



# UI Navigation

Mike Nakhimovich



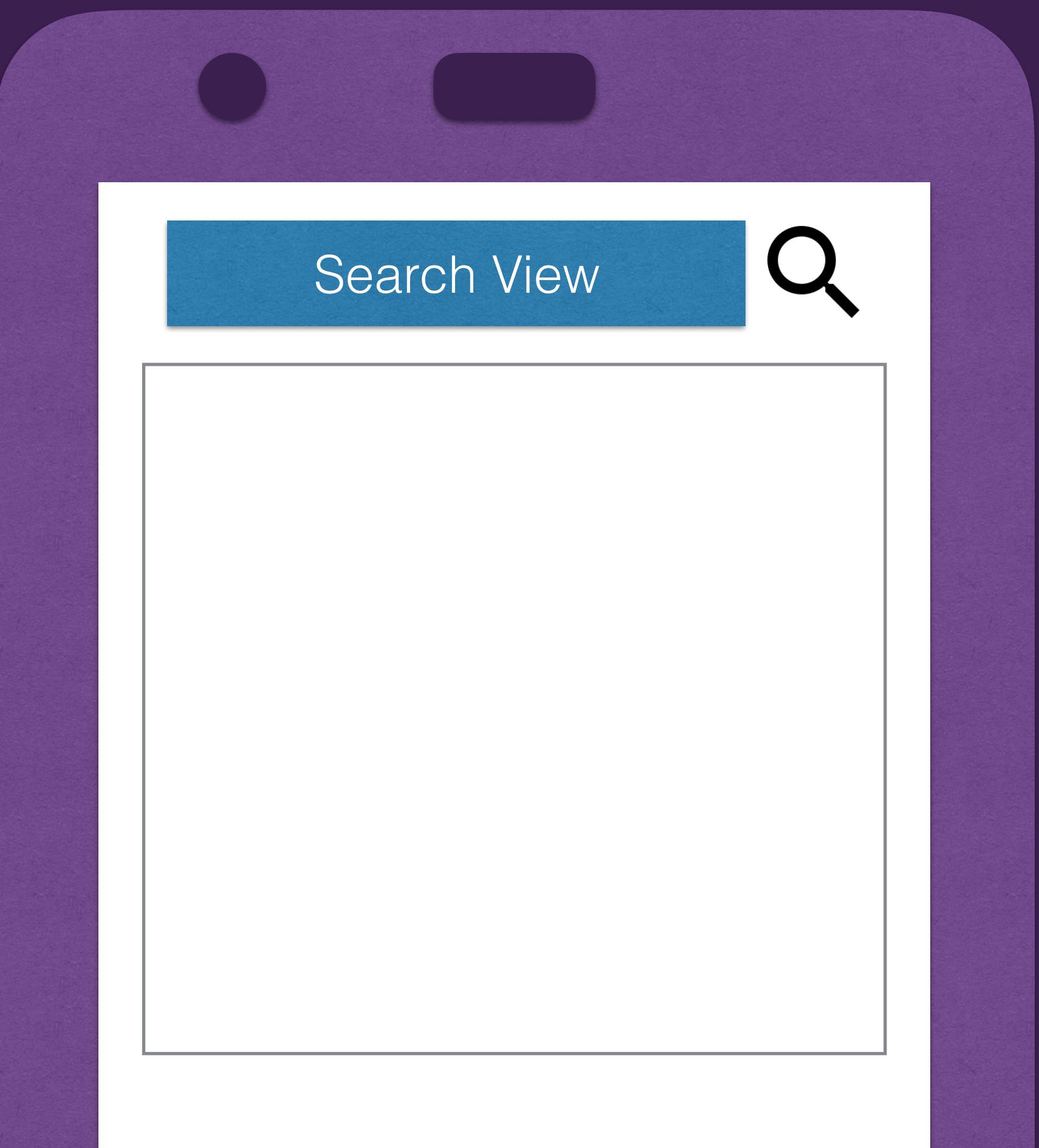
# Dispatching State

Mike Nakhimovich

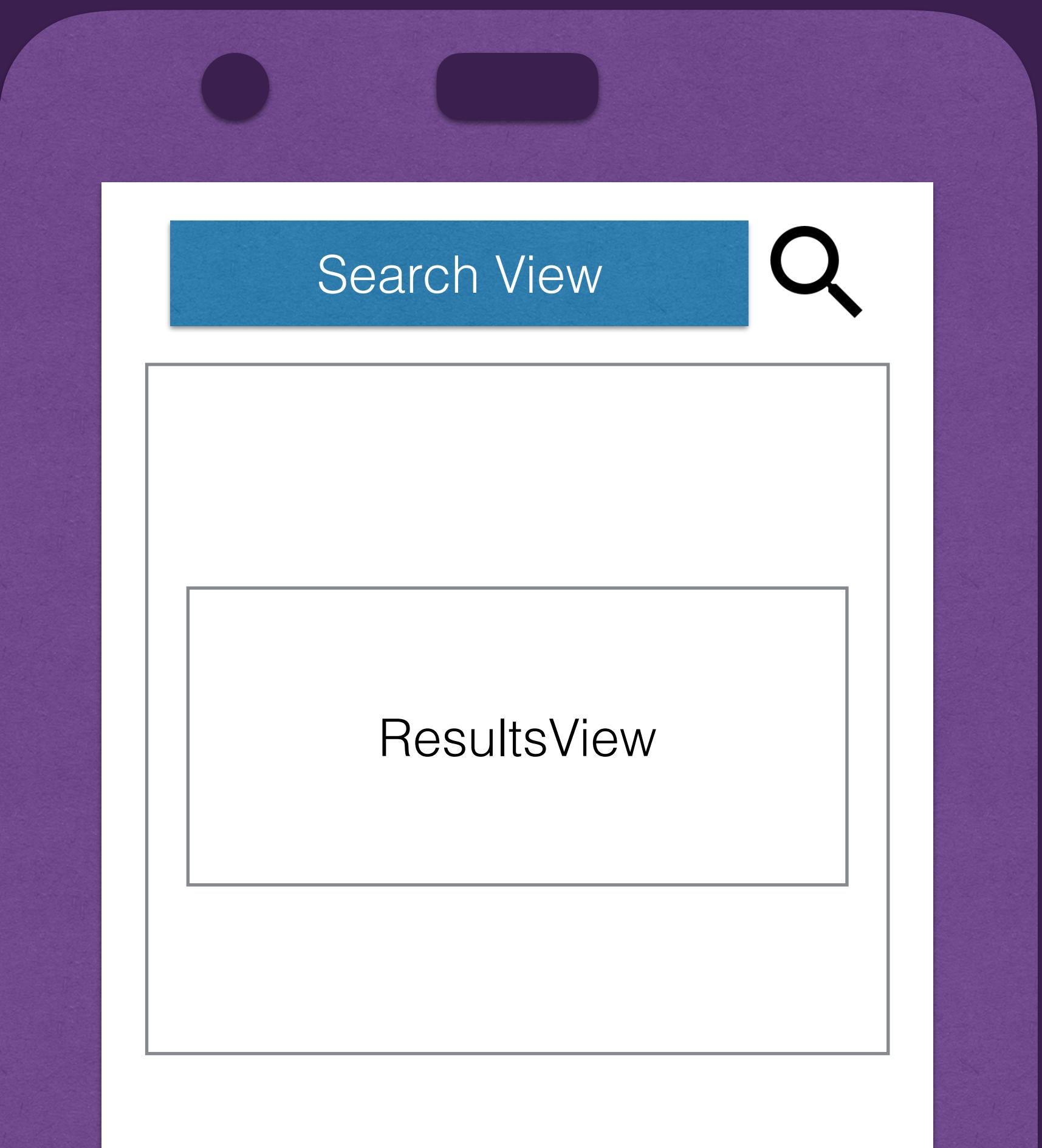
# Apps have a UI



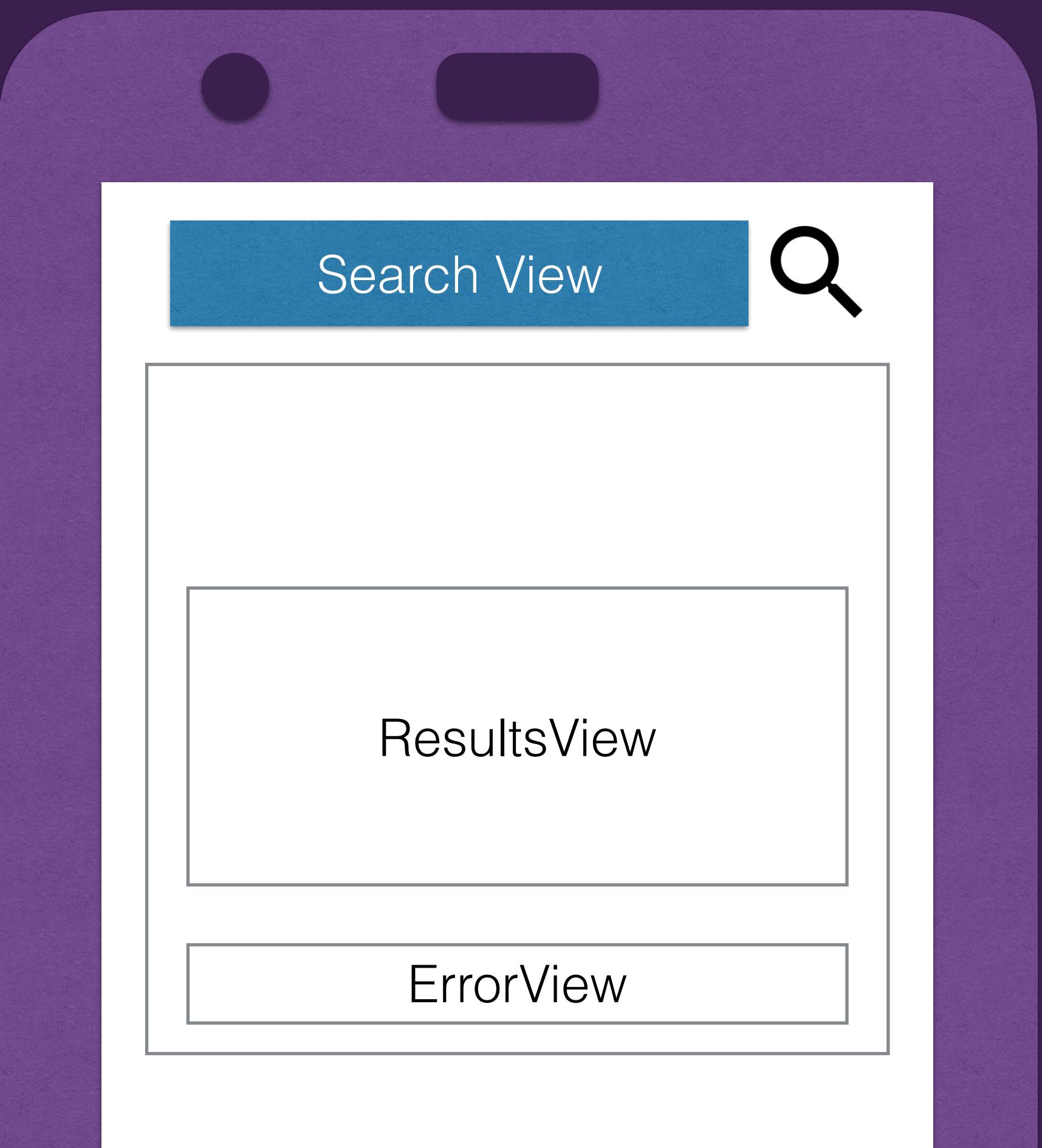
# Apps have a UI



# Apps have a UI



# Apps have a UI



# Apps have a UI



# Users interact with the UI



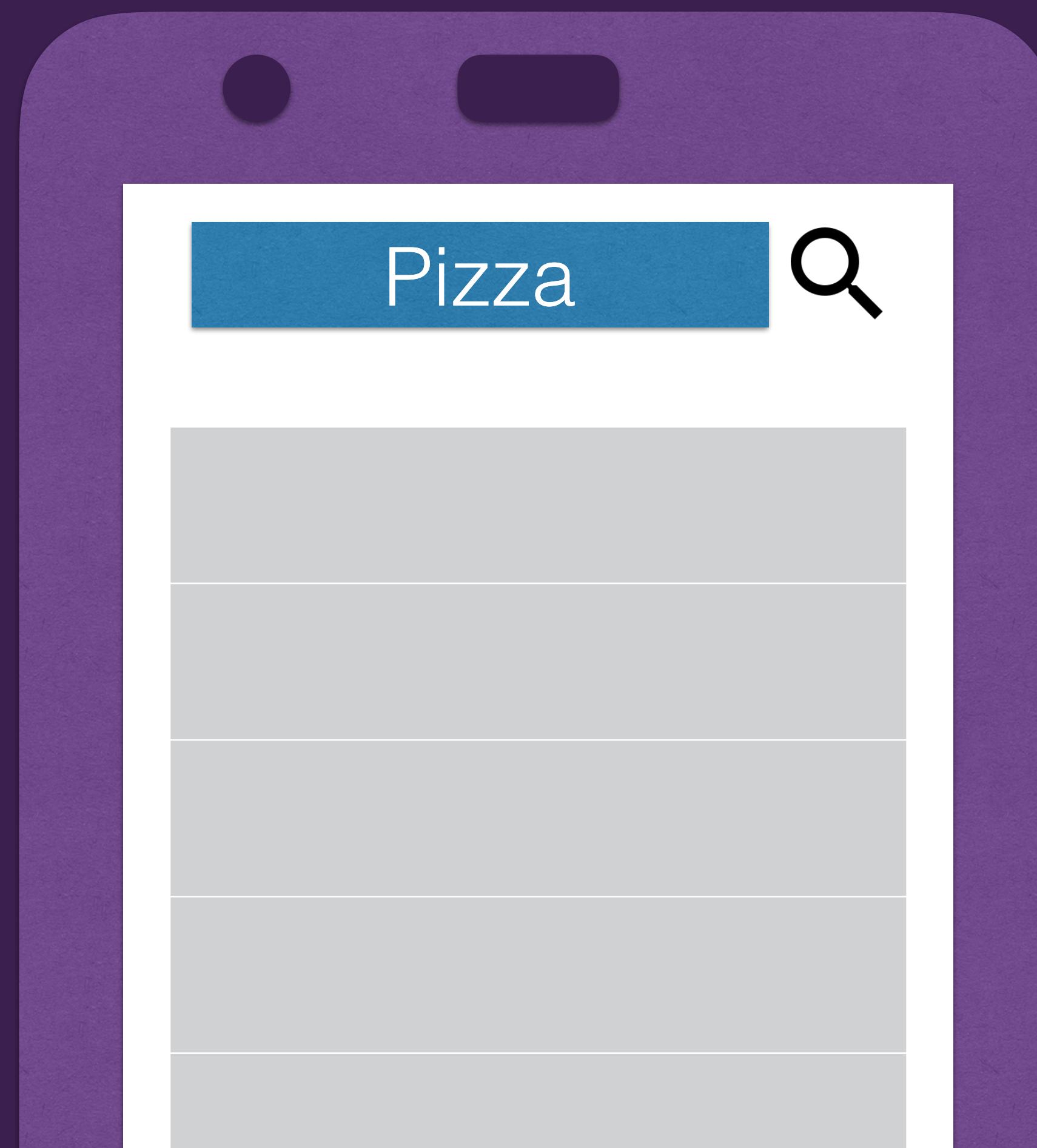
# Interactions lead to State change



# Interactions lead to State change



# State change need to be rendered



# What is state?

What is state?

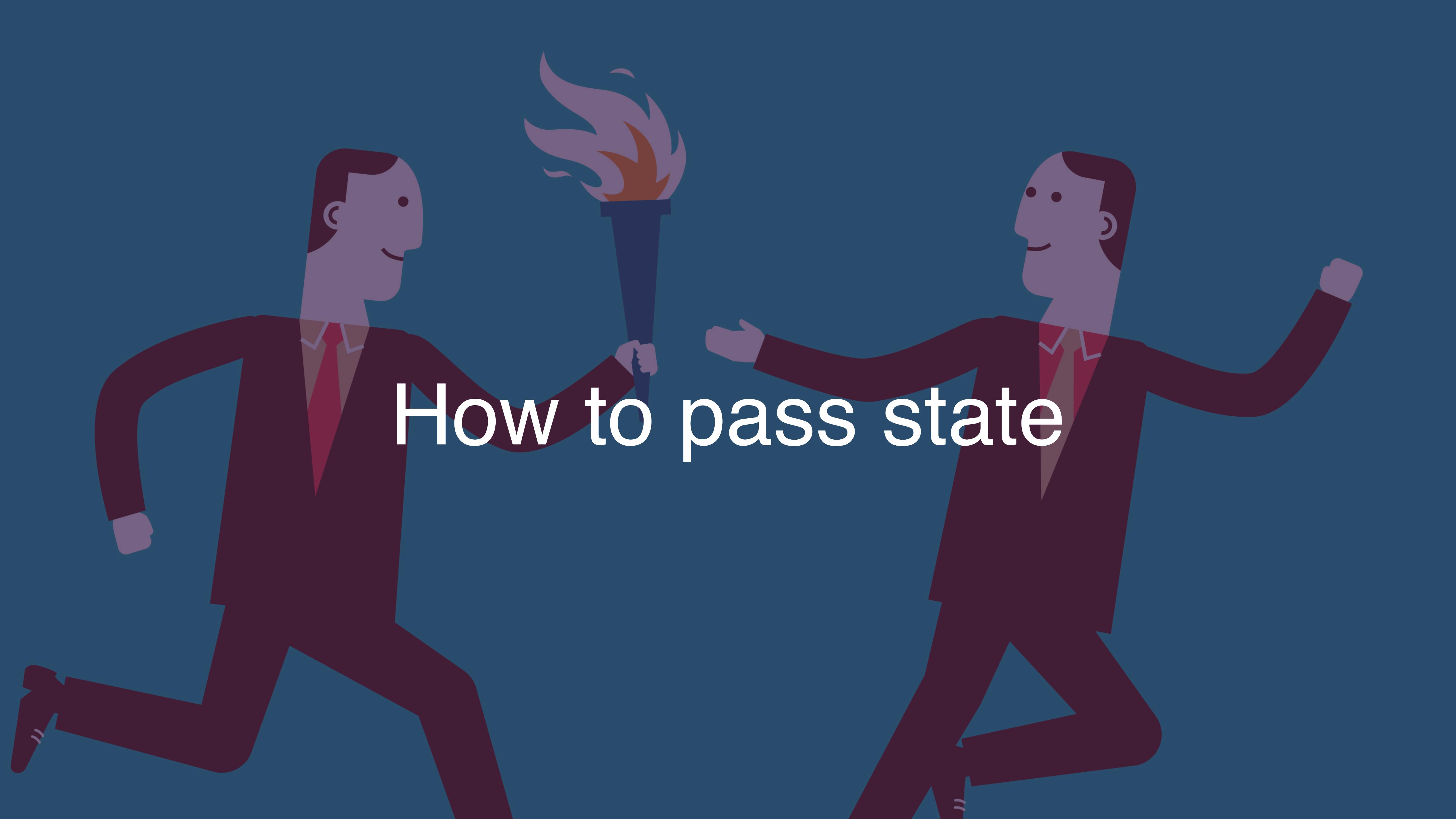
A representation of UI  
at a certain point in time

State determines how the  
UI renders & behaves

Android lacks patterns for  
working with state & views

A wide-angle photograph of a mountain range during sunset or sunrise. The sky is filled with soft, warm colors ranging from deep purple to bright orange and yellow. The mountains in the background are partially obscured by mist or low-hanging clouds, creating a sense of depth. In the foreground, there's a dense forest of evergreen trees, and some green shrubs and small white flowers are visible in the lower-left corner.

# How to create state

A stylized illustration of two men in dark blue suits and red ties running towards the right. The man on the left is holding a lit torch with a blue and orange flame. The man on the right has his arms raised in a celebratory or cheering gesture. They are set against a dark blue background.

# How to pass state

A photograph of a port terminal showing a large stack of shipping containers. The containers are stacked in several layers, with some containers on the ground level. The containers are of various colors, including white, blue, red, yellow, and green. Some containers have the Maersk logo and name on them. In the background, there are several large blue cranes used for moving the containers. The sky is overcast and grey.

How to hold state



How do we think about state?

# Example: state modeled as a single data class

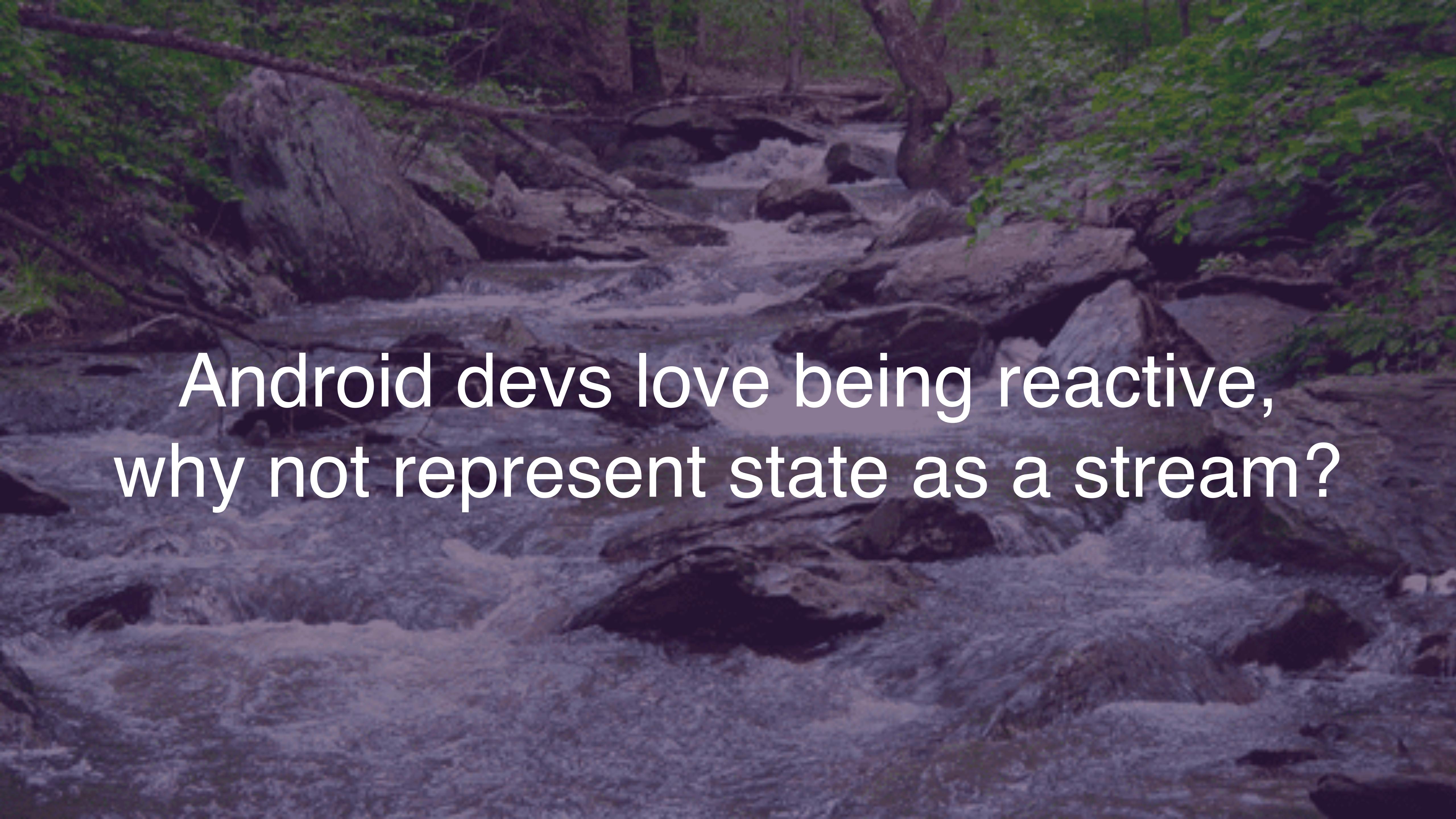
```
data class ScreenState(  
    val searchResults:List<Results>,  
    val searchTerm:String,  
    val isLoading:Boolean)
```

Example: state modeled as a single growing data class

```
data class ScreenState(  
    val searchResults:List<Results>,  
    val searchTerm:String,  
    val isLoading:Boolean,  
    val CartItems:List<Items>,  
    val userId:Long
```

~~State modeled as a God class~~

```
data class ScreenState(  
    val searchResults:List<Results>,  
    val searchTerm:String,  
    val isLoading:Boolean,  
    val CartItems:List<Items>,  
    val userId:Long
```

A photograph of a waterfall cascading down a dark, rocky cliff face. The water is white and turbulent as it falls. The cliff is covered in patches of green moss and small trees. The background is blurred, suggesting motion.

Android devs love being reactive,  
why not represent state as a stream?

Lately, I've been using a  
centralized dispatcher to model  
state as a reactive stream

# What's a dispatcher?

A dispatcher is an object that receives  
and sends messages about state

A dispatcher is an object that receives  
and sends messages about state

that the UI reacts to

# Why a state dispatcher?

# Why a state dispatcher?

Decouple producers of state  
change from its consumers

A silhouette of a person walking away from a two-headed arrow sign. The background is a sunset or sunrise over mountains.

# Journey to dispatcher

# User types a search term



# User clicks search



# Loading should become visible



# Followed by results



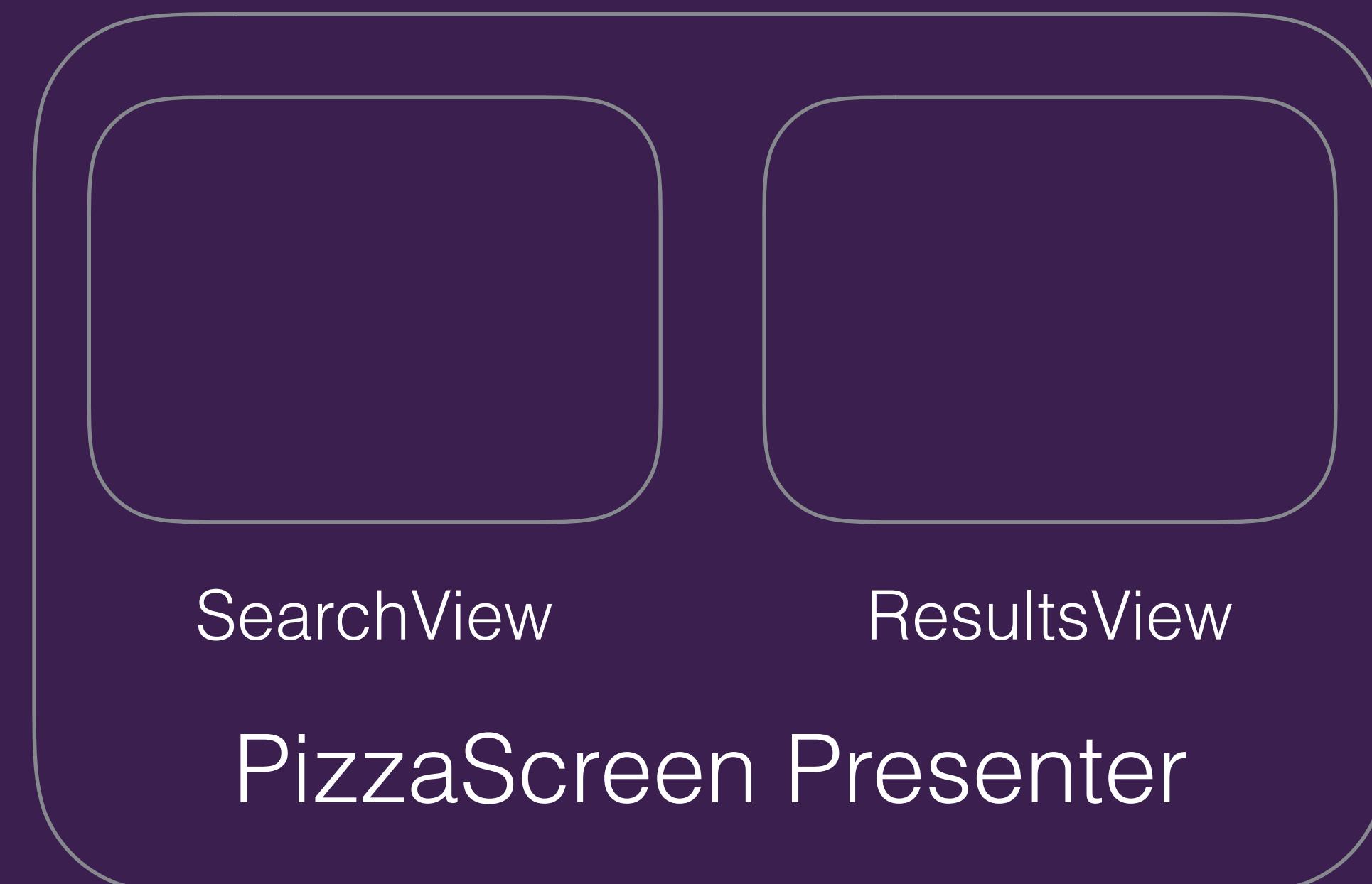
# How do we tell views what to display?

