How to Use Each Completed Feature:

Intro

This document is pretty long. It describes the design of each feature in the app and gives some description of the code structure for each. While reading this document completely will certainly help you develop the project, it will also take a long time. So using the table of contents to reference specific features is recommended.

Project File Structure

The Project is split into two main portions:

1. Flutter App: The flutter app contains all of the logic and UI for our frontend. It handles data views for researchers, provides login and authentication, and provides all of the word tracking and recording features of the app.
2. Firebase Project: This section of the app currently only contains cloud functions which are used to run secure actions. However, the firebase console also contains rules for the database that control access to data.

The project is split into the following folders:

+---baby_words_tracker (Flutter App)
+---docs (Documentation Website)
\---firebase-project (Firebase Project)

Flutter App

The flutter app is found in the baby_words_tracker folder in the root of the repository and is structured as follows:

```
baby_words_tracker/        # Root project directory
├── android/               # Android app platform code
├── assets/                # App assets like images or data files
│   └── LECS_mascot.png    # The mascot of the LECS lab
│   # Note: Add assets to pubspec.yaml to use them in the app
├── doc/                   # Flutter-generated documentation
│   └── api/
│     └── index.html  # Main page of the generated docs website
├── ios/                   # iOS app platform code
├── lib/                   # Main application code and logic
│   ├── auth/              # Authentication and user model services
│   ├── data/              # Database interaction logic
│   │   ├── listeners/     # Listeners for document change events
│   │   ├── models/        # Data models that mirror Firestore docs
│   │   ├── repositories/  # Abstractions to read/write data
│   │   └── services/      # Logic that uses models and repositories
│   ├── l10n/              # Localization (translations)
│   ├── pages/             # App screens and navigation
│   │   └── shared/        # Shared components (top bar, bottom nav)
│   ├── util/              # Utilities and enums (language, role…)
│   ├── video/             # Video-related functions (single file)
│   └── main.dart          # App entry point: provider setup, routing
├── linux/                 # Linux app platform code
├── macos/                 # macOS app platform code
├── test/                  # Widget tests from Flutter template
├── web/                   # Web app platform code
```

```
└── windows/                # Windows app platform code
```

Firebase Project

The Firebase Project is used to develop cloud functions. It can also be used to develop database rules and other Firebase things. The structure is as follows:

```
firebase-project/                  # Firebase project root folder
├── emulator-sources/      # Local Firestore emulator data
│   ├── auth_export/
│   └── firestore_export/
│     └── all_namespaces/
│           └── all_kinds/
├── functions/             # Cloud Functions
│   ├── auth/              # Functions to manage custom claims  |   |
                             and check roles. Also a role enum.
│   ├── node_modules/      # Dependencies (auto-generated)
│   ├── util/              # Utility functions
│   │                             (e.g. logging a generic data object)
│   └── index.js           # Main file for all cloud functions
└── commands.txt           # Helpful Firebase emulation CLI
                                  commands
```

Project Features

Add Words - text and video:

Path: <repo_folder>/baby_words_tracker/lib/pages/landing_page.dart

The add words feature is at the bottom of the landing page and includes both text and video input. The user can input a string of words, phrases, or sentences into the text box, and it will be parsed into individual unique words that are in either the english or spanish dictionary. The words are sent to spell check using the check_and_update_words() function to ensure that the given words are valid English or Spanish words. Any new words (project wide not just for the specific child) will be added to the word bank. For more information on this function, please see the spell check section. The user can also select a video from their device. The video selector will only allow .mp4 videos to be selected. To do this we use the file_picker library. This library handles the direct calls to the user's device's file system and will only show files that match the specified file type. For more information about the library see the following documentation. https://pub.dev/packages/file_picker

After the words have been verified the app will use the cloud functions to fetch a signed url. A signed url is a https API call to our Google Cloud created by firebase functions. For more information, please see the cloud function or video section. That url will then be used to upload

the selected video, if there is one, to the user's (parent) folder in the google cloud storage bucket. The folder name for each user is their user id. Then, for each unique word, a word tracker will be created or updated in the selected child's word tracker collection. You can identify the current child by making a call to the current child service. The current child's name is also displayed in the top right hand corner of the Top Bar of the user interface.

