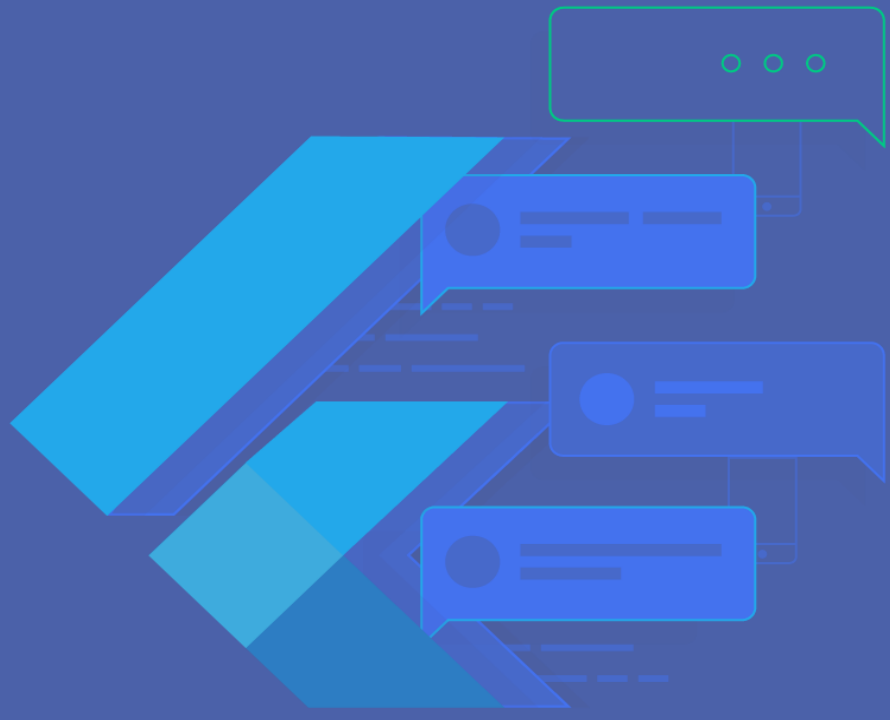
State Management



University American College Skopje

Welcome

- Managing and lifting State up
- Local and App State
- List of State management approaches
- Provider Package
  - ChangeNotifier
  - ChangeNotifierProvider
  - Consumer
- Accessing the State
- Providers & Listeners
- Reading a value
- Working with Multiple Providers
- Exploring alternate Provider Syntaxes
- Shop App Creation

State

- **State** is information that can be read synchronously when the widget is built and might change during the lifetime of the widget.
- **State** objects are created by the framework by calling the StatefulWidget.**createState** method when inflating a StatefulWidget to insert it into the tree.
- It is the responsibility of the widget implementer to ensure that the **State** is promptly notified when such state changes, using State.**setState**.
- Because a given StatefulWidget instance *can be inflated multiple times* (e.g., the widget is incorporated into the tree in multiple places at once), *there might be more than one* **State** object associated with a given StatefulWidget instance.
- Similarly, if a StatefulWidget is removed from the tree and later inserted in to the tree again, the framework will call StatefulWidget.**createState** again to create a fresh **State** object, simplifying the lifecycle of **State** objects.

State Management

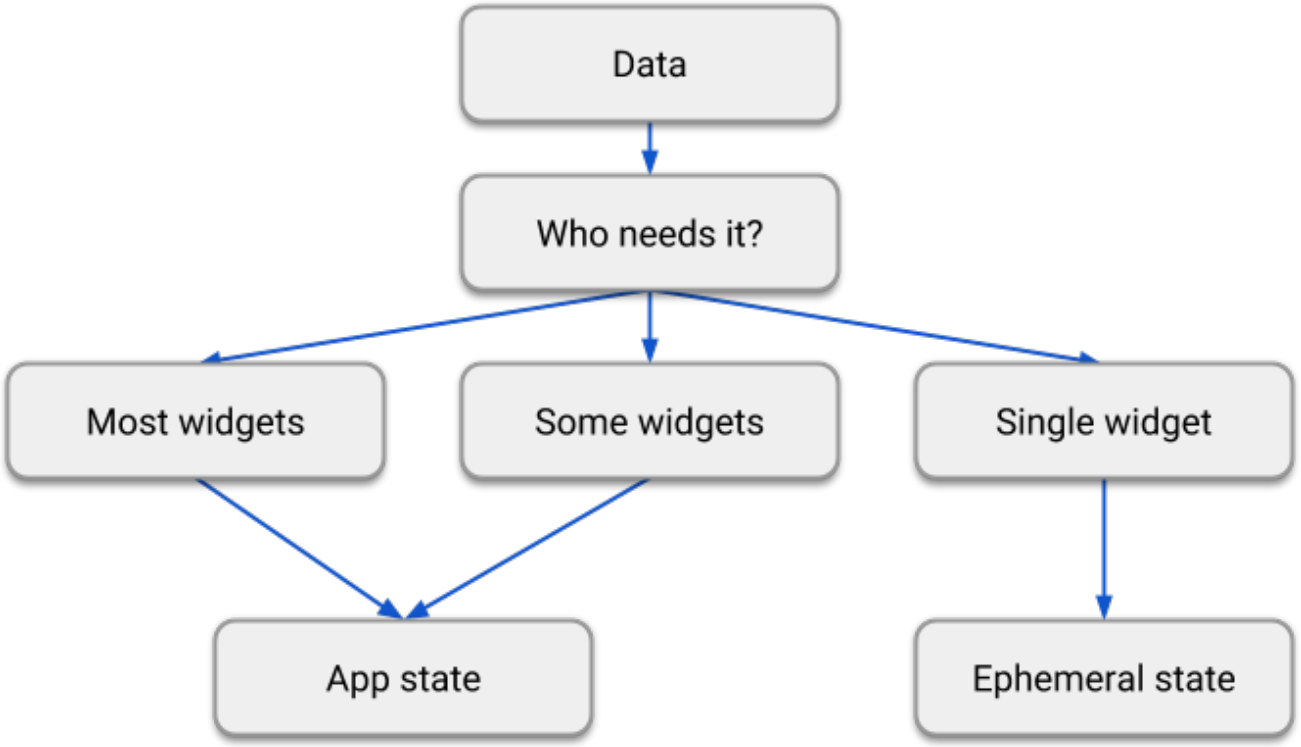
- The **state management** is one of the most *popular and necessary* processes in the lifecycle of an application.
- Overall, **state management** makes the state of an app visible in the form of a data structure.
- State management libraries provide developers with the tools needed to create the data structures and change them when new actions occur.
- According to official documentation, Flutter is declarative. It means Flutter builds its UI by reflecting the current state of your app.

Local and App State

- In the broadest possible sense, the **state** of an app is everything that exists in memory when the app is running.
- This includes the app's assets, all the variables that the Flutter framework keeps about the UI, animation state, textures, fonts, and so on.
- First, you don't even manage some state (like textures). The framework handles those for you. So a more useful definition of state is "whatever data you need in order to rebuild your UI at any moment in time".
- Second, the state that you do manage yourself can be separated into two conceptual types: **ephemeral** state and **app** state.
- **Ephemeral** state can be implemented using State and *setState*(), and is often local to a single widget. The rest is your **app** state.
- Both types have their place in any Flutter app, and the split between the two depends on your own preference and the complexity of the app.

Using Ephemeral and App State

- To be clear, you can use *State* and *setState*() to manage all of the **state** in your app. If you have some simple state that **only affects the behavior of a single widget**, *StatefulWidget* is all you need.
- On the other hand, when state needs to be shared across multiple widgets or even the entire app, you're dealing with **shared/global app state**.
- There is no clear-cut, universal rule to distinguish whether a particular variable is **ephemeral** or **app** state.
- Sometimes, you'll have to refactor one into another. For example, you'll start with some clearly **ephemeral** state, but as your application grows in features, it might need to be moved to **app** state.


Data
Who needs it?
Most widgets
Some widgets
Single widget
App state
Ephemeral state

Ephemeral state

- The state that is local to any widget is known as **ephemeral** state (sometimes also called UI state).
- As the state is contained within a single widget, there is no need for any complex state management techniques – just using a simple *StatefulWidget* to rebuild the UI is enough.
- Other parts of the widget tree seldom need to access this kind of state. There is no need to serialize it, and it doesn't change in complex ways.
- Few examples:

  - current page in a *PageView*

  - current progress of a complex animation

  - current selected tab in a *BottomNavigationBar*

```
class MyHomepage extends StatefulWidget {
  const MyHomepage({Key? key}) : super(key: key);

  @override
  _MyHomepageState createState() => _MyHomepageState();
}

class _MyHomepageState extends State<MyHomepage> {
  int _index = 0;

  @override
  Widget build(BuildContext context) {
    return BottomNavigationBar(
      currentIndex: _index,
      onTap: (newIndex) {
        setState(() {
          _index = newIndex;
        });
      },
      // ... items ...
    );
  }
}
```

App state

- The state that is shared across many parts of your app, and that you want to keep between user sessions, is known as the **app state** (sometimes also called shared state).
- This is where state management solutions help a lot.
- Examples of application state:

  - User preferences

  - Login info

  - Notifications in a social networking app

  - The shopping cart in an e-commerce app

  - Read/unread state of articles in a news app

- For managing app state, you'll want to research your options. Your choice depends on the complexity and nature of your app, your team's previous experience, and many other aspects.

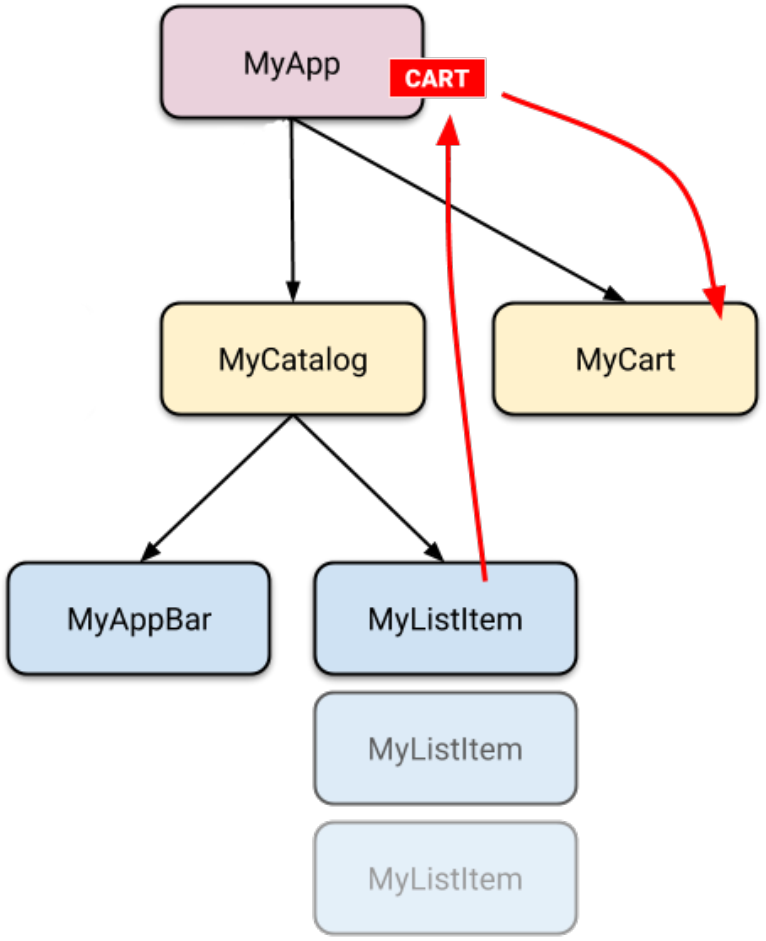
List of state management approaches

- Provider
  A recommended approach.
- InheritedWidget & InheritedModel
  The low-level approach used to communicate between ancestors and children in the widget tree.
  This is what provider and many other approaches use under the hood.
- SetState
  The low-level approach to use for widget-specific, ephemeral state.
- Redux
  A state container approach familiar to many web developers.
- Fish-Redux
  Fish Redux is an assembled flutter application framework based on Redux state management.
  It is suitable for building medium and large applications.
- BloC/Rx
  A family of stream/observable based patterns.
- Flutter Commands
  Reactive state management that uses the Command Pattern and is based on ValueNotifiers.
  Best in combination with GetIt, but can be used with Provider or other locators too.
- GetIt
  A service locator based state management approach that doesn't need a BuildContext.
- MobX
  A popular library based on observables and reactions.
- Riverpod
  Is similar to Provider and is compile-safe and testable. Riverpod doesn't have a dependency on the Flutter SDK.
- GetX
  A simplified reactive state management solution.

Lifting state up

- In Flutter, it makes sense to keep the **state** above the widgets that use it.
- Why? In declarative frameworks like Flutter, if you want to change the UI, you have to rebuild it. In other words, it's hard to imperatively change a widget from outside, by calling a method on it. And even if you could make this work, you would be fighting the framework instead of letting it help you.
- You would need to take into consideration the current state of the UI and apply the new data to it. It's hard to avoid bugs this way.
- In Flutter, you construct a new widget every time its contents change. Because you can only construct new widgets in the build methods of their parents, if you want to change contents, it needs to live in parent element or above.

> *In flutter we have this concept of lifting the state up, which implies, we take that state which is being dependent upon by the set of widgets and place it as far up as the parent widget that enclose all those widgets, such that you can pass down that state to all the widgets that need it(This is similar to the concept of prop drilling for those coming from react).*


MyApp
CART
MyCatalog
MyCart
MyAppBar
MyListItem
MyListItem
MyListItem

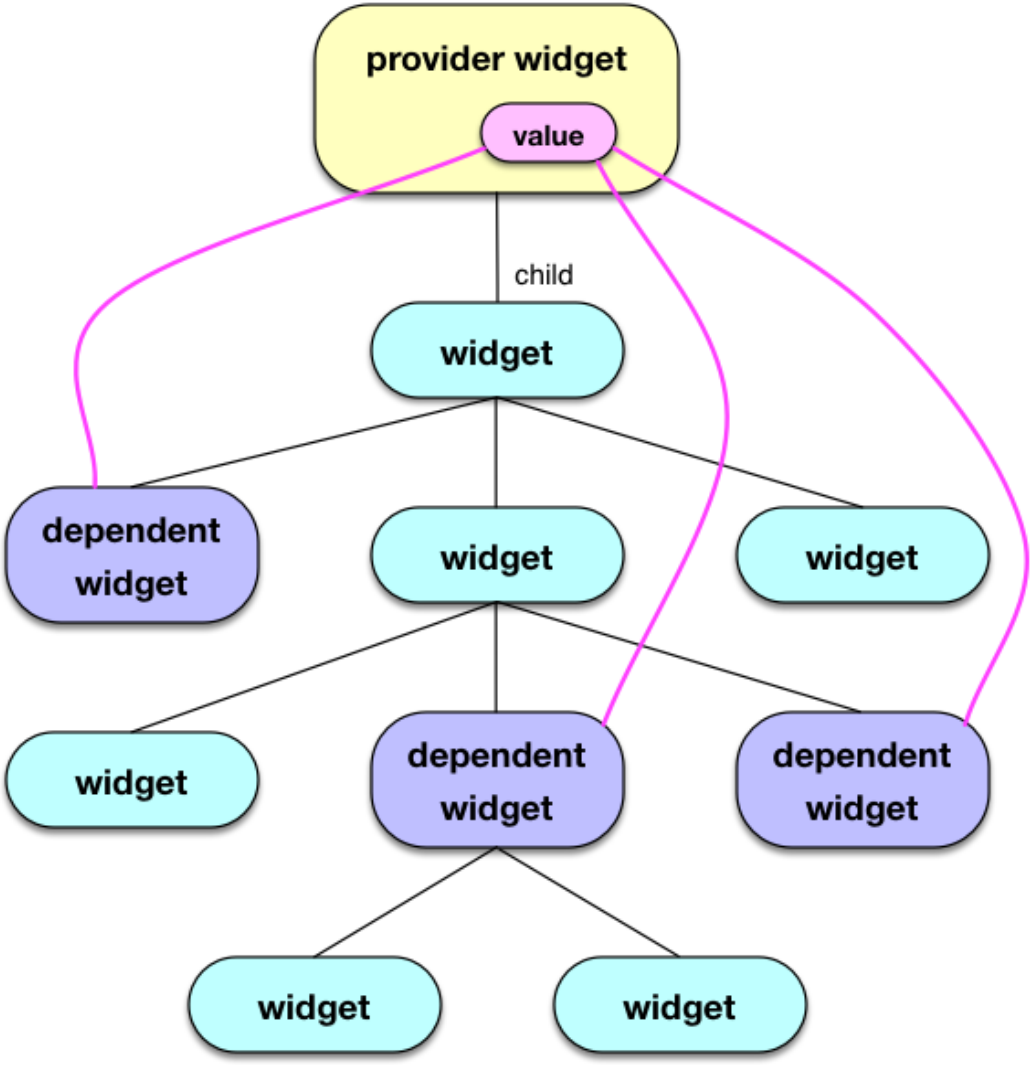
Provider Package

- The simplest example of app **state management** can be learned by using the **provider package**.
- A provider is a third-party library.
- **Provider package** is a wrapper around **InheritedWidgets** to make them more reusable and easy to use.
- Wrapping Provider around the application makes the state accessible in all the widgets.
- Provider allows you to not only expose a value, but also create, listen, and dispose of it.
- The state management with the **provider** is easy to understand and requires less coding.

