Muze Innovation Co., Ltd.

# Flutter Riverpod

ณ ห้องประชุมของบริษัท มิวซ์ อินโนเวชั่น จำกัด
อาคารเอไอเอ แคปปิตอล เซ็นเตอร์ ชั้นที่ 25 ห้องเลขที่ 2501-2503
89 ถนนรัชดาภิเษก แขวงดินแดง เขตดินแดง กรุงเทพฯ 10400

## agenda

- **introduction**
- **flutter provider**
    - who is remi rousselet?
    - changeNotifierProvider
    - multiProvider
    - proxyProvider
- **flutter riverpod**
    - provider vs riverpod
    - make your first provider with network request
    - performing side effects
    - passing arguments to your requests
    - websockets
    - combining requests
    - clearing cache
    - eager initialization of providers
    - logging and error reporting

## agenda

- **bonus**
  - types of providers
  - about code generation

## (Mo) Atikom

Experienced Mobile Developer with a demonstrated history of working in the banking and commercial industry. Skilled in Flutter, SwiftUI, and Objective-C. Strong engineering professional with a Bachelor of Science - BS focused in Computer Science from Burapha University.

## Project Example

https://github.com/atikomtancharoen/riverpod-training

1

## Provider

The provider package is a popular third-party library available on the Dart packages repository (pub.dev). It implements the provider design pattern, offering a structured and convenient approach to managing application state and data.

## Who is Remi Rousselet?

Rémi is an active member of the Flutter community since its early days. He is the author of various popular packages such as Provider/Riverpod, flutter_hooks or Freezed. At Invertase, he is working on open-source projects to simplify the life of other developers.

Rémi Rousselet

Provider: https://github.com/rrousselGit/provider

Riverpod: https://github.com/rrousselGit/riverpod

Freezed: https://github.com/rrousselGit/freezed

etc.

## Provider

A wrapper around InheritedWidget to make them easier to use and more reusable.

**By using provider instead of manually writing Inherited Widget, you get:**

- simplified allocation/disposal of resources
- lazy-loading
- a vastly reduced boilerplate over making a new class every time
- devtool friendly – using Provider, the state of your application will be visible in the Flutter devtool
- a common way to consume these InheritedWidgets
- increased scalability for classes with a listening mechanism that grows exponentially in complexity (such as ChangeNotifier, which is O(N) for dispatching notifications)

## ChangeNotifierProvider

Listens to a ChangeNotifier, expose it to its descendants and rebuilds dependents whenever ChangeNotifier.notifyListeners is called.

```
ChangeNotifierProvider(
  create: (_) => new MyChangeNotifier(),
  child: ...
)
```

## MultiProvider

A provider that merges multiple providers into a single linear widget tree.

```
MultiProvider(
  providers: [
    Provider<Something>(create: (_) => Something()),
    Provider<SomethingElse>(create: (_) => SomethingElse()),
    Provider<AnotherThing>(create: (_) => AnotherThing()),
  ],
  child: someWidget,
)
```

## ProxyProvider

ProxyProvider is a provider that combines multiple values from other providers into a new object and sends the result to Provider.

```
Widget build(BuildContext context) {
  return MultiProvider(
    providers: [
      ChangeNotifierProvider(create: (_) => Counter()),
      ProxyProvider<Counter, Translations>(
        update: (_, counter, __) => Translations(counter.value),
      ),
    ],
    child: Foo(),
  );
}

class Translations {
  const Translations(this._value);

  final int _value;

  String get title => 'You clicked $_value times';
}
```

2

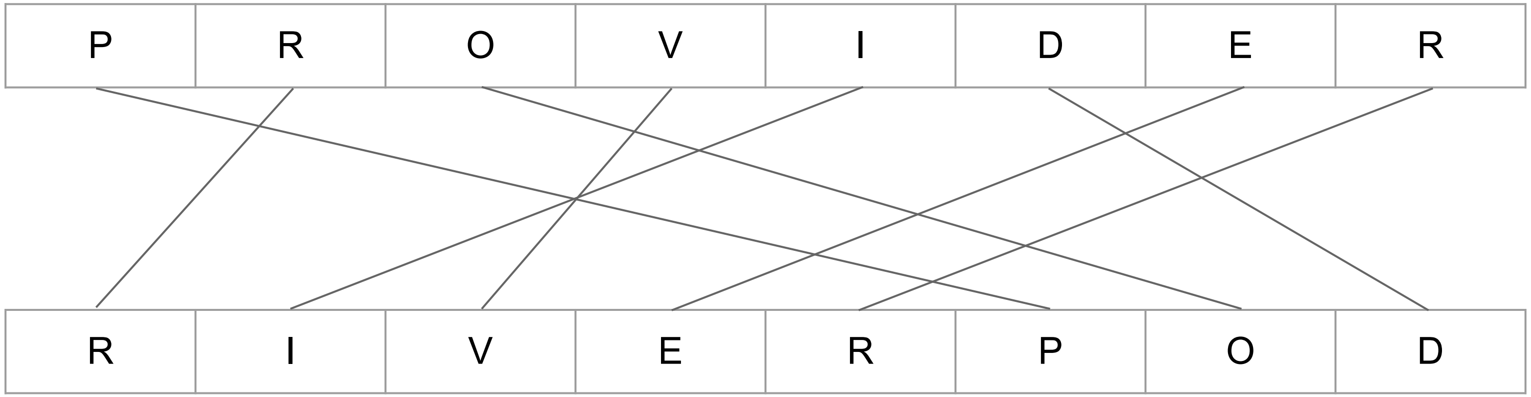
## Riverpod

Riverpod (anagram of Provider) is a reactive caching framework for Flutter/Dart.

Using declarative and reactive programming, Riverpod takes care of a large part of your application's logic for you. It can perform network-requests with built-in error handling and caching, while automatically re-fetching data when necessary.

## Anagram

An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.



## Motivation

the data is often fetched asynchronously from a server.

**The problem is, working with asynchronous code is hard. Although Flutter comes with some way to create state variables and refresh the UI on change, it is still fairly limited. A number of challenges remain unsolved:**

- Asynchronous requests need to be cached locally, as it would be unreasonable to re-execute them whenever the UI updates.
- Since we have a cache, our cache could get out of date if we're not careful.
- We also need to handle errors and loading states

2.

## Provider vs Riverpod

Recaps the differences and the similarities between Provider and Riverpod.

## Defining providers

The primary difference between both packages is how "providers" are defined.

- Provider, providers are widgets and as such placed inside the widget tree
- Provider, one way of reading providers is to use a Widget's BuildContext.

- Riverpod, providers are not widgets. Instead they are plain Dart objects.
- Riverpod's snippet extends ConsumerWidget instead of StatelessWidget.
- Instead of BuildContext.watch, in Riverpod we do WidgetRef.watch, using the WidgetRef which we obtained from ConsumerWidget.
- Riverpod does not rely on generic types. Instead it relies on the variable created using provider definition.

