Programmer's Guide for LSFB Project

Documentation of the Francophone Belgian Sign Language Project

Loïc Franck KAMGAIN MEUPIAP

INFO B302 - Individual Project (2024 - 2025)
Faculty of Computer Science, University of Namur, Belgium

Contents

1 Preface

The **Programmer's Guide** aims to provide detailed information about the LSFB (French Sign Language of Belgium) project, to guide any developer who wishes to participate in its development or improvement. This guide focuses on the software architecture, the technical choices made, and the different ways to effectively extend or maintain the project.

2 Introduction

2.1 Purpose of the Guide

This guide aims to provide developers with a **clear starting point** for further development of the **LSFB project**. It provides a detailed overview of the project's structure, its main components, and the reasons behind certain architectural choices. The goal is to allow a new developer to quickly understand the project's architecture, business logic, and how the backend and frontend interact with the database.

2.2 Target audience

The guide is primarily intended for software developers who want to work on this project, especially those with a good grasp of Python, **FastAPI**, **SQLAlchemy**, and **JavaScript**. It is also useful for front-end developers with **HTML**, **CSS (TailwindCSS)**, and **JavaScript skills.**

3 Where should I start if I were to continue developing this software?

3.1 Where are the sources?

The project sources are available on **GitHub**. To clone the repository and get the source code, you can run the following command:

```
git clone https://github.com/UNamurCSFaculty/2425_INFOB318_LSD_01
```

This will give you access to all back-end (FastAPI, Python) and front-end (HTML, JavaScript) code.

3.2 Where is the documentation?

The complete project documentation is divided into two parts:

- **User's Guide** : This guide is for end users and explains how to install and use the application.

- **Programmer's Guide** : This guide is intended for developers and explains the software architecture, how the code is structured, as well as the technical choices made.

The full documentation can be found in the `docs/` directory of the GitHub repository.

3.3 Who are the developers?

The project development was mainly carried out by **Loïc Franck KAMGAIN MEUPIAP**, a student in Block 3 Computer Science at the University of Namur, as the main developer. I was accompanied by assistant **Jérôme Fink**, who provided advice on the database design and proposals on best practices for managing annotations and videos. External contributions mainly focused on the database architecture and query optimization processes.

3.4 Where are the specifications?

There are no **formal specifications** written for this project. The project specifications were developed through **several meetings with the client**. Expectations and needs were discussed during these meetings, and the specifications were extracted directly from these discussions with the client.

Screenshots of the emails and meetings detailing these discussions are available in the docs/ directory of the GitHub repository. These screenshots contain essential information about the expected features, technical requirements, and specific constraints of the project.

3.5 Who are the sponsors?

The **LSFB (French Sign Language of Belgium)** project was sponsored by several stakeholders, each with a specific role in the design and monitoring of the project. This project was primarily supported by the **University of Namur** and benefited from the expertise of several researchers and teachers specializing in the fields of sign language linguistics and computer science. The project aims to analyze Belgian French Sign Language through an interactive platform allowing the analysis and visualization of sign videos in order to offer researchers, students and linguistics professionals a powerful tool for the study of signs, annotated videos, and interactions in Belgian French Sign Language.

- **Academic background:** The University of Namur, through its departments, enabled the interdisciplinarity of the project. Sign language linguistics researchers provided detailed specifications on the types of data and annotations required, while computer science students took charge of developing the platform. This collaboration between linguists and computer scientists resulted in the creation of a comprehensive tool, integrating **databases, statistical analyses, and interactive data visualization**.

3.6 What was the development environment?

The development environment for the **LSFB** project was designed to ensure **optimal productivity** and **easy maintenance** of the code while meeting the specific requirements of the project. This section details the tools, technologies, and practices used throughout the development process.

3.6.1 Programming Languages and Technologies Used

The **LSFB** project mainly uses **Python** for backend development and **JavaScript** for frontend development, with additional tools and libraries to handle interfaces, database interactions, and video management. Here is an overview of the main technologies used:

Backend (Python)

- **Python 3.12+**: Python is the primary language used for backend development, specifically version **3.12** or later. Python was chosen for its **simplicity**, **readability**, and the extensive ecosystem of libraries and frameworks that facilitate rapid development of robust solutions.
- **FastAPI**: FastAPI was selected to handle the backend server of the project. It is a modern framework for building fast and efficient web APIs in Python. FastAPI offers exceptional **performance** due to its asynchronous architecture and efficient request handling. Here's why **FastAPI** was chosen:
  - **High Performance**: FastAPI is built on **Starlette** for managing requests and **Pydantic** for data validation, allowing performance close to **Node.js** and other JavaScript frameworks.
  - **Ease of Development**: FastAPI simplifies development with automatic data validation and native support for asynchronous requests.
  - **Automatic Documentation**: FastAPI automatically generates interactive documentation using **Swagger UI**, making it easier for developers to test APIs directly from the web interface.

