

Building Forms with Flutter

 codingwithjoe.com/building-forms-with-flutter/

March 12, 2018



Over the years many of the apps I have built dealt with data - searching for data, listing data, submitting new data, etc. In all of those apps there was one thing they all shared - a requirement to allow a user to submit data to a back-end via some sort of form.

Goal of this Tutorial

We will build a form in Flutter, show how to perform validation, and finally how to send that data over HTTP to a service.

Creating the App

If you are not yet familiar with the basics of creating a flutter app, please see my first post in this series on flutter – [From Zero to App with Flutter](#).

Note – for this post we'll be using VS Code although other IDEs may be used. Open up VS Code and from the command palette (CTRL+SHIFT+P) choose 'Flutter:New Project'. Enter a new project name and hit Enter. VS Code will setup your new project, which we will be heavily modifying next...

As you might expect, there is a Form widget within the Flutter framework and we will put together a form with several fields inside of it. This is what the form will look like when we are finished:

As you can see, this is a simple contact form. Here is the code for your main.dart file that would produce what you see in our example.

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

void main() => runApp(new MyApp());

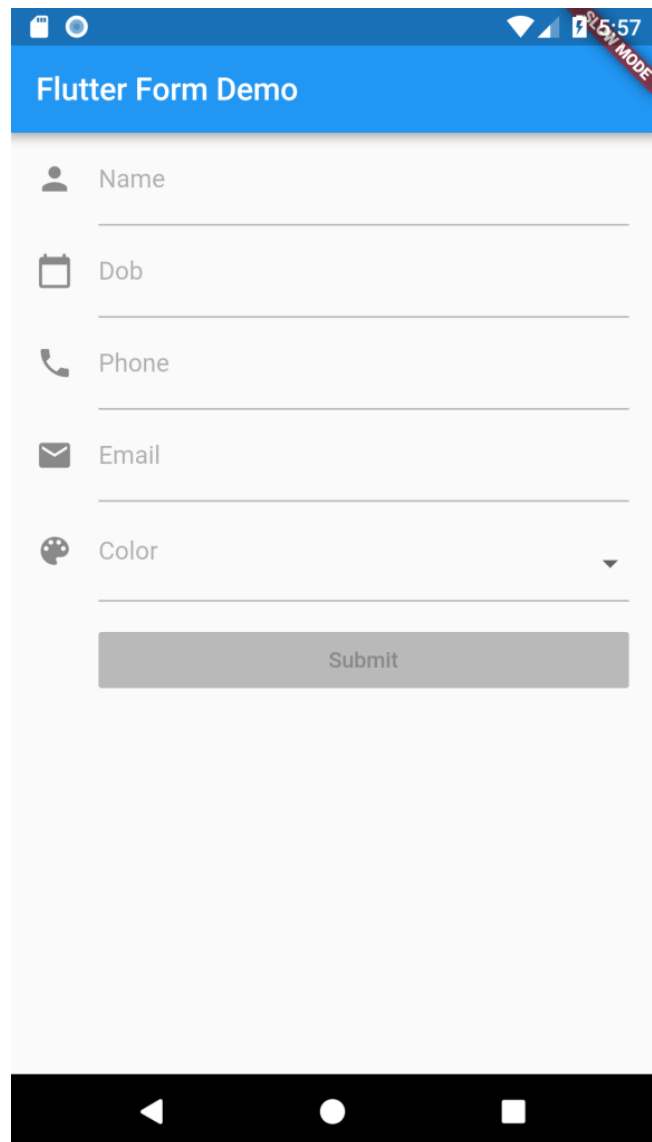
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Form Demo',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new MyHomePage(title: 'Flutter
Form Demo'),
    );
  }
}

class MyHomePage extends StatefulWidget
{
  MyHomePage({Key key, this.title}) :
super(key: key);
  final String title;

  @override
  _MyHomePageState createState() =>
new _MyHomePageState();
}

class _MyHomePageState extends
State<MyHomePage> {
  final GlobalKey<FormState> _formKey =
new GlobalKey<FormState>();
  List<String> _colors = <String>['', 'red',
'green', 'blue', 'orange'];
  String _color = '';

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new SafeArea(
        top: false,
        bottom: false,
        child: new Form(
          key: _formKey,
```

A screenshot of a mobile application titled "Flutter Form Demo". The app has a blue header bar with the title. Below the header, there is a form with five input fields, each preceded by an icon: a person icon for "Name", a calendar icon for "Dob", a phone icon for "Phone", an envelope icon for "Email", and a color palette icon for "Color". Each field has a horizontal line for text entry. Below the "Color" field, there is a grey button labeled "Submit". The app is running on an Android device, as indicated by the navigation bar at the bottom and the status bar at the top showing the time as 5:57 and battery level at 57%.

