**Implementa Arquitectura a tu proyecto Flutter usando el patrón BLOC**

[](https://medium.com/@cesavegl?source=post_elevate_sequence_page---------------------------)

[Cesar Vega](https://medium.com/@cesavegl?source=post_elevate_sequence_page---------------------------)

[Mar 23](https://medium.com/comunidad-flutter/implementa-arquitectura-a-tu-proyecto-flutter-usando-el-patrón-bloc-2cb031722166?source=post_elevate_sequence_page---------------------------) · 11 min read

*Este artículo es una traducción de un artículo publicado originalmente en inglés por* [**Sagar Suri**](https://medium.com/@sagarsuri56?source=post_elevate_sequence_page---------------------------) *en* [Medium](https://medium.com/?source=post_elevate_sequence_page---------------------------)*. Por favor, visita el siguiente* [enlace](https://medium.com/flutterpub/architecting-your-flutter-project-bd04e144a8f1?source=post_elevate_sequence_page---------------------------) *y recomienda el artículo original si te gusta el contenido:*

¡Hola amigos! Estoy de vuelta con otro nuevo artículo sobre Flutter. En esta ocasión hablaré y les demostraré “*cómo arquitectar a sus proyectos Flutter*”. De esta forma podrás mantener, escalar y probar tus proyectos Flutter fácilmente. Antes de sumergirnos en el tema actual, me gustaría compartir una pequeña historia sobre por qué debemos centrarnos en la construcción de una arquitectura sólida de nuestros proyectos.

**Actualización**: La Parte 2 de este artículo ha sido publicada con algunos cambios en el diseño actual para tratar algunos temas y mostrar algunas implementaciones sorprendentes. Aquí está el [enlace](https://medium.com/@cesavegl/implementa-arquitectura-a-tu-proyecto-flutter-usando-el-patrón-bloc-parte-2-91a8956980b4?source=post_elevate_sequence_page---------------------------) ([En inglés](https://medium.com/flutterpub/architect-your-flutter-project-using-bloc-pattern-part-2-d8dd1eca9ba5?source=post_elevate_sequence_page---------------------------)).

**¿Por qué necesitas arquitectar tu proyecto?**

“Érase una vez en el año 2015. Yo era un programador competitivo novato (perfil de [Hackerearth](http://www.hackerearth.com/@sagarsuri56?source=post_elevate_sequence_page---------------------------)) y también estaba aprendiendo el desarrollo de aplicaciones android. Como programador competitivo, sólo me importaba el rendimiento y la eficiencia del programa que escribí. Nunca pensé en estructurar mis programas o proyectos que escribí. Esta tendencia y estilo se refleja también en mis proyectos de Android. Estaba escribiendo aplicaciones para Android con una mentalidad de programador competitivo. Al principio, cuando estaba trabajando en mis propios proyectos, todo estaba bien, ya que nunca tuve un jefe o gerente que pudiera darme los requisitos para añadir nuevas características o cambiar las existentes en la aplicación. Cuando empecé a trabajar en una startup y a crear aplicaciones Android para ellos. Siempre me tomé mucho tiempo para cambiar una característica existente en la aplicación. No sólo eso, incluso he añadido errores como una funcionalidad adicional en el proceso de creación de la aplicación. La causa principal de todos estos problemas fue: “**Nunca seguí ningún patrón arquitectónico ni estructuré mis proyectos**”. A medida que pasó el tiempo y empecé a entender el mundo del software, pasé de ser un programador competitivo a un ingeniero de software. Hoy en día, cada vez que comienzo un nuevo proyecto, mi objetivo principal es construir una estructura o arquitectura sólida del proyecto. Eso me ayudó a ser un mejor y más estresante ingeniero de software libre 😄”.

Poniendo fin a mi aburrida historia 😅. Empecemos a trabajar en el objetivo principal de este artículo. “ Implementa arquitectura a tu proyecto Flutter usando el patrón BLOC”.

