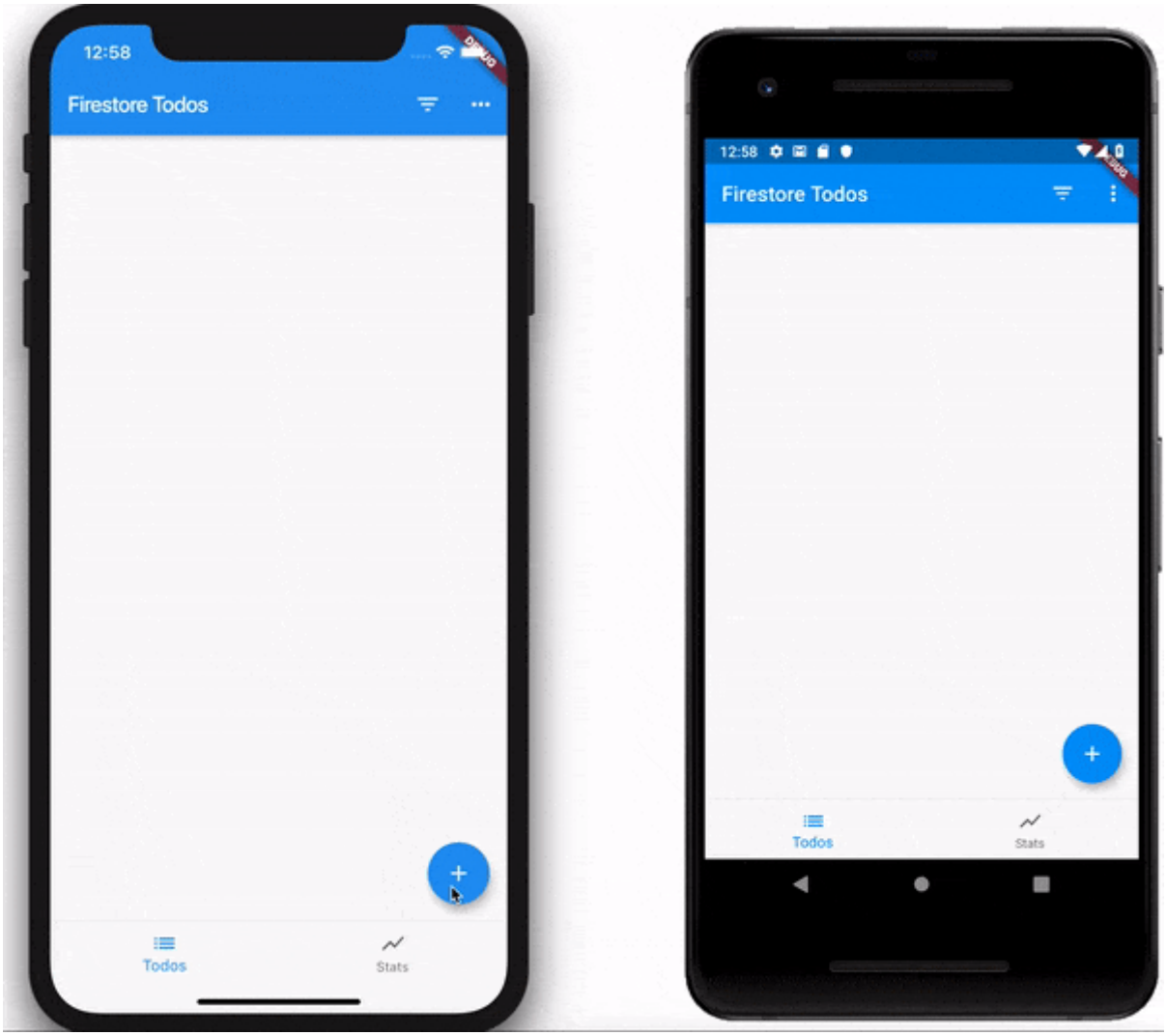
Flutter Firestore Todos Tutorial

level advanced

> In the following tutorial, we're going to build a reactive Todos App which hooks up to Firestore. We're going to be building on top of the flutter todos example so we won't go into the UI since it will all be the same.