Frontend (HTML, CSS, JavaScript)

- **HTML5** and **CSS3**: The user interface is built using **HTML5** for page structure and **CSS3** for styling.
- **TailwindCSS**: **TailwindCSS** is a CSS framework used to build responsive, modern interfaces. It allows rapid styling without writing custom CSS, using a utility-first approach.
  - **Why TailwindCSS?** It was chosen for its **flexibility** and ability to quickly build modular and responsive interfaces, reducing the need for custom CSS and ensuring a uniform design.
- **JavaScript**: **JavaScript** is used to manage dynamic interactions on the interface, such as user clicks, video interactions, etc.

- **Pyplot**: For plotting the skeleton data, we used PyPlot, a Python library that allows you to create 2D/3D graphs.
  - **Why Pyplot?** PyPlot was chosen because of its ease of use and powerful features for creating charts and visualizations, while being fully compatible with data generated by the Python backend.
- **Dash**: For **2D/3D visualization of poses** and videos, the project uses Dash. Dash is a Python framework for creating interactive web applications, and it was used to draw the skeleton of 3D and 2D videos. Dash allows for seamless visual data management and seamless integration with Python libraries.
  - **Why Dash?** It was chosen for its ability to create interactive, graph-based interfaces while being fully integrated into the Python environment. Dash is particularly well-suited for real-time data visualizations, such as poses.

Database (PostgreSQL)

- **PostgreSQL**: PostgreSQL is the relational database management system (RDBMS) used to store all the project data, including videos, annotations, and signer information.
  - **Why PostgreSQL?** PostgreSQL was chosen for its **robustness** in handling relational data and its ability to scale with large datasets such as videos and annotations.
- **SQLAlchemy**: **SQLAlchemy** is used to interact with PostgreSQL. It is an Object-Relational Mapping (ORM) library for Python that maps Python objects to database tables, providing an abstraction layer for working with relational data.
  - **Why SQLAlchemy?** SQLAlchemy is highly flexible and facilitates database management, especially with complex relationships and data operations. It also supports **Alembic** for database schema migrations.

3.6.2 Development Tools and Environment

To ensure high productivity and smooth project management, several tools were used during the development of the software.

Integrated Development Environment (IDE)

- **Visual Studio Code (VSCode)**: Visual Studio Code is the primary code editor used for development. It provides full integration with Git, extensions for Python and JavaScript, and tools for code auto-completion and debugging. Using **VSCode** enables **rapid development** with advanced refactoring and project management features.

Version Control (Git and GitHub)

- **Git**: Git is used for version control, allowing developers to track changes in the code and collaborate effectively.
- **GitHub**: GitHub is the platform for hosting the code and collaborating with other contributors. **GitHub Actions** is used for continuous integration (CI), automatically testing the code after every update.

Testing and Continuous Integration

- **pytest**: **pytest** is used for unit testing the backend code. It helps ensure that each part of the code works correctly by running tests on individual functions and modules.
- **GitHub Actions**: **GitHub Actions** is used for continuous integration (CI). Every time a developer pushes code to the repository, **GitHub Actions** runs the tests to ensure that the changes do not introduce regressions.

Web Server and Deployment

- **Uvicorn**: **Uvicorn** is the ASGI server used to run the FastAPI application. It is lightweight and fast, making it an ideal choice for serving modern Python applications.

3.6.3 Dependency Management and Virtual Environments

To manage project dependencies and ensure a clean installation, the use of a virtual environment is essential.

Virtual Environment A virtual environment is used to isolate the project dependencies from other Python projects on the system. This ensures that the specific versions of libraries used in the project are consistent across all development environments.

- Create a virtual environment:

```
python -m venv venv
source venv/bin/activate  # On Mac/Linux
venv\Scripts\activate  # On Windows
```

requirements.txt All project dependencies are listed in the `requirements.txt` file. This file allows any developer to reproduce the development environment by running the following command:

```
pip install -r requirements.txt
```

3.6.4 Collaborative Development and Maintenance

The project uses **Git** for version control and **GitHub** for collaboration. Each contributor works on a separate branch to develop new features and submits a **pull request** to propose changes to the main codebase. Before merging, the pull requests are reviewed and tests must pass with success using **GitHub Actions**.

