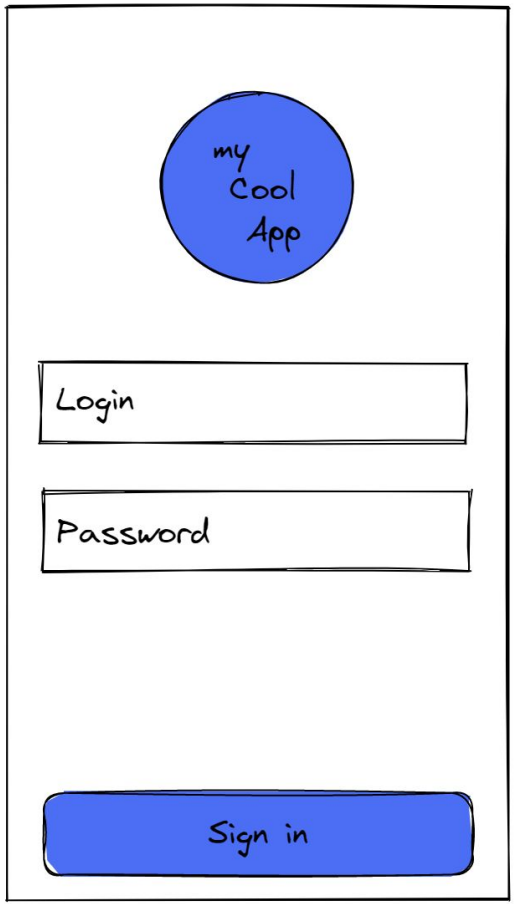
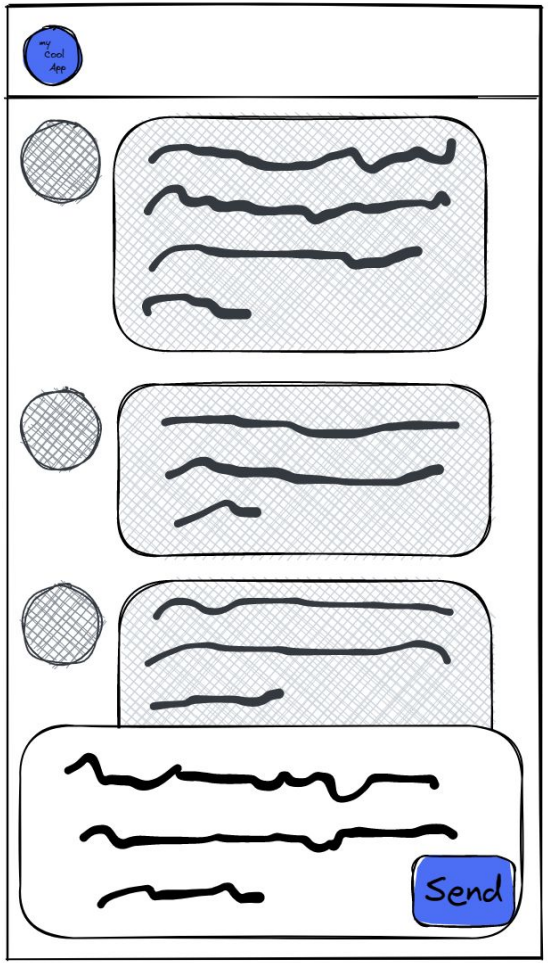
LeanCode