## **Defining providers** (Provider)

With Provider, providers are widgets and as such placed inside the widget tree, typically inside a MultiProvider:

```
class Counter extends ChangeNotifier { ... }

void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider<Counter>(create: (context) => Counter()),
      ],
      child: MyApp(),
    )
  );
}
```

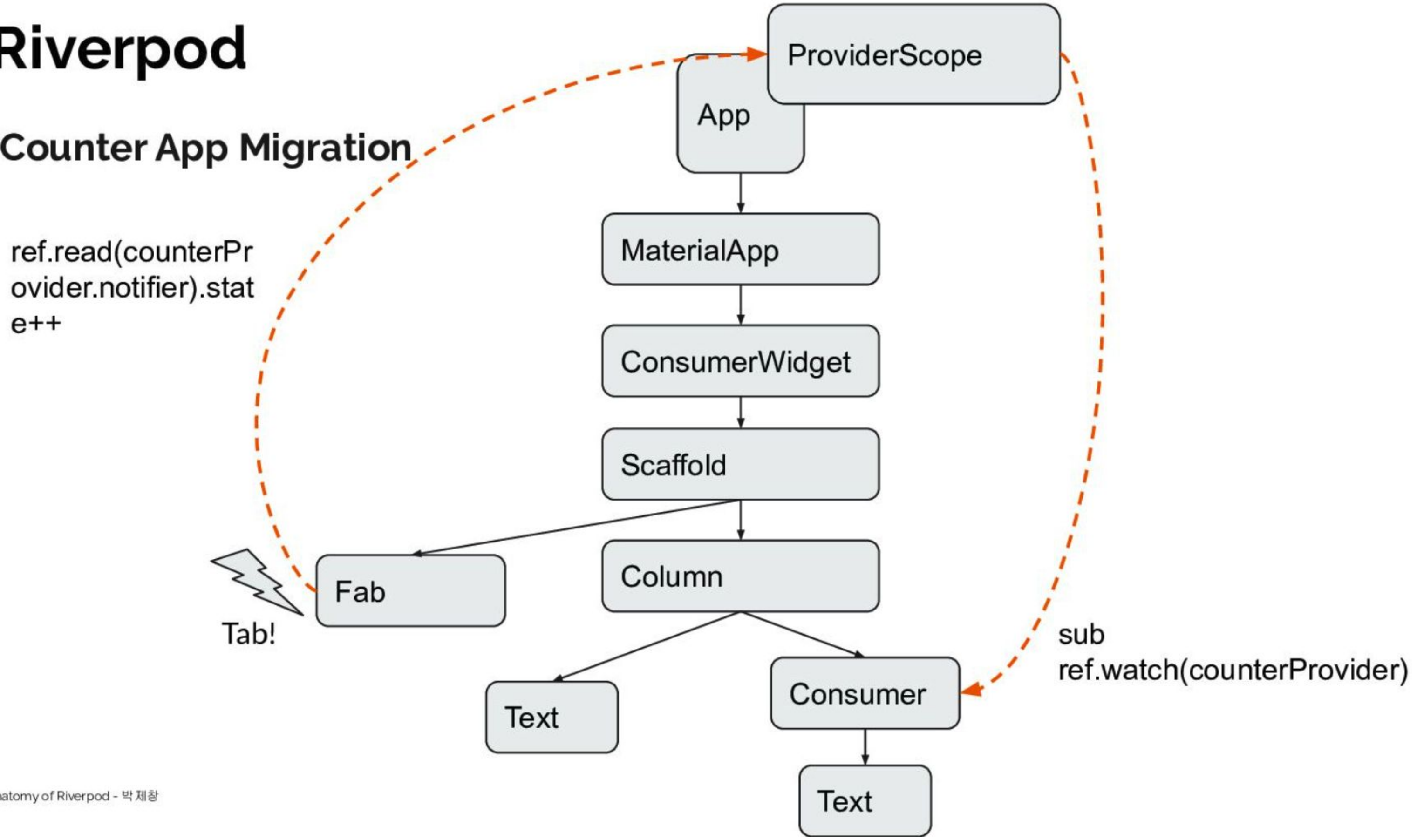
## Defining providers (Riverpod)

For Riverpod, it is necessary to add a ProviderScope widget above the entire application.

```
// Providers are now top-level variables
final counterProvider = ChangeNotifierProvider<Counter>((ref) => Counter());

void main() {
  runApp(
    // This widget enables Riverpod for the entire project
    ProviderScope(
      child: MyApp(),
    ),
  );
}
```

## Riverpod

### Counter App Migration



Anatomy of Riverpod - 박제창

https://speakerdeck.com/itsmedreamwalker/anatomy-of-riverpod-bagjecang-at-modupop

## **Reading providers:** BuildContext (Provider)

With Provider, one way of reading providers is to use a Widget's BuildContext.

The equivalent in Riverpod would be:

```
Provider<Model>(...);
```

then reading it using Provider is done with:

```
class Example extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    Model model = context.watch<Model>();
    ...
  }
}
```

## **Reading providers:** BuildContext (Riverpod)

The equivalent in Riverpod would be:

```
final modelProvider = Provider<Model>(...);

class Example extends ConsumerWidget {
  @override Widget build(BuildContext context, WidgetRef ref) {
    Model model = ref.watch(modelProvider);
    ...
  }
}
```

## **Reading providers:** BuildContext (Provider & Riverpod)

Riverpod uses the same terminology as Provider for reading providers.

- BuildContext.watch -> WidgetRef.watch
- BuildContext.read -> WidgetRef.read
- BuildContext.select -> WidgetRef.watch(myProvider.select)

**The rules for context.watch vs context.read applies to Riverpod too:**

Inside the build method, use "watch". Inside click handlers and other events, use "read". When in need of filtering out values and rebuilds, use "select".

## **Consumer** (Provider)

helpful as a performance optimization, by allowing more granular rebuilds of the widget tree

A provider was defined as:

```
Provider<Model>(...);
```

Provider allows reading that provider using Consumer with:

```
Consumer<Model>(
  builder: (BuildContext context, Model model, Widget? child) {
    ...
  }
)
```

## **Consumer** (Riverpod)

Riverpod, too, has a widget named Consumer for the exact same purpose.

we defined a provider as:

```
final modelProvider = Provider<Model>(...);
```

Provider allows reading that provider using Consumer with:

```
Consumer(
  builder: (BuildContext context, WidgetRef ref, Widget? child) {
    Model model = ref.watch(modelProvider);
    ...
  }
)
```

## Combining providers: ProxyProvider

## **Combining providers:** ProxyProvider

When using Provider, the official way of combining providers is using the ProxyProvider widget (or variants such as ProxyProvider2).

## **Combining providers:** ProxyProvider with stateless objects (Provider)

When using Provider, the official way of combining providers is using the ProxyProvider widget

For example we may define:

```
class UserIdNotifier extends ChangeNotifier {
  String? userId;
}

// ...

ChangeNotifierProvider<UserIdNotifier>(create: (context) => UserIdNotifier()),
```

From there we have two options. We may combine UserIdNotifier to create a new "stateless" provider. Such as:

```
ProxyProvider<UserIdNotifier, String>(
  update: (context, userIdNotifier, _) {
    return 'The user ID of the the user is ${userIdNotifier.userId}';
  }
)
```

## **Combining providers:** ProxyProvider with stateless objects (Riverpod)

We can do something similar in Riverpod, but the syntax is different.

