BLoC Software Design

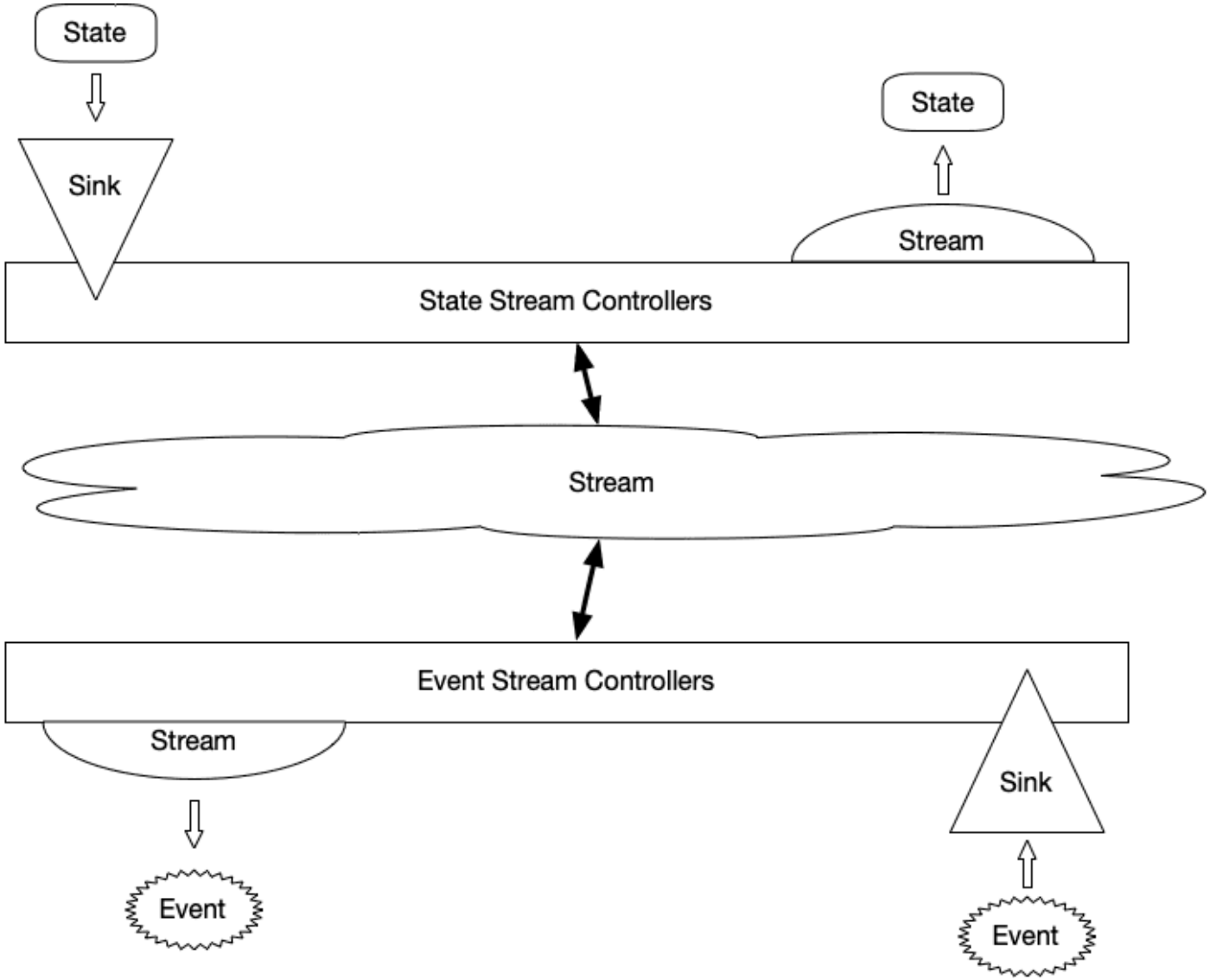
Design Patterns Used in BLoC

- BLoC is similar to Provider but more complicated with added features.
- It uses Dart Stream feature.
- It uses the Observer + Command Design Patterns, and CounterEvent is the Command in this example.

The Process to use BLoC

- We define a state.
- We define an event to alert any change in the state.
- We use a stream to give an input (sink) or listen to an event, so we need a stream controller.

