

# Introducción al patrón Bloc y al manejo de estado de la aplicación

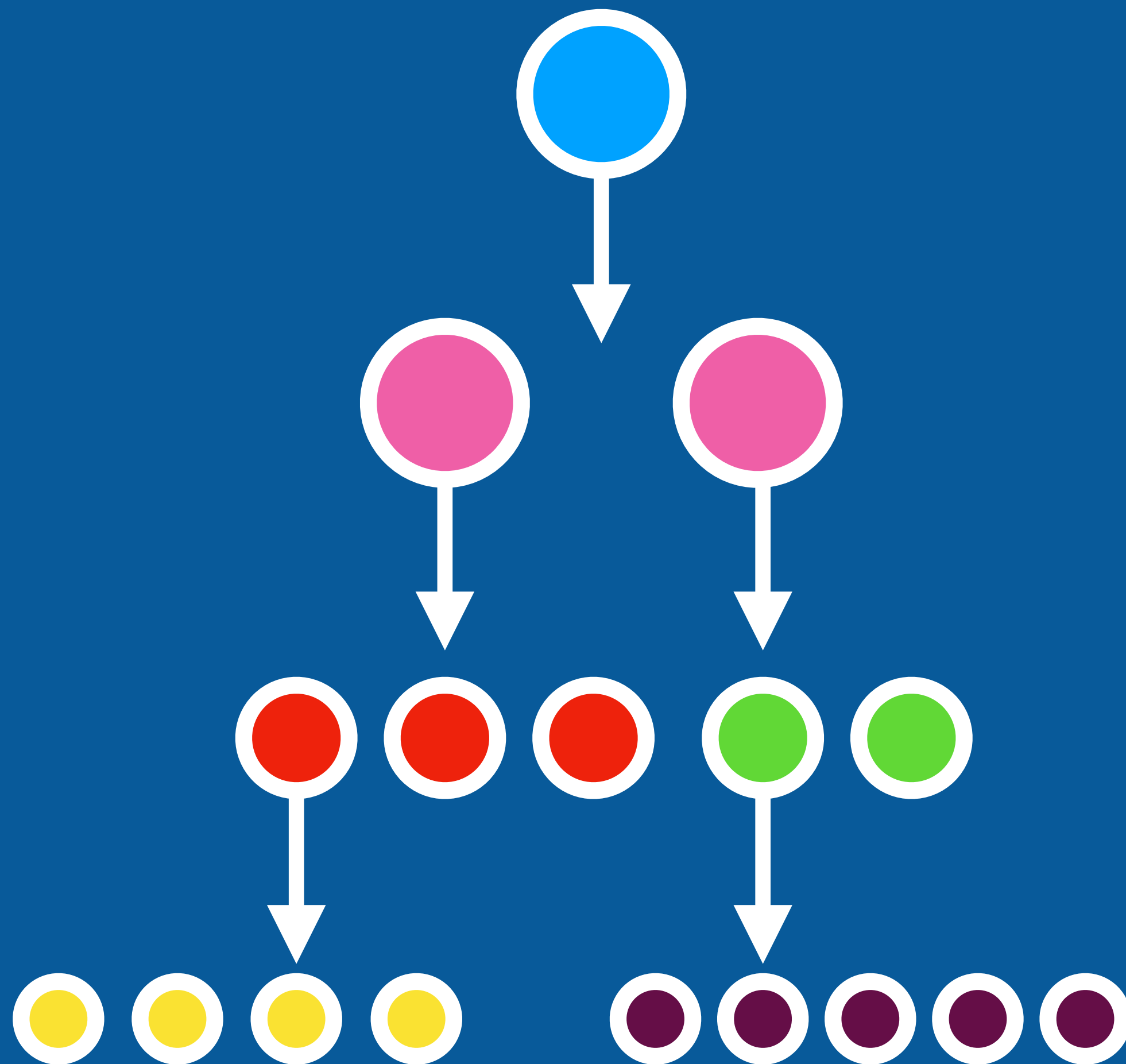
# Patrón Bloc

Es una forma para manejar el estado de la data de nuestra aplicación

- **ScopedModel**
- **Patrón Redux**
- **Inherited Widgets**
- **Provider**
- **Y otros...**

¿Qué es? ¿Por qué usarlo?

