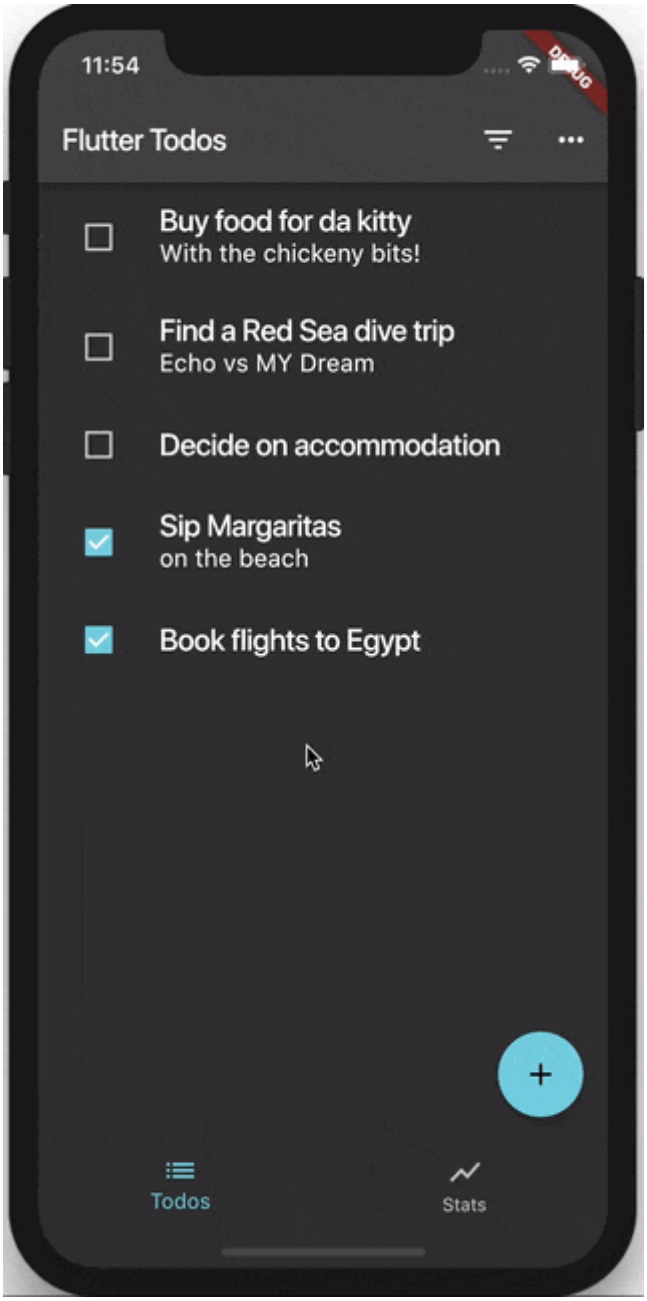
Flutter Todos Tutorial

level advanced

> In the following tutorial, we're going to build a Todos App in Flutter using the Bloc library.



Key Topics

- Observe state changes with BlocObserver.
- BlocProvider, Flutter widget which provides a bloc to its children.
- BlocBuilder, Flutter widget that handles building the widget in response to new states.
- Using Bloc instead of Cubit. What's the difference?
- Prevent unnecessary rebuilds with Equatable.
- MultiBlocProvider, a Flutter widget that merges multiple BlocProvider widgets into one.

Setup

We'll start off by creating a brand new Flutter project

```bash
flutter create flutter_todos
```

We can then replace the contents of `pubspec.yaml` with

```yaml
name: flutter_todos
description: A new Flutter project.

environment:
  sdk: ">=2.6.0 <3.0.0"

dependencies:
  meta: ^1.1.6
  equatable: ^1.0.0
  flutter_bloc: ^5.0.0
  flutter:
    sdk: flutter

dependency_overrides:
  todos_app_core:
    git:
      url: https://github.com/felangel/flutter_architecture_samples
      path: todos_app_core
      ref: rxdart/0.23.0
```

```
  todos_repository_core:
    git:
      url: https://github.com/felangel/flutter_architecture_samples
      path: todos_repository_core
      ref: rxdart/0.23.0
  todos_repository_simple:
    git:
      url: https://github.com/felangel/flutter_architecture_samples
      path: todos_repository_simple
      ref: rxdart/0.23.0

flutter:
  uses-material-design: true
```

and then install all of the dependencies

```bash
flutter packages get
```

Note: We're overriding some dependencies because we're going to be reusing them from Brian Egan's Flutter Architecture Samples.

App Keys

Before we jump into the application code, let's create `flutter_todos_keys.dart`. This file will contain keys which we will use to uniquely identify important widgets. We can later write tests that find widgets based on keys.

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

class FlutterTodosKeys {
  static final extraActionsPopupMenuButton =
      const Key('__extraActionsPopupMenuButton__');
  static final extraActionsEmptyContainer =
      const Key('__extraActionsEmptyContainer__');
  static final filteredTodosEmptyContainer =
      const Key('__filteredTodosEmptyContainer__');
```

```
  static final statsLoadInProgressIndicator = const Key('__statsLoadInPro
  static final emptyStatsContainer = const Key('__emptyStatsContainer__')
  static final emptyDetailsContainer = const Key('__emptyDetailsContainer
  static final detailsScreenCheckBox = const Key('__detailsScreenCheckBox
}
```

We will reference these keys throughout the rest of the tutorial.

> **Note:** You can check out the integration tests for the application here. You can also check out unit and widget tests here.

Localization

One last concept that we will touch on before going into the application itself is localization. Create `localization.dart` and we'll create the foundation for multi-language support.

```dart
import 'dart:async';

import 'package:flutter/material.dart';

class FlutterBlocLocalizations {
  static FlutterBlocLocalizations of(BuildContext context) {
    return Localizations.of<FlutterBlocLocalizations>(
      context,
      FlutterBlocLocalizations,
    );
  }

  String get appTitle => "Flutter Todos";
}

class FlutterBlocLocalizationsDelegate
    extends LocalizationsDelegate<FlutterBlocLocalizations> {
  @override
  Future<FlutterBlocLocalizations> load(Locale locale) =>
      Future(() => FlutterBlocLocalizations());
```

```
  @override
  bool shouldReload(FlutterBlocLocalizationsDelegate old) => false;

  @override
  bool isSupported(Locale locale) =>
      locale.languageCode.toLowerCase().contains("en");
}
```

We can now import and provide our `FlutterBlocLocalizationsDelegate` to our `MaterialApp` (later in this tutorial).

For more information on localization check out the official flutter docs.

Todos Repository

In this tutorial we're not going to go into the implementation details of the `TodosRepository` because it was implemented by Brian Egan and is shared among all of the Todo Architecture Samples. At a high level, the `TodosRepository` will expose a method to `loadTodos` and to `saveTodos`. That's pretty much all we need to know so for the rest of the tutorial we'll focus on the Bloc and Presentation layers.

Todos Bloc

> Our `TodosBloc` will be responsible for converting `TodosEvents` into `TodosStates` and will manage the list of todos.

Model

The first thing we need to do is define our `Todo` model. Each todo will need to have an id, a task, an optional note, and an optional completed flag.

Let's create a `models` directory and create `todo.dart`.