# Forms

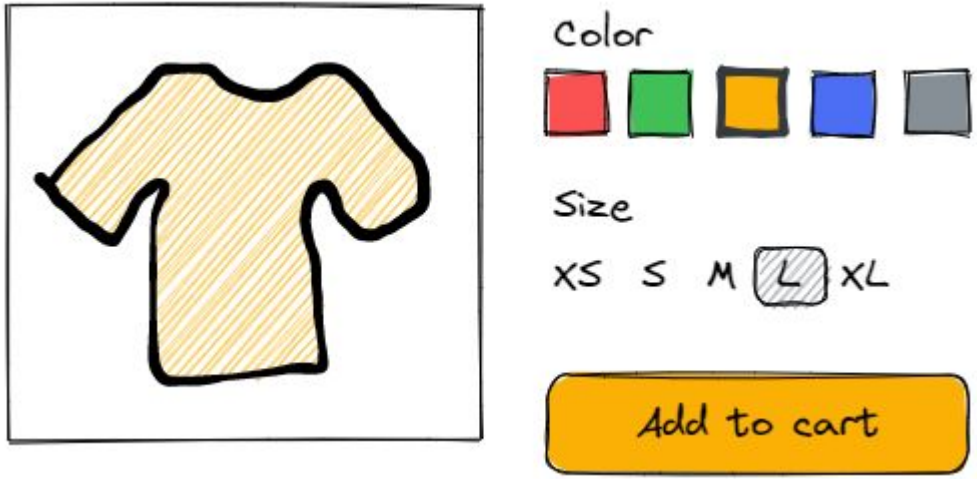
# What is a form




Example: Login screen

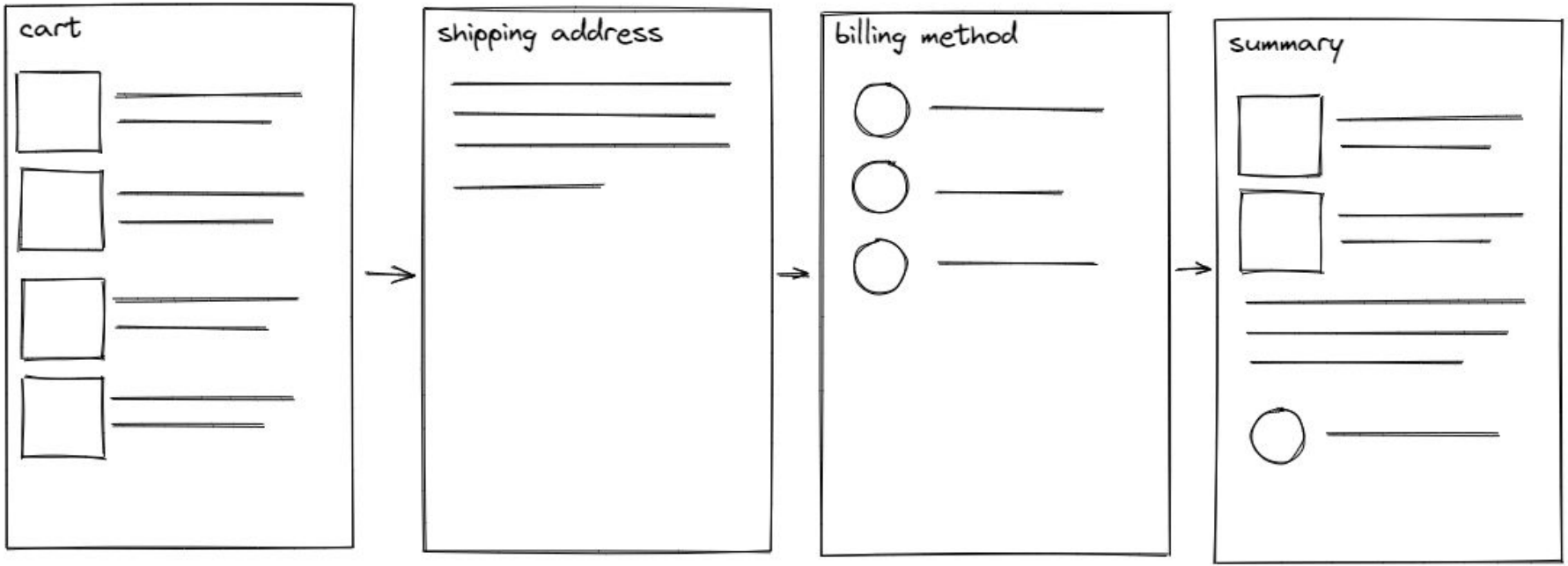



Example: social media posting




Example: adding items to cart

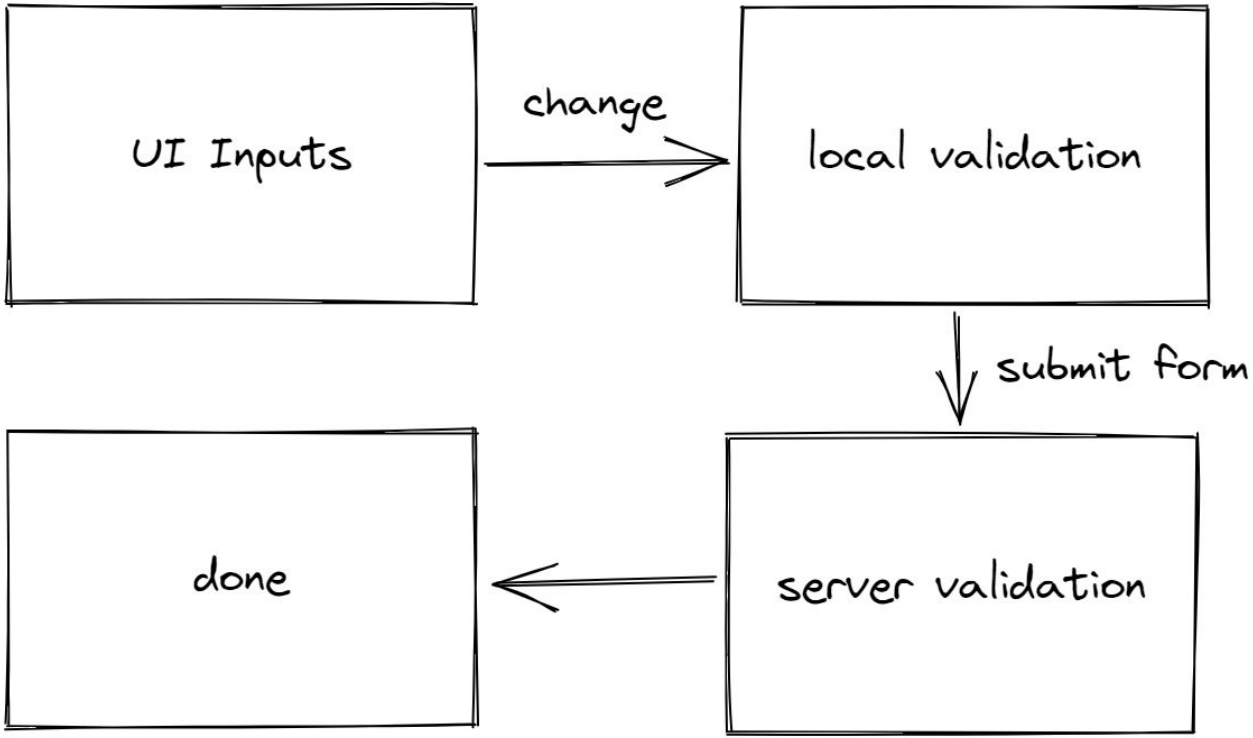
# But wait there’s more



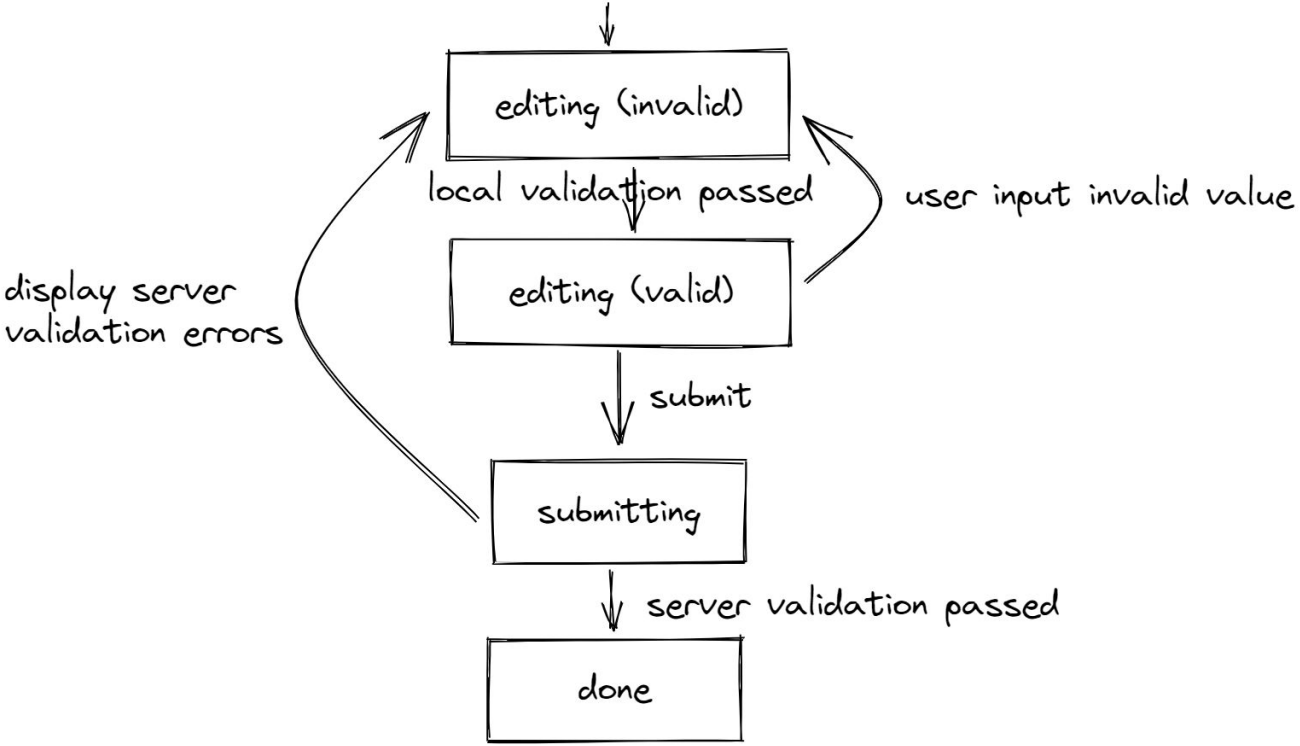

Example: shop checkout

# Breakdown

# Basic form flow



# Basic form flow



# Validation

| Local validation | Server validation |
|---|---|
| instant → better UX | requires a network call |
| costs nothing | uses server resources and bandwidth |
| you might not be the only client/can be bypassed | protects your data |
| is optional | is required |
| is a subset of the other | |

# Before validation: normalize!

- Trim spaces from text fields like email, chat message
  - is a tweet consisting of 300 spaces too long or empty?
  - should you even be able to submit it?

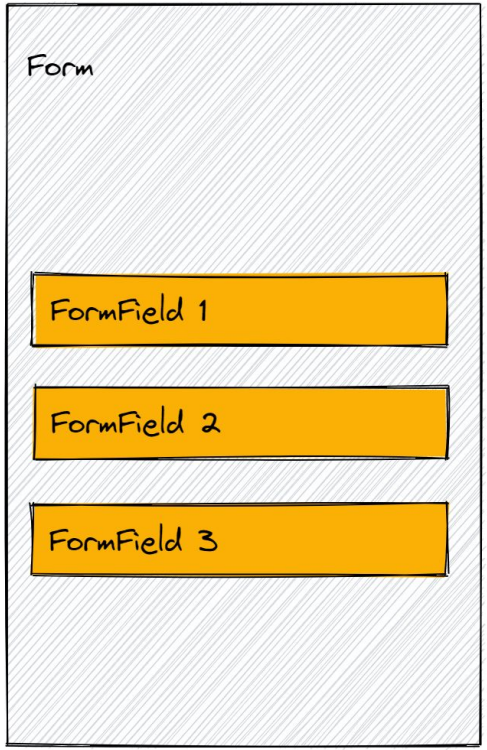
# Forms in Flutter

# Basic widgets

| input type / library | Material | Cupertino |
|---|---|---|
| text | TextField, TextFormField | CupertinoTextField |
