Automatic Code Translation (A.C.T)

JID 4310

Mekhi Carter, Carla Sabala, Margaret Song, Isaac Kim, Minjun Kim

CLIENT: KIRA BOWDEN OF ROBINS AFB ENGINEERING GROUP

REPOSITORY: HTTPS://GITHUB.COM/CARLAMSABALA/ACT-JID-4310

Table of Contents

List of Figures

Terminology

Term	Definition	Context
API	Application Programming Interface: a set of functions and protocols that allow applications to communicate with one another.	Used to handle communication between the frontend and backend of the ACT application.
Authentication	The process of verifying a user's identity through a trusted identity provider.	Handled via Authentik and OpenID Connect (OIDC) for secure login and session control.
Authorization	The process of determining user permissions for accessing specific features or data.	Role-based access is managed through Authentik.
CouchDB	A NoSQL document database used to store JSON-formatted data.	Stores user translation history, including Delphi and C# code.
Delphi	A legacy programming language commonly used in older enterprise software systems.	Translated to modern C# by the ACT system.
Docker	A platform used to containerize and run applications in isolated environments.	Used to run the ACT system's components consistently across all development and deployment setups.
FAISS	Facebook AI Similarity Search: a library for efficient similarity search of dense vectors.	Used to retrieve similar Delphi examples for improving translation accuracy in the RAG pipeline.

Frontend	The graphical user interface that users interact with, built using HTML, CSS, and JavaScript.	Includes interfaces for login, code input/output, and translation history.
JSON	JavaScript Object Notation: a lightweight, human-readable format for structuring data.	Used for data exchange between frontend, backend, and the database.
Node.js	A JavaScript runtime used for building scalable server-side applications.	Powers the ACT backend server logic and API endpoints.
OpenAI API	A hosted API that provides access to various large language models for natural language and code generation tasks.	Used to translate Delphi code to C# using GPT-4o-mini.
OpenID Connect (OIDC)	An identity layer on top of OAuth 2.0 used for secure user authentication and session management.	Integrated via Authentik for user authentication.
RAG (Retrieval-Augmented Generation)	A method of enhancing LLM outputs by providing them with relevant, pre-retrieved context.	Used to boost translation accuracy by combining similar examples with GPT-based translation. [More Here]
Static File Hosting	Serving HTML/CSS/JS and other assets from a defined directory without dynamic rendering.	The ACT frontend is hosted using Node.js's static file serving capabilities.
System Architecture	The high-level structure and flow of information within a software system.	Encompasses the frontend, backend, CouchDB, Authentik, and Docker container interactions.

Translation Record	A stored entry containing the user's Delphi input code and the system's C# translation output.	Managed within CouchDB and retrievable by the user for later reference.
UML (Unified Modeling Language)	A standardized visual language for modeling the structure and behavior of systems.	Used in design documentation to show component relationships and interaction flows.

Introduction

Background

The Automated Code Translation (ACT) project is developed to assist software engineers at Robins AFB-402 Software Engineering Group in modernizing legacy systems. Specifically, it focuses on translating Delphi code into C#, enabling developers to integrate older systems with contemporary technology seamlessly. The project addresses the challenges of manual translation, which is prone to human error, time-consuming, and often inconsistent. By automating the process, ACT aims to streamline workflows, ensure accuracy, and improve maintainability of modernized systems. The design integrates user authentication, translation storage, and a secure and scalable architecture, leveraging technologies such as Docker for containerization, PostgreSQL for user credentials, CouchDB to maintain user translation history, and Node.js for development.

Document Summary

System Architecture: Provides a high-level overview of the application's static components and their interactions, along with a dynamic sequence diagram for runtime behavior.

Data Storage Design: Describes the schema and storage mechanisms, focusing on the MySQL database structure for translations and user authentication data.

Component Design: Breaks down the application into frontend and backend components, detailing their static class relationships and dynamic interactions.

UI Design: Walks through the user flow and demonstrates the different heuristics considered to provide an aesthetic and efficient user interface for our clients.

System Architecture

Introduction

Our application, ACT Translation Tool, is a cross-platform desktop application developed using the Electron framework. It is designed with a layered architecture, which separates concerns between the frontend (user interface), backend (business logic), and database (data storage). This architectural choice ensures modularity, scalability, and ease of maintenance, allowing individual layers to be updated or replaced independently. This is particularly important since the project may be handed off to other engineers after this course, making clarity and reusability essential.

In the following section, we present a high-level overview of our system. We use two diagrams to illustrate the architecture:

1. **Static Diagram** (Figure 1): This diagram showcases the major components of the system and their logical relationships. While it does not depict specific runtime scenarios, it provides a clear understanding of how the parts of the system interact.
2. **Dynamic Diagram** (Figure 2): This diagram illustrates the runtime behavior of the system during a specific feature execution. It provides a realistic example of how components interact to handle tasks like storing translations or processing user requests.

Our system architecture can be broken into the following major components:

- **Frontend**: The graphical user interface (GUI) built using HTML, CSS, and JavaScript. It is served as a static web application through an Express.js server. The interface includes views for login, translation, and translation history. Syntax-highlighted code editing is enabled through the CodeMirror library. Users can paste or upload Delphi code for translation, view translated C# output, and interact with previously saved translations through a file drive view.

- **Backend**: The Node.js-powered logic layer responsible for handling authentication, translation requests, and database interactions. It communicates with OpenAI's API to generate C# translations using a retrieval-augmented prompt strategy. It also provides REST endpoints for saving, retrieving, updating, and deleting translation records from the database.
- **Database**:
  A CouchDB database is used to store translation history, including the original Delphi code, generated C# code, timestamps, and titles. Authentication and user session management are handled separately through an integrated Authentik instance backed by PostgreSQL. PostgreSQL also stores authentication metadata and service configuration for the identity provider.