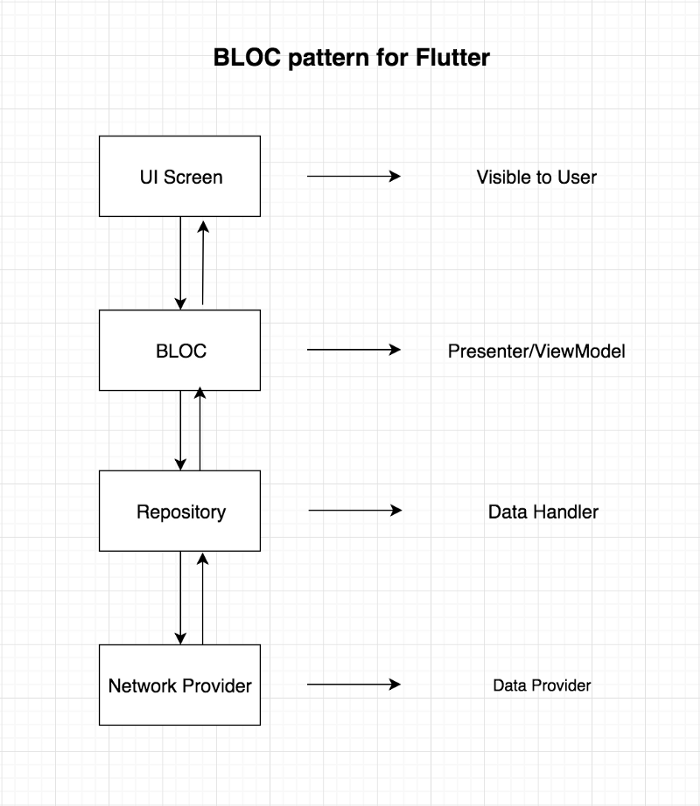
**Nuestra meta**

Estaré construyendo una aplicación muy simple. Tendrá una pantalla en la que podrá ver una lista de elementos de la grilla. Estos elementos se obtendrán del servidor. La lista de artículos serán películas populares del sitio de [**The Movies DB**](https://www.themoviedb.org/?source=post_elevate_sequence_page---------------------------).

**Nota**: Antes de seguir adelante, **asumo que entiendes los** [**Widgets**](https://docs.flutter.io/flutter/widgets/Widget-class.html?source=post_elevate_sequence_page---------------------------)**,** [**cómo hacer una llamada de red en Flutter**](https://medium.com/flutterpub/making-a-network-call-in-flutter-f712f2137109?source=post_elevate_sequence_page---------------------------) **y tener conocimientos intermedios en Dart**. Este artículo será poco extenso y estará muy vinculado a otros recursos para que puedas leer más sobre temas específicos.

*Así que vamos a empezar el espectáculo*. 😍

Antes de sumergirse directamente en el código. Permítanme darles una experiencia visual de la arquitectura que seguiremos para estructurar esta aplicación.



El Patrón BLOC

El diagrama anterior muestra cómo fluyen los datos de la interfaz de usuario a la capa de datos y viceversa. BLOC nunca tendrá ninguna referencia de los Widgets en la pantalla de la interfaz de usuario. La pantalla UI sólo observará los cambios procedentes de la clase BLOC. Hagamos una pequeña pregunta y respuesta para entender este diagrama:

**1. ¿Qué es el patrón BLOC?**

Es un sistema de gestión de estado para Flutter recomendado por los desarrolladores de Google. Ayuda en la gestión del estado y en el acceso a los datos desde un lugar central en su proyecto.

**2. ¿Puedo co-relacionar esta arquitectura con cualquier otra arquitectura?**

Sí, por supuesto. MVP y MVVM son un buen ejemplo. Lo único que cambiará es: BLOC será reemplazado por ViewModel en MVVM.

**3. ¿Qué hay bajo el manto del Bloc? o ¿Cuál es esa cosa central que maneja el estado en un solo lugar?**

Enfoque STREAMS o REACTIVE. En términos generales, los datos fluirán del Bloc a la UI o de la UI al Bloc en forma de flujos. Si nunca has oído hablar de los streams. Lee esta respuesta de [Stack Overflow](https://stackoverflow.com/a/1216397/8327394?source=post_elevate_sequence_page---------------------------).

