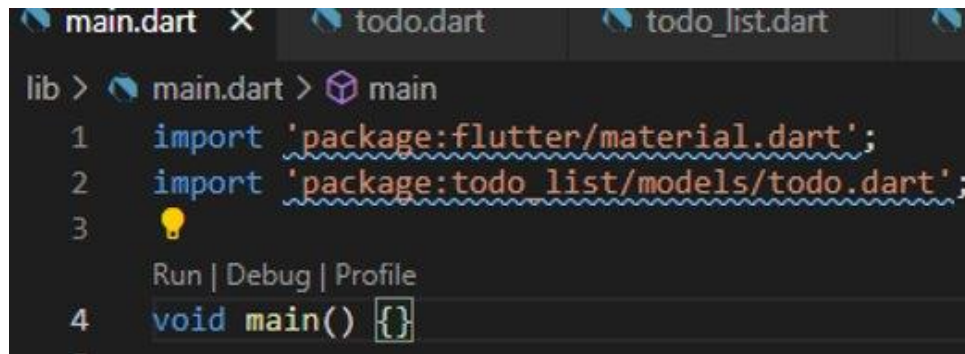


Base App Structure

Building the core application functionality for the Todo App

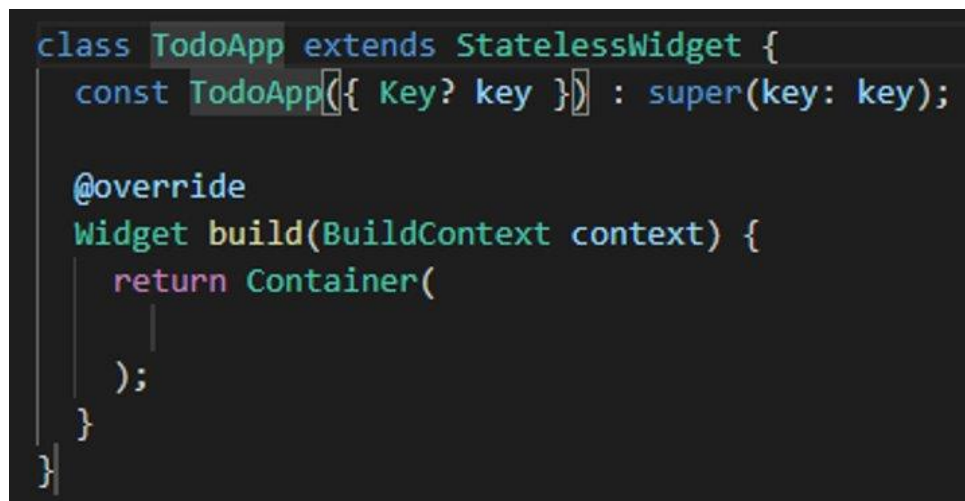
Part 1: Declutter the flutter

Start by opening VSCode and creating a new flutter project using the flutter create command. Call your app todo_list. Remove the templated counter app code from the main.dart file, leaving the main() entry point that dart requires, it should look like this:



```
lib > main.dart > main
1 import 'package:flutter/material.dart';
2 import 'package:todo_list/models/todo.dart';
3
Run | Debug | Profile
4 void main() {}
```

For flutter we need to initialise the flutter application by running a built-in command runApp(). To this we pass our main app, which is a widget we create. The easiest way to do this with VS Code is to type "st" which should open an intellisense window with options. The second item should be stateless widget, press enter to insert the snippet. Now enter the name of your app: TodoApp. We can call runApp(const TodoApp()); in main to launch the application.



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

  @override
  Widget build(BuildContext context) {
    return Container(
      );
  }
}
```

To create our application, we're going to use the widget called Material app it sets up the theming and lets us set a home page (which we're going to create). Replace Container() with material app and press return.

Note: Remember to use the Ctrl + Space to get a helper of what parameters can be passed to this material app constructor.

Add title & home to the parameters. Home accepts the page (Still a widget) we are going to display as our homepage. Enter a string in the title for the app.

Home Page

Now we need to create our Stateful Widget for the home page. This is going to be managing its own internal state through the built-in `setState()` function. Like we did before, create a new stateful widget this time called `TodoHomePage`. Stateful widgets are split up into two parts, the widget and the state. The state controls the build method and any dynamic properties that might define or influence its state. As the app is rebuilt it calls the state to determine what to display.

Now go back to our `MaterialApp` widget and add the `TodoHomePage` into its `home` parameter. (You may need to apply `const` appropriately here)

Now you should have the following elements:

- Main with `runApp()` function passing in our stateless `TodoApp` widget.
- `TodoApp` that builds a material app that takes a title and a homepage of a stateful `TodoHomePage` widget.
- Inside that then we finally have our build method that will display the widgets we want to build on our page.
- Replace the build methods container with a scaffold with an app bar that has a title of a `Text` Widget.
- Test that we have implemented this correctly by running the app and seeing if it launches correctly.

Todo Class

To the project inside the `lib` folder create a new folder called `models`. This is going to hold our `Todo` class. Create a new class by creating a new file do not forget to name it appropriately, including the `.dart` extension. Open the file and create a class inside called `Todo`. Create a constructor and three properties for `name`, `description` and `complete`. Finally, we are going to override an inbuilt `toString` method. Type `String` and press space and the override will pop up, press enter. This will add the `@override` annotation for us. Remove the default code and return a string that is the name and the description in a single string.

