Hot Reload vs Hot Restart

في Flutter، هناك ميزتين رئيسيتين لتحديث التطبيق أثناء التطوير وهما: Hot Reload وHot Restart.

Hot Reload

تُستخدم ميزة Hot Reload لإجراء تحديث سريع للتطبيق دون الحاجة إلى إعادة تشغيله بالكامل. عند إجراء تغيير في الكود (مثل تعديل واجهة المستخدم أو تغيير النصوص (Texts)) وضغط زر Hot Reload، يقوم Flutter بتحديث الكود الذي تم تعديله فقط ويعيد تحميله مباشرة في التطبيق.

متي نستخدم ال Hot Reload؟

ال Hot Reload مثالي لإجراء تغييرات صغيرة وسريعة مثل التعديلات على الواجهات أو النصوص.

ال Hot Reload يُستخدم في تحديث واجهة المستخدم وإجراء تغييرات لا تؤثر على حالة (State) التطبيق.

النتائج:

يحتفظ التطبيق بحالته (State) السابقة، أي أنه إذا كنت في صفحة معينة وأجريت Hot Reload، ستبقى في نفس الصفحة وستظهر التغييرات دون فقدان بيانات أو إعادة تعيين الحالة.

Hot Restart

أما Hot Restart فهو يعيد تشغيل التطبيق من الصفر، ولكنه لا يقوم بعملية بناء كاملة مثل تشغيل التطبيق من جديد.

متي نستخدم ال Hot Restart؟

ال Hot Restart مفيد عندما تُجري تغييرات جوهرية في الكود مثل تغيير بعض الخصائص الثابتة أو التعديلات التي تؤثر على حالة (State) التطبيق بالكامل.

تُستخدم أيضًا عند وجود مشكلة أو خلل في التطبيق لا يمكن إصلاحه باستخدام Hot Reload فقط.

النتائج:

يقوم Hot Restart بإعادة تشغيل التطبيق وإعادة تعيين حالته (State) بالكامل، مما يعني أن جميع البيانات والحالات السابقة سيفقدها التطبيق ويبدأ من البداية.

مثال على استخدامات Hot Reload و Hot Restart:

استخدم الـ Hot Reload إذا كُنت تقوم بتغيير لون زر (Button) أو تعديل نص (Text) في الواجهة، فاضغط على Hot Reload لترى التغييرات فوراً.

استخدم الـ Hot Restart إذا قمت بتعديل القيمة الابتدائية لأحد المتغيرات الرئيسية التي تؤثر على التطبيق ككل، فيجب استخدام Hot Restart لكي تتأكد أن التغييرات فعّالة.

لبناء تطبيق Flutter وفهم كيفية بدء تشغيله، يمكننا تقسيم العملية إلى خطوات مرتبة وفقاً لشجرة (Tree) الـ Widgets والـ classes في الكود، حيث يتم تنفيذ الكود من الأعلى إلى الأسفل. لنبدأ بشرح هذه الخطوات بالتفصيل:

main()

دالة main() هي نقطة البداية في تطبيق Flutter، وهي الدالة التي يتم استدعاؤها أولاً عند تشغيل التطبيق.

داخل main()، يتم استدعاء runApp() مع تمرير MyApp كـ widget رئيسي.

ال runApp() تقوم بتحميل التطبيق بالكامل وتجعله جاهزاً للعمل.

مثال:

```
void main() {
  runApp(MyApp());
}
```

MyApp()

هي ال Root widget للتطبيق، والتي يبدأ من خلالها بناء شجرة (Tree) الـ widgets لعرض واجهة المستخدم. بمعنى آخر، ال MyApp هي المدخل الرئيسي للتطبيق، وتعمل كحاوية أولية تحتوي على بقية المكونات التي تشكل واجهة التطبيق.