View Stats

Path to page: <repo_folder>/baby_words_tracker/lib/pages/stats.dart

Page Info:

The View Stats (or Learning Summary) page of the app allows a parent to look at graphs showing statistics about their child's learning. The Syncfusion package is used for graph display. The user is able to select a graph to view using a drop down at the middle of the screen, and is able to select the time horizon of the graph using a numerical input box below the graph.

State Info:

The page is stateful to allow the above configurations to update the pages state, which determines how the graph is built. Functions to update this state information are also declared within a state, to allow features built in the built function to change their values. The type of graph to be displayed is stored as a GraphType Enum, which should be expanded whenever new graph types are added. There is also a global final list of GraphTypes called graphsWithLength, that indicates which graphs should have the ability to have their length adjusted. This should also be updated whenever more graphs are added. A TextEditingController is declared once to allow user submission of the graphLength.

Also stored in this page's state is the graphCache, a map from 3-tuples of (GraphType, int, String) to dynamic.

graphCache element:

- (GraphType, int, String)
  - GraphType = The graphtype of the stored information
  - int = The length (in days) of the stored information
  - String = the childID of the child the information is about
- dynamic = Data used to generate a graph of GraphType, stored as a List

This map is used to prevent re-querying the same data during a single session on the stats page, in order to save database hits. This functionality could absolutely improved to persist over an entire session of the app, over a certain amount of time, or to update only whenever the relevant info changes.

Key Functions in Build:

- wordsKnownFeature = Simple wrapper function for a database call to get the number of words a child knows
- graphSwitcher = Wrapper for a switch statement that selects the proper graph to be displayed. Needs to be expanded whenever a new graph is added
- lengthChangeFeature = wrapper for the text box and submit button that use the function in State to change the graph length

- graphTypeSelectDropdown = What it says on the box

Graph Builder Functions:

Every graph is a FutureBuilder – A function that is itself synchronous but depends on the result of an asynchronous function. which depends on the results of a Graph Data Function that makes queries and organizes them into a list to be turned into a graph. Once the future is done being computed, its result is added to the snapshot, which the graph function then grabs the data from, and builds a graph. Before the data is added to the snapshot, it can be in various loading or error states, where it can be set to return another widget to indicate its state e.g. loading wheel. Explanations on how to create graphs (synfusion charts) can be found in the syncfusion documentation here.

Graph Data Functions:

These functions make the proper database queries based on the request of the user, then sort or otherwise organize the data and return a list for their coupled Graph Builder to turn into a graph.

Notes:

Graphs for (and therefore queries on) the entire history of a child's learning can grow large and expensive quickly. It would be wise to add better caching and/or more database end statistics so that they are not done via many database reads.

Authentication:

Authentication in our app is handled by firebase. The authentication UI pages are provided by the package firebase_ui_auth. These pages provide the user with a sign in screen and a profile page. Users are given a randomly generated userID (UID) on account creation and stored in the firebase authentication database. Additionally, users are given the parent role by default and can be given more roles later.

Sign in Methods:

Sign in methods can be enabled and disabled in the Firebase console. Password and email based sign in is enabled by default. But other providers like google and icloud can and should be enabled.The setting for this can be found in the firebase console in the authentication page under the sign-in methods tab.

Handling Authentication Status in the App:

Authentication status and user roles are checked and stored by the AuthenticationService class (found in ‘baby_words_tracker/lib/auth/authentication_service.dart’). The class notifies users whenever the user uid, email, displayname, or custom claims changeA provider for this class is created in main and provided throughout the project. This class should be used to access user email, uid, displayname, and custom claims. Important note: the

UserModelService is a listener to the AuthenticationService; however, it takes a moment to load new data, so in cases where user data is not required, the authentication service should be used to access information or listened to for changes. Otherwise the UserModelService should be used. A consumer widget should be used to access the information the AuthenticationService stores.

User database data is stored and synchronized using the UserModelService class in the same folder as the AuthenticationService. This class is also initialized and provided using a provider in main. This class handles syncing authentication data with the database and pulling the user model from the database when the current user changes. Additionally, it listens to the current user model document in the database and notifies listeners when a change occurs. This class should be used to retrieve user specific information such as childIDs for a parent. The class should be accessed using a consumer widget and provider.