```dart
import 'package:todos_app_core/todos_app_core.dart';
import 'package:equatable/equatable.dart';
```

```dart
import 'package:todos_repository_core/todos_repository_core.dart';

class Todo extends Equatable {
  final bool complete;
  final String id;
  final String note;
  final String task;

  Todo(
    this.task, {
    this.complete = false,
    String note = '',
    String id,
  })  : this.note = note ?? '',
        this.id = id ?? Uuid().generateV4();

  Todo copyWith({bool complete, String id, String note, String task}) {
    return Todo(
      task ?? this.task,
      complete: complete ?? this.complete,
      id: id ?? this.id,
      note: note ?? this.note,
    );
  }

  @override
  List<Object> get props => [complete, id, note, task];

  @override
  String toString() {
    return 'Todo { complete: $complete, task: $task, note: $note, id: $id
  }

  TodoEntity toEntity() {
    return TodoEntity(task, id, note, complete);
  }

  static Todo fromEntity(TodoEntity entity) {
    return Todo(
      entity.task,
      complete: entity.complete ?? false,
      note: entity.note,
      id: entity.id ?? Uuid().generateV4(),
```

```
      );
    }
  }
```

Note: We're using the **Equatable** package so that we can compare instances of `Todos` without having to manually override `==` and `hashCode`.

Next up, we need to create the `TodosState` which our presentation layer will receive.

States

Let's create `blocs/todos/todos_state.dart` and define the different states we'll need to handle.

The three states we will implement are:

- `TodosLoadInProgress` - the state while our application is fetching todos from the repository.
- `TodosLoadSuccess` - the state of our application after the todos have successfully been loaded.
- `TodosLoadFailure` - the state of our application if the todos were not successfully loaded.

dart

```dart
import 'package:equatable/equatable.dart';
import 'package:flutter_todos/models/models.dart';

abstract class TodosState extends Equatable {
  const TodosState();

  @override
  List<Object> get props => [];
}

class TodosLoadInProgress extends TodosState {}

class TodosLoadSuccess extends TodosState {
  final List<Todo> todos;
```

```
  const TodosLoadSuccess([this.todos = const []]);

  @override
  List<Object> get props => [todos];

  @override
  String toString() => 'TodosLoadSuccess { todos: $todos }';
}

class TodosLoadFailure extends TodosState {}
```

Next, let's implement the events we will need to handle.

Events

The events we will need to handle in our `TodosBloc` are:

- `TodosLoadSuccess` - tells the bloc that it needs to load the todos from the `TodosRepository`.
- `TodoAdded` - tells the bloc that it needs to add a new todo to the list of todos.
- `TodoUpdated` - tells the bloc that it needs to update an existing todo.
- `TodoDeleted` - tells the bloc that it needs to remove an existing todo.
- `ClearCompleted` - tells the bloc that it needs to remove all completed todos.
- `ToggleAll` - tells the bloc that it needs to toggle the completed state of all todos.

Create `blocs/todos/todos_event.dart` and let's implement the events we described above.

dart

```
import 'package:equatable/equatable.dart';
import 'package:flutter_todos/models/models.dart';

abstract class TodosEvent extends Equatable {
  const TodosEvent();

  @override
  List<Object> get props => [];
}
```

```dart
class TodosLoadSuccess extends TodosEvent {}

class TodoAdded extends TodosEvent {
  final Todo todo;

  const TodoAdded(this.todo);

  @override
  List<Object> get props => [todo];

  @override
  String toString() => 'TodoAdded { todo: $todo }';
}

class TodoUpdated extends TodosEvent {
  final Todo todo;

  const TodoUpdated(this.todo);

  @override
  List<Object> get props => [todo];

  @override
  String toString() => 'TodoUpdated { todo: $todo }';
}

class TodoDeleted extends TodosEvent {
  final Todo todo;

  const TodoDeleted(this.todo);

  @override
  List<Object> get props => [todo];

  @override
  String toString() => 'TodoDeleted { todo: $todo }';
}

class ClearCompleted extends TodosEvent {}

class ToggleAll extends TodosEvent {}
```

Now that we have our `TodosStates` and `TodosEvents` implemented we can implement our `TodosBloc`.

Bloc

Let's create `blocs/todos/todos_bloc.dart` and get started! We just need to set the initial state and implement `mapEventToState`.

```dart
import 'dart:async';
import 'package:bloc/bloc.dart';
import 'package:meta/meta.dart';
import 'package:flutter_todos/blocs/todos/todos.dart';
import 'package:flutter_todos/models/models.dart';
import 'package:todos_repository_simple/todos_repository_simple.dart';

class TodosBloc extends Bloc<TodosEvent, TodosState> {
  final TodosRepositoryFlutter todosRepository;

  TodosBloc({@required this.todosRepository}) : super(TodosLoadInProgress

  @override
  Stream<TodosState> mapEventToState(TodosEvent event) async* {
    if (event is TodosLoadSuccess) {
      yield* _mapTodosLoadedToState();
    } else if (event is TodoAdded) {
      yield* _mapTodoAddedToState(event);
    } else if (event is TodoUpdated) {
      yield* _mapTodoUpdatedToState(event);
    } else if (event is TodoDeleted) {
      yield* _mapTodoDeletedToState(event);
    } else if (event is ToggleAll) {
      yield* _mapToggleAllToState();
    } else if (event is ClearCompleted) {
      yield* _mapClearCompletedToState();
    }
  }

  Stream<TodosState> _mapTodosLoadedToState() async* {
    try {
      final todos = await this.todosRepository.loadTodos();
      yield TodosLoadSuccess(
```

```
        todos.map(Todo.fromEntity).toList(),
      );
    } catch (_) {
      yield TodosLoadFailure();
    }
  }

  Stream<TodosState> _mapTodoAddedToState(TodoAdded event) async* {
    if (state is TodosLoadSuccess) {
      final List<Todo> updatedTodos = List.from((state as TodosLoadSucces
        ..add(event.todo);
      yield TodosLoadSuccess(updatedTodos);
      _saveTodos(updatedTodos);
    }
  }

  Stream<TodosState> _mapTodoUpdatedToState(TodoUpdated event) async* {
    if (state is TodosLoadSuccess) {
      final List<Todo> updatedTodos = (state as TodosLoadSuccess).todos.m
        return todo.id == event.updatedTodo.id ? event.updatedTodo : todo
      }).toList();
      yield TodosLoadSuccess(updatedTodos);
      _saveTodos(updatedTodos);
    }
  }

  Stream<TodosState> _mapTodoDeletedToState(TodoDeleted event) async* {
    if (state is TodosLoadSuccess) {
      final updatedTodos = (state as TodosLoadSuccess)
          .todos
          .where((todo) => todo.id != event.todo.id)
          .toList();
      yield TodosLoadSuccess(updatedTodos);
      _saveTodos(updatedTodos);
    }
  }

  Stream<TodosState> _mapToggleAllToState() async* {
    if (state is TodosLoadSuccess) {
      final allComplete =
          (state as TodosLoadSuccess).todos.every((todo) => todo.complete
      final List<Todo> updatedTodos = (state as TodosLoadSuccess)
          .todos
```

```
            .map((todo) => todo.copyWith(complete: !allComplete))
            .toList();
        yield TodosLoadSuccess(updatedTodos);
        _saveTodos(updatedTodos);
      }
    }

    Stream<TodosState> _mapClearCompletedToState() async* {
      if (state is TodosLoadSuccess) {
        final List<Todo> updatedTodos =
            (state as TodosLoadSuccess).todos.where((todo) => !todo.complet
        yield TodosLoadSuccess(updatedTodos);
        _saveTodos(updatedTodos);
      }
    }