By organizing the system into these layers, we ensure clear separation of concerns, which facilitates concurrent development and simplifies debugging and future enhancements. The subsequent diagrams provide a detailed view of how these components interact statically and dynamically.

Goals and Objectives

The ACT system aims to achieve the following objectives:

1. Automate Code Translation: Efficiently convert Delphi code into C# while preserving the intended functionality and structure.
2. Enhance Developer Efficiency: Minimize the time and effort required for code translation, enabling developers to focus on higher-priority tasks.
3. Ensure Accuracy and Consistency: Eliminate human errors in the translation process and maintain consistent code quality across the system.
4. Facilitate Seamless Integration: Enable legacy systems to integrate with contemporary technology stacks using translated C# code.
5. Support User Management: Provide user-friendly authentication and account management for a secure and personalized experience.
6. Enable Translation History Tracking: Store translation logs securely to allow users to reference and retrieve prior translations when needed.
7. Provide Scalability and Security: Design the system to accommodate increasing user demand while adhering to security standards.

Architecture Rationale

The architectural choices for ACT were driven by the need for modularity, scalability, and security while keeping the system user-friendly and maintainable.

1. Frontend: HTML, CSS, JavaScript
    - The user interface is built using standard web technologies (HTML, CSS, JavaScript) and served statically through an Express server. CodeMirror is used to provide a robust, interactive code editing experience with syntax highlighting and editing capabilities. This allows users to write, view, and edit Delphi and C# code efficiently.

- This choice promotes broad browser compatibility and eliminates the need for platform-specific deployment tools like Electron, resulting in a lighter and more portable application.

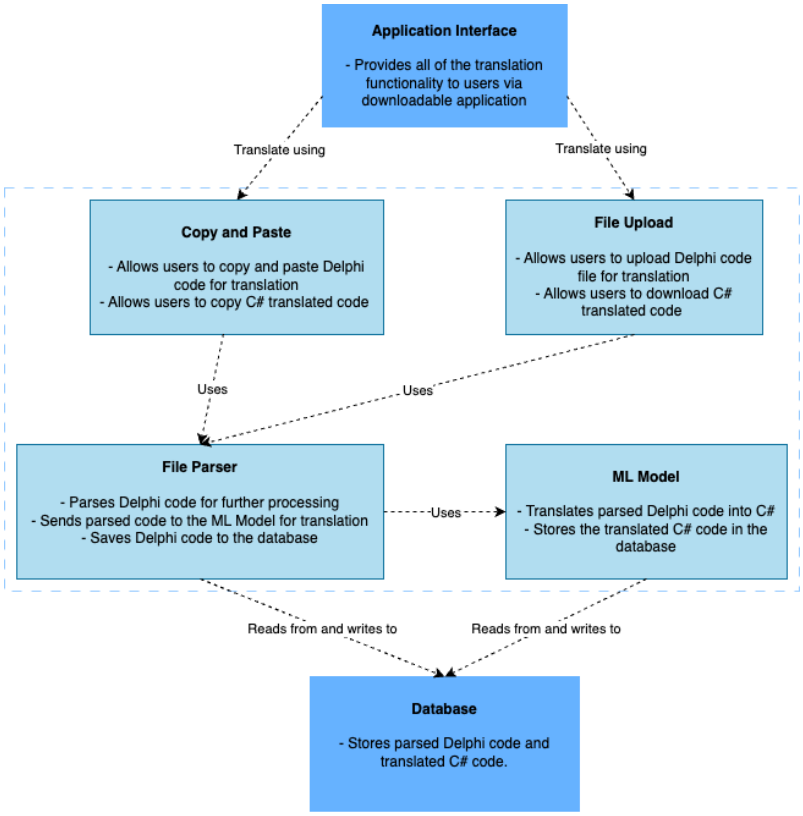
2. Backend: Node.js + Express
    - The backend uses Node.js with Express.js to create a lightweight and asynchronous web server. This enables:
    - Efficient handling of multiple concurrent translation requests.
    - Integration with OpenAI's API for translation generation using a retrieval-augmented generation (RAG) technique.
    - Management of user sessions and secure communication with the frontend and databases.
    - Modular API design that allows the frontend to access translation, authentication, and storage services without coupling.
3. CouchDB and PostgreSQL for Data Storage
    - CouchDB offers a robust and secure database system for storing user translation records without sacrificing scalability.
    - PostgreSQL (through Authentik) offers privacy as developers can set up specific rules for data access, which allows for better control and privacy over the user credentials and authentication tokens stored in this database.
4. Security Considerations
    - OIDC Authentication: Secure, token-based login system via Authentik using the OIDC protocol.
    - Transport Security: All traffic between the frontend, backend, and Authentik services is encrypted using HTTPS and TLS.
    - Database Security:
        - PostgreSQL uses role-based access control for Authentik credentials.
        - CouchDB access is restricted using credentials and runs within Docker.
    - Input Validation: Frontend and backend validations are in place to ensure inputs are sanitized before processing or saving.

- Error Handling: Consistent error logging and feedback to the user allow for faster debugging and enhanced user experience.
- Environment Isolation: Services are containerized using Docker, preventing unintended exposure of local configurations or services.

5. Separation of Concerns
    - The layered architecture ensures that the frontend, backend, and database operate independently, making it easier to debug, maintain, and scale specific components without disrupting the entire system.

