

Методические указания к Практической работе №6

по дисциплине «Разработка кроссплатформенных мобильных приложений»

Тема работы: Навигация и маршрутизация в приложении. Адаптивный дизайн приложения

План практической работы:

- Знакомство с императивной навигацией (Navigator 1.0 , Navigator API)
- Знакомство с декларативной навигацией (Navigator 2.0, Pages API, Router)
- Знакомство с виджетом NavigationBar
- Знакомство с адаптивным дизайном приложения
- Выполнение практической работы №6 в соответствии с заданием

Последовательность выполнения практической работы:

1. Знакомство с императивной навигацией (Navigator 1.0 , Navigator API)

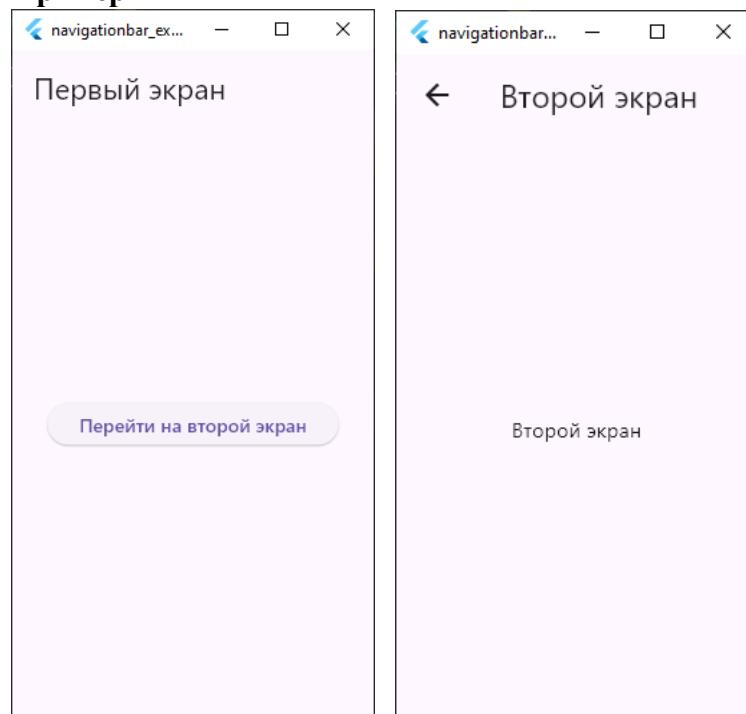
Navigator API базируется на стеке маршрутов, где каждый маршрут представляет собой экран приложения. Navigator предоставляет методы добавления, удаления и замены маршрутов в стеке, а также управляет анимациями перехода между экранами.

Для управления стеком виджетов используются следующие методы:

- *Navigator.push()* – добавляет новую страницу(маршрут) в вершину стека (т.е. страница отображается поверх текущей).
- *Navigator.pushReplacement()* – заменяет текущий маршрут в стеке навигатора. В отличие от Navigator.push(), pushReplacement() удаляет текущий маршрут, чтобы освободить место для нового. Это полезно, когда вам нужно перейти на новый экран и не позволить пользователю вернуться на предыдущий с помощью кнопки «Назад» (например, переход со страницы заставки).
- *Navigator.pop()* – удаляет маршрут из вершины стека (т.е. пользователю отображается предыдущая страница). Если в стеке больше нет маршрутов, приложение будет закрыто.
- *Navigator.pushNamed()* – переход на страницу по имени (добавление именованного маршрута в вершину стека).
- *Navigator.pushReplacementNamed()* – переход на страницу по имени с заменой текущей.
- *Navigator.pushAndRemoveUntil()* – очистка истории (полезно при выходе из профиля, когда нужно удалить все предыдущие маршруты).

При использовании методов Navigator.push() или Navigator.pushNamed() для перехода на новую страницу, Flutter сам добавляет стрелку «←» (Назад, которая вызывает *Navigator.pop()*) в AppBar новой страницы, если в стеке есть хотя бы один предыдущий экран.