Espero que esta pequeña sección de preguntas y respuestas haya aclarado algunas de tus dudas. Si necesitas más claridad o quieres hacer una pregunta específica, puedes comentar abajo o conectarte directamente conmigo en [LinkedIn](https://www.linkedin.com/in/sagar-suri/?source=post_elevate_sequence_page---------------------------).

**Comencemos a construir el proyecto con el patrón BLOC**

1. Primero crea un nuevo proyecto y borra todo el código del archivo main.dart. Escribe abajo el comando en tu terminal:

flutter create myProjectName

2. Escribe el siguiente código en tu archivo main.dart:

**import** 'package:flutter/material.dart';  
**import** 'src/app.dart';  
  
**void** main(){  
 runApp(App());  
}

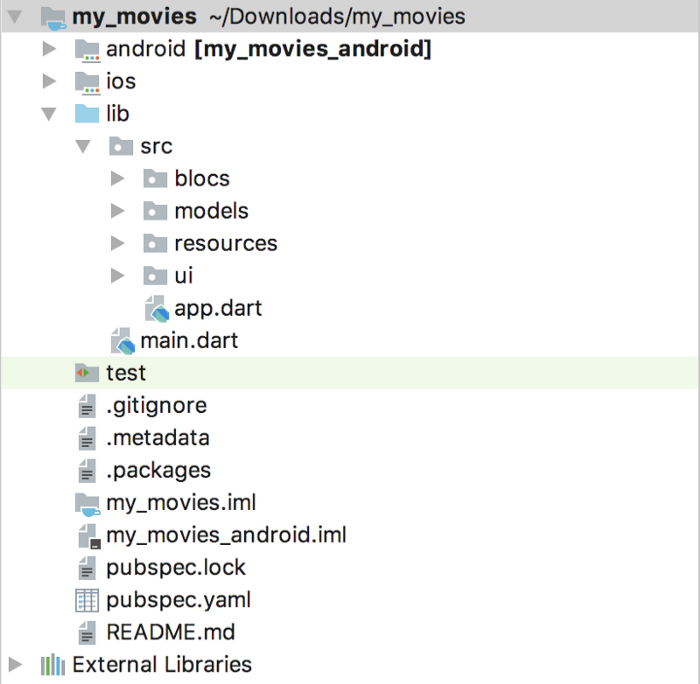
Debes estar recibiendo un error en la segunda línea. Lo resolveremos en los próximos pasos.

3. Crea un paquete **src** bajo el paquete **lib**. Dentro del paquete src creas un archivo y le asignas como nombre **app.dart**. Copia y pega el siguiente código dentro del archivo app.dart.

**import** 'package:flutter/material.dart';  
**import** 'ui/movie\_list.dart';  
  
**class App extends StatelessWidget** {  
 @override  
 **Widget** build(**BuildContext** context) {  
 // *TODO: implement build* **return** MaterialApp(  
 theme: ThemeData.dark(),  
 home: Scaffold(  
 body: MovieList(),  
 ),  
 );  
 }  
}

4. Crea un nuevo paquete dentro del paquete src y nombrarlo como **resources**.

Ahora crea algunos paquetes nuevos, es decir, bloques, modelos, recursos y ui, como se muestra en el siguiente diagrama, y luego nos quedaremos con el esqueleto del proyecto:



Estructura del Projecto

El paquete **blocs** contendrá nuestros archivos relacionados con la implementación de BLOC. el paquete **models** contendrá la clase POJO o la clase modelo de la respuesta JSON que obtendremos del servidor. el paquete **resources** contendrá la clase repositorio y la clase implementada de la llamada de red. el paquete **ui** contendrá nuestras pantallas que serán visibles para el usuario.