Static System Architecture

Diagram

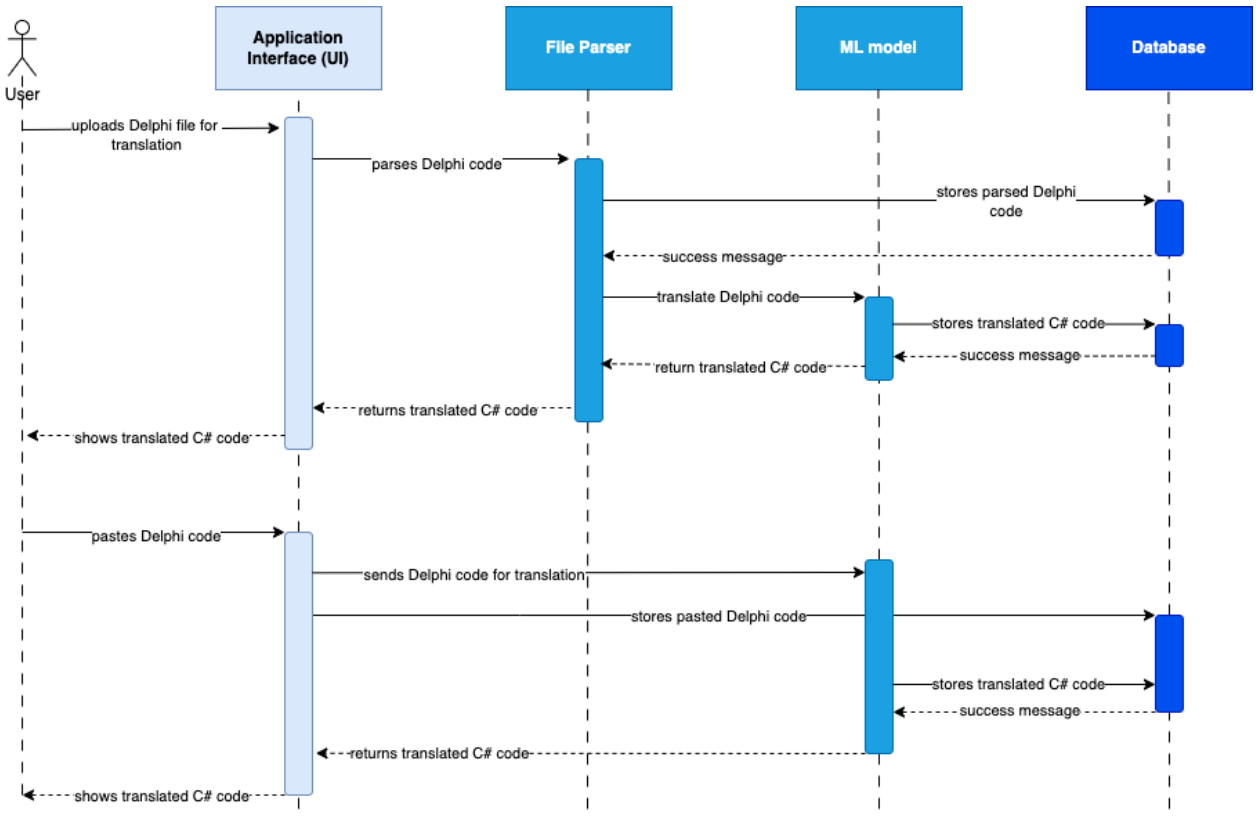


(Figure 1 – Static System Architecture showing the major components of the system and their logical relationships.)

We chose a Component Diagram to represent the structure of the system, focusing on the organization of major components such as the Application Interface, File Parser, ML Model, and Database. This diagram effectively showcases the dependencies between the components, providing a simple visual of how the different parts of the system rely on each other. The diagram further prioritizes the logical relationships of our main feature, translating Delphi code to C#, over runtime behavior.

Dynamic System Architecture

Diagram



(Figure 2 - Dynamic System Design demonstrating runtime behavior of code translation)

We specifically chose a System Sequence Diagram (SSD) to be able to show the different components of our system and how they interact with each other with a given use case. This diagram also captures the user-system interactions clearly. The feature I chose,

code translation (both copy and paste and file upload options to include the file parser component), is the main use of our application, therefore this would be the best feature to showcase with the SSD, to cover all the interactions and components possibly involved.

Data Storage Design

Introduction

The ACT project modernizes legacy **Delphi code** by translating it into **C#**. To support this process, the system is containerized and uses dedicated storage systems for different types of data:

- **CouchDB** – A NoSQL document store for **user translation data**. Allows for scalability and unstructured data such as the translation text and files.
- **PostgreSQL** – Managed by **Authentik**, responsible for **user authentication and storage**. Allows for privacy and security through developer settings.
- **Static Assets** – Images, stylesheets, and scripts are stored as files and served over **HTTP**.
- **Data Exchange** – System components communicate using **JSON**.

Database Use

CouchDB for User Translation (NoSQL Document Store)

- **Purpose:** Stores translation documents along with metadata.
- **Document Structure Example:**

```
Json CopyEdit
{
  "_id": "translation:unique-id",
  "user_email": "user@example.com",
  "delphi_code": "begin ... end;",
  "csharp_code": "public void Method() { ... }",
  "timestamp": "2025-03-04T12:00:00Z"
}
```

- **Relationships:**

- Each translation document includes a user_email field linking it to the corresponding user.
- As **CouchDB** is **schema-less**, these relationships are managed at the application level.

