Flutter 개념 심화 (코딩 쉐프 영상 참조)

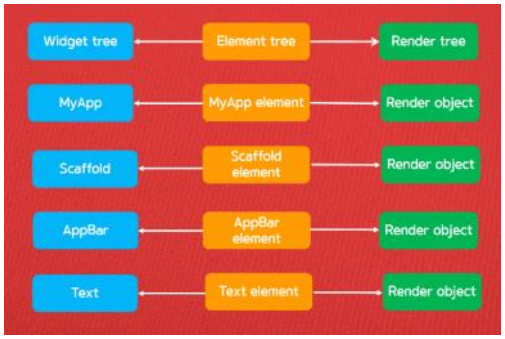
2020년 7월 8일 수요일 오전 1:08

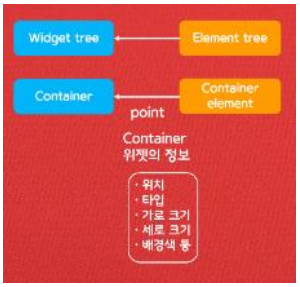
Stateless 위젯 & State

- **Flutter에서 State의 의미**는 "**UI가 변경되도록 영향을 미치는 데이터**"이다.
- App 수준(사용자 인증 or 서버에서 가져온 데이터)과 Widget 수준(radio 버튼 선택 or 텍스트 입력 유무 등 App의 상태를 바꾸는 모든 행위들)의 데이터가 있다.

- **Stateless widget**은 "**State가 변하지 않는 위젯**"임.
  즉, 한번 지정된 레이아웃이나 텍스트, 컬러 정보 들은 앱 자체가 rebuilding 되지 않는 한 바뀌지 않는다.

Flutter에서의 Tree 종류





- **Widget-Tree는 코드 상에서 언제든지 제어가 가능**하다.
  하지만 **Element-Tree 와 Render-Tree는 Flutter가 내부적으로 제어하는 Tree**이다.
  그리고 **이 2개의 Tree(Element & Render)는 개발자가 생성한 Widget-tree에 근거해서 생성**된다.

- **Widget-Tree는 개발자가 작성한 코드에 근거**해서 **Flutter가 build메소드를 호출해서 생성**한다. 그러나 **이 Widget-tree들은 하나의 구성일 뿐, 직접적으로 앱 화면에 그려지지 않는다.**

  Widget-Tree는 하나의 설계도로써, **Flutter에게 앱 화면에 개발자가 생성한 Widget-Tree 순서와 모양에 따라 UI를 보여줘라 요청하는 것**임.

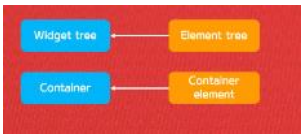
- 중요한 것은 **Element-Tree가 가장 중요**하다. 이 **Element-Tree는 Widget-Tree와 Render-Tree를 연결**해준다. **Element-Tree는 Flutter가 자동으로 개발자가 만든 Widget-Tree에 근거해서 Widget을 생성**해준다. 그리고 **Element-Tree는 Widget-Tree의 모든 위젯과 1:1 관계로 형성**된다.

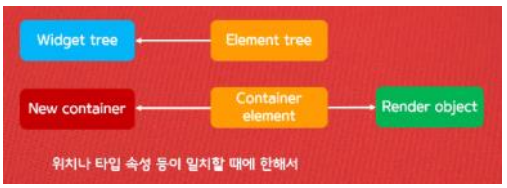
  Render-Tree는 직접적으로 앱 화면에 UI를 그리는 역할을 수행한다. Element-Tree는 Widget-Tree와 Render-Tree 중간에서 Render-Tree가 실질적으로 Rendering 하기 위해서 가지고 있는 Render 객체와 1:1 관계로 형성된다.
  우리가 눈으로 확인가능한 화면의 작업 결과들은 전부 Render-Tree의 작업 결과이다.

- 즉, Flutter는 개발자가 **Widget-Tree를 생성될 때마다, 즉시 Element-Tree도 함께 생성**한다. 정리하자면, **Element라는 것은 결국 위젯들의 정보를 담고있는 하나의 객체일 뿐**이며, **Element도 메모리에 저장되어 Flutter에 의해 사용되는 요소**이다.

