Riverpod

Made by **Remi Rousselet**

Twitter: @remi_rousselet

Github: @rrousselGit

Riverpod

Table of Contents:

1. Introduction

> Riverpod

A Reactive State-Management and Dependency Injection framework.

Packages:

riverpod	
flutter_riverpod	
hooks_riverpod	

https://riverpod.dev/

2. Providers

Providers are the most important part of a `Riverpod` application.
A provider is an object that encapsulates a piece of state and allows listening to that state.

For providers to work, you must add `ProviderScope` at the root of your Flutter applications:

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

https://riverpod.dev/docs/concepts/providers

2. Providers

Why use providers? (1)

By wrapping a piece of state in a provider, this:

- Allows easily accessing that state in multiple locations.
  - Providers are a complete replacement for patterns like `Singletons`, `Service Locators`, `Dependency Injection` or `InheritedWidgets`.
- Simplifies combining this state with others.
  - Ever struggled to merge multiple objects into one?
    - This scenario is built directly inside providers, with a simple syntax.

https://riverpod.dev/docs/concepts/providers#why-use-providers

2. Providers

Why use providers? (2)

By wrapping a piece of state in a provider, this:

- Enables performance optimizations.
  - Whether for filtering widget rebuilds or for caching expensive state computations;
    - providers ensure that only what is impacted by a state change is recomputed.
- Increases the testability of your application.
  - With providers, you do not need complex setUp/tearDown steps.
  - Furthermore, any provider can be overridden to behave differently during test, which allows easily testing a very specific behavior.
- Easily integrate with advanced features, such as logging or pull-to-refresh.

https://riverpod.dev/docs/concepts/providers#why-use-providers

3. Creating a Provider (1)

Providers come in many variants, but they all work the same way.

```
final myProvider = Provider<MyValue>(
  (ref) {
    return MyValue();
  },
  name: 'myProvider', // name used in debug
);
```

we can have two providers expose a state of the same "type":

```
final cityProvider = Provider<String>((ref) => 'London');
final countryProvider = Provider<String>((ref) => 'England');
```

https://riverpod.dev/docs/concepts/providers#creating-a-provider

3. Creating a Provider (2)

Performing actions before the state destruction

```
final example = StreamProvider.autoDispose((ref) {
  final streamController = StreamController<int>();

  ref.onDispose(() {
    // Closes the StreamController when the state of this provider is destroyed.
    streamController.close();
  });

  return streamController.stream;
});
```

> Note: Depending on the provider used, it may already take care of the clean-up process. For example, `StateNotifierProvider` will call the dispose method of a `StateNotifier`.

https://riverpod.dev/docs/concepts/providers#performing-actions-before-the-state-destruction

3. Creating a Provider (3)

Creating Provider with Modifiers

```
final myAutoDisposeProvider = StateProvider.autoDispose<String>((ref) => 0);
final myFamilyProvider = Provider.family<String, int>((ref, id) => '$id');

// combine 2 modifiers (autoDispose & family)
final userProvider = FutureProvider.autoDispose.family<User, int>((ref, userId) async {
  return fetchUser(userId);
});
```

At the moment, there are two modifiers available:

- `.autoDispose`, which will make the provider automatically destroy its state when it is no-longer listened.
- `.family`, which allows creating a provider from external parameters.

https://riverpod.dev/docs/concepts/providers#provider-modifiers

3. Reading a Provider (1)

Obtaining a "ref" object

First and foremost, before reading a provider, we need to obtain a "ref" object.