    Future _saveTodos(List<Todo> todos) {
      return todosRepository.saveTodos(
        todos.map((todo) => todo.toEntity()).toList(),
      );
    }
  }
```

> When we yield a state in the private `mapEventToState` handlers, we are always yielding a new state instead of mutating the `state`. This is because every time we yield, bloc will compare the `state` to the `nextState` and will only trigger a state change (`transition`) if the two states are **not equal**. If we just mutate and yield the same instance of state, then `state == nextState` would evaluate to true and no state change would occur.

Our `TodosBloc` will have a dependency on the `TodosRepository` so that it can load and save todos. It will have an initial state of `TodosLoadInProgress` and defines the private handlers for each of the events. Whenever the `TodosBloc` changes the list of todos it calls the `saveTodos` method in the `TodosRepository` in order to keep everything persisted locally.

Barrel File

Now that we're done with our `TodosBloc` we can create a barrel file to export all of our bloc files and make it convenient to import them later on.

Create `blocs/todos/todos.dart` and export the bloc, events, and states:

```dart
export './todos_bloc.dart';
export './todos_event.dart';
export './todos_state.dart';
```

Filtered Todos Bloc

> The `FilteredTodosBloc` will be responsible for reacting to state changes in the `TodosBloc` we just created and will maintain the state of filtered todos in our application.

Model

Before we start defining and implementing the `TodosStates`, we will need to implement a `VisibilityFilter` model that will determine which todos our `FilteredTodosState` will contain. In this case, we will have three filters:

- `all` - show all Todos (default)
- `active` - only show Todos which have not been completed
- `completed` only show Todos which have been completed

We can create `models/visibility_filter.dart` and define our filter as an enum:

```dart
enum VisibilityFilter { all, active, completed }
```

States

Just like we did with the `TodosBloc`, we'll need to define the different states for our `FilteredTodosBloc`.

In this case, we only have two states:

- `FilteredTodosLoadInProgress` - the state while we are fetching todos
- `FilteredTodosLoadSuccess` - the state when we are no longer fetching todos

Let's create `blocs/filtered_todos/filtered_todos_state.dart` and implement the two states.

```dart
import 'package:equatable/equatable.dart';
import 'package:flutter_todos/models/models.dart';

abstract class FilteredTodosState extends Equatable {
  const FilteredTodosState();

  @override
  List<Object> get props => [];
}

class FilteredTodosLoadInProgress extends FilteredTodosState {}

class FilteredTodosLoadSuccess extends FilteredTodosState {
  final List<Todo> filteredTodos;
  final VisibilityFilter activeFilter;

  const FilteredTodosLoadSuccess(
    this.filteredTodos,
    this.activeFilter,
  );

  @override
  List<Object> get props => [filteredTodos, activeFilter];

  @override
  String toString() {
    return 'FilteredTodosLoadSuccess { filteredTodos: $filteredTodos, act
  }
}
```

Note: The `FilteredTodosLoadSuccess` state contains the list of filtered todos as well as the active visibility filter.

Events

We're going to implement two events for our `FilteredTodosBloc` :

- `FilterUpdated` - which notifies the bloc that the visibility filter has changed
- `TodosUpdated` - which notifies the bloc that the list of todos has changed

Create `blocs/filtered_todos/filtered_todos_event.dart` and let's implement the two events.

```dart
import 'package:equatable/equatable.dart';
import 'package:flutter_todos/models/models.dart';

abstract class FilteredTodosEvent extends Equatable {
  const FilteredTodosEvent();
}

class FilterUpdated extends FilteredTodosEvent {
  final VisibilityFilter filter;

  const FilterUpdated(this.filter);

  @override
  List<Object> get props => [filter];

  @override
  String toString() => 'FilterUpdated { filter: $filter }';
}

class TodosUpdated extends FilteredTodosEvent {
  final List<Todo> todos;

  const TodosUpdated(this.todos);

  @override
  List<Object> get props => [todos];

  @override
  String toString() => 'TodosUpdated { todos: $todos }';
}
```

We're ready to implement our `FilteredTodosBloc` next!

Bloc

Our `FilteredTodosBloc` will be similar to our `TodosBloc`; however, instead of having a dependency on the `TodosRepository`, it will have a dependency on the `TodosBloc` itself. This will allow the `FilteredTodosBloc` to update its state in response to state changes in the `TodosBloc`.

Create `blocs/filtered_todos/filtered_todos_bloc.dart` and let's get started.

```dart
import 'dart:async';
import 'package:bloc/bloc.dart';
import 'package:meta/meta.dart';
import 'package:flutter_todos/blocs/filtered_todos/filtered_todos.dart';
import 'package:flutter_todos/blocs/todos/todos.dart';
import 'package:flutter_todos/models/models.dart';

class FilteredTodosBloc extends Bloc<FilteredTodosEvent, FilteredTodosSta
  final TodosBloc todosBloc;
  StreamSubscription todosSubscription;

  FilteredTodosBloc({@required this.todosBloc})
      : super(
          todosBloc.state is TodosLoadSuccess
              ? FilteredTodosLoadSuccess(
                  (todosBloc.state as TodosLoadSuccess).todos,
                  VisibilityFilter.all,
                )
              : FilteredTodosLoadInProgress(),
        ) {
    todosSubscription = todosBloc.listen((state) {
      if (state is TodosLoadSuccess) {
        add(TodosUpdated((todosBloc.state as TodosLoadSuccess).todos));
      }
    });
  }

  @override
  Stream<FilteredTodosState> mapEventToState(FilteredTodosEvent event) as
    if (event is FilterUpdated) {
```

```
      yield* _mapUpdateFilterToState(event);
    } else if (event is TodosUpdated) {
      yield* _mapTodosUpdatedToState(event);
    }
  }

  Stream<FilteredTodosState> _mapUpdateFilterToState(
    FilterUpdated event,
  ) async* {
    if (todosBloc.state is TodosLoadSuccess) {
      yield FilteredTodosLoadSuccess(
        _mapTodosToFilteredTodos(
          (todosBloc.state as TodosLoadSuccess).todos,
          event.filter,
        ),
        event.filter,
      );
    }
  }

  Stream<FilteredTodosState> _mapTodosUpdatedToState(
    TodosUpdated event,
  ) async* {
    final visibilityFilter = state is FilteredTodosLoadSuccess
        ? (state as FilteredTodosLoadSuccess).activeFilter
        : VisibilityFilter.all;
    yield FilteredTodosLoadSuccess(
      _mapTodosToFilteredTodos(
        (todosBloc.state as TodosLoadSuccess).todos,
        visibilityFilter,
      ),
      visibilityFilter,
    );
  }

  List<Todo> _mapTodosToFilteredTodos(
      List<Todo> todos, VisibilityFilter filter) {
    return todos.where((todo) {
      if (filter == VisibilityFilter.all) {
        return true;
      } else if (filter == VisibilityFilter.active) {
        return !todo.complete;
      } else {
```

```
          return todo.complete;
        }
      }).toList();
    }

    @override
    Future<void> close() {
      todosSubscription.cancel();
      return super.close();
    }
  }
```

> We create a `StreamSubscription` for the stream of `TodosStates` so that we can listen to the state changes in the `TodosBloc`. We override the bloc's close method and cancel the subscription so that we can clean up after the bloc is closed.

Barrel File

Just like before, we can create a barrel file to make it more convenient to import the various filtered todos classes.

Create `blocs/filtered_todos/filtered_todos.dart` and export the three files:

```dart
export './filtered_todos_bloc.dart';
export './filtered_todos_event.dart';
export './filtered_todos_state.dart';
```

Next, we're going to implement the `StatsBloc`.

Stats Bloc

> The `StatsBloc` will be responsible for maintaining the statistics for number of active todos and number of completed todos. Similarly, to the

> `FilteredTodosBloc`, it will have a dependency on the `TodosBloc` itself so that it can react to changes in the `TodosBloc` state.

State

Our `StatsBloc` will have two states that it can be in:

- `StatsLoadInProgress` - the state when the statistics have not yet been calculated.
- `StatsLoadSuccess` - the state when the statistics have been calculated.

Create `blocs/stats/stats_state.dart` and let's implement our `StatsState`.

