P5. Expense App

> Expense is a money spending and analysis tracking app. Using this app, we trace how much we spend on what items.

1. Data Structure

- Transaction class

Transaction Class

- The Transaction class contains the information to constitute one transaction entity.

```
class Transaction {
  final String id; final String title;
  final double amount; final DateTime date;

  Transaction({
    this.id, this.title,
    this.amount, this.date,
  });
}
```

- Normally, a database system is used to store this transaction.
- For simplicity, use a list of transactions.

```
final List<Transaction> _userTransactions = [];
```

- We use the `_addNewTransaction()` service function to add a new transaction to the list.

```
void _addNewTransaction(
  String txTitle, double txAmount, DateTime chosenDate) {
  final newTx = Transaction(
    title: txTitle, amount: txAmount, date: chosenDate,
    id: DateTime.now().toString(),
  );
  // add to the list and redraw
  setState(() {_userTransactions.add(newTx);});
}
```

2. Service functions

- This application has many service functions.
- They are all closely related to corresponding user interfaces, so they are explained in the user interface section.

3. User interface

- main.dart
- NewTransaction (new_transaction.dart)
- TransactionList (transaction_list.dart)
- ChartBar (chart_bar.dart)
- Chart (chart.dart)

main.dart

- It has the classic Flutter program structure.

```
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  Widget build(BuildContext context) {
    return MaterialApp(home: MyHomePage(),);
  }
}
class MyHomePage extends StatefulWidget {
  _MyHomePageState createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> { ... }
```

_MyHomePageState widget structure

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Personal Expenses', ),
      actions: <Widget>[IconButton(... ),],
    ),
    body: SingleChildScrollView(
      child: Column(
        children: <Widget>[
          Chart(_recentTransactions),
          TransactionList(_userTransactions, _deleteTransaction),
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      child: Icon(Icons.add),
      onPressed: () => _startAddNewTransaction(context),
    ),
  );
}
```

Scaffold: appBar

- It has a string and an IconButton (+) mark on the right.
- `_startAddNewTransaction` is called with the button.

```
title: Text('Personal Expenses',),
actions: <Widget>[
  IconButton(
    icon: Icon(Icons.add),
    onPressed: () => _startAddNewTransaction(context),
  ),
],
```

Scaffold: body

- The Column widget has the `Chart` and the `TransactionList` widget.

```
SingleChildScrollView(
  child: Column(
    children: <Widget>[
      Chart(_recentTransactions),
      TransactionList(_userTransactions, _deleteTransaction),
    ],
  ),
),
```

Scaffold: FloatingActionButton

- We can also start the transaction with the `_startAddNewTransaction` here.

```
floatingActionButton: FloatingActionButton(
  child: Icon(Icons.add),
  onPressed: () => _startAddNewTransaction(context),
),
```

- From the `_MyHomePageState` widget, we need to understand these methods or classes.
    - _startAddNewTransaction
    - Chart
    - TransactionList

APIs (service functions)

- The _recentTransactions method returns a list of Transactions for the last 7 days.

```
List<Transaction> get _recentTransactions {
  return _userTransactions.where((tx) {
    return tx.date.isAfter(
      DateTime.now().subtract(
        Duration(days: 7),
      ),
    );
  }).toList();
}
```

- The _addNewTransaction method creates a Transaction and redraws widgets.

```
void _addNewTransaction(
    String txTitle, double txAmount, DateTime chosenDate) {
  final newTx = Transaction(
    title: txTitle, amount: txAmount,
    date: chosenDate, id: DateTime.now().toString(),
  );

  setState(() {
    _userTransactions.add(newTx);
  });
}
```

- The _startAddNewTransaction method displays a showModalBottomSheet to get inputs through the NewTransaction widget.

```
void _startAddNewTransaction(BuildContext ctx) {
  showModalBottomSheet(
    context: ctx,
    builder: (_) {
      return NewTransaction(_addNewTransaction);
    },
  );
}
```

- The _deleteTransaction method deletes the transaction and redraws widgets.

```
void _deleteTransaction(String id) {
  setState(() {
    _userTransactions.removeWhere((tx) => tx.id == id);
  });
}
```

NewTransaction (new_transaction.dart)

- The `NewTransaction` is a stateful widget.
- The first argument is a function to be called.