مثال:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Home'),
        ),
        body: Center(
          child: Text('Hello World!'),
        ),
```

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

MaterialApp

هي widget رئيسية تُستخدم لتكوين التطبيق بالكامل بتصميم Material Design من Flutter.

ال MaterialApp يوفر العديد من الخصائص الأساسية مثل:

ال **home**: والتي تحدد الصفحة أو الواجهة الرئيسية للتطبيق.

ال **theme**: لاختيار الثيم أو التصميم الخاص بالألوان والعناصر الأخرى.

Scaffold

هي widget تُوفر هيكلًا أساسيًا لشاشات التطبيق، ويُعد بمثابة الحاوية لواجهة المستخدم.

Scaffold يوفر مناطق محددة مثل:

ال **appBar**: وهو شريط العنوان الذي يظهر في الجزء العلوي من التطبيق.

ال **body**: وهو محتوى الشاشة الرئيسي.

وال floatingActionButton وغيرها من العناصر المساعدة.

AppBar

هي widget تُمثل شريط العنوان العلوي في التطبيق، ويُستخدم غالبًا لعرض عنوان الشاشة أو الأيقونات.

Center

هي widget تُستخدم لتوسيط العناصر في الشاشة.

Text

هي widget تُستخدم لعرض النصوص (Texts) في واجهة المستخدم.

تسلسل التنفيذ من البداية إلى النهاية:

يتم استدعاء الدالة ()main، والتي تستدعي ()runApp وتبدأ تشغيل التطبيق.

يقوم ()runApp بتشغيل MyApp.

داخل MyApp، يتم إنشاء MaterialApp كعنصر رئيسي يمثل تطبيقًا بتصميم Material Design.

ال MaterialApp يحتوي على خاصية home، التي تحتوي على Scaffold، وهو العنصر الأساسي الذي يبني هيكل التطبيق.

داخل Scaffold، لدينا:

ال **AppBar** في الأعلى لعرض شريط العنوان.

ال **body** يحتوي على Center، الذي يقوم بتوسيط المحتوى.

ال **Center** يحتوي على widget Text يعرض كلمة "Hello World!" في منتصف الشاشة.

```
runApp(MyApp)
 └── MyApp
     └── MaterialApp
         └── Scaffold
             ├── AppBar
             │     └── title: Text('Home')
             └── body: Center
                   └── child: Text('Hello World!')
```

بهذا الشكل، يبدأ التطبيق من main() وينتهي عند عرض المحتوى المحدد في شاشة التطبيق.

في Flutter، الـ Widgets تُقسم إلى نوعين رئيسيين هما Visible و Invisible.

Visible Widgets

هي العناصر التي يمكن رؤيتها بشكل مباشر على واجهة المستخدم.

تتضمن عناصر مثل:

Text: لعرض النصوص.

Image: لعرض الصور.

Icon: لعرض الأيقونات.

Button: لعرض الأزرار التفاعلية.

هذه الـ Widgets تساهم في تكوين المظهر العام للتطبيق وتحديد العناصر المرئية للمستخدم.

Invisible Widgets

هي عناصر تُستخدم لتنظيم وتخطيط واجهة المستخدم، لكن لا يتم رؤيتها مباشرةً كعناصر.

تتضمن Widgets مثل:

Column: لترتيب العناصر عموديًا.

Row: لترتيب العناصر أفقياً.

Container: للتحكم في المسافات، والتنسيقات، وتطبيق الخلفيات.

Stack: لوضع العناصر فوق بعضها.

هذه الـ Widgets تساعد في تنظيم شكل واجهة المستخدم وتوزيع العناصر، لكنها لا تظهر بنفسها كمكونات مرئية.

Column

هي Widget من نوع Invisible، تُستخدم لترتيب العناصر عموديًا، بحيث يتم عرض كل عنصر أسفل الآخر.

تحتوي على خاصية children، التي تقبل قائمة من الـ Widgets التي سيتم ترتيبها عموديًا داخل الـ Column.

Row

هي أيضًا Widget من نوع Invisible، تُستخدم لترتيب العناصر أفقيًا، بحيث يتم عرض العناصر بجانب بعضها البعض.

مثل الـ Column، تحتوي على خاصية children لاستقبال قائمة من العناصر التي سيتم ترتيبها أفقيًا.

Center

هي Widget من نوع Invisible تُستخدم لوضع العناصر في وسط الشاشة.

تُعتبر حاوية (Container) تقوم بتوسيط الـ child الخاص بها ضمن المساحة المتاحة.

TextButton

هي Widget تفاعلية من نوع Visible، تُستخدم لإنشاء زر يظهر كنص بسيط.

تحتوي على خاصية onPressed لتحديد دالة (Function) تُنفذ عند الضغط على الزر (Button).

تتضمن أيضًا خاصية child، التي تُحدد العنصر الذي سيتم عرضه داخل الزر (Button)، وغالبًا يكون Text.

ElevatedButton

هي أيضًا Widget تفاعلية من نوع Visible، تُستخدم لإنشاء زر (Button) مرتفع (بارز) و تُعطي تأثيرًا ثلاثي الأبعاد.

تحتوي على خاصية onPressed لتحديد دالة (Function) تُنفذ عند الضغط على الزر (Button)، مثل TextButton.

تتضمن خاصية child لعرض محتوى داخل الزر (Button)، ويمكن أن يكون Text أو أي عنصر آخر.

child Vs children

ال **child** تستخدم لاحتواء عنصر واحد داخل الـ Widget. تُستخدم عندما تحتاج الـ Widget لاستقبال عنصر واحد فقط. مثال: في TextButton و ElevatedButton، يتم استخدام child لتحديد العنصر المعروض داخل الزر (Button).

ال **children** تُستخدم لاحتواء قائمة من العناصر (List) داخل الـ Widget. تُستخدم عندما يمكن للـ Widget استقبال أكثر من عنصر، مثل Column و Row.

لربط الدوال (Functions) مع الأزرار (Buttons) في Flutter، يمكنك الاستفادة من خاصية onPressed في Button Widgets مثل TextButton وElevatedButton. هذه الخاصية تتطلب دالة (Function) تُنفذ عند الضغط على الزر (Button).

مثال:

تعريف الدالة (Function):