Пример



Программный код:

```
import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Навигация Flutter',
      home: const FirstPage(),
    );
  }
}

class FirstPage extends StatelessWidget {
  const FirstPage({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Первый экран')),
      body: Center(
        child: ElevatedButton(
          child: const Text('Перейти на второй экран'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => SecondPage(),
              ),
            );
          },
        ),
      ),
    );
  }
}

class SecondPage extends StatelessWidget {
  const SecondPage({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Второй экран')),
      body: Center(child: Text('Второй экран')),
    );
  }
}
```

Использование именованных маршрутов навигатора (named routes)

Если наше приложение состоит из большого количества экранов, то каждый раз создавать маршрут для перехода на новый экран становится неудобно. Лучшим подходом является использование именованных маршрутов, которые присваивают каждому экрану в приложении уникальный идентификатор.

Имена маршрутов принято использовать как пути в директориях: '/', '/login', '/login/123', и т.д.

Маршрут главного окна по умолчанию: '/'.

MaterialApp(

```
  initialRoute: '/',
  routes: {
    '/': (context) => const FirstPage(),
    '/second': (context) => const SecondPage(),
  },
);
```

Если нам нужно изменить маршрут по умолчанию при открытии приложения, нужно указать параметр initialRoute со значением маршрута, к примеру: initialRoute: '/second'.

Для открытия окна по маршруту нужно использовать Navigator.pushNamed(), например:
Navigator.pushNamed(context, '/second');

Можно вынести имена маршрутов в отдельный класс и для того, чтобы изменения маршрутов производить только в одном файле, вынести его в отдельный файл routes.dart в папке lib:

```
class Routes {  
    static const loading = '/';  
    static const login = '/login';  
}
```

Тогда файл main.dart:

```
import 'package:flutter/material.dart';  
import 'routes.dart'; // импортируем Routes  
import 'pages/loading_page.dart';  
import 'pages/login_page.dart';  
  
void main() => runApp(const MyApp());  
  
class MyApp extends StatelessWidget {  
    const MyApp({super.key});  
  
    @override  
    Widget build(BuildContext context) {  
        return MaterialApp(  
            initialRoute: Routes.loading,  
            routes: {  
                Routes.loading: (context) => const LoadingPage(),  
                Routes.login: (_) => const LoginPage(),  
            },  
        );  
    }  
}
```

Для открытия окна по маршруту:

```
Navigator.pushNamed(context, Routes.login);
```

Пример экрана заставки (файл pages/loading_page.dart) с навигацией по именованному маршруту к экрану авторизации через 3 секунды

```
import 'package:flutter/material.dart';  
import '../routes.dart'; // для доступа к Routes  
  
class LoadingPage extends StatefulWidget {  
    const LoadingPage({super.key});  
  
    @override  
    State<LoadingPage> createState() => _LoadingPageState();  
}  
  
class _LoadingPageState extends State<LoadingPage> {  
    @override  
    void initState() {  
        super.initState();  
        //Имитация инициализации (загрузка настроек, БД)  
        Future.delayed(const Duration(seconds: 3), () {  
            if (!mounted) return;  
            Navigator.pushReplacementNamed(context, Routes.login);  
        });  
    }  
}
```

```
}

@Override
Widget build(BuildContext context) {
    return Scaffold(
        body:
            // Код страницы загрузки
        ),
    );
}
}
```

2. Знакомство с декларативной навигацией (Navigator 2.0, Pages API, Router)

Декларативная навигация (Navigator 2.0, Pages API, Router)

С выпуском Flutter 1.22 был представлен Navigator 2.0, который позволил описывать навигационное состояние, как часть состояния приложения и, по задумке разработчиков Flutter, должен был облегчить реализацию сложных сценариев навигации, таких как: глубокие ссылки (deep links) и веб-навигация.

Важно отметить, что вы не найдете в Flutter SDK класса Navigator 2.0. Ведь изменился не сам класс, а концепция в навигации и подход, как ее реализовывать.

API Flutter Navigator 2.0 относительно редко применяется в чистом виде: он лежит в основе многих популярных пакетов для навигации (мощные библиотеки маршрутизации, такие как GoRouter, Beamer и AutoRoute, поддерживающие вложенную навигацию с отслеживанием состояния).