In Riverpod, the definition of our UserIdNotifier would be:

```
class UserIdNotifier extends ChangeNotifier {
  String? userId;
}

// ...

final userIdNotifierProvider = ChangeNotifierProvider<UserIdNotifier>(
  (ref) => UserIdNotifier(),
);
```

From there, to generate our String based on the userId, we could do:

```
final labelProvider = Provider<String>((ref) {
  UserIdNotifier userIdNotifier = ref.watch(userIdNotifierProvider);
  return 'The user ID of the the user is ${userIdNotifier.userId}';
});
```

## **Combining providers:** ProxyProvider with stateful objects (Provider)

When combining providers, another alternative use-case is to expose stateful objects, such as a ChangeNotifier instance.

We can define a new ChangeNotifier that is based on UserIdNotifier.userId. For example we could do:

```
class UserNotifier extends ChangeNotifier {
  String? _userId;
  void setUserId(String? userId) {
    if (userId != _userId) {
      print('The user ID changed from $_userId to $userId');
      _userId = userId;
    }
  }
}

// ...

ChangeNotifierProxyProvider<UserIdNotifier, UserNotifier>(
  create: (context) => UserNotifier(),
  update: (context, userIdNotifier, userNotifier) {
    return userNotifier!
      ..setUserId(userIdNotifier.userId);
  },
);
```

## **Combining providers:** ProxyProvider with stateful objects (Riverpod)

First, in Riverpod, the definition of our UserIdNotifier would be:

```
class UserIdNotifier extends ChangeNotifier {
  String? userId;
}

// ...

final userIdNotifierProvider = ChangeNotifierProvider<UserIdNotifier>(
  (ref) => UserIdNotifier(),
);
```

## **Combining providers:** ProxyProvider with stateful objects (Riverpod)

First, in Riverpod, the definition of our UserIdNotifier would be:

```
class UserNotifier extends ChangeNotifier {
  String? _userId;
  void setUserId(String? userId) {
    if (userId != _userId) {
      print('The user ID changed from $_userId to $userId');
      _userId = userId;
    }
  }
}

// ...

final userNotifierProvider = ChangeNotifierProvider<UserNotifier>((ref) {
  final userNotifier = UserNotifier();
  ref.listen<UserIdNotifier>( userIdNotifierProvider, (previous, next) {
      if (previous?.userId != next.userId) {
        userNotifier.setUserId(next.userId);
      }
    },
  );
  return userNotifier;
});
```

## Riverpod and Provider can coexist

Keep in mind that it is entirely possible to use both Provider and Riverpod at the same time.

## Provider Documentation

See the API documentation:

provider

riverpod

2.2

## Make your first provider with network request

## Network requests are the core of any application. But there are a lot of things to consider when making a network request:

- The UI should render a loading state while the request is being made
- Errors should be gracefully handled
- The request should be cached if possible

## Performing your network request in a "provider"

Performing a network request is usually what we call "business logic". In Riverpod, business logic is placed inside "providers". A provider is a super-powered function. They behave like normal functions, with the added benefits of:

- being cached
- offering default error/loading handling
- being listenable
- automatically re-executing when some data changes

## Creating the provider

The syntax for defining a provider is as followed:

```
final name = SomeProvider.someModifier<Result>((ref) {
  <your logic here>
});
```

| | |
|---|---|
| The provider variable | This variable is what will be used to interact with our provider.<br>The variable must be final and "top-level" (global). |
| The provider type | Generally either Provider, FutureProvider or StreamProvider.<br>The type of provider used depends on the return value of your function. For example, to create a Future<Activity>, you'll want a FutureProvider<Activity>.<br><br>FutureProvider is the one you'll want to use the most.<br><br>**TIP:** Don't think in terms of "Which provider should I pick". Instead, think in terms of "What do I want to return". The provider type will follow naturally. |

## Creating the provider(2)

```
final name = SomeProvider.someModifier<Result>((ref) {
  <your logic here>
});
```

| | |
|---|---|
| Modifiers (optional) | Often, after the type of the provider you may see a "modifier".<br>Modifiers are optional, and are used to tweak the behavior of the provider in a way that is type-safe.<br><br>There are currently two modifiers available:<br>● autoDispose, which will automatically clear the cache when the provider stops being used.<br>● family, which enables passing arguments to your provider. |
| Ref | An object used to interact with other providers.<br>All providers have one; either as parameter of the provider function, or as a property of a Notifier. |
| The provider function | This is where we place the logic of our providers. This function will be called when the provider is first read.<br>Subsequent reads will not call the function again, but instead return the cached value. |

## 2.3 Performing side effects

Applications often implement a CRUD (Create, Read, Update, Delete) API. When doing so, it is common that an update request (typically a POST) should also update the local cache to have the UI reflect the new state.

To do that, we will use a new concept: Notifiers.

## Defining a Notifier

Notifiers are the "stateful widget" of providers.Notifiers are the "stateful widget" of providers. This new syntax is as follows:

```
final name = SomeNotifierProvider.someModifier<MyNotifier, Result>(MyNotifier.new);

class MyNotifier extends SomeNotifier<Result> {
  @override
  Result build() {
    <your logic here>
  }
  <your methods here>
}
```

## Defining a Notifier (2)

```
final name = SomeNotifierProvider.someModifier<MyNotifier, Result>(MyNotifier.new);
```

| | |
|---|---|
| The provider variable | This variable is what will be used to interact with our provider.<br>The variable must be final and "top-level" (global). |
| The provider type | Generally either NotifierProvider, AsyncNotifierProvider or StreamNotifierProvider.<br>The type of provider used depends on the return value of your function. For example, to create a Future<Activity>, you'll want a AsyncNotifierProvider<Activity>.<br><br>AsyncNotifierProvider is the one you'll want to use the most.<br><br>**TIP:** Don't think in terms of "Which provider should I pick". Instead, think in terms of "What do I want to return". The provider type will follow naturally. |

## Defining a Notifier (3)

```
final name = SomeNotifierProvider.someModifier<MyNotifier, Result>(MyNotifier.new);
```

| | |
|---|---|
| Modifiers (optional) | Often, after the type of the provider you may see a "modifier".<br>Modifiers are optional, and are used to tweak the behavior of the provider in a way that is type-safe.<br><br>There are currently two modifiers available:<br>● autoDispose, which will automatically clear the cache when the provider stops being used.<br>● family, which enables passing arguments to your provider. |
| The Notifier's constructor | The parameter of "notifier providers" is a function which is expected to instantiate the "notifier".<br>It generally should be a "constructor tear-off". |

## Defining a Notifier (4)

```
class MyNotifier extends SomeNotifier<Result> {
  @override
  Result build() {
    <your logic here>
  }
  <your methods here>
}
```

| | |
|---|---|
| The Notifier | If NotifierProvider is the "StatefulWidget" class, then this part is the State class.<br><br>This class is responsible for exposing ways to modify the state of the provider. Public methods on this class are accessible to consumers using ref.read(yourProvider.notifier).yourMethod().<br><br>**NOTE:** Notifiers should not have public properties besides the built-in state, as the UI would have no mean to know that state has changed. |
| The Notifier type | The base class extended by your notifier should match that of the provider + modifiers. Some examples would be:<br>● NotifierProvider -> Notifier<br>● AsyncNotifierProvider -> AsyncNotifier<br>● AsyncNotifierProvider.autoDispose -> AutoDisposeAsyncNotifier<br>● AsyncNotifierProvider.autoDispose.family -> AutoDisposeFamilyAsyncNotifier |

## Defining a Notifier (5)

```
class MyNotifier extends SomeNotifier<Result> {
  @override
  Result build() {
    <your logic here>
  }
  <your methods here>
}
```

| | |
|---|---|
| The build method | All notifiers must override the build method.<br>This method is equivalent to the place where you would normally put your logic in a non-notifier provider.<br><br>This method should not be called directly. |

## Using ref.invalidateSelf() to refresh the provider.

One option is to have our provider re-execute the GET request.

