**Break down of the file structure**

The main folder

* /lib

Under that is the sub folders:

* /constants
* /data
* /models
* /routes
* /shared
* /utils
* /views

and the main dart file

* Main.dart

Main.dart:  
  
This code is a Dart program that sets up and runs a Flutter application. It uses the Hive package for local data storage and follows a structured approach to manage app settings and navigation. The application's UI and behavior depend on the values stored in the Hive box named Constants.appSettings.

* **Main Function (Entry Point):**
  + main() function is the entry point of the application.
  + It initializes Hive for data storage and opens a Hive box named Constants.appSettings.
  + Finally, it runs the Flutter app by wrapping it in a ProviderScope.
* **MyApp Widget:**
  + MyApp is the main application widget.
  + It uses a ValueListenableBuilder to listen for changes in the Constants.appSettings Hive box.
  + Inside the builder, it retrieves the values of showStart and hasPassword settings from the box.
  + It creates a MaterialApp widget to define the overall theme and initial route of the app.
  + The initial route is determined based on the showStart and hasPassword settings.

**/constants**

Constants.dart:  
  
This class serves as a centralized repository for various constants used in the application, including text strings, language keys, month names, and more. It's organized for easy access and localization of content.

**/data**

Budget\_notifier.dart:  
  
This BudgetNotifier class is used to manage the state of budget manager file data in the application using Riverpod. It provides methods for reading, saving, and deleting budgets, and it keeps the state updated accordingly.

Password\_notifier.dart  
  
This PasswordNotifier class is used to manage the state of password data in the application using Riverpod. It provides methods for reading, writing, and deleting passwords and keeps the state updated accordingly.

Settings\_notifier.dart  
  
This SettingsNotifier class is used to manage the state of app settings/user preferences data in the application using Riverpod. It provides methods for reading, writing, and deleting passwords and keeps the state updated accordingly.

Transaction\_notifier.dart  
  
This TransactionNotifier class is used to manage the state of app settings/user preferences data in the application using Riverpod. It provides methods for reading, writing, and deleting passwords and keeps the state updated accordingly.

**/models**  
  
budget\_model.dart  
  
This Budget class is a simple data class used to represent budget information. It has three properties: year, month, and budgetName, which are used to store details about the main budget manager file.  
This class is used to store and manipulate budget data in the application. You can create instances of Budget, and if needed, use the copyWIth() method to create a new instance with modified properties

Password\_model.dart

This Password class is a data class used to represent user password information. It has three properties: password, email, and salt, which are used to store details related to a user's password.  
This class is typically used for managing user passwords and associated information in the application. You can create instances of Password, and if needed, use the copyWith method to create a new instance with modified properties.

transaction\_model.dart  
  
This Transaction class is a data class used to represent transaction information. It has four properties: transactionType, date, amount, and note, which are used to store details about a financial transaction.  
This class is typically used to store and manage transaction data in the application. You can create instances of Transaction, and if needed, use the copyWith method to create a new instance with modified properties.

/routes  
  
routes.dart:

This code defines a mapping of named navigation routes for the Flutter application. Each route is associated with a specific screen (widget) that will be displayed when the corresponding route is navigated to.

Here's what the code does:

appRoutes is a variable that holds a Map where the keys are route names (e.g. StartPage.pageKey) and the values are functions that build and return the corresponding screen widgets.

For example, when you navigate to the route StartPage.pageKey, it will build and display the StartPage widget.

Similarly, other routes such as HomePage, AddExpensePage, EditExpensePage, etc., are associated with their respective screen widgets.

These routes are typically used with Flutter's Navigator to navigate between screens within the app.

Note that the screen widgets are specified as constants, such as const StartPage(), const HomePage(), etc., which indicates that they are stateless widgets and do not depend on external data.

**/shared**

Custom\_appbar.dart

The CustomAppBar class defines a custom app bar for the Flutter application. This custom app bar has certain visual properties that differ from the default AppBar in terms of colors and elevation. It provides a method showAppBar to create and return an instance of this custom app bar.

Here's a brief explanation of the code:

* CustomAppBar is a class that encapsulates the customization of the app bar.
* The showAppBar method takes three optional parameters:
  + title: A widget that represents the title of the app bar.
  + leading: A widget that represents the leading widget in the app bar (usually used for back buttons or navigation).
  + actions: A list of widgets that represent the action items in the app bar (e.g., buttons).
* Inside the showAppBar method, an AppBar widget is created with the following properties:
  + backgroundColor: The background color of the app bar (white in this case).
  + shadowColor: The color of the app bar's shadow (white in this case, effectively making it invisible).
  + surfaceTintColor: The color of the app bar's surface (white in this case).
  + elevation: The elevation (shadow) of the app bar (set to 0, making it flat).
  + title: The title widget passed as a parameter.
  + leading: The leading widget passed as a parameter.
  + actions: The list of action widgets passed as a parameter.

This custom app bar can be used in the Flutter application by creating an instance of CustomAppBar and calling its showAppBar method to create the app bar with the desired customization. It's especially useful if you want to have an app bar with a specific look and feel that differs from the default app bar.

Custom\_button.dart

The CustomButton class defines a custom button widget for the app. This custom button has certain visual properties and is designed to be easily customized in terms of text, width, and press action. It provides a method showButton to create and return an instance of this custom button.

Here's an explanation of the code:

* CustomButton is a class that encapsulates the customization of the button.
* The showButton method takes the following parameters:
  + context: A required BuildContext used to get the device's screen width.
  + text: A required String that represents the text displayed on the button.
  + onPress: A required VoidCallback that defines the action to be executed when the button is pressed.
  + width: An optional double that represents the width of the button. If not provided, a default width of 120 is used.
* Inside the showButton method:
  + The device's screen width is obtained using MediaQuery.
  + A TextStyle for the button text is created with a font size based on the device's width.
  + An ElevatedButton widget is created with the following properties:
    - style: Styling properties for the button, including text color, background color, size, padding, and shape.
    - onPressed: The callback function to be executed when the button is pressed.
    - child: The button's text, styled with the previously defined TextStyle.

This custom button can be used in the app by creating an instance of CustomButton and calling its showButton method to create a button with the desired customization. Useful when you want to create buttons with consistent styling throughout the app.