```

        autovalidate: true,
        child: new ListView(
          padding: const
EdgeInsets.symmetric(horizontal: 16.0),
          children: <Widget>[
            new TextFormField(
              decoration: const
InputDecoration(
              icon: const
Icon(Icons.person),
              hintText: 'Enter your first and
last name',
              labelText: 'Name',
            ),
            ),
            new TextFormField(
              decoration: const
InputDecoration(
              icon: const
Icon(Icons.calendar_today),
              hintText: 'Enter your date of
birth',
              labelText: 'Dob',
            ),
            ),
            keyboardType:
TextInputType.datetime,
            ),
            new TextFormField(
              decoration: const
InputDecoration(
              icon: const Icon(Icons.phone),
              hintText: 'Enter a phone
number',
              labelText: 'Phone',
            ),
            ),
            keyboardType:
TextInputType.phone,
            inputFormatters: [

WhitelistingTextInputFormatter.digitsOnly,
            ],
            ),
            new TextFormField(
              decoration: const
InputDecoration(
              icon: const Icon(Icons.email),
              hintText: 'Enter a email
address',
              labelText: 'Email',
            ),
            ),
            keyboardType:
TextInputType.emailAddress,
            ),
            new FormField(

```

```

        builder: (FormFieldState state)
    {
        return InputDecorator(
            decoration: InputDecoration(
                icon: const
Icon(Icons.color_lens),
                labelText: 'Color',
            ),
            isEmpty: _color == "",
            child: new
DropdownButtonHideUnderline(
                child: new
DropdownButton(
                    value: _color,
                    isDense: true,
                    onChanged: (String
newValue) {
                        setState(() {

newContact.favoriteColor = newValue;
                            _color = newValue;

state.didChange(newValue);
                                });
                            },
                                items: _colors.map((String
value) {
                                    return new
DropdownMenuItem(
                                        value: value,
                                        child: new Text(value),
                                    );
                                }).toList(),
                            ),
                        ),
                    );
                },
            ),
            new Container(
                padding: const
EdgeInsets.only(left: 40.0, top: 20.0),
                child: new RaisedButton(
                    child: const Text('Submit'),
                    onPressed: null,
                )),
        ],
    )),
    );
}
}

```

This is not yet a functional form, but lets look at what we have so far. Flutter can seem a bit verbose at first but in reality it's no more complex or verbose than other approaches - its just different. If you dissect it piece by piece you'll soon get a clear picture just how

well thought out and organized the framework is. You just have to dig in. Let's do just that and look at the above build method from the beginning.

Body

This is where your main content will go. You'll need a Widget to go in here. Popular choices are Padding, Column, Row, SafeArea, ListView, etc. - basically whatever you see as needing to be your top level Widget that will contain one or more child Widgets. I chose SafeArea for this example because it advertises itself as being able to add enough padding to prevent intrusions into its space from the operating system. I don't know about you but I can't stand it when something intrudes on my form space!

Form

In our example, Form is the child of SafeArea.

Notice the autovalidate property being set to true? We do this so that the framework can invoke validation as data is entered. If this is set to false, validation will not occur until we call validate explicitly. We don't have any validation logic in place just yet but don't worry, we will get to that. Next, move your attention to the child property. Here is where we add a Widget that can contain the form fields we want - we chose ListView but we could have chose another Widget such as Column or Row.

```
body: new SafeArea(  
  top: false,  
  bottom: false,  
  child: new Form(  
    key: _formKey,  
    autovalidate: true,  
    child: new ListView(  
      children: [
```

ListView

ListView is a great choice for having, well a list of items. It's a good choice for us because we have a form we want displayed vertically with one input per 'row'. Moreover ListView is scrollable so it gives us room to grow this form without worrying about running out of space. ListView has a children array property where we can supply our form fields. In the screenshot below I have collapsed all of the children under ListView so you can get a sense of the overall structure.