This can be done by calling ref.invalidateSelf() after the POST request:

```
Future<void> addTodo(Todo todo) async {
  // We don't care about the API response
  await http.post(
    Uri.https('your_api.com', '/todos'),
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode(todo.toJson()),
  );

  // Once the post request is done, we can mark the local cache as dirty.
  // This will cause "build" on our notifier to asynchronously be called again,
  // and will notify listeners when doing so.
  ref.invalidateSelf();

  // (Optional) We can then wait for the new state to be computed.
  // This ensures "addTodo" does not complete until the new state is available.
  await future;
}
```

## Updating the local cache manually

Another option is to update the local cache manually.

This would involve trying to mimic the backend's behavior. For instance, we would need to know whether the backend inserts new items at the start or at the end.

```
Future<void> addTodo(Todo todo) async {
  await http.post(
    Uri.https('your_api.com', '/todos'),
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode(todo.toJson()),
  );

  final previousState = await future;

  state = AsyncData([...previousState, todo]);
}
```

## Updating the local cache manually (2)

**NOTE:** This example uses immutable state. This is not required, but recommended.
If you want to use mutable state instead, you can alternatively do:

```
final previousState = await future;
// Mutable the previous list of todos.
previousState.add(todo);
// Manually notify listeners.
ref.notifyListeners();
```

2.4

## Passing arguments to your requests

In a previous article, we saw how we could define a "provider" to make a simple GET HTTP request.

But often, HTTP requests depend on external parameters.

## Updating our providers to accept arguments

For example, we defined our provider like this:

```
// A "functional" provider
final activityProvider = FutureProvider.autoDispose((ref) async {
  // TODO: perform a network request to fetch an activity
  return fetchActivity();
});

// Or alternatively, a "notifier"
final activityProvider2 = AsyncNotifierProvider<ActivityNotifier, Activity>(
  ActivityNotifier.new,
);

class ActivityNotifier extends AsyncNotifier<Activity> {
  @override
  Future<Activity> build() async {
    // TODO: perform a network request to fetch an activity
    return fetchActivity();
  }
}
```

## Updating our providers to accept arguments (2)

we need to add .family after the type of our provider, and an extra type parameter corresponding to the argument type. For example, we could update our provider to accept a String argument corresponding to the type of activity desired:

```
final activityProvider = FutureProvider.autoDispose
    // We use the ".family" modifier.
    // The "String" generic type corresponds to the argument type.
    // Our provider now receives an extra argument on top of "ref": the activity type.
    .family<Activity, String>((ref, activityType) async {
  // TODO: perform a network request to fetch an activity using "activityType"
  return fetchActivity(activityType);
});

// Again, for notifier we use the ".family" modifier, and specify the argument as type "String".
final activityProvider2 = AsyncNotifierProvider.autoDispose
    .family<ActivityNotifier, Activity, String>(
  ActivityNotifier.new,
);
```

## Updating our providers to accept arguments (3)

```
// When using ".family" with notifiers, we need to change the notifier subclass:
// AsyncNotifier -> FamilyAsyncNotifier
// AutoDisposeAsyncNotifier -> AutoDisposeFamilyAsyncNotifier
class ActivityNotifier extends AutoDisposeFamilyAsyncNotifier<Activity, String> {
  /// Family arguments are passed to the build method and accessible with this.arg
  @override
  Future<Activity> build(String activityType) async {
    // Arguments are also available with "this.arg" print(this.arg);
    // TODO: perform a network request to fetch an activity
    return fetchActivity(activityType);
  }
}
```

## Caching considerations and parameter restrictions

When passing parameters to providers, the computation is still cached. The difference is that the computation is now cached per-argument.

If two widgets consumes the same provider with the same parameters, only a single network request will be made.

But if two widgets consumes the same provider with different parameters, two network requests will be made.

For this to work, Riverpod relies on the == operator of the parameters.

As such, it is important that the parameters passed to the provider have consistent equality.

## Caching considerations and parameter restrictions (2)

**CAUTION!:** A common mistake is to directly instantiate a new object as the parameter of a provider, when that object does not override ==.
For example, you may be tempted to pass a List like so:

```
// We could update activityProvider to accept a list of strings instead.
// Then be tempted to create that list directly in the watch call.
ref.watch(activityProvider(['recreational', 'cooking']));
```

The problem with this code is that ['recreational', 'cooking'] == ['recreational', 'cooking'] is false. As such, Riverpod will consider that the two parameters are different, and attempt to make a new network request. This would result in an infinite loop of network requests, permanently showing a progress indicator to the user.

To fix this, you could either use a const list (const ['recreational', 'cooking']) or use a custom list implementation that overrides ==.

## Caching considerations and parameter restrictions (3)

With that in mind, you may wonder how to pass multiple parameters to a provider.

The recommended solution is either to:

- Switch to code-generation, which enables passing any number of parameters
- Use Dart 3's records

The reason why Dart 3's records come in handy is because they naturally override == and have a convenient syntax.

2.5

## Websockets

A common use-case of modern applications is to interact with websockets, such as with Firebase or GraphQL subscriptions.

Interacting with those APIs is often done by listening to a Stream.

## Updating our providers to accept arguments (2)

To help with that, Riverpod naturally supports Stream objects. Like with Futures, the object will be converted to an AsyncValue:

```
final streamExampleProvider = StreamProvider.autoDispose<int>((ref) async* {
  // Every 1 second, yield a number from 0 to 41.
  // This could be replaced with a Stream from Firestore or GraphQL or anything else.
  for (var i = 0; i < 42; i++) {
    yield i;
    await Future<void>.delayed(const Duration(seconds: 1));
  }
});
```

## Disabling conversion of Streams/Futures to AsyncValue

By default, Riverpod will convert Streams and Futures to AsyncValue. Although rarely needed, it is possible to disable this behavior by wrapping the return type in a Raw typedef.

```
@riverpod Raw<Stream<int>> rawStream(RawStreamRef ref) {
  // "Raw" is a typedef. No need to wrap the return
  // value in a "Raw" constructor.
  return const Stream<int>.empty();
}

class Consumer extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // The value is no-longer converted to AsyncValue,
    // and the created stream is returned as is.
    Stream<int> stream = ref.watch(rawStreamProvider);
    return StreamBuilder<int>(
      stream: stream, builder: (context, snapshot) {
        return Text('${snapshot.data}');
      },
    );
  }
}
```

## 2.6 Combining requests

## The basics: Obtaining a "ref"

All possible ways of combining requests have one thing in common: They are all based on the Ref object.