BLoC Architecture Overview



Define State

- We define a state `_counter` in the CounterBloc class.

Code Example:

```
class CounterBloc {
  int _counter = 0;
```

Define Event: CounterEvent

- We define an abstract event (CounterEvent) and concrete event (IncrementEvent)

Code Example:

```
abstract class CounterEvent {}
class IncrementEvent extends CounterEvent {}
```

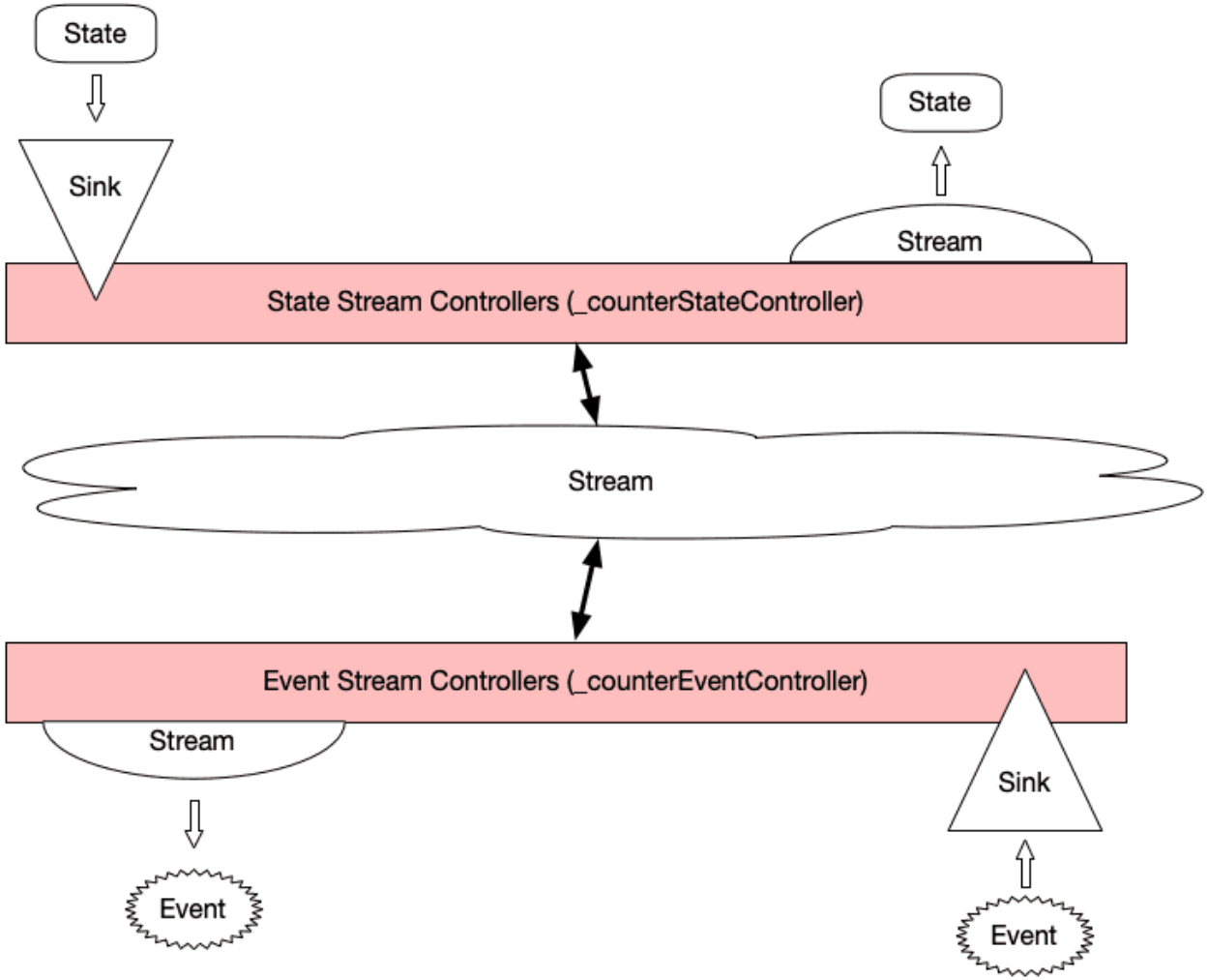
Use the Stream: Stream Controller

- We make two stream controllers.
- `_counterEventController` is for managing events.
- `_counterStateController` is for managing states.