Custom\_date\_picker.dart

The DatePicker class provides methods for selecting and displaying a date picker dialog in the app. It also has a method to set the current date in a TextEditingController.

Here's an explanation of the code:

* DatePicker is a class that encapsulates date selection and formatting functionality.
* \_selectedDate is a private variable that holds the currently selected date. It's initially set to null.
* selectDate is a method that shows a date picker dialog and allows the user to select a date. It takes two parameters:
  + context: A required BuildContext used to display the date picker dialog.
  + textEditingController: A required TextEditingController used to update the text field with the selected date.
* Inside selectDate:
  + The showDatePicker function is called, which displays the date picker dialog.
  + It specifies the initial date, date range (from 2000 to 2040), and the visual appearance of the date picker using a custom builder.
* If the user selects a new date (newSelectedDate is not null):
  + \_selectedDate is updated with the selected date.
  + The selected date is formatted using DateFormat to the format 'dd MMMM yyyy'.
  + The textEditingController is updated with the formatted date, and the cursor position is set to the end of the text.
* setCurrentDate is a method that sets the current date in a TextEditingController. It takes one parameter:
  + textEditingController: A required TextEditingController used to update the text field with the current date.
* Inside setCurrentDate:
  + The current date is obtained using DateTime.now().
  + The current date is formatted using DateFormat to the format 'dd MMMM yyyy'.
  + The textEditingController is updated with the formatted current date, and the cursor position is set to the end of the text.

These methods allow you to easily add date selection and formatting functionality to the Flutter app's text fields. The selectDate method is particularly useful for displaying a date picker dialog, while setCurrentDate sets the current date in a text field.

Custom\_dialog.dart

The CustomDialog class provides a method for displaying a custom dialog box in the app. This dialog includes a title, content, and two buttons for confirmation and denial.

Explanation of the code:

* CustomDialog is a class that encapsulates the creation of custom dialogs.
* The showDialogBuilder method takes the following parameters:
  + context: A required BuildContext used to display the dialog.
  + title: An optional string for the dialog's title (default is null).
  + content: A required string for the dialog's content.
  + deny: A required string for the text displayed on the denial button.
  + confirm: A required string for the text displayed on the confirmation button.
* Inside the showDialogBuilder method:
  + The device's screen width is obtained using MediaQuery to calculate the size of various UI elements.
  + Text styles for the title, content, and button text are defined based on the device's width.
* The showDialog function is called to display the dialog with the following properties:
  + context: The current BuildContext.
  + barrierDismissible: Set to false to prevent the user from dismissing the dialog by tapping outside.
  + builder: A callback function that returns an AlertDialog.
* Inside the AlertDialog:
  + The title and content are displayed with the specified text styles.
  + The actions section contains two buttons:
    - A "Deny" button (TextButton) with the specified text and an onPressed callback that pops the dialog with a value of false.
    - A "Confirm" button (TextButton) with the specified text and an onPressed callback that pops the dialog with a value of true.
* The showDialogBuilder method returns a Future<bool?>, which represents the user's choice (true for confirmation, false for denial, and null if dismissed).

You can use this CustomDialog class to display custom dialogs with the specified title, content, and button text in the app. It provides flexibility in customizing the appearance and behavior of the dialogs.

Custom\_snackbar.dart

The CustomSnackbar class provides a method for displaying a custom snackbar in app. This snackbar includes a message and an optional duration.

Explanation of the code:

* CustomSnackbar is a class that encapsulates the creation and display of custom snackbars.
* The showSnackBar method takes the following parameters:
  + context: A required BuildContext used to display the snackbar.
  + text: A required string for the snackbar's message.
  + ms: An optional integer representing the duration of the snackbar in milliseconds (default is 1500).
* Inside the showSnackBar method:
  + The device's screen width is obtained using MediaQuery to calculate the font size for the snackbar text.
  + A TextStyle for the snackbar text is defined based on the device's width.
* The ScaffoldMessenger.of(context) is used to access the scaffold's messenger for displaying the snackbar.
* Before showing a new snackbar, any existing snackbar is removed using removeCurrentSnackBar() to ensure that only one snackbar is displayed at a time.
* A new SnackBar is displayed with the following properties:
  + backgroundColor: The background color of the snackbar.
  + duration: The duration for which the snackbar is displayed.
  + content: The snackbar's content, which includes the message with the specified text style.
* The if (context.mounted) check ensures that the snackbar is displayed only if the widget is still mounted. This prevents issues when trying to display a snackbar after the widget has been disposed.

You can use this CustomSnackbar class to easily show custom snackbars with a specified message and duration in the app. It provides flexibility in customizing the appearance and behavior of the snackbars.

Disable\_focus\_node.dart

The AlwaysDisabledFocusNode class is a custom implementation of a FocusNode in Flutter. This class is designed to override the behavior of the hasFocus property, ensuring that it always returns false, effectively making the focus node permanently disabled.

Explanation of the code:

* AlwaysDisabledFocusNode is a class that extends FocusNode, inheriting its properties and methods.
* Inside the AlwaysDisabledFocusNode class, the hasFocus property is overridden by providing a custom getter.
* The custom hasFocus getter always returns false, indicating that the focus node should never have focus. This effectively makes the focus node permanently disabled, preventing it from gaining focus under any circumstances.

shared.dart

This is a common practice to organize and reuse code across different parts of the app.  
By exporting these files, you make these custom utilities and widgets available for use in other parts of the Flutter application.

**/utils or utilities**

File\_manager.dart

This class handles various file operations such as reading, writing, editing, and deleting data from different types of files.