- Stateless 위젯 내에 Container 위젯이 있다고 가정한다면, Flutter는 즉시 Container 위젯에 대응하는 Container element를 Element-Tree 상에 Container를 추가하고, Widget-Tree상의 Container 위젯을 가리키게 된다. (다만 가리키는 위젯에 대한 정보(위치, 타입, 가로세로 크기, 배경색 등)도 함께 가지고 있게 된다. (포인터와 유사한 개념)

Reload 와 Rebuild의 차이





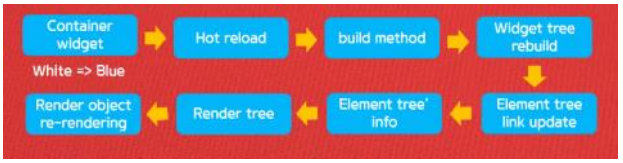
- **Widget-Tree는 build 메소드가 호출될 때마다 새롭게 Rebuild** 된다.
  Reload는 어떤 프레임은 그대로 둔 채, 부수적인 요소들만 바꾸는 것.
  Rebuild는 어떠한 요소를 바꾸기 위해 바꾼 요소를 포함하여 전체를 재구성하는 것.

  # 만약 Element-Tree가 중개자 역할을 하지 않는다면? #
  Widget-Tree가 매번 Rebuild된다는 것은 Widget-Tree 자체도 Stateless 처럼 한번 생성되면 요소들이 변경될 수 없으므로 요소들을 변경시키려면 새로 만들어야 한다는 것을 의미하며, Widget-Tree 내의 모든 Widget들도 Rebuild 된다는 의미이다.

  만약, 문자 하나 바꾸기 위해서 Hot-Reload를 사용하면, 매번 Build 메소드가 호출되면서 Widget-Tree가 Rebuild 될 것이며, 그리고 Widget-Tree와 연결된 Element-Tree와 Render-Tree도 Rebuild될 것이다.

  => 그렇기 때문에 Element-Tree가 Widget-Tree에서 대응되는 widget에 대한 포인터를 가지고 있다가 Widget-Tree 상의 widget을 변경시킨 후, Rebuild 하면 기존처럼 Widget-Tree와 Element-Tree, Render-Tree까지 모두 Rebuild 되는 것이 아니라,
  Element-Tree에서 새롭게 생성(변경)된 Widget을 가리키게 된다.
  => 아마 내부구조가 Linked-list 방식으로 동작하지 않을까 싶다...(개인적인 내 생각)

  그리고 **Element-Tree가 Render-Tree에서 대응되는 widget에게 Widget-Tree 내의 Widget의 변경된 부분에 대한 데이터를 전달해줌**으로써, **Render object는 바뀐 부분만 Rendering 하게 된다.**



- 정리하자면, **Hot-Reload 기능이 사용될 때마다** 모든 위젯들을 화면에 다시 그리는 것이 아닌, **Element-Tree를 활용해서 변경된 부분만을 다시 그리는 방식을 사용**하므로, **빠르고 효율적**이다.

  또한, **Stateless 위젯은 한번 Build되면 절대로 State가 변경되지 않으므로, 새로운 State를 적용하려면 Rebuild만을 통해서 적용이 가능**하다.

  예를 들면, Stateless 위젯인 Text 위젯의 내용을 바꿀 때도 생성자를 통해서 외부에서 데이터를 전달하는 것일 뿐, Text 위젯 내부의 State가 바뀌는 것이 아니다.
  => 그리고 이러한 효율적인 시스템으로 인해 Flutter는 초당 60프레임 속도로 변경된 내용을 보여주기 때문에 개발자는 Hot-Reload 사용 시, 변경된 부분을 바로 확인이 가능한 것임.

Stateful 위젯과 StateLess 위젯의 공통점 & 차이점

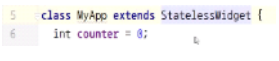
- **공통점 :**
  => **두 위젯 모두 생성자를 통해서 외부에서 데이터가 입력이 되면**, 그 결과를 반영하기 위해서 **Build 메소드가 호출**되면서 **widget들이 Rebuild**되고 **필요한 부분의 UI를 다시 렌더링**하게 된다.

