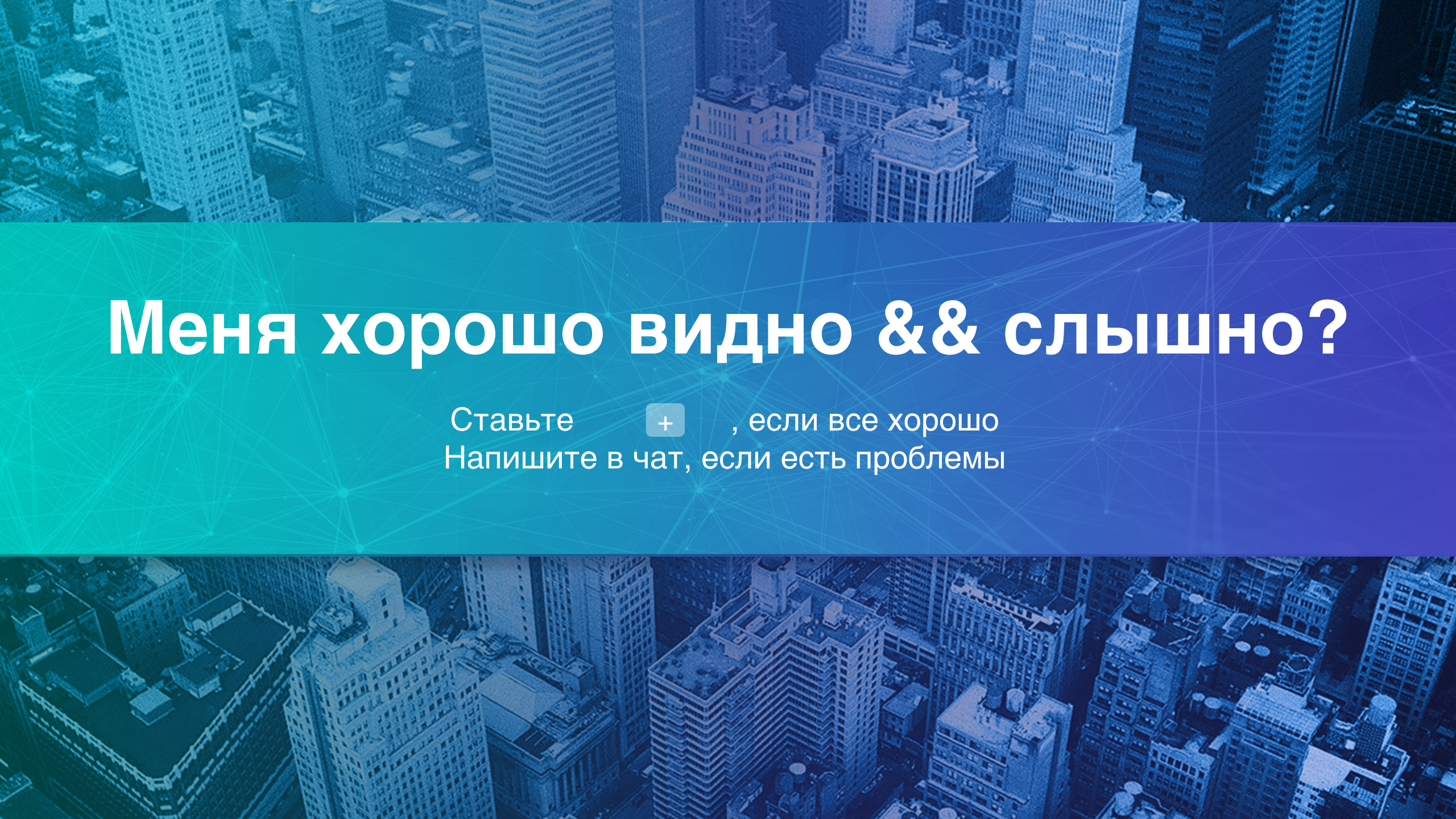
OTUS

ОНЛАЙН-ОБРАЗОВАНИЕ

Онлайн-образование

Проверить, идет ли запись!



Меня хорошо видно && слышно?

