Riverpod 3.0의 새로운 기능

Riverpod 3.0에 오신 것을 환영합니다!
이번 업데이트에는 오랫동안 기다려온 많은 기능들, 버그 수정, 그리고 API 단순화가 포함되어 있습니다.

이 버전은 더 간단하고 통합된 Riverpod로의 전환기입니다.

> **주의**
>
> 이 버전은 몇 가지 생명주기 변경사항을 포함하고 있습니다. 이런 변경사항들이 미묘한 방식으로 앱을 손상시킬 수 있습니다. 신중하게 업그레이드하세요. 마이그레이션 가이드는 마이그레이션 페이지를 참고해주세요.

주요 하이라이트는 다음과 같습니다:

- 오프라인 지속성 (experimental) - Provider를 데이터베이스에 저장하도록 선택할 수 있습니다
- Mutation (experimental) - 인터페이스가 사이드이펙트에 반응할 수 있게 하는 새로운 메커니즘
- 자동 재시도 - Provider가 실패할 때 지수적 백오프로 자동으로 새로고침 됩니다
- `Ref.mounted` - `BuildContext.mounted`와 유사하지만 `Ref`용입니다.
- 제네릭 지원 (코드 생성) - 생성된 Provider가 이제 타입 매개변수를 정의할 수 있습니다
- 일시정지/재개 지원 - `ref.listen` 사용 시 리스너를 일시적으로 일시정지할 수 있습니다
- 공개 API 통합 - 동작이 통합되고 중복 인터페이스가 융합됩니다
- Provider 생명주기 변경 - 최신 코드에 더 잘 맞도록 Provider 동작에 약간의 조정
- 새로운 테스팅 유틸리티:
  - `ProviderContainer.test` - 컨테이너를 생성하고 테스트 종료 후 자동으로 해제하는 테스트 유틸리티입니다.
  - `NotifierProvider.overrideWithBuild` - 전체 notifier를 모킹하지 않고 `Notifier.build`만 모킹합니다.
  - `Future/StreamProvider.overrideWithValue` - 기존 유틸리티가 복원되었습니다
  - `WidgetTester.container` - 위젯 테스트 내에서 `ProviderContainer`를 얻는 헬퍼 메서드입니다.
- 정적으로 안전한 스코핑 - 오버라이드가 누락된 경우를 감지하는 새로운 린트 규칙이 추가되었습니다

오프라인 지속 기능 (experimental)

> **정보**
>
> 이 기능은 실험적이며 아직 안정적이지 않습니다. 사용 가능하지만, 주요 버전 업데이트 없이도 API가 파괴적인 방식으로 변경될 수 있습니다.

오프라인 지속 기능은 Provider를 장치에 로컬로 캐시할 수 있게 하는 새로운 기능입니다. 오프라인 지속 기능을 사용하면 앱이 닫히고 다시 열렸을 때 Provider를 캐시에서 복원할 수 있습니다. 이 기능은 선택적이며, 모든 "Notifier" Provider에서 지원되고, 코드 생성 사용 여부와 관계없이 작동합니다.

Riverpod는 데이터베이스 자체는 포함하지 않고, 데이터베이스와 상호작용하는 인터페이스만 포함합니다. 인터페이스를 구현하면 원하는 모든 데이터베이스를 사용할 수 있습니다. SQLite용 공식 패키지가 관리되고 있습니다: riverpod_sqflite.

간단한 데모로서 오프라인 지속성을 사용하는 방법을 보여드립니다:

riverpod **riverpod_generator**

```dart
// 코드 생성 없이 JsonSqFliteStorage를 보여주는 예제입니다.
final storageProvider = FutureProvider<JsonSqFliteStorage>((ref) async {
  // SQFlite를 초기화합니다. Provider 간에 Storage 인스턴스를 공유해야 합니다.
  return JsonSqFliteStorage.open(
    join(await getDatabasesPath(), 'riverpod.db'),
  );
});

/// 직렬화 가능한 Todo 클래스입니다.
class Todo {
  const Todo({
    required this.id,
    required this.description,
    required this.completed,
  });

  Todo.fromJson(Map<String, dynamic> json)
      : id = json['id'] as int,
        description = json['description'] as String,
        completed = json['completed'] as bool;
```

```
  final int id;
  final String description;
  final bool completed;

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'description': description,
      'completed': completed,
    };
  }
}

final todosProvider =
    AsyncNotifierProvider<TodosNotifier, List<Todo>>(TodosNotifier.new);

class TodosNotifier extends AsyncNotifier<List<Todo>>{
  @override
  FutureOr<List<Todo>> build() async {
    // 'build' 메서드 시작 부분에서 persist를 호출합니다.
    // 이렇게 하면:
    // - 이 메서드가 처음 실행될 때 DB를 읽고 지속된 값으로 상태를 업데이트합니다.
    // - 이 Provider의 변경사항을 수신하고 그 변경사항을 DB에 씁니다.
    persist(
      // JsonSqFliteStorage 인스턴스를 전달합니다. Future를 "await"할 필요가 없습니
다.
      // Riverpod가 처리합니다.
      ref.watch(storageProvider.future),
      // 이 상태의 고유 키입니다.
      // 다른 Provider가 같은 키를 사용해서는 안 됩니다.
      key: 'todos',
      // 기본적으로 상태는 2일 동안만 오프라인으로 캐시됩니다.
      // 캐시 지속 시간을 변경하려면 선택적으로 다음 줄의 주석을 해제할 수 있습니다.
      // options: const StorageOptions(cacheTime:
StorageCacheTime.unsafe_forever),
      encode: jsonEncode,
      decode: (json) {
        final decoded = jsonDecode(json) as List;
        return decoded
            .map((e) => Todo.fromJson(e as Map<String, Object?>))
            .toList();
      },
    );
```

```
      // 서버에서 비동기적으로 할일을 가져옵니다.
      // await 중에 지속된 할일 목록을 사용할 수 있습니다.
      // 네트워크 요청이 완료된 후 서버 상태가
      // 지속된 상태보다 우선합니다.
      final todos = await fetchTodos();
      return todos;
    }

    Future<void> add(Todo todo) async {
      // 상태를 수정할 때 변경사항을 지속하기 위한 추가 로직이 필요하지 않습니다.
      // Riverpod가 자동으로 새 상태를 캐시하고 DB에 씁니다.
      state = AsyncData([...await future, todo]);
    }
  }
```

Mutation (experimental)

> **정보**
>
> 이 기능은 실험적이며 아직 안정적이지 않습니다. 사용 가능하지만, 주요 버전 업데이트 없이도 API가 파괴적인 방식으로 변경될 수 있습니다.

Riverpod 3.0에서 "Mutation"이라는 새로운 기능이 도입되었습니다.
이 기능은 아래 두 가지 문제를 해결합니다:

- UI가 "사이드이펙트"(폼 제출, 버튼 클릭 등)에 반응하여 로딩/성공/오류 메시지를 표시할 수 있게 해줍니다. "Form이 성공적으로 제출되었을 때 Toast로 보여주기" 같은 기능을 생각해보세요.
- Ref.read와 Automatic disposal를 결합한 `onPressed` 콜백이 사이드이펙트이 아직 진행 중인 동안 Provider가 해제되는 문제를 해결합니다.

간단히 말하면, 새로운 Mutation 객체가 추가되었습니다. 이는 Provider처럼 최상위 final 변수로 선언됩니다:

```
final addTodoMutation = Mutation<void>();
```

그 후, UI에서 `ref.listen` / `ref.watch`를 사용하여 Mutation의 상태를 수신할 수 있습니다:

```dart
class AddTodoButton extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // "addTodo" 사이드이펙트의 상태를 수신합니다
    final addTodo = ref.watch(addTodoMutation);

    return switch (addTodo) {
      // 진행 중인 사이드이펙트이 없습니다
      // 제출 버튼을 보여줍니다
      MutationIdle() => ElevatedButton(
        // 클릭 시 사이드이펙트을 트리거합니다
        onPressed: () {
          // TODO 코드 스니펫 이후 설명을 참조하세요
        },
        child: const Text('제출'),
      ),
      // 사이드이펙트이 진행 중입니다. 스피너를 보여줍니다
      MutationPending() => const CircularProgressIndicator(),
      // 사이드이펙트이 실패했습니다. 재시도 버튼을 보여줍니다
      MutationError() => ElevatedButton(
        onPressed: () {
          // TODO 코드 스니펫 이후 설명을 참조하세요
        },
        child: const Text('재시도'),
      ),
      // 사이드이펙트이 성공했습니다. 성공 메시지를 보여줍니다
      MutationSuccess() => const Text('할일이 추가되었습니다!'),
    };
  }
}
```

마지막으로, `onPressed` 콜백 내에서 다음과 같이 사이드이펙트을 트리거할 수 있습니다:

```dart
onPressed: () {
  addTodoMutation.run(ref, (tsx) async {
    // 여기서 사이드이펙트을 실행합니다.
    // 보통 여기서 Notifier를 가져와서 메서드를 호출합니다.
    await tsx.get(todoListProvider.notifier).addTodo('새 할일');
  });
}
```

> **노트**
>
> 여기서 Ref.read 대신 tsx.get을 호출했다는 점에 주목하세요. 이는 Mutation 고유의 기능입니다. tsx.get은 Provider의 상태를 가져오지만, Mutation이 완료될 때까지 그것을 유지합니다.