Code Example:

```
final _counterEventController =
  StreamController<CounterEvent>();
final _counterStateController =
  StreamController<int>();
```

Stream Controllers Architecture



Public Properties

- We need to provide the `counterEventSink` property, which is the sink for the event.
- Users use this sink (input) to give an event to the stream.

Code Example:

```
get counterEventSink =>
  _counterEventController.sink;
}
```

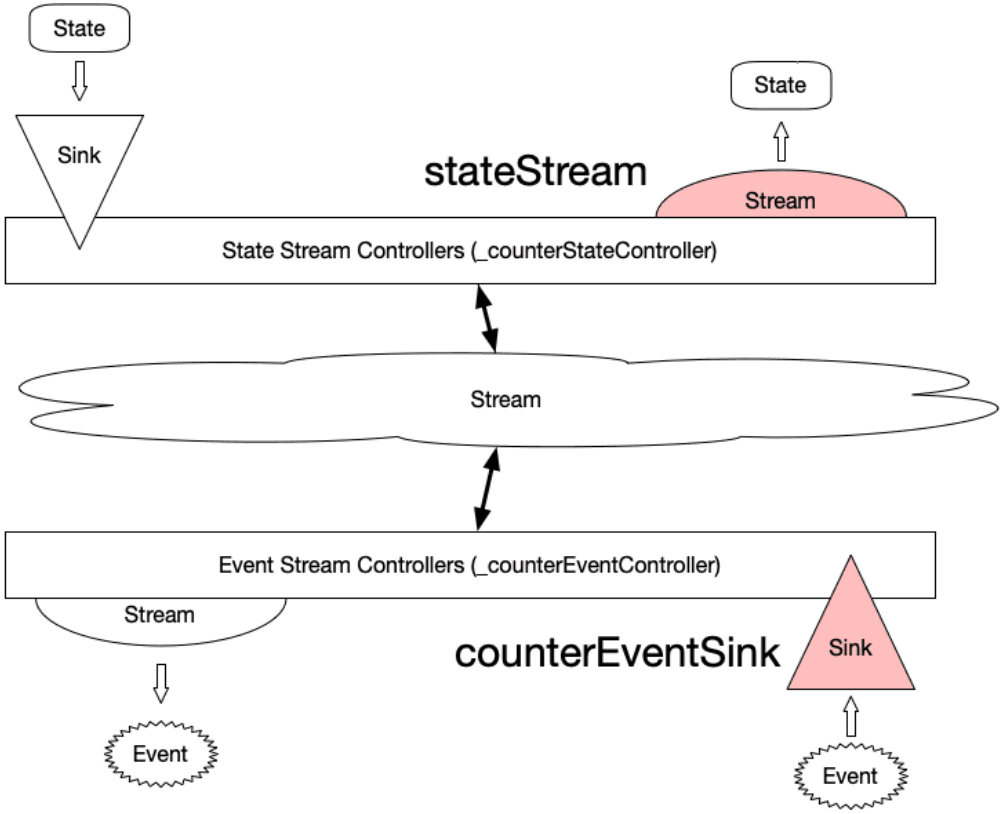
State Stream Property

- We need to provide the `stateStream` property, which is the stream of the state controller for the state.

