issues. Additionally, a logout button provides a secure way to end the session, making the page a seamless and integral part of the platform's initiative-sharing workflow.

## HOW FRONTEND CONNECTS TO YOUR WEB SERVICE.

The frontend uses the HttpClient service to send HTTP requests to the backend API endpoints. These requests enable CRUD (Create, Read, Update, Delete) operations by passing data to and retrieving data from the backend. Authentication is handled using JWT tokens stored in the browser, which are included in request headers for secure communication. The HttpClient service is injected into Blazor components using the @inject directive.

# #4 User Management:

To distinguish between the registration processes for regular users and administrators, I have decided that all new users registering on the platform will be assigned the default role of "User" (*Atelier Shared/Model/User or Register Request screenshot*). This means

newly created accounts will have permissions limited to editing and deleting only their own initiatives.

## IN THE CLASS `ATELIERCONTEXT.CS`

5 references

```
public string Role { get; set; } = "User"; // Default role
```

Administrator accounts, on the other hand, are created directly in the backend and provided to designated administrators. These accounts are granted elevated privileges, allowing administrators to edit and delete any initiative, irrespective of the account that originally created it. The primary distinction between an administrator and a regular user is this broader access control. (*SQLite screenshot*)

```
modelBuilder.Entity<User>().HasData(  
    new User  
    {  
        Id = 1,  
        Username = "admin",  
        Password = "hashed_password",  
        Role = "Admin",  
        Email = "316333@viauc.dk",  
        DateJoined = DateTime.UtcNow  
    }  
);
```

## #5 Data Access:

The introduction of an Object-Relational Mapper (ORM), like Entity Framework Core (EF Core), has significantly changed how I handle data in my project. It simplifies the process by providing an abstraction over the database, allowing me to work with objects in code instead of writing raw SQL queries. This makes the development process more intuitive and aligns well with the object-oriented principles used in the rest of the application.

With the ORM, data access and CRUD (Create, Read, Update, Delete) operations have become much easier. Instead of writing complex SQL statements, I can use straightforward methods like `Add`, `Update`, or `Remove` to manipulate data. For example, adding a new initiative to the database is as simple as creating an object and calling `context.InitiativePosts.Add(newInitiative)` followed by `context.SaveChanges()`. This has reduced the amount of repetitive code and improved the overall efficiency of development.

Using LINQ with the ORM has also streamlined querying the database. Queries are now written in C#, making them easier to read and maintain. For instance, retrieving initiatives for a specific category is as simple as writing `context.InitiativePosts.Where(i => i.CategoryId == 3)`. This eliminates the need for manual SQL, making it faster and less error-prone.



The ORM also helps with database schema management. Through migrations, I can define changes to the database structure in code, and the ORM generates the necessary scripts to apply those changes. For example, when I add a new property to a model class, such as `DateCreated`, I can create a migration and update the database automatically. This ensures the code and database stay in sync.

Data seeding has been particularly useful for initializing the database with essential data during development and testing. For example, I've preloaded the database with an admin user, a sample initiative, and various categories. This ensures that the system is ready to use as soon as the database is created, saving time during setup.

Overall, using an ORM in my project has made data management more straightforward, secure, and maintainable. It allows me to focus on the application logic rather than worrying about the details of database operations, which has been a significant improvement to my workflow.

This project utilizes Entity Framework Core (EF Core) for efficient and robust data management. The integration is evident from the inclusion of key EF Core packages such as `Microsoft.EntityFrameworkCore`, `Microsoft.EntityFrameworkCore.Sqlite`, and `Microsoft.EntityFrameworkCore.Tools`, enabling object-relational mapping, SQLite database support, and migration tools. The database context (`AtelierContext`) defines `DbSet` properties for entities like `Users`, `InitiativePosts`, and `CategoryModel`, allowing seamless interaction with database tables using LINQ queries and strongly-typed objects. Additionally, EF Core facilitates database configuration, schema migrations, and data seeding, ensuring a scalable and maintainable architecture for the Atelier platform.

```

<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="BCrypt.Net-Next" Version="4.0.3" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="8.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="8.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="9.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="8.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="9.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.11" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="9.0.0" />
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    <PrivateAssets>all</PrivateAssets>
  </ItemGroup>
  <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
  <ProjectReference Include="..\AtelierShared\AtelierShared.csproj" />

</Project>

