Arabic Sign Language Translator

2024/2025

Team Members

Abdelrahman Mostafa Samy

Abdelrahman Hassan Gabry

Hamza Mohammed Ahmed

Hazem Yasser Ibrahim

Under Supervision Of

Dr. Michael Nasif

Eng. Hassan Talal

Misr International Computer Academy

Artificial Intelligence Department

Productivity and Vocation Training Department

Github

Acknowledgment

We would like to express our deepest gratitude to everyone who has contributed to the success of this project.

First and foremost, we extend our sincere appreciation to our supervisor, **Dr. Michael Nasif** and **Eng. Hassan Talal** for their invaluable guidance, continuous support, and insightful feedback throughout the research and development process. Their expertise and encouragement have been instrumental in shaping this project.

We would also like to thank our institution and faculty members for providing the necessary resources and a stimulating learning environment that enabled us to explore and implement advanced AI technologies.

Special thanks go to our families and friends, whose unwavering support, patience, and encouragement have motivated us throughout this journey.

Finally, we acknowledge the contributions of the deaf community, researchers, and developers in the field of sign language recognition. Their work has inspired and driven us to develop an innovative solution that promotes accessibility and inclusivity.

This project is a testament to the power of collaboration, perseverance, and the pursuit of knowledge, and we are grateful for everyone who played a part in its realization.

Abstract

The **AI-Powered Sign Language Recognition System** is an advanced **deep learning**-based solution designed to bridge the communication gap between the deaf and hearing communities. This system leverages **computer vision**, **sequential modeling**, and natural human-computer interaction to provide real-time **translation of sign language gestures into text**.

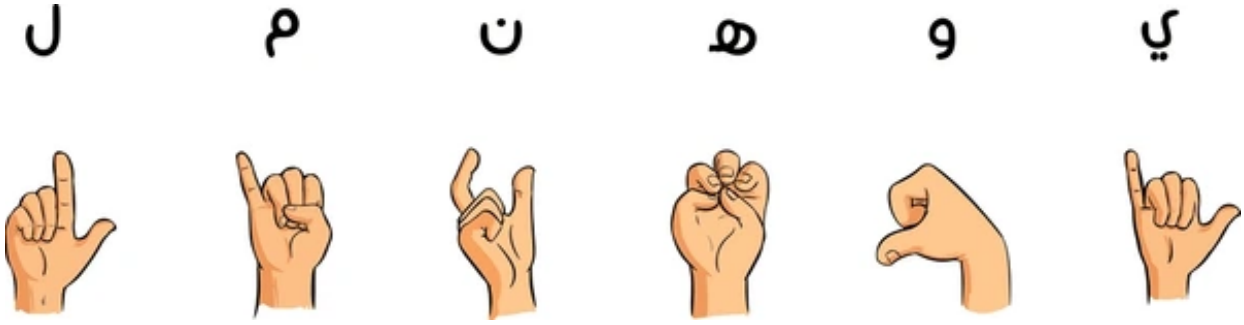
The core of the system is built on MediaPipe, which extracts precise hand, face, and body landmarks from video input. These extracted features are **processed** by a **Sequential LSTM model**, trained on a carefully curated dataset to ensure robust gesture classification. To enhance recognition accuracy and adaptability, CNN-LSTM and Transformer-based architectures have been explored, enabling the system to capture both spatial and temporal dependencies in sign language expressions.

Designed for scalability and real-world integration, this **system** is implemented in an intuitive Tkinter-based graphical user interface (GUI) and is adaptable for **deployment on mobile** (Flutter-based) and kiosk platforms. The technology is particularly **suited for public service** applications in **restaurants**, **hospitals**, **banks**, **airports**, **government offices**, and educational institutions, providing an automated, cost-effective, and inclusive **communication tool**.

This project represents a significant step toward accessible AI-driven communication for the hearing-impaired community. Future enhancements will focus on expanding linguistic coverage, increasing model efficiency, and integrating multi-modal recognition systems, ensuring a seamless and universally accessible experience.

Chapter 1 | Introduction and Objectives

Sign language is one of the most **essential communication tools** for the **deaf and hard-of-hearing community**, enabling them to **express thoughts, emotions, and ideas effectively**. However, a significant challenge persists: the **lack of widespread understanding of sign language** among the **general population**. This **communication barrier** often leads to **social isolation** and **limited accessibility** in various **daily interactions**, particularly in **public places** such as **restaurants**, **hospitals**, **banks**, and **educational institutions**.



1-1 Project Idea

Communication is a **fundamental human right**, yet millions of **deaf and hard-of-hearing individuals** face significant challenges in **daily interactions** due to the **lack of widespread sign language proficiency** among the **general population**. This **communication gap** often leads to **social exclusion**, **limited access to essential services**, and **reduced opportunities in education and employment**.

Traditional solutions, such as **human interpreters** and **written communication**, are not always **accessible**, **practical**, or **cost-effective**. Many **public and private institutions** struggle to provide **effective**

communication support for deaf individuals, further limiting their **independence**.

To address this challenge, our project introduces an **AI-powered Sign Language Recognition System**, a **technological solution** that leverages **computer vision** and **deep learning** to **translate sign language gestures into text in real time**.

By integrating **MediaPipe** for **feature extraction** and **LSTM-based deep learning models** for **classification**, our system provides **highly accurate gesture recognition**, making it a **valuable tool** for enhancing **accessibility and inclusivity**.

The project is designed for **scalability** and **real-world implementation**, allowing **deployment in public service areas, businesses, and educational settings**. The system features an **intuitive Tkinter-based GUI**, with future plans for **mobile** and **kiosk-based applications** to ensure **ease of use across different platforms**.

This **graduation project** not only demonstrates the **potential of artificial intelligence** in **human-computer interaction** but also highlights the **importance of technological inclusivity** in **modern society**. Our objective is to create a **practical**, **efficient**, and **scalable solution** that **bridges the gap** between the **deaf and hearing communities**, fostering a **more inclusive world for everyone**.

1-2 Project Objectives

Real-Time Sign Recognition :

Develop a system that accurately detects and interprets Arabic sign language gestures in real-time using a camera feed.

Gesture Classification：

Implement a machine-learning model to classify hand gestures into Arabic letters and words with high accuracy.

User Authentication：

Enable users to create accounts, log in, and securely track their learning progress and translation history.

History and Progress Tracking：

Store and display users' past translations and learning progress to help them review and improve over time.

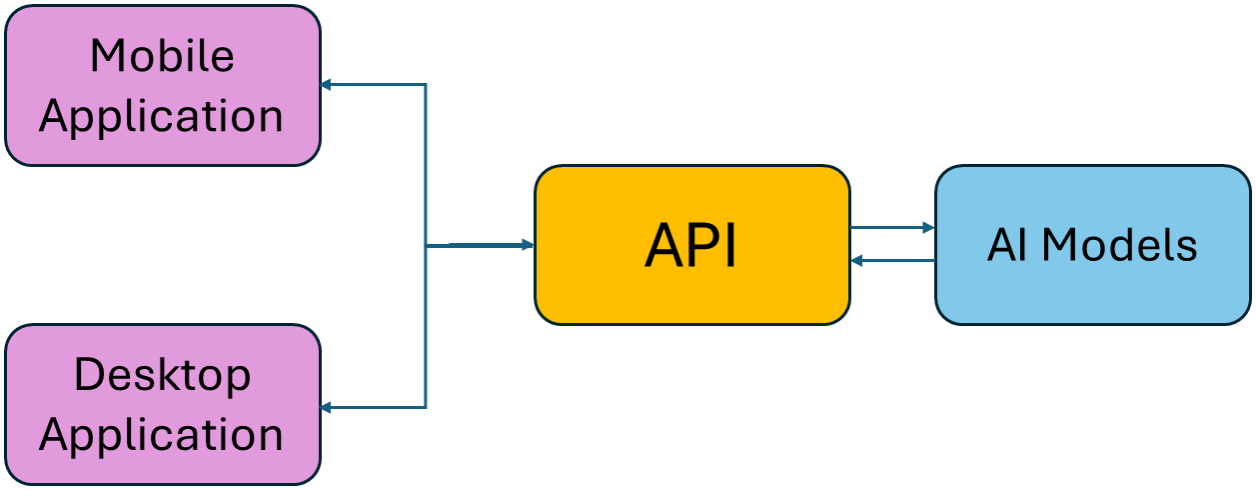
Text-to-Sign Language Conversion：

Provide a feature that converts Arabic text into corresponding sign language gestures, supporting interactive learning.

User-Friendly Interface：

Design an intuitive and accessible UI that allows users to interact with the application effortlessly.

1-3 Project Block Diagram



Chapter 2 | Scientific Background

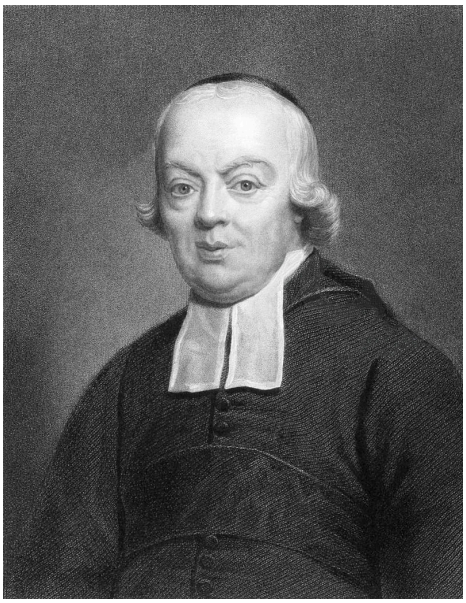
Scientific Background on Sign Language and Brain Processing

2-1 Introduction

Sign language serves as a **vital communication tool** for the **deaf and hard-of-hearing community**, offering a **structured linguistic system** that enables **interaction through visual and gestural means**. Over time, different regions have developed their own **sign languages**, each with **unique grammatical rules** and **cultural influences**. Understanding the **history of sign language** and the way the **brain processes it** provides insight into its **significance** and **cognitive impact**. Additionally, **Arabic Sign Language (ArSL)** holds a **distinctive place** within the broader framework of sign languages, reflecting the **linguistic diversity** of the **Arab world**.

2-2 Concept 1: General History of Sign Language

Sign language has been an essential mode of communication for deaf individuals throughout history. While informal gesture-based communication likely existed for centuries, the **formalization** of sign languages **began in the 18th century**.



One of the key figures in early sign language development was **Charles-Michel de l'Épée**, a French priest who established the first public school for the deaf in the **1760s**.

He is often credited with laying the foundation for modern sign languages, particularly French Sign Language (LSF), which later influenced American Sign Language (ASL) and many others. Throughout the 19th and 20th centuries, different regions developed their own sign languages, often based on local deaf communities and cultural influences. Despite attempts to unify sign languages, linguistic diversity remains a defining characteristic, with each language possessing its own grammar and vocabulary.

2-3 Concept 2: How the Brain Processes Sign Language

Research in neuroscience has shown that the brain processes sign language in ways similar to spoken language. Studies using functional MRI (fMRI) and other neuroimaging techniques indicate that the same brain regions—such as Broca's area and Wernicke's area—are activated when processing both spoken and sign languages. This suggests that the brain recognizes sign language as a linguistic system rather than merely as a set of gestures.

Interestingly, sign language users demonstrate heightened visual-spatial processing skills, as sign language relies on movement, handshapes, and spatial relationships. The left hemisphere of the brain, responsible for language processing, works alongside the right hemisphere, which specializes in spatial and visual information, to decode and produce sign language efficiently. This interplay between hemispheres highlights the cognitive flexibility required for sign language comprehension and production.

2-4 Arabic Sign Language (ASL) and Its History

Arabic Sign Language (ASL) refers to a collection of sign languages used across Arabic-speaking countries. Unlike spoken Arabic, which has a standardized version (Modern Standard Arabic), sign languages in the Arab world vary significantly by region. Efforts to unify these languages have led to the development of "Unified Arabic Sign Language," though local variations persist.

Historically, deaf communities in the Arab world have relied on indigenous sign languages, but formal recognition and development of ArSL began in the 20th century. Organizations such as the Arab Federation of the Deaf and government institutions have worked toward standardizing sign language education and resources.

ArSL shares structural similarities with other sign languages but also incorporates unique cultural and linguistic elements from Arabic. Hand movements, facial expressions, and spatial orientation play a crucial role in communication. Despite growing awareness and support, challenges such as limited educational resources and societal acceptance continue to affect ArSL users.

Chapter 3 | Artificial Intelligence Software

3-1 Introduction

Our ASL recognition system utilizes **Artificial Intelligence (AI) and Deep Learning** to accurately detect and classify Arabic Sign Language gestures. We have developed two advanced models:

1. **CNN-Based Model (MobileNetV2 with Data Augmentation)** – This model is designed for efficient and lightweight image-based sign recognition. We use **MobileNetV2**, a convolutional neural network (CNN) optimized for speed and accuracy. To improve generalization and robustness, we apply **data augmentation techniques** such as rotation, flipping, and brightness adjustment. This ensures the model can handle variations in lighting, hand positions, and backgrounds, making it more adaptable to real-world scenarios.
2. **Transformer-Based Model** – To enhance performance and context understanding, we implement a **transformer-based model** for sign language recognition. Unlike CNNs, which process spatial features, transformers leverage **self-attention mechanisms** to capture **temporal and spatial relationships** in sign gestures. This allows the model to recognize complex hand movements more effectively, improving accuracy in real-time gesture classification.

Both models are designed to work in **real-time** and provide a reliable solution for translating ASL into text or speech. Our goal is to enhance accessibility for the Deaf and Hard-of-Hearing (DHH) community by developing an AI-powered tool that is **efficient, scalable, and adaptable** for various applications, including **education, accessibility, and human-computer interaction**.

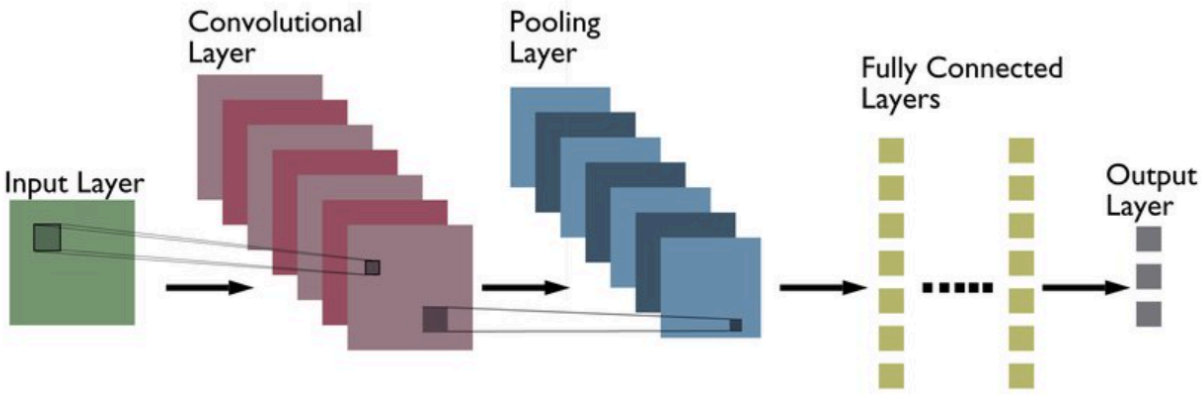
3-2 Concepts

3-2-1 CNN model

What a CNN model is? What are the most fundamental components of a CNN architecture?

Convolutional Neural Networks, commonly referred to as CNNs, are a specialized kind of neural network architecture that is designed to process data with a grid-like topology. This makes them particularly well-suited for dealing with spatial and temporal data, like images and videos, that maintain a high degree of correlation between adjacent elements.

CNNs are similar to other neural networks, but they have an added layer of complexity due to the fact that they use a series of convolutional layers. Convolutional layers perform a mathematical operation called convolution, a sort of specialized matrix multiplication, on the input data. The convolution operation helps to preserve the spatial relationship between pixels by learning image features using small squares of input data. . The picture below represents a typical CNN architecture



Convolutional Layers

Convolutional layers slide **filters** (kernels) **across input data to detect** features like edges, textures, or complex **patterns** in deeper layers. As the filters **scan the image**, they produce a feature map **highlighting detected features**. Stacking multiple convolutional layers allows the network to learn more intricate patterns, making them essential for **extracting** meaningful **features** from images.

Pooling Layers

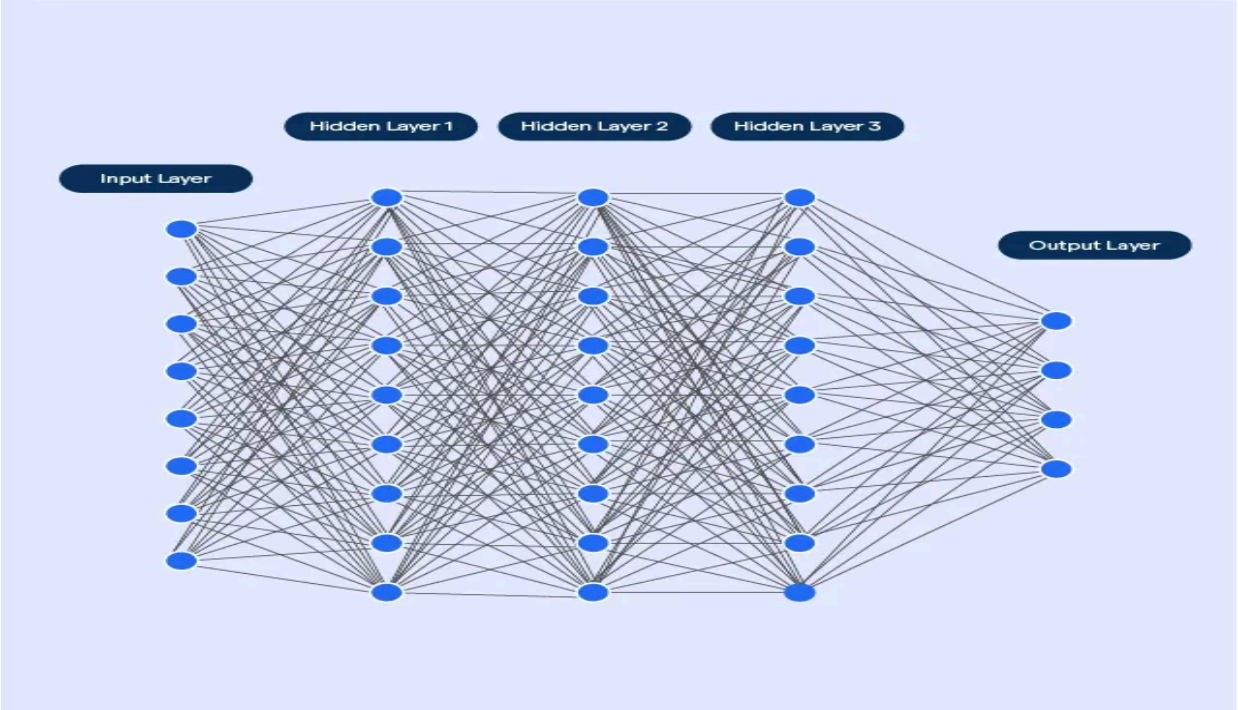
Pooling layers follow convolutional layers to reduce spatial dimensions (height and width), lowering computational cost and preventing overfitting. By downsampling feature maps, pooling layers make the model more robust to small shifts or rotations. The two main types are:

- Max Pooling: **Selects** the **highest value** in a region, **preserving** the most **prominent feature**.
- Average Pooling: **Computes** the **average value**, providing a **smoother representation**.

Max pooling is especially useful for **reducing complexity** while **maintaining key features**, ensuring the model remains efficient and invariant to small transformations.

Fully Connected Layers

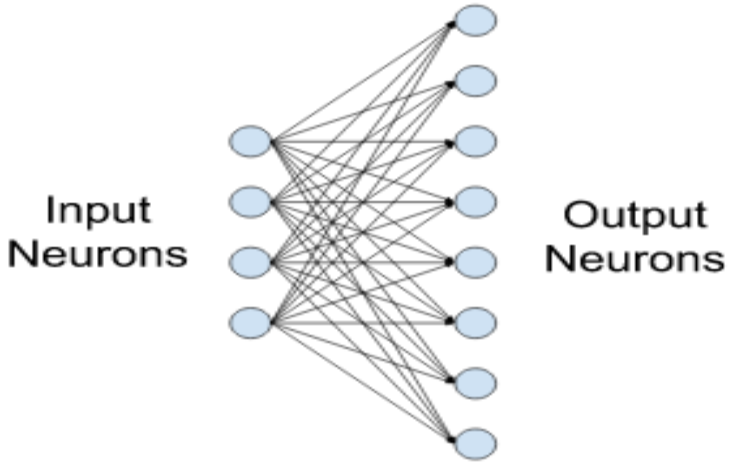
Fully connected layers appear at the end of a CNN, where **each neuron connects** to all **neurons in the previous layer**. These layers take the extracted features, flatten them into a one-dimensional vector, and use them for classification. For instance, in an image classifier, the final fully connected layer assigns labels like "dog" or "cat" based on learned features.



Output Layer

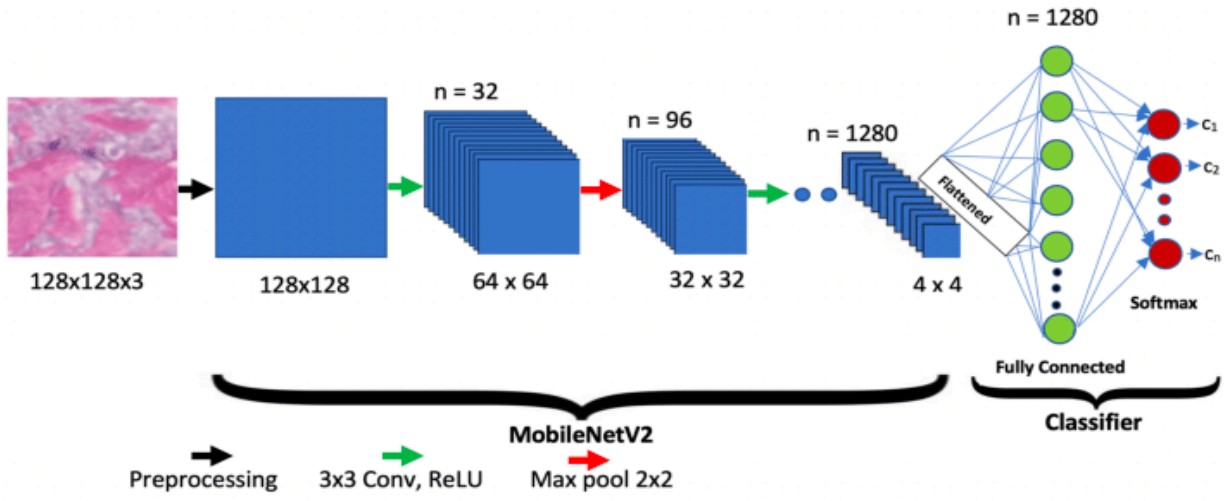
The output layer in a Convolutional Neural Network (CNN) plays a critical role as it's the final layer that produces the actual output of the network, typically in the form of a classification or regression result. Its importance can be outlined as follows:

1. Transformation of Features to Final Output: The earlier layers of the CNN (convolutional, pooling, and fully connected layers) are responsible for extracting and transforming features from the input data. The output layer takes these high-level, abstracted features and transforms them into a final output form, which is directly interpretable in the context of the problem being solved.
2. Task-Specific Formulation:
    - For classification tasks, the output layer typically uses a softmax activation function, which converts the input from the previous layers into a probability distribution over the predefined classes. The softmax function ensures that the output probabilities sum to 1, making them directly interpretable as class probabilities.
    - For regression tasks, the output layer might consist of one or more neurons with linear or no activation function, providing continuous.