```dart
import 'package:equatable/equatable.dart';

abstract class StatsState extends Equatable {
  const StatsState();

  @override
  List<Object> get props => [];
}

class StatsLoadInProgress extends StatsState {}

class StatsLoadSuccess extends StatsState {
  final int numActive;
  final int numCompleted;

  const StatsLoadSuccess(this.numActive, this.numCompleted);

  @override
  List<Object> get props => [numActive, numCompleted];

  @override
  String toString() {
    return 'StatsLoadSuccess { numActive: $numActive, numCompleted: $numC
  }
}
```

Next, let's define and implement the `StatsEvents`.

Events

There will just be a single event our `StatsBloc` will respond to: `StatsUpdated`. This event will be added whenever the `TodosBloc` state changes so that our `StatsBloc` can recalculate the new statistics.

Create `blocs/stats/stats_event.dart` and let's implement it.

```dart
import 'package:equatable/equatable.dart';
import 'package:flutter_todos/models/models.dart';

abstract class StatsEvent extends Equatable {
  const StatsEvent();
}

class StatsUpdated extends StatsEvent {
  final List<Todo> todos;

  const StatsUpdated(this.todos);

  @override
  List<Object> get props => [todos];

  @override
  String toString() => 'StatsUpdated { todos: $todos }';
}
```

Now we're ready to implement our `StatsBloc` which will look very similar to the `FilteredTodosBloc`.

Bloc

Our `StatsBloc` will have a dependency on the `TodosBloc` itself which will allow it to update its state in response to state changes in the `TodosBloc`.

Create `blocs/stats/stats_bloc.dart` and let's get started.

```dart
import 'dart:async';
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_todos/blocs/blocs.dart';

class StatsBloc extends Bloc<StatsEvent, StatsState> {
  final TodosBloc todosBloc;
  StreamSubscription todosSubscription;

  StatsBloc({@required this.todosBloc}) : super(StatsLoadInProgress()) {
    todosSubscription = todosBloc.listen((state) {
      if (state is TodosLoadSuccess) {
        add(StatsUpdated(state.todos));
      }
    });
  }

  @override
  Stream<StatsState> mapEventToState(StatsEvent event) async* {
    if (event is StatsUpdated) {
      int numActive =
          event.todos.where((todo) => !todo.complete).toList().length;
      int numCompleted =
          event.todos.where((todo) => todo.complete).toList().length;
      yield StatsLoadSuccess(numActive, numCompleted);
    }
  }

  @override
  Future<void> close() {
    todosSubscription.cancel();
    return super.close();
  }
}
```

That's all there is to it! Our `StatsBloc` recalculates its state which contains the number of active todos and the number of completed todos on each state change of our `TodosBloc`.

Now that we're done with the `StatsBloc` we just have one last bloc to implement: the `TabBloc`.

Tab Bloc

> The `TabBloc` will be responsible for maintaining the state of the tabs in our application. It will be taking `TabEvents` as input and outputting `AppTabs`.

Model / State

We need to define an `AppTab` model which we will also use to represent the `TabState`. The `AppTab` will just be an `enum` which represents the active tab in our application. Since the app we're building will only have two tabs: todos and stats, we just need two values.

Create `models/app_tab.dart`:

```dart
enum AppTab { todos, stats }
```

Event

Our `TabBloc` will be responsible for handling a single `TabEvent`:

- `TabUpdated` - which notifies the bloc that the active tab has updated

Create `blocs/tab/tab_event.dart`:

```dart
import 'package:equatable/equatable.dart';
import 'package:flutter_todos/models/models.dart';

abstract class TabEvent extends Equatable {
  const TabEvent();
}

class TabUpdated extends TabEvent {
  final AppTab tab;

  const TabUpdated(this.tab);

  @override
```

```
  List<Object> get props => [tab];

  @override
  String toString() => 'TabUpdated { tab: $tab }';
}
```

Bloc

Our `TabBloc` implementation will be super simple. As always, we just need to set the initial state and implement `mapEventToState`.

Create `blocs/tab/tab_bloc.dart` and let's quickly do the implementation.

```dart
import 'dart:async';
import 'package:bloc/bloc.dart';
import 'package:flutter_todos/blocs/tab/tab.dart';
import 'package:flutter_todos/models/models.dart';

class TabBloc extends Bloc<TabEvent, AppTab> {
  TabBloc() : super(AppTab.todos);

  @override
  Stream<AppTab> mapEventToState(TabEvent event) async* {
    if (event is TabUpdated) {
      yield event.tab;
    }
  }
}
```

I told you it'd be simple. All the `TabBloc` is doing is setting the initial state to the todos tab and handling the `TabUpdated` event by yielding a new `AppTab` instance.

Barrel File

Lastly, we'll create another barrel file for our `TabBloc` exports. Create `blocs/tab/tab.dart` and export the two files:

```dart
export './tab_bloc.dart';
export './tab_event.dart';
```

Bloc Observer

Before we move on to the presentation layer, we will implement our own `BlocObserver` which will allow us to handle all state changes and errors in a single place. It's really useful for things like developer logs or analytics.

Create `blocs/simple_bloc_observer.dart` and let's get started.

```dart
import 'package:bloc/bloc.dart';

class SimpleBlocObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    print(event);
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print(transition);
  }

  @override
  void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
    print(error);
    super.onError(bloc, error, stackTrace);
  }
}
```

All we're doing in this case is printing all state changes (`transitions`) and errors to the console just so that we can see what's going on when we're running our app. You can hook up your `BlocObserver` to google analytics, sentry, crashlytics, etc...

Blocs Barrel

Now that we have all of our blocs implemented we can create a barrel file. Create `blocs/blocs.dart` and export all of our blocs so that we can conveniently import any bloc code with a single import.

```dart
export './filtered_todos/filtered_todos.dart';
export './stats/stats.dart';
export './tab/tab.dart';
export './todos/todos.dart';
export './simple_bloc_observer.dart';
```

Up next, we'll focus on implementing the major screens in our Todos application.

Screens

Home Screen

> Our `HomeScreen` will be responsible for creating the `Scaffold` of our application. It will maintain the `AppBar`, `BottomNavigationBar`, as well as the `Stats` / `FilteredTodos` widgets (depending on the active tab).

Let's create a new directory called `screens` where we will put all of our new screen widgets and then create `screens/home_screen.dart`.

```dart
import 'package:flutter/material.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:flutter_todos/blocs/blocs.dart';
import 'package:flutter_todos/widgets/widgets.dart';
import 'package:flutter_todos/localization.dart';
import 'package:flutter_todos/models/models.dart';

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
      return BlocBuilder<TabBloc, AppTab>(
        builder: (context, activeTab) {
          return Scaffold(
            appBar: AppBar(
              title: Text(FlutterBlocLocalizations.of(context).appTitle),
              actions: [
                FilterButton(visible: activeTab == AppTab.todos),
                ExtraActions(),
              ],
            ),
            body: activeTab == AppTab.todos ? FilteredTodos() : Stats(),
            floatingActionButton: FloatingActionButton(
              key: ArchSampleKeys.addTodoFab,
              onPressed: () {
                Navigator.pushNamed(context, ArchSampleRoutes.addTodo);
              },
              child: Icon(Icons.add),
              tooltip: ArchSampleLocalizations.of(context).addTodo,
            ),
            bottomNavigationBar: TabSelector(
              activeTab: activeTab,
              onTabSelected: (tab) =>
                  BlocProvider.of<TabBloc>(context).add(TabUpdated(tab)),
            ),
          );
        },
      );
    }
  }
```

The `HomeScreen` accesses the `TabBloc` using `BlocProvider.of<TabBloc>(context)` which will be made available from our root `TodosApp` widget (we'll get to it later in this tutorial).

Next, we'll implement the `DetailsScreen`.

Details Screen

> The `DetailsScreen` displays the full details of the selected todo and allows the user to either edit or delete the todo.

Create `screens/details_screen.dart` and let's build it.