Ставьте + , если все хорошо

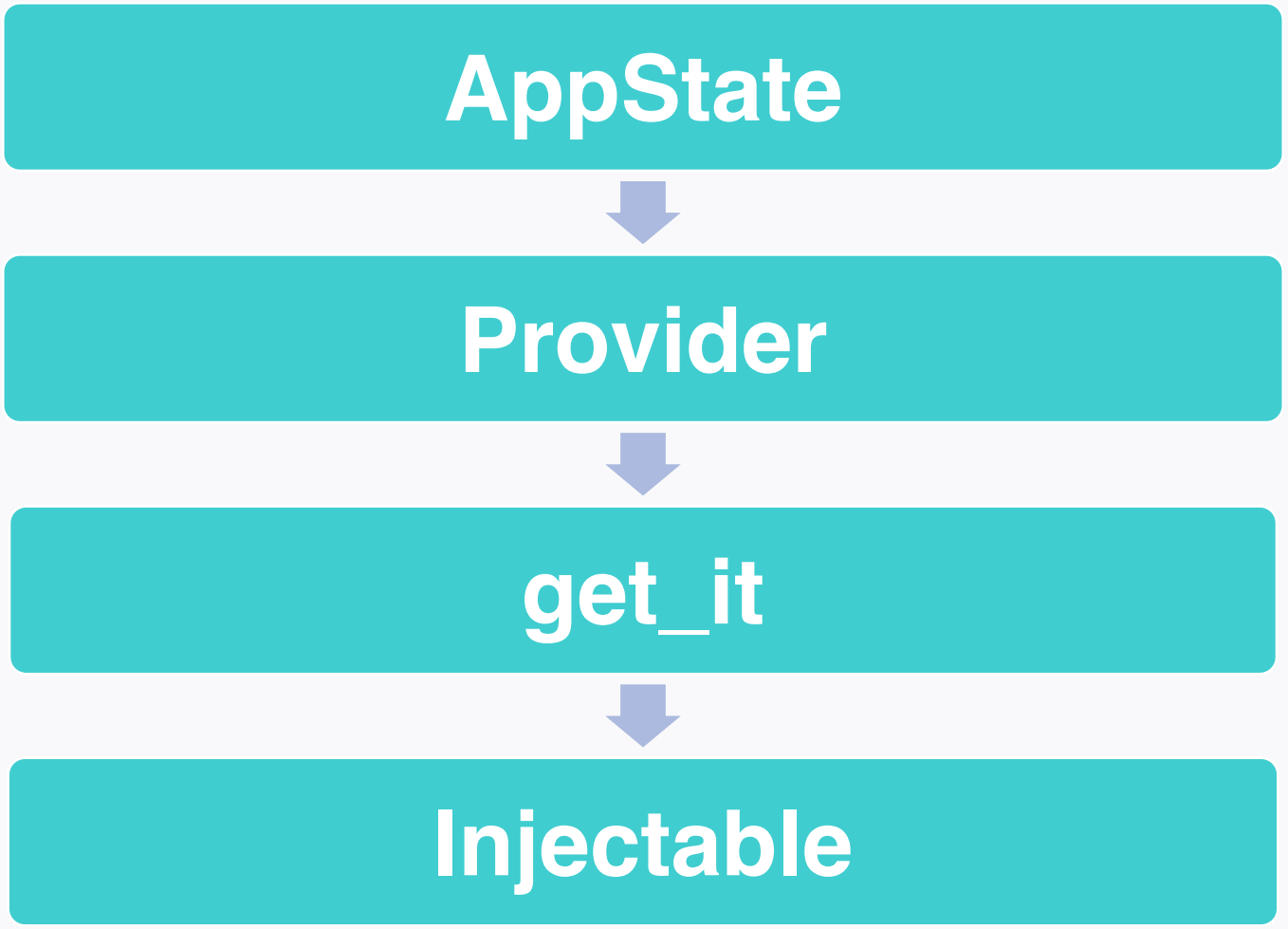
Напишите в чат, если есть проблемы

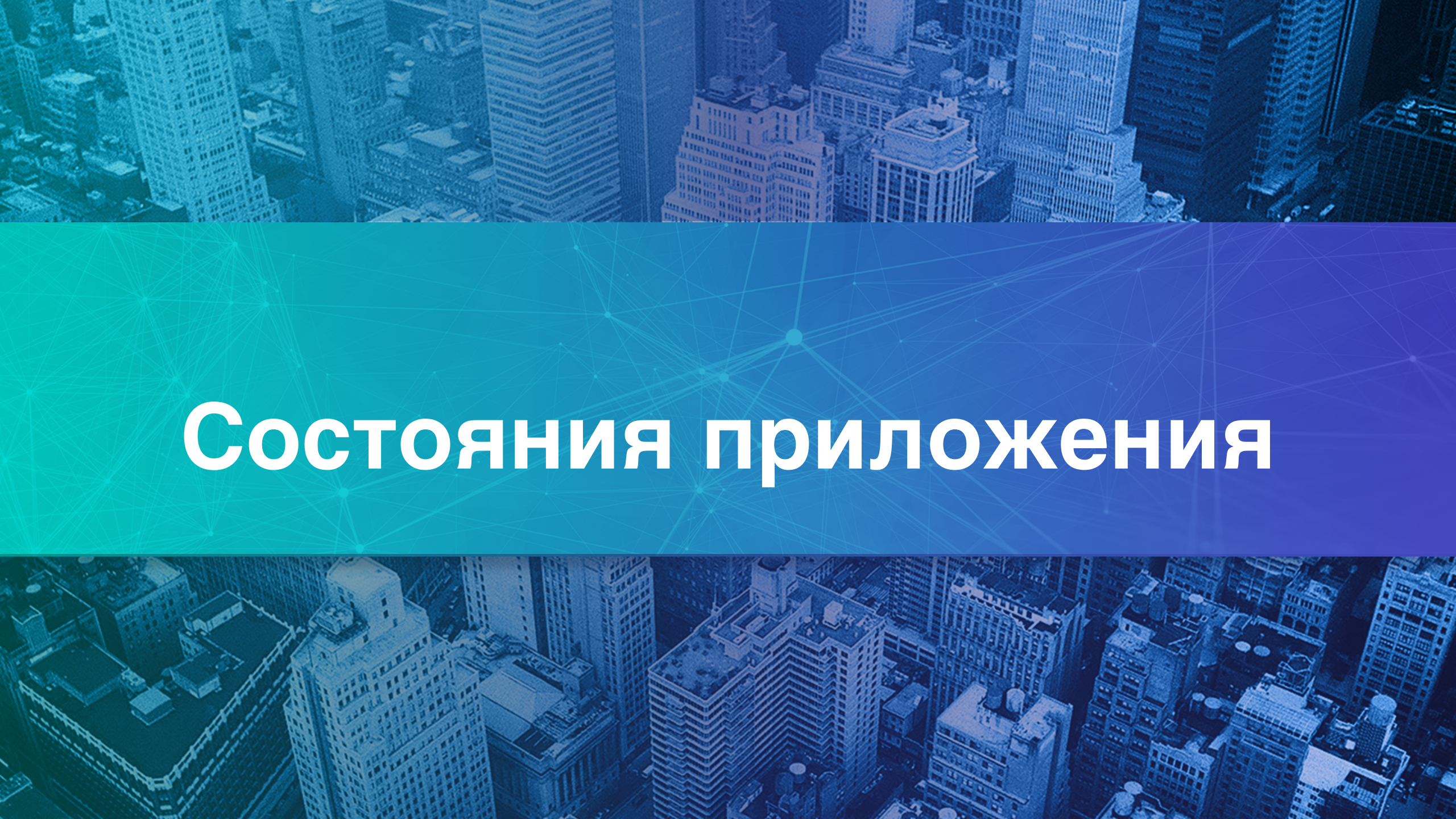
State Management + DI

Цели вебинара

1. Понять какие бывают состояния у приложения
2. Научиться работать с Provider
3. Научиться внедрять зависимости в нашем приложении с помощью get_it и injectable

Маршрут вебинара





Состояния приложения

Flutter.State

Flutter.

Полезное определение состояния - это «любые данные, которые вам нужны для отображения вашего пользовательского интерфейса в любой момент времени».

Состояние в приложении можно разделить на два типа:

Ephemeral state

App state

Flutter.

Ephemeral state

Эфемерное(Временное) состояние (иногда называемое *состоянием пользовательского интерфейса* или *локальным состоянием*) - это состояние, которое вы можете аккуратно включить в один виджет.

Это намеренно расплывчатое определение, поэтому вот несколько примеров.

- текущая страница в `PageView`
- текущий прогресс сложной анимации
- текущая выбранная вкладка в `BottomNavigationBar`

Другие части дерева виджетов редко нуждаются в доступе к этому виду состояния. Нет необходимости сохранять его, и он не меняется сложным образом.

Нет необходимости использовать инструменты управления состоянием (ScopedModel, Redux и т. Д.) Для этого типа состояния. Все, что вам нужно, это файл `StatefulWidget`.

Flutter.

App State

Состояние, которое не является временным, которое вы хотите использовать во многих частях своего приложения и которое вы хотите сохранить между пользовательскими сеансами

Примеры состояния приложения:

- Предпочтения пользователей
- Информация для входа
- Корзина покупок в приложении электронной коммерции
- Состояние прочитанных / непрочитанных статей в новостном приложении