```

The AtelierContext class serves as the central database context for the Atelier project, leveraging Entity Framework Core (EF Core) to manage data interactions. By inheriting from DbContext, it provides a bridge between the application and the underlying SQLite database. The class defines DbSet properties for the key entities in the system, including Users, InitiativePosts, and CategoryModel. These DbSet properties represent tables in the database, allowing developers to query and manipulate data using LINQ and strongly-typed objects, which enhances maintainability and reduces errors.

```

17 references
public class AtelierContext : DbContext
{
    0 references
    public AtelierContext(DbContextOptions<AtelierContext> options)
        : base(options)

    // DbSet
    12 references
    public DbSet<User>? Users { get; set; }
    7 references
    public DbSet<InitiativePost>? InitiativePosts { get; set; }
    1 reference
    public DbSet<CategoryModel>? Category { get; set; }
    0 references
    public object Categories { get; internal set; }
}

```

The `OnConfiguring` method plays a critical role in configuring the database connection. It uses the SQLite provider to define how the application connects to the database, ensuring the connection string is correctly set. Additionally, the method suppresses warnings about pending model changes during debugging using the `ConfigureWarnings` method. This suppression helps streamline the development process by minimizing unnecessary interruptions during testing and debugging, improving the developer experience.

```
0 references
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    base.OnConfiguring(optionsBuilder);

    optionsBuilder.UseSqlite(
        optionsBuilder.Options.FindExtension<Microsoft.EntityFrameworkCore.Sqlite.Infrastructure.Internal.SqliteOptionsExtension>()?.ConnectionString
        ?? throw new InvalidOperationException("No connection string configured.")
    );

    // Suppress the pending model changes warning (use only during debugging)
    optionsBuilder.ConfigureWarnings(warnings =>
    {
        warnings.Ignore(RelationalEventId.PendingModelChangesWarning);
    });
}
```

The `OnModelCreating` method is where database schema customization and data seeding occur. It initializes the database with essential data, including an admin user, a sample initiative, and predefined categories. For instance, the admin user is seeded with details such as Username, Role, and Email, while the `InitiativePosts` table is seeded with a sample initiative linked to the admin user.

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // Seed data for Admin
    modelBuilder.Entity<User>().HasData(
        new User
        {
            Id = 1,
            Username = "admin",
            Password = "hashed_password",
            Role = "Admin",
            Email = "316333@viauc.dk",
            DateJoined = DateTime.UtcNow
        }
    );

    // Seed data for InitiativePost
    modelBuilder.Entity<InitiativePost>().HasData(
        new InitiativePost
        {
            Id = 1,
            Title = "Improve Blade Manufacturing Efficiency",
            Content = "Implement automated robotic systems for precision cutting and assembly of wind turbine blades. This will reduce production time by 20%",
            CategoryId = 3,
            AuthorId = 1 // Link to the seeded User
        }
    );

    // Seed Categories
    modelBuilder.Entity<CategoryModel>().HasData(
        new CategoryModel { Id = 1, Category = "Concept Development" },
        new CategoryModel { Id = 2, Category = "Design & Prototyping" },
        new CategoryModel { Id = 3, Category = "Manufacturing & Production" },
        new CategoryModel { Id = 4, Category = "Sustainability & Green Technology" },
        new CategoryModel { Id = 5, Category = "Sustainability and ESG (Environmental, Social, Governance)" },
        new CategoryModel { Id = 6, Category = "Installation and Deployment" },
        new CategoryModel { Id = 7, Category = "Operational Efficiency" },
        new CategoryModel { Id = 8, Category = "Cost Optimization" },
        new CategoryModel { Id = 9, Category = "Safety and Risk Management" },
        new CategoryModel { Id = 10, Category = "Regulatory Compliance" },
        new CategoryModel { Id = 11, Category = "Customer-Focused Initiatives" },
        new CategoryModel { Id = 12, Category = "Digitalization and Data Management" }
    );
}
```

The categories provide a predefined taxonomy for organizing initiatives, covering various aspects such as "Concept Development," "Sustainability & Green Technology," and "Cost Optimization." This seed data ensures that the database is populated with initial, meaningful data upon creation, simplifying setup and testing.

In summary, the `AtelierContext` class encapsulates all necessary configurations and initializations for the database in the Atelier project. It showcases the power of EF Core by abstracting complex database interactions and offering an object-oriented approach to managing relational data. This design ensures a maintainable, scalable, and developer-friendly architecture for the project.

The screenshot below shows the database schema generated by Entity Framework Core in the Atelier project. This schema was created based on the defined models (`User`, `InitiativePost`, and `CategoryModel`) and relationships configured in the `AtelierContext` class.

The database structure for the Atelier project consists of three main tables: `Users`, `InitiativePosts`, and `Category`, which are interconnected to support a well-structured and organized data model. The `Users` table stores information about all registered users, including their roles (e.g., Admin or User) and account details, enabling role-based access control. The `InitiativePosts` table captures user-generated content, such as initiatives, and links each initiative to its author (`AuthorId`) and category (`CategoryId`), ensuring clear ownership and categorization. The `Category` table provides a predefined list of categories for organizing initiatives, enhancing navigation and filtering capabilities. These relationships, enforced through foreign keys, create a normalized database that is efficient for querying and managing data. Seeded data across all tables, including admin accounts, example initiatives, and categories, ensures the system is functional and test-ready from the start.

