

CMPS 312

State Management with



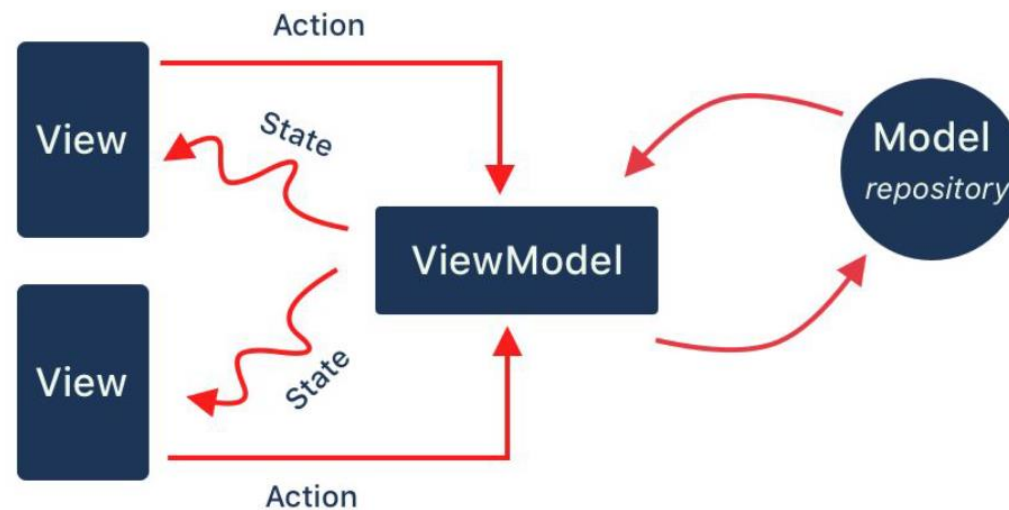
Dr. Abdelkarim Erradi

CSE@QU

Outline

1. Model-View-ViewModel (MVVM)
2. Riverpod Providers (ViewModel)

MVVM Architecture



Model-View-ViewModel (MVVM) Architecture



View = UI to display state & collect user input

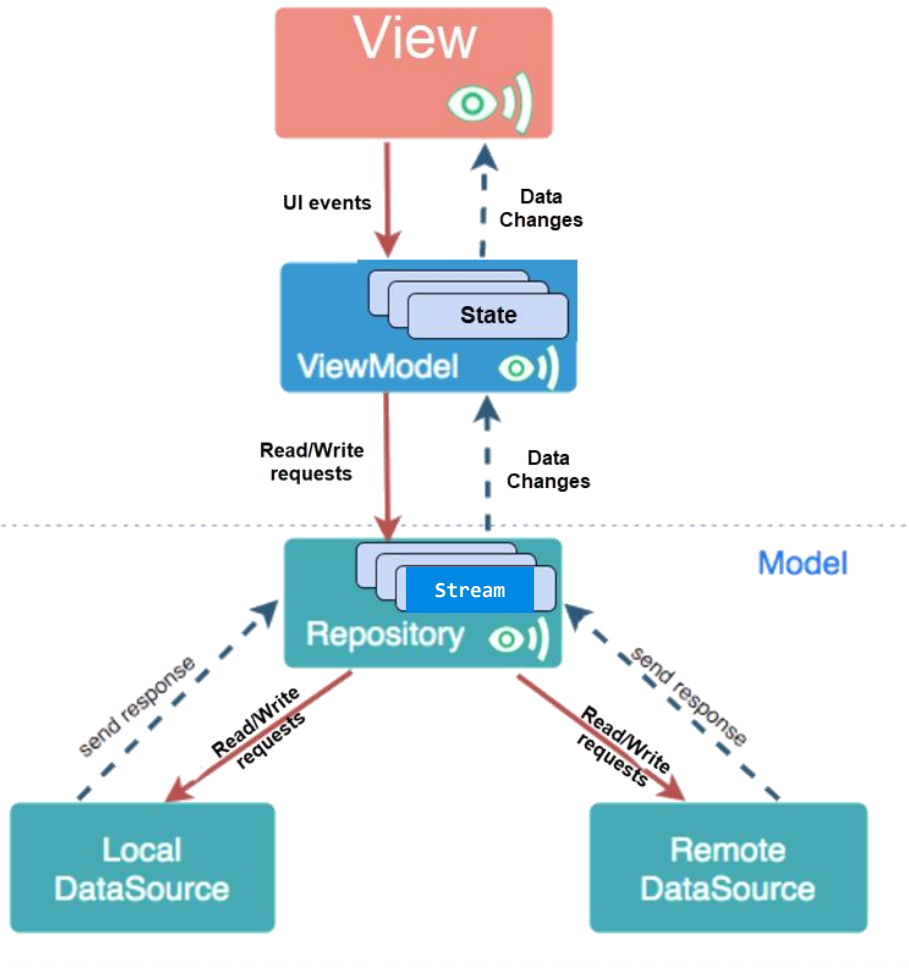
- It **observes** state changes from the ViewModel to update the UI accordingly
- Calls the ViewModel to handle events such as button clicks, form input, etc.