Here's what this class does:

1. **Writing Budget Data to File**:
   * The writeDataToBudgetManagerFile method writes budget-related data (related to the budget management feature) to a file. It formats the data and appends it to the file if the file already exists, or creates a new file if it doesn't.
2. **Reading Budget Data from File**:
   * The readDataFromBudgetManagerFile method reads budget data from a file and parses it into a list of Budget objects. It splits the lines of the file and constructs Budget objects from the parsed data.
3. **Writing Transaction Data to File**:
   * The writeDataToTxnsFile method writes transaction-related data to a file. Similar to the budget data, it formats and appends data to the file if it exists or creates a new file.
4. **Reading Transaction Data from File**:
   * The readDataFromTxnsFile method reads transaction data from a file and parses it into a list of Transaction objects. It splits the lines of the file and constructs Transaction objects from the parsed data.
5. **Writing Password Data to File**:
   * The writeDataToPasswordFile method writes password-related data (for user authentication) to a file.
6. **Reading Password Data from File**:
   * The readDataFromPasswordFile method reads password data from a file and parses it into a list of Password objects.
7. **Editing a Line in a File**:
   * The editLineInFile method allows you to edit an existing line in a file. It identifies the line by its line number and replaces it with new data.
8. **Deleting a Line from a File**:
   * The deleteLineFromFile method deletes a specific line from a text file, identified by its line number.
9. **Deleting a File**:
   * The deleteFile method is used to delete an existing file.

This class provides essential functionality for managing data persistence in the app, particularly for budget and transaction-related features. It's a central component for interacting with files and storing structured data on the device.

Notification\_manager.dart

The NotificationManager class you provided is responsible for managing notifications in the app using the flutter\_local\_notifications package.

Here's what this class does:

1. **Initialization**:
   * In the init method, the FlutterLocalNotificationsPlugin is initialized with Android-specific settings. This is typically called when the app starts to set up the notification system.
2. **Scheduling a Monthly Reminder Notification**:
   * The scheduleReminderNotification method schedules a notification reminding the user to create a budget.
   * It initializes the time zone settings using the timezone package.
   * It calculates the date for the 28th of the current month and the next month.
   * It defines Android-specific notification settings such as channel ID, description, importance, and priority.
   * It schedules the notification using \_flutterLocalNotificationsPlugin.zonedSchedule with the following parameters:
     + id: An identifier for the notification.
     + title: Title of the notification.
     + body: Notification message.
     + scheduledDate: Date and time when the notification should be shown (28th of the current or next month).
     + platformChannelSpecifics: Notification details, including Android-specific settings.
     + androidScheduleMode: Android schedule mode to determine when to allow scheduling while the device is in an idle state.
     + uiLocalNotificationDateInterpretation: Interpretation of the scheduled date.
   * It logs a message indicating that the notification has been scheduled.
3. **Cancelling a Scheduled Notification**:
   * The cancelScheduledNotification method cancels the scheduled notification with ID 0 using \_flutterLocalNotificationsPlugin.cancel. This is for when you want to cancel the reminder notification at some point.

Password\_manager.dart

The PasswordManager class provides functionality for managing passwords within the app.

Here's a explanation of what this class does:

1. **generateSecureSalt(int length)**:
   * This method generates a secure salt, which is a random sequence of bytes used to enhance password security.
   * It takes an integer length as a parameter, indicating the length of the salt to be generated.
   * Inside the method, a secure random number generator is created using Random.secure().
   * The salt is generated as a list of random integers, each ranging from 0 to 255, with a total length specified by length.
   * The salt is then converted into a Uint8List and returned.
2. **hashValue(String value, Uint8List salt)**:
   * This method takes a String value (password or email) and a Uint8List salt as parameters.
   * It uses the SHA-256 cryptographic hash function to hash the combination of the value and salt.
   * The value is first encoded to bytes using UTF-8 encoding.
   * The salt is combined with the encoded value by concatenating their bytes.
   * The resulting data is hashed using SHA-256, and the hash is returned as a Uint8List.
3. **isValueMatching**:
   * This method compares a given hash value (givenHashValue) with a stored hash value (storedHashValue) to check if they match.
   * It also takes a String parameter hashSalt, which represents the salt used in hashing.
   * The method converts the hashSalt from its string representation to a Uint8List using the convertStringToUint8List method.
   * It then computes the hash of the givenHashValue using the converted hashSalt.
   * Finally, it compares the computed hash with the storedHashValue and returns true if they match or false if they don't.
4. **convertStringToUint8List({required String hashSalt})**:
   * This method converts a string representation of byte values (e.g., "[1, 2, 3]") into a Uint8List.
   * It takes a required parameter hashSalt, which is the string representation of the byte values.
   * The method first removes square brackets and splits the string by ', ' to obtain individual byte values as strings.
   * It converts these byte strings to integers and stores them in a list.
   * Finally, it creates a Uint8List from the list of integers and returns it.

The PasswordManager class provides methods for generating secure salts, hashing values with salts, and checking if a given value matches a stored hash value using a salt. It also includes a utility method for converting a string representation of byte values into a Uint8List. These functions are useful for secure password management within a Flutter application.

utils.dart  
exports the above files again

**/Views**

Start\_page.dart

The StartPage class is part of the app and represents the initial screen that users see when they first launch the app.

Here's a breakdown of what this class does:

1. **Imports**:
   * The dart:io library is imported as io to handle file operations.
   * Various other dependencies from the app, including constants, providers, and widgets, are imported.
2. **Class Declaration**:
   * StartPage is defined as a ConsumerWidget. It takes no named arguments and is marked as const.
   * The pageKey is defined as a static constant with the route identifier for this page.
3. **Build Method**:
   * The build method is implemented, as required by the ConsumerWidget interface. It takes the BuildContext and WidgetRef as parameters.
   * WidgetRef - An object that allows widgets to interact with providers.
   * The CustomButton class is instantiated, which is used to create a custom button widget.
   * The language and indexValue variables are declared. language is obtained from a Riverpod provider (notiProvLanguages), and indexValue is set to 0.
   * The start variable is initialized with a localized start text obtained from the language map using the indexValue and the Constants.startText key.
4. **Scaffold Widget**:
   * The Scaffold widget is used as the main layout container for the page.
5. **Centered Content**:
   * The Center widget is used to center its child vertically and horizontally within the screen.
6. **Custom Button**:
   * The customButton.showButton method is called to create a custom button widget.
   * The context is passed as a parameter.
   * The button's text is set to the start variable, which represents the localized start text.
   * The onPress callback is defined, which is executed when the button is pressed.
7. **Button's OnPress Callback**:
   * When the button is pressed, the following actions are performed:
     + User Default Preferences:
       - User default preferences are stored in a Hive box (a local NoSQL database). These preferences include settings related to showing the start page, using a password, enabling reminders, setting the app's language, and indicating whether a new budget has been created.
     + Creating an Initial Budget:
       - An initial budget is created when the app is first launched. It generates a budget file with the current year and the previous month, e.g., "202308.txt."
       - The budget data is written to a file in the app's documents directory.
     + Navigation:
       - The app navigates to the HomePage using Navigator.pushAndRemoveUntil.