# ¿Qué es? ¿Por qué usarlo?



**Bloc**

Usuario

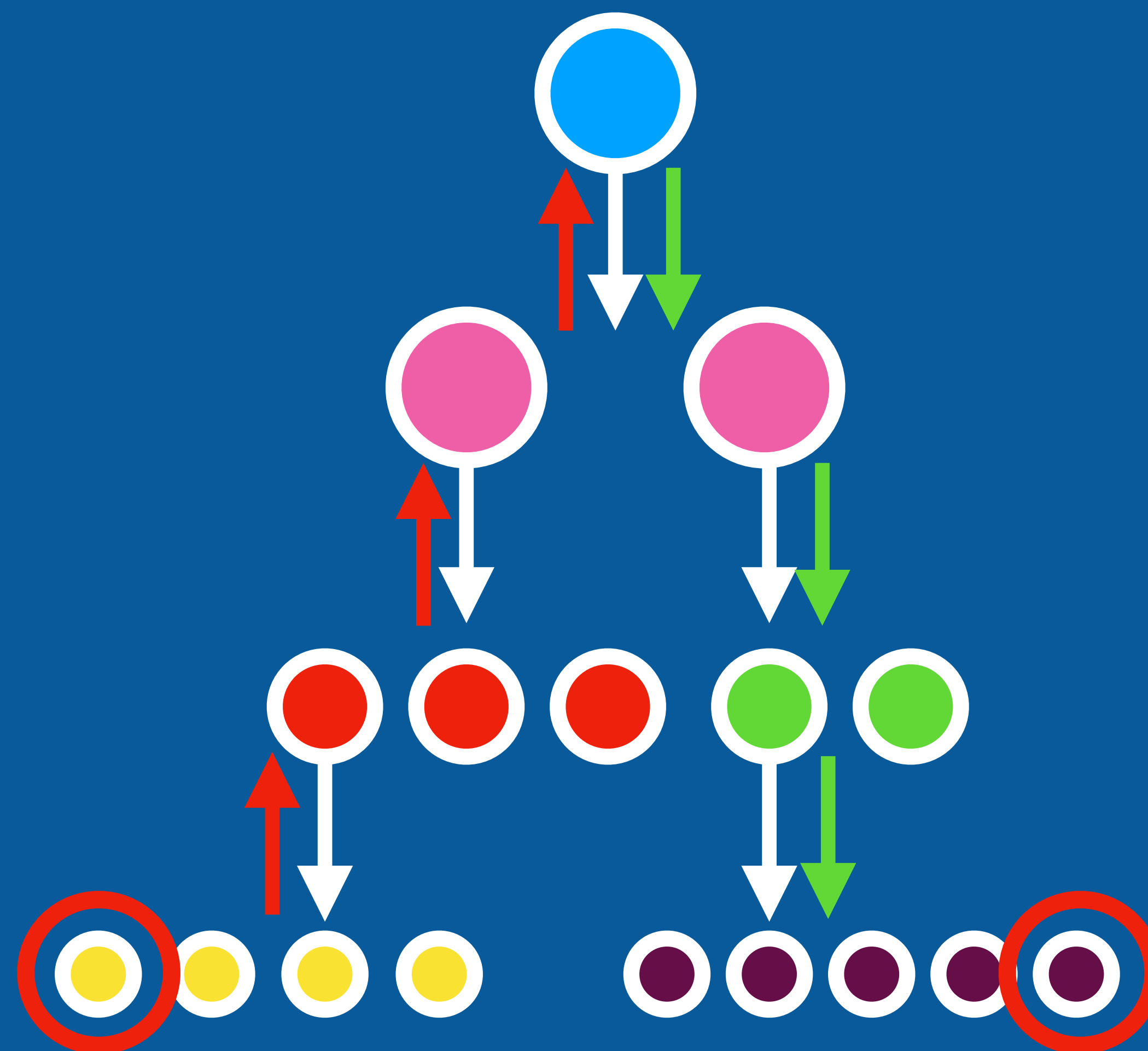
Productos

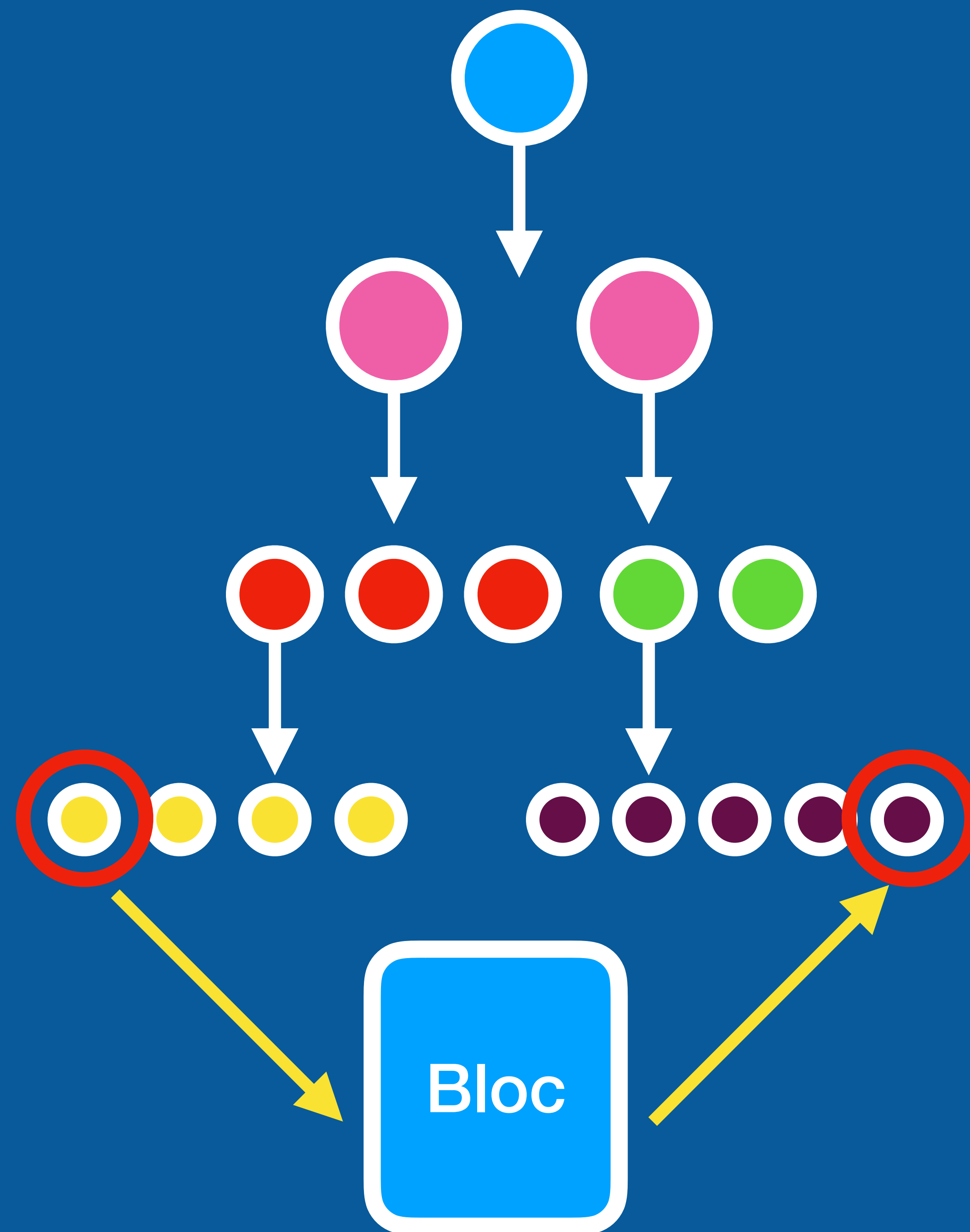
Películas

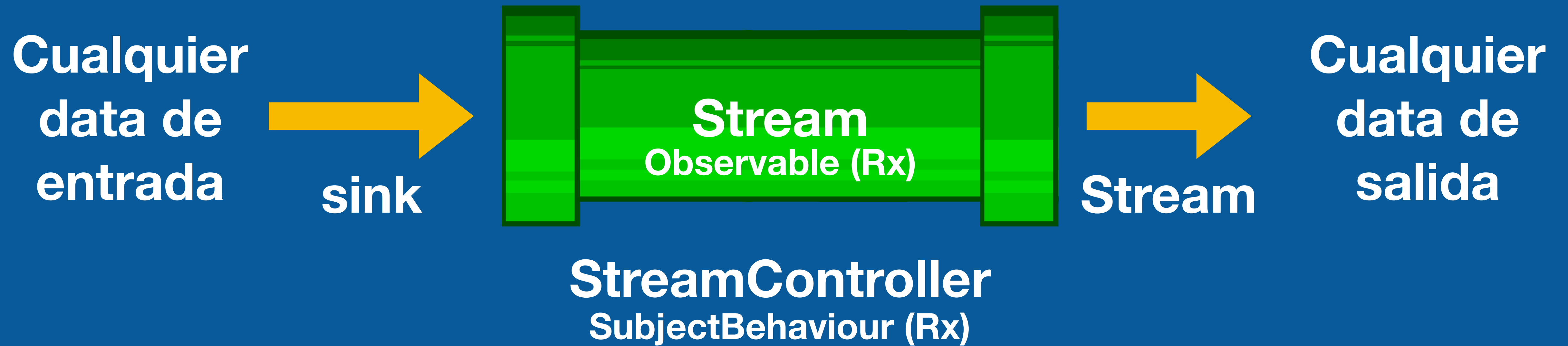