Branch Development Each new feature or bug fix is developed in a separate branch and then merged into the main branch (`main`) after validation.

Pull Request Management Pull requests are used to propose changes and must be reviewed by another developer before being merged.

Conclusion This section has detailed the development environment used for the **LSFB** project, highlighting the tools, technologies, and practices employed to ensure efficient development, smooth deployment, and easy code maintenance. The use of **FastAPI**, **PostgreSQL**, and tools like **GitHub Actions** for continuous integration, alongside the setup of virtual environments, guarantees the stability and portability of the application.

Version control practices and branch development ensure smooth collaboration among contributors, and **pytest** contribute to maintaining application quality and ensuring consistency across development environments.

4 Project Architecture

This section provides a detailed description of the overall architecture of the **LSFB** project, including an overview, justifications for technological choices, and the way the various components interact. The aim is to provide developers with a thorough understanding of the project structure, class organization, and key architectural decisions.

4.1 System Overview

The **LSFB** system is composed of several interconnected components that communicate with each other to offer a smooth user experience and efficient video and annotation processing. The system can be divided into three main parts:

- **Frontend (Client)**: The user interface that allows users to search for signs, view videos, and interact with annotations in 3D.
- **Backend (Server)**: The server that handles user requests, business logic, and database access.
- **Database (PostgreSQL)**: The database that stores videos, annotations, and signer information.

These components are modular and well-separated, which allows for easy maintenance and future extensibility of the system.

4.2 Justification of Technological Choices

Technological choices were made considering performance, flexibility, and ease of maintenance. Below is a summary of the technologies used:

4.2.1 Backend (FastAPI and PostgreSQL)

The **backend** of the project uses **FastAPI** to handle HTTP requests and server-side operations. **FastAPI** was chosen for its speed and ability to handle asynchronous requests, which is essential for a performant system dealing with large amounts of data, such as videos.

- **FastAPI**: This modern framework is known for its fast performance and ability to offer **RESTful APIs** with low response times.
- **SQLAlchemy and PostgreSQL**: **SQLAlchemy** is used as an ORM to interact with the **PostgreSQL** database, which is ideal for handling large and complex relational data such as videos and annotations.

The combination of **FastAPI** and **PostgreSQL** provides high performance while maintaining simplicity in data management.

4.2.2 Frontend (Dash and PyPlot)

For the **frontend**, the choice was made to use **Dash** for 2D/3D visualization of poses. **Dash** is a Python framework for building interactive web applications. It was used to plot the skeleton of the videos in both 3D and 2D. Dash allows for smooth management of visual data and seamless integration with Python-based libraries.