5. Una última cosa que tenemos que añadir es [RxDart](https://pub.dartlang.org/packages/rxdart?source=post_elevate_sequence_page---------------------------), una biblioteca de terceros. Abre tu archivo **pubspec.yaml** y añade **rxdart**: ^0.18.0 como se muestra abajo:

**dependencies:  
 flutter:  
 sdk:** flutter  
  
 # The following adds the Cupertino Icons font to your application.  
 # Use with the CupertinoIcons class for iOS style icons.  
 **cupertino\_icons:** ^0.1.2  
 **rxdart:** ^0.18.0  
 **http**: ^0.12.0+1

sincroniza tu proyecto o escribe el comando de abajo en la terminal. Asegúrate de ejecutar este comando dentro del directorio de tu proyecto.

flutter packages get

6. Ahora estamos completos con el esqueleto del proyecto. Tiempo para ocuparse de la capa más baja del proyecto, es decir, la **capa de red**. Entendamos el punto final de la API que vamos a consumir. Pulsa este [enlace](https://www.themoviedb.org/account/signup?source=post_elevate_sequence_page---------------------------) y serás llevado al sitio de la API de la base de datos de películas. Regístrate y obtén tu clave de API en la página Configuración. Estaremos pulsando la url de abajo para obtener la respuesta:

[http://api.themoviedb.org/3/movie/popular?api\_key=“](http://api.themoviedb.org/3/movie/popular?api_key=802b2c4b88ea1183e50e6b285a27696e&source=post_elevate_sequence_page---------------------------)your\_api\_key”

Pon tu clave de API en el enlace anterior y haz clic en (Eliminar también las comillas dobles). Puedes ver la respuesta de JSON algo así:

{  
 "page": 1,  
 "total\_results": 19772,  
 "total\_pages": 989,  
 "results": [  
 {  
 "vote\_count": 6503,  
 "id": 299536,  
 "video": false,  
 "vote\_average": 8.3,  
 "title": "Avengers: Infinity War",  
 "popularity": 350.154,  
 "poster\_path": "\/7WsyChQLEftFiDOVTGkv3hFpyyt.jpg",  
 "original\_language": "en",  
 "original\_title": "Avengers: Infinity War",  
 "genre\_ids": [  
 12,  
 878,  
 14,  
 28  
 ],  
 "backdrop\_path": "\/bOGkgRGdhrBYJSLpXaxhXVstddV.jpg",  
 "adult": false,  
 "overview": "As the Avengers and their allies have continued to protect the world from threats too large for any one hero to handle, a new danger has emerged from the cosmic shadows: Thanos. A despot of intergalactic infamy, his goal is to collect all six Infinity Stones, artifacts of unimaginable power, and use them to inflict his twisted will on all of reality. Everything the Avengers have fought for has led up to this moment - the fate of Earth and existence itself has never been more uncertain.",  
 "release\_date": "2018-04-25"  
 },

7. Vamos a construir un modelo o clase POJO para este tipo de respuesta. Crea un nuevo archivo dentro del paquete **models** y nómbralo como **item\_model.dart**. Copia y pega el siguiente código dentro del archivo item\_model.dart:

**class ItemModel** {  
 **int** \_page;  
 **int** \_total\_results;  
 **int** \_total\_pages;  
 **List**<**\_Result**> \_results = [];  
  
 **ItemModel**.fromJson(**Map**<**String**, **dynamic**> parsedJson) {  
 print(parsedJson['results'].length);  
 \_page = parsedJson['page'];  
 \_total\_results = parsedJson['total\_results'];  
 \_total\_pages = parsedJson['total\_pages'];  
 **List**<**\_Result**> temp = [];  
 **for** (**int** i = 0; i < parsedJson['results'].length; i++) {  
 **\_Result** result = \_Result(parsedJson['results'][i]);  
 temp.add(result);  
 }  
 \_results = temp;  
 }  
  
 **List**<**\_Result**> **get** results => \_results;  
  
 **int get** total\_pages => \_total\_pages;  
  
 **int get** total\_results => \_total\_results;  
  
 **int get** page => \_page;  
}  
  
**class \_Result** {  
 **int** \_vote\_count;  
 **int** \_id;  
 **bool** \_video;  
 **var** \_vote\_average;  
 **String** \_title;  
 **double** \_popularity;  
 **String** \_poster\_path;  
 **String** \_original\_language;  
 **String** \_original\_title;  
 **List**<**int**> \_genre\_ids = [];  
 **String** \_backdrop\_path;  
 **bool** \_adult;  
 **String** \_overview;  
 **String** \_release\_date;  
  
