KIV/PIA-E

Semester Project Documentation

Author:

Kevin

January 16, 2026

Contents

1 Introduction

This document provides technical documentation of the **KIV/PIA-E Translation Management System** developed as a semester project.

The system supports an end-to-end workflow for handling customer translation requests, assigning translators, uploading translated files, collecting feedback, and managing project lifecycle states. The backend is implemented using **Python (Flask)** with a **REST API** and **MySQL** database.

This documentation focuses primarily on:

- REST API endpoints and example requests
- Authentication and authorization model
- Role-based access control (RBAC)
- Project lifecycle and state machine
- Database schema overview
- Common error responses and status codes

2 Project Setup and Execution

2.1 Prerequisites

- Docker + Docker Compose
- (Optional) Python environment for running tests locally

2.2 Environment Configuration

Create a `.env` file from `.env.example` and fill environment-specific values. Ensure `.env` is in `.gitignore`.

Google OAuth2 Configuration

If Google OAuth2 is enabled, configure:

- `GOOGLE_CLIENT_ID`
- `GOOGLE_CLIENT_SECRET`
- (If used) a Google client secret JSON file referenced via environment variable (recommended), or mounted as a Docker secret/volume.

Security note: never commit real secrets into Git. Keep placeholder values in `.env.example`.

2.3 Run with Docker Compose

```
docker compose up --build
```

Stop services:

```
docker compose down
```

2.4 Run Automated Tests

On host:

```
pytest -q
```

Inside a running container:

```
docker compose exec <SERVICE_NAME> pytest -q
```

3 System Overview

The system supports three types of users:

- **Customer** – creates translation projects, downloads results, and provides approval/rejection feedback
- **Translator** – translates assigned projects and uploads (or re-uploads) translated files
- **Administrator** – oversees the workflow (assignment and lifecycle control), and may notify customers/translators (optional)

The project workflow consists of:

1. Project creation by customer
2. Automatic translator assignment based on target language (if available)
3. Translator uploads translated file
4. Customer reviews the translation and approves or rejects it
5. If rejected, translator re-uploads a corrected translation
6. Administrator closes the project (and may notify customers/translators)

4 Authentication and Authorization

4.1 Authentication

Sensitive routes are protected using session-based authentication.

4.2 Authorization (RBAC)

Role checks are enforced via decorators.

Example:

```
@login_required
@require_role('ADMIN')
def administrator_dashboard():
    ...
```

4.3 UI vs API behavior

Unauthorized access is handled differently depending on route type:

- **UI pages:** redirect to login page (browser-friendly).
- **API endpoints:** return JSON with appropriate HTTP status codes (client-friendly).

5 Project State Machine

Projects move through a predefined lifecycle.

5.1 States

- **CREATED** – customer submitted a new project (original file uploaded)
- **ASSIGNED** – translator assigned automatically by language
- **COMPLETED** – translator uploaded the translated file
- **APPROVED** – customer approved the translation
- **REJECTED** – customer rejected the translation; translator must re-upload a corrected version
- **CLOSED** – project closed by administrator (final state)

6 Database Schema Overview

The application uses a MySQL database (`pia_db`) with the following tables: `Users`, `Languages`, `Projects`, and `Feedbacks`. All identifiers are stored as `char(36)` UUID strings.

6.1 Users

Stores registered user accounts.

- **id** (`char(36)`, PK) – UUID of the user
- **name** (`varchar(255)`) – display/name identifier
- **email** (`varchar(255)`, UNIQUE) – unique email address
- **password** (`varchar(255)`) – hashed password
- **role** (`enum('CUSTOMER','TRANSLATOR','ADMINISTRATOR')`) – user role
- **created_at** (`datetime`, default `CURRENT_TIMESTAMP`) – creation timestamp

6.2 Languages

Defines which languages a user (typically a translator) supports. This is a many-to-many style mapping represented as one row per user-language pair.

- **user_id** (`char(36)`, FK → `Users.id`) – user UUID
- **language** (`char(2)`) – language code (e.g., `en`, `de`)

Primary key: composite (`user_id`, `language`) to prevent duplicates.

6.3 Projects

Stores translation projects and their lifecycle state. Files are stored as paths/filenames (not BLOBs).