자동 재시도

3.0부터, 초기화 중에 실패하는 Provider는 자동으로 재시도됩니다. 재시도는 지수적 백오프로 수행되며, Provider는 성공하거나 해제될 때까지 재시도됩니다. 이는 네트워크 연결 부족과 같은 임시적인 문제로 인한 작업 실패에 도움이 됩니다.

기본 동작은 모든 오류를 재시도하고, 200ms 지연으로 시작하여 각 재시도 후 최대 6.4초까지 두 배씩 증가합니다.
이는 `retry` 매개변수를 전달하여 ProviderContainer/ProviderScope에서 모든 Provider에 대해 사용자 정의할 수 있습니다:

ProviderScope **ProviderContainer**

```
void main() {
  runApp(
    ProviderScope(
      // 특정 오류를 건너뛰거나 재시도 횟수에 제한을 추가하거나
      // 지연을 변경하는 등 재시도 로직을 사용자 정의할 수 있습니다
      retry: (retryCount, error) {
        if (error is SomeSpecificError) return null;
        if (retryCount > 5) return null;

        return Duration(seconds: retryCount * 2);
      },
      child: MyApp(),
    ),
  );
}
```

또는 Provider 생성자에 `retry` 매개변수를 전달하여 Provider별로 구성할 수 있습니다:

riverpod riverpod_generator

```dart
final todoListProvider = NotifierProvider<TodoList, List<Todo>>(
  TodoList.new,
  retry: (retryCount, error) {
    if (error is SomeSpecificError) return null;
    if (retryCount > 5) return null;

    return Duration(seconds: retryCount * 2);
  },
);
```

`Ref.mounted`

오래 기다렸던 `Ref.mounted`가 마침내 등장했습니다! `BuildContext.mounted`와 유사합니다. 비동기 작업 후에 Provider가 아직 마운트되어 있는지 확인하는 데 사용할 수 있습니다:

riverpod riverpod_generator

```dart
class TodoList extends Notifier<List<Todo>> {
  @override
  List<Todo> build() => [];

  Future<void> addTodo(String title) async {
    // 새 할일을 서버에 게시합니다
    final newTodo = await api.addTodo(title);
    // 비동기 작업 후에
    // Provider가 아직 마운트되어 있는지 확인합니다
    if (!ref.mounted) return;

    // 마운트되어 있다면 상태를 업데이트합니다
    state = [...state, newTodo];
  }
}
```

이것이 작동하려면 상당한 생명주기 변경이 필요했습니다. 생명주기 변경 섹션을 꼭 읽어보세요.

제네릭 지원 (코드 생성)

코드 생성을 사용할 때 이제 생성된 Provider에 대해 타입 매개변수를 정의할 수 있습니다. 타입 매개변수는 다른 Provider 매개변수와 같이 작동하며, Provider를 감시할 때 전달되어야 합니다.

```
@riverpod
T multiply<T extends num>(T a, T b) {
  return a * b;
}

// ...

int integer = ref.watch(multiplyProvider<int>(2, 3));
double decimal = ref.watch(multiplyProvider<double>(2.5, 3.5));
```

일시정지/재개 지원

2.0에서 Riverpod는 이미 어느 정도 일시정지/재개 지원이 있었지만 상당히 제한적이었습니다. 3.0에서는 모든 `ref.listen` 리스너를 필요에 따라 수동으로 일시정지/재개할 수 있습니다:

```
final subscription = ref.listen(
  todoListProvider,
  (previous, next) {
    // 새 값으로 뭔가를 합니다
  },
);

subscription.pause();
subscription.resume();
```

동시에, Riverpod는 이제 다양한 상황에서 Provider를 일시정지합니다:

- Provider가 더 이상 보이지 않으면 일시정지됩니다 (TickerMode 기반).
- Provider가 재구축되면, 재구축이 완료될 때까지 구독이 일시정지됩니다.
- Provider가 일시정지되면, 모든 구독도 일시정지됩니다.

자세한 내용은 생명주기 변경 섹션을 참조하세요.

공개 API 통합

Riverpod 3.0의 목표 중 하나는 API를 단순화하는 것입니다. 여기에는 아래 세부 목표를 포함했습니다.

- 권장하는 것과 그렇지 않은 것을 강조
- 불필요한 인터페이스 중복 제거
- 모든 기능이 일관된 방식으로 작동하도록 보장

이를 위해 몇 가지 변경사항이 있었습니다

[StateProvider]/[StateNotifierProvider]와 [ChangeNotifierProvider]는 권장되지 않으며 다른 임포트로 이동됨

이러한 Provider는 제거되지 않고 단순히 다른 임포트로 이동되었습니다. 다음 대신:

```
import 'package:riverpod/riverpod.dart';
```

이제 다음을 사용해야 합니다:

```
import 'package:riverpod/legacy.dart';
```

이는 이러한 Provider가 더 이상 권장되지 않음을 강조하기 위함입니다.
동시에 이들은 하위 호환성을 위해 보존됩니다.