Working with Providers & Listeners

- ***Providers*** are widgets that make values available to the descendent widgets that need it. These descendant widgets are called "*dependents*." A *provider* is fundamentally a means for sharing a value.
- In general, ***Provider*** goes with ***ChangeNotifier***. *ChangeNotifier* is a built-in class of Flutter that provides change notification to its listeners. With *Provider*, *ChangeNotifier* can encapsulate the app state.
- To expose a newly created object, use the default constructor of a provider. Do **not** use the .**value** constructor if you want to create an object, or you may otherwise have undesired side effects.
- **DON'T** create your object from variables that can change over time.
- To notify listening widgets whenever the state change and propel these widgets to rebuild (thus update the UI), you can call the **notifyListeners**() method.

```
class MyClass with ChangeNotifier {
   final _myList = [
     /* some data here */
   ];
   List get myList => _myList;
  /* ... */

  void updateMyList(){
    // do something with _myList
    // Then rebuild listening widgets:
    notifyListeners()
  }
}
```


provider widget
value
child
widget
dependent widget
widget
widget
widget
dependent widget
dependent widget
widget
widget

Understanding the Provider Package

- With provider, you don't need to worry about callbacks or InheritedWidgets.
  But you do need to understand 3 main concepts to use this library:

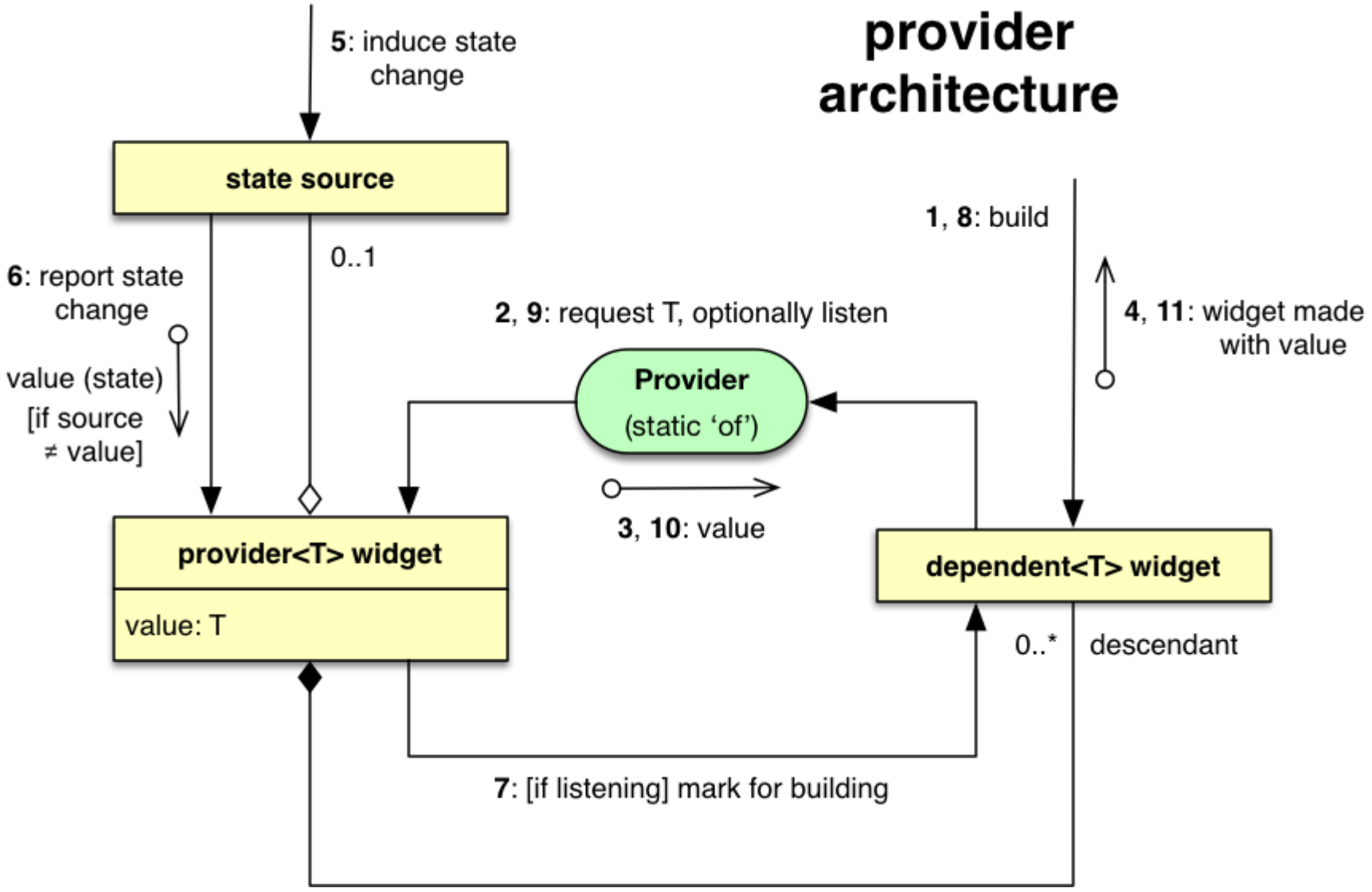
  - ChangeNotifier

  - ChangeNotifierProvider

  - Consumer

- **ChangeNotifier** is a central point that manages state of the screen. If the state inside it gets changed, then it notifies the framework to rebuilt the screen again.
- Each view will have it's own model that extends the **ChangeNotifier**.
- **ChangeNotifierProvider** provides an instance of **ChangeNotifier** in the view.
- **Consumer** is a widget that allows us to use the **ChangeNotifier**.

> *Place your consumer just above the code that needs to access the change notifier. It is because it rebuilds its descendent widgets when there are some changes and you don't want to rebuild your whole screen every time there are some changes.*


provider architecture
5: induce state change
state source
0..1
6: report state change
value (state)
[if source ≠ value]
2, 9: request T, optionally listen
Provider
(static 'of')
3, 10: value
1, 8: build
4, 11: widget made with value
provider<T> widget
value: T
dependent<T> widget
0..*
descendant
7: [if listening] mark for building

ChangeNotifier

- **ChangeNotifier** is a simple class, which provides change notification to its listeners.
- It is easy to understand, implement, and optimized for a small number of listeners.
- It is used for the listener to observe a model for changes.
- After updating the state of our data, we call **notifyListeners**() to notify all the widgets who are listening to this change so that they rebuild and update themselves.

```dart
class CartModel extends ChangeNotifier {
  /// Internal, private state of the cart.
  final List<Item> _items = [];

  /// An unmodifiable view of the items in the cart.
  UnmodifiableListView<Item> get items => UnmodifiableListView(_items);

  /// The current total price of all items (assuming all items cost $42).
  int get totalPrice => _items.length * 42;

  /// Adds [item] to cart. This and [removeAll] are the only ways to modify the
  /// cart from the outside.
  void add(Item item) {
    _items.add(item);
    // This call tells the widgets that are listening to this model to rebuild.
    notifyListeners();
  }

  /// Removes all items from the cart.
  void removeAll() {
    _items.clear();
    // This call tells the widgets that are listening to this model to rebuild.
    notifyListeners();
  }
}
```

Do I have to use ChangeNotifier for complex states? **No**.
You can use any object to represent your **state**.
For example, an alternate architecture is to use **Provider.value()** combined with a **StatefulWidget**.

```
class Example extends StatefulWidget {
  const Example({Key key, this.child}) : super(key: key);

  final Widget child;

  @override
  ExampleState createState() => ExampleState();
}

class ExampleState extends State<Example> {
  int _count;

  void increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Provider.value(
      value: _count,
      child: Provider.value(
        value: this,
        child: widget.child,
      ),
    );
  }
}
```

where we can read the state by doing:

```
return Text(context.watch<int>().toString());
```

and modify the state with:

```
return FloatingActionButton(
  onPressed: () => context.read<ExampleState>().increment(),
  child: Icon(Icons.plus_one),
);
```

ChangeNotifierProvider

- This class is defined to listen to a **ChangeNotifier** to expose the instance to its descendants and rebuild dependents whenever *ChangeNotifier.notifyListeners* is called.
- You don’t want to place ChangeNotifierProvider higher than necessary (because you don’t want to pollute the scope).
- The parent widget is wrapped in **ChangeNotifierProvider** with the datatype of the ChangeNotifier Data class.
- This includes a builder method that returns an instance of **ChangeNotifier** ‘Data’ class, which is used to expose the instance to all the children.
- Use create when a new object is created else use value to refer to an existing instance of **ChangeNotifier**.

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CartModel(),
      child: const MyApp(),
    ),
  );
}
```

Consumer

- It is a type of provider that does not do any fancy work. It just calls the provider in a new widget and delegates its build implementation to the builder.
- The **Consumer** class also works with values that aren't state or that aren't changing. If the value doesn't change, its dependent widgets don't rebuild.
- The only required argument of the **Consumer** widget is the *builder*, which is called whenever the *ChangeNotifier* changes.
- The builder is called with **three** arguments.
  - The first argument, context, contain in every build() method.
  - The second argument is the instance of the ChangeNotifier.
  - The third argument is the child that is used for optimization.
- It is the best idea to put the **consumer** widget as deep as in the tree as possible.

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text("Total price: ${cart.totalPrice}");
  },
);
```

```
// DON'T DO THIS
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return HumongousWidget(
      // ...
      child: AnotherMonstrousWidget(
        // ...
        child: Text('Total price: ${cart.totalPrice}'),
      ),
    );
  },
);
```

Instead:

```
// DO THIS
return HumongousWidget(
  // ...
  child: AnotherMonstrousWidget(
    // ...
    child: Consumer<CartModel>(
      builder: (context, cart, child) {
        return Text('Total price: ${cart.totalPrice}');
      },
    ),
  ),
);
```

Working with Multiple Providers

- If there is a need to provide more than one class, you can use **MultiProvider**.
- Another useful class from the provider package that merges multiple providers into a single linear widget tree.
- The **MultiProvider** is a list of all the different Providers being used within its scope.
- Used to improve readability and reduce boilerplate code of having to nest multiple layers of providers.
- Without using this, we would have to nest our Providers with one being the child of another and another.

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => CartModel()),
        Provider(create: (context) => SomeOtherClass()),
      ],
      child: const MyApp(),
    ),
  );
}
```

Provider.of

- Each dependent widget requests the value it needs from **Provider.of**.
- The **Provider.of** method retrieves the value from the appropriate provider widget and returns it back to the dependent.
- **Provider.of** must know where in the tree the requesting dependent resides so that it queries the nearest ancestor provider whose value has that type.
- The **Provider.of** method does the job by taking the type of value needed and the dependent's build context, and by returning a value of this type:

```
class Provider<T> {
  static T of<T>(BuildContext context, {bool listen = true}) {...}
}
```

Reading a value

- The easiest way to read a value is by using the extension methods on [**BuildContext**]:
  - context.**watch**<T>(), which makes the widget listen to changes on T
  - context.**read**<T>(), which returns T without listening to it
  - context.**select**<T, R>(R cb(T value)), which allows a widget to listen to only a small part of T.
- One can also use the static method **Provider.of**<T>(context), which will behave similarly to **watch** and when the listen parameter is set to false like **Provider.of**<T>(context, listen: false), then it will behave similarly to **read**.
- It's worth noting that context.**read**<T>() won't make a widget rebuild when the value changes and it cannot be called inside StatelessWidget.build/State.build. On the other hand, it can be freely called outside of these methods.
- These methods will look up in the widget tree starting from the widget associated with the BuildContext passed and will return the nearest variable of type T found (or throw if nothing is found).
- This operation is O(1). It doesn't involve walking in the widget tree.

```
Provider<User>(
  builder: (context) => User(), // provided value
  child: someWidgetTree
)
```

