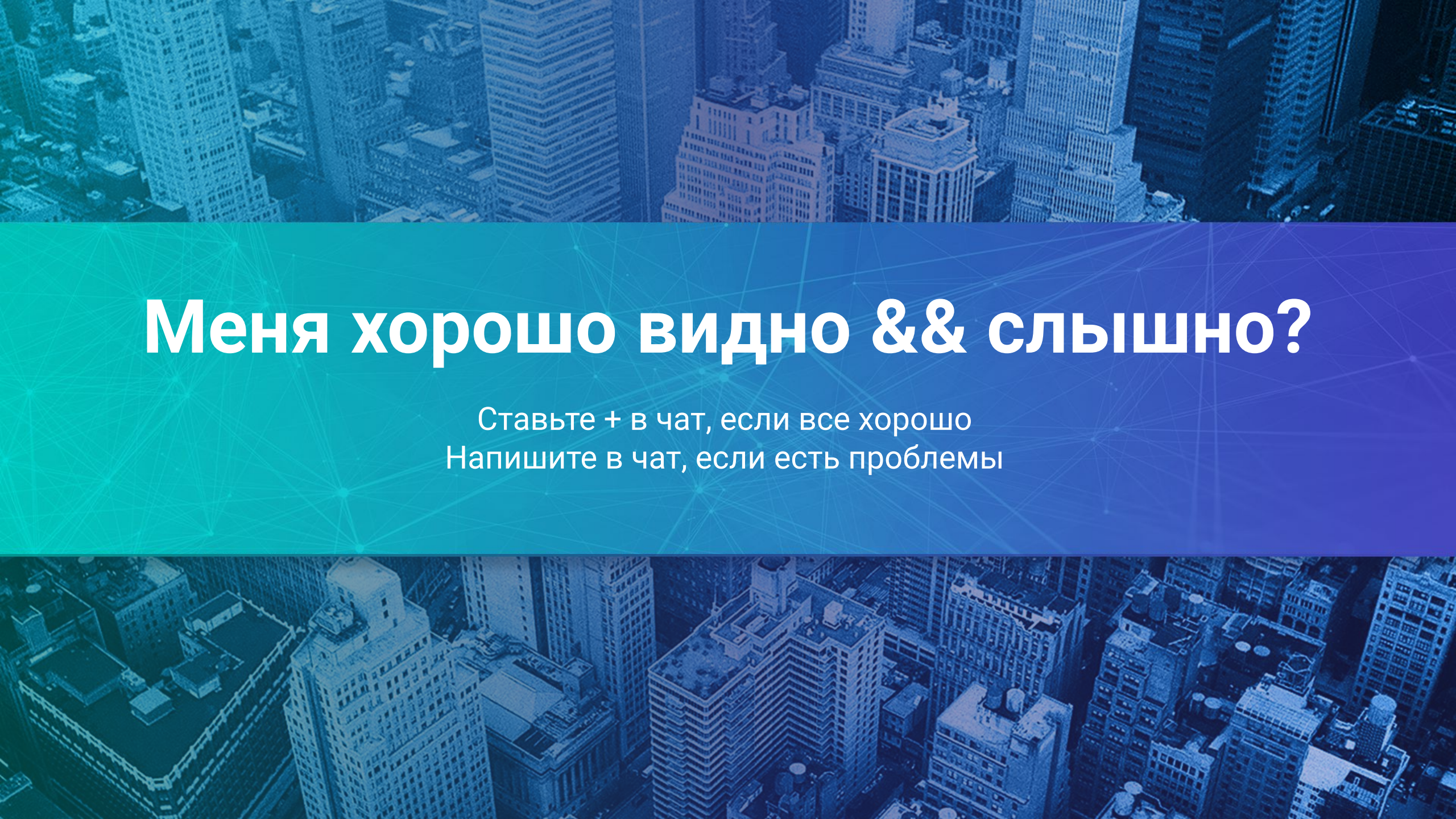


Меня хорошо видно && слышно?

Ставьте + в чат, если все хорошо
Напишите в чат, если есть проблемы

Проверить, идет ли запись!

