ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

КОНСПЕКТ ЛЕКЦІЙ
(опорний конспект лекцій)

ТЕХНОЛОГІЇ КРОС-ПЛАТФОРМНОЇ РОЗРОБКИ

для студентів	4 курсу, 2 курсу (за скороченим терміном навчання)
підготовки	першого (бакалаврського) рівня вищої освіти
галузі знань	12 «Інформаційні технології»
спеціальності	121 «Інженерія програмного забезпечення»
освітньо-професійних програм	Програмна інженерія
факультету	Інформаційних технологій та дизайну

Хмельницький – 2024

Конспект лекцій (опорний конспект лекцій) з дисципліни «Технології крос-платформної розробки» підготовки фахівців на першому (бакалаврському) рівні вищої освіти спеціальності 121 «Інженерія програмного забезпечення».

РОЗРОБНИКИ: Хохлов В.А., к.т.н.

РЕЦЕНЗЕНТ: Вороненко М.О., к.т.н., доцент

Затверджено на засіданні кафедри програмних засобів і технологій

Протокол № 1 від «04» вересня 2024 року

Завідувач кафедри

_____ доц., к.т.н. О.Є.Огнєва

УЗГОДЖЕНО:
З НАВЧАЛЬНО-МЕТОДИЧНИМ ВІДДІЛОМ

Реєстраційний номер № _____

Зміст

Лекція №1. Основні відомості про Flutter

Завдяки різноманітним операційним системам, які працюють на мобільних і настільних пристроях у світі, кросплатформна розробка вже давно є метою розробки додатків. Можливість написати одну кодову базу та розгорнути її на кількох платформах значно економить час і зусилля для вашої компанії та команди.

В наш час існує ряд фреймворків, що дозволяют вести крос-платформену розробку. Як правило, це Android, iOS (і можливо Web):

- React Native
- Xamarian
- Qt

У цих фреймворків завжди багато недоліків, один з основних недоліком є їх низька продуктивність.

Одним з останніх фреймворків, що вийшли на міжплатформну арену, є Flutter від Google. Хоча спочатку Flutter підтримував лише мобільні платформи Android та iOS, згодом він розширився, включивши підтримку Інтернету, macOS, Windows, Linux, Fuchsia та вбудованих пристроїв. Це — у поєднанні з швидким циклом розробки Flutter, гнучким дизайном інтерфейсу користувача та продуктивністю рідного додатка — робить його дуже привабливою метою як для нових, так і для досвідчених розробників.

Основна сторінка документації: https://docs.flutter.dev/

Фреймворк Flutter пропонує розробникам такі функції:

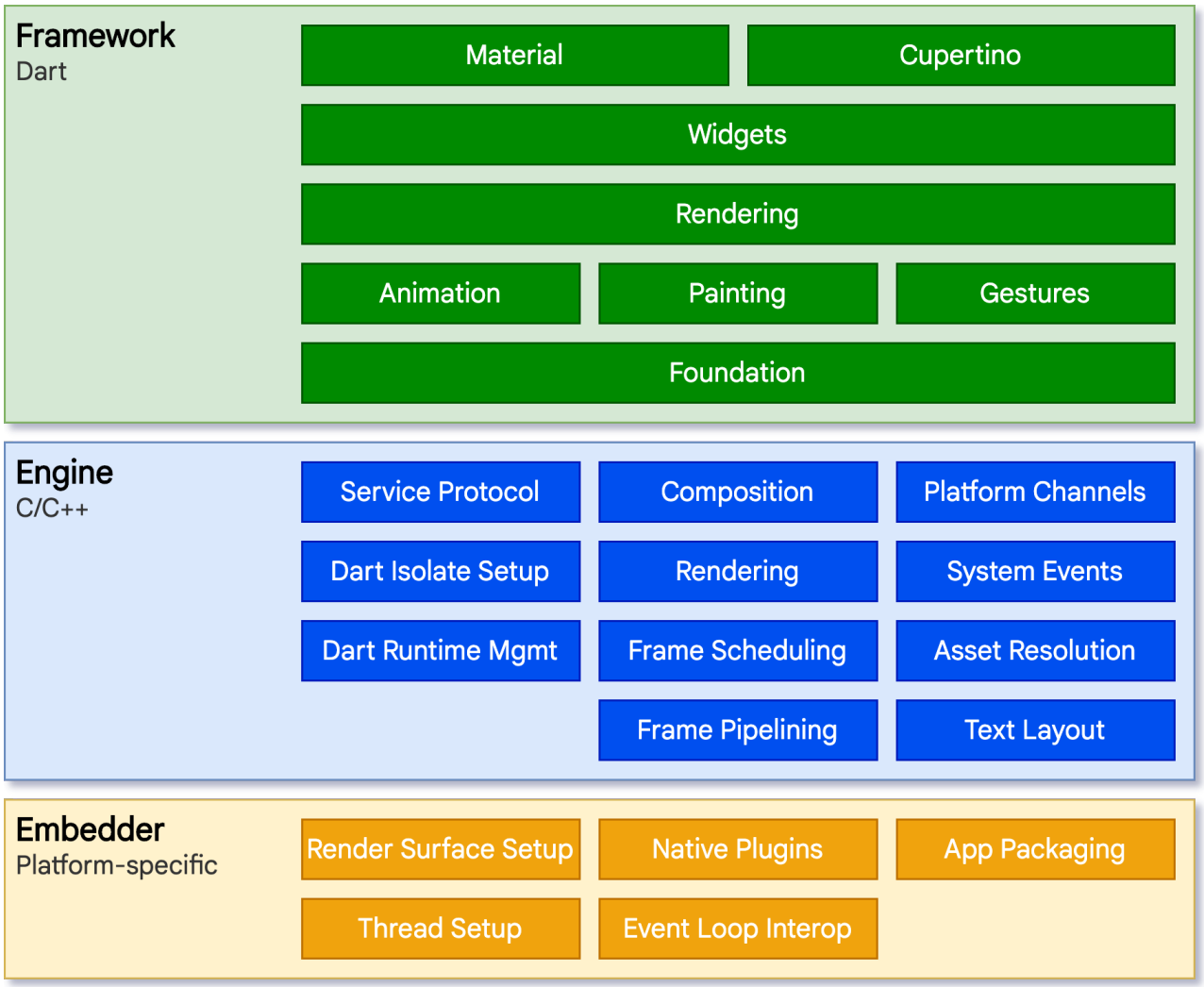
- Сучасна та реактивна структура.
- Використовує мову програмування Dart, і її дуже легко вивчити.
- Швидкий процес розробки, у тому числі hot reload.
- Красиві та плавні інтерфейси користувача.
- Величезний каталог віджетів.
- Використовує той самий інтерфейс користувача для кількох платформ.
- Високопродуктивний додаток.
- Підтримка Ahead-Of-Time (AOT) та Just-In-Time (JIT) режимів компіляції.

Незважаючи на численні переваги, флаттер має такі недоліки:

- Оскільки він закодований мовою Dart, розробнику необхідно вивчити нову мову.
- Сучасний фреймворк намагається максимально розділити логіку та інтерфейс користувача, але у Flutter інтерфейс користувача та логіка змішуються. Це можна подолати це за допомогою розумного кодування та використання модуля високого рівня для розділення інтерфейсу користувача та логіки.

- Flutter – це ще одна платформа для створення мобільного додатка.

Архитектура Flutter



Виджеты Flutter

Основна концепція фреймворка Flutter — у Flutter все є віджетом. Віджети - це в основному компоненти інтерфейсу користувача, які використовуються для створення інтерфейсу програми.

У Flutter сама програма є віджетом верхнього рівня та її інтерфейс створюється з використанням одного або кількох дочірніх (віджетів), які знову ж таки будуються за допомогою своїх дочірніх елементів-віджетів. Такий підхід створювати інтерфейс користувача будь-якої складності.

Віджети Flutter підтримують взаємодію за допомогою спеціального віджета GestureDetector. GestureDetector — це невидимий віджет, який має можливість фіксувати такі дії користувачів як торкання, перетягування, тощо його дочірнього віджета.

Віджети Flutter підтримують підтримку стану, надаючи спеціальний віджет StatefulWidget. Віджети Flutter є реактивними, тобто StatefulWidget буде автоматично перебудовуватись щоразу, як змінюється його внутрішній стан. Повторна перебудова оптимізується шляхом

пошуку різниці між старим і новим інтерфейсами віджетів і відтворення лише необхідних змін.