This StartPage serves as the entry point to the app and allows users to start using the budget management application. It initializes user preferences and creates an initial budget file if it's the first app launch.

Home\_page.dart

The HomePage class is a key part of the app and represents the main page where users manage their budget.

Breakdown of what this class does:

1. **Imports**:
   * The dart:developer library is imported to allow logging.
   * Various dependencies from the app, including other views, constants, data providers, and utilities, are imported.
2. **Class Declaration**:
   * HomePage is defined as a ConsumerWidget. It takes no named arguments and is marked as const.
   * The pageKey is defined as a static constant with the route identifier for this page.
3. **Build Method**:
   * The build method is implemented, as required by the ConsumerWidget interface. It takes the BuildContext and WidgetRef as parameters.
   * Several widgets and variables are declared and initialized within this method.
4. **Initialization**:
   * Instances of CustomAppBar and NotificationManager are created.
   * The deviceWidth variable is used to calculate font sizes based on the device's screen width.
   * Text styles (mainTextStyle, unselectedTextStyle, selectedTextStyle) are defined for various UI elements.
   * Several Riverpod providers (budgetManagerData, budgetIndex, transactionData, budgetData, languages, months, index, isBudgetCreated, isBefore28th, rescheduleReminder) are watched to obtain relevant data.
5. **Rescheduling Reminder Notification**:
   * The rescheduleReminder provider is watched to determine if the user has enabled notifications. If so, it schedules a reminder notification using the NotificationManager.
6. **IndexValue for Language**:
   * The index provider is watched to determine the selected language index.
   * The home, transaction, and file variables are set to localized text based on the selected language.
7. **Budget Data Retrieval**:
   * The budgetManagerData provider is watched to obtain budget-related data, such as year, month, name, filename, and the number of budgets.
   * If data is available, it populates the relevant variables.
8. **Budget Creation Logic**:
   * The isBudgetCreated provider is watched to check if the user has created a budget before the 28th of the month.
   * If the app is launched before the 28th, it sets the value to false to prevent creating a new budget for the next month.
9. **Tab Widget**:
   * A tab function is defined to create a styled tab widget with a given text and width.
10. **App Bar**:
    * A custom app bar is displayed at the top of the page. It includes the budget name and a settings icon that allows users to navigate to the settings page.
11. **Page Body**:
    * The page body is divided into two expanded sections: one for navigation buttons and the other for the tab bar and tab view.
    * Navigation buttons (IconButton) for navigating between budgets are included.
    * A DefaultTabController is used to manage the tabs.
    * The TabBar and TabBarView widgets are used to create a tabbed interface with three tabs: Home, Transactions, and File.
    * Each tab content is displayed based on the selected tab.
12. **Tab Content**:
    * Three tabs are implemented:
      + **HomeTab**: Displays budget summary information and allows users to create transactions for the selected budget.
      + **TransactionTab**: Lists and manages transactions for the selected budget.
      + **ViewBudgetTab**: Displays budget details.

This HomePage serves as the main interface for budget management, providing access to different tabs for budget-related actions. It also supports localization and handles notifications.

**/home\_tabs**

Home\_tabs.dart

Exports home tabs

Tab\_home.dart

The HomeTab class in your Flutter application represents one of the tabs on the home page where users can manage their budget.

Here's what this class does:

1. **Imports**:
   * The dart:developer library is imported to allow logging.
   * Various dependencies from the app, including constants, data providers, and models, are imported.
2. **Class Declaration**:
   * HomeTab is defined as a ConsumerWidget that takes three required parameters: transactionData, currentFileName, and index.
3. **Build Method**:
   * The build method is implemented, as required by the ConsumerWidget interface. It takes the BuildContext and WidgetRef as parameters.
   * Widgets and variables are declared and initialized within this method.
4. **Initialization**:
   * The deviceWidth variable is used to calculate font sizes based on the device's screen width.
   * A text style (textStyle) and icon size (iconSize) are defined for UI elements.
   * Three variables (totalIncome, totalExpense, totalBalance) are initialized to calculate transaction totals.
5. **Provider Watchers**:
   * Various Riverpod providers are watched to obtain data, including language, language index, budget index, and index value.
6. **Language Localization**:
   * The indexValue variable is set based on the language index.
   * Text strings for various UI elements (e.g., "Get Started," "Income," "Expense," "Balance") are localized based on the selected language.
7. **Column Widget**:
   * The Column widget is used to arrange child widgets vertically.
8. **Transaction Data Handling**:
   * Inside the transactionData.when block:
     + If there is no transaction data, it displays a message to guide the user on how to get started.
     + If there is transaction data:
       - It calculates the total income, total expense, and total balance by iterating through the transactions.
       - Transactions are categorized as income or expense based on their type.
       - The total balance is calculated as the difference between total income and total expense.
9. **Display Transaction Totals**:
   * The calculated transaction totals (income, expense, and balance) are displayed using Text widgets arranged in a Column.
   * These totals are displayed alongside their corresponding labels (e.g., "Income: 500.00").
10. **Add Expense and Income Buttons**:
    * Two columns are displayed side by side.
    * Each column contains a label (either "Expense" or "Income") and an "Add" button.
    * The "Add" button is an IconButton that navigates the user to the respective expense or income page when clicked.

This HomeTab class provides an interface for viewing transaction totals and navigating to add expense or income transactions. It also supports localization for different languages.

Tab\_transaction.dart:

The provided code defines two Flutter widgets: TransactionTab and TransactionList. These widgets are part of a budget management app.