AutoDispose 인터페이스가 제거됨

AudoDispose 인터페이스가 제거된거지, 기능이 제거된 것은 아닙니다. 2.0에서 모든 Provider, Ref, Notifier는 auto-dispose를 위해 복제되었습니다(`Ref` vs `AutoDisposeRef`, `Notifier` vs `AutoDisposeNotifier` 등). 이는 일부 엣지 케이스에서 컴파일 오류를 위한 것이었지만, 나쁜 API라는 대가를 치뤘습니다.

3.0에서는 인터페이스가 통합되고, 이전 컴파일 오류는 이제 린트 규칙(riverpod_lint 사용)으로 구현됩니다. 구체적으로 말하면 `AutoDisposeNotifier`의 모든 참조를 `Notifier`로 교체할 수 있습니다. 코드의 동작은 변경되지 않습니다.

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

- class MyNotifier extends AutoDisposeNotifier<int> {
+ class MyNotifier extends Notifier<int> {
}
```

"FamilyNotifier"와 "Notifier" 통합

이전 점과 마찬가지로, `FamilyNotifier`와 `Notifier` 인터페이스가 이제 통합되었습니다.

간단히 말하면, 아래 코드 대신:

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

class CounterNotifier extends FamilyNotifier<int, Argument> {
  @override
  int build(Argument arg) => 0;
}
```

이제 아래 코드와 같이 사용 합니다:

```
final provider = NotifierProvider.family<CounterNotifier, int, Argument>(
  CounterNotifier.new,
);

class CounterNotifier extends Notifier<int> {
  CounterNotifier(this.arg);
  final Argument arg;
  @override
  int build() => 0;
}
```

이는 `Notifier` + `FamilyNotifier` + `AutoDisposeNotifier` + `AutoDisposeFamilyNotifier` 대신, 항상 `Notifier` 클래스를 사용한다는 의미입니다.

이 변경사항은 코드 생성에 영향을 주지 않습니다.

모두를 지배하는 하나의 `Ref`

Riverpod 2.0에서 각 Provider는 자체 Ref 서브클래스(`FutureProviderRef`, `StreamProviderRef` 등)와 함께 제공되었습니다. 일부 `Ref`에는 `state` 속성이, 일부에는 `future` 또는 `notifier` 등이 있었습니다. 유용했지만, 얻는 것에 비해 많은 복잡성이었습니다. 그 이유 중 하나는 Notifier가 이미 가지고 있던 추가 속성들이 있어서 인터페이스가 중복되었기 때문입니다.

3.0에서는 `Ref`가 통합되었습니다. `Ref<T>`와 같은 제네릭 매개변수나, `FutureProviderRef`는 더 이상 없습니다. 하나만 있습니다: `Ref`. 실제로 이는 생성된 Provider의 구문이 단순화됨을 의미합니다:

```
-Example example(ExampleRef ref) {
+Example example(Ref ref) {
  return Example();
}
```

> **정보**
>
> 이는 그대로 유지되는 WidgetRef에 관한 것이 아닙니다. Ref와 WidgetRef는 두 가지 다른 것입니다.

모든 `updateShouldNotify`가 이제 `==`를 사용함

`updateShouldNotify`는 상태 변경이 발생했을 때 Provider가 리스너에게 알림을 보낼지 결정하는 데 사용되는 메서드입니다. 하지만 2.0에서는 이 메서드의 구현이 Provider마다 상당히 달랐습니다. 일부 Provider는 `==`를, 일부는 `identical`을, 일부는 더 복잡한 로직을 사용했습니다.

3.0부터 모든 Provider는 알림을 필터링하기 위해 `==`를 사용합니다. 이는 몇 가지 방식으로 영향을 줄 수 있습니다:

- 일부 Provider가 특정 상황에서 더 이상 리스너에게 알림을 보내지 않을 수 있습니다.
- 일부 리스너가 이전보다 더 자주 알림을 받을 수 있습니다.
- `==`를 오버라이드하는 큰 데이터 클래스가 있다면 약간의 성능 영향을 볼 수 있습니다.