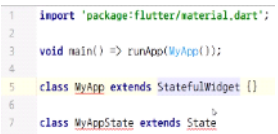
- **차이점 :**
  => 하지만 **Stateful 위젯은 내부에 State라는 클래스를 가지고 있다. 즉, 두 개의 클래스가 결합이 되어서 Stateful 위젯을 형성하는 것**이다.
  Stateful 위젯에서 build 메소드는 stateful 내부의 클래스인 State 클래스가 가지고 있다.
  이 말은, **Stateful 위젯의 Rebuild를 하는 주체는 State 클래스**이며, **이 과정을 통해 화면의 UI를 다시 렌더링** 하게 된다.

- ***Staeful widget 이 Rebuild 되는 2가지 경우***

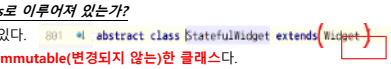
1) Child 위젯의 생성자를 통해서 데이터가 전달 될 때
2) Internal state가 변경될 때

Stateless 위젯을 Stateful 위젯으로 바꾸는 과정



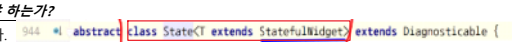


- **Stateful Widget은 State와 결합된 형태**이므로, MyApp에 대한 State클래스인 MyAppState 클래스를 정의하고 State클래스를 상속받는다.

- ***<u>의문점 1 : 왜 Stateful Widget은 두 개의 class로 이루어져 있는가?</u>***
  => Stateful Widget의 소스코드를 보면 알 수 있다. 

  보다시피, **Widget 클래스는 기본적으로 Immutable(변경되지 않는)한 클래스**다.
  <u>즉, 한번 생성되면 State가 변하지 않는다.</u>
  그래서 **Stateful Widget 클래스를 상속받은 MyApp 위젯은 Stateful Widget임과 동시에 Stateless Widget 처럼 Immutable한 특징을 가지게 된다.**

  하지만 **Stateful Widget은 반드시 State의 변화를 반영해야 하므로**, 이 문제를 해결하기 위해 **2개의 클래스로 나누어서 Stateful Widget인 MyApp 위젯은 Immutable한 특징을 유지**하고 **MyAppState 클래스는 Mutable한 특징을 가지도록 구성한 것**이다.

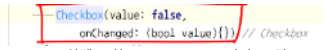
- ***<u>의문점 2 : 어떻게 이 2개의 클래스를 연결해야 하는가?</u>***
  => State 클래스의 소스코드를 보면 알 수 있다. 

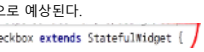
  보다시피, **Generic 타입으로 구성**되어 있다. **어떤 Type이 오던지 그 Type은 Stateful Widget을 상속받게 되어 있다.**

  MyAppState 클래스는 State 클래스를 상속받았고, 이것은 MyAppState 클래스는 State 클래스 Type이 되었다는 의미이다.

  그리고 상속받은 State 클래스의 Generic Type을 MyApp 클래스로 지정해준다면 이 State 클래스는 오직 MyApp 클래스 Type만 가질 수 있게 된다.

  정리하자면, MyAppState 클래스가 StatefulWidget인 MyApp 위젯에 연결되는 것이라고 Flutter에게 알려줄 수 있게 된다.

- ***<u>의문점 3 : 왜 State 클래스는 Generic Type을 가지게 되었을까?</u>***
  => **코드의 재사용성과 안정성 때문**이다.
  => 앱을 만드는 과정에서 여러 개의 Stateful 위젯을 사용하게 될 경우, 그때마다 함께 결합하는 State 클래스가 필요할 것이고, 이때 State 클래스를 Generic Type으로 사용한다면 연결이 필요한 Stateful 위젯을 선택해서 지정만 해주면 간단히 두 클래스가 연결되기 때문이다.
  

  예를 들면, Checkbox 위젯을 생성했고, 이 Checkbox 위젯 또한 Stateful Widget이다. 그럼 당연하게도 2개의 클래스로 이루어져 있을 것으로 예상된다.
  Checkbox의 소스코드를 보자. 

  그림과 같이 Stateful Widget을 상속받는 것을 볼 수 있다.

  State 클래스를 Generic Type으로 만들어서 필요한 Type을 편하게 지정하고,
  그 외의 Type은 전달받지 않도록 안전장치를 해 놓은 셈이다.