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.
- BlocListener, a Flutter widget which invokes the listener code in response to state changes in the bloc.
- MultiBlocProvider, a Flutter widget that merges multiple BlocProvider widgets into one.
- Using Bloc instead of Cubit. What's the difference?
- Prevent unnecessary rebuilds with Equatable.

The only things we're going to be refactoring in our existing todos example are the repository layer and parts of the bloc layer.

We'll start off in the repository layer with the `TodosRepository`.

Todos Repository

Create a new package at the root level of our app called `todos_repository`.

> **Note:** The reason for making the repository a standalone package is to illustrate that the repository should be decoupled from the application and can be reused across multiple apps.

Inside our `todos_repository` create the following folder/file structure.

```sh
├── lib
|   ├── src
|   |   ├── entities
|   |   |   ├── entities.dart
|   |   |   └── todo_entity.dart
|   |   ├── models
|   |   |   ├── models.dart
|   |   |   └── todo.dart
|   |   ├── todos_repository.dart
|   |   └── firebase_todos_repository.dart
|   └── todos_repository.dart
└── pubspec.yaml
```

Dependencies

The `pubspec.yaml` should look like:

```yaml
name: todos_repository

version: 1.0.0+1

environment:
  sdk: ">=2.6.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  cloud_firestore: ^0.12.10+2
  rxdart: ^0.23.0
  equatable: ^1.0.0
  firebase_core: ^0.4.0+8
```

Note: We can immediately see our `todos_repository` has a dependency on `firebase_core` and `cloud_firestore`.

Package Root

The `todos_repository.dart` directly inside `lib` should look like:

```dart
library todos_repository;

export 'src/firebase_todos_repository.dart';
export 'src/models/models.dart';
export 'src/todos_repository.dart';
```

This is where all of our public classes are exported. If we want a class to be private to the package we should make sure to omit it.

Entities

> Entities represent the data provided by our data provider.

The `entities.dart` file is a barrel file that exports the `todo_entity.dart` file.

```dart
export 'todo_entity.dart';
```

Our `TodoEntity` is the representation of our `Todo` inside Firestore. Create `todo_entity.dart` and let's implement it.

```dart
// Copyright 2018 The Flutter Architecture Sample Authors. All rights res
// Use of this source code is governed by the MIT license that can be fou
// in the LICENSE file.

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

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

  const TodoEntity(this.task, this.id, this.note, this.complete);

  Map<String, Object> toJson() {
    return {
      "complete": complete,
      "task": task,
      "note": note,
      "id": id,
    };
  }

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

  @override
```

```
  String toString() {
    return 'TodoEntity { complete: $complete, task: $task, note: $note, i
  }

  static TodoEntity fromJson(Map<String, Object> json) {
    return TodoEntity(
      json["task"] as String,
      json["id"] as String,
      json["note"] as String,
      json["complete"] as bool,
    );
  }

  static TodoEntity fromSnapshot(DocumentSnapshot snap) {
    return TodoEntity(
      snap.data['task'],
      snap.documentID,
      snap.data['note'],
      snap.data['complete'],
    );
  }

  Map<String, Object> toDocument() {
    return {
      "complete": complete,
      "task": task,
      "note": note,
    };
  }
}
```

The `toJson` and `fromJson` are standard methods for converting to/from json. The `fromSnapshot` and `toDocument` are specific to Firestore.

> **Note:** Firestore will automatically create the id for the document when we insert it. As such we don't want to duplicate data by storing the id in an id field.

Models

> Models will contain plain dart classes which we will work with in our Flutter Application. Having the separation between models and entities allows us to switch our data provider at any time and only have to change the the `toEntity` and `fromEntity` conversion in our model layer.

Our `models.dart` is another barrel file. Inside the `todo.dart` let's put the following code.

```dart
import 'package:meta/meta.dart';
import '../entities/entities.dart';

@immutable
class Todo {
  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;

  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
  int get hashCode =>
      complete.hashCode ^ task.hashCode ^ note.hashCode ^ id.hashCode;

  @override
  bool operator ==(Object other) =>
      identical(this, other) ||
      other is Todo &&
          runtimeType == other.runtimeType &&
          complete == other.complete &&
```

```
            task == other.task &&
            note == other.note &&
            id == other.id;

  @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,
    );
  }
}
```

Todos Repository

> `TodosRepository` is our abstract base class which we can extend whenever we want to integrate with a different `TodosProvider`.