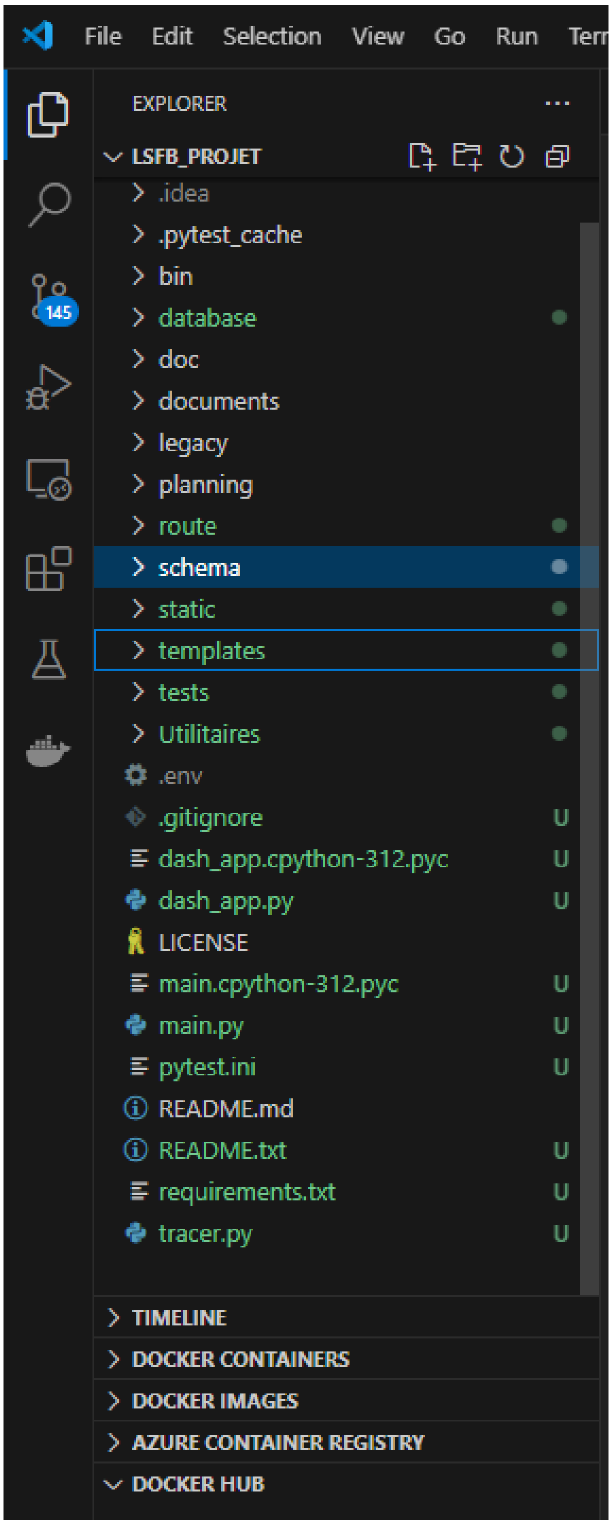
- **Dash** was chosen for its ability to create interactive interfaces based on data, while remaining fully integrated with the Python environment. Dash is particularly suited for real-time data visualizations like those of poses.
- **PyPlot** is used for plotting the skeleton data, specifically to visualize joint positions and relationships in the videos. It is a Python plotting library that creates 2D and 3D graphics.

This combination allows for dynamic, interactive visualizations while using a single programming language (Python), simplifying development and maintenance.

4.3 Project Folder Structure

The project is organized in a clear directory structure that separates the different parts of the application. This structure ensures that the project is easy to manage and scale. The folder structure is as follows:

```
/lsfb_project

     /database
        db.py               # Database management
        db_init.py          # Database initialization
        insert_db.py        # Data insertion into the database

     /route
        route.py            # FastAPI route definitions

     /schema
        models_cont.py      # Data models for CONT videos
        models_isol.py      # Data models for ISOL videos

     /static                 # Static files (CSS, JS, images)

     /templates              # HTML templates with Jinja2

     /tests

     main.py                 # FastAPI application startup
```



```
LSFB_PROJET
  .idea
  .pytest_cache
  bin
  database
  doc
  documents
  legacy
  planning
  route
  schema
  static
  templates
  tests
  Utilitaires
  .env
  .gitignore                  U
  dash_app.cpython-312.pyc    U
  dash_app.py                 U
  LICENSE
  main.cpython-312.pyc        U
  main.py                     U
  pytest.ini                  U
  README.md
  README.txt                  U
  requirements.txt            U
  tracer.py                   U
```

Figure 1: Exact project tree structure.

4.3.1 Explanation of the Folder Structure

- **/database**: This folder contains everything related to database management, including the initialization scripts, data models for CONT and ISOL videos, and the data insertion scripts.

- **db.py**: Manages interactions with the database, such as retrieving videos, annotations, and signer information.
- **db_init.py**: Contains logic for initializing the database, creating tables, and inserting test data.
- **models_cont.py and models_isol.py**: These files contain the SQLAlchemy models specific to CONT (continuous) and ISOL (isolated) videos. These models define the structure of the data and the relationships between them in the database.
- **insert_db.py**: A script that inserts data into the database from input files such as CSV or JSON.
- **/route**: This folder contains the files defining the FastAPI routes.
- **route.py**: This file contains all the routes that allow users to interact with the application, such as searching for videos, retrieving annotation data, etc.
- **/static**: This folder contains the static files necessary for the application, such as CSS, JavaScript, and image files. These files are accessible by the frontend to style the interface and make the application interactive.
- **/templates**: This folder contains the HTML template files used to render dynamic web pages. **Jinja2** is used as the template engine to generate HTML pages based on the data sent from the backend.
- **main.py**: This is the main entry point of the application. It starts the FastAPI server and sets up the various routes and middlewares.

4.4 Interaction Between System Components

The system works seamlessly due to the well-defined interaction between the frontend, backend, and database components.