이렇게 고침으로써, 이제 MyApp 클래스와 MyAppState의 결합 1단계는 된 것이다.

```
class MyApp extends StatefulWidget {}

class MyAppState extends State<MyApp>{
```

다음 단계는 MyApp 위젯 내에서 CreateState 메소드를 호출해야 한다.
보다시피, **createState 메소드는 State 타입으로 지정**되어 있으며,
이 State는 Generic 타입으로 Stateful Widget이 지정되어 있다.

=> **createState 메소드는 반드시 State 타입의 객체를 return** 해야 하는데
결국 Stateful Widget 타입만이 올 수 있는 객체이여야 하며,
이 객체는 MyAppState 클래스에 근거해서 만들어진 것을 의미한다.

```
class MyApp extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    // TODO: implement createState
    return MyAppState();
  }
}

class MyAppState extends State<MyApp>{
```

그리고 **createState 메소드는 StatefulWidget이 생성될 때 마다 호출되는 메소드**이다.

정리하자면, **Stateless 위젯은 build 메소드에서 생성한 객체를 바로 return** 하지만,
Stateful 위젯은 createState 메소드에서 생성한 객체를 return 한다.

- ***하지만 여전히 에뮬레이터에서 카운터 증가 버튼을 눌러도 화면 상에서는 증가되지 않는다. 이유가 무엇일까?***
  => **출력을 담당하는 Text 위젯은 Stateless 위젯**이므로, **State를 변화시키려면 반드시 build 메소드를 호출해서 위젯을 Rebuild 하는 방법 밖에 없다.**

```
Text(
  '$counter',
```

- ***그럼 어떻게 버튼을 누르면 변화할 수 있도록 할 수 있는가?***
  => Flutter가 제공하는 **setState 메소드를 호출**한다. 이 setState 메소드 역할은 2가지이다.
  첫째 : **매개변수로 전달된 함수를 호출**하는 것
  둘째 : **build 메소드를 호출**하는 것

- setState를 사용하여 소스코드 변경

```
floatingActionButton: FloatingActionButton(
  child: Icon(Icons.add),
  onPressed: (){
    counter++;
    print("$counter");
  }), // FloatingActionButton
```

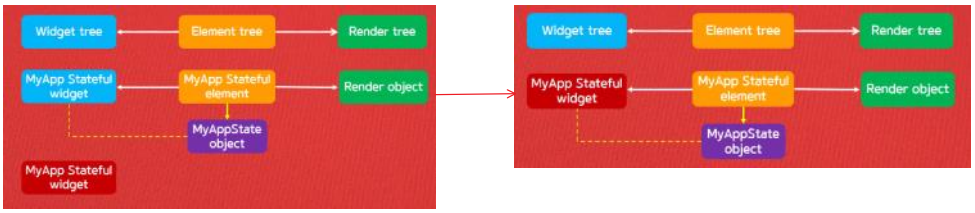
```
floatingActionButton: FloatingActionButton(
  child: Icon(Icons.add),
  onPressed: (){
    setState(() {
      counter++;
      print("$counter");
    });

  }), // FloatingActionButton
```

- 이제 버튼을 누르면 숫자가 1씩 증가할 것이고, 이를 감지한 setState 메소드가 호출된다. 이때 setState 메소드는 counter 변수의 숫자가 증가함에 따라 counter 변수를 사용하고있는 위젯들을 표시해주게 된다.

  정리하자면, **숫자의 증가로 State가 변했고 이 변화를 반영해서 다시 Rebuild 해야 하기 때문에 setState 메소드가 호출되어 기능을 수행**하게 되는 것이다.

```
Text(
  '$counter',
  style: Theme.of(context).textTheme.display1,
), // Text
```

- Flutter에서는 **setState가 표시해준 이러한 위젯들을 "dirty"라고 표현**한다.
  위 코드에선 Text 위젯이 dirty가 된다.



- **createState 메소드를 호출**해서 **MyApp Stateful element와 연결된 MyAppState 객체도 생성**한다. 이 MyAppState 객체는 **MyApp Stateful widget과 간접적으로만 연결**되어 있다.

  MyApp Stateful element는 위젯 관련 중요 정보를 가지고 있지만, 메모리 상에서 어디에도 종속되지 않은 **독립된 객체인 MyAppState 객체에 대한 정보도 가지게 된다.**

  이제 setState 메소드가 호출되고 build 메소드로 인해 State 객체가 Rebuild 되면서 **최종적으로 새로운 State를 반영한 새로 생성된 MyApp Stateful widget이 Rebuild** 된다.