이러한 변경사항에 영향을 받는다면, `updateShouldNotify`를 오버라이드하여 사용자 정의 구현을 사용할 수 있습니다:

riverpod riverpod_generator

```dart
class TodoList extends Notifier<List<Todo>> {
  @override
  List<Todo> build() => [];

  @override
  bool updateShouldNotify(List<Todo> previous, List<Todo> next) {
    // 사용자 정의 구현
    return true;
  }
}
```

Provider 생명주기 변경

Provider 읽기가 예외를 발생시킬 때, 오류가 이제 ProviderException으로 래핑

이전에는 Provider가 오류를 던지면, Riverpod가 때때로 그 오류를 직접 다시 던졌습니다:

riverpod riverpod_generator

```dart
final exampleProvider = FutureProvider<int>((ref) async {
  throw StateError('오류');
});

// ...
ElevatedButton(
  onPressed: () async {
    // 이것은 StateError를 다시 던집니다
    ref.read(exampleProvider).requireValue;

    // 이것도 StateError를 다시 던집니다
    await ref.read(exampleProvider.future);
  },
```

```
  child: Text('클릭하세요'),
);
```

3.0에서는 이것이 변경되었습니다. 대신, 오류는 원본 오류와 스택 추적을 모두 포함하는 `ProviderException`에 캡슐화됩니다.

> **정보**
>
> `AsyncValue.error`, `ref.listen(..., onError: ...)`와 ProviderObserver는 이 변경사항의 영향을 받지 않으며, 여전히 변경되지 않은 오류를 받습니다.

이는 여러 이점이 있습니다:

- 훨씬 더 나은 스택 추적으로 디버깅이 개선됩니다
- 이제 Provider가 실패했는지, 아니면 실패한 다른 Provider에 의존하고 있는 오류 상태인지 판단할 수 있습니다.

예를 들어, ProviderObserver는 이를 사용하여 같은 오류를 두 번 로깅하지 않을 수 있습니다:

```
class MyObserver extends ProviderObserver {
  @override
  void providerDidFail(ProviderObserverContext context, Object error,
StackTrace stackTrace) {
    if (error is ProviderException) {
      // Provider가 직접 실패한 것이 아니라, 실패한 Provider에 의존하고 있습니다.
      // 따라서 오류가 이미 로깅되었습니다.
      return;
    }

    // 오류를 로그합니다
    print('Provider failed: $error');
  }
}
```

이는 Riverpod의 자동 재시도 메커니즘에서 내부적으로 사용됩니다. 기본 자동 재시도는 `ProviderException`을 무시합니다:

```
ProviderContainer(
  // 기본 재시도 동작의 예
```

```
  retry: (retryCount, error) {
    if (error is ProviderException) return null;

    // ...
  },
);
```

보이지 않는 위젯 내의 리스너 일시정지

Riverpod에 리스너를 일시정지하는 방법이 있으므로, Riverpod는 이를 사용하여 위젯이 보이지 않을 때 리스너를 기본적으로 일시정지합니다. 이는 보이는 위젯 트리에서 사용되지 않는 Provider가 일시정지됨을 의미합니다.

구체적인 예로, 두 경로가 있는 애플리케이션을 생각해보세요:

- Provider를 사용해 웹소켓을 수신하는 홈 페이지
- 해당 웹소켓에 의존하지 않는 설정 페이지

일반적인 애플리케이션에서 사용자는 먼저 홈 페이지를 열고 *그 다음* 설정 페이지를 엽니다. 이는 설정 페이지가 열려 있는 동안 홈페이지도 열려 있지만 보이지 않음을 의미합니다.

2.0에서는 홈페이지가 웹소켓을 계속 수신했을 것입니다. 3.0에서는 웹소켓 Provider가 대신 일시정지되어, 리소스를 절약할 수 있습니다.

어떻게 작동하는가:
Riverpod는 TickerMode에 의존하여 위젯이 보이는지 여부를 판단합니다. 그리고 false일 때, Consumer의 모든 리스너가 일시정지됩니다.

또한 TickerMode를 직접 사용하여 소비자의 일시정지 동작을 수동으로 제어할 수 있습니다. 값을 자의적으로 true/false로 설정하여 강제로 재개/일시정지할 수 있습니다:

```
class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return TickerMode(
      enabled: false, // 리스너를 일시정지합니다
      child: Consumer(
        builder: (context, ref, child) {
          // 이 "watch"는 TickerMode가
          // true로 설정될 때까지 일시정지됩니다
          final value = ref.watch(myProvider);
```

```
          return Text(value.toString());
        },
      ),
    );
  }
}
```

Provider가 일시정지된 다른 Provider만 사용되는 경우, 함께 일시정지

Riverpod 2.0에는 이미 어느 정도 일시정지/재개 지원이 있었습니다. 하지만 제한적이었고 일부 엣지 케이스를 다루지 못했습니다.
다음을 생각해보세요:

riverpod **riverpod_generator**

```
final exampleProvider = Provider<int>((ref) {
  ref.onCancel(() => print('일시정지됨'));
  ref.onResume(() => print('재개됨'));
  return 0;
});
```

2.0에서 이 Provider에 대해 `ref.read`를 한 번 호출하면, Provider의 상태는 유지되지만 '일시정지됨'이 출력됩니다. 이는 `ref.read`가 Provider를 "수신"하지 않기 때문입니다. 그리고 Provider가 "수신"되지 않으면 일시정지됩니다.