MobileNets – CNN Architecture for Mobile Devices

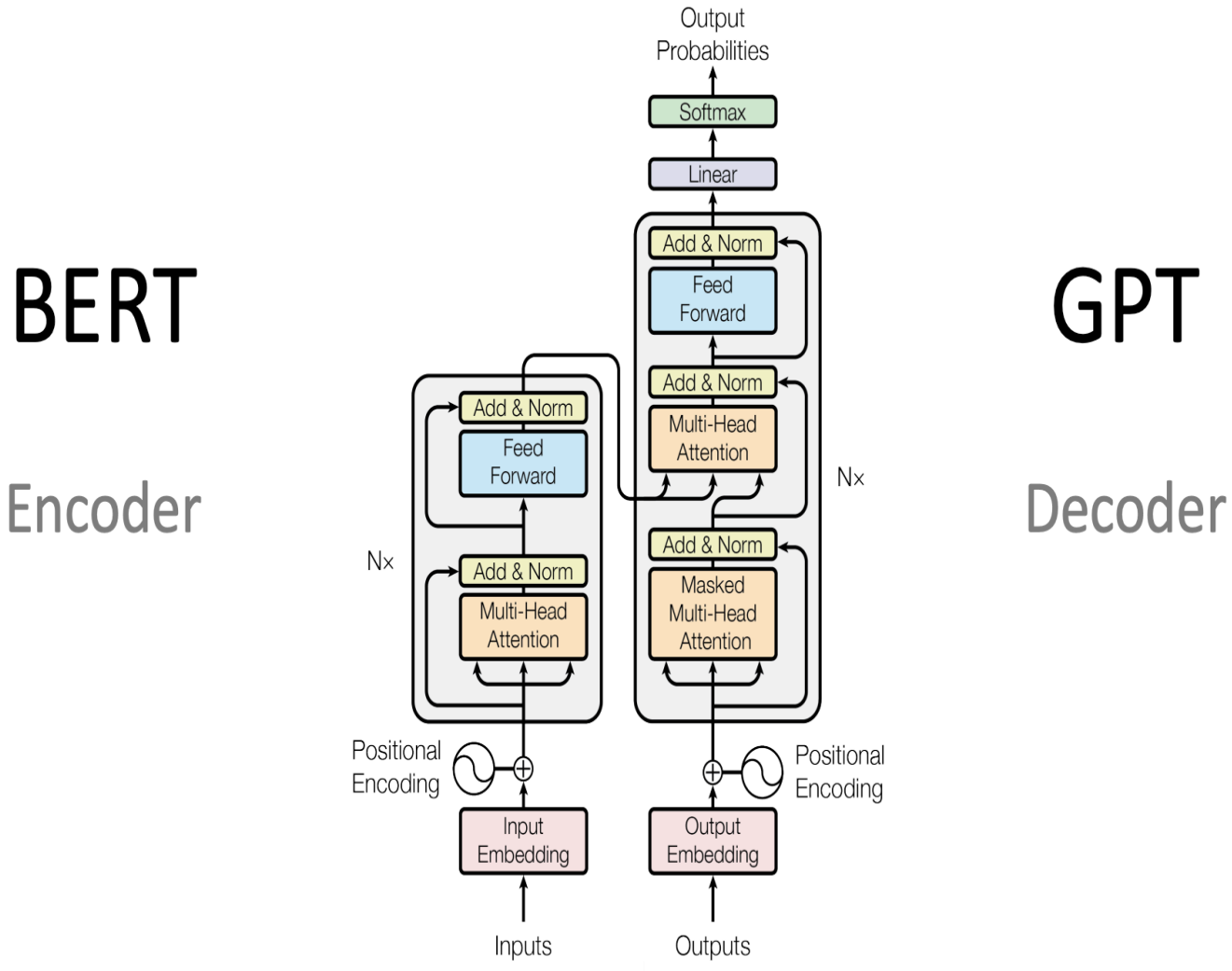
MobileNets are CNNs that can be fit on a mobile device to classify images or detect objects with low latency. MobileNets have been developed by Andrew G Trillion et al.. They are usually very small CNN architectures, which makes them easy to run in real-time using embedded devices like smartphones and drones. The architecture is also flexible so it has been tested on CNNs with 100-300 layers and it still works better than other architectures like VGGNet. Real-life examples of MobileNets CNN architecture include CNNs that is built into Android phones to run Google's Mobile Vision API, which can automatically identify labels of popular objects in images



1. Transformer Models

The **Transformer model**, **introduced in** the **2017** paper *"Attention Is All You Need"* by Vaswani et al., **revolutionized deep learning** with its **self-attention mechanism**, enabling parallel processing and a strong grasp of global context. Unlike traditional models such as RNNs, LSTMs, or CNNs, **Transformers analyze all input elements simultaneously**, dynamically weighting relationships between them. Core innovations include multi-head attention, positional encodings, and an encoder-decoder architecture built from stacked attention layers.

These features allow Transformers to efficiently handle long-range dependencies and scale across various domains, including NLP, computer vision, and generative AI. In our ASL recognition system, the Transformer plays a key role in learning spatial-temporal dependencies across sequences of hand movements, improving gesture interpretation accuracy.



2. How the Transformer Works in ASL Recognition

Our Transformer-based model is designed to **analyze video sequences** where each frame contains hand gesture information. The process follows these main steps:

Step 1: Input Data Representation

Before passing data to the Transformer, we first **extract key features** from each frame of the video. This can be done in two ways:

- **Using Keypoint Detection**: We use **MediaPipe** to extract keypoints (e.g., hand positions, joint angles).
- **Using Image Features**: Instead of raw pixels, we can use a **CNN (like MobileNetV2)** to extract meaningful features from each frame.

Each frame (or set of keypoints) is then transformed into a **vector representation** and organized into a sequence.

Step 2: Positional Encoding

Unlike CNNs and LSTMs, Transformers **do not inherently understand the order of a sequence**. To overcome this, we add **positional encodings** to the input data. This helps the model recognize that a specific frame occurs before or after another, preserving the **temporal order of gestures**.

Step 3: Multi-Head Self-Attention

The **self-attention mechanism** allows the Transformer to focus on different parts of the sequence **simultaneously**. For ASL recognition, this is crucial because:

- Some gestures depend on previous hand movements.
- Not all frames contribute equally to recognizing a sign.
- The model needs to distinguish between **similar but different** gestures.

The **multi-head attention** mechanism applies multiple attention layers, allowing the model to learn **different aspects** of the gesture in parallel.

Step 4: Feed-Forward Network (FFN)

After attention processing, each vector passes through a **feed-forward neural network (FFN)** to further refine feature extraction. The FFN is applied **independently to each frame**, improving the model's ability to recognize **complex hand movements**.

Step 5: Classification Output

3. Why Use Transformers for ASL Recognition?

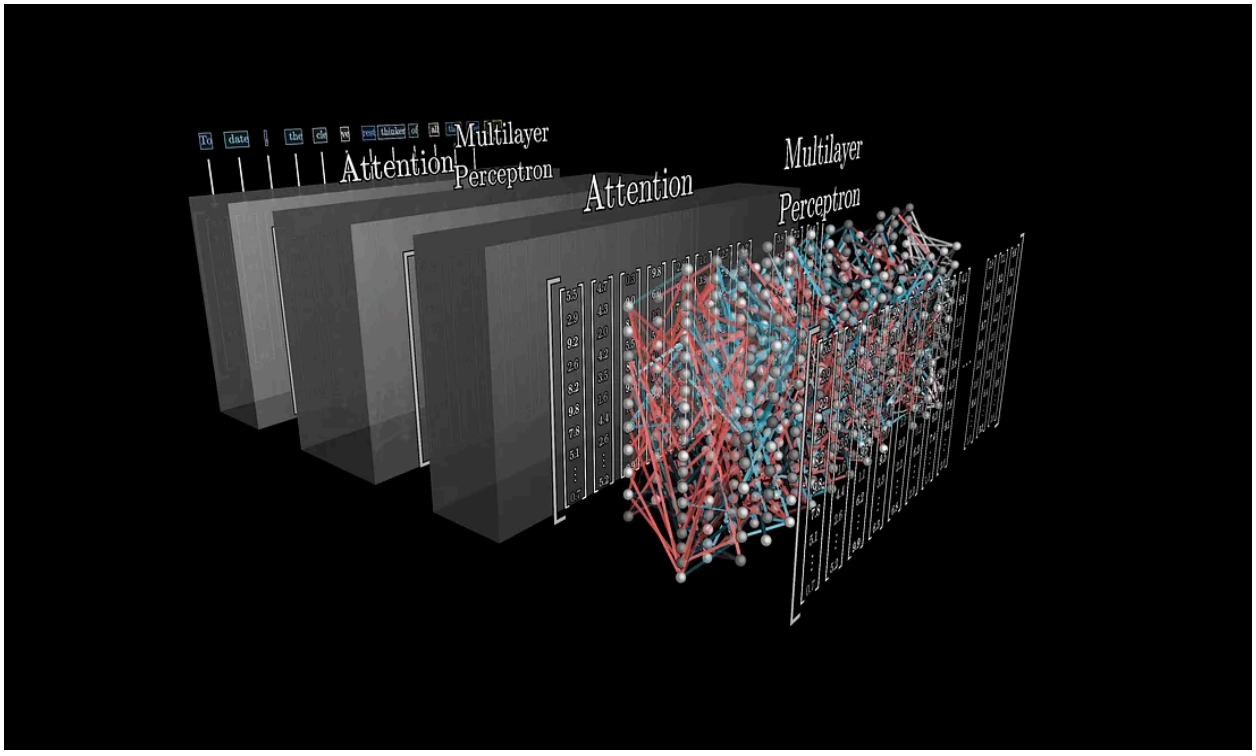
Advantages of using Transformer models over traditional approaches:

Better Context Understanding – The self-attention mechanism allows the model to recognize long-range dependencies between frames, improving classification accuracy.

Parallel Processing – Unlike LSTMs, which process sequences **step by step**, Transformers analyze **entire sequences simultaneously**, leading to faster inference.

Improved Accuracy – Transformers can learn **complex hand movements and variations**, reducing misclassification errors.

Scalability – The model can be adapted to recognize a **large vocabulary of signs**, making it suitable for real-world ASL applications.



3-3 Static Hand Prediction

```
    def detect_and_crop_hand(self, frame):
        img_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        results = hands.process(img_rgb)
        if results.multi_hand_landmarks:
            for hand_landmarks in results.multi_hand_landmarks:
                h, w, _ = frame.shape
                x_min, y_min, x_max, y_max = w, h, 0, 0

                for lm in hand_landmarks.landmark:
                    x, y = int(lm.x * w), int(lm.y * h)
                    x_min, y_min = min(x_min, x), min(y_min, y)
                    x_max, y_max = max(x_max, x), max(y_max, y)

                padding = 20
                x_min = max(0, x_min - padding)
                y_min = max(0, y_min - padding)
                x_max = min(w, x_max + padding)
                y_max = min(h, y_max + padding)

                hand_crop = frame[y_min:y_max, x_min:x_max]
                if hand_crop.shape[0] > 0 and hand_crop.shape[1] > 0:
                    hand_crop = cv2.resize(hand_crop, (224, 224))
                    return hand_crop, (x_min, y_min, x_max, y_max)
        return None, None
```

This function detects a hand in a video frame, crops it, and resizes it to **224×224 pixels**, which is commonly used for deep learning models like **MobileNetV2**.

```
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)

train_generator = datagen.flow_from_directory(
        PROCESSED_PATH,
        target_size=(224, 224),
        batch_size=32,
        class_mode='categorical',
        subset='training'
)

val_generator = datagen.flow_from_directory(
        PROCESSED_PATH,
        target_size=(224, 224),
        batch_size=32,
        class_mode='categorical',
        subset='validation'
)
```

This code sets up **data augmentation and preprocessing** for training a **CNN (MobileNetV2) model** using **Keras' `ImageDataGenerator`**.

Key Points:

1. **Data Normalization:**
   - `rescale=1./255` scales pixel values from **[0, 255]** to **[0, 1]** (helps CNN training).
2. **Dataset Splitting:**
   - `validation_split=0.2` reserves **20%** of the data for validation.
3. **Data Loading:**
   - `train_generator`: Loads the **training** set (80%).
   - `val_generator`: Loads the **validation** set (20%).
4. **Batch Processing:**
   - Uses **batch size = 32** for efficient training.
5. **Class Mode:**
   - `class_mode='categorical'` for **multi-class classification** (each sign language gesture is a category).

```
base_model = MobileNetV2(input_shape=(224, 224, 3), include_top=False, weights="imagenet")
base_model.trainable = False  # Freeze pre-trained layers

x = Flatten()(base_model.output)
x = Dense(512, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(28, activation='softmax')(x)  # 27 classes

model = Model(inputs=base_model.input, outputs=x)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train model
model.fit(train_generator, validation_data=val_generator, epochs=10, verbose=1)

# Save model
model.save("asl_model7.h5")
print("Model saved successfully!")
```

Loads MobileNetV2 (pre-trained on ImageNet).

include_top=False → Removes the final classification layers.

trainable=False → Freezes pre-trained layers (only the new layers will be trained).

Flatten(): Converts feature maps to a 1D vector.

Dense(512, activation='relu'): Adds a fully connected layer with 512 neurons.

Dropout(0.5): Prevents overfitting by randomly dropping 50% of neurons.

Dense(28, activation='softmax'): Outputs 28 probabilities (one for each class).

Optimizer: adam (adaptive learning for faster convergence).

Loss Function: categorical_crossentropy (for multi-class classification).

Metric: accuracy (to measure performance).

3-4 Motion Hand Prediction

```python
def load_data() -> tuple:
    sequences, labels = [], []
    for action in ACTIONS:
        for sequence in range(NO_SEQUENCES):
            window = []
            for frame_num in range(SEQUENCES_LENGTH):
                file_path = os.path.join(DATA_PATH, action, str(sequence), f"{frame_num}.npy")
                try:
                    res = np.load(file_path)
                    # Ensure the feature size is 1662 (truncate or pad if necessary)
                    if res.shape[0] > FEATURE_SIZE:
                        res = res[:FEATURE_SIZE]  # Truncate to 1662 features
                    elif res.shape[0] < FEATURE_SIZE:
                        res = np.pad(res, (0, FEATURE_SIZE - res.shape[0]))  # Pad with zeros
                    window.append(res)
                except FileNotFoundError as e:
                    print(f"Error loading file: {e}")
                    # If file is missing, append a zero array of size 1662
                    window.append(np.zeros(FEATURE_SIZE))
            sequences.append(window)
            labels.append(LABEL_MAP[action])

    # Convert sequences to a numpy array and ensure the shape is (num_sequences, SEQUENCES_LENGTH, FEATURE_SIZE)
    sequences = np.array(sequences)
    print(f"Shape of sequences: {sequences.shape}")  # Debugging: Check the shape
    return sequences, to_categorical(labels).astype(int)
```

This function **loads preprocessed sign language data** from `.npy` files and structures it into sequences for training a machine learning model. It ensures **each sequence is properly formatted** by handling missing or inconsistent feature sizes.

```python
def create_transformer_model(input_shape: tuple) -> Model:
    inputs = Input(shape=input_shape)

    # Multi-Head Attention Layer
    attention = MultiHeadAttention(num_heads=8, key_dim=64, dropout=0.2)(inputs, inputs)
    attention = LayerNormalization()(attention)

    # Add Residual Connection
    attention = Add()([attention, inputs])  # Adding residual connection
    attention = LayerNormalization()(attention)

    # Feed-Forward Network with Layer Normalization and Dropout
    ffn = Dense(512, activation='relu')(attention)
    ffn = Dropout(0.3)(ffn)
    ffn = Dense(256, activation='relu')(ffn)
    ffn = Dropout(0.3)(ffn)

    # Another Attention Layer for capturing more complex features
    attention2 = MultiHeadAttention(num_heads=8, key_dim=64, dropout=0.2)(ffn, ffn)
    attention2 = LayerNormalization()(attention2)
    attention2 = Add()([attention2, ffn])
    attention2 = LayerNormalization()(attention2)

    # Flatten and Output Layer
    flatten = Flatten()(attention2)
    output = Dense(ACTIONS.shape[0], activation='softmax')(flatten)

    model = Model(inputs=inputs, outputs=output)
    model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy',
                  metrics=['categorical_accuracy'])
    return model
```

This layer **accepts input sequences** (frames with features).

`input_shape`: The shape of each input sample (e.g., `(30, 1662)`, meaning 30 frames, each with 1662 features).

Multi-Head Attention allows the model to **focus on different parts** of the input sequence at once.

`num_heads=8`: Uses **8 attention heads** to learn multiple representations.

`key_dim=64`: Each attention head has **64-dimensional key/query embeddings**.

`dropout=0.2`: Helps prevent **overfitting**.

```

    # Multi-Head Attention Layer
    attention = MultiHeadAttention(num_heads=8, key_dim=64, dropout=0.2)(inputs, inputs)
    attention = LayerNormalization()(attention)

```

Another **residual connection** and **Layer Normalization** ensure better gradient flow.

```
    # Add Residual Connection
    attention = Add()([attention, inputs])  # Adding residual connection
    attention = LayerNormalization()(attention)
```

A **fully connected network** enhances the model's ability to learn **complex patterns**.

Uses **ReLU activation** for non-linearity.

Dropout (0.3) is added to **reduce overfitting**.\

```

    # Feed-Forward Network with Layer Normalization and Dropout
    ffn = Dense(512, activation='relu')(attention)
    ffn = Dropout(0.3)(ffn)
    ffn = Dense(256, activation='relu')(ffn)
    ffn = Dropout(0.3)(ffn)

```

A second Multi-Head Attention layer refines features extracted by the **FFN**.

Another **residual connection** and **Layer Normalization** ensure better gradient flow.

```
    # Another Attention Layer for capturing more complex features
    attention2 = MultiHeadAttention(num_heads=8, key_dim=64, dropout=0.2)(ffn, ffn)
    attention2 = LayerNormalization()(attention2)
    attention2 = Add()([attention2, ffn])
    attention2 = LayerNormalization()(attention2)
```

Flattens the tensor to feed into a **fully connected layer**.

The **output layer** has neurons equal to the **number of actions (ACTIONS.shape[0])**.

Uses **Softmax activation** for **multi-class classification**.

Adam optimizer (`learning_rate=0.0001`) for adaptive learning.

Categorical Crossentropy Loss since this is a **multi-class classification task**.

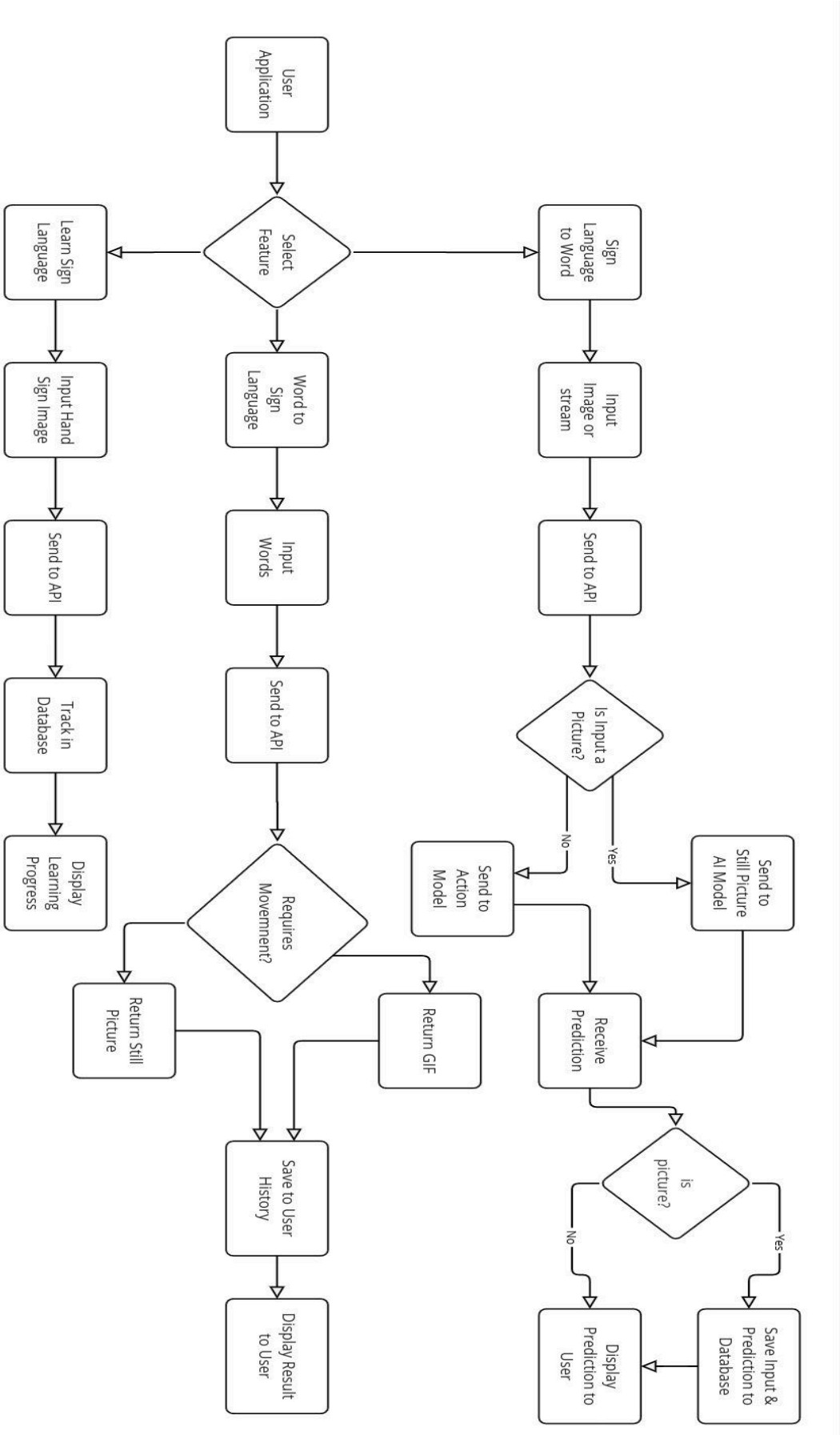
Categorical Accuracy to measure performance.

```
    # Flatten and Output Layer
    flatten = Flatten()(attention2)
    output = Dense(ACTIONS.shape[0], activation='softmax')(flatten)

    model = Model(inputs=inputs, outputs=output)
    model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy',
                  metrics=['categorical_accuracy'])
```

Chapter 4 | API & Applications

4-1 Introduction & Block Diagram

4-1-1 Block Diagram


User Application
Select Feature
Sign Language to Word
Input Image or stream
Send to API
Is Input a Picture?
No
Send to Action Model
Yes
Send to Still Picture AI Model
Receive Prediction
is picture?
No
Display Prediction to User
Yes
Save Input & Prediction to Database
Word to Sign Language
Input Words
Send to API
Requires Movemment?
Return Still Picture
Return GIF
Save to User History
Display Result to User
Learn Sign Language
Input Hand Sign Image
Send to API
Track in Database
Display Learning Progress

4-1-2 Main Introduction

Our project is dedicated to assisting individuals with **special needs**, specifically those who are **deaf and mute**. Through our program, we aim to give them a **voice** by enabling **communication via their mobile phones & desktop computers**. This is achieved through a **seamless integration of AI technology** and a **robust backend infrastructure** powered by an **API (Application Programming Interface)**. The API serves as the **core communication bridge**, connecting the mobile and desktop applications with **advanced AI models** that **detect and interpret sign language**.

The API is built with **Flask**, a **lightweight and flexible web framework** in **Python**. Flask was chosen for its **simplicity and scalability**, allowing us to efficiently manage **user authentication**, **data processing**, and **communication** between the mobile and desktop applications. Its **modular design** enabled **seamless integration** with our **AI models** for sign language detection, ensuring **fast and reliable real-time processing**. By leveraging Flask, we created a **robust backend** that enhances **performance**, **security**, and provides an **efficient platform** for users to **communicate effortlessly**.