```dart
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/blocs/todos/todos.dart';
import 'package:flutter_todos/screens/screens.dart';
import 'package:flutter_todos/flutter_todos_keys.dart';

class DetailsScreen extends StatelessWidget {
  final String id;

  DetailsScreen({Key key, @required this.id})
      : super(key: key ?? ArchSampleKeys.todoDetailsScreen);

  @override
  Widget build(BuildContext context) {
    return BlocBuilder<TodosBloc, TodosState>(
      builder: (context, state) {
        final todo = (state as TodosLoadSuccess)
            .todos
            .firstWhere((todo) => todo.id == id, orElse: () => null);
        final localizations = ArchSampleLocalizations.of(context);
        return Scaffold(
          appBar: AppBar(
            title: Text(localizations.todoDetails),
            actions: [
              IconButton(
                tooltip: localizations.deleteTodo,
                key: ArchSampleKeys.deleteTodoButton,
                icon: Icon(Icons.delete),
                onPressed: () {
                  BlocProvider.of<TodosBloc>(context).add(TodoDeleted(tod
                  Navigator.pop(context, todo);
                },
              )
            ],
          ),
          body: todo == null
              ? Container(key: FlutterTodosKeys.emptyDetailsContainer)
              : Padding(
```

```
                padding: EdgeInsets.all(16.0),
                child: ListView(
                  children: [
                    Row(
                      crossAxisAlignment: CrossAxisAlignment.start,
                      children: [
                        Padding(
                          padding: EdgeInsets.only(right: 8.0),
                          child: Checkbox(
                              key: FlutterTodosKeys.detailsScreenCheckB
                              value: todo.complete,
                              onChanged: (_) {
                                BlocProvider.of<TodosBloc>(context).add
                                  TodoUpdated(
                                    todo.copyWith(complete: !todo.compl
                                  ),
                                );
                              }),
                        ),
                        Expanded(
                          child: Column(
                            crossAxisAlignment: CrossAxisAlignment.star
                            children: [
                              Hero(
                                tag: '${todo.id}__heroTag',
                                child: Container(
                                  width: MediaQuery.of(context).size.wi
                                  padding: EdgeInsets.only(
                                    top: 8.0,
                                    bottom: 16.0,
                                  ),
                                  child: Text(
                                    todo.task,
                                    key: ArchSampleKeys.detailsTodoItem
                                    style:
                                        Theme.of(context).textTheme.hea
                                  ),
                                ),
                              ),
                              Text(
                                todo.note,
                                key: ArchSampleKeys.detailsTodoItemNote
                                style: Theme.of(context).textTheme.subt
```

```
                                ),
                              ],
                            ),
                          ),
                        ],
                      ),
                    ],
                  ),
                ),
              floatingActionButton: FloatingActionButton(
                key: ArchSampleKeys.editTodoFab,
                tooltip: localizations.editTodo,
                child: Icon(Icons.edit),
                onPressed: todo == null
                    ? null
                    : () {
                        Navigator.of(context).push(
                          MaterialPageRoute(
                            builder: (context) {
                              return AddEditScreen(
                                key: ArchSampleKeys.editTodoScreen,
                                onSave: (task, note) {
                                  BlocProvider.of<TodosBloc>(context).add(
                                    TodoUpdated(
                                      todo.copyWith(task: task, note: note),
                                    ),
                                  );
                                },
                                isEditing: true,
                                todo: todo,
                              );
                            },
                          ),
                        );
                      },
              ),
            );
          },
        );
      }
    }
```

Note: The `DetailsScreen` requires a todo id so that it can pull the todo details from the `TodosBloc` and so that it can update whenever a todo's details have been changed (a todo's id cannot be changed).

The main things to note are that there is an `IconButton` which adds a `TodoDeleted` event as well as a checkbox which adds an `TodoUpdated` event.

There is also another `FloatingActionButton` which navigates the user to the `AddEditScreen` with `isEditing` set to `true`. We'll take a look at the `AddEditScreen` next.

Add/Edit Screen

> The `AddEditScreen` widget allows the user to either create a new todo or update an existing todo based on the `isEditing` flag that is passed via the constructor.

Create `screens/add_edit_screen.dart` and let's have a look at the implementation.

```dart
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/models/models.dart';

typedef OnSaveCallback = Function(String task, String note);

class AddEditScreen extends StatefulWidget {
  final bool isEditing;
  final OnSaveCallback onSave;
  final Todo todo;

  AddEditScreen({
    Key key,
    @required this.onSave,
    @required this.isEditing,
    this.todo,
  }) : super(key: key ?? ArchSampleKeys.addTodoScreen);
```

```
  @override
  _AddEditScreenState createState() => _AddEditScreenState();
}

class _AddEditScreenState extends State<AddEditScreen> {
  static final GlobalKey<FormState> _formKey = GlobalKey<FormState>();

  String _task;
  String _note;

  bool get isEditing => widget.isEditing;

  @override
  Widget build(BuildContext context) {
    final localizations = ArchSampleLocalizations.of(context);
    final textTheme = Theme.of(context).textTheme;

    return Scaffold(
      appBar: AppBar(
        title: Text(
          isEditing ? localizations.editTodo : localizations.addTodo,
        ),
      ),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: ListView(
            children: [
              TextFormField(
                initialValue: isEditing ? widget.todo.task : '',
                key: ArchSampleKeys.taskField,
                autofocus: !isEditing,
                style: textTheme.headline5,
                decoration: InputDecoration(
                  hintText: localizations.newTodoHint,
                ),
                validator: (val) {
                  return val.trim().isEmpty
                      ? localizations.emptyTodoError
                      : null;
                },
                onSaved: (value) => _task = value,
```

```
              ),
              TextFormField(
                initialValue: isEditing ? widget.todo.note : '',
                key: ArchSampleKeys.noteField,
                maxLines: 10,
                style: textTheme.subtitle1,
                decoration: InputDecoration(
                  hintText: localizations.notesHint,
                ),
                onSaved: (value) => _note = value,
              )
            ],
          ),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        key:
            isEditing ? ArchSampleKeys.saveTodoFab : ArchSampleKeys.saveN
        tooltip: isEditing ? localizations.saveChanges : localizations.ad
        child: Icon(isEditing ? Icons.check : Icons.add),
        onPressed: () {
          if (_formKey.currentState.validate()) {
            _formKey.currentState.save();
            widget.onSave(_task, _note);
            Navigator.pop(context);
          }
        },
      ),
    );
  }
}
```

There's nothing bloc-specific in this widget. It's simply presenting a form and:

- if `isEditing` is true the form is populated it with the existing todo details.
- otherwise the inputs are empty so that the user can create a new todo.

It uses an `onSave` callback function to notify its parent of the updated or newly created todo.

That's it for the screens in our application so before we forget, let's create a barrel file to export them.

Screens Barrel

Create `screens/screens.dart` and export all three.