이는 현재 사용되지 않는 Provider를 일시정지하는 데 유용합니다! 문제는 많은 경우에 이 최적화가 작동하지 않는다는 것입니다. 예를 들어, Provider가 다른 Provider를 통해 간접적으로 사용될 수 있습니다.

riverpod **riverpod_generator**

```
final anotherProvider = Provider<int>((ref) {
  return ref.watch(exampleProvider);
});

 class MyWidget extends ConsumerWidget {
  @override
```

```
  Widget build(BuildContext context, WidgetRef ref) {
    return Button(
      onPressed: () {
        ref.read(anotherProvider);
      },
      child: Text('클릭하세요'),
    );
  }
}
```

이 시나리오에서 버튼을 한 번 클릭하면, `anotherProvider`가 `exampleProvider`를 수신하기 시작합니다. 하지만 `anotherProvider`는 더 이상 사용되지 않고 일시정지됩니다. 그러나 `exampleProvider`는 일시정지되지 않습니다. 왜냐하면 아직 사용되고 있다고 생각하기 때문입니다. 따라서 버튼을 클릭해도 더 이상 '일시정지됨'이 출력되지 않습니다.

3.0에서는 이것이 수정되었습니다. Provider가 일시정지된 Provider에 의해서만 사용되는 경우, 그것도 일시정지됩니다.

Provider가 다시 빌드될 때, 빌드가 완료될 때 까지 이전 상태가 유지

2.0에서는 비동기 Provider와 'auto-dispose'를 결합할 때 알려진 불편함이 있었습니다. 구체적으로, 비동기 Provider가 `await` 후에 auto-dispose Provider를 감시하면, "auto dispose"가 예기치 않게 트리거될 수 있었습니다.

다음을 생각해보세요:

riverpod **riverpod_generator**

```
final autoDisposeProvider = StreamProvider.autoDispose<int>((ref) {
  ref.onDispose(() => print('해제됨'));
  ref.onCancel(() => print('일시정지됨'));
  ref.onResume(() => print('재개됨'));
  // 매초 값을 방출하는 스트림
  return Stream.periodic(Duration(seconds: 1), (i) => i);
});

final asynchronousExampleProvider = FutureProvider<int>((ref) async {
  print('비동기 간격 전');
  // Provider 내의 비동기 간격; 일반적으로 API 호출.
```

```
  // 이것은 비동기 작업이 완료되기 전에
  // "autoDispose" Provider를 해제합니다
  await null;

  print('비동기 간격 후');
  // 비동기 작업 후에
  // auto-dispose Provider를 수신합니다
  return ref.watch(autoDisposeProvider.future);
});

void main() {
  final container = ProviderContainer();
  // 이것은 매초 '해제됨'을 출력하고,
  // 계속 0을 출력합니다
  container.listen(asynchronousExampleProvider, (_, value) {
    if (value is AsyncData) print('${value.value}\n----');
  });
}
```

이것을 Dartpad에서 실행하면 다음이 출력됩니다:

```
// 첫 번째 출력
비동기 간격 전
비동기 간격 후
0
---- // 두 번째 이후 출력
일시정지됨
비동기 간격 전
해제됨 // 'autoDispose' Provider가 비동기 간격 중에 해제되었습니다!
비동기 간격 후
0
----
일시정지됨
비동기 간격 전
해제됨
비동기 간격 후
0
----
... // 매초 계속
```

보시다시피, 이것은 매초 일관되게 `0`을 출력합니다. 왜냐하면 `autoDispose` Provider가 비동기 간격 중에 반복적으로 해제되기 때문입니다. 해결 방법은 `ref.watch` 호출을 `await` 문 앞으로 이동하는 것이었습니다. 하

지만 이는 오류가 발생하기 쉽고, 직관적이지 않으며, 항상 가능하지 않았습니다.

3.0에서는 리스너의 해제를 지연시켜 이것이 수정되었습니다. Provider가 재구축될 때, 모든 리스너를 즉시 제거하는 대신, 일시정지합니다.

완전히 같은 코드가 이제 대신 다음을 출력합니다:

```
// 첫 번째 출력
비동기 간격 전
비동기 간격 후
0
----
일시정지됨
비동기 간격 전
비동기 간격 후
재개됨
1
----
일시정지됨
비동기 간격 전
비동기 간격 후
재개됨
2
----
... // 매초 계속
```

Provider의 예외가 "ProviderException"으로 다시 던져짐.

"Provider가 실패함"과 "Provider가 실패한 Provider에 의존함"을 구별하기 위해, Riverpod 3.0은 이제 예외를 원본을 포함하는 `ProviderException`으로 래핑합니다.

