Bewertung

Branching

You left the main branch at some point and started "allow-double-input" branch which you never left since then. That is bad in general. Migrate back to your main branch.

Files and directories

In general I like the structure. Sure, your project is rather small and this helps. But you seem to have lost your way a bit in recent days. Examples:

- bug_report_form.dart: Why under models?
- my_text_field.dart: Why under models?

Moved both of them into presentation.

GUI & UX

The app is quite usable. There's details I do not understand why you do not change them - or why you do not say that you do not know how to change them:

- On the start page
  - I do find the input of the money value optically not nice (it's not super bad, but not beautiful)
  - I do find repeating the same value in the next line totally useless.
  - I like the stars and the smiley.
  - I think the tip value is too much text
  - I think the suggested payment value is too small
  - I think the country chooser is too small and hidden.
  - In general, I think it is not a well-rounded screen.
- I like the differentiation between calculcation and settings. There is no need to have a seprate Setting page. I do not understand why a Setting Page is still part of the project. Delete what you do not use. Well, I did it now.
- In the settings section
  - the oval buttons do not have enough padding, at least to the right.
  - the "bug report" button is in a style not fitting to the other elements.
  - the definition of your own tip settings is not fitting. And the current name is too complicated / not really fitting.
  - the button with the dark mode is a complete affront to your overall app looks.
- On the bug report page
  - there is a render flex error
  - the site is too empty and just a skeleton
  - You have to put some text there, make bigger elements, distribute everything more within the space,
  - You should think about using a dialog instead. This is smaller anyhow.

In general, I think you have to learn to look beyond "does it work" and develop a little bit of a feeling for "does it look good enough".

main.dart

Much too much comments for my taste, but as it mainly happens under the real code, it is kinda okay for now.

I do not think that you need `WidgetRef? globalRef;`. It may sound tempting to be able to access ref at any point in the app, but I think this is always possible anyhow. I deletd this line, wanted to fix it wherever it is used right now. Then I found out it is used nowhere. That's why it obviously wasnÄt needed.

I kinda do not like globalLanguageLibrary. To have a global variable (always in memory) with two potentially big data objects, just to draw a button on a single page, seems overblown. I solved this locally within the settings page.

I do like your theming and dark mode approach. It is not much code and quite elegant.

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

This wanst used. The linter said so. Why didnt you delete the line? 😃

appstate_provider.dart

In general I really liked this file. It's the strongest class in this project and quite exemplary. I find the code - granted that it is partially written by other people who helped you - concise and readable. There is always a clear goal that is pursued in the code. I would even say it is the best provider of all students.

Small details though:

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

Why do you ignore the lint? The lint says that selectedCountry cannot be null, therefore you need no null check. If there is a reason for intValue potentially being null, then selectedCountryObject is not written properly, because it never returns null. Okay, that said, of course I looked into selectedCountryObject:

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

```
    }
    return countries.first;
  }
```

You see: If there is no fitting country, the first country will be returned. If there was an empty list of countries, you would get a range error. That's why we can safely say that there will never be a null value. Please stop ignoring lints and start analyizing them. Ignoring them should be a rare exception.

All data classes

Ich changed all attributes to final. This allowed me to use const constructors for everything. This will boost the performance. In geberal: Our data classses will always be final. They do not change. If there is a change, we create new non-changing objects. Only in rare cases a const sonstructor might not be possible.

Your comments are good. Unfortunately they are on German. For training purposes I prefer English. But: They are quite good.

In appstate_provider.dart i saw this:

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

I immediately remembered that gros was a getter. It calculates the applicable tip and then calculated gros. That's why it should not be settable at all. See?

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

A getter calculates its value automatically. An attribut can be set. Why does state.copyWith accept gros at all?

```
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,
    );
```

So, gros is a parameter of copyWith and is handed over to the constructor. Why?

```
  /// 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,
  });
```

And here we see that the constructor demands a value for gros, but does not do anything with its value. All other values are directly wirtten into attributes. The gros parameter makes no sense. That is why I deleted it and fixed it here and all the places where it was used immeditaly were red.

Another part of the code that was affected by this change:

```
/// 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}');
}
```

The compiler sees the copyWith part and marks gros as an error. And if we just delete it, it marks tip as not needed. If we delete tip, it marks tippDouble as not needed and so on. And really ... if you cannot set gros, why calculate here at all? Thats why I changed it to:

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

You should learn from it to really analyze links and not just ignore them. It will lead to our code having a better structure.

GUI

All GUI classes were quite okay. But let's look at some code anyhow:

```
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,
    );
    ...
```

Do you see that you construct the EmojiParser in every build run? This could potentially be an expensive object in memory. There are several ways to avoid this:

1. We could make emojiParser an attribute of this class. It would mean that is only instantiated once per Widget.
2. We could instantiate emojiParser in a global Variable. It would only happen once but always be in memory, even if our page does not require any emojis.
3. We could provide emojiParser via a Riverpod provider. We could access it from anywhere and it would be created only once per when it is needed.

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

Fazit

All in all the project is fine. There is certainly some details missing and there are some raw edges. But it is really good. I want to stress again that the provider is written very well. It does what it should and nothing else - in every method. I liked it.