Proposal 1: Encapsulate Screen  
in a ScreenPresenter

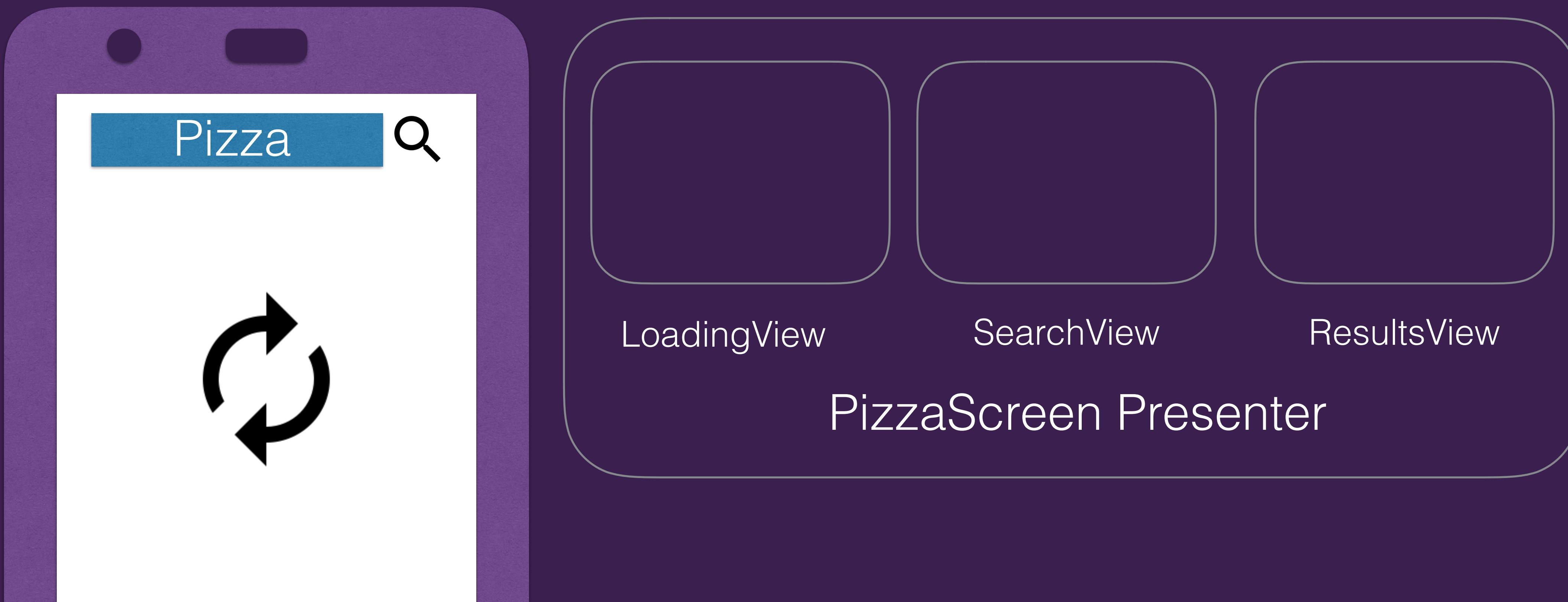


# How do we tell ResultsViews what to display?

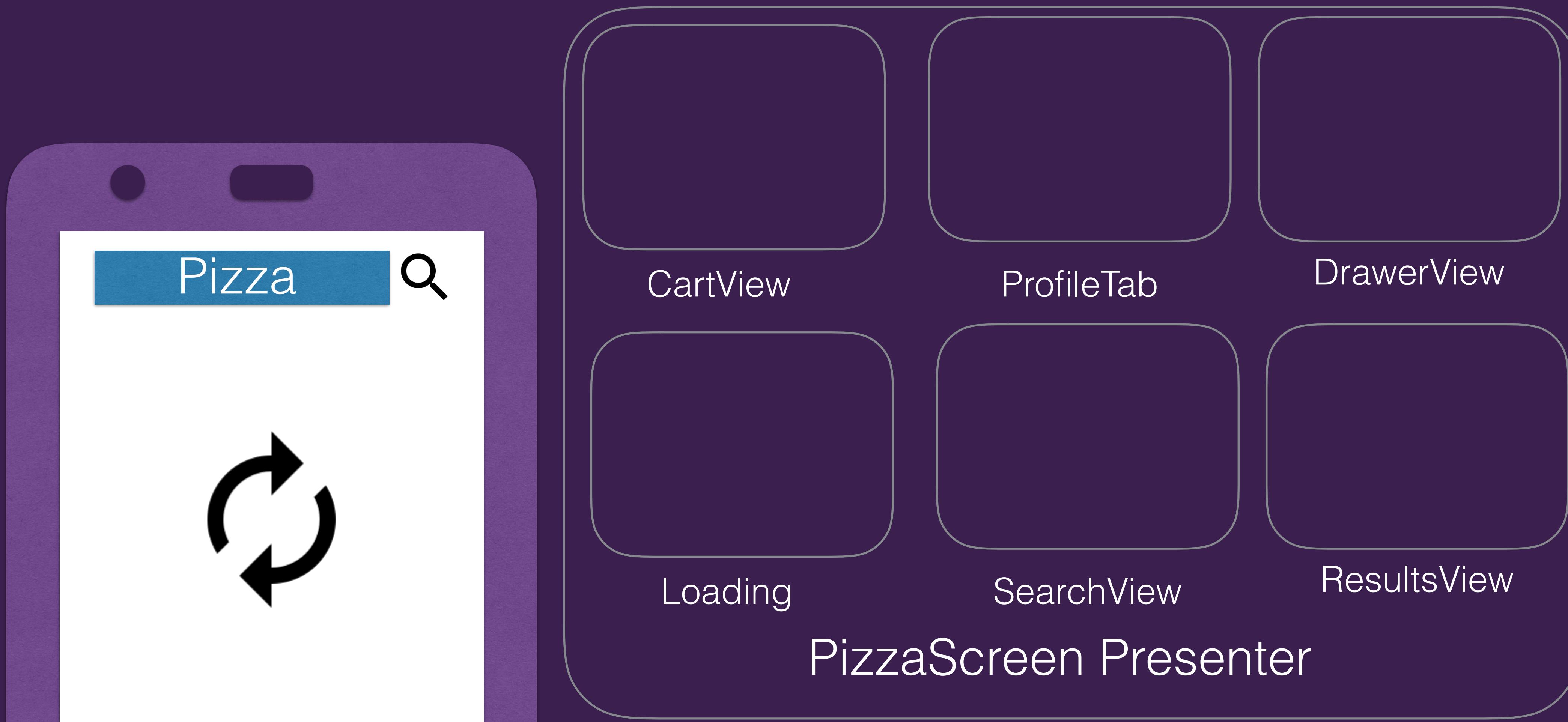
## Proposal 1: Encapsulate Screen in a ScreenPresenter



# But then... Product adds another view



# And then... Product adds 3 more views



# How do we tell ResultsView what to display?

~~Proposal 1: Create a ScreenPresenter  
which contains both Presenters~~



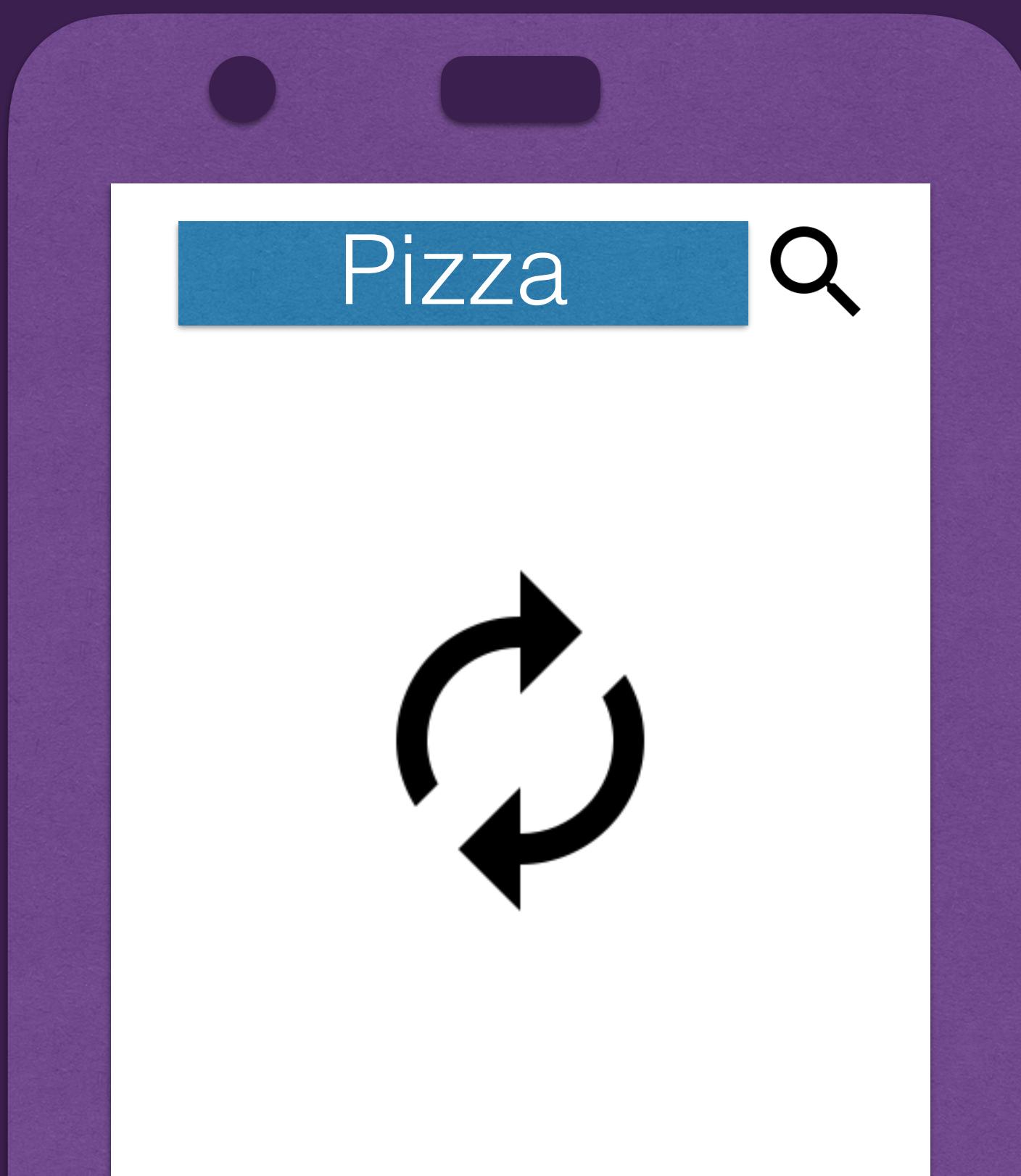
Not very scalable

# How do we tell ResultsView what to display?

## Proposal 2: Create a Listener



# Proposal 2: Create a Listener



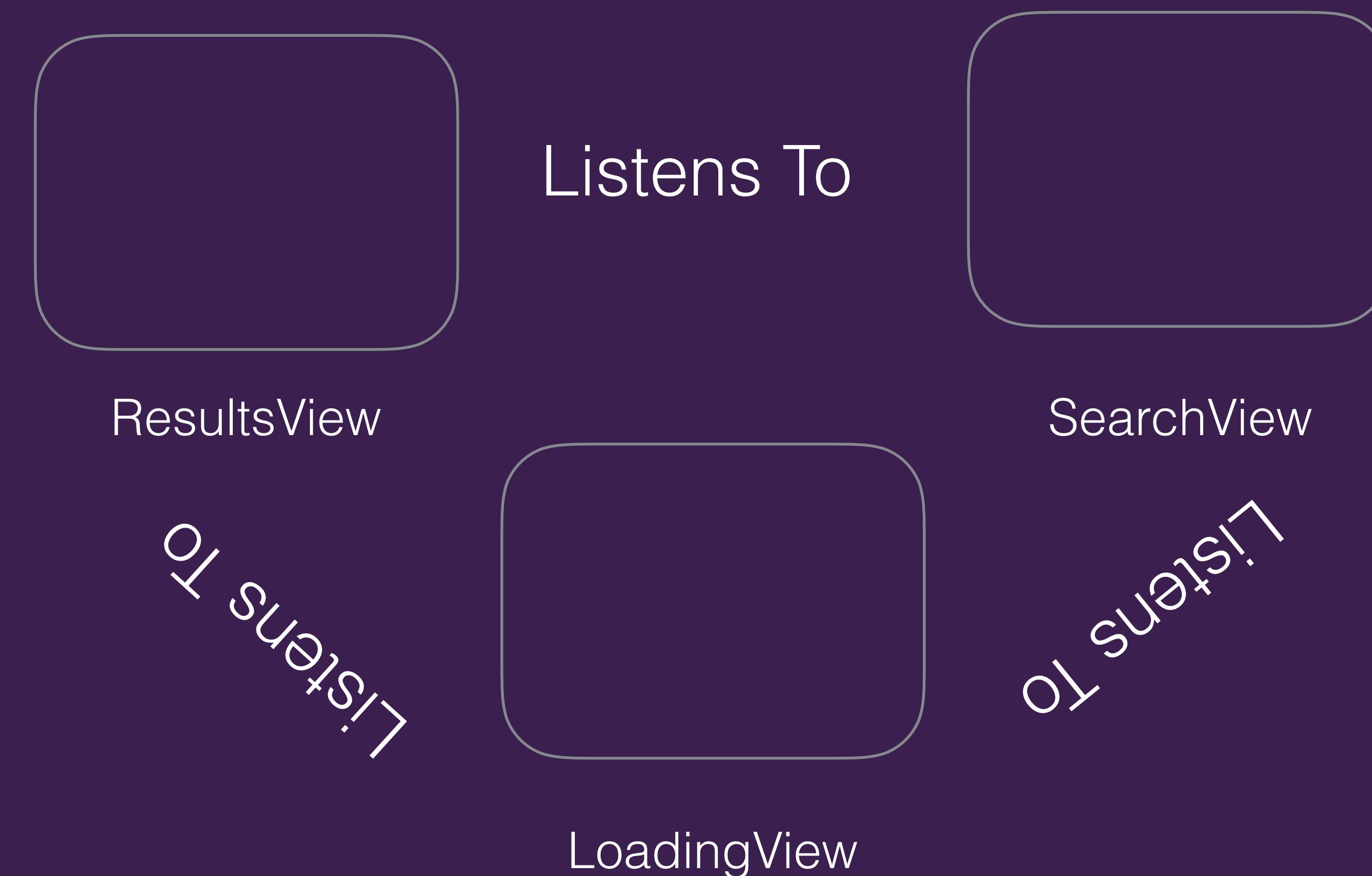
ResultsView

Listens To



SearchView

# Proposal 2: Create a Listener



## Proposal 2: Create a Listener

Listeners lead to circular dependencies

# Last Proposal

Use a centralized dispatcher  
for ALL state change

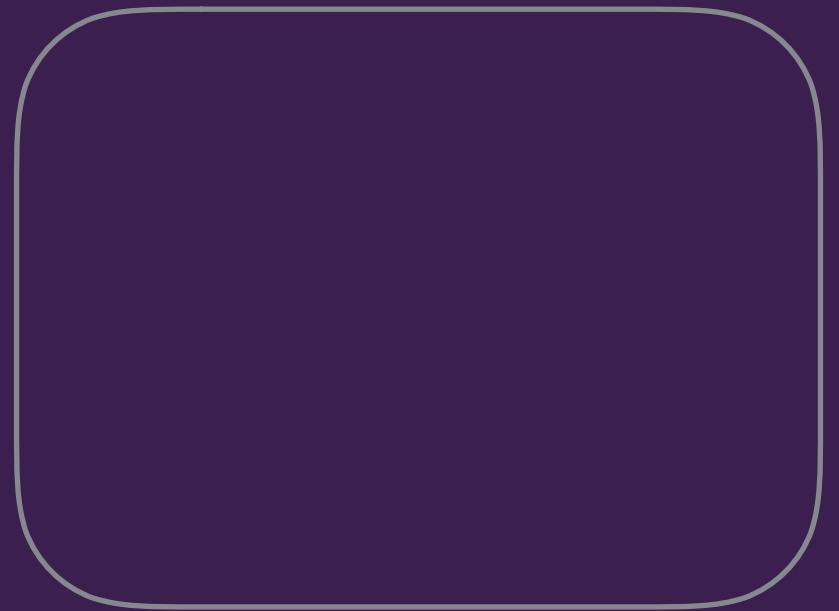
Use a centralized dispatcher  
for ALL state change

A Dispatcher is responsible for receiving  
state changes and emitting them

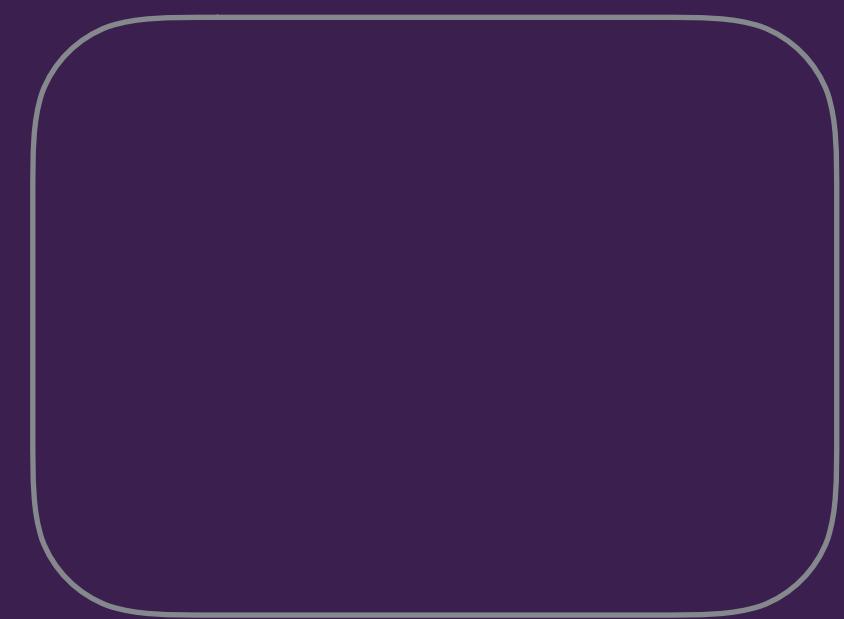
# Why a state dispatcher?

# Views become coupled to the dispatcher not each other

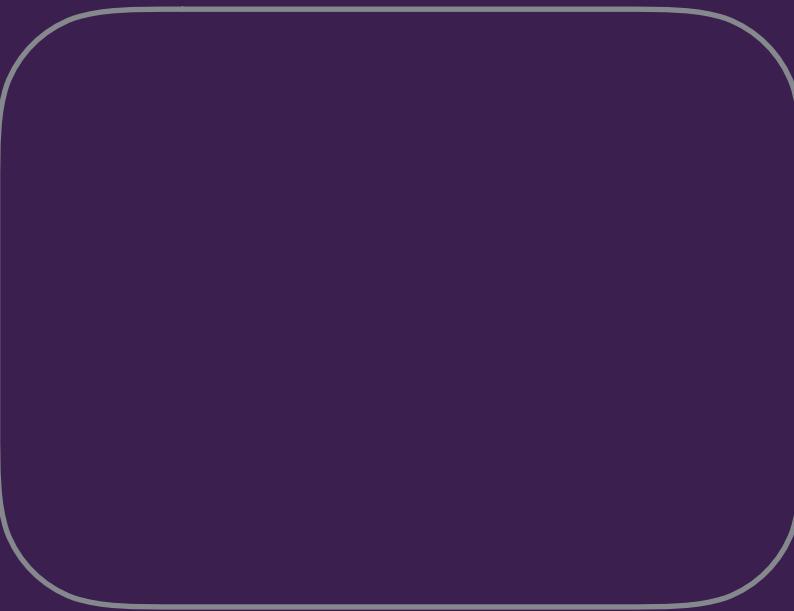
Dispatcher



ResultsView



LoadingView



SearchView

# Reactive state

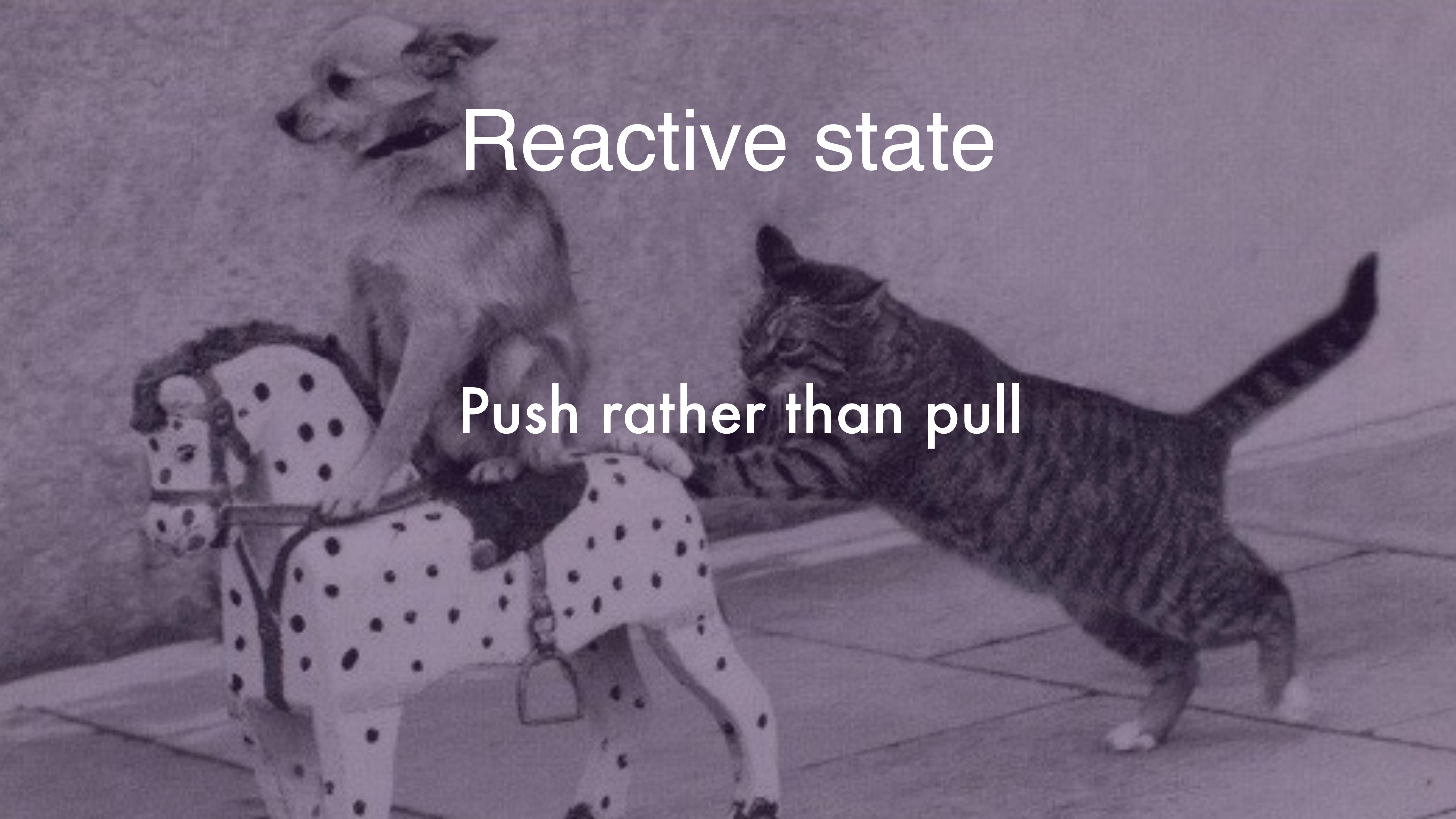
UI creates new state

# Reactive state

UI reacts to new state

# Reactive state

With no hard references between UI elements



# Reactive state

Push rather than pull

A black and white photograph showing a large, dense crowd of people from behind, seated in rows, likely in an auditorium or lecture hall. The perspective is from the back of the room, looking towards the front where a stage or presentation area would be.

# Implementing reactive state

# Reactive state

Represent your state as immutable value objects

```
Sealed class State{  
    data class ShowingLoading():State  
    data class ShowingResults(val results>List<String>):State  
    data class ShowingError(val error>Error):State  
}
```

# Reactive state

Push new state through a reactive stream

Interface Dispatcher

```
fun dispatch(state: State)  
{
```

interface RxState {

```
    fun <T> ofType(clazz: Class<T>): Observable<T>  
}
```

Sealed class State{

```
data class ShowingLoading():State  
data class ShowingResults(val results>List<String>):State  
data class ShowingError(val error>Error):State  
}
```

# Reactive state

Anytime UI needs to change, dispatch a new state

```
dispatcher.dispatch(State.ShowingResults(resultData))
```

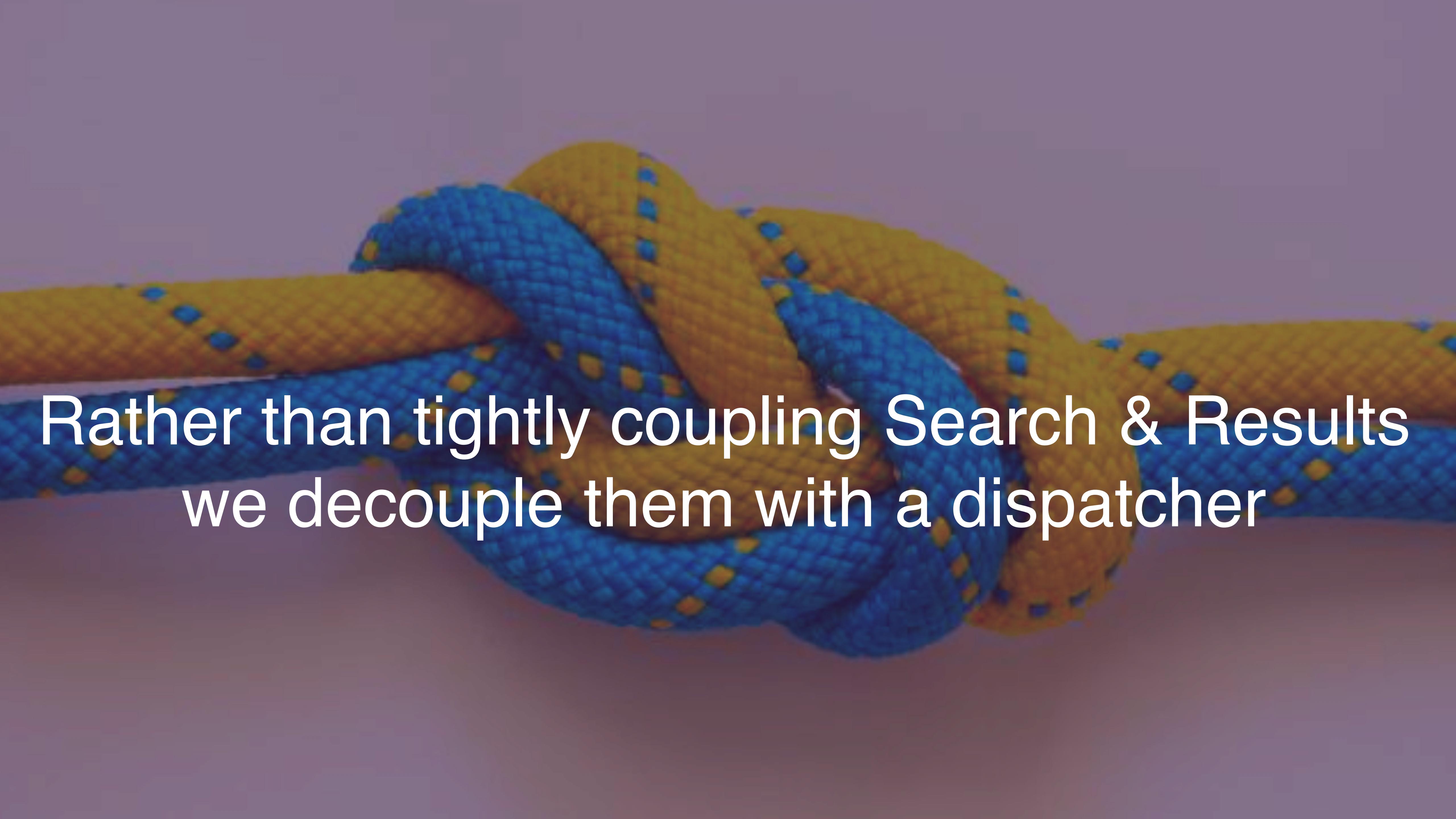
```
dispatcher.dispatch(State.ShowingLoading())
```

```
dispatcher.dispatch(State.ShowingError(errors))
```

# Reactive state

**Anytime state changes, react to new state**

```
rxState ofType(State.Results)  
    .map{it.data}  
    .subscribe{ mvpView.updateUI(data) }
```



Rather than tightly coupling Search & Results  
we decouple them with a dispatcher

# Reactive state visualized

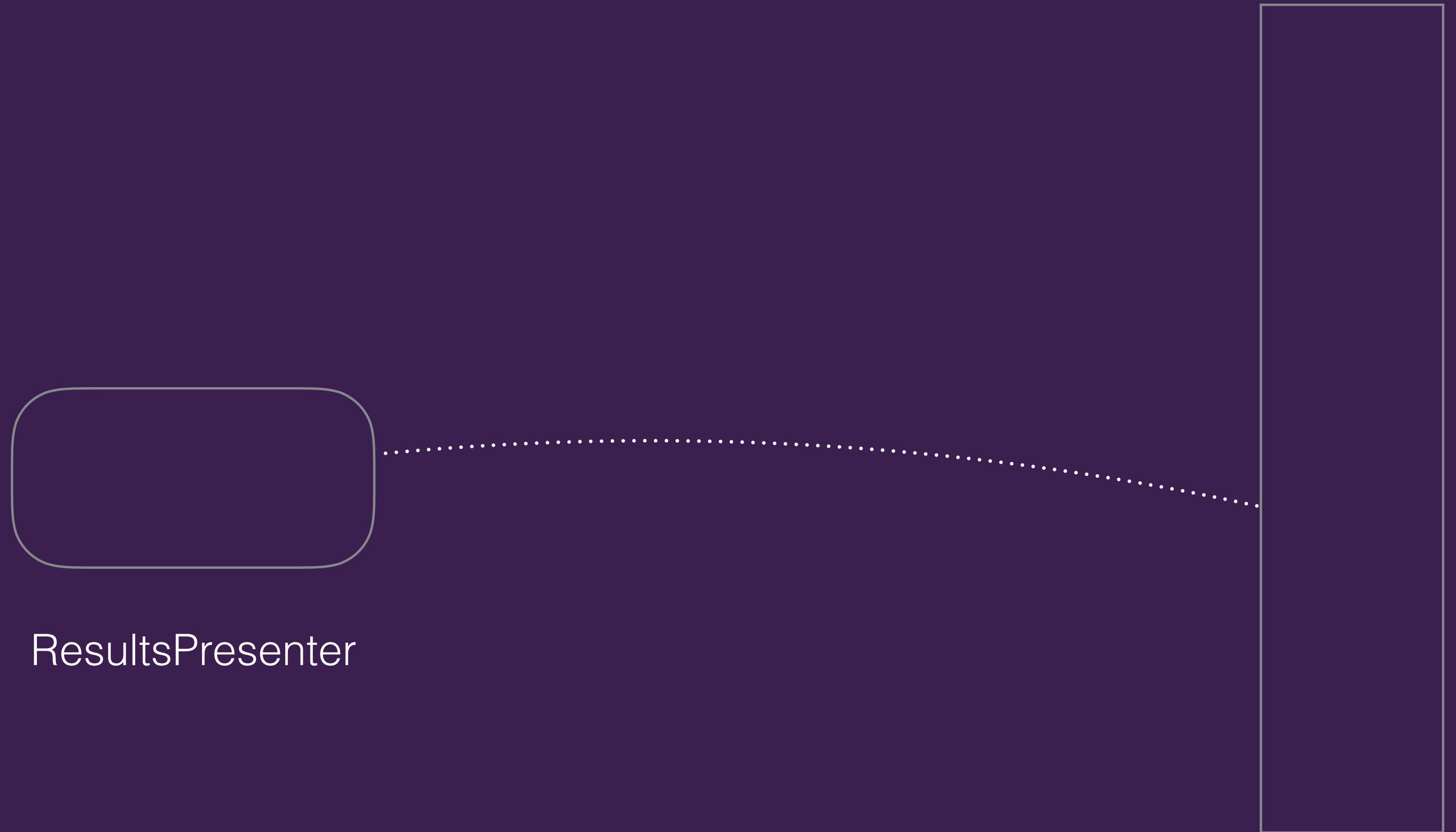
Dispatcher



# Reactive state

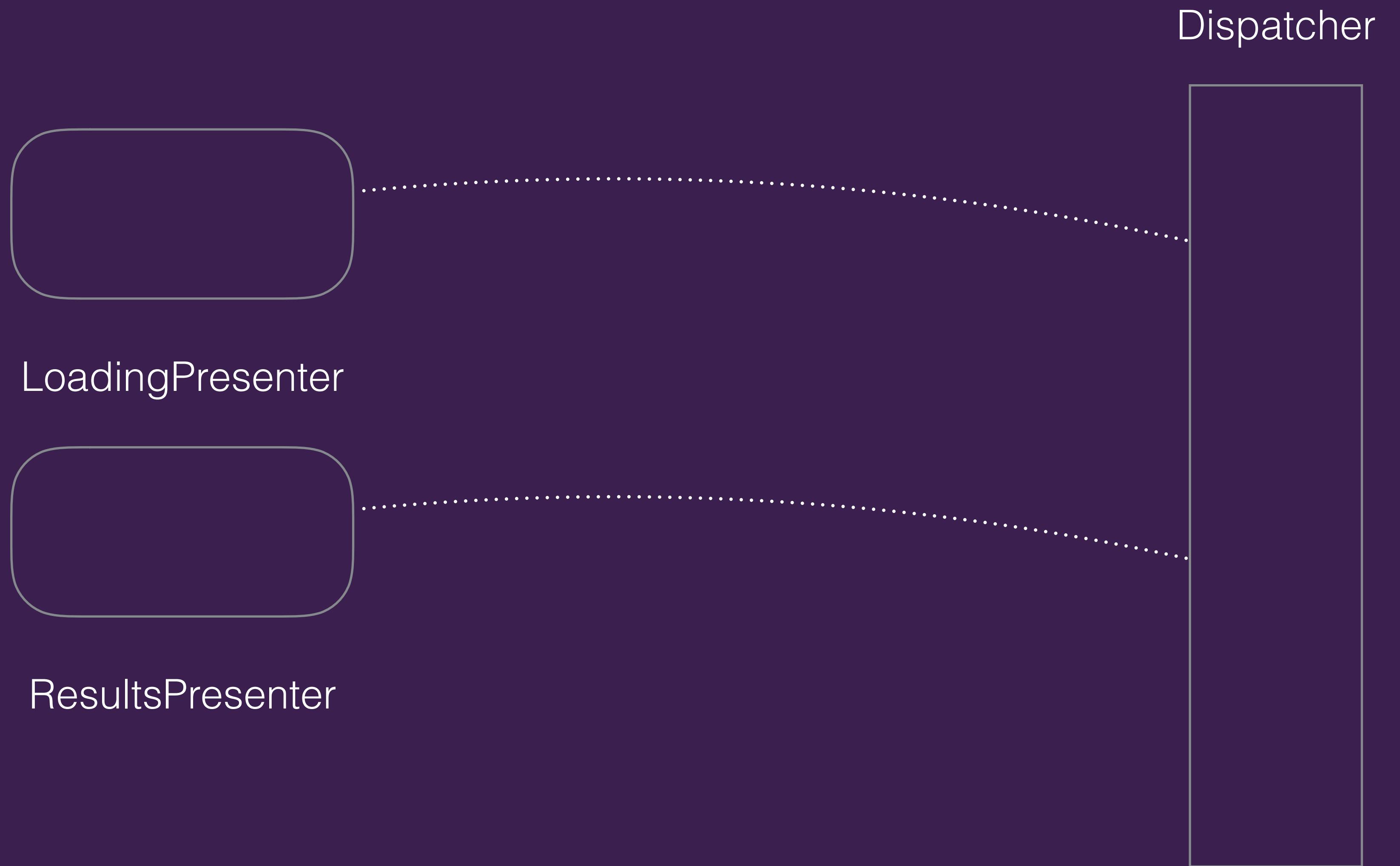
**ResultsPresenter Subscribes to ShowResults states change**