| true/false | Checkbox, Switch | CupertinoCheckbox, CupertinoSwitch |
| radio group | Radio | CupertinoSegmentedControl |
| dropdown | DropdownButton | CupertinoPicker |
| number | Slider | CupertinoSlider |
| date/time picker | showDatePicker | CupertinoDatePicker, CupertinoTimePicker |

# Form + FormField

# Form + FormField




```
Form(
  key: formKey,
  child: ListView(
    children: [
      TextFormField(/* ... */),
      const SizedBox(height: 16),
      FormField(/* ... */),
      const SizedBox(height: 16),
      FormField<double>(/* ... */),
      const SizedBox(height: 24),
      ElevatedButton(
        onPressed: () {/*...*/},
        child: const Text('Submit'),
      ),
    ],
  ),
)
```

# FormField

```
FormField<T>(
  initialValue: /* value */,
  autovalidateMode: AutovalidateMode.always /* always, disabled, onUserInteraction */,
  enabled: true /* true, false */,
  validator: (T value) => /* String message or null */,
  onSaved: (T value) { /* ... */ },
  builder: (FormFieldState<T> field) => /* ... */,
),
```

# FormFieldState

Contains

- current value
- errorText from validator
- isValid, hasError

Has methods to

- update value (didChange)
- reset field
- validate field
- save field

```
class CheckboxFormField extends StatelessWidget {
  const CheckboxFormField({
    super.key,
    this.autovalidateMode,
    this.enabled = true,
    required this.label,
  });

  final AutovalidateMode? autovalidateMode;
  final bool enabled;
  final String label;

  @override
  Widget build(BuildContext context) {
    return FormField<bool>(
      autovalidateMode: autovalidateMode,
      enabled: enabled,
      builder: (FormFieldState<bool> field) => Column(
        mainAxisSize: MainAxisSize.min,
        children: [
          Row(
            children: [
              Checkbox(
                value: field.value,
                onChanged: enabled ? field.didChange : null,
              ),
              Expanded(
                child: Text(label),
              ),
            ],
          ),
          if (field.hasError)
            Text(
              field.errorText ?? '',
              style: const TextStyle(color: Colors.red),
            ),
        ],
      ),
    );
  }
}
```

