Glam Link

FASHION APPLICATION

--CLOSET AND OUTFIT ORGANIZER--

Project by:

- Cuc Ana Alexia (Team leader)
- Ciubăncan Mara (Software developer)
- Capac Teodora (Software developer)
- Moroșanu Ștefania (Software tester/Software developer)

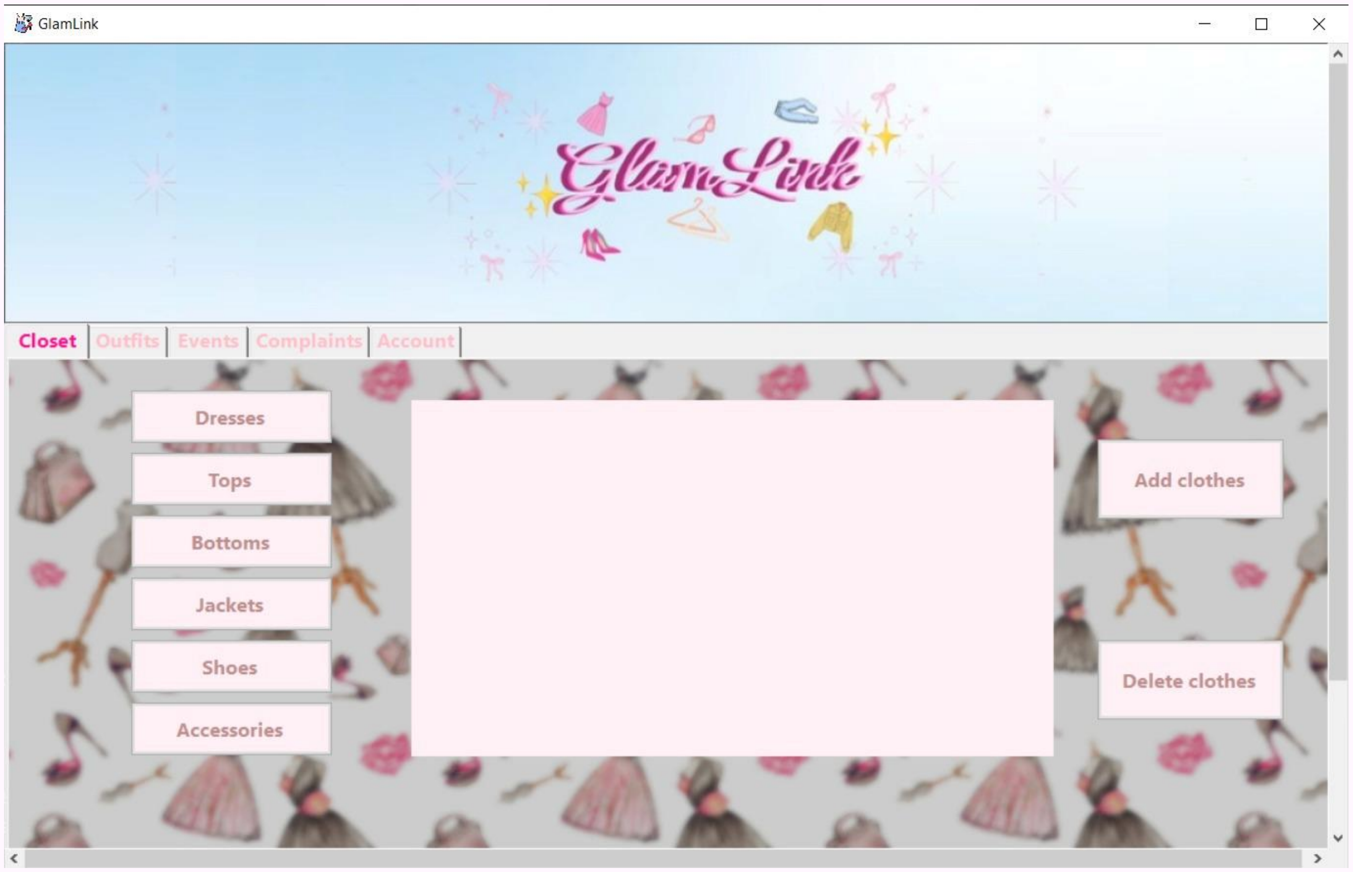
Table of contents

1. Introduction

The GlamLink application is a **fashion planning and wardrobe management tool** that helps users organize their clothing items, create and manage outfits and plan events. It is designed as a multi-tier C# application using Visual Studio, **Windows Forms** for the user interface, and an **ASP.NET Web Core API** for backend logic, with **SQL Server Management Studio** for the relational database.

The GlamLink application is developed using a three-tier architecture, which ensures maintainability, modularity, and clear division of responsibilities and separates the system into three distinct layers:

I. **Data Access Layer** (SQL Server using Entity Framework Core).
II. **Logic Layer** (ASP.NET Web Core API),
III. **Presentation Layer** (Windows Forms UI);



Goals:

Provide an intuitive and visually appealing interface for creating outfits by combining clothing items like dresses, tops, bottoms, jackets, shoes, and accessories.	
Help users plan and manage events by linking selected outfits, making outfit preparation easier.	
Allow administrators to manage and respond to user complaints to ensure app quality.	

Objectives:

Implement a structured three-tier architecture to separate the user interface, business logic, and data access layers.	
Support various clothing categories with responsive UI elements based on user selections.	
Link clothing items, outfits, and events to specific users through unique IDs.	

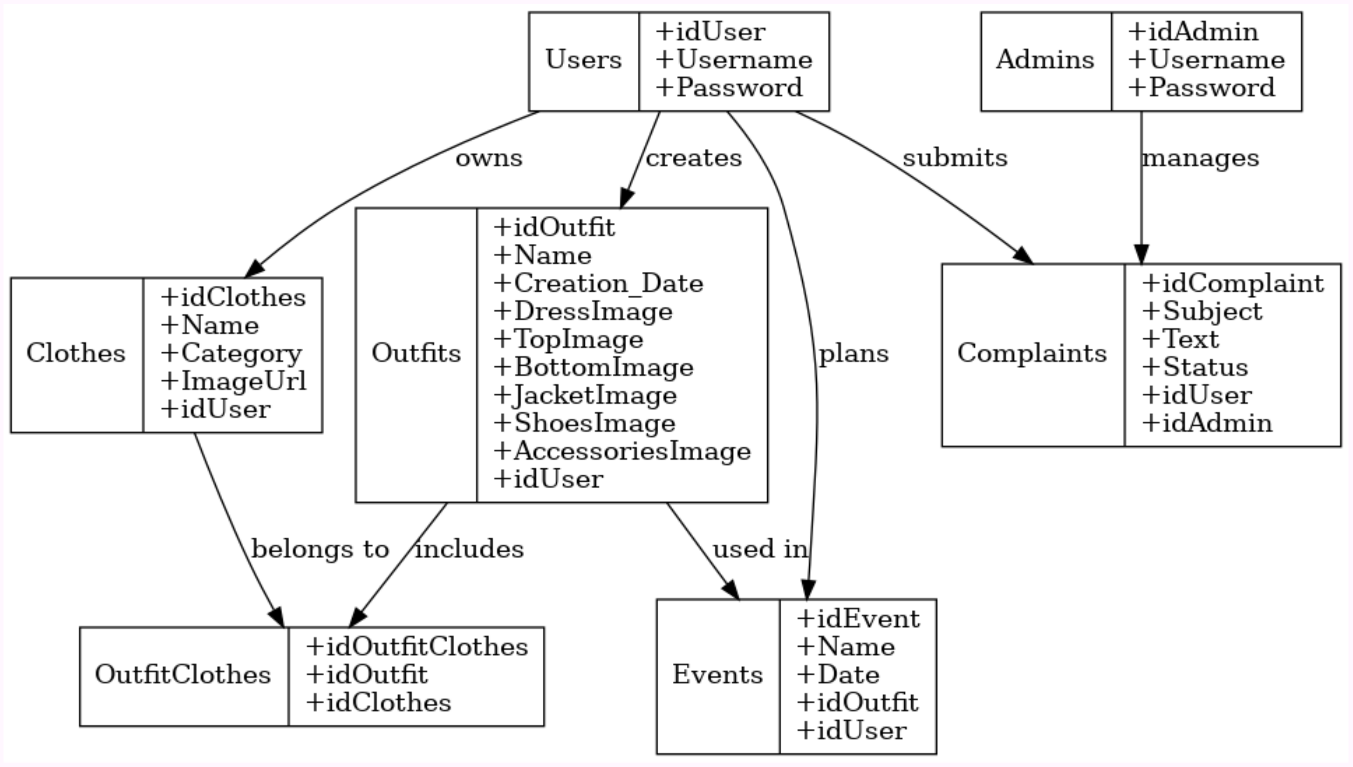
Functions:

Enable secure user registration and login with role-based access (user/admin).	
Allow users to add clothing items and view them in categories with image previews.	
Let users create outfits by selecting one item per category, with options to update or delete them.	
Allow event planning by selecting a date, time, and linking an outfit.	
Provide a way for users to submit complaints that admins can review and resolve.	

