Stateful Life Cycle of Stateful Widget

Advanced Flutter R2
NAME: Tasneem Samy

Stateful Widgets:

Stateful widgets are the dynamic workhorses of Flutter UIs. They maintain internal state, which can change over time and trigger UI updates. Understanding their lifecycle is crucial for building efficient and responsive apps.

1-Birth: Creation():

It consists of 2 stages Construction and createState()

1. Constructor:

- **Purpose:**
  - Establishes initial properties and configuration for the StatefulWidget.
  - Defines the parameters that can be passed to the widget when it's created
- **Example:**

```
class MyStatefulWidget extends StatefulWidget {
  final String title; // Configuration property
  final int initialValue; // Another configuration property

  MyStatefulWidget({this.title, this.initialValue = 0}); // Constructor

  @override
  _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}
```

- **Key Points:**
  - Similar to constructors in regular Dart classes.
  - Can be used to set default values for properties.
  - Defines the interface for how the widget is created and configured.

2. createState():

- **Purpose:**
  - Called by Flutter to instantiate the associated State object.
  - The State object holds the widget's mutable state, which can change over time.
- **Example:**

```
@override
_MyStatefulWidgetState createState() => _MyStatefulWidgetState();
```

- **Key Points:**
    - Indirectly invoked by Flutter when a StatefulWidget is inserted into the widget tree.
    - Always returns a subclass of State that manages the widget's state.
    - Marks the beginning of the State object's lifecycle.
- **Relationship:**
    - The constructor defines how a StatefulWidget is configured and what properties it holds.
    - The createState method then creates the State object that will manage those properties and any dynamic changes to the widget's appearance or behavior.

2. Mounting:

This phase involves creating a widget instance, associating it with an element, and adding the element to the widget tree.

1-initState()

Executed when the widget is first created and inserted into the tree.

- **Key tasks:**
    - **Initialization**: Set initial values for state variables.
    - **Setting up listeners**.
    - **Access widget properties**: Retrieve values passed from parent widgets.
    - **Schedule tasks:** Set up timers, subscriptions, or other asynchronous operations.
- **Example :**

```
@override
void initState() {
  super.initState();
  _count = widget.initialValue; // Accessing widget properties
  _timer = Timer.periodic(Duration(seconds: 1), (timer) => _increment());
}
```

2-didChangeDependencies()

Called after initState() and whenever the widget's dependencies change (e.g., parent widget rebuilds).

- **Key tasks**:
    - **Access inherited data or context**: Retrieve data from inherited widgets or the BuildContext.
    - **Perform actions based on dependencies**: React to changes in external data or context.

- **Example**

```
@override
void didChangeDependencies() {
  super.didChangeDependencies();
  // Accessing inherited data or context
}
```

3- Build State

The "build state" refers to the active phase when a stateful widget is rendering its UI on the screen. It's primarily associated with the build() method, which plays a crucial role in defining and updating the widget's visual representation.

1-build()

- **Called whenever:**
  - The widget is first inserted into the tree (during mounting).
  - The widget's state or its dependencies change.
  - The parent widget rebuilds and requests its children to update.
- **Responsibilities:**
  - Assembling the widget tree based on current state and context.
  - Returning a Widget object that represents the UI structure.

- **Example**

```
@override
Widget build(BuildContext context) {
  return Column(
    children: [
      Text('Count: $_count'),
      TextButton(
        onPressed: _increment,
        child: Text('Increment'),
      ),
    ],
  );
}
```

4- Updates:

- **setState:**

Called to signal that state has changed, triggering a rebuild.

  - Wrap any changes to state variables within setState to ensure UI updates.

. Handling State Updates:

- **Within build():** Access and use updated state values to construct the UI.
- **In Event Handlers:** Call setState() within event handlers (e.g., button taps) to update state based on user interactions.
- **In Asynchronous Operations**: Call setState() within Future callbacks or streams to reflect changes from data fetches or other asynchronous processes.

Example

```
void _increment() {
  setState(() => _count++); // Triggering a rebuild
}
```

5- Interaction:

Refers to the state that a stateful widget enters when it's actively interacting with the user or responding to external events. It's a temporary state that enables widgets to dynamically adapt their behavior and UI based on user actions or changing conditions.

didUpdateWidget: Called when the StatefulWidget's configuration changes.

  - Use this to update state or configuration based on new widget properties.

Example:

```
@override
void didUpdateWidget(CounterWidget oldWidget) {
  super.didUpdateWidget(oldWidget);
  // Handling changes in widget configuration
}
```

6. Deactivation:

- When it occurs: A StatefulWidget is deactivated when it's removed from the widget tree, temporarily disappearing from the screen.
- **Key stages:**
  1. **deactivate() method**: Called when deactivation begins.
     - **Used for:**
       - Pausing animations or timers to conserve resources.
       - Saving state that needs to persist across deactivation/activation cycles (e.g., text input fields).
  2. Widget stays in memory: The widget and its state are still accessible, but its UI is no longer visible.

Example:

```
@override
void deactivate() {
  super.deactivate();
  // Handling temporary removal from the widget tree
}
```

7. Disposal:

- When it occurs: A StatefulWidget is disposed when it's permanently removed from the tree and its resources are released.
- **Key stages:**
  1. **dispose() method:** Called before the widget is destroyed.
     - **Used for:**
       - Cleaning up resources (closing connections, canceling timers, unsubscribing from streams).
       - Preventing memory leaks and ensuring proper resource management.
  2. Widget and state removed: The widget, its state, and associated elements are removed from memory.

Example:

```
@override
void dispose() {
  super.dispose();
  _timer?.cancel(); // Ensure timer is canceled when disposed
}
```

Conclusion

Stateful widgets are the dynamic workhorses of the Flutter world. They hold the reins on state, that ever-changing entity that breathes life into user interfaces. Understanding them is crucial for crafting responsive and engaging Flutter apps.