ViewModel

- Manages **state** (i.e., data needed by the UI)
 - Interacts with the Model to read/write data based on user input
 - Expose the state as **Observables** that the UI can subscribe-to to get data changes
- Implements UI logic / computation (e.g., Filtering or Sorting Data, Validate user input, check correct email format or check both the password and confirm password fields match)

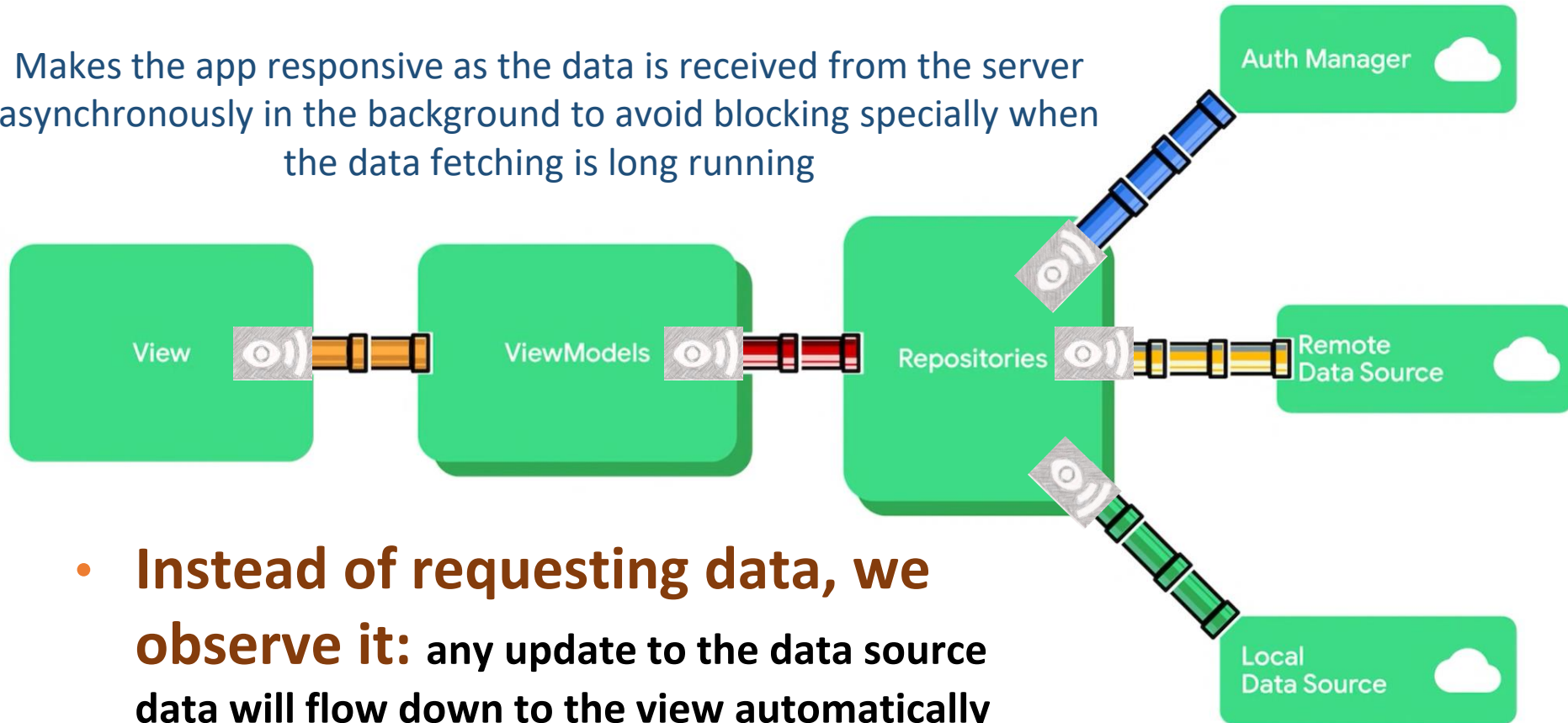
Model - handles data operations

- Model has **entities** that represent app data
- **Repositories** read/write data from either a Local Database or a Remote Web API
- Implements data-related logic / computation