A breakdown of what each of these widgets does:

**TransactionTab:**

1. **Imports and Class Declaration**:
   * Import statements for necessary libraries and dependencies are included.
   * The TransactionTab class is defined as a ConsumerWidget and takes three required parameters: transactionData, currentFileName, and budgetIndex.
2. **Build Method**:
   * The build method is implemented as required by the ConsumerWidget interface.
   * Inside this method, various variables and styles are initialized.
   * Riverpod providers are watched to obtain language-related data.
3. **Language Localization**:
   * The indexValue variable is set based on the language index.
   * A text string (getStartedTwoText) is localized based on the selected language.
4. **Widget Building**:
   * The transactionData.when block is used to handle different states of the transactionData (e.g., loading, data available, or error).
   * If there is no transaction data, it displays a "Get Started" message to guide the user on how to get started.
   * If transaction data is available, it displays a TransactionList widget to show a list of transactions.

**TransactionList:**

1. **Imports and Class Declaration**:
   * Import statements for necessary libraries and dependencies are included.
   * The TransactionList class is defined as a ConsumerWidget and takes three required parameters: transactionData, currentFileName, and budgetIndex.
2. **Widget Building**:
   * The ListView.separated widget is used to display a list of transactions.
   * The itemBuilder parameter is used to build each item in the list.
   * Each transaction item is wrapped in an InkWell to make it tappable.
   * The transaction details, including the note and amount, are displayed within each item.
   * Tapping on a transaction item navigates the user to either the EditIncomePage or EditExpensePage based on the transaction type.
   * A separatorBuilder is used to insert dividers between each transaction item.

These widgets provide a user interface for displaying and interacting with a list of transactions in a budget management app. The transactions are localized based on the selected language, and users can edit them by tapping on individual items.

Tab\_view\_budget\_history.dart:

The provided code defines two Flutter widgets: ViewBudgetTab and BudgetList. These widgets are part of a budget management app.

Explanation of what each of these widgets does:

**ViewBudgetTab:**

1. **Imports and Class Declaration**:
   * Import statements for necessary libraries and dependencies are included.
   * The ViewBudgetTab class is defined as a ConsumerWidget and takes one required parameter: budgetData.
2. **Build Method**:
   * The build method is implemented as required by the ConsumerWidget interface.
   * Inside this method, the budgetData.when block is used to handle different states of the budgetData (e.g., loading, data available, or error).
   * If data is available, it returns a BudgetList widget to display a list of budget items.

**BudgetList:**

1. **Imports and Class Declaration**:
   * Import statements for necessary libraries and dependencies are included.
   * The BudgetList class is defined as a ConsumerStatefulWidget and takes one required parameter: budgetData.
2. **State Class**:
   * \_BudgetListState is defined as the state class for BudgetList. This class extends ConsumerState<BudgetList>.
3. **Build Method (State)**:
   * The build method in the state class is implemented to create the widget's UI.
   * Inside this method, various widgets and styles are initialized.
   * Riverpod providers are watched to obtain language-related data and information about whether the next budget has been created.
4. **Language Localization**:
   * The indexValue variable is set based on the language index.
   * Text strings for creating and viewing budgets are localized based on the selected language.
5. **Budget List**:
   * A ListView.separated widget is used to display a list of budget items.
   * Each item in the list shows the budget month and year.
   * Tapping on a budget item navigates the user to the ViewBudgetPage with relevant arguments.
6. **Create Budget Button**:
   * The visibility of the "Create Budget" button is controlled based on conditions like the current date being after the 28th of the month and whether the next budget has been created.
   * Tapping the button triggers the creation of a new budget and shows a snackbar with a message.

These widgets provide a user interface for displaying a list of budgets, allowing the user to view and create new budgets in a budget management app. The user interface is localized based on the selected language, and the creation of a new budget is controlled based on the current date and the status of the next budget.

Add\_expense.dart:

The provided code defines a Flutter widget

Here's a breakdown of what this widget does:

1. **Imports**:
   * The widget imports the necessary libraries, including add\_transaction\_page.dart and flutter/material.dart.
2. **Class Declaration**:
   * The AddExpensePage class is defined as a stateless widget that takes no named parameters.
3. **Static PageKey**:
   * A static constant string pageKey is defined, which represents the route key for this page. It is set to '/add\_expense'.
4. **Build Method**:
   * Inside the build method, a const AddTransactionPage widget is created. This widget is responsible for adding transactions and is passed the pageType parameter set to AddPageType.addExpense. This parameter determines the type of transaction being added.

The AddExpensePage serves as a wrapper for the AddTransactionPage with a specific page type, indicating that it's for adding an expense transaction. When navigating to this page, it opens the AddTransactionPage for adding expense details.

Add\_income.dart:

The provided code defines another Flutter widget, AddIncomePage, which is similar in structure to the previously explained AddExpensePage.

Here's a breakdown of what this widget does:

1. **Imports**:
   * The widget imports the necessary libraries, including add\_transaction\_page.dart and flutter/material.dart.
2. **Class Declaration**:
   * The AddIncomePage class is defined as a stateless widget that takes no named parameters (super.key is specified but not used).
3. **Static PageKey**:
   * A static constant string pageKey is defined, representing the route key for this page. It is set to '/add\_income'.
4. **Build Method**:
   * Inside the build method, a const AddTransactionPage widget is created. This widget is responsible for adding transactions and is passed the pageType parameter set to AddPageType.addIncome. This parameter helps determine the type of transaction being added.

Add\_transaction\_page.dart:

This widget can handle both income and expense transactions based on the pageType specified when creating an instance of this widget.

Here's a breakdown of what this widget does:

1. **Imports**:
   * The widget imports the necessary libraries, including various custom widgets and flutter/material.dart.
2. **Enum AddPageType**:
   * This enum defines two types of pages: addIncome and addExpense. It is used to specify the type of transaction being added.
3. **Class Declaration**:
   * The AddTransactionPage class is defined as a stateful widget that takes two named parameters:
     + pageType: An instance of the AddPageType enum that specifies whether it's an income or expense transaction page.
4. **State Class \_AddTransactionPageState**:
   * This class extends ConsumerState<AddTransactionPage>, indicating that it manages the state for the AddTransactionPage.
5. **Controller Variables**:
   * Several TextEditingController instances are declared to manage the text input fields for date, amount, and note.
6. **Dispose Method**:
   * The dispose method is overridden to dispose of the text editing controllers when the widget is no longer needed.
7. **Build Method**:
   * It sets up various UI elements for adding a transaction, including text fields for date, amount, and note.
   * It uses custom widgets such as CustomAppBar and CustomButton for consistent styling.
   * The UI elements are localized based on the selected language.
8. **Logic for Saving Transaction**:
   * When the "Save" button is pressed, the widget checks if the date, amount, and note fields are empty. If any of them are empty, it displays a snackbar indicating that fields cannot be empty.
   * If all fields are filled, it creates a Transaction object and adds it to the transaction data using the asyncTransactionProvider. It also displays a snackbar indicating that the transaction has been saved.