```

child: new ListView(
  padding: const EdgeInsets.symmetric(horizontal: 16.0),
  children: <Widget>[
    new TextFormField(
    ), // TextFormField
    new TextFormField(
    ), // TextFormField
    new TextFormField(
    ), // TextFormField
    new TextFormField(
    ), // TextFormField
    new InputDecorator(
    ), // InputDecorator
    new Container( // RaisedButton // Container
  ], // <Widget>[]
)), // ListView // Form // SafeArea

```

Inputs & Decorators

Most of our inputs are TextFormField widgets. This is one of the more common widgets you'll be using. What gives it the nice material look is the InputDecoration. This includes the border, labels, icons, and styles. Take a look at the Name field for example. It's InputDecoration gives us a 'person' icon, a nice hint, and the text for a label.

```

new TextFormField(
  decoration: const InputDecoration(
    icon: const Icon(Icons.person),
    hintText: 'Enter your first and last name',
    labelText: 'Name',
  ), // InputDecoration
), // TextFormField

```

We also played around with a few other options in our form fields. Our phone field uses a keyboard type of phone, which makes it easier for a user to enter a phone number. We also set an inputFormatter of type `WhitelistingTextInputFormatter.digitsOnly`. This makes it so the user can only enter digits but has the side effect of not allowing the entering of a formatted phone # such as (444) 444-4444.


```

new TextFormField(
  decoration: const InputDecoration(
    icon: const Icon(Icons.phone),
    hintText: 'Enter a phone number',
    labelText: 'Phone',
  ), // InputDecoration
  keyboardType: TextInputType.phone,
  inputFormatters: [
    WhitelistingTextInputFormatter.digitsOnly,
  ],
), // TextFormField

```

You may wish to explore the various available formatters:

- [BlacklistingTextInputFormatter](#)
- [LengthLimitingTextInputFormatter](#)
- [WhitelistingTextInputFormatter](#)

Making a Dropdown

Dropdowns are put together in dart in a very specific manner. Dropdowns are essentially a [DropDownButton](#) widget that contains a list of items. Items are represented by one or more [DropDownMenuItem](#) widgets. DropDownButton is a generic type meaning it is built as DropDownButton<T> where the generic type T must represent the type of items in your dropdown. In our case we construct as DropDownButton<String> and DropDownMenuItem<String>.

In addition, two other widgets are needed to make this all look the way we want it to - [DropDownButtonHideUnderline](#) and another Input

Decorator. DropdownButtonHideUnderline only becomes necessary to hide the underline that by default accompanies our DropDownButton. If we did not take this approach, we would see the underline from the ListView as well as from the DropDownButton - which of course would look odd. As for the InputDecorator - you have seen what these are used for before - to provide icons, labels, and styles to our widget. But there is something new here - the [isEmpty](#) property. This property is important in our example because it is how we tell the decorator whether or not the input (in this case the DropDownButton is empty or not). Why do we need to do that? Because if we do not, the floating label will always be pushed to the top as if there is a value in the dropdown even when it is in fact empty.

Phew - that was a lot of code. I've heard rumors that the Flutter team will be making improvements in this area but for now we have to do a bit of extra work to put together a dropdown. I suggest that if you find yourself doing this more than once, you consider separating out this code to a widget that you can reuse.

There is one more thing to do before we are ready to have a functional dropdown for our form - we have to wrap the dropdown in a `FormField`. This is necessary if we want it to act like the other form fields and add things to it like validation (more on that later).

Here, take a look at the structure for our dropdown:

```
new FormField<String>(
  builder: (FormFieldState<String> state) {
    return InputDecorator(
      decoration: InputDecoration(
        icon: const Icon(Icons.color_lens),
        labelText: 'Color',
      ),
      isEmpty: _color == "",
      child: new DropdownButtonHideUnderline(
        child: new DropdownButton<String>(
          value: _color,
          isDense: true,
          onChanged: (String newValue) {
            setState(() {
              newContact.favoriteColor = newValue;
              _color = newValue;
              state.didChange(newValue);
            });
          },
          items: _colors.map((String value) {
            return new DropdownMenuItem<String>(
              value: value,
              child: new Text(value),
            );
          }).toList(),
        ),
      ),
    );
  },
);
```

Let's review the above widget tree. We start off by creating a `FormField` of type string. We'll find out later in the section where we add validation as to why this is important. Next, we place inside the `FormField` an `InputDecorator` widget, It is the `InputDecorator` that gives us the `Color` label and the icon - both to the left of the dropdown and aligned nicely with the rest of this form. Next we create the `DropDownButtonHideUnderline` widget (again so that the `DropDownButton` inside doesn't have it's underline) and stuff inside of it our `DropDownButton`. To complete this, and ultimately to provide the

actual list for the dropdown, we have a `DropDownButton.items` property to set. In the code example at the beginning of this post, you likely noticed the `_colors` array the the `_color` variables.

```
List<String> _colors = <String>['', 'red', 'green', 'blue', 'orange'];  
String _color = '';
```

To set the `DropDownButton.items` property, we call `_colors.map` which allows us to transform our colors array into a list of `DropDownMenuItem` Widgets. Now we have a functional and aesthetically pleasing dropdown that also validates the user selected a color!

Making a Date Picker

Coming from a web development background, I have a love/hate relationship with date pickers. Seems like we are always fussing with them to make them work across browsers and integrate them with the latest frameworks. Plus, I am a keyboard guy and these things usually just get in the way - but on a mobile device, they can be useful. So our approach here is going to be a little different than what you might think. We will allow the user to manually input a date as well as click a button to invoke a date picker.

Steps:

Update your `.yaml` file. We will be needing the `DateFormat` class which is in the [internationalization package](#). Here is what your updated dependencies section should now look like:

```
dependencies:  
  flutter:  
    sdk: flutter  
  intl: 0.15.2
```

Next, let's import a few more things in our `main.dart`. Add these at the top:

```
import 'dart:async';  
import 'package:intl/intl.dart';
```

Next, let's add some code - place this snippet directly above your build method:

```

final TextEditingController _controller = new TextEditingController();
Future _chooseDate(BuildContext context, String initialDateString) async {
  var now = new DateTime.now();
  var initialDate = convertToDate(initialDateString) ?? now;
  initialDate = (initialDate.year >= 1900 && initialDate.isBefore(now) ? initialDate : now);

  var result = await showDatePicker(
    context: context,
    initialDate: initialDate,
    firstDate: new DateTime(1900),
    lastDate: new DateTime.now());

  if (result == null) return;

  setState(() {
    _controller.text = new DateFormat.yMd().format(result);
  });
}

DateTime convertToDate(String input) {
  try
  {
    var d = new DateFormat.yMd().parseStrict(input);
    return d;
  } catch (e) {
    return null;
  }
}

```

Let's take a minute to cover the above snippet, starting with our variable declarations. The TextEditingController will allow us to set the text of the TextFormField it is associated with in code (we'll do that association in the next section). This is what is happening within our new _chooseDate method. As for the _chooseDate method, it displays the flutter date picker dialog via the showDatePicker method. So to recap, we show a date picker and use the _controller.text property to capture what was returned from the date picker. And since we want to show it in a specific format, we use the DateFormat class. I chose to display the date in the en_US format MM/dd/yy (e.g. 3/9/18) but if you are in a different locale this code could be easily modified. Formatting the date that comes back from the date picker is important in our scenario because we want to have a display format that does not conflict with what a user would manually enter. Skipping this step would result in the date picker returning 3/9/18 as March 9th, 2018.

Within the _chooseDate method you will also notice some logic that depends on the helper method, convertToDate. This helper method is necessary because our TextFormField stores our date as text and we need to convert to a DateTime in order to satisfy the needs of the date picker. In addition, we are also supplying an initialDate, firstDate and lastDate since these are all required in order to use the date picker. You probably noticed there is some logic to ensure the initialDate falls within the boundaries

of `firstDate` and `lastDate` - fail to do this and you will get an exception from the date picker. If this were production code, I would advise considering some exception handling in the `_chooseDate` method just to be on the safe side.

Next up, we need to modify how we construct the `TextFormField` for the date of birth. Replace the entire previous declaration of this field with the following snippet:

```
new Row(children: <Widget>[
  new Expanded(
    child: new TextFormField(
      decoration: new InputDecoration(
        icon: const Icon(Icons.calendar_today),
        hintText: 'Enter your date of birth',
        labelText: 'Dob',
      ),
      controller: _controller,
      keyboardType: TextInputType.datetime,
    ),
    new IconButton(
      icon: new Icon(Icons.more_horiz),
      tooltip: 'Choose date',
      onPressed: () {
        _chooseDate(context, _controller.text);
      },
    ),
  ],
),
```

Let's go over this new snippet in detail. We start off with a `Row` widget, which contains an `Expanded` widget as well as an `IconButton`. Within the `Expanded` widget we have our `TextFormField` that represents our `Dob` field. Why the `Expanded` widget? This widget tells the framework to ensure its child widget uses all available space left. This works out well for us because we want the new `IconButton` to take some space at the end of this row and the `TextFormField` to use the rest. Now that the structure has been explained, let's talk about two key pieces to this new snippet of code - the `TextFormField.Controller` and the `IconButton.onPressed`.

As mentioned above, we would eventually be associating the `TextEditingController` with a `TextFormField`. Assigning the `TextFormField.controller` property does just that. As for the `IconButton.onPressed`, it is pretty straight forward - when the button is pressed the `onPressed` event fires and we call our new `_chooseDate` method. And that is pretty much all there is to it.

Note - if you want to build a standalone datepicker widget that you can use over and over again, and don't mind the fact that you cannot manually edit the date - then I recommend looking at the flutter [demo on datepicker](#).

Adding Validation

Just about all forms need some sort of validation. Fortunately flutter is up to the task. Let's look at a few different opportunities to explore when it comes to our current form.

DateTime Validation

Firstly, we might not want someone to enter a Dob that is a future date. Secondly, since we allow manual entry of a date, we need to check if that date is properly formatted. We will create a simple validation method for our needs. Add this new method to your main.dart file:

```
bool isValidDob(String dob) {  
  if (dob.isEmpty) return true;  
  var d = convertToDate(dob);  
  return d != null && d.isBefore(new DateTime.now());  
}
```

Next, locate the Dob TextFormField and add this line of code directly below the keyboardType property (this should be the very last line of code):

```
validator: (val) => isValidDob(val) ? null : 'Not a valid date',
```

The above code specifies a validator for our Dob TextFormField so that whenever validation needs to occur, the isValidDob method is called. Essentially how this works is if the Dob is considered valid, null is returned which signifies there are no validation issues. On the other hand if the Dob is not considered valid, an appropriate error message is returned.

String Length Validation

Flutter makes this easy by providing the LengthLimitingTextInputFormatter input formatter. Simply add this code to any TextFormField to limit the length of characters to the passed in value. In our case we will limit the name to 30 characters.

```
inputFormatters: [new LengthLimitingTextInputFormatter(30)],
```

Required Field Validation

There is no way to simply mark a field as required in flutter. But that is not to say meeting this requirement is difficult. You can supply a validator to the TextFormField much like what we did in the Dob validation. Here is one that makes sure the user specifies a value for the name TextFormField:

```
validator: (val) => val.isEmpty ? 'Name is required' : null,
```

Phone Number Validation

For our validation, we'll use a standard 10-digit US phone format (###)###-####. Since we are asking users to enter the parenthesis and hyphens, we need to replace the WhitelistingTextInputFormatter.digitsOnly we used previously with the following custom format pattern:

```
new WhitelistingTextInputFormatter(new RegExp(r'^([\d -]{1,15})$')),
```

What the above pattern does is essentially create a whitelist of characters we want to allow the user to enter. If a character doesn't match this whitelist, it is discarded and not allowed to be entered. This keeps the user from entering anything other than parentheses, hyphens and digits.

In addition to whitelisting characters, we need to make sure we have the proper overall format and tell the user when they have an issue with their entry. Add this code to the phone number TextFormField:

```
validator: (value) => isValidPhoneNumber(value) ? null : 'Phone number must be entered as (###)###-####',
```

As well as this method to handle the phone number validation:

```
bool isValidPhoneNumber(String input) {  
  final RegExp regex = new RegExp(r'^\\(\\d\\d\\d\\)\\d\\d\\d\\-\\d\\d\\d\\d$');  
  return regex.hasMatch(input);  
}
```

With this combination you can limit the user to only entering characters that make up a phone number and tell them when their entry does not meet your requirements.

Note: There are opportunities for improvement here. For example you could write your own formatter that would result in not requiring the user to enter in the parentheses and hyphen parts of you phone numbers. Let's be honest - entering those on a phone is not user friendly! Here is a great example put together by the flutter team that you could potentially use (hint: look for the `_UsNumberTextInputFormatter` class) - [Flutter Demo](#).

Email Address Validation

Validating email addresses is notoriously difficult if you shoot for 100% accuracy. I find this expression used here to be good enough. Besides the ultimate validation for email addresses is sending an email from your system to the email address entered by the user to confirm they aren't entering a valid formatted, yet fictitious email address. Add this validator code to your phone number TextFormField:

```
validator: (value) => isValidEmail(value) ? null : 'Please enter a valid email address',
```

And this code to your main.dart file. The email address supplied by the user will be validated against this regular expression and if it doesn't pass, then the user will see a message informing them to correct their entry.

```
bool isValidEmail(String input) {  
  final RegExp regex = new RegExp(r"^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-  
  ][email protected][a-zA-Z0-9](?:[a-zA-Z0-9-]{0,253}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,253}[a-zA-Z0-9])?)*$");  
  return regex.hasMatch(input);  
}
```

Dropdown validation

Given that we have a `FormField` to work with, adding a validator is now a snap. You need only make two modifications to the code for the dropdown: specify an `errorText` to display if the `FormField` has an error and the addition of the validator itself. Here is the updated code:

```
new FormField<String>(
  builder: (FormFieldState<String> state) {
    return InputDecorator(
      decoration: InputDecoration(
        icon: const Icon(Icons.color_lens),
        labelText: 'Color',
        errorText: state.hasError ? state.errorText : null,
      ),
      isEmpty: _color == "",
      child: new DropdownButtonHideUnderline(
        child: new DropdownButton<String>(
          value: _color,
          isDense: true,
          onChanged: (String newValue) {
            setState(() {
              newContact.favoriteColor = newValue;
              _color = newValue;
              state.didChange(newValue);
            });
          },
          items: _colors.map((String value) {
            return new DropdownMenuItem<String>(
              value: value,
              child: new Text(value),
            );
          }).toList(),
        ),
      ),
    );
  },
  validator: (val) {
    return val != "" ? null : 'Please select a color';
  },
),
```

Big thanks to [Jason Jerome](#) for this idea and the code to make this work!

Validation wrap-up

Whew - we made it through validation! When doing your own validation and expanding on what we have covered here, I advise you to take the following into account.

- Where are your users located? In my examples we assumed continental United States. But that might not work for you and complicates the date and phone number validation we used here. In addition, if you ever allow your users to enter in date + time, then you need to consider how you will handle time zones properly. And finally, don't forget localization/internationalization!
- Malicious intent! Always validate your input within your application but always do it again on your backend. This prevents someone from submitting something to your backend that you might have assumed the frontend validation handled. Always double-check everything and assume that at some point somebody can try and subvert the intended use of your app.
- Test, Test, Test - Always write unit tests against your validation logic! I'd bet that you'll find scenarios that you did not think of and find ways to fix those bugs before you users find them.
- Consider the long term. Don't take the easy route and place a lot of validation logic within the same code where your view resides. Although we did it here, it was mainly to keep the examples easy to follow. Validation logic should be separated out into another framework or library that is highly reusable and highly testable.

Submitting the Form

Finally, we have come to the last section of this tutorial. We have covered how to build a form in flutter, how to validate form fields in flutter, and now we will move on to submitting the form to a backend service. Since we don't have a real backend setup, we'll show you how to fake one quickly for testing purposes. First things first, we need to put together the view model for this form. Typically you would do this first but I purposefully kept it out of previous examples to keep things simple. Create a new file under lib and call it contact.dart. Here is the code for your new file:

```
class Contact {
  String name;
  DateTime dob;
  String phone = "";
  String email = "";
  String favoriteColor = "";
}
```

Now, back in main.dart, import your Contact by adding this line near your other import declarations:

```
import 'contact.dart';
```

Now that we have access to contact.dart, let's instantiate a new instance of contact. At the top of your `_MyHomePageState` class, add this single line of code:

```
Contact newContact = new Contact();
```

Next, we need to update how our form works so that when the Submit button is clicked, we save each form field back to our newContact we just created. This is where the FormField onSaved event comes in handy. This works well for all of our fields except Color - which is not a TextFormField but rather a dropdown. In order to make sure the value selected in the dropdown makes its way back to the Contact.favoriteColor property, we need to add a single line of code to the onChanged event to sync things up. Here is an updated code snippet for the Form we have been working with.