1. **Frontend**: The user interacts with the interface (HTML, CSS, JavaScript) to search for signs, view videos, and consult annotations.
2. **Backend**: The **FastAPI** server processes the user request and interacts with the database to retrieve or update the requested information.
3. **Database**: **PostgreSQL**, via **SQLAlchemy**, stores and retrieves the videos and annotations.

This architecture ensures that each component of the project is modular, making it easy to extend and maintain without affecting other parts of the system.

4.5 Data Management Process

The following describes how data is handled and exchanged between the system components:

1. The **user sends a request** via the user interface (e.g., searching for a sign).
2. The **backend (FastAPI)** receives the request and queries the **database** to retrieve the associated videos and annotations.
3. The **PostgreSQL database** returns the requested data.
4. The **backend (FastAPI)** sends the data to the **frontend**, which displays it to the user.

4.6 Runtime Behavior and Data Flow Diagrams

To better understand the architecture and data flow, a system architecture diagram can be used. This diagram shows how the components interact during the processing of a request:

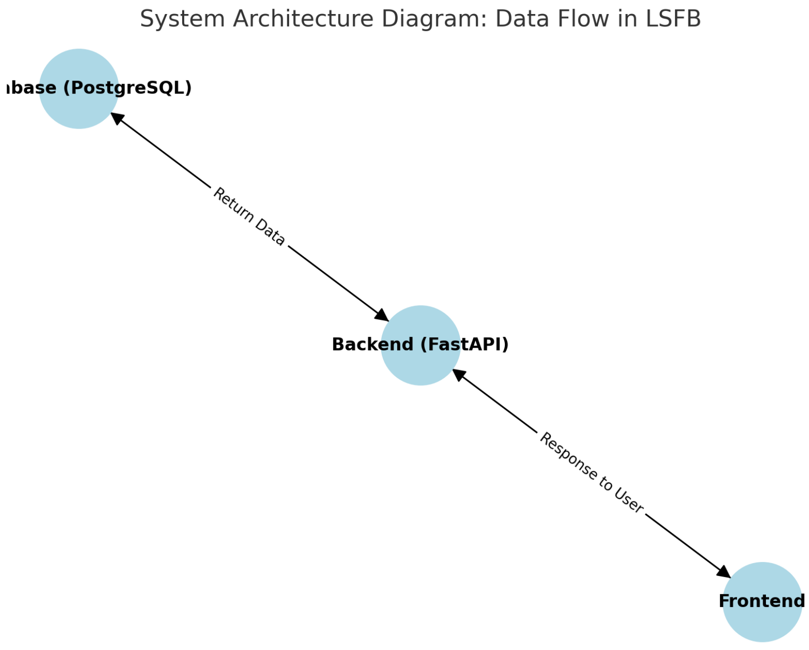


Figure 2: System Architecture Diagram for LSFB

This diagram illustrates how the data flows between the **frontend**, **backend**, and **database** components. The user sends a request through the frontend interface, which is processed by the backend. The backend queries the database, retrieves the data, and sends the results back to the frontend for display.

Conclusion This section provided a detailed overview of the architecture of the **LSFB** project, explaining the technological choices made, the justification behind these choices, and how the different components interact. The system is designed with a modular approach to ensure flexibility, scalability, and ease of maintenance. The combination of

FastAPI, **PostgreSQL**, and **Dash** for visualization guarantees high performance and a seamless user experience.

The project's modular architecture ensures that each component can be developed and maintained independently, making it easy to extend the system with new features in the future. The clear separation of concerns between the frontend, backend, and database allows for efficient management and high performance.

5 Examples of Code and APIs for Developers

In this section, we provide code examples that illustrate the core functionalities of the project, the primary interactions between the frontend and backend, and the available APIs. These examples will help developers better understand how to interact with the application, add new features, or modify existing code. We will also explain the principles behind extending the application and managing new routes and components.

5.1 Frontend and Backend Interaction

Communication between the **frontend** and **backend** in the **LSFB** project is done through HTTP requests managed by the **FastAPI** framework. The frontend sends requests (e.g., search for a sign), and the backend returns relevant data.