```
Schema
CREATE TABLE "Category" ( "Id" INTEGER NOT NULL CONSTRAINT "PK_Category" PRIMARY KEY AUTOINCREMENT, "Category" TEXT NOT NULL )
"Id" INTEGER NOT NULL
"Category" TEXT NOT NULL
CREATE TABLE "InitiativePost" ( "Id" INTEGER NOT NULL CONSTRAINT "PK_InitiativePost" PRIMARY KEY AUTOINCREMENT, "AuthorId" INTEGER NOT NULL, "CategoryId" INTEGER NOT NULL, "Content" TEXT NOT NULL, "DatePublished" TEXT NOT NULL, "Title" TEXT NOT NULL, CONSTRAINT "FK_InitiativePost_Category_CategoryId" FOREIGN KEY ("CategoryId") REFERENCES "Category" ("Id") ON DELETE CASCADE )
"Id" INTEGER NOT NULL
"AuthorId" INTEGER NOT NULL
"CategoryId" INTEGER NOT NULL
"Content" TEXT NOT NULL
"DatePublished" TEXT NOT NULL
"Title" TEXT NOT NULL
CREATE TABLE "User" ( "Id" INTEGER NOT NULL CONSTRAINT "PK_User" PRIMARY KEY AUTOINCREMENT, "Username" TEXT NOT NULL, "Email" TEXT NOT NULL, "Password" TEXT NOT NULL, "Role" TEXT NOT NULL, "IsAdmin" TEXT NOT NULL )
"Id" INTEGER NOT NULL
"Username" TEXT NOT NULL
"Email" TEXT NOT NULL
"Password" TEXT NOT NULL
"Role" TEXT NOT NULL
"IsAdmin" TEXT NOT NULL
CREATE TABLE "InitiativePostVersion" ( "MigrationId" TEXT NOT NULL CONSTRAINT "PK_InitiativePostVersion" PRIMARY KEY, "ProductVersion" TEXT NOT NULL )
"MigrationId" TEXT NOT NULL
"ProductVersion" TEXT NOT NULL
CREATE TABLE "InitiativePost" ( "Id" INTEGER NOT NULL CONSTRAINT "PK_InitiativePost" PRIMARY KEY, "Title" TEXT NOT NULL )
"Id" INTEGER NOT NULL
"Title" TEXT NOT NULL
CREATE TABLE "InitiativePost" ( "Id" INTEGER NOT NULL CONSTRAINT "PK_InitiativePost" PRIMARY KEY, "Title" TEXT NOT NULL )
"Id" INTEGER NOT NULL
"Title" TEXT NOT NULL
CREATE INDEX "IX_InitiativePost_CategoryId" ON "InitiativePost" ("CategoryId")
```

The first screenshot illustrates the `Category` table in the database. This table stores a predefined list of categories that are used to organize initiatives. Each category is represented by two columns:

- **Id:** A unique identifier for each category (primary key).
- **Category:** The name of the category, describing its purpose or focus (e.g., "Concept Development" or "Sustainability and ESG").

These categories are essential for ensuring that initiatives are categorized correctly, making it easier for users to filter and navigate through the system. This table was seeded with initial data during database setup, ensuring consistency and usability from the start.

File Edit View Tools Help

New Database Open Database Write Changes Revert Changes

Database Structure Browse Data Execute SQL

Table: Category Filter in any column

	<u>Id</u>	Category
	Filter	Filter
1	1	Concept Development
2	2	Design & Prototyping
3	3	Manufacturing & Production
4	4	Sustainability & Green Technology
5	5	Sustainability and ESG (Environmental, Social, Governance)
6	6	Installation and Deployment
7	7	Operational Efficiency
8	8	Cost Optimization
9	9	Safety and Risk Management
10	10	Regulatory Compliance
11	11	Customer-Focused Initiatives
12	12	-Digitalization and Data Management

DB Browser for SQLite - C:\Users\gabit\OneDrive - Via\JC\Skribebord\Atelier source code\Atelier\AtelierAPI\NextGenAtelier.db

File Edit View Tools Help

New Database Open Database Write Changes Revert Changes Undo Open Project Save Project Attach Databases

Database Structure Browse Data Execute SQL

Table: InitiativePosts Filter in any column

	<u><a href="#">Id</a></u>	<a href="#">AuthorId</a>	<a href="#">CategoryId</a>	<a href="#">Content</a>	<a href="#">DatePublished</a>	<a href="#">Title</a>	