Each user can **create an account** in the mobile & desktop applications using their **name, email, and password**. The **registration and authentication processes** are securely managed through the API, ensuring **user data protection** and **streamlined access**. Once registered, users can access a variety of **features designed to enhance their daily interactions**.

4-1-2 Features

One of the **key features** of the program is its ability to **convert a user's gestures into text**. This is made possible through **real-time video streaming**, where the **Flask API** efficiently handles the **video data**, sending it to the **AI model** for processing. The **interpreted gestures** are then **converted into speech**, allowing users to **communicate effectively** in various situations, such as **shopping**, **seeking assistance**, or **social interactions**.

Additionally, the program includes a feature that **converts text into sign language movements**, enabling users to **understand and become educated** on spoken conversations more easily. This is facilitated by the **Flask API**, which processes the **text input** and triggers the corresponding **sign language animations** on the user's device.

To further support **learning and communication**, the program offers a **letter recognition assistance** feature, helping users **familiarize themselves with the Arabic alphabet**. The **AI model**, connected via the **Flask API**, **recognizes the user's hand gestures** and **verifies the letters**, ensuring **accurate learning experiences**. Each interaction is **logged through the API**, recording the **date**, **time**, and **type of feature used**. This allows users to **track their history** and **manage their records**, with the flexibility to **delete specific entries** or **clear their history** entirely.

The application also includes a **sign language learning section**, where users can **practice and learn all the Arabic letters**. Each letter comes with **five attempts**, and users are considered to have **learned a letter** if they succeed in at least **three attempts**. Otherwise, they can **retry learning the letter** or

restart the learning process. The **Flask API** keeps track of **users' progress**, ensuring a **personalized and effective learning experience**.

To ensure **accessibility** for a **broader audience**, the program is available in both **Arabic and English**, offering **English as the global language** while ensuring accessibility for **Arabic-speaking users across Arab countries**. The application also includes a **support feature**, allowing users to **seek help** or **report issues directly** through the **API**. Additionally, an **"About the Program"** section provides **comprehensive details** about the application, its **features**, and its **intended purpose**.

By integrating these features through a **powerful Flask API**, our goal is to **make life easier and more efficient** for individuals with **speech and hearing impairments**. We aim to **empower them to communicate seamlessly** with others and **navigate daily tasks with greater independence**. Through this project, we strive to **foster inclusivity** and provide **meaningful support** to those who rely on **sign language for communication**.

4-2 API

The **Flask API** acts as the central hub **connecting** the AI model, the mobile (Flutter) application, and the desktop application, **enabling real-time communication** and data processing. It provides endpoints for **user authentication**, **image prediction**, learning progress tracking, history management, and more. Powered by **SQLite3 technology**, the API utilizes a **database** for efficient **data storage** and retrieval. This ensures reliable management of user data, learning progress, and history records. The API supports the **backend logic** for **detecting and translating** Arabic Sign Language, ensuring **seamless integration** and interaction between the frontend applications and the AI model.

Main code:

- Database Initialization & Account model creation

```
from flask_sqlalchemy import SQLAlchemy

_db = SQLAlchemy()

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' +
os.path.join(app.root_path, 'database.sqlite3')

app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = True

class Account(_db.Model):
    __tablename__ = 'accounts'
    id = _db.Column(_db.Integer, primary_key=True)
    username = _db.Column(_db.String(100), nullable=False)
    email = _db.Column(_db.String(100), nullable=False)
    password = _db.Column(_db.String(100), nullable=False)
    history = _db.Column(_db.String(2000), nullable=False)
    learn_progress = _db.Column(_db.String(2000),
nullable=False)
    image_path = _db.Column(_db.String(200), nullable=True)
```

Initializes the database and creates the template of account

- User Authentication and Account Management

  Endpoints:

  - /ajax/signup, /ajax/login, /ajax/accountdata

Authentication - Signup Endpoint

```python
@app.route("/ajax/signup", methods=["POST"])
def back_signup():
    username = request.json["username"]
    email = request.json["email"]
    password = request.json["password"]
    image_base = request.json.get("image_base")

    if not all([username, email, password]):
        return jsonify({"error": "All fields are required"}), 400

    if Account.query.filter_by(username=username).first() or
Account.query.filter_by(email=email).first():
        return jsonify({"error": "Username or email already exists"}), 400

    if image_base != None:
        image_filename = f"{uuid4().hex}.png"
        image_path = os.path.join(app.config['UPLOAD_FOLDER'], image_filename)
        with open(image_path, "wb") as f:
            f.write(base64.b64decode(image_base))
    else:
        image_filename = None

    new_acc = Account(
        username=username, email=email, password=password, image_path=image_filename,
        history="[]", learn_progress='[{"Name":"أ","Description":"حرف أ هو أول حروف اللغة
العربية","Done":"None"},{"Name":"ب","Description":"حرف ب هو ثاني حروف اللغة العربية","Done":"None"}]')


    _db.session.add(new_acc)
    _db.session.commit()

    return jsonify({'message': 'Account created'}), 200
```

Handles the creation of new accounts to the database

Authentication - Login endpoint

```
@app.route("/ajax/login", methods=["POST"])
def back_login():
    r = request.json
    user = Account.query.filter_by(email=r["email"]).first()
    if user:
        if r["password"] == user.password:
            url = url_for('static', filename=f'profile_images/{user.image_path}',
_external=True) if user.image_path else None
            return jsonify({'message': 'Correct', "id": user.id, "username": user.username,
"email": user.email, "pfp_url": url}), 200
        else:
            return jsonify({'message': 'Wrong password'}), 401
    else:
        return jsonify({'message': 'Not found'}), 404
```

Handles logging in and the authentication of password and username

Authentication - Account data Endpoint

```
@app.route("/ajax/accountdata", methods=["GET"])
def back_getaccountdata():
    r = request.json

    if "id" in r.keys():
        user = Account.query.filter_by(id=r["id"]).first()
    elif "email" in r.keys():
        user = Account.query.filter_by(email=r["email"]).first()
    elif "username" in r.keys():
        user = Account.query.filter_by(username=r["username"]).first()

    if not user:
        return jsonify({'message': 'Not found'}), 404

    url = url_for('static', filename=f'profile_images/{user.image_path}', _external=True) if
user.image_path else None

    return jsonify({'message': 'Correct', "id": user.id, "username": user.username, "email":
user.email, "pfp_url": url}), 200
```

Returns the data of specified user

- Image Receiving and Prediction Endpoints

  Endpoints:

  - /predict, /predict_motion

Prediction - Static Hand Prediction Endpoint

```python
@app.route('/predict', methods=['POST'])
def predict_page():
    try:
        data = request.json.get('image')
        user_id = request.json.get('id')
        user = Account.query.filter_by(id=user_id).first()

        if not user:
            return jsonify({'error': 'User not found'}), 404
        if not data:
            return jsonify({'error': 'No image'}), 400

        image_data = base64.b64decode(data)
        np_image = np.frombuffer(image_data, np.uint8)
        frame = cv2.imdecode(np_image, cv2.IMREAD_COLOR)

        os.makedirs(os.path.join("static", "prediction_images"), exist_ok=True)
        image_path = os.path.join("prediction_images", f"{uuid4().hex}.png").replace("\\",
"/")
        cv2.imwrite(os.path.join("static", image_path.replace("/", "\\")), frame)

        if frame is None:
            return jsonify({'error': 'Failed to decode image'}), 400

        prediction = predict_sign(frame)

        if prediction is None:
            print("[ERROR] No valid prediction")
            return jsonify({'error': 'No hand detected'}), 400

        os.makedirs(os.path.join("static", "prediction_images"), exist_ok=True)
        image_path = os.path.join("prediction_images", f"{uuid4().hex}.png").replace("\\",
"/")
        cv2.imwrite(os.path.join("static", image_path.replace("/", "\\")), frame)

        his = eval(user.history)
        if request.json.get("type") == "Sign Language To Words":
            his.append({
                "id": len(his) + 1,
                "date": datetime.datetime.now().strftime("%d-%m-%Y %H:%M"),
                "prediction": prediction,
                "type": "Sign Language To Words",
                "image": url_for('static', filename=image_path, _external=True)
            })

        user.history = str(his)
        _db.session.commit()

        return jsonify({'prediction': prediction}), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Processes received image, adds prediction to user’s history and returns prediction

Prediction - Motion Hand Prediction Endpoint

```
@socketio.on("video_stream", namespace="/motion_stream")
def handle_video_stream(frame_data):
    global sequence

    img_data = base64.b64decode(frame_data)
    np_arr = np.frombuffer(img_data, np.uint8)
    frame = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)

    with mp_holistic.Holistic(min_detection_confidence=0.5, min_tracking_confidence=0.5) as
holistic:
        image, results = mediapipe_detection(frame, holistic)

        keypoints = extract_landmark_data(results)
        sequence.append(keypoints)
        sequence = sequence[-SEQUENCE_LENGTH:]

        if len(sequence) == SEQUENCE_LENGTH:
            res = motion_model.predict(np.expand_dims(sequence, axis=0))[0]
            prediction = MOTION_ACTIONS[np.argmax(res)]
            print(prediction)

            socketio.emit("prediction", prediction)
```

Using the library socket-io, the endpoint opens a video stream between the server and the applications, the server listens for frames sent from live stream and process the frames through the motion prediction AI model and sends the prediction back to the applications, this loop continues until the connection is closed then adds prediction to user’s history

- Text to Sign Language

  Endpoints:

  - /ajax_text_to_signlanguage, /predict_motion

Prediction - Static Hand Prediction Endpoint

Turns received arabic text into a GIF of the corresponding sign language gestures.

```
def generate_gif(sentence):

    letter_dir = "./letters/"
    available_files = {os.path.splitext(f)[0]: f for f in
os.listdir(letter_dir)}
    sorted_keys = sorted(available_files.keys(), key=len,
reverse=True)

    sentence = sentence.lower()
    image_files = []
    notfound = []
    used_words = set()

    for phrase in sorted_keys:
        if phrase in sentence and phrase not in used_words:
            image_files.append(os.path.join(letter_dir,
available_files[phrase]))
            sentence = sentence.replace(phrase, "", 1).strip()
            used_words.add(phrase)

    for word in sentence.split():
        if word in available_files and word not in used_words:
            image_files.append(os.path.join(letter_dir,
available_files[word]))
        else:
            notfound.append(word)

    if notfound:
        return jsonify({'message': f"{', '.join(notfound)} not in
the database."}), 204

    images = [Image.open(img).convert("RGBA") for img in
image_files[::-1]]
    file_name = f"signlanguage_gifs/{uuid4().hex[:6]}.gif"
    file_path = f"./static/{file_name}"
    images[0].save(file_path, save_all=True,
append_images=images[1:], duration=500, loop=0)

    file_url = url_for('static', filename=file_name,
_external=True)
    return file_url
```

Function that gathers saved images in a folder and makes a gif

Main endpoint function

```
@app.route("/ajax/text_to_signlanguage", methods=["POST"])
def text_to_signlanguage():
    user =
Account.query.filter_by(id=request.json.get("id")).first()
    if not user:
        return jsonify({'message': 'Not found'}), 404

    sentence = request.json.get("sentence")
    file_url = generate_gif(sentence)

    his = eval(user.history)
    his.append({
        "id": len(his) + 1,
        "date": datetime.datetime.now().strftime("%d-%m-%Y
%H:%M"),
        "prediction": sentence,
        "type": "Words To Sign Language",
        "image": file_url
    })
    user.history = str(his)
    _db.session.commit()

    return jsonify({'message': 'Correct', 'file_name':
file_url}), 200
```

The endpoint calls the **generate_gif** function to generate a gif of the words from images in a folder

4-3 Mobile Application

4-3-1 What Do We Use to Make a Mobile Application?

To develop a **mobile application**, we use **Flutter**, an **open-source UI framework** by **Google**. Flutter enables developers to build **high-performance**, **cross-platform applications** using a **single codebase**, making development **faster** and more **efficient**.

Flutter offers a **rich widget library**, allowing for **modern and customizable UI designs**. Its **fast rendering engine** ensures **smooth performance**, and the **hot reload feature** enables **instant updates** during development.

Built using **Dart**, Flutter provides **smooth animations**, **flexible UI designs**, and **native-like performance**. It also **integrates easily with backend services**, making it a great choice for **scalable applications**.

Another advantage is its ability to support **Android, iOS, web, and desktop** from a **single codebase**, reducing **development time and effort**. Flutter's **powerful tools** allow developers to create **feature-rich**, **high-quality apps** with a **seamless user experience**.

Photo of the Dart & Flutter logos

4-3-2 Pros of Using Flutter for Mobile Applications

1. Cross-Platform Development – A single codebase works for Android, iOS, web, and desktop, saving time and effort.
2. Hot Reload for Fast Development – Instantly see code changes without restarting the app.
3. Beautiful UI – A rich widget library allows for customizable, modern designs.
4. High Performance – Uses Dart and Skia rendering engine for smooth animations.
5. Single Codebase – Reduces development and maintenance costs.
6. Strong Community & Google Support – Regular updates and an active developer community.
7. Backend Integration – Supports APIs, cloud storage, and authentication.
8. Open Source & Free – Constant improvements from the global developer community.
9. Less Testing Effort – A single codebase reduces the need for platform-specific testing.
10. Ideal for Complex Apps – Great for applications requiring animations and advanced UI.

4-3-4 What are the packages used (with version)?

Cupertino_icons: 1.0.5

- Description: This package provides a set of high-quality icons following the iOS design language (Cupertino style) for use in Flutter applications.

animated_splash_screen: 1.3.0

- Description: A Flutter package that allows developers to create customizable animated splash screens with various transition effects.

image_picker: 1.0.0

- Description: A Flutter plugin that enables the selection of images and videos from the device's gallery or camera.

video_player: 2.1.15

- Description: A Flutter plugin for displaying inline video with support for both network and local videos.

url_launcher: 6.1.10

- Description: A Flutter plugin for launching URLs in the mobile platform. Supports web, phone, SMS, and email schemes.

http: 0.13.5

- Description: A composable, Future-based library for making HTTP requests in Dart.

awesome_dialog: 2.1.1

- Description: A Flutter package that helps to create beautiful and customizable dialogs with various animations.

shared_preferences: 2.0.15

- Description: A Flutter plugin for reading and writing simple key-value pairs to persistent storage**.**

camera: ^0.10.0

- Enables camera access for capturing photos/videos in Flutter apps.

socket_io_client: ^3.0.2

- Facilitates real-time communication with a Socket.IO server.

4-3-5 Overview and pictures of mobile application

First Page

Signup Page

Login Page

Drawer Page

Page Choose Type

Sign Language To Words Page

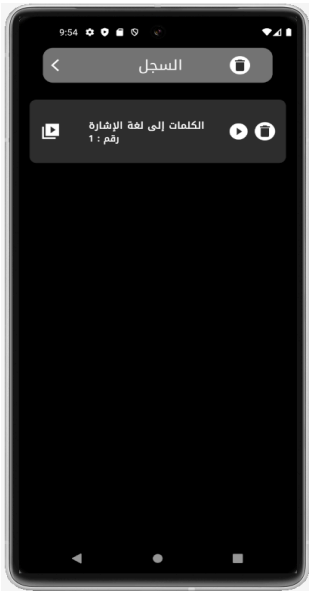
Words To Sign Language Page

Category Learn Page

Letters Learn Page

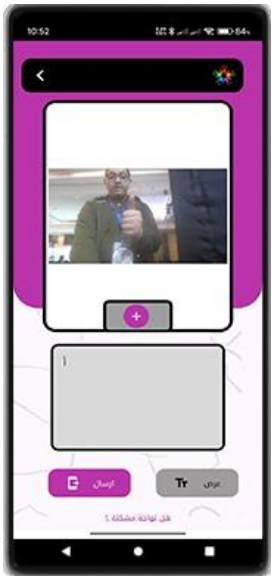
Learn Page

Data of Words To Sign Language



History Page

History Data



Sign Language To Words Page

4-3-6 Key Points About the Codebase Purpose

- **Core Features :**
  1. **Sign Language to Words** : Converts gestures (images/videos) to text.
  2. **Words to Sign Language :** Converts text to sign language (GIFs).
  3. **Learn Letters :** Practice and learn sign language letters.
  4. **History :** Tracks and displays user interactions.

- **Multi-LanguageSupport:** Supports Arabic and English , Language-specific text is managed in Language.dart.

- **Technologies Used :**
  1. **Frontend:** Flutter (Dart).
  2. **Packages:** image_picker, video_player, url_launcher and more.
  3. **Backend:** API integration for processing and data management (Flask).

Error Handling :

1. Feedback via Snack Bars and Dialogs.
2. Handles API errors and invalid user inputs.

Code Structure :

1. **Screens :** SignLanguageToWords, WordsToSignLanguage, LearnLetter, History, Settings.
2. **Data Management:** Language.dart, crud.dart, linksapp.dart.
3. **Utilities:** Image/video picking, validation, navigation.

4-3-7 Code :

4-3-7-1 API Functions Template for Code Implementation :

1. **getRequest()**: Retrieves data from an **API** endpoint. For example, to **fetch** sign language data for a word, provide the **URL** of the target page.
2. **postRequest()**: **Sends data** to an **API** endpoint for saving or processing. For instance, to submit login credentials, send a dictionary containing **key-value pairs** along with the target **URL**.

```
import 'package:http/http.dart' as http;
import 'dart:convert';

class API {
  getRequest(String url) async {
    try {
      var response = await http.get(Uri.parse(url));
      if (response.statusCode == 200) {
        var responsebody = jsonDecode(response.body);
        return responsebody;
      } else {
        print("Error : ${response.statusCode}");
      }
    } catch (e) {
      print("Error is : $e");
    }
  }

  Future<Map<String, dynamic>?> postRequest(
      String url, Map<String, dynamic> data) async {
    try {
      var response = await http.post(Uri.parse(url),
          headers: {'Content-Type':
'application/json'},
          body: jsonEncode(data));
      if (response.statusCode == 200) {
        return json.decode(response.body);
      } else {
        print("Error: ${response.statusCode}");
        return null;
      }
    } catch (e) {
      print("Exception: $e");
      return null;
    }
  }
}
```

4-3-7-1-1 Links of project :

```
const String linkServerName = "http://192.168.176.58:5000";   //  Not Fixed

const String linkSignUp = "$linkServerName/ajax/signup";
const String linkLogin = "$linkServerName/ajax/login";

const String linkHistory = "$linkServerName/ajax/get_user_history";
const String LinkGetLetters = "$linkServerName/ajax/get_saved_letters";
const String LinkSendVideoModel = "$linkServerName/predict_motion";
const String LinkModel = "$linkServerName/predict";

const String linkLearn = "$linkServerName/ajax/learn";
const String linkGetLearn = "$linkServerName/ajax/get_user_progress";
const String linkResetLearn = "$linkServerName/ajax/reset_learn";
const String linkResetAllLearn = "$linkServerName/ajax/clear_all_letters";

const String linkTextToSign = "$linkServerName/ajax/text_to_signlanguage";

const String LinkDeleteHistory = "$linkServerName/ajax/delete_history_item";
const String LinkDeleteAllHistory = "$linkServerName/ajax/clear_all_history";
```

4-3-7-2 Sign Up Page :

This page validates the inputs for **name**, **profile picture**, **email,** and **password**

If any errors are detected, an error message is returned.

If all inputs are valid, the data is sent to the API, and the **onSignupSuccess()** function is triggered.

There are two options for adding a profile picture :

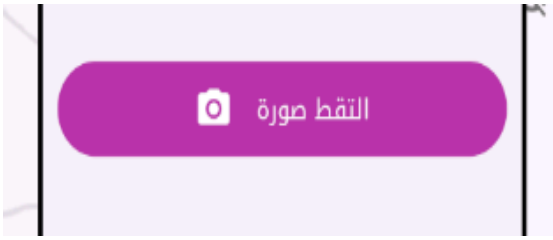
- **Pick Image From Gallery :**

  A function that **retrieves images** from **the phone's storage**.


```dart
// Pick an image from the gallery
_pickImageFromGallery() async {
  final XFile? pickedFile = await picker.pickImage(
    source: ImageSource.gallery,
    imageQuality: 50,
  );

  if (pickedFile != null) {
    setState(() {
      _image = File(pickedFile.path);
    });
  }
}
```

- **Pick Image From Camera :**

  A function that **captures images** using **the phone's camera.**




```dart
// Pick an image from the camera
  _pickImageFromCamera() async {
    final XFile? pickedFile = await picker.pickImage(
      source: ImageSource.camera,
      imageQuality: 50,
    );

    if (pickedFile != null) {
      setState(() {
        _image = File(pickedFile.path);
      });
    }
  }
```

```
// Function to handle signup
  void _signup() async {
    if (_formKey.currentState!.validate()) {
      // Validate form inputs
      if (widget.image == null) {
        // Check if an image is selected
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text(
              widget.isArabic ? "أدخل الصورة" : 'Please select an image',
              style: TextStyle(
                  fontFamily: widget.isArabic ? "Arabic-sans" : "Racing-sans"),
            ),
            backgroundColor: Colors.red,
          ),
        );
        return;
      }

      widget.isLoading = true; // Set loading state to true
      setState(() {});

      Uint8List imageBytes =
          await widget.image!.readAsBytes(); // Read image as bytes
      var response = await api.postRequest(linkSignUp, {
        // Send signup request
        "image_base": base64Encode(imageBytes), // Encode image to base64
        "username": widget.nameController.text, // Get username
        "email": widget.emailController.text, // Get email
        "password": widget.passwordController.text, // Get password
      });

      widget.isLoading = false; // Set loading state to false
      setState(() {});

      if (response?['message'] == 'Account created') {
        // Check if account creation was successful
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text(
              widget.isArabic ? "تم انشاء الحساب" : response?['message'],
```

```
            style: TextStyle(
                fontFamily: widget.isArabic ? "Arabic-sans" : 'Racing-sans'),
          ),
          backgroundColor: Color(0xFFBA33AB),
        ),
      );
      widget.onSignupSuccess(); // Trigger success callback
    } else {
      // Handle signup failure
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text(
            widget.isArabic
                ? "فشل انشاء الحساب"
                : response?['message'] ?? "Eroor",
            style: TextStyle(
                fontFamily: widget.isArabic ? "Arabic-sans" : 'Racing-sans'),
          ),
        ),
      );
    }
  }
}
```

onSignupSuccess() Function:

It **navigates** to the login page, **closes** the signup page, and **hides** the language switch button.

```
// Handle sign-up success
void _onSignupSuccess() {
  setState(() {
    _login = true;
    _signup = false;
    _change_language = false;
  });
}
```

4-3-7-3 Login Page :

This function takes an email and password as input, sends them to the API, and verifies the existence of the associated account. If found, the user's ID, profile image, email, and name are stored, the `onLoginSuccess()` function is triggered, and a **"Login Success"** message is displayed.

If the credentials are incorrect, the message

"Username or password incorrect" is shown.