```
class MyDependent extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final user = Provider.of<User>(context);
    return Text("User name: ${user.name}");
  }
)
```

Using “Consumer” instead of “Provider.of”

- The provider package includes a convenience class for listening to state changes, Consumer, that calls Provider.of for us. Using Consumer, we can instead write:

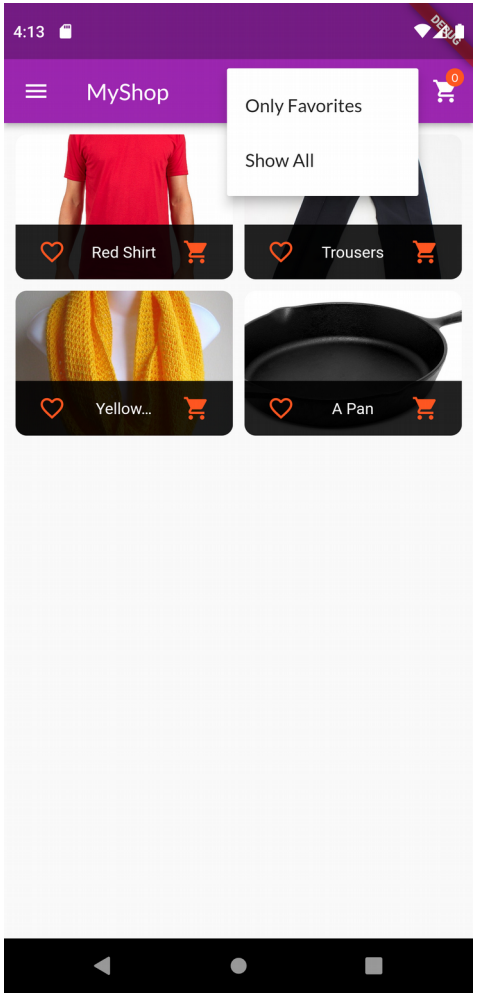
```
Consumer<User>(
  builder: (context, user, child) {
    return Text("User name: ${user.name}");
  }
)
```

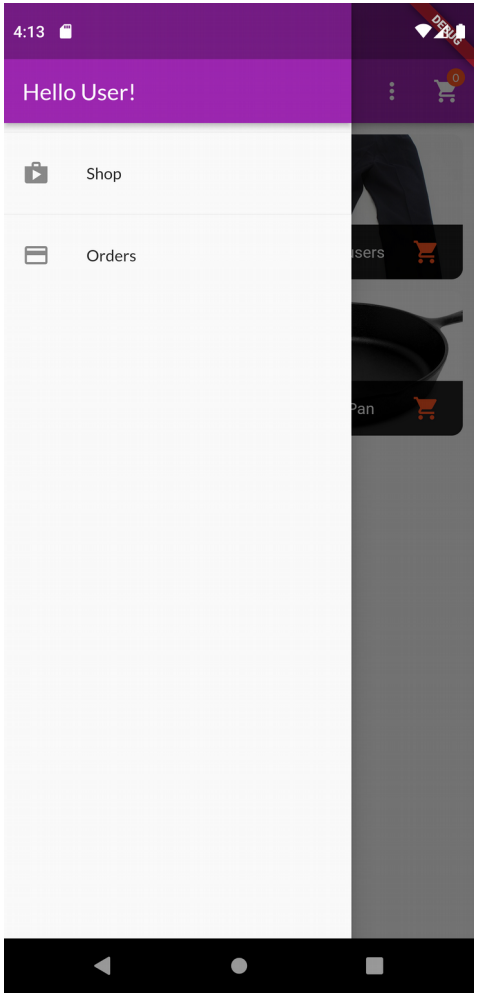
- These can be useful for performance optimizations or when it is difficult to obtain a **BuildContext** descendant of the provider.

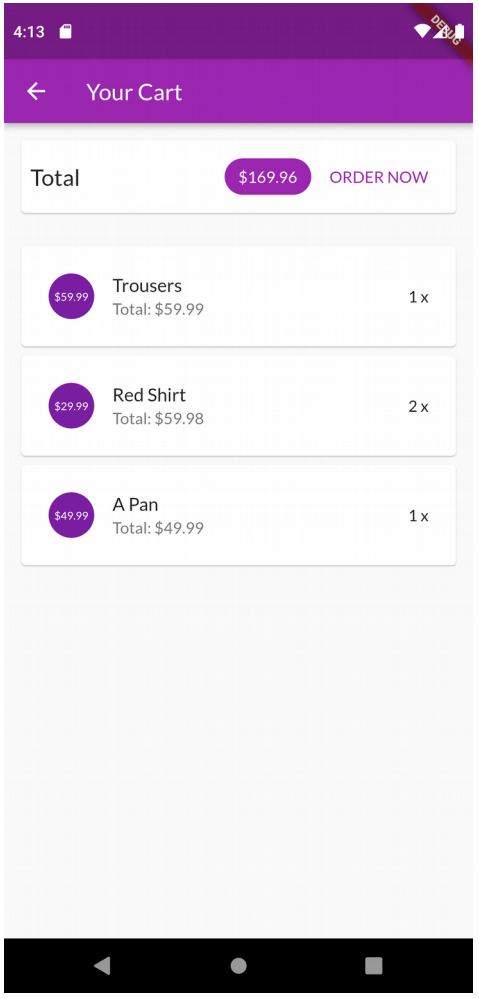
Exploring Alternate Provider Syntaxes

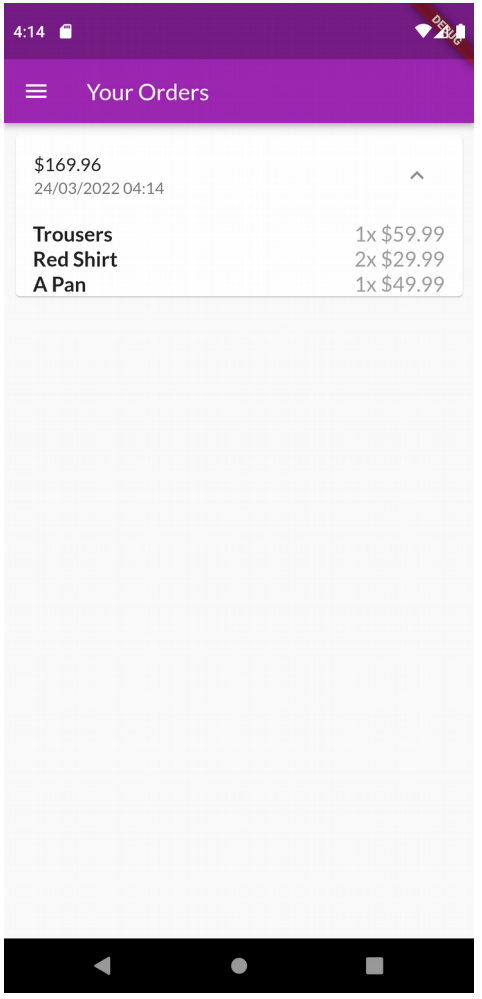
- There are at least **two** versions of every kind of provider.
- One version is responsible for creating and disposing the state source. This version “owns” the state source and manages its lifetime. We can call them “*disposing providers*.”
- The other version only references the state source and does not manage its lifetime. This version of a provider has a constructor ending in .**value**. We call them ".*value providers*”.
- In **disposing** providers, the builder parameter provides a function for creating the state source.
- In .**value** providers, there is instead a value parameter, which takes a reference to the state source.
- When using a .**value** provider, you are responsible for creating and disposing the state source as needed.

Exercise – Building Shop Application



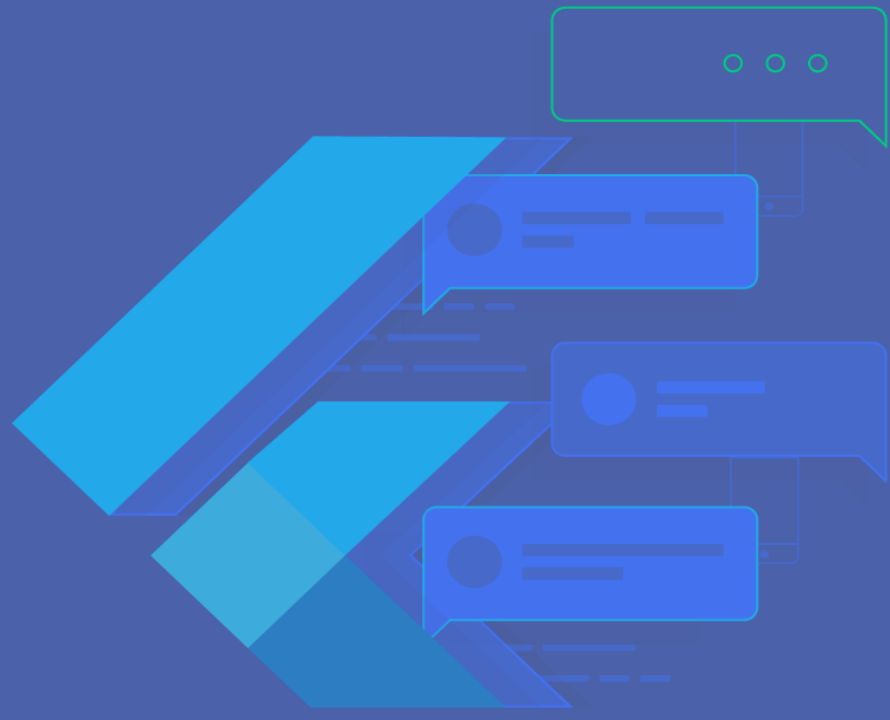






Questions ?

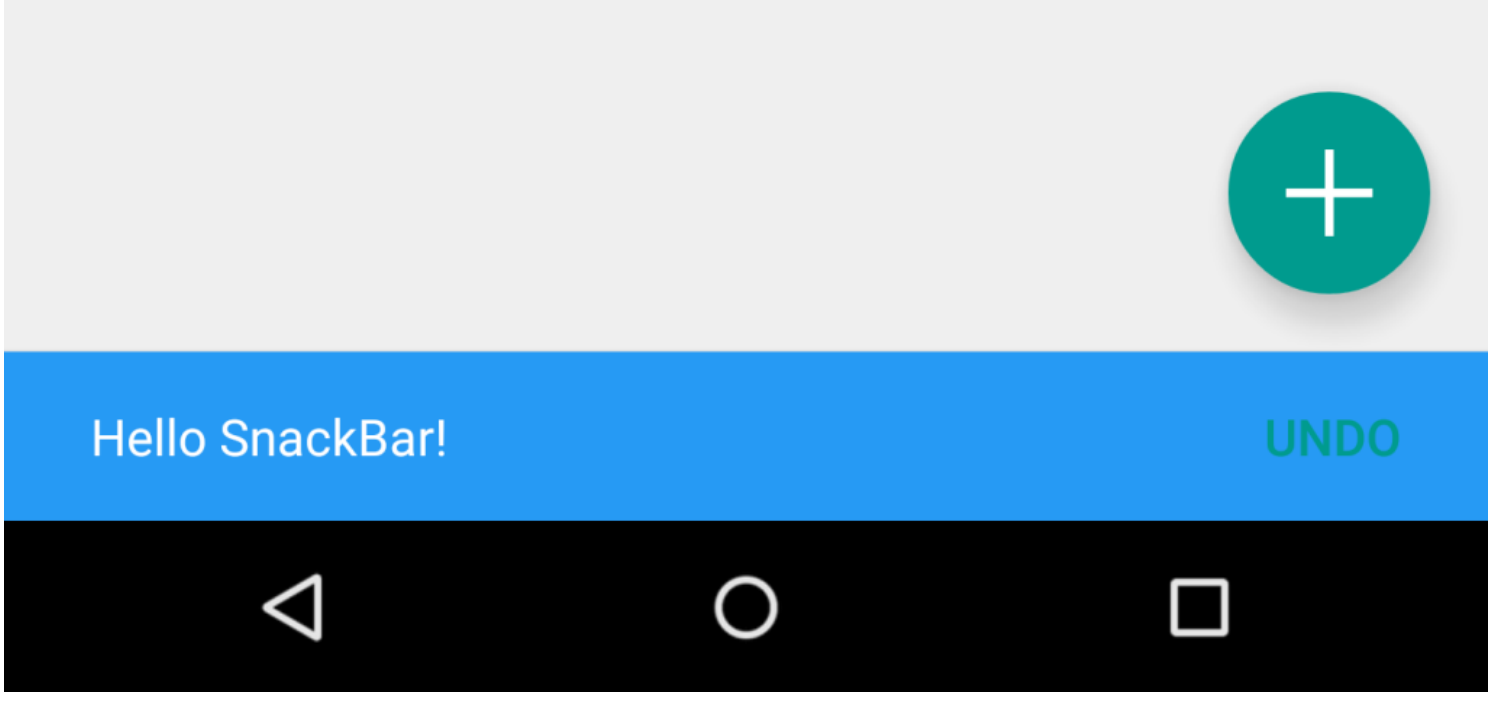
Working with User Inputs & Forms



University American College Skopje

Welcome

- SnackBar
- AlertDialog
- Using Forms & Working with Form Inputs
- Managing Form Input Focus
- Multiline Inputs & Disposing Objects
- Submitting Forms
- Validating User Input
- Improving the Shop Application



Flutter SnackBar Widget

Snackbar

- **SnackBar** is a material widget in flutter used to notify the user with a short message.
- We can create a **snackbar** in flutter by calling its constructor. There is only one required property to create a snackbar which is **content**.
- The Properties of a snackbar in flutter are:

  - backgroundColor
  - shape
  - width
  - elevation
  - action
  - dismissDirection
  - padding
  - behavior
  - margin
  - duration
  - animation
  - onVisible
- We can’t display a **snackbar** continuously like other widgets. Generally, it will be triggered by click events and lasts for few seconds and disappears.
- To control how long the SnackBar remains visible, specify a **duration**.

SnackBarAction

- A button for a **SnackBar**, known as an "**action**".
- **Snackbar** actions are always enabled. If you want to disable a **snackbar** action, simply don't include it in the snack bar.
- **Snackbar** actions can only be pressed once. Subsequent presses are ignored.
- Properties:

  - disabledTextColor → Color?

    The button disabled label color. This color is shown after the SnackBarAction is dismissed.

  - label → String

    The button label.

  - onPressed → VoidCallback

    The callback to be called when the button is pressed. Must not be null.

  - textColor → Color?

    The button label color. If not provided, defaults to SnackBarThemeData.actionTextColor.

SnackBarThemeData

- Customizes default property values for **SnackBar** widgets.
- Descendant widgets obtain the current **SnackBarThemeData** object using Theme.of(context).**snackBarTheme**.
- Instances of **SnackBarThemeData** can be customized with SnackBarThemeData.copyWith.
- Typically a **SnackBarThemeData** is specified as part of the overall Theme with ThemeData.snackBarTheme.
- The default for ThemeData.snackBarTheme provides all null properties.
- All **SnackBarThemeData** properties are null by default.
- When null, the SnackBar will provide its own defaults.

ScaffoldMessenger

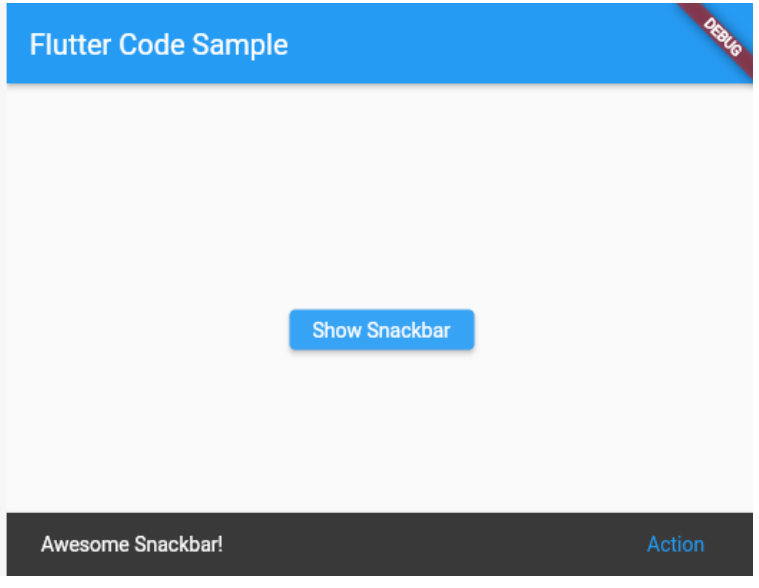
- Manages **SnackBars** and MaterialBanners for descendant Scaffolds.
- This class provides APIs for showing **snack bars** and material banners at the bottom and top of the screen, respectively.
- When the **ScaffoldMessenger** has nested Scaffold descendants, the **ScaffoldMessenger** will only present the notification to the root Scaffold of the subtree of Scaffolds.
- In order to show notifications for the inner, nested Scaffolds, set a new scope by instantiating a new **ScaffoldMessenger** in between the levels of nesting.

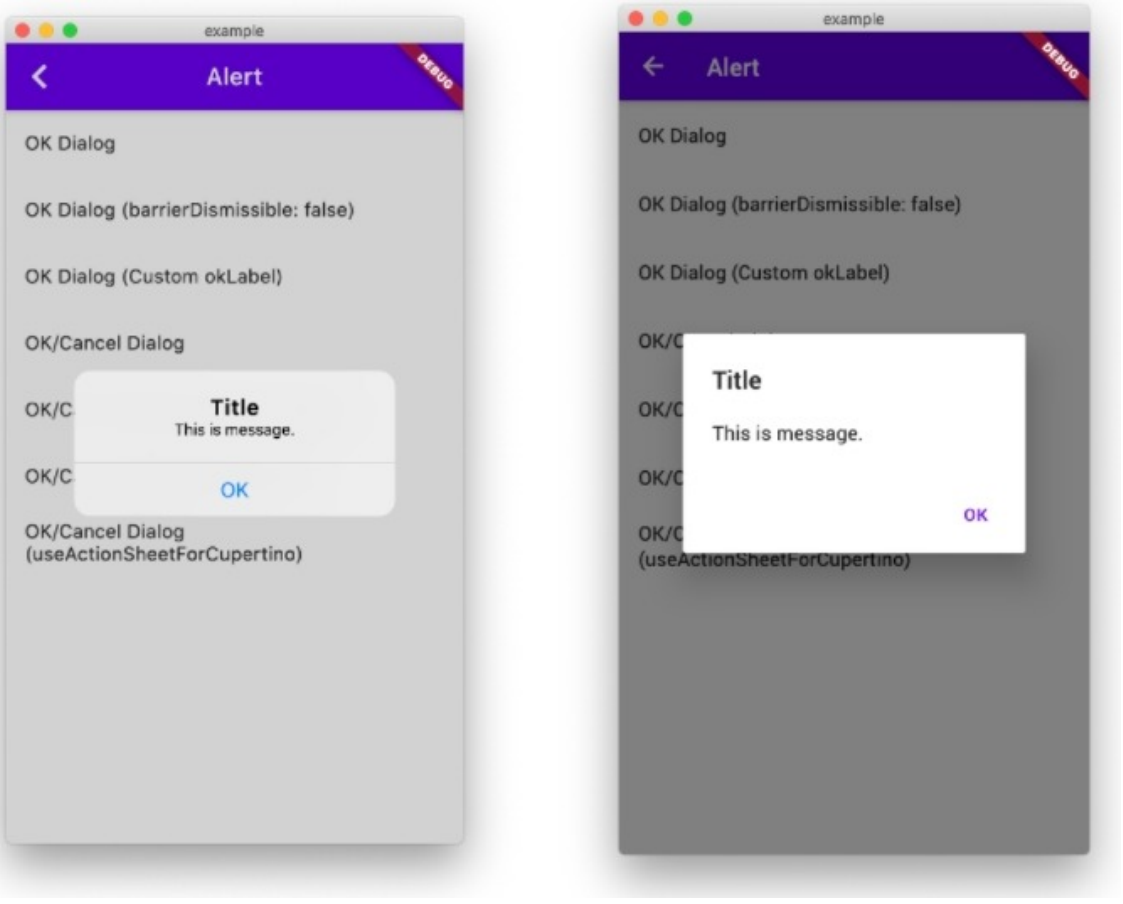
```
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return OutlinedButton(
      onPressed: () {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(
            content: Text('A SnackBar has been shown.'),
          ),
        );
      },
      child: const Text('Show SnackBar'),
    );
  }
}
```

SnackBars managed by the ScaffoldMessenger

- The temporary notification is generally shown near the bottom of the app using the **ScaffoldMessengerState.showSnackBar** method.
- **ScaffoldMessenger.of**, to obtain the current ScaffoldMessengerState for the current BuildContext, which manages the display and animation of snack bars.
- **ScaffoldMessengerState.showSnackBar**, which displays a **SnackBar**.
- To remove the **SnackBar** with an exit animation, use **hideCurrentSnackBar** or call **ScaffoldFeatureController.close** on the returned ScaffoldFeatureController.
- To remove a **SnackBar** suddenly (without an animation), use **ScaffoldMessengerState.removeCurrentSnackBar**, which abruptly hides the currently displayed snack bar, if any, and allows the next to be displayed.

```dart
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      child: const Text('Show Snackbar'),
      onPressed: () {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: const Text('Awesome Snackbar!'),
            action: SnackBarAction(
              label: 'Action',
              onPressed: () {
                // Code to execute.
              },
            ),
          ),
        );
      },
    );
```





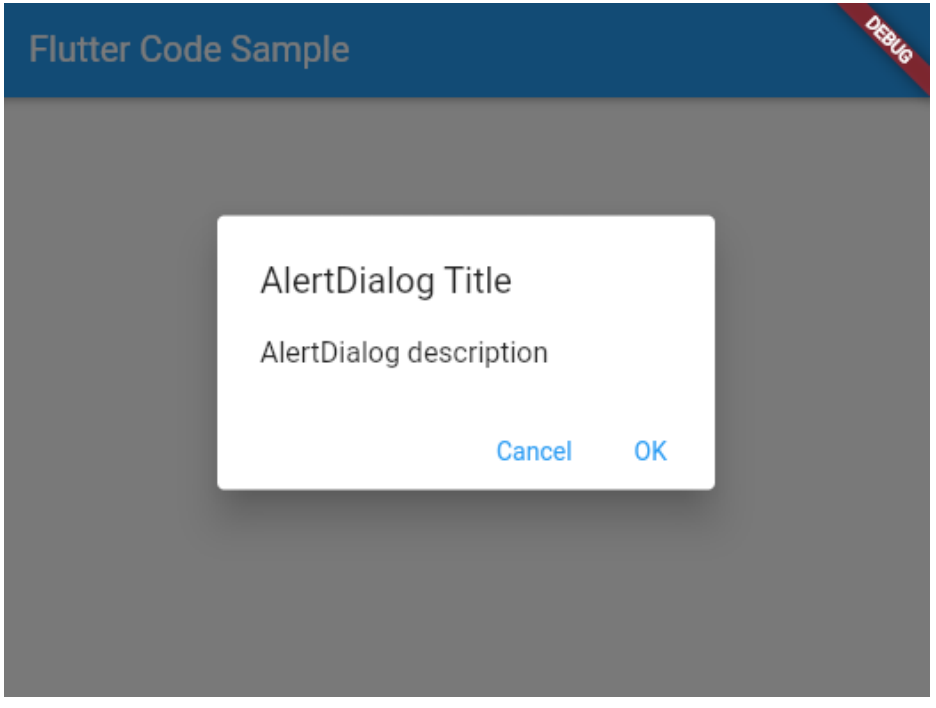
Flutter AlertDialog Widget

Showing Alert Dialogs

- An **alert dialog** informs the user about situations that require acknowledgement.
- An alert dialog has an optional title and an optional list of actions. The title is displayed above the content and the actions are displayed below the content.
- We can create an alert dialog in flutter by using the constructor. To display the **alert dialog** we have to use **showDialog()** method.
- If the content is too large to fit on the screen vertically, the dialog will display the title and the actions and let the content overflow, which is rarely desired.
- Consider using a scrolling widget for content, such as SingleChildScrollView, to avoid overflow. (However, be aware that since AlertDialog tries to size itself using the intrinsic dimensions of its children, widgets such as ListView, GridView, and CustomScrollView, which use lazy viewports, will not work. If this is a problem, consider using Dialog directly.)

Tip: For dialogs that offer the user a choice between several options, consider using a SimpleDialog.