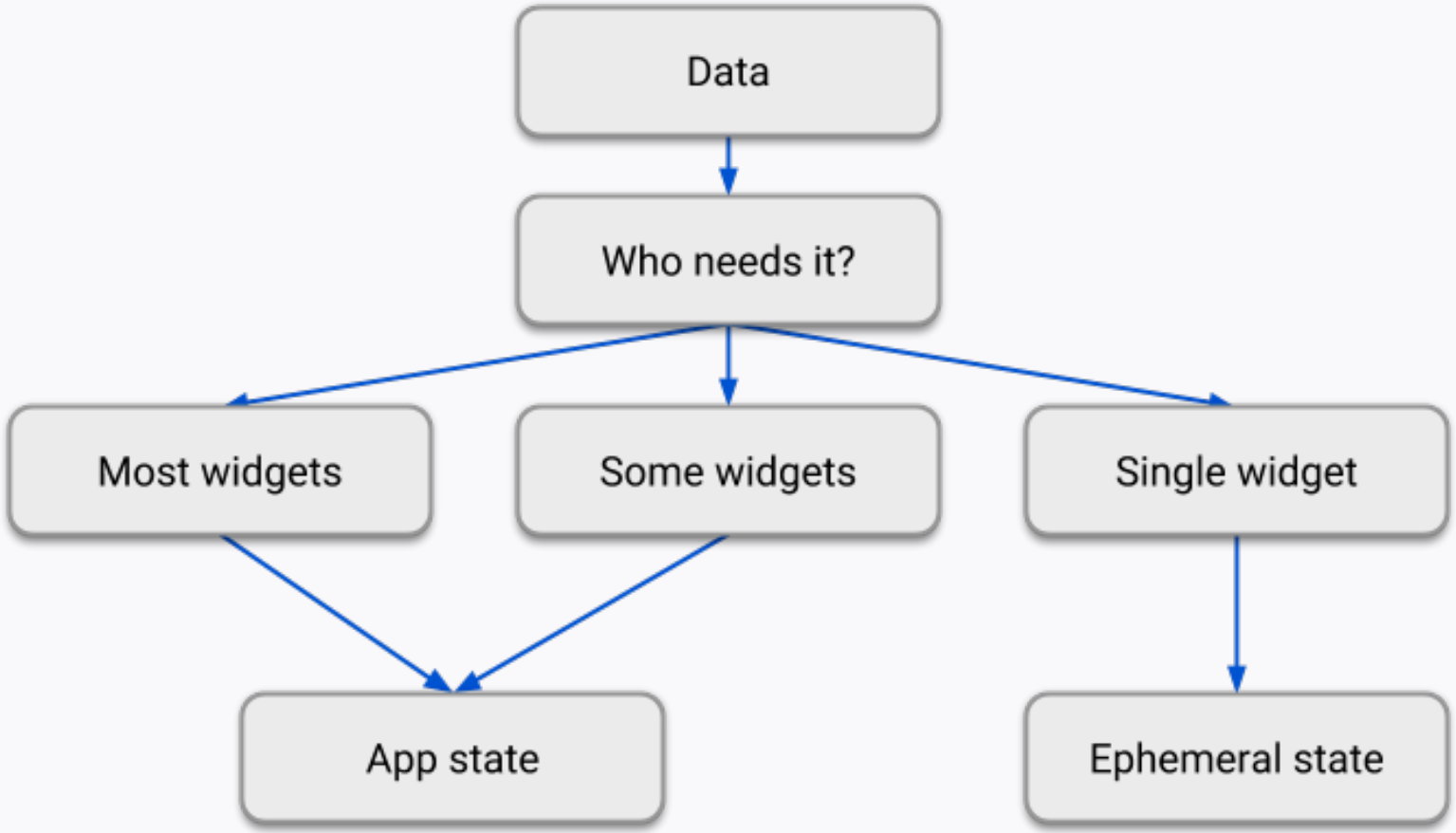
Примечание

Не существует четкого универсального правила, позволяющего различать, является ли конкретная переменная эфемерной или состоянием приложения.

Иногда вам придется преобразовывать одно в другое.

Например, вы начнете с явно недолговечного состояния, но по мере роста функций вашего приложения, возможно, потребуется перевести его в состояние приложения.

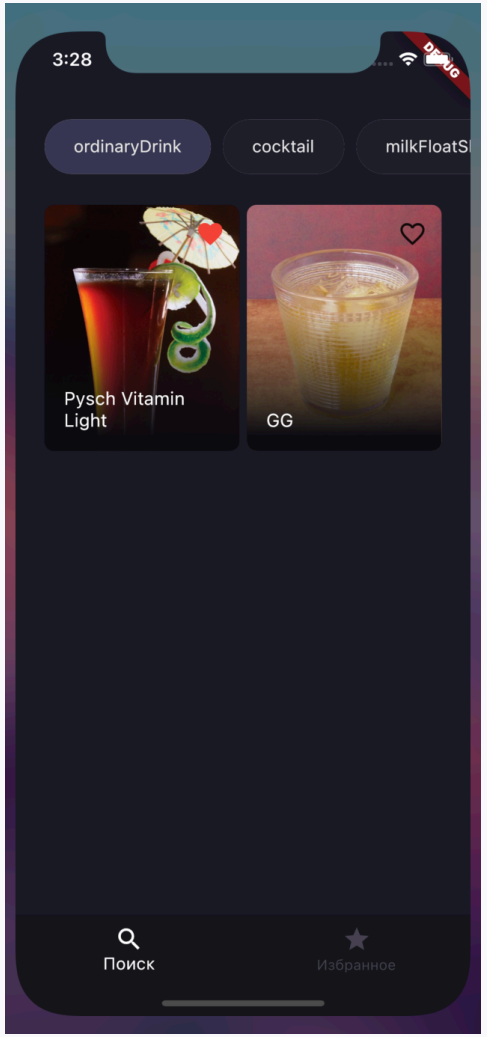
Flutter. Состояние приложения

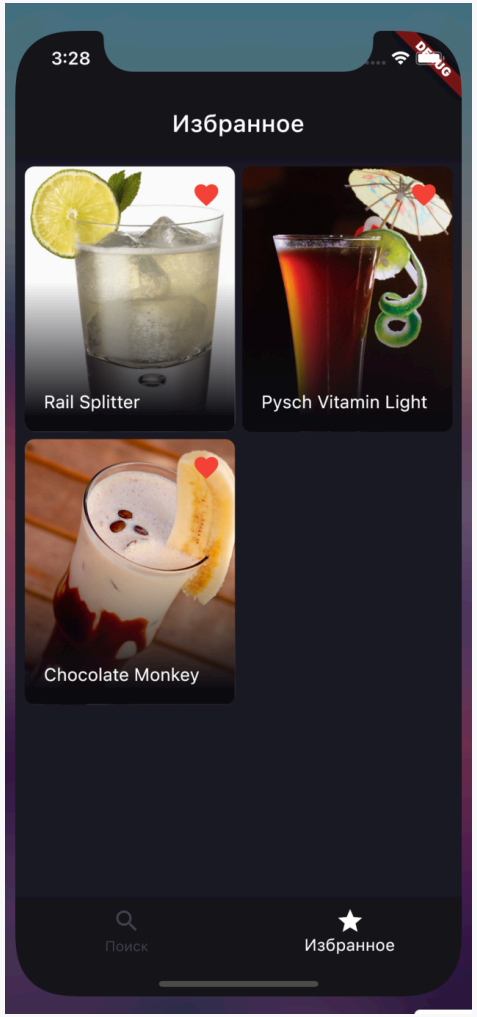


Flutter. Состояние приложения

Что из этого относится к AppState
а что к Ephemeral state ?

1. Выбранная категория коктейлей.
2. Список коктейлей.
3. Список избранных коктейлей.
4. Индикатор выбранного коктейля.