- **id** (`char(36)`, PK) – project UUID
- **customerId** (`char(36)`, FK → `Users.id`) – project owner (customer)
- **name** (`varchar(255)`) – project name
- **description** (`text`) – project description

- **translatorId** (`char(36)`, FK → `Users.id`, nullable) – assigned translator
- **languageCode** (`char(2)`) – target language code
- **originalFile** (`varchar(255)`, nullable) – stored path/filename of uploaded source file
- **translatedFile** (`varchar(255)`, nullable) – stored path/filename of translated output
- **state** (`enum('CREATED','ASSIGNED','COMPLETED','APPROVED','REJECTED','CLOSED')`, default `'CREATED'`) – project lifecycle state
- **createdAt** (`datetime`, default `CURRENT_TIMESTAMP`) – creation timestamp

6.4 Feedbacks

Stores customer feedback per project. The design enforces **one feedback record per project**.

- **projectId** (`char(36)`, PK, FK → `Projects.id`) – project UUID
- **text** (`text`) – feedback text
- **createdAt** (`datetime`, default `CURRENT_TIMESTAMP`) – timestamp

6.5 Relationships and Constraints

- **Projects.customerId → Users.id**
  ON DELETE CASCADE, ON UPDATE CASCADE (deleting a customer deletes their projects)
- **Projects.translatorId → Users.id**
  ON DELETE SET NULL, ON UPDATE CASCADE (deleting a translator unassigns the project)
- **Languages.user_id → Users.id**
  ON DELETE CASCADE, ON UPDATE CASCADE (deleting a user deletes their language mappings)
- **Feedbacks.projectId → Projects.id**
  ON DELETE CASCADE, ON UPDATE CASCADE (deleting a project deletes its feedback)

6.6 Indexes

- **Users**: PK on `id`, UNIQUE index on `email`
- **Languages**: composite PK on (`user_id`, `language`)
- **Projects**: PK on `id`, indexes on `customerId` and `translatorId`
- **Feedbacks**: PK on `projectId`

7 REST API Documentation

All endpoints are prefixed with `/api` unless noted otherwise. The API uses a mix of JSON payloads (primarily for user/auth operations) and multipart form-data (for file uploads). Authentication is session-based; some endpoints may use the authenticated session user if identifiers are not explicitly provided.

7.1 Users

POST /api/users

Creates a new user.

Request body: JSON

- `name` (string, required)
- `email` (string, required)
- `password` (string, required)
- `role` (string, required) – `CUSTOMER` — `TRANSLATOR` — `ADMINISTRATOR`
- `languages` (array of strings, optional) – defaults to `[]`

```
curl.exe -X POST http://localhost:5000/api/users ^
  -H "Content-Type: application/json" ^
  -d "{\"name\":\"alice\",\"email\":\"alice@example.com\",\"password
     \":\"Secret123\",\"role\":\"CUSTOMER\",\"languages\":[\"en
     \",\"de\"]}"
```

GET /api/users

Returns all users.

```
curl.exe -X GET http://localhost:5000/api/users
```

GET /api/users/¡name¿

Returns a single user by name.

```
curl.exe -X GET http://localhost:5000/api/users/alice
```

GET /api/customer

Returns the customer UI/endpoint (role-protected).

GET /api/translator

Returns the translator UI/endpoint (role-protected).

GET /api/administrator

Returns the administrator UI/endpoint (role-protected).

7.2 Projects

POST /api/projects

Creates a new translation project.
Request type: multipart/form-data (file upload)
Form fields:

- `customer_id` (UUID string, optional) – if missing, backend uses `session.user.user_id`
- `project_name` (string, required)
- `description` (string, required)
- `language` (string, required) – target language code (e.g., `en`)
- `source_file` (file, required)

```
curl.exe -X POST http://localhost:5000/api/projects ^
  -F "customer_id=<CUSTOMER_UUID>" ^
  -F "project_name=Sample project" ^
  -F "description=Test upload project" ^
  -F "language=en" ^
  -F "source_file=@C:\path\to\test.txt"
```

GET /api/projects

Returns projects (scope depends on role/implementation; commonly ADMIN gets all, CUSTOMER gets own).

```
curl.exe -X GET http://localhost:5000/api/projects
```