# FormField

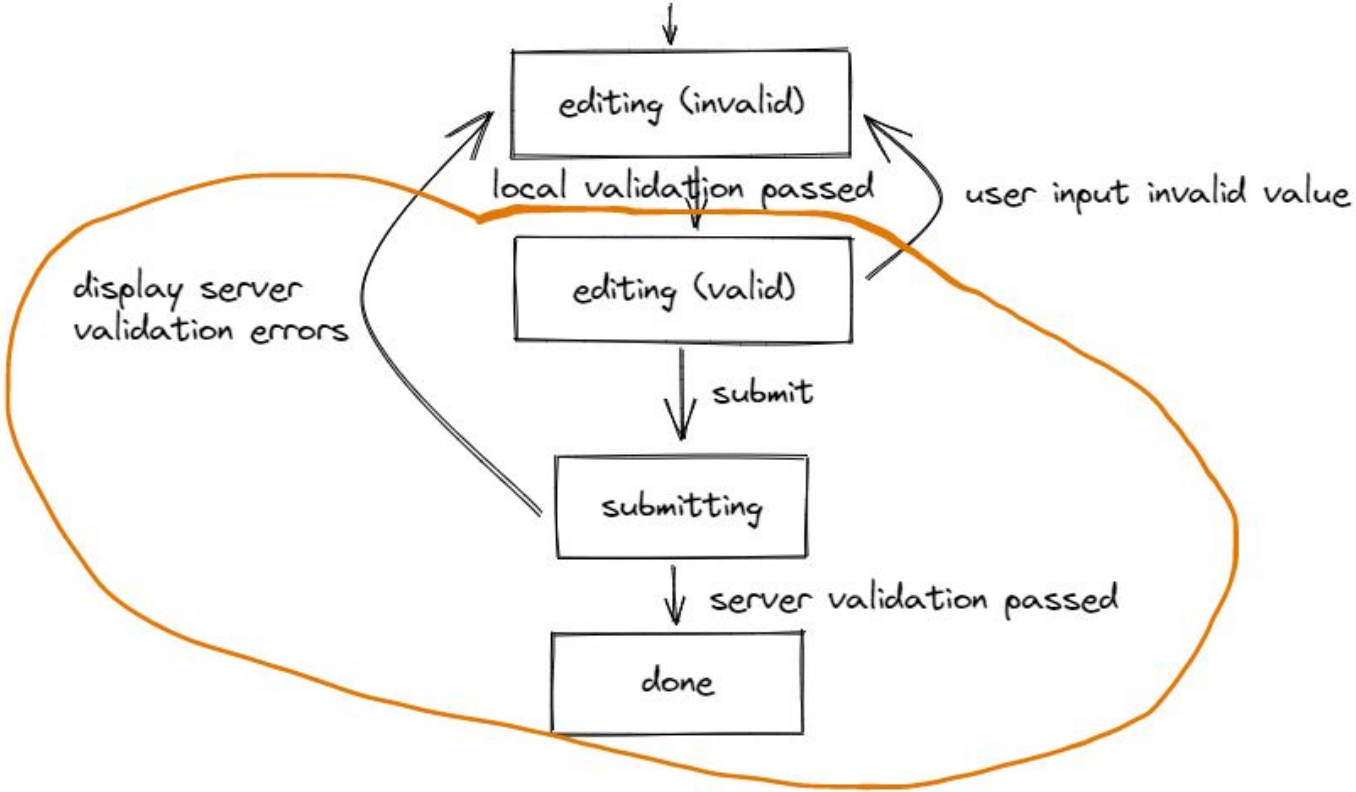
You can extend FormField and FormFieldState to implement custom form field widgets

# Form

- Useful when there are multiple fields
- Allows to validate, reset and save all fields at once with a single method call
- Does **not** provide access to field values
- Field values are best accessed:
  - via GlobalKeys on respective fields,
  - by writing them into a value bag/bundle object in onSave

# Submitting the form

# Basic form flow



# Submitting the form

- Form & FormField have no opinionated server error handling
- You can override FormField's error text by passing errorText to InputDecoration

```
ElevatedButton(
  onPressed: () async {
    if (!(_formKey.currentState?.validate() ?? false)) {
      return;
    }

    final text = _textKey.currentState?.value;
    final checkbox = _checkboxKey.currentState?.value;
    final slider = _sliderKey.currentState?.value;

    final result = await callApiOrSomething(
      text: text,
      checkbox: checkbox,
      slider: slider,
    );

    if (result.wasSuccessful) {
      // Navigate to a success page or show a toast
    }
    else {
      setState(() {
        // Store server validation errors to display
        _serverValidationErrors = result.validationErrors;
      });
    }
  },
  child: const Text('Submit'),
)
```