Dispatcher

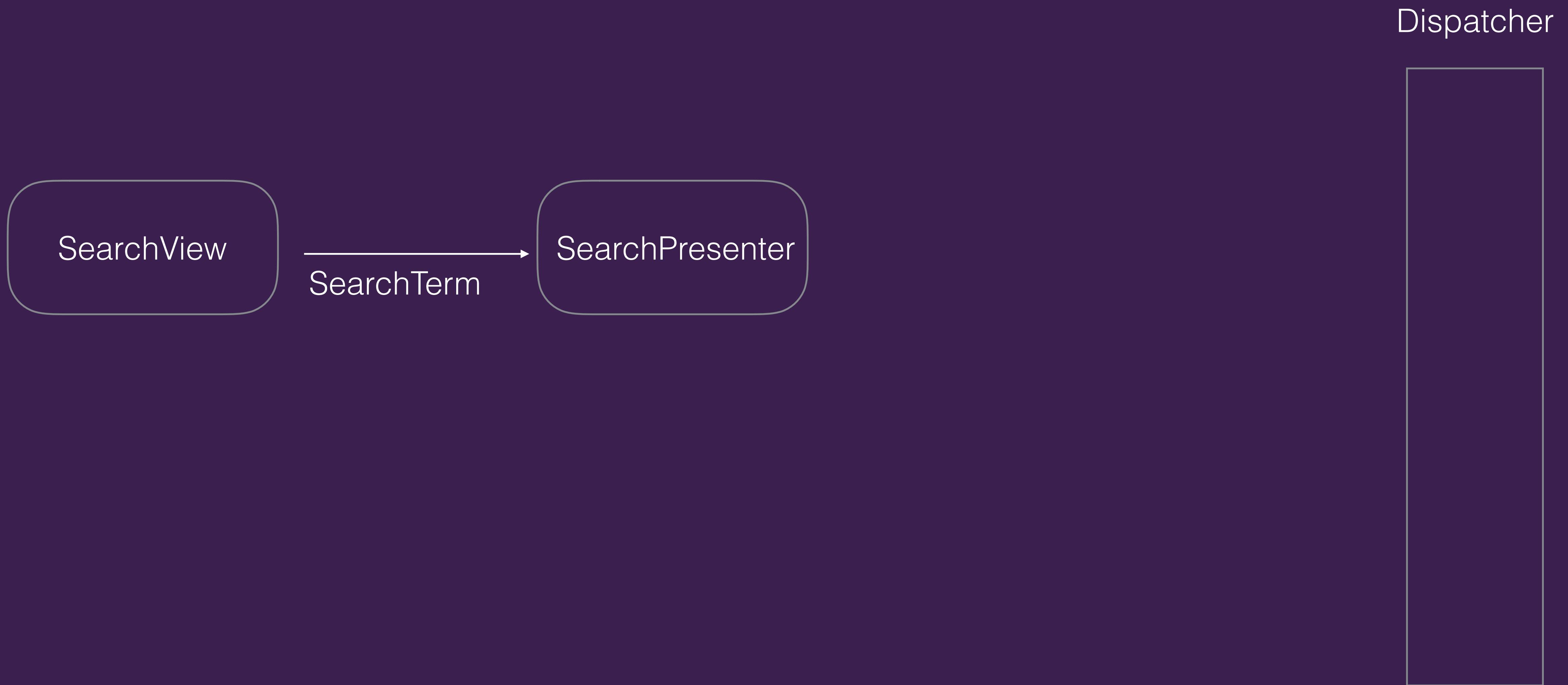


# Reactive state

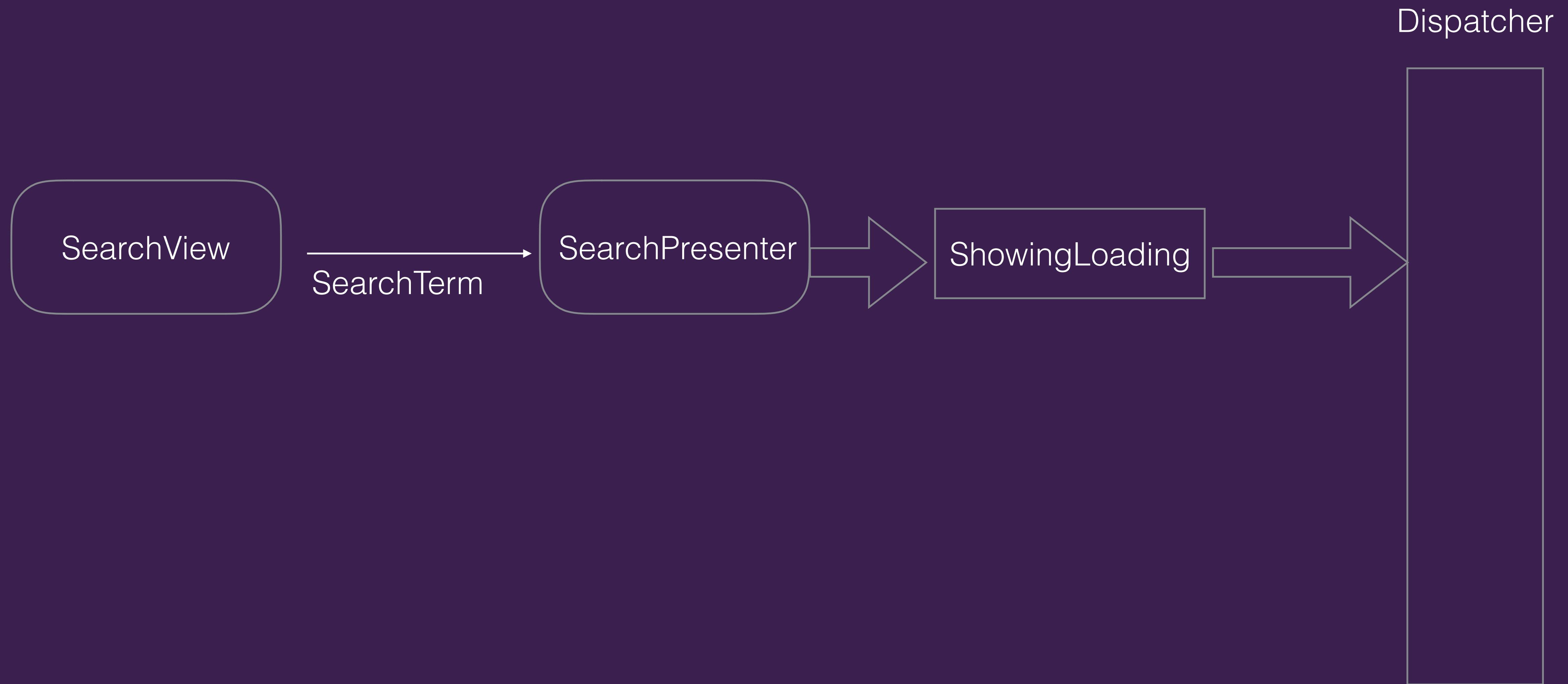
LoadingPresenter Subscribes to ShowLoading states change