- ***왜 MyAppState 객체는 MyApp Stateful widget 처럼 widget-tree 상에서 매번 Rebuild 되지 않는 걸까?***
  => Flutter가 변경된 State 객체를 비용이 저렴한 Stateful widget으로 만들어서 Rebuild하고 MyAppState 객체는 element-tree에서 mutable 특징을 가진 채로 존재하면서 필요할 때마다 새로운 State를 저장함과 동시에 새롭게 Rebuild된 Stateful 위젯과의 연결만을 업데이트 한다.

- 그러면 **MyApp Stateful element에 연결 되어있는 MyAppState 객체에 새로운 State 객체가 저장**이 되고, 이제 **MyAppState 객체는 새롭게 Rebuild된 MyApp Stateful widget을 가리키게 된다.**

final & const 차이

- **Final 변수**는 **런타임 때 초기화**되며, 한번 초기화 된 후에는 값을 변경할 수 없음
  => 새로운 값으로 변경하려면 build 메소드를 통해 Rebuild 되어야 함.

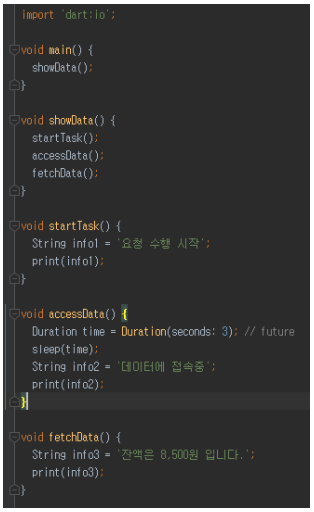
- **Const 변수**는 런타임이 아닌 **컴파일 시, 초기화(상수화)** 된 이후에는 값을 변경할 수 없음.
  선언과 동시에 초기화 되어야 함. 컴파일 이후 런타임 때는 컴파일 시 초기화한 값으로 유지.
  => 런타임 시에도 반드시 값이 유지되어야 할 변수는 const로 사용한다.

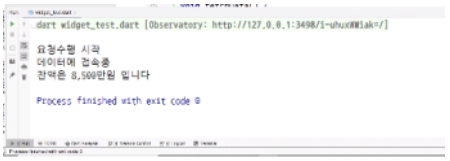
Future / Async / await 개념

- Future 클래스는 비동기 작업을 할 때 사용한다.
- Future는 일정 소요시간 후에 실제 데이터나 에러를 반환
- Async 클래스는 await 메소드를 가지고 있음
  => await로 선언된 메소드는 응답이 처리될 때 까지 대기.
- FutureBuilder<>는 데이터를 전부 받아 오기전에, 데이터 없이 화면에 그릴 수 있는 UI를 화면에 그려준다.

- const를 위젯에 적재적소 잘 사용하면, rebuild 시, const를 사용한 위젯은 다시 화면에 그려주지 않으므로 고정적인 UI요소들에 사용하여 선택적으로 rebuild를 사용하지 않음으로써, 성능을 향상 시킬 수 있다.
- 정리하자면, Future는 어떤 작업의 결과값을 나중에 받기로 약속한 것이다. 즉 요청한 작업의 결과를 기다리지 않고 바로 다음 작업으로 넘어가면서 작업하다가 future를 사용한 작업이 완료되면 결과를 받는 방식의 비동기 처리를 한다.

  하지만, 작업이 완료될 때까지 기다렸다가 결과값을 받고 다음 작업으로 넘어가야 할 경우도 있을 것이다. 그럴 경우, async와 await를 사용하면 된다.

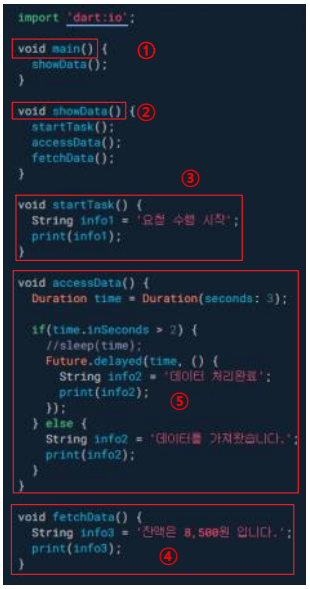


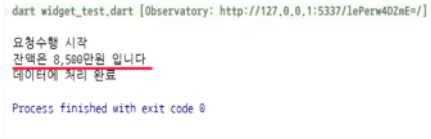


=> Duration 옵션에 3초를 부여하고, sleep()함수를 사용에 time값(3초)를 전달한 후, 실행한 결과이다.

startTask()함수가 수행된 후, 3초 뒤, accessData() 함수가 호출되어 수행되고, 이어서 fetchData()함수가 수행된다.