Let's create `todos_repository.dart`

```dart
import 'dart:async';

import 'package:todos_repository/todos_repository.dart';

abstract class TodosRepository {
  Future<void> addNewTodo(Todo todo);

  Future<void> deleteTodo(Todo todo);
```

```
  Stream<List<Todo>> todos();

  Future<void> updateTodo(Todo todo);
}
```

Note: Because we have this interface it is easy to add another type of datastore. If, for example, we wanted to use something like sembast all we would need to do is create a separate repository for handling the sembast specific code.

Firebase Todos Repository

> `FirebaseTodosRepository` manages the integration with Firestore and implements our `TodosRepository` interface.

Let's open `firebase_todos_repository.dart` and implement it!

dart

```
import 'dart:async';

import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:todos_repository/todos_repository.dart';
import 'entities/entities.dart';

class FirebaseTodosRepository implements TodosRepository {
  final todoCollection = Firestore.instance.collection('todos');

  @override
  Future<void> addNewTodo(Todo todo) {
    return todoCollection.add(todo.toEntity().toDocument());
  }

  @override
  Future<void> deleteTodo(Todo todo) async {
    return todoCollection.document(todo.id).delete();
  }

  @override
```

```
  Stream<List<Todo>> todos() {
    return todoCollection.snapshots().map((snapshot) {
      return snapshot.documents
          .map((doc) => Todo.fromEntity(TodoEntity.fromSnapshot(doc)))
          .toList();
    });
  }

  @override
  Future<void> updateTodo(Todo update) {
    return todoCollection
        .document(update.id)
        .updateData(update.toEntity().toDocument());
  }
}
```

That's it for our `TodosRepository`, next we need to create a simple `UserRepository` to manage authenticating our users.

User Repository

Create a new package at the root level of our app called `user_repository`.

Inside our `user_repository` create the following folder/file structure.

```sh
├── lib
|   ├── src
|   |   └── user_repository.dart
|   └── user_repository.dart
└── pubspec.yaml
```

Dependencies

The `pubspec.yaml` should look like:

```yaml
name: user_repository
```

```
version: 1.0.0+1

environment:
  sdk: ">=2.6.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  firebase_auth: ^0.15.0+1
```

Note: We can immediately see our `user_repository` has a dependency on `firebase_auth`.

Package Root

The `user_repository.dart` directly inside `lib` should look like:

```dart
library user_repository;

export 'src/user_repository.dart';
```

User Repository

> `UserRepository` is our abstract base class which we can extend whenever we want to integrate with a different provider`.

Let's create `user_repository.dart`

```dart
abstract class UserRepository {
  Future<bool> isAuthenticated();

  Future<void> authenticate();
```

```
  Future<String> getUserId();
}
```

Firebase User Repository

> `FirebaseUserRepository` manages the integration with Firebase and implements our `UserRepository` interface.

Let's open `firebase_user_repository.dart` and implement it!

```dart
import 'package:firebase_auth/firebase_auth.dart';
import 'package:user_repository/user_repository.dart';

class FirebaseUserRepository implements UserRepository {
  final FirebaseAuth _firebaseAuth;

  FirebaseUserRepository({FirebaseAuth firebaseAuth})
      : _firebaseAuth = firebaseAuth ?? FirebaseAuth.instance;

  Future<bool> isAuthenticated() async {
    final currentUser = await _firebaseAuth.currentUser();
    return currentUser != null;
  }

  Future<void> authenticate() {
    return _firebaseAuth.signInAnonymously();
  }

  Future<String> getUserId() async {
    return (await _firebaseAuth.currentUser()).uid;
  }
}
```

That's it for our `UserRepository`, next we need to setup our Flutter app to use our new repositories.

