So if we think about in this list we've already come across set state,

we know how to use that to control and manage local state really well

and that of course has to be used in conjunction with a stateful widget. But because stateful widgets

are a lot more complex than stateless widgets, for one stateful widgets actually needs have a memory.

They need to remember what their state is so that it can be changed whereas stateless widgets are much

more simple to work with and they could just get created, destroyed and rebuilt.

Now we also saw lifting state up earlier on using a combination of callbacks and lifting the state up

in the widget tree,

we were able to manage our state. But of course as our app gets more complex and if we need to have callbacks

that traverse the entire tree or if we need our state to be passed multiple levels down then we might

encounter some problems.

So currently the most flexible way of managing state is through the use of the provider package.

And this is currently the recommendation from the Google Flutter team for us to use.

But I guess what I'm trying to say is that keep in mind that even though this is the officially recommended

way of managing state in your Flutter apps, it's not the only way.

And in a lot of cases you might find you need a different way or you might find that you need a combination

of these different ways of managing state.

And that's not wrong.

It really depends on your use case,

so your requirements, your particular sense of style for how you want to organize your code and also

the maturity of the framework of Flutter.

But let's talk about Provider because it's a really really great tool.

It's a package that was actually created by the community.

Google had a similar package called Provide which did something kind of similar but along came this

guy Remi and he decided to build a version that was actually deemed to be better. So full credit to the

Google Flutter team

they realized the potential of this package and actually put their full weight behind it.

And now not only is it officially recommended, but it's also supported both by Remi himself as well as

the Google Flutter team.

So it has all the resources it needs to achieve its full potential.

Now in order to fully appreciate why Provider is so useful we first have to consider what is the problem

that its actually trying to solve.

So firstly from our code, you might have already realized that there's a couple of things that's getting

a little bit hairy right?

We have all of these callbacks in order to pass data back up the tree and we have a whole bunch of state

ful widgets which needs to use sets date to call the build method of the child widgets and lots of

things are getting rebuilt constantly. And it's probably not the most efficient way of going about this.

But because our app is getting quite large and it's quite hard to see at a glance what's going on when

we are managing our state, I've created a really simple app to be able to illustrate the problem a little

bit better.

So in this case I have a top level class which is MyApp which then has two children and one of which

is something called Level 1. And then Level 1 creates another one called level 2 and Level 2 Creates

Level 3.

So here's where that code looks like. It's pretty simple

and the important part is that we've got a MyApp class that creates a level 1 class that then creates

a level 2 and Level 3 class in that order.

So here is our material app which creates level 1 and then Level 1 creates Level 2, Level 2 Creates Level

3.

We basically have exactly this structure.

Now here's the problem.

Let's say that I had a property that needed to be used over here as well as over here. In order to do

that we have to lift the state up for it to live within my app.

Now that's all very well and good but when I pass that property or the state of that property down to

Level 3, in order to get it there I have to traverse through Level 1 and level 2 and Level 3 which means

that that data has to be updated within these two classes which doesn't actually need it.

So I have to essentially drill down through my tree in order to be able to reach the lower levels. And

in reactive programming

this is something that we might call Prop Drilling,

so drilling through the tree in order to pass down the property. Let's see this in action in our code.

Let's say that we have our string which is just called data and it's, I don't know, a 'Top Secret Data'.

And this needs to be used within both the AppBar here

so it's gonna be used inside the title as a text widget and that is gonna contain that data. But it also

needs to be used down here in Level 3,

so this level 3 text widget also needs data. But in order to get it down into level 3 I need to drill

through all my levels.

So it means that inside Level 1 I'll need a property which I'll just call data and when I initialize

Level 1, I'm going to initialize it by passing over this data property. And then I'm going to pass it

over to level 1 through its constructor and then now that data is inside Level 1.

But in order to get it down to Level 3, I have to keep passing it to Level 2

right?

So let's pass it over to level 2 and let's create that final String property which we call data

and then when Level 2 gets initialized, we also pass over the property to this.data.

And then finally we just need to pass it over to Level 3.

So again I'm going to create my final string data and when I initialize Level 3, then I pass over the

value of the data that I need.

And now finally I have the data where I need it down in Level 3 so that I can actually put it inside

my text widget.

So now if I hit save and I run my app, then you can see that it works right?

My data which lives inside MyApp, my top level widget, is able to pass this data property both down to

my AppBar as well as to my level 3 widget which is this one that has the text widget. But in order to

do that I had to pass that data through a whole bunch of unrelated and really widgets that didn't care

for this data at all, because you can see that in level 2 I don't need that data in Level 1 I don't need

that data. I'm just passing this data through in order to get it down to Level 3.

So for any lazy programmers out there, this is pretty painful to watch and it also massively increases

the complexity.

Say, I don't know, have a typo in here or say I had some sort of other piece of data inside level 1 and

I get confused between the data that gets passed in and the local version of it,

it's really not pretty.

So what can we do instead?

Well if we have the Provider package well then we could move the state up to the top but then we can

have widgets that simply subscribe to the state. So we might have our level 3 widget put its hand up

saying I'm going to listen for any changes in the state. And when it does change I'll be notified and

instead of traversing through level 1 and level 2 in order to get to level 3,

it's simply only passes the data to the level 3 widget which actually cares for this updated state instead

of rebuilding level 1, level 2 and level 3.

Instead of having that tree rebuild to pass the state down so that we actually get it to level 3,

we simply just let level 3 listen to updates.

So how do we use provider to get all of these cool features?

Well we first have to include the provider package in our app.

So let's head over to installing and let's add provider as a dependency.

So I'm gonna open up my pubspec.yaml and just below my Cupertino icons where all my dependencies

go, I'm going to add the provider package and I'm going to call packages get to fetch the code from that

package.

And now I should be able to import the provider.dart file.

So now that I've got provider in my code, I'm able to use it.

The first thing I have to do is to expose my data, so to say what is the data that we're going to be

providing? In order to do that

it's as you can imagine and we're in Flutterland so it's gonna be achieved using a widget. And I'm going

to put that provider as high up as it needs be given that I need the data both in my AppBar as well

as all the way down in level 3.

Well then I'm going to put it at the very top so that my data is provided to my entire material app.

So here I'm going to wrap with a new widget and that widget is going to be a provider widget.

Now with the provider widget, we also have to specify a data type of the kind of data that this provider

is going to provide. That data type is just going to be a string

because our data is just a really simple string right?

So now that I've created my provider widgets and it has a child which all subsequent levels are going

to be able to listen to the data that I provide,

we also have to add one more property which is the provider builder.

The builder takes the current build context so it knows where it is within the widget tree, and then

it has to return the data that needs to be available to all the children. That is going to be our data

right here.

Now that I've got my data being provided at the very top level of the tree, then I can ask the provider

to give me that data wherever I need it.

So firstly let's simplify this code.

We have a simple one line return so we don't need all of this curly brackets etc. and the return keyword.

We could simply just use our fat arrow, does exactly the same thing.

So now we are providing our data at the very top level of our app above our material app so we can now

reach into it and use it wherever we need to.

The first place where we need it is within our text.

So now instead of having to pass our data through our level 1 level 2 widgets which really don't

need this data, we can't actually go ahead and delete all of this code

all of it here and here and here.

And instead we're just going to be using it within level 3.

And in order to access that data, all we have to do is say provider.of

and again, the 'of' method expects us to specify what kind of data we're trying to get hold of.

So this is going to be the string that we're providing right here to all of the children in our tree.

So now if I hit save and I check my app, you can see that I