```
void printWord() {
  debugPrint('I am pressed!!!');
}
```

استخدام الدالة (Function) مع الزر(Button):

مثال:

```
ElevatedButton(
  onPressed: printWord, // سيتم استدعاء printWord عند الضغط على الزر
  child: Text('Submit'),
),
```

مثال:

```
void printHello(int num) {
  debugPrint('Hello World!');
  debugPrint('This is the number $num');
}
ElevatedButton(
                                    // استدعاء printHello وتمرير 5 كمعامل
  onPressed: () => printHello(5),
  child: Text('Submit'),
),
```

إذا كانت الدالة (Function) تحتوي على معاملات (Parameters)، يمكنك استخدام Anonymous function لتمرير قيمة (Value).

Anonymous Functions

في Flutter، تُستخدم الـ Anonymous Functions لتنفيذ مهام بسيطة بدون الحاجة لتسمية الدالة (Function). تُعتبر مفيدة خاصةً عند تمرير الدوال (Functions) كمعاملات (Parameters) لـ Widgets مثل الأزرار.

يوجد طريقتان رئيسيتان لكتابة الـ Anonymous Functions في Flutter باستخدام => أو {}.

استخدام =>

هذا الأسلوب مناسب إذا كانت الدالة (Function) تحتوي على تعبير (Statement) واحد فقط.

مثال:

```
ElevatedButton(
  onPressed: () => print("Button pressed!"),
  child: Text("Click Me"),
);
```

هنا، يتم تنفيذ print("Button pressed!") عند الضغط على الزر.

استخدام الأقواس {}

يُستخدم هذا الأسلوب إذا كانت الدالة (Function) تحتاج إلى عدة تعليمات برمجية (Statements).

مثال:

```
ElevatedButton(
  onPressed: () {
    print("Button pressed!");
    print("Another action executed.");
  },
  child: Text("Click Me"),
);
```

في هذا المثال، تُنفذ التعليمات داخل الأقواس {} عند الضغط على الزر (Button). يسمح لنا هذا الأسلوب بإضافة أي عدد من التعليمات البرمجية (Statements) داخل الدالة (Function).

Stateless Widgets Vs Stateful Widgets

Stateless Widgets

هي نوع من الـ Widgets التي لا تحتوي على State داخلي خاص بها. بمجرد أن يتم إنشاؤها، فإنها لا تتغير ولا تعيد بناء نفسها تلقائيًا. تُستخدم الـ Stateless Widgets عندما لا تحتاج واجهة المستخدم إلى التفاعل مع المستخدم أو إلى التحديث بعد الإنشاء الأول.

الخصائص:

Input Data → Widget → Renders UI

تأخذ بيانات المدخلات (Input Data).

ترسم نفسها على الشاشة مرة واحدة عند إنشائها.

أمثلة على Stateless Widgets:

Text

Icon

Image

Stateful Widgets

هي نوع من الـ Widgets التي تحتوي على State داخلي يمكن تغييره. عندما يتغير الـ State، تُعيد الـ Stateful Widget بناء نفسها تلقائيًا لتعرض التغيير في واجهة المستخدم. تُستخدم Stateful Widgets عندما تحتاج واجهة المستخدم إلى التفاعل أو إلى التحديث بشكل مستمر.

الخصائص:

Input Data → Widget/Internal State → Renders UI

تأخذ بيانات المدخلات وتحتوي على State داخلي.

تُعيد بناء نفسها عند تحديث الـ State.

يُمكنك التحكم في الـ State من داخل الـ Widget نفسها، مما يسمح بإنشاء واجهات ديناميكية تتفاعل مع المستخدم.

أمثلة على Stateful Widgets:

Checkbox

TextField

في Flutter عند استخدام Stateful Widget، تُستخدم setState() لتحديث البيانات (الـ State) وإعادة بناء واجهة المستخدم (UI) لتعرض القيم التي تم تحديثها.

ال setState() هي الدالة (Function) المسؤولة عن إخبار Flutter بأن هناك تغييرًا في الـ State يتطلب تحديث واجهة المستخدم (UI).

مثال عملي - عداد (Counter) باستخدام setState

في هذا المثال، سنقوم بإنشاء عداد (Counter) بسيط يزيد عدده كلما ضغط المستخدم على زر (Button). كلما ضغط المستخدم على الزر (Button)، يتم تحديث قيمة العداد (Counter) باستخدام setState()، مما يؤدي إلى إعادة بناء واجهة المستخدم (UI) لتظهر القيمة الجديدة.

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

void main() => runApp(MaterialApp(

  home: CounterApp(),

));

class CounterApp extends StatefulWidget {

  @override
```

```
  _CounterAppState createState() => _CounterAppState();
}
class _CounterAppState extends State<CounterApp> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++; // زيادة قيمة العداد بمقدار 1
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Counter Example')),
      body: Center(
        child: Column(
```

```
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      Text(
        'You have pressed the button this many times:',
      ),
      Text(
        '$_counter',
        style: TextStyle(fontSize: 40, fontWeight: FontWeight.bold),
      ),
    ],
  ),
),
floatingActionButton: FloatingActionButton(
  onPressed: _incrementCounter,
  child: Icon(Icons.add),
```

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

شرح المثال:

قمنا بتعريف متغير counter_ في CounterAppState_ ليخزن قيمة العداد (Counter) الحالية، وبدأناه بالقيمة 0.

الدالة ()incrementCounter_ مسؤولة عن زيادة قيمة المتغير counter_ بمقدار 1 في كل مرة يتم استدعاؤها.

نستخدم ()setState داخل الدالة لتحديث قيمة counter_.

ثم نقوم بزيادة قيمة العداد (Counter)، ثم نطلب من Flutter إعادة بناء واجهة المستخدم لتحديث البيانات.

زر FloatingActionButton يحتوي على أيقونة +، وعند الضغط عليه، يتم استدعاء()incrementCounter_، مما يؤدي إلى زيادة العداد (Counter) وإعادة بناء واجهة المستخدم.

في Flutter، يمكنك استخدام mainAxisAlignment و crossAxisAlignment لتحديد كيفية توزيع وترتيب العناصر داخل Row أو Column.

mainAxisAlignment

ال mainAxisAlignment يتحكم في توزيع العناصر على طول المحور الأساسي (Main Axis) لـ Row أو Column. بالنسبة لـ Row، يكون المحور الأساسي أفقيًا، أما بالنسبة لـ Column فيكون عموديًا.