User Roles and Administration:

The admin page is available only to users with the admin custom claim (role). If a user attempts to run the functions on the admin page without the admin role they will fail a permissions check.

User roles can be managed in the admin page. The admin page is accessible from the researcher home screen by clicking on the shield icon in the top right corner.
The top of the admin page features a search bar. This bar allows the user to search for a specific email address to manage roles for or to get custom claims for. Below the search bar is a text field showing the currently selected email address.
The admin page provides 3 main features:

1. Fetching Custom Claims: The top button “Get Custom Claims,” gets the custom claims for the email in the search bar. If the email is not valid an error message will be returned instead. Custom claims will appear in a text element below the search bar labeled User Roles.
2. Assigning/Removing Roles: Below the “Get Custom Claims” button are a series of buttons dedicated to managing user roles. The buttons each do what they say. Adding or removing a custom claim from the current email address. After a button is pressed, a cloud function will be called and the role will be removed or added to custom claims for the selected user. After the function completes, a message will be displayed in a popup bar at the bottom of the screen indicating success or failure and the reason for failure.
3. Fetching the UID-Email Table: The bottom button fetches the UID-Email table. This is a table containing the email address, user ID, custom claims, and account status of every user. The table is then displayed in a scrollable list, and downloaded as a CSV. This table is quite large, so it may take some time for the function to complete. The table allows a user to deanonymize the database if necessary for research or administration purposes; however, it should be considered sensitive since the database is supposed to be anonymous for all users except for administrators.

Add Child to Parent:

Path to utils: <parent_directory>\baby_words_tracker\lib\util\child_utils.dart
Path to GUI location: <parent_directory>\baby_words_tracker\lib\pages\settings.dart

Adding a new child is done with this utility function. The GUI has the user input a name, birth date, and set of languages that they speak, then that information is passed into the util function addChildToCurrParent. If the current parent exists, the util function uses the childDataService to create a new child and add it to the database with the current parent set as its parent. It then uses to ParentDataService to add the newly created child to the list of children of the current parent. The child and parent mutually having each others ids in their parents and children lists respectively is all that it takes for the relationship to be known by the app.

Give Another Parent Access to Current Child:

Path to utils: <parent_directory>\baby_words_tracker\lib\util\child_utils.dart
Path to GUI location: <parent_directory>\baby_words_tracker\lib\pages\settings.dart

Similar to adding a child to the current parent, this feature allows the user to add their currently selected child to another user of the app. This was chosen to be done simply by passing in a valid email, because people who are mutually taking care of a child can reasonably be expected to know each others emails. Because the current user typically will not have permission to directly change database information about another user, and the other parents ID must be found based on their email, this functionality is implemented through a firebase cloud function "addChildToOtherParent", preventing exposure of these calls to a malicious end user. Reference Cloud Functions for more information on how to modify this.

Video Functions and Playback:

Path to utils: <parent_directory>/baby_words_tracker/video/video_functions.dart
Path to display: <parent_directory>/baby_words_tracker/pages/video_display.dart

Video function implements all of the logic needed for video upload on the landing page. There are three steps to the video upload process: file selection, get signed url, upload file. They must be implemented in that order because the file path/name is needed for the signed url. File selection is handled by the file_picker library and enables users on android, iOS, macOS, and windows to pick any .mp4 file on their system. The base file path (i.e. just the filename without any of the system path information) is passed to the get signed url cloud function. These functions are implemented in the firebase-project directory and more information on them can be found in the Cloud Functions section. The response to this call is the signed url that should be used for the upload. This will only allow uploads to the specified file path for 5 minutes, so the signed url needs to be generated for each video upload. To upload files use the signed url like an api call and add the video file written in bytes to the body of the request. The response from the request will be 200 if the request is successful. Note we are not

currently compressing the videos because the flutter library that has been the standard is no longer available and no suitable alternative has been found as of Apr 23, 2025.