# Dependent fields

# Dependent fields

Example 1: Validate a field using another field’s value

```
TextFormField(
  key: _textKey,
  decoration: const InputDecoration(
    label: Text('Text field'),
  ),
  validator: (value) {
    // _checkboxKey is a GlobalKey<FormFieldState<bool>>
    // passed to another FormField rendering a Checkbox
    final checkbox = _checkboxKey.currentState?.value ?? false;
    if (checkbox) {
      return null;
    } else {
      return value!.isEmpty ? 'This field is required' : null;
    }
  },
)
```

# Dependent fields

Example 2: Display some fields conditionally based on whether a checkbox is checked or not

# Async validation

# Async validation

Example: currency conversion in a bank money transfer form
There’s 2000 PLN in your bank account and you want to make a 500 € transfer. Do you have enough funds?

# Async validation

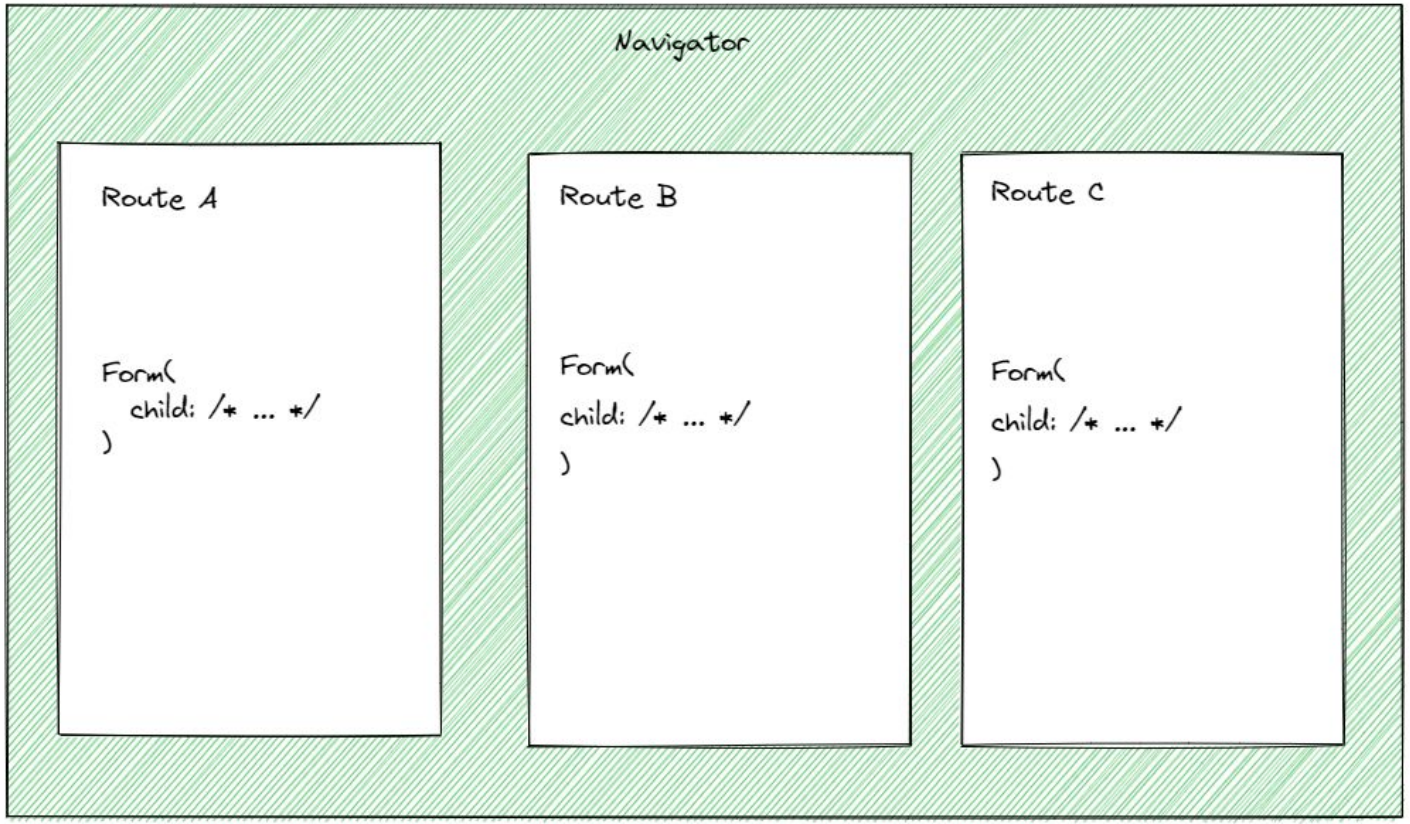
- It’s good to materialize the asynchronous call – create a data structure that describes the process: whether it’s loading, what result it returned
- An example of such structure: AsyncSnapshot materializes the concepts of a stream so that its state can be known at any point in time