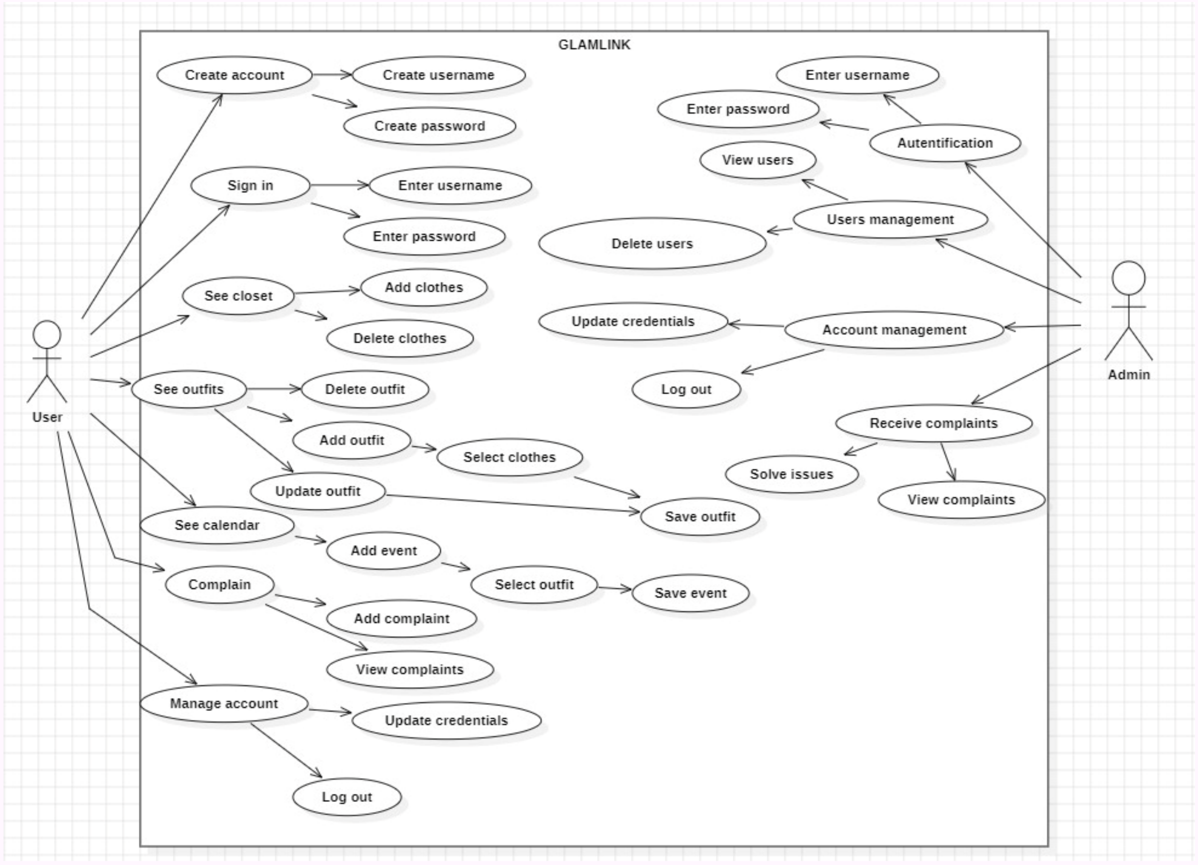
2. Application design

Database Design

The GlamLink database design models key entities along with their relationships; the design ensures clear links between users, their content, and supported actions.

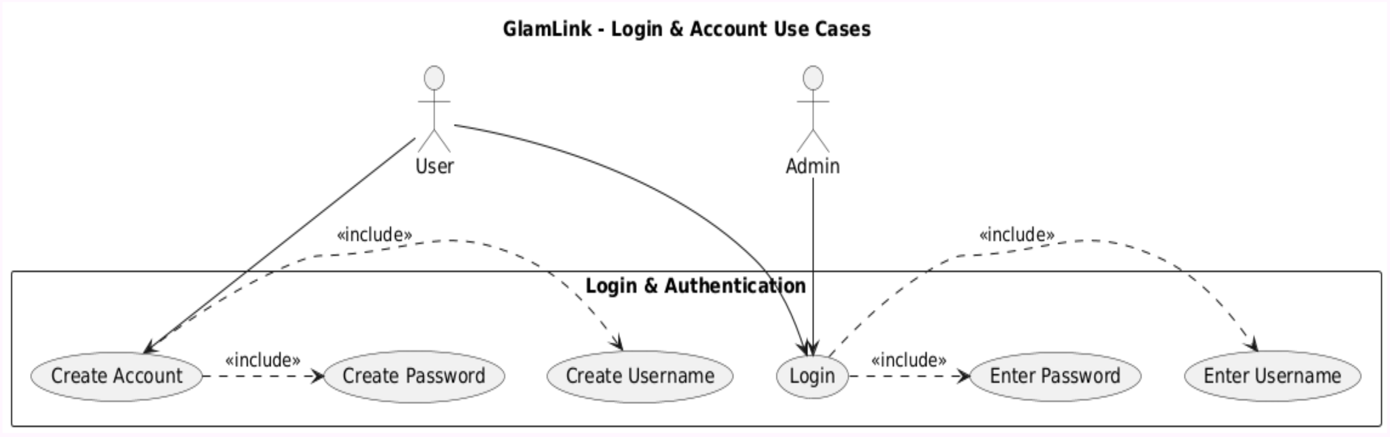


The application's design includes UML diagrams for use case and class diagrams. The architecture ensures good extensibility and a clear separation between **logic** and **interface**.

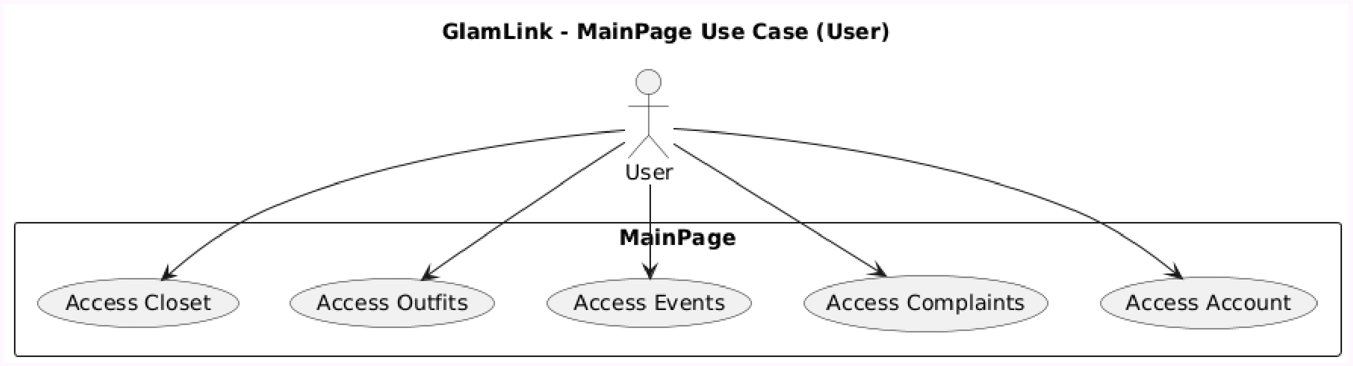


Key forms include:

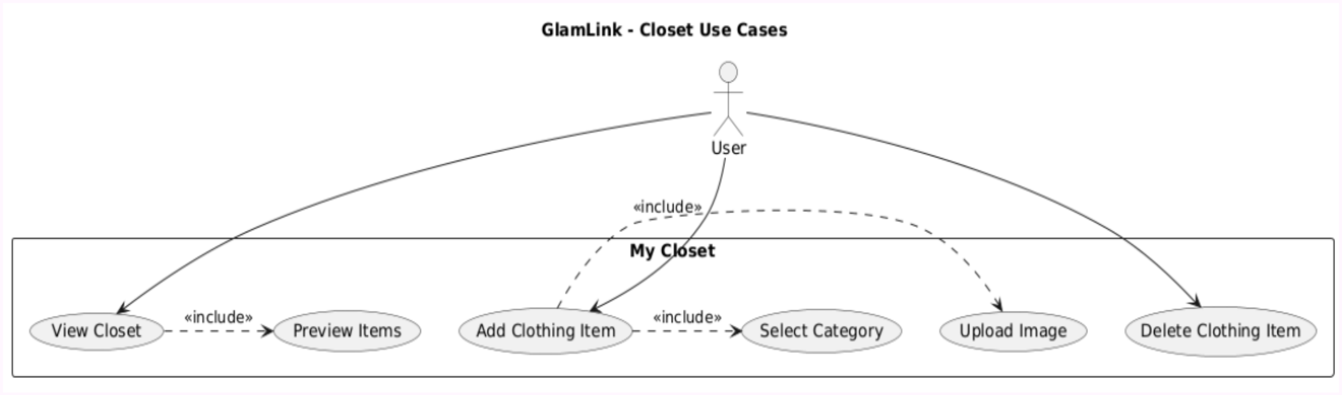
- *LoginForm.cs*: Authenticates users. Below is the use case diagram for login and account creation: Account). Related use case:



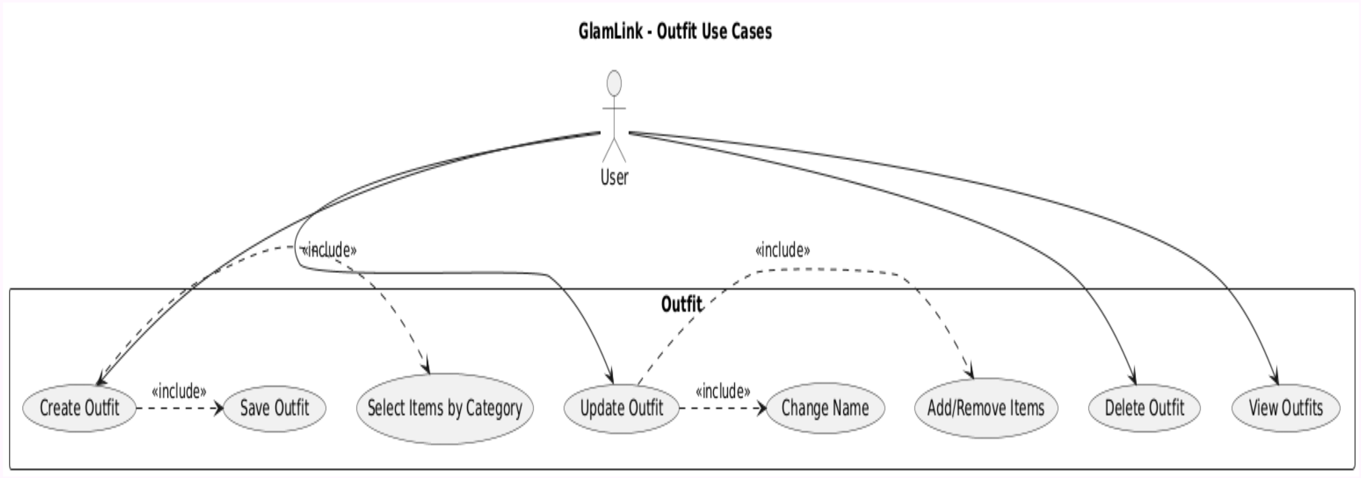
- *MainPage.cs*: Hosts navigation tabs (Closet, Outfits, Events, Complaints, Account). Related use case:



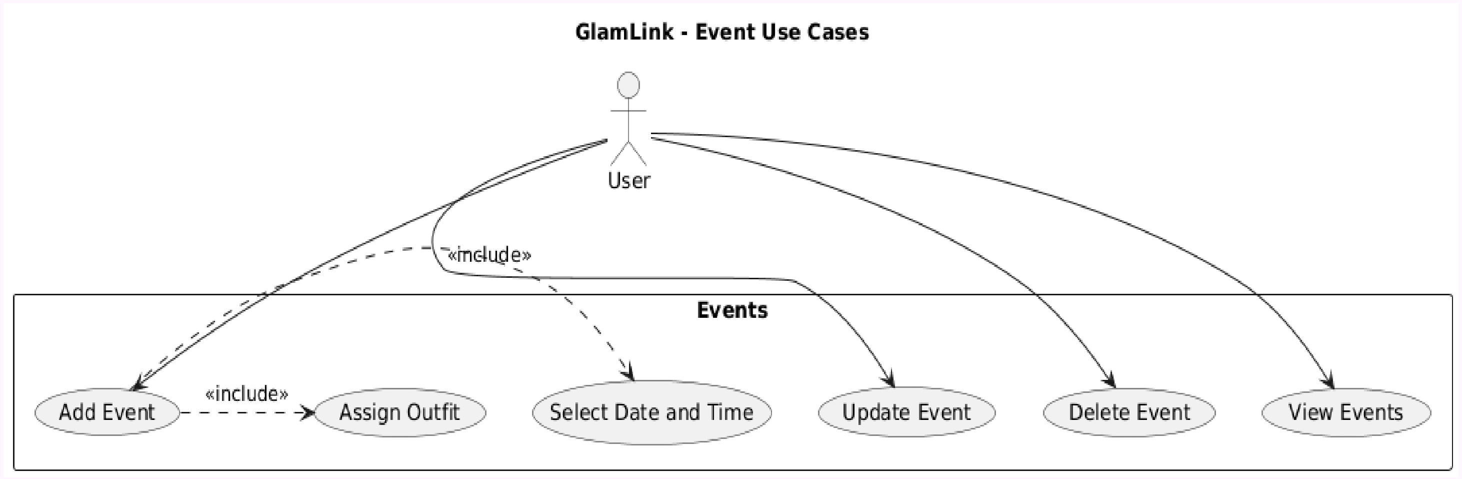
- *AddClothesForm.cs*: The **Closet tab** allows users to add, view, and delete clothing items. Related use case:



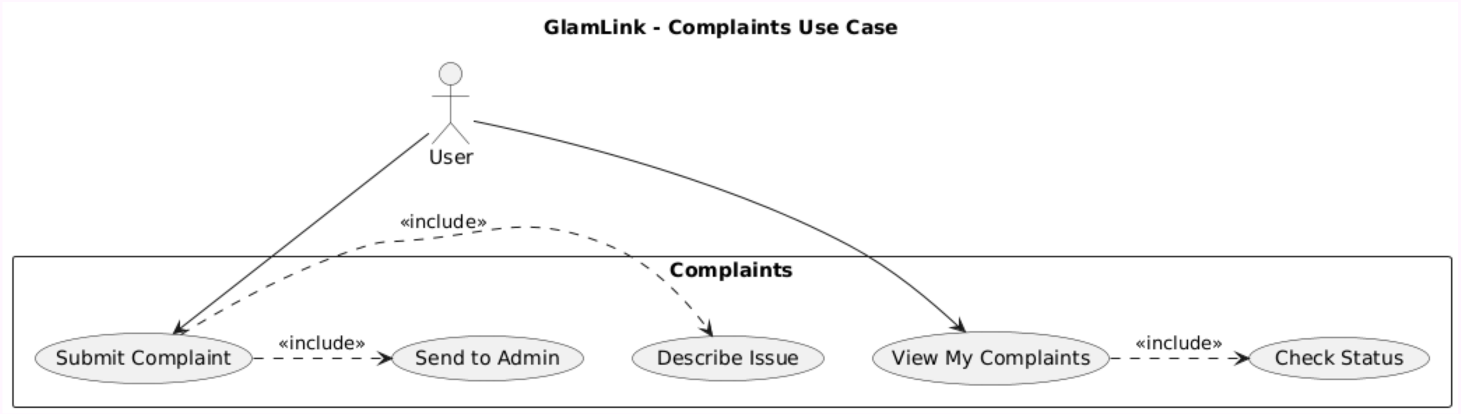
- *AddOutfitsForm.cs*, *UpdateOutfitForm.cs:* The **Outfits tab** enables outfit creation and editing using selected clothing:



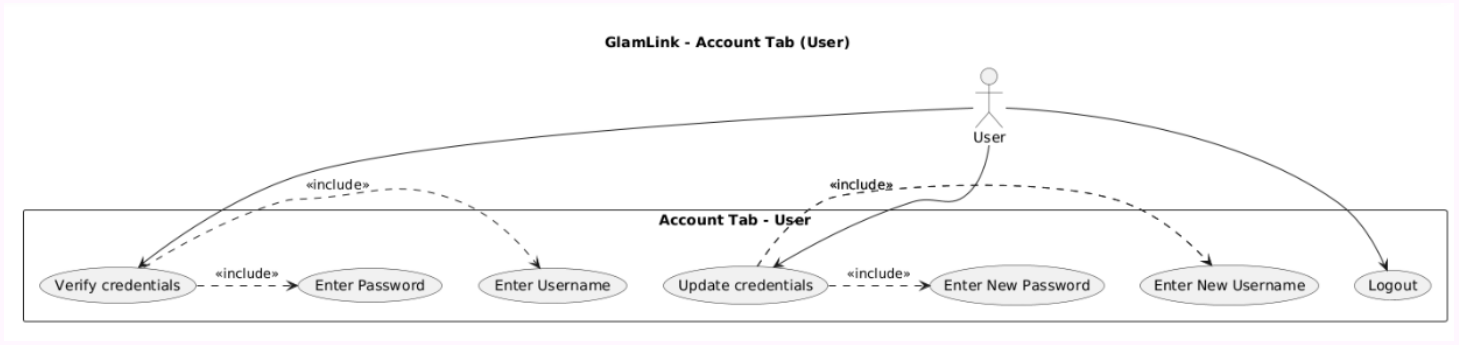
- *AddEventForm.cs, UpdateEventForm.cs*: The **Events tab** handles outfit-linked event scheduling. Its use cases are shown below:



- *UserComplaintsForm.cs:* Users can file complaints and check their status in the **Complaints tab**. This diagram outlines those actions:

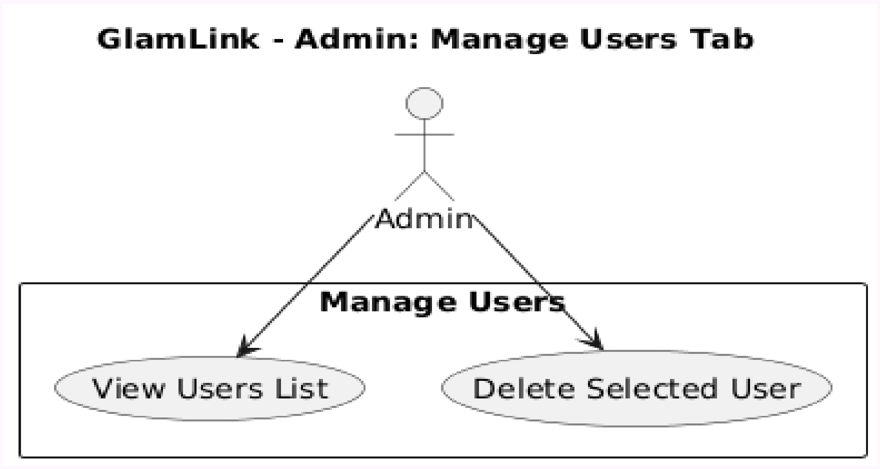


In the **Account tab** the user can update its credentials, after verifying them and to logout.