The Ref object is an object to which all providers have access. It grants them access to various life-cycle listeners, but also various methods to combine providers.

In functional providers, the Ref is passed as a parameter to the provider's function:

```
final provider = Provider<int>((ref) {
  // "Ref" can be used here to read other providers
  final otherValue = ref.watch(otherProvider);

  return 0;
});
```

## The basics: Obtaining a "ref" (2)

In class variants, the Ref is a property of the Notifier class:

```
final provider = NotifierProvider<MyNotifier, int>(MyNotifier.new);

class MyNotifier extends Notifier<int> {
  @override int build() {
    // "Ref" can be used here to read other providers
    final otherValue = ref.watch(otherProvider);

    return 0;
  }
}
```

## Using ref to read a provider.

## The ref.watch method.

The ref.watch method takes a provider, and returns its current state. Then, whenever the listened provider changes, our provider will be invalidated and rebuilt next frame or on next read.

## The ref.watch method.

```
final locationProvider = StreamProvider<({double longitude, double latitude})>((ref) {
  // TO-DO: Return a stream which obtains the current location
  return someStream;
});

final restaurantsNearMeProvider = FutureProvider<List<String>>((ref) async {
  // We use "ref.watch" to obtain the latest location.
  // By specifying that ".future" after the provider, our code will wait
  // for at least one location to be available.
  final location = await ref.watch(locationProvider.future);

  // We can now make a network request based on that location.
  // For example, we could use the Google Map API:
  // https://developers.google.com/maps/documentation/places/web-service/search-nearby
  final response = await http.get(
    Uri.https('maps.googleapis.com', 'maps/api/place/nearbysearch/json', {
      'location': '${location.latitude},${location.longitude}',
      ...
    }),
  );

  // Obtain the restaurant names from the JSON
  final json = jsonDecode(response.body) as Map;
  final results = (json['results'] as List).cast<Map<Object?, Object?>>();
  return results.map((e) => e['name']! as String).toList();
});
```

## The ref.listen methods.

The `ref.listen` method is an alternative to `ref.watch`.

It is similar to your traditional "listen"/"addListener" method. It takes a provider and a callback, and will invoke said callback whenever the content of the provider changes.

Refactoring your code such that you can use ref.watch instead of ref.listen is generally recommended, as the latter is more error-prone due to its imperative nature.

**INFO:** It is entirely safe to use ref.listen during the build phase of a provider. If the provider somehow is recomputed, previous listeners will be removed.

Alternatively, you can use the return value of ref.listen to remove the listener manually when you wish.

## The ref.listen methods. (2)

We could rewrite the ref.watch example to use ref.listen instead

```
final provider = Provider<int>((ref) {
  ref.listen(otherProvider, (previous, next) {
    print('Changed from: $previous, next: $next');
  });

  return 0;
});
```

## The ref.read method.

The last option available is ref.read. It is similar to ref.watch in that it returns the current state of a provider. But unlike ref.watch, it doesn't listen to the provider.

As such, ref.read should only be used in places where you can't use ref.watch, such as inside methods of Notifiers.

```
final notifierProvider = NotifierProvider<MyNotifier, int>(MyNotifier.new);

class MyNotifier extends Notifier<int> {
  @override
  int build() {
    // Bad! Do not use "read" here as it is not reactive
    ref.read(otherProvider);

    return 0;
  }

  void increment() {
    ref.read(otherProvider); // Using "read" here is fine
  }
}
```

## The ref.read method. (2)

**CAUTION!:** Be careful when using ref.read on a provider as, since it doesn't listen to the provider, said provider may decide to destroy its state if it isn't listened.

## 2.7

## Clearing cache and reacting to state disposal

Riverpod offers various ways to interact with state disposal. This ranges from delaying the disposal of state to reacting to destruction.

## When is state destroyed and how to change this?

To enable automatic disposal, you can use .autoDispose next to the provider type:

```
// We can specify autoDispose to enable automatic state destruction.
final provider = Provider.autoDispose<int>((ref) {
  return 0;
});
```

## When is state destroyed and how to change this? (2)

When using code-generation, by default, the state is destroyed when the provider stops being listened to. This happens when a listener has no active listener for a full frame. When that happens, the state is destroyed.

This behavior can be opted out by using keepAlive: true.

Doing so will prevent the state from getting destroyed when all listeners are removed.

```
// We can specify "keepAlive" in the annotation to disable
// the automatic state destruction
@Riverpod(keepAlive: true)
int example(ExampleRef ref) {
  return 0;
}
```

## Manually forcing the destruction of a provider, using ref.invalidate

Sometimes, you may want to force the destruction of a provider. This can be done by using `ref.invalidate`, which can be called from another provider or from a widget.

Using ref.invalidate will destroy the current provider state. There are then two possible outcomes:

- If the provider is listened, a new state will be created.
- If the provider is not listened, the provider will be fully destroyed.

```
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'),
    );
  }
}
```

## Fine-tuned disposal with ref.keepAlive

As mentioned, when automatic disposal is enabled, the state is destroyed when the provider has no listeners for a full frame.

But you may want to have more control over this behavior. For instance, you may want to keep the state of successful network requests, but not cache failed requests.

```
final provider = FutureProvider.autoDispose<String>((ref) async {
  final response = await http.get(Uri.parse('https://example.com'));
  // We keep the provider alive only after the request has successfully completed.
  // If the request failed (and threw an exception), then when the provider stops being
  // listened to, the state will be destroyed.
  final link = ref.keepAlive();

  // We can use the `link` to restore the auto-dispose behavior with:
  // link.close();

  return response.body;
});
```

2.8

## Eager initialization of providers

All providers are initialized lazily by default. This means that the provider is only initialized when it is first used. This is useful for providers that are only used in certain parts of the application.

The recommended approach is to simply "watch" a provider in a Consumer placed right under your ProviderScope:

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

class MyApp extends StatelessWidget {
  @override Widget build(BuildContext context) {
    return const _EagerInitialization(
      // TODO: Render your app here
      child: MaterialApp(),
    );
  }
}

class _EagerInitialization extends ConsumerWidget {
  const _EagerInitialization({required this.child});
  final Widget child;