# Multi-step forms

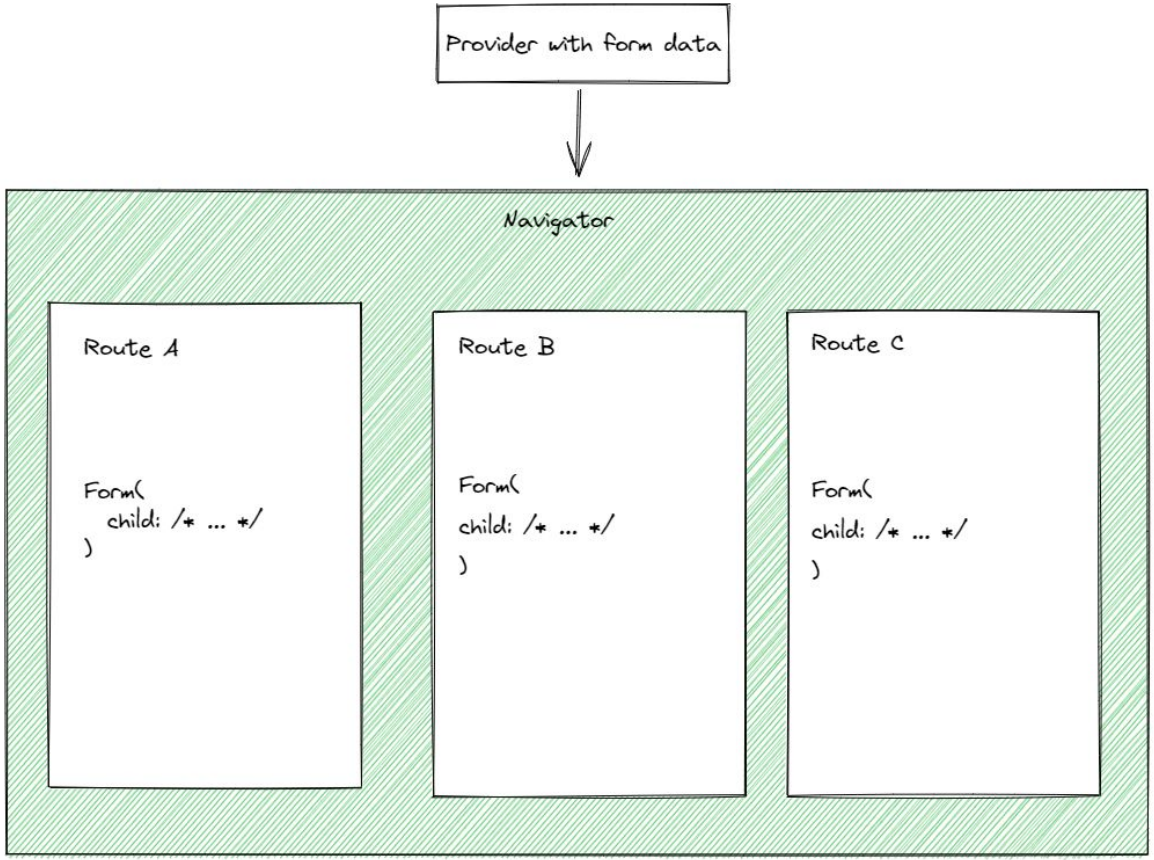
# Multi-step forms

One possible approach:

- each step is a separate navigator route
- each route contains its own Form
- data can be passed between steps either by:
  - keeping it in an InheritedWidget/Provider/Riverpod/etc. above their closest common Navigator
  - accepting previous step’s data as a parameter to route creation; this is only possible when using imperative navigation




