Bewertung

Branching

Du hast irgendwann den Branch main verlassen und den Branch "allow-double-input" eröffnet. Den hast Du dann nie mehr verlassen. Das ist prinzipiell schlecht. Ich hab auch erst Stunden versenkt, weil ich deinen main-Branch analysiert habe und nicht den neueren "allow-double-input".

Dateistruktur

Prinzipiell finde ich die Struktur gut. Dein Projekt ist natürlich eines der kleineren und das hilft dabei, das sauber zu gestalten, aber eine saubere Struktur hilft immer, das ganze wartbar zu halten.

Allerdings hast Du in letzter Zeit - nach dem Eröffnen von "allow-double-input" einiges versaut in Sachen Struktur. Beispiele:

- bug_report_form.dart: Wieso ist das unter models?
- my_text_field.dart: Wieso ist das unter models?

Ich hab beides in den presentation-Bereich verschoben.

GUI & Bedienkonzept

Prinzipiell ist die App natürlich schön nutzbar, aber es gibt Punkte, wo ich nicht ganz verstehe, wieso Du sie nicht änderst oder - wenn Du nicht weißt wie - nicht deutlicher kommunizierst, was Du ändern willst:

- Auf der Startseite
    - finde ich die Eingabe des Rechnungsbetrags optisch suboptimal (aber nicht schlimm)
    - finde ich die Wiederholung des Rechnungsbetrags in der nächsten Zeile natürlich Quatsch
    - finde ich die Sterne und den Smiley okay (vielleicht ist der Smiley auch optional)
    - Trinkgeld ist irgendwie zuviel Text
    - Und der zu zahlende Betrag ist viel zu klein.
    - Die Länderauswahl ist zu versteckt.
    - Insgesamt ist das optisch nicht rund.
- Die generelle Unterteilung in Rechnen und Einstellung finde ich okay. Es braucht keine eigene Settings-Seite. Aber warum ist die existierende Settings-Seite dann nicht gelöscht? Jetzt ist sie es.
- Auf der Settings-Seite
    - haben die Ovalen Buttons / Auswahlfelder zu wenig Padding, zumindest nach Rechts.
    - Ist das "Fehler melden" nicht im selben Stil.
    - Ist die Eigentrinkgeld Eingabe nicht im selben Stil. Außerdem sollte es wohl anders heißen. Es geht ja eher darum, die vorgegebenen Sätze anzupassen.
    - Ist der Button mit dem DarkMode ein kompletter Bruch
- Auf der Fehlermeldungsseite
    - Gibt es einen Renderflex-Fehler
    - Ist die Seite viel zu leer.
    - Entweder kommt da Text drumherum, die Fehler können größer oder sonst irgendwas

- Man könnte auch drüber nachdenken, ob das Melden eines Fehlers nicht einfach per Dialog passieren sollte, der sich nur vor den Screen schiebt - und nicht als eigener Screen, weil einfach zu leer.

Also, generell glaube ich einfach, dass hier der Anspruch an Dich selbst zu gering ist. Es ist erkennbar noch nicht perfekt, also warum zufrieden sein? 😃

main.dart

Natürlich zu viel Grün für meinen Geschmack, aber solang das unter dem eigentlichen Code passiert, ist das ok.

Die Zeile `WidgetRef? globalRef;` glaube ich eigentlich nicht, dass Du sie brauchst. Es klingt verlockend, von überall auf ref zugreifen zu können, aber ich glaube eigentlich das geht sowieso. Ich hab sie einfach mal gelöscht und wollte es überall dort fixen, wo es deswegen zu Problemen kam. Hab es getan und - nirgendwo rot, also brauchte man das nirgendwo. 😃

Die globalLanguageLibrary nutzt Du ausschließlich in den Settings. Ich tendiere dazu, das dann auch nur dort lokal zu nutzen. Hab das entsprechend verschoben. Es wäre ein bisschen (unerhebliche) Platzverschwendung, hier global immer zwei Objekte (die theoretisch ja groß werden könnten) im Speicher zu behalten, wenn man sie nur braucht, um zwei Buttons auf dem Settings-Screen zu zeichnen.