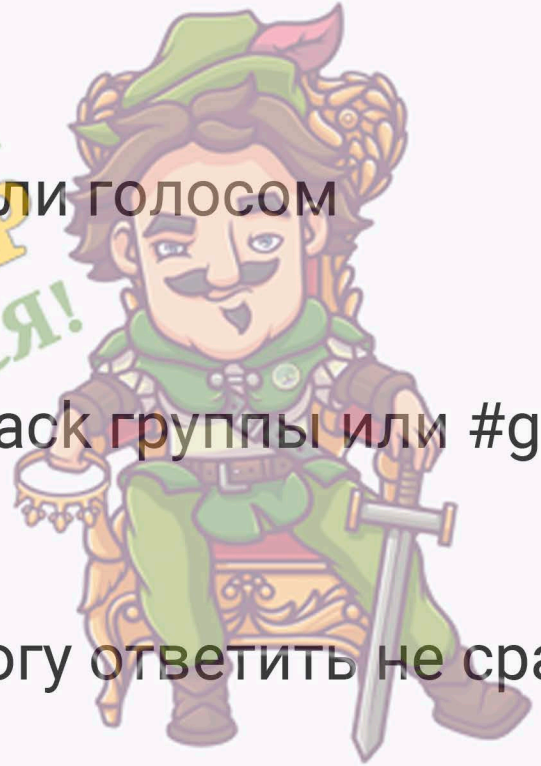
Правила вебинара

Активно участвуем

Задаем вопросы в чат или голосом

Off-topic обсуждаем в Slack группы или #general

Вопросы вижу в чате, могу ответить не сразу



Обзор MobX

Всеволод Краснов
Flutter Developer
t.me/@vs_krasnov

Преподаватель

Всеволод Краснов

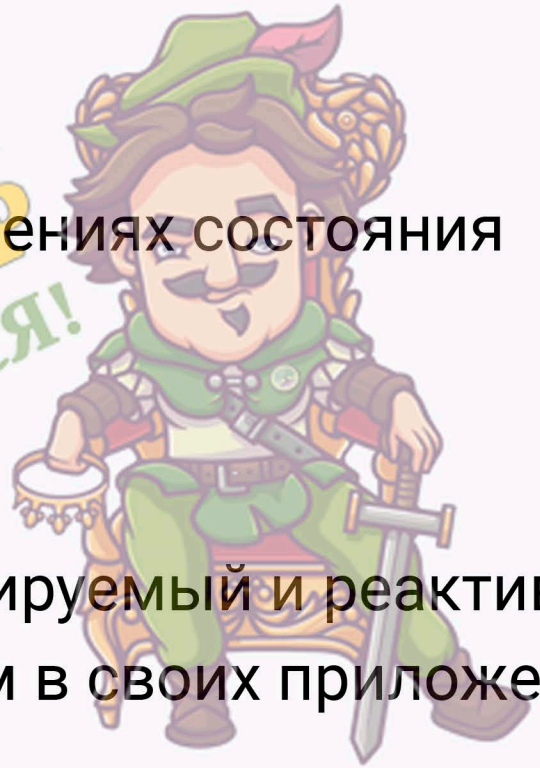
Flutter Developer @ Supernova.io

- 8 лет в ИТ
- Два года с Flutter
- Опыт мобильной и веб-разработки на Flutter

Цель вебинара | После занятия вы сможете

1. Хранить и обновлять состояние в MobX Store

2. Обновлять UI при изменениях состояния



Смысл:

Использовать масштабируемый и реактивный подход к управлению состоянием в своих приложениях