Основные сущности Navigator 2.0:

- Router (виджет, который отвечает за образование стека страниц Page навигатора в зависимости от состояния приложения)
- BackButtonDispatcher (класс, сообщающий Router через callBack, что пользователь нажал на системную кнопку «Назад» на платформах, поддерживающих данную функциональность (например, Android OS). Параметр backButtonDispatcher не является обязательным и может быть опущен.
- RouteInformationProvider (провайдер путей навигации для виджета Router)
- RouteInformationParser (парсер поступающей от RouteInformationProvider информации о пути в состояние (конфигурацию) навигации)
- RouterDelegate (класс, отвечающий за то, как именно Router узнаёт об изменениях состояния приложения и реагирует на них. Для этого он прослушивает RouteInformationParser и состояние навигации, а затем встраивает в дерево виджет Navigator с готовым стеком страниц)
- RouterConfig (интерфейс, который позволяет инкапсулировать создание раннее перечисленных компонент в единую сущность и передавать их в Router как один объект)

Router (Navigator 2.0) требует много кода и труден в освоении, кроме того, сложен в поддержке. Когда в приложении количество экранов достигает 10-15 штук, тогда конфигурация навигации становится чересчур громоздкой и поддерживать декларативность становится затруднительно.

Поэтому разработчики используют либо проверенный и простой (но, к сожалению, слабо конфигурируемый) Navigator, либо какую-нибудь библиотеку, основанную на Router (GoRouter, Beamer и AutoRoute и другие).

Использование библиотеки go_router

1) Файл pubspec.yaml

```
dependencies:
  go_router: ^16.2.4
```

2) В файле dart

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

3) Базовая настройка маршрутов: необходимо создать экземпляр объекта GoRouter, в котором указаны все маршруты приложения:

```
// Определяем маршруты
final GoRouter _router = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) => const HomePage(),
    ), GoRoute
    GoRoute(
      path: '/details',
      builder: (context, state) => const DetailsPage(),
    ), GoRoute
  ],
); GoRouter
```

- 4) Подключение к приложению: добавить в виджет MaterialApp.router (используем вместо MaterialApp) routerConfig:

```
@override
Widget build(BuildContext context) {
  return MaterialApp.router(
    routerConfig: _router,
    debugShowCheckedModeBanner: false,
  );
}
```

5) Основные методы навигации:

- context.go('/path') – заменяет текущий стек экранов в стеке навигации экранами, настроенными для маршрута назначения (удаляет все существующие маршруты);
- context.push('/path') – добавляет новый экран в стек;
- context.pop() – возврат на предыдущий экран;
- context.replace('/path') – заменяет текущий экран на новый, не трогая остальные маршруты в стеке;
- context.pushReplacement('/path') – заменяет текущий экран, но именно тот, с которого вызван переход;
- context.canPop() – проверяет, есть ли куда возвращаться.

3. Знакомство с виджетом NavigationBar

NavigationBar — нижняя панель навигации, которая позволяет пользователю переключаться между разными наборами виджетов в пределах одного экрана приложения, без необходимости использовать класс Navigator. Используется внутри Scaffold в параметре bottomNavigationBar.