Notifiers are used to **keep the View in synch** with the data sources

Makes the app responsive as the data is received from the server asynchronously in the background to avoid blocking specially when the data fetching is long running



- **Instead of requesting data, we observe it:** any update to the data source data will flow down to the view automatically
- Repo observes data changes from data sources
- ViewModel observes data changes from the Repo
- View observes data changes from the ViewModel

MVVM Key Principles

- Separation of concerns:
 - View, ViewModel, and Model are **separate components** with distinct roles
- Loose coupling:
 - ViewModel **has no direct reference to the View**
 - View never accesses the model directly
 - Model unaware of the view
- Observer pattern:
 - View observes the ViewModel (to get data changes)
 - ViewModel observes the Model (to get data changes)



Advantages of MVVM



- ***Separation of concerns*** = separate UI from app logic
 - App logic is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change
 - Allow changing a component without significantly disturbing the others (e.g., View can be completely changed without touching the model)
 - Easier **testing** of the App components

MVVM => Easily **maintainable** and **testable** app

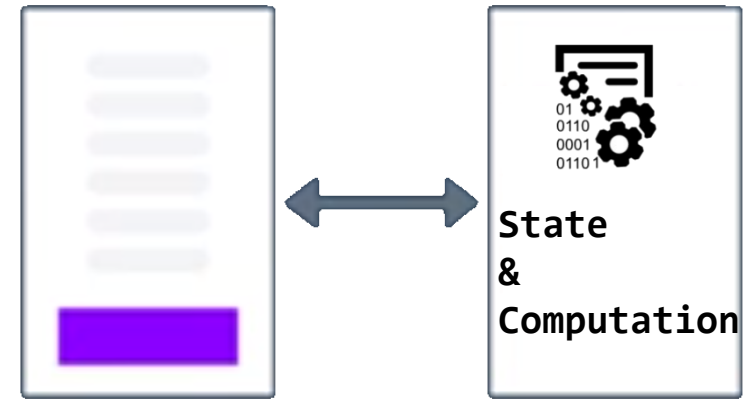
Riverpod Providers (ViewModel)

Riverpod

- Allows you to efficiently create, manage, and share states across the app
- Promotes clean code by **separating** business logic from UI, which simplifies testing and maintenance
- It provides different types of providers to manage various kinds of state
- Allows data caching

Provider

- **Provider** acts as the ViewModel is used to **provide and manage state** (i.e., data needed by the UI)
- Provider exposes **State** variables, which as **observable data holder** that the View watches to trigger UI rebuild
 - This decouples the Provider from the View: the **Provider does NOT have any direct reference to the View**
 - The View can observe the Provider State variables for changes then update the UI