Code Example:

```
get stateStream =>
  _counterStateController.stream;
```

Public Properties Overview



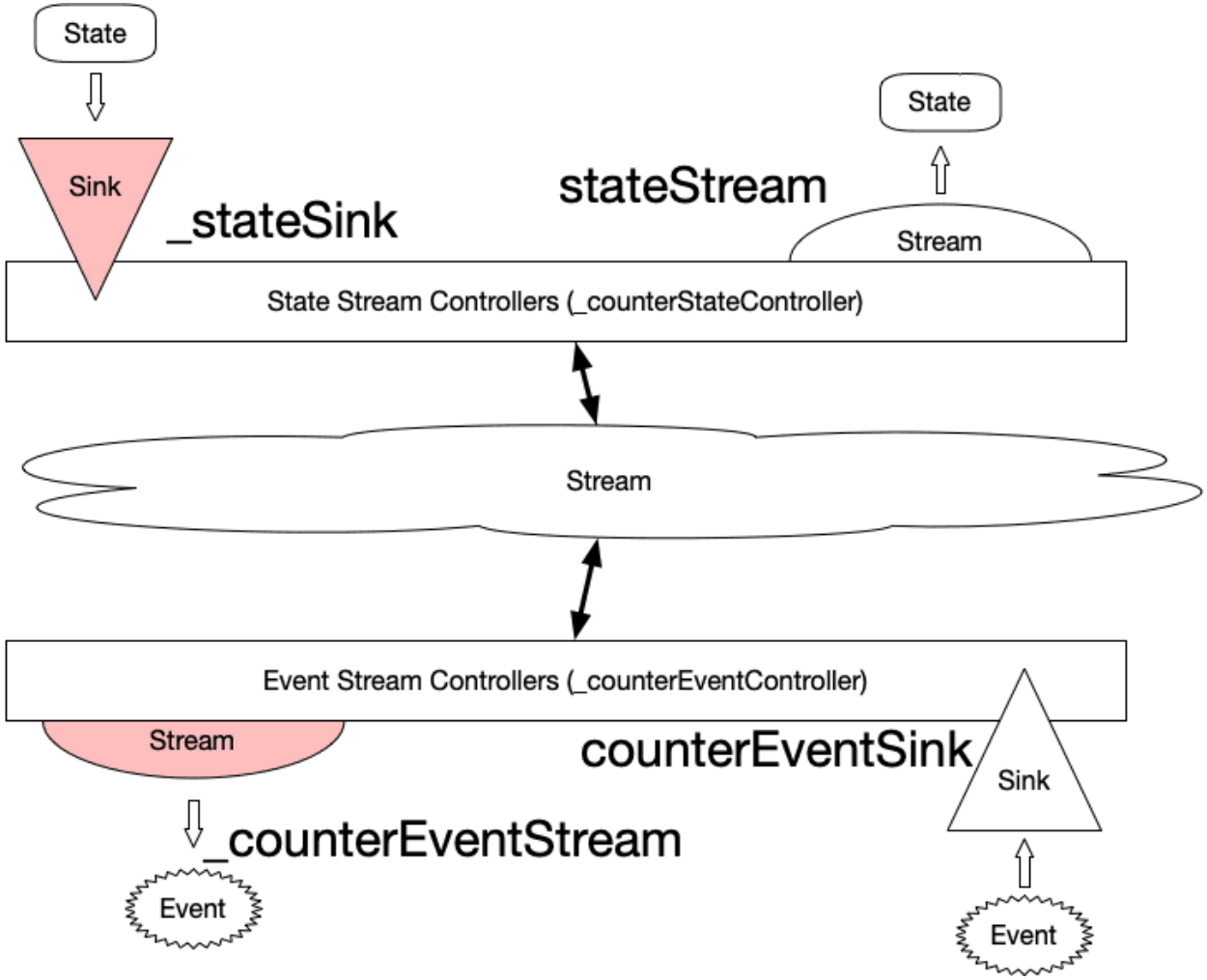
State Management

- We use `_stateSink` property, which is the sink (input) to the stream for the state (`_counter`) from the _conterStateController.
- We get stateStream from the same controller.

Code Example:

```
get _stateSink => _counterStateController.sink;
get stateStream =>
  _counterStateController.stream;
```