Управление состоянием приложения

Управление состоянием приложения

Существует множество библиотек которые позволяют нам работать с состоянием приложения

Provider	2866 ♥
Get	2283 ♥
Bloc	1482 ♥
MobX	468 ♥
Riverpod	346 ♥
Redux	184 ♥
ScopedModel	112 ♥

Управление состоянием приложения

В рамках нашего курса мы рассмотрим

- Provider
- Get
- Bloc
- MobX
- Riverpod
- Redux
- ScopedModel


Provider

Provider

https://pub.dev/packages/provider

`Provider` – написан в 2018 году Реми Русле, похожий на `ScopedModel`, но функции которого не ограничиваются предоставлением подкласса Model.

Он содержит в себе `InheritedWidget`, но Provider может предоставлять любые объекты состояния, в том числе, bloc, streams, futures и другие.

Google анонсировала на конференции Google I/O ’19, что в дальнейшем `Provider` будет предпочтительным пакетом для управления состоянием.

Разумеется, допускается и использование других пакетов, но, если у вас есть какие-либо сомнения, Google рекомендует остановиться на `Provider`.

Provider

Библиотека Provider предоставляет нам следующие виджеты

Поставщики

- Provider
- MultiProvider
- ProxyProvider
- StreamProvider
- FutureProvider
- ChangeNotifierProvider
- ValueListenableProvider

Потребители

- Consumer
- Selector

Provider

Provider

Действует аналогично InheritedWidget. Предоставляет модель вниз по дереву.

Методы и параметры:

create	Метод в котором создается наша текущая модель
lazy	Если true, то create вызывается только при первом обращении к модели
dispose	Вызывается когда наш виджет удален из дерева виджетов
child	Дочерний виджет
builder	Обертка для создания дочернего виджета, если нам нужен BuildContext

Получение данных от провайдера

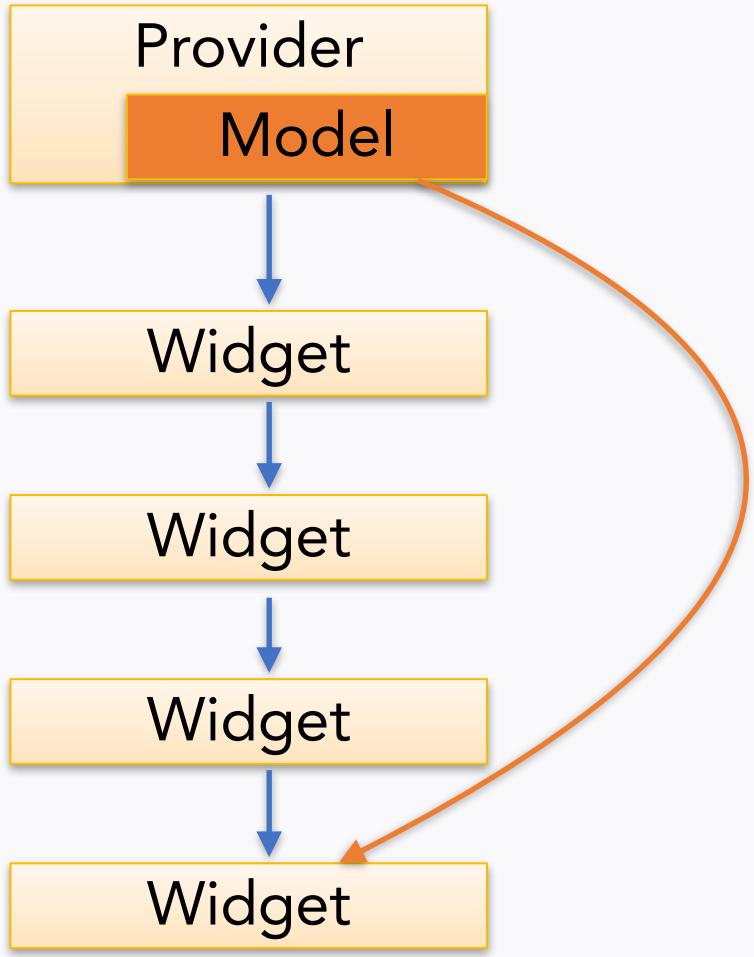
С помощью extension:

T - тип данных, которые передает провайдер

context.read<T>()

context.watch<T>()

context.select<T>()



Получение данных от провайдера

С помощью extension:

T - тип данных, которые передает провайдер

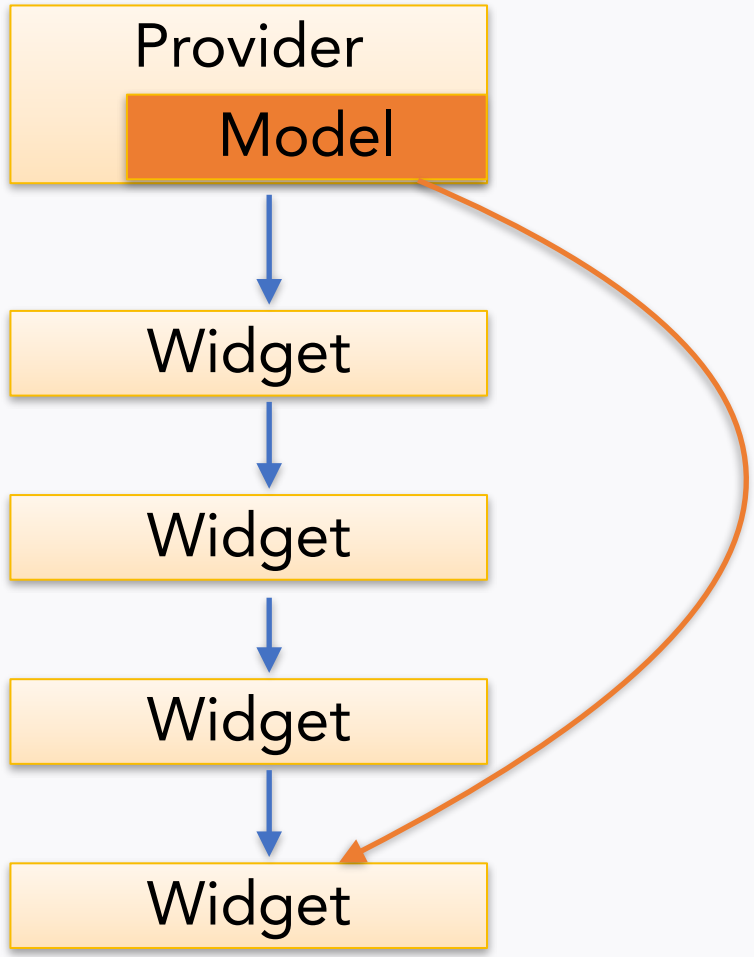
context.read<T>()