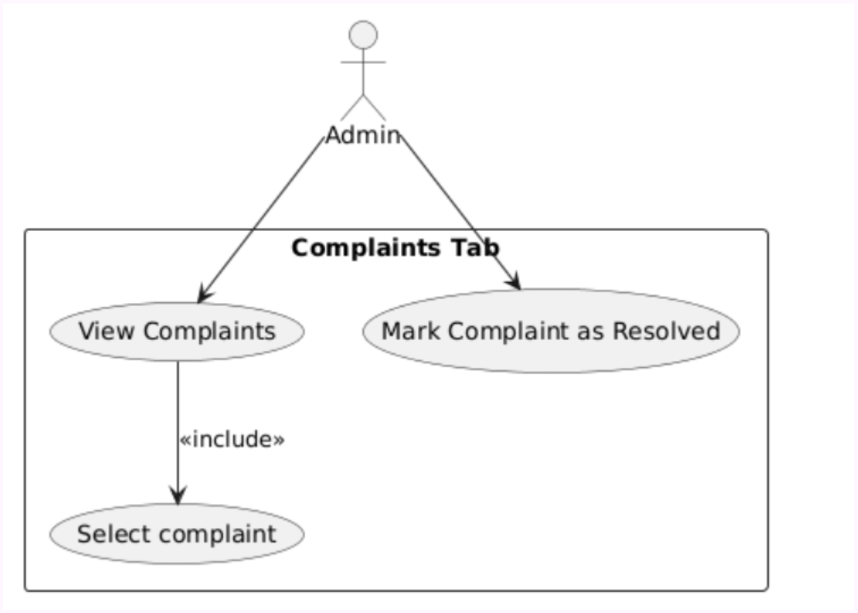


- *AdminForm.cs*: Allow administrators to view and handle user-submitted complaints:

  The **Manage Users tab** allows the admin to view and delete user accounts:



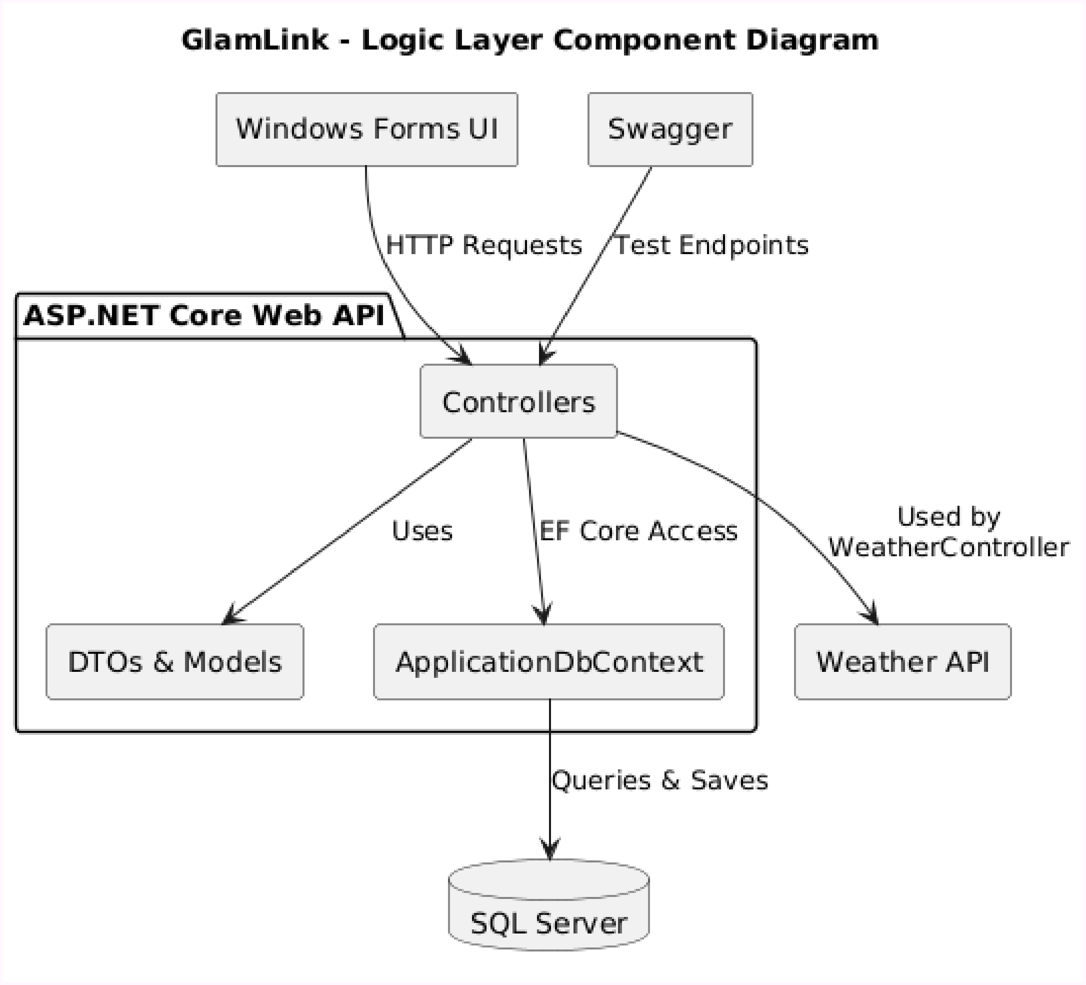
Through the **Complaints tab**, admins can handle issues submitted by users. They select the complaints, mark as resolved, or delete complaints:



The **Account Tab** for admins has the exact same functionality as the one for users.

The following component diagram illustrates how the Windows Forms UI interacts with the ASP.NET Core Web API. It shows how API controllers use internal models and DTOs, access the database via Entity Framework Core, and call external services like the Weather API.