 **\_Result**(result) {  
 \_vote\_count = result['vote\_count'];  
 \_id = result['id'];  
 \_video = result['video'];  
 \_vote\_average = result['vote\_average'];  
 \_title = result['title'];  
 \_popularity = result['popularity'];  
 \_poster\_path = result['poster\_path'];  
 \_original\_language = result['original\_language'];  
 \_original\_title = result['original\_title'];  
 **for** (**int** i = 0; i < result['genre\_ids'].length; i++) {  
 \_genre\_ids.add(result['genre\_ids'][i]);  
 }  
 \_backdrop\_path = result['backdrop\_path'];  
 \_adult = result['adult'];  
 \_overview = result['overview'];  
 \_release\_date = result['release\_date'];  
 }  
  
 **String get** release\_date => \_release\_date;  
  
 **String get** overview => \_overview;  
  
 **bool get** adult => \_adult;  
  
 **String get** backdrop\_path => \_backdrop\_path;  
  
 **List**<**int**> **get** genre\_ids => \_genre\_ids;  
  
 **String get** original\_title => \_original\_title;  
  
 **String get** original\_language => \_original\_language;  
  
 **String get** poster\_path => \_poster\_path;  
  
 **double get** popularity => \_popularity;  
  
 **String get** title => \_title;  
  
 **double get** vote\_average => \_vote\_average;  
  
 **bool get** video => \_video;  
  
 **int get** id => \_id;  
  
 **int get** vote\_count => \_vote\_count;  
}

Espero que puedas mapear este archivo con la respuesta de JSON. Si no es así, estamos interesados en el **poster\_path** dentro de la clase **Results** que es lo que necesitas saber para avanzar. Mostraremos todos los posters de las películas más populares en nuestra interfaz de usuario principal. El método fromJson()es simplemente obtener el json decodificado y poner los valores en las variables correctas.

8. Ahora es el momento de trabajar en la implementación de la red. Crea un archivo dentro del paquete de **resources** y nómbralo como **movie\_api\_provider.dart**. Copia y pega el siguiente código dentro del archivo y te lo explicaré:

**import** 'dart:async';  
**import** 'package:http/http.dart' **show Client**;  
**import** 'dart:convert';  
**import** '../models/item\_model.dart';  
  
**class MovieApiProvider** {  
 **Client** client = Client();  
 **final** \_apiKey = 'your\_api\_key';  
  
 **Future**<**ItemModel**> fetchMovieList() **async** {  
 print("entered");  
 **final** response = **await** client  
 .get("http://api.themoviedb.org/3/movie/popular?api\_key=$\_apiKey");  
 print(response.body.toString());  
 **if** (response.statusCode == 200) {  
 // If the call to the server was successful, parse the JSON  
 **return** ItemModel.fromJson(json.decode(response.body));  
 } **else** {  
 // If that call was not successful, throw an error.  
 **throw** Exception('Failed to load post');  
 }  
 }  
}

Nota: Por favor coloca tu clave API en la variable \_apiKey dentro del archivo **movie\_api\_provider.dart** o no funcionará.

fetchMovieList()es hacer la llamada de red a la API. Una vez completada la llamada de red, se devolverá un objeto [Future](https://api.dartlang.org/stable/2.0.0/dart-async/Future-class.html?source=post_elevate_sequence_page---------------------------) ItemModel si la llamada de red se ha realizado correctamente o se lanzará una Excepción.

9. A continuación vamos a crear un nuevo archivo dentro del paquete de **resources** y lo llamaremos **repository.dart**. Copia y pega el siguiente código dentro del archivo:

**import** 'dart:async';  
**import** 'movie\_api\_provider.dart';  
**import** '../models/item\_model.dart';  
  
**class Repository** {  
 **final** moviesApiProvider = MovieApiProvider();  
  
 **Future**<**ItemModel**> fetchAllMovies() => moviesApiProvider.fetchMovieList();  
}

Estamos importando el archivo **movie\_api\_provider.dart** y llamando a su método fetchMovieList(). Esta clase Repository es el punto central desde donde los datos fluirán hacia el Bloc.

