Jelena Jovanoski Follow
Nov 24, 2020 · 4 min read ★ · Listen

Save

DISCOVER FLUTTER — WEEK #13

The Stateful Widget Lifecycle

One of the most common questions in Flutter job interviews

Photo by Ken Cheung on Unsplash.

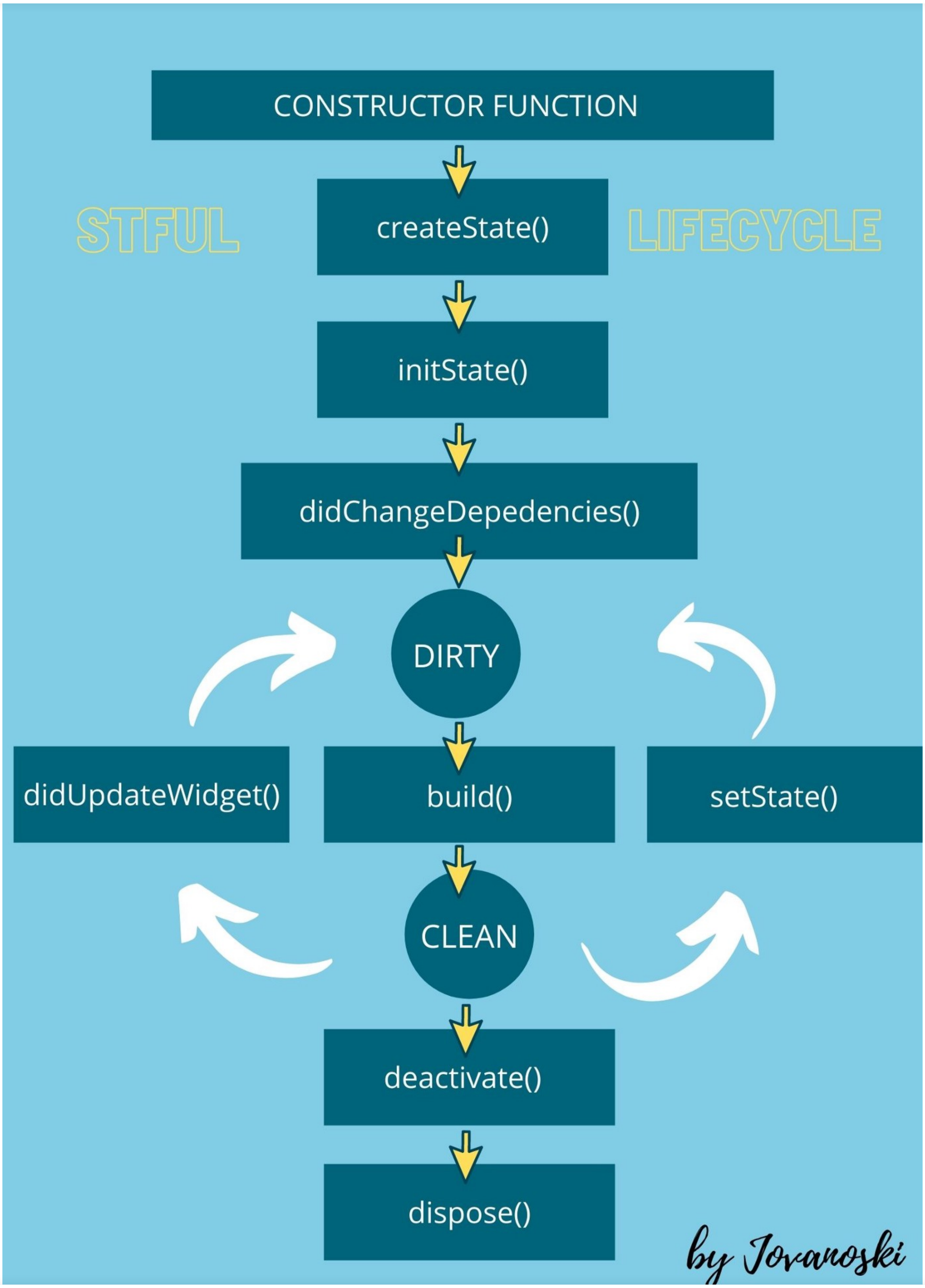
One of the questions that often appear in job interviews revolves around the lifecycle of a stateful widget, hence the article dedicated to this topic.