Connectivity Flow

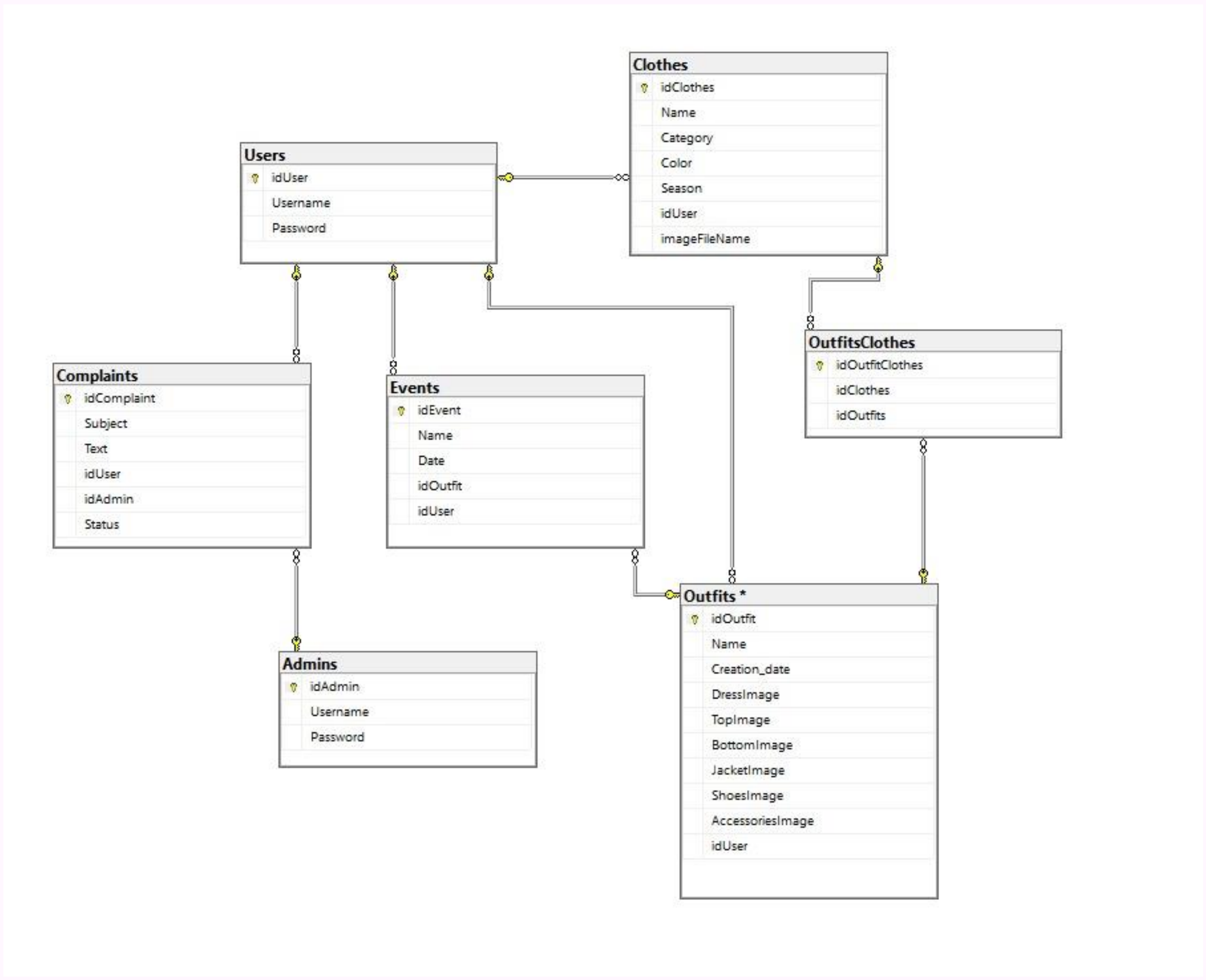


1. The **user interacts** with a Windows Form (e.g., clicks "Add Outfit").
2. The form sends a **HTTP request** to the Web API endpoint.
3. The **API controller** processes the request, applies any rules, and interacts with the database through EF Core.
4. **SQL Server** executes queries and returns the result to the API. The API responds, and the UI updates accordingly.

3. Application implementation

Database Implementation

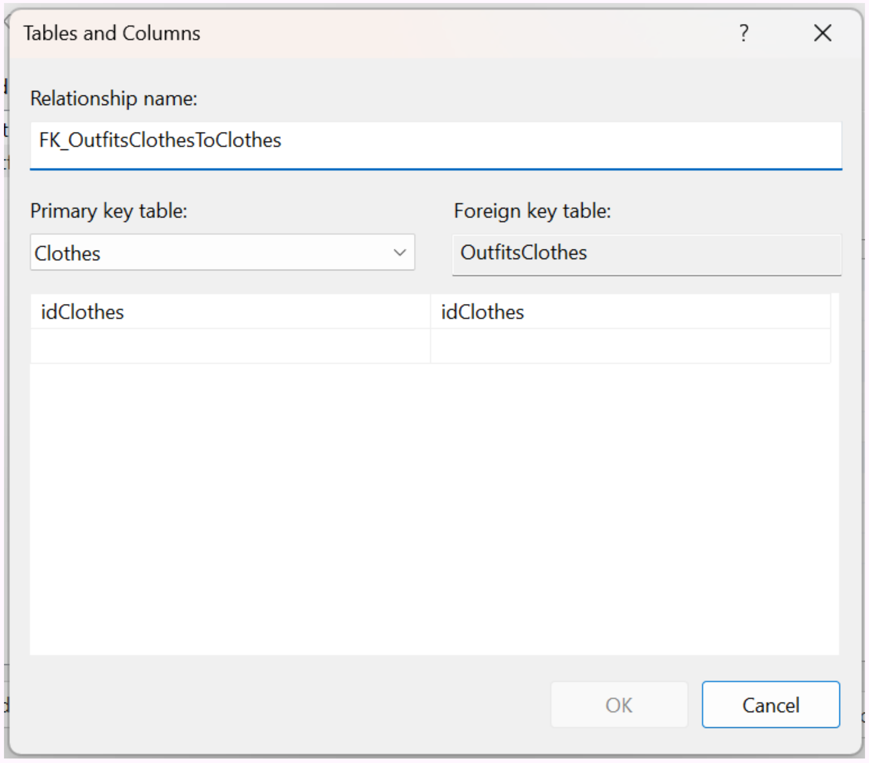
Data is stored in a **SQL Server database**, managed through **SQL Server Management Studio (SSMS)**. The Entity Framework Core is used as the Object-Relational Mapper, with ApplicationDbContext acting as the bridge between code and database.



Main Tables:

Table Name	Purpose
Users	Stores user accounts and login info
Admins	Admin-level accounts
Clothes	Clothing items by category, user ID, image URL
Outfits	Stores created outfits with image fields
OutfitsClothes	(if used) Links outfits to clothing items
Events	Stores events, dates, and linked outfit
Complaints	Stores complaints submitted by users

In the next capture is presented an example of how the Foreign Keys were set in our database; the relationship ensures that each clothing item referenced in the OutfitsClothes table must exist in the Clothes table, enforcing referential integrity between the two tables. It supports the **many-to-many relationship** between outfits and clothing items.



Relationship name: FK_OutfitsClothesToClothes

Primary key table: Clothes

Foreign key table: OutfitsClothes

Clothes	OutfitsClothes
idClothes	idClothes

Services and API Routing

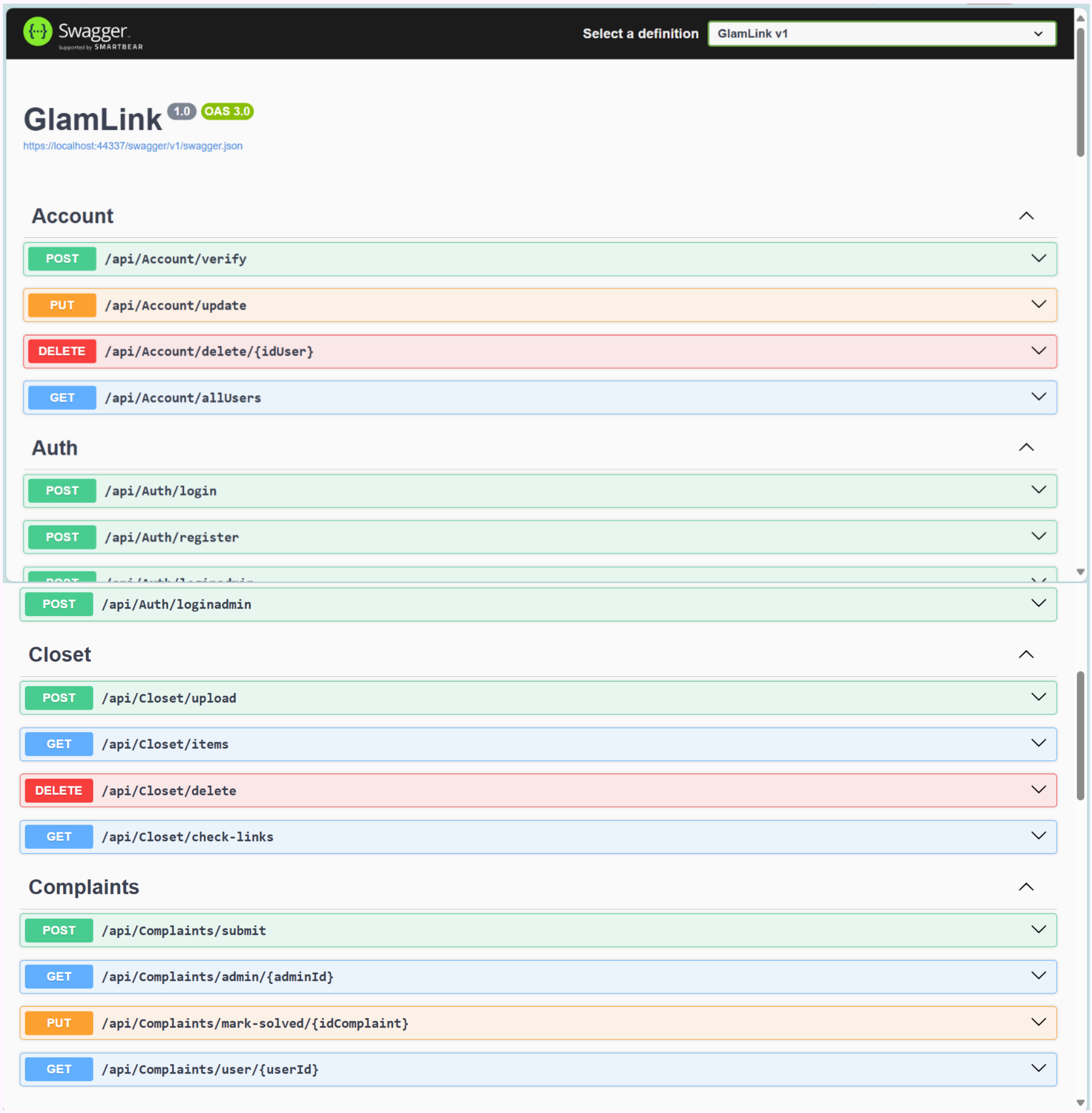
Swagger - Automatically documents and allows testing of the API.

GlamLink uses Swagger UI to automatically generate interactive documentation for all available Web API endpoints. This enables developers to browse, test, and validate API routes in real time during development.

Each controller accepts HTTP requests (GET, POST, PUT, DELETE), processes input (often via DTOs), interacts with the *ApplicationDbContext.cs*, and sends a structured response. The Swagger main view displays the list of all API controllers (Account, Auth, Closet, etc.), each grouped by domain.