즉, accessData함수에 약간의 딜레이를 부여하였지만 결국은 모든 함수가 동기적(절차적)으로 수행된 것을 볼 수 있다.





=> 실행순서는 다음과 같다.

1) Main함수가 실행이 되고, showData()함수가 실행이 된다.
2) showData함수 내의 startTask함수가 실행이 된다.
3) accessData함수가 실행이 된다.
4) accessData함수 내에 있는 Future.delayed함수를 만나면 인자 값으로 전달된 time값 동안 Future.delayed함수의 실행을 멈추고 다음 실행 분기인 fetchData함수로 실행흐름이 넘어간다.
5) FetchData함수 실행 후, 다시 Future.delayed로 넘어와서 작업을 수행한다.

- Future 클래스는 Generic 처럼 원하는 자료형을 지정할 수 있는데, 위 dart 코드를 보면 fetchUserOrder함수는 Future<void>형으로 되어있다.

  즉, 일반 void형 함수와의 차이점은 시간을 지정해서 지정 시간동안 Future.delayed 함수가 실행되고 나서 결과를 return한다는 것 뿐이다.

  위 코드에서는 Future.delayed(Duration(seconds: 2)를 사용하여 Future.delayed함수를 2초간 실행하게되고, 2초 후에 print('Large Latte')를 수행하여 결과를 return 하도록 한 것이다.

Isolate 개념

- Flutter App을 실행하게 되면 Isolate라는 새로운 쓰레드가 생성되면서 이 쓰레드가 해당 앱을 전체를 수행(총괄)한다.
  Why? => Dart라는 언어 자체가 애초에 설계되었을 때, 단일 스레드 방식으로 동작하도록 설계된 언어이고, 이 Dart언어와 Flutter 프레임워크를 사용하여 만든 App을 동작시키는 것이니까.

- App을 실행하면 Dart 언어의 내부 동작 방식은 어떻게 되는가??

1) 단일 isolate 스레드가 생성되고 Dart는 내부적으로 FIFO방식인 Event-Queue를 생성하고, Event들을 적재한다.

2) Micro-Task(Event Loop의 실행이 Event-Queue로 넘어가기 전에 짧은 시간동안 비동기적으로 먼저 실행되고 끝나는 작업들)을 수행한다.

3) 더 이상 내부적으로 수행할 Micro-Task가 존재하지 않는다면, Event Loop는 각종 event들(gesture, drawing, Reading files, Fetching Data, Button tap, Future, Stream 등)을 Event-Queue에 적재(Event와 관련된 코드들이 Event-Queue에 등록)한다.

4) main 함수를 실행한다.

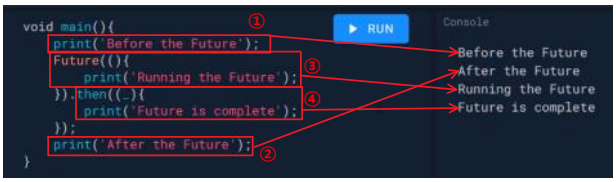
5) Event loop를 실행하여 Event-Queue에 적재된 Event들을 큐에 적재된 순서대로 처리.

Future / Async method 개념

- 비동기 방식으로 미래의 어느 시점에 완성되어서 실제적인 데이터가 되거나 에러를 반환하는 객체.

- ***Future 객체의 동작 방식***

1. Dart에 의해서 future 객체가 내부적인 배열에 등록됨
2. Future를 사용하여 수행되는 코드들이 Event-Queue에 등록됨
3. 불완전한 future 객체가 반환(비동기 연산의 결과를 받기 전에 객체 반환)
4. Synchronous(동기) 방식으로 실행되어야 할 코드 먼저 실행
5. 최종적으로 실제적인 data값이 future 객체로 전달.