Der Einsatz von Theming und DarkMode ist super. Wenig und elegant. Konstrukte wie:

```
final a = ThemeData.light(useMaterial3: true);
```

kann man auch weglassen, wenn die Variable eh nicht verwendet wird. Da weist ja sogar der Linter drauf hin. 😃

appstate_provider.dart

Prinzipiell finde ich, dass es das Deine stärkste Klasse im ganzen Projekt ist. Ich finde den Code, auch wenn sicherlich Hilfe an der einen oder anderen Stelle erfolgte, durchgängig gut lesbar und auch immer klar zielorientiert. Ich würde sogar so weit gehen, dass es der beste Provider aller Teilnehmer ist.

Trotzdem Detailkritik. In der Methode setNet findet sich zB Folgendes:

```
void setNet(int intValue) {
  final quality = state.quality;
  final selectedCountry = state.selectedCountryObject;
  // ignore: unnecessary_null_comparison
  if (selectedCountry == null) {
    resetNet();
    return;
  }
```

Warum hebelst Du den Lint durch das ignore aus? Der Lint sagt Dir selectedCountry kann nicht null sein, also braucht es keinen null-Check. Wenn Du meinst, dass null ein möglicher Wert ist, dann stimmt selectedCountryObject nicht, denn das gibt offensichtlich `Country` zurück und nicht `Country?`. Also gucken wir uns selectedCountryObject an:

```
Country get selectedCountryObject {
  for (final c in countries) {
    if (c.id == selectedCountry) {
      return c;
    }
  }
  return countries.first;
}
```

Wenn kein passendes Land gefunden wird, dann wird das erste genommen. Wäre die Liste der Countries leer, dann würde diese Zeile einen Fehler werfen. Also können wir davon ausgehen, dass der null-Check wirklich sinnlos ist. Diese Lints sind oft sehr sinnvoll und sie zu ignorieren sollte die wohlüberlegte Ausnahme sein!

Alle Datenklassen

Ich hab die Attribute jeweils auf final umgestellt. Dadurch konnte ich die Constructors mit const definieren und das wiederum führt dazu, dass zum Beispiel der Ausgangszustand vom AppStateProvider jetzt eine Konstante ist, was (natürlich nicht messbar) die Performance erhöht. Generell gilt: Unsere Arten von Datenklassen haben eigentlich immer finale Attribute, denn sie ändern sich ja nicht, sondern werden im Zweifelsfall durch was Neues ersetzt. Nur in Ausnahmen sollte ein const Constructor nicht möglich sein.

Die Doku-Kommentare sind in der Regel gut, wenn auch auf Deutsch. Da würde ich in Zukunft zu Trainingszwecken gern Englisch sehen. Aber: Alles ist im Prinzip gut.

In der appstate_provider.dart fiel mir Folgendes ins Auge:

```
void resetNet() {
  state = state.copyWith(
    net: 0,
    gros: 0,
  );
}
```

Ich meinte, mich zu erinnern, dass gros eigentlich ein Getter ist. Stimmt auch, denn in appstate.dart steht:

```
  /// Methode zur Berechnung des Bruttobetrags (Nettobetrag + Trinkgeld)
  int get gros => net + tipp;
```

Ein Getter berechnet sich automatisch. Ein Attribut kann gesetzt werden. Also handelt es sich nicht um ein Attribut, aber wieso akzeptiert state.copyWith dann überhaupt gros?

```
Appstate copyWith({
  List<Country>? countries,
  int? net,
  int? gros,
  Quality? quality,
  String? selectedCountry,
  bool? darkMode,
  List<TippOverride>? overrides,
  Language? selectedLanguage,
  int? ownTippingAmount,
}) =>
    Appstate(
      countries: countries ?? this.countries,
      net: net ?? this.net,
      gros: gros ?? this.gros,
      quality: quality ?? this.quality,
      selectedCountry: selectedCountry ?? this.selectedCountry,
      darkMode: darkMode ?? this.darkMode,
      overrides: overrides ?? this.overrides,
      selectedLanguage: selectedLanguage ?? this.selectedLanguage,
      ownTippingAmount: ownTippingAmount ?? this.ownTippingAmount,
    );
```

gros ist also ein Parameter von copyWith und wird auch durchgereicht an den Constructor von AppState. Wieso das?

```
  /// Konstruktor für die Initialisierung des Zustands
  const Appstate({
    required this.countries,
    required this.net,
    required this.quality,
    required this.selectedCountry,
    required this.darkMode,
    required this.overrides,
    required this.selectedLanguage,
    required this.ownTippingAmount,
    required int gros,
  });
```