Controllers - Entry points for API requests:

- AccountController: (/api/account) Verifies and updates account information (users/admins).
- AuthController: (/api/auth) Handles registration and login authentication.
- ClosetController: (/api/closet) Manages clothing item uploads, previews, and deletions.
- OutfitController: (/api/outfits) Creates, updates, and deletes outfits.
- EventController: (/api/events) Handles event creation, updates, and deletions.
- ComplaintController: (/api/complaints) Manages complaint submissions and admin resolution.
- WeatherController: (/api/weather) Retrieves weather data from an external API.



Swagger supported by SMARTBEAR

Select a definition GlamLink v1

GlamLink 1.0 OAS 3.0

https://localhost:44337/swagger/v1/swagger.json

Account

POST /api/Account/verify

PUT /api/Account/update

DELETE /api/Account/delete/{idUser}

GET /api/Account/allUsers

Auth

POST /api/Auth/login

POST /api/Auth/register

POST /api/Auth/loginadmin

Closet

POST /api/Closet/upload

GET /api/Closet/items

DELETE /api/Closet/delete

GET /api/Closet/check-links

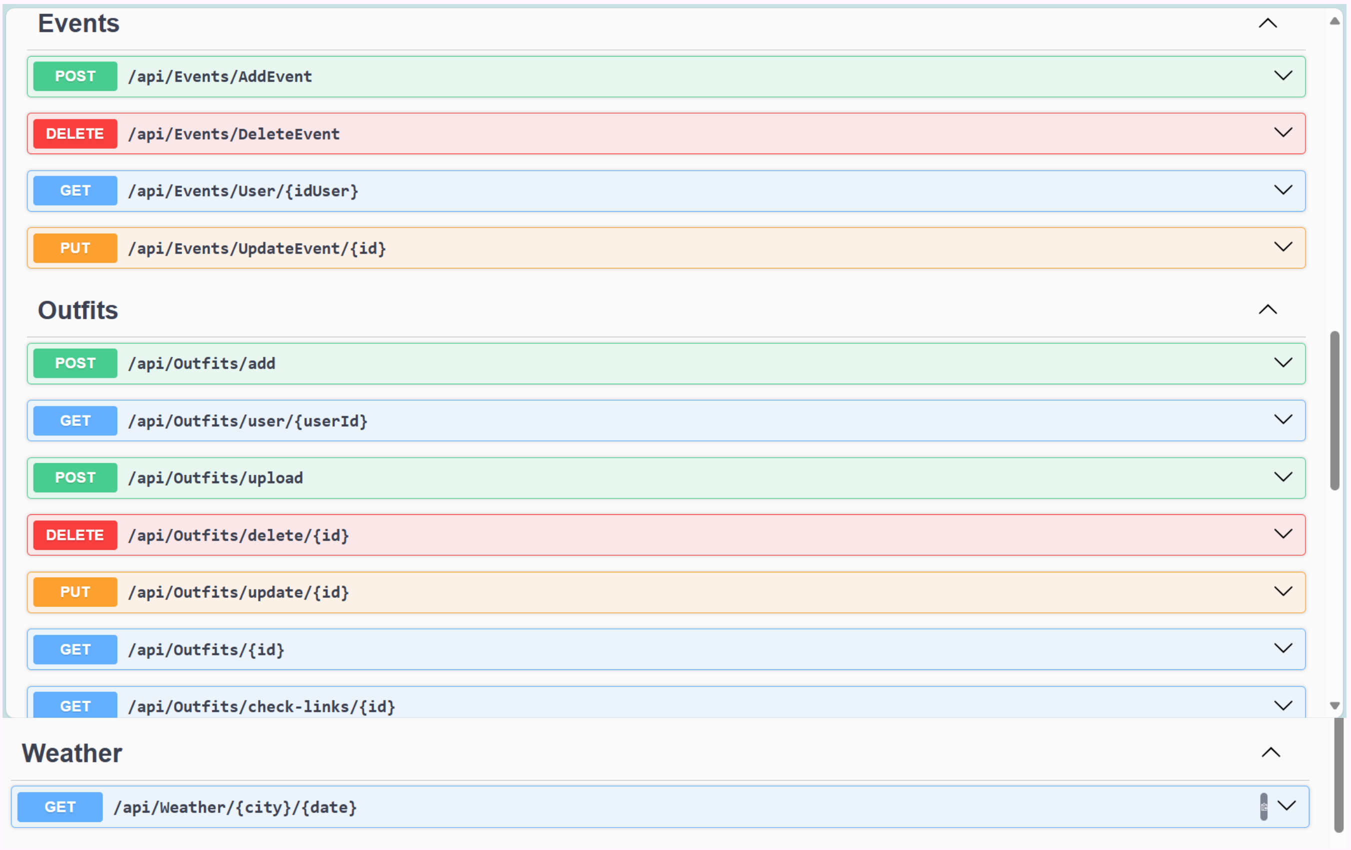
Complaints

POST /api/Complaints/submit

GET /api/Complaints/admin/{adminId}

PUT /api/Complaints/mark-solved/{idComplaint}

GET /api/Complaints/user/{userId}



Events

POST /api/Events/AddEvent

DELETE /api/Events/DeleteEvent

GET /api/Events/User/{idUser}

PUT /api/Events/UpdateEvent/{id}

Outfits

POST /api/Outfits/add

GET /api/Outfits/user/{userId}

POST /api/Outfits/upload

DELETE /api/Outfits/delete/{id}

PUT /api/Outfits/update/{id}

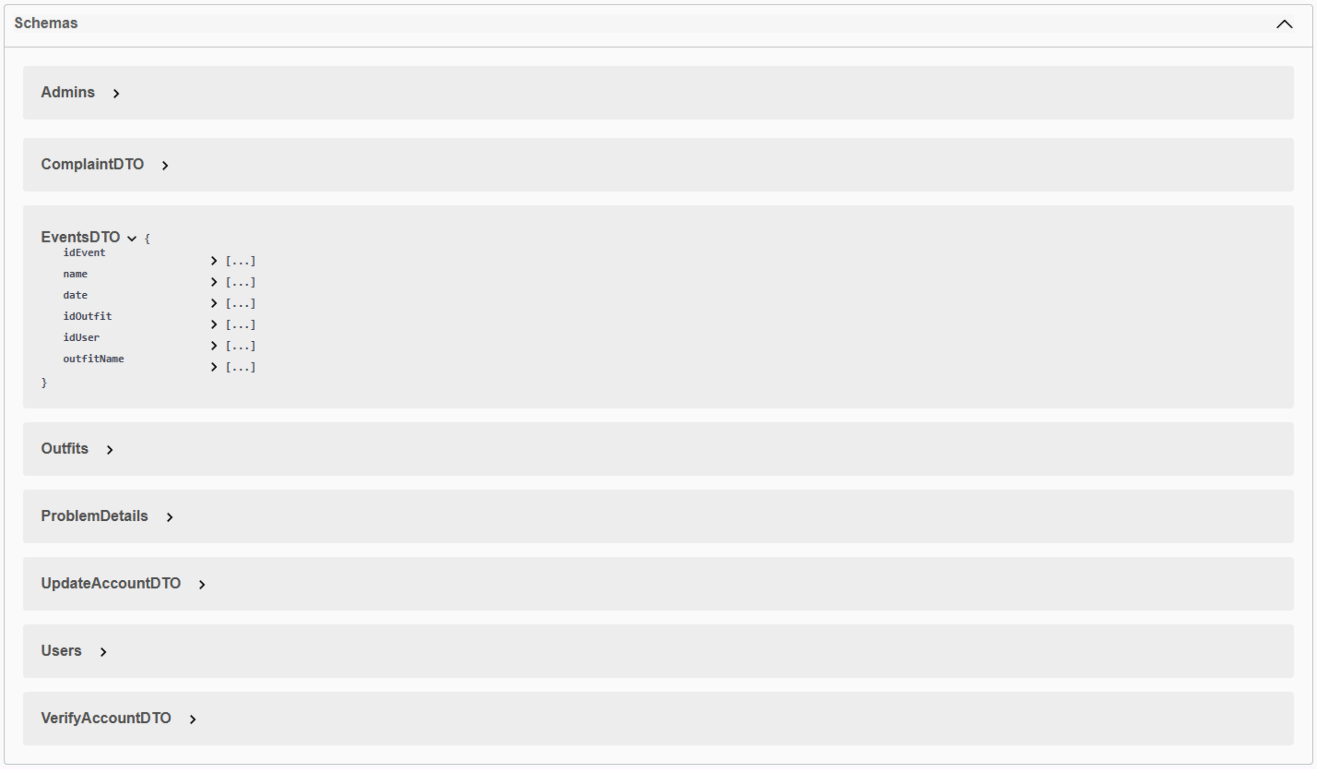
GET /api/Outfits/{id}

GET /api/Outfits/check-links/{id}

Weather

GET /api/Weather/{city}/{date}

The Schemas section shows all DTOs and models used in the API: EventsDTO, ComplaintDTO, Users, VerifyAccountDTO, etc. Each can be clicked to preview its structure.



Schemas

Admins ›

ComplaintDTO ›

```
EventsDTO ⌄ {
    idEvent          › [...]
    name             › [...]
    date             › [...]
    idOutfit         › [...]
    idUser           › [...]
    outfitName       › [...]
}
```

Outfits ›

ProblemDetails ›

UpdateAccountDTO ›

Users ›

VerifyAccountDTO ›

How does it work?