View

Provider

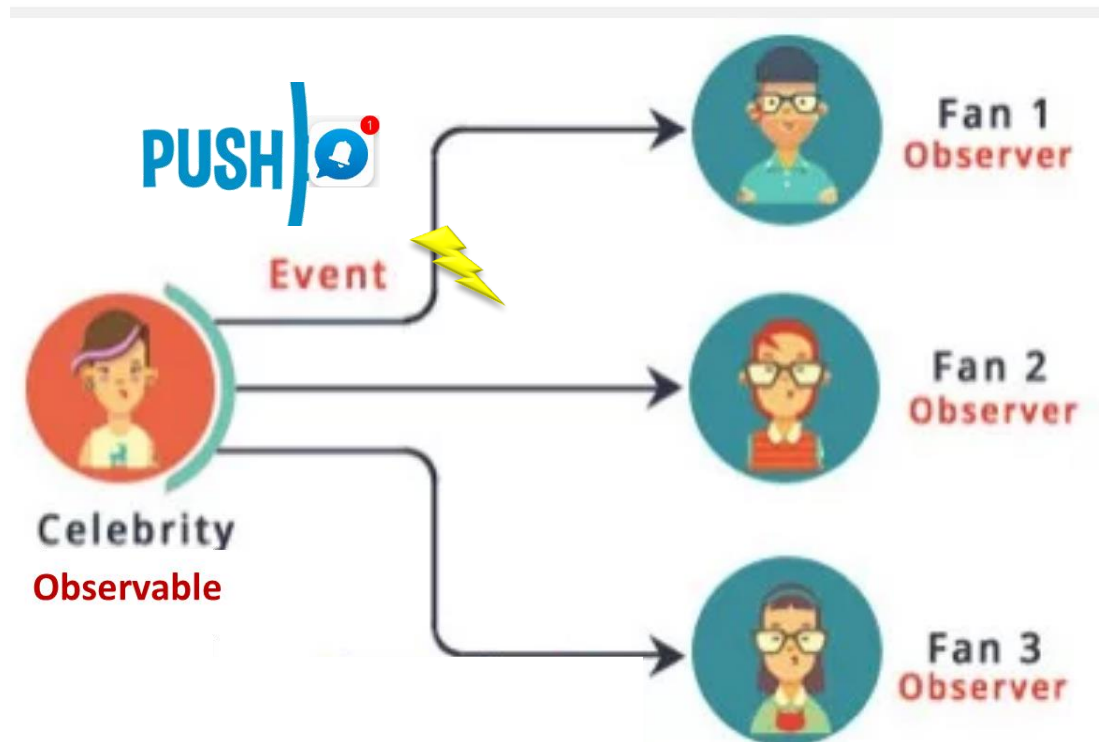


Use **Provider**:

- Manages state
- Read/write data from a Repository

Observable - Real-Life Example

- A celebrity who has many fans on Instagram
 - Fans want to get all the latest updates (photos, videos, posts etc.)
 - Here fans are **Observers** and celebrity is an **Observable**



“no reference to the View in ViewModels” rule

- ViewModel should **not be aware of the View** who is interacting with
=> It should be **decoupled** from the View



- ViewModel should not hold a reference to Widgets
- Should not have any Flutter framework related code
- As this defeats the purpose of separating the UI from the data
- Can lead to **memory leaks** and **crashes** (due to null pointer exceptions) as the ViewModel outlives the View
 - if you rotate a screen 3 times, 3 three different screen instances will be created, but you only have one ViewModel instance

Main Providers in Riverpod

Provider Type	Description	Key Features	Typical Usage
Provider	Provides immutable data that doesn't change over time	<ul style="list-style-type: none">- Used for static, read-only values- Ideal for objects created once and shared across the app- No reactivity or updates	<ul style="list-style-type: none">- Providing configuration data, constants, or dependencies that don't change- Example: API URLs, theme settings
NotifierProvider	Provides a stateful object that extends Notifier<T>	<ul style="list-style-type: none">- Exposes a notifier object that implements state management logic	<ul style="list-style-type: none">- Managing complex state logic such as shopping cart, authentication state
FutureProvider	Handles asynchronous data	<ul style="list-style-type: none">- Automatically rebuilds when the Future completes- Returns a loading/error state while waiting	<ul style="list-style-type: none">- Fetching data from an API- Asynchronous initialization like loading user data, preferences, or remote configuration at startup
StreamProvider	Handles continuous asynchronous data from a Stream	<ul style="list-style-type: none">- Provides real-time data updates- Returns a loading/error state while waiting for updates	<ul style="list-style-type: none">- Real-time data streams like a chat messages- Listening to database changes or sensor data- WebSocket connections

Providers - Example Usage Scenarios

- **Provider:** Static configurations like app themes or localization settings
- **NotifierProvider:** Managing shopping cart in eCommerce app
- **FutureProvider:** Fetching weather data
- **StreamProvider:** Listening to Firebase Firestore document updates or real-time messaging
- All providers are **initialized lazily by default**
 - The provider is only initialized when it is first used
 - Useful for providers that are only used in certain parts of the app

Provider

- Provider provides a shared **read-only value** to the parts of your app that need it
 - Used to provide a value that doesn't change, such as configuration data, API URLs, theme settings
 - It does not rebuild UI when the value changes
- Can't we just use simple static variable?
 - Provider is lazily initialized
 - Static variable is **globally** accessible vs. Provider can scope data to a specific part of the widget tree
 - Promoting better state management and avoiding unnecessary data exposure
 - Static variable remains in memory for the entire duration of the app vs. **Provider.autoDispose** automatically releases resources when the state is no longer needed (e.g., when the widget using it is removed from the widget tree), helping optimize memory usage