PostgreSQL via Authentik for User Information (Relational Database)

- **Purpose:** Securely manages user authentication and stores user information.
- **Entity-Relationship:** Authentik utilizes **PostgreSQL** to store user details.
- **Simplified Users Table Structure:**

Column	Type	Attributes
user_id	UUID	Primary Key
email	VARCHAR(255)	Unique
password_hash	TEXT	Secure Hash
metadata	JSONB	Additional user info

File Use

Static Files

- **Purpose:** Static assets (e.g., images, CSS, JavaScript) are stored in designated directories.
- **Directory Structure:**
  - src/assets/Images/ – Stores **PNG, SVG** image assets.
  - src/assets/stylesheets/ – Contains **CSS** files.
  - src/pages/ – Holds **HTML** files (e.g., index.html, translation.html).
- **Documentation:**
  - Any **non-standard file formats** should be documented separately.
  - Otherwise, reference **standard formats** (HTML, CSS, JS, JSON).

Data Exchange

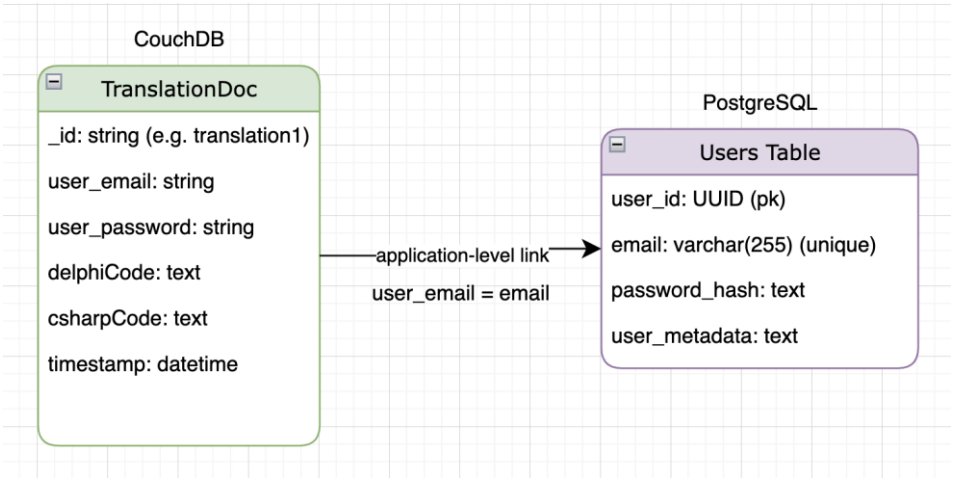
Format and Protocols

- **Data Format:**
  - **JSON** is used for API responses and token exchanges.
  - **Static files** are served in standard web formats (HTML, CSS, JS).
- **Protocols:**
  - API endpoints communicate via **HTTP**.
  - Authentication follows **OAuth 2.0 / OpenID Connect (OIDC)** protocols.

Security Considerations

- **Data Encryption:**
  - In **production**, all communication must occur over **HTTPS** to protect data in transit.
- **Authentication:**
  - User authentication is managed via **Authentik**, utilizing **OIDC**.
  - **JWT (JSON Web Tokens)** are used for secure session management.
  - **User credentials are never stored in plain text**.
- **Privacy & Data Exchange Security:**
  - **Personally Identifiable Information (PII)** is handled securely using **encrypted channels** and **strict access controls**.
  - **All API interactions** adhere to best practices for **data integrity and confidentiality**.

Document Model Diagram



(Figure 3 - Document Model Diagram (JSON schema) demonstrating both databases)

This combined diagram was chosen to represent our mixed database approach, where CouchDB stores flexible, JSON-based translation documents and PostgreSQL securely manages user authentication data. By using a NoSQL document model for the translation data alongside a SQL table for user information, the diagram logically organizes our system and demonstrates the application-level relationship: the user_email field in each translation document links to the user email recorded in PostgreSQL. This approach highlights the benefits of using each database for its intended purpose while showing the interaction between the two and how they interconnect and store data within the overall system.

Summary

- **CouchDB** stores **translation documents** in a flexible **JSON format**, linking them to users via metadata.
- **PostgreSQL (via Authentik)** manages **user authentication** and stores structured user data.
- **Static files** (HTML, CSS, JS, images) are stored in organized directories and served directly.

- **Data exchange** occurs using **JSON over secure HTTP/HTTPS** connections, ensuring **data integrity**.
- **Security Measures** include **JWT authentication, HTTPS enforcement, and encryption** for sensitive data.

This system design ensures both **flexibility** in data storage (**NoSQL for translations**) and **strong security & integrity** for user data (**PostgreSQL and Authentik**).