Розглянемо програму, що створюється автоматчино при створенні нового flutter-проекту.

```
// імпорт  пакету fliutter/material. Цей пакет дозволяє створювати
//интерфейс відповідно до вказаних рекомендацій щодо дизайну на Android.

import 'package:flutter/material.dart';

//Це точка входу до програми Flutter. Викликає функцію runApp і передає
//їй об’єкт класу MyApp. Метою функції runApp є приєднання віджету до екрана.

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

//Віджети використовуються для створення інтерфейсу  у фреймворку flutter.
//StatelessWidget є віджетом, який не підтримує жодного стану віджета.
//MyApp розширює StatelessWidget і перевизначає його метод build. Метою цього
//методу є створення частини інтерфейсу користувача програми.

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

//Метод build створює стандартний віджет   MaterialApp і передає йому
//в якості дочірнього новий віджет  MyHomePage

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
        title: 'Flutter Demo',
        home: MyHomePage(title: 'Flutter Demo Home Page'));
  }
}

//Вигляд віджету  MyHomePage залежить від значення внутрішнтої змінної, тому він
//розширює класс  StatefulWidget

class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key: key);

  final String title;
```

```
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

//З кожним  віджетом StatefulWidget пов’язана окрема змінна классу  State

class _MyHomePageState extends State<MyHomePage> {

//Змінна, що впливає на вигляд віджету
  int _counter = 0;

  void _incrementCounter() {

//Щоб повідомити Flutter про необхідність перебудови віджету необхідно
//викликати метод setState
    setState(() { _counter++; });

  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(widget.title)),
      body: Center(
          child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
            const Text('You have pushed the button this many times:'),
            Text('$_counter')
          ])),
//Кнопка, що по натискані викликає вказанний метод
floatingActionButton: FloatingActionButton(
          onPressed: _incrementCounter,
          tooltip: 'Increment',
          child: const Icon(Icons.add)),
    );
  }
}
```

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz,

Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №2. Основні відомості про мову програмування Dart

Dart — це об’єктно-орієнтована мова програмування із синтаксисом у стилі C. Вона підтримує концепції програмування, такі як інтерфейси, класи. На відміну від інших мов програмування Dart не підтримує масиви. Замість цього Dart підтримує такі колекції як списки (List) та словники (Map).

Основна сторінка https://dart.dev/.

Основні концепції Dart

Все, що ви можете помістити в змінну, є об’єктом, а кожен об’єкт є екземпляром класу. Навіть числа, функції та null є об'єктами.

Хоча Dart є строго типізованою мовою, вказувати тип необов’язково, оскільки Dart може виводити типи.

Dar є null-безпечною мовою. Це означає, що змінні не можуть містити null, якщо ви не скажете, що можуть. Ви можете зробити змінну нульовою, поставивши знак питання (?) в кінці її типу. Наприклад, змінна типу int? може бути цілим числом, а може бути нульовим. Якщо ви знаєте, що вираз ніколи не має значення null, але Dart не погоджується, ви можете додати ! стверджувати, що він не є нульовим.

Якщо ви хочете явно сказати, що будь-який тип дозволений, використовуйте тип Object?, Object або — якщо потрібно відкласти перевірку типу до часу виконання — спеціальний тип dynamic.

Dart підтримує загальні типи, такі як List<int> (список цілих чисел) або List<Object> (список об’єктів будь-якого типу).

Dart підтримує функції верхнього рівня (наприклад, main()), а також функції, пов’язані з класом або об’єктом (статичні методи та методи екземпляра відповідно). Ви також можете створювати функції всередині функцій (вкладені або локальні функції).

Аналогічно, Dart підтримує змінні верхнього рівня, а також змінні, прив’язані до класу чи об’єкта (статичні змінні та змінні екземпляра). Змінні екземпляра іноді називають полями або властивостями.

На відміну від Java, Dart не має ключових слів public, protected і private. Якщо ідентифікатор починається з символу підкреслення (_), він є приватним для його бібліотеки.

Ідентифікатори можуть починатися з літери, знаку $ або підкреслення (_), за якими слідує будь-яка комбінація цих символів і цифр.

Dart має як вирази (які мають значення під час виконання), так і оператори (які їх не мають). Наприклад, результат умовного виразу ? expr1 : expr2 має значення expr1 або expr2. Порівняйте це з оператором if-else, який не має значення. Оператор часто містить один або кілька виразів, але вираз не може містити безпосередньо оператор.

Змінні

Змінні в Dart декларуються та ініціалізуються наступним чином:

```
var name = 'Bob';
```

Змінні зберігають посилання. Змінна під назвою name містить посилання на об’єкт String зі значенням «Bob». Тип змінної імені вважається String, але ви можете змінити цей тип, вказавши його.

Якщо ви ніколи не збираєтеся змінювати змінну, використовуйте final або const замість var або на додаток до типу. final змінну можна встановити лише один раз; const змінна є константою часу компіляції.

Вбудовані типи

Мова Dart має вбудовану підтримку наступних типів:

- Числа (int, double)
- Рядки (String)
- Логічні значення (bool)
- Списки (List)
- Набори (Set)
- Словники (Map)
- Руни (Runes; часто замінюються API символів)
- Символи (Symbol)
- Значення null (Null)

Ця підтримка включає можливість створювати об’єкти за допомогою літералів. Наприклад, «це рядок» є рядковим літералом, а true — булевим літералом.

Оскільки кожна змінна в Dart посилається на об’єкт — екземпляр класу, можна використовувати конструктори для ініціалізації змінних. Деякі з вбудованих типів мають власні конструктори.

Записи (records)

Записи — це анонімні агрегатні незмінні (immutable) типи даних, що дозволяють об’єднувати кілька об’єктів в один. Записи — це реальні значення, вони можуть бути привласнені змінним, передані у функцію як аргумент, поертатися функцією, як результат, зберігатися в списках (list) або словниках (maps), тощо.

Створення запису за допомогою виразу:

```
var record = ('first', a: 2, b: true, 'last');
```

Анотація типу запису — це перелічені через кому типи полів, що взяті в дужки. Приклад функції, що приймає в якості аргументу запис з двух полів типу int й повертає результат такого ж типу:

```
(int, int) swap((int, int) record) {
  var (a, b) = record;
  return (b, a);
}
```

Перший рядок функції — це так званий destructuring, тобто “розбір” запису на окремі змінні.

Поля в записах можуть бути як іменовані, так і позиційні. Для доступа до полів використовуються вбудовані гетери.

```
var record = ('first', a: 2, b: true, 'last');
print(record.$1);
print(record.a);
```

Оскількі записи незмінні, то вони не мають сетерів.

Функції

Dart — це справжня об’єктно-орієнтована мова, тому навіть функції є об’єктами і мають тип Function. Це означає, що функції можна призначати змінним або передавати як аргументи іншим функціям. Також можна викликати екземпляр класу Dart так, ніби це функція.

Dart має вираз => expr , що є скороченням для { return expr; }. Позначення => іноді називають синтаксисом зі стрілками.

Функція може мати будь-яку кількість необхідних позиційних параметрів. За ними можуть слідувати або іменовані параметри, або додаткові позиційні параметри (але не обидва варианти).

Іменовані параметри є необов’язковими, якщо вони спеціально не позначені як обов’язкові. Викликаючи функцію, ви можете вказати іменовані параметри за допомогою paramName: value. Наприклад:

```
enableFlags(bold: true, hidden: false);
```

Визначаючи функцію, використовуйте {param1, param2, …}, щоб вказати іменовані параметри:

```
void enableFlags({bool? bold, bool? Hidden}) {...}
```

Обгортання набору параметрів функції в [] позначає їх як необов’язкові позиційні параметри.

Функція може використовувати = для визначення значень за замовчуванням для необов'язкових параметрів, як іменованих, так і позиційних.

Кожна програма повинна мати функцію main() верхнього рівня, яка служить точкою входу до програми. Функція main() повертає void і має необов’язковий параметр List<String> для аргументів.

Можна створити безіменну функцію, яка називається анонімною функцією, або іноді лямбда або замикання. Анонімну функцію можна призначити змінній, щоб, наприклад, додати або видалити її з колекції.

Керування потоком коду

Можна керувати потоком коду Dart, використовуючи будь-яке з наведеного нижче:

- if and else
- for loops
- while and do-while loops
- break and continue
- switch and case
- assert

Dart також пдтримує виключення: try-catch та throw.

Класи

Dart — це об’єктно-орієнтована мова з класами та успадкуванням на основі міксинів (mixin-based inheritance). Кожен об’єкт є екземпляром класу, і всі класи, крім Null, походять від Object. Спадкування на основі міксинів означає, що хоча кожен клас (крім верхнього класу, Object?) має рівно один суперклас, тіло класу можна повторно використовувати в кількох ієрархіях класів. Методи розширення (Extension methods) – це спосіб додати функціональність до класу без зміни класу або створення підкласу.

Об’єкт створюються за допомогою конструктора. Імена конструкторів можуть бути як ClassName, так і ClassName.identifier. Другий вариант — це так звані іменовані конструктори.

Окрім виклику конструктора суперкласу, можна також ініціалізувати змінні екземпляра перед запуском тіла конструктора. Ініціализатори мають розділятися комами.