```
// Function to handle login
  void _login() async {
    if (_formKey.currentState!.validate()) {
      // Validate form inputs
      widget.isLoading = true; // Set loading state to true
      if (mounted) setState(() {});

      var response = await api.postRequest(linkLogin, {
        // Send login request
        "email": _emailController.text, // Get email
        "password": _passwordController.text, // Get password
      });

      widget.isLoading = false; // Set loading state to false
      if (mounted) setState(() {});

      if (response?['message'] == 'Correct') {
        // Check if login was successful
        sharedPref.setString("id",
            response!['id'].toString()); // Save user ID to shared preferences
        sharedPref.setString("pfp_url",
            response!['pfp_url'].toString()); // Save profile picture URL
        sharedPref.setString(
            "username", response!['username']); // Save username
        sharedPref.setString("email", response!['email']); // Save email
        print("Login sucsess: ${response?['message']}");

        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            duration: Duration(seconds: 1),
            content: Text(
              widget.isArabic ? "تسجيل ناجح" : 'Login Sucsessful',
              style: TextStyle(
                  fontFamily: widget.isArabic ? 'Arabic-sans' : 'Racing-sans'),
            ),
            backgroundColor: Color(0xFFBA33AB),
          ),
        );
```

```
        widget.onLoginSuccess(); // Notify the parent widget of successful login
      } else {
        // Handle login failure
        print("Login failed: ${response?['message']}");
        AwesomeDialog(
          context: context,
          dialogType: DialogType.error,
          title: widget.isArabic ? "مشكله" : 'Error',
          desc: widget.isArabic
              ? "اسم المستخدم أو كلمه المرور غير صحيحه"
              : 'Username or password incorrect',
          btnOkOnPress: () {},
          dialogBackgroundColor: Color(0xFFBA33AB),
          titleTextStyle: TextStyle(
              color: Colors.white,
              fontFamily: widget.isArabic ? "Arabic-sans" : "Racing-sans"),
          descTextStyle: TextStyle(
              color: Colors.white,
              fontFamily: widget.isArabic ? "Arabic-sans" : "Racing-sans"),
        ).show();
      }
    }
  }
```

4-3-7-3-1 onLoginSuccess() Function :

This function closes both the login and signup pages and navigates to the **"Choose Type"** page. It **expands** the background container, **hides** the language switch button to prevent issues, **updates** the logo and icon with new images, and **displays** the user's name.

```
  // Handle login success
  void _onLoginSuccess() {
    setState(() {
      _login = false;
      _signup = false;
      showPageChoose = true;
      heightBackGround = 320;
      _change_language = false;
      iconn = "images/ASLP Logo3.png";
      logo = "images/menu_icon2.jpeg";
      _image = null;
      _Name = true;
    });
  }
```

4-3-7-4 Get All History :

This function **retrieves and displays** all history records from the **API**.

A User ID must be provided to **fetch** the corresponding data.

```
// Function to fetch history data from the API
  fetchHistory() async {
    var response = await api.postRequest(linkHistory, {
      'id': sharedPref.getString('id') // Send user ID to fetch history
    });

    if (response == null) {
      // Handle null response
      setState(() {
        historyList = [];
      });
    } else if (response['message'] == 'Not found') {
      // Handle no history found
      setState(() {
        historyList = [];
      });
    } else {
      // Process response data
      setState(() {
        historyList = List<Map<String, dynamic>>.from(
            response['data']); // Map response data to list
      });
    }
  }
```

4-3-7-5 Clear One History Item :

This function **deletes** a specific history record from the **API**.

To **proceed** with the deletion, both the **User ID** and **History ID** must be provided.

```
// Function to delete a specific history item
  ClearHistory(int id) async {
    var response = await api.postRequest(LinkDeleteHistory, {
      'id': sharedPref.getString('id'), // User ID
      'history_id': id // History item ID
    });

    if (response!['message'] == 'History item deleted successfully') {
      // Handle success
      _showSnackBar(widget.isArabic
          ? 'نجح مسح السجل'
          : 'History Deleted'); // Show success message
      setState(() {
        historyList.clear(); // Clear the list
        fetchHistory(); // Fetch updated history
      });
    } else {
      // Handle failure
      _showSnackBar(widget.isArabic
          ? 'فشل مسح السجل'
          : 'Failed to delete history'); // Show error message
    }
  }
```

4-3-7-6 Clear All History :

This function **deletes** all history records from the **API**.

The **User ID** must be **provided** to perform the deletion.

```
// Function to delete all history items
  ClearAllHistory() async {
    var response = await api.postRequest(LinkDeleteAllHistory, {
      'id': sharedPref.getString('id') // User ID
    });

    if (response == null) {
      // Handle null response
      _showSnackBar(widget.isArabic
          ? 'فشل مسح السجل'
          : 'Failed to delete history'); // Show error message
      return;
    }

    if (response['message'] == 'All history cleared successfully') {
      // Handle success
      _showSnackBar(widget.isArabic
          ? 'نجح مسح السجل'
          : 'History Deleted'); // Show success message
      setState(() {
        historyList.clear(); // Clear the list
      });
    } else {
      // Handle failure
      _showSnackBar(widget.isArabic
          ? 'فشل مسح السجل'
          : 'Failed to delete history'); // Show error message
    }
  }
```

4-3-7-7 Get All Letters :

This function **displays** all letters for learning, indicating which ones have been **completed** and which are **still pending**.

It also **provides** a relearn button for any letter as well as an option to **relearn all letters**.

```
// Function to fetch learning history from the API
  fetchLetters() async {
    var response = await api.postRequest(
      linkGetLearn, // API endpoint for fetching learning history
      {'id': sharedPref.getString('id')}, // Request body with user ID
    );

    if (response == null) {
      // Handle null response
      return;
    }

    if (response['message'] == 'Not found') {
      // Handle no data found
      setState(() {
        LearnList = [];
      });
    } else {
      // Process response data
      setState(() {
        LearnList = List<Map<String, dynamic>>.from(
            response["data"]); // Map response data to list
      });
    }
  }
```

4-3-7-8 Reset Learn One Letter :

This function allows **relearning** of a specific character from the set of **learning characters**.

To **proceed**, the **User ID** and the desired **character** must be provided.

```
// Function to reset learning progress for a specific letter
  reset_learn(String Letter) async {
    var response = await api.postRequest(
      linkResetLearn, // API endpoint for resetting learning progress
      {
        'id': sharedPref.getString('id'),
        'letter': Letter
      }, // Request body with user ID and letter
    );

    if (response == null) {
      // Handle null response
      _showSnackBar('فشل إعادة ضبط جميع الحروف'); // Show error message
      return;
    }

    if (response['message'] == 'Reset Success') {
      // Handle success
      _showSnackBar('نجح إعادة ضبط جميع الحروف'); // Show success message

      setState(() {
        LearnList.clear(); // Clear the list
        fetchLetters(); // Fetch updated data
      });
    } else {
      // Handle failure
      _showSnackBar('فشل إعادة ضبط جميع الحروف'); // Show error message
    }
  }
```

4-3-7-9 Reset Learn All Letters :

This function **enables** the relearning of all learning characters.

To **initiate the process**, the **User ID** must be provided,

and all characters will be **reset for relearning**.

```
// Function to clear learning progress for all letters
  resetAllLetters() async {
    var response = await api.postRequest(
      linkResetAllLearn, // API endpoint for resetting all learning
progress
      {'id': sharedPref.getString('id')}, // Request body with user ID
    );

    if (response == null) {
      // Handle null response
      _showSnackBar('فشل مسح جميع الحروف'); // Show error message
      return;
    }

    if (response['message'] == 'All letters cleared successfully') {
      // Handle success
      _showSnackBar('نجح مسح جميع الحروف'); // Show success message
      setState(() {
        LearnList.clear(); // Clear the list
        fetchLetters(); // Fetch updated data
      });
    } else {
      // Handle failure
      _showSnackBar('فشل مسح جميع الحروف'); // Show error message
    }
  }
```

4-3-7-10 Learn Letter :

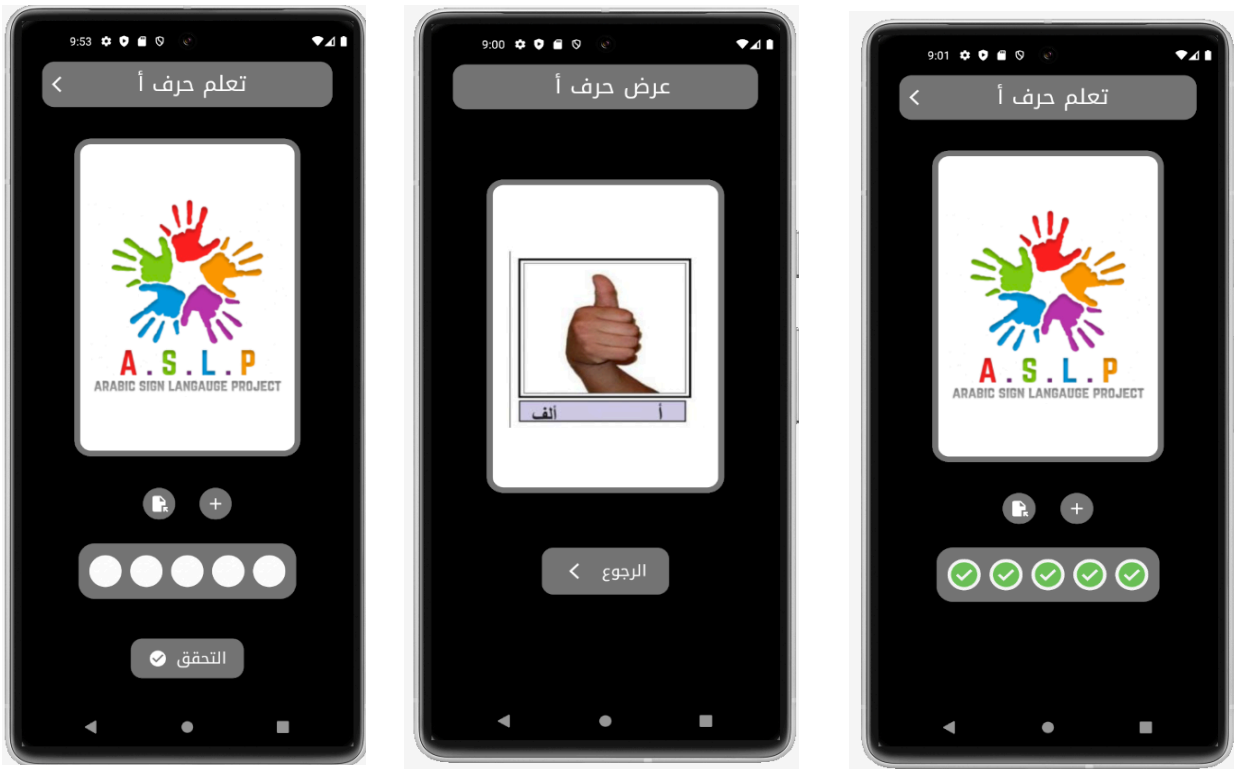
In this process, letters are learned by capturing images and performing hand gestures to form each letter.

If the user correctly performs the letter **3, 4, or 5 times**, **it is considered learned.** However, if the user f**ails to perform the letter correctly** within those attempts, it remains **unlearned.**

Exiting before completing all **5 attempts** also results in the letter

being **unlearned .**

The final results are **recorded**, and the status of each letter is **displayed**, indicating whether it has been **successfully learned** or **not**.



```
// Function to update the learning status in the database
 checkLearn(String Letter, String Done) async {
   var response = await api.postRequest(linkLearn,
       {'letter': Letter, 'id': sharedPref.getString('id'), "done": Done});
 }

 // Function to check the user's attempt
 check() async {
   setState(() {
     widget.isLoading = true; // Set loading state to true
   });
   if (_cameraImage == null) {
     // Check if an image is selected
     setState(() {
       widget.isLoading = false; // Set loading state to false
     });
     _showSnackBar('أدخل صورة أولا'); // Show error message
     return;
   }

   try {
     Uint8List bytes =
         await _cameraImage!.readAsBytes(); // Read image as bytes
     setState(() {
       widget.isLoading = true; // Set loading state to true
     });

     // Send the image to the API for prediction
     var response = await api.postRequest(LinkModel, {
       'image': base64Encode(bytes),
       'type': 'learn',
       'id': sharedPref.getString('id')
     });
     setState(() {
       widget.isLoading = false; // Set loading state to false
     });

     if (response == null) {
       // Handle null response
       setState(() {
         _cameraImage = null; // Clear the selected image
         count += 1; // Increment the attempt count
         // Update the check status based on the attempt count
         if (count == 1) {
           check1 = "False";
         } else if (count == 2) {
           check2 = "False";
         } else if (count == 3) {
           check3 = "False";
         } else if (count == 4) {
           check4 = "False";
         } else if (count == 5) {
           check5 = "False";
         }
       });
```

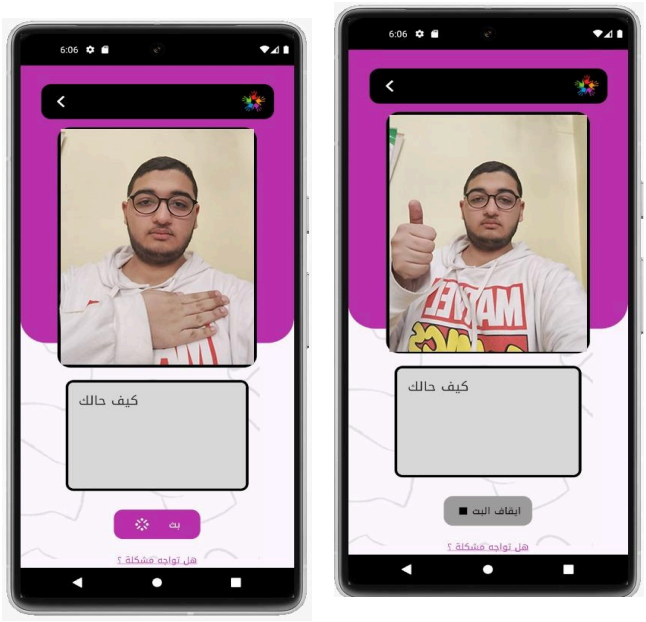
```
  setState(() {
    if (count == 5) {
      // If all attempts are used
      button = false; // Disable the button
      int trueCount = [check1, check2, check3, check4, check5]
          .where((c) => c == 'True')
          .length;
      // Check if the learning is successful based on the check status

      if (trueCount >= 3) {
        checkLearn(widget.Letter, "True"); // Mark learning as successful
      } else {
        checkLearn(widget.Letter, "False"); // Mark learning as failed
      }
    }
  });
  return;
}
```

```
else if (response['prediction'] != null) {
        // Handle successful response
        if (response['prediction'] == widget.Letter) {
          // Check if the prediction matches the letter
          _showSnackBar(
              '${response['prediction']} : الناتج  '); // Show success message
          setState(() {
            _cameraImage = null; // Clear the selected image
            count += 1; // Increment the attempt count
            // Update the check status based on the attempt count
            if (count == 1) {
              check1 = "True";
            } else if (count == 2) {
              check2 = "True";
            } else if (count == 3) {
              check3 = "True";
            } else if (count == 4) {
              check4 = "True";
            } else if (count == 5) {
              check5 = "True";
            }
          });
```

```
      } else {
        // Handle incorrect prediction
        setState(() {
          _cameraImage = null; // Clear the selected image
          count += 1; // Increment the attempt count
          // Update the check status based on the attempt count
          if (count == 1) {
            check1 = "False";
          } else if (count == 2) {
            check2 = "False";
          } else if (count == 3) {
            check3 = "False";
          } else if (count == 4) {
            check4 = "False";
          } else if (count == 5) {
            check5 = "False";
          }
        });
      }
      setState(() {
        if (count == 5) {
          // If all attempts are used
          button = false; // Disable the button
          int trueCount = [check1, check2, check3, check4, check5]
              .where((c) => c == 'True')
              .length;
          // Check if the learning is successful based on the check status

          if (trueCount >= 3) {
            checkLearn(widget.Letter, "True"); // Mark learning as successful
          } else {
            checkLearn(widget.Letter, "False"); // Mark learning as failed
          }
        }
      });
    } else {
      // Handle unexpected response
      _showSnackBar('Unexpected response');
    }
  } catch (e) {
    // Handle exceptions
    setState(() {
      widget.isLoading = false; // Set loading state to false
    });
    _showSnackBar('An error occurred: $e'); // Show error message
  }
}
```

4-3-7-11 Page Sign Language To Words (Word) :

In this function, a **live stream** is opened with the model to process the movements and gestures




```
void _initSocket() {
    // Connect to the Socket.IO server
    socket = IO.io('http://192.168.1.11:5000', <String, dynamic>{
      'transports': ['websocket'],
      'autoConnect': true,
    });

    // Listen for connection events
    socket.onConnect((_) {
      print('Connected to Socket.IO server');
    });

    // Listen for the 'test' event from the server
    socket.on('prediction', (data) {
      print("Received processed from server");
      if (mounted) {
        setState(() {
          _processed = data;
          sharedPref.setString('prediction', data);
        });
      }
    });

    // Listen for disconnection events
    socket.onDisconnect((_) {
      print('Disconnected from Socket.IO server');
    });

    // Handle errors
    socket.onError((error) {
      print("Socket.IO error: $error");
    });
  }
```

```
Future<void> _initCamera() async {
    try {
      final cameras = await availableCameras();

      // Find the front camera
      final frontCamera = cameras.firstWhere(
        (camera) => camera.lensDirection == CameraLensDirection.front,
        // orElse: () => null, // If no front camera is found, return null
      );

      if (frontCamera != null) {
        // Initialize the front camera
        final CameraController cameraController =
            CameraController(frontCamera, ResolutionPreset.low);
        await cameraController.initialize();

        if (!mounted)
          return; // Ensure widget is still mounted before updating state

        setState(() {
          _cameraController = cameraController;
        });
      } else {
        print('No front camera available');
      }
    } catch (e) {
      print('Error initializing camera: $e');
    }
  }
```

```
void _startStreaming() {
    if (_cameraController == null || !_cameraController!.value.isInitialized) {
      print("Camera not initialized yet");
      return;
    }
    isStreaming = true;
    _streamFrames();
  }
```

```
bool _isCapturing = false; // Flag to ensure captures don't overlap

  void _streamFrames() async {
    while (isStreaming) {
      if (_cameraController == null ||
          !_cameraController!.value.isInitialized) {
        print("Camera stopped streaming due to unavailability");
        return;
      }

      if (_isCapturing) {
        // If still capturing, skip this frame
        await Future.delayed(
            Duration(milliseconds: 100)); // Wait before checking again
        continue;
      }

      _isCapturing = true; // Mark the start of capturing

      try {
        final XFile frame = await _cameraController!.takePicture();
        Uint8List imgBytes = await frame.readAsBytes();
        String base64String = base64Encode(imgBytes);

        // Send the base64-encoded frame to the server using the 'video_stream'
event
        socket.emit('video_stream', base64String);
      } catch (e) {
        print('Error capturing frame: $e');
      } finally {
        _isCapturing = false; // Reset the flag after capture
      }

      // Ensure a proper delay between captures to manage FPS
      await Future.delayed(
          Duration(milliseconds: 100)); // Adjust FPS rate as needed
    }
  }

  void _stopStreaming() {
    isStreaming = false;
  }
```

4-3-7-12 Page Words To Sign Language :

This function **processes** a given word or sentence by **retrieving** and **displaying corresponding** word images as a **video** or **GIF**.

To perform the translation, the **User ID** and **the desired word(s)** must be sent to the **API.**

All actions are **recorded** in **the history**.

```
// Function to convert text to sign language
 Future<void> Convert(String sentence) async {
   var response = await api.postRequest(
     linkTextToSign, // API endpoint for text-to-sign conversion
     {'id': sharedPref.getString('id'), "sentence": sentence}, // Request body
   );

   if (sentence.isEmpty) {
     // Handle empty input
     ScaffoldMessenger.of(context).showSnackBar(
       SnackBar(
           duration: Duration(seconds: 1),
           content: Text(
             widget.isArabic ? "لو سمحت أدخل الكلمات" : 'Please enter a words',
             style: TextStyle(
                 fontFamily: widget.isArabic ? "Arabic-sans" : "Racing-sans"),
           )),
     );
   }
   setState(() {
     if (response == null || response['message'] == "Not Found in DataBase") {
       Gif = "images/ASLP Logo4.png"; // Default image if no result is found
     } else if (response['message'] == 'Correct') {
       Gif = response['file_name']; // Set GIF image from response
       print(Gif); // Print GIF path for debugging
     }
   });
 }
```

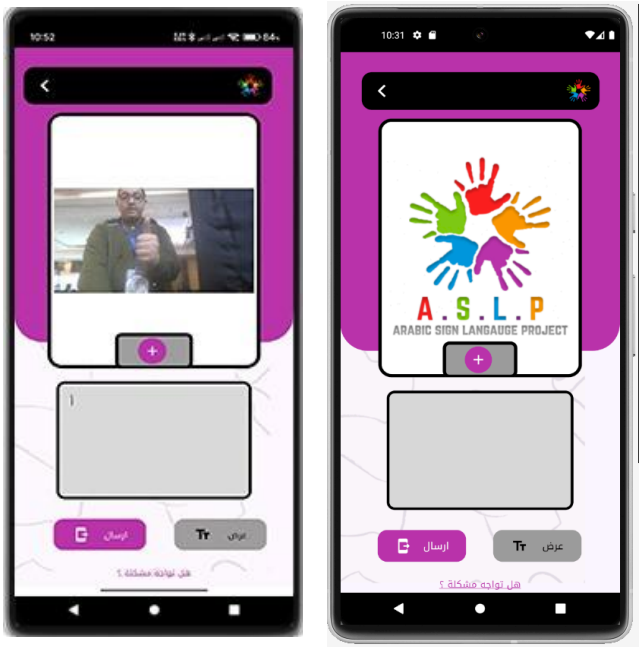
4-3-7-13 Data Of Word To Sign Language :

All the words and letters available for sign language translation combine to help people with special needs, who are deaf and dumb, interact with others easily.


```
// Function to fetch letters from the API
dynamic fetchLetters() async {
  var response =
      await api.getRequest(LinkGetLetters); // Fetch letters from API

  if (response == null) {
    // Handle null response
    setState(() {
      LettersList = [];
    });
  } else {
    // Process response
    setState(() {
      LettersList = List<Map<String, dynamic>>.from(response['data'].map(
          (item) => {
                'word': item[0],
                'image': item[1]
              })); // Map response data to list

      print("$LettersList"); // Print fetched letters for debugging
    });
  }
}
```

4-3-7-14 Page Sign Language To Word (Letter) :

This page enables the conversion of sign language or gestures into a specific letter. The process can be carried out by either selecting an image from the device or capturing one using the camera.

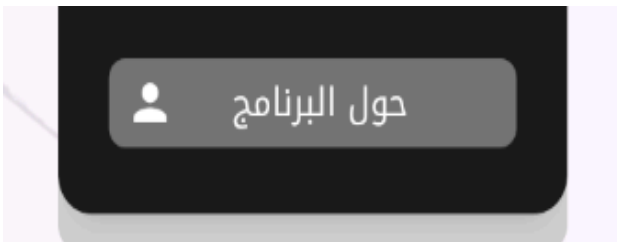



```
void check() async {
  if (widget.image == null) {
    _showSnackBar(widget.isArabic ? "لا يوجد صورة" : 'No image provided');
    return;
  }

  try {
    Uint8List bytes = await widget.image!.readAsBytes();
    setState(() {
      widget.isLoading = true;
    });

    var response = await api.postRequest(LinkModel, {
      'image': base64Encode(bytes),
      'type': "Sign Language To Words",
      'id': sharedPref.getString('id'),
    });
    setState(() {
      widget.isLoading = false;
    });

    if (response == null) {
      _showSnackBar(widget.isArabic
          ? "يوجد مشكله فى الصورة"
          : 'No response from server');
      sharedPref.setString("prediction", "No Result");
      return;
    }
```

```
    if (response['error'] == 'No image') {
      _showSnackBar(
          widget.isArabic ? "يجب أدخال صورة" : 'Please input a valid image');
    } else if (response['error'] == 'No hand detected') {
      sharedPref.setString("prediction", "No Hand Detected");
      _showSnackBar(widget.isArabic ? "لا يوجد يد" : 'No hand detected');
    } else if (response['prediction'] != null) {
      sharedPref.setString("prediction", response['prediction']);
      _showSnackBar('${response['prediction']} : الناتج ');
      result = true;
    } else {
      sharedPref.setString("prediction", "No Result");
      _showSnackBar('Unexpected response');
    }
  } catch (e) {
    setState(() {
      widget.isLoading = false;
    });
    sharedPref.setString("prediction", "No Result");
    _showSnackBar('An error occurred: $e');
  }
}
```

4-3-7-15 About Project :

Clicking this button directs you to a webpage that provides a detailed **overview of the project**, including its objectives, the team behind it, and its **timeline**.