Flutter App

Setup

Let's create a new Flutter app called `flutter_firestore_todos`. We can replace the contents of the `pubspec.yaml` with the following:

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

version: 1.0.0+1

environment:
  sdk: ">=2.6.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  flutter_bloc: ^5.0.0
  todos_repository:
    path: todos_repository
  user_repository:
    path: user_repository
  equatable: ^1.0.0

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

Note: We're adding our `todos_repository` and `user_repository` as external dependencies.

Authentication Bloc

Since we want to be able to sign in our users, we'll need to create an `AuthenticationBloc`.

If you haven't already checked out the flutter firebase login tutorial, I highly recommend checking it out now because we're simply going to reuse the same `AuthenticationBloc`.

Authentication Events

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

abstract class AuthenticationEvent extends Equatable {}

class AppStarted extends AuthenticationEvent {
  @override
  List<Object> get props => [];
}
```

Authentication States

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

abstract class AuthenticationState extends Equatable {
  const AuthenticationState();

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

class Uninitialized extends AuthenticationState {}

class Authenticated extends AuthenticationState {
  final String userId;

  const Authenticated(this.userId);

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

  @override
  String toString() => 'Authenticated { userId: $userId }';
}

class Unauthenticated extends AuthenticationState {}
```

Authentication Bloc

```dart
import 'dart:async';
import 'package:bloc/bloc.dart';
import 'package:meta/meta.dart';
import 'package:user_repository/user_repository.dart';
import 'package:flutter_firestore_todos/blocs/authentication_bloc/bloc.da

class AuthenticationBloc
    extends Bloc<AuthenticationEvent, AuthenticationState> {
  final UserRepository _userRepository;

  AuthenticationBloc({@required UserRepository userRepository})
      : assert(userRepository != null),
        _userRepository = userRepository,
        super(Uninitialized());

  @override
  Stream<AuthenticationState> mapEventToState(
    AuthenticationEvent event,
  ) async* {
    if (event is AppStarted) {
      yield* _mapAppStartedToState();
    }
  }

  Stream<AuthenticationState> _mapAppStartedToState() async* {
    try {
      final isSignedIn = await _userRepository.isAuthenticated();
      if (!isSignedIn) {
        await _userRepository.authenticate();
      }
      final userId = await _userRepository.getUserId();
      yield Authenticated(userId);
    } catch (_) {
      yield Unauthenticated();
    }
  }
}
```

Now that our `AuthenticationBloc` is finished, we need to modify the `TodosBloc` from the original Todos Tutorial to consume the new `TodosRepository`.

Todos Bloc

```dart
import 'dart:async';
import 'package:bloc/bloc.dart';
import 'package:meta/meta.dart';
import 'package:flutter_firestore_todos/blocs/todos/todos.dart';
import 'package:todos_repository/todos_repository.dart';

class TodosBloc extends Bloc<TodosEvent, TodosState> {
  final TodosRepository _todosRepository;
  StreamSubscription _todosSubscription;

  TodosBloc({@required TodosRepository todosRepository})
      : assert(todosRepository != null),
        _todosRepository = todosRepository,
        super(TodosLoading());

  @override
  Stream<TodosState> mapEventToState(TodosEvent event) async* {
    if (event is LoadTodos) {
      yield* _mapLoadTodosToState();
    } else if (event is AddTodo) {
      yield* _mapAddTodoToState(event);
    } else if (event is UpdateTodo) {
      yield* _mapUpdateTodoToState(event);
    } else if (event is DeleteTodo) {
      yield* _mapDeleteTodoToState(event);
    } else if (event is ToggleAll) {
      yield* _mapToggleAllToState();
    } else if (event is ClearCompleted) {
      yield* _mapClearCompletedToState();
    } else if (event is TodosUpdated) {
      yield* _mapTodosUpdateToState(event);
    }
  }

  Stream<TodosState> _mapLoadTodosToState() async* {
    _todosSubscription?.cancel();
```

```
    _todosSubscription = _todosRepository.todos().listen(
          (todos) => add(TodosUpdated(todos)),
        );
  }

  Stream<TodosState> _mapAddTodoToState(AddTodo event) async* {
    _todosRepository.addNewTodo(event.todo);
  }

  Stream<TodosState> _mapUpdateTodoToState(UpdateTodo event) async* {
    _todosRepository.updateTodo(event.updatedTodo);
  }

  Stream<TodosState> _mapDeleteTodoToState(DeleteTodo event) async* {
    _todosRepository.deleteTodo(event.todo);
  }

  Stream<TodosState> _mapToggleAllToState() async* {
    final currentState = state;
    if (currentState is TodosLoaded) {
      final allComplete = currentState.todos.every((todo) => todo.complet
      final List<Todo> updatedTodos = currentState.todos
          .map((todo) => todo.copyWith(complete: !allComplete))
          .toList();
      updatedTodos.forEach((updatedTodo) {
        _todosRepository.updateTodo(updatedTodo);
      });
    }
  }

  Stream<TodosState> _mapClearCompletedToState() async* {
    final currentState = state;
    if (currentState is TodosLoaded) {
      final List<Todo> completedTodos =
          currentState.todos.where((todo) => todo.complete).toList();
      completedTodos.forEach((completedTodo) {
        _todosRepository.deleteTodo(completedTodo);
      });
    }
  }

  Stream<TodosState> _mapTodosUpdateToState(TodosUpdated event) async* {
    yield TodosLoaded(event.todos);
```

```
  }

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

The main difference between our new `TodosBloc` and the original one is in the new one, everything is `Stream` based rather than `Future` based.