- The controller receives a **DTO** (e.g., OutfitDTO) from an HTTP request.
- The API transforms it into a **Model** (e.g., Outfit) and saves it to the database.
- When data is retrieved, it is returned as a DTO again, not as a raw Model.

Models

In the GlamLink application, models play a crucial role in representing the structure of the database by using objected-oriented classes for each table in the database. Using **Entity Framework Core**, each model class corresponds directly to a database table (e.g., Users, Admins, Clothes, Outfits, Events, Complaints, OutfitClothes).

These models define the **data schema** through their corresponding properties. They are essential for:

- **Efficient communication** with the database.
- Supporting **data migrations** and **schema updates**.
- Allowing the application to perform **CRUD operations** (Create, Read, Update, Delete) without writing raw SQL.

By clearly mapping real-world entities to structured classes, models ensure that the application remains maintainable, scalable, and easy to integrate with the frontend and backend layers.

To illustrate the connection between the swagger and backend code we chose to **describe two random key Web API endpoints** from the GlamLink application, which are documented and interactable via Swagger UI.

Submit Complaint Endpoint

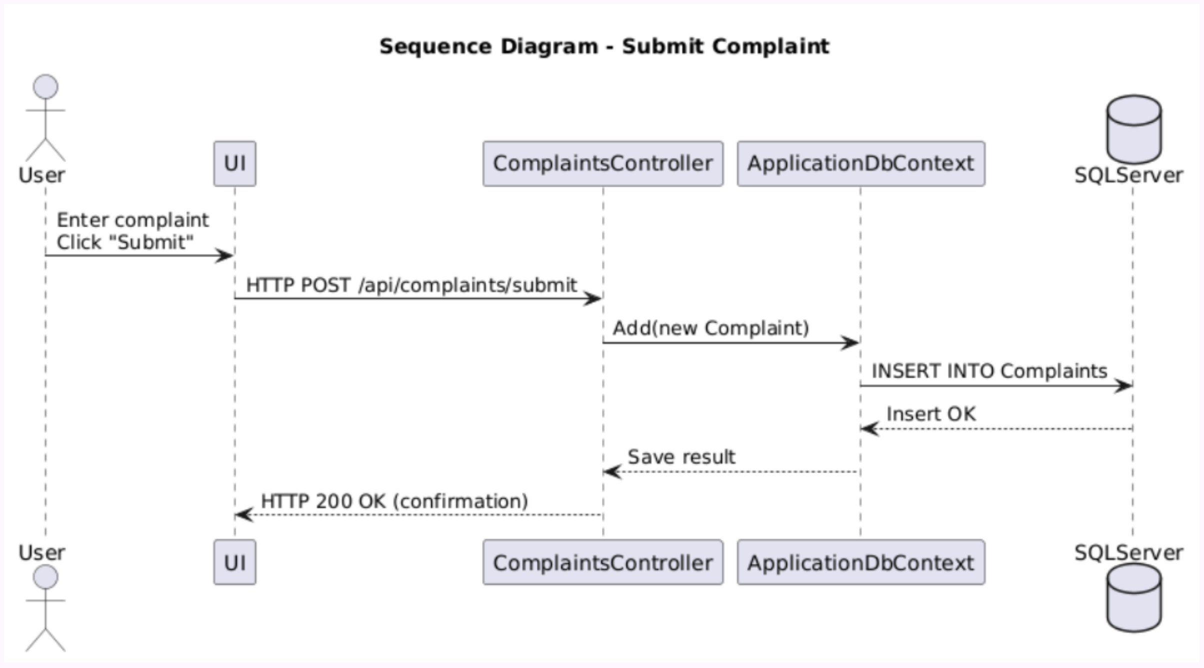
Route: POST /submit

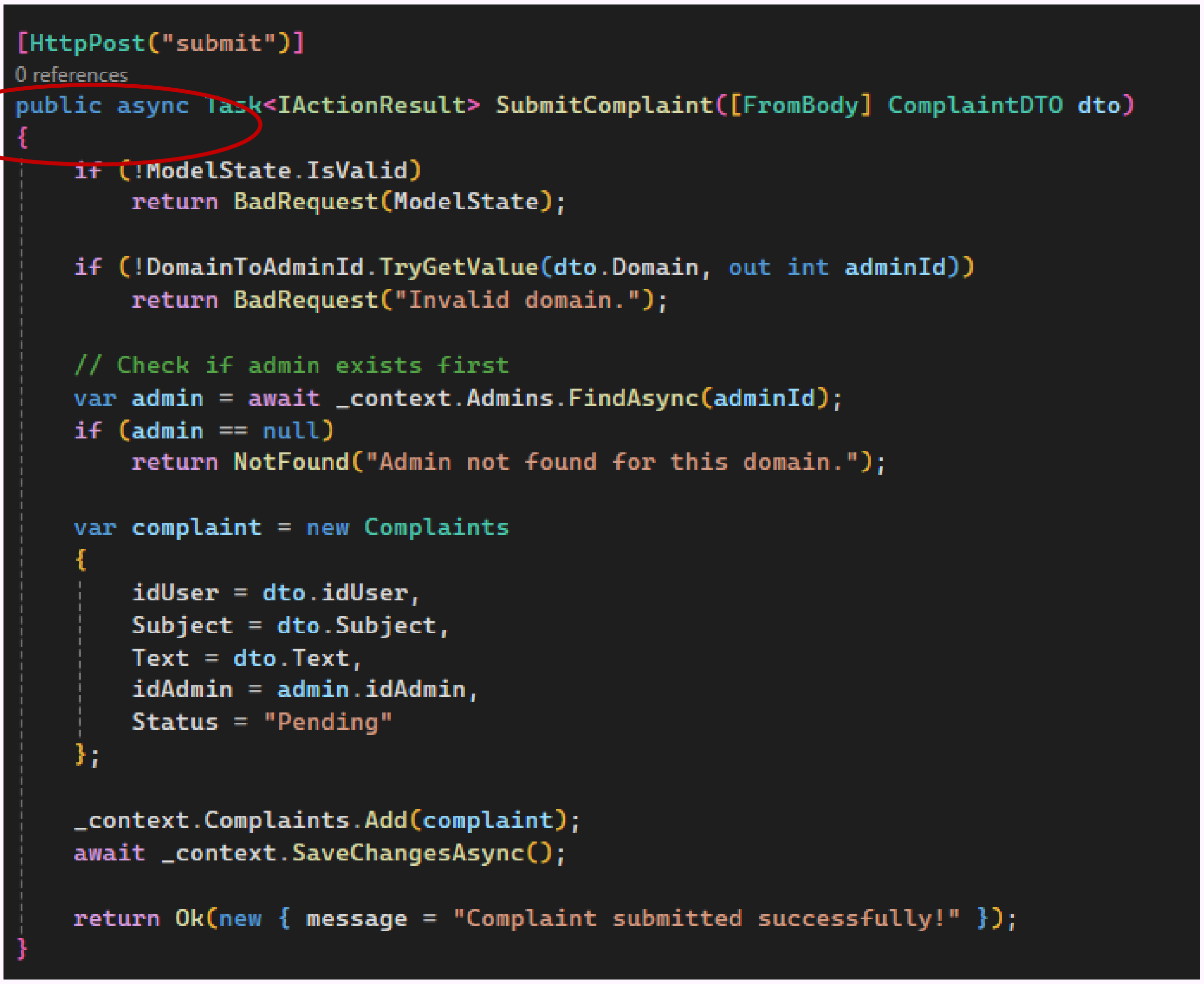
Purpose: Allows a user to submit a complaint that is automatically assigned to an appropriate admin based on the domain.

Implementation:
This method receives a ComplaintDTO object from the request body. It performs several validations:

- Ensures the model is valid.
- Maps the complaint domain to an admin using DomainToAdminId.
- Checks whether the admin exists in the database.
- If all checks pass, it creates a new Complaints object, sets its initial status to "Pending", and saves it to the database.