```
// Launch the about page URL
Future<void> _launchAboutPage() async {
  final Uri url = Uri.parse('https://sites.google.com/view/aslp-info/home');
  try {
    if (!await launchUrl(
      url,
      mode: LaunchMode.externalApplication,
    )) {
      throw 'Could not launch $url';
    }
  } catch (e) {
    debugPrint('Error: $e');
  }
}
```

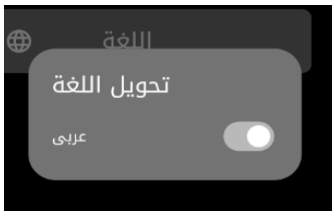
4-3-7-16 Need More Help ?

This button enables you to report any problems to the developers to solve them, and this button makes you go to a group on WhatsApp to solve all the problems


```
  // Function to launch the WhatsApp group link
  Future<void> _launchNeedProblem() async {
    final Uri url = Uri.parse(
        'https://chat.whatsapp.com/GAgm3l2a1zNISYsVYHuWM7'); // WhatsApp group
link
    try {
      if (!await launchUrl(
        url,
        mode: LaunchMode.externalApplication, // Launch in external application
      )) {
        throw 'Could not launch $url';
      }
    } catch (e) {
      debugPrint('Error: $e');
    }
  }
```

4-3-7-17 Change Language :

Pressing this button changes the language of the program between Arabic and English, and this is because the program is dedicated to Arabs, so the main language is Arabic and English because it is the language of the world




```dart
bool isArabic = true; // Default language is Arabic
// Toggle language (Arabic/English)
void updateisArabic() {
  setState(() {
    isArabic = !isArabic;
  });
}
```

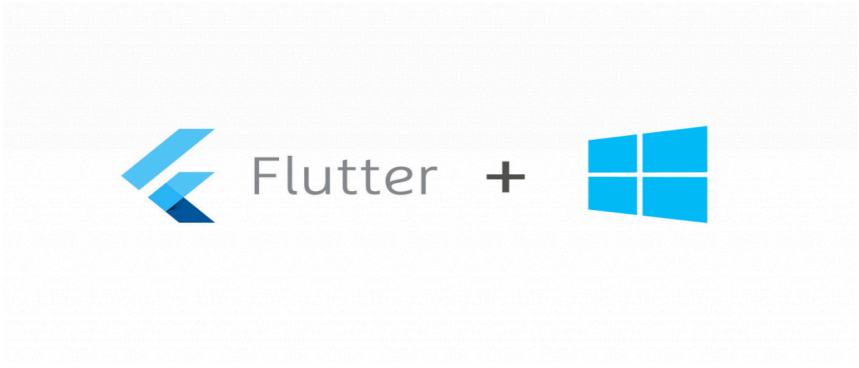
4-4 Desktop Application

4-4-1 What is used to make desktop applications ?

Flutter applications for Windows are primarily built using **Dart**, **C++**, and the **Flutter framework** itself. **Dart** is the **main programming language**, responsible for **defining the UI**, **handling user interactions**, and **managing state** within the application. It provides a **reactive, widget-based approach** that ensures **smooth performance** and **flexibility**.

C++ plays a **crucial role** in the **Flutter engine**, which powers **rendering**, **input handling**, and **communication between Dart and the native Windows platform**. It integrates with **Windows-specific APIs** like **Win32** and **Direct3D** to **render UI elements efficiently**. Developers can also use **C++** to create **native plugins** for **system-level tasks** that require **direct access to Windows functionalities**.

The **Flutter framework** bridges **Dart and native code**, providing **tools for UI development** and **system interaction**. It relies on **platform channels** and the **Foreign Function Interface (FFI)** to enable **communication between Dart and C++** when necessary. This setup allows developers to create **high-performance Windows applications** while maintaining the **flexibility** of **Flutter's cross-platform capabilities**.



4-4-2 Pros of using flutter for windows applications

1. **Cross-Platform Compatibility** – Flutter allows you to write code once and deploy it on Windows, macOS, Linux, Android, and iOS, reducing development effort.
2. **Fast Development with Hot Reload** – The Hot Reload feature lets developers see changes instantly without restarting the entire application, speeding up development.
3. **Beautiful UI with Customization** – Flutter provides a rich set of widgets that enable modern and visually appealing UI designs, with full customization flexibility.
4. **Dart's Performance and Efficiency** – Dart's Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation provide a good balance between development speed and runtime performance.
5. **Good Performance** – Flutter's Skia graphics engine ensures smooth animations and rendering, even on lower-end hardware.
6. **Active Community and Google Support** – Flutter is backed by Google and has a strong developer community, with frequent updates and new features.
7. **Access to Native Windows APIs** – Using FFI (Foreign Function Interface) and platform channels, Flutter apps can interact with native C++ code to access Windows features like file system operations, system dialogs, and hardware interactions.
8. **Open-Source and Free** – Flutter is completely free to use, making it an excellent choice for developers and businesses looking for cost-effective solutions.

4-4-3 What are the packages used (With version)?

animated_splash_screen: ^1.3.0

This package makes it easy to create a splash screen with animations. A splash screen is the first screen that users see when they open your app, and it's often used to display the app's logo or name.

flutter_acrylic: ^1.1.4

This package allows you to add acrylic effects to your app's UI. Acrylic is a visual effect that makes windows appear translucent and blurry, and it's often used in Windows 11.

window_manager: ^0.4.2

This package allows you to control the window of your Flutter app on desktop platforms. You can use it to change the window's size, position, and title, among other things.

url_launcher: ^6.1.8

This package allows you to open URLs in the user's default browser. You can use it to link to websites, send emails, or make phone calls.

Image_picker:^0.8.7+5

This package allows users to select images from their device's gallery or take photos with the camera.

Http:^1.1.0

This package provides a way to make HTTP requests, which is how your app can communicate with web servers. You can use it to fetch data from APIs or send data to them.

camera_windows: ^0.2.6+1

This package provides access to the camera on Windows devices.

Camera: ^0.11.0+2

This package provides access to the camera on Android and iOS devices.

image: ^4.5.2

This package provides a set of tools for working with images. You can use it to resize, crop, and manipulate images.

shared_preferences: ^2.5.1

This package provides a way to store simple data in key-value pairs. This is useful for storing user settings or other small bits of data that you need to persist between app sessions.

video_player: ^2.9.2

This package allows you to play videos in your app.

Path_provider: ^2.1.5

This package provides access to commonly used directories on the device, such as the documents directory or the temporary directory.

video_player_media_kit: ^1.0.5

This package is an alternative video player plugin that uses the media_kit library.

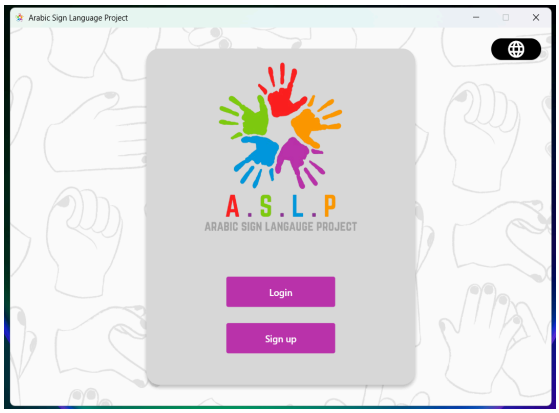
Video_player_win: ^3.1.1

This package provides access to the video player on Windows devices.

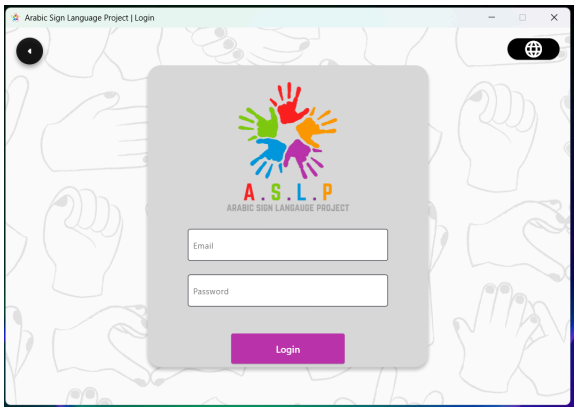
Socket_io_client: ^3.0.2

This package provides a way to connect to Socket.IO servers. Socket.IO is a protocol for real-time, bidirectional communication between clients and servers.

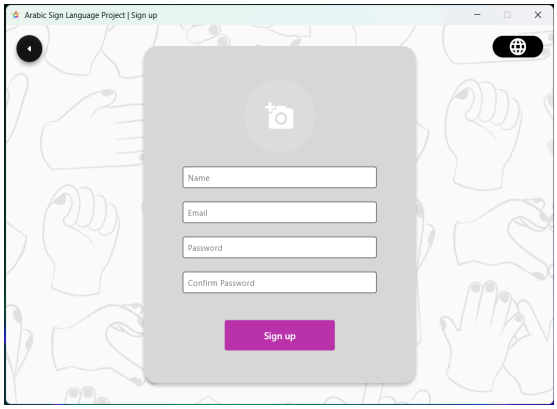
4-4-4 Overview of Desktop Application:



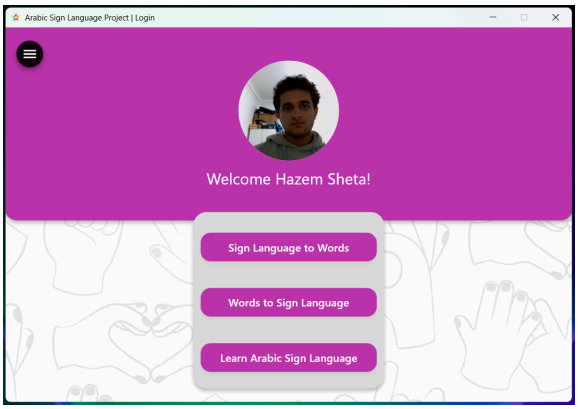
Onboarding Screen



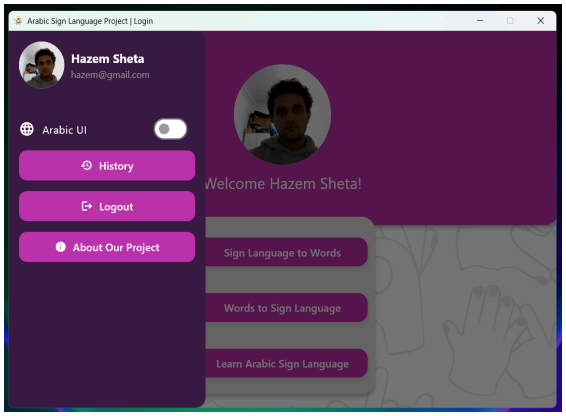
Login Screen



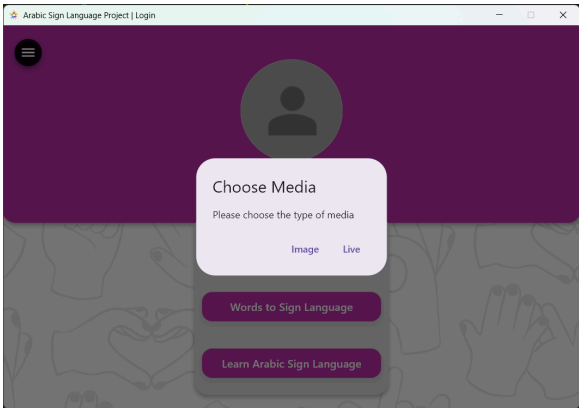
Sign-Up Screen



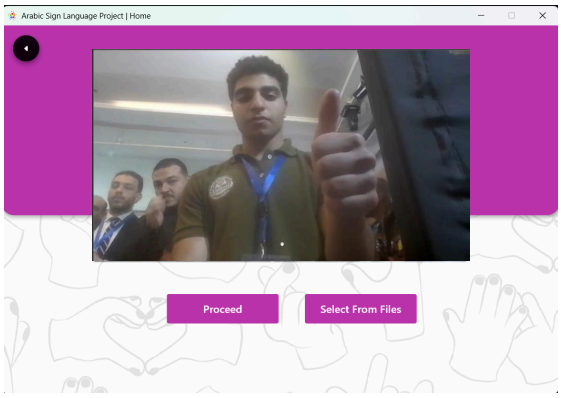
Home Screen



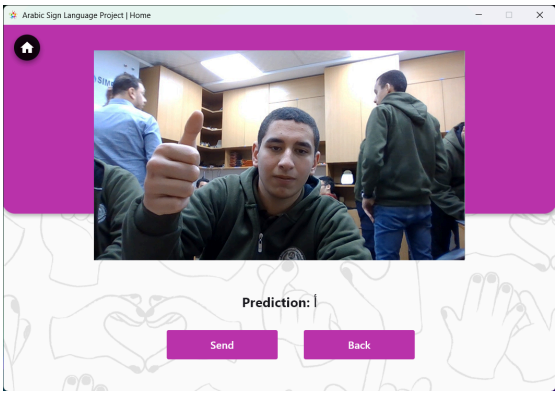
Side Menu



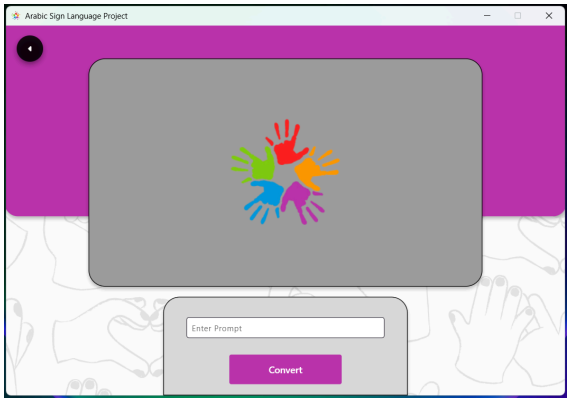
Alert Window
(When entering SL to word)



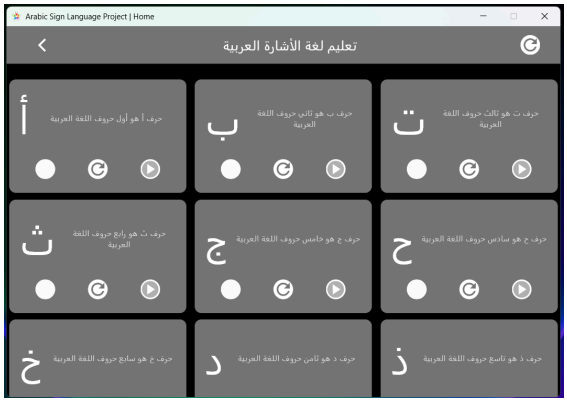
Camera Screen



Prediction Screen



Words to sign language



Learn sign language

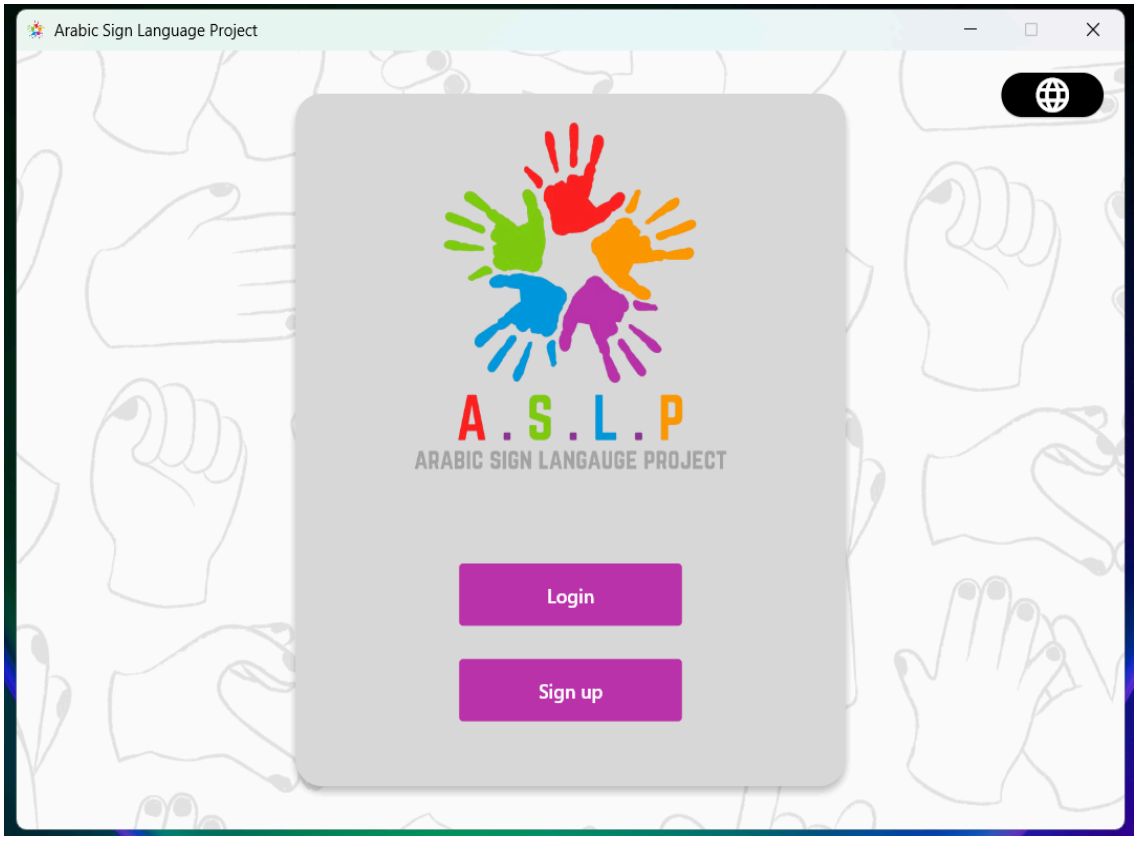
Learn letter Screen

Show letter screen

4-4-5 The making of desktop application:

Onboarding Screen:

The On boarding screen is arguably one of the most important screens as it asks the user to either sign up and make an account or Login with an already made account. There is also a language switch button on the top right which can be seen all throughout the Onboarding, Login and Sign-Up



Important sections of the code:

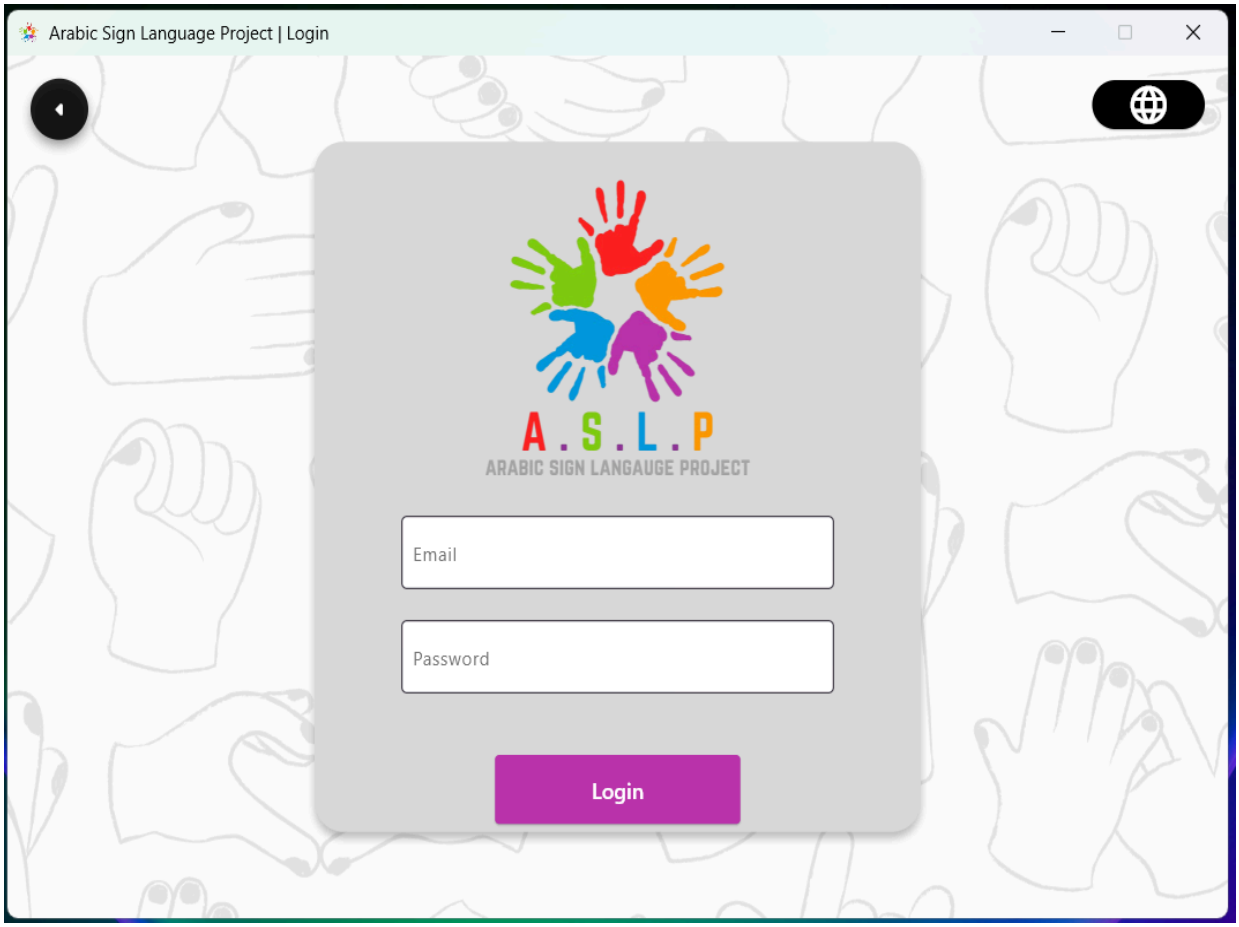
This contains all the window settings like **setSize**, **SetTitle**, etc inside the widget builder which are the most important functions inside this screen and the function of the language switch button with the actual button at the end.

```dart
Future<void> _toggleLanguagePreference() async {
  //Get Language Preference using Shared Preferences package
  final prefs = await SharedPreferences.getInstance();
  isArabic = !(prefs.getBool('isArabic') ?? false);
  //set boolean variable "isArabic" to either true or false
  await prefs.setBool('isArabic', isArabic!);
}
class choice extends StatefulWidget {
  const choice({super.key});

  @override
  State<choice> createState() => _choiceState();
}

class _choiceState extends State<choice> {
  @override
  Widget build(BuildContext context) {
    windowManager.setSize(const Size(860, 600)); // set size to 860 x 600
    windowManager.setResizable(false); //set windows to unresizable
    windowManager.setTitleBarStyle(TitleBarStyle.normal); //set title bar to normal
    windowManager.setTitle("Arabic Sign Language Project"); //set name for window
    windowManager.setAlignment(Alignment.center); // align window to center
    //apply effect
    Window.setEffect(
      effect: WindowEffect.mica,
      dark: false,
```

```dart
Positioned(
            top: 16,
            right: 16,
            child: ElevatedButton(
                onPressed: () async {
                  await _toggleLanguagePreference();
                  setState(() {});
                  if (isArabic!) {
                    ScaffoldMessenger.of(context).showSnackBar(
                      const SnackBar(content: Text('تم التبديل بنجاح!')),
                    );
                  } else {
                    ScaffoldMessenger.of(context).showSnackBar(
                      const SnackBar(content: Text('Switched Successfully!')),
                    );
                  }
                },
                style: ElevatedButton.styleFrom(backgroundColor: Colors.black),
                child: const Icon(
                  Icons.language,
                  color: Colors.white,
                  size: 30,
                )),
          ),
```

4-4-5-2 Login Screen:

The login screen has **2 Text fields** one for the user's **email** and the other for the **password** and a **Login button** at the end which allows the user to login to his account if they have entered their **credentials** correctly.



Important sections of the code:

After the user is logged in, the information is called from the api and stored into global variables that can be used across the application.

ie.. Name, Email, ID, and Picture.

2 Text Editing Controllers are made for **Email** and **Password** text fields where each one has their own **Focus node** to understand where the user is focused (Either typing in Email text field or Password text field).