```
class NewTransaction extends StatefulWidget {
  final Function addTx;
  NewTransaction(this.addTx);
  @override
  _NewTransactionState createState() => _NewTransactionState();
}
```

_NewTransactionState

- Its state widget is `_NewTransactionState` that has all the GUI variables, controller, and `update` method.

```
class _NewTransactionState extends State<NewTransaction> {
  final _titleController = TextEditingController();
  final _amountController = TextEditingController();
  DateTime _selectedDate = DateTime.now();
  ...
```

widget structure

```
TextField(...),
TextField(...),
Container(
  child: Row(
    children: <Widget>[
      Expanded(child: Text(...),
      TextButton(
        style: TextButton.styleFrom(...)
        child: Text(...),
        onPressed: _presentDatePicker,
      ),
    ],
  ),
),
ElevatedButton(
  child: Text('Add Transaction'),
  onPressed: _submitData,
),
```

getNow

- This function returns the current date in a formatted string.

```
String getNow() {
  final DateTime now = DateTime.now();
  final DateFormat formatter = DateFormat("yyyy-MM-dd");
  String formatted = formatter.format(now);
  return formatted;
}
```

_submitData

- This function uses the _amountController to get the users' input, parse it into a number, and return the value using the given function.
- It uses the technique to get a function as an argument (widget.addTx) and returns to the caller using the pop method.

- We know that a stateful widget and state are a pair.
- The addTx is a member of the state widget.
- To access the state widget member from the state, we should prepend `widget` to it.

```
final Function addTx; // at the state widget
widget.addTx(...) // at the state
```

```
void _submitData() {
  if (_amountController.text.isEmpty) {
    return;
  }
  final enteredTitle = _titleController.text;
  final enteredAmount = double.parse(_amountController.text);

  if (enteredTitle.isEmpty || enteredAmount <= 0) { return; }
  widget.addTx(enteredTitle, enteredAmount, _selectedDate,
  );
  Navigator.of(context).pop();
}
```

_presentDatePicker

- This function uses the showDatePicker so users can choose the date.

```
void _presentDatePicker() {
  showDatePicker(
    context: context,
    ...
  ).then((pickedDate) {
    if (pickedDate == null) {return;}
    setState(() {_selectedDate = pickedDate;});
  });
}
```

TransactionList (transaction_list.dart)

- Using the _addNewTransaction function, users can add Transactions to the list.
- This TransactionList displays the transaction information.

- It has the transaction list and a function to delete a Transaction.

```
class TransactionList extends StatelessWidget {
  final List<Transaction> transactions;
  final Function deleteTx;

  TransactionList(this.transactions, this.deleteTx);
```

Widget Structure

- It uses ListView.builder to display Card widgets.
- This technique displays a series of information on a widget.

```
child:ListView.builder(
  itemBuilder: (ctx, index) {
    return Card(
      child: ListTile(
        leading: CircleAvatar(...)
        ...
    );
  },
  itemCount: transactions.length,
),
```

- When there is no transaction, it shows a text and an image.

```
Column(
  children: <Widget>[
    Text(
      'No transactions added yet!',
      style: Theme.of(context).textTheme.titleMedium,
    ),
    Container(
        height: 200,
        child: Image.asset(
          'assets/images/waiting.png',
          fit: BoxFit.cover,
        )),
  ],
)
```

IconButton to delete a Transaction

- The last item on the card is the icon button.
- When the button is pressed, the transaction at the index is deleted.

```
trailing: IconButton(
  icon: Icon(Icons.delete),
  onPressed: () => deleteTx(transactions[index].id),
),
```

ChartBar(chart_bar.dart)

- ChartBar is a widget that displays the amount of money spent on a day.
- In this example, the user spent $600 on Tuesday.

Widget Structure

- It is a column with a text, FractionallySizedBox, and a label.

```
return Column(
  children: <Widget>[
    Container(child: FittedBox(...)),
    Container(
      child: Stack(
        children: <Widget>[FractionallySizedBox(...),],
      ),
    ),
    Text(label),
  ],
...
```

FractionallySizedBox

- FractionallySizedBox is a widget that sizes itself based on a fraction of its parent's size.
- It can align elements with relative sizing.