قيم mainAxisAlignment:

start

يجعل العناصر تصطف في بداية المحور الأساسي.

end

يجعل العناصر تصطف في نهاية المحور الأساسي.

center

يجعل العناصر تصطف في منتصف المحور الأساسي.

spaceBetween

يضع المسافات بالتساوي بين العناصر بدون مسافة في البداية والنهاية.

spaceAround

يضع مسافات بالتساوي حول كل عنصر، مع اضافة نصف المسافة التي بين العناصر في البداية والنهاية.

spaceEvenly

يوزع العناصر مع مسافات متساوية بينهم، بما في ذلك البداية والنهاية.

crossAxisAlignment

ال crossAxisAlignment يتحكم في توزيع العناصر على طول المحور المتقاطع (Cross Axis) للـ Row أو Column. بالنسبة لـ Row، يكون المحور المتقاطع عموديًا، أما بالنسبة لـ Column فيكون أفقيًا.

قيم crossAxisAlignment:

start

يجعل العناصر تصطف عند بداية المحور المتقاطع.

end

يجعل العناصر تصطف عند نهاية المحور المتقاطع.

center

يجعل العناصر تصطف في منتصف المحور المتقاطع.

stretch

يجعل العناصر تمتد لتشغل المساحة المتاحة على طول المحور المتقاطع.

baseline

وهي (لـ Row فقط، وتجعل العناصر تصطف على طول الخط الأساسي للنص.

مثال:

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

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

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      home: Scaffold(
```

```
      appBar: AppBar(title: Text("Alignment Example")),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.spaceAround,
crossAxisAlignment: CrossAxisAlignment.center,
        children: <Widget>[
          Container(color: Colors.red, width: 50, height: 50),
          Container(color: Colors.green, width: 50, height: 50),
          Container(color: Colors.blue, width: 50, height: 50),
        ],
      ),
    ),
  );
}
}
```

يمكنك تغيير القيم في mainAxisAlignment و crossAxisAlignment لتجربة كيفية توزيع العناصر في التطبيق.

Model

ال Model هو فئة (Class) يُستخدم لتمثيل كائن (Object) يحتوي على البيانات وانواع البيانات التي يتم التعامل معها في التطبيق. يتم تعريف النموذج (Model) بشكل أساسي لتبسيط إدارة البيانات وتسهيل نقلها بين واجهة المستخدم (UI) والطبقات المختلفة للتطبيق.

كيفية إنشاء نموذج (Model) في Flutter

لإنشاء نموذج (Model)، نبدأ بتعريف فئة (Class) تحتوي على الخصائص المطلوبة. إليك مثال عن كيفية إنشاء نموذج AnswerItemModel:

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

class AnswerItemModel {

  final String title;

  final VoidCallback onPressed;

  AnswerItemModel({required this.title, required this.onPressed});

}
```

كلمة final تعني أن القيم ثابتة ولا تتغير بعد التهيئة (Initialization).

كلمة required تشير إلى أن هذه القيم مطلوبة.

بهذا الشكل، يمكنك استخدام AnswerItemModel في التطبيق.

في Flutter، تُعتبر الكلمة const مهمة لتحسين الأداء. عندما تستخدم const مع الكائنات (Objects) التي لا تتغير، يخبر ذلك Flutter أن يقوم بإنشاء هذا الكائن (Object) مرة واحدة فقط وتخزينه في الذاكرة. هذا يعني أنه إذا كان الكائن (Object) ثابتًا (لا يتغير)، فإن Flutter لا يحتاج إلى إعادة بنائه في كل مرة يُعاد فيها بناء الواجهة.

في حالة النص أو TextStyle الذي لن يتغير، يمكنك وضع const قبله ليتم إنشاؤه مرة واحدة فقط:

```
const Text(
  'Congratulations!',
  style: TextStyle(
    fontSize: 36,
    fontWeight: FontWeight.w600,
  ),
),
```

بهذه الطريقة، Flutter لن يعيد بناء هذا النص أو هذا عند إعادة بناء الواجهة، مما يحسن الأداء بتقليل العمليات الزائدة و استهلاك الذاكرة.