@override Widget build(BuildContext context, WidgetRef ref) {
  // Eagerly initialize providers by watching them.
  // By using "watch", the provider will stay alive and not be disposed.
  ref.watch(myProvider);
  return child;
  }
}
```

2.9

## Logging and error reporting

Riverpod natively offers a way to listen to all events happening in the provider tree. This can be used to log all the events, or to report errors to a remote service.

This is achieved by using the ProviderObserver class, and passing it to ProviderScope/ProviderContainer.

## Defining a ProviderObserver

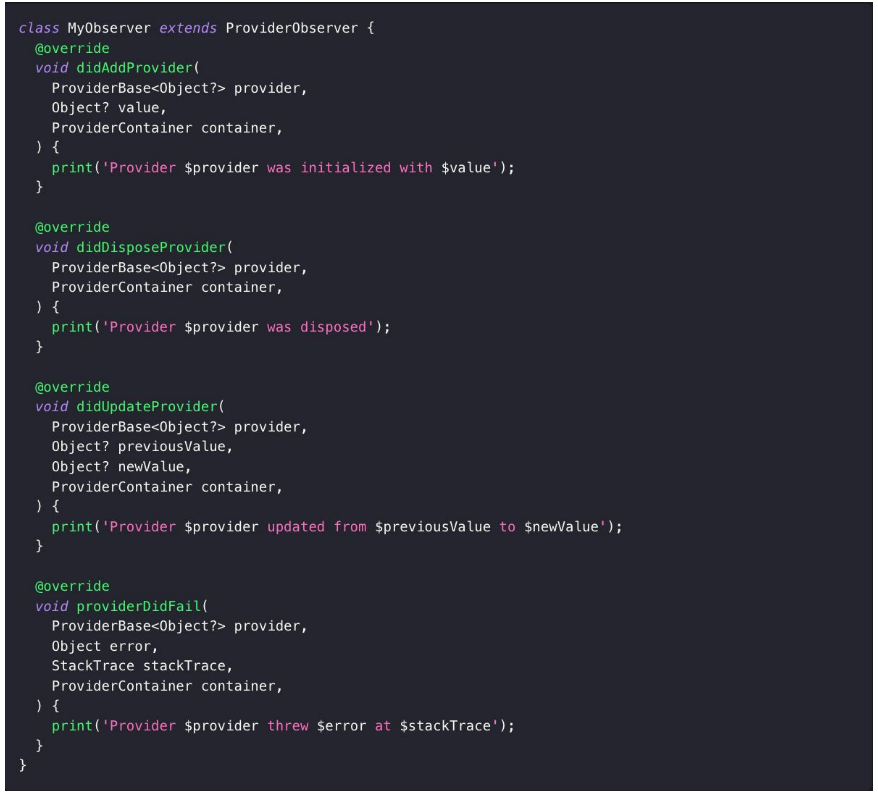
A ProviderObserver is a class that should be extended.

It offers various methods which can be overridden to listen to events:

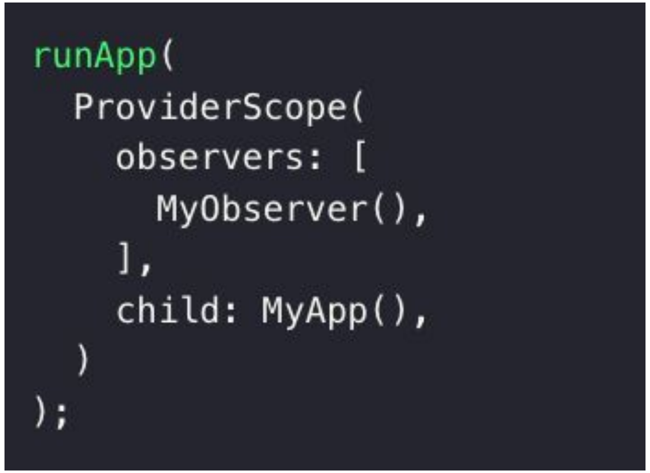
- didAddProvider, called when a provider is added to the tree
- didUpdateProvider, called when a provider is updated
- didDisposeProvider, called when a provider is disposed
- providerDidFail, when a synchronous provider throws an error

## Using a ProviderObserver

Now that we've defined an observer, we need to use it.



```dart
class MyObserver extends ProviderObserver {
  @override
  void didAddProvider(
    ProviderBase<Object?> provider,
    Object? value,
    ProviderContainer container,
  ) {
    print('Provider $provider was initialized with $value');
  }

  @override
  void didDisposeProvider(
    ProviderBase<Object?> provider,
    ProviderContainer container,
  ) {
    print('Provider $provider was disposed');
  }

  @override
  void didUpdateProvider(
    ProviderBase<Object?> provider,
    Object? previousValue,
    Object? newValue,
    ProviderContainer container,
  ) {
    print('Provider $provider updated from $previousValue to $newValue');
  }

  @override
  void providerDidFail(
    ProviderBase<Object?> provider,
    Object error,
    StackTrace stackTrace,
    ProviderContainer container,
  ) {
    print('Provider $provider threw $error at $stackTrace');
  }
}
```



```dart
runApp(
  ProviderScope(
    observers: [
      MyObserver(),
    ],
    child: MyApp(),
  )
);
```

3

## Bonus

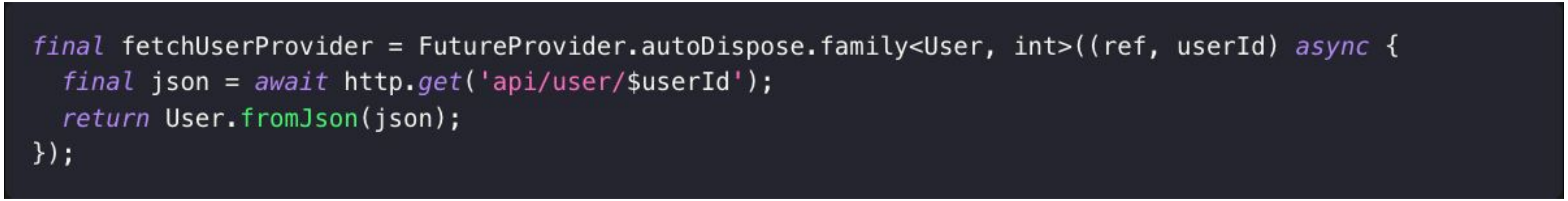
## Types of Providers

| Provider Type | Provider Create Function | Example Use Case |
|---|---|---|
| Provider | Return any type | A service class / computed property (filtered list) |
| StateProvider (legacy) | Return any type | A filter condition / simple state object |
| StateNotifierProvider (legacy) | Return a Future of any type | A result from an API call |
| FutureProvider | Return a Stream of any type | A stream of results from an API |
| StreamProvider | Return a subclass of StateNotifier | A complex state object that is immutable except through an interface |
| ChangeNotifierProvider (legacy) | Return a subclass of ChangeNotifier | A complex state object that requires mutability |
| (Async)NotifierProvider (new 2.0) | | |

## About code generation

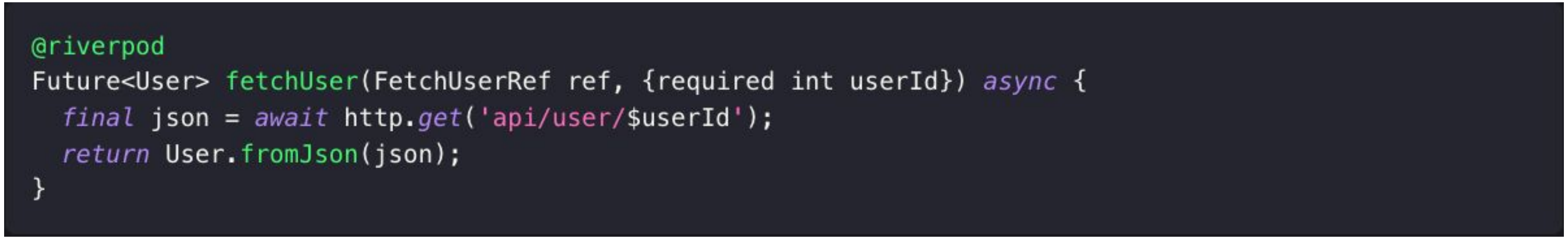
Code generation is the idea of using a tool to generate code for us. In Dart, it comes with the downside of requiring an extra step to "compile" an application. Although this problem may be solved in the near future, as the Dart team is working on a potential solution to this problem.

In the context of Riverpod, code generation is about slightly changing the syntax for defining a "provider". For example, instead of:



```dart
final fetchUserProvider = FutureProvider.autoDispose.family<User, int>((ref, userId) async {
  final json = await http.get('api/user/$userId');
  return User.fromJson(json);
});
```

Using code generation, we would write:



```dart
@riverpod
Future<User> fetchUser(FetchUserRef ref, {required int userId}) async {
  final json = await http.get('api/user/$userId');
  return User.fromJson(json);
}
```

## Migrate from non-code-generation variant:

When using non-code-generation variant, it is necessary to manually determine the type of your provider. The following are the corresponding options for transitioning into code-generation variant:

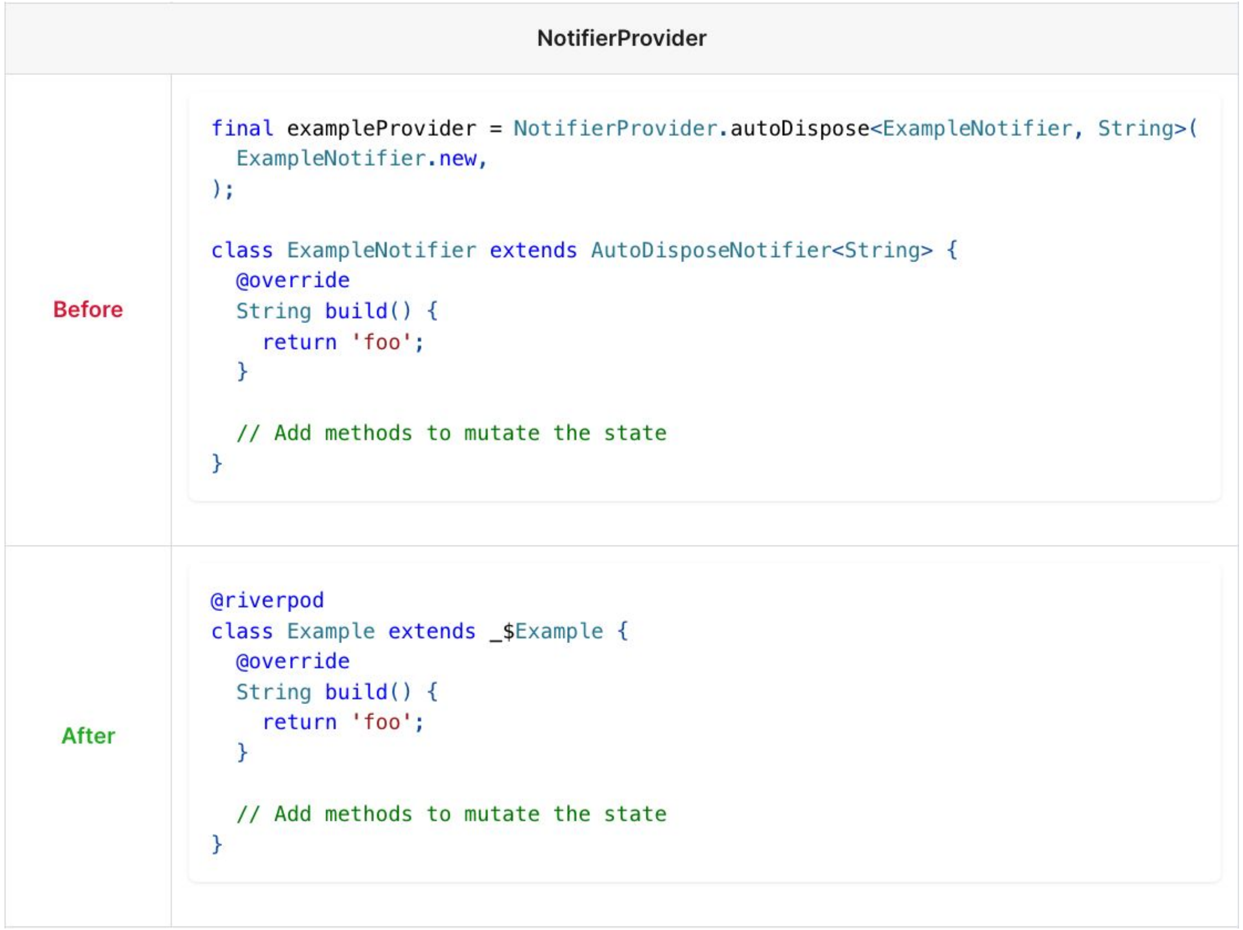
**Provider**

**Before**

```dart
final exampleProvider = Provider.autoDispose<String>(
  (ref) {
    return 'foo';
  },
);
```

**After**

```dart
@riverpod
String example(ExampleRef ref) {
  return 'foo';
}
```



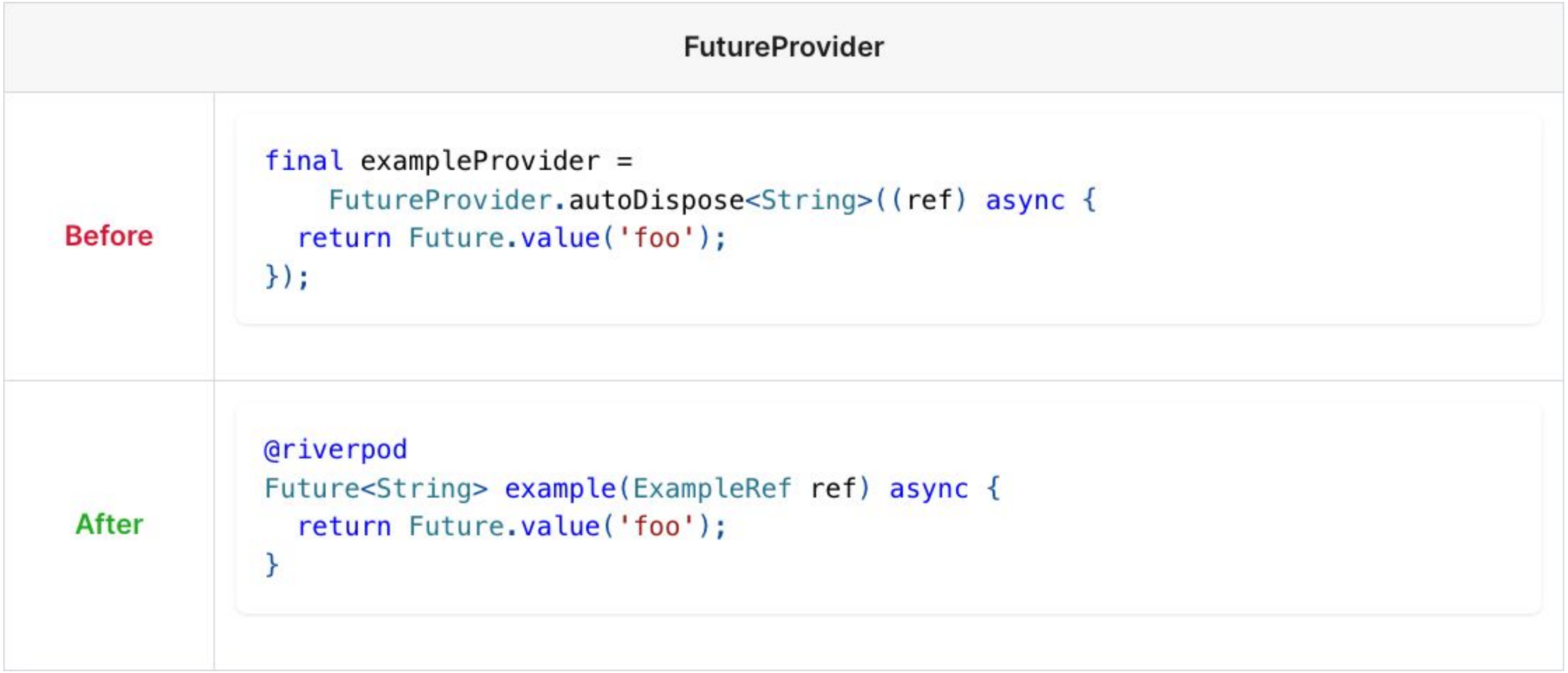
**NotifierProvider**

**Before**