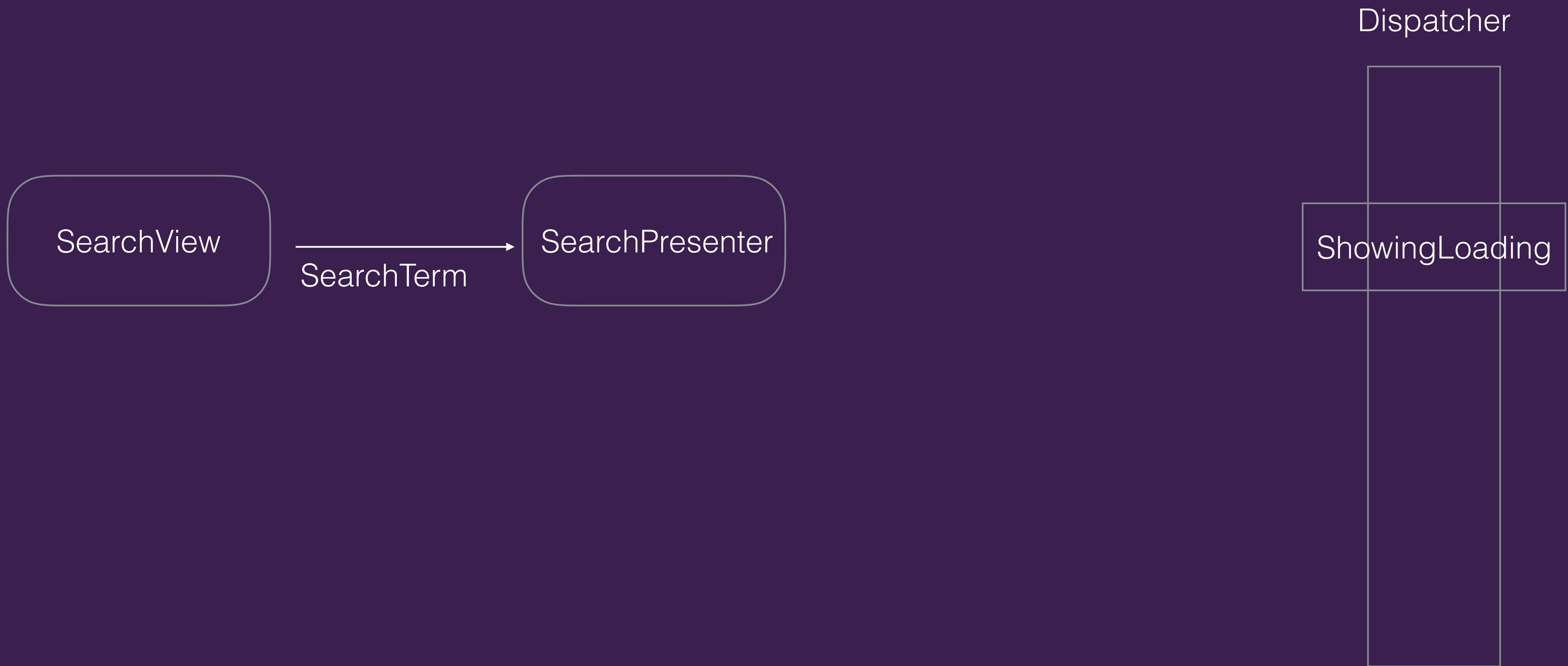
# Reactive state



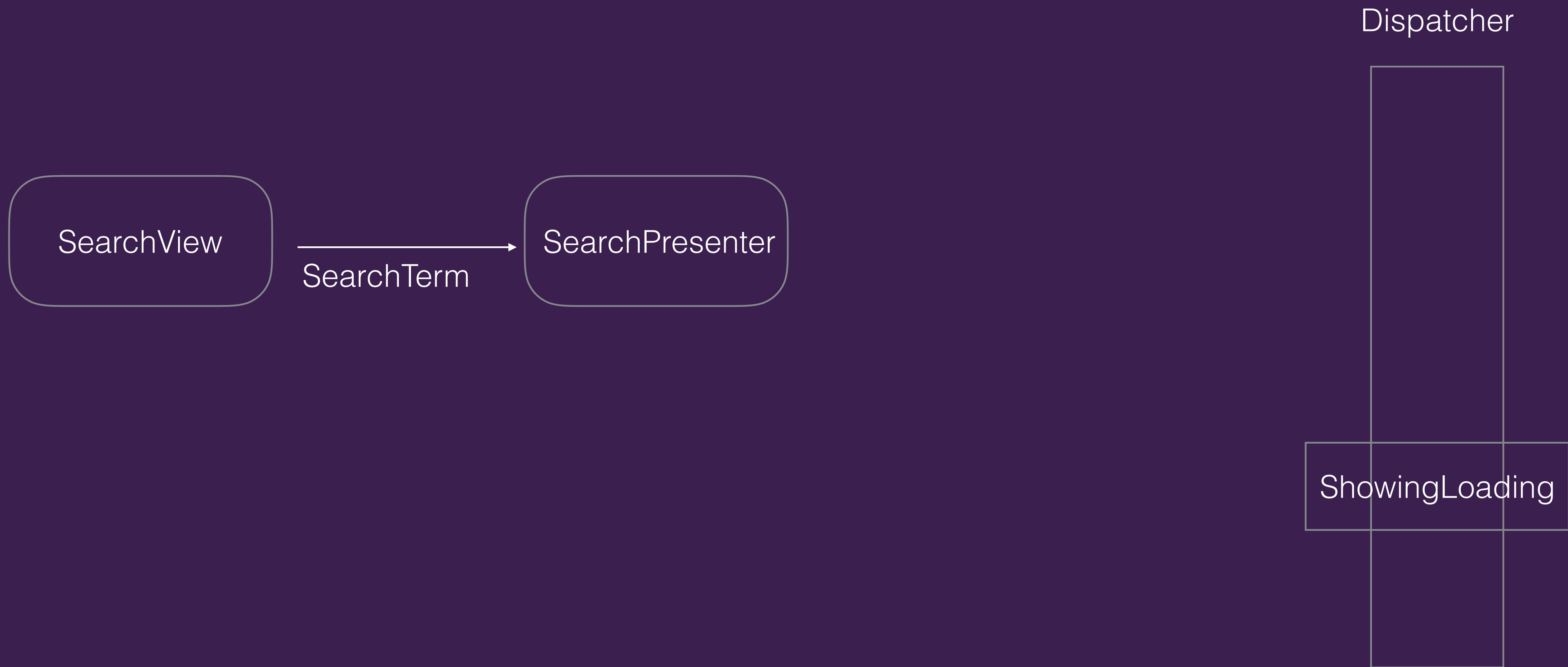
# Reactive state



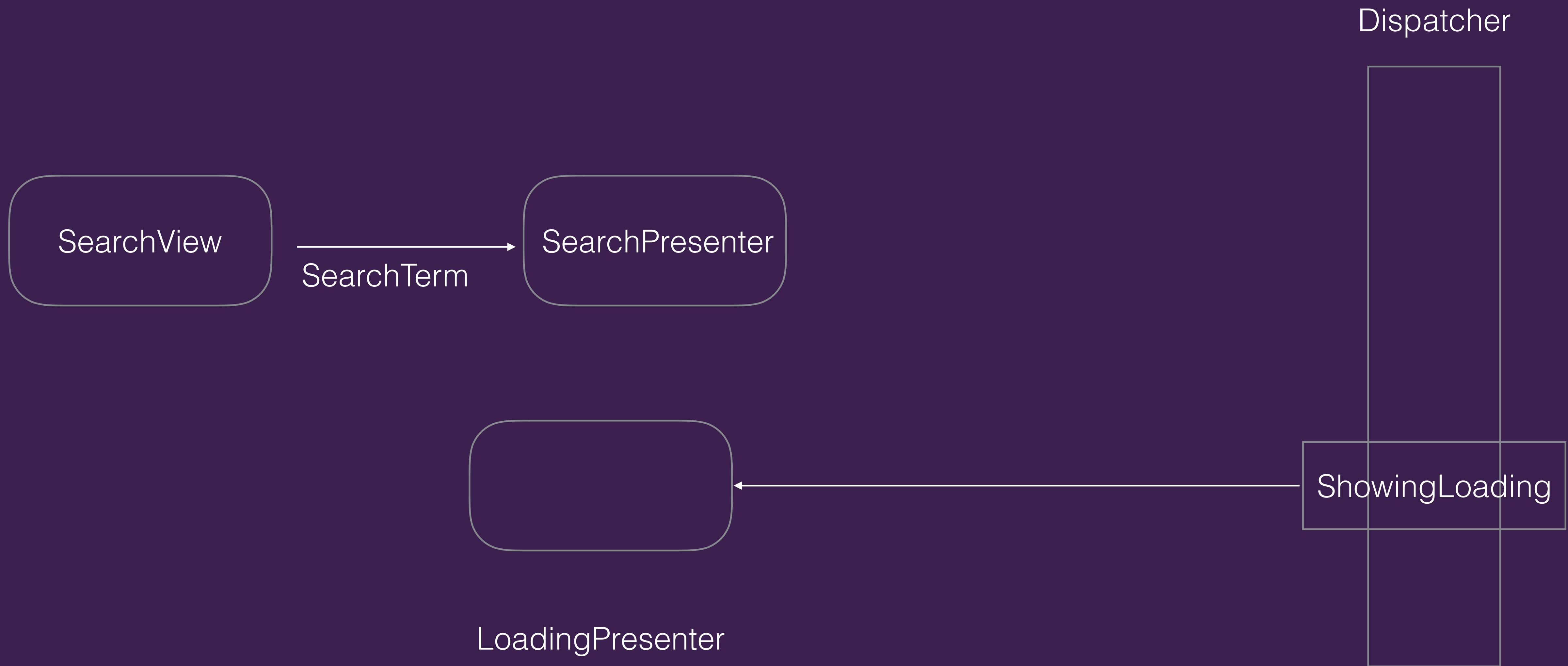
# Reactive state



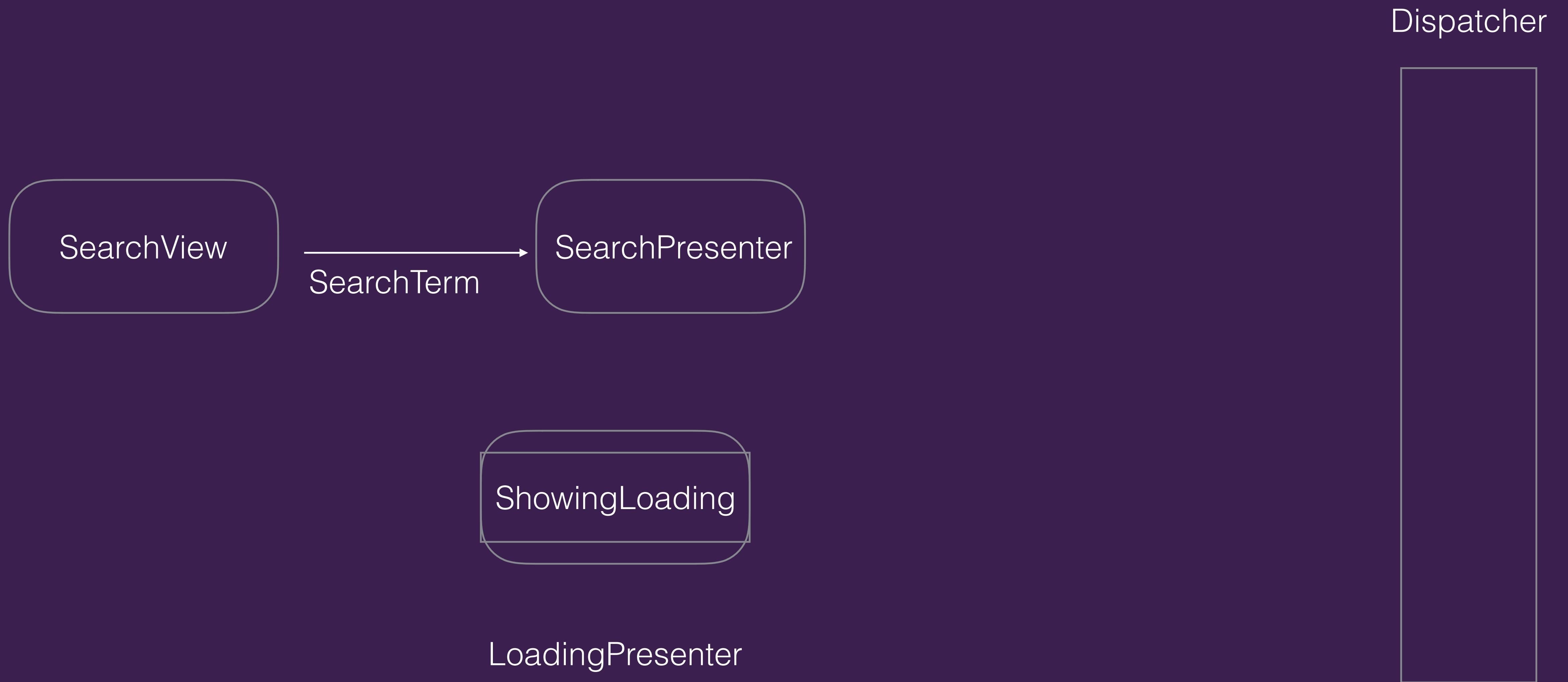
# Reactive state



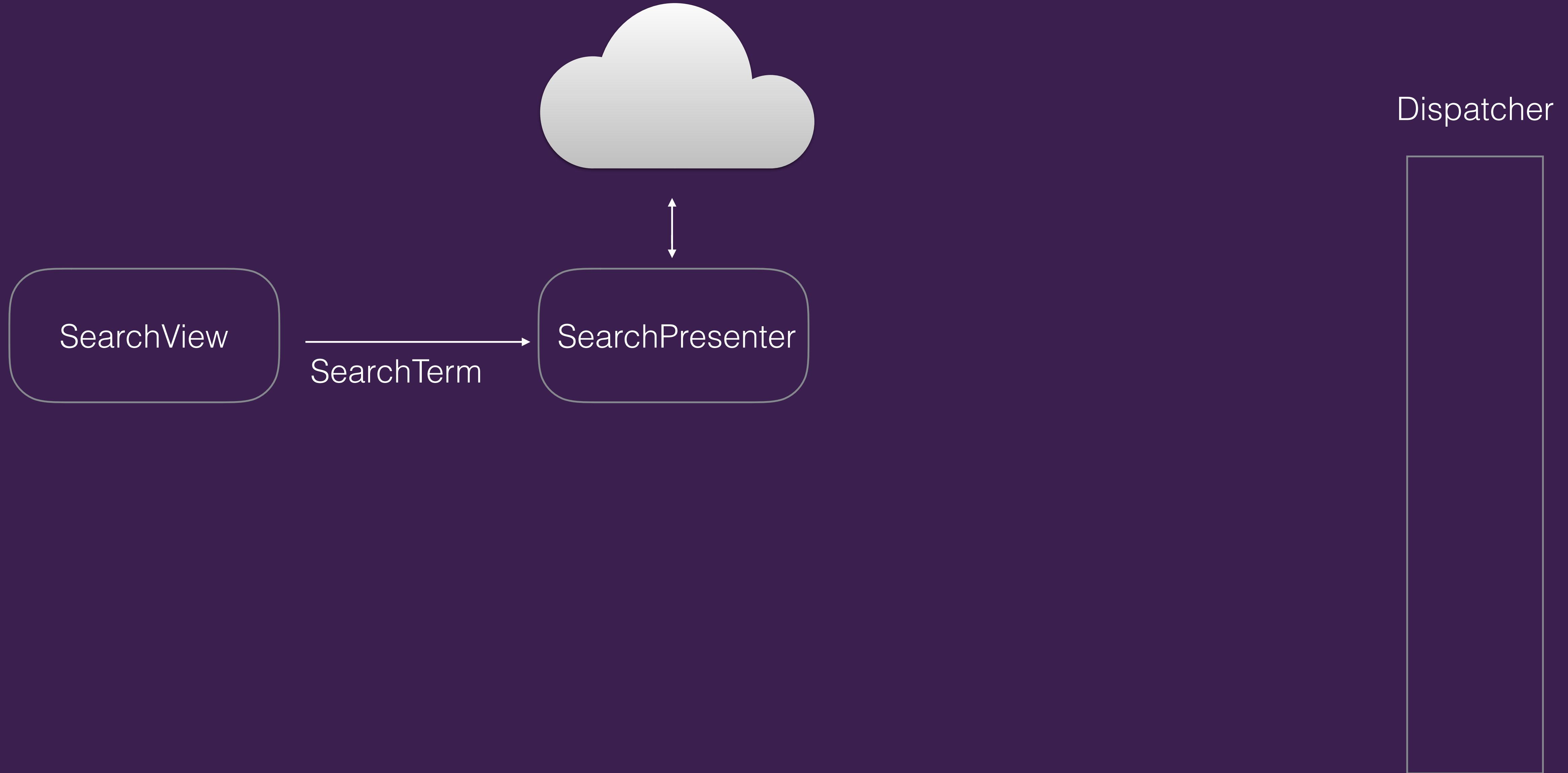
# Reactive state



# Reactive state

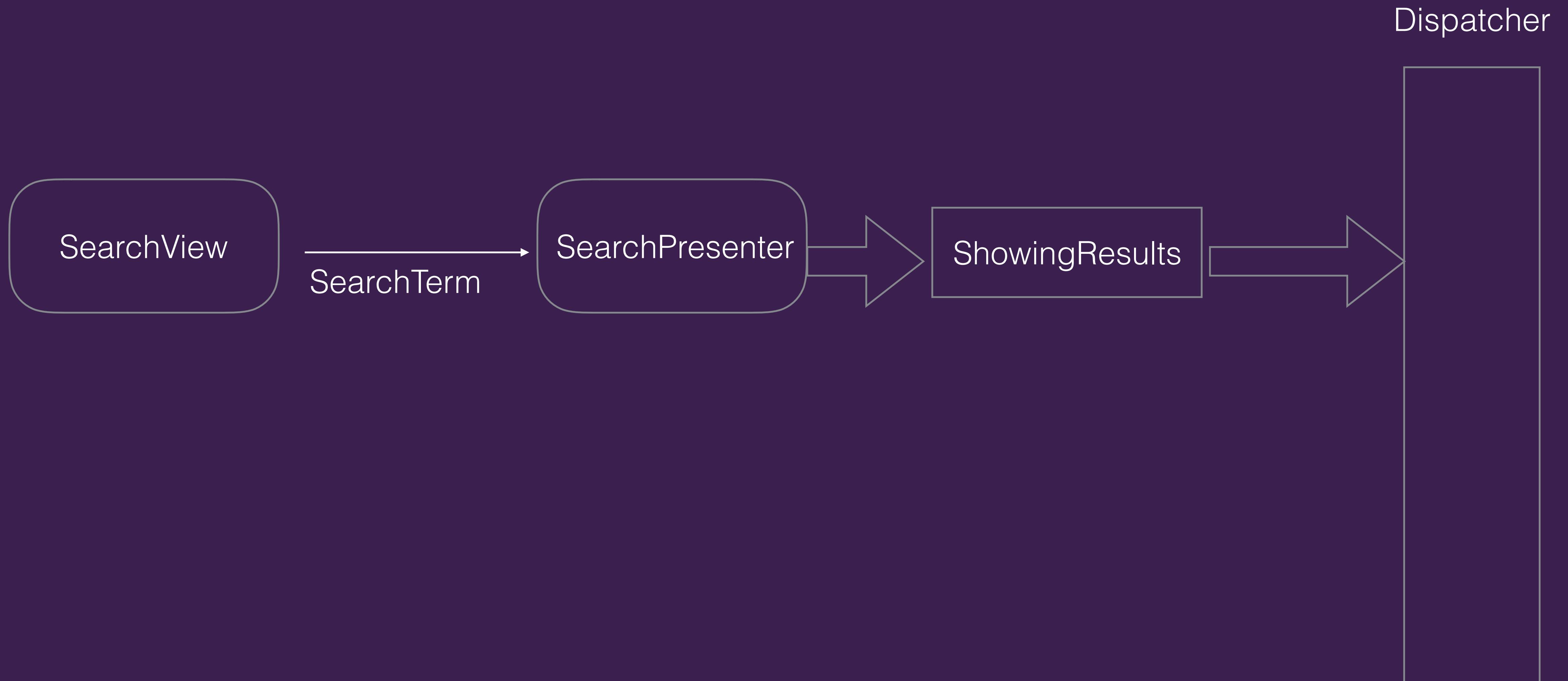


# Reactive state



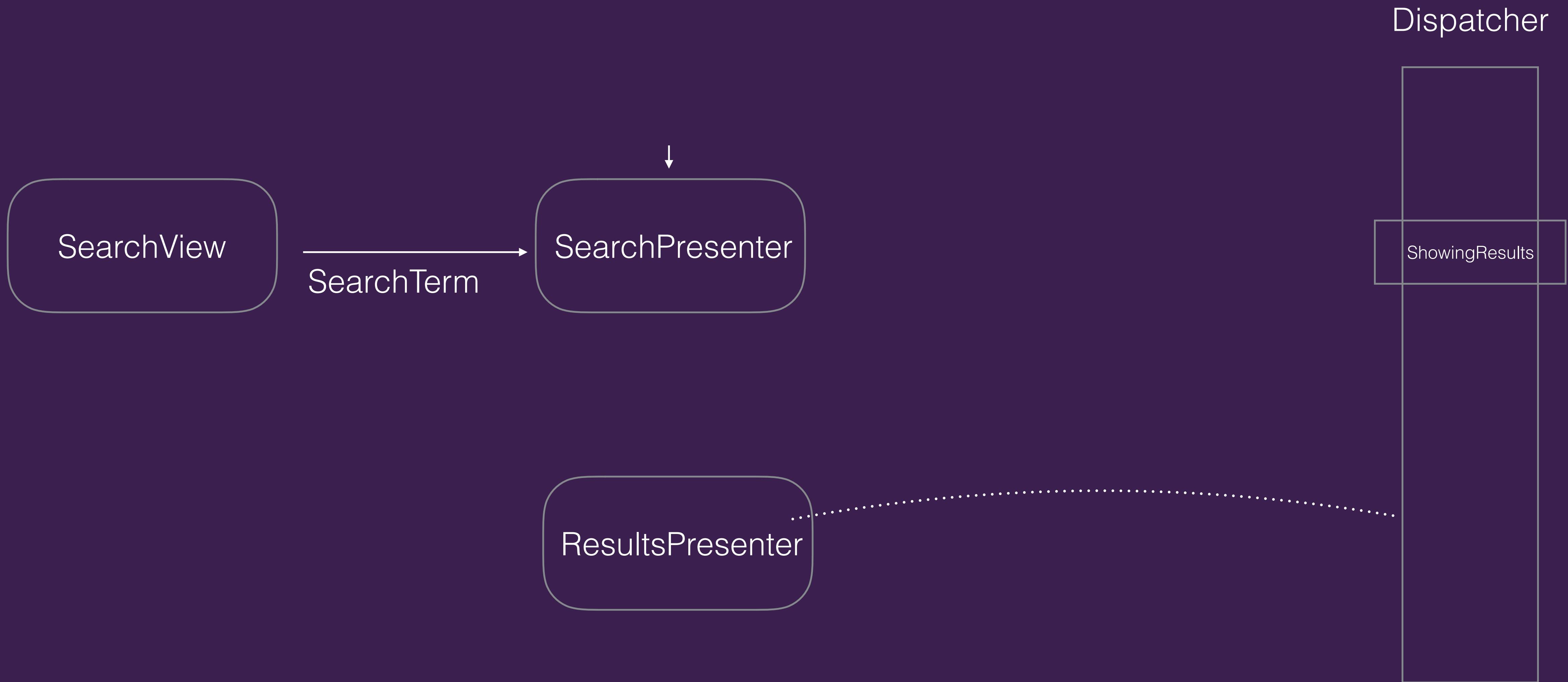
# Reactive state

Search Presenter calls `dispatcher.dispatch(State.ShowingResults(resultData))`



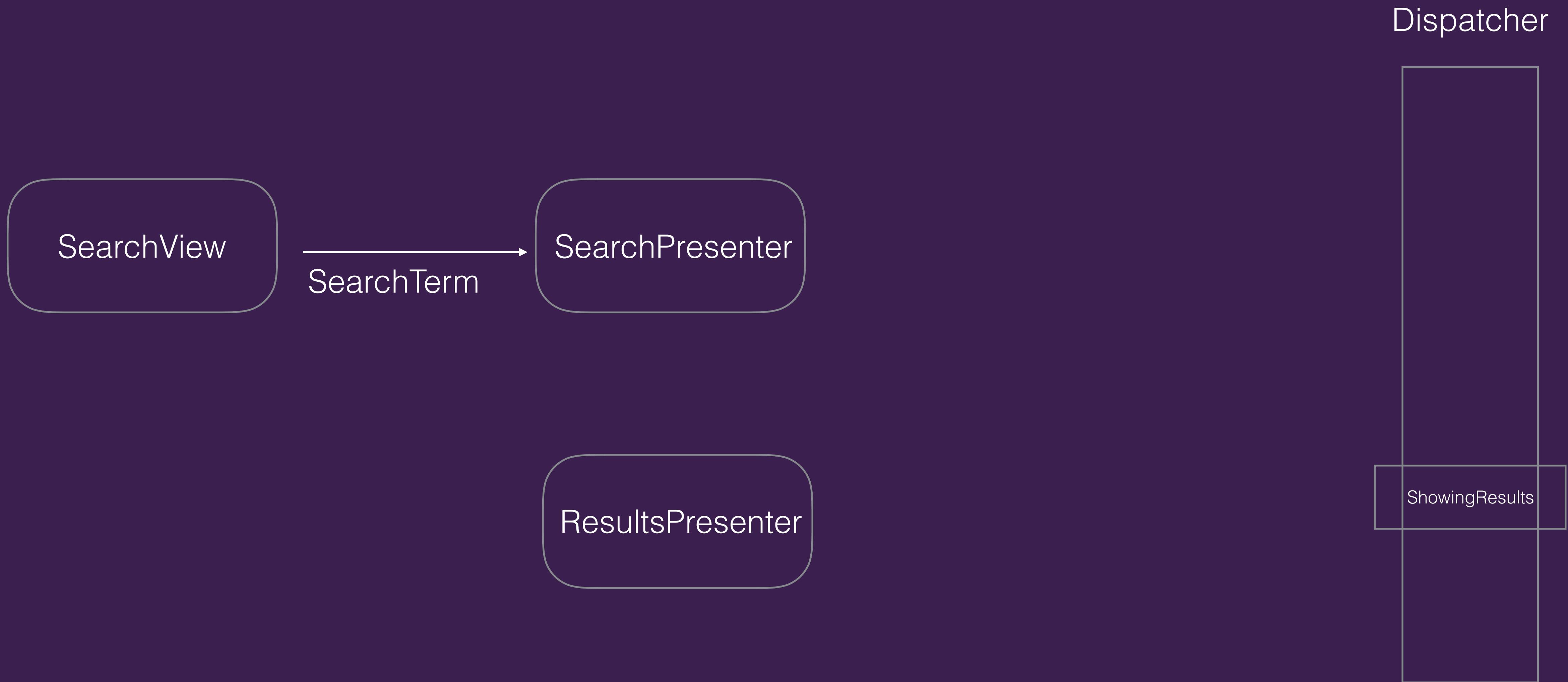
# Reactive state

**Dispatcher needs to emit a new State to subscribers**



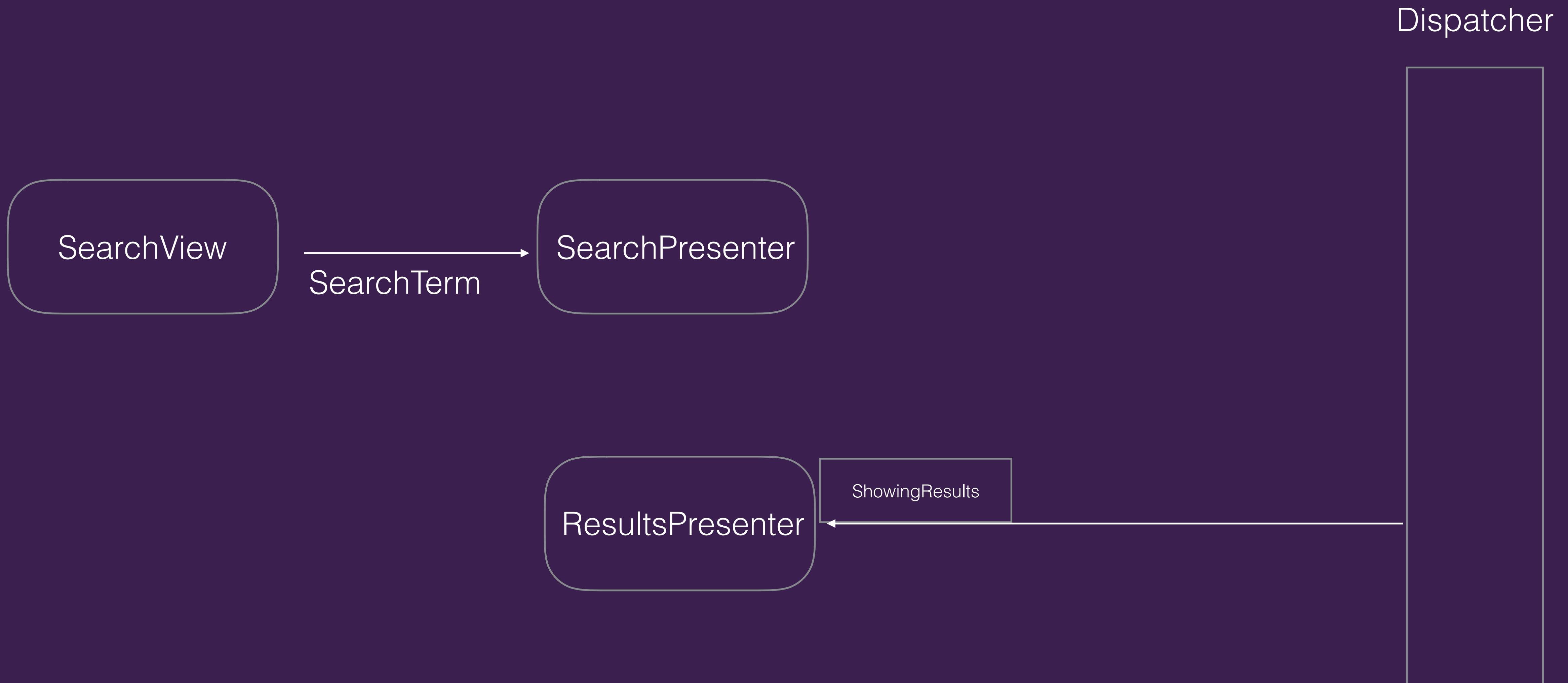
# Reactive state

Dispatcher Emits new ShowingResults State to ResultsPresenter



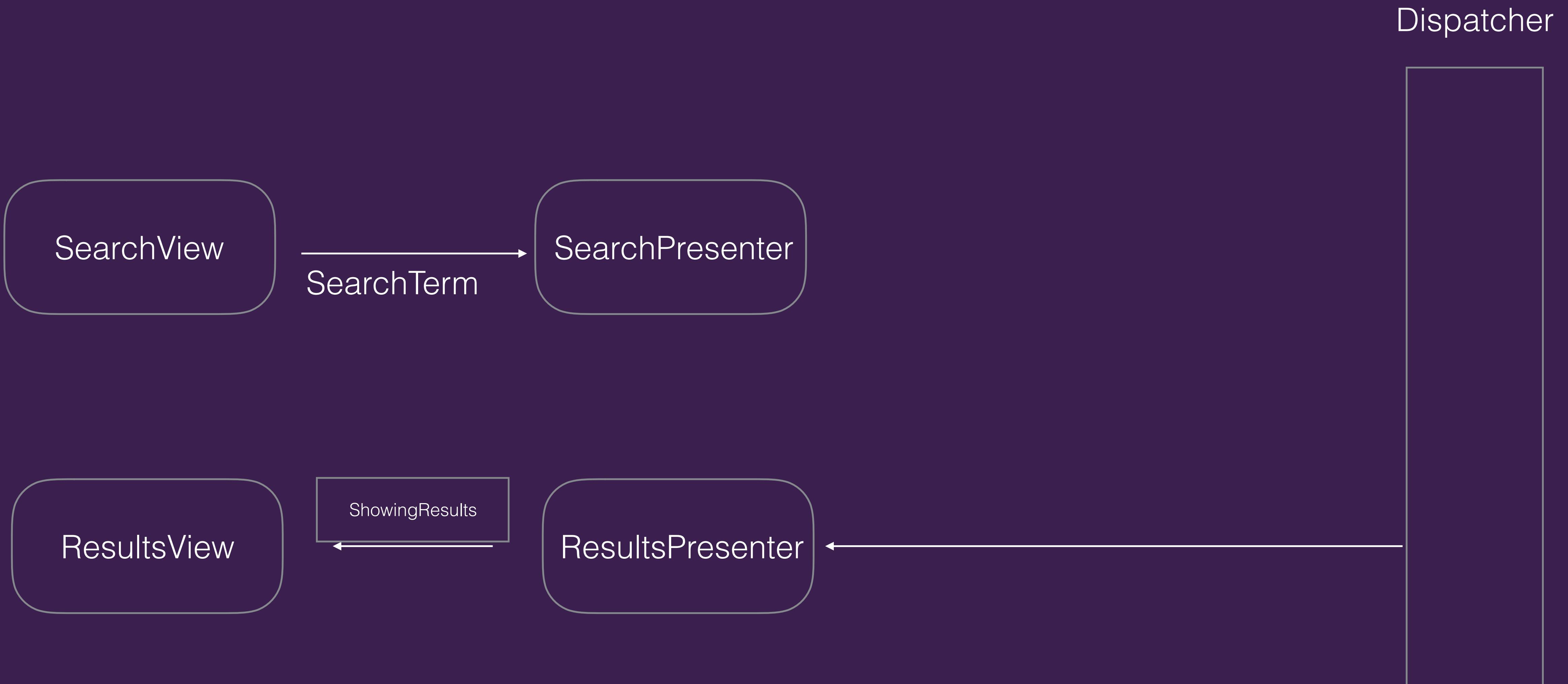
# Reactive state

**Dispatcher Emits new ShowingResults State to subscribers**



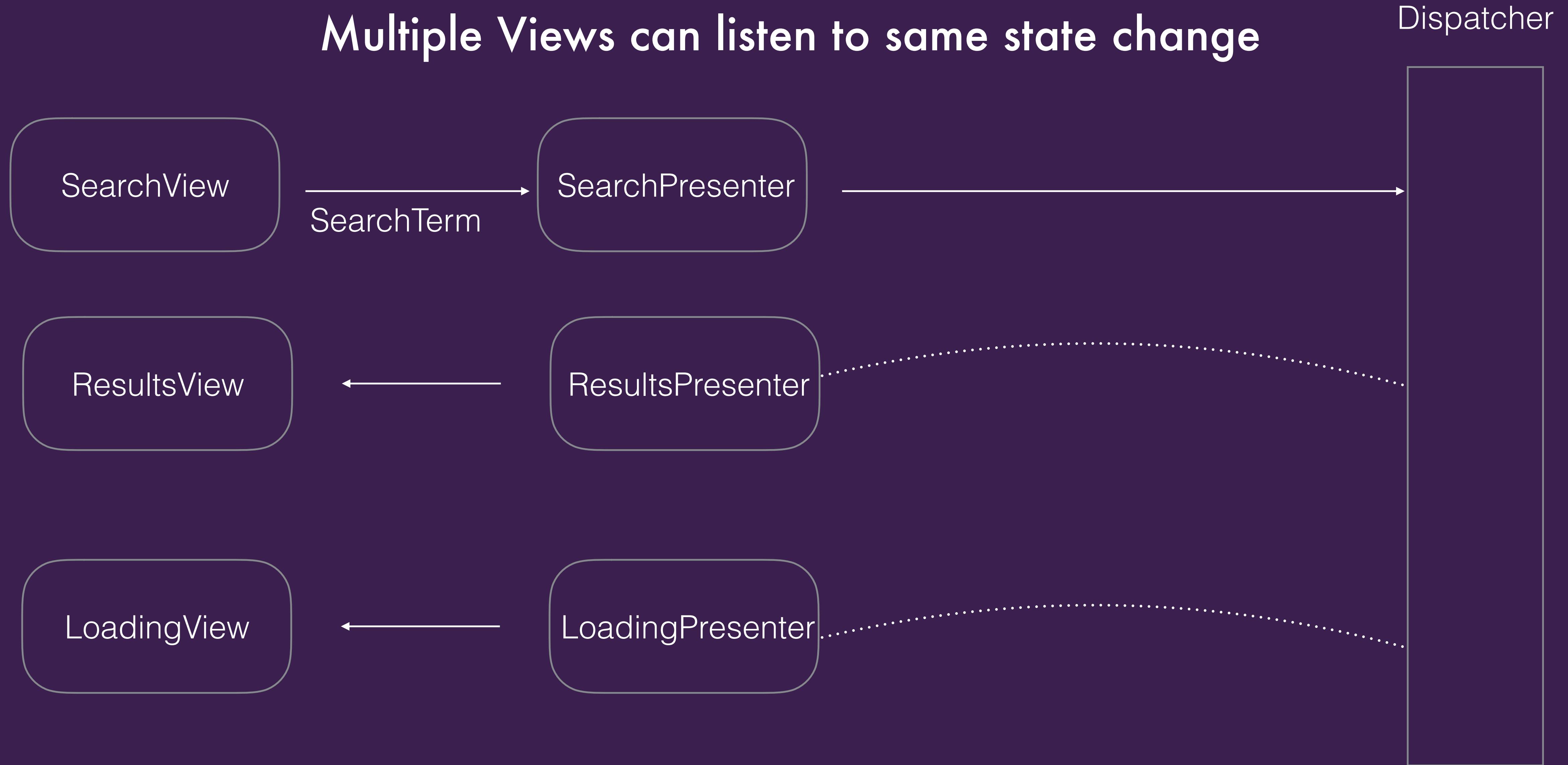
# Reactive state

Dispatcher Emits new ShowingResults State to subscribers

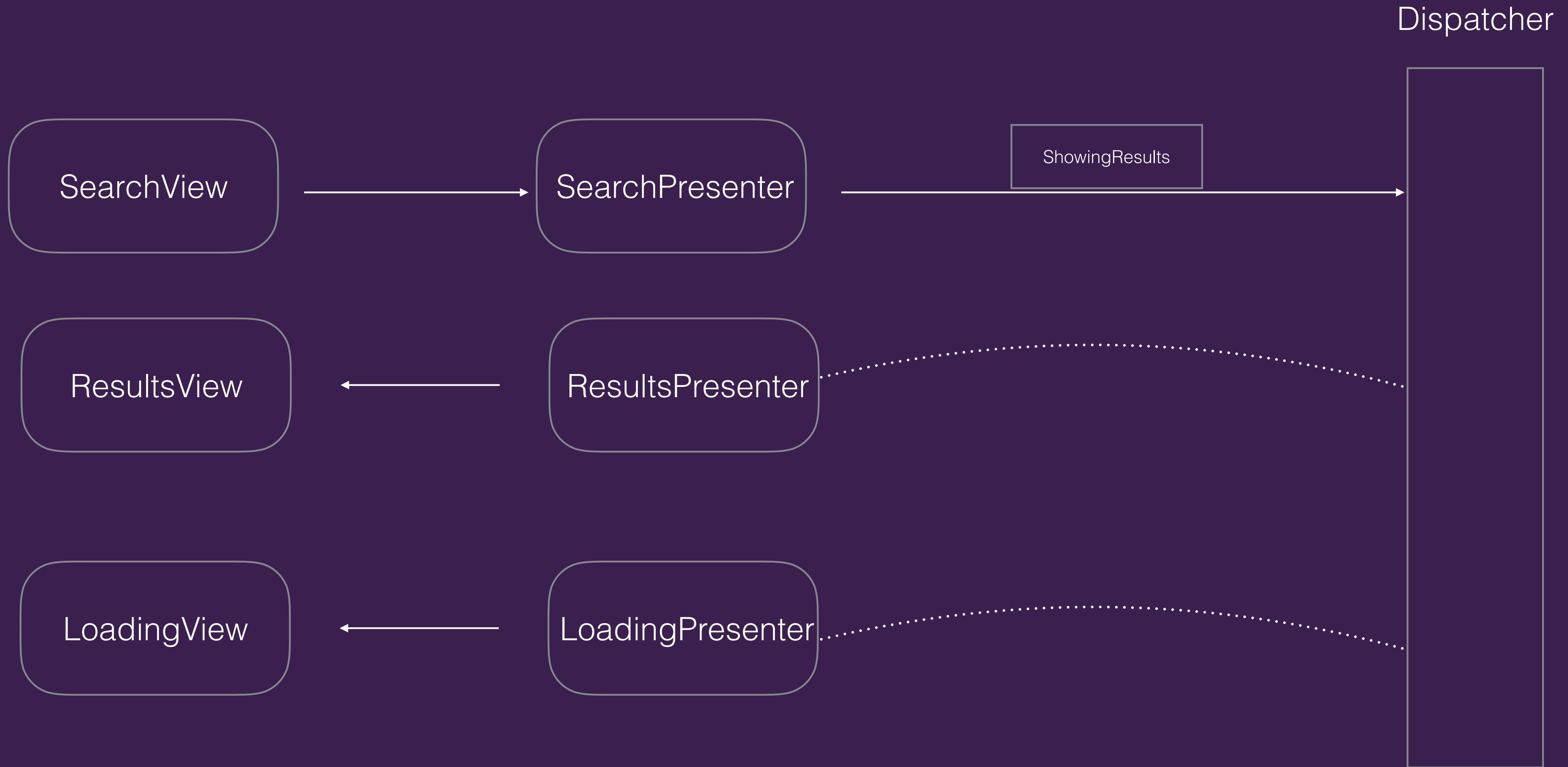


# Reactive state

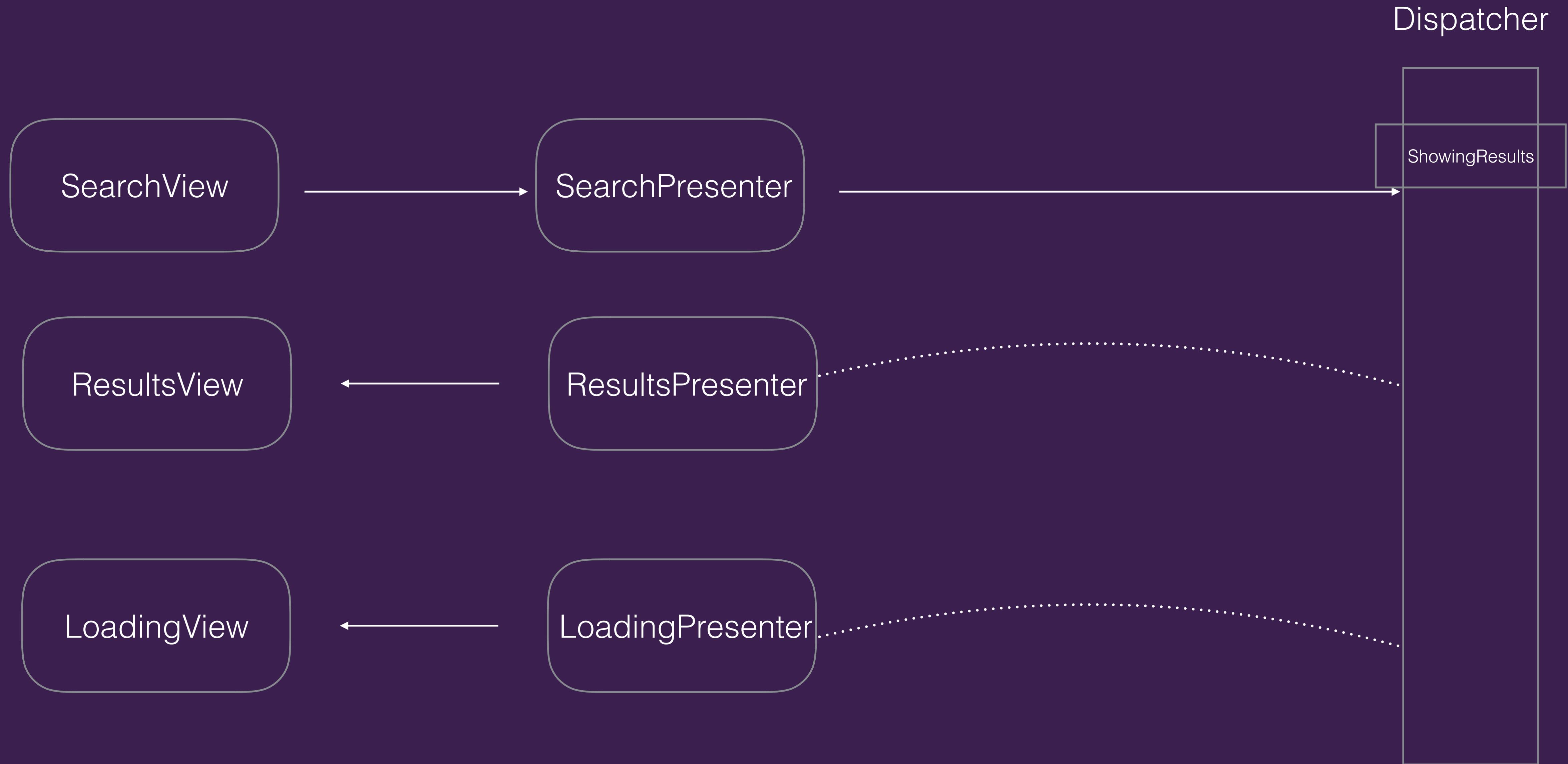
**Multiple Views can listen to same state change**



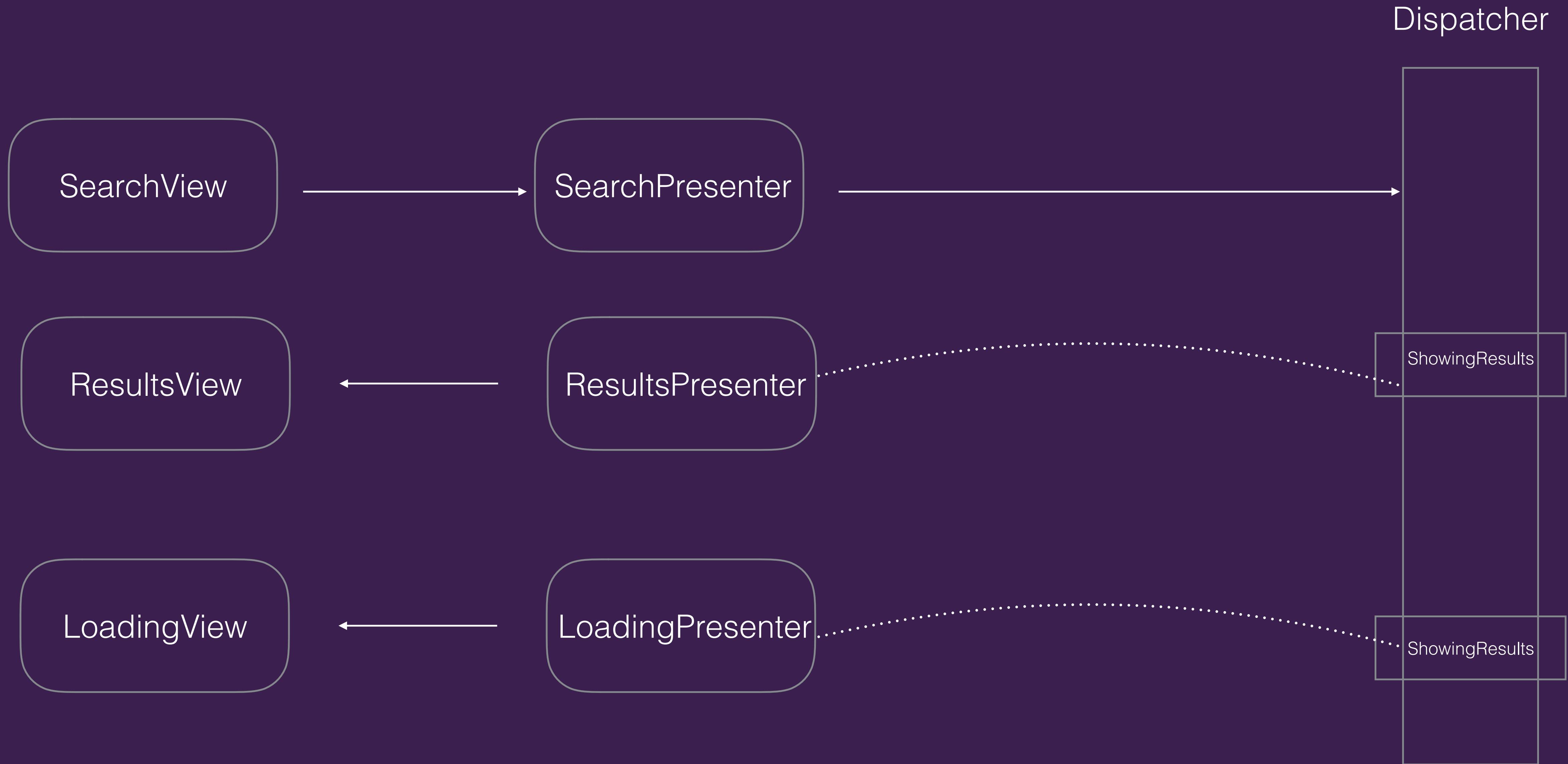
# Reactive state



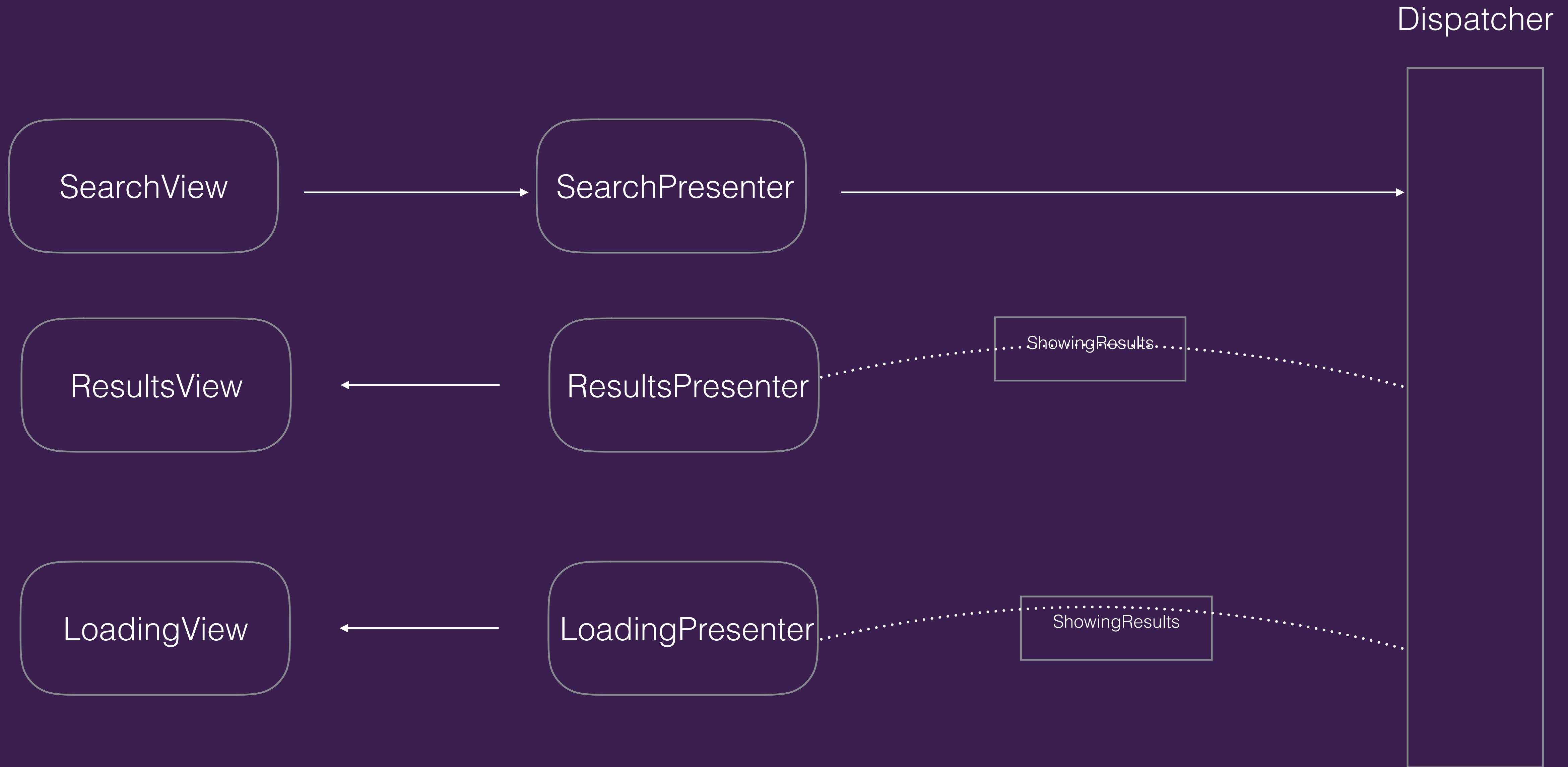
# Reactive state



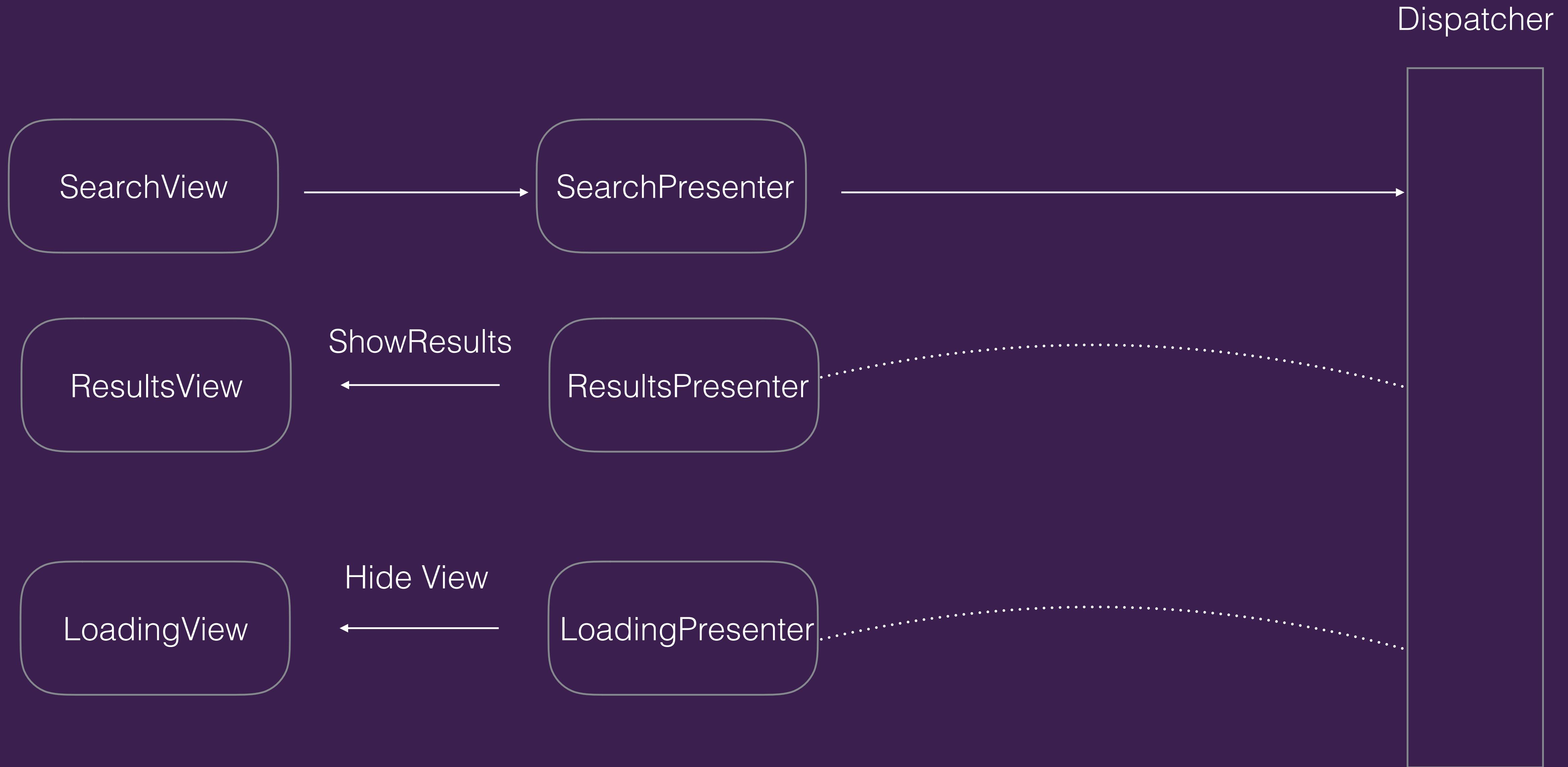
# Reactive state



# Reactive state



# Reactive state



A close-up photograph of two kittens. One kitten is white with brown spots, looking up with its mouth slightly open. The other kitten has dark brown fur with light stripes and is looking directly at the camera with wide, greenish-yellow eyes. They appear to be playing or communicating.