```dart
Stream<TodosState> _mapLoadTodosToState() async* {
  _todosSubscription?.cancel();
  _todosSubscription = _todosRepository.todos().listen(
      (todos) => add(TodosUpdated(todos)),
    );
}
```

When we load our todos, we are subscribing to the `TodosRepository` and every time a new todo comes in, we add a `TodosUpdated` event. We then handle all `TodosUpdates` via:

```dart
Stream<TodosState> _mapTodosUpdateToState(TodosUpdated event) async* {
  yield TodosLoaded(event.todos);
}
```

Putting it all together

The last thing we need to modify is our `main.dart`.

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

```dart
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:flutter_firestore_todos/blocs/authentication_bloc/bloc.da
import 'package:todos_repository/todos_repository.dart';
import 'package:flutter_firestore_todos/blocs/blocs.dart';
import 'package:flutter_firestore_todos/screens/screens.dart';
import 'package:user_repository/user_repository.dart';

void main() {
  Bloc.observer = SimpleBlocObserver();
  runApp(TodosApp());
}

class TodosApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiBlocProvider(
      providers: [
        BlocProvider<AuthenticationBloc>(
          create: (context) {
            return AuthenticationBloc(
              userRepository: FirebaseUserRepository(),
            )..add(AppStarted());
          },
        ),
        BlocProvider<TodosBloc>(
          create: (context) {
            return TodosBloc(
              todosRepository: FirebaseTodosRepository(),
            )..add(LoadTodos());
          },
        )
      ],
      child: MaterialApp(
        title: 'Firestore Todos',
        routes: {
          '/': (context) {
            return BlocBuilder<AuthenticationBloc, AuthenticationState>(
              builder: (context, state) {
                if (state is Authenticated) {
                  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(),
                );
              }
              if (state is Unauthenticated) {
                return Center(
                  child: Text('Could not authenticate with Firestore'),
                );
              }
              return Center(child: CircularProgressIndicator());
            },
          );
        },
        '/addTodo': (context) {
          return AddEditScreen(
            onSave: (task, note) {
              BlocProvider.of<TodosBloc>(context).add(
                AddTodo(Todo(task, note: note)),
              );
            },
            isEditing: false,
          );
        },
      },
    ),
  );
 }
}
```

The main differences to note are the fact that we've wrapped our entire application in a `MultiBlocProvider` which initializes and provides the `AuthenticationBloc` and `TodosBloc`. We then, only render the todos app if the `AuthenticationState` is `Authenticated` using `BlocBuilder`. Everything else remains the same as in the previous `todos tutorial`.

That's all there is to it! We've now successfully implemented a firestore todos app in flutter using the bloc and flutter_bloc packages and we've successfully separated our presentation layer from our business logic while also building an app that updates in real-time.

The full source for this example can be found here.

PREVIOUS
Firebase Login ✨