```
child: new Form(  
  key: _formKey,  
  autovalidate: true,  
  child: new ListView(  
    padding: const EdgeInsets.symmetric(horizontal: 16.0),  
    children: <Widget>[  
      new TextFormField(  
        decoration: const InputDecoration(  
          icon: const Icon(Icons.person),  
          hintText: 'Enter your first and last name',  
          labelText: 'Name',  
        ),  
        inputFormatters: [new LengthLimitingTextInputFormatter(30)],  
        validator: (val) => val.isEmpty ? 'Name is required' : null,  
        onSaved: (val) => newContact.name = val,  
      ),  
      new Row(children: <Widget>[  
        new Expanded(  
          child: new TextFormField(  
            decoration: new InputDecoration(  
              icon: const Icon(Icons.calendar_today),  
              hintText: 'Enter your date of birth',  
              labelText: 'Dob',  
            ),  
            controller: _controller,  
            keyboardType: TextInputType.datetime,  
            validator: (val) =>  
              isValidDob(val) ? null : 'Not a valid date',  
            onSaved: (val) => newContact.dob = convertToDate(val),  
          )),  
        new IconButton(  
          icon: new Icon(Icons.more_horiz),  
          tooltip: 'Choose date',  
          onPressed: (() {  
            _chooseDate(context, _controller.text);  
          })),  
      ]),  
      new TextFormField(  
        decoration: const InputDecoration(  
          icon: const Icon(Icons.phone),  
          hintText: 'Enter a phone number',  
          labelText: 'Phone',  
        ),  
        keyboardType: TextInputType.phone,
```

```

inputFormatters: [
    new WhitelistingTextInputFormatter(
        new RegExp(r'^([\d -]{1,15}$')),
    ],
    validator: (value) => isValidPhoneNumber(value)
        ? null
        : 'Phone number must be entered as (###)###-####',
    onSave: (val) => newContact.phone = val,
),
new TextFormField(
    decoration: const InputDecoration(
        icon: const Icon(Icons.email),
        hintText: 'Enter a email address',
        labelText: 'Email',
    ),
    keyboardType: TextInputType.emailAddress,
    validator: (value) => isValidEmail(value)
        ? null
        : 'Please enter a valid email address',
    onSave: (val) => newContact.email = val,
),
new FormField<String>(
    builder: (FormFieldState<String> state) {
        return InputDecorator(
            decoration: InputDecoration(
                icon: const Icon(Icons.color_lens),
                labelText: 'Color',
                errorText: state.hasError ? state.errorText : null,
            ),
            isEmpty: _color == '',
            child: new DropdownButtonHideUnderline(
                child: new DropdownButton<String>(
                    value: _color,
                    isDense: true,
                    onChanged: (String newValue) {
                        setState(() {
                            newContact.favoriteColor = newValue;
                            _color = newValue;
                            state.didChange(newValue);
                        });
                    },
                    items: _colors.map((String value) {
                        return new DropdownMenuItem<String>(
                            value: value,
                            child: new Text(value),
                        );
                    }).toList(),
                ),
            ),
        );
    },
    validator: (val) {
        return val != '' ? null : 'Please select a color';
    },

```

```

    ),
    new Container(
      padding: const EdgeInsets.only(left: 40.0, top: 20.0),
      child: new RaisedButton(
        child: const Text('Submit'),
        onPressed: _submitForm,
      )),
  ],
  )),

```

Now that we have the form all squared away, let's implement the `onPressed` of the `RaisedButton`. This is where we will ensure the form is valid, and submit to our web service. Here is the implementation of our `_submitForm` method:

```

void _submitForm() {
  final FormState form = _formKey.currentState;

  if (!form.validate()) {
    showMessage('Form is not valid! Please review and correct.');
```