```dart
export './add_edit_screen.dart';
export './details_screen.dart';
export './home_screen.dart';
```

Next, let's implement all of the "widgets" (anything that isn't a screen).

Widgets

Filter Button

> The `FilterButton` widget will be responsible for providing the user with a list of filter options and will notify the `FilteredTodosBloc` when a new filter is selected.

Let's create a new directory called `widgets` and put our `FilterButton` implementation in `widgets/filter_button.dart`.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/blocs/filtered_todos/filtered_todos.dart';
import 'package:flutter_todos/models/models.dart';

class FilterButton extends StatelessWidget {
  final bool visible;

  FilterButton({this.visible, Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final defaultStyle = Theme.of(context).textTheme.bodyText2;
    final activeStyle = Theme.of(context)
```

```
          .textTheme
          .bodyText2
          .copyWith(color: Theme.of(context).accentColor);
      return BlocBuilder<FilteredTodosBloc, FilteredTodosState>(
          builder: (context, state) {
        final button = _Button(
          onSelected: (filter) {
            BlocProvider.of<FilteredTodosBloc>(context).add(FilterUpdated(f
          },
          activeFilter: state is FilteredTodosLoadSuccess
              ? state.activeFilter
              : VisibilityFilter.all,
          activeStyle: activeStyle,
          defaultStyle: defaultStyle,
        );
        return AnimatedOpacity(
          opacity: visible ? 1.0 : 0.0,
          duration: Duration(milliseconds: 150),
          child: visible ? button : IgnorePointer(child: button),
        );
      });
    }
  }

  class _Button extends StatelessWidget {
    const _Button({
      Key key,
      @required this.onSelected,
      @required this.activeFilter,
      @required this.activeStyle,
      @required this.defaultStyle,
    }) : super(key: key);

    final PopupMenuItemSelected<VisibilityFilter> onSelected;
    final VisibilityFilter activeFilter;
    final TextStyle activeStyle;
    final TextStyle defaultStyle;

    @override
    Widget build(BuildContext context) {
      return PopupMenuButton<VisibilityFilter>(
        key: ArchSampleKeys.filterButton,
        tooltip: ArchSampleLocalizations.of(context).filterTodos,
```

```
        onSelected: onSelected,
        itemBuilder: (BuildContext context) => <PopupMenuItem<VisibilityFil
              PopupMenuItem<VisibilityFilter>(
                key: ArchSampleKeys.allFilter,
                value: VisibilityFilter.all,
                child: Text(
                  ArchSampleLocalizations.of(context).showAll,
                  style: activeFilter == VisibilityFilter.all
                      ? activeStyle
                      : defaultStyle,
                ),
              ),
              PopupMenuItem<VisibilityFilter>(
                key: ArchSampleKeys.activeFilter,
                value: VisibilityFilter.active,
                child: Text(
                  ArchSampleLocalizations.of(context).showActive,
                  style: activeFilter == VisibilityFilter.active
                      ? activeStyle
                      : defaultStyle,
                ),
              ),
              PopupMenuItem<VisibilityFilter>(
                key: ArchSampleKeys.completedFilter,
                value: VisibilityFilter.completed,
                child: Text(
                  ArchSampleLocalizations.of(context).showCompleted,
                  style: activeFilter == VisibilityFilter.completed
                      ? activeStyle
                      : defaultStyle,
                ),
              ),
            ],
        icon: Icon(Icons.filter_list),
      );
    }
  }
```

The `FilterButton` needs to respond to state changes in the `FilteredTodosBloc` so it uses `BlocProvider` to access the `FilteredTodosBloc` from the `BuildContext`.

It then uses `BlocBuilder` to re-render whenever the `FilteredTodosBloc` changes state.

The rest of the implementation is pure Flutter and there isn't much going on so we can move on to the `ExtraActions` widget.

Extra Actions

> Similarly to the `FilterButton`, the `ExtraActions` widget is responsible for providing the user with a list of extra options: Toggling Todos and Clearing Completed Todos.

Since this widget doesn't care about the filters it will interact with the `TodosBloc` instead of the `FilteredTodosBloc`.

Let's create the `ExtraAction` model in `models/extra_action.dart`.

```dart
enum ExtraAction { toggleAllComplete, clearCompleted }
```

And don't forget to export it from the `models/models.dart` barrel file.

Next, let's create `widgets/extra_actions.dart` and implement it.

```dart
import 'package:flutter/material.dart';
import 'package:flutter/foundation.dart';
import 'package:flutter/widgets.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/blocs/todos/todos.dart';
import 'package:flutter_todos/models/models.dart';
import 'package:flutter_todos/flutter_todos_keys.dart';

class ExtraActions extends StatelessWidget {
  ExtraActions({Key key}) : super(key: ArchSampleKeys.extraActionsButton)

  @override
  Widget build(BuildContext context) {
    return BlocBuilder<TodosBloc, TodosState>(
```

```
        builder: (context, state) {
          if (state is TodosLoadSuccess) {
            bool allComplete =
                (BlocProvider.of<TodosBloc>(context).state as TodosLoadSucc
                    .todos
                    .every((todo) => todo.complete);
            return PopupMenuButton<ExtraAction>(
              key: FlutterTodosKeys.extraActionsPopupMenuButton,
              onSelected: (action) {
                switch (action) {
                  case ExtraAction.clearCompleted:
                    BlocProvider.of<TodosBloc>(context).add(ClearCompleted(
                    break;
                  case ExtraAction.toggleAllComplete:
                    BlocProvider.of<TodosBloc>(context).add(ToggleAll());
                    break;
                }
              },
              itemBuilder: (BuildContext context) => <PopupMenuItem<ExtraAc
                PopupMenuItem<ExtraAction>(
                  key: ArchSampleKeys.toggleAll,
                  value: ExtraAction.toggleAllComplete,
                  child: Text(
                    allComplete
                        ? ArchSampleLocalizations.of(context).markAllIncomp
                        : ArchSampleLocalizations.of(context).markAllComple
                  ),
                ),
                PopupMenuItem<ExtraAction>(
                  key: ArchSampleKeys.clearCompleted,
                  value: ExtraAction.clearCompleted,
                  child: Text(
                    ArchSampleLocalizations.of(context).clearCompleted,
                  ),
                ),
              ],
            );
          }
          return Container(key: FlutterTodosKeys.extraActionsEmptyContainer
        },
      );
    }
  }
```

Just like with the `FilterButton`, we use `BlocProvider` to access the `TodosBloc` from the `BuildContext` and `BlocBuilder` to respond to state changes in the `TodosBloc`.

Based on the action selected, the widget adds an event to the `TodosBloc` to either `ToggleAll` todos' completion states or `ClearCompleted` todos.

Next we'll take a look at the `TabSelector` widget.

Tab Selector

> The `TabSelector` widget is responsible for displaying the tabs in the `BottomNavigationBar` and handling user input.

Let's create `widgets/tab_selector.dart` and implement it.

```dart
import 'package:flutter/cupertino.dart';
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/models/models.dart';

class TabSelector extends StatelessWidget {
  final AppTab activeTab;
  final Function(AppTab) onTabSelected;

  TabSelector({
    Key key,
    @required this.activeTab,
    @required this.onTabSelected,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BottomNavigationBar(
      key: ArchSampleKeys.tabs,
      currentIndex: AppTab.values.indexOf(activeTab),
      onTap: (index) => onTabSelected(AppTab.values[index]),
```

```
        items: AppTab.values.map((tab) {
          return BottomNavigationBarItem(
            icon: Icon(
              tab == AppTab.todos ? Icons.list : Icons.show_chart,
              key: tab == AppTab.todos
                  ? ArchSampleKeys.todoTab
                  : ArchSampleKeys.statsTab,
            ),
            title: Text(tab == AppTab.stats
                ? ArchSampleLocalizations.of(context).stats
                : ArchSampleLocalizations.of(context).todos),
          );
        }).toList(),
      );
    }
  }
```

You can see that there is no dependency on blocs in this widget; it just calls `onTabSelected` when a tab is selected and also takes an `activeTab` as input so it knows which tab is currently selected.