Component Design

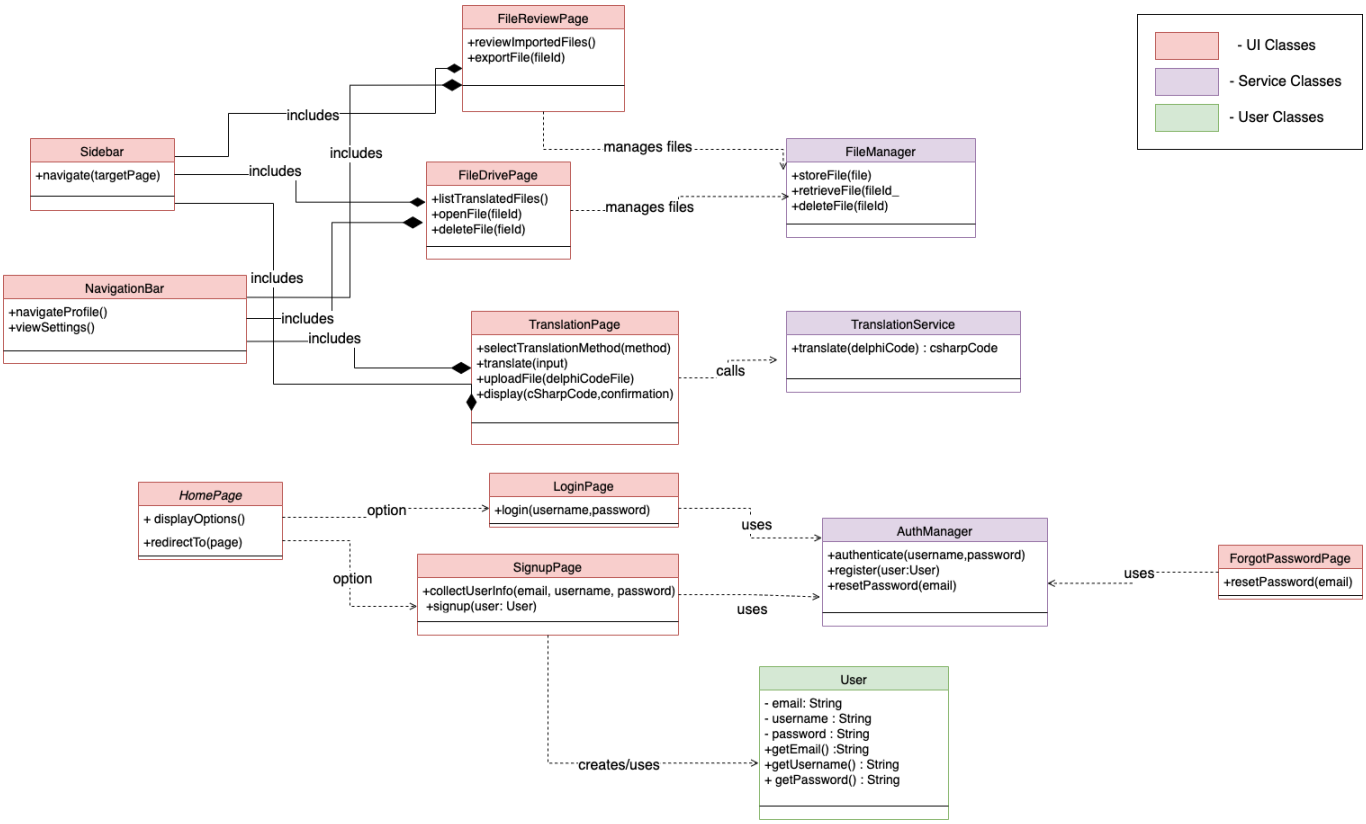
Introduction

This document presents the component design for our system, which manages user authentication, file handling, and code translation workflows. The design is illustrated with two complementary UML diagrams:

1. A **Static Component Diagram**, showing how classes and components are organized and how they relate at design time.
2. A **Dynamic (Sequence) Diagram**, showing how those same components interact at runtime to fulfill user requests.

Together, these diagrams ensure **conceptual integrity** by connecting the overall architecture (static view) with the actual process flow (dynamic view). They clarify which components handle authentication, file management, translation services, and user interface interactions, providing a picture of the system's structure and behavior.