	<a href="#">Filter</a>	<a href="#">Filter</a>	<a href="#">Filter</a>	<a href="#">Filter</a>	<a href="#">Filter</a>	<a href="#">Filter</a>	
1	1	1	3	Implement automated robotic systems for precision cutting and assembly ...	2024-12-06 11:08:04.7691788	Improve Blade Manufacturing Efficiency	
2	5	4	4	Implement recycling programs in schools to reduce waste and educate...	2024-12-06 07:27:58.1707923	Recycling in Schools	
3	6	4	3	Leverage artificial intelligence to optimize production lines and ...	2024-12-06 07:28:31.402357	AI for Manufacturing	
4	7	4	2	Design and develop innovative prototypes for renewable energy ...	2024-12-06 07:28:48.5960692	Renewable Energy Prototypes	
5	8	4	1	Provide a platform for students to develop and pitch their innovative ...	2024-12-06 07:29:03.2881859	Idea Incubator for Students	
6	9	4	3	Develop eco-friendly packaging materials to reduce plastic waste i...	2024-12-06 07:29:19.4929967	Sustainable Packaging	
7	15	8	4	This initiative focuses on developing highly efficient and ...	2024-12-06 11:00:22.3486289	Innovative Solar Panel Design	
8	16	4	10	This initiative aims to develop an AI-powered system to assist ...	2024-12-06 11:09:02.3571038	Automated Compliance Monitoring System	
9	18	9	10	ehg3e4heht4eh	2025-01-05 21:42:04.7558905	fw3rg	
10	19	9	3	efgh	2025-01-05 23:23:10.6584431	abc	

The second screenshot depicts the InitiativePosts table, which stores all initiatives created by users. This table highlights the relationship between initiatives and their authors, as well as the categorization of initiatives. The seeded data includes sample initiatives, providing examples for testing and illustrating how real-world content might look within the system.

**Id:** A unique identifier for each initiative.

**AuthorId:** A foreign key referencing the Users table, linking each initiative to its creator.

**CategoryId:** A foreign key referencing the Category table, indicating the category of the initiative.

**Content:** A detailed description of the initiative.

**DatePublished:** A timestamp recording when the initiative was published.

**Title:** A brief title summarizing the initiative.

The third screenshot shows the Users table, which maintains data about all registered users. The table includes seeded data for an admin account, which has elevated privileges, such as managing all initiatives. Regular user accounts have more restricted permissions, such as managing only their own initiatives. This structure supports a role-based access control system, ensuring appropriate levels of access for different users.

**Id:** A unique identifier for each user (primary key).

**Username:** The name the user uses to log in and identify themselves.

**Email:** The user's email address, which could also be used for communication or authentication.

**Password:** A hashed version of the user's password for secure storage.

**Role:** Specifies whether the user is an Admin or a User, determining their privileges in the system.

**DateJoined:** The timestamp indicating when the user registered on the platform.

DB Browser for SQLite - C:\Users\gabid\OneDrive - ViaUC\Skribebord\Atelier source code\Atelier\AtelierAPI\NextGenAtelier.db

File Edit View Tools Help

New Database Open Database Write Changes Revert Changes Undo Open Project Save Project Attach Database

Database Structure Browse Data Execute SQL

Table: Users Filter in any column