each step is a separate navigator route and contains its own Form




Option A: Data provided from above navigator

```dart
class Step1Data {}

class Step2Data {}

class Step3Data {}

class MyFormData extends ChangeNotifier {
  Step1Data? _step1Data;
  Step1Data? get step1Data => _step1Data;
  set step1Data(Step1Data? value) {
    if (value != _step1Data) {
      _step1Data = value;
      notifyListeners();
    }
  }

  Step2Data? _step2Data;
  Step2Data? get step2Data => _step2Data;
  set step2Data(Step2Data? value) {
    if (value != _step2Data) {
      _step2Data = value;
      notifyListeners();
    }
  }

  Step3Data? _step3Data;
  Step3Data? get step3Data => _step3Data;
  set step3Data(Step3Data? value) {
    if (value != _step3Data) {
      _step3Data = value;
      notifyListeners();
    }
  }
}
```

```dart
ChangeNotifierProvider<MyFormData>(
  create: (context) => MyFormData(),
)
```

```dart
final formData = context.watch<MyFormData>();
```

```
class FormStep1Route extends MaterialPageRoute {
  /* ... */
}

class FormStep1Data {}

class FormStep2Route extends MaterialPageRoute {
  FormStep2Route({required FormStep1Data step1Data})
      : super(
          builder: (context) => Step2(step1Data: step1Data),
        );
}

class FormStep2Data {}

class FormStep3Route extends MaterialPageRoute {
  FormStep3Route({
    required FormStep1Data step1Data,
    required FormStep2Data step2Data,
  }) : super(
          builder: (context) => Step3(
            step1Data: step1Data,
            step2Data: step2Data,
          ),
        );
}
```

Option B: data passed directly to routes

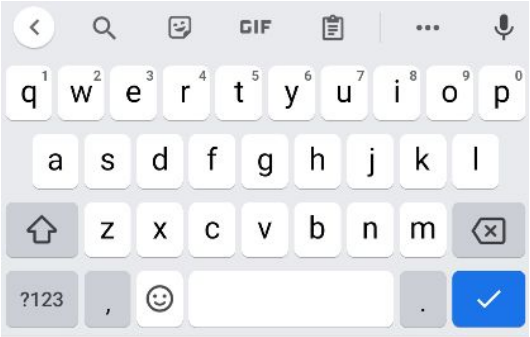
# Extras

# File upload

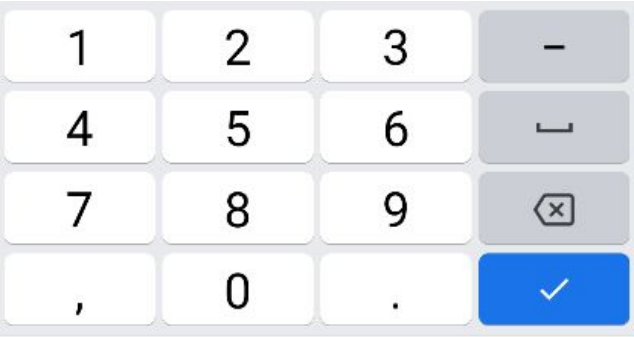
- Flutter has no built-in file pickers
- use libraries like file_picker or image_picker or file_selector
- media files might need to be compressed or otherwise preprocessed – in the mobile app or via a proxy (or both)
- depending on the app you might have to upload the file to a storage server separate from other form data

# Neat TextField properties - keyboardType

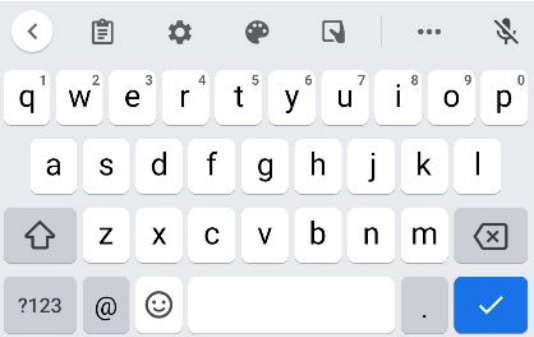
- allows to optimize the keyboard for the field type
- does not prevent pasting! – use input formatters




Default – text

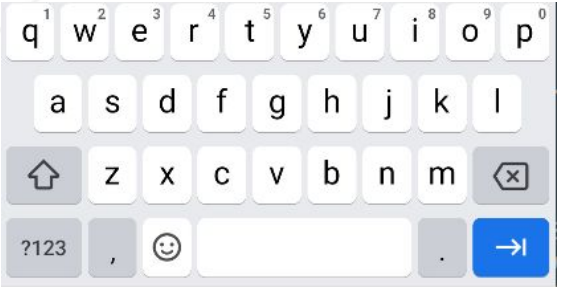



number




email

# Neat TextField properties - textInputAction




next


newline


search


done

# Packages that might prove useful

- leancode_forms - our solution for complex forms
- reactive_forms - model-driven approach to forms
- freezed – create data types like unions/ADTs using code generation