Internal Architecture



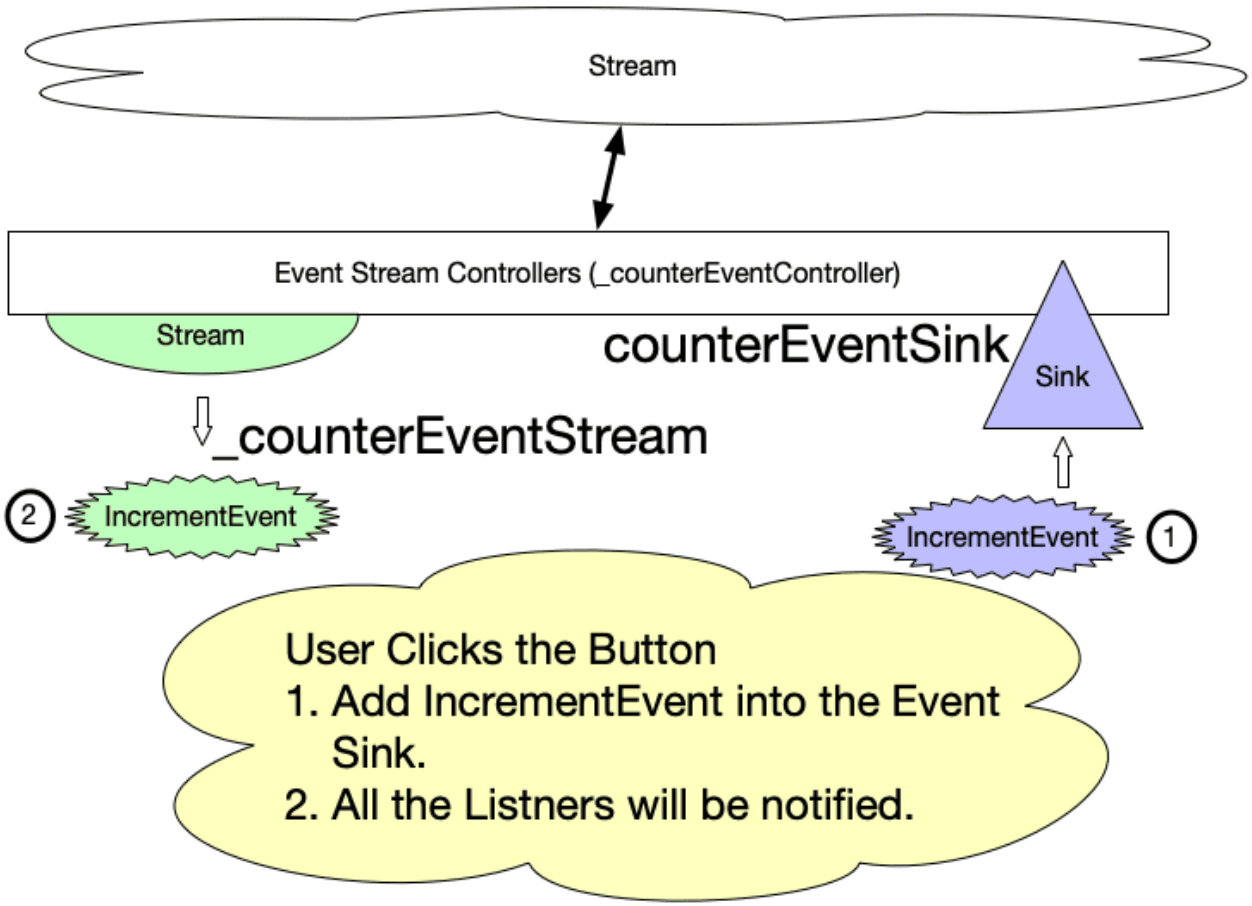
The BLoC process

1. User Makes the Event

- We use the event sink.
- User clicks the button to add an event to the event sink.

Code Example:

```
final _bloc = CounterBloc();
...
FloatingActionButton(
  onPressed:
    () =>
      _bloc.counterEventSink.add(IncrementEvent()),
)
```