Getters і setters – це спеціальні методи, які надають доступ для читання та запису до властивостей об’єкта. Кожна змінна екземпляра має неявний геттер, а також сетер, якщо доречно. Можна створити додаткові властивості, реалізувавши гетери та сетери, використовуючи ключові слова get і set.

Міксини (mixins) – це спосіб повторного використання коду класу в кількох ієрархіях класів. Щоб використовувати міксин, використовуйте ключове слово with з одним або кількома іменами міксинів. У наступному прикладі показано два класи, які використовують міксини:

```
class Musician extends Performer with Musical {
  // ...
}
class Maestro extends Person
    with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

Щоб реалізувати міксин, створіть клас, який розширює Object і не оголошує конструкторів.

```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;
}
```

Генеріки або параметризовані типи (Generics)

Dart підтримує параметризовані тири. Нарпиклад, щоб отб’явити список рядків, можна використати запис List<String>.

Літерари для для list, set, and map :

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

Підтримка асинхронного програмування

Бібліотеки Dart сповнені функцій, які повертають об’єкти Future або Stream. Ці функції є асинхронними: вони повертаються після налаштування, можливо, тривалої операції (наприклад, введення-виводу), не чекаючи завершення цієї операції.

Ключові слова async і await підтримують асинхронне програмування, що дозволяє писати асинхронний код, схожий на синхронний код.

Код, який використовує async і await, є асинхронним, але він дуже схожий на синхронний код. Наприклад, ось деякий код, який використовує await для очікування результату асинхронної функції:

```
await lookUpVersion();
```

Щоб використовувати await, код має бути в асинхронній функції — функції, позначеній як асинхронна:

```
Future<void> checkVersion() async {
  var version = await lookUpVersion();
  // Do something with version
}
```

Хоча асинхронна функція може виконувати тривалі операції, вона не чекає цих операцій. Натомість асинхронна функція виконується лише до тих пір, поки не зустріне свій перший вираз await. Потім він повертає об’єкт Future, відновлюючи виконання лише після завершення виразу await.

Більшість комп’ютерів, навіть на мобільних платформах, мають багатоядерні ЦП. Щоб скористатися перевагами всіх цих ядер, розробники традиційно використовують потоки спільної пам’яті, що працюють одночасно. Однак паралельність спільного стану схильна до помилок і може призвести до складного коду. Замість потоків весь код Dart виконується всередині ізолятів (isolates). Кожен ізолят Dart має єдиний потік виконання і не має жодних змінюваних об’єктів з іншими ізолятами.

Null-безпечне програмування

Використання змінних зі значення null часто є причиною помилок, які важко знаходити і виправляти (так звана мопила на міліард доларів). Dart, як і інші скучані мови підтримує null-безпечне програмування. В термінах Dart це називається sound null safety. Такий підхід означає, что змінні не можуть мати значення null (non-nulabel типи даних), якщо програміст явним чином на повідомить про це. Таким чином помилки часу виконання (runtime null-dereference) стають помилками моменту редаугвання (edit-time analysis errors).

Напримклад, в наступному фрагменті жодна зі змінних не може отримати значення null:

```
var i = 42; // Inferred to be an int.
String name = getFileName();
final b = Foo();
```

Щоб показанати, що змінна може в якийсь час отримати значення null, необхідно додати символ ‘?’ до її типу:

```
int? aNullableInt = null;
```

Для підтримки sound null safety Dart має кілька спеціальних операторів і ключових слів:

- late — non-nulable змінні мають бути ініциалізовані перед використанням. Якщо в місці декларації присвоїти значення змінній ще неможливо, використовується слово late щоб вказати, що змінна обов’язково отримає значення перед її першим використанням.
- required — вказує, що іменований параметр функції обов’язково має бути переданий.
- Оператор ??= - привласнює нове значення змінній у випадку, коли вона має значення null
- Оператор ?. - null-safety вариант оператора ‘.’ - якщо змінна має значення null, повертає null, не звертаючись до вказаного поля зміної.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way

(The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №3. Підтримка об’єкто-орієнтованого програмування в Dart

Dart — це об’єктно-орієнтована мова з класами та успадкуванням на основі міксинів (mixin-based inheritance). Кожен об’єкт є екземпляром класу, і всі класи, крім Null, походять від Object. Спадкування на основі міксинів означає, що хоча кожен клас (крім верхнього класу, Object?) має рівно один суперклас, тіло класу можна повторно використовувати в кількох ієрархіях класів.

Декларація класу може містити:

- поля
- конструктори
- гетери та сетери
- методи

Приклад декларації класу:

```
class Employee {
      // поле
      String name;
      // конструктор
      Employee(String name) {
           this.name = name;
      }
      //getter method
      String get emp_name {
            return name;
      }
      //setter method
      void set emp_name(String name) {
            this.name = name;
      }
      //method definition
      void printName() {
            print(name);
      }
}
```

Приклад використання:

```
void main() {
 //object creation
```

```
 Employee emp = Employee(“FirstLastName”);
 emp.name="employee1";
 emp.printName ();
}
```

В Dart немає окремих ключових слів для позначення видимості методів або полів класу, таких як private або paublic. Якщо ім’я поля чи метода починається з ‘_’, воно є приватним, в іншому випадку — публичним.

Розглянемо більш реалистичний приклад класу, що представляє геометричну фігуру прямокутник.

```
class Rectangle {
  Rectangle(double w, double h) {
    width = w;
    height = h;
  }
  double area() => width * height;
  late double width;
  late double height;
}

void main() {
  final r = Rectangle(100, 200);
  print(r.area());
}
```

Оскількі поля width та height є ненульовими, вони мають бути проініціалізовані до першого використання. Але з одного боку ми не знаємо значень до моменту створення об’єкту цього класу, а з іншого конструктор класу обов’ язково буде викликаний до використання об’єкту, то ми можемо використати слово late й проініціалізувати ці поля в тілі конструктора. Також конструктор може мати список ініціалізації для полів класу, який виконається ще до виконання тіла конструктора:

```
class Rectangle {
  Rectangle(double w, double h):width = w,height = h;
  double width;
  double height;
}
```

При такому варіанті поля отримають значення при створенні об’єкту ще до виконання конструктора. Тому ми можемо опустити слово late, а тіло конструктора зробити пустим.

Dart дозволяє використовувати ще коротший запис:

```
Rectangle(this.width, this.height);
```

Конструктор, як і будь-який метод чи фугкція можуть мати іменовані параметри:

```
Rectangle({required this.width, required this.height});
```

Використання такого метода більш наглядно:

```
final r = Rectangle(width:100, height:200);
```

Dart дозволяє створювати також іменовані констурктори для особливих випадків. Наприклад, наступний конструктор сворює прямокутник з однаковими шириною та довжиною, тобто квадрат:

```
Rectangle.square({required double size}):this(width:size, height:size);
```

Також конструктори можуть викликати один одного, як в наведенному прикладі. Використання цього конструктора:

final s = Rectangle.square(size: 50);

print(s.area());

Розглянемо ще один клас для фігури-круга:

```
class Circle  {
  Circle({required this.radius});
  double area() => 3.14 * radius * radius;
  double radius;
}
```

А також його використання:

```
Circle c = Circle(radius: 10);
print(c.area());
```

Часто є можливість створювати ієрархії класів, утворюючи нові класи шляхом додавання якоїсь спеціалізації до вже існуючих класів. Наприклад і прямокутник, і круг є фігурами, тому доцільно створити базовий клас Shape, від якого унаслідувати вже Rectangle та Circle:

```
class Shape {
  double area() => 0;
}

class Rectangle extends Shape {
  Rectangle({required this.width, required this.height});
  Rectangle.square({required double size}):this(width:size, height:size);
  @override
  double area() => width * height;
  double width;
  double height;
}

class Circle extends Shape {
  Circle({required this.radius});
```

```
  @override
  double area() => 3.14 * radius * radius;
  double radius;
}
```

Такий підхід дозволяє створювати списки фігур й обробляти їх:

```
void main() {
  final List<Shape> shapes = [Rectangle(width:100, height:200),
                              Rectangle.square(size:50),
                              Circle(radius:10),
                             ];
  for(final shape in shapes) {
    print(shape.area());
  }
}
```

В даному випадку клас Shape потрібен лише для створення початку ієрархії. Нам не потрібно створювати його об’єкти. Наступний запис можливий:

```
final List<Shape> shapes = [Rectangle(width:100, height:200), Shape()];
```

але він позбавлений сенсу.

Для унеможливлювання створення об’єктив подібних класів в dart використовується поннятя абстрактного класу:

```
abstract class Shape {
  double area();
}
```

Якщо абстрактний клас лише визначає інтерфейс якогось метода, який має бути обов’язково реалізований в класа-нащадках, то його тіло не треба вказувати.

Окрім модифікатору abstract в dart також модуть застосовуватись наступні можифікатори:

- base
- interface
- final
- sealed

Доволі часто виникає ситуація, коли різні класи з різних ієрархій мають містити якийсь загальний код. Для вирішення цієї проблеми в dart використовується концепція міксінів (mixin):

```
mixin M {
  int m = 10;
  void printM() => print(m);
}
```

```
class A with M {}

class B with M {}

void main() {
  final a = A();
  a.printM();

  final b = B();
  b.printM();
}
```

Хоча класи A та B не споріднені, dart додасть вміст міксіну M в кожен з них. Концепція міксінів дуже широко застосовється у flutter.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №4. Основні відомості про віджети

Основна ідея Flutter — все є віджет. Розглянемо приклад.