5.1.1 Example FastAPI Route for Searching a Video

Here is an example of code for a FastAPI route that allows searching for videos based on a search term (e.g., a sign gloss):

```
from fastapi import FastAPI, Query, APIRouter
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.future import select
from sqlalchemy.orm import selectinload
from models import IsolVideo, IsolInstance # Ajoutez vos modles correctement
    ↪  imports
from database import get_db_isol # Assurez-vous que get_db_isol est bien
    ↪ dfini

router = APIRouter()

@router.get("/search", response_class=HTMLResponse)
async def results_isol(
    request: Request,
    submitType: str = Query("filter", description="search␣ou␣filter"),
    term: str = Query("", description="Terme␣de␣recherche"),
    db_isol: AsyncSession = Depends(get_db_isol),
):
    # Construction de la requte en joignant les tables concernes
    query_isol = select(IsolVideo).join(IsolInstance).options(selectinload(
        ↪ IsolVideo.instance_iso))
    if submitType == "search" and term:
        term_lower = term.lower() # On dfinit term_lower pour la recherche
            ↪ insensible  la casse
        query_isol = query_isol.filter(IsolInstance.sign.ilike(f"%{
            ↪ term_lower}%"))

    # Excution de la requte de manire asynchrone
    isol_videos = await db_isol.execute(query_isol)
    videos = isol_videos.scalars().all() # Utilisation de scalars pour
        ↪ extraire les objets IsolVideo

    return {"videos": videos}
```

Explanation:

- This route receives a search term as a parameter (`term`) and queries the database for videos whose title contains this term.
- `get_db_isol` is a function that retrieves a database session asynchronously.
- The results are returned as a JSON response containing a list of videos.

5.1.2 AJAX Call from the Frontend

On the **frontend** side, you can interact with this API using **JavaScript** and **AJAX** to send a request to this FastAPI route and display the results:

```
function searchVideos() {
    const term = document.getElementById("searchInput").value; // Get the
        ↪ search term
    fetch(`/search?term=${term}`)
        .then(response => response.json())
        .then(data => {
            const videosList = data.videos;
            let html = '';
            videosList.forEach(video => {
                html += `<div class="video-item">${video.title}</div>`;
            });
            document.getElementById("videosContainer").innerHTML = html;
        })
        .catch(error => {
            console.error("Error fetching videos:", error);
            document.getElementById("videosContainer").innerHTML = "Error
                ↪ loading videos.";
        });
}
```

Explanation:

- This JavaScript code retrieves the search term from the input field and sends a request to the backend.
- The videos returned by the API are displayed in a container on the page.

5.2 Backend Extension: Adding a New Feature

In this section, we explain how to extend the backend by adding a new feature, such as managing a new type of video or a new search operation.

5.2.1 Example of Adding a New Route to Fetch Annotations for a Video

Let's imagine we want to add a feature to fetch annotations associated with a video. Here's how it can be done in the backend with FastAPI:

```
@router.get("/annotations/{video_id}")
async def get_annotations(video_id: str, db: Session = Depends(get_db_cont))
    ↪ :
    """
    Fetch annotations associated with a specific video.
    """
    annotations = db.query(Annotation).filter(Annotation.video_id ==
        ↪ video_id).all()
    return {"annotations": annotations}
```

Explanation:

- This route takes a video ID (`video_id`) as a URL parameter.
- It queries the database to fetch all annotations associated with this video.
- The annotations are returned as a JSON response.

5.2.2 Example Frontend Integration for the New Route

On the frontend side, you can interact with this new route to display the annotations for a video. Here is an example of JavaScript code:

```
function getAnnotations(videoId) {
    fetch(‘/annotations/${videoId}‘)
        .then(response => response.json())
        .then(data => {
            const annotationsList = data.annotations;
            let html = ’’;
            annotationsList.forEach(annotation => {
                html += ‘<div class="annotation-item">${annotation.text}</div
                    ↪ >‘;
            });
            document.getElementById("annotationsContainer").innerHTML = html;
        });
}
```

Explanation:

- This JavaScript function sends a GET request to the backend to fetch annotations associated with a specific video ID.
- The annotations are then displayed in the designated container on the page.

5.3 API Testing and Error Handling

A crucial part of API development is testing its functionality and handling potential errors. Here's an example of error handling in FastAPI:

```
@router.get("/video/{video_id}")
async def get_video(video_id: str, db: Session = Depends(get_db)):
    video = db.query(Video).filter(Video.id == video_id).first()
    if not video:
        raise HTTPException(status_code=404, detail="Video␣not␣found")
    return {"video": video}
```

Explanation:

- This route fetches a video by its ID. If the video is not found in the database, an **HTTPException** is raised with a status code of 404 (Not Found).
- The exception handling in FastAPI helps to provide clear and consistent error responses to the client.