# Interacting with a dispatcher

# Presenter subscribes on attach



```
override fun attachView(view: NearYouMVPView) {  
    rxState ofType(State.Results)  
        .map{it.data}  
        .subscribe{ mvpView.updateUI(data) }  
}
```

# Another presenter dispatches state change



```
fun searchFor(searchTerm:String){  
    store.getResults(searchTerm)  
    .subscribe{dispatcher.dispatch(State.Results(data=it.results))}
```

# Dispatcher emits state change



```
fun searchFor(searchTerm:String){  
    store.getResults(searchTerm)  
        .subscribe{dispatcher.dispatch(State.Results(data=it.results))}
```

```
override fun attachView(view: NearYouMVPView) {  
    rxState ofType(State.Results)  
        .map{it.data}  
        .subscribe{ mvpView.updateUI(data) } ←
```

# Consumers & Producers of state stay decoupled



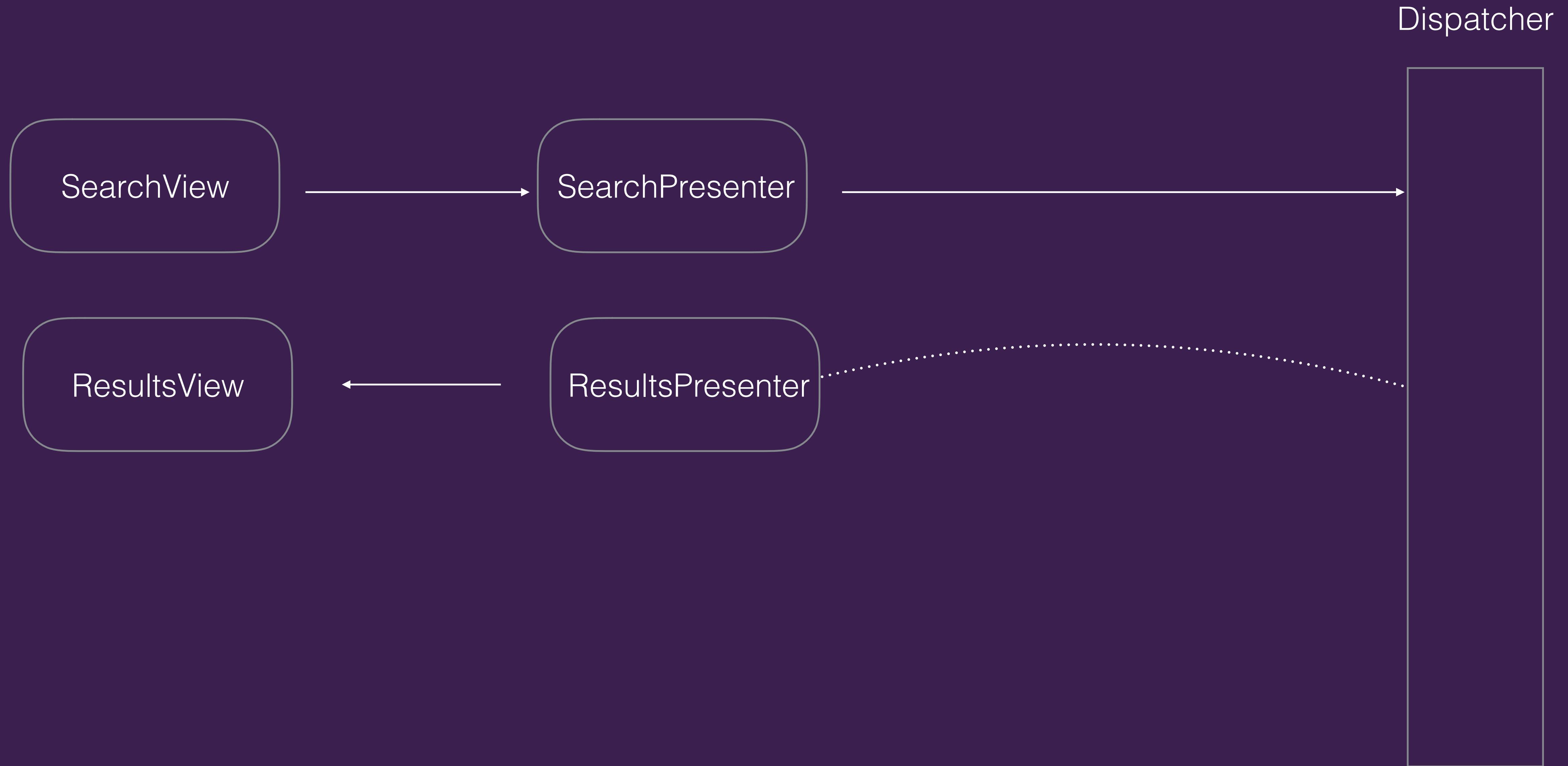
```
fun searchFor(searchTerm:String){  
    store.getResults(searchTerm)  
        .subscribe{dispatcher.dispatch(State.Results(data=it.results))}
```

```
override fun attachView(view: NearYouMVPView) {  
    rxState ofType(State.Results)  
        .map{it.data}  
        .subscribe{ mvpView.updateUI(data) } ←
```

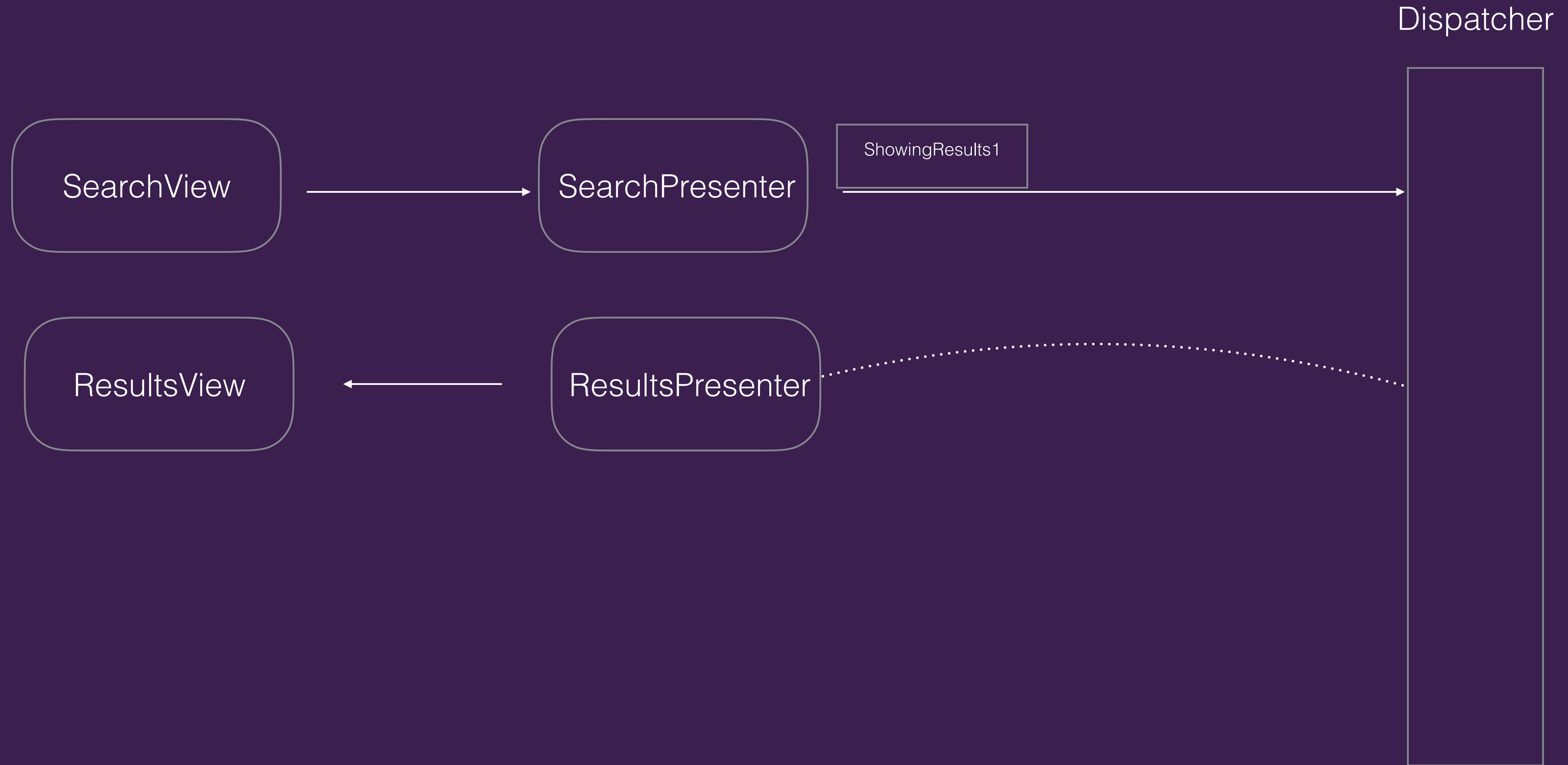


How does data flow?