10. Ahora viene una parte un poco complicada. Implementación de la lógica de bloc. Vamos a crear un nuevo archivo dentro del paquete **blocs** y lo llamaremos **movies\_bloc.dart**. Copia y pega debajo del código y te explicaré el código con poco detalle:

**mport** '../resources/repository.dart';  
**import** 'package:rxdart/rxdart.dart';  
**import** '../models/item\_model.dart';  
  
**class MoviesBloc** {  
 **final** \_repository = Repository();  
 **final** \_moviesFetcher = PublishSubject<**ItemModel**>();  
  
 **Observable**<**ItemModel**> **get** allMovies => \_moviesFetcher.stream;  
  
 fetchAllMovies() **async** {  
 **ItemModel** itemModel = **await** \_repository.fetchAllMovies();  
 \_moviesFetcher.sink.add(itemModel);  
 }  
  
 dispose() {  
 \_moviesFetcher.close();  
 }  
}  
  
**final** bloc = MoviesBloc();

Estamos importando un package import ‘package:rxdart/rxdart.dart'; el cual eventualmente importará todos los métodos y clases relacionados con [RxDart](https://pub.dartlang.org/packages/rxdart?source=post_elevate_sequence_page---------------------------) en este archivo. Dentro de la clase MoviesBloc estamos creando el objeto de clase Repository que se utilizará para acceder a fetchAllMovies().(). Estamos creando un objeto [PublishSubject](https://pub.dartlang.org/documentation/rxdart/latest/rx/PublishSubject-class.html?source=post_elevate_sequence_page---------------------------) cuya responsabilidad es añadir los datos que obtuvo del servidor en forma de objeto ItemModel y pasarlos a la pantalla de la interfaz de usuario como stream. Para pasar el objeto ItemModel como stream hemos creado otro método AllMovies()cuyo tipo de retorno es [Observable](https://youtu.be/XbOuCBuQepI?source=post_elevate_sequence_page---------------------------) (mira este vídeo si no entiendes Observables). Si ves la última línea estamos creando el objeto bloque. De esta manera estamos dando acceso a una única instancia de la clase MoviesBloc a la pantalla de la interfaz de usuario.

Si no sabes lo que es la programación reactiva. Por favor, lee esta sencilla [explicación](https://dzone.com/articles/only-introduction-reactive?source=post_elevate_sequence_page---------------------------). En pocas palabras, siempre que haya nuevos datos procedentes del servidor. Tenemos que actualizar la pantalla UI. Para que esta tarea de actualización sea sencilla, le estamos diciendo a la pantalla de la interfaz de usuario que siga observando cualquier cambio que provenga de la clase MoviesBloc y que actualice su contenido en consecuencia. Esta “observación” de nuevos datos puede hacerse usando RxDart.

11. Ahora la última sección. Crea un nuevo archivo dentro del paquete **ui** y nómbralo como **movie\_list.dart**. Copia y pega el código de abajo:

**import** 'package:flutter/material.dart';  
**import** '../models/item\_model.dart';  
**import** '../blocs/movies\_bloc.dart';  
  
**class MovieList extends StatelessWidget** {  
 @override  
 **Widget** build(**BuildContext** context) {  
 bloc.fetchAllMovies();  
 **return** Scaffold(  
 appBar: AppBar(  
 title: Text('Popular Movies'),  
 ),  
 body: StreamBuilder(  
 stream: bloc.allMovies,  
 builder: (context, **AsyncSnapshot**<**ItemModel**> snapshot) {  
 **if** (snapshot.hasData) {  
 **return** buildList(snapshot);  
 } **else if** (snapshot.hasError) {  
 **return** Text(snapshot.error.toString());  
 }  
 **return** Center(child: CircularProgressIndicator());  
 },  
 ),  
 );  
 }  
  