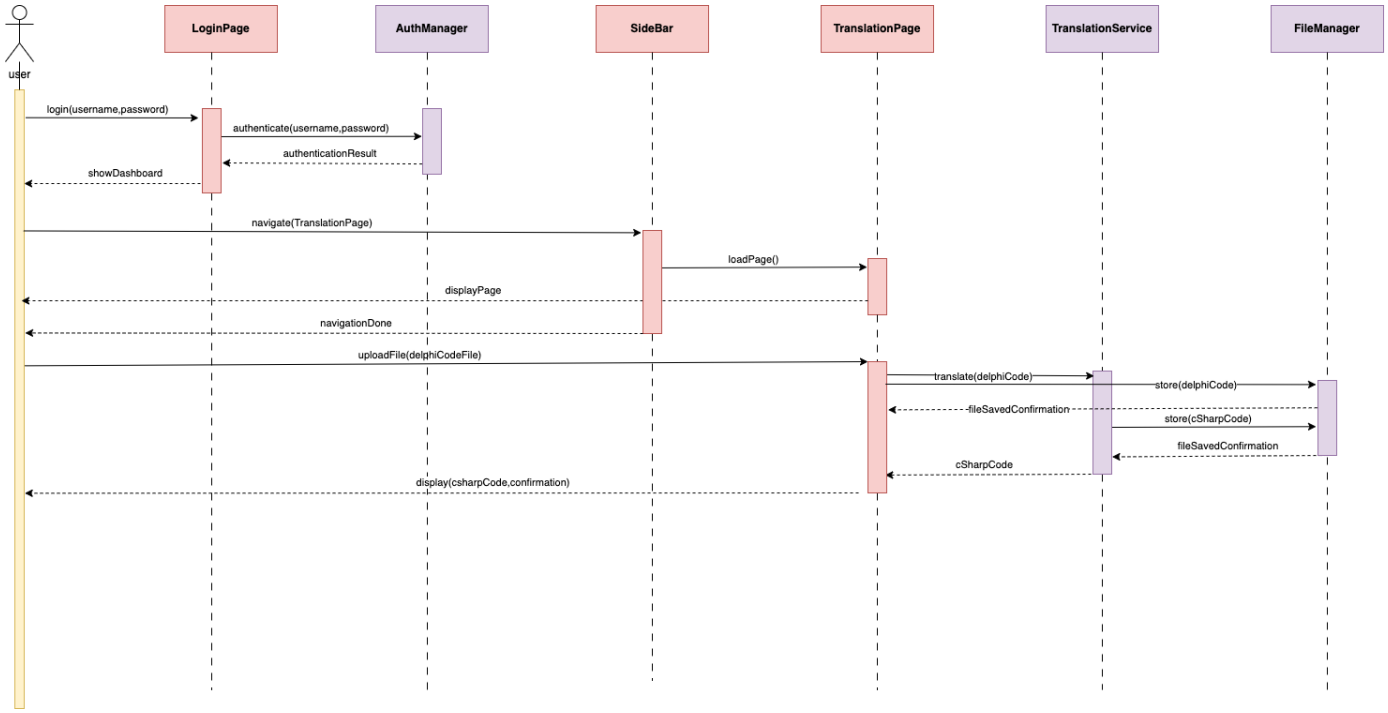
Static Elements



(Figure 4 – Static Class Diagram)

The static component diagram shows how the system's classes and components relate at design time, emphasizing structure over behavior. Each component is color-coded according to its role (for example, UI, service, or data), making it easy to distinguish which parts handle user interactions, which perform core logic, and which store information. You can see each component's attributes, methods, and relationships, illustrating how data and functionality are organized. These relationships include associations such as "includes," "uses," or "creates," clarifying the high-level architecture.

Dynamic Elements



(Figure 5 – Dynamic Component System Sequence Diagram)

The dynamic (sequence) diagram captures how those same components from the static diagram work together to fulfill user requests at runtime. This diagram traces the flow of messages—such as login requests, file uploads, or translation calls—through various components like AuthManager, FileManager, or TranslationService.

UI Design

Introduction

This portion of the document will demonstrate a walkthrough of our service. The goal of this document is to show how users will interact with our application, guiding them through key screens and features.

When users open our app and login, they'll be able to access three key pages: their home, translator, and file drive pages. The UI walkthrough will go in depth about these main pages that the user interacts with, and heuristics associated with each of the screens.

UI Walkthrough

Before Translation

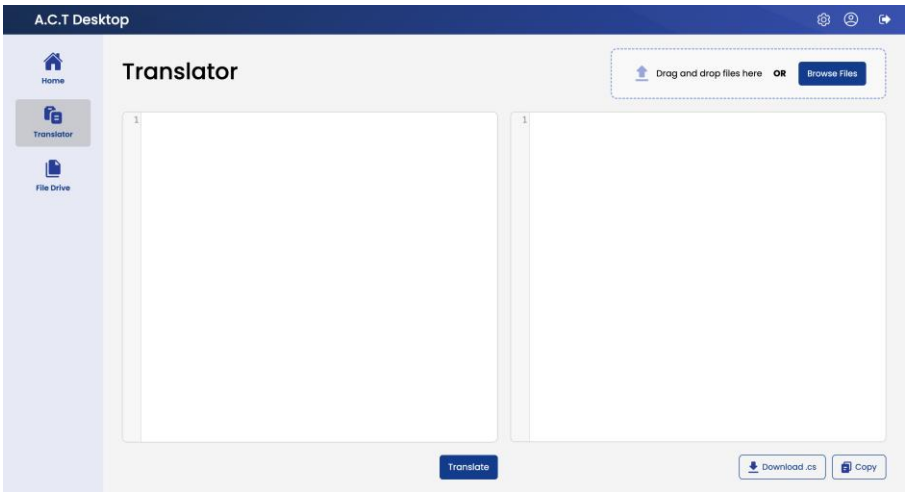


Figure 6. Translator Empty Page

After Translation

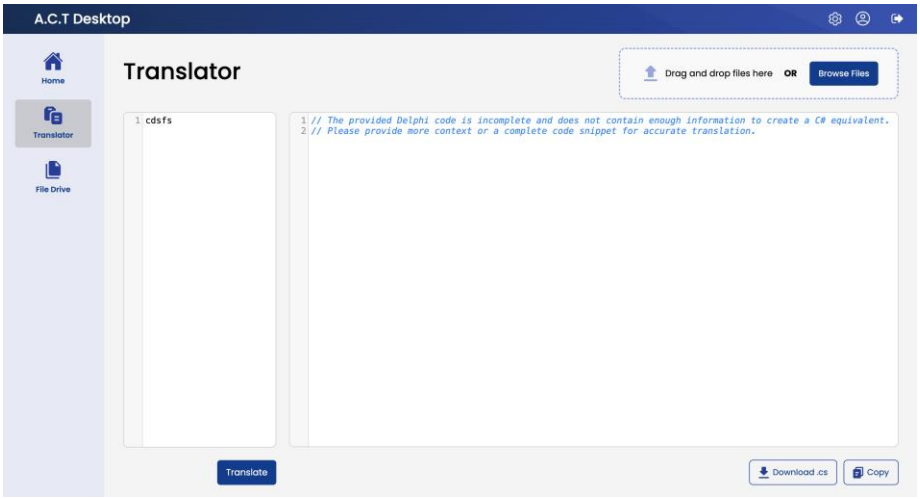


Figure 7. Translator Error Page

- One of the main features of these screens is allowing users to directly start translating their Delphi code using our copy and paste features or by uploading files. Users can also navigate to other pages such as their Home and File Drive with all their previously translated files and text. Users also get an error message if they do not provide valid Delphi code.
    - Out of the 10 usability heuristics, we have applied the “match between system and the real world”, “aesthetic and minimalist design”, and “error prevention”.
        - **Match between System and the Real World**: The interface uses familiar concepts, such as “drag and drop” for file uploads, mirroring how users typically move files on their computer. This intuitive approach lets users translate code just as they would handle files in a standard desktop environment.
        - **Aesthetic and Minimalist Design**: Only essential elements—like the input box, output box, and key action buttons—are displayed. This uncluttered layout ensures users can focus on translating Delphi code without unnecessary distractions, helping them quickly identify where to paste or upload their code.
        - **Error Prevention**: The system provides an error message if users submit invalid Delphi code, preventing them from continuing with

invalid inputs. By showing an error message to users early on, the application helps them correct issues before proceeding, reducing wasted effort and frustration for later.