This object is what allows us to interact with providers, be it from a widget or another provider.

https://riverpod.dev/docs/concepts/reading#obtaining-a-ref-object

3. Reading a Provider (2)

Obtaining a "ref" from a provider

All providers receive a "ref" as parameter:

```
final provider = Provider((ref) {
  // use ref to obtain other providers
  final repository = ref.watch(repositoryProvider);

  return SomeValue(repository);
})
```

https://riverpod.dev/docs/concepts/reading#obtaining-a-ref-from-a-provider

3. Reading a Provider (3)

(Obtaining a "ref" from a provider)

This parameter is safe to pass to the value exposed by the provider.

```
final counter = StateNotifierProvider<Counter, int>((ref) {
  return Counter(ref);
});

class Counter extends StateNotifier<int> {
  Counter(this.ref): super(0);

  final ProviderRefBase ref;

  void increment() {
    // Counter can use the "ref" to read other providers
    final repository = ref.read(repositoryProvider);
    repository.post('...');
  }
}
```

https://riverpod.dev/docs/concepts/reading#obtaining-a-ref-from-a-provider

3. Reading a Provider (4)

Obtaining a "ref" from a widget

> When a widget obtains a "ref", this "ref" should not be passed around. It should be used only by the widget that created this object.

Extending `ConsumerWidget` instead of `StatelessWidget`

```
class HomeView extends ConsumerWidget {
  const HomeView({Key? key}): super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // use ref to listen to a provider
    final counter = ref.watch(counterProvider);
    return Text('$counter');
  }
}
```

https://riverpod.dev/docs/concepts/reading#obtaining-a-ref-from-a-widget

3. Reading a Provider (5)

```dart
class HomeView extends ConsumerStatefulWidget {
  const HomeView({Key? key}): super(key: key);
  @override
  HomeViewState createState() => HomeViewState();
}

class HomeViewState extends ConsumerState<HomeView> {
  @override
  void initState() {
    super.initState();
    // "ref" can be used in all life-cycles of a StatefulWidget.
    ref.read(counterProvider);
  }
  @override
  Widget build(BuildContext context) {
    // We can also use "ref" to listen to a provider inside the build method
    final counter = ref.watch(counterProvider);
    return Text('$counter');
  }
}
```

https://riverpod.dev/docs/concepts/reading#obtaining-a-ref-from-a-widget

3. Reading a Provider (6)

A final solution for obtaining a "ref" inside widgets is to rely on `Consumer`.

```
Scaffold(
  body: Consumer(
    builder: (context, ref, child) {
      // We can also use the ref parameter to listen to providers.
      final counter = ref.watch(counterProvider);
      return Text('$counter');
    },
  ),
)
```

https://riverpod.dev/docs/concepts/reading#consumer-and-hookconsumer-widgets

3. Reading a Provider (7)

Three primary usages for "ref":

- ref.watch
- ref.listen
- ref.read

> Whenever possible, prefer using `ref.watch` over `ref.read` or `ref.listen` to implement a feature.
> By changing your implementation to rely on `ref.watch`, it becomes both reactive and declarative, which makes your application more maintainable.

https://riverpod.dev/docs/concepts/reading#using-ref-to-interact-with-providers

3. Reading a Provider (8)

Using ref.watch to observe a provider

- inside the build method of a widget
- inside the body of a provider to have the widget/provider listen to provider

```
final filterTypeProvider = StateProvider<FilterType>((ref) => FilterType.none);
final todosProvider = StateNotifierProvider<TodoList, List<Todo>>((ref) => TodoList());

final filteredTodoListProvider = Provider((ref) {
  // obtains both the filter and the list of todos
  final FilterType filter = ref.watch(filterTypeProvider).state;
  final List<Todo> todos = ref.watch(todosProvider);
  switch (filter) {
    case FilterType.completed:
      // return the completed list of todos
      return todos.where((todo) => todo.isCompleted).toList();
    ...
  }
});
```

https://riverpod.dev/docs/concepts/reading#using-refwatch-to-observe-a-provider

3. Reading a Provider (9)

Using ref.listen to react to a provider change

Similarly to `ref.watch`, it is possible to use `ref.listen` to observe a provider.

The main difference between them is that, rather than rebuilding the widget/provider if the listened provider changes, using `ref.listen` will instead call a custom function.