```dart
// Global variables to store user information
String? globalName;
String? globalEmail;
String? globalPFP;
int? globalID;
// Function to toggle language preference and save it in shared preferences
Future<void> _toggleLanguagePreference() async {
  final prefs = await SharedPreferences.getInstance();
  isArabic = !(prefs.getBool('isArabic') ?? false);
  await prefs.setBool('isArabic', isArabic!);
}

// Login widget
class Login extends StatefulWidget {
  const Login({super.key});

  @override
  State<Login> createState() => _LoginState();
}

class _LoginState extends State<Login> {
  // Controllers for email and password input fields
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();

  // Focus nodes for email and password input fields
  final FocusNode _emailFocusNode = FocusNode();
  final FocusNode _passwordFocusNode = FocusNode();

  // Form key to validate the form
  final _formKey = GlobalKey<FormState>();
```

The Email text field is made so it **dynamically** switches the **hint text** on its own when the **user changes** the **language** to **Arabic** or back to **English**.

The **Controller** that was made for the text field is used on this field to save the user **Email** to the **Global Email** variable.

The **focus node** allows the user to move on from the focused text field right after pressing enter to the **next** field (Password field).

```
Form(
 key: _formKey,
 child: Column(
   children: [
     // Email Field
     Padding(
       padding: const EdgeInsets.symmetric(horizontal: 20),
       child: SizedBox(
         height: 48,
         width: 300,
         child: TextFormField(
           controller: _emailController,
           focusNode: _emailFocusNode,
           textInputAction: TextInputAction.next,
           decoration: InputDecoration(
             hintText:
                 isArabic! ? "البريد الإلكتروني" : "Email",
             hintStyle: TextStyle(
               color: Colors.grey[600],
             ),
             border: OutlineInputBorder(
               borderRadius: BorderRadius.circular(4),
             ),
             filled: true,
             fillColor: Colors.white,
             contentPadding: const EdgeInsets.all(8),
           ),
           validator: (value) {
             if (value == null || value.isEmpty) {
               return isArabic!
                   ? "يرجى إدخال بريد إلكتروني"
                   : 'Please enter an email';
             }
             if (!value.contains("@")) {
               return isArabic!
                   ? 'يرجى إدخال بريد إلكتروني صالح'
                   : 'Please enter a valid email';
             }
             return null;
           },
           style: const TextStyle(
             fontFamily: 'Segoe UI',
             fontSize: 12,
             fontWeight: FontWeight.normal,
           ),
           onFieldSubmitted: (_) {
             FocusScope.of(context)
                 .requestFocus(_passwordFocusNode);
           },
         ),
       ),
     ),
     const SizedBox(height: 20),
```

Password **Text field** is also made so it **dynamically** changes the **hint text** depending on the **language** the **user** is **using**.

The **Controller** that was made for the text field is used on this field to send the **Password** of the user to the **API**.

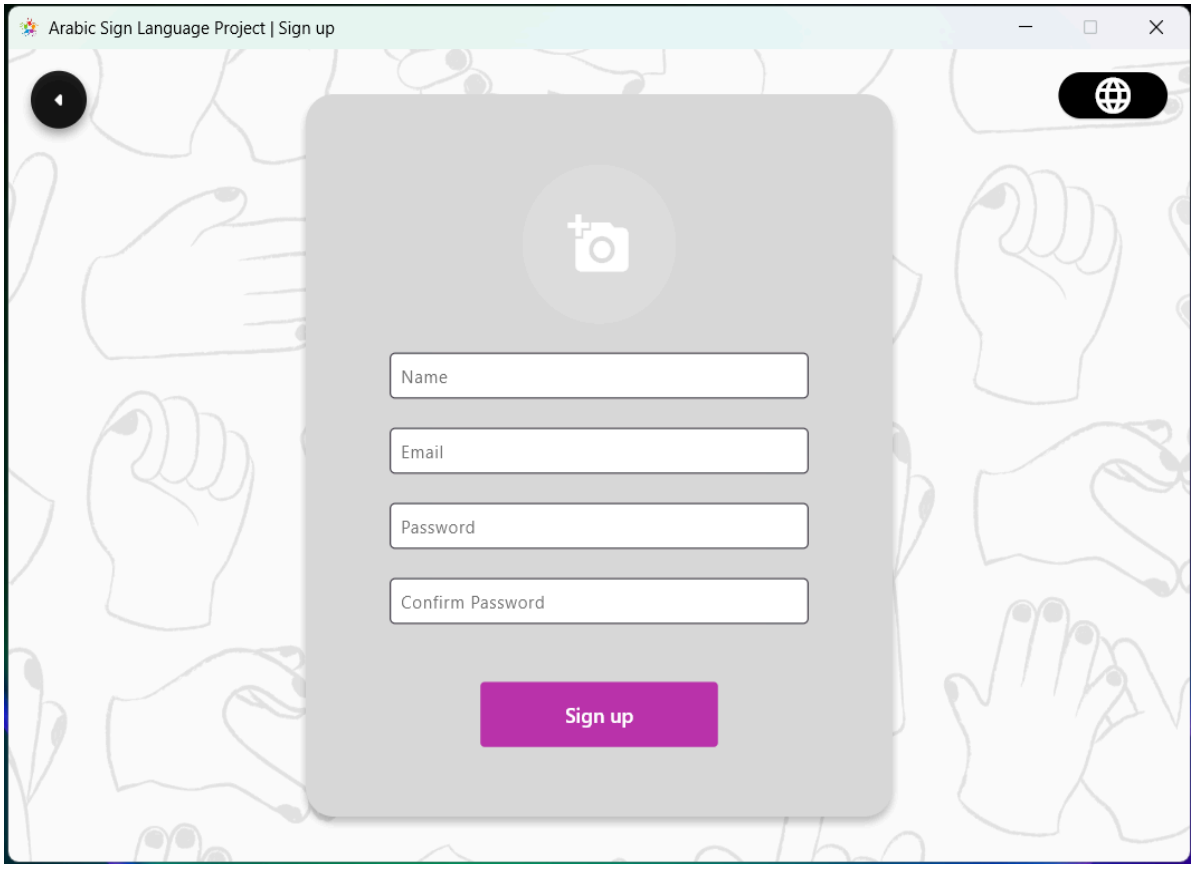
The **focus node** here allows the user to **Login** right after pressing the **Enter** key.

```
Padding(
  padding: const EdgeInsets.symmetric(horizontal: 20),
  child: SizedBox(
    height: 48,
    width: 300,
    child: TextFormField(
      controller: _passwordController,
      focusNode: _passwordFocusNode,
      textInputAction: TextInputAction.done,
      obscureText: true,
      decoration: InputDecoration(
        hintText:
            isArabic! ? "كلمة المرور" : "Password",
        hintStyle: TextStyle(
          color: Colors.grey[600],
        ),
        border: OutlineInputBorder(
          borderRadius: BorderRadius.circular(4),
        ),
        filled: true,
        fillColor: Colors.white,
        contentPadding: const EdgeInsets.all(8),
      ),
      validator: (value) {
        if (value == null || value.isEmpty) {
          return isArabic!
              ? 'يرجى إدخال كلمة مرور'
              : 'Please enter a password';
        }
        return null;
      },
      style: const TextStyle(
        fontFamily: 'Segoe UI',
        fontSize: 12,
        fontWeight: FontWeight.normal,
      ),
      onFieldSubmitted: (_) {
        if (_formKey.currentState!.validate()) {
          _submitForm();
        }
      },
    ),
  ),
),
const SizedBox(height: 40),
```

- **Sign-Up Screen**:

The Sign up screen is made of a **Profile picture** entry and **4 Text fields**, **Name**, **Email**, **Password** and **Confirm Password**.

Lastly **Sign up button** at the bottom which takes all the data that has been entered and delivers it to the **API** so a **new account** is made for the user.



Important sections of the code:

Due to the **amount** of text fields we have opted to use a **builder widget** to build as **many** text fields as **needed** without having **too many lines** used for the same purpose.

```
Widget _buildTextField(String hintText, TextEditingController controller,
    {bool isPassword = false}) {
  return Padding(
    padding: const EdgeInsets.symmetric(horizontal: 20),
    child: SizedBox(
      height: 32,
      width: 300,
      child: TextField(
        controller: controller,
        textInputAction: controller == confirmPasswordController
            ? TextInputAction.done
            : TextInputAction.next,
        focusNode: controller == nameController
            ? _nameFocusNode
            : controller == emailController
            ? _emailFocusNode
            : controller == passwordController
            ? _passwordFocusNode
            : _confirmPasswordFocusNode,
        onSubmitted: (_) {
          if (controller == nameController) {
            FocusScope.of(context).requestFocus(_emailFocusNode);
          } else if (controller == emailController) {
            FocusScope.of(context).requestFocus(_passwordFocusNode);
          } else if (controller == passwordController) {
            FocusScope.of(context).requestFocus(_confirmPasswordFocusNode);
          }
        },
        decoration: InputDecoration(
          hintText: hintText,
          hintStyle: TextStyle(
            color: Colors.grey[600],
          ),
          border: OutlineInputBorder(
            borderRadius: BorderRadius.circular(4),
            borderSide: const BorderSide(color: Colors.grey),
          ),
          filled: true,
          fillColor: Colors.white,
          contentPadding: const EdgeInsets.all(8),
        ),
        obscureText: isPassword,
        style: const TextStyle(
          fontFamily: 'Segoe UI',
          fontSize: 12,
          fontWeight: FontWeight.normal,
        ),
        cursorColor: Colors.blue,
        cursorWidth: 2,
        cursorRadius: const Radius.circular(4),
      ),
    ),

  );
 }
}
```

After the user has entered all their **information**, all of it will get sent to the **API** to make an account for them that is registered inside the database using the following function along with error handling if the user for example,

has not confirmed the password where an error snackbar will pop up saying "**Passwords do not match**." or if the user has left an empty field which will result in a snackbar saying "**Please fill in all the fields**".

```
Future<void> _submitForm() async {
  final name = nameController.text.trim();
  final email = emailController.text.trim();
  final password = passwordController.text.trim();
  final confirmPassword = confirmPasswordController.text.trim();
  final profileImageBase64 = _encodeImageToBase64(_selectedImage);

  if (name.isEmpty || email.isEmpty || password.isEmpty || confirmPassword.isEmpty) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text("Please fill in all the fields.")),
    );
    return;
  }

  if (password != confirmPassword) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        backgroundColor: Colors.red,
        content: Text("Passwords do not match."),

      ),
    );
    return;
  }
  try {
    final response = await http.post(
      Uri.parse(linkSignUp),
      headers: {"Content-Type": "application/json"},
      body: jsonEncode({
        "username": name,
        "email": email,
        "password": password,
        "image_base": profileImageBase64,
      }),
    );

    if (response.statusCode == 200) {
      Navigator.of(context).push(
        MaterialPageRoute(builder: (context) => const Login()),
      );
      if (isArabic!) {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text("تم إنشاء الحساب ، يرجى تسجيل الدخول!")),
        );
      } else {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text("Account Created, Please Login!")),
        );
      }
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text("Error: ${response.body}")),
      );
    }
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("Error: $e")),
    );
  }
}
```

- **Home Screen**:

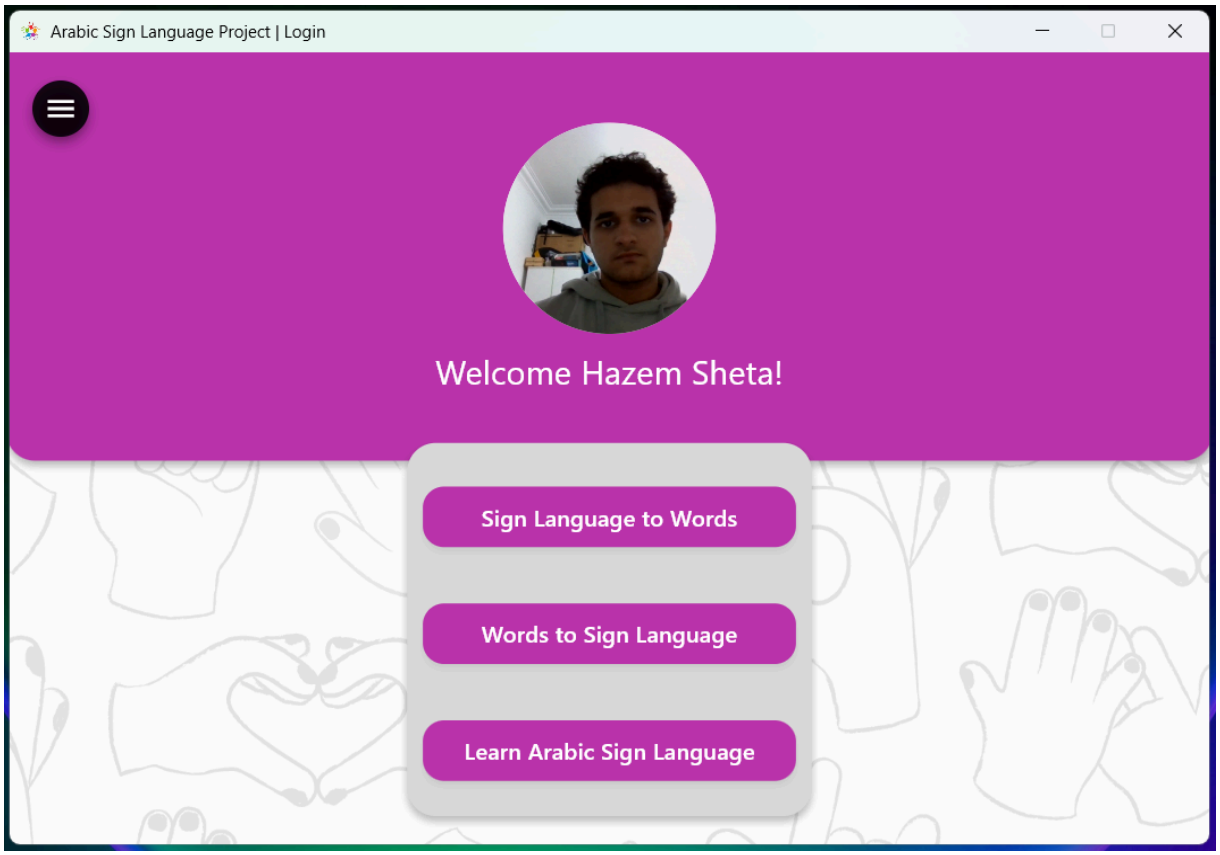
After the user either creates an account and logs in they will be greeted inside the **Home Screen**. The Home Screen **contains 3 buttons** at the **bottom center**.

First, The "**Sign Language to Words**" button where the user will be prompted to pick a media type to go into the camera screen which we will get to later, then be sent to the **API** and have the model predict what the sign language the user has done.

Second, The "**Words to Sign Language**" button where the user will be taken to another screen to input words or letters that will be sent to the **API**, Then **returned** as an **image OR** if there are **multiple** words/letters it will be **returned** as a **GIF**.

Third, The **Most Important Feature** "**Learn Arabic Sign Language**"

Where the user will be taken to a screen to **pick a letter to learn**.

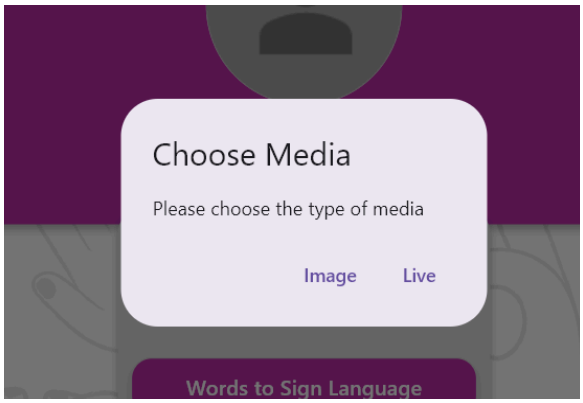


Important sections of the code:

As said before, when pressing the "**Sign Language to Words**" button an **alert box** will appear to make the user **pick** between **either** an **image or Live**.

After the user picks he will be **taken** to a **screen** that **takes a picture** or to a **live screen** that **will stream** the **pictures** depending on the **user's choice**.

```
_buildButton(
text: isArabic! ? "لغة الإشارة إلى الكلمات" :"Sign Language to Words",
onTap: () {
showDialog(
  context: context,
  builder: (BuildContext context) {
    return AlertDialog(
      title: Text(isArabic! ? "اختر الوسائط" : "Choose Media"),
      content: Text(isArabic! ? "يرجى اختيار نوع الوسائط" : "Please choose the type of media"),
      actions: <Widget>[
        TextButton(
          child: Text(isArabic! ? "صورة" : "Image"),
          onPressed: () {
            Navigator.of(context).push(MaterialPageRoute(
              builder: (context) => CameraScreen(),
            ));
          },
        ),
        TextButton(
          child: Text(isArabic! ? "مباشر" : "Live"),
          onPressed: () {
            Navigator.of(context).push(MaterialPageRoute(
              builder: (context) => LiveScreen(),
            ));
          },
        ),
      ],
    );
  },
);
},
),
```



Another **Builder widget** is used here in this screen where it takes a **String** called **text** then **displays** it as the **button text** and an **onTap function** which determines what the button does **when it is pressed**. The function of **onTap** in each button is to take the user to a specific Screen using the navigator built-in function which will take the user to said screen.

```
  Widget _buildButton({required String text, required VoidCallback onTap}) {
    return SizedBox(
      height: 43,
      width: 263,
      child: ElevatedButton(
        onPressed: onTap,
        style: ElevatedButton.styleFrom(
          backgroundColor: const Color(0xffBA33AB),
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(15),
          ),
          shadowColor: Colors.black.withOpacity(0.25),
          elevation: 4,
        ),
        child: Text(
          text,
          style: const TextStyle(
            fontSize: 16,
            fontFamily: "Racing Sans One",
            color: Colors.white,
          ),
        ),
      ),
    );
  }
}
```

```
Navigator.of(context).push(MaterialPageRoute(
  builder: (context) => LiveScreen(),
));
```

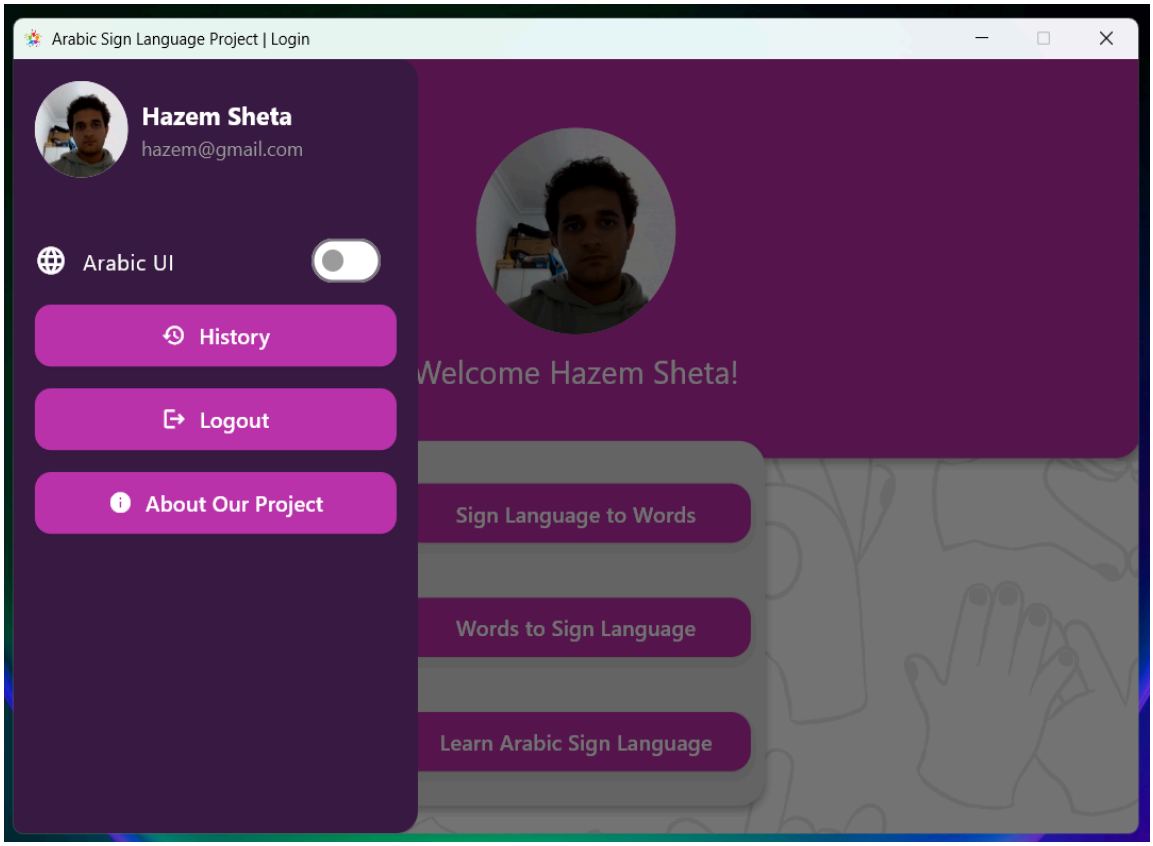
Last but not least, **the Side menu/Drawer** which can be **accessed** by pressing **the 3 line menu icon** in the **top left corner** which will **open** up the menu with **3 buttons** and **one switch**.

First, The **Arabic UI** Switch which when pressed will **switch** the **entire UI** text from **Arabic** to **English**.

Second, The **history** button which will take the **user** to a **screen** that shows all the **past usages** of the app including the **picture/video** used.

Third, a **Logout button** which will **delete all user global variables** like **ID**, **Name**, **Password**, **PFP** and **go back** to the **Onboarding Screen** where the **user** can **make a new account** or **log back in**.

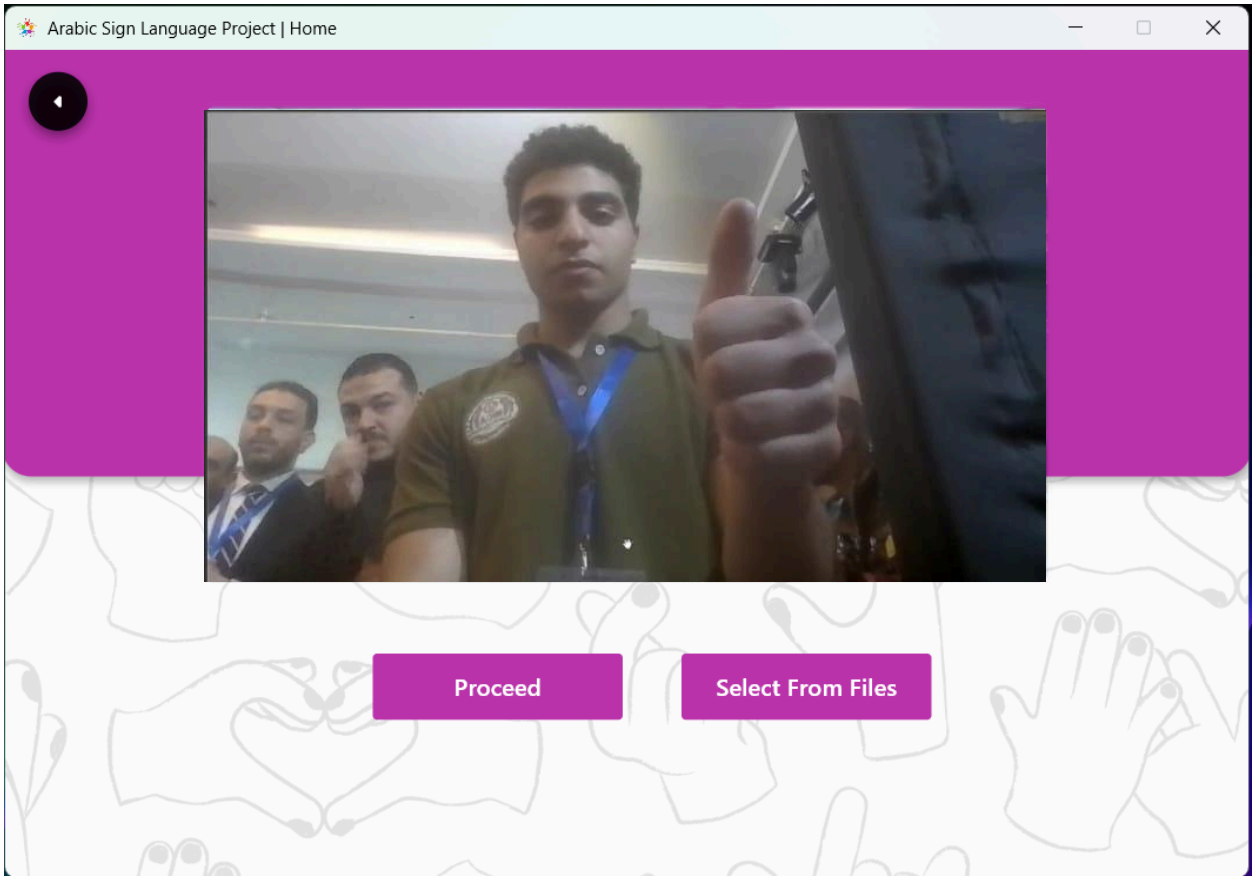
Fourth, is **a button** that will **take the user** to the **project website** made on **google websites** that contains all the **general project information**. (`https://sites.google.com/view/aslp-info/home`)



- **Camera Screen**:

After the user picks what type of **media** they want to **input** they will be taken inside the **Camera Screen** where their **webcam** will **turn on** and then be displayed in the middle with **2 buttons** below to **Capture** or pick an **Image/Video** from the **storage** on the **computer**.

Once the **capture button** is pressed it will **display** the image or if its a video the **capture button** will turn into a **stop recording button** which will then also **display** the **video** in place of the **Camera preview**.



Important sections of the code:

First we must **initialize** important **variables** so we can use the **camera** and make a **File** variable called **_selectedImage**, **_controller** and **ImagePicker**.