Video download/playback has a very similar process. When a video is uploaded, the word tracker for the associated word is also updated to include the base file path. A list of words with videoIDs that are not null is pulled down using the WordTracker service. A dropdown menu is displayed to the user to allow them to select which word they would like to see a video for. Once a word is selected, a signed url for the file path in google cloud is generated and used to request the file. The file is returned in bytes and it is written to a temporary directory. This directory will be wiped from the user's device when the app is closed, so there are no concerns about exponentially increasing the storage needed on a user's device. Once the file is written, it is theoretically available for playback. However, we have not been able to get this feature to work as of Apr 23, 2025.

Researcher Page:

Path: <repo_folder>/baby_words_tracker/lib/pages/researcher_home_page.dart

The researcher page is only accessible to users with the researcher user type. After logging in, the researcher is sent to the researcher page which displays a scrollable table of all of the word utterances submitted by the users of the application. Each word utterance entry in the table contains the child ID, child age, the word that was uttered, the language and part of speech of the word, and the date of the first utterance. The researcher can filter the table by selecting a field name from the dropdown menu and entering the value of that field to filter by in the text box. The text box provides autocomplete suggestions based on the field selected. Selecting the “filter” button will display the filtered table, and the “clear filter” button will return the table to its original state. The table can also be sorted by any field by clicking on the corresponding header, with an up arrow by the header indicating that it is sorted in ascending order and a down arrow indicating that it is sorted in descending order. The researcher can download a CSV file of the current data table by pressing the “Download as CSV” button. The CSV will contain the content of what is currently displayed in the table, so if a filter is being applied then that will be reflected in the CSV file.

Cloud Functions:

Path: <parent directory>/firebase_project/functions

Cloud functions run on a firebase hosted server and are used whenever secure code execution is needed (role limited activities, sensitive data filtering, etc…) or when the adminSDK is necessary (listing users, changing custom claims). Cloud functions are edited and uploaded by the app editors and run out of reach of end users. The logic behind this is simple. If we check

the role of a user in the frontend to deny a certain functionality, they can simply edit the app or send the http request on their own. The system must deny them on the backend so they cannot hijack our app.

In our application, cloud functions are used to do the following:

1. Assign/Remove Roles: Roles are managed as custom claims on our users' JWT (authentication) token.
    a. Role management requires that a user be authenticated and have their custom claims checked.Editing custom claims can only be done using the adminSDK which can only be securely accessed in a hosted context. Otherwise, credentials would be shipped with the app giving end users full access to admin functionality.
    b. The cloud functions for role management check that the user is authorized and meets the required role. If they meet the requirements, the desired role is added or removed.

Role	Min Role to Remove	Min Role to Add
Parent	Admin	Researcher
Researcher	Admin	Admin
Admin	Admin	Admin

2. Get Custom Claims for a User: Retrieving custom claims can only be done with the AdminSDK. To retrieve custom claims, the user must be an admin. This function returns the custom claims for a user with a specified email address. The custom claims are returned in a map which has the format {"<name>":<bool_value>}
3. Get Email-UID Table: This function allows an admin user to retrieve the full list of userIDs and email addresses along with each user's custom claims and account status. The table is formatted with the following format:

email	uid	disabled	admin	researcher	parent	unauthenticated
String	String	true/false	true/false	true/false	true/false	true/false

The table's current format reflects the current number of roles. If another role was added, it would be placed in a new column ordered based on where it is defined in the user_roles.dart enum in the utils folder. For this reason, **it is best practice to check the header of the table in any function that uses it to check user roles.** Otherwise, changes to the enum could adversely affect your code.

4. Generate Video Upload/Download Signed URLs: These functions use the authentication service to get the current users id, create the path the file in the Google Cloud bucket, and then make the current url with the given meta data. They only allow either read or write privileges to the specified file for 5 minutes for added security. Note that for upload, a blank entry to the path is made before the upload because you can not upload to a file path that does not exist.