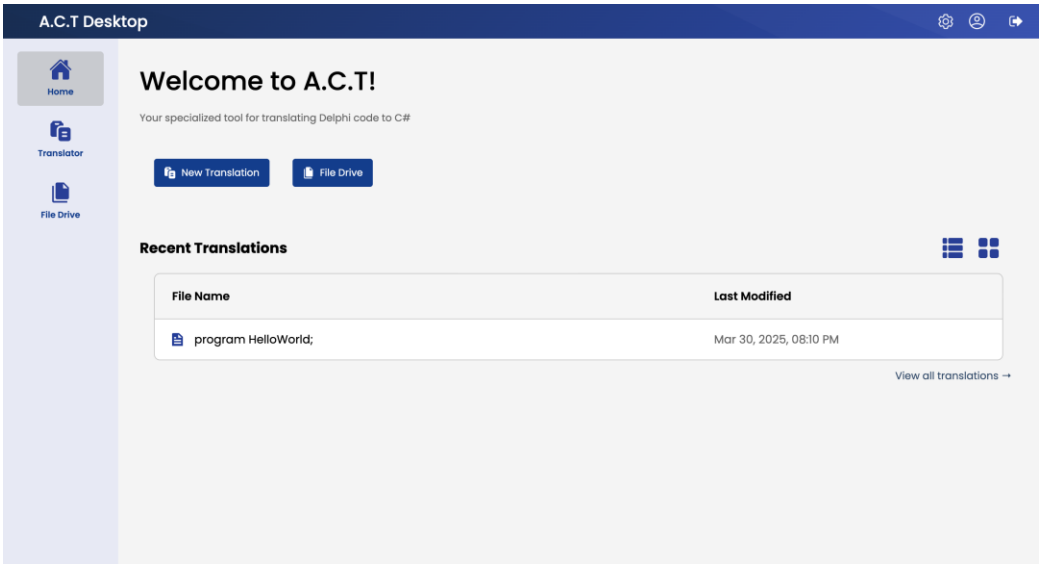
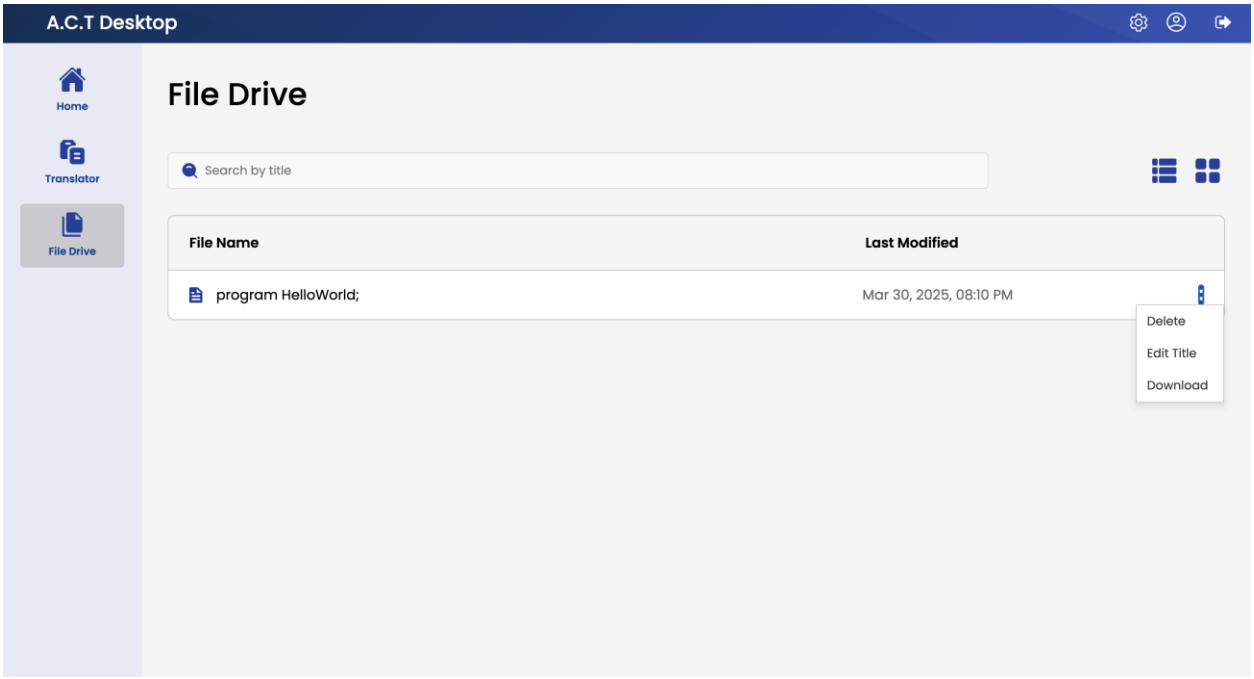


Figure 8. Home Page

- After users are authenticated using our client's authentication system (Authentik), users will be greeted with the Home Screen. The screen demonstrates recent translations and allows users to quickly access our other pages: "New Translation" to head to the Translator page and "File Drive" to head to the File Drive page.
    - Out of Nielsen's 10 usability heuristics, for this screen we have applied "*match between the system and real world*", "*consistency and standards*", and "*flexibility and efficiency of use*."
        - **Match between the System and Real World**: The Home Page uses familiar terminology and iconography that mirror everyday experiences. For example, labels like "Recent Translations" and buttons such as "New Translation" and "File Drive" reflect common actions, making the system intuitive. Also, the UI of the page resembles common UI of similar services like Google Docs, making it intuitive and easier for users to navigate.