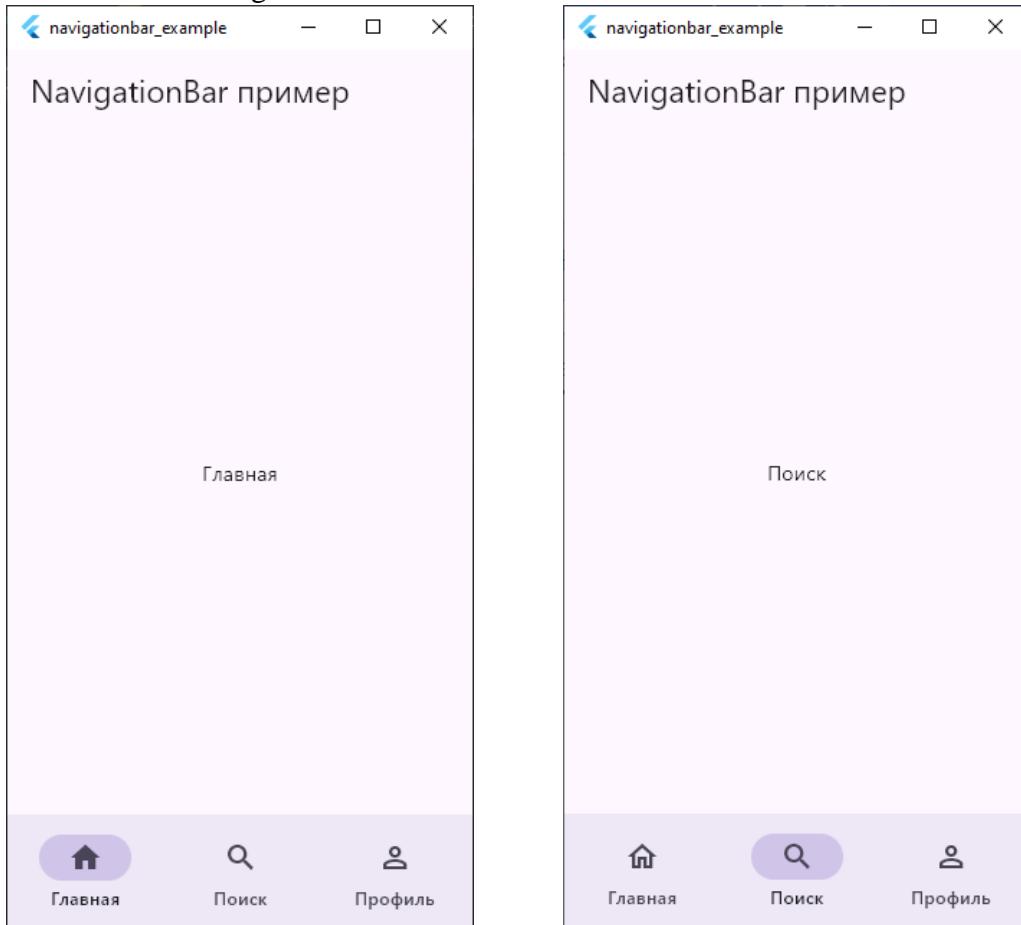
До появления Material Design 3 использовался виджет BottomNavigationBar.

Параметры виджета NavigationBar

Параметр	Назначение
destinations	Список пунктов навигации (объектов NavigationDestination)
selectedIndex	Индекс выбранного пункта
onDestinationSelected	Обратный вызов при выборе пункта (меняет активный индекс)
backgroundColor	Цвет фона панели
indicatorColor	Цвет подсветки выбранного пункта

labelBehavior	Управляет отображением подписей (например, NavigationDestinationLabelBehavior.alwaysShow, .alwaysHide)
height	Высота панели
elevation	Тень под панелью
shadowColor	Цвет границы выбранного элемента
indicatorShape	Форма индикатора выбранного элемента

Пример использования NavigationBar



Программный код:

```

import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      debugShowCheckedModeBanner: false,
      home: NavigationExample(),
    );
  }
}

class NavigationExample extends StatefulWidget {
  const NavigationExample({super.key});

  @override
  State<NavigationExample> createState() => _NavigationExampleState();
}

class _NavigationExampleState extends State<NavigationExample> {
  int _selectedIndex = 0;
}

```

```

static const List<Widget> _screens = [
  Center(child: Text('Главная')),
  Center(child: Text('Поиск')),
  Center(child: Text('Профиль')),
];

@Override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('NavigationBar пример')),
    body: _screens[_selectedIndex],
    bottomNavigationBar: NavigationBar(
      backgroundColor: Colors.deepPurple.shade50,
      indicatorColor: Colors.deepPurple.shade100,
      labelBehavior: NavigationDestinationLabelBehavior.alwaysShow,
      animationDuration: const Duration(milliseconds: 500),
      selectedIndex: _selectedIndex,
      onDestinationSelected: (index) => setState(() => _selectedIndex = index),
      destinations: const [
        NavigationDestination(
          icon: Icon(Icons.home_outlined),
          selectedIcon: Icon(Icons.home),
          label: 'Главная',
        ),
        NavigationDestination(
          icon: Icon(Icons.search_outlined),
          selectedIcon: Icon(Icons.search),
          label: 'Поиск',
        ),
        NavigationDestination(
          icon: Icon(Icons.person_outline),
          selectedIcon: Icon(Icons.person),
          label: 'Профиль',
        ),
      ],
    ),
  );
}
}

```

4. Знакомство с адаптивным дизайном приложения

Адаптивный дизайн – подход к созданию интерфейсов, при котором приложение подстраивается под разные устройства, платформы, размеры экранов, ориентацию и т.д.