	<u>Id</u>	Username	Email	Password	Role	DateJoined
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	admin	316333@viauc.dk	hashed_password	Admin	2024-12-06 11:08:04.7685366
2	2	sa	sa	\$2a\$11\$UFWLrU/kpe2j3PMhrslceL1CS0c52ExhULay3pHQuBi...	User	2024-12-06 06:21:28.7428201
3	3	bogdan	bogdan	\$2a\$11\$D3tCym76uF91phTQj5N0DuiPx5uB...AAa	User	2024-12-06 06:50:21.4372642
4	4	a	a	\$2a\$11\$H13twYfRh.9QHGFtQqAoYQjvruqakuV444QAPHJNZ5nn...	User	2024-12-06 07:15:54.9216757
5	5	bo	bo	\$2a\$11\$IT/K9Tm6tqu3d09m8yuSh.9fsQmZyU2gz1RQTQ42Gy14nR22Ye2hu	User	2024-12-06 10:04:32.0052766
6	6	q	q	\$2a\$11\$czeSHx8zB2Z4AOPGfGwrW8.4p4j69...	User	2024-12-06 10:39:52.6886516
7	7	Ga	Ga	\$2a\$11\$N4AddJ0HRCLKEIzWbWBpVeZdStJ...	User	2024-12-06 11:50:37.2141739
8	8	gabriela	gabriela	\$2a\$11\$PjYwJb2JvKdiBw7.5V8c7ORwSu00...mdF.7NHATZW4e	User	2024-12-06 11:59:18.9445882
9	9	Gabriela	Gabriela	\$2a\$11\$LEeAqbBeGly.EiK4jhRqp..0fDgbN37yFLdLqN2CdqPhbhlGShIm0	User	2025-01-05 21:07:33.1857181
10	10	ggg	string	\$2a\$11\$zsKQTetYTopHaeBF2YNGc.1AbVq/Qsob1wjUJYV/q61MH8BO7GuUm	User	2025-01-05 22:35:41.041933

## #6 Project Conclusion & Demonstration:

The Atelier project has a RESTful web service that acts as the backend. It handles all the heavy lifting, like managing database queries and processing requests from the frontend. The service is built using ASP.NET Core, and it follows REST principles, making it easy for the frontend to interact with it over HTTP. Through this web service, the system can perform operations like creating, reading, updating, and deleting data for users, initiatives, and categories. While it doesn't include HATEOAS (which wasn't required), it still gets the job done efficiently.

The frontend for Atelier is a Blazor WebAssembly application, which provides the user interface. It's where users can log in, browse initiatives, and manage their data. Blazor makes it easy to build an interactive, responsive frontend that looks good and works seamlessly. It communicates with the backend via HTTP, so all the data is fetched, updated, and stored by the backend while the frontend focuses on giving users a smooth experience.

In Atelier, the web service and the frontend are two completely separate programs. The backend is the RESTful API built with ASP.NET Core, and the frontend is the Blazor application. They communicate via HTTP requests. This separation is a smart design choice because it makes the system modular—each part can be updated or scaled independently without breaking the other.

The Atelier project successfully fulfills most of the outlined user stories, including user registration and secure login with hashed passwords, allowing users to create accounts and manage their initiatives. Logged-in users can submit, edit, and delete their own initiatives, while admins have full control to moderate, edit, or delete any initiative to maintain content quality. Both visitors and logged-in users can view all initiatives, with guests also able to browse initiatives by category without needing an account. The system effectively uses role-based access control to differentiate between admin and regular user privileges. However, while the project implements most functionalities, it currently lacks pagination for initiatives, meaning all initiatives are displayed at once without being split into pages. Overall, the project meets 10 out of 11 requirements, with pagination being the only feature yet to be implemented.