```
File? _selectedImage;
final ImagePicker _imagePicker = ImagePicker();
late CameraController _controller;
late List<CameraDescription> _cameras;
bool _isCameraInitialized = false;

@override
void initState() {
  super.initState();
  _initializeCamera();
}
```

To take a picture we must initialize the camera by making a controller for the camera and make it check for the first available camera on the computer, if an error does happen it will display said error in-place of the Camera preview.

```
Future<void> _initializeCamera() async {
  try {
    _cameras = await availableCameras();
    if (_cameras.isEmpty) {
      throw Exception('No cameras available');
    }
    _controller = CameraController(
      _cameras[0], // Select the first available camera
      ResolutionPreset.high,
    );
    await _controller.initialize();
    setState(() {
      _isCameraInitialized = true;
    });
  } catch (e) {
    print('Error initializing camera: $e');
  }
}
```

After the **camera** is **initialized** the user can now **take** a **picture** or **video** using **said webcam** but a **minor issue** has to be **resolved**. **This issue** occurs in the **flutter Camera package** where it **displays** the **picture** taken **opposite** to the **preview** shown **before** taking the picture.

This issue can be solved by **manipulating** the **image** and **flipping it** using the **flutter Image package**.

All **images** are **saved** inside of the users **pictures folder** so they can be **viewed** by the **user outside the app** at any time on the computer.

```dart
Future<void> _captureImage() async {
  if (_isCameraInitialized) {
    try {
      final XFile? image = await _controller.takePicture();
      if (image != null) {
        // Read the new image file
        File originalFile = File(image.path);
        Uint8List imageBytes = await originalFile.readAsBytes();

        // Decode the image
        img.Image? decodedImage = img.decodeImage(imageBytes);
        if (decodedImage != null) {
          // Flip the image horizontally
          img.Image flippedImage = img.flipHorizontal(decodedImage);

          // Get the directory to save the modified image
          final directory = await getTemporaryDirectory();
          String timestamp = DateTime.now().millisecondsSinceEpoch.toString();
          String flippedImagePath = '${directory.path}/flipped_$timestamp.jpg';

          // Save the flipped image
          File flippedFile = File(flippedImagePath);
          await flippedFile.writeAsBytes(img.encodeJpg(flippedImage));

          // Update UI
          setState(() {
            _selectedImage = flippedFile;
          });
        }
      }
    } catch (e) {
      print("Error capturing image: $e");
    }
  }
}
```

If the user decides to **pick** an **image** off the storage of the computer they will be greeted with a **file picker window** to pick the **media** they wish to send.

This is **achieved** in flutter by using the **Image_Picker package**.

```
Future<void> _pickImage() async {
  final XFile? image =
  await _imagePicker.pickImage(source: ImageSource.gallery);
  if (image != null) {
    setState(() {
      _selectedImage = File(image.path);
    });
  }
}
```

After a **media** is **taken/selected** the **user** can then press the "**Proceed**" **button** to navigate to the **Result screen** with the **selected media**.

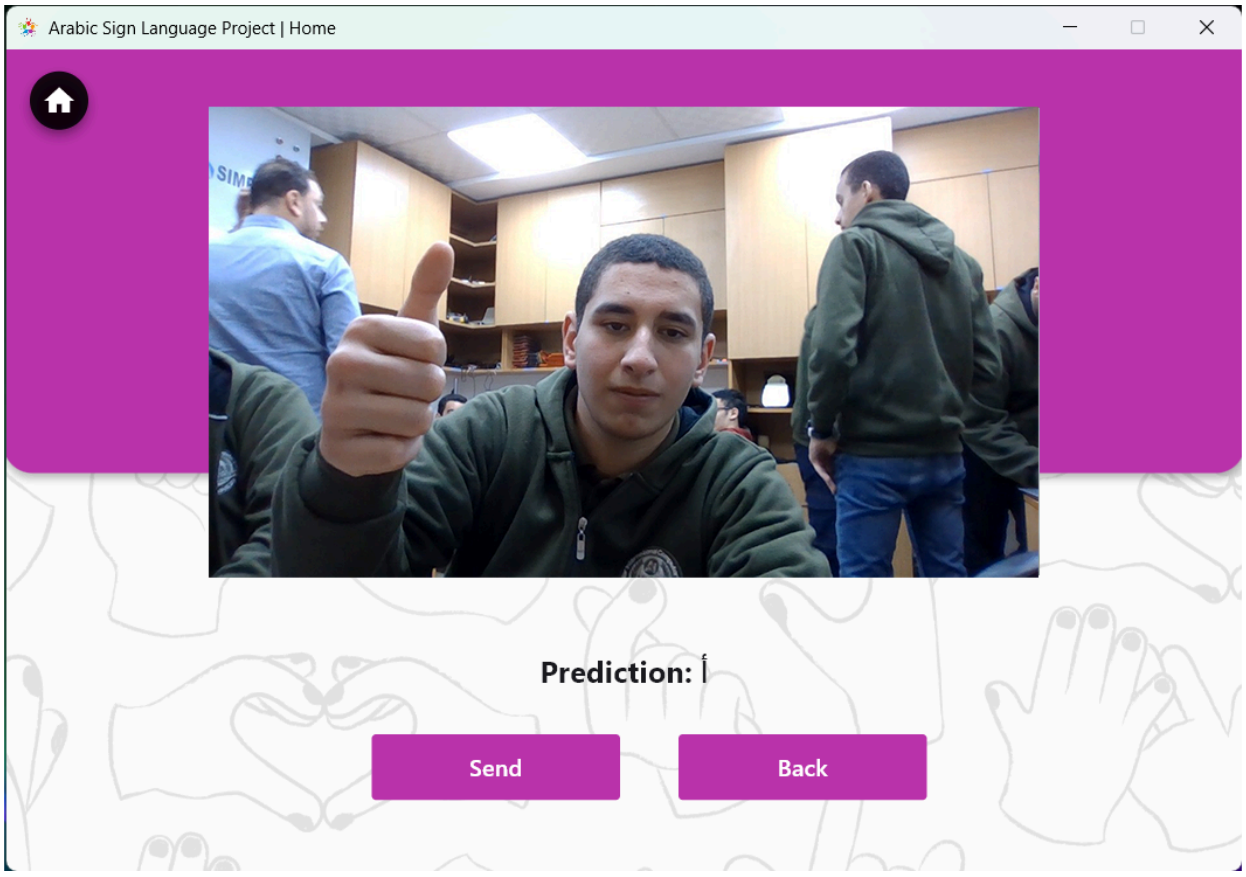
```
Container(
  height: 45,
  width: 170,
  decoration: const BoxDecoration(
    color: Color(0xffBA33AB),
    borderRadius: BorderRadius.all(Radius.circular(3)),
  ),
  child: MaterialButton(
    onPressed: _selectedImage != null
        ? () {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (context) =>
              ResultScreen(media: _selectedImage!,isVideo: false,),
        ),
      );
      dispose();
    }
        : _captureImage,
    child: Text(
     _selectedImage != null ? isArabic! ?
          'متابعة' : 'Proceed' : isArabic! ? 'التقاط' : 'Capture',
      style: const TextStyle(
        color: Colors.white,
        fontSize: 16,
        fontFamily: 'Segoe UI',
      ),
    ),
  ),
),
```

- **Result Screen**:

After the **user** either **picks** a media file from their files or takes a new one they can click "**proceed**" from the **Camera screen** where they will be taken to the **Result Screen** where they can send the image to the **API** which will return the **AI Model prediction** of their **signs**.

In this **screen** their **media of choice** (Image/Video) is displayed in the middle with **2 buttons** under it, "**Send**" and "**Back**".

When the **send button** is pressed it will then **send the Media file** to the **API** which will then **return the prediction** of the **Media file** and save it to the user's **History**.



Important sections of the code:

After an **image** is taken it is sent to the **API** in **base64** format and then **decoded** back to **normal** when it has reached the **API**.

Then it returns back the **prediction** of the **AI model** and saves the **image** to the user's **history** so it can be **accessed later**.

```
Future<void> _sendImage() async {
  setState(() {
    _isSending = true;
  });

  try {
    final bytes = await widget.media!.readAsBytes();
    final base64Image = base64Encode(bytes);

    final response = await http.post(
      Uri.parse(LinkModel), // Ensure correct endpoint
      headers: {'Content-Type': 'application/json'},
      body: jsonEncode({
        'image': base64Image,
        'type': "Sign Language To Words",
        'id': globalID, // Ensure userId is provided
      }),
    );

    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      setState(() {
        prediction = data['prediction'] ?? 'No prediction available';
      });
    } else {
      final errorData = jsonDecode(response.body);
      _showError("Error: ${errorData['error'] ?? 'Unknown error'}");
    }
  } catch (e) {
    _showError("Error sending image: $e");
  } finally {
    setState(() {
      _isSending = false;
    });
  }
}
```

Any **Image** is **saved** in the **Pictures folder** so the **user** can **access** it later on **through** their **file manager**.

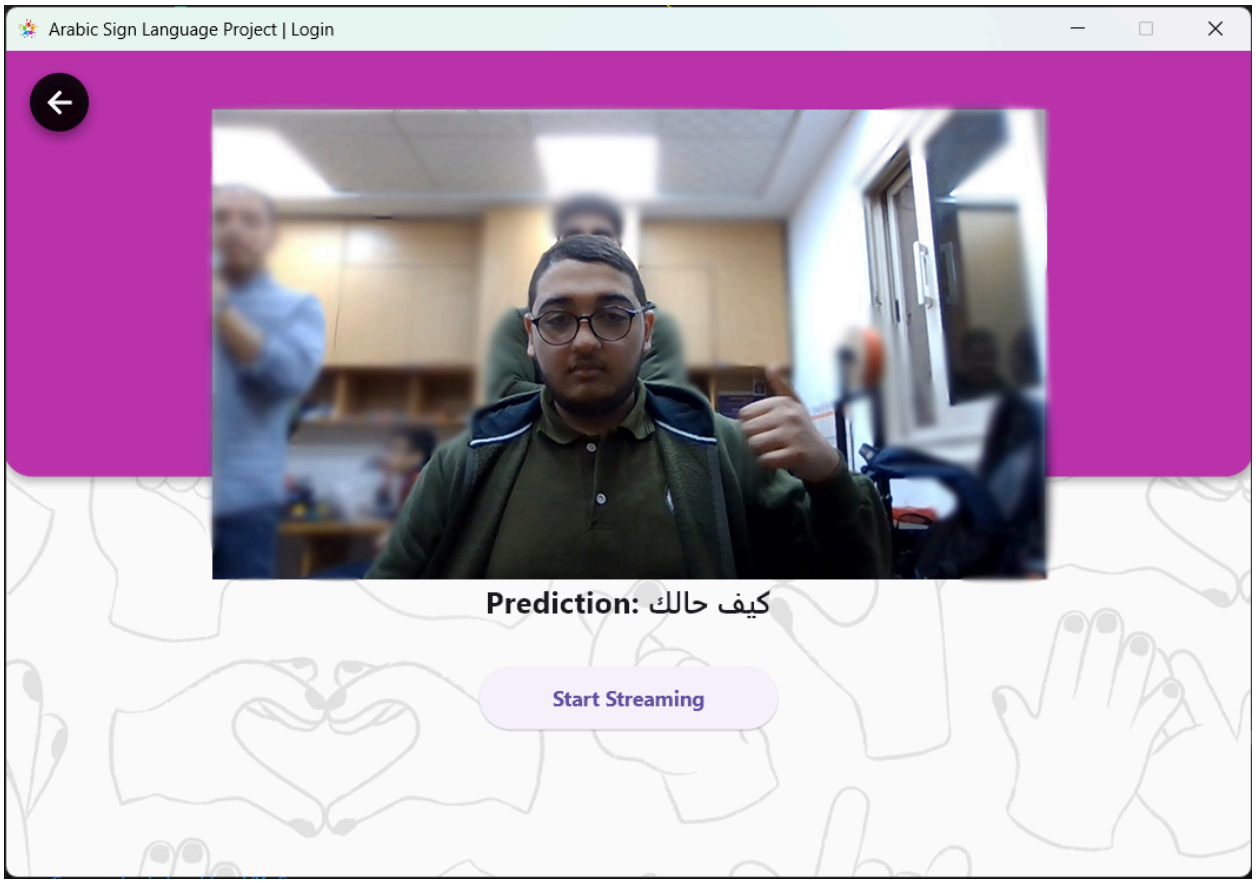
Any **images** saved to the **user history** can be **deleted** at **any time** and **are not stored forever** to **free up storage** in the **database**.

```
Future<void> uploadVideo() async {
   if (widget.media == null && widget.media == null) {

     return;
   }
   setState(() {
     _isSending = true;
   });
   try {
     var url = Uri.parse(LinkSendVideoModel); // API endpoint for video upload

     var request = http.MultipartRequest('POST', url)
       ..files.add(await http.MultipartFile.fromPath(
           'video',
           widget.media!.path ??
               widget.media!.path)); // Add video file to request

     var response = await request.send(); // Send request

     if (response.statusCode == 200) {
       // Check if request was successful
       var responseBody = await response.stream.bytesToString();
       var jsonData = jsonDecode(responseBody); // Decode response
       print("Prediction: ${jsonData['prediction']}");

       setState(() {
         prediction = jsonData['prediction'] ?? 'No prediction';
         _isSending = false;
       });
     } else {
       // Handle request failure
       print("Error: ${response.reasonPhrase}");
       setState(() {
         _isSending= false;
       });
     }
   } catch (e) {
     // Handle exceptions
     print(e);
     setState(() {
       _isSending = false;
     });
   }
 }
```

- **Live Screen**:

After the user picks the **Live** option from the **alert box** they will be **taken** to the **Live Screen**.

In this screen the **camera** is **initialized** in the **center** and under it is a "**start streaming**" **button** that will **start streaming** each **frame** from the **camera** at **30fps** to a **web socket** that will **process each frame** from the **motion/action model** and **send back** the **prediction**.



Important sections of the code:

After initializing the camera the application checks for a connection to the web socket IP address using "Socket IO client" Package.

If there is a **connection**, the **application** starts **streaming** the **frames** from the **camera** right when the **user presses** the "**Start Streaming**" button.

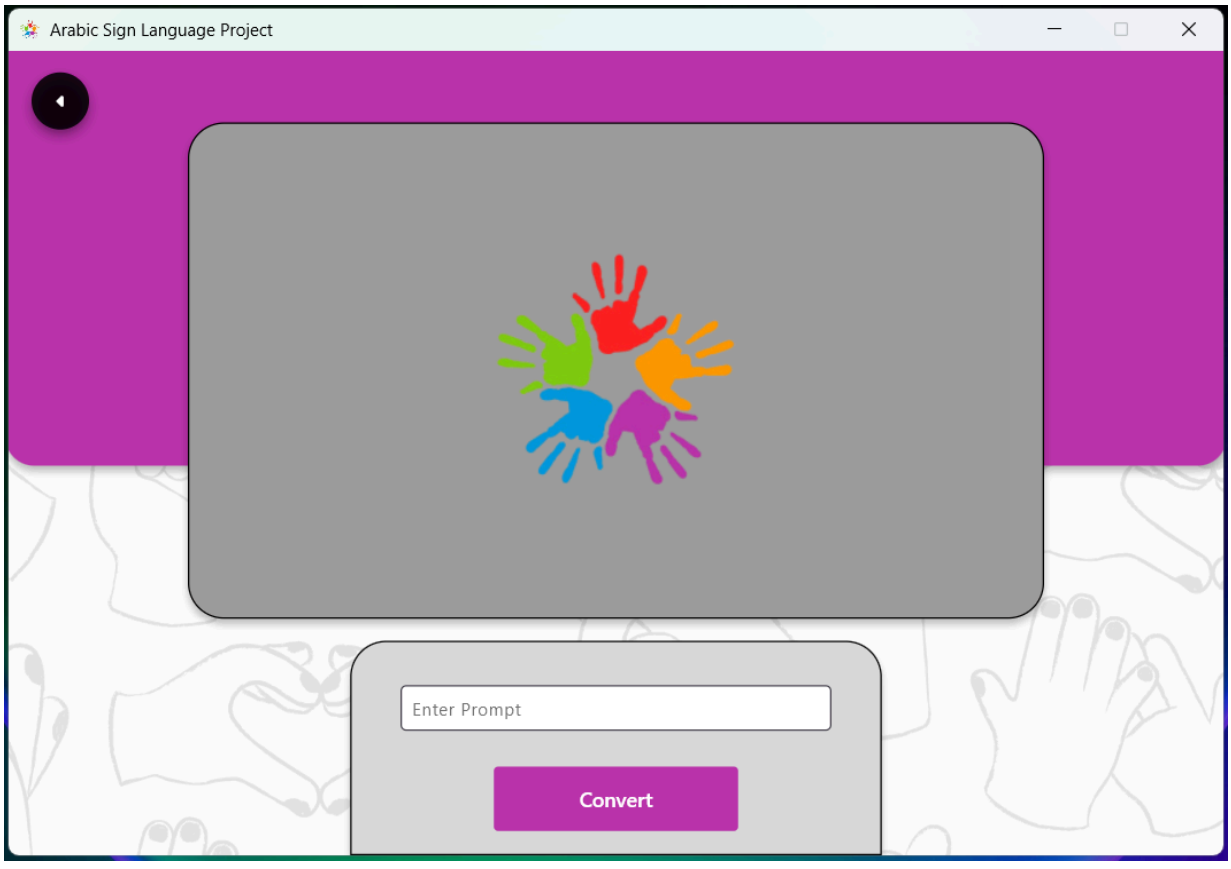
```
void _initSocket() {
   // Connect to the Socket.IO server
   socket = IO.io(linkServerName, <String, dynamic>{
     'transports': ['websocket'],
     'autoConnect': true,
   });

   // Listen for connection events
   socket.onConnect((_) {
     print('Connected to Socket.IO server');
   });

   // Listen for the 'test' event from the server
   socket.on('prediction', (data) {
     print("Received data from server");
     setState(() {
       _processed = data;
       _prediction = _processed.toString();
     });
   });

   // Listen for disconnection events
   socket.onDisconnect((_) {
     print('Disconnected from Socket.IO server');
   });

   // Handle errors
   socket.onError((error) {
     print("Socket.IO error: $error");
   });
 }
```

```
void _streamFrames() async {
   while (isStreaming) {
     if (_cameraController == null || !_cameraController!.value.isInitialized) {
       print("Camera stopped streaming due to unavailability");
       return;
     }

     final XFile frame = await _cameraController!.takePicture();
     Uint8List imgBytes = await frame.readAsBytes();
     String base64String = base64Encode(imgBytes);

     // Send the base64-encoded frame to the server using the 'video_stream' event
     socket.emit('video_stream', base64String);

     await Future.delayed(Duration(milliseconds: 100)); // Adjust FPS
   }
 }
```

- **Words to Sign Language Screen**:

After **pressing** the **Words to Sign Language button** in the home screen the user is taken to the **Words to Sign Language screen** where they have a rounded container in the middle with the **ASLP logo** by **default**.

A cut-off rounded square is found at the bottom containing a **text field** to enter the **words or letters** that the **user** would like to **convert** into **sign language** and a "**Convert**" **button** under it.

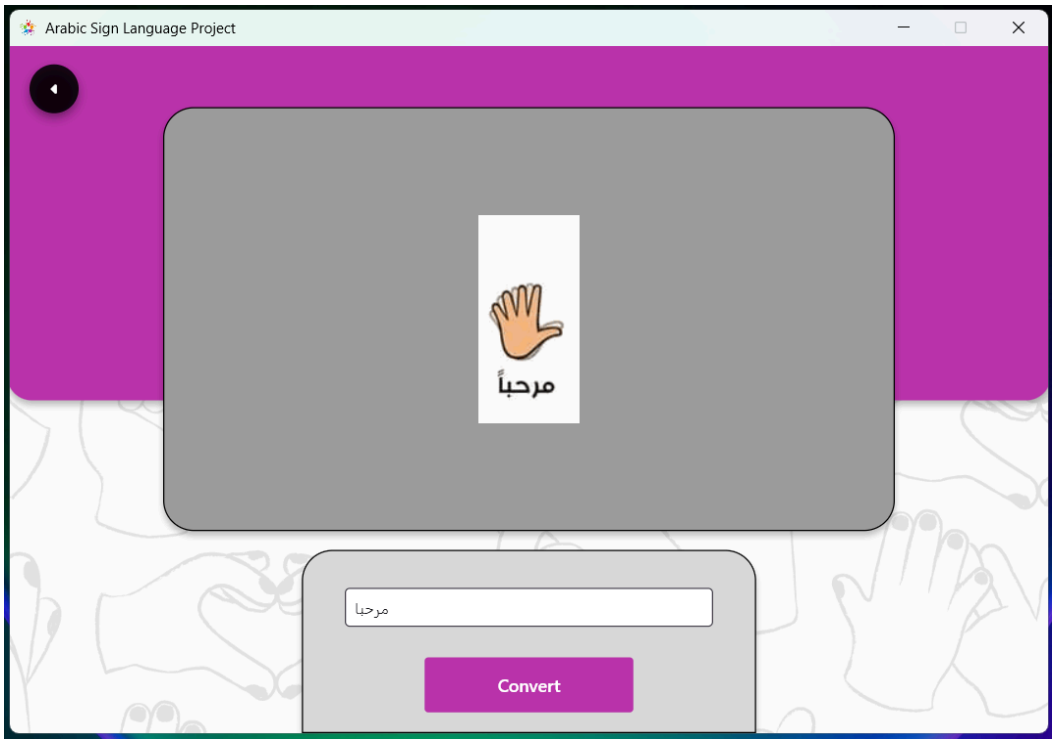


After the user has entered a **string** of **words** or **letters** they **wish** to **convert**.

That **String** to will be **sent** to the **API** and then will be **returned** as an **image if** it's a **single word or letter**.

If that **String** is **more** than **one word or letter** a **GIF** will be **sent**.

It works by **separating each letter or word** with a **space**.



Important sections of the code:

To **convert** the **Words to Sign Language**, First make a **string called GIF** so when the **API returns** back the **image or GIF** we can **save its image link** and **display it** to the **user**.

```
String Gif = "";

  Future<void> Convert(String sentence) async {
    var response = await crud.postRequest(
      linkTextToSign,
      {'id': globalID.toString(), "sentence": sentence},
    );

    if (response == null) {
      return;
    }

    if (response['message'] == 'Not found') {
      print("Error");
    } else if (response['message'] == 'Correct') {
      setState(() {
        Gif = response['file_name'];
      });
    }
  }
```

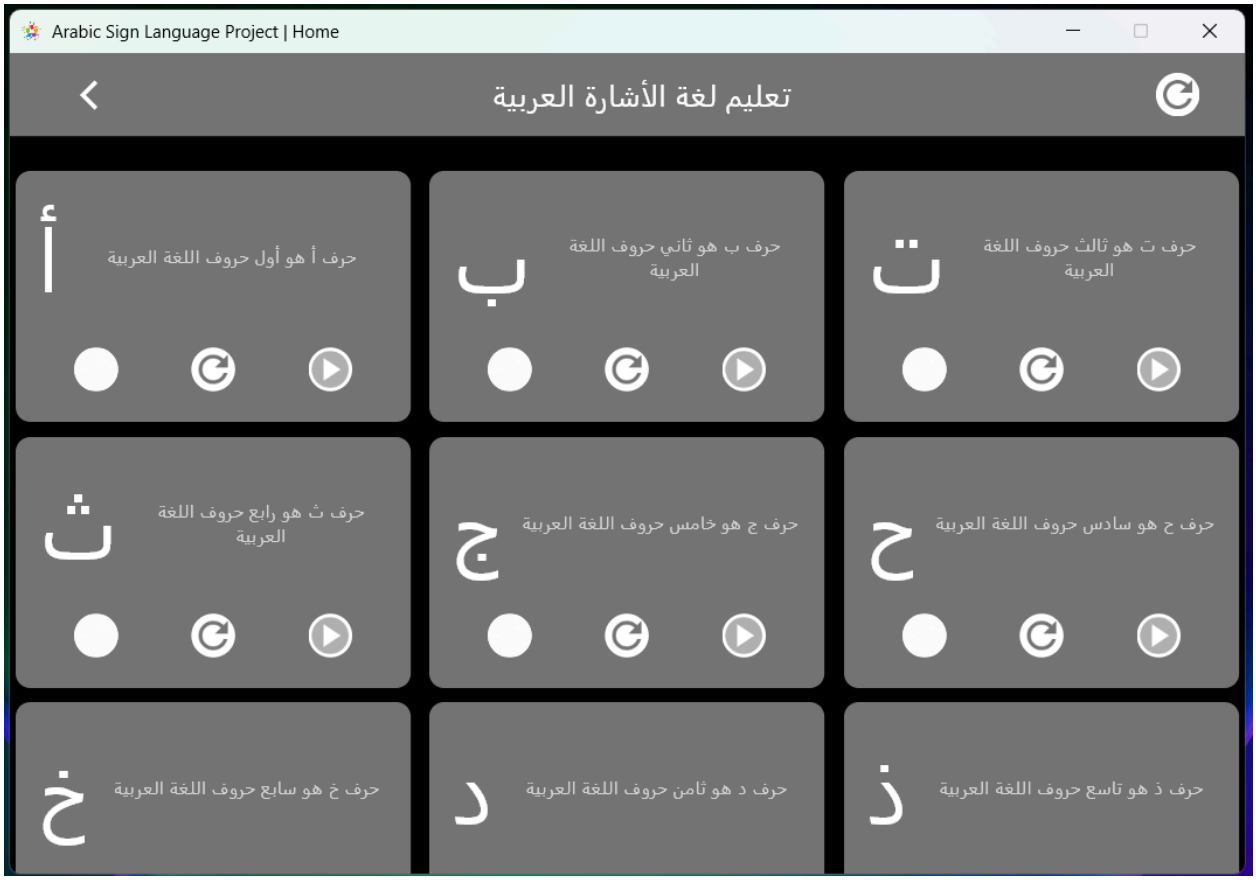
- **Learn Arabic Sign Language Screen**:

After pressing the **Learn Arabic Sign Language** button the user will then be taken to this screen where he can pick any letter they wish to learn by pressing the play button.

All **Arabic letters** are listed. Once the user **fully learns** a letter the **UI** will **automatically update** to **show** whether they have **successfully learned** it or **failed** at **learning it**.

If the user has **failed** they can **press** the **looping arrow** icon to retry the letter.

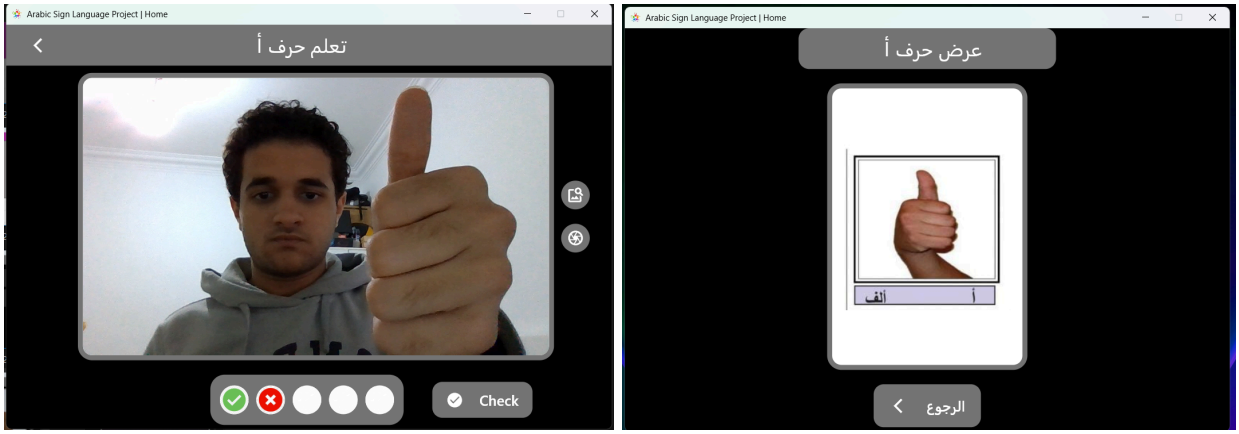
If they wish to **reset everything** they can **press** the **looping arrow** at the top to **reset everything**.



Once the **user picks** which **letter** they want to **learn** they will be taken to the **learning screen** where the **webcam** will **open** and show **the preview** in the middle with **2 buttons** on the **side**.

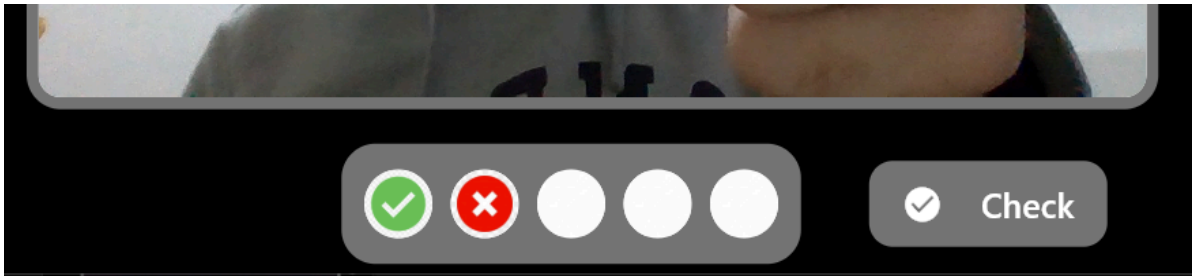
The **First button** shows **how** the **letter hand sign** is done.

The **Second button captures** an **image** from the **webcam** and **displays** it as the **preview**.



After a **picture** is taken the user can press the "**check**" **button** to send the picture to the **API** to **check the hand sign**.

If the **hand sign** is **correct** it will **update** the **bottom bar** with **either a right or wrong icon** depending on if the **hand sign was done correctly or not**.



Important sections of the code:

So, when an **image** is **sent** to the **API** it **checks** how **many times** the **user** has **sent** an **image** and how **many times** the **user** has **failed or** got it **correct**.

```
// Function to update the learning status in the database
 checkLearn(String Letter, String Done) async {
   var response = await api.postRequest(linkLearn,
       {'letter': Letter, 'id': sharedPref.getString('id'), "done": Done});
 }

 // Function to check the user's attempt
 check() async {
   setState(() {
     widget.isLoading = true; // Set loading state to true
   });
   if (_cameraImage == null) {
     // Check if an image is selected
     setState(() {
       widget.isLoading = false; // Set loading state to false
     });
     _showSnackBar('أدخل صورة أولا'); // Show error message
     return;
   }

   try {
     Uint8List bytes =
         await _cameraImage!.readAsBytes(); // Read image as bytes
     setState(() {
       widget.isLoading = true; // Set loading state to true
     });

     // Send the image to the API for prediction
     var response = await api.postRequest(LinkModel, {
       'image': base64Encode(bytes),
       'type': 'learn',
       'id': sharedPref.getString('id')
     });
     setState(() {
       widget.isLoading = false; // Set loading state to false
     });

     if (response == null) {
       // Handle null response
       setState(() {
         _cameraImage = null; // Clear the selected image
         count += 1; // Increment the attempt count
         // Update the check status based on the attempt count
         if (count == 1) {
           check1 = "False";
         } else if (count == 2) {
           check2 = "False";
         } else if (count == 3) {
           check3 = "False";
         } else if (count == 4) {
           check4 = "False";
         } else if (count == 5) {
           check5 = "False";
         }
       });
```

```
        setState(() {
          if (count == 5) {
            // If all attempts are used
            button = false; // Disable the button
            int trueCount = [check1, check2, check3, check4, check5]
                .where((c) => c == 'True')
                .length;
            // Check if the learning is successful based on the check status

            if (trueCount >= 3) {
              checkLearn(widget.Letter, "True"); // Mark learning as successful
            } else {
              checkLearn(widget.Letter, "False"); // Mark learning as failed
            }
          }
        });
        return;
      }
```

```
else if (response['prediction'] != null) {
        // Handle successful response
        if (response['prediction'] == widget.Letter) {
          // Check if the prediction matches the letter
          _showSnackBar(
              '${response['prediction']} : الناتج  '); // Show success message
          setState(() {
            _cameraImage = null; // Clear the selected image
            count += 1; // Increment the attempt count
            // Update the check status based on the attempt count
            if (count == 1) {
              check1 = "True";
            } else if (count == 2) {
              check2 = "True";
            } else if (count == 3) {
              check3 = "True";
            } else if (count == 4) {
              check4 = "True";
            } else if (count == 5) {
              check5 = "True";
            }
          });
```

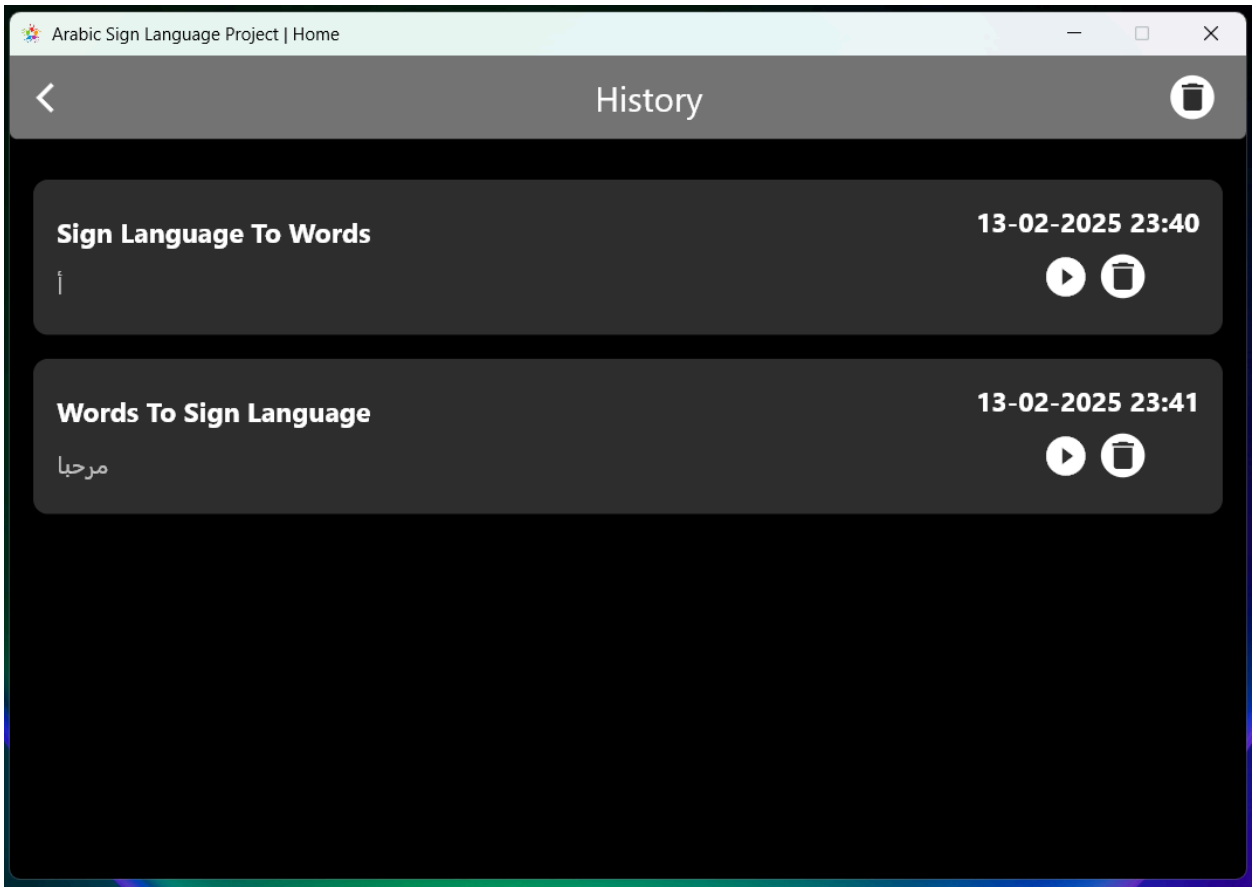
```
      } else {
        // Handle incorrect prediction
        setState(() {
          _cameraImage = null; // Clear the selected image
          count += 1; // Increment the attempt count
          // Update the check status based on the attempt count
          if (count == 1) {
            check1 = "False";
          } else if (count == 2) {
            check2 = "False";
          } else if (count == 3) {
            check3 = "False";
          } else if (count == 4) {
            check4 = "False";
          } else if (count == 5) {
            check5 = "False";
          }
        });
      }
      setState(() {
        if (count == 5) {
          // If all attempts are used
          button = false; // Disable the button
          int trueCount = [check1, check2, check3, check4, check5]
              .where((c) => c == 'True')
              .length;
          // Check if the learning is successful based on the check status

          if (trueCount >= 3) {
            checkLearn(widget.Letter, "True"); // Mark learning as successful
          } else {
            checkLearn(widget.Letter, "False"); // Mark learning as failed
          }
        }
      });
    } else {
      // Handle unexpected response
      _showSnackBar('Unexpected response');
    }
  } catch (e) {
    // Handle exceptions
    setState(() {
      widget.isLoading = false; // Set loading state to false
    });
    _showSnackBar('An error occurred: $e'); // Show error message
  }
}
```

- **History Screen**:

This screen is accessed by pressing the **history button** in the **Home Screen drawer**.

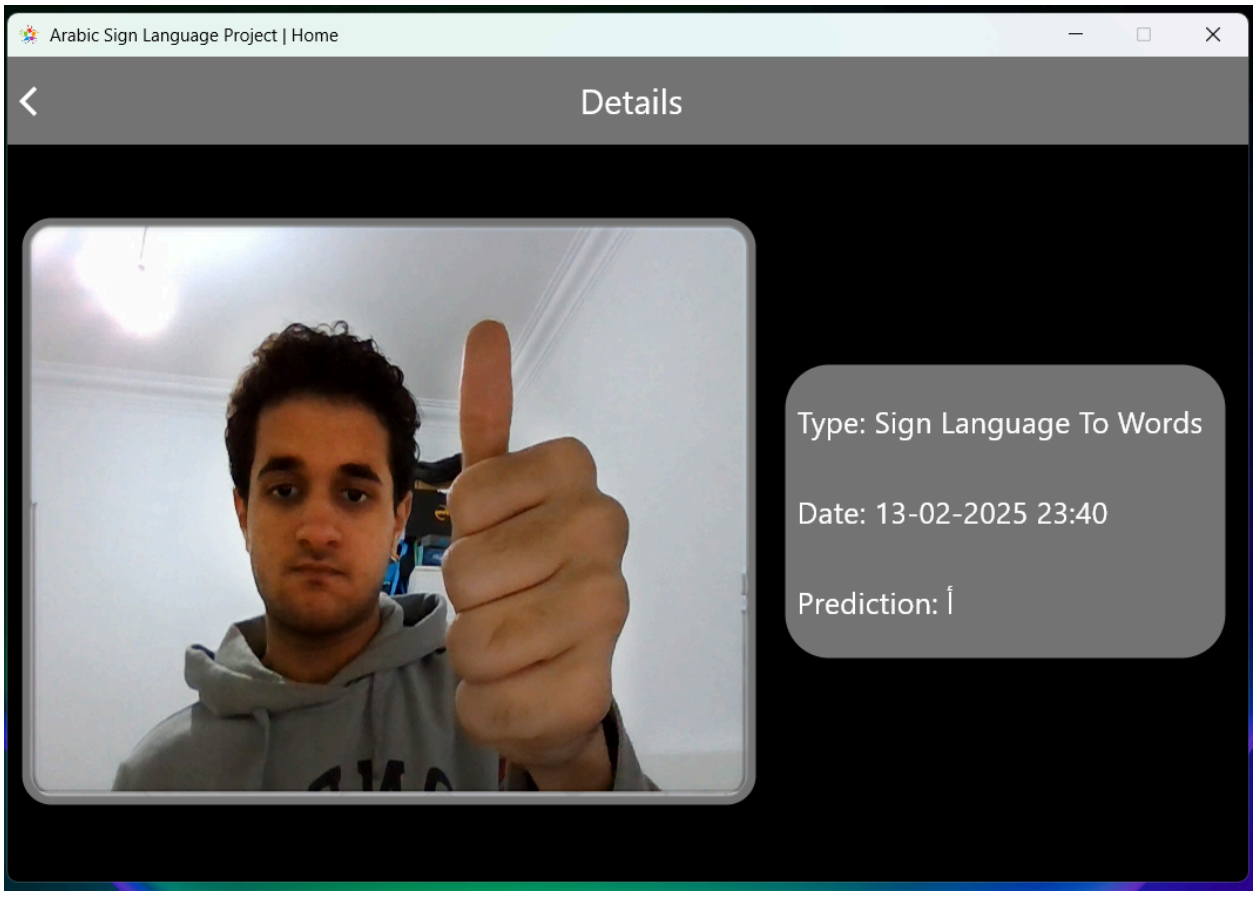
Upon entering the screen all the **user's history** will be **fetched** and **placed** in a tile inside of a list with the **type** and **date** and **time** and the **AI Model prediction** at the **bottom left**.

It shows all the past **usages** of the **Sign Language to Words** or **Words to Sign Language**.



The **user** can **manually delete all** the **history** by pressing the **trash icon in the app bar** on the **top right corner** or **delete a single instance/usage** by pressing on the **trash icon** on the **desired tile**.

In addition there is also a **play icon** then **when pressed** will **show** the **details** of the **usage/instance** including the **Image**.



Chapter 5 | Final Results & Future work

5-1 Final Results

Static Model

Motion Model

Videos

Static Model Video

Motion Model Video

5-1-1 Overview of Results

Our project, an **Arabic Sign Language Translator**, aimed to facilitate real-time communication between the deaf and hearing communities by translating Arabic Sign Language into text or from text to Signs. The system successfully captures sign language gestures through a camera, processes them using an AI model, and provides accurate translations. Through rigorous testing, we validated the accuracy, speed, and usability of our solution

5-1-2 Real-World Impact

Our Arabic Sign Language Translator has significant real-world applications:

- **Education**: Helps deaf students communicate more effectively in schools and universities.
- **Customer Service**: Enhances accessibility in banks, hospitals, and government offices.
- **Social Inclusion**: Bridges the communication gap between the deaf and hearing communities, promoting inclusion.
- **Emergency Situations**: Enables faster and more effective communication for deaf individuals in critical scenarios.

By providing an intuitive and accessible translation system, our project has the potential to improve the lives of thousands of people in the Arabic-speaking deaf community by using our Desktop & Mobile applications.

5-1-3 Limitations & Challenges

Despite its success, our system has some limitations:

- **Complex Gestures**: Some multi-hand gestures and dynamic gestures involving motion over time were harder to recognize accurately. The system struggles with distinguishing subtle variations in hand positioning and movement speed.
- **Background Noise**: Variations in lighting, background clutter, and differences in hand shape due to skin tone or camera angles slightly impacted recognition. Ensuring optimal conditions for gesture recognition remains a challenge.
- **Limited Dataset**: The dataset used for training the model was not exhaustive and lacked certain regional variations and dialects in Arabic Sign Language. Collecting a more diverse and comprehensive dataset could enhance the system's accuracy.
- **Contextual Understanding**: The system translates individual signs without considering sentence structure or context, leading to potential misinterpretations. Implementing natural language processing (NLP) techniques could help improve contextual accuracy.
- **Hardware Constraints**: The system requires a high-quality camera Lower-end devices may experience reduced recognition performance, limiting accessibility for some users.
- **User Adaptability**: Some users found it difficult to adjust their signing styles to match the model's training data, leading to occasional misclassification of gestures.

5-2 Conclusion

In conclusion, the development of our **Arabic Sign Language Translator** demonstrates the **potential of artificial intelligence** in enhancing **accessibility** for the **deaf community**. The system has shown **promising results** in accurately **translating sign language into text and speech**, providing a **crucial communication tool**. Despite challenges such as **complex gestures** and **dataset limitations**, the project lays a **solid foundation for future improvements**. By **incorporating additional features** and **refining the model**, this technology could greatly impact **education**, **customer service**, and **social inclusion**. Ultimately, our work contributes to a **more inclusive society** where **language barriers** no longer hinder **communication**.

5-3 Future work

To enhance our system, future work could focus on:

- **Expanding the Training Dataset**: Collecting and labeling more Arabic sign gestures to improve accuracy.
- **Integrating NLP**: Using natural language processing to improve contextual understanding.
- **Web Implementation**: Developing a web-based solution for broader accessibility.
- **Improving the Live option:** Current applications rely on cloud processing, requiring users to upload videos or images for translation, and a working live option that needs to be improved and worked on, reducing delays and enhancing user experience.

Our project represents a meaningful step toward bridging the communication gap for the Arabic-speaking deaf community. With continued development, this translator holds the potential to become an essential tool for improving accessibility and fostering inclusion across a wide range of sectors.