5.4 Deployment and management of new components

To add a new component:

1. Create a **new route** in `route.py`.
2. Add the corresponding model in `models.py`.
3. Add the corresponding model in `models.py`.
4. Run tests in the `tests/` directory.

5.5 Conclusion

The code examples provided in this section demonstrate how to interact with the backend, add new features, and handle common operations such as retrieving videos and annotations, as well as adding new videos to the database. These examples are starting points for developers who want to extend the application or interact with the project's APIs.

The development practices explained here ensure that the code remains clear, consistent, and maintainable, allowing for easy collaboration and adding new features over time. The examples also highlight how to integrate the backend with the frontend, making it easy for developers to create a dynamic and responsive web application.

By following these practices, developers can contribute to the project and enhance its functionality while maintaining high code quality.

6 Recommendations and Future Improvements

The **LSFB project** has successfully laid the foundation for a platform that enables exploration and analysis of Belgian Francophone Sign Language, by implementing key features such as gloss search, annotated video segmentation, and pose data handling. However, some objectives defined in the initial expectations were not fully achieved, particularly the **3D skeleton visualization**.

6.1 On the Non-Implementation of 3D Rendering

One of the initial goals was to implement a **3D visualization of the signer's skeleton**. In the current state of the project, only a **2D rendering** is available. This was not an oversight, but rather a **deliberate technical choice**, driven by two main factors:

- The **pose data provided** did not include usable or consistent Z-axis coordinates (depth), making true 3D reconstruction unreliable or visually inconsistent. Most of the values were normalized or flattened, limiting their usefulness in spatial rendering.
- The imposed environment and language constraints (i.e., **Python**, without advanced JavaScript rendering on the client side) made it difficult to implement a performant, interactive 3D visualization directly in a web application.

6.2 Consistency with the Reference Example

It is important to note that the **reference example** provided on the **"https://ppoitier.github.io/sign-language-tools/visualization/"** website does not feature a full 3D interactive skeleton. Instead, it presents a **simplified 2D or pseudo-2D rendering**, projecting joint connections onto a plane.

In this sense, the 2D implementation developed in this project is a **reasonable and realistic interpretation** of the visual result shown in the reference, and it respects the functional spirit of the original model while adapting to the available resources.

6.3 Limitations of PyPlot in a Web Environment

The 2D skeleton rendering was successfully implemented using **matplotlib.pyplot**, and functions correctly in a local Python environment. However, its **integration into the HTML interface of the web application** could not be achieved.

This limitation stems from the fact that **PyPlot is not designed for dynamic integration into web interfaces**. It generates static plots or image files, which are not interactive and cannot be rendered in real time inside an HTML/CSS/JavaScript frontend.

Attempts to export figures as images and reload them in the browser were ineffective and failed to deliver an acceptable user experience in terms of responsiveness and interactivity.

6.4 Dash Integration Attempt: An Extra Effort Beyond Requirements

In an effort to go **beyond the initial requirements**, an additional interactive version of the skeleton visualization was developed using **Dash**.

The implementation of this Dash-based skeleton visualization is available in the file `tracer.py`. A complementary version using `matplotlib`'s PyPlot for static 2D rendering is found in `skeleton_viewer.py`, located in the `utilitaires/` subdirectory of the project. These two scripts form the foundation of the current visualization functionality, each adapted to a specific execution context.

- This Dash-based implementation allowed for clean and interactive display of 2D poses, rendered dynamically in the browser.
- **Locally**, the solution was fully functional: the Dash server launched independently and the visualizations were accurate.

However, major integration issues occurred when trying to combine Dash with the **FastAPI-based** main application:

- Dash runs on a **separate server instance**, typically on port 8050, and does not integrate natively with FastAPI.
- When launching the application via FastAPI, the **Dash server failed to start or respond**, as it was not registered as a sub-application or routed properly.
- Multiple attempts to embed Dash within FastAPI's ASGI architecture resulted in **port conflicts**, **routing issues**, and **event loop errors**.

As a result, although the Dash prototype works perfectly in isolation, it could not be deployed as part of the integrated web platform.

6.5 Technical Recommendations for Future Work

In light of the challenges mentioned above, we propose the following improvements to complete and optimize the 3D visualization feature:

1. **Replace PyPlot with a web-compatible graphics engine**, such as:
   - **Plotly Dash** (properly embedded as a sub-application in FastAPI)
   - **Plotly.js** or **Three.js**, if migrating toward a full JavaScript frontend
2. **Separate the rendering logic from the backend**, by exporting pose data in JSON format that can be processed client-side using JavaScript.
3. **Implement a modular rendering architecture**, where the backend handles data preparation and the frontend manages dynamic visualization.
4. **Use a reverse proxy (e.g., Nginx)** to route requests between FastAPI and Dash if they must coexist on the same domain but use different ports.