GET /api/projects/¡customer_id¿

Returns projects for a specific customer.

```
curl.exe -X GET http://localhost:5000/api/projects/<CUSTOMER_UUID>
```

GET /api/projects/¡project_id¿

Returns project details for a specific project.

```
curl.exe -X GET http://localhost:5000/api/projects/<PROJECT_UUID>
```

PUT /api/projects/¡project_id¿/status

Updates project status/state (state machine enforced).

```
curl.exe -X PUT http://localhost:5000/api/projects/<PROJECT_UUID>/
    status ^
  -H "Content-Type: application/json" ^
  -d "{\"state\":\"COMPLETED\"}"
```

PUT /api/projects/¡project_id¿/assign

Assigns a translator to the project.

```
curl.exe -X PUT http://localhost:5000/api/projects/<PROJECT_UUID>/
    assign ^
  -H "Content-Type: application/json" ^
  -d "{\"translator_id\":\"<TRANSLATOR_UUID>\"}"
```

POST /api/project/¡project_id¿/accept

Customer accepts/approves translation.

```
curl.exe -X POST http://localhost:5000/api/project/<PROJECT_UUID>/
   accept
```

POST /api/project/¡project_id¿/reject

Customer rejects translation.

```
curl.exe -X POST http://localhost:5000/api/project/<PROJECT_UUID>/
   reject ^
  -H "Content-Type: application/json" ^
  -d "{\"reason\":\"Incorrect terminology\"}"
```

POST /api/project/¡project_id¿/close

Administrator closes/archives the project.

```
curl.exe -X POST http://localhost:5000/api/project/<PROJECT_UUID>/
   close
```

GET /api/project/¡project_id¿/download/original

Downloads the original uploaded file.

```
curl.exe -X GET http://localhost:5000/api/project/<PROJECT_UUID>/
   download/original
```

GET /api/project/¡project_id¿/download/translation

Downloads the translated file.

```
curl.exe -X GET http://localhost:5000/api/project/<PROJECT_UUID>/
   download/translation
```

POST /api/project/¡project_id¿/uploat

Translator uploads (or re-uploads) the translated file.
Request type: multipart/form-data
Form fields:

- `translated_file` (file, required)

```
curl.exe -X POST http://localhost:5000/api/project/<PROJECT_UUID>/
    uploat ^
  -F "translated_file=@C:\path\to\translation.txt"
```

7.3 Email

POST /api/email/respond

Sends an email response/notification (if enabled).

```
curl.exe -X POST http://localhost:5000/api/email/respond ^
  -H "Content-Type: application/json" ^
  -d "{\"to\":\"alice@example.com\",\"subject\":\"Update\",\"message
      \":\"Your project was updated.\"}"
```

7.4 Authentication (Auth Blueprint)

Authentication endpoints are provided under the `/auth/api` prefix.

POST /auth/api/register

Registers a new user (if enabled).

```
curl.exe -X POST http://localhost:5000/auth/api/register ^
  -H "Content-Type: application/json" ^
  -d "{\"name\":\"alice\",\"email\":\"alice@example.com\",\"password
      \":\"Secret123\",\"role\":\"CUSTOMER\"}"
```

POST /auth/api/login

Logs in a user and creates a session.

```
curl.exe -X POST http://localhost:5000/auth/api/login ^
  -H "Content-Type: application/json" ^
  -d "{\"name\":\"alice\",\"password\":\"Secret123\"}"
```

POST /auth/api/logout

Logs out the current user (session invalidation).

```
curl.exe -X POST http://localhost:5000/auth/api/logout
```

8 Error Handling

Common error responses returned by API endpoints:

400 Bad Request

Invalid input, missing required fields, invalid state transition.

```
{
  "error": "Invalid request payload."
}
```

401 Unauthorized

Authentication required.

```
{
  "error": "Authentication required."
}
```

403 Forbidden

User authenticated but lacks permissions.

```
{
  "error": "Insufficient permissions."
}
```

404 Not Found

Resource not found.

```
{
  "error": "Resource not found."
}
```

500 Internal Server Error

Generic fallback for unexpected issues.

9 Conclusion

The semester project implements a complete workflow for handling translation requests with database persistence, RBAC, and session-based secured access. The modular REST API and clear state machine enable maintainability and further extension.