Edit\_expense.dart:

This widget is a wrapper for the EditTransactionPage, specifying that it's an expense edit page.

Here's what this widget does:

1. **Imports**:
   * The widget imports the necessary libraries, including budget\_app/views/edit\_transaction\_page.dart and flutter/material.dart.
2. **Class Declaration**:
   * The EditExpensePage class is defined as a stateful widget that takes no named parameters.
   * It specifies the type of state it returns as \_EditExpensePageState.
3. **Static String pageKey**:
   * This static constant string, pageKey, is defined as '/edit\_expense', which likely represents the route name for this page.
4. **State Class \_EditExpensePageState**:
   * This class extends State<EditExpensePage>, indicating that it manages the state for the EditExpensePage.
5. **Build Method**:
   * It returns an instance of EditTransactionPage, specifying that it's an expense edit page (EditPageType.editExpense).

It sets the EditPageType to editExpense when creating an instance of the EditTransactionPage.

Edit\_income.dart:

Similar to the EditExpensePage, this widget is a wrapper for the EditTransactionPage, specifying that it's an income edit page.

Here's what this widget does:

1. **Imports**:
   * The widget imports the necessary libraries, including budget\_app/views/edit\_transaction\_page.dart and flutter/material.dart.
2. **Class Declaration**:
   * The EditIncomePage class is defined as a stateful widget that takes no named parameters.
   * It specifies the type of state it returns as \_EditIncomePageState.
3. **Static String pageKey**:
   * This static constant string, pageKey, is defined as '/edit\_income', which likely represents the route name for this page.
4. **State Class \_EditIncomePageState**:
   * This class extends State<EditIncomePage>, indicating that it manages the state for the EditIncomePage.
5. **Build Method**:
   * It returns an instance of EditTransactionPage, specifying that it's an income edit page (EditPageType.editIncome).

Just like the EditExpensePage, the EditIncomePage widget serves as a wrapper for the EditTransactionPage, specifically designed for editing income transactions. It sets the EditPageType to editIncome when creating an instance of the EditTransactionPage.

Edit\_transaction\_page.dart:

The provided code defines a Flutter widget called EditTransactionPage, which is a page for editing both income and expense transactions in a budget management app.

Here's a breakdown of what this widget does:

1. **Imports**:
   * The widget imports various libraries, including budget\_app/constants/constants.dart, budget\_app/data/settings\_notifier.dart, budget\_app/data/transaction\_notifier.dart, and others, to access necessary functionalities and widgets.
2. **Enum EditPageType**:
   * This enum defines two values: editIncome and editExpense, which represent the type of page for editing a transaction (either income or expense).
3. **Class Declaration**:
   * The EditTransactionPage class is defined as a stateful widget that takes a named parameter pageType of type EditPageType. This parameter determines whether the page is for editing income or expense transactions.
   * It specifies the type of state it returns as \_EditTransactionPageState.
4. **State Class \_EditTransactionPageState**:
   * This class extends State<EditTransactionPage>, indicating that it manages the state for the EditTransactionPage.
5. **Instance Variables**:
   * Several instance variables are declared, including \_datePicker, \_customSnackbar, \_customButton, \_customDialog, and three TextEditingController instances (\_dateTextController, \_amountTextController, \_noteTextController). These are used for handling date input, displaying snackbar messages, buttons, dialogs, and managing text input fields.
6. **Build Method**:
   * It retrieves arguments passed to the page using ModalRoute.of(context)!.settings.arguments and sets the initial values of date, amount, and note text controllers based on these arguments.
   * It calculates screen size-related properties, such as deviceWidth and iconSize, and defines TextStyle objects for text styling.
   * The UI consists of a title, input fields for date, amount, and note, and two buttons for saving and deleting the transaction.
   * The user can edit the transaction details, and there are checks for empty fields before saving.
   * A delete confirmation dialog is displayed when the user clicks the delete button.
   * Snackbar messages are shown to confirm actions (e.g., transaction deletion or update).

The EditTransactionPage widget provides a user interface for editing income and expense transactions, with the ability to save changes or delete the transaction. The specific functionality depends on the pageType parameter passed when creating an instance of this widget.

Password\_create\_page.dart:

The provided code defines a Flutter widget called PassCreatePage, which is a page for creating a password in a budget management app.

Here's a breakdown of what this widget does:

1. **Imports**:
   * The widget imports various libraries, including budget\_app/constants/constants.dart, budget\_app/data/password\_notifier.dart, budget\_app/data/settings\_notifier.dart, and others, to access necessary functionalities and widgets.
2. **Class Declaration**:
   * The PassCreatePage class is defined as a stateful widget without any required parameters. It specifies the type of state it returns as \_CreatePasscodePageState.
3. **State Class \_CreatePasscodePageState**:
   * This class extends State<PassCreatePage>, indicating that it manages the state for the PassCreatePage.
4. **Instance Variables**:
   * Several instance variables are declared, including passwordField, confirmPasswordField, and emailField. These are used for handling user input for creating a password.
5. **Build Method**:
   * It calculates screen size-related properties, such as deviceWidth and deviceHeight, and defines TextStyle objects for text styling.
   * The UI consists of a title, input fields for a new password, confirm password, and email address. It also includes a button for saving the password.
   * The user can create a new password, and there are checks for empty fields, matching passwords, and valid email addresses.
   * If the password creation is successful, the user is redirected to the password login screen.

The PassCreatePage widget provides a user interface for creating a password for the budget management app, with validation checks for password matching and valid email addresses.

Password\_forgot\_page.dart:

The provided code defines a Flutter widget called PassForgotPage, which is a page for handling password recovery in a budget management app.