이는 Provider에서 오류를 잡는다면, `ProviderException`의 내용을 검사하도록 try/catch를 업데이트해야 함을 의미합니다:

```dart
try {
  ref.watch(failingProvider);
} on ProviderException catch (e) {
  switch (e.exception) {
    case SomeSpecificError():
      // 특정 오류를 처리합니다
```

```
    default:
      // 다른 오류를 처리합니다
      rethrow;
  }
}
```

새로운 테스팅 유틸리티

`ProviderContainer.test`

2.0에서 일반적인 테스트 코드는 `createContainer`라는 사용자 정의 유틸리티에 의존했습니다. 3.0에서는 이 유틸리티가 이제 Riverpod의 일부이며, `ProviderContainer.test`라고 불립니다. 새 컨테이너를 생성하고 테스트 종료 후 자동으로 해제합니다.

```
void main() {
  test('My test', () {
    final container = ProviderContainer.test();
    // 컨테이너를 사용합니다
    // ...
    // 테스트 종료 후 컨테이너가 자동으로 해제됩니다
  });
}
```

`createContainer`를 `ProviderContainer.test`로 안전하게 전역 검색-교체할 수 있습니다.

`NotifierProvider.overrideWithBuild`

이제 전체 notifier를 모킹하지 않고 `Notifier.build` 메서드만 모킹할 수 있습니다. 이는 notifier를 특정 상태로 초기화하면서도 notifier의 원래 구현을 계속 사용하고 싶을 때 유용합니다.

riverpod **riverpod_generator**

```
class MyNotifier extends Notifier<int> {
  @override
  int build() => 0;
```

```
  void increment() {
    state++;
  }
}

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

void main() {
  final container = ProviderContainer.test(
    overrides: [
      myProvider.overrideWithBuild((ref) {
        // build 메서드를 모킹하여 42에서 시작합니다.
        // "increment" 메서드는 영향을 받지 않습니다.
        return 42;
      }),
    ],
  );
}
```

Future/StreamProvider.overrideWithValue

얼마 전에 `FutureProvider.overrideWithValue`와 `StreamProvider.overrideWithValue`가 Riverpod에서 "일시적으로" 제거되었습니다. 다시 돌아왔습니다!

riverpod **riverpod_generator**

```
final myFutureProvider = FutureProvider<int>((ref) async {
  return 42;
});

void main() {
  final container = ProviderContainer.test(
    overrides: [
      // Provider를 값으로 초기화합니다.
      // 오버라이드를 변경하면 값이 업데이트됩니다.
      myFutureProvider.overrideWithValue(AsyncValue.data(42)),
    ],
  );
}
```

`WidgetTester.container`

위젯 트리에서 `ProviderContainer`에 접근하는 간단한 방법입니다.

```
void main() {
  testWidgets('can access a ProviderContainer', (tester) async {
    await tester.pumpWidget(const ProviderScope(child: MyWidget()));
    ProviderContainer container = tester.container();
  });
}
```

자세한 정보는 WidgetTester.container 확장을 참조하세요.

사용자 정의 ProviderListenable

Riverpod 3.0에서는 사용자 정의 ProviderListenable을 생성할 수 있습니다. 이는 SyncProviderTransformerMixin을 사용하여 수행할 수 있습니다.

다음 예제는 콜백이 선택된 값 대신 boolean을 반환하는 `provider.select`의 변형을 구현합니다.

```
final class Where<T> with SyncProviderTransformerMixin<T, T> {
  Where(this.source, this.where);
  @override
  final ProviderListenable<T> source;
  final bool Function(T previous, T value) where;

  @override
  ProviderTransformer<T, T> transform(
    ProviderTransformerContext<T, T> context,
  ) {
     return ProviderTransformer(
       initState: (_) => context.sourceState.requireValue,
       listener: (self, previous, next) {
         if (where(previous, next))
           self.state = next;
       },
     );
  }
}
```

```
extension<T> on ProviderListenable<T> {
  ProviderListenable<T> where(
    bool Function(T previous, T value) where,
  ) => Where<T>(this, where);
}
```

`ref.watch(provider.where((previous, value) => value > 0))`로 사용됩니다.

정적으로 안전한 스코핑 (코드 생성 전용)

riverpod_lint를 통해, Riverpod는 이제 스코핑이 잘못 사용되었을 때를 감지하는 방법을 포함합니다. 이 린트는 오버라이드가 누락되어 런타임 오류를 방지하는 것을 감지합니다.

다음을 생각해보세요:

```
// 일반적인 "스코프된 Provider"
@Riverpod(dependencies: [])
Future<int> myFutureProvider() => throw UnimplementedError();
```

이 Provider를 사용하려면 두 가지 옵션이 있습니다.
다음 옵션 중 어느 것도 사용하지 않으면, Provider는 런타임에 오류를 던집니다.