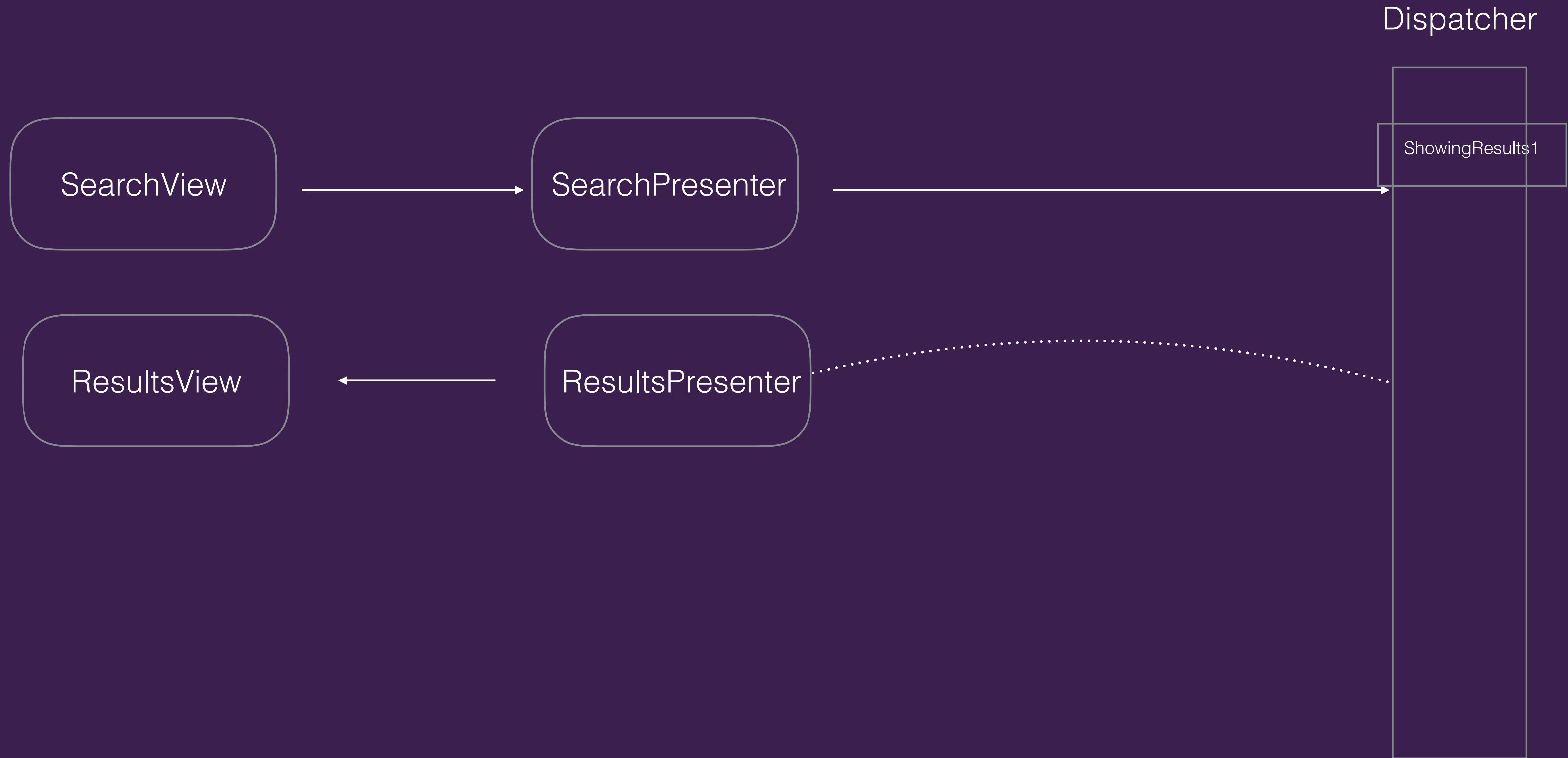
# Reactive state



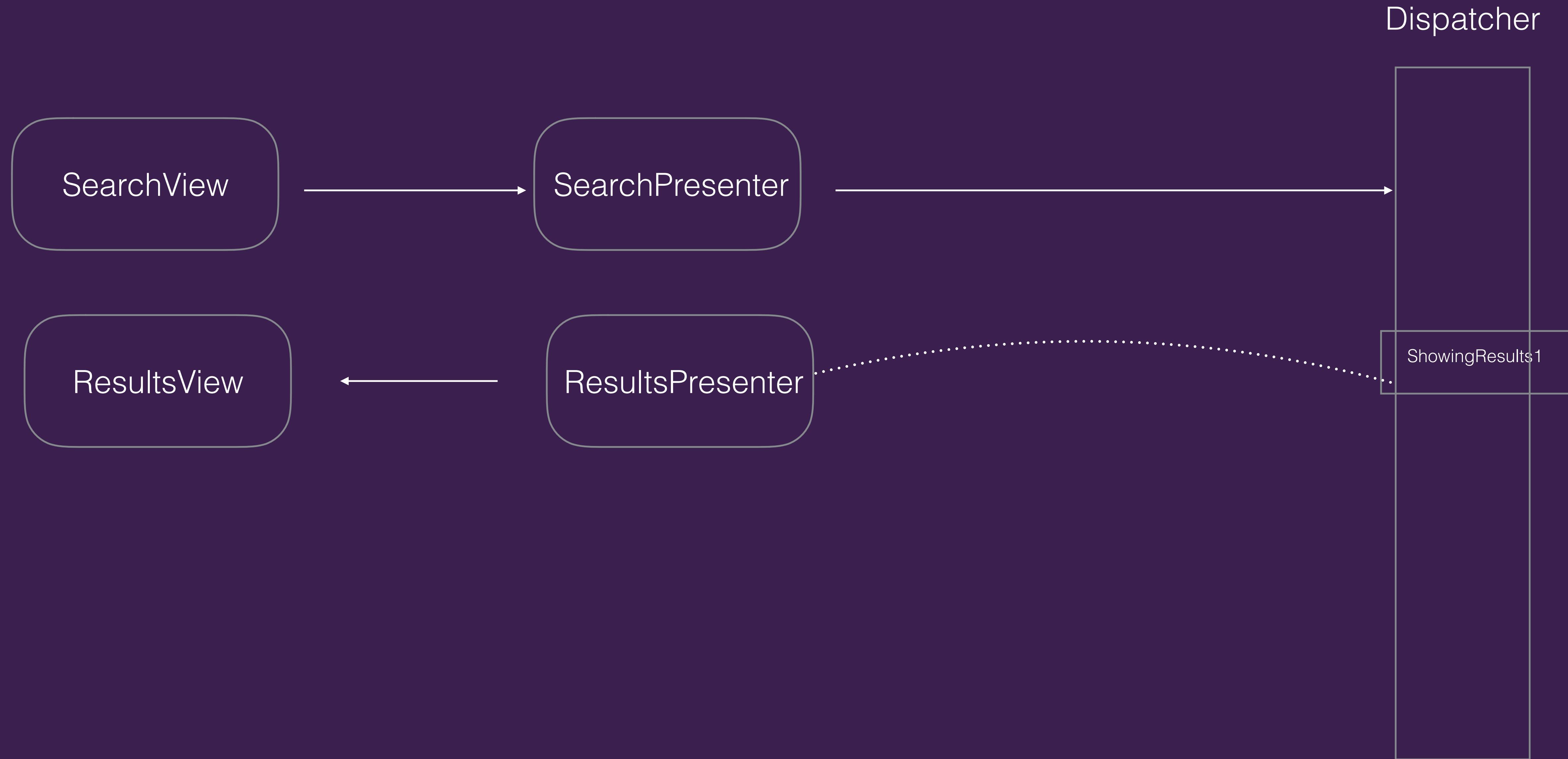
# Reactive state



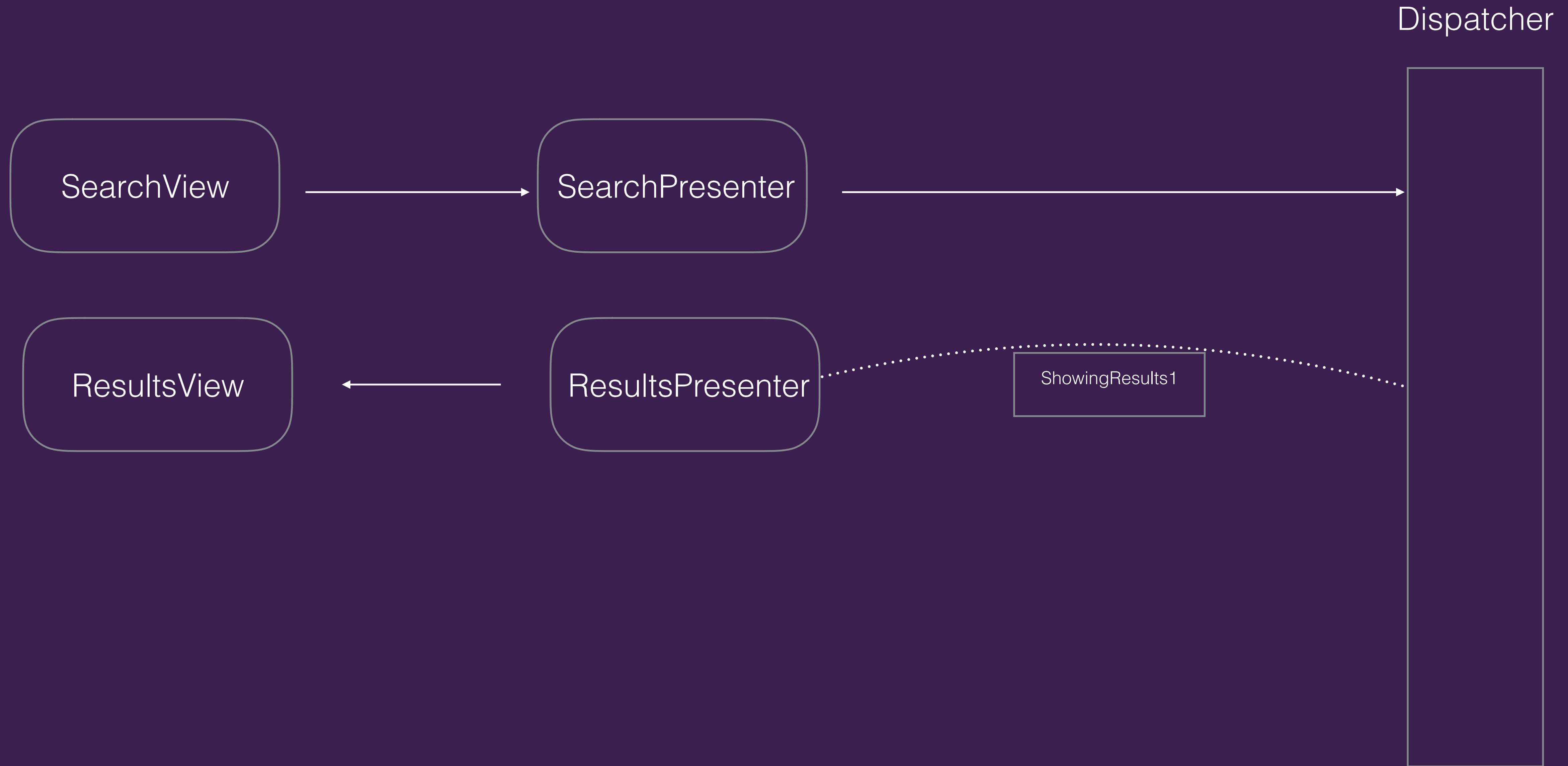
# Reactive state



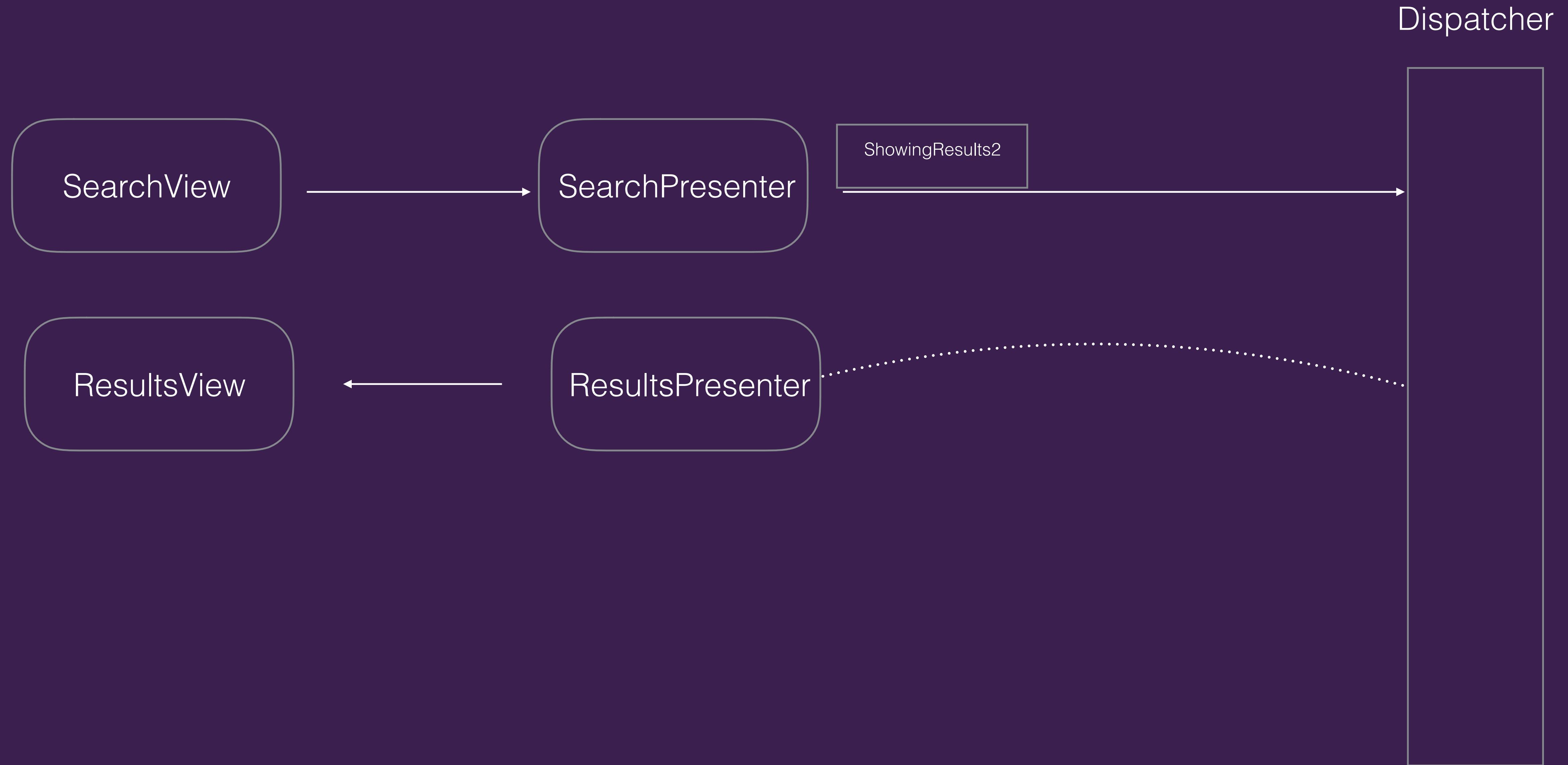
# Reactive state



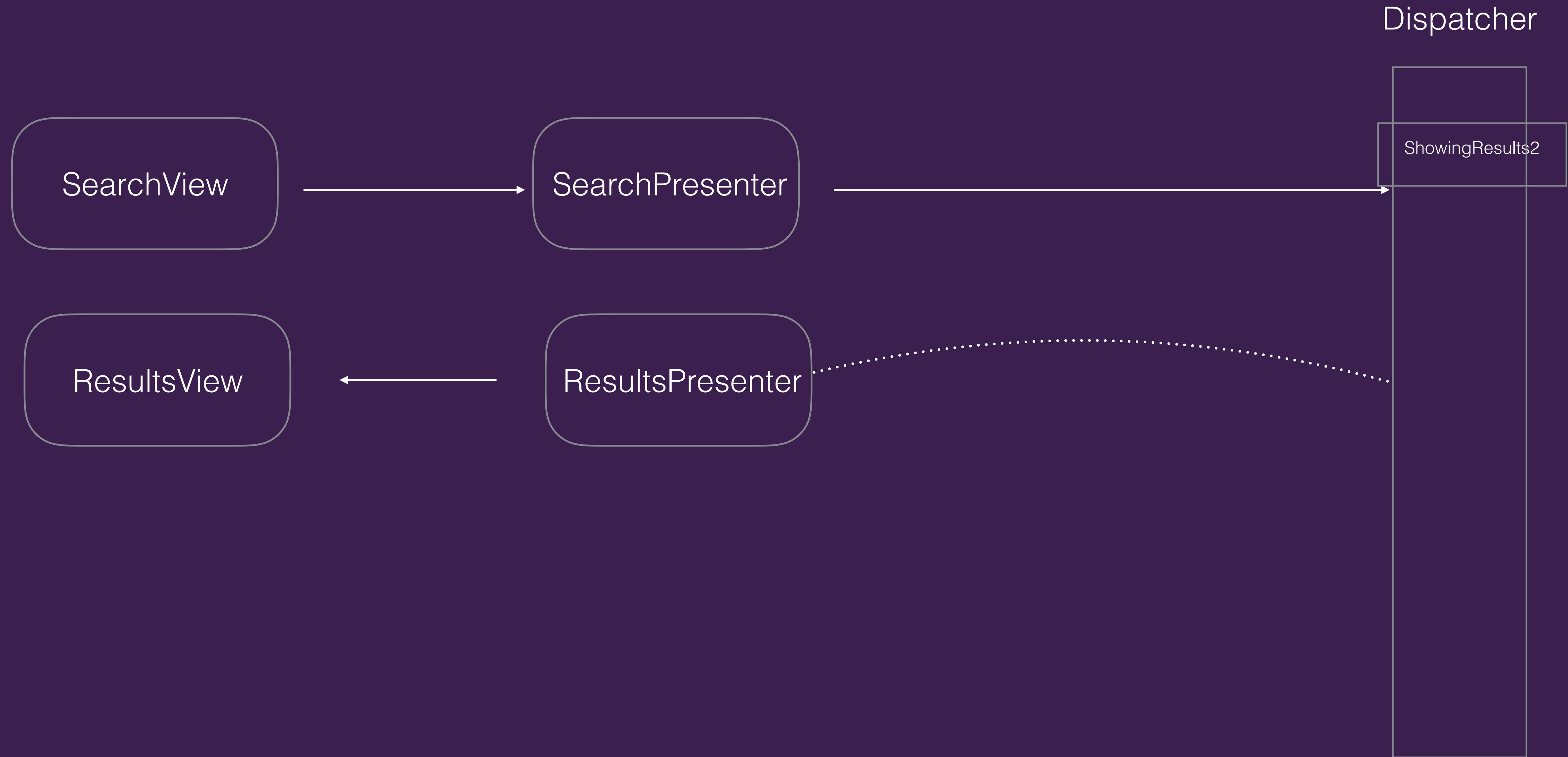
# Reactive state



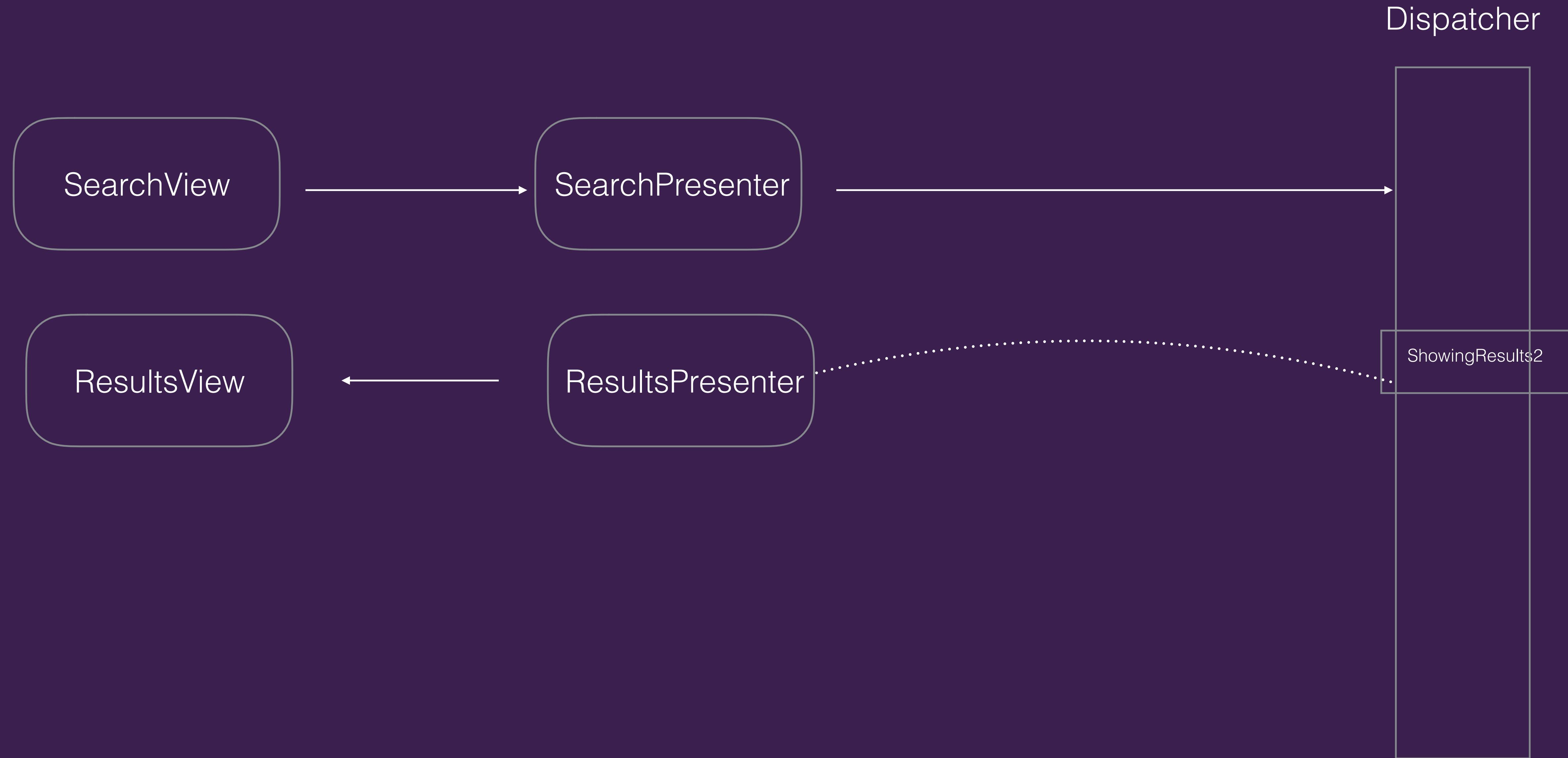
# Reactive state



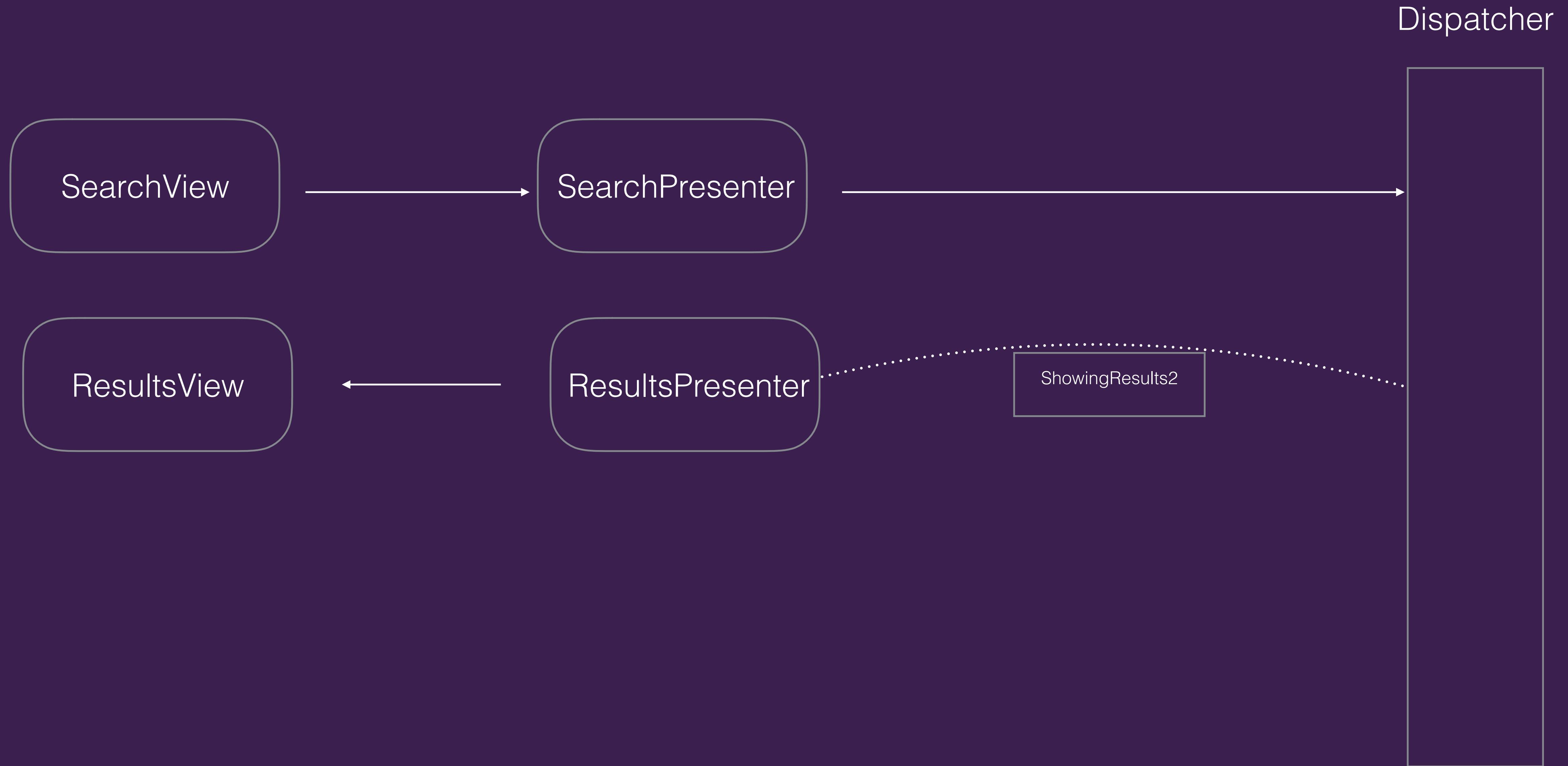
# Reactive state



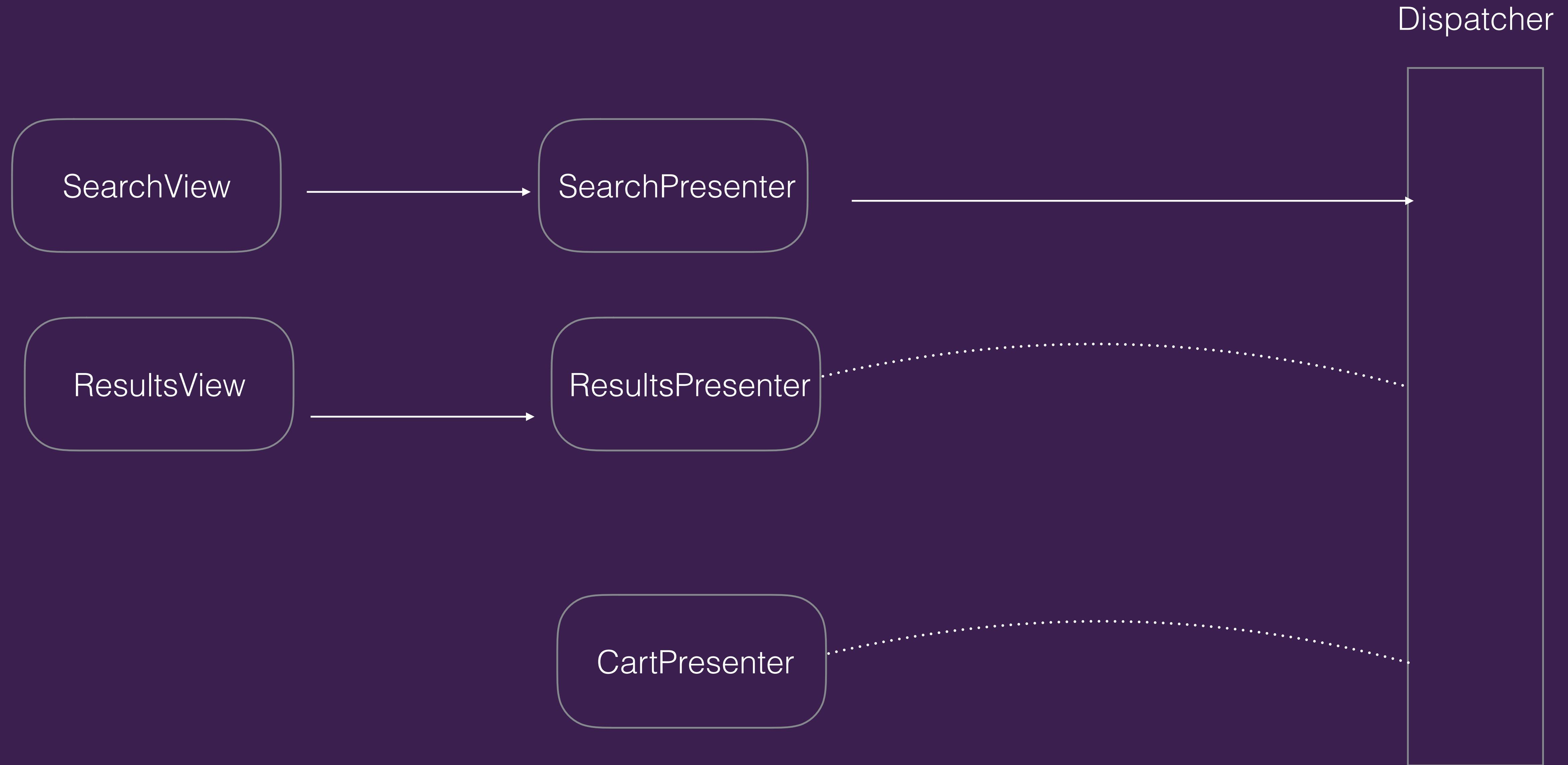
# Reactive state



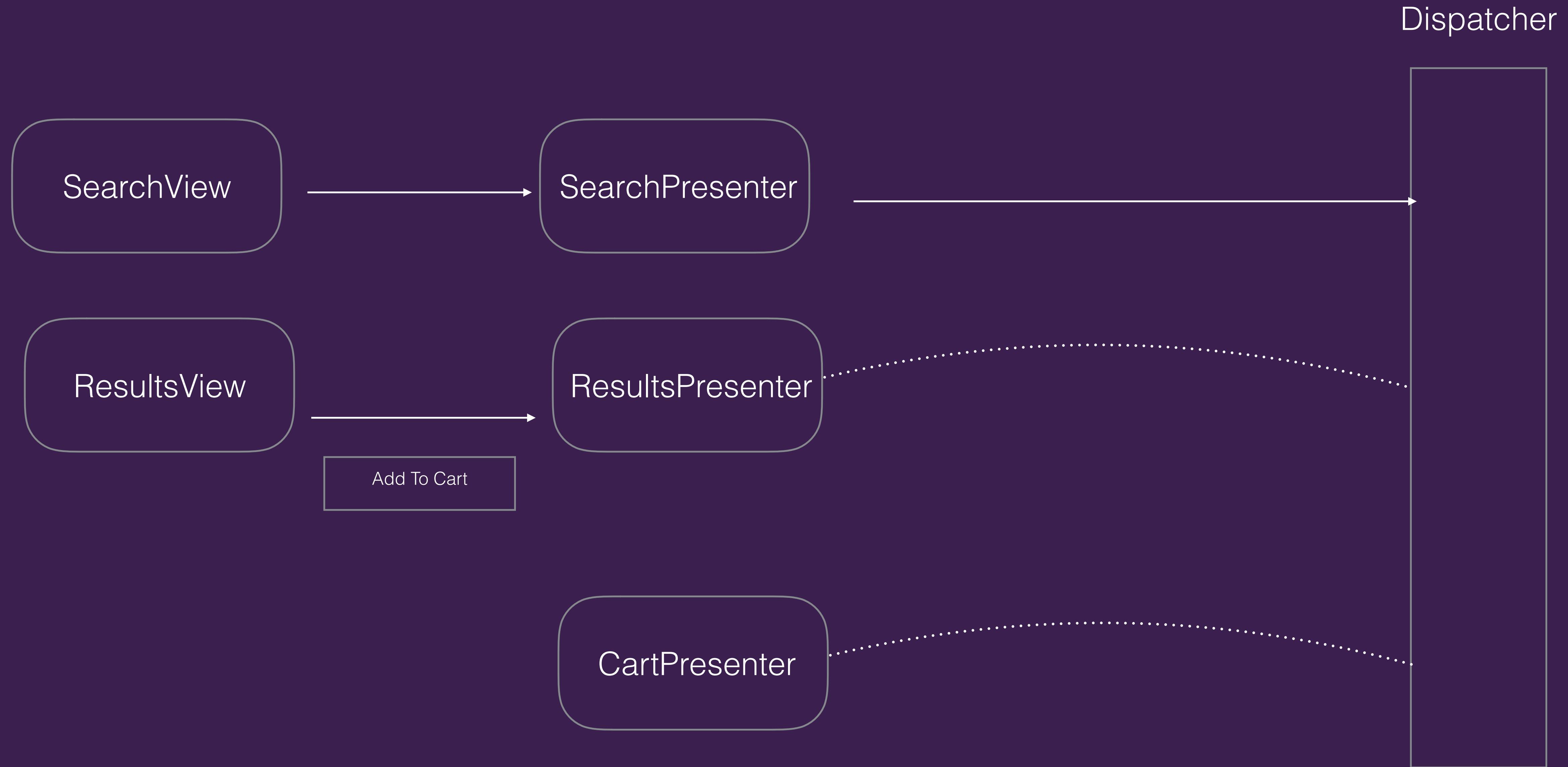
# Reactive state



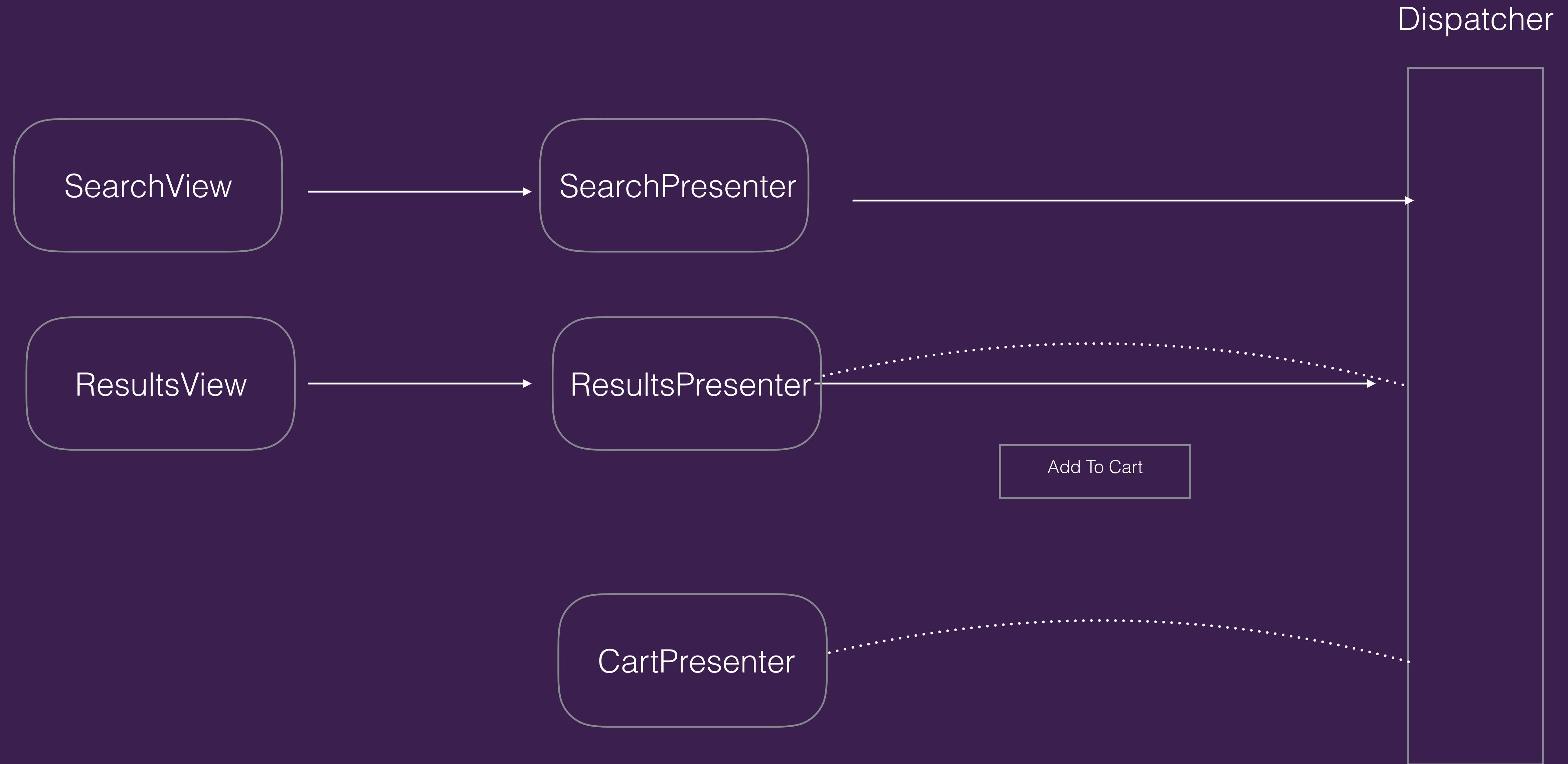
# Reactive state



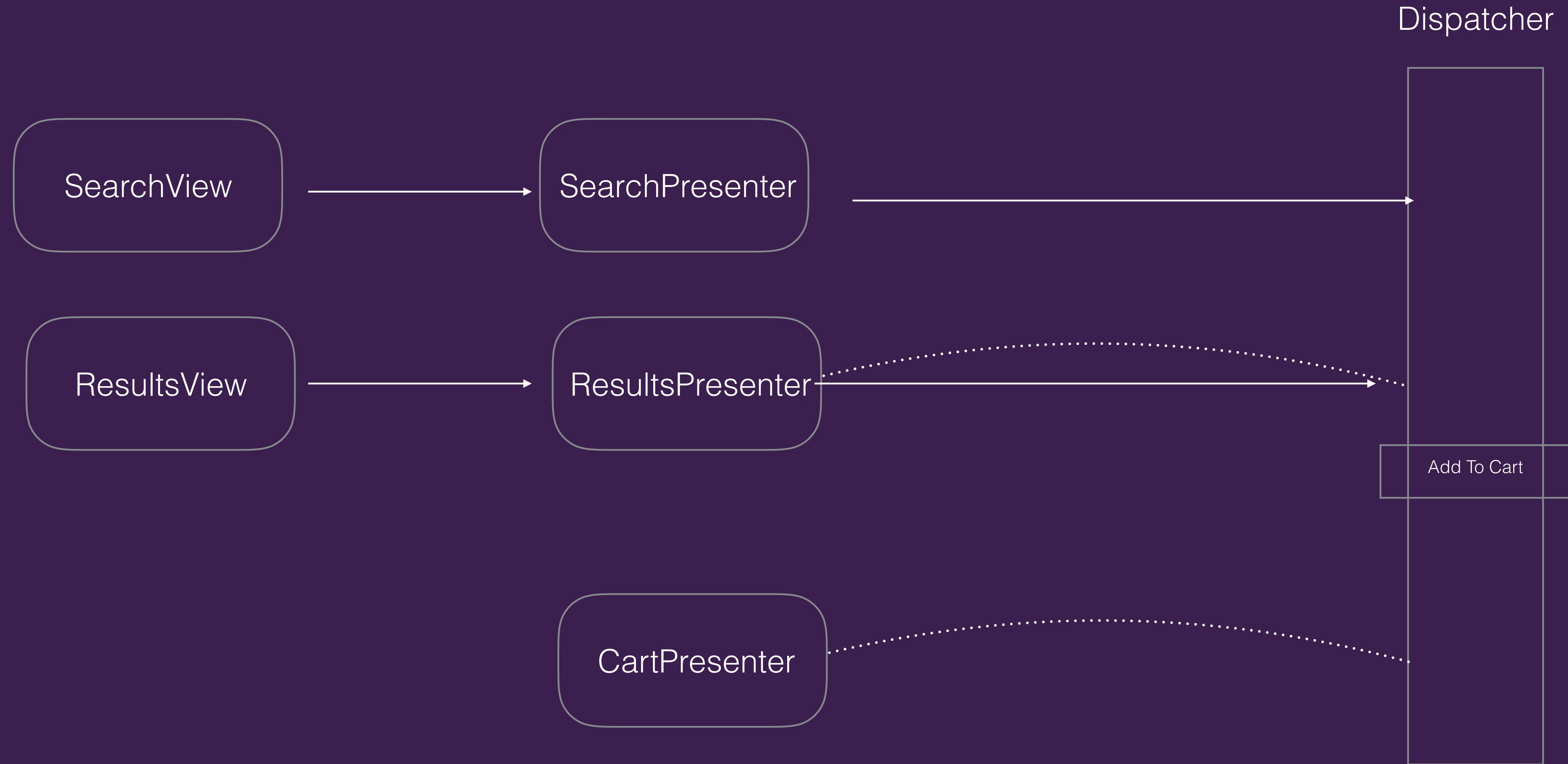
# More state



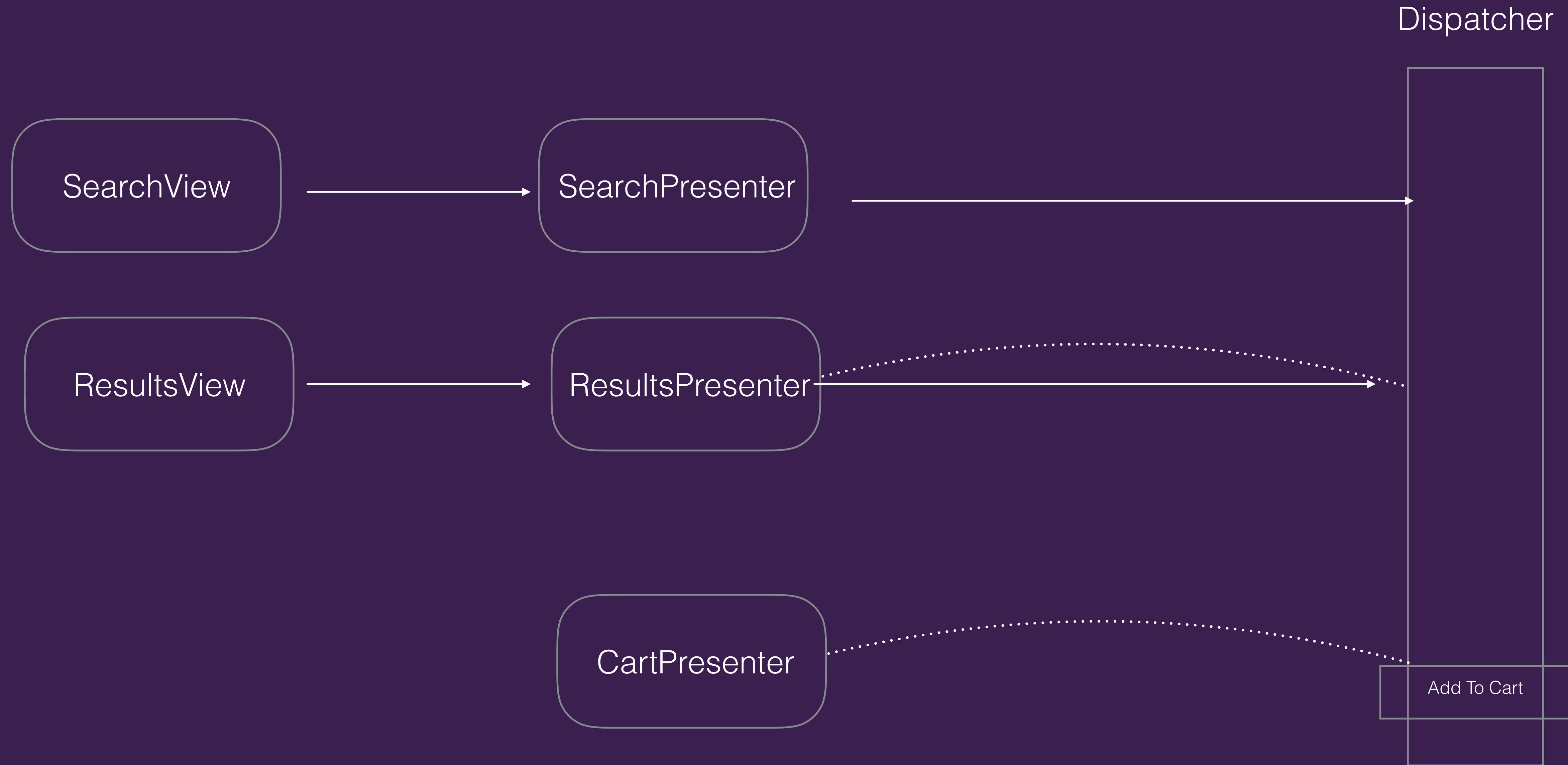
# More state



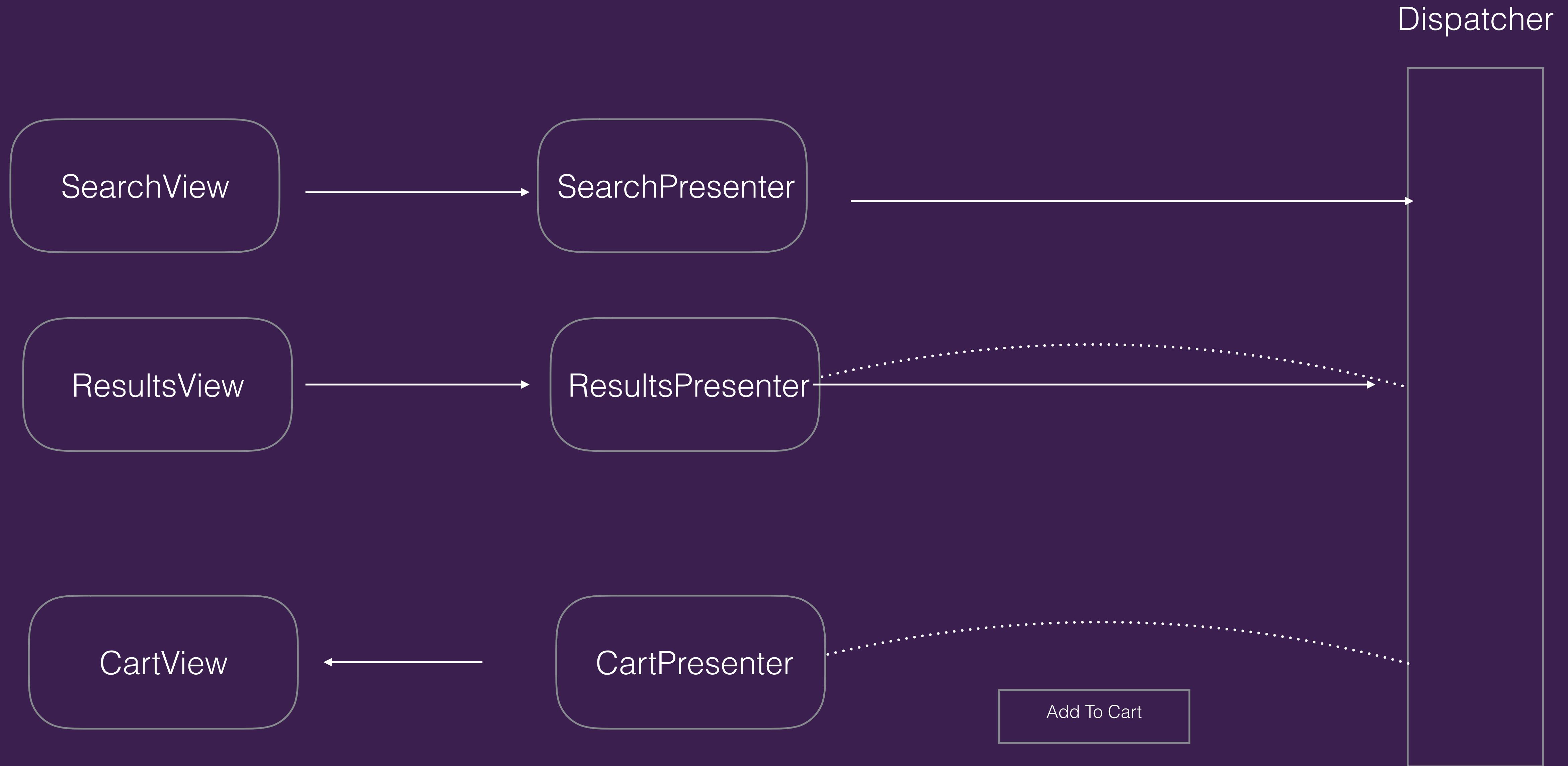
# More state



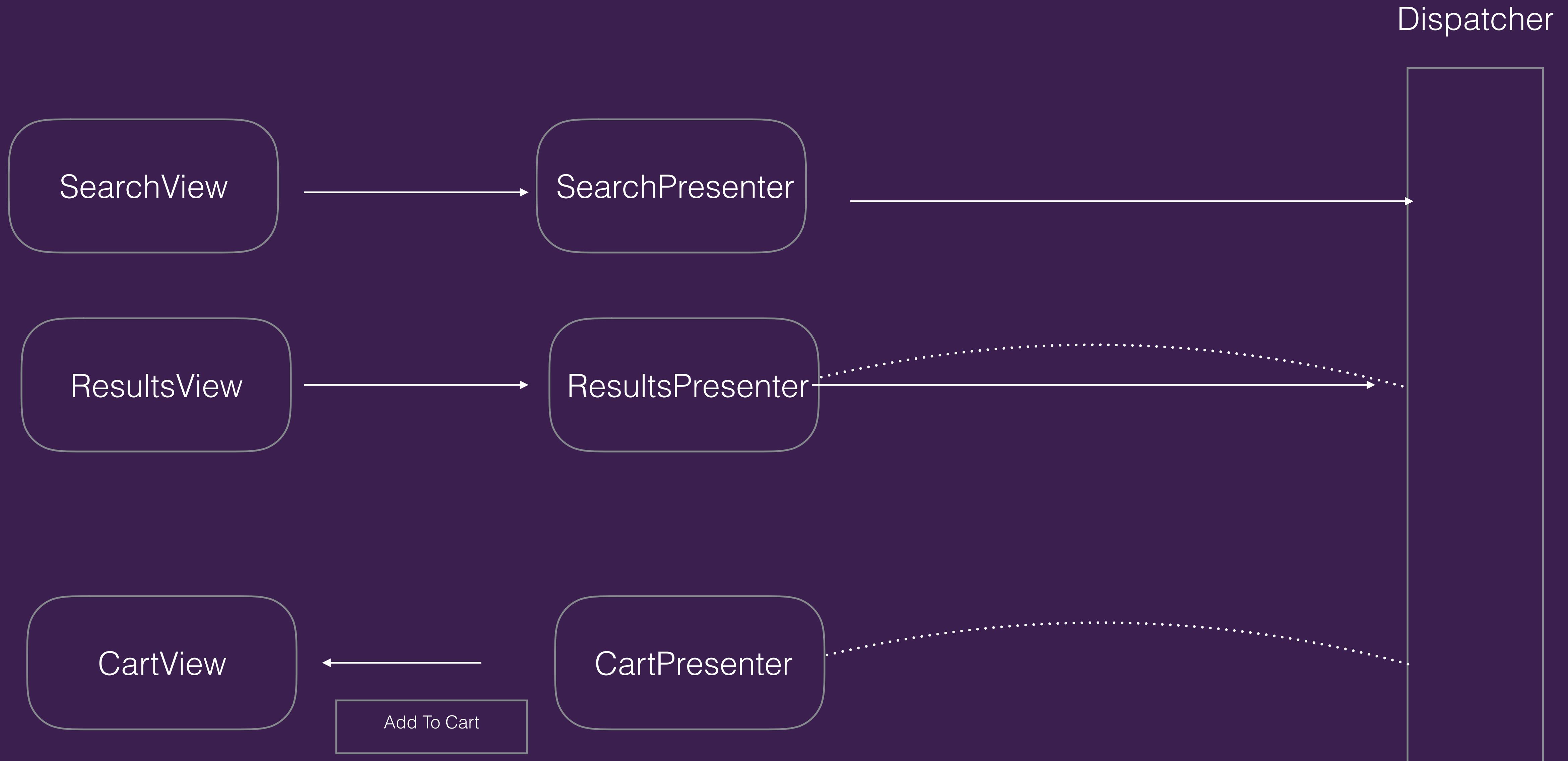
# More state



# More state



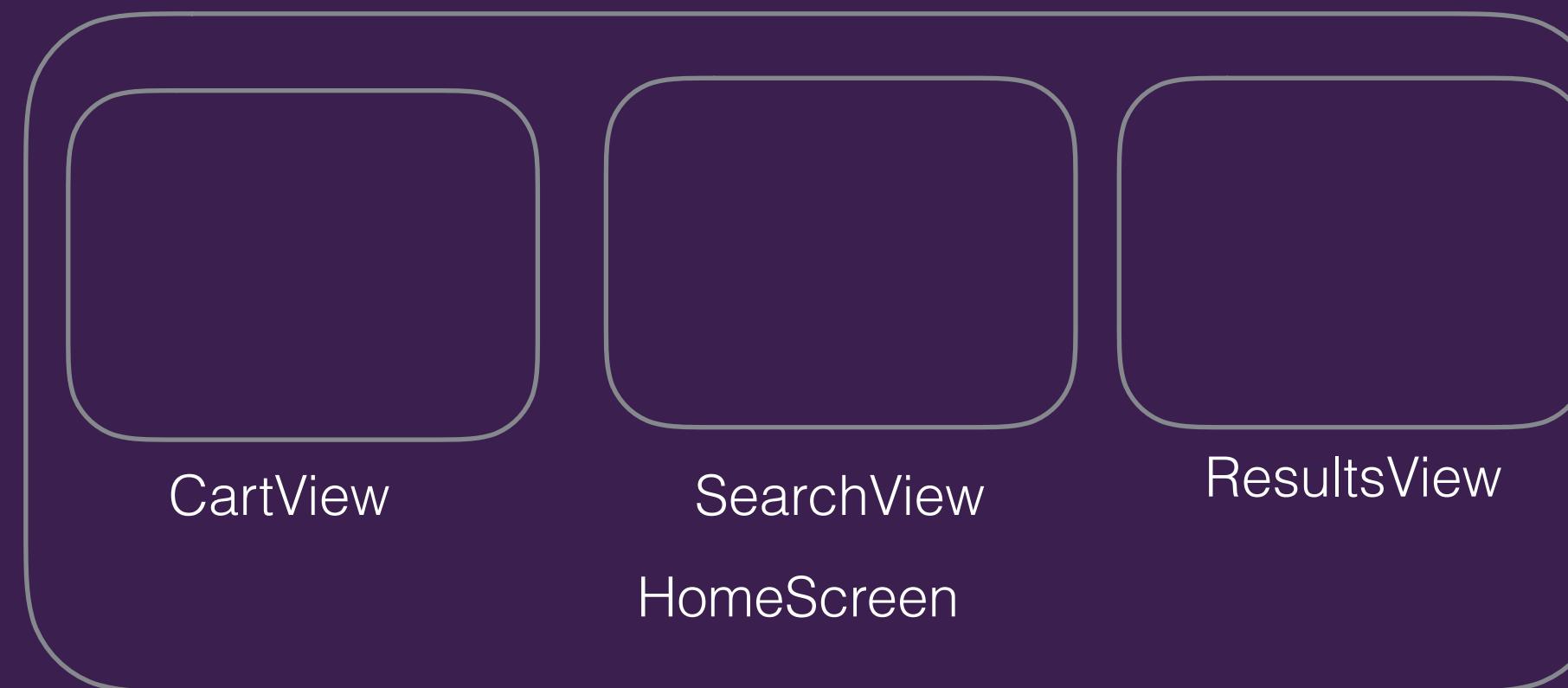
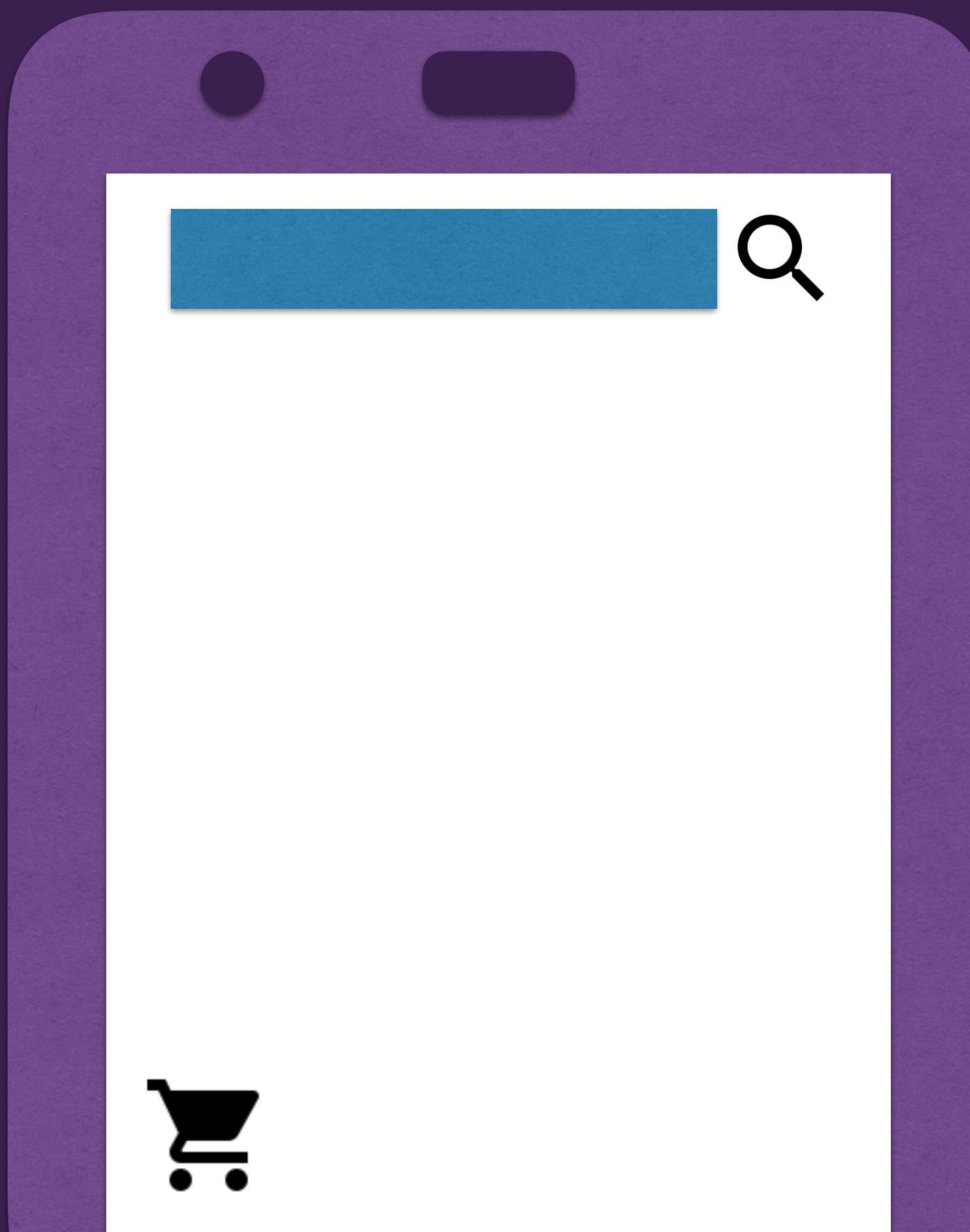
# More state



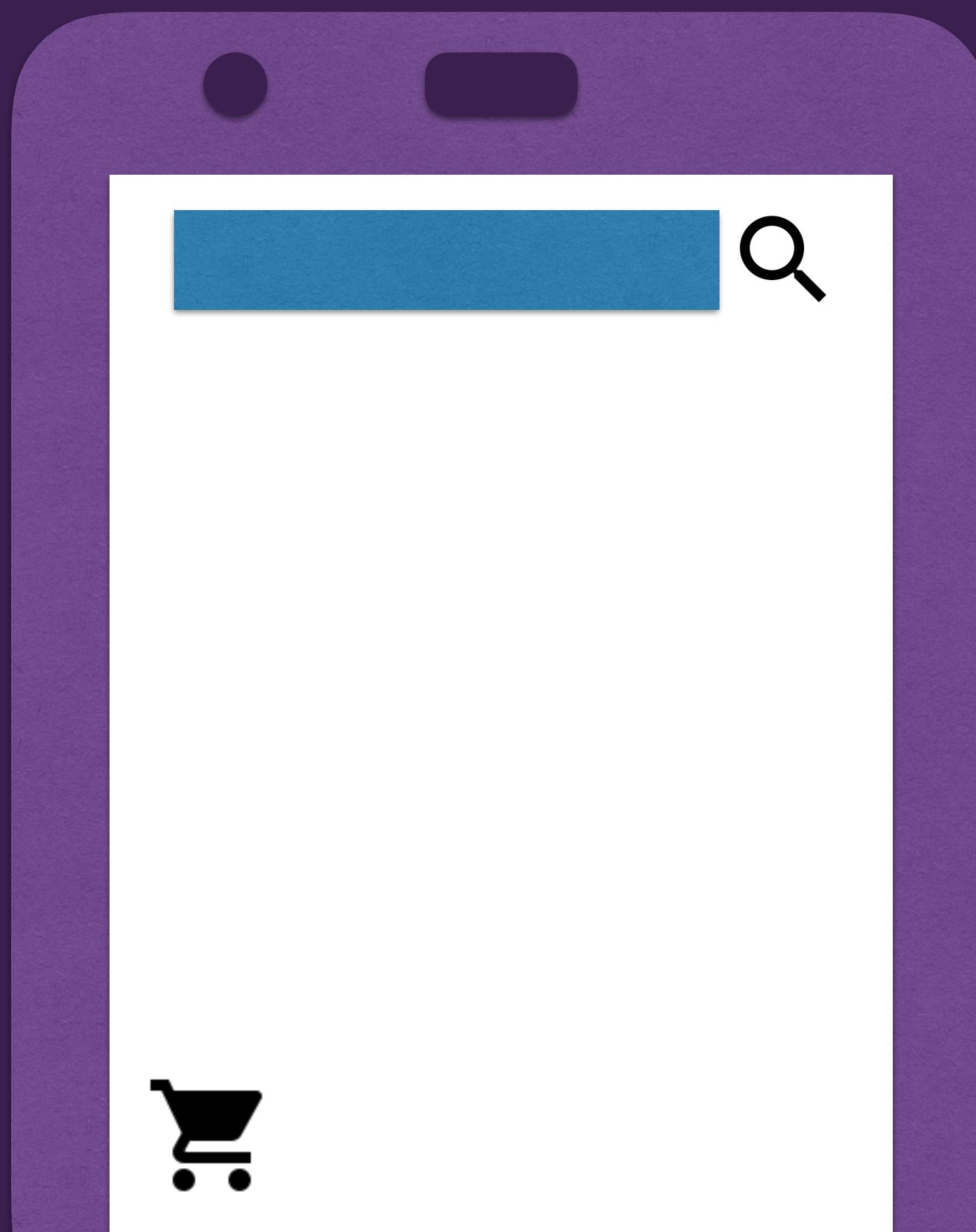


Emitting data is nice...  
How about showing/hiding views?

# When user clicks on cart, we should show the cart view



# How?

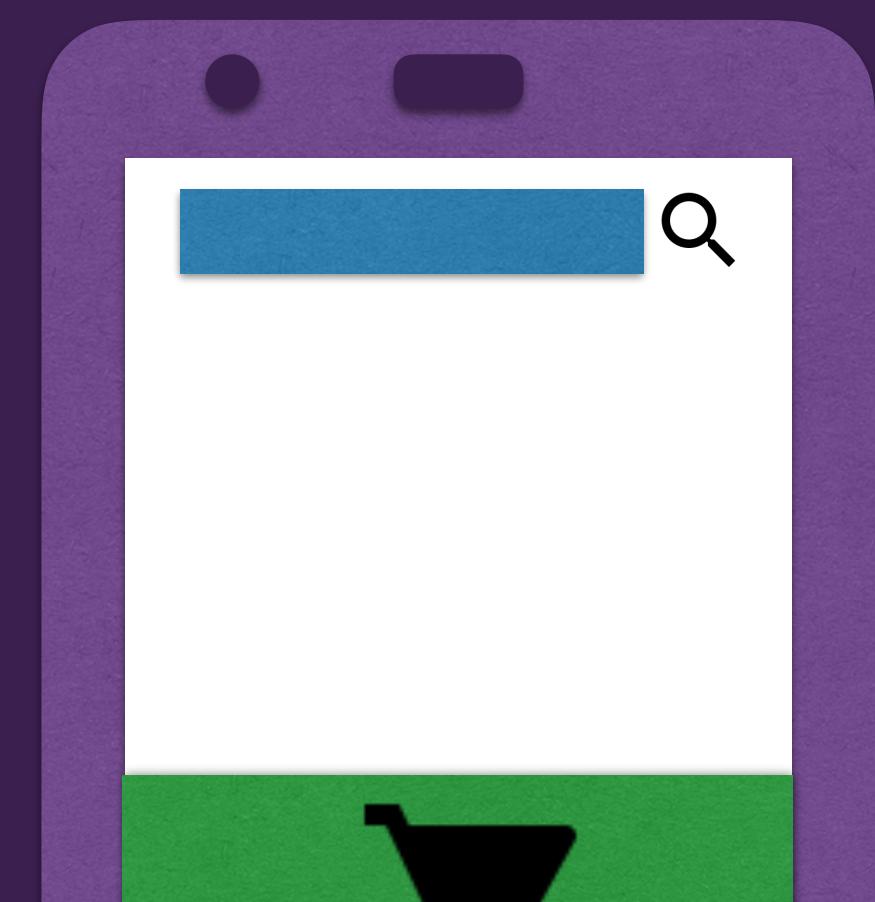


The background of the slide is a dark, textured brick wall, providing a rustic and industrial feel.

We can combine state changes  
through composition

# Showing states

```
sealed class State {  
    object Cart : State()  
    data class Showing(val state: State) : State()  
}
```



# Showing states

```
sealed class State {  
    object Cart : State()  
    data class Showing(val state: State) : State()  