```dart
Widget build(BuildContext context) {
  return TextButton(
    onPressed: () => showDialog<String>(
      context: context,
      builder: (BuildContext context) => AlertDialog(
        title: const Text('AlertDialog Title'),
        content: const Text('AlertDialog description'),
        actions: <Widget>[
          TextButton(
            onPressed: () => Navigator.pop(context, 'Cancel'),
            child: const Text('Cancel'),
          ),
          TextButton(
            onPressed: () => Navigator.pop(context, 'OK'),
            child: const Text('OK'),
          ),
        ],
      ),
    ),
    child: const Text('Show Dialog'),
  );
```



Forms

- **Forms** are an integral part of all modern mobile and web applications.
- It is mainly used to interact with the app as well as gather information from the users.
- They can perform many tasks, which depend on the nature of your business requirements and logic, such as authentication of the user, adding user, searching, filtering, ordering, booking, etc.
- A **form** can contain text fields, buttons, checkboxes, radio buttons, etc.
- There are **two** ways to handle **forms** in Flutter:

  - **Controller driven fields**

  - **The Form widget**

Using Forms

- Flutter provides a **Form** widget to create a form. The form widget acts as a container, which allows us to group and validate the multiple form fields.
- When you create a form, it is necessary to provide the **GlobalKey**. This key uniquely identifies the form and allows you to do any validation in the form fields.
- Each individual form field should be wrapped in a **FormField** widget, with the **Form** widget as a common ancestor of all of those.
- Call methods on **FormState** to save, reset, or validate each **FormField** that is a descendant of this Form.
- To obtain the FormState, you may use **Form.of** with a context whose ancestor is the Form, or pass a GlobalKey to the Form constructor and call **GlobalKey.currentState**.
- This widget renders a material design text field and also allows us to display validation errors when they occur.

> **Tip:** Using a `GlobalKey` is the recommended way to access a form. However, if you have a more complex widget tree, you can use the `Form.of()` method to access the form within nested widgets.

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

// Define a custom Form widget.
class MyCustomForm extends StatefulWidget {
  const MyCustomForm({Key? key}) : super(key: key);

  @override
  MyCustomFormState createState() {
    return MyCustomFormState();
  }
}

// Define a corresponding State class.
// This class holds data related to the form.
class MyCustomFormState extends State<MyCustomForm> {
  // Create a global key that uniquely identifies the Form widget
  // and allows validation of the form.
  //
  // Note: This is a `GlobalKey<FormState>`,
  // not a GlobalKey<MyCustomFormState>.
  final _formKey = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    // Build a Form widget using the _formKey created above.
    return Form(
      key: _formKey,
      child: Column(
        children: <Widget>[
          // Add TextFormFields and ElevatedButton here.
        ],
      ),
    );
  }
}
```

FormField Class

- To implement form fields, Flutter provides us with the **FormField** class, which all form fields extend.
- **FormField** widget maintains the current state of the form field, so that updates and validation errors are visually reflected in the UI.
- When used inside a **Form**, you can use methods on **FormState** to query or manipulate the form data as a whole.
- Use a **GlobalKey** with **FormField** if you want to retrieve its current state.
- A **Form** ancestor is not required. The **Form** simply makes it easier to save, reset, or validate multiple fields at once.
- To use without a **Form**, pass a **GlobalKey** to the constructor and use **GlobalKey.currentState** to save or reset the form field.

FormField Class Properties

- **initialValue**
  An optional value to initialize the form field to, or null otherwise.
- **builder**
  A callback that is responsible for building the widget inside the form field.
- **onSaved**
  An optional method to call with the final value when the form is saved via FormState.save.
- **validator**
  An optional method that validates an input. Returns an error string to display if the input is invalid, or null otherwise.
- **autovalidateMode**
  A boolean that specifies whether we want to call validator every time the field changes, or only when the field is submitted.
- **restorationId**
  Restoration ID to save and restore the state of the form field.
- **enabled**
  Whether the form is able to receive user input.

Working with Form Inputs

- **TextFields** allow users to type text into an app. They are used to build forms, send messages, create search experiences, and more.
- **TextFormField** is a convenience widget that wraps a **TextField** widget in a **FormField** widget. This provides additional functionality, such as validation and integration with other FormField widgets.
- To integrate the **TextField** into a **Form** with other **FormField** widgets, consider using **TextFormField**.
- If a controller is not specified, **initialValue** can be used to give the automatically generated controller an initial value.
- When a controller is specified, its **TextEditingController.text** defines the **initialValue**.
- If this **FormField** is part of a scrolling container that lazily constructs its children, like a *ListView* or a *CustomScrollView*, then a controller should be specified. The controller's lifetime should be managed by a stateful widget ancestor of the scrolling container.

Important: Call `dispose` of the `TextEditingController` when you've finished using it. This ensures that you discard any resources used by the object.

Managing Form Input Focus

- **Managing focus** is a fundamental tool for creating forms with an intuitive flow.
- When a text field is selected and accepting input, it is said to have "**focus**."
- Generally, users shift focus to a text field by tapping.
- We can set focus on a text field at a later point in time using **focusNode** property.
- It involves three steps:

  1. Create a **FocusNode**.
     FocusNode is an object that can be used by a stateful widget to obtain the keyboard focus and to handle keyboard events.
  2. Pass the **FocusNode** to a **TextField**.
  3. Give focus to the **TextField**.
     Use the requestFocus() method to perform this task.

- **FocusNodes** are *ChangeNotifiers*, so a listener can be registered to receive a notification when the focus changes.

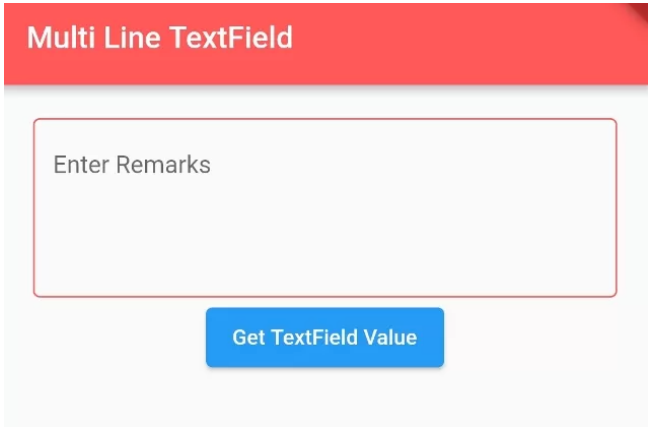
Disposing Objects

- **Listenable** is a base class of **ChangeNotifier** which is used by **TextEditingController**, **FocusNode**, **ValueNotifier** and many other flutter classes.
- **FocusNodes** are persistent objects that form part of a focus tree that is a sparse representation of the widgets in the hierarchy that are interested in receiving keyboard events.
- They must be managed like other persistent state, which is typically done by a StatefulWidget that owns the node.
- A stateful widget that owns a focus scope node must call dispose from its **State.dispose** method.

```
@override
void dispose() {
  _node.removeListener(_handleFocusChange);
  // The attachment will automatically be detached in dispose().
  _node.dispose();
  super.dispose();
}
```

Multiline Inputs

- By default, a **TextField** widget is limited to single line.
- So to make a **TextField** accept multiple lines, simply make use of **maxLines** property in **TextField** or **TextFormField** widget.
- And keyboardType property set it to **TextInputType.multiline**, so that user will get a button using which he/she can move the cursor to next line.



maxLines Property

- **maxLines** is the maximum number of lines to show at one time, wrapping if necessary.
- This affects the height of the field itself and does not limit the number of lines that can be entered into the field
- If this is 1 (the default), the text will not wrap, but will scroll horizontally instead.
- If this **is null**, there is no limit to the number of lines, and the text container will start with enough vertical space for one line and automatically grow to accommodate additional lines as they are entered, up to the height of its constraints.
- If this **is not null**, the value must be greater than zero, and it will lock the input to the given number of lines and take up enough horizontal space to accommodate that number of lines. Setting minLines as well allows the input to grow and shrink between the indicated range.

The full set of behaviors possible with **minLines** and **maxLines** are as follows.

Input that occupies a single line and scrolls horizontally as needed.

```
TextField()
```

Input whose height grows from one line up to as many lines as needed for the text that was entered. If a height limit is imposed by its parent, it will scroll vertically when its height reaches that limit.

```
TextField(maxLines: null)
```

The input's height is large enough for the given number of lines. If additional lines are entered the input scrolls vertically.

```
TextField(maxLines: 2)
```

Input whose height grows with content between a min and max. An infinite max is possible with `maxLines: null`.

```
TextField(minLines: 2, maxLines: 4)
```

These examples apply equally to **TextField, TextFormField, CupertinoTextField,** and **EditableText**.

Submitting Forms

- Here's where our **GlobalKey** plays its role. With **_formKey.currentState** we can access the state of the form. This does not mean that we can directly access the fields' values, but it provides methods to act on the form.
- When the user attempts to **submit** the form, check if the form is **valid**. You can use the **validate** and the **save** methods of the form's state.
- The **validate** function will call all the validator callbacks on the fields of the form. If there's an error, the validate will return false and rebuild the form showing the errors; otherwise, it returns true.
- If the user submits incorrect information, stop the submission and display a friendly error message letting them know what went wrong.
- If the values are ok, the next step is calling the **save** function, which will call the **onSaved** callback on the fields. In this callback function, you want to add the logic to process the value of the field. Usually, you add the value to a global object and then process the object elsewhere as you don't want to mix business logic with the presentation layer.

```
ElevatedButton(
  onPressed: () {
    // Validate returns true if the form is valid, or false otherwise.
    if (_formKey.currentState!.validate()) {
      // If the form is valid, display a snackbar. In the real world,
      // you'd often call a server or save the information in a database.
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Processing Data')),
      );
    }
  },
  child: const Text('Submit'),
),
```

Validating User Input

- Each field comes with a callback function to check the value of the field.
- **Validate** the input by providing a **validator()** function.
- If the user's input isn't valid, the validator function returns a String containing an error message.
- If there are **no** errors, the validator must return **null**.

```
TextFormField(
  // The validator receives the text that the user has entered.
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Please enter some text';
    }
    return null;
  },
),
```

Optional: Using Regular Expressions

- Like any other programming language, Dart supports *RegEx* (Regular Expression).
- Regular expressions are Patterns, and can as such be used to match strings or parts of strings.
- *RegEx* is part of the Dart code library, implemented in **RegExp** Class.
- This class offers utility methods like stringMatch(), firstMatch() or *hasMatch*() which checks whether the string you pass as an argument to hasMatch() matches the provided regular expression.
- This allows you to perform powerful checks for special/advanced patterns.

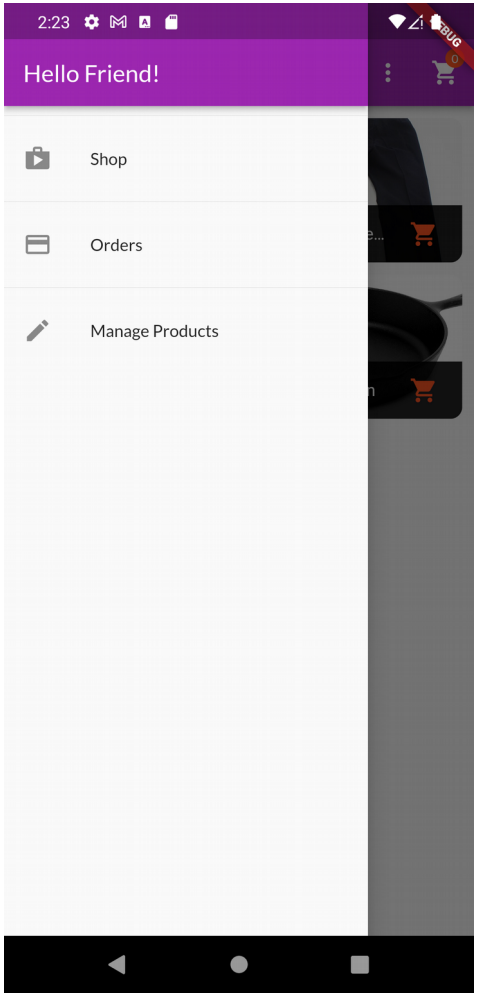
```
var urlPattern = r"(https?|ftp)://([-A-Z0-9.]+)(/[-A-Z0-9+&@#/%=~_|!:,.;]*)?(\?[A-Z0-9+&@#/%=~_|!:,.;]*)?";
var result = new RegExp(urlPattern, caseSensitive: false).firstMatch('https://www.google.com');
```

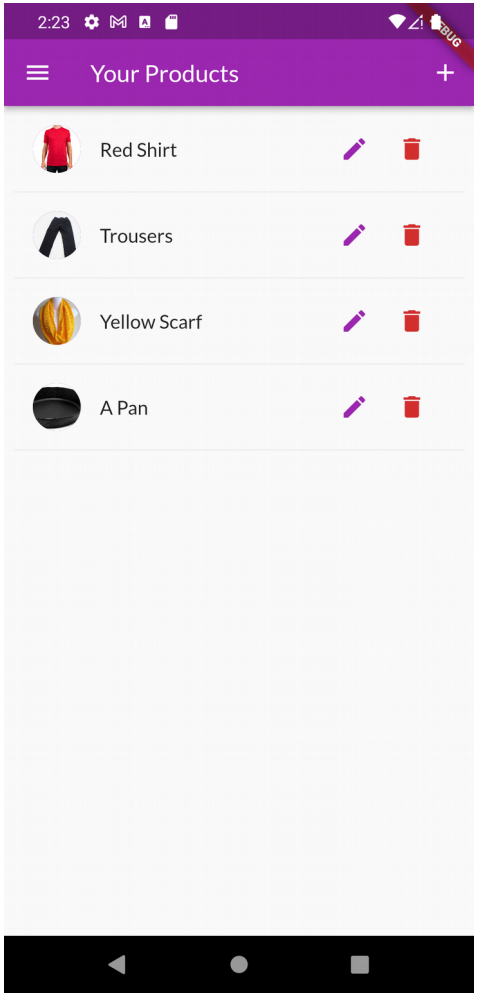
Form validation in Flutter can be implemented using simple regex validators

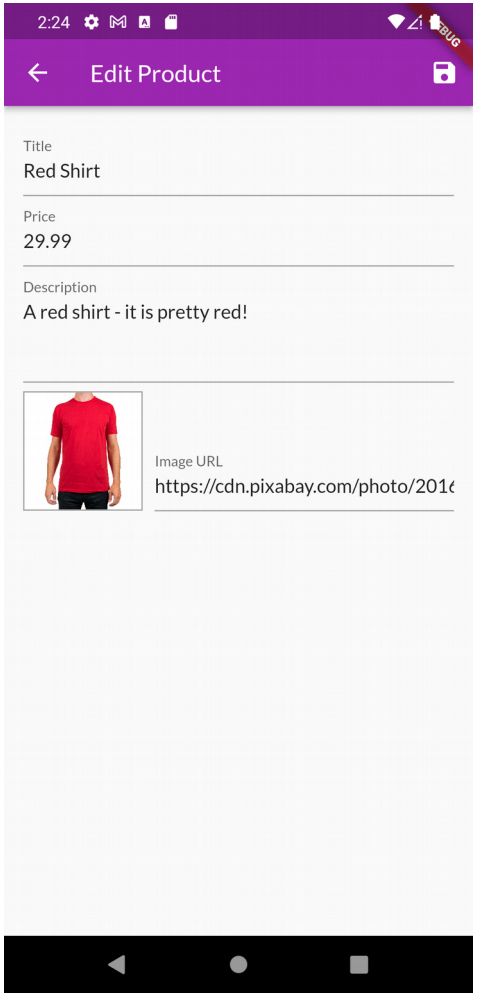
```
new TextFormField(
    obscureText: true, // To display typed char with *
    decoration: new InputDecoration(
      hintText: 'Password',
      labelText: 'Enter your password'
    ),
    validator: validatePassword(),
    onSaved: (String value) { this.userData.password = value; }
)
```

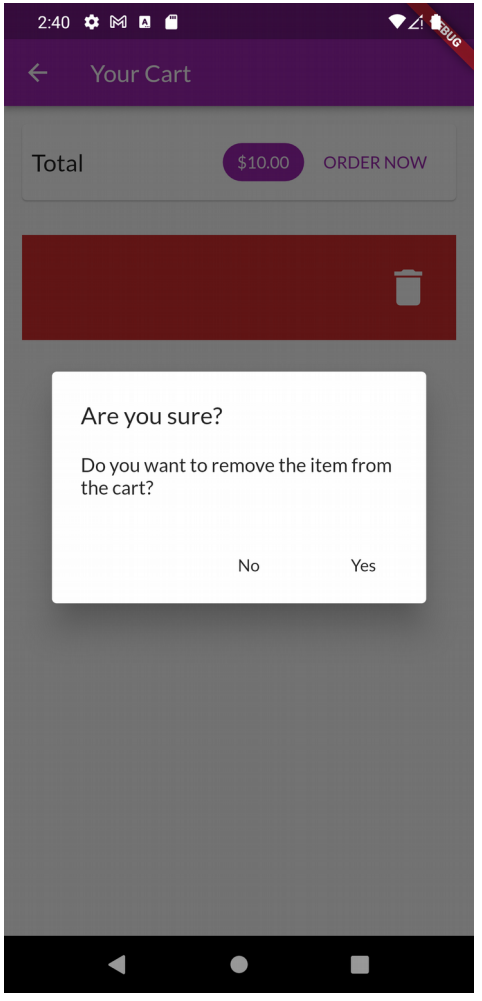
```
static FormFieldValidator<String> validateRegex(String regex, String errorMsg, bool not) {
    return (value) {
      if (RegExp(regex).hasMatch(value))
        return not? errorMsg:null;
      return not? null:errorMsg;
    };
}
static FormFieldValidator<String> validatePassword() {
    return (value) {
        Function validate = validateRegex(r"[A-Z]+", "Password should contain an uppercase character", true) ;
        String err = validate(value);
        if(err == null) {
            validate = validateNotRegex(r"[#]+", "Password should not contain #", false);
            err = validate(value);
        }
        return err;
    };
}
```

Exercise – Improving Shop Application



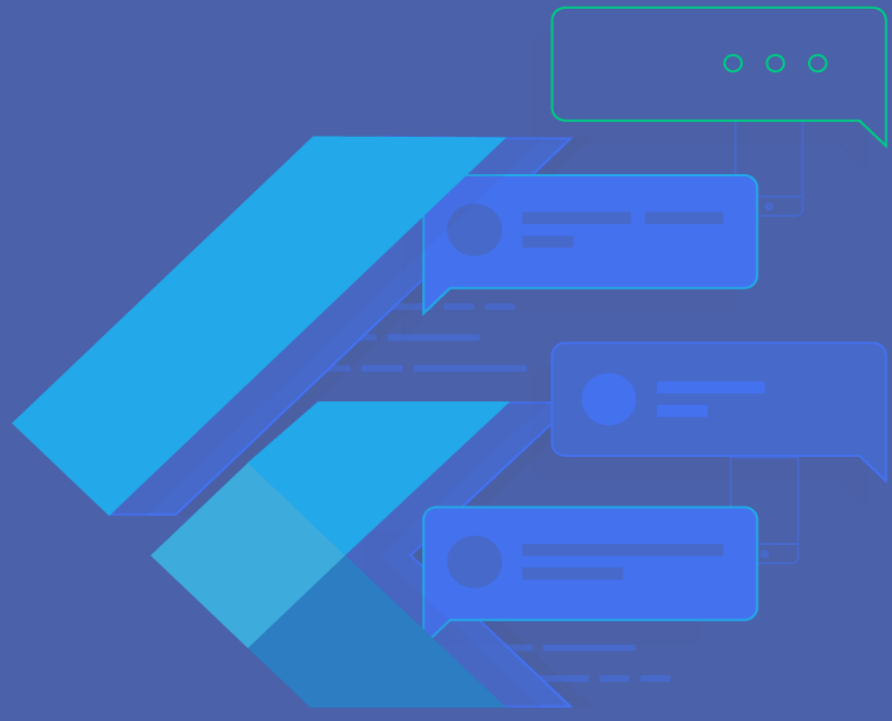






Questions ?