A breakdown of what this widget does:

1. **Imports**:
   * The widget imports various libraries, including budget\_app/constants/constants.dart, budget\_app/data/password\_notifier.dart, budget\_app/data/settings\_notifier.dart, and others, to access necessary functionalities and widgets.
2. **Class Declaration**:
   * The PassForgotPage class is defined as a stateful widget without any required parameters. It specifies the type of state it returns as \_PassForgotPageState.
3. **State Class \_PassForgotPageState**:
   * This class extends State<PassForgotPage>, indicating that it manages the state for the PassForgotPage.
4. **Instance Variables**:
   * Several instance variables are declared, including \_emailController for handling email input, and various custom widgets like \_customAppBar, \_customSnackbar, \_passwordManager, and \_customButton.
5. **Build Method**:
   * It calculates screen size-related properties, such as deviceWidth and deviceHeight, and defines TextStyle objects for text styling.
   * The UI consists of a title, an input field for email, and a button for confirming the password recovery.
   * The user can enter their email, and there are checks for empty fields, matching email addresses, and valid email format.
   * If the email matches, the stored password is removed and the user is redirected to the home page.

The PassForgotPage widget provides a user interface for password recovery in the budget management app, with validation checks for email matching and valid email addresses.

Password\_login\_page.dart:

The provided code defines a Flutter widget called PassLoginPage, which is a login page for a budget management app that uses a password for authentication.

Here's a breakdown of what this widget does:

1. **Imports**:
   * The widget imports various libraries, including budget\_app/constants/constants.dart, budget\_app/data/password\_notifier.dart, budget\_app/data/settings\_notifier.dart, and others, to access necessary functionalities and widgets.
2. **Class Declaration**:
   * The PassLoginPage class is defined as a stateful widget without any required parameters. It specifies the type of state it returns as \_PassLoginPageState.
3. **State Class \_PassLoginPageState**:
   * This class extends State<PassLoginPage>, indicating that it manages the state for the PassLoginPage.
4. **Instance Variables**:
   * Several instance variables are declared, including \_passwordController for handling the password input, and various custom widgets like \_customSnackbar, \_passwordManager, and \_customButton.
5. **Build Method**:
   * It calculates screen size-related properties, such as deviceWidth and deviceHeight, and defines TextStyle objects for text styling.
   * The UI consists of a title, an input field for the password, a "Forgot Password?" button, and a login button.
   * The user can enter their password, and if it matches the stored password, they are redirected to the home page. Otherwise, an error message is displayed.

The PassLoginPage widget provides a user interface for password-based login in the budget management app, with password validation checks and the option to reset the password through the "Forgot Password?" button.

Settings\_page.dart

The provided code defines a Flutter widget called SettingsPage, which is a settings page for a budget management app.

Here's what this widget does:

1. **Imports**:
   * The widget imports various libraries, including budget\_app/constants/constants.dart, budget\_app/data/settings\_notifier.dart, and others, to access necessary functionalities and widgets.
2. **Class Declaration**:
   * The SettingsPage class is defined as a stateless widget without any required parameters. It overrides the build method to define the widget's UI.
3. **Build Method**:
   * It calculates screen size-related properties, such as deviceWidth and iconSize, and defines TextStyle objects for text styling.
   * The UI consists of:
     + A dropdown menu for selecting the app's language.
     + A switch for enabling/disabling a password setting.
     + A switch for enabling/disabling reminder notifications.
     + An "Exit" button for exiting the app.
   * The selected language, password setting, and reminder setting values are retrieved from ref (a ProviderContainer).
   * The language can be changed via the dropdown menu, and the password and reminder settings can be toggled using the switches.
   * Depending on the settings, the app may schedule reminder notifications or delete saved passwords.
   * The "Exit" button uses SystemNavigator.pop() to close the app.

The SettingsPage widget provides a user interface for configuring language preferences and app settings related to password management and reminders in the budget management app.

View\_budget\_page.dart:

Two widgets ViewBudgetPage and BudgetList.

Here's what this widget does:

**ViewBudgetPage Widget**

1. **Imports**: This section imports various libraries and dependencies required for the widget to function. These libraries provide access to fundamental Flutter and custom functionalities.
2. **Class Declaration**: The ViewBudgetPage class is defined as a stateless widget. This means it doesn't hold any mutable state and can't be changed after it's created.
3. **Build Method**: The build method constructs the user interface for the page. Here's a breakdown of the UI elements within this widget:
   * **AppBar**: The top part of the page contains an app bar with a back button (IconButton). When clicked, this button will navigate the user back to the previous screen. The app bar also displays the selected month and year as the title.
   * **Transaction List**: Below the app bar, there's a section for displaying a list of transactions. It uses the BudgetList widget to render this list. The transactions are fetched from the transactionData provider, which presumably provides a list of transactions for the selected budget.
   * **Delete Button**: At the bottom of the page, there's a "Delete" button. When clicked, it triggers a confirmation dialog using the customDialog.showDialogBuilder method. The dialog asks the user to confirm the deletion of either the entire budget or specific transactions.
   * **Localization**: The widget also handles localization. It uses the language and index variables to determine the language for text displayed on the page. Text such as "Income," "Expense," "Balance," and others are localized based on the selected language.
   * **Calculating Totals**: The widget calculates and displays the total income, total expense, and total balance for the selected budget based on the transaction data. These totals are displayed at the bottom of the transaction list.

**BudgetList Widget**

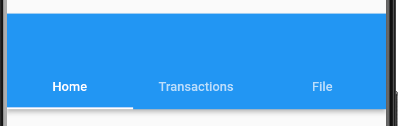
1. **Imports**: This section imports libraries and dependencies required for the BudgetList widget. It includes custom classes and constants for the app's functionality.
2. **Class Declaration**: The BudgetList class is defined as a stateless widget. It takes three required parameters: transactionData, fileName, and selectedBudgetIndex. These parameters are used to customize the appearance and behavior of the widget.
3. **Build Method**: The build method constructs the user interface for displaying a list of transactions. Here's a breakdown of the UI elements within this widget:
   * **Transaction List**: The core of this widget is a ListView.separated widget. It creates a scrollable list of transactions separated by dividers. The list is populated using the transactionData provided as input.
   * **Transaction Items**: For each transaction in the transactionData list, a ListTile is created. Each ListTile represents a transaction and contains:
     + A title displaying the transaction type ("Income" or "Expense").
     + A subtitle showing the transaction's note.
     + A trailing text displaying the transaction amount in the app's currency format.
   * **Calculating Totals**: The widget calculates and displays the total income, total expense, and total balance based on the provided transaction data. These totals are displayed in a separate Expanded widget at the bottom of the list.