}
```

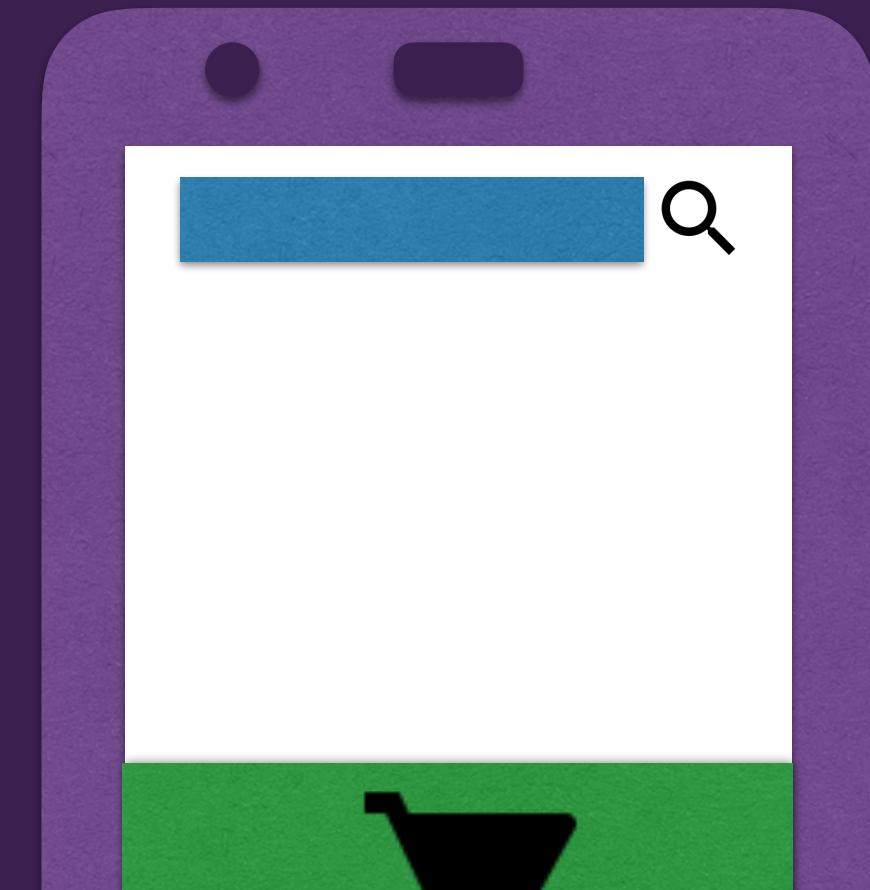
```
dispatcher.dispatch(State.Showing(Cart))
```



# Showing states

```
interface RxState {  
    fun showingNot(clazz: Class<out Screen>): Observable<Showing>  
    fun showing(clazz: Class<out Screen>): Observable<Showing>  
}
```

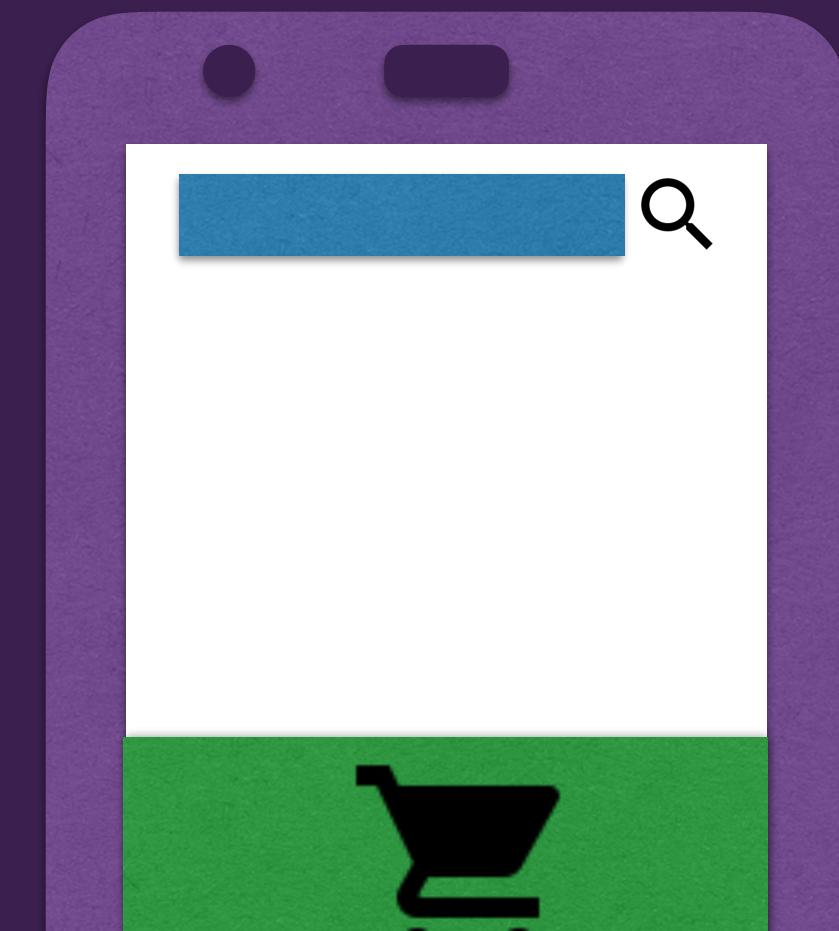
```
rxState.showing(Cart::class.java)  
    .subscribe({  
        mvpView.showCart()  
    }))
```



# We can show/hide views based on showing states

```
rxState.showing(Cart::class.java))  
    .subscribe({  
        mvpView.showCart()  
    }))
```

```
rxState.showingNot(Cart::class.java))  
    .subscribe({  
        mvpView.hide()  
    }))
```



# Dispatching showing cart

```
dispatcher.dispatch(State.Showing(Cart))
```



```
rxState.showing(Cart::class.java))  
.subscribe({  
    mvpView.showCart()  
}))
```

```
rxState.showingNot(Cart::class.java))  
.subscribe({  
    mvpView.hide()  
}))
```

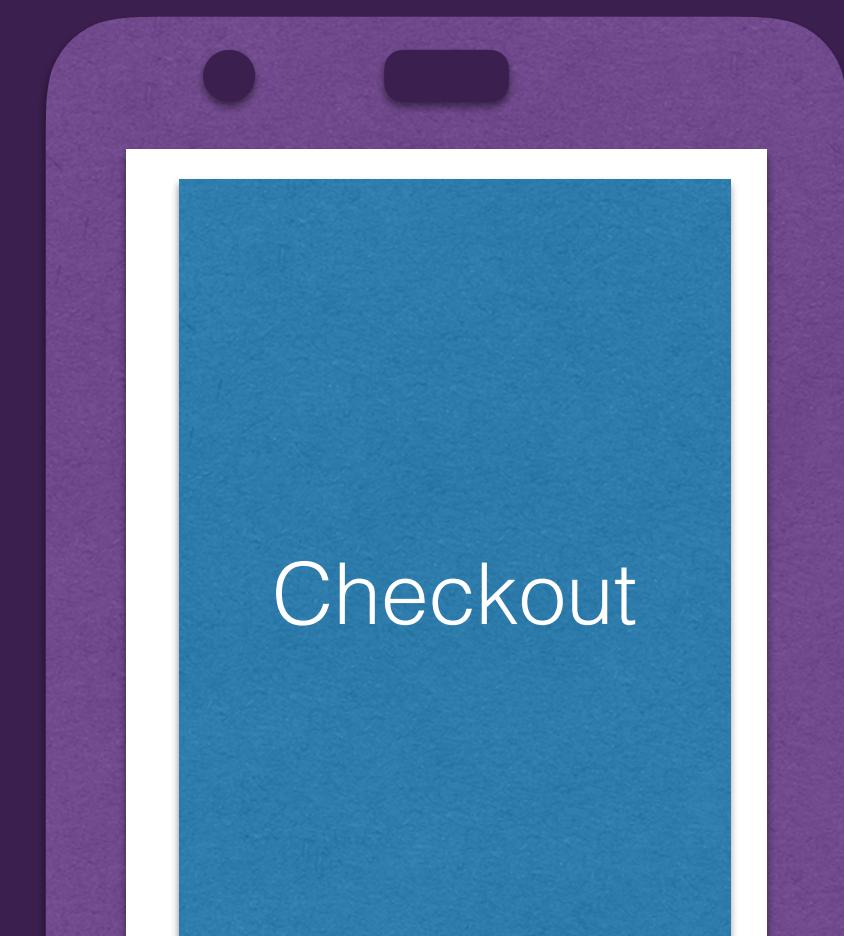


# Dispatching Showing Checkout

```
dispatcher.dispatch(State.Showing(Checkout))
```

```
rxState.showing(Cart::class.java))  
.subscribe({  
    mvpView.showCart()  
}))
```

```
rxState.showingNot(Cart::class.java))  
.subscribe({  
    mvpView.hide()  
}))
```



Showing/hiding works well  
when all views attached

Showing/hiding works well  
when all views attached

Not very scalable in real apps

# How do we deal with deep flows?

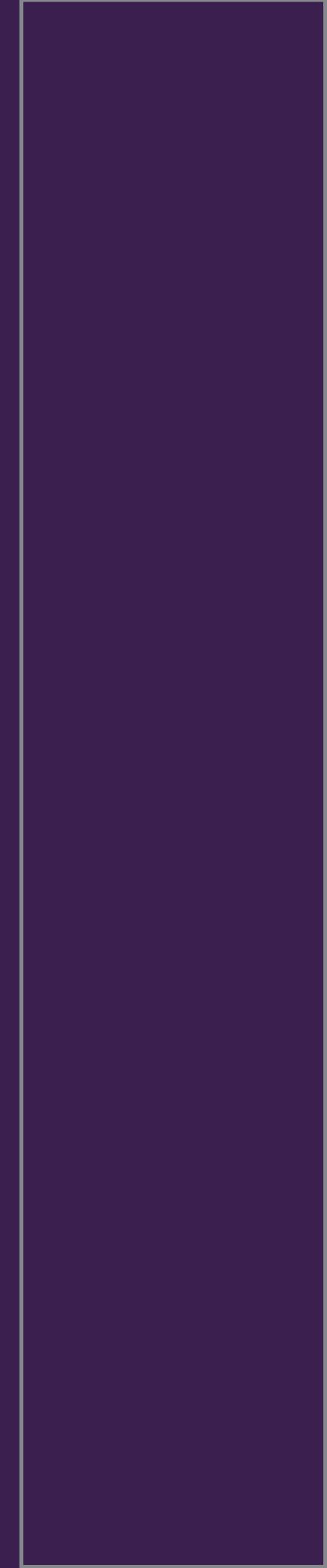
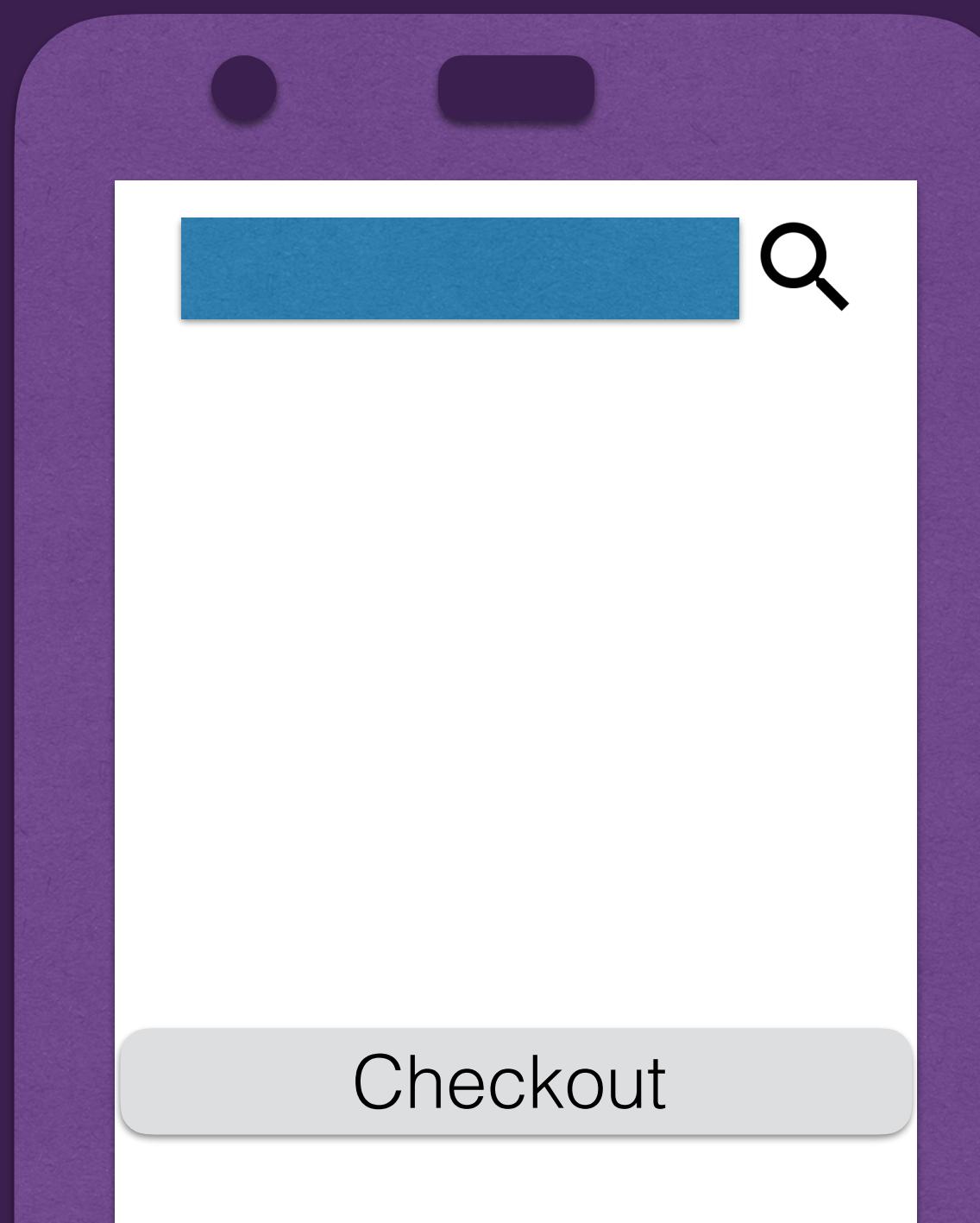
How do we deal with deep flows?