- **Consistency and Standards**: The design elements (colors, layout, navigation icons) follow the same structure as the other pages, so users recognize where to find key features at a glance. Standard terminology (e.g., “File Drive,” “New Translation”) aligns with the rest of the application and helps maintain a uniform experience.
- **Flexibility and Efficiency of Use**: Users can quickly access both the Translator page and File Drive from the Home Page without extra steps, speeding up their workflow. The “Recent Translations” list also provides a convenient shortcut for revisiting or updating recent tasks, reducing the time needed to locate recently and frequently accessed files.



File 9. File Drive Page

- This page demonstrates the user’s File Drive. This will display all previous translated texts and files. Users can search through these previous translations and organize them in the way they want to: list versus grid view. With each of the previous translations, users also have the option to delete the translation from their

file drive, download the translation as a .cs file, and/or edit the title of the translation.

- Out of the 10 usability heuristics, this page incorporates “flexibility and efficiency of use”, “aesthetic and minimalist design”, and “user control and freedom.”
    - **Flexibility and Efficiency of Use**: The File Drive allows users to quickly search through previous translations and switch between list or grid view based on their preferences. This flexibility ensures they can locate and manage files more efficiently, tailoring the interface to their individual workflow. With the sidebar, users are also easily able to access other pages, making flexibility between different pages easier and more efficient.
    - **Aesthetic and Minimalist Design**: The interface presents only the most essential elements—such as the file name, last modified date, and key actions (delete, download, edit). This uncluttered layout keeps users focused on the core tasks and reduces cognitive load, aligning with minimalist design principles.
    - **User Control and Freedom**: Users are empowered to manage their files as they see fit—by editing the title, downloading the translation as a .cs file, or deleting it entirely. These options provide a sense of control and freedom, ensuring that users can correct mistakes or reorganize their files whenever necessary.

Appendix

GET /auth/login

Redirects to Authentik login flow via OpenID Connect.

GET /auth/callback

Callback URL after Authentik authentication.

GET /auth/logout

Logs the user out and redirects to the Authentik logout endpoint.

GET /auth/status

Returns whether the user is currently authenticated.

Response:

```
json

CopyEdit

{
  "authenticated": true,
  "user": {
    "email": "...",
    "name": "..."
  }
}
```

POST /translate

Translates Delphi Pascal code to C# using a Retrieval-Augmented Generation (RAG) approach powered by OpenAI.

- **Content-Type**: application/json
- **Authentication**: None
- **Request Body**:

```
json
CopyEdit
{
  "delphi_code": "procedure Greet; begin
Writeln('Hello, World!'); end;"
}
```

- **Response**:

```
json
CopyEdit
{
  "translated_csharp": "public static void Greet() {\n
Console.WriteLine(\"Hello, World!\");\n}"
}
```

Resource Information

- Method: POST
- Content-Type: application/json
- Auth: None
- Rate Limit: Depends on OpenAI plan that the client will use

Parameters

- **delphi_code**
    - The Delphi Pascal code to be translated
    - Type: String
    - Required

Response

- **translated_csharp**
    - The translated C# code generated by the model
    - Type: String
- **error**
    - If an error occurs, contains error message
    - Type: String

POST /save-translation

Stores a new translation record in the CouchDB database.

- **Request Body**:

```
json
CopyEdit
{
  "delphiCode": "...",
  "csharpCode": "...",
  "documentTitle": "..."
}
```

- **Response**:

```
json
CopyEdit
{
  "success": true,
  "id": "<doc_id>",
  "documentTitle": "..."
}
```

GET /get-translations

Returns all stored translations, sorted by timestamp.

Response:

```
json
CopyEdit
{
  "success": true,
  "translations": [
    {
      "_id": "...",
      "documentTitle": "...",
      "delphiCode": "...",
      "csharpCode": "...",
      "timestamp": "..."
    }
  ]
```

```
}
```

GET /get-translation/:id

Fetches a single translation document by its ID.

Response:

```
json
CopyEdit
{
  "success": true,
  "translation": {
    "_id": "...",
    "documentTitle": "...",
    "delphiCode": "...",
    "csharpCode": "...",
    "timestamp": "..."
  }
}
```

PUT /update-translation/:id

Updates the title of a stored translation.

Request Body:

```
json
CopyEdit
{
  "documentTitle": "New Title"
```

```
}
```

Response:

```
json
CopyEdit
{
  "success": true,
  "id": "...",
  "rev": "..."
}
```

DELETE /delete-translation/:id

Deletes a translation document using its ID and revision.

Response:

```
json
CopyEdit
{
  "success": true,
  "id": "..."
}
```

Team Members and Contact Information

Name	**Email**	**Contributions**
Carla Marie Sabala	carlamsabala@gmail.com	Front and Backend Developer
Minjun Kim	ckandrew04@gmail.com	ML Developer

Margaret Song	msong321@gatech.edu	UI/UX Developer
Mekhi Carter	mekhi@gatech.edu	Team and Software Lead
Isaac Kim	ikim319@gatech.edu	Front and Backend Developer