Next, we'll take a look at the `FilteredTodos` widget.

Filtered Todos

> The `FilteredTodos` widget is responsible for showing a list of todos based on the current active filter.

Create `widgets/filtered_todos.dart` and let's implement it.

dart

```
import 'package:flutter/material.dart';
import 'package:flutter/foundation.dart';
import 'package:flutter/widgets.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/blocs/blocs.dart';
import 'package:flutter_todos/widgets/widgets.dart';
import 'package:flutter_todos/screens/screens.dart';
import 'package:flutter_todos/flutter_todos_keys.dart';
```

```
class FilteredTodos extends StatelessWidget {
  FilteredTodos({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final localizations = ArchSampleLocalizations.of(context);

    return BlocBuilder<FilteredTodosBloc, FilteredTodosState>(
      builder: (context, state) {
        if (state is FilteredTodosLoadInProgress) {
          return LoadingIndicator(key: ArchSampleKeys.todosLoading);
        } else if (state is FilteredTodosLoadSuccess) {
          final todos = state.filteredTodos;
          return ListView.builder(
            key: ArchSampleKeys.todoList,
            itemCount: todos.length,
            itemBuilder: (BuildContext context, int index) {
              final todo = todos[index];
              return TodoItem(
                todo: todo,
                onDismissed: (direction) {
                  BlocProvider.of<TodosBloc>(context).add(TodoDeleted(tod
                  ScaffoldMessenger.of(context).showSnackBar(DeleteTodoSn
                    key: ArchSampleKeys.snackbar,
                    todo: todo,
                    onUndo: () =>
                        BlocProvider.of<TodosBloc>(context).add(TodoAdded
                    localizations: localizations,
                  ));
                },
                onTap: () async {
                  final removedTodo = await Navigator.of(context).push(
                    MaterialPageRoute(builder: (_) {
                      return DetailsScreen(id: todo.id);
                    }),
                  );
                  if (removedTodo != null) {
                    ScaffoldMessenger.of(context).showSnackBar(DeleteTodo
                      key: ArchSampleKeys.snackbar,
                      todo: todo,
                      onUndo: () => BlocProvider.of<TodosBloc>(context)
                          .add(TodoAdded(todo)),
```

```
                    localizations: localizations,
                  ));
              }
            },
            onCheckboxChanged: (_) {
              BlocProvider.of<TodosBloc>(context).add(
                TodoUpdated(todo.copyWith(complete: !todo.complete)),
              );
            },
          );
        },
      );
    } else {
      return Container(key: FlutterTodosKeys.filteredTodosEmptyContai
    }
  },
);
}
}
```

Just like the previous widgets we've written, the `FilteredTodos` widget uses `BlocProvider` to access blocs (in this case both the `FilteredTodosBloc` and the `TodosBloc` are needed).

> The `FilteredTodosBloc` is needed to help us render the correct todos based on the current filter

> The `TodosBloc` is needed to allow us to add/delete todos in response to user interactions such as swiping on an individual todo.

From the `FilteredTodos` widget, the user can navigate to the `DetailsScreen` where it is possible to edit or delete the selected todo. Since our `FilteredTodos` widget renders a list of `TodoItem` widgets, we'll take a look at those next.

Todo Item

> `TodoItem` is a stateless widget which is responsible for rendering a single todo and handling user interactions (taps/swipes).

Create `widgets/todo_item.dart` and let's build it.

```dart
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/models/models.dart';

class TodoItem extends StatelessWidget {
  final DismissDirectionCallback onDismissed;
  final GestureTapCallback onTap;
  final ValueChanged<bool> onCheckboxChanged;
  final Todo todo;

  TodoItem({
    Key key,
    @required this.onDismissed,
    @required this.onTap,
    @required this.onCheckboxChanged,
    @required this.todo,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Dismissible(
      key: ArchSampleKeys.todoItem(todo.id),
      onDismissed: onDismissed,
      child: ListTile(
        onTap: onTap,
        leading: Checkbox(
          key: ArchSampleKeys.todoItemCheckbox(todo.id),
          value: todo.complete,
          onChanged: onCheckboxChanged,
        ),
        title: Hero(
          tag: '${todo.id}__heroTag',
          child: Container(
            width: MediaQuery.of(context).size.width,
            child: Text(
```

```
              todo.task,
              key: ArchSampleKeys.todoItemTask(todo.id),
              style: Theme.of(context).textTheme.headline6,
            ),
          ),
        ),
        subtitle: todo.note.isNotEmpty
            ? Text(
                todo.note,
                key: ArchSampleKeys.todoItemNote(todo.id),
                maxLines: 1,
                overflow: TextOverflow.ellipsis,
                style: Theme.of(context).textTheme.subtitle1,
              )
            : null,
      ),
    );
  }
}
```

Again, notice that the `TodoItem` has no bloc-specific code in it. It simply renders based on the todo we pass via the constructor and calls the injected callback functions whenever the user interacts with the todo.

Next up, we'll create the `DeleteTodoSnackBar`.

Delete Todo SnackBar

> The `DeleteTodoSnackBar` is responsible for indicating to the user that a todo was deleted and allows the user to undo his/her action.

Create `widgets/delete_todo_snack_bar.dart` and let's implement it.

```dart
import 'package:flutter/material.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/models/models.dart';

class DeleteTodoSnackBar extends SnackBar {
  final ArchSampleLocalizations localizations;
```

```
  DeleteTodoSnackBar({
    Key key,
    @required Todo todo,
    @required VoidCallback onUndo,
    @required this.localizations,
  }) : super(
          key: key,
          content: Text(
            localizations.todoDeleted(todo.task),
            maxLines: 1,
            overflow: TextOverflow.ellipsis,
          ),
          duration: Duration(seconds: 2),
          action: SnackBarAction(
            label: localizations.undo,
            onPressed: onUndo,
          ),
        );
}
```

By now, you're probably noticing a pattern: this widget also has no bloc-specific code. It simply takes in a todo in order to render the task and calls a callback function called `onUndo` if a user presses the undo button.

We're almost done; just two more widgets to go!

Loading Indicator

> The `LoadingIndicator` widget is a stateless widget that is responsible for indicating to the user that something is in progress.

Create `widgets/loading_indicator.dart` and let's write it.