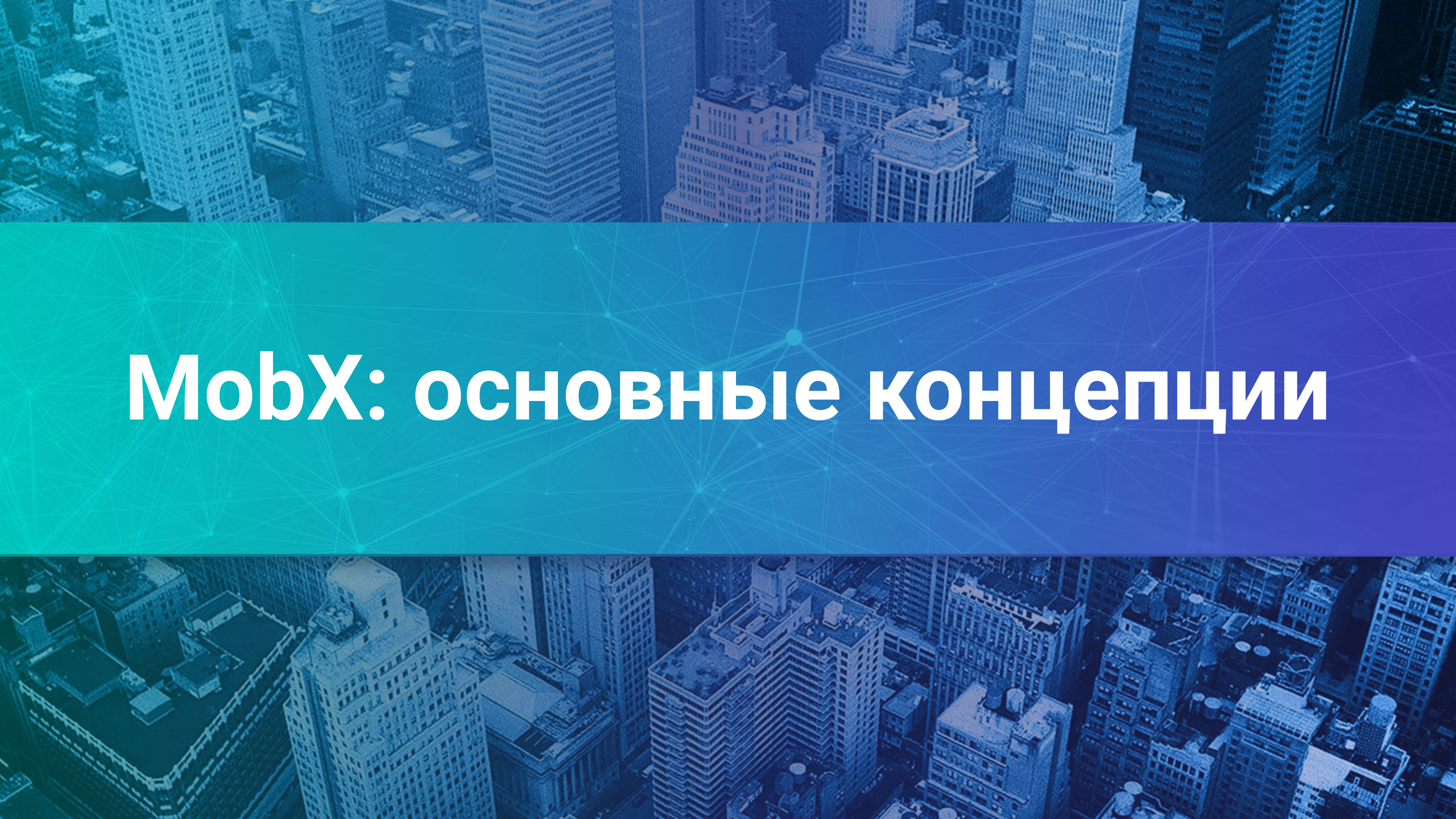
Маршрут вебинара

MobX: основные концепции

Best practices

Практика

Рефлексия



MobX: основные концепции

MobX

- Библиотека управления состоянием
- Реактивный подход
- Порт JS-библиотеки

MobX

Chris Sells @csells · 29 Jun 2019

If you haven't seen MobX.dart @ mobx.pub, check it out. In combination with with Provider, it's PFM (Pure Flutter Magic :). I use it when I build anything real. #recommended #Flutter

1 8 19

"It's like BLoC, but actually works."

Remi Rousselet

Flutter enthusiast, creator of the flutter_hooks, provider packages.

Star 1,719

MobX: зависимости

pubspec.yaml

```
dependencies:
 mobx: ^1.2.1+4
 flutter_mobx: ^1.1.0+2

dev_dependencies:
 build_runner: ^1.10.4
 mobx_codegen: ^1.1.1+3
```

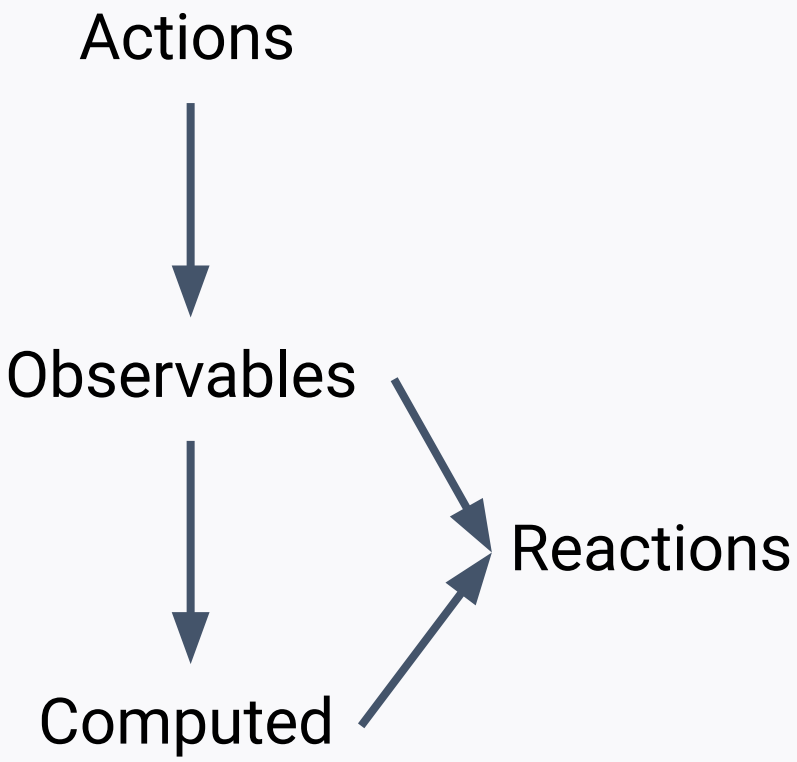
Пример

Store

- Хранилище состояния
- Операции управления состоянием

```
abstract class CalculatorStore with Store {

}
```

Процесс



Observable

- Основное состояние (core state)
- Уведомляет слушателей об изменениях

```
final observable = Observable<String>("Initial value");

observable.value = "New value";
```