```
class MyHomePage extends StatelessWidget {
      MyHomePage({Key? key, required this.title}) : super(key: key);
      final String title;
      @override
      Widget build(BuildContext context) {
            return Scaffold(
                  appBar: AppBar(title: Text(this.title)),
                  body: Center(child: Text('Hello World')));
            }
}
```

В прикладі шляхом наслідвування від StatelessWidget створюється новий віджет.

StatelessWidget вимагає, щоб у його похідному класі був реалізован лише один метод build. Цей метод отримує контекстне середовище, необхідне для створення віджетів, за допомогою параметра BuildContext і повертає віджет, який він створює. У коді використовується title як обов’язковий аргументів конструктора, а також Key як інший опціональний аргумент. Заголовок використовується для відображення заголовка, а ключ — для ідентифікації віджета в дереві віджетів. Тут метод build викликає метод build класу Scaffold, який, у свою чергу, викликає метод build класу AppBar і Center для побудови інтерфейсу користувача. Нарешті, метод build класу center викликає метод build класу Text.

У Flutter віджети можна згрупувати в кілька категорій на основі їхніх функцій, як наведено нижче:

- Спеціальні віджети платформи
- Віджети макета
- Віджети обслуговування стану
- Незалежні від платформи

Спеціальні віджети для платформ

У Flutter є віджети, специфічні для певної платформи - Android або iOS.

Спеціальні віджети для Android розроблені відповідно до рекомендацій щодо Material дизайну ОС Android. Спеціальні віджети Android називаються віджетами Material.

Спеціальні віджети iOS розроблені відповідно до рекомендацій Apple Human Interface Guidelines і вони називаються віджетами Cupertino.

Ось деякі з найбільш використовуваних Material віджетів:

- Scaffold
- AppBar

- BottomNavigationBar
- TabBar
- RaisedButton
- FloatingActionButton
- FlatButton
- Date & Time Pickers

Ось деякі з найбільш використовуваних Cupertino віджетів:

- CupertinoButton
- CupertinoPicker
- CupertinoDatePicker
- CupertinoTabScaffold
- CupertinoTabView
- CupertinoTextField

Віджети макету (Layout widgets)

У Flutter віджет можна створити шляхом створення одного або кількох віджетів. Щоб скласти кілька віджетів в один віджет, Flutter надає велику кількість віджетів з можливістю створювати макети. Наприклад, дочірній віджет можна центрувати за допомогою віджета Center.

Ось деякі з популярних віджетів макета:

- Container: прямокутник, оформлений за допомогою віджетів BoxDecoration

  фону, кордону та тіні.
- Center: центрує його дочірній віджет
- Row: розташовує його дочірні елементи в горизонтальному напрямку.
- Column: розташовує його дочірні елементи у вертикальному напрямку.
- Stack: розташовує один над одним.

Віджети обслуговування стану

У Flutter всі віджети є похідними від StatelessWidget або StatefulWidget.

Віджет, похідний від StatelessWidget, не має жодної інформації про стан, але може містити віджет, похідний від StatefulWidget. Динамічний характер програми підтримується через інтерактивну поведінку віджетів і зміни стану під час взаємодії. Наприклад, натискання кнопки лічильника збільшить/зменшить внутрішній стан лічильника на один і реактивний характер віджета Flutter автоматично повторно відтворить віджет за допомогою інформації про новий стан.

Основні віджети

Flutter надає велику кількість базових віджетів для створення як простих, так і складних інтерфейсів у незалежний від платформи спосіб.

Text

Віджет Text використовується для відображення фрагмента рядка. Стиль рядка можна встановити за допомогою параметру style і класу TextStyle. Зразок коду для цієї мети виглядає так:

```
Text('Hello World!', style: TextStyle(fontWeight: FontWeight.bold))
```

Найважливіші властивості віджета Text наступні:

- maxLines, int: максимальна кількість рядків для відображення
- overflow, TextOverFlow: вказує, як обробляється візуальне переповнення
- style, TextStyle: стиль рядка за допомогою класу TextStyle
- textAlign, TextAlign: вирівнювання тексту, наприклад, праворуч, ліворуч, вирівнювання тощо
- textDirection, TextDirection: напрямок потоку тексту, зліва направо або справа наліво

Icon

Віджет Icon використовується для відображення гліфа зі шрифту, описаного в класі IconData. Відобразити просту піктограму електронної пошти можна так:

```
Icon(Icons.email)
```

Повний код сторінки додатка з піктограмою в ценрті:

```
class MyHomePage extends StatelessWidget {
      MyHomePage({Key? key, required this.title}) : super(key: key);
      final String title;
      @override
      Widget build(BuildContext context) {
            return Scaffold(
                  appBar: AppBar(title: Text(this.title)),
                  body: Center(child: Icon(Icons.email, color: Colors.red)),
            );
      }
}
```

TextButton

Віджет TextButton відображає кнопку (як правило, з текстом), що реагує на натискання.

Найважливіши властивості:

- child, Widget: віджет для візуального представлення кнопки

- style, ButtonStyle: задає стилі кнопки
- onPressed, VoidCallback?: функція, що буде викликана при натисканні на кнопку; якщо null, то кнопка буде неактивною

Приклад:

```
TextButton(
      style: TextButton.styleFrom(textStyle: const TextStyle(fontSize: 20)),
      onPressed: () {},
      child: const Text('Enabled'),
)
```

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №5. Віджети макетів

Оскільки основна концепція Flutter полягає в тому, що все є віджетом, Flutter включає функціональність макетування інтерфейсу в самих віджетах. Flutter надає досить багато спеціально розроблених віджетів, такі як Container, Center, Align тощо, лише з метою макетування інтерфейсу. Віджети створюються шляхом композиції інших віджетів, зазвичай з використуванням віджетів макету.

Віджети макета можна згрупувати у дві окремі категорії залежно від їх дочірнього елементу:

- віджет для підтримки одного дочірнього елементу
- віджет, який підтримує декілька дочірних елементів

Single Child Widgets

У цій категорії віджети мають лише один дочірній віджет, і кожен віджет має спеціальну функціональність.

Наприклад, віджет Center просто центрує його дочірній віджет відносно його батьківського віджета, а віджет Container забезпечує повну гнучкість, щоб розмістити його дочірній у будь-якому місці всередині. Він використовує різні параметри, такі як відступи, оформлення тощо.

Single child widgets – це чудові варіанти для створення високоякісних віджетів, які мають одну функцію, як кнопка, мітка тощо.

Наприклад, для створення 3d-кнопки можна використати віджет Container наступним чином:

```
class MyButton extends StatelessWidget {
      MyButton({Key? key}) : super(key: key);
      @override
      Widget build(BuildContext context) {
            return Container(
                  decoration: const BoxDecoration(
                  border: Border(
                    top: BorderSide(width: 1.0, color: Color(0xFFFFFFFFF)),
                    left: BorderSide(width: 1.0, color: Color(0xFFFFFFFFF)),
                    right: BorderSide(width: 1.0, color: Color(0xFFFF000000)),
                    bottom: BorderSide(width: 1.0, color: Color(0xFFFF000000)),
                  ),
                  ),
                  child: Container(
                        padding: const EdgeInsets.symmetric(horizontal: 20.0,
vertical: 2.0),
                  decoration: const BoxDecoration(
                        border: Border(
                        top: BorderSide(width: 1.0, color: Color(0xFFFFDFDFDF)),
```