!!! read - нельзя использовать внутри build()

context.watch<T>()

!!! watch - нельзя использовать вне метода build()

context.select<T>()



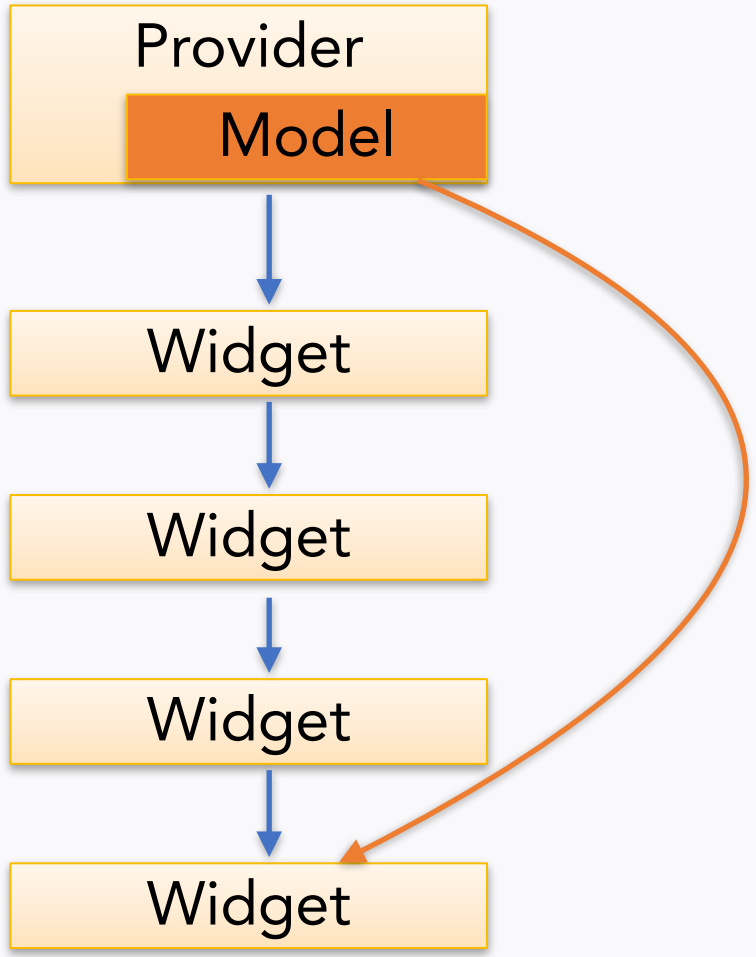
Получение данных от провайдера

С помощью метода

Provider.of<T>(context)

listenable:true - аналог context.watch()

listenable:false - аналог context.read()