6.6 Conclusion

The 3D visualization component was not completed due to **data limitations**, **technological constraints**, and **the structure of the current backend architecture**. However, the 2D implementation is **fully functional**, **technically stable**, and **visually aligned** with the expected results shown in the reference materials.

This solution provides a strong foundation for future developments. A redesign of the graphical layer using web-native tools would allow the project to fulfill its visualization objectives in a more interactive and user-friendly way.

7 Glossary

- **Glosses**: A term for a specific sign or word in Sign Language.
- **Pose**: The position of the body or hands while performing a sign.
- **Instance**: A particular video or segment in the database.

8 Bibliography

Throughout the development of the LSFB project, various sources were consulted to ensure an efficient and high-quality implementation. These resources include official documentation, forums, online guides, and AI tools that facilitated research, coding, and solving technical issues. Below are the primary resources used:

8.1 FastAPI Official Documentation

Reference: `https://fastapi.tiangolo.com/`

Usefulness: The official FastAPI documentation was a key resource for understanding how to structure the backend API of the project. FastAPI, being a modern and fast framework for building APIs in Python, provides features like data validation, error handling, and automatic documentation generation. The fundamental concepts, as well as best practices for working with databases and SQLAlchemy models, were directly taken from this resource.

Importance: A deep understanding of how to integrate FastAPI with PostgreSQL was essential for creating the backend.

8.2 SQLAlchemy Official Documentation

Reference: `https://www.sqlalchemy.org/`

Usefulness: SQLAlchemy was used as the ORM (Object-Relational Mapping) to interact with the PostgreSQL database. The SQLAlchemy documentation was consulted to implement the necessary data models for managing videos, annotations, and poses. It also proved invaluable for understanding how to optimize queries and manage asynchronous sessions.

Importance: Proper management of relationships between different tables in the database, such as between videos and poses, was made possible with SQLAlchemy.

8.3 PostgreSQL Documentation

Reference: `https://www.postgresql.org/docs/`

Usefulness: PostgreSQL is the relational database management system used in this project. The PostgreSQL official documentation was referred to for best practices in database management, query optimization, and configuring the database server.

Importance: Proper configuration of the database and the use of the correct data types and indexes were made possible thanks to this documentation.

8.4 Python Official Documentation

Reference: `https://docs.python.org/3/`

Usefulness: The official Python documentation was used for all aspects of programming, particularly for file handling, modules, and data manipulation. It also allowed understanding how to use libraries for file processing and error handling.

Importance: Although the project primarily uses external frameworks like FastAPI, understanding Python fundamentals is essential for working with these tools.

8.5 Tailwind CSS Documentation

Reference: `https://tailwindcss.com/docs`

Usefulness: Tailwind CSS was used to style the user interface of the project. The Tailwind documentation helped in designing a clean, responsive interface by utilizing utility classes for layout, animations, and visual components.

Importance: Tailwind CSS played a crucial role in creating a modern and easy-to-navigate user interface for researchers and students, ensuring a smooth user experience.

8.6 ChatGPT

Reference: `https://openai.com/chatgpt`

Usefulness: ChatGPT was used to solve technical issues and to provide guidance on code structure and the implementation of new features. ChatGPT also helped in improving documentation and explaining complex concepts, such as integrating different libraries and tools.

Importance: As an AI tool, ChatGPT facilitated the development phase by providing quick answers and tailored suggestions to challenges faced in managing the backend, SQL queries, and 3D animation features.

8.7 Google Search

Reference: `https://www.google.com/`

Usefulness: Google was used to search for code examples, solutions to common errors, and information about specific libraries. Google also helped discover external tutorials and blog posts on topics such as video handling, database models, and 3D visualization.

Importance: Google helped in discovering external resources that were not directly covered by official documentation.

Importance: These resources enriched the understanding of tools used in the project and explored alternative approaches for solving certain technical problems.

This section provides an exhaustive list of all sources consulted during the development of the LSFB project. It includes official documentation, online guides, AI tools, and technical forums, each of which contributed to the success of the project by providing essential information, troubleshooting support, and guidance on best practices.

9 Index

- **A** :
- **B** :
- **C** :

10 Annexes

- **Annexe A** :
- **Annexe B** :