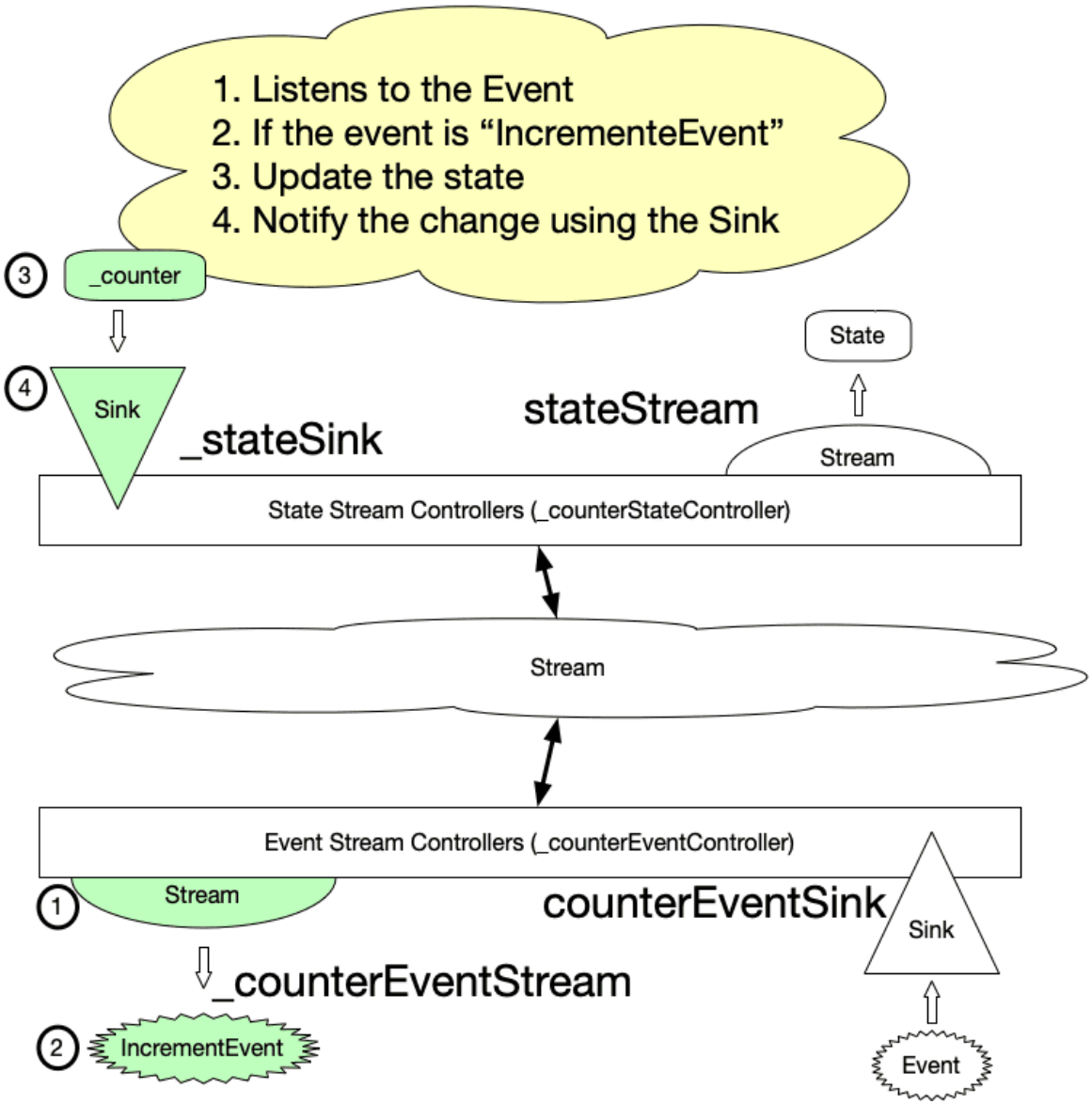

Stream
Event Stream Controllers (_counterEventController)
Stream
counterEventSink
Sink
_counterEventStream
2
IncrementEvent
IncrementEvent
1
User Clicks the Button
1. Add IncrementEvent into the Event Sink.
2. All the Listners will be notified.

2. Events to State

- We need to listen to an event from the event controller's stream.
- When an event is observed, we update the state and notify the change using the state sink.

Code Example:

```
CounterBloc() {
  _counterEventStream.listen(
    _mapEventToState);
}
void _mapEventToState(CounterEvent event) {
  if (event is IncrementEvent) {
    _counter++;
  }
  _stateSink.add(_counter);
}
```


1. Listens to the Event
2. If the event is “IncrementeEvent”
3. Update the state
4. Notify the change using the Sink
3
_counter
4
Sink
_stateSink
stateStream
State
Stream
State Stream Controllers (_counterStateController)
Stream
Event Stream Controllers (_counterEventController)
1
Stream
_counterEventStream
counterEventSink
Sink
2
IncrementEvent
Event

3. The StreamBuilder

- The StreamBuilder widget observes the stream (`_bloc.stateStream`) and captures the data in the `snapshot.data`.

Code Example:

```
StreamBuilder(
  stream: _bloc.stateStream,
  initialData: 0,
  builder: (BuildContext context, AsyncSnapshot<int> snapshot) {
    return Column(
      children: <Widget>[
        const Text(...),
        Text(
          '${snapshot.data}',
...
```

4. Cleanup and Disposal

- We need to close both controllers.

Code Example:

```
void dispose() {
  _counterStateController.close();
  _counterEventController.close();
}
```

Overall Architecture