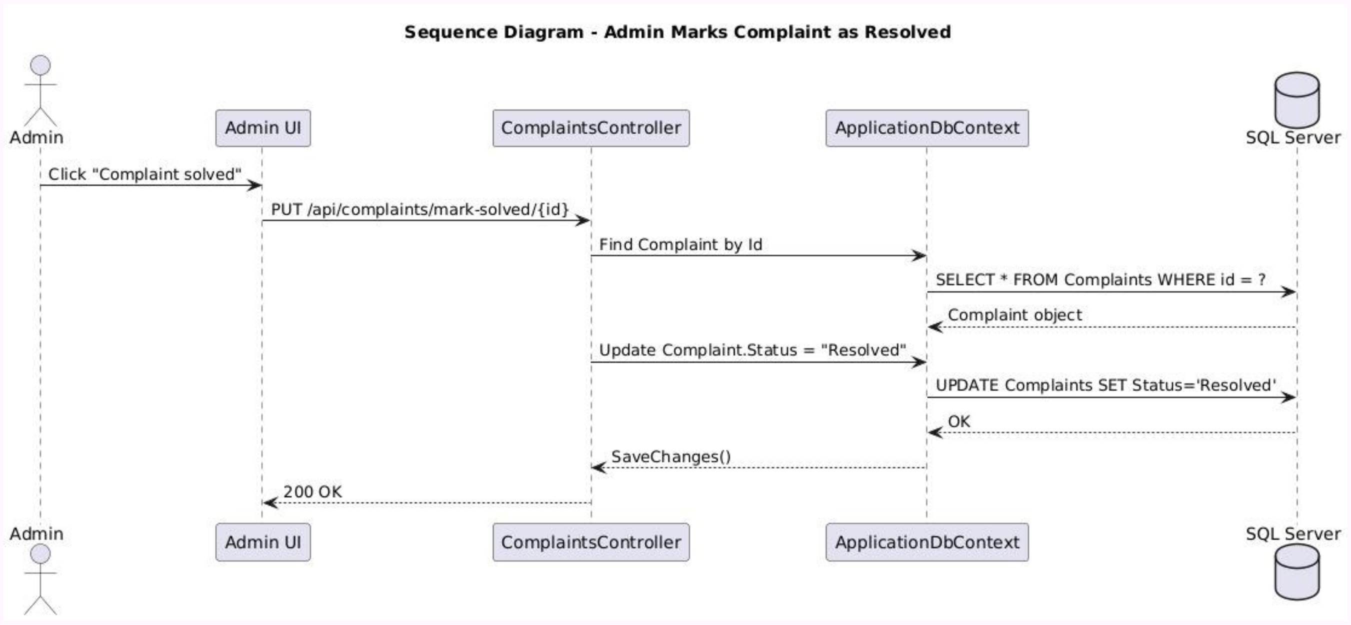
Swagger Usage:
Through Swagger UI, users can send a JSON request with complaint details (Subject, Text, Domain, etc.). If the domain is valid and an admin exists, the complaint is stored and a message is returned:
"Complaint submitted successfully!"

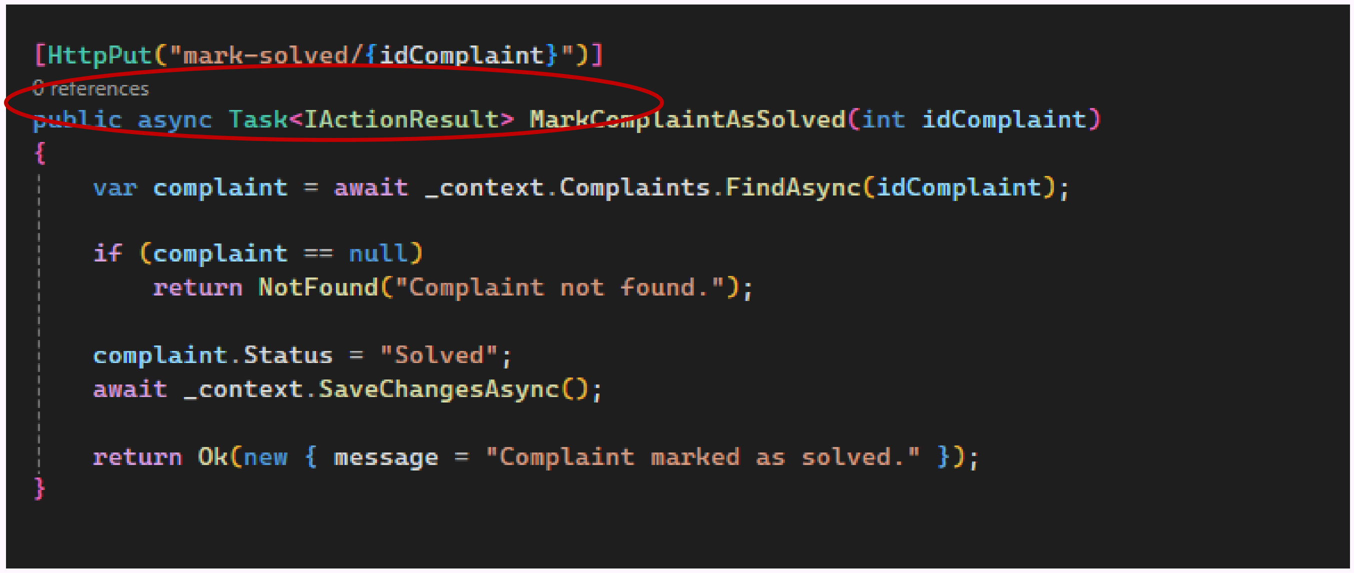




```
[HttpPost("submit")]
0 references
public async Task<IActionResult> SubmitComplaint([FromBody] ComplaintDTO dto)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    if (!DomainToAdminId.TryGetValue(dto.Domain, out int adminId))
        return BadRequest("Invalid domain.");

    // Check if admin exists first
    var admin = await _context.Admins.FindAsync(adminId);
    if (admin == null)
        return NotFound("Admin not found for this domain.");

    var complaint = new Complaints
    {
        idUser = dto.idUser,
        Subject = dto.Subject,
        Text = dto.Text,
        idAdmin = admin.idAdmin,
        Status = "Pending"
    };

    _context.Complaints.Add(complaint);
    await _context.SaveChangesAsync();

    return Ok(new { message = "Complaint submitted successfully!" });
}
```

1. **Mark Complaint as Solved**
   - **Route:** PUT /mark-solved/{idComplaint}
   - **Purpose:** Allows an admin to mark a complaint as resolved by updating its Status field in the Complaints table.
   - **Implementation:** The method retrieves the complaint by ID, checks for existence, updates the status to "Solved", and saves changes.
   - **Swagger Usage:** Provides a clear interface where an admin can input a complaint ID and confirm that the complaint has been resolved.


Sequence Diagram - Admin Marks Complaint as Resolved
Admin
Admin UI
ComplaintsController
ApplicationDbContext
SQL Server
Click "Complaint solved"
PUT /api/complaints/mark-solved/{id}
Find Complaint by Id
SELECT * FROM Complaints WHERE id = ?
Complaint object
Update Complaint.Status = "Resolved"
UPDATE Complaints SET Status='Resolved'
OK
SaveChanges()
200 OK



```
[HttpPut("mark-solved/{idComplaint}")]
0 references
public async Task<IActionResult> MarkComplaintAsSolved(int idComplaint)
{
    var complaint = await _context.Complaints.FindAsync(idComplaint);

    if (complaint == null)
        return NotFound("Complaint not found.");

    complaint.Status = "Solved";
    await _context.SaveChangesAsync();

    return Ok(new { message = "Complaint marked as solved." });
}
```

4. Application testing

To verify the functionality, reliability, and usability of the GlamLink application, extensive manual testing was carried out using both the Windows Forms interface and Swagger UI for backend endpoints. Testing covered the core features available to users and administrators.

The full set of test cases, including actions, expected results, and outcomes, is documented in Annex 1 – Test Case Sheet (Excel). The Excel file is organized into sheets corresponding to specific modules or tabs of the application.

Testing Scope

Each major area of the application was tested as follows:

Module / Tab	Test Focus
User – Login Page	User and admin login validation
User – Closet Tab	Adding, viewing, and deleting clothing items
User – Outfits Tab	Creating and updating outfits
User – Events Tab	Scheduling events and linking outfits
User – Complaints Tab	Submitting and reviewing user complaints
User – Account Tab	Updating user credentials
Admin – Manage Users Tab	Viewing and deleting user accounts
Admin – Complaints Tab	Viewing, resolving, and deleting complaints
Admin – Update Account Tab	Updating admin credentials

Test Case Structure

Each test case in the annex includes:

1. Test Case ID - A unique identifier for tracking the test
2. Test Case description – A summary of the feature or action being tested
3. Test Data– Input values used in the test (e.g., login credentials, outfit names)
4. Pre-Conditions – Required state before the test (e.g., user must be logged in)
5. Test steps– A step-by-step outline of user actions

6. Expected Test Result – What the application should do if the feature works correctly
7. Actual Test Result – The observed behavior during the test
8. Status Pass/Fail – Whether the result met expectations

Test sheets are clearly separated by tab or feature name (e.g., "Main Page (Events Tab)", "Admin (Complaints)").

Results and Findings

All critical features of GlamLink were successfully validated. The following conclusions were drawn from the test execution:

- All major functions passed without failure, including login, CRUD operations, and navigation between tabs.
- Data integrity was confirmed: records in the database accurately reflected user actions.
- Edge cases (e.g., empty inputs, duplicate values) were handled gracefully with user feedback and appropriate error messages.
- Minor UI issues (e.g., spacing, text alignment) were noted but did not affect functionality or performance.

Annex Reference

A full list of test cases, organized by feature and tab, is included in:

Annex - GlamLink Test Case Sheet (Excel) GlamLink Test Case.xlsx

This document contains detailed entries for each test, including expected and actual results, ensuring traceability and coverage of all application functionality.

5. Conclusion

The GlamLink project was a comprehensive and rewarding development experience that combined technical skills, teamwork, and creativity. Over the course of the project, we designed and implemented a fully functional fashion planning application that allows users to manage their clothing items, create personalized outfits, and link them to scheduled events—all within an intuitive interface.

Future possible improvements of our application

- **AI-Based Outfit Suggestions -** based on weather, events, and user preferences
- **Weather-Aware Planning -** using real-time forecast data
- **Calendar Syncing -** with platforms like Google Calendar or Outlook
- **Outfit History & Statistics -** showing frequently or rarely worn items
- **Event alerts -** when deleting a clothing item linked to an upcoming outfit.