Sending HTTP Requests



University American College Skopje

Welcome

- Sending Http Requests
- Working with “async” & “await”
- Working with Futures
- Showing a Loading Indicator
- Implementing Pull-to-Refresh
- Using HTTP with Firebase Database
- Create Custom Exceptions & Error Handling
- Improving the Shop Application

HTTP Requests in Flutter

- Fetching data from the internet is necessary for most apps.
- **HTTP request** is used to fetch the data from the internet, It can be **JSON** format or in any other format.
- Getting to know **HTTP requests** and interfacing with web services is an essential part of many apps.
- **HTTP package** provides the simplest way to fetch data from the internet.
- Other open-source packages that can help when making HTTP requests in Flutter:

  - Dio
  - Retrofit
  - Chopper

HTTP Package

- **HTTP Package** is a composable, Future-based library f**or making HTTP requests** published by the Dart team and currently the most-liked HTTP packages on pub.dev.
- This package contains a set of high-level functions and classes that make it easy to consume HTTP resources.
- It's multi-platform and supports mobile, desktop, and browser.
- The plugin supports retrying requests. You can use the **RetryClient** class to retry failing requests.
- It also contains **IOClient**, a *dart:io*-based HTTP client and **BrowserClient**, a *dart:html*-based **HTTP client** that runs in the browser and is backed by XMLHttpRequests.
- To install the **HTTP package**, add it to the dependencies section of the pubspec.yaml file. You can find the latest version of the HTTP package the pub.dev.

Using the HTTP Package

- The easiest way to use this library is **via the top-level functions**. They allow you to make individual HTTP requests with minimal hassle.
- If you're making **multiple requests** to the same server, you can keep open a persistent connection by using a **Client** rather than making one-off requests. If you do this, make sure to close the client when you're done.
- You can also exert more fine-grained control over your requests and responses by creating **Request** or **StreamedRequest** objects yourself and passing them to **Client.send**.
- This package is designed to be composable. This makes it easy for external libraries to work with one another to add behavior to it.

```
var client = http.Client();
try {
  var response = await client.post(
      Uri.https('example.com', 'whatsit/create'),
      body: {'name': 'doodle', 'color': 'blue'});
  var decodedResponse = jsonDecode(utf8.decode(response.bodyBytes)) as Map;
  var uri = Uri.parse(decodedResponse['uri'] as String);
  print(await client.get(uri));
} finally {
  client.close();
}
```

HTTP Library Classes

BaseClient
The abstract base class for an HTTP client. [...]

BaseRequest
The base class for HTTP requests. [...]

BaseResponse
The base class for HTTP responses. [...]

ByteStream
A stream of chunks of bytes representing a single piece of data.

Client
The interface for HTTP clients that take care of maintaining persistent connections across multiple requests to the same server. [...]

MultipartFile
A file to be uploaded as part of a MultipartRequest. [...]

MultipartRequest
A `multipart/form-data` request. [...]

Request
An HTTP request where the entire request body is known in advance.

Response
An HTTP response where the entire response body is known in advance.

StreamedRequest
An HTTP request where the request body is sent asynchronously after the connection has been established and the headers have been sent. [...]

StreamedResponse
An HTTP response where the response body is received asynchronously after the headers have been received.

Request & StreamedRequest Class

- **Request:** An HTTP request where the entire request body is known in advance.
- **StreamedRequest:** An HTTP request where the request body is sent asynchronously after the connection has been established and the headers have been sent.

  - When the request is sent via BaseClient.send, only the headers and whatever data has already been written to StreamedRequest.sink will be sent immediately.

  - More data will be sent as soon as it's written to StreamedRequest.sink, and when the sink is closed the request will end.

  Sink property:

  - The sink to which to write data that will be sent as the request body.

  - This may be safely written to before the request is sent; the data will be buffered.

  - Closing this signals the end of the request.

Response & StreamedResponse Class

- **Response:** An HTTP response where the entire response body is known in advance.

  - The **body property** contains the body of the response as a string.
  This is converted from bodyBytes using the charset parameter of the Content-Type header field, if available. If it's unavailable or if the encoding name is unknown, latin1 is used by default, as per RFC 2616.

  - Response class also contains the static method

  fromStream(StreamedResponse response) → Future<Response>

  which creates a new HTTP response by waiting for the full body to become available from a StreamedResponse.

- **StreamedResponse:** An HTTP response where the response body is received asynchronously after the headers have been received.

  - Stream property contains the stream from which the response body data can be read and should always be a single-subscription stream.

HTTP Library Functions

delete(Uri url, {Map<String, String>? headers, Object? body, Encoding? encoding}) → Future<Response>
Sends an HTTP DELETE request with the given headers to the given URL. [...]

get(Uri url, {Map<String, String>? headers}) → Future<Response>
Sends an HTTP GET request with the given headers to the given URL. [...]

head(Uri url, {Map<String, String>? headers}) → Future<Response>
Sends an HTTP HEAD request with the given headers to the given URL. [...]

patch(Uri url, {Map<String, String>? headers, Object? body, Encoding? encoding}) → Future<Response>
Sends an HTTP PATCH request with the given headers and body to the given URL. [...]

post(Uri url, {Map<String, String>? headers, Object? body, Encoding? encoding}) → Future<Response>
Sends an HTTP POST request with the given headers and body to the given URL. [...]

put(Uri url, {Map<String, String>? headers, Object? body, Encoding? encoding}) → Future<Response>
Sends an HTTP PUT request with the given headers and body to the given URL. [...]

read(Uri url, {Map<String, String>? headers}) → Future<String>
Sends an HTTP GET request with the given headers to the given URL and returns a Future that completes to the body of the response as a String. [...]

readBytes(Uri url, {Map<String, String>? headers}) → Future<Uint8List>
Sends an HTTP GET request with the given headers to the given URL and returns a Future that completes to the body of the response as a list of bytes. [...]

get/delete/head function

```
Future<Response> get(
        Uri url,
        {Map<String, String>? headers}
)
```

```
Future<Response> head(
        Uri url,
        {Map<String, String>? headers}
)
```

- **HTTP GET, DELETE** or **HEAD** respectively, sends a request with the given headers to the given URL.
- This automatically initializes a new **Client** and closes that client once the request is complete. If you're planning on making multiple requests to the same server, you should use a single Client for all of those requests.
- For more fine-grained control over the request, use Request instead.

```
Future<Response> delete(
        Uri url,
        {Map<String, String>? headers,
        Object? body,
        Encoding? encoding}
)
```

post/patch/put function

```
Future<Response> post(
        Uri url,
        {Map<String, String>? headers,
        Object? body,
        Encoding? encoding}
)
```

```
Future<Response> patch(
        Uri url,
        {Map<String, String>? headers,
        Object? body,
        Encoding? encoding}
)
```

```
Future<Response> put(
        Uri url,
        {Map<String, String>? headers,
        Object? body,
        Encoding? encoding}
)
```

- **HTTP POST, PATCH** or **PUT** respectively, sends a request with the given headers and body to the given URL.
- **body** sets the body of the request. It can be a String, a List<int> or a Map<String, String>.
  If it's a String, it's encoded using **encoding** and used as the body of the request. The content-type of the request will default to "text/plain".
- If **body** is a List, it's used as a list of bytes for the body of the request.
- If body is a **Map**, it's encoded as form fields using **encoding**. The content-type of the request will be set to "application/x-www-form-urlencoded"; this cannot be overridden.
- **encoding** defaults to utf8.
- For more fine-grained control over the request, use Request or StreamedRequest instead.

Working with “async” & “await”

- To perform an asynchronous computation, you use an **async** function which always produces a **future**.
- Inside such an asynchronous function, you can use the **await** operation to delay execution until another asynchronous computation has a result.
- While execution of the awaiting function is delayed, the program is **not** blocked, and can continue doing other things.
- The **async** and **await** keywords provide a declarative way to define asynchronous functions and use their results. Remember these two basic guidelines when using **async** and **await**:

  - To define an async function, add async before the function body

  - The await keyword works only in async functions.

Futures

- **Dart is a single-threaded programming language**. However, Dart and Flutter have its answer when it comes to asynchronous operations.
- They together perform long-running operations with the help of **Future API**, **async**, **await** keywords, and **then** functions.
- **Future<T>** object represents the result of an asynchronous operation which produces a result of type T.
- If the result is not usable value, then the future's type is **Future<void>**.
- There are 2 ways to handle Futures:

  - Using the **Future API**

  - Using the **async** and **await** operation.

Futures & Async Code

- **Future** gives us a promise token and says that a value will be returned at some point in future.
- A future represents the result of an asynchronous operation, and can have two states:

  - uncompleted or **completed**.
- With a Future, you can manually register callbacks that handle the value, or error, once it is available.
- Since a Future can be completed in two ways, either with a value (if the asynchronous computation succeeded) or with an error (if the computation failed), you can install callbacks for either or both cases.
- If a **future** completes with an error, awaiting that future will (re-)throw that error.
- If a future does not have any registered handler when it completes with an error, it forwards the error to an "uncaught-error handler". This behavior ensures that no error is silently dropped.
- A future may also fail to ever complete. In that case, **no** callbacks are called.

Note: *Uncompleted* is a Dart term referring to the state of a future before it has produced a value.

Working with Futures in Dart

- There are **two ways** to execute a Future and use the value it returns. If it returns any.
- The most common way is to **await** on the **Future** to return. For this to work your function that's calling the code has to be marked **async**.

```
FlatButton(
  child: Text('Run Future'),
  onPressed: () async {
    var value = await myTypedFuture();
  },
)
```

- Sometimes you **don't** want to turn the function into a **Future** or mark it async, so the other way to handle a future is by using the .**then** function. It takes in a function that will be called with the value type of your Future.

```
void runMyFuture() {
  myTypedFuture().then((value) {
    // Run the code here using the value
  });
}
```

FutureBuilder Class

- **FutureBuilder** is a widget that builds itself based on the latest snapshot of interaction with a **Future**.
- The **future** must have been obtained earlier, e.g. during *State.initState, State.didUpdateWidget*, or *State.didChangeDependencies*.
- It must **not** be created during the *State.build* or *StatelessWidget.build* method call when constructing the **FutureBuilder**.
- If the **future** is created at the same time as the **FutureBuilder**, then every time the FutureBuilder's parent is rebuilt, the asynchronous task will be restarted.
- **FutureBuilder** is Stateful by nature i.e it maintains its own state as we do in StatefulWidgets.
- When we use the **FutureBuilder** widget we need to check for future state i.e future is resolved or not and so on.

FutureBuilder Properties

Key? key: The widget's key, used to control how a widget is replaced with another widget.

Future<T>? future: A Future whose snapshot can be accessed by the builder function.

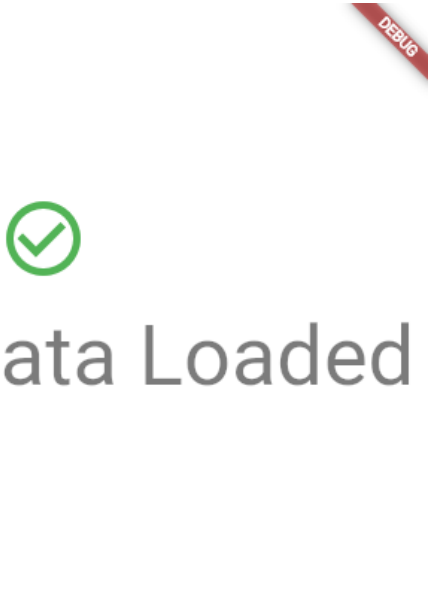
T? initialData: The data that will be used to create the snapshots until a non-null Future has completed.

required AsyncWidgetBuilder<T> builder: The build strategy used by this builder.

AsyncWidgetBuilder

- You are required to pass an **AsyncWidgetBuilder** function which is used to build the widget.
- The function has **two** parameters. The first parameter's type is **BuildContext**, while the second parameter's type is **AsyncSnapshot<T>**. You can use the value of the second parameter to determine the content that should be rendered.
- **AsyncSnapshot** is described as an immutable representation of the most recent interaction with an asynchronous computation.
- There are some important properties of **AsyncSnapshot** that can be useful. The first one is **connectionState** whose type is ConnectionState enum. (none, waiting, active, done).
- Another property you need to know is **hasError**. It can be used to indicate whether the snapshot contains a non-null error value. If the last result of the asynchronous operation was failed, the value will be true.
- To check whether the snapshot contains non-null data, you can use the **hasData** property. The asynchronous operation has to be complete with non-null data in order for the value to become true.
- However, if the asynchronous operation completes without data (e.g. Future<void>), the value will be false. If the snapshot has data, you can obtain it by accessing the data property.

```dart
class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  final Future<String> _calculation = Future<String>.delayed(
    const Duration(seconds: 2),
    () => 'Data Loaded',
  );

  @override
  Widget build(BuildContext context) {
    return DefaultTextStyle(
      style: Theme.of(context).textTheme.headline2!,
      textAlign: TextAlign.center,
      child: FutureBuilder<String>(
        future: _calculation, // a previously-obtained Future<String> or null
        builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
          List<Widget> children;
          if (snapshot.hasData) {
            children = <Widget>[
              const Icon(
                Icons.check_circle_outline,
                color: Colors.green,
```



Handling Errors

- You use a normal **try/catch** to catch the failures of awaited asynchronous computations.
- In general, when writing asynchronous code, **you should always await a future** when it is produced, and not wait until after another asynchronous delay.
- That ensures that you are ready to receive any error that the future might produce, which is important because an asynchronous error that no-one is awaiting is an uncaught error and may terminate the running program.
- **Futures** has its own way of **handling errors**.
- Future has a strategy called **catchError** which is utilized to deal with errors transmitted by the Future. It's what asynchronous be compared to a **catch** block.
- You need to pass a callback that will be called when the Future emits an error. The passed callback can have one or two parameters.
- When the callback is called, **the error** is passed as the first argument. If the callback accepts two parameters, **the stack trace** will be passed as the second argument.

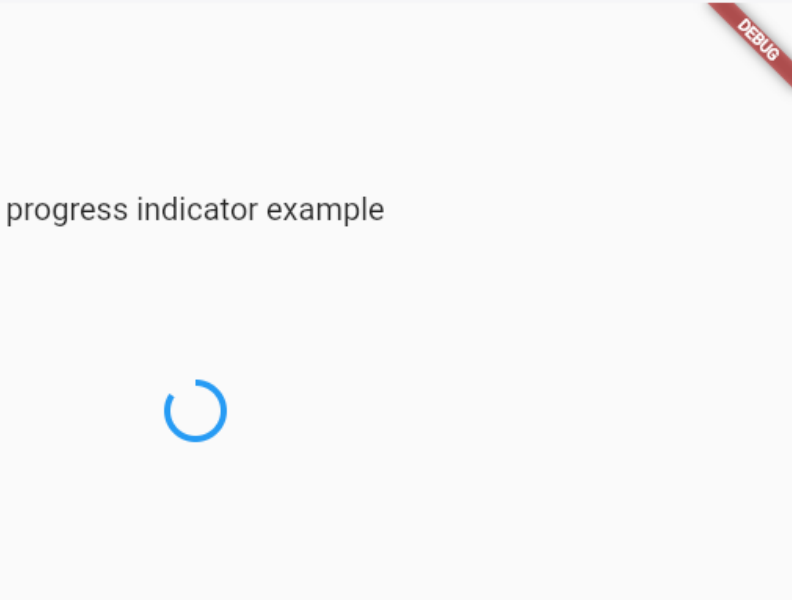
CircularProgressIndicator Class

- A widget that **shows progress along a circle**, which spins to indicate that the application is busy.
- There are **two** kinds of circular progress indicators:

  - *Determinate*. Determinate progress indicators have a specific value at each point in time, and the value should increase monotonically from 0.0 to 1.0, at which time the indicator is complete. To create a determinate progress indicator, use a non-null value between 0.0 and 1.0.

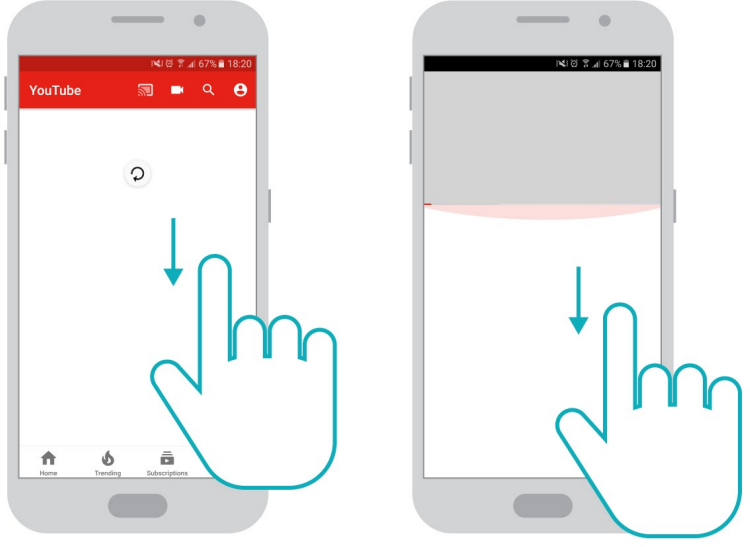
  - *Indeterminate*. Indeterminate progress indicators do not have a specific value at each point in time and instead indicate that progress is being made without indicating how much progress remains. To create an indeterminate progress indicator, use a null value.