That can be useful to perform actions when a certain change happens, such that to show a snackbar when an error happens.

https://riverpod.dev/docs/concepts/reading#using-refwatch-to-observe-a-provider

3. Reading a Provider (10)

Using ref.listen to react to a provider change

Used inside body of a provider:

```
final counterProvider = StateNotifierProvider<Counter, int>((ref) => Counter());

final anotherProvider = Provider((ref) {
  ref.listen<int>(counterProvider, (int count) {
    print('The counter changed ${count}');
  });
  ...
});
```

https://riverpod.dev/docs/concepts/reading#using-reflisten-to-react-to-a-provider-change

3. Reading a Provider (11)

Using ref.listen to react to a provider change

Inside the build method of a widget:

```
final counterProvider = StateNotifierProvider<Counter, int>((ref) => Counter());

class HomeView extends ConsumerWidget {
  const HomeView({Key? key}): super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    ref.listen<int>(counterProvider, (int count) {
      print('The counter changed ${count}');
    });
    ...
  }
}
```

https://riverpod.dev/docs/concepts/reading#using-reflisten-to-react-to-a-provider-change

3. Reading a Provider (12)

Using ref.read to obtain the state of a provider once

The `ref.read` method is a way to obtain the state of a provider, without any extra effect.

It is commonly used inside functions triggered on user intereactions.

```
final counterProvider = StateNotifierProvider<Counter, int>((ref) => Counter());
class HomeView extends ConsumerWidget {
  ...
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Scaffold(
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          // Call `increment()` on the `Counter` class
          ref.read(counterProvider.notifier).increment();
        },
  ...
}
```

https://riverpod.dev/docs/concepts/reading/#using-refread-to-obtain-the-state-of-a-provider-once

3. Reading a Provider (13)

Notes

- The watch method should not be called asynchronously, like inside `onPressed` or a `ElevatedButton`. Not should it be used inside initState and other State life-cycles.
  In those cases, consider using `ref.read` instead.
- The listen method should not be called asynchronously, like inside `onPressed` or a `ElevatedButton`. Not should it be used inside initState and other State life-cycles.
- Using `ref.read` should be avoided as much as possible.
  It exists as a work-around for cases where using watch or listen would be otherwise too inconvenient to use.
  If you can, it is almost always better to use watch/listen, especially watch.

https://riverpod.dev/docs/concepts/reading/#using-refread-to-obtain-the-state-of-a-provider-once

3. Reading a Provider (14)

Notes

- DON'T use ref.read inside the build method or provider
- You might be tempted to use ref.read to optimize the performance of a widget by doing:

```
// BAD
final counter = ref.read(counterProvider);

// GOOD
StateController<int> counter = ref.watch(counterProvider.notifier);

return ElevatedButton(
  onPressed: () => counter.state++,
);
```

https://riverpod.dev/docs/concepts/reading/#using-refread-to-obtain-the-state-of-a-provider-once

3. Reading a Provider (15)

Notes

On the other hand, the second approach supports cases where the counter is reset.
For example, another part of the application could call:

```
ref.refresh(counterProvider);
```

which would recreate the `StateController` object.

If we used `ref.read` here, our button would still use the previous `StateController` instance, which was disposed and should no-longer be used.
Whereas using `ref.watch` correctly rebuilt the button to use the new `StateController`.

https://riverpod.dev/docs/concepts/reading/#using-refread-to-obtain-the-state-of-a-provider-once

3. Reading a Provider (16)

Deciding what to read

As example, consider the following StreamProvider:

```
final userProvider = StreamProvider<User>(...);
```

When reading this userProvider, you can:

- synchronously read the current state by listening to userProvider itself
- obtain the associated Stream, by listening to userProvider.stream
- obtain a Future that resolves with the latest value emitted, by listening to userProvider.last

https://riverpod.dev/docs/concepts/reading/#deciding-what-to-read

3. Reading a Provider (17)

Using "select" to filter rebuilds

By default, listening to a provider listens to the entire object. But in some cases, a widget/provider may only case about some properties instead of the whole object.

For example, a provider may expose a User:

```
abstract class User {
  String get name;
  int get age;
}
```

But a widget may only use the user name:

```
  User name = ref.watch(userProvider).user; // NOT GOOD
  String name = ref.watch(userProvider.select((user) => user.name)); // GOOD
```

https://riverpod.dev/docs/concepts/reading#using-select-to-filter-rebuilds

3. Reading a Provider (18)

Using "select" to filter rebuilds

It is possible to use select with ref.listen too:

```
ref.listen<String>(
  userProvider.select((user) => user.name),
  (String name) {
    print('The user name changed $name');
  ...
```

Doing so will call the listener only when the name changes.