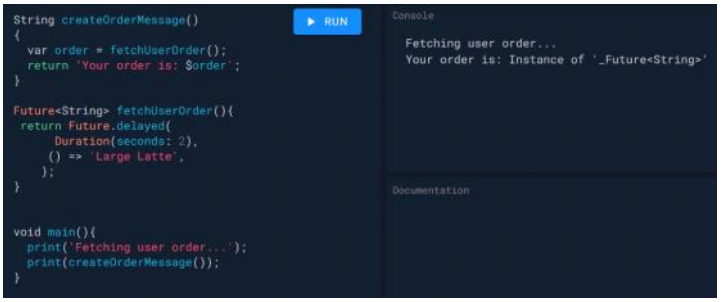
 # 코드 동작 과정 #

1) print('Before the Future'); 출력
2) Future는 비동기로 처리되는 이벤트 이므로 관련 코드가 Event-Queue에 등록되며, 잠시 후에 처리되기 위해 건너뛴다.
3) Then() 도 Future가 수행된 후에 실행되어야 하므로 건너뛰게 된다. 하지만 Then()은 Event가 아니므로, Event-Queue에 등록되지는 않는다.
4) 동기방식으로 처리되는 마지막의 print()함수를 실행하여 2번째로 출력되게 된다.

- 정리 : Future는 Event Loop를 통해서 Event Queue에서 FIFO방식으로 처리되는 하나의 Event이다.

- ***Async Method의 동작 방식***

1. Async 메소드를 통해서 나오는 결과물은 Future이다.
2. await 키워드를 만날 때까지 Synchronous(동기) 방식으로 코드 처리
3. await 키워드를 만나면 future가 완료될 때까지 대기
4. Future가 완료되면 바로 다음 코드들을 실행한다.



 # 코드 해석#

1) Main 함수가 실행되고, print('Fetching user order...');를 수행한다.
2) createOrderMessage()메소드를 수행한다.
3) fetchUserOrder() 메소드를 수행하게 되는데, Future 객체를 사용한 메소드 이므로 비동기 방식으로 진행하게 된다. Future.delayed와 Duration을 사용하여 2초 뒤에 'Large Latte'를 출력하도록 한다.
4) 하지만 결과는 불완전한 Future객체가 반환되었다.

- 우리가 기대한 결과가 안 나온 이유가 뭘까??
  => Future.delayed를 사용하여 2초 뒤 'Large Latte'를 출력 시키도록 하였으나, fetchUserOrder 메소드의 코드에서는 'Large Latte' 출력 코드 이후에는 실행할 코드가 없으므로, 2초가 지나기도 전에 fetchUserOrder() 메소드가 return을 하여 불완전한 Future 객체가 반환되어 createOrderMessage() 메소드의 order변수에 저장되고 출력 된 것이다.

- 그럼 어떻게 해야 정상적으로 처리될까?????????

- createOrderMessage 함수와 main 함수에 async 키워드와 await를 사용하여 'Large Latte'가 return 될 때까지 대기한 후, return되면 그제서야 return 값을 출력하고 종료하게 된다.
- Main 함수에 async와 await를 쓴 이유는 어찌보면 당연하다. Await를 사용하지 않았다면 createOrderMessage 함수를 호출한 print함수는 createOrderMessage가 return 하는 'Large Latte' 데이터를 받기 전에 불완전한 Future객체를 출력하고 종료될 것이다.

Future / async & await 예제

FutureBuilder 개념

- Future를 사용하여 비동기 방식으로 데이터를 서버든 어디서든 모두 받아 오기전에, 앞으로 받아올 데이터 없이도 화면을 구성할 수 있는 위젯들을 먼저 화면을 그리기 위해 사용되는 기능.
  만약, 이 FutureBuilder가 없다면 데이터를 다 받기까지 기다린 후, 화면에 UI를 그리거나 데이터의 변경을 setState()를 통해 바꾸어 줘야 한다.

- 인자 값 중, snapshot은 특정 시점의 future 데이터를 복사해서 보관하는 용도의 인자이다.

- Snapshot이 복사해서 가지고 있는 return된 future 데이터가 실제로 존재하는지 확인하는 절차가 필요하다.
  FutureBuilder 내의 builder 속성은 AsyncSnapshot 객체와 함께 제공되는데, connectionState값은 AsyncSnapshot.connectionState 속성의 value 값인 ConnectionState.none / ConnectionState.waiting / ConnectionState.done중에 하나의 값을 가진다.