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Padding(
      padding: const EdgeInsets.all(20.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Text(
            'progress indicator example',
            style: Theme.of(context).textTheme.headline6,
          ),
          CircularProgressIndicator(
            value: controller.value,
            semanticsLabel: 'Linear progress indicator',
          ),
        ],
      ),
    ),
  );
```



RefreshIndicator Class

- A widget that supports the Material "*swipe to refresh*" idiom.
- When the child's Scrollable descendant overscrolls, **an animated circular progress indicator** is faded into view.
- When the scroll ends, if the indicator has been dragged far enough for it to become completely opaque, the **onRefresh** callback is called.
- The callback is expected to update the scrollable's contents and then complete the Future it returns. The refresh indicator disappears after the callback's Future has completed.
- The trigger mode is configured by **RefreshIndicator.triggerMode**.
- A **RefreshIndicator** can only be used with a vertical scroll view.



Here's the list of **named parameters** you can pass to the constructor of **RefreshIndicator**.

Key key: The widget's key.

Widget child *: The widget below this widget in the tree. It contains the content that can be refreshed.

double displacement: Where the refresh indicator will settle measured from the child's top or bottom edge.

double edgeOffset: The offset where RefreshProgressIndicator starts to appear on drag start.

Future<void> onRefresh *: A callback function to be called when the refresh indicator has been dragged far enough which means a refresh should be performed.

Color color: The foreground color of the refresh indicator.

Color backgroundColor: The background color of the refresh indicator.

bool notificationPredicate: Specifies whether a [ScrollNotification] should be handled by this widget. Defaults to defaultScrollNotificationPredicate.

String semanticsLabel: Semantics.label for the widget.

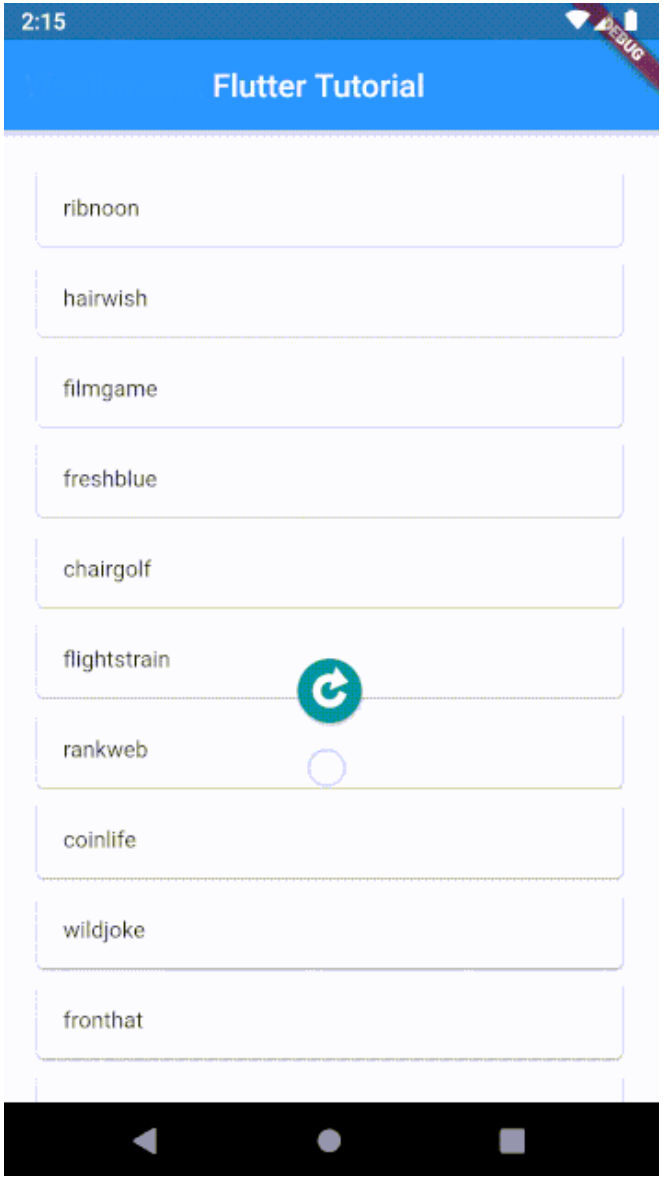
String semanticsValue: Semantics.value for the widget.

double strokeWidth: The stroke width of the refresh indicator. Defaults to 2.0.

RefreshIndicatorTriggerMode triggerMode: Defines how this RefreshIndicator can be triggered when users overscroll.

*: required

```dart
RefreshIndicator(
  onRefresh: _refreshData,
  backgroundColor: Colors.teal,
  color: Colors.white,
  displacement: 200,
  strokeWidth: 5,
  child: ListView.builder(
    padding: EdgeInsets.all(20.0),
    itemBuilder: (context, index) {
      WordPair wordPair = _data[index];

      return _buildListItem(wordPair.asString, context);
    },
    itemCount: _data.length,
  ),
)
```



Using HTTP with Firebase Database

1. Preparing our backend

- There are two types of database available cloud firestore and realtime database. We will be using a **realtime database** because it is easy to implement.

The Firebase Realtime Database is a cloud-hosted database.

- Data is stored as JSON and synchronized in realtime to every connected client.
- When you build cross-platform apps Flutter & Firebase, all of your clients can share one Realtime Database instance and automatically receive updates with the newest data.

2. Sending POST request

- To generate the post request we need the **URL** of our database.

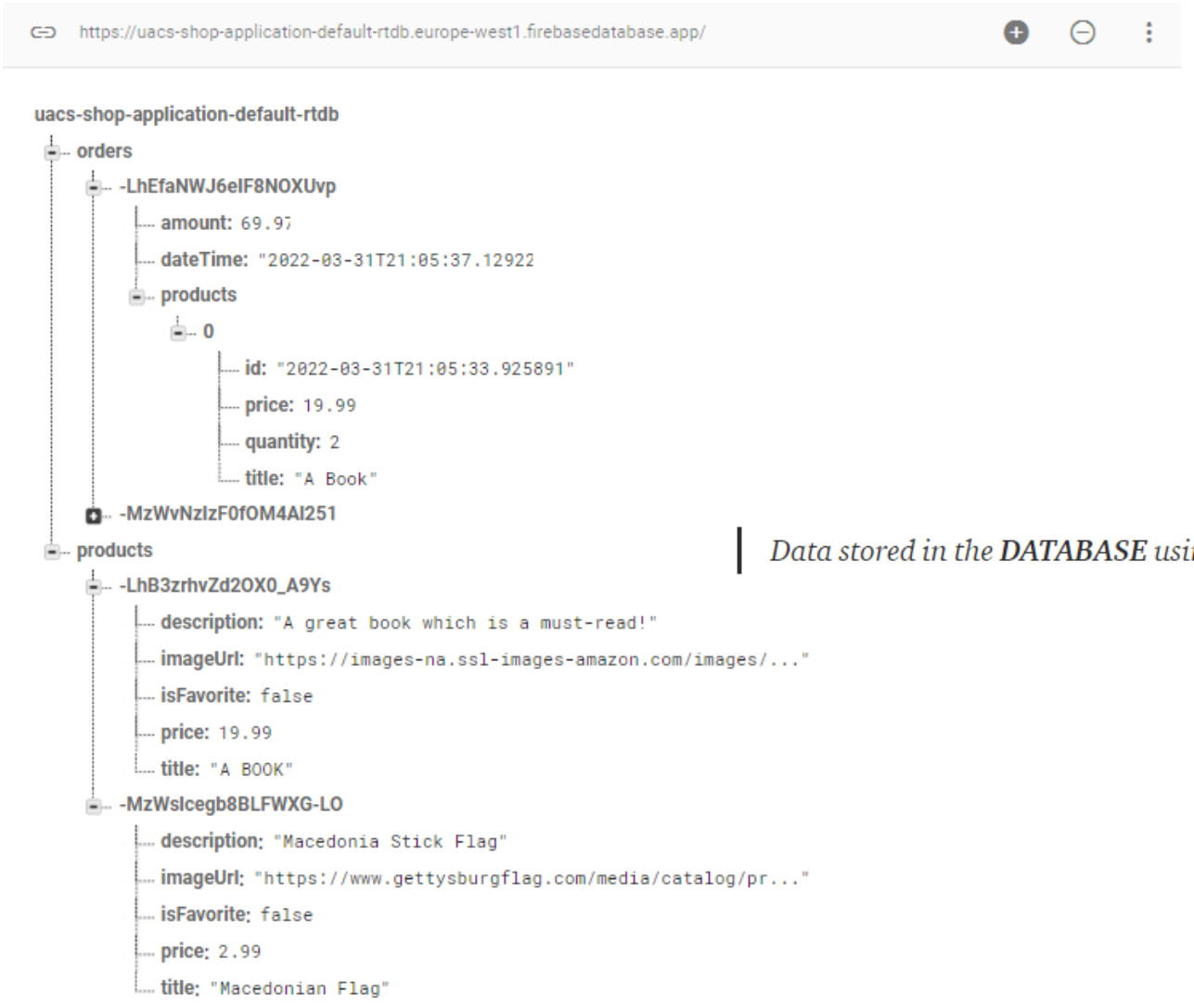
- Also we will have to append “/products.json” or “/orders.json” at the end of the URL. This is very important to add because the names “**products**” and “**orders**” are used as fields which are used to store the various attributes with a unique Id auto-generated by the firebase.

3. Sending GET request

- To fetch the data we will again need the same **URL**.

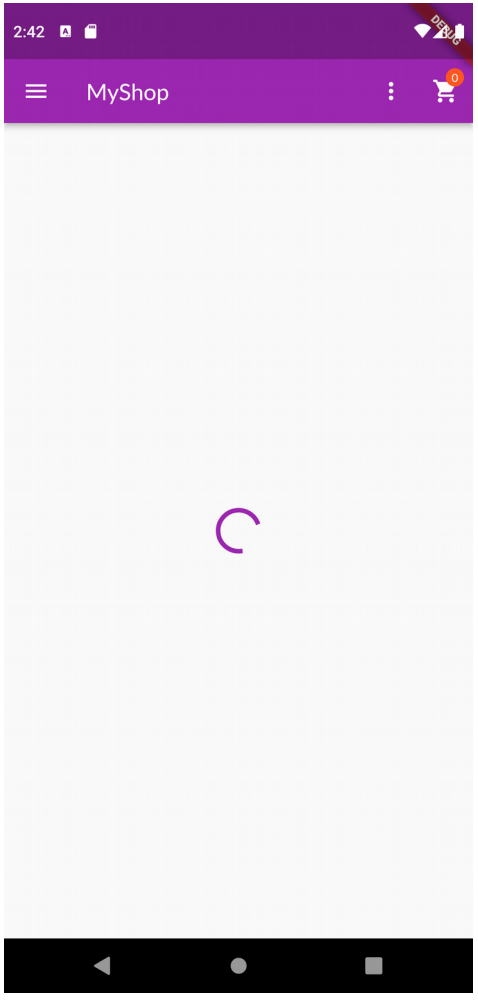
- We will get the **JSON** data from the database, therefore, we will need to add a **decode()** statement.

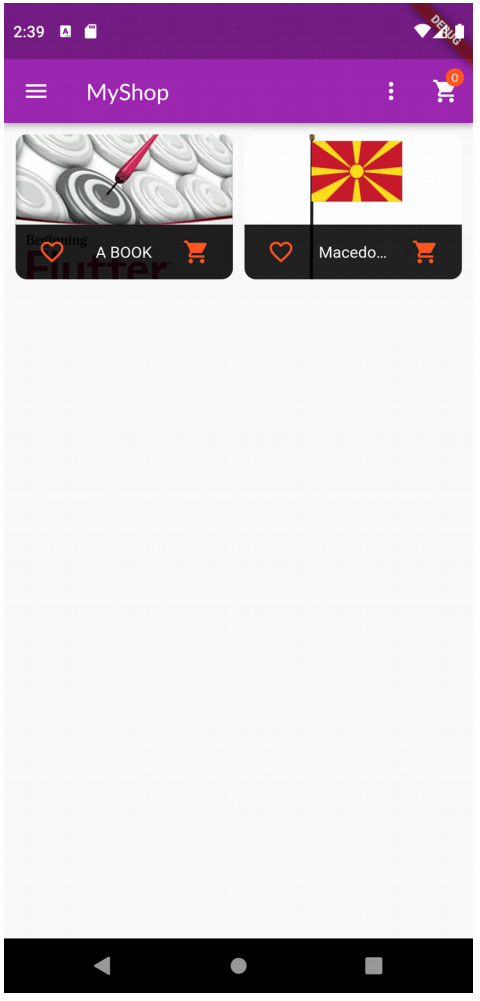
Note: HTTP status code are used to determine if a request was successful or unsuccessful. A status code of `200` represents a successful HTTP request.

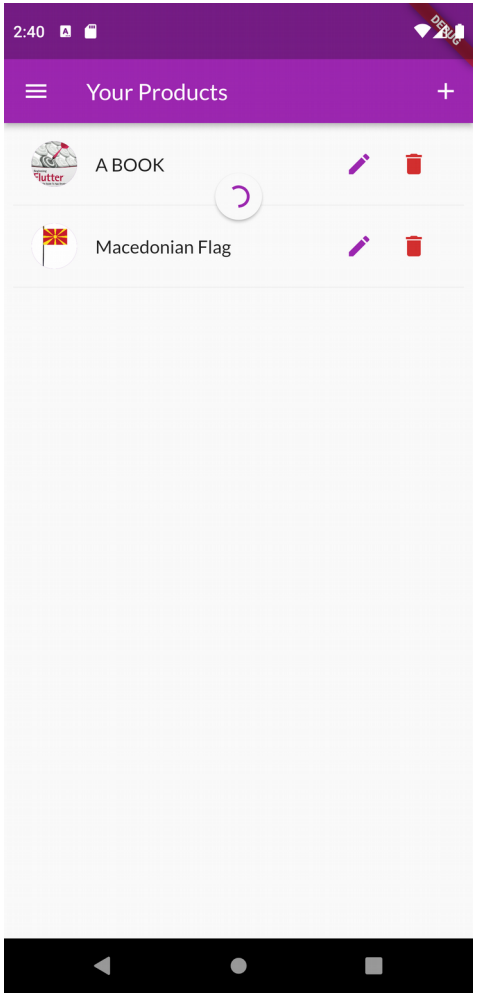


*Data stored in the **DATABASE** using **post** request looks like this .....*

Exercise – Improving Shop Application







Questions ?

Adding User Authentication

University American College Skopje

Welcome

- User Authentication and How It Works?
- Local Authentication
- Common Authentication Types
- Persisting User Auth State
- Authentication Packages for Flutter
- Flutter Firebase Authentication
- ProxyProvider and ChangeNotifierProxyProvider
- Timer Class
- Improving the Shop Application

User Authentication

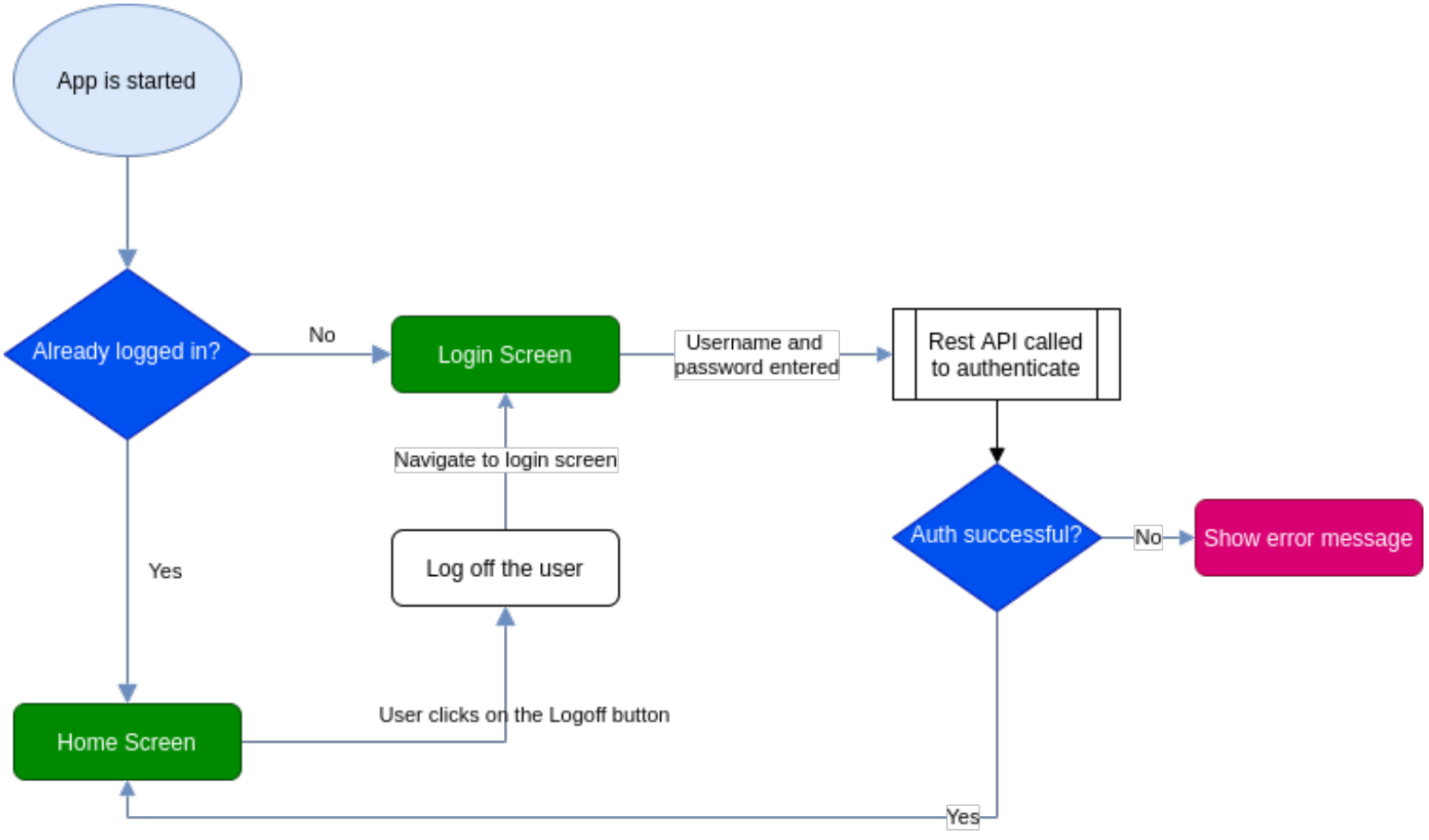
- **Authentication** is undoubtedly one of the most important things in eCommerce application development.
- **User authentication** is about how users prove that they are the legitimate app users.
- **Signing user up, in and out** are nearly universal features for every type of app.

Interplay between User/App Authentication

- In addition to the user need to be authenticated also the mobile app will need to authenticate to the back-end system in some way.
- **App authentication** covers how the app authenticates towards the backend.
- On one end of the spectrum **user authentication** and **app authentication** are decoupled.
- The user authenticates first, after which the app is ready for use. All subsequent business logic request to the back-end server rely on **app authentication**.
- On the other end of the spectrum, **user authentication** and **app authentication** coincide, i.e. every time the app needs to authenticate, the user will interact.
- Clearly some trade-off is required between both extremes, as security requirements often mandate some level of user interaction (within a certain time frame), but without overdoing it and constantly asking the user to authenticate.

How Authentication Works?

- In order to sign a user into your **Flutter** app, first, you have to get some pieces of information from your targeted users.
- These can be the user's **email address and password**, or an **OAuth token** from a federated identity provider.
- There are lots of ways to achieve this, especially in **Flutter**.
- Some of the popular ways include adding **authentication** to our app by employing *Firebase* or building your own *custom API* and integrating it in your applications to deal with **user authentication**.


App is started
Already logged in?
No
Login Screen
Username and password entered
Rest API called to authenticate
Navigate to login screen
Auth successful?
No
Show error message
Yes
Log off the user
Home Screen
User clicks on the Logoff button
Yes

Local Authentication

- Verifying the user locally i.e. without the use of a web server, using facelock or fingerprint can be termed as **Local Authentication**.

  1. biometric authentication on iOS

  2. fingerprint APIs on Android

- Sometimes **device authentication** is also mentioned, i.e. some device fingerprint being sent towards the backend.
- Flutter provides *local_auth* plugin to provide means to perform local, on-device authentication of the user.
- Comparing to native frameworks, *local_auth* provides more generic biometric authentication functionalities that suffice basic needs.

Note that device authentication is accessible to all apps running on your user's mobile phone and generally easy to spoof.

Common Authentication Types

- **Password**-based authentication. (Passwords are the most common methods of authentication.)
- **Multi-factor** authentication.
- **Certificate**-based authentication.
- **Biometric** authentication.
- **Token**-based authentication.

User Experience

- Users only see the tip of the iceberg of the **authentication process**.
- Namely, whenever **they are asked to**:

  - enter some information (e.g., username, PIN, password, OTP),
  - perform biometric verification (e.g., fingerprint, faceID),
  - confirm an action.
- All the real complexity, such as cryptography and protocols, are hidden from the user.
- Regarding the user experience, the key questions are **how and how often (when) a user should authenticate**.
- **On the one hand**, all user interactions create friction, so reducing the number of interactions is generally good for the user experience.
- **On the other hand**, long lasting sessions (where no user interaction is required) are a potential security disaster and might leave some of your users wondering if your mobile app is secure (e.g., when their phone is left unattended, will someone else be able to use the mobile app?).
- The answer is thus **not** straightforward and highly depends on the value and risk associated with the usage of the mobile app.
- What kind of data is processed and can be accessed? What is the risk of unauthorized data access, the ability to approve transactions or the ability to sign documents? Should locally stored data be protected?

User Auth State

- The first challenge when it comes to **authentication** is to determine if we have a user or not and which page to open when the application starts up.
- In most apps, you only have to log in once and it remembers the status on subsequent visits — that is, it automatically signs you into the app so that you don’t have to provide your credentials every time.
- The management and handling of such a state are known as **data persistence** which can be done by applying a **persistent storage mechanism** in the Flutter application. It is necessary to retain the data that was before the application has been shut off.
- In Flutter there are many solutions to store data locally on users' devices instead of using remote cloud servers or APIs.

Secure Storage

- The *flutter_secure_storage* plugin provides secure storage via **iOS Keychain** and **Android KeyStore**.
- For a persistent store of simple data, the *shared_preferences* plugin is available.
- However, quoting from its official documentation, neither platform can guarantee that writes will be persisted to disk after returning; hence the plugin should not be used for storing critical data.

Authentication Packages for Flutter

google_sign_in

A Google Sign-In authentication system for signing in with a Google account on Android and iOS using the Flutter plugin.

http_auth

The library deploys a HTTP basic or digest authentication from Dart.

firebase_auth

The Flutter plugin for Firebase Auth enables Android and iOS authentication using passwords, phone numbers, and identity providers like Google, Facebook, and Twitter.

local_auth

Using the Flutter plugin for Android and iOS devices, you may now enable local authentication with fingerprint, touch ID, face ID, passcode, pin, or pattern.

fresh_dio

Fresh, which is a package:dio module for refreshing authentication tokens without affecting page load speed. Fresh is built on top of package:dio and handles authentication tokens transparently.

pinput

After pressing the number keys, you may need to type in your personal Pin code (OTP) text field. It supports custom numpads as well.

Flutter Firebase Authentication

- **Firebase Authentication** provides backend services & easy-to-use SDKs to authenticate users to your app.
- It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.
- Before using any **sign-in methods**, ensure you have configured the sign-in methods on the Firebase console.
- In Flutter you can query the Firebase Auth backend through a REST API or with firebase_auth plugin.

HTTPS is required. Firebase only responds to encrypted traffic so that your data remains safe.

Firebase Email/Password Auth

- You can **create a new email and password user** by issuing an HTTP POST request to the Firebase Auth REST API **signupNewUser** endpoint or with a method called **createUserWithEmailAndPassword**() when using *firebase_auth* plugin for Flutter.
- You can **sign in a user with an email and password** by issuing an HTTP POST request to the Firebase Auth REST API **verifyPassword** endpoint or with **signInWithEmailAndPassword**() when using *firebase_auth* plugin for Flutter.
- Before using *firebase_auth*, you must first have ensured you have initialized *FlutterFire*.