```
                left: BorderSide(width: 1.0, color: Color(0xFFFFDFDFDF)),
                right: BorderSide(width: 1.0, color: Color(0xFFFF7F7F7F)),
                bottom: BorderSide(width: 1.0, color: Color(0xFFFF7F7F7F)),
                      ),
                color: Colors.grey,
                ),
                child: const Text('OK',
                      textAlign: TextAlign.center, style: TextStyle(color:
                            Colors.black)),
                ),
          );
    }
}
```

Flutter має велику кількість Single child widgets, зокрема:

- Padding: Використовується для впорядкування його дочірнього віджета за заданим відступом.
- Align: Вирівняйте його дочірній віджет всередині себе, використовуючи значення властивості alignment.
- FittedBox: Масштабує дочірній віджет, а потім розташовує його відповідно до вказаної властивості fit.
- AspectRatio: Намагається встановити розмір дочірнього віджета до вказаного співвідношення сторін.
- Container: Універсальний віджет-контейнер для одного дочірнього віджету на основі прямокутника з вирівнюванням, відступами багатими функціями стилю.

Multiple Child Widgets

У цій категорії віджет матиме більше одного дочірнього віджета та макет кожного віджету унікальний.

Наприклад, віджет Row дозволяє розташовувати його дочірні елементи в горизонтальному напрямку, тоді як віджет Column дозволяє розташовувати його дочірні елементи у вертикальному напрямку. Складаючи Row і Column, можна створити віджет будь-якого рівня складності.

Найчастіше використовуються наступні віджети:

- Row: дозволяє розташувати його дочірні елементи горизонтально.
- Column: дозволяє розташувати його дочірні елементи у вертикальному положенні.
- ListView: дозволяє впорядкувати його дочірні елементи у вигляді списку.
- GridView: дозволяє розташувати його дочірні елементи як галерею.

- Expanded: використовується для того, щоб дочірні віджети Row і Column займали максимально можливу площу.
- Table: віджет на основі таблиці.
- Flow: віджет на основі потоку.
- Stack: віджет на основі стека.

Розглянемо приклда екрану, що демонструє список товарів. Кожен елемент списку — це картка опису товару що містить: назву, опис товару та ціну.

```
class ProductBox extends StatelessWidget {
 ProductBox({Key? key, required this.name,
                        required this.description,
                        required this.price, required this.image,})
            : super(key: key);
      final String name;
      final String description;
      final int price;
      @override
      Widget build(BuildContext context) {
            return Container(
                  padding: EdgeInsets.all(2),
                        height: 120,
                        child: Card(
                        child: Column(
                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
                  Text(this.name,
                  style: TextStyle(fontWeight: FontWeight.bold)),
                  Text(this.description),
                  Text("Price: " + this.price.toString()),
                  ],
            ))); }
}
```

Цей віджет буде використовуватись як дочірний для ListView:

```
class MyHomePage extends StatelessWidget {
 MyHomePage({Key? key, required this.title}) : super(key: key);
      final String title;
      @override
      Widget build(BuildContext context) {
```

```
            return Scaffold(
            appBar: AppBar(title: Text("Product Listing")),
            body: ListView(
            padding: const EdgeInsets.fromLTRB(2.0, 10.0, 2.0, 10.0),
            children: <Widget>[
                  ProductBox(
                        name: "iPhone",
                        description: "iPhone is the stylist phone ever",
                        price: 1000),
                  ProductBox(
                        name: "Pixel",
                        description: "Pixel is the most featureful phone ever",
                        price: 800),
                  ProductBox(
                        name: "Laptop",
                        description: "Laptop is most productive  tool",
                        price: 2000),
                  ProductBox(
                        name: "Tablet",
                        description: "Tablet is the most useful device ",
                        price: 1500),
                  ],
            ));
      }
}
```

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №6. Керування станом flutter-додатка

Flutter — це фреймворк для побудови інтерфейсу користувача (UI). Але UI рідко коли буває сталим. Як правило, екран додатка має різний вигляд в залежності від тої чи іншої умови. Наприклад, фінансові додатки можуть відображати прибуток чи збиток різними кольорами, програма-календар може показувати поточний день іншим кольором, а сам календар відображати або списком днів, або таблицею, в залежності від налаштувань.

Таким чином, можна сказати, що інтерфейс — це функція від стану: UI = f(state)

Базовий підхід до керування станом у Flutter

За замовчанняи для зберігання стану у Flutter-додатках використовуються поля класу відповідного віджету. Розглянемо, наприклад, класс MyHomePage проекту, який створюється командою flutter create project_name:

```
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});
  final String title;
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  void _incrementCounter() { setState(() { _counter++; }); }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(widget.title)),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text('You have pushed the button this many times:'),
            Text('$_counter', style: Theme.of(context).textTheme.headline4),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
```

```
      ),
    );
  }
}
```

Цей клас виконує дві функції:

1. зберігає свій стан за допомогою поля _counter
2. відображає себе на екрані

Хоча такий підхід працює (і навіть інколи використовується), але він має ряд недоліків:

1. змішування в одному класі логічної та інтерфейсної частин призводить до складнощів в супроводженні коду, його масштабуванні та тестуванні
2. зміна значення _counter призводить для перебудови всього екрану, хоча змінюватись мав бы лише Text-віджет, що відображає значення _counter

Керування станом за допомогою Provider

Для Flutter розроблено доволі багато інструментів керування станом, які роз’язують згадані проблеми: ScopedModel, BloC (business logic component).

Розглянемо використання ще однієї бібліотеки — Provider. Для передачі даних з якогось рівня дерева віджетів на нижчі рівні у Flutter використовується спеціальний тип віджету — InheritedWidget. Provider — це обгортка над InheritedWidget, що спрошує роботи з ним.

Бібліотека Provider, як і інші додаткові бібліотеки для Flutter поставляється у вигляді спеціального плагіну (наявні плагіни можна переглянути на https://pub.dev). Щоб додати плагін до проекту, необхідно вказати його в файлі pubspec.yaml. Це спеціальний файл, який містить ріщноманітну інформацію про проект:

```
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.4
```

Необхідно уважно ставитись до віступів у рядках цього файлу, оскільки формат суттєво залежить від них. Після оновлення файлу необхідно виконати команду:

```
flutter pub get
```

Тепер винесемо в окремий файл логіку роботи з лічильником (це буде наша модель):

```
import 'package:flutter/material.dart';
class CounterModel extends ChangeNotifier {
  int _counter = 0;
  int get counter => _counter;
  set counter(int value) {
    if (value != _counter) {
      _counter = value;
```

```
      notifyListeners();
    }
  }
}
```

ChangeNotifier — це спеціальний клас, що за допомогою метода notifyListeners() може повідомляти всім об’єктам, що підписані на нього (listeners) про зміни свого стану.

Плагін provider містить ряд різноманітних провайдерів, в нашому випадку необхідно використовувати ChangeNotifierProvider.

Загальний код з використанням такого провайдера має вигляд:

```
ChangeNotifierProvider<Model>(
      create: (_) => Model(),
      child: Consumer<Model>(
        builder: (context, model, child) {
          // Here you can use the model variable
          // to read and alter the state
        },
      ),
    )
```

В нашому випадку:

```
@override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterModel(),
      child: Consumer<CounterModel>(
        builder: (context, model, child) => Scaffold(
          appBar: AppBar(title: Text(widget.title)),
          body: Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                const Text('You have pushed the button this many times:'),
                Text('${model.counter}', style:
Theme.of(context).textTheme.headline4),
              ],
            ),
          ),
          floatingActionButton: FloatingActionButton(
            onPressed: () => model.counter++,
            tooltip: 'Increment',
            child: const Icon(Icons.add),
```

```
          ),
        ),
      ),
    );
  }
```

Оскільки ми вже не використовуємо поле _counter та метод _incrementCounter, їх можна видалити і навіть перетворити MyHomePage на віджет без стану:

```
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});
  final String title;
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterModel(),
      child: Consumer<CounterModel>(
        builder: (context, model, child) => Scaffold(
          appBar: AppBar(title: Text(title)),
          body: Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                const Text('You have pushed the button this many times:'),
                Text('${model.counter}', style:
Theme.of(context).textTheme.headline4),
              ],
            ),
          ),
          floatingActionButton: FloatingActionButton(
            onPressed: () => model.counter++,
            tooltip: 'Increment',
            child: const Icon(Icons.add),
          ),
        ),
      ),
    );
  }
}
```

Таким чином, за допомогою Provider ми відокремили логічну частину програми від інтерфейсної. Така архітектура програми, коли є окремі частини програми виділяються в модель та вигляд, має назву Model-View-ViewModel — MVVM.

Додаткові вожливості Provider

Змінимо текст програми наступним чином:

```
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});
  final String title;
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterModel(),
      child: Consumer<CounterModel>(
        builder: (context, model, child) {
          print('BUILD Home widget');
          return Scaffold(
            appBar: AppBar(title: Text(title)),
            body: Center(
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Label(),
                  Text('${model.counter}', style:
Theme.of(context).textTheme.headline4),
                ],
              ),
            ),
            floatingActionButton: FloatingActionButton(
              onPressed: () => model.counter++,
              tooltip: 'Increment',
              child: const Icon(Icons.add),
            ),
          );
        },
      ),
    );
  }
}

class Label extends StatelessWidget {
  const Label({Key? key}) : super(key: key);
  @override
```