```

  } else {
    form.save(); //This invokes each onSave event

    print('Form save called, newContact is now up to date...');
    print('Email: ${newContact.name}');
    print('Dob: ${newContact.dob}');
    print('Phone: ${newContact.phone}');
    print('Email: ${newContact.email}');
    print('Favorite Color: ${newContact.favoriteColor}');
    print('=====');
    print('Submitting to back end...');
    print('TODO - we will write the submission part next...');
  }
}

```

The above code is dependent on three additional changes. The first change is that we need to implement the new `showMessage` method as follows:

```

void showMessage(String message, [MaterialColor color = Colors.red]) {
  _scaffoldKey.currentState
    .showSnackBar(new SnackBar(backgroundColor: color, content: new Text(message)));
}

```

This new method shows a brief message (toast) at the bottom of the screen - referred to as a **SnackBar**. As you can see, there is a new class variable here named `_scaffoldKey` that needs to be implemented. This `_scaffoldKey` is needed in order to show the `SnackBar`. Add this line to the top of your `_MyHomePageState` class.

```

final GlobalKey<ScaffoldState> _scaffoldKey = new GlobalKey<ScaffoldState>();

```

And our third required change is to set the key property of `Scaffold` to our `_scaffoldKey`. Like this:

```
@override
Widget build(BuildContext context) {
  return new Scaffold(
    key: _scaffoldKey,
    appBar: new AppBar(
      title: new Text(widget.title),
    ),
```

Now you have everything you need to validate the form and prepare the newContact to be submitted to a service. Let's take care of this final piece. For starters, find the TODO in our code about 'writing the submission part next' inside of the `_submitForm` and replace that line with the following code:

```
var contactService = new ContactService();
contactService.createContact(newContact)
  .then((value) =>
    showMessage('New contact created for ${value.name}!', Colors.blue)
  );
```

The above code calls a `ContactService` class which we will create next and will have the responsibility of submitting our contact information to a web API. Once the contact comes back from the server, we will show a confirmation message to the user. Let's go ahead and add a new file under `lib` and call it `contact_service.dart`. At this point, also return to the top of `main.dart` and import your new `contact_services.dart`. Here is the code for `contact_services.dart`:

```

import 'package:http/http.dart' as http;
import 'dart:async';
import 'dart:convert';
import 'package:intl/intl.dart';

import 'contact.dart';

class ContactService {
  static const _serviceUrl = 'http://mockbin.org/echo';
  static final _headers = {'Content-Type': 'application/json'};

  Future<Contact> createContact(Contact contact) async {
    try {
      String json = _toJson(contact);
      final response =
        await http.post(_serviceUrl, headers: _headers, body: json);
      var c = _fromJson(response.body);
      return c;
    } catch (e) {
      print('Server Exception!!!');
      print(e);
      return null;
    }
  }

  Contact _fromJson(String json) {
    Map<String, dynamic> map = JSON.decode(json);
    var contact = new Contact();
    contact.name = map['name'];
    contact.dob = new DateFormat.yMd().parseStrict(map['dob']);
    contact.phone = map['phone'];
    contact.email = map['email'];
    contact.favoriteColor = map['favoriteColor'];
    return contact;
  }

  String _toJson(Contact contact) {
    var mapData = new Map();
    mapData["name"] = contact.name;
    mapData["dob"] = new DateFormat.yMd().format(contact.dob);
    mapData["phone"] = contact.phone;
    mapData["email"] = contact.email;
    mapData["favoriteColor"] = contact.favoriteColor;
    String json = JSON.encode(mapData);
    return json;
  }
}

```

Let's review our new ContactService class. Towards the top you can see that we are using mockbin.org to setup a test API endpoint. Calling echo basically returns back to us the same thing we posted - useful for testing our form submission. The only public

method in here, `createContact` accepts a `Contact` instance, serializes it to `Json` and submits it to the `mockbin echo` endpoint. It then awaits a response, converts the response back to a `Contact` instance from `Json` and returns it to the caller.

Summary

In this article we learned how to create a form, validate its fields, and submit it to an API. We've just scratched the surface on what Flutter can offer and in future articles we might spend more time on validation and creating widgets like a full blown date picker. I hope you enjoyed this article and let me know if there are other topics you are interested in.