- 사용하기 전에 `ProviderScope`를 사용하여 Provider를 오버라이드합니다:

  ```
  class MyWidget extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return ProviderScope(
        overrides: [
          myFutureProvider.overrideWithValue(AsyncValue.data(42)),
        ],
        // 오버라이드된 Provider에 접근하려면 Consumer가 필요합니다
        child: Consumer(
          builder: (context, ref, child) {
            // Provider를 사용합니다
            final value = ref.watch(myFutureProvider);
            return Text(value.toString());
          },
        ),
  ```

```
    );
  }
}
```

- 스코프된 Provider를 사용하는 것에 `@Dependencies`를 지정하여 그것에 의존한다는 것을 나타냅니다.

```
@Dependencies([myFuture])
class MyWidget extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // Provider를 사용합니다
    final value = ref.watch(myFutureProvider);
    return Text(value.toString());
  }
}
```

`@Dependencies`를 지정한 후, `MyWidget`의 모든 사용은 위와 같은 두 가지 옵션이 필요합니다:

  - `MyWidget` 사용하기 전에 `ProviderScope`를 사용하여 Provider를 오버라이드하거나

```
void main() {
  runApp(
    ProviderScope(
      overrides: [
        myFutureProvider.overrideWithValue(AsyncValue.data(42)),
      ],
      child: MyWidget(),
    ),
  );
}
```

  - `MyWidget`을 사용하는 것에 `@Dependencies`를 지정하여 그것에 의존한다는 것을 나타냅니다.

```
@Dependencies([myFuture])
class MyApp extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
     // MyApp은 MyWidget을 통해 스코프된 Provider를 간접적으로 사용합니다
     return MyWidget();
  }
}
```

기타 변경사항

AsyncValue

AsyncValue는 다양한 변경사항을 받았습니다.

- 이제 "sealed"입니다. 이는 완전한 패턴 매칭을 가능하게 합니다:

```
AsyncValue<int> value;
switch (value) {
  case AsyncData():
    print('data');
  case AsyncError():
    print('error');
  case AsyncLoading():
    print('loading');
  // 기본 케이스가 필요하지 않습니다
}
```

- `valueOrNull`이 `value`로 이름이 바뀌었습니다. 기존 `value`는 오류와 관련된 동작이 이상해서 제거되었습니다. 마이그레이션하려면 `valueOrNull` → `value`로 전역 검색-교체하세요.
- `AsyncValue.isFromCache`가 추가되었습니다. 이 플래그는 오프라인 지속성을 통해 값을 얻을 때 설정됩니다. 이는 UI가 데이터베이스에서 오는 상태와 서버에서 오는 상태를 구별할 수 있게 해줍니다.
- `AsyncLoading`에서 선택적 `progress` 속성을 사용할 수 있습니다. 이는 Provider가 요청의 현재 진행률을 정의할 수 있게 해줍니다:

riverpod **riverpod_generator**

```
class MyNotifier extends AsyncNotifier<User> {
  @override
  Future<User> build() async {
    // AsyncLoading에 선택적으로 "progress"를 전달할 수 있습니다
    state = AsyncLoading(progress: .0);
    await fetchSomething();
    state = AsyncLoading(progress: 0.5);

    return User();
```

```
    }
  }
```

Ref 리스너가 구독 취소할 수 있는 함수 반환

이제 다양한 생명주기 리스너를 "구독 취소"할 수 있습니다:

riverpod riverpod_generator

```dart
final exampleProvider = FutureProvider<int>((ref) {
  // onDispose와 다른 생명주기 리스너가
  // 리스너를 제거하는 함수를 반환합니다.
  final removeListener = ref.onDispose(() => print('dispose));
  // 단순히 함수를 호출하여 리스너를 제거합니다:
  removeListener();

  // ...
});
```

리스너에 weak 설정 - auto-dispose를 방지하지 않고 Provider를 수신합니다.

`Ref.listen`을 사용할 때 선택적으로 `weak: true`를 지정할 수 있습니다:

riverpod riverpod_generator

```dart
final exampleProvider = FutureProvider<int>((ref) {
  ref.listen(
    anotherProvider,
    // 플래그를 지정합니다
    weak: true,
    (previous, next) {},
  );

  // ...
});
```

이 플래그를 지정하면 수신된 Provider가 사용되지 않으면 여전히 해제될 수 있다고 Riverpod에 알려줍니다.

이 플래그는 단일 Provider에서 여러 "진실의 원천"을 결합하는 것과 관련된 일부 틈새 사용 사례를 돕는 고급 기능입니다.

✏ 이 페이지 수정하기