Und hier sehen wir dann, dass der Constructor einen int-Wert namens gros akzeptiert, aber nix damit macht. Alle anderen Werte werden direkt in Attribute geschrieben (das geschieht mit dem this). Der gros-Parameter ist aber unsinnig. Wir löschen ihn hier und ganz automatisch haben wir nun überall dort, wo er adressiert wurde, Compilefehler und können ihn entfernen. Und dabei kommen wir dann auch zu folgendem Fehler:

```
/// Methode zum Festlegen des Nettobetrags im App-Zustand
void setNet(int intValue) {
  final quality = state.quality;
  final selectedCountry = state.selectedCountryObject;
```

```
    final percentage = state.getRealTipPercentage(selectedCountry, quality);
    log('percentage: $percentage');
    final tippDouble = intValue * percentage / 100;
    final tipp = tippDouble.toInt();
    state = state.copyWith(
      net: intValue,
      gros: intValue + tipp,
    );
    log('${state.gros}, ${state.net} state  values}');
  }
```

Moniert wird die copyWith, bei der wir eben ein gros übergeben, obwohl das jetzt nicht mehr geht. Und wenn wir uns hier fragen, ob das denn sein kann, dann lernen wir etwas über die Funktionsweise. Denn der tip wird an anderer Stelle ausgerechnet, das gros auch. Die gesamte Berechnung ist an dieser Stelle sinnlos, weil das nicht ist, was angezeigt wird. Folglich kann die Funktion reduziert werden auf:

```
    /// Methode zum Festlegen des Nettobetrags im App-Zustand
    void setNet(int intValue) {
      state = state.copyWith(net: intValue);
    }
```

Also, wir lernen daraus: Wenn wir solchen angezeigten Problemen und Lints wirklich intensiv nachgehen, dann machen wir unseren Code einfacher!

GUI

Die GUI-Dateien, die ich angeguckt haben, waren allesamt sinnvoll und gut strukturiert. Aber wir picken auch hier mal Beispiele raus, was man anders machen kann:

```
class SettingsSection extends ConsumerWidget {
  /// Konstruktor für SettingsSection mit optionalem Schlüssel
  const SettingsSection({
    super.key,
  });

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    const settingsButtonsBorder = OutlineInputBorder(
      borderRadius: BorderRadius.all(
        Radius.circular(
          30,
        ),
      ),
    );
    final emojiLibrary = EmojiParser();
    final appstate = ref.watch(refAppState);
    final translate = appstate.selectedLanguage;
    final appstateProvider = ref.read(
```

```
        refAppState.notifier,
      );
      ...
```

Es ist nur eine Kleinigkeit, aber die emojiLibrary würde nun bei jedem Aufbau des Bildschirms (also bei Animationen durchaus 60mal die Sekunde) neu instanziiert. Wenn Du Dir den Code anguckst, siehst Du, dass da erstmal eine Übersetzungstabelle für Emojis aufgebaut wird. Das ist also durchaus eine aufwändige Operation. Es gibt mehrere Wege, wie wir das umgehen können:

1. Wir könnten den Parser als Attribut der jeweiligen Klasse definieren und dort instanziieren. Das wäre besser, weil es nur einmal pro Widget passiert. Nachteil wäre, dass unser Widget nicht mehr const sein kann. Und das heißt es baut langsamer auf, auch wenn das nicht viel sein mag.
2. Wir könnten den Parser in einer globalen Variable ininitialisieren. Initialisierung dann nur einmal, aber niemand mag globale Variablen.
3. Wir könnten den Parser in einen Riverpod-Provider packen. Damit können wir ihn von überall aufrufen.

Weg 3 wurde beschritten. Auf die Weise wird er nur einmal initialisiert. Und er kann gelesen werden genauso wie der AppState.

```
final refEmojiParser = Provider<EmojiParser>((ref) => EmojiParser());
```

Fazit

In der Summe ist das schon ein sehr gutes Projekt. Es fehlt eindeutig Feinschliff in Sachen Optik und ein bisschen mehr Detailorientierung im Code, aber es war wirklich gut. Hervorheben würde ich den Provider, bei dem fast jede Funktion präzise genau das macht, was sie tun soll - und sonst nichts.