Provider Example

```
final apiUrlProvider = Provider.autoDispose<String>(
  (ref) => "https://api.example.com"
);
```

This Provider exposes a read-only String value, the URL
<https://api.example.com>

Vs. `static apiUrlProvider = "https://api.example.com"`

```
class ApiUrlScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // Reading the API URL using the provider
    final apiUrl = ref.read(apiUrlProvider);

    return Text('API URL: $apiUrl');
  }
}
```

The value can be accessed by any **consumer** widget that needs it using `ref.read`

Consuming Providers from the UI

- To allow widgets to read or watch providers, the root widget must be wrapped in a **ProviderScope** widget
 - The ProviderScope serves as the **container** for all providers' states, allowing widgets to read and watch those providers

```
void main() {  
  runApp(ProviderScope(child: MyApp()))  
}
```

- Widgets should extend **ConsumerWidget** instead of StatelessWidget (or **ConsumerStatefulWidget** instead of StatefulWidget) to get a **WidgetRef** object to access the declared providers
 - Use **ref.read** / **ref.watch** for reading/watching providers

```
class CounterScreen extends ConsumerWidget {  
  @override  
  Widget build(BuildContext context, WidgetRef ref) {  
    final counter = ref.watch(counterProvider);  
    ...  
  }  
}
```

How to read provider state

- **ref.watch()**: Rebuilds the widget whenever the provider's state changes, ensuring a reactive UI
- **ref.read()**: Accesses the provider's current state without listening to future updates
 - typically used in event handlers such as **onPressed**
- Best Practice: Prefer using **ref.watch()** over **ref.read()** to maintain a reactive UI whenever state changes
 - Use **ref.read()** for one-time access, like in event handlers

NotifierProvider

- **NotifierProvider** is responsible for creating and providing an instance of a **Notifier class** to the parts of your app that need it
 - Acts as the mechanism that makes the state available to the app
- **Key Features:**
 - Encapsulation: Keeps state management logic encapsulated within the Notifier class
 - The Provider allows Widgets to **listen to changes** in the state managed by the Notifier and rebuild themselves when the state changes

NotifierProvider - Notifier class

- A Notifier is a class that **holds** the mutable state and the **logic** to manipulate that state (i.e., methods to update or compute it) => the "**how**" of state changes
 - Must extend the **Notifier<T>**
 - Must override the **build** method to initialize the state
 - It encapsulates state management methods that mutate the state
 - Listeners get notified of the changes, making the UI reactive to these changes
 - E.g. CounterNotifier class holds the counter state and provides methods to increment and decrement the counter
- Consumer widgets can call public methods of the Notifier class using:

```
ref.read(yourProvider.notifier).yourMethod()
```

NotifierProvider - Example

```
class CounterNotifier extends Notifier<int> {  
  @override  
  int build() => 0;  
  void increment() => state++;  
}
```

NotifierProvider creates an instance of **CounterNotifier** to allow widgets to listen for state changes

```
final counterProvider =  
  NotifierProvider<CounterNotifier, int>(() => CounterNotifier());
```

```
class CounterScreen extends ConsumerWidget {  
  @override  
  Widget build(BuildContext context, WidgetRef ref) {  
    final counter = ref.watch(counterProvider);  
    ...  
    ElevatedButton(  
      onPressed: () =>  
        ref.read(counterProvider.notifier).increment(),  
      child: const Text('Increment'),  
    ); ...  
  }  
}
```

Consumer widget can call public methods of the Notifier class

NotifierProvider - Example

- **NotifierProvider** creates an instance of **CounterNotifier** to allow widgets to listen for state changes
- **NotifierProvider<CounterNotifier, int>** has two generic data types:
 - **CounterNotifier**: specifies the type of the Notifier class that will manage the state
 - _ The Notifier is responsible for managing how the state is updated
 - **int** : specifies the type of the state being managed by the Notifier
 - _ The state is what the UI listens to and rebuilds when it changes

FutureProvider


- FutureProvider is used to handle asynchronous operations, like fetching data from an API or database queries
 - **UI rebuilds when the future is completed:** it listens to a Future and triggers a UI rebuild once the operation completes and data is received
 - Handles the **loading**, **error**, and **data** states in a structured manner, e.g.:
 - **loading**: show a spinner until data is available
 - **error**: display error message if something fails
 - **data**: show the received data