This BudgetList widget is reusable and can be used in various parts of the app to display transaction lists. It's designed to provide a clean and organized representation of budget transactions.

Overall, the ViewBudgetPage and BudgetList widgets work together to create a user-friendly interface for viewing and managing budget transactions within the app. Users can navigate through transactions, calculate totals, and delete specific transactions or entire budgets as needed.

Misc. Information I had to look up to complete that application

Other links were added in the in description of the process.

* Googled how to write and read a text file in flutter  
  - <https://stackoverflow.com/questions/54122850/how-to-read-and-write-a-text-file-in-flutter>
* How to get tabs on the page  
  - <https://docs.flutter.dev/cookbook/design/tabs>

the code from the link above added an extra app bar space above the tabs, so I searched how to remove that  
- <https://stackoverflow.com/questions/50609252/flutter-tabbar-without-appbar>

* Changing the tab colour and Keeping state of the tabs

- <https://blog.logrocket.com/flutter-tabbar-a-complete-tutorial-with-examples/>

Last section in article

* How to add in the vertical dividers in the tab bar

- <https://stackoverflow.com/questions/74532801/how-to-make-vertical-lines-between-tabs-in-tabbar>

* Horizontal line  
  - <https://flutterforyou.com/how-to-create-a-horizontal-line-in-flutter/>
* Tab bar

- <https://blog.logrocket.com/flutter-tabbar-a-complete-tutorial-with-examples/>

* Vertical lines in tab

- <https://stackoverflow.com/questions/74532801/how-to-make-vertical-lines-between-tabs-in-tabbar>

* Date picker

- <https://stackoverflow.com/questions/54127847/flutter-how-to-display-datepicker-when-textformfield-is-clicked>

* For the navigation part of the app, destroy all previous screens when a certain action is done

- <https://stackoverflow.com/questions/69114951/how-can-i-pop-all-routes-and-push-a-new-one-popuntil-if-it-had-not-been-called>

* Only allow numbers and 2 decimals

- <https://stackoverflow.com/a/64947831>

* Exit app

- <https://stackoverflow.com/questions/45109557/flutter-how-to-programmatically-exit-the-app>

* Passing value through NamedRoute

- <https://stackoverflow.com/a/65846263>

* Find file on emu device  
  - <https://developer.android.com/studio/debug/device-file-explorer>
* Read and writing

- <https://docs.flutter.dev/cookbook/persistence/reading-writing-files>

* Passing arguments

- <https://docs.flutter.dev/cookbook/navigation/navigate-with-arguments>

* Convert datetime to tzdatetime

- <https://stackoverflow.com/a/66664835/17882863>

Added UI fields for the data and then looked up:

* Searched how to show a date picker  
  - <https://stackoverflow.com/a/62529777>
* What the intl package was and added it to the pubspec.yaml

- <https://pub.dev/packages/intl>

Resources for reading and writing and what not:

- <https://stackoverflow.com/questions/54122850/how-to-read-and-write-a-text-file-in-flutter#54122851>

- <https://docs.flutter.dev/cookbook/persistence/reading-writing-files>

- <https://stackoverflow.com/a/55025228>

- <https://stackoverflow.com/questions/51717700/is-there-any-way-to-write-to-a-pre-existing-text-file-in-flutter>

- <https://stackoverflow.com/questions/60137910/flutter-how-to-delete-a-file-inside-the-flutter-app-directory>

Added a snackbar (toast pop up message) to the deletion function to tell user if it’s been deleted or not in the add\_expense.dart file. And came across the error below, that’s the fix for it.

- <https://stackoverflow.com/questions/68871880/do-not-use-buildcontexts-across-async-gaps>

Edit functionality:  
- <https://stackoverflow.com/questions/22926306/replace-a-word-from-a-specific-line-in-a-text-file#22926445>

Added riverpod package for state management  
- <https://pub.dev/packages/flutter_riverpod>  
  
relearnt riverpod because I forgot a lot of it

- <https://riverpod.dev/docs/getting_started>

- <https://www.youtube.com/watch?v=grcgR9tUGiU>

- <https://docs-v2.riverpod.dev/docs/providers/notifier_provider>

- <https://github.com/rrousselGit/riverpod/discussions/1813#discussioncomment-3961971>

Reason for using riverpod/state management:

It’s the process of managing the data and the user interface of an application to ensure that changes in one part of the app are reflected consistently throughout the entire app.

For example, when a user adds an expense or income, itll update the balance in the app

Wrapped the main widget MyApp() in main.dart in a providerscope widget, reason for this is so that you can access the data throughout the app and allow the state to be controlled.

For the months, saving each month’s transactions to a text file as “20238.txt”, “20239.txt” The budget history list and the home page date scroller at the top, uses the main created budget file BudgetManager.txt where it keeps track of the different budget names eg “20238.txt”, “20239.txt”  
  
  
For app settings/user preference  
- <https://pub.dev/packages/hive/>

For password encryption the crypto package its used  
- <https://pub.dev/packages/crypto>

I had a problem with the initial file creation through riverpod, I hardcoded the it create the initial file instead.  
  
  
For notification I used timezone package with flutter local notification package  
- <https://pub.dev/packages/timezone>  
- <https://pub.dev/packages/flutter_local_notifications>  
  
  
app icon by <https://www.flaticon.com/authors/afif-fudin> free to use as long as dudes attributed

For handing permission = permission\_handler package used.  
different permissions used for different android versions, device\_info\_plus package used to get the android version of the device  
- <https://pub.dev/packages/permission_handler>  
- <https://pub.dev/packages/device_info_plus>  
  
  
For the languages setting I created a Map<int,Map<String,String>> so that I can use to change the users language preference for the app  
  
for example reading the value would look something like:  
  
language[language preference index][Language constant key]