```
https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=[API_KEY]
```

API_KEY refers to the Web API Key, which can be obtained on the project settings page in your admin console.

ProxyProvider

- **ProxyProvider** is a provider that builds a value based on other providers.
- The exposed value is built through either **create** or **update**, then passed to *InheritedProvider*.
- As opposed to **create**, **update** may be called more than once.
- It will be called once the first time the value is obtained, then once whenever **ProxyProvider** rebuilds or when one of the providers it depends on updates.
- **ProxyProvider** comes in different variants such as **ProxyProvider2**.
  This is syntax sugar on the top of **ProxyProvider0**.

As such, `ProxyProvider<A, Result>` is equal to:

```
ProxyProvider0<Result>(
  update: (context, result) {
    final a = Provider.of<A>(context);
    return update(context, a, result);
  }
);
```

Whereas `ProxyProvider2<A, B, Result>` is equal to:

```
ProxyProvider0<Result>(
  update: (context, result) {
    final a = Provider.of<A>(context);
    final b = Provider.of<B>(context);
    return update(context, a, b, result);
  }
);
```

This last parameter of `update` is the last value returned by either `create` or `update`. It is `null` by default.

`update` must not be `null`.

ChangeNotifierProxyProvider

- ChangeNotifierProxyProvider is a ChangeNotifierProvider that builds and synchronizes a ChangeNotifier with external values.

  ChangeNotifierProxyProvider
  ({Key? key, required Create<R> create, required ProxyProviderBuilder<T, R> update,
  bool? lazy, TransitionBuilder? builder, Widget? Child}) - Initializes **key** for subclasses.

- First, create the **_proxy_** object, the one that you'll use in your UI.
- Next, define a function to be called on `**update**`. It will return the same type as the create method.

```
ChangeNotifierProvider(
  create: (context) {
    return MyChangeNotifier(
      myModel: Provider.of<MyModel>(context, listen: false),
    );
  },
  child: ...
)
```

```
ChangeNotifierProxyProvider<MyModel, MyChangeNotifier>(
  create: (_) => MyChangeNotifier(),
  update: (_, myModel, myNotifier) => myNotifier
    ..update(myModel),
  child: ...
);
```

A typical implementation of such `MyChangeNotifier` could be:

```
class MyChangeNotifier with ChangeNotifier {
  void update(MyModel myModel) {
    // Do some custom work based on myModel that may call `notifyListeners`
  }
}
```

- **DON'T** create the ChangeNotifier inside `update` directly.
  This will cause your state to be lost when one of the values used updates. It will also cause unnecessary overhead because it will dispose the previous notifier, then subscribes to the new one.

Instead reuse the previous instance, and update some properties or call some methods.

```
ChangeNotifierProxyProvider<MyModel, MyChangeNotifier>(
  // may cause the state to be destroyed involuntarily
  update: (_, myModel, myNotifier) => MyChangeNotifier(myModel: myModel),
  child: ...
);
```

- **PREFER** using ProxyProvider when possible.
  If the created object is only a combination of other objects, without http calls or similar side-effects, then it is likely that an immutable object built using ProxyProvider will work.

Timer Class

- **Timer** is a class that represents a count-down timer that is configured to trigger an action once end of time is reached, and it can fire **once** or **repeatedly**.

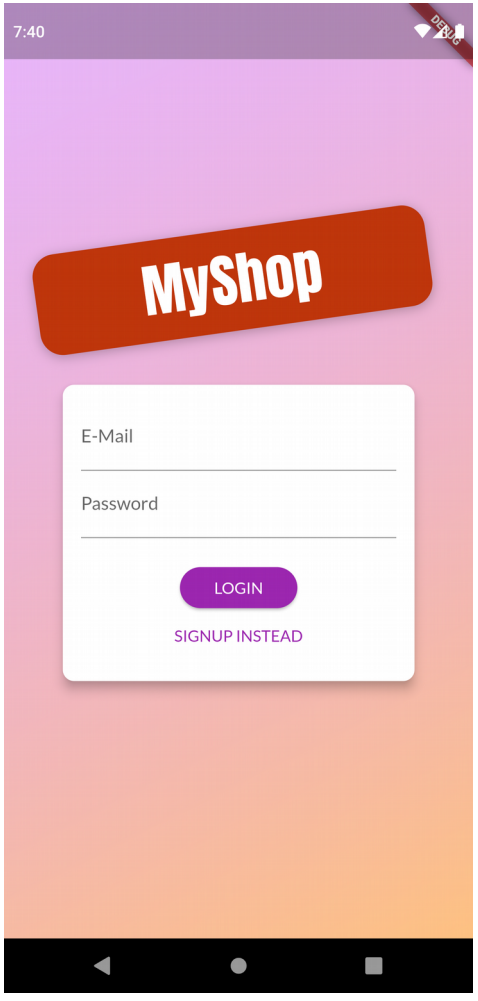
  Timer(Duration duration, void callback()) - Creates a new timer.
  Timer.periodic(Duration duration, void callback(Timer timer)) - Creates a new repeating timer.

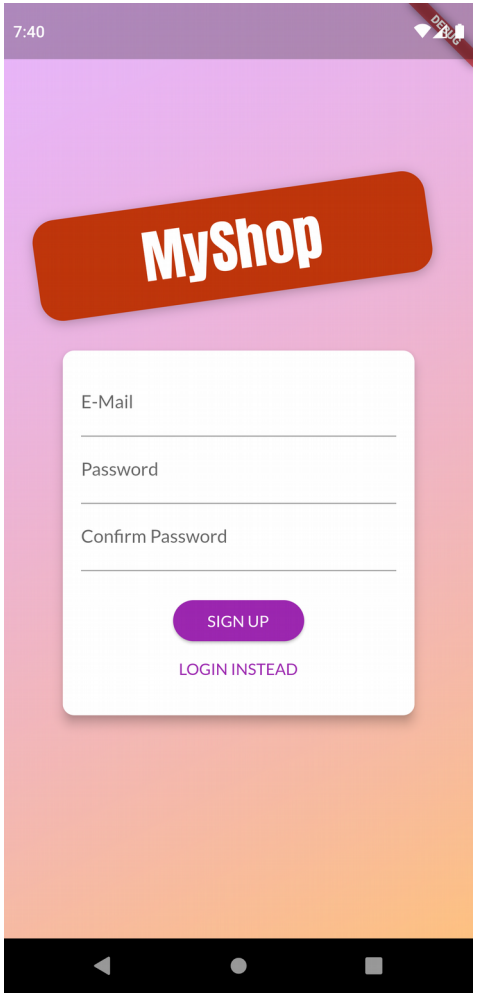
- The callback function of **Timer** is being executed asynchronously, it doesn't block the main thread.
- **The timer** counts down from the specified duration to 0. When the timer reaches 0, the timer invokes the specified callback function.
- Use a **periodic timer** to repeatedly count down the same interval.
- A negative duration is treated the same as a duration of 0. If the duration is statically known to be 0, consider using **run**.
- To stop timer when it reaches some certain expectations, call timer.**cancel**().

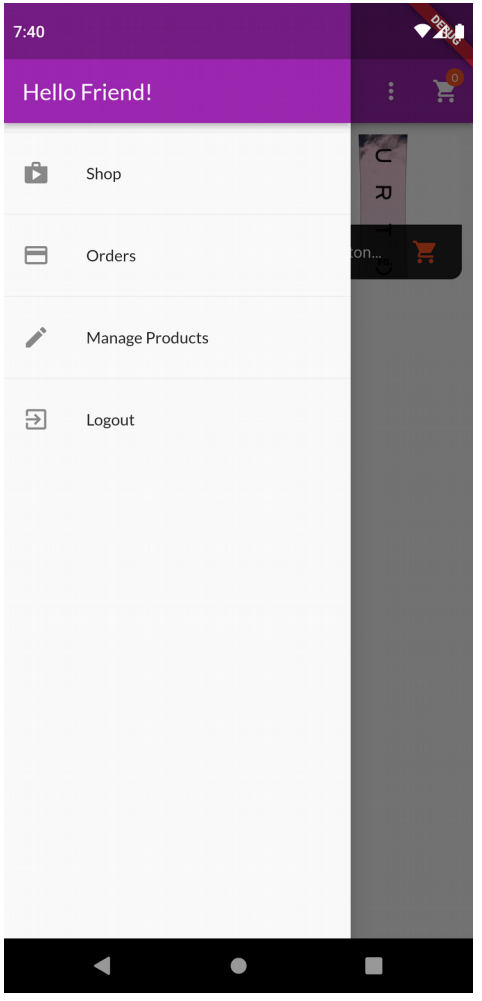
```dart
void main() {
  scheduleTimeout(5 * 1000); // 5 seconds.
}

Timer scheduleTimeout([int milliseconds = 10000]) =>
    Timer(Duration(milliseconds: milliseconds), handleTimeout);

void handleTimeout() {  // callback function
  // Do some work.
}
```

Note: If Dart code using Timer is compiled to JavaScript, the finest granularity available in the browser is 4 milliseconds.

Exercise – Improving Shop Application

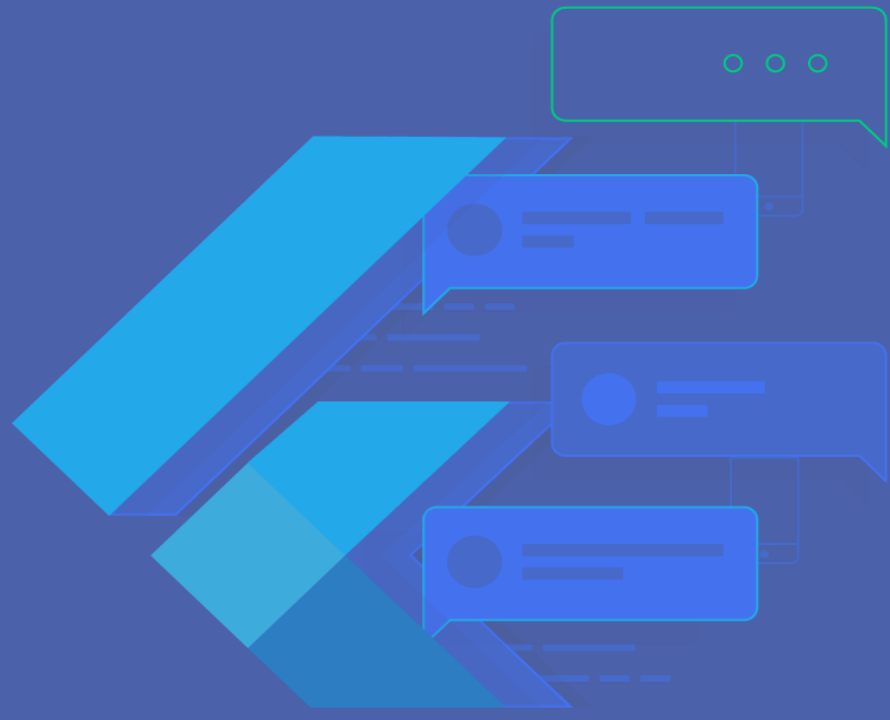






Questions ?

Persisting Data Locally in Flutter



Reading and Writing Values using SharedPreferences/Sembast/SQLite/Writing to Files

University American College Skopje

Welcome

- Store key-value data on disk
- Read and Write Files
- Persist data with SQLite
- Sembast
- Other Persisting Storage Packages
- Exercises: Persist data with SharedPreferences, Files, SQLite and Sembast

Store key-value data on disk

- If you have a relatively small collection of key-values to save, you can use the **shared_preferences** plugin.
- Normally, you would have to write native platform integrations for storing data on both iOS and Android.
- Fortunately, the **shared_preferences** plugin can be used to persist key-value data on disk.
- The **shared preferences** plugin wraps *NSUserDefaults* on **iOS** and *SharedPreferences* on **Android**, providing a persistent store for simple data.
- Data may be persisted to disk asynchronously, and there is no guarantee that writes will be persisted to disk after returning, so this plugin must not be used for storing critical data.
- Supported data types are **int**, **double**, **bool**, **String** and **List<String>**.

SharedPreferences

- To use this plugin, add **shared_preferences** as a dependency in your **pubspec.yaml** file.
- To persist data, use the setter methods provided by the SharedPreferences class. Setter methods are available for various primitive types, such as **setInt**, **setBool**, and **setString**.
- **Setter methods** do two things:
  First, synchronously update the key-value pair in-memory. Then, persist the data to disk.
- To read data, use the appropriate **getter method** provided by the SharedPreferences class. For each **setter** there is a corresponding **getter**. For example, you can use the **getInt**, **getBool**, and **getString** methods.
- To delete data, use the **remove**() method.
- Although key-value storage is easy and convenient to use, it has limitations:

  - Only primitive types can be used: **int**, **double**, **bool**, **string**, and **stringList**.

  - It's not designed to store a lot of data.

- Save data