```dart
final exampleProvider = NotifierProvider.autoDispose<ExampleNotifier, String>(
  ExampleNotifier.new,
);

class ExampleNotifier extends AutoDisposeNotifier<String> {
  @override
  String build() {
    return 'foo';
  }

  // Add methods to mutate the state
}
```

**After**

```dart
@riverpod
class Example extends _$Example {
  @override
  String build() {
    return 'foo';
  }

  // Add methods to mutate the state
}
```



## FutureProvider

**Before**

```dart
final exampleProvider =
    FutureProvider.autoDispose<String>((ref) async {
  return Future.value('foo');
});
```

**After**

```dart
@riverpod
Future<String> example(ExampleRef ref) async {
  return Future.value('foo');
}
```

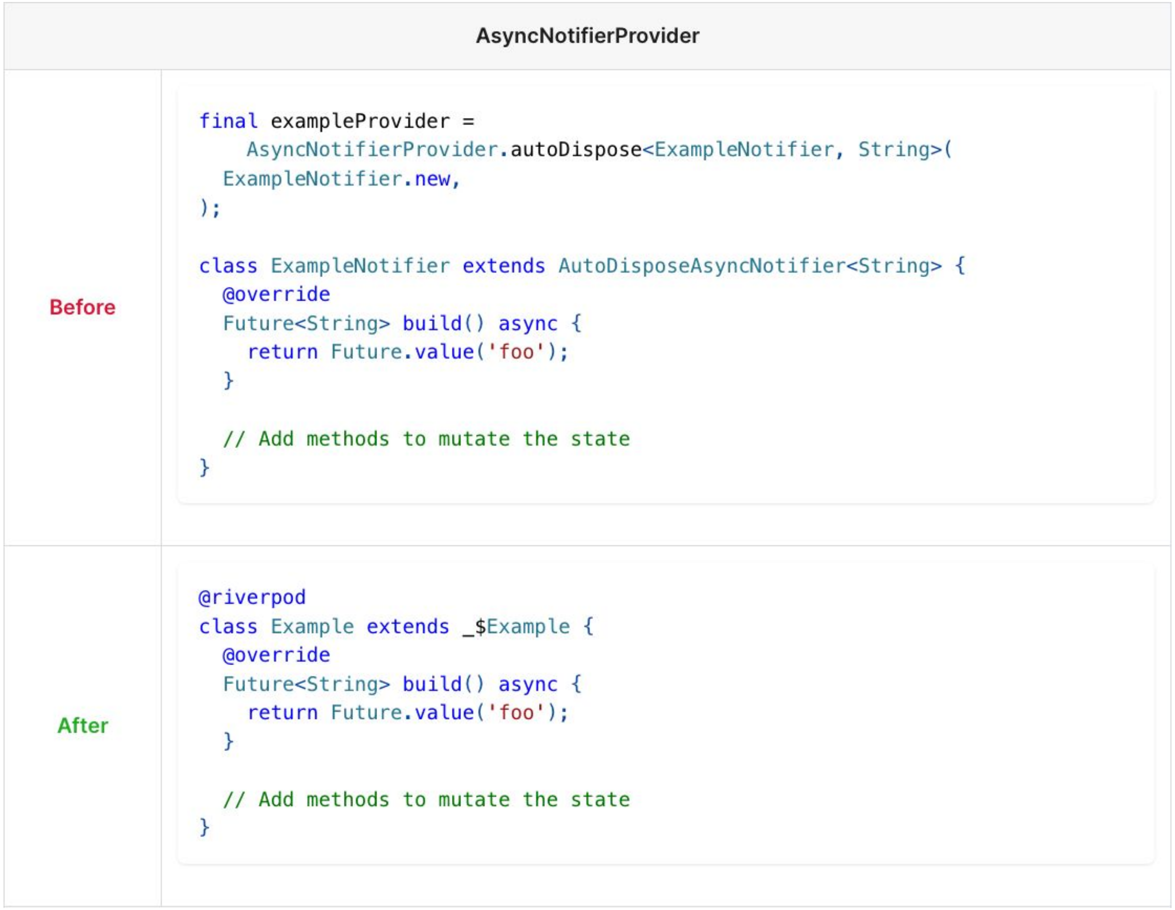
## StreamProvider

**Before**

```dart
final exampleProvider =
    StreamProvider.autoDispose<String>((ref) async* {
  yield 'foo';
});
```

**After**

```dart
@riverpod
Stream<String> example(ExampleRef ref) async* {
  yield 'foo';
}
```



**AsyncNotifierProvider**

**Before**

```dart
final exampleProvider =
    AsyncNotifierProvider.autoDispose<ExampleNotifier, String>(
  ExampleNotifier.new,
);

class ExampleNotifier extends AutoDisposeAsyncNotifier<String> {
  @override
  Future<String> build() async {
    return Future.value('foo');
  }

  // Add methods to mutate the state
}
```

**After**

```dart
@riverpod
class Example extends _$Example {
  @override
  Future<String> build() async {
    return Future.value('foo');
  }

  // Add methods to mutate the state
}
```

## StreamNotifierProvider

**Before**

```dart
final exampleProvider =
    StreamNotifierProvider.autoDispose<ExampleNotifier, String>(() {
  return ExampleNotifier();
});

class ExampleNotifier extends AutoDisposeStreamNotifier<String> {
  @override
  Stream<String> build() async* {
    yield 'foo';
  }

  // Add methods to mutate the state
}
```

**After**

```dart
@riverpod
class Example extends _$Example {
  @override
  Stream<String> build() async* {
    yield 'foo';
  }

  // Add methods to mutate the state
}
```

THANK YOU

## Flutter Riverpod Meeting