 **Widget** buildList(**AsyncSnapshot**<**ItemModel**> snapshot) {  
 **return** GridView.builder(  
 itemCount: snapshot.data.results.length,  
 gridDelegate:  
 **new** SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 2),  
 itemBuilder: (**BuildContext** context, **int** index) {  
 **return** Image.network(  
 'https://image.tmdb.org/t/p/w185${snapshot.data  
 .results[index].poster\_path}',  
 fit: **BoxFit**.cover,  
 );  
 });  
 }  
}

*Lo mejor e interesante de esta clase es que no estoy usando un StatefulWidget. Pero en su lugar estoy usando un StreamBuilder que hará el mismo trabajo que StatefulWidget, es decir, actualizar la interfaz de usuario.*

Una cosa a señalar aquí, que *estoy haciendo una llamada de red dentro del método build lo cual* ***no debería hacerse*** ya que el método buildContext() puede ser llamado varias veces. Estaré actualizando este artículo con un mejor enfoque. Pero por ahora, a medida que el artículo se hace más grande y complicado, estoy manteniendo las cosas simples, es decir, haciendo la llamada de red dentro del método buildContext().

Como les dije, nuestra clase MoviesBloc está pasando los nuevos datos como stream. Así que para tratar con los streams tenemos una buena clase incorporada, es decir, StreamBuilder, que escuchará los streams entrantes y actualizará la interfaz de usuario en consecuencia. StreamBuilder está esperando un parámetro streamdonde estamos pasando el método allMovies()de MovieBloc al devolver un stream. Por lo tanto, en el momento en que se reciba un stream de datos, StreamBuilder volverá a mostrar el widget con los datos más recientes. Aquí los datos de la instantánea contienen el objeto ItemModel. Ahora puedes usar cualquier Widget para mostrar lo que sea que haya en el objeto (aquí tu creatividad entra en la imagen). Utilizá un GridView para mostrar todos los pósters que están en la lista de resultados del objeto ItemModel. Es la salida del producto final:

Pequeña demo. El video no estaba capturando los fotogramas completos, supongo 😐

Así que hemos llegado al final de este artículo. Buen trabajo, amigos, por aguantar hasta el final. Espero que les guste este artículo. Si tienes alguna duda o pregunta, conéctate conmigo en [**LinkedIn**](https://www.linkedin.com/in/sagar-suri/?source=post_elevate_sequence_page---------------------------)o [**Twitter**](https://twitter.com/SagarSuri94?source=post_elevate_sequence_page---------------------------). Aprecien el artículo con algunos aplausos y comentarios.

Si quieres el código completo. Aquí está el [repositorio github](https://github.com/SAGARSURI/MyMovies?source=post_elevate_sequence_page---------------------------) del proyecto.

El [**Flutter Pub**](https://medium.com/flutterpub/architect-your-flutter-project-using-bloc-pattern-part-2-d8dd1eca9ba5?source=post_elevate_sequence_page---------------------------)es un medio de publicación para traerle los últimos y sorprendentes recursos como artículos, videos, códigos, podcasts, etc. sobre esta gran tecnología para enseñarle cómo construir hermosas aplicaciones con ella. Puedes encontrarnos en [Facebook](https://www.facebook.com/FlutterPub?source=post_elevate_sequence_page---------------------------), [Twitter](https://twitter.com/FlutterPub?source=post_elevate_sequence_page---------------------------) y [Medium](https://medium.com/flutterpub/architect-your-flutter-project-using-bloc-pattern-part-2-d8dd1eca9ba5?source=post_elevate_sequence_page---------------------------) o aprender más sobre nosotros [aquí](https://medium.com/flutterpub/welcome-to-flutter-pub-8480678ed212?source=post_elevate_sequence_page---------------------------). ¡Nos encantaría conectarnos! Y si eres un escritor interesado en escribir para nosotros, entonces puedes hacerlo a través de estas [pautas](https://medium.com/flutterpub/how-to-submit-your-article-s-on-flutterpub-7b6bf37dfc43?source=post_elevate_sequence_page---------------------------).