```
class Todo {
  final String name;
  final String description;
  final bool complete;

  Todo({required this.name, required this.description, this.complete = false});

  @override
  String toString() {
    //return name + " (" + description + ") "; old operation on string
    return "$name - ($description) "; //String interpolation
  }
}
```

Part 2: Todo UI

Reviewing the core counter app code, we can see that when we want the internal state (and therefore the view) to refresh we call a `setState()` function. This function tells flutter to reload the UI build methods and determine any changes it needs to make. However, let us say we have a list of items that we want to update, the UI will need to respond to an added item. To do this we use a builder to infer the collection of widgets using the number of items.

Inside the `_TodoHomePageState` class, add a final list of todos like this:

```
final List<Todo> todos = <Todo>[
  Todo(name: "Shopping", description: "Pick up groceries"),
  Todo(name: "Paint", description: "Recreate the Mona Lisa"),
  Todo(name: "Dance", description: "I wanna dance with somebody"),
]; // <Todo>[]
```

This will give us some temporary/fake data to work with in our application. Now we can start building our UI using an `ItemBuilder`. In the body of the scaffold, add a new centre widget. In its child we are going to create a listview by dynamically calling the `ListView.builder()` function. This takes a few parameters that will allow us to generate a template style widget. For now, we're going to just output the list. The builder function takes a few options, but the once we are going to add are the `itemCount` and `itemBuilder`.

`ItemCount`: tells the `ListView.Builder()` how many iterations

Inside the `itemBuilder` parameter, we need to pass a function that flutter uses to build the UI elements. It returns the widgets by looping over a collection.

`ItemBuilder` function:

```
child: ListView.builder(
  itemCount: todos.length,
  itemBuilder: (BuildContext context, int i) {
    return Container(
      padding: const EdgeInsets.all(5),
      child: Text(todos[i].toString()),
    ); // Container
  }), // ListView.builder
```

Exercise

Where the returned container contains the basic todo as a string. Update this returned widget to be more visually appealing using the data from the todo model. Think of the information you have

available and how you may want to present this. I recommend having a design in mind before starting. Use a combination of widgets to create this design.

Recommended widgets:

- Box Decoration
- Box Shadow
- Colours
- Margin/Padding: `EdgeInsets.all()`

Feel free to apply other styles throughout the application customising other widgets.

Part 3: Adding A Todo

Floating Action Button

In the example counter project, we had a floating action button that represented the counter action. We're going to use this to create the add button that's always available.

In the scaffold look for the `floatingActionButton` parameter. Add a new floating action button that takes three parameters:

- `OnPressed` – The event handler that will be called when the button is pressed.
- `Tooltip` – simple text. When hovered over this pops up and is displayed to the user.
- `Child` – The widget elements that represent the button to be shown.

The `OnPressed()` is an event handler that can accept an anonymous function, call-back or the name of a scoped function that exists.

Inside the state object create a new function that will be our add Todo function. Call it:

`_openAddTodo`

This doesn't need a return type. The underscore implying this is a private function. It does not require any input parameters.

Add the function to the on pressed parameter without the parenthesis so you're referencing the function itself.

```
floatingActionButton: FloatingActionButton(  
  onPressed: _openAddTodo,
```

Note:

- *Calling the function: `_openAddTodo()`*
- *Passing Reference to function: `_openAddTodo`*

Show Dialog

To easily present a pop up we can use the `ShowDialog()` function. It takes a context and a builder (like our list builder) which expects a widget to be returned. For the popup we'll return an `AlertDialog` widget. Create this widget and in its content create a column with children that represent the fields for the Todo data. To position this Column, set its `mainAxisSize` to `MainAxisSize.min`. In the children we can lay out our visual elements from top to bottom.

This is created by using `TextFormField` using a controller which handles the editing.

At the top of the **`TodoHomePageState`** class add the two controllers which are the class: `TextEditingController` with appropriate names:

```
final TextEditingController _controlName = TextEditingController();
final TextEditingController _controlDescription = TextEditingController();
```

Add these controllers to our form ensuring they match the correct one.

```
const Padding(
  padding: EdgeInsets.fromLTRB(5, 8, 5, 0),
  child: Text("Name"),
), // Padding
Padding(
  padding: const EdgeInsets.fromLTRB(5, 0, 5, 8),
  child: TextFormField(
    controller: _controlName,
  ), // TextFormField
), // Padding
```

Getting the inputs

Inside the `onPressed`: of the elevated button create an anonymous function with a body:

```
() {}
```

Call the special function `setState()` which can then call the `todos.add()` to add a new todo. Create the new todo using the controllers created previously fetching the `controller.text` property.

`Todo(name: <controller>.text, description: <controller>.text)`

Don't forget to finally call: `Navigator.pop(context)` to close the pop up once this has been completed.

```
child: ElevatedButton(  
  child: const Text("Submit"),  
  onPressed: () {  
    setState(() {  
      todos.add(Todo(  
        name: _controlName.text,  
        description: _controlDescription.text)); // Todo  
      });  
    Navigator.pop(context);  
  }), // ElevatedButton
```