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

class LoadingIndicator extends StatelessWidget {
  LoadingIndicator({Key key}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    return Center(
      child: CircularProgressIndicator(),
    );
  }
}
```

Not much to discuss here; we're just using a `CircularProgressIndicator` wrapped in a `Center` widget (again no bloc-specific code).

Lastly, we need to build our `Stats` widget.

Stats

> The `Stats` widget is responsible for showing the user how many todos are active (in progress) vs. completed.

Let's create `widgets/stats.dart` and take a look at the implementation.

dart

```
import 'package:flutter/material.dart';
import 'package:flutter/foundation.dart';
import 'package:flutter/widgets.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/blocs/stats/stats.dart';
import 'package:flutter_todos/widgets/widgets.dart';
import 'package:flutter_todos/flutter_todos_keys.dart';

class Stats extends StatelessWidget {
  Stats({Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocBuilder<StatsBloc, StatsState>(
      builder: (context, state) {
        if (state is StatsLoadInProgress) {
          return LoadingIndicator(key: FlutterTodosKeys.statsLoadInProgre
        } else if (state is StatsLoadSuccess) {
```

```
            return Center(
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  Padding(
                    padding: EdgeInsets.only(bottom: 8.0),
                    child: Text(
                      ArchSampleLocalizations.of(context).completedTodos,
                      style: Theme.of(context).textTheme.headline6,
                    ),
                  ),
                  Padding(
                    padding: EdgeInsets.only(bottom: 24.0),
                    child: Text(
                      '${state.numCompleted}',
                      key: ArchSampleKeys.statsNumCompleted,
                      style: Theme.of(context).textTheme.subtitle1,
                    ),
                  ),
                  Padding(
                    padding: EdgeInsets.only(bottom: 8.0),
                    child: Text(
                      ArchSampleLocalizations.of(context).activeTodos,
                      style: Theme.of(context).textTheme.headline6,
                    ),
                  ),
                  Padding(
                    padding: EdgeInsets.only(bottom: 24.0),
                    child: Text(
                      "${state.numActive}",
                      key: ArchSampleKeys.statsNumActive,
                      style: Theme.of(context).textTheme.subtitle1,
                    ),
                  )
                ],
              ),
            );
          } else {
            return Container(key: FlutterTodosKeys.emptyStatsContainer);
          }
        },
      );
```

```
    }
  }
```

We're accessing the `StatsBloc` using `BlocProvider` and using `BlocBuilder` to rebuild in response to state changes in the `StatsBloc` state.

Putting it all together

Let's create `main.dart` and our `TodosApp` widget. We need to create a `main` function and run our `TodosApp`.

```dart
void main() {
  Bloc.observer = SimpleBlocObserver();
  runApp(
    BlocProvider(
      create: (context) {
        return TodosBloc(
          todosRepository: const TodosRepositoryFlutter(
            fileStorage: const FileStorage(
              '__flutter_bloc_app__',
              getApplicationDocumentsDirectory,
            ),
          ),
        )..add(TodosLoadSuccess());
      },
      child: TodosApp(),
    ),
  );
}
```

Note: We are setting our observer to the `SimpleBlocObserver` we created earlier so that we can hook into all transitions and errors.

Note: We are also wrapping our `TodosApp` widget in a `BlocProvider` which manages initializing, closing, and providing the `TodosBloc` to our entire

widget tree from flutter_bloc. We immediately add the `TodosLoadSuccess` event in order to request the latest todos.

Next, let's implement our `TodosApp` widget.

```dart
class TodosApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: FlutterBlocLocalizations().appTitle,
      theme: ArchSampleTheme.theme,
      localizationsDelegates: [
        ArchSampleLocalizationsDelegate(),
        FlutterBlocLocalizationsDelegate(),
      ],
      routes: {
        ArchSampleRoutes.home: (context) {
          return MultiBlocProvider(
            providers: [
              BlocProvider<TabBloc>(
                create: (context) => TabBloc(),
              ),
              BlocProvider<FilteredTodosBloc>(
                create: (context) => FilteredTodosBloc(
                  todosBloc: BlocProvider.of<TodosBloc>(context),
                ),
              ),
              BlocProvider<StatsBloc>(
                create: (context) => StatsBloc(
                  todosBloc: BlocProvider.of<TodosBloc>(context),
                ),
              ),
            ],
            child: HomeScreen(),
          );
        },
        ArchSampleRoutes.addTodo: (context) {
          return AddEditScreen(
            key: ArchSampleKeys.addTodoScreen,
            onSave: (task, note) {
              BlocProvider.of<TodosBloc>(context).add(
```

```
                TodoAdded(Todo(task, note: note)),
              );
            },
            isEditing: false,
          );
        },
      },
    );
  }
}
```

Our `TodosApp` is a `StatelessWidget` which accesses the provided `TodosBloc` via the `BuildContext`.

The `TodosApp` has two routes:

- `Home` - which renders a `HomeScreen`
- `TodoAdded` - which renders a `AddEditScreen` with `isEditing` set to `false`.

The `TodosApp` also makes the `TabBloc`, `FilteredTodosBloc`, and `StatsBloc` available to the widgets in its subtree by using the `MultiBlocProvider` widget from flutter_bloc.

```dart
MultiBlocProvider(
  providers: [
    BlocProvider<TabBloc>(
      create: (context) => TabBloc(),
    ),
    BlocProvider<FilteredTodosBloc>(
      create: (context) => FilteredTodosBloc(todosBloc: todosBloc),
    ),
    BlocProvider<StatsBloc>(
      create: (context) => StatsBloc(todosBloc: todosBloc),
    ),
  ],
  child: HomeScreen(),
);
```

is equivalent to writing

```dart
BlocProvider<TabBloc>(
  create: (context) => TabBloc(),
  child: BlocProvider<FilteredTodosBloc>(
    create: (context) => FilteredTodosBloc(todosBloc: todosBloc),
    child: BlocProvider<StatsBloc>(
      create: (context) => StatsBloc(todosBloc: todosBloc),
      child: Scaffold(...),
    ),
  ),
);
```

You can see how using `MultiBlocProvider` helps reduce the levels of nesting and makes the code easier to read and maintain.

The entire `main.dart` should look like this:

```dart
import 'package:flutter/material.dart';
import 'package:bloc/bloc.dart';
import 'package:path_provider/path_provider.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:todos_repository_simple/todos_repository_simple.dart';
import 'package:todos_app_core/todos_app_core.dart';
import 'package:flutter_todos/localization.dart';
import 'package:flutter_todos/blocs/blocs.dart';
import 'package:flutter_todos/models/models.dart';
import 'package:flutter_todos/screens/screens.dart';

void main() {
  Bloc.observer = SimpleBlocObserver();
  runApp(
    BlocProvider(
      create: (context) {
        return TodosBloc(
          todosRepository: const TodosRepositoryFlutter(
            fileStorage: const FileStorage(
              '__flutter_bloc_app__',
              getApplicationDocumentsDirectory,
            ),
          ),
        )..add(TodosLoadSuccess());
```

```
      },
      child: TodosApp(),
    ),
  );
}

class TodosApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: FlutterBlocLocalizations().appTitle,
      theme: ArchSampleTheme.theme,
      localizationsDelegates: [
        ArchSampleLocalizationsDelegate(),
        FlutterBlocLocalizationsDelegate(),
      ],
      routes: {
        ArchSampleRoutes.home: (context) {
          return MultiBlocProvider(
            providers: [
              BlocProvider<TabBloc>(
                create: (context) => TabBloc(),
              ),
              BlocProvider<FilteredTodosBloc>(
                create: (context) => FilteredTodosBloc(
                  todosBloc: BlocProvider.of<TodosBloc>(context),
                ),
              ),
              BlocProvider<StatsBloc>(
                create: (context) => StatsBloc(
                  todosBloc: BlocProvider.of<TodosBloc>(context),
                ),
              ),
            ],
            child: HomeScreen(),
          );
        },
        ArchSampleRoutes.addTodo: (context) {
          return AddEditScreen(
            key: ArchSampleKeys.addTodoScreen,
            onSave: (task, note) {
              BlocProvider.of<TodosBloc>(context).add(
                TodoAdded(Todo(task, note: note)),
```

```
                );
              },
              isEditing: false,
            );
          },
        },
      );
    }
  }
```

That's all there is to it! We've now successfully implemented a todos app in flutter using the bloc and flutter_bloc packages and we've successfully separated our presentation layer from our business logic.

The full source for this example can be found here.

PREVIOUS
Weather

NEXT
Firebase Login ✨