Formulario

**Bloc**

Business Logic Component  
Lógica de negocio de componente







Trabaja en base a Streams

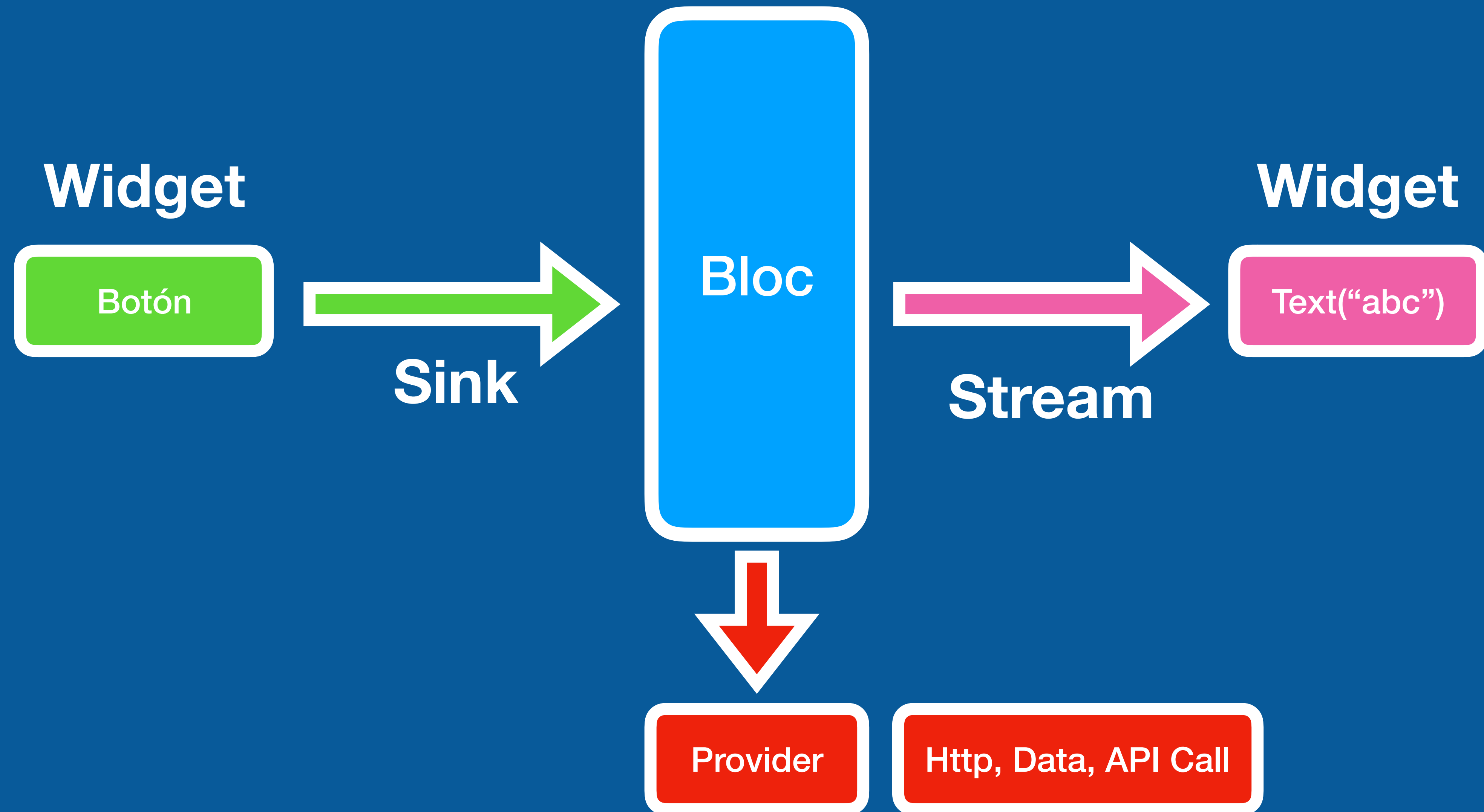


Bloc

Tiene 3 puntos importantes

- Trabaja únicamente con entradas y salidas
- Para introducir información usaremos el sink y para la salida usaremos un stream
- Es importante cerrar el stream cuando ya no lo necesitamos