```
  Widget build(BuildContext context) {
    print('BUILD Text Label');
    return const Text('You have pushed the button this many times:');
  }
}
```

Ми виділили текстове повідомлення в окремий віджет і додали друк рядків тексту в консоль. Якщо запустити додаток і натискати на кнопку, то в консолі будуть з’являться рядки:

```
flutter: BUILD Home widget
flutter: BUILD Text Label
```

Але нам не потрібно перебудовувати Label кожен раз, оскількі цей віджет ніколи не змінюється, оскільки він не залежить від значення лічильника. Якщо додати const перед цим віджетом в методі build:

```
const Label(),
```

то він не буде змінюватись (рядок “flutter: BUILD Text Label” зникне з консолі), але не завжди віджет може бути константним. Але треба намагатись використовувати константні віджети якогомога частіше.

Пакет provider окрім віждету Consumer містить ряд віджетів, які дозволяють реагувати лише на зміну окремих полів моделі, зокрема віджет Select. Також Provider додає додаткові методи (select, read, тощо) до класу BuildContext, що спрощують використання виджетів Select.

З використанням цих додаткових можливостей нащ код буде мати вигляд:

```
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key, required this.title});
  final String title;
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterModel(),
      builder: (BuildContext context, __) {
        print('BUILD Home widget');
        return Scaffold(
          appBar: AppBar(title: Text(title)),
          body: Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                Label(),
                CounterLabel(),
              ],
            ),
          ),
```

```
          floatingActionButton: FloatingActionButton(
            onPressed: () => context.read<CounterModel>().counter++,
            tooltip: 'Increment',
            child: const Icon(Icons.add),
          ),
        );
      },
    );
  }
}

class Label extends StatelessWidget {
  const Label({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    print('BUILD Text Label');
    return const Text('You have pushed the button this many times:');
  }
}
class CounterLabel extends StatelessWidget {
  const CounterLabel({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    final value = context.select<CounterModel, int>((model) => model.counter);
    print('BUILD Counter label');
    return Text('$value', style: Theme.of(context).textTheme.headline4);
  }
}
```

В консолі пергий раз з’являться рядки:

```
flutter: BUILD Home widget
flutter: BUILD Text Label
flutter: BUILD Counter label
```

А після натискання на кнопку лише:

```
flutter: BUILD Counter label
```

Таким чином, при зміні стану буде перебудовуватись лише потрібна частина UI.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.

- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №7. Навігація у flutter-додатках

Навігація є основною концепцією розробки мобільних додатків. Це дозволяє користувачам переходити з одного екрана на інший. Добре реалізоавна навігація забезпечує організованість програми та розширює її можливості.

Flutter пропонує імперативний API (Navigator 1.0) і декларативний API (Navigator 2.0) механізми маршрутизації. У Navigator 1.0 ви можете додати сторінку лише на вершину стеку навігації та видалити сторінку з вершини стеку. З іншого боку, декларативний механізм дозволяє вам повністю контролювати навігаційний стек. Також Navigator 2.0 спрощує навігацію у web-додатках.

Navigator 1.0

Керує навігацією у fluter віджет Navigator. Для додавання сторінки на вершину стеку навігації використовується метод push. Класовий варіант цього метода приймає два аргументы: BuildContext та сторінку, які в термінах flutter называються маршрутами (routes). Фрагмент коду, що виконує перехід на нову сторінку має вигляд:

```
Navigator.push(
      context,
      MaterialPageRoute<bool>(builder: (BuildContext context) => GamePage6()));
```

Для повернення на попередню сторінку використовується метод pop. Класовий варіант цього метода приймає один обов’язковий аргумент — BuildContext:

```
Navigator.pop(context)
```

Якщо сторінці необхідно повернути якесь значення на попередній маршрут, то його можна передати через другий аргумент метода pop. Це значення буде доступне як значення, що повернув метод push. Таким чином, загальний фрагмент коду для навігації може мати наступний вигляд:

```
   final bool? shouldShowAd = await Navigator.push(
      context,
      MaterialPageRoute<bool>(
        builder: (BuildContext context) => GamePage4())
    );
...
Navigator.pop(context, true);
```

Недоліком такого підходу є те, що інформація про маршрути додатка може знаходитись в різних мисцях коду, що ускладнює процес програмування.

Navigator 1.0 має також інший підхід для визначення маршрутів у додатку. Для цього віджету MaterialApp передати в якості одного з аругментів словник (map) марщрутів та їх ідентифікаторів. Ідентифікатором маршруту може бути довільний рядок символів. Також необхідно вказати початковий маршрут:

```
MaterialApp(
...
```

```
initialRoute: '/',
routes: {
    '/': (BuildContext context) => const MenuPage(),
    '/help': (BuildContext context) => const HelpPage(),
    '/settings': (BuildContext context) => const SettingsPage(),
   },
)
```

Для переходу на один з зазначених маршрутів використовується метод pushNamed:

```
Navigator.pushNamed(context, '/help')
```

Navigator 2.0

Можливості Navigator 1.0 в цілому достатні для використання в невеличких мобільних додатках. Але недоліками є те, що маршрутизация задається в імперативному стилі, в той час, як інтрефейс користувача у flutter програмується декларативно. Також цих можливостей недосттньо для великих додатків зі складною навігацією, для підтримки deep links (переходів в середину додатка за допомогою спеціальних інтернет-адрес) та для навігації у web-додатках. Для розв’язання цих проблем команда flutter запропонувала Navigator 2.0. Слід зазначити, що він не є заміною існуючого підходу, оскількі обидві версії навігаторів можна використовувати одночасно.

Для декларативного стилю визначення маршрутів в додатку Navigator 2.0 додає додаткові аргументи для віджета Navigator.

Можливі сторінки додатка задаються за допомогою аргумента pages. Цей список може перебудовуватись в залежності від стану додатку. Перебудова списку призводить до переходів між сторінками:

```
return MaterialApp(
      title: 'Navigation Demo',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: Navigator(
        pages: [
          MaterialPage(
            key: const ValueKey('homePage'),
            child: MyHomePage(
              title: 'Navigation Demo Home Page',
              onButtonTap: () {
                setState(() {
                  _isDetailedPageOpened = true;
                });
              },
            ),
          ),
],
```

...

Для керування станом в найпростішому випадку можна використовувати свої змінні і метод setState.

При завданні параметра pages обов’язковим є також аргумент onPopPage — це функція, яка буде викликана прі поверненні з будь-якої сторінки назад по стеку навігації.

Розглянемо невеличкий приклад додатка, що містить дві сторінки:

1. основна, з кнопкою, що виконує перехід на детальну сторінку

2. сторінку з детальною інформацією

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

  @override
  State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  bool _isDetailedPageOpened = false;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Navigation Demo',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: Navigator(
        pages: [
          MaterialPage(
            key: const ValueKey('homePage'),
            child: MyHomePage(
              title: 'Navigation Demo Home Page',
              onButtonTap: () {
                setState(() {
                  _isDetailedPageOpened = true;
                });
              },
            ),
          ),
          if (_isDetailedPageOpened)
            const MaterialPage(
              key: ValueKey('detailsPage'),
```

```
            child: DetailsPage(),
          ),
        ],
        onPopPage: (route, result) {
          if (!route.didPop(result)) {
            return false;
          }
          if (_isDetailedPageOpened) {
            setState(() {
              _isDetailedPageOpened = false;
            });
          }
          return true;
        },
      ),
    );
  }
}
```

Для більш складної навігації та підтримки deep links Navigator 2.0 містить спеціальний віджет Router, а також додаткові класи: RouteInformationProvider, RouteInformationParser, RouterDelegate, BackButtonDispatcher.

Але конфігурування безпосередньо віджету Router виявляється доволі складним. Тому користувачі flutter розробили декілька плагінів, що спрощують цей процес. Найбільш досконалим та популярним є плагін go_router.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №8. Анімація у fluter-додатках

Анімація є невід’ємною частиною інтерфейсу сучасних мобільних додатків. Плавні переходи та інтерактивні елементи роблять програму приємною у використанні. З іншого боку, погано зроблена анімація може зробити програму незграбною або, що ще гірше, взагалі зламати програму.

Аніміровані віджети flutter

Flutter містить велику вількість анімірованих віджетів. Всі вони міняють один або декілька параметрів віджета за вказаний проміжок часу. За початкове значення параметра береться його попереднє значення, на кінцеве — нове. Тривалість анімації (duration) є обов’язковим параметром таких віджетів.

Розглянемо деякі з них:

- AnimatedAlign — міняє вирівнювання (ліворуч, праворуч, тощо...) віджета.
- AnimatedContainer — віджет Container, який змінює свої параметри.
- AnimatedOpacity — анімірована версія віджету Opacity, що задає прозорість свого дочірнього віджету.
- ScaleTransition — анімує масштаб віджету, що трасформується.
- AnimatedBuilder — загальний віджет для побудови анімації.