You don't have to return a property of the object. Any value that overrides == will work. For example you could do:

```
final label = ref.watch(userProvider.select((user) => 'Mr ${user.name}'));
```

https://riverpod.dev/docs/concepts/reading#using-select-to-filter-rebuilds

4. Combining providers

As an example, consider the following provider:

```
final cityProvider = Provider((ref) => 'London');
```

We can now create another provider that will consume our cityProvider:

```
final weatherProvider = FutureProvider((ref) async {
  // We use `ref.watch` to listen to another provider, and we pass it the provider
  // that we want to consume. Here: cityProvider
  final city = ref.watch(cityProvider);

  // We can then use the result to do something based on the value of `cityProvider`.
  return fetchWeather(city: city);
});
```

That's it. We've created a provider that depends on another provider.

https://riverpod.dev/docs/concepts/combining_providers#combining-providers

4. Combining providers

FAQ:

- What if the value listened changes over time?
- Can I read a provider without listening to it?
- How to test an object that receives read as parameter of its constructor?
- My provider updates too often, what can I do?

https://riverpod.dev/docs/concepts/combining_providers#combining-providers

5. ProviderObserver

Listens to the changes of a ProviderContainer.
Has four methods:

- didAddProvider
- didDisposeProvider
- didUpdateProvider
- mayHaveChanged (`Deprecated` , will be removed)

Usage:

- Use to log the info of a provider and its state.

https://riverpod.dev/docs/concepts/provider_observer

6. Modifiers (1)

At the moment, there are two modifiers available:

- `.autoDispose`, which will make the provider automatically destroy its state when it is no-longer listened.
- `.family`, which allows creating a provider from external parameters.

Example:

```
final myAutoDisposeProvider = StateProvider.autoDispose<String>((ref) => 0);
final myFamilyProvider = Provider.family<String, int>((ref, id) => '$id');

// combine 2 modifiers (autoDispose & family)
final userProvider = FutureProvider.autoDispose.family<User, int>((ref, userId) async {
  return fetchUser(userId);
});
```

https://riverpod.dev/docs/concepts/providers#provider-modifiers

6. Modifiers (2)

`.family`

The `.family` modifier has one purpose:

> Creating a provider from external values.

Some common use-cases for family would be:

- Combining FutureProvider with .family to fetch a Message from its ID
- Passing the current Locale to a provider, so that we can handle translations:
- Connecting a provider with another provider without having access to its variable.

https://riverpod.dev/docs/concepts/modifiers/family

6. Modifiers (3)

`.family`

Example:

```
final messagesFamily = FutureProvider.family<Message, String>((ref, id) async {
  return dio.get('http://my_api.dev/messages/$id');
});

...

Widget build(BuildContext context, WidgetRef ref) {
  final response = ref.watch(messagesFamily('id1'));
  // final response2 = ref.watch(messagesFamily('id2'));
}
```

https://riverpod.dev/docs/concepts/modifiers/family#usage

6. Modifiers (4)

`.family`

Parameter restrictions:

> For families to work correctly, it is critical for the parameter passed to a provider to have a consistent hashCode and ==.

Parameter should be:

- A primitive (bool/int/double/String), a constant (providers), or an immutable object that overrides == and hashCode.
- No support for multiple values/parameters

https://riverpod.dev/docs/concepts/modifiers/family#usage

6. Modifiers (5)

`.autoDispose`

A common use-case when using providers is to want to destroy the state of a provider when it is no-longer used.

There are multiple reasons for doing such, such as:

- When using Firebase, to close the connection and avoid unnecessary cost
- To reset the state when the user leaves a screen and re-enters it.

Providers comes with a built-in support for such use-case, through the `.autoDispose` modifier.

https://riverpod.dev/docs/concepts/modifiers/auto_dispose

Thank you

https://riverpod.dev/docs/concepts/modifiers/auto_dispose