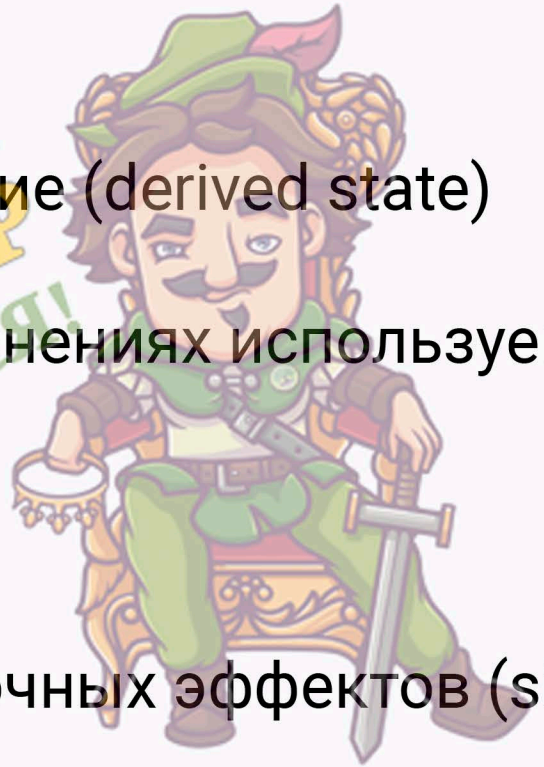
Observable

- `Observable<T>`
- `ObservableList<T>`
- `ObservableMap<K, V>`
- `ObservableSet<T>`
- `ObservableFuture<T>`
- `ObservableStream<T>`

Observable

```
abstract class CalculatorStore with Store {

 final firstNumber = Observable(0);

 final secondNumber = Observable(0);

}
```

Computed



- Производное состояние (derived state)
- Обновляется при изменениях используемых `Observable/Computed`
- Геттер
- Не должен иметь побочных эффектов (side effects)
- Кэширует результат

Computed

```
abstract class CalculatorStore with Store {
 final firstNumber = Observable(0);
 final secondNumber = Observable(0);

 @computed
 int get result => firstNumber.value + secondNumber.value;
}
```

Action

- Операция обновления основного состояния
- Метод с аннотацией `@action`
- Может быть асинхронным

Action

```
abstract class CalculatorStore with Store {
 // Observables and computed

 @action
 void setFirstNumber(int value) {
   firstNumber.value = value;
 }

 @action
 void setSecondNumber(int value) {
   secondNumber.value = value;
 }
}
```

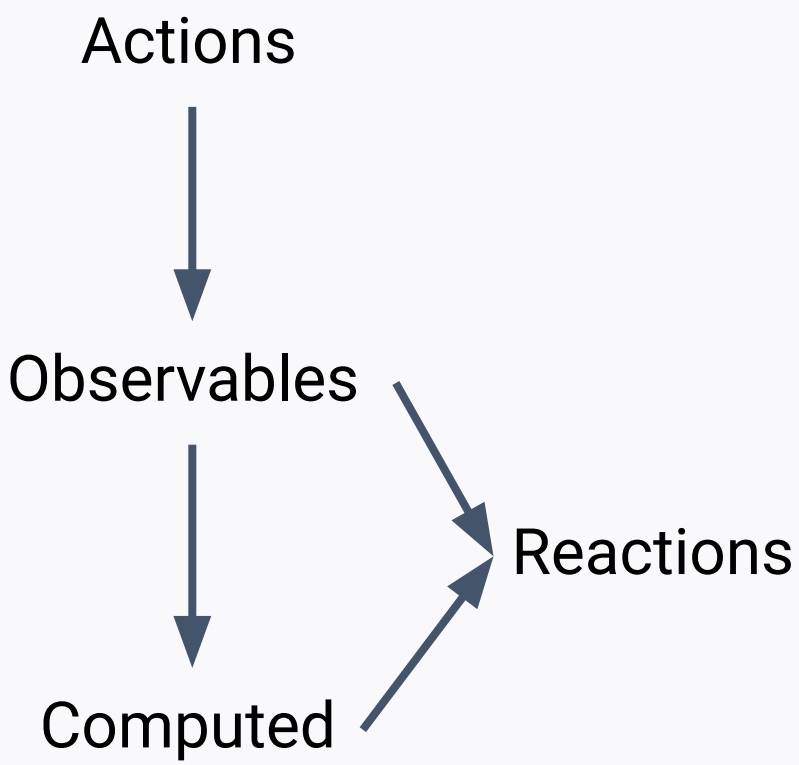
Reaction

- Действие на обновление состояния
- Слушает и основное, и производное состояние
- Может создавать побочные эффекты (side effects)
- Самый частый кейс - обновление UI

Reaction: виджет Observer

```
Observer(
 builder: (context) {
   return Text(
     "Result: ${store.result}",
   );
 },
),
```

Процесс



Action - операция обновления состояния

Observable - основное состояние

Computed - производное состояние

Reaction - действие на обновление состояния

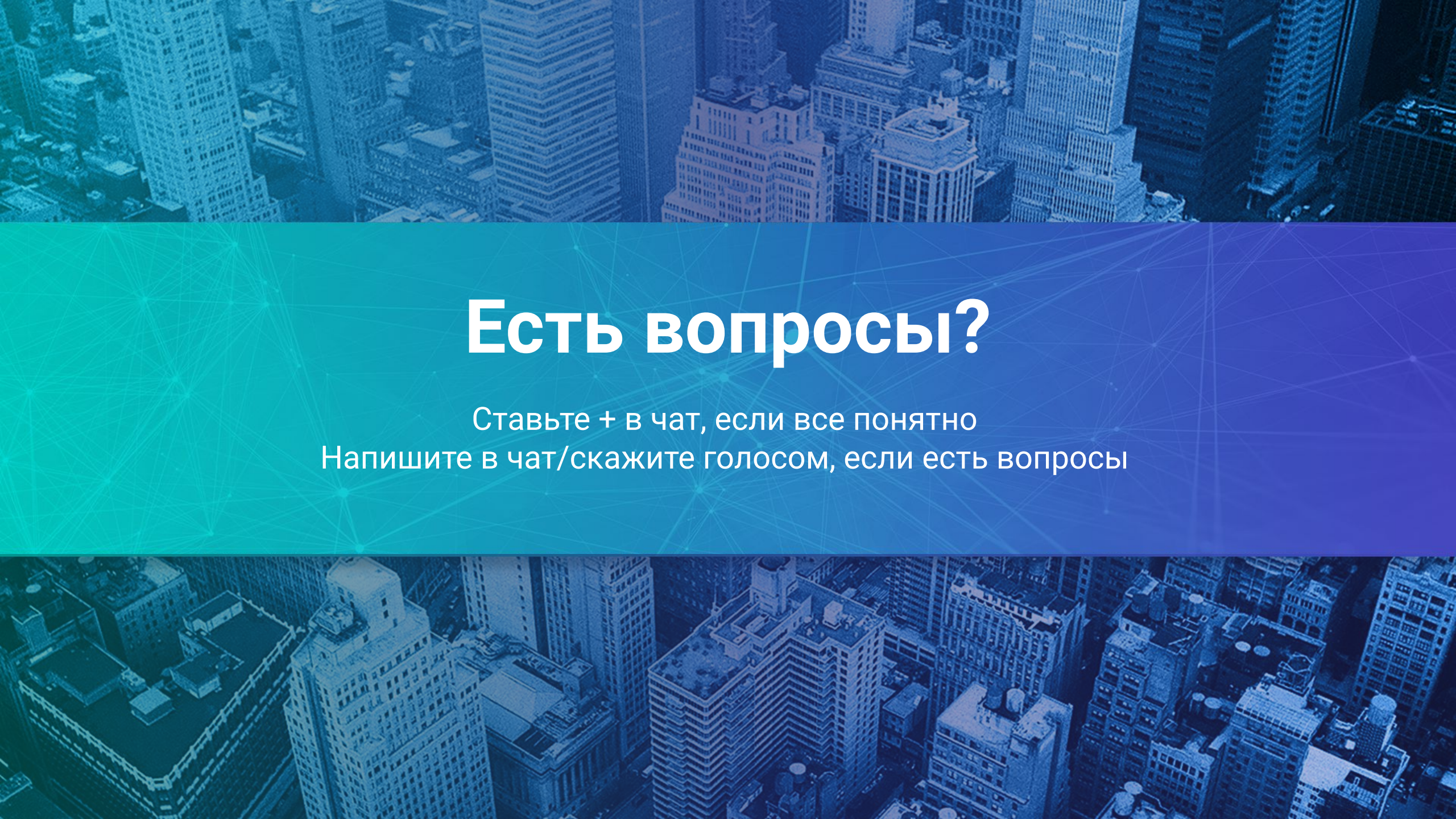
Codegen

```dart
part 'calculator_store.g.dart';

class _CalculatorStoreImpl extends CalculatorStore
with _$_CalculatorStoreImpl {}
```

```
% flutter pub run build_runner build --delete-conflicting-outputs

% flutter pub run build_runner watch --delete-conflicting-outputs
```

Если сгенерированные
файлы в репозитории

Пример



Есть вопросы?

Ставьте + в чат, если все понятно
Напишите в чат/скажите голосом, если есть вопросы

Задание

1. Забрать проект
2. Исправить ошибки в реализации MobX
3. Убедиться, что при нажатии на кнопку курсы обновляются и среднее значение пересчитывается

10:00

GitHub

готовы

Задание: решение

- Отсутствует `Observer`
- Отсутствует аннотация `@action`

Custom reactions

- Можно создавать собственные Reactions
- Reactions могут вызывать другие actions
- Требуют ручной `dispose`

Custom reactions

```
ReactionDisposer autorun(Function(Reaction) fn)

ReactionDisposer reaction<T>(T Function(Reaction) fn, void Function(T) effect)

ReactionDisposer when(bool Function(Reaction) predicate, void Function() effect)

Future<void> asyncWhen(bool Function(Reaction) predicate)
```

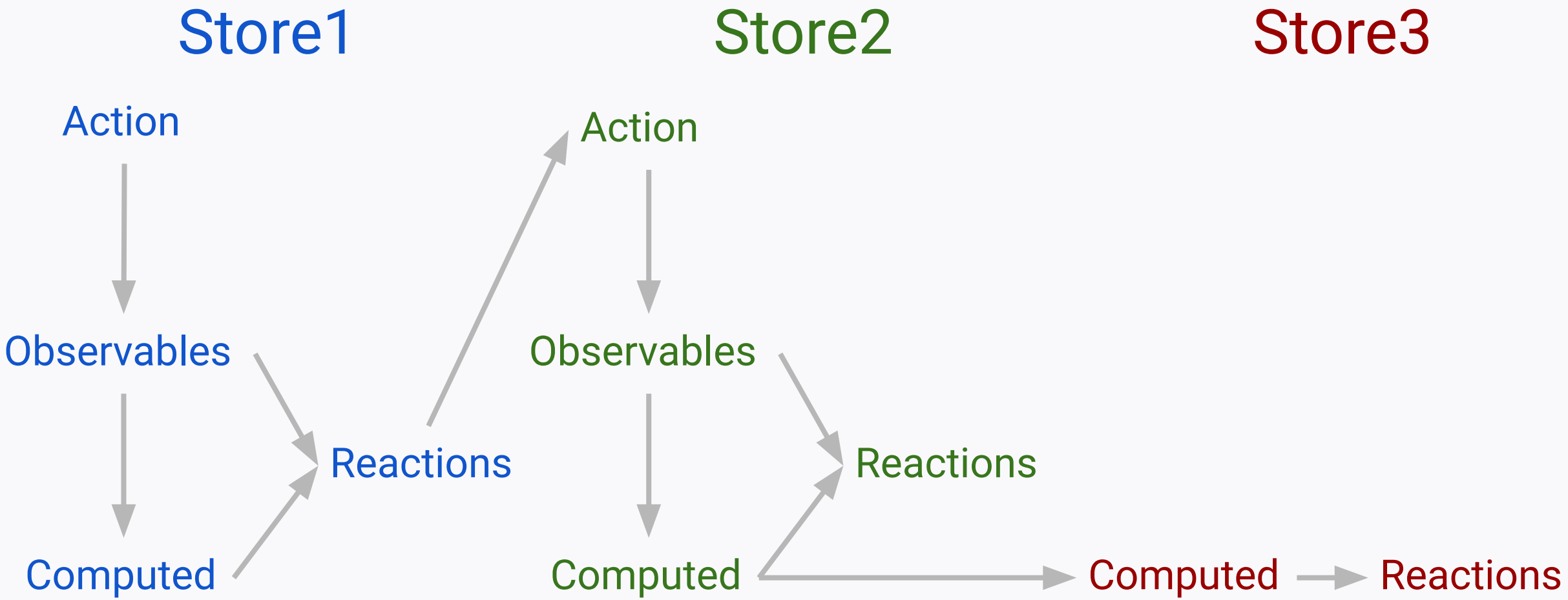
Пример

Best practices

Store: разделение ответственности

- Множество Store-ов в одном приложении
- Store-ы могут иметь разный уровень в дереве виджетов
- Каждый store отвечает за свою часть состояния

Store: цепочка обновлений

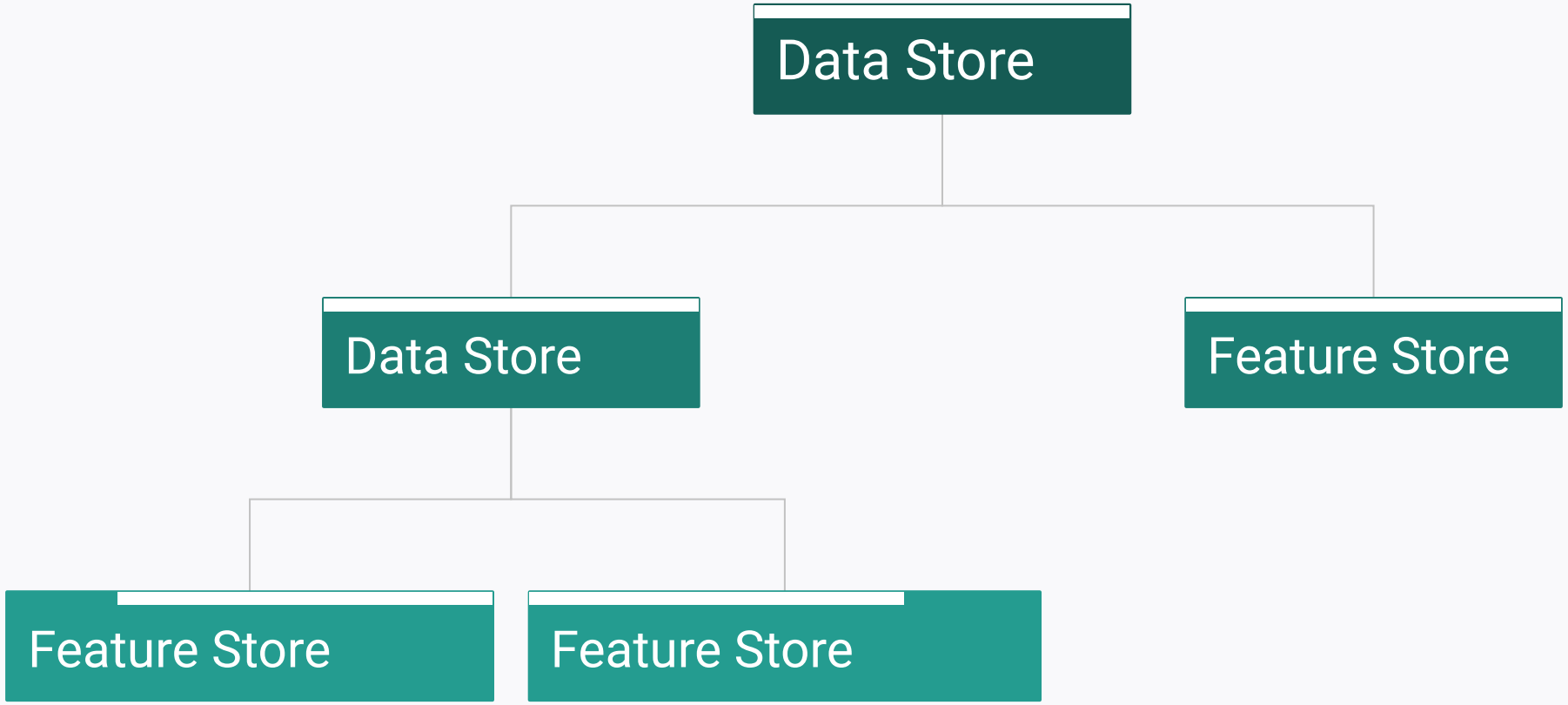


Store: агрегация

- Внутри одного Store можно слушать состояние другого
- В Reactions одного Store можно вызывать Actions другого
- Агрегация для связи Store-ов

Пример

Store: агрегация



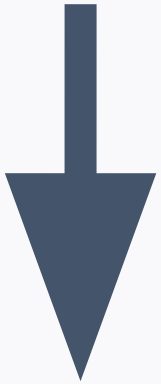
Store: управление жизненным циклом

- MobX + Provider = ❤️
- Provider:
  - передает Store по дереву виджетов
  - управляет жизненным циклом Store (`create/dispose`)
  - связывает цикл Store с UI

Store: структура файла

Приватный Instance класс

Публичный Abstract класс



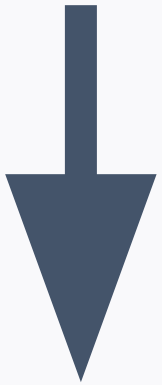
- Добавление дополнительного factory конструктора

+ Упрощение доступа к элементам Store в IDE

Observables и разделение ответственности

Не использовать `@observable` аннотацию

Установить `ReactiveWritePolicy.always`



Запрет на редактирование состояния напрямую из UI

Отделение бизнес-логики от представления

Observables и изменяемые модели

- `Observable` уведомляет только при присвоении нового `value`
- Изменения внутри модели НЕ отслеживаются
- Стоит избегать изменяемых (мутабельных) моделей
- `ObservableList` / `ObservableMap` / `ObservableSet` уведомляют при изменении состава коллекции

Observables и изменяемые модели

```
class Person {
 String firstName;
 String lastName;

 Person({this.firstName, this.lastName})
}
```

```
var person = Observable<Person>(Person(firstName: "John"));
person.value = Person(firstName: "Mark"); // Слушатели будут уведомлены
person.value.firstName = "Mike"; // Слушатели не будут уведомлены
```

Observer: размещение

- `Observer` как можно ближе в дереве виджетов к чтению состояния
- Получение `Store` наверху функции `builder`
- Чтение состояния должно быть в immediate execution context функции `builder` виджета `Observer`

Observer: размещение

```
return Observer(
 builder: (context) {
   return LayoutBuilder(
     builder: (context, constraints) {
       final store = context.watch<DataStore>();


       return Text(store.importantData);
     },
   );
 },
);
```

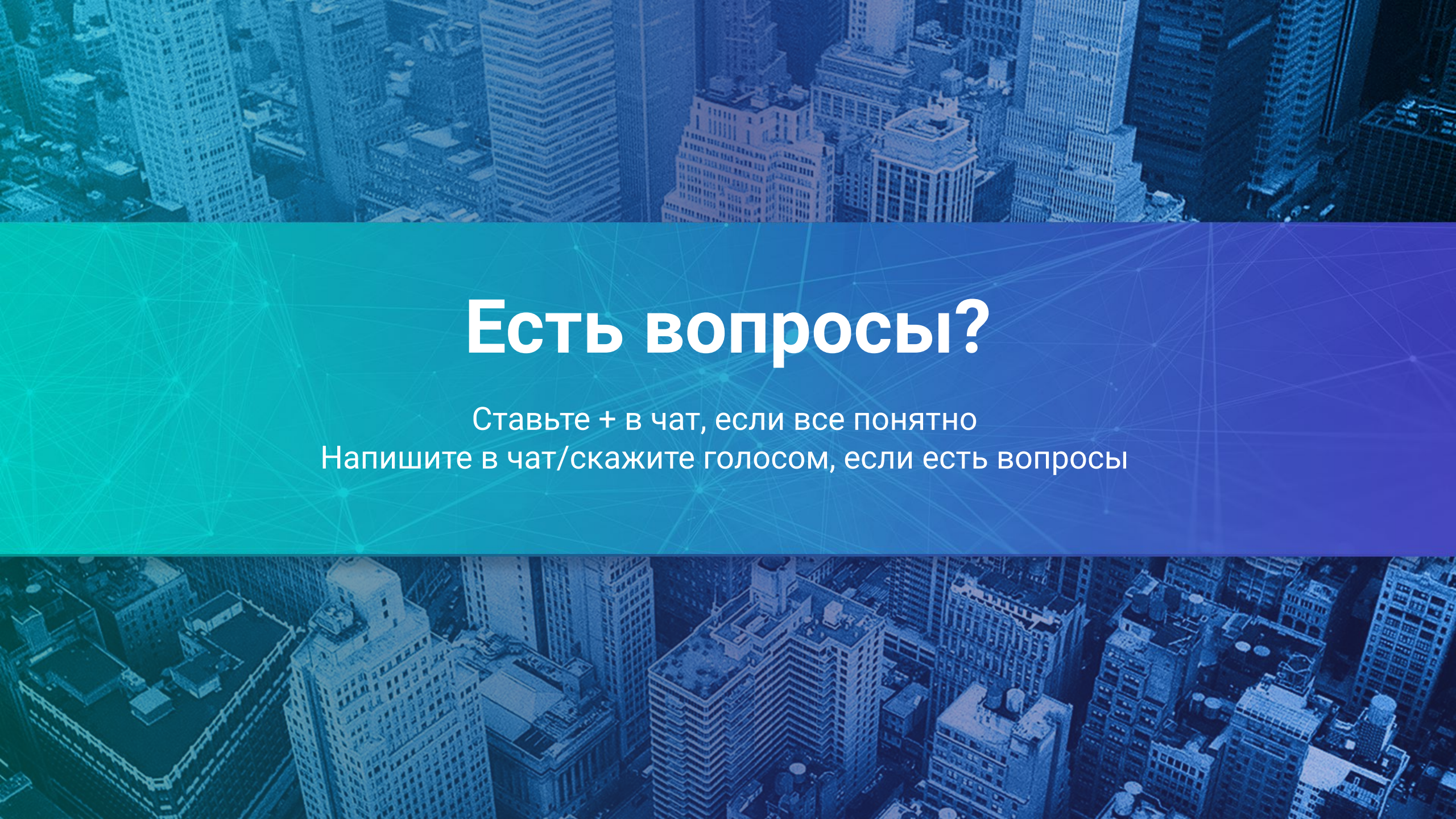
Обновление UI не произойдет!

Observer: размещение

```dart
class MyWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Observer(
     builder: (context) => InnerWidget(),
   );
 }
}
```

```dart
class InnerWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   final store = context.watch<DataStore>();

   return Text(store.importantData);
 }
}
```

Обновление UI не произойдет!



Есть вопросы?

Ставьте + в чат, если все понятно
Напишите в чат/скажите голосом, если есть вопросы

Практика

Задание

1. Забрать проект
2. Связать `ExchangeStore` с `CurrenciesStore`
3. Изменить `ExchangeStore` для обеспечения конвертации по актуальному курсу введенной пользователем суммы
4. Изменить дерево виджетов, чтобы UI менялся при обновлениях состояния `ExchangeStore`

25:00

GitHub

готовы

Итоги - тезисы

1. MobX - библиотека управления состоянием
2. Store состоит из Observable, Computed, Action, Reaction
3. Виджет Observer обновляет UI при изменении состояния
4. Агрегация для связи разных Store

MobX: плюсы и минусы

+ Мало boilerplate-а
+ Простая концепция
+ Легкий доступ к состоянию из UI
+ Хорошая масштабируемость

- Кодогенерация
- Нельзя использовать `@observable`
- `Observer` слушает изменения только в immediate execution context

Цель вебинара | Проверка достижения целей

1 Хранить и обновлять состояние в MobX Store

2 Обновлять UI при изменениях состояния

Смысл:

Использовать масштабируемый и реактивный подход к управлению состоянием в своих приложениях

Рефлексия

? Что думаете о MobX?

? Чем отличается от остальных библиотек управления состоянием?

Список материалов для изучения

- Документация MobX.dart: mobx.netlify.app
- The fundamental principles behind MobX
- Becoming fully reactive: an in-depth explanation of MobX

Домашнее задание

1. Забрать Github репозиторий

2. Внести изменения в классы описания состояний для экрана FavouriteCocktailsScreen (помечены /// todo)

3. Изменить FavouriteCocktailsScreen, выделив логику для получения состояния

4. Изменить CocktailDetailsScreen и isFavourite текущей модели: коктейль должен появиться или удалиться в списке избранного

5. Убедиться, что изменения состояния isFavourite для конкретного коктейля отражается в поведении экрана FavouriteCocktailsScreen: появляется новый избранный коктейль, удаляется ранее убранный из favourites

Домашнее задание

Форма сдачи:

Сдается в виде ссылки на Github репозиторий с проектом

Куда сдать ДЗ:

Отправляется через личный кабинет OTUS

Заполните, пожалуйста,
опрос о занятии по ссылке в чате

Спасибо за внимание!
Приходите на следующие вебинары

Всеволод Краснов
t.me/vs_krasnov