В якості прикладу додамо в стартовий проект (лічильник) віджет AnimatedContainer:

```
...
child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            AnimatedContainer(
              duration: const Duration(milliseconds: 300),
              color: Colors.green,
              height: 50,
              width: _counter * 20,
            ),
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headlineMedium,
            ),
          ],
...
```

Такі зміни призведуть до появи над текстом зеленої полоски, яка буде отримувати нову ширину після кожного натискання на кнопку. Зміна буде відбуватися протягом 300 мілісекунд.

Керування анімацією у flutter

Розглянемо маханізми, які надає flutter для керування анімацією.

Однією з основних концепцій, пов’язаних з анімацією у flutter, є tweens. Ці об’єкти задають початкове та кінцеве значення якогось параметра (назва походить від слова beTWEEN). tween-об’єкти створюються або за допомогою generic-класу Tween<T> або за допомогою спеціалізованих класів, наприклад, ColorTween.

Для керування процесом анімації використовується Animation Controllers. Керування відбувається таким чином, що анімація змінює своє значення та повертає його на основі прогресу анімації.

Таким чином, кожна анімація Flutter потребує створення принаймні двох елементів:

- Tween, що генерує значення для анімації
- AnimationController, що керує прогресом анімації

AnimationController задає прогрес анімації від 0 до 1, тоді як Tween дає фактичне значення анімації, що очікується віджетом.

AnimationController також дає нам контроль над тим, як поводиться анімація за допомогою таких методів, як forward(), reverse() або repeat().

При створенні об’єкта AnimationController необхідно вказати як мінімум два переметри:

duration — час анімації

vsync — об’єкт, який надає так званні ticks — моменти часу, в які необхідно обчислити нові значення анімації. Для цього може застосовуватись, наприклад, міксін SingleTickerProviderStateMixin.

Модифікуємо стартовий проект flutter таким чином, щоб він демонстрував анімірований прямокутник, що змінює свій розмір і колір. Для цього змінимо класс _MyHomePageState наступним чином:

```
class _MyHomePageState extends State<MyHomePage>
    with SingleTickerProviderStateMixin {
  late AnimationController controller;
  late Animation colorAnimation;
  late Animation sizeAnimation;

  @override
  void initState() {
    super.initState();
    controller =
```

```
        AnimationController(vsync: this, duration: const Duration(seconds: 2));
    colorAnimation =
        ColorTween(begin: Colors.blue, end: Colors.yellow).animate(controller);
    sizeAnimation = Tween<double>(begin: 100.0, end: 200.0).animate(controller);

    controller.addListener(() {
      setState(() {});
    });

    controller.repeat();
  }

  @override
  void dispose() {
    controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: Text(widget.title),
      ),
      body: Center(
        child: Container(
          height: sizeAnimation.value,
          width: sizeAnimation.value,
          color: colorAnimation.value,
        ),
      ),
    );
  }
}
```

Ми створюємо два об’єкти tween: один для розміру інший для кольору, а також контролер анімації. Оскількі вони буду створюватись пізніше, ніж об’єкт класу _MyHomePageState, ми використовуємо ключове слово late.

Для безперервною анімації, запускаємо її методом repeat(). Для того, щоб віджет перебудовувався кожен раз, коли зміниться значення анімації, додаємо до контролера лістенер:

```
controller.addListener(() {
      setState(() {});
    });
```

При побудові контейнера відповідні значення беруться з відповідних tween:

```
Container(
    height: sizeAnimation.value,
    width: sizeAnimation.value,
    color: colorAnimation.value,
),
```

Рекомендується звільняти ресурси, що зайняті анімацією як тільки зникає потреба в них:

```
@override
  void dispose() {
    controller.dispose();
    super.dispose();
  }
```

Метод dispose викликається фреймворком flutter в момент, коли віджет видаляється з дерева віджетів.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/

Лекція №9. Робота з Internet

Робота з Internet є важливою складовою сучасних мобільних додатків. Досить часто мобільний додаток працює в парі з серверною частиною, виконуючи різноманітні запити по мережеі Internet до сервера і відображаючи отриману інформацію в потрібному вигляді.

Пакет http

Для Flutter розроблено декілька пакетів, що спрощують роботу з Internet. Базовим пакетом є http. Встановлюється пакет звичаним чином:

```
flutter pub add http
```

або додається беспосередньо у файл pubspec.yaml в секцію dependencies:

```
dependencies:
  http: ^1.2.2
```

Для Android-додатків потрібно також вказати в AndroidManifest.xml дозвіл на роботу з Internet:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Пакет http міть набір функцій першого рівня, для виконання різноманітних запитів, згідно протоколу HTTP: get, create, put, patch, delete...

```
import 'package:http/http.dart' as http;
void main() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');
  final response = await http.get(url);
  print('Status code: ${response.statusCode}');
  print('Body: ${response.body}');
  print('Headers: ${response.headers}');
 }
```

Ці функції повертають Future, що містить об’єкт класу Response. Цей клас містить різноманітні поля з інформацією про відповідь сервера: statusCode, headers, body, тощо.

Сайт jsonplaceholder.typicode.com дозволяє тестувати різноманитні http-запити. Він має API, що повертає фейкові значення для ряду об’єктів: user, post, album. Більш детально описано в документації. Запит GET https://jsonplaceholder.typicode.com/posts/{id} повертає пост з вказаним id. Результат роботи програми може бути наступним:

```
Status code: 200
Body: {
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
```

```
}
Headers: {cache-control: max-age=43200, content-type: application/json;
charset=utf-8, expires: -1, pragma: no-cache}
```

Асинхронні операції в dart

Звертання до будь-якого серверу може займати доволі багато часу. Для кращого користувацього досвіду бажано, щоб підчас такого очікування додаток залишався придатним для користувача. В інакшому випадку буде виникати ілюзія, що додаток «завис». В сучасних мовах програмування з цією метою використовуються асинхроні функції. Це спеціальний вид функцій, які по-першн, повертають одразу не результат, а спеціальний об’єкт, який отримає результат пізніше, а по-друге, можуть бути призуминені і продовжені пізніше з точки останову. В dart асихнроні функції повертають об’єкт класу **Future**, позначаються словом **async**, а призупиняються в очікуванні відповіді довгої операції за допомогою слова **await**.

Обробка JSON-даних

Велика кількість серверів повертає відповіді у форматі JSON (java script object notation). Це словник, ключами якого є рядки — назви властивостей об’єкта, а значеннями — значення цих властивостей. Значення можуть бути різних типів: bool, int, double, String, List, Map. Для представлення таких структур в dart використовується тип даних Map<String, dymanic>. Але доцільніше створювати відповідний клас, щоб компілятор зміг перевірити відповідність типів. З цією метою створюється клас, у якого один з конструкторів створює об’єкт з json-структури.

Для представлення об’єкту посту з https://jsonplaceholder.typicode.com можна сторити наступний клас:

```
 class Post {
    Post({required this.id, required this.userId, required this.title, required
this.body});
    Post.fromJson(Map<String, dynamic> json)
      : id = json['id'] as int,
        userId = json['userId'] as int,
        title = json['title'] as String,
        body = json['body'] as String;
    final int id;
    final int userId;
    final String title;
    final String body;
}
```

Іменований конструктор fromJson створює об’єкт з переданого json-запису. Конструктор fromJson можна також створити, використовуючи співставлення за зразком:

```
factory Post.fromJson(Map<String, dynamic> json) {
    return switch (json) {
      {
```

```
        'userId': int userId,
        'id': int id,
        'title': String title,
        'body': String body,
      } => Post(userId: userId, id: id, title: title, body: body),
      _ => throw const FormatException('Failed to load album.'),
    };
  }
```

Конвертація відповіді сервера в об’єкт класу Post відбувається настйпним чином:

```
if (response.statusCode == 200) {
    final post = Post.fromJson(jsonDecode(response.body) as Map<String,
dynamic>);
  } else {
    throw Exception('Failed to load album');
  }
```

Функція jsonDecode з пакету 'dart:convert' оброблює json-рядок і створює словник, якій потім передається конструктору Post.fromJson.

Пакет dio

Іншим популярним пакетом для роботи з Internet є dio.

Розглянемо приклад функції, що отримує з сатйту https://jsonplaceholder.typicode.com і повератє об’єкт List<Post>

```
import 'package:dio/dio.dart';
Future<List<Post>?> fetchPosts() async {
  final url = 'posts';
  List<Post>? posts;
  try {
      final Response resp = await _httpClient.get<List<dynamic>>(url);
      posts = (resp.data as List<dynamic>)
          .map((e) => Post.fromJson(e as Map<String, dynamic>))
          .toList();
   } on DioException catch (e) {
      print('Posts api error: $e');
   }
  return posts;
}
```

Змінна _httpClient визначається наступним чином:

```
final Dio _httpClient = Dio(BaseOptions(baseUrl:
'https://jsonplaceholder.typicode.com/'));
```

Якщо додаток має сторінку, яка відображає список постів при її відкритті, цей метод можна викликати в методі initState відповідного класу для StatefullWidget або як аргумент future віджету FutureBuilder.