```
// obtain shared preferences
final prefs = await SharedPreferences.getInstance();

// set value
prefs.setInt('counter', counter);
```

- Read data

```
final prefs = await SharedPreferences.getInstance();

// Try reading data from the counter key. If it doesn't exist, return 0.
final counter = prefs.getInt('counter') ?? 0;
```

- Remove data

```
final prefs = await SharedPreferences.getInstance();

prefs.remove('counter');
```

Read and write files

- In some cases, you need to **read and write files to disk**. For example, you may need to persist data across app launches, or download data from the internet and save it for later offline use.
- **To save files to disk**, combine the **path_provider** plugin with the **dart:io** library.
- The **path_provider** package provides a platform-agnostic way to access commonly used locations on the device's file system.
- The plugin currently supports access to two file system locations:

  1. **Temporary** and

  2. **Documents directory**

- Once you know where to store the file, you can use the **File** class from the **dart:io** library to create a reference to the file's full location.

File Class

- **A File** holds a path on which operations can be performed. You can get the parent directory of the file using parent, a property inherited from FileSystemEntity.
- You can create a new **File** object with a pathname to access the specified file on the file system from your program.
- The **File class** contains methods for manipulating files and their contents. Using methods in this class, you can open and close files, read to and write from them, create and delete them, and check for their existence.
- When **reading** or **writing a file**, you can use streams (with **openRead**), random access operations (with **open**), or convenience methods such as **readAsString**,
- Most methods in this class occur in synchronous and asynchronous pairs, for example, **readAsString** and **readAsStringSync**. Unless you have a specific reason for using the synchronous version of a method, prefer the asynchronous version to avoid blocking your program.
- If path is a symbolic **link**, rather than a file, then the methods of **File** operate on the ultimate target of the link, **except** for **delete** and **deleteSync**, which operate on the link.

Read from a file

```dart
import 'dart:async';
import 'dart:io';

void main() {
  File('file.txt').readAsString().then((String contents) {
    print(contents);
  });
}
```

```dart
import 'dart:io';
import 'dart:convert';
import 'dart:async';

void main() async {
  final file = File('file.txt');
  Stream<String> lines = file.openRead()
    .transform(utf8.decoder)       // Decode bytes to UTF-8.
    .transform(LineSplitter());    // Convert stream to individual lines.
  try {
    await for (var line in lines) {
      print('$line: ${line.length} characters');
    }
    print('File is now closed.');
  } catch (e) {
    print('Error: $e');
  }
}
```

Write to a file

```dart
import 'dart:io';

void main() async {
  final filename = 'file.txt';
  var file = await File(filename).writeAsString('some content');
  // Do something with the file.
}
```

```dart
import 'dart:io';

void main() {
  var file = File('file.txt');
  var sink = file.openWrite();
  sink.write('FILE ACCESSED ${DateTime.now()}\n');

  // Close the IOSink to free system resources.
  sink.close();
}
```

Persist data with SQLite

- If you are writing an app that needs to persist and query large amounts of data on the local device, consider using a **database** instead of a local file or key-value store.
- In general, databases provide **faster inserts**, **updates**, and **queries** compared to other local persistence solutions.
- Flutter apps can make use of the **SQLite** databases via the **sqflite** plugin available on **pub.dev**.
- The API is largely inspired from Android ContentProvider where a typical SQLite implementation means opening the database once on the first request and keeping it open.
- If you don't use singleInstance, keeping a reference (at the app level or in a widget) can cause issues with hot reload if the reference is lost (and the database not closed yet).

Sqflite

- **Sqflite** provides a basic location strategy using the databases path on Android and the Documents folder on iOS, as recommended on both platform.
- **A SQLite database** is a **file** in the file system identified by a path. The location can be retrieved using getDatabasesPath.
- If relative, this path is relative to the path obtained by **getDatabasesPath**(), which is the default database directory on **Android** and the documents directory on **iOS/MacOS**.
- Opening a database in **read-write mode** is the default. One can specify a version to perform migration strategy, can configure the database and its version.
- It is strongly suggested to **open a database only once**. By default a database is open as a single instance (singleInstance: true). i.e. re-opening the same file is safe and it will give you the same database.
- You might want to **preload** your database when opened the first time. You can either

  - Import an existing SQLite file checking first whether the database file exists or not

  - Populate data during onCreate

Get Database Path

```
// Check if we have an existing copy first
var databasesPath = await getDatabasesPath();
String path = join(databasesPath, "demo_asset_example.db");
```

Open a Database

```
// try opening (will work if it exists)
Database db;
try {
  db = await openDatabase(path, readOnly: true);
} catch (e) {
  print("Error $e");
}
```

Create a Database Table

```
var db = await openDatabase(path);
await db.execute("CREATE TABLE IF NOT EXISTS Test(id INTEGER PRIMARY KE
Y)");
```

Insert into a database table

```
await db.insert("Test", {"id": 1});
```

Query or Select

```
await db.query("Test")
```

Count number of items

```
Future<int> getCount() async {
  Database db = await this.db;
  var result = Sqflite.firstIntValue(
    await db.rawQuery("SELECT COUNT (*) FROM $tblTodo")
  );
  return result;
}
```

Update Data

```
Future<int> updateTodo(Todo todo) async {
  var db = await this.db;
  var result = await db.update(tblTodo, todo.toMap(),
    where: "$colId = ?", whereArgs: [todo.id]);
  return result;
}
```

Delete From SQLite database

```
Future<int> deleteTodo(int id) async {
  int result;
  var db = await this.db;
  result = await db.rawDelete('DELETE FROM $tblTodo WHERE $colId = $id');
  return result;
}
```

Close Database

```
    await db.close();
```

Sembast

- As preferred by name **Sembast** db stands for **S**imple **Emb**edded **A**pplication **St**ore database.
- Sembast is **NoSQL** persistent store database solution for single process **io** applications.
- The whole document based database resides in a single file and is loaded in memory when opened. Changes are appended right away to the file and the file is automatically compacted when needed.
- Works on **Dart VM** and **Flutter** (no plugin needed, 100% Dart so works on all platforms - MacOS/Android/iOS/Linux/Windows).
- **Sembast** also supports encryption using user-defined codec but the thing which makes it different among all NoSQL databases is its working compatibility with de-normalized data as you know that it is just a **JSON file** and viewing and accessing the content from the database is easy as compared to others.

Other Persisting Storage Packages

flutter_secure_storage

Null safety 1617

Flutter Secure Storage provides API to store data in secure storage. Keychain is used in iOS, KeyStore based solution is used in Android.

get_storage

Null safety 822

A fast, extra light and synchronous key-value storage written entirely in Dart

drift

Null safety 357

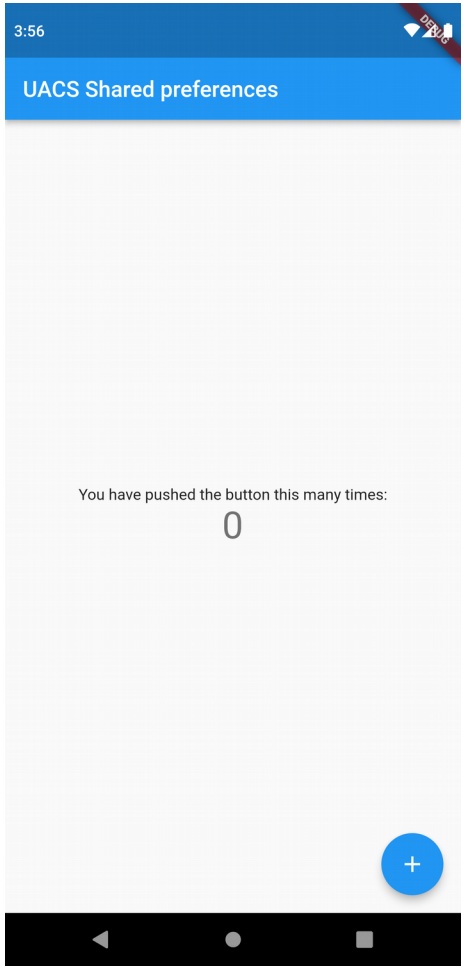
Drift is a reactive library to store relational data in Dart and Flutter applications.

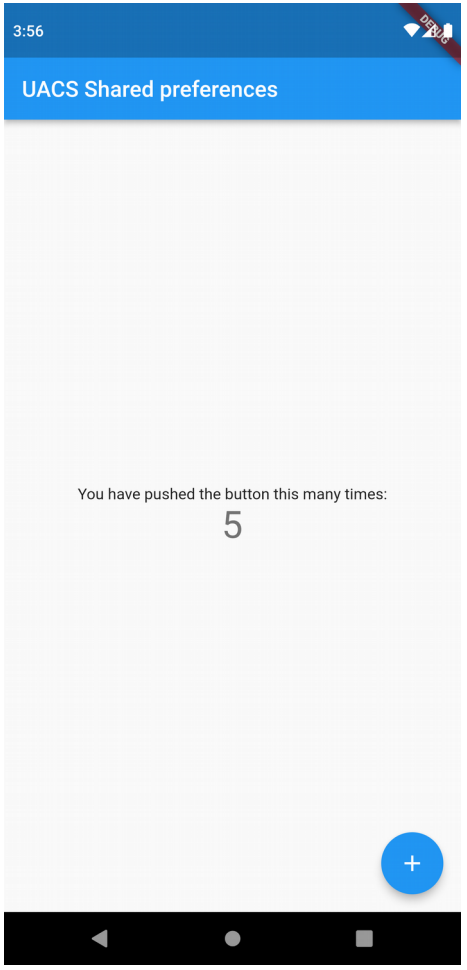
hive

Null safety 3095

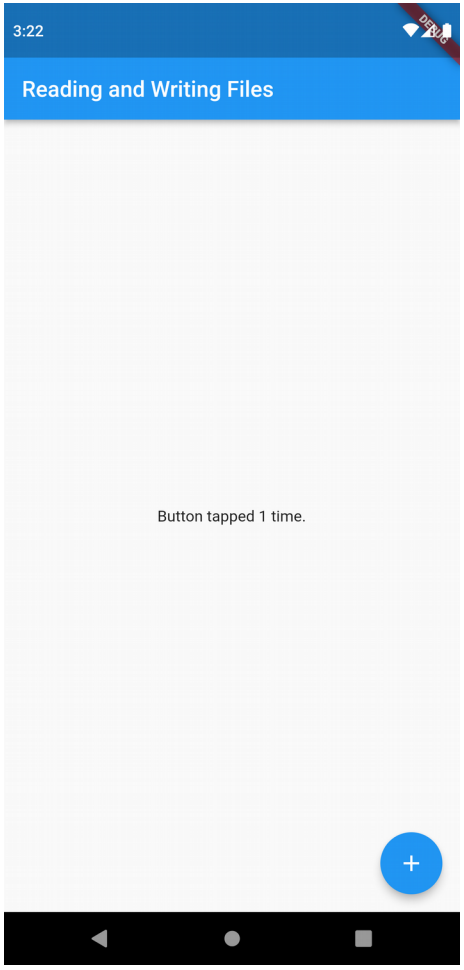
Lightweight and blazing fast key-value database written in pure Dart. Strongly encrypted using AES-256.

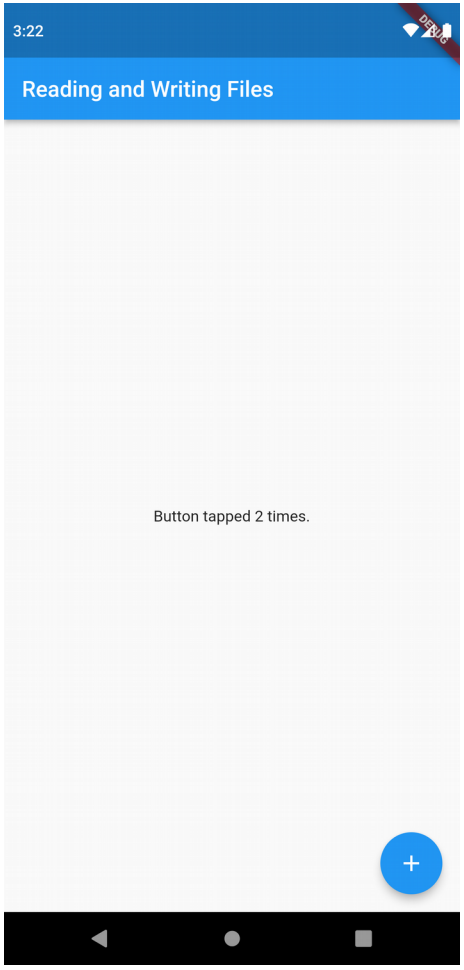
Exercise – Using SharedPreferences





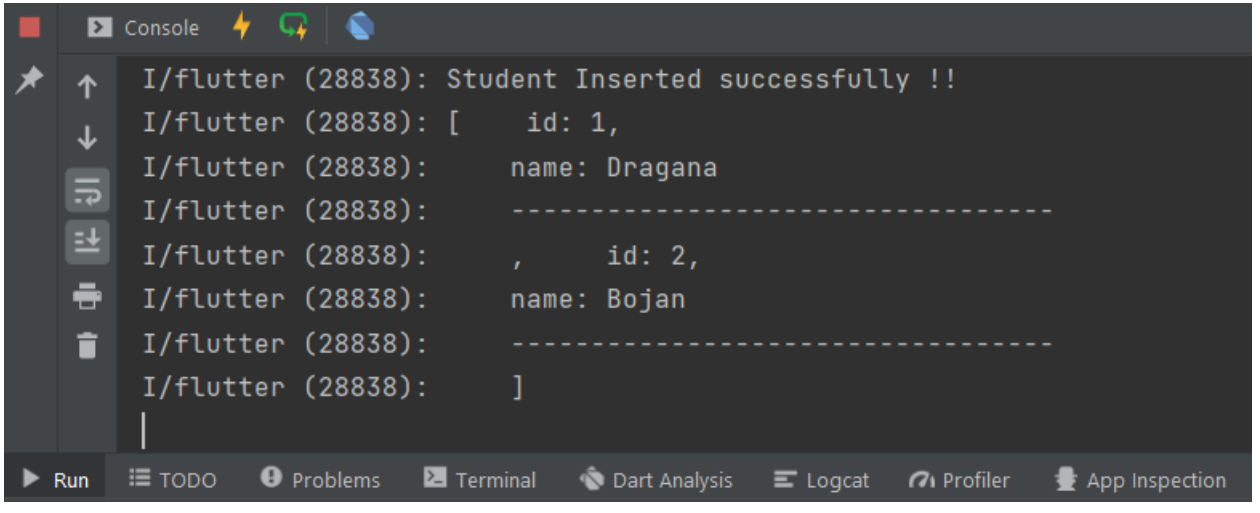
Exercise – Using Files

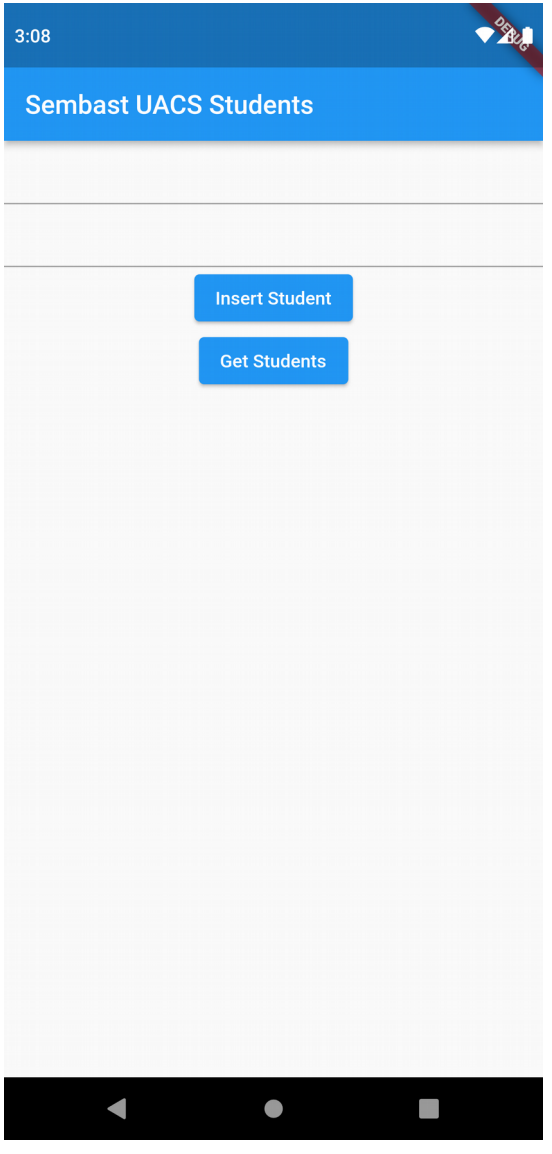




Exercise – Using SQLite

Exercise – Using Sembast





Questions ?