FutureProvider Example

```
final weatherProvider = FutureProvider<String>((ref) async {  
  await Future.delayed(const Duration(seconds: 2)); // Simulate network call  
  return "Sunny"; // Data returned from API  
});
```

```
class WeatherScreen extends ConsumerWidget {  
  @override  
  Widget build(BuildContext context, WidgetRef ref) {  
    final weatherAsync = ref.watch(weatherProvider);  
  
    return Scaffold(  
      appBar: AppBar(title: const Text('Weather Forecast')),  
      body: weatherAsync.when(  
        loading: () => const CircularProgressIndicator(), // Loading state  
        error: (err, stack) => Text('Error: $err'), // Error state  
        data: (weather) => Text('Weather: $weather'), // Success state  
      ),  
    );  
  }  
}
```


StreamProvider

- StreamProvider is used to listen to asynchronous data streams
 -  It returns a stream of values produced incrementally over time, allowing for live updates (e.g., receiving updates from a database or Web API to refresh the UI)
 - It provides the latest emitted value from the stream to update widgets when new data arrives
 - Ideal for real-time data, such as stock prices, chat messages, or sensor readings
 - Handles the **loading**, **error**, and **data** states in a structured manner

StreamProvider Example

```
final stockPriceProvider = StreamProvider<double>((ref) async* {  
  // Simulate fetching stock prices from an API.  
  await Future.delayed(const Duration(seconds: 1));  
  yield 150.0; // Initial price  
  await Future.delayed(const Duration(seconds: 2));  
  yield 152.5; // New price update  
  await Future.delayed(const Duration(seconds: 2));  
  yield 151.0; // Another update  
});
```

```
class StockPriceScreen extends ConsumerWidget {  
  @override  
  Widget build(BuildContext context, WidgetRef ref) {  
    final stockPriceAsync = ref.watch(stockPriceProvider);  
  
    return Center(  
      child: stockPriceAsync.when(  
        loading: () => const CircularProgressIndicator(),  
        error: (err, stack) => Text("Error: $err"),  
        data: (price) => Text("Stock Price: \${price}"),  
      );  
    );  
  }  
}
```

Provider.autoDispose

- .autoDispose is used to automatically dispose the provider when no longer needed (i.e., when the UI is no longer listening)
 - improving performance and reducing memory usage

```
final weatherProvider =  
FutureProvider..autoDispose<String>((ref) async {  
  // Simulate network call  
  await Future.delayed(const Duration(seconds: 2));  
  return "Sunny"; // Data returned from API  
});
```

Provider destruction using ref.invalidate

- Sometimes, you may want to force the destruction of a provider using ref.invalidate

```
class MyWidget extends ConsumerWidget {  
  @override  
  Widget build(BuildContext context, WidgetRef ref) {  
    return ElevatedButton(  
      onPressed: () {  
        // On click, destroy the provider.  
        ref.invalidate(someProvider);  
      },  
      child: const Text('dispose a provider'),  
    );  
  }  
}
```

Combining Providers

- The **ref** object is accessible to all providers, allowing them to read or watch other providers
 - Use **ref.watch** to get the current state of a provider. Whenever the listened provider updates, the consumer provider will be invalidated and recomputed

```
final otherValue = ref.watch(otherProvider);
```

- e.g., a provider that listens to the user's location and fetches nearby restaurants

```
// We use "ref.watch" to get the latest location  
final location = await ref.watch(locationProvider);  
// Then get the nearby restaurants based on that location
```

Summary

- Provider-based State Management: Enables a clear separation between state and UI components
- Reactivity: Automatically rebuilds widgets when the state changes, ensuring the UI remains in sync with the data
- Provider Types: Supports various provider types (e.g., Provider, NotifierProvider, FutureProvider, StreamProvider) for different use cases and state management needs
 - NotifierProvider: Supplies an instance of a Notifier class to app Widgets that require it
 - Supports Asynchronous Data: Easily handles asynchronous operations through FutureProvider and StreamProvider

Resources

- Riverpod Documentation

https://riverpod.dev/docs/introduction/getting_started

- Riverpod complete guide

<https://resocoder.com/2022/04/22/riverpod-2-0-complete-guide-flutter-tutorial/>