Получение данных от провайдера

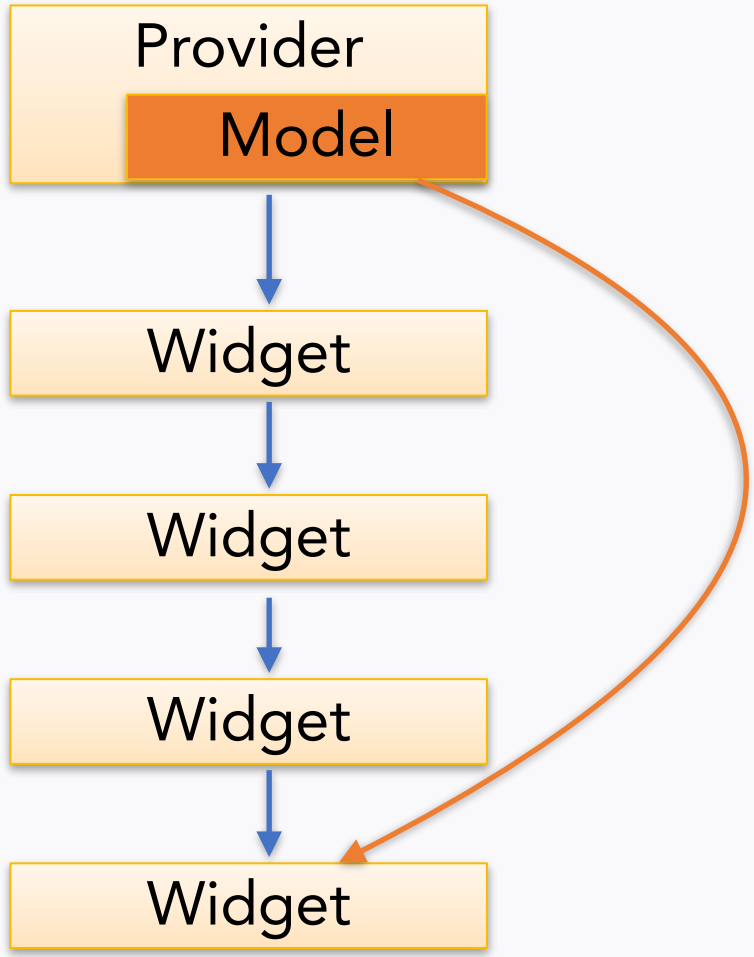
С помощью виджетов

Consumer

Consumer - обертка метода Provider.of<T>()

Selector

Selector - позволяет контролировать перестроение дочернего виджета и конвертировать полученное значение от Provider



Provider

Т.к Provider - это виджет то можно создавать множество провайдеров и вкладывать их друг в друга.

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

Provider

MultiProvider

Предоставляет множество провайдеров вниз по дереву виджетов.

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

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

Provider.

StreamProvider — Слушает [Stream] и предоставляет его содержимое дочернему элементу и потомкам.

FutureProvider — Слушает [Future] и …

ChangeNotifierProvider — Слушает [ChangeNotifier] и…

ValueListenableProvider — Слушает [ValueListenable] и…

Provider. Асинхронные провайдеры

ProxyProvider

Создает свое значение на основании другого(других) провайдеров

```
MultiProvider(
  providers: [
    Provider<Title>(create: (context) => Title('Title')),
    Provider<int>(create: (context) => 10),
    ProxyProvider2<int, Title, String>(
      update: (context, t1, t2, result) {
        return '${t2.title} $t1';
      },
    )
  ],
```

Provider.

Вопросы:

Есть Provider и есть **SomeWidget(Something something)** внизу дерева виджетов.
Как получить и передать модель Something в SomeWidget?

```
Provider<Something>(
  create: (_) => Something(),
  child: … )
```

???

```
…
Widget build(BuildContext context) {
 return SomeWidget(….)
}
…
```


Dependency Injection (DI)

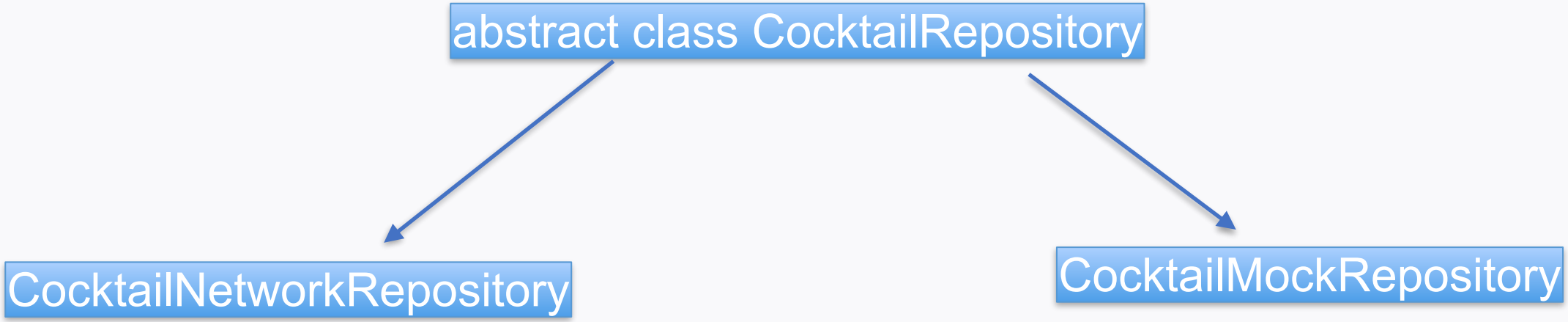
Dependency Injection

Dependency injection — процесс предоставления внешней зависимости программному компоненту.
В полном соответствии с принципом единственной ответственности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Dependency Injection

Зачем это нужно ???

Dependency Injection



Dependency Injection

Где и как CocktailInfoView может получить repository?

```
class CocktailInfoView {
  …
  void someMethod(){
    …
    final cocktail = await repository.getCocktail(id);
    …
  }
}
```

Dependency Injection

Вариант 1. Создать внутри CocktailInfoView

```
class CocktailInfoView {
  …
  void someMethod(){
    …
    CocktailRepository repository = CocktailNetworkRepository();
    final cocktail = await repository.getCocktail(id);
    …
  }
}
```

Плюсы:

- Быстро

Минусы:

- CocktailNetworkRepository нельзя заменить
- CocktailInfoView нельзя протестировать
- Нарушен принцип единственной ответственности

Dependency Injection

Вариант 1. Создать внутри CocktailInfoView

```
class CocktailInfoView {
  …
  void someMethod(){
    …
    CocktailRepository repository = CocktailNetworkRepository();
    final cocktail = await repository.getCocktail(id);
    …
  }
}
```

Плюсы:

- Быстро

Минусы:

- CocktailNetworkRepository нельзя заменить
- CocktailInfoView нельзя протестировать
- Нарушен принцип единственной ответственности