La idea, es que los Widgets se encarguen únicamente de dibujarse y no de controlar el estado de la información





# Hay dos tipos de StreamControllers


- SingleSubscription
- Broadcast

Widget

Solo yo puedo  
escuchar

```
class MiBloc {  
    StreamController<String> _streamController = new StreamController<String>( );  
}
```

# Ocuparemos 2 cosas, un SINK y un STREAM



```
class MiBloc {  
    StreamController<String> _streamController = new StreamController<String>();  
  
    // ¿Qué clase de información entrará al sink?  
    Sink<String> get inputSink => streamController.sink;  
  
    // ¿Qué clase de información saldrá?  
    Stream<String> get outputStream => streamController.stream;  
}
```

# Para transformar un Stream, usaremos un StreamTransformer



```
class MiBloc {  
  
    StreamController<String> _streamController = new StreamController<String>();  
  
    // ¿Qué clase de información entrará al sink?  
    Sink<String> get inputSink => streamController.sink;  
  
    // ¿Qué clase de información saldrá?  
    Stream<String> get outputStream => streamController.stream.transform( algoParaTransformar );  
  
}
```

# Cerrar el Stream cuando ya no será necesario



```
class MiBloc {  
    ...  
    ...  
    ...  
  
    void dispose() {  
        _streamController?.close();  
    }  
  
}
```

# ¿Cómo funciona esto en Flutter?

**Widget**



StreamBuilder

**Dibujar un widget cada vez que se recibe  
información de un Stream**

# Widget A



```
floatingActionButton(  
  onPressed: () => MyBloc().inputSink.add('Hola Mundo!')
```

# Widget B



```
StreamBuilder(  
  stream: MyBloc.outputStream,  
  builder: (BuildContext context, AsyncSnapshot snapshot) {  
  
    return Container(); // El nuevo Widget a dibujar  
  
  }
```

# Widget B

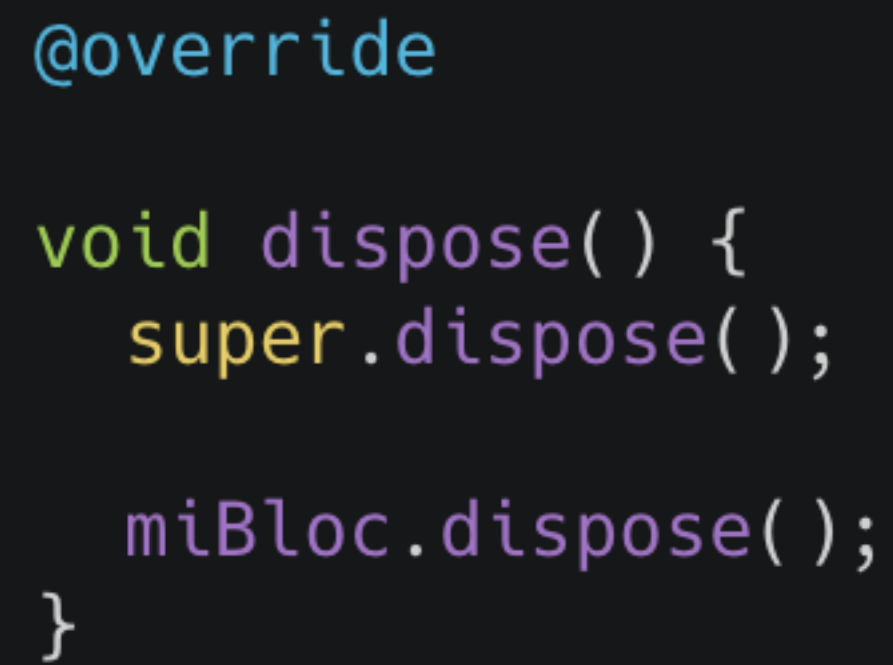


```
StreamBuilder(  
  stream: MyBloc.outputStream,  
  builder: ( BuildContext context, AsyncSnapshot snapshot ) {  
  
    return Container(); // El nuevo Widget a dibujar  
  
  }  
)
```

**Context:** Contiene la información del árbol de widgets y más...

**Snapshot:** Información referente al estado del Stream y la información que sale del Stream

# El lugar típico para llamar el dispose del Bloc



```
@override  
  
void dispose() {  
    super.dispose();  
  
    miBloc.dispose();  
}
```

Es en el paso del ciclo de vida de un StatefulWidget  
**dispose();**