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/
- Проект «Facts about numbers» з вручную написаним кодом – Режим доступу: https://github.com/xvadim/number_facts

Лекція №10. Кодогенерація у Flutter

В процесі розробки часто виникає необхідність написання стандртного шаблоного коду для багатьох випадків. Прикладами може бути код конвертації json-даних в об’єкти класів. Для спрощення цього процесу застосовється кодогенерація: підхід, який дозволяє спеціальними анотаціями описани необхідний функціонал і використовуючи спеціальні інструменти сгенерувати потрібний програмний код.

Пакет build_runner

Для Flutter розроблено багато пакетів для кодогенерації. Пакет build_runner є особливим: він забезпечує функінування інших кодогенеруючих пакетів й керує їми. Цей пакет (як і інші для кодогенерації) використовується не під час роботи додатка, а під час його створення, тому він додається не в секцію dependencies, а в секцію dev_dependencies файла pubspec.yaml:

```
dev_dependencies:
  build_runner: ^2.4.12
```

Для запуску процесу генерації кода використовується команда:

```
dart run build_runner build -d
```

Параметр -d вказує на необхідність видалити при необхідності старі сгенеровані файли.

Анотації починаються з символу @, за яким йде назва анотації з можливими параметрами.

Пакет json_serializable

Пакет json_serializable використовується для генерації методів fromJson та toJson, які створюють Dart-об’єкти з json-словників та представляють Dart-об’єкти в json-форматі відповідно.

Для використання цього пакету його треба додати в секцію dev_dependencies та допоміжний пакет json_annotation:

```
dependencies:
  flutter:
    sdk: flutter
  json_annotation: ^4.9.0
dev_dependencies:
  build_runner: ^2.4.12
  json_serializable: ^6.8.0
```

Клас, для якого треба сгенерувати методи fromJson та toJson, необхідно відмітити анотацією @JsonSerializable():

```
import 'package:json_annotation/json_annotation.dart';
part 'post.g.dart';

@JsonSerializable()
class Post {
```

```
  Post({
    required this.id,
    required this.userId,
    required this.title,
    required this.body,
  });

  factory Post.fromJson(Map<String, dynamic> json) => _$PostFromJson(json);
  Map<String, dynamic> toJson() => _$PostToJson(this);

  final int id;
  final int userId;
  final String title;
  final String body;
}
```

fromJson — це фабричний конструктор, який викликає сгенерований метод _$PostFromJson, метод toJson — сгенерований метод _$PostToJson. Код цих методів знаходиться в файлі post.g.dart, який включається в файл post.dart директивою part.

Сгенерований файл виглядає наступним чином:

```
part of 'post.dart';
Post _$PostFromJson(Map<String, dynamic> json) => Post(
      id: (json['id'] as num).toInt(),
      userId: (json['userId'] as num).toInt(),
      title: json['title'] as String,
      body: json['body'] as String,
    );

Map<String, dynamic> _$PostToJson(Post instance) => <String, dynamic>{
      'id': instance.id,
      'userId': instance.userId,
      'title': instance.title,
      'body': instance.body,
    };
```

Існує доволі багато анотацій для керування сгенерованим кодом. Наприклад, якщо поле в json-словнику не співпадає з назвою поля класа, використовується анотація @JsonKey:

```
@JsonKey(name: 'id')
final int postId;
```

Більше детально наявні анотації вказані в документації пакету.

Пакет retrofit

Пакет retrofit використовується для генерації коду, що за допомогою пакету dio виконує мережеві запити й обробку відповідей. Для його використання потрібно додати відповідні залежності у файл pubspec.yaml:

```
dependencies:
  flutter:
    sdk: flutter
  dio: ^5.5.0+1
  json_annotation: ^4.9.0
  retrofit: ^4.1.0
dev_dependencies:
  build_runner: ^2.4.12
  json_serializable: ^6.8.0
  retrofit_generator: ^8.1.2
```

Приклад опису класу, що використовується для запитів до https://jsonplaceholder.typicode.com/, зокрема отримання посту за його ідентифікатором (id):

```
import 'package:dio/dio.dart';
import 'package:retrofit/retrofit.dart';

import '../domain/post.dart';

part 'retrofit_client.g.dart';

@RestApi(baseUrl: 'https://jsonplaceholder.typicode.com/')
abstract class RetrofitClient {
  factory RetrofitClient(Dio dio) = _RetrofitClient;

  @GET('/posts/{id}')
  Future<Post> getPost(@Path() int id);
}
```

Пакет містить багато анотацій, таких як @GET, @POST, @DELETE, @UPDATE, тощо для опису відповідних http-запитів, а також @Path, @Body для формування тіл запитів.

Сгенеровані класи можуть використовуватись наступним чином:

```
Future<Post?> _fetchPost() async {
    final dio = Dio();
    final client = RetrofitClient(dio);
    final post = await client.getPost(1);
    print('Title: ${post.title}');
    return post;
```

```
}
```

Підготовка додатка для публікації в Google Play

Файл з Android-додатком має бути підписаний цифровим підписом для публікації його в Google Play. Файл підпису створюється утілітою keytool. Для створення файлу Linux або macOS треба виконати команду:

```
keytool -genkey -v -keystore ~/upload-keystore.jks -keyalg RSA \
        -keysize 2048 -validity 10000 -alias upload
```

В Windows файл створюється командою PowerShell:

```
keytool -genkey -v -keystore $env:USERPROFILE\upload-keystore.jks `
        -storetype JKS -keyalg RSA -keysize 2048 -validity 10000 -alias upload
```

Ці команди стоврюють файл upload-keystore.jks в домашньому каталозі користувача. Параметр alias задає спеціальний псевдонім, який треба вказувати при підпису. Також треба буде придумати пароль й ввести його на запит команди.

Інформація про цифровий підпис вказується в підсекції signingConfigs секції android в файлі android/app/build.gradle. Для того щоб, по-перше випадково не помістити конфеденційну інформацію в систему контроля версій (наприклад, git), а по-друге автоматизувати процес підпису додатку, поступають настйпним чином. В каталозі android створюють файл key.properties з інформацією для підпису:

```
storePassword=you_password
keyPassword=you_password
keyAlias=upload
storeFile=path_to_upload_keystore.jks
```

В файлі android/app/build.gradle цей файл зчитується в словник:

```
def keystoreProperties = new Properties()
def keystorePropertiesFile = rootProject.file('key.properties')
if (keystorePropertiesFile.exists()) {
    keystoreProperties.load(new FileInputStream(keystorePropertiesFile))
}
```

А потім отримана інформація вказується в секції підпису:

```
android {
...
signingConfigs {
        release {
            keyAlias keystoreProperties['keyAlias']
            keyPassword keystoreProperties['keyPassword']
            storeFile file(keystoreProperties['storeFile'])
            storePassword keystoreProperties['storePassword']
        }
    }
```

```
...
}
```

Android-додатки можуть розповсюджуватись у двох форматах: apk- або aab-файлів. Aab-формат більш сучасний і зручний. Для створеня aab-файлу використовується команда:

```
flutter build appbundle --tree-shake-icons
```

Параметр —tree-shake-icons вказує необхідність видалити зі створюваного файлу іконки, які не використовуються.

Apk-файл створюється командою:

```
flutter build apk --tree-shake-icon
```

Рекомендована література

- Flutter Apprentice (Second Edition): Learn to Build Cross-Platform Apps / Mike Katz, Kevin David Moore, Vincent Ngo – Вид-во: “Razeware LLC”, 2021. – 615 с.
- Flutter TutorialPoints / Вид-во: “Tutorials Point (I) Pvt. Ltd.”, 2019. – 185 с.
- Dart Apprentice / Matt Galloway, Jonathan Sande – Вид-во: “Razeware LLC”, 2020. – 436 с.
- Gilad Bracha. The Dart Programming Language– Вид-во: “Addison-Wesley”, 2016р. – 201 с.
- Carmine Zaccagnino. Programming Flutter: Native, Cross-Platform Apps the Easy Way (The Pragmatic Programmers) 1st Edition – Вид-во: “Pragmatic Bookshelf”, 2020. – 370 с.
- Dart Portal – Режим доступу: https://dart.dev/
- Flutter Portal – Режим доступу: https://flutter.dev/
- Нетворкінг у Flutter додатках — про просте і складне на прикладі Tide – Режим доступу: https://dou.ua/forums/topic/39051/
- Проект «Facts about numbers» з вручную написаним кодом – Режим доступу: https://github.com/xvadim/number_facts
- Проект «Facts about numbers» зі сгенерованим кодом – Режим доступу: https://github.com/xvadim/number_facts2