Платформа

```

import 'dart:io' //show Platform;
import 'package:flutter/foundation.dart' //show kIsWeb;
...
final double platformPadding =
  (kIsWeb || Platform.isMacOS || Platform.isLinux || Platform.isWindows) //десктоп, веб
    ? 50
    : 20;
...
SizedBox(height: platformPadding);

```

Размер экрана

```

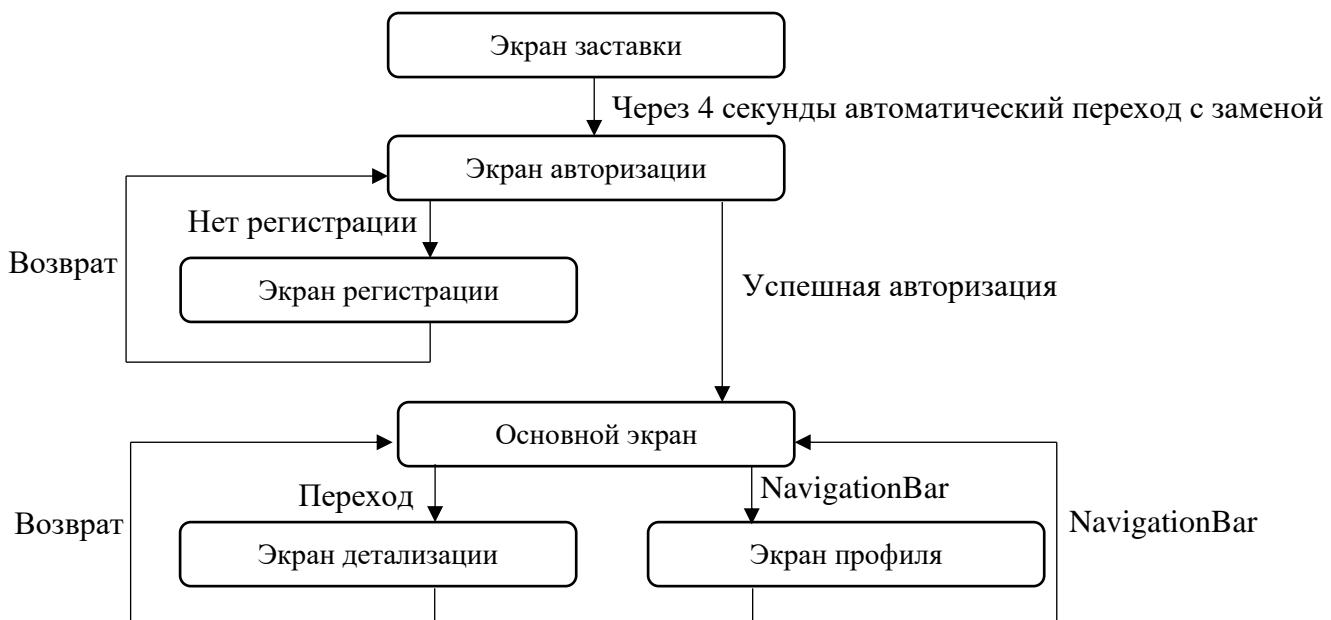
final mediaQuery = MediaQuery.of(context);
final double platformPadding =
  (mediaQuery.size.width > 600)
    ? 50
    : 20;
...
SizedBox(height: platformPadding);

```

5. Задание на практическую работу

Изменить приложение, созданное в предыдущей практической работе, выполнив следующие действия:

- 1) Добавить в проект в папку lib/pages файлы loading_page.dart (экран заставки), detail_page.dart (экран детализации).
- 2) Создайте экраны заставки и детализации пунктов основного экрана согласно макетам ниже.
- 3) Скопируйте код основного экрана, созданный в практической работе №4, из файла main.dart в файл lib/pages/home_page.dart, добавьте на страницу основного экрана нижнюю панель приложения согласно макету ниже.
- В файле main.dart определите маршруты приложения, используя императивный или декларативный подход к навигации (класс объявления имен маршрутов вынесите в отдельный файл routes.dart).
- 4) Добавьте следующую навигацию в приложение:



- 5) На основном экране приложения (home_page.dart) выполните адаптивный дизайн: при ширине экрана более 600 – вывод карточек пунктов в 2 столбца (GridView), а также добавьте отступы по вертикали между элементами для десктопной и веб платформ приложения.

Макеты экранов приложения:

