

Пошаговая стратегия

Примеры и референсы



Задачи

- Развития навыков работы на Flutter
- Портфолио
- Умение делать и доводить до конца длинные проекты
- Реализации давней идеи

Управление состоянием (App State) :

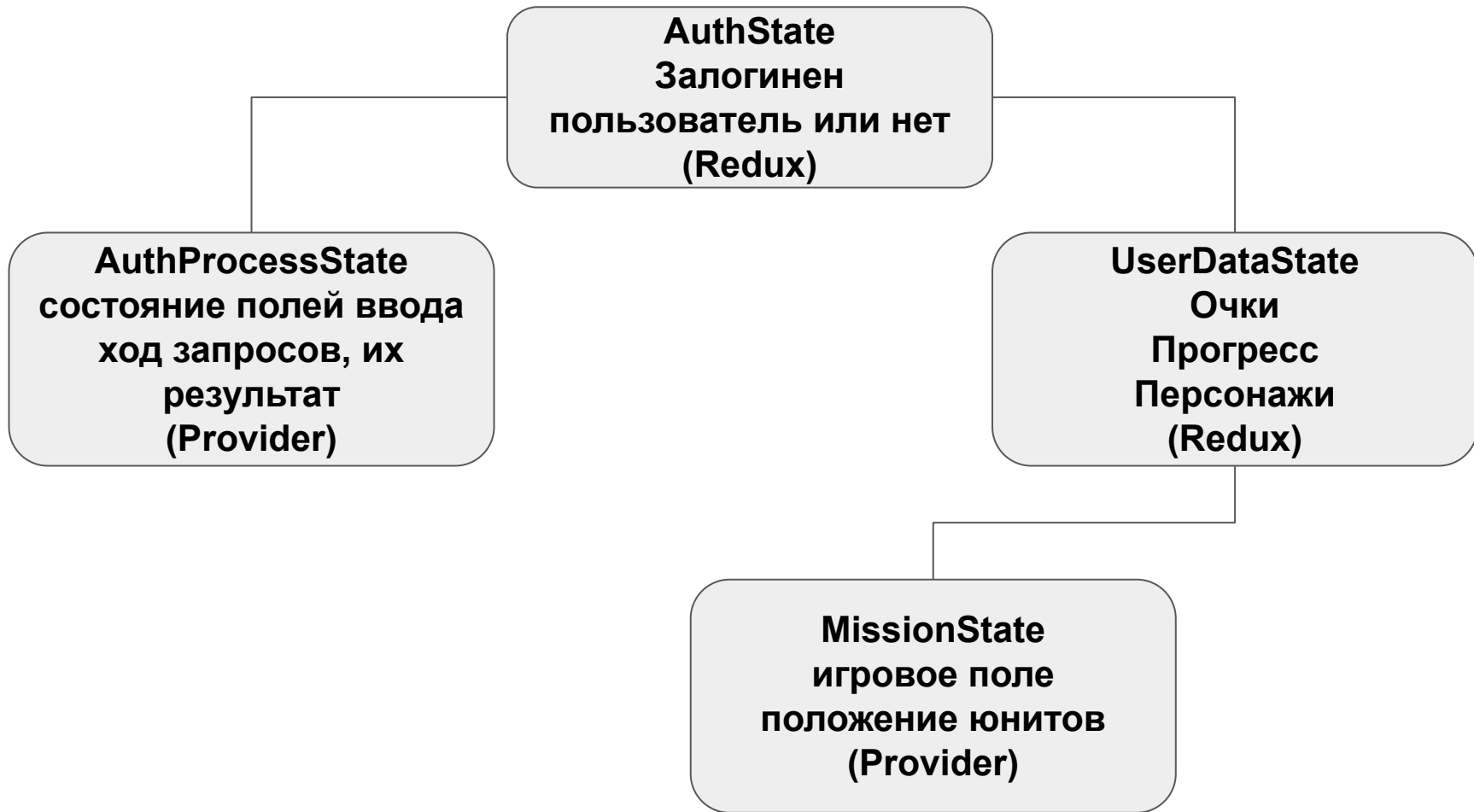
- Provider, ChangeNotifier ([попробовал](#))
- Bloc (использован, внутренний проект)
- **Redux (был выбран этот подход)**
- Mobx (на очереди)

Redux (App State):

- Состояние логина пользователя
- Очков пользователя
- Текущей команды пользователя
- Прогресса сюжетной линии

Provider (Ephemeral State):

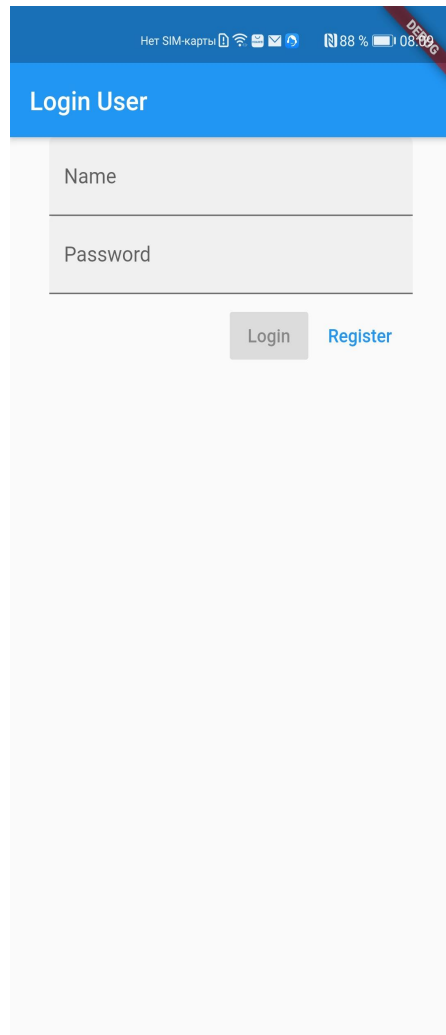
- Состояния формы логина/регистрации
- Хранение состояния текущей миссии
- Отображения списка доступных персонажей



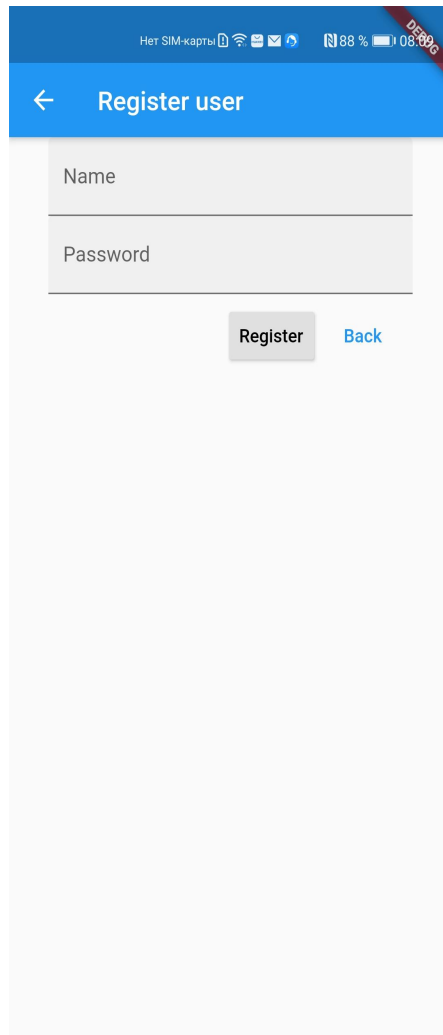
Хранение данных :

- В файле - очень не удобно, не рационально
- SharedPreferences (UserDefaults) - используется временно для логина
- Database - будет использоваться именно этот подход

Хранение данных с помощью БД наиболее подходящий метод. Он требует чуть больше времени на настройку и создание нужных классов для доступа/записи. Но после этого позволяет с удобством работать и выполнять необходимые операции. Легко расширяется и изменяется.



Mockup of a mobile application screen titled "Login User". The screen features a blue header bar with the title. Below the header, there are two input fields: "Name" and "Password". At the bottom, there are two buttons: "Login" (gray) and "Register" (blue).



Mockup of a mobile application screen titled "Register user". The screen features a blue header bar with a back arrow and the title. Below the header, there are two input fields: "Name" and "Password". At the bottom, there are two buttons: "Register" (gray) and "Back" (blue).

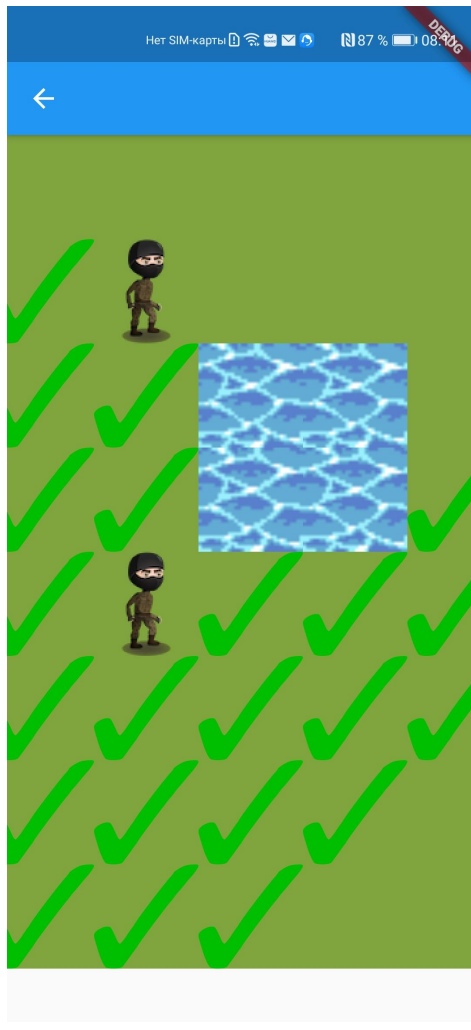
Логин и регистрация.

Внешний вид экрана логина и регистрации целиком взяты из обучения [материал дизайну](#). На этом проекте UI элементы будут максимально соответствовать принятым правилам. Сами картинки может сменятся, но layout и поведение будет таким же.

Состояние самих экранов (доступность полей ввода и кнопок, значение полей ввода, отправка запросов) обрабатывается с помощью Provider.

Для хранения информации о залогиненом пользователе использованы SharedPreferences (временно).

Запросы к серверу выполнены в отдельном Isolate.

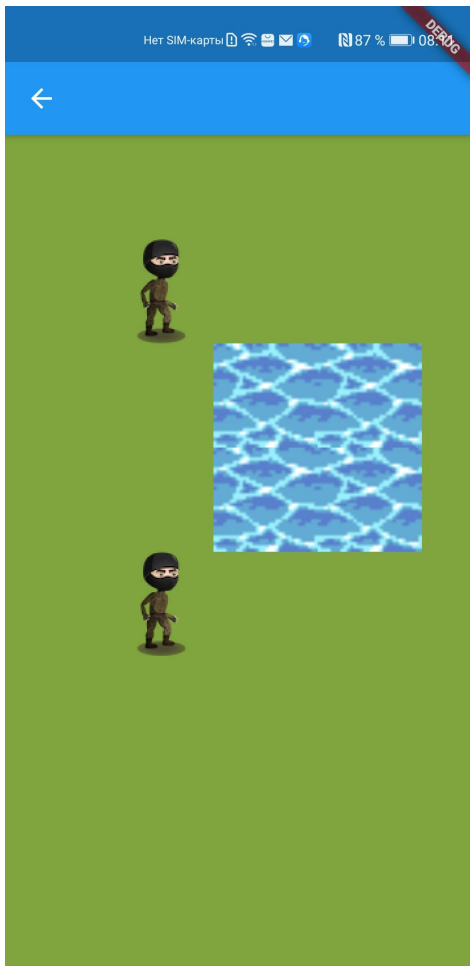


Основной игровой экран (карта миссии).

Внутри лежит Stack с несколькими слоями :

- Виджет поля игры (почва, разные препятствия)
- Виджет со всеми юнитами
- Виджет с UI элементами (индикаторы доступных клеток)

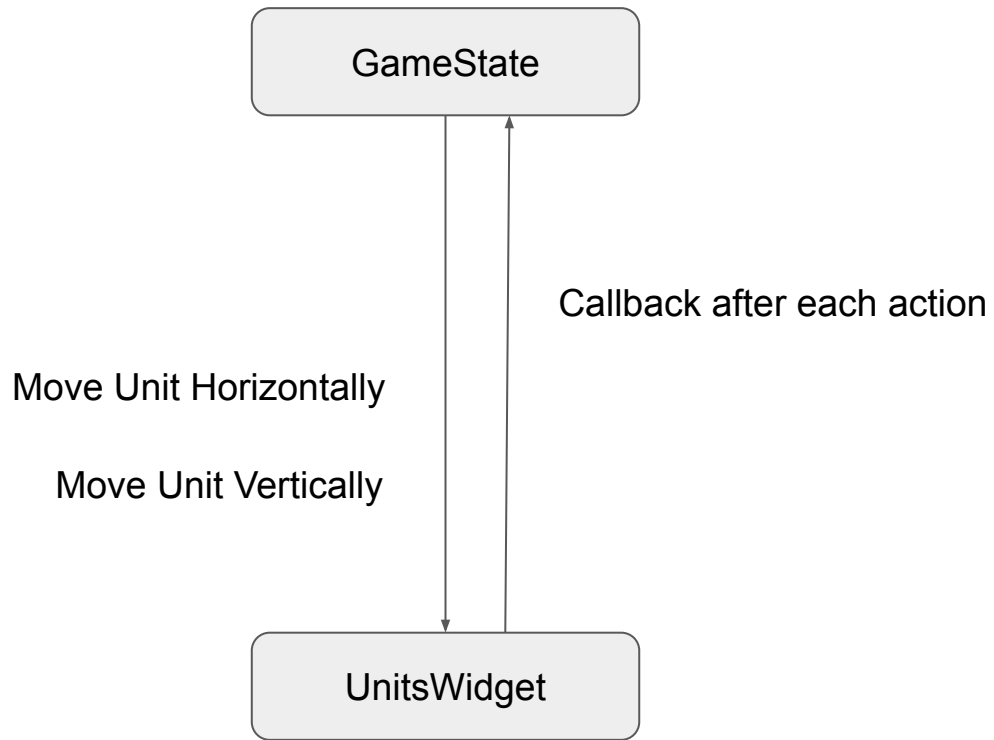
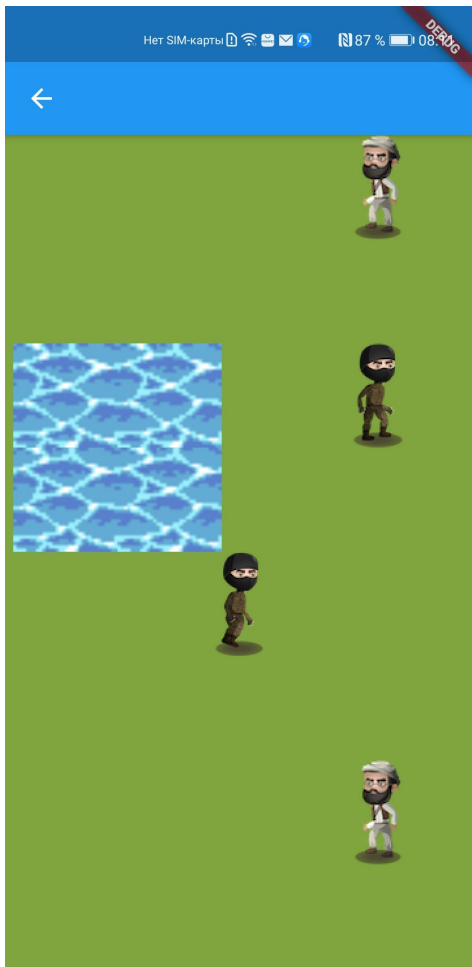
```
body: SafeArea(  
  child: SingleChildScrollView(  
    scrollDirection: Axis.horizontal,  
    child: SingleChildScrollView(  
      scrollDirection: Axis.vertical,  
      child: Stack(  
        children: [  
          GamePoleWidget(),  
          UnitsWidget(),  
          UITilesWidget()  
        ],  
      ),  
    ),  
  ),  
)
```



Каждый юнит - это просто [виджет](#). Который ничего не знает о том, какой юнит он представляет. Его задача просто отрисовывать анимацию которую ему передают.

Для отрисовки анимации используется AnimationController + Tween для более удобного задания времени анимации и отображения кадров. UnitWidget не занимается передвижением юнита. Он только рисует переданную ему анимацию.

Передвижение и управления самими юнитами осуществляется с помощью класса [GameState](#) (ChangeNotifier). У которого есть Stream - по которому он передает действия которые нужно сделать и в каком порядке их нужно выполнять. Это нужно для того, чтобы выстраивать цепочку анимаций. Например, когда игрок переходит на клетку, которая расположена по диагонали от начального местоположения - юнит сначала пройдет по вертикали, а потом по горизонтали. Потом к этой цепочке будут добавляться события атаки/уничтожения юнитов.



Тесты

- Есть тесты на проверку правильности генерации поля. С помощью матчера `findBySemanticLabel` проверяется, что было создано нужное количество ячеек соответствующего типа
- Есть тест на проверку поиска доступных клеток. Проверяем, что алгоритм поиска доступных для движения клеток возвращает клетки с учетом препятствий и других юнитов.

Тесты писать довольно долго и сложно, но они помогают удостовериться, что алгоритмы, сами по себе, работают верно.