Seven Cycles of StatefulWidget

When Flutter builds a stateful widget, it first executes the `constructor` function of the widget and then calls the `createState()` method. If we look at the stateful widget, the constructor function is executed first. On the other hand, if we look at the `State` object of the stateful widget, its lifecycle starts when the `createState()` method is called.

Note: The `constructor` function is not part of the lifecycle because the state of the `widget` property is empty at that time.

Open in app Get started



This method creates a `State` object. This object is where all the mutable state for that widget is held. This method is `required` within the `StatefulWidget` :

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

mounted (true)

Once we create a `State` object, the framework will associate the `State` object with `BuildContext` by setting the boolean property called `mounted` to `true` . This property gives us information on whether this `State` object is currently in a widget tree.

Note: This step is not marked as a real step in the lifecycle, but it is important to know what is going on in the background.

2. initState()

When the object is inserted into the tree (`mounted` property is set to `true`), this method is automatically executed after the class constructor. `initState()` is called only once, when the state object is created for the first time.

Note: You cannot use `BuildContext` from this method.

Tip: Use this method to manage HTTP requests and subscribe to streams or any other object that could change the data on this widget.

```
@override
void initState() {
  super.initState();
  // TODO: implement initState
}
```

3. didChangeDependencies()

The framework will call this method immediately after the `initState()` . It will also be

4. build()

This method is `required` and it will be called many times during the lifecycle, but the first time is after the `didChangeDependencies()` method. So whenever the widget that belongs to the state is updated, the framework will always execute this method (i.e. every time `didUpdateWidget()` or `setState()` method is called).

5. didUpdateWidget()

Gets called if the parent widget changes its configuration and has to rebuild the widget. The framework gives you the old widget as an argument that you can use to compare it with the new widget. Flutter will call the `build()` method after it.

Tip: Use this method if you need to compare the new widget to the old one.

```
@override
void didUpdateWidget(covariant MyHomePage oldWidget) {
  super.didUpdateWidget(oldWidget);
  // TODO: implement didUpdateWidget
}
```

setState()

This method is often called from the Flutter framework itself and from the developer. The `setState()` method notifies the framework that the internal state of the current object is "dirty," which means that it has been changed in a way that might impact the UI. After this notification, the framework will call the `build()` method to update and rebuild a widget.

Tip: Whenever you change the internal state of a `State` object, make that change in the `setState()` method.

```
setState(() {
  // implement setState
});
```

This method is called when the widget is removed from the widget tree, but it can be reinserted before the current frame changes are finished when the state is moved from one point in a tree to another.

```
@override
void deactivate() {
  super.deactivate();
  // TODO: implement deactivate
}
```

7. dispose()

This is called when the `State` object is removed permanently from the widget tree.

Tip: Use this method for cleaning up 308 1 *connections.*

```
@override
void dispose() {
  super.dispose();
  // TODO: implement dispose
}
```

mounted (false)

After the `dispose()` method, the `State` object is not currently in a tree, so the `mounted` property is now `false`. The state object can never remount.

Conclusion

If you're a fan of short, interesting articles covering various Flutter topics and you want to get into the habit of learning Flutter with me over the next 18 weeks, you can read my articles every Tuesday.

If you have any questions or comments about this article, let me know in the comments section.

- Week #1 — "Roadmap for Learning Flutter"
- Week #2 — "How to Create Intro Sliders"
- Week #3 — "How to Easily Generate Routes"
- Week #4 — "Recommendation for Certain Flutter Packages"
- Week #5 — " Bottom Navigation Bar"
- Week #6 — "Holy Trinity of Every Animation"
- Week #7 — "New Material Buttons in Flutter Version 1.22"
- Week #8 — " My October Recommendations for Flutter Packages"
- Week #9 — "Flutter Version Manager — FVM"

See you next week. Don't break the streak!