Treat screen creation/routing as a state change

# Dispatching screens workflow

CartView

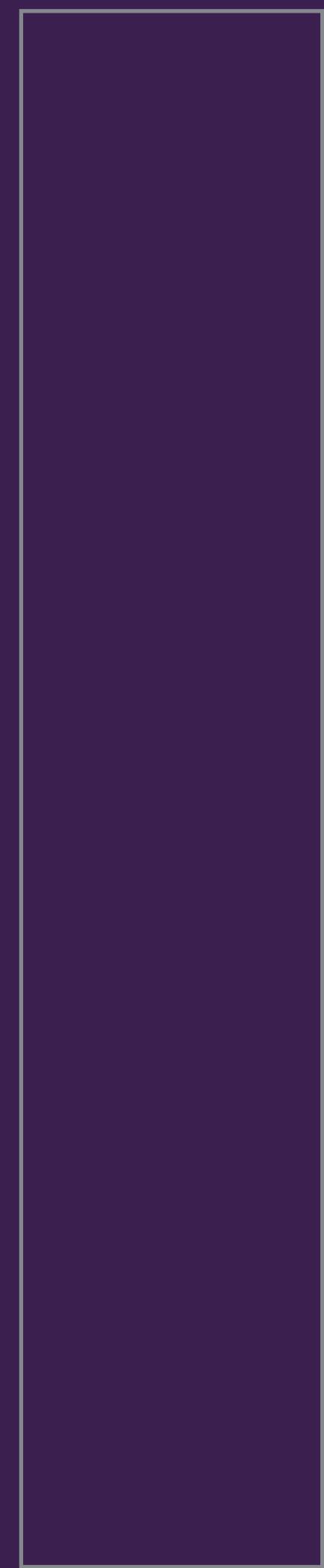
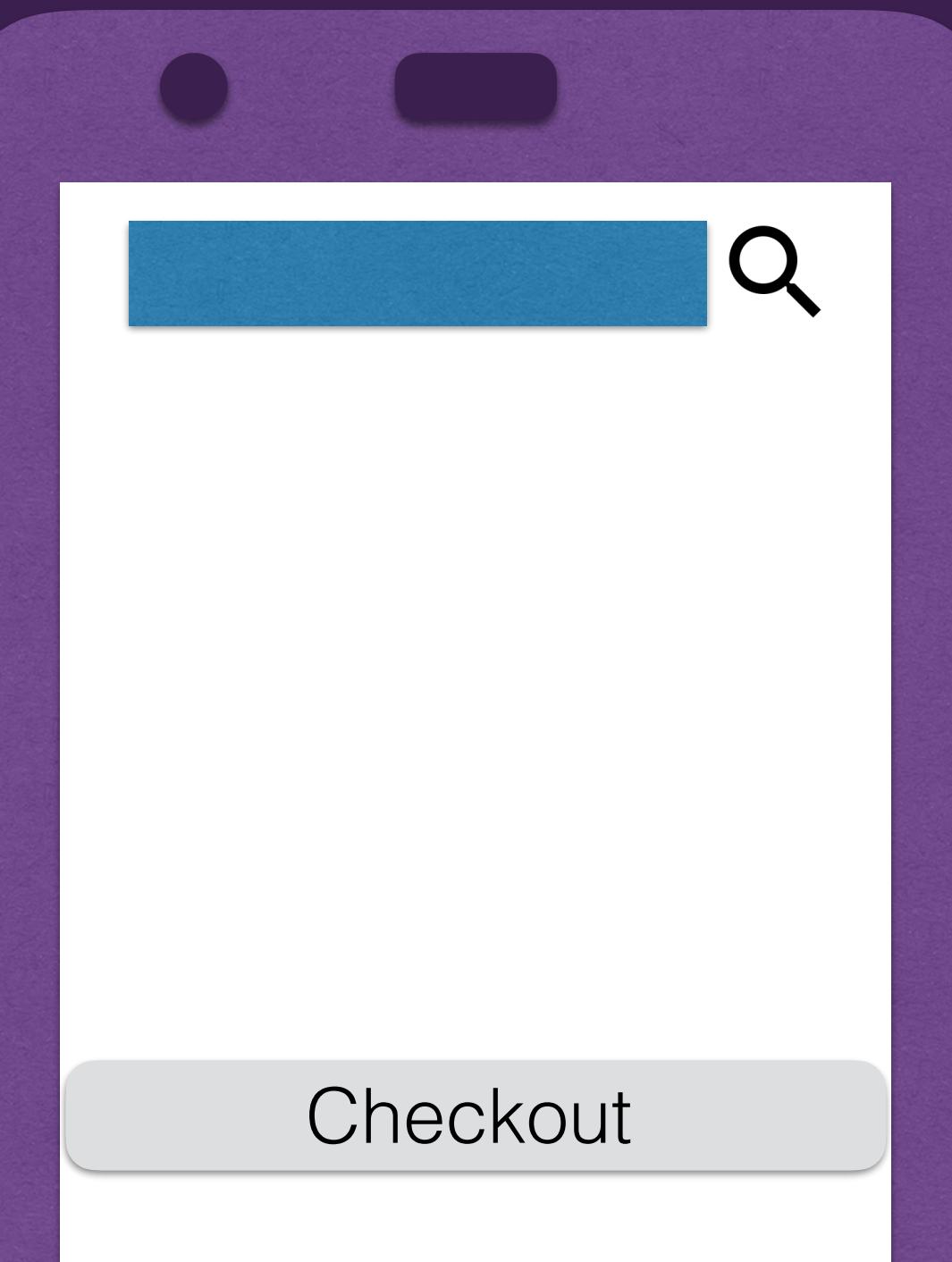
CartPresenter



CartView

Go To Checkout

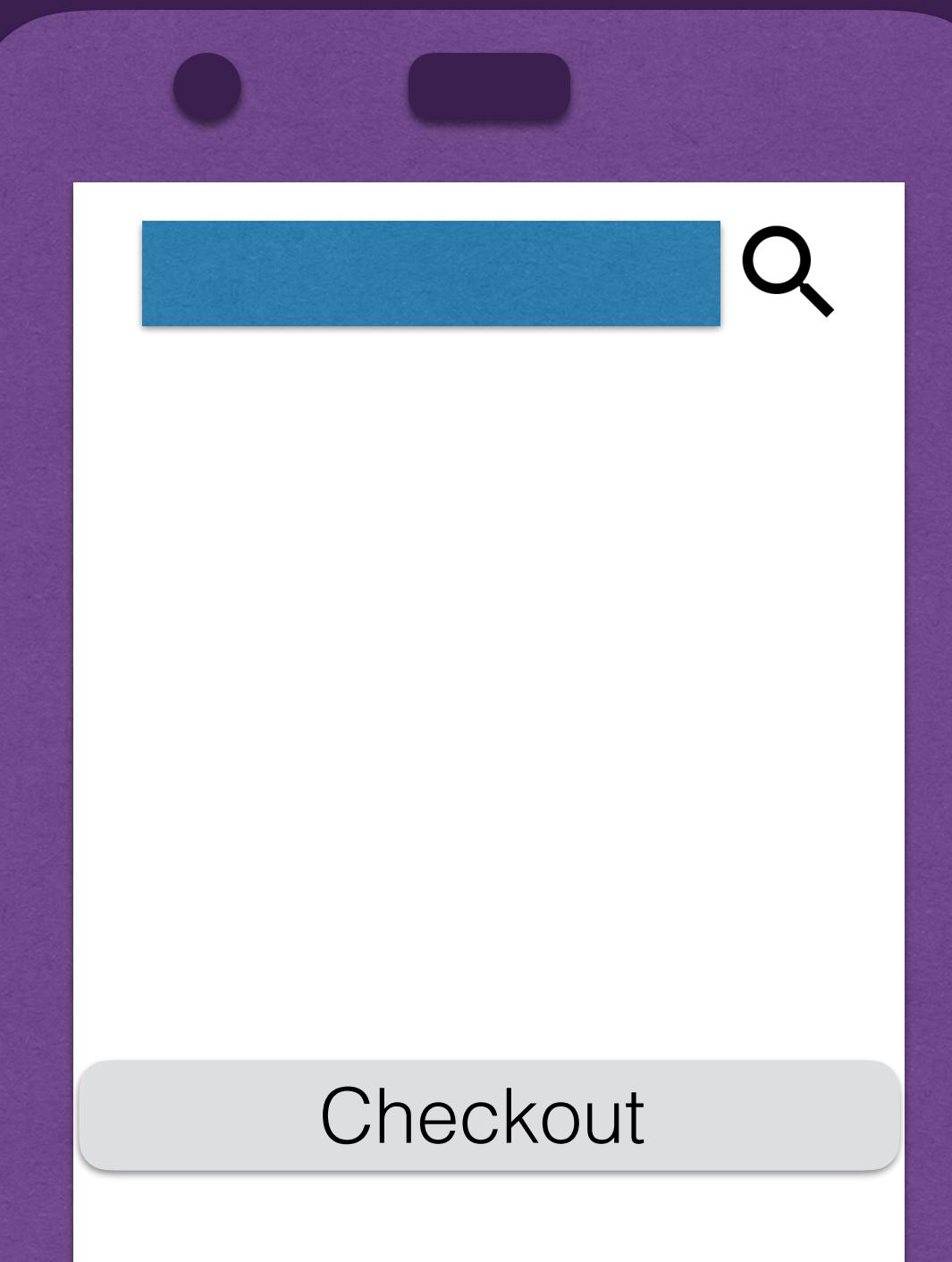
CartPresenter



CartView

CartPresenter

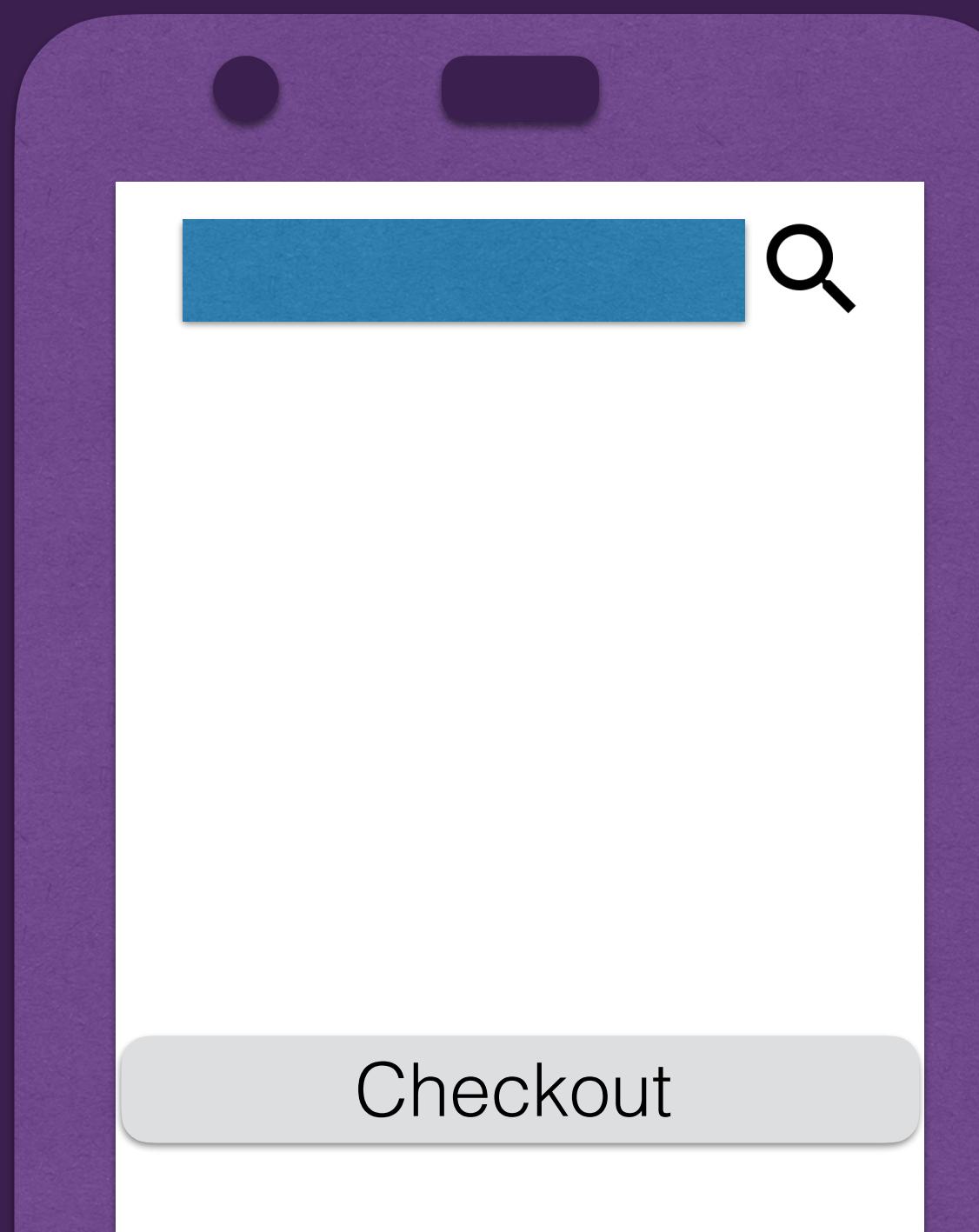
Screen.Checkout



CartView

CartPresenter

Screen.Checkout

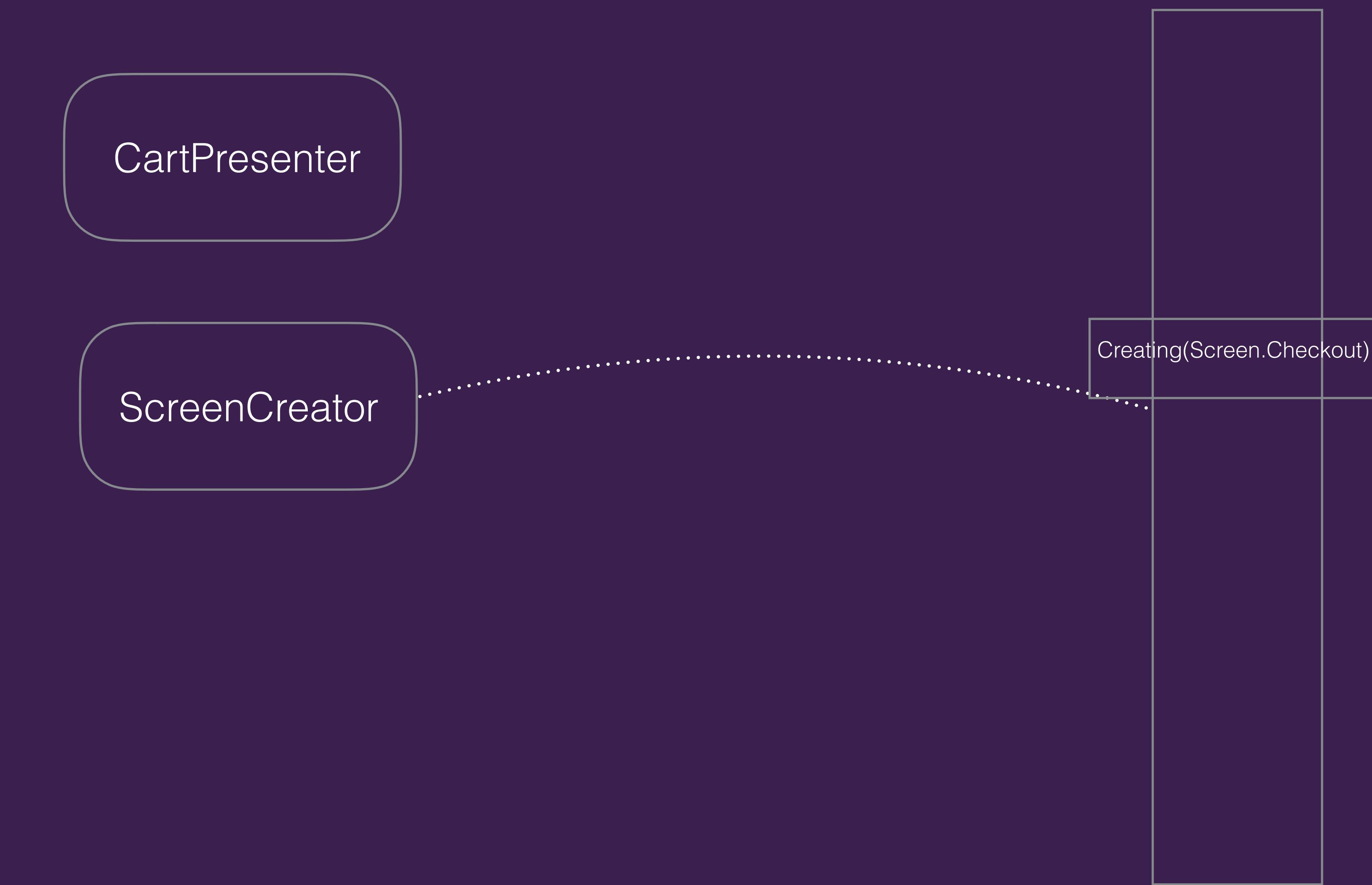
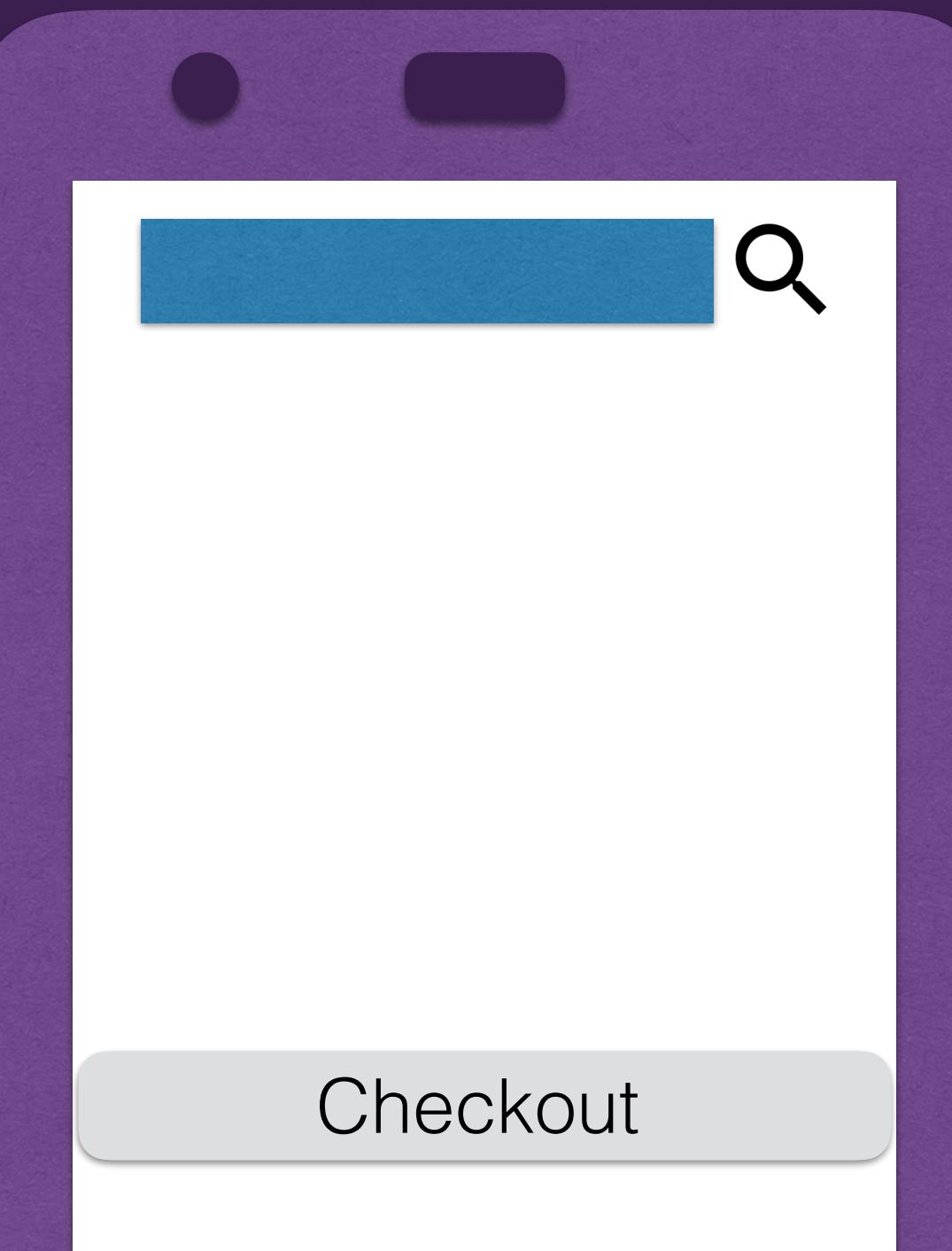


CartView

CartPresenter

ScreenCreator

Creating(Screen.Checkout)



CartView

CartPresenter

Creating(Screen.Checkout)

ScreenCreator

Checkout

CartView

CartPresenter

Screen.Checkout  
**ScreenCreator**

CheckoutPresenter

CheckoutView

Hidden CheckoutView

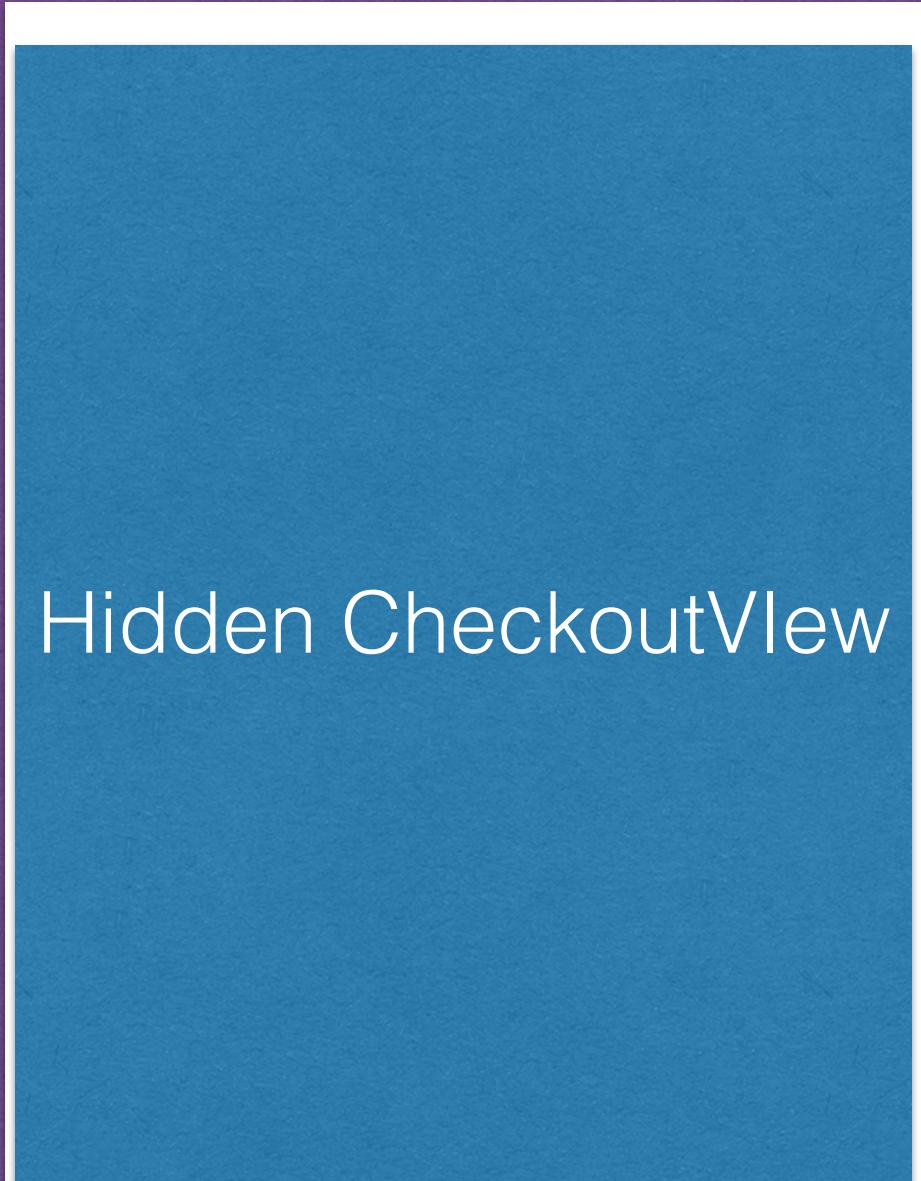
CartView

CartPresenter

ScreenCreator

CheckoutPresenter

CheckoutView



Hidden CheckoutView

Showing(Screen.Checkout)

CartView

CartPresenter

ScreenCreator

CheckoutPresenter

CheckoutView

Hidden CheckoutView

Showing(Screen.Checkout)

CartView

CartPresenter

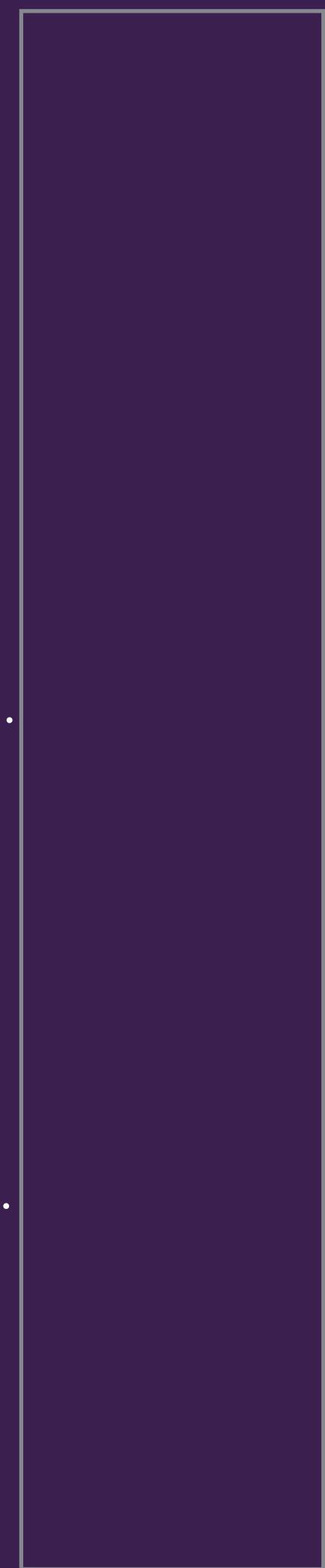
ScreenCreator

CheckoutPresenter

Showing(Screen.Checkout)

CheckoutView

Hidden CheckoutView



CartView

CartPresenter

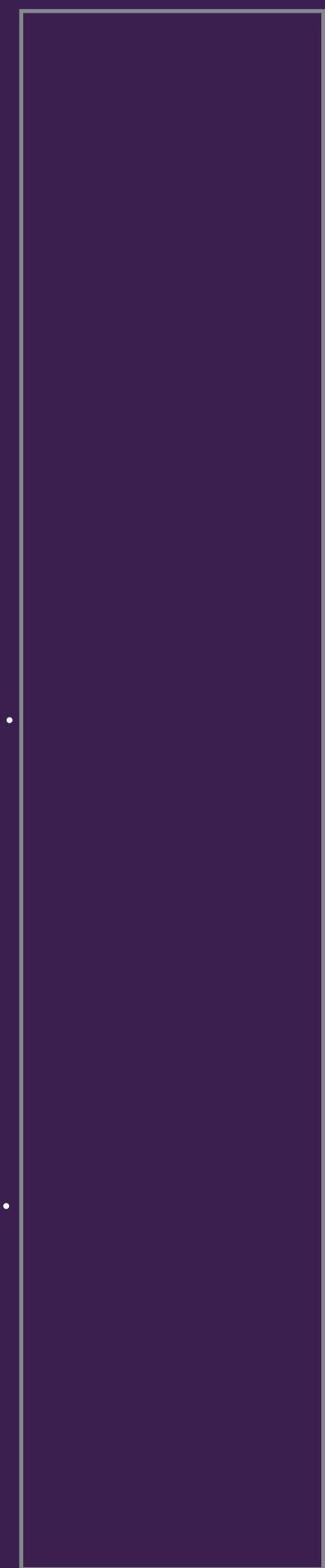
ScreenCreator

CheckoutPresenter

Showing(Screen.Checkout)

CheckoutView

CheckoutView  
With Data



View tells presenter to go to new screen

presenter.openCart()

# Presenter dispatches new screen state

```
sealed class Screen : State() {  
    data class Checkout(val payload:String) : Screen()  
}  
dispatcher.dispatch(Screen.Checkout)
```

# Dispatcher encapsulates in a creating state

```
dispatcher.dispatch(Creating(Screen.Checkout))
```

# Screen creator creates view/presenter

```
rxState.creating()  
    .subscribe({  
        createScreen(it)  
    })
```

# Dispatches a showing state

```
rxState.creating()  
    .subscribe({  
        createScreen(it)  
        dispatcher.dispatch(Showing(it.screen))  
    })
```

# Dispatcher adds each showing event to a back stack

```
override fun dispatch(state: State) {  
    when (state) {  
        is Showing->{  
            backstack.push(state)  
            rxState.push(state)  
        }  
        else -> rxState.push(state)  
    }  
}
```

# Dispatcher adds each showing event to a back stack

```
override fun dispatch(state: State) {  
    when (state) {  
        is Showing->{  
            backstack.push(state) ← val backstack: Stack<Showing> = Stack()  
            rxState.push(state)  
        }  
        else -> rxState.push(state)  
    }  
}
```

# Dispatcher pushes new state to subscribers

```
override fun dispatch(state: State) {  
    when (state) {  
        is Showing->{  
            backstack.push(state)  
            rxState.push(state)  
        }  
        else -> rxState.push(state)  
    }  
}
```

# Presenter reacts to the showing state

```
override fun attachView(mvpView: CheckoutMVPView) {  
    rxState.showing(Screen.Checkout::class.java)  
        .observeOn(AndroidSchedulers.mainThread())  
        .subscribe { mvpView.show() }  
}
```

# Screens = States

```
sealed class Screen : State() {  
    object Search : Screen()  
    object Cart:Screen()  
    Data class Checkout(val items>List<Item>) : Screen()  
    data class Payment(val items>List<Item>) : Screen()  
}
```

# Routing becomes a state change

```
sealed class Screen : State() {  
    object Search : Screen()  
    object Cart:Screen()  
  
    Data class Checkout(val items>List<Item>) : Screen()  
  
    data class Payment(val items>List<Item>) : Screen()  
}  
  
    fun goToSearch() {dispatcher.dispatch(Screen.Search)}  
    fun openCart() { dispatcher.dispatch(Screen.Cart) }  
  
    fun submitNames(firstName: String, lastName: String) {  
        userRepository.update(firstName, lastName)  
        .subscribe { dispatcher.dispatch(Screen.CheckoutPayment(cartItems)) }  
    }  
}
```

Need to go back?



# Dispatcher.goBack()

```
interface Dispatcher {  
    fun dispatch(state: State)  
    fun goBack()  
}  
  
override fun onBackPressed() {  
    dispatcher.goBack()  
}
```

# Dispatcher.goBack()

```
interface Dispatcher {  
    fun dispatch(state: State)  
    fun goBack()  
}  
  
override fun onBackPressed() {  
    dispatcher.goBack()  
}
```

Dispatcher will pop current showing state and dispatch previous one again

# State Stack = Back Stack

```
interface Dispatcher {  
    fun dispatch(state: State)  
    fun goBack()  
}  
  
override fun onBackPressed() {  
    dispatcher.goBack()  
}
```

# Back Stack = State Stack

```
interface Dispatcher {  
    fun dispatch(state: State)  
    fun goBack()  
}  
  
override fun onBackPressed() {  
    dispatcher.goBack()  
}
```



Routing becomes a state change

Every view interaction becomes a state change

Your UI becomes a representation  
of dispatched state

Which is dispatched by your UI

# TLDR: Your app becomes a cycle

TLDR: Your app becomes a cycle  
User interacts with view

TLDR: Your app becomes a cycle  
User interacts with view  
View dispatches state change

TLDR: Your app becomes a cycle

User interacts with view

View dispatches state change

View reacts to state change

TLDR: Your app becomes a cycle

User interacts with view

View dispatches state change

View reacts to state change

User interacts with view



[github.com/  
digitalbuddha/Dispatcher](https://github.com/digitalbuddha/Dispatcher)

Big thank you to Efeturi  
Money for the UI