Dependency Injection

Вариант 2. Передать CocktailRepository в конструкторе

```
class CocktailInfoView {

CocktailRepository repository;

CocktailInfoView(this.repository)
  …
  void someMethod(){
    …
    final cocktail = await repository.getCocktail(id);
    …
  }
}
```

Dependency Injection

Вариант 3. Использовать ServiceLocator, InheritedWidget,Provider итд

```
class CocktailInfoView {
  …
  void someMethod(){
    …
    CocktailRepository repository = Provider.of< CocktailRepository >(context);
    или
    CocktailRepository repository =  locator.get< CocktailRepository >();
    final cocktail = await repository.getCocktail(id);
    …
  }
}
```

Dependency Injection

Вариант 3. Передать CocktailRepository в конструкторе

```
class CocktailInfoView {

CocktailRepository repository;

CocktailInfoView(this.repository)
  …
  void someMethod(){
    …
    final cocktail = await repository.getCocktail(id);
    …
  }
}
```

А) С помощью ServiceLocator(Provider) итд

Б) Создать инстанс класса


get_it

Dependency Injection. Get_it

https://pub.dev/packages/get_it

Библиотека использующая шаблон ServiceLocator.

Позволяет управлять зависимостями внутри приложения вне дерева виджетов.

Соответственно без использования BuildContext.

Dependency Injection. Get_it

Работа с ServiceLocator делится на 2 этапа

1. Регистрация объектов

2. Получение объектов

Dependency Injection. Get_it

1. Регистрация объектов

registerFactory<T>	Регистрирует «Фабрику» для класса T. Каждый запрос на получение T будет создавать новый экземпляр класса T
registerFactoryParam<T, P1,P2>	Регистрирует «Фабрику» для класса T. + Передает параметры P1 и P2 в качестве аргументов конструктора в T(P1,P2)
registerSingleton<T>	Регистрирует «Синглтон» для класса T. Каждый запрос на получение T будет возвращать один и тот же экземпляр класса T
registerLazySingleton<T>	Регистрирует «Синглтон» для класса T. Каждый запрос на получение T будет возвращать один и тот же экземпляр класса T
registerSingletonWithDependencies	Регистрирует «Синглтон» после регистрации объектов от которых он зависит

Dependency Injection. Get_it

1. Регистрация объектов (Асинхронные версии)

registerFactoryAsync<T>

registerFactoryParamAsync<T, P1,P2>

registerSingletonAsync<T>

registerLazySingletonAsync<T>

Dependency Injection. Get_it

2. Получение объектов

`get<T>(name, param1, param2)` — Синхронное получение экземпляра класса T

`getAsync<T>(name, param1, param2)` — Асинхронное получение экземпляра класса T. Возвращает Future<T>.

Как получить класс строго зависит от того каким способом он регистрировался!!!


injectable

Injectable

https://pub.dev/packages/injectable

Injectable - удобный генератор кода для get_it.

Injectable

1. Добавляем зависимости

```
dependencies:
  get_it:
  injectable:

dev_dependencies:
  injectable_generator:
```

Injectable

2. Создаем файл в котором конфигурируем injectable

```dart
final getIt = GetIt.instance;

@InjectableInit(
  initializerName: r'$initGetIt', // default
  preferRelativeImports: true, // default
  asExtension: false, // default
)
void configureDependencies() => $initGetIt(getIt);
```

Injectable

3. Добавляем аннотации.

```
@injectable
class ServiceA {}

@injectable
class ServiceB {
    ServiceB(ServiceA serviceA);
}
```

4. Запускаем build_runner

Injectable

Создание factory/singletone

```
@injectable *
class ServiceA {
  final ServiceB serviceB;

  ServiceA(this.serviceB);
}
```

Создаст

```
sl.factory<ServiceA>(() => ServiceA(get<ServiceB>()));
```

* injectable / singletone / lazySingleton

Injectable

Создание factory/singletone

```
@Injectable(as:Service, env: [Environment.prod])*
class ServiceA extends Service {

}
```

Создаст

```
sl.factory<Service>(() => ServiceA(), registerFor: {_prod});
```

as - определяет тип для регистрации

env - определяет enviroments для который будет зарегистрирован

* Injectable/Singletone/LazySingleton

Injectable

Создание asynchronous injectables

```
@injectable
class ApiClient {
  final String baseUrl;
  ApiClient._(this.baseUrl);

  @factoryMethod
  static Future<ApiClient> create(Config config) async {
    ...
    return ApiClient._(config.baseUrl);
  }
}
```

Создаст

```
sl.factoryAsync<ApiClient>(() => ApiClient.create(get<Config>()));
```

Injectable

Использование factoryParam

```
@injectable
class ApiClient {
  ApiClient(@factoryParam String baseUrl);
}
```

Создаст

```
gh.factoryParam<ApiClient, String, dynamic>(
    (baseUrl, _) => ApiClient(baseUrl));
```

При получении необходимо передать param

```
final api = GetIt./.get<ApiClient>(param1: 'url');
```

Injectable

Module

Позволяет группировать регистрируемые объекты и

```
@module
abstract class RegisterModule {
  final Service service = ServiceA();
  Future<SharedPreferences> get prefs =>
                    SharedPreferences.getInstance();
}
```

```
final registerModule = _$RegisterModule();
gh.factory<Service>(() => registerModule.service);

gh.factoryAsync<SharedPreferences>(() => registerModule.prefs);
```

Заполните, пожалуйста,
опрос о занятии по ссылке в чате