```
FractionallySizedBox(
  widthFactor: 0.5, // 50% of the parent's width
  heightFactor: 0.3, // 30% of the parent's height
  alignment: Alignment.center,
  child: YourWidget(),
)
```

- In this example, it uses spendingPctOfTotal to display the spend information.

```
ChartBar(this.label, this.spendingAmount, this.spendingPctOfTotal);
...
FractionallySizedBox(
  heightFactor: spendingPctOfTotal, // <--
  child: Container(
    decoration: BoxDecoration(
      color: Theme.of(context).primaryColor,
      borderRadius: BorderRadius.circular(10),
    ),
  ),
),
```

Chart (chart.dart)

- The Chart displays multiple ChartBar widgets.
- In this example, it shows ChartBar for one week.

Widget Structure

- The ChartBar widget displays ChartBar widgets in the Row using the Card widget.

```
return Card(
  child: Row(
    ...
        child: ChartBar(
          (data['day'] as String),
          (data['amount'] as double),
          (data['amount'] as double) / totalSpending,
        ),
    ...
  ),
);
```

- The ChartBar widget requires (1) label, (2) spendingAmount, and (3) spendingPctOfTotal.
- The information is stored in the `groupedTransactionValues`.

```
groupedTransactionValues.map((data) {
  return Flexible(
    child: ChartBar(
      (data['day'] as String),
      (data['amount'] as double),
      (data['amount'] as double) / totalSpending,
    ),
  );
}).toList(),
```

groupedTransactionValues

- groupedTransactionValues is a property: a function that can be used as if it were a variable.
- It returns a map (dictionary) of the date string and the total sum.

```
List<Map<String, Object>> get groupedTransactionValues {
  return List.generate(7, (index) {
    ...
    return {
      'day': ...
      'amount': totalSum,
    };
  }).reversed.toList();
```

Getting the first character of weekdays

- This function retrieves the first character of the week of the day.
- For example, 'T' is extracted from 'Tuesday'.

```
DateFormat.E().format(weekDay).substring(0, 1)
```

Getting the total amount of money spent on a date.

- We can use DateTime.now().subtract() function to get the date information from now.
- The index is used to specify the date we want to get.

```
final weekDay = DateTime.now().subtract(
  Duration(days: index),
);
```

- Then, it computes the total money (totalSum) spent on the date.

```
var totalSum = 0.0;

for (var i = 0; i < recentTransactions.length; i++) {
  if (recentTransactions[i].date.day == weekDay.day &&
      recentTransactions[i].date.month == weekDay.month &&
      recentTransactions[i].date.year == weekDay.year) {
    totalSum += recentTransactions[i].amount;
  }
}
```

totalSpending

- totalSpending is another property that computes the total amount of money in the list.
- It uses a fold method to get the sum.
- Let's say we have a list l = [1,2,3,4,5].

- fold(0, ...) starts with an **initial value** of 0.
- (acc, curr) => acc + curr is a **function** that adds the current item (curr) to the accumulated total (acc).

```
int sum = l.fold(0, (acc, curr) => acc + curr);
```

- The groupedTransactionValues is a list of maps with keys (day, amount).
- We can use the fold method to sum all the values with the amount key.

```
double get totalSpending {
  return groupedTransactionValues.fold(0.0, (sum, item) {
    return sum + (item['amount'] as double);
  });
}
```

4. Program Structure

- This application uses MVC (Model-View-Controller) architecture.

```
├── main.dart
├── model
│   └── transaction.dart
├── theme
│   └── main_theme.dart
└── view
    ├── chart_bar.dart
    ├── chart.dart
    ├── new_transaction.dart
    └── transaction_list.dart
```

MVC Software architecture

- The Model is in the `model` directory.
- The Views (widgets) are in the `view` directory.
- The Controllers are also in the views directory.

This is OK, but we can do better

- This is OK for simple applications.
- However, as the view and control are intermingled in the same files, this will add unnecessary complexity.
- We need to address this issue.

Self-grading for HW

- You analyze the whole code once (30%).
- You analyze the whole code twice using a different method (60%).
  - Make a summary of widgets that you did not know before (what and how to use them).
- You understand how the code works (80%).
- You can use the programming techniques in this example to make team and individual projects (100%).