Updating Cloud Functions

Functions are updated by changing the index.js file in the functions folder and then running a firebase tools command. Any function that is exported from index.js will be treated as a cloud function. The steps are as follows

1. Modify and save index.js
2. From the firebase_project directory run: `firebase deploy --only functions`
    a. If you want to deploy other things as well, firebase deploy deploys all applicable things in the folder

Note: Our cloud functions use ESLint for formatting. **Functions will not deploy if they do not match the ESLint rules.** Set up ESLint by doing the following:

1. In the <parent directory>/firebase_project/functions folder run the following command to install dependencies including ESLint:
    a. `npm install`
2. Install the ESLint extension for VSCode: https://marketplace.visualstudio.com/items/?itemName=dbaeumer.vscode-eslint
3. Add the following text to your VSCode settings.json file.

```
"editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
},
"eslint.validate": ["javascript"]
```

    a. Access the file by pressing `ctrl + ,` to open settings and clicking the file icon with an arrow over it at the top right to open the json file.
    b. Or by pressing `ctrl + shift + p` and typing settings then looking for the options with (JSON) at the end.

Your index.js and other javascript files in the functions folder should now be automatically formatted on save.

Spell Check:

Path: <parent directory>/baby_words_tracker/utils/check_and_update_words.dart

Spell check is a feature built to assist the add words feature, and can be found in the check_and_update_words.dart file in the utils folder. This function is only invoked by the add words section of the landing page. When a user submits text, each unique word is passed to a spell check function. The goal of this function is to check if the word is in the english or spanish language and add newly found words to the word collection in the database.

There are 2 ways to check if a word is in the dictionary. The first is to check the current word collection to see if a doc exists for the word. This is the first method that is used because it is faster than making the API call. The second is to check the Wiki Text API for an input for the given word. We are unaware of any official documentation for this API outside of the base url used in the function. The word will be checked for both English and Spanish. If the API call is successful (response code == 200), then the language is added to the found language list, and the part of speech is parsed out of the response. The definition for the word is also needed, but

cannot be found in this response. Therefore, a separate API is made to another wiki API to retrieve the definition. The first definition given is assumed to be valid. It is possible for a word to exist without returning a part of speech or definition. This is actually incredibly common and should not be seen as a sign of an issue. The definitions will initialize to null and the part of speech will be “unknown” in this case. We then use the part of speech and definition for each language to form a Word model that will be returned and it also added to the word collection. If no response is received for either language, then the word is considered to not exist and null is returned and an error is thrown.

Localization:

Path: <parent_directory>/baby_words_tracker/l10n/

Localization refers to the ability to switch between languages. The name comes from the fact that traditional localization would initialize a users language based on their location, and then use their language preferences on sign in. Our app, on the other hand, should only ever be used within the United States, so we automatically initialize to English and then have a language switch setting on the setting page for the user to flip to Spanish. All text within the app is localized using the localizationService. The service reads in a key and uses the all_localizations class to pull the actual text that will be displayed based on the current locale. The locale is updated anytime the user flips that language setting. To use the localization service the widget must be wrapped in a consumer widget that implements the LocalizationService class. Then the translation method can be called on the key for the text field and the proper text for the current locale will be displayed. It is best practice to make keys as similar to the actual text or as descriptive as possible so that code still accurately portrays the meaning of the text that will be displayed. It should be noted that the user profile and sign in pages do not use our localization system, but instead implement the firebase localization service because those pages were built by firebase. This is implemented in main.dart, and the correct firebase localization is determined by listening to the locale of our localization service.

Firestore Database:

We use firestore to store all in app information not related to authentication. Firestore is a NoSQL database that uses collections and documents (docs) to store data. Docs are similar in structure to a JSON file, and collections are simply a container for those files, like a table would be in a SQL database. A key feature that is unique to firestore is that docs can have subscollections of data. Subcollections support scaling data storage and should be used for large amounts of varying storage under a document. It is recommended to use a subcollection of a Map field if the amount of data stored will need to be scaled in any way. These subcollections will persist even if the document that hosts the subcollection has been deleted, so they must be explicitly deleted before their parent document is.

Our current database structure can be seen below:

```
Parent (Collection)
├── id (String, Document ID)
├── childIDs (String[], list of child document ids)
├── language (String)

Researcher (Collection)
├── id (String, Document ID)
├── name (String)
├── email (String)
├── institution (String)

Child (Collection)
├── id (String, Document ID)
├── name (String)
├── birthday (Timestamp)
├── parentIds (String[], list of parent document ids)
├── language (String[], language codes (es or en))
├── wordCount (Int)
├── Word Tracker (Subcollection)
    ├── id (String, Document ID is the word)
    ├── firstUtterance (Timestamp)
    ├── videoID (String, filename of video in google cloud)

Word (Collection)
├── id (String, Document ID)
├── definition (Map<String, String>: “language_code” : definition)
├── languageCodes (String[])
├── partOfSpeech (Map<String, String>: “language_code”: partOfSpeech)
```

Note that the document IDs are not actual fields in the database, but instead how to reference a specific document. However, the ids are a field of the user model. These ids are identical to the ones found in the authentication database, so the correct user can be found using the user model after sign in. There is no identifying information for the parents in the database to ensure anonymity of the users and to minimize potential bias from the research team.

To access the database directly you will need access to the firebase console. To get this, Dr. Lisa Hsin will need to add you as either a collaborator or an owner to the Google project.

Firebase Repository:

Path: <repo_folder>/baby_words_tracker/lib/data/repositories/

The FirebaseRepository class provides functions for interacting directly with the Firestore database. It is used by the data service classes to work with documents in all collections and subcollections. Documents are generally sent to the FirestoreRepository class in a Map<String, dynamic> format. Documents are retrieved from the FirestoreRepository class as a DocumentWithId class or list of DocumentWithId classes.

Currently, the FirestoreRepository class catches and prints errors then returns null, false, or an empty list on a command failure. Work needs to be done in the future to enhance error handling at the repository and project levels.

Data Models:

Path: <repo_folder>/baby_words_tracker/lib/data/models/

Data model classes mirror a document stored in the Firestore database. Each model class generally reflects a type of document stored in a single collection. These classes let us define a consistent structure for documents and retrieve and work with them in the code. Each model supports basic functions for serialization and deserialization from a JSON string, a map object, and a DataWithId object. Map is used to send data to the repository and DataWithId is used to convert retrieved data into a model object. These models are generally a one-to-one conversion with the exception of timestamps and language codes. The timestamp object in firestore can be translated to a DateTime object in dart easily, and this is done throughout. Languages codes are initialized as an enum in the data models to ensure codes remain consistent with the international abbreviation. Use the displayName function to get the language code as a string of the full language name (ex: LanguageCode.en.displayName = “English”), and use the displayCode function to pass the actual code as a string. By convention, the language code is passed to the database when a language is listed.

Data Services:

Path: <repo_folder>/baby_words_tracker/lib/data/services/

Data services add a layer of abstraction between the database layer (repository) and the front end. These functions are the only place that a repository function should ever be called. These functions are specific to the needs of the specific model, and provide more fine grained CRUD functionality. For example if you need to know the value of a specific field for a document in a collection, then you should use a getter from the corresponding data service. Note that most of the services do not implement update and delete functionality because parent users do not have database permissions to perform those operations. This is also where we do error handling at the moment. Errors from the repository are caught but thrown to the next level. Instead all of these functions, except those that return a standard data type, return a nullable data model. The model will only be null if the repository function call fails.

FAQs

Q: How do I add new words that my child has said?
A: Select the text box on the homepage below “My Child Said…”, enter a word or sentence, then press submit. If a word is not found in the dictionary, an error message will display and you will need to retype your input. To attach a video to any of these words, press the select video box and navigate to the video you would like to include. This video can not be more than 5 MB.

Q: How do I add a new child to my account?
A: Select the settings icon at the bottom of the screen, then under the “Add Child” heading, type the child’s name in the “Choose Name” text field, then select the child’s birthday from the “Tap to choose birthday…” field. Click the box next to the language(s) that your child speaks so that a check mark appears, then press the submit button. You can toggle between your children on the home screen by pressing the three dots on the upper right and selecting a child.

Q: How do I give another parent access to my child?
A: First, make sure that the other parent has made an account. Next, select the settings icon at the bottom of the screen, and scroll down to the header that reads “Give another Parent Access To Current Child.” Type in the email address of the parent you would like to add the child to, then press the submit button.

Q: How can I see statistics of my child’s learning progress?
A: Select the bar graph icon to navigate to a learning summary page. To see the “New Words Per Day” graph, select “Words Learned / Day” from the dropdown menu. You can change the number of days that are displayed by typing a number into the “Over how many days…” text field and pressing submit. To see the “Total Number of Words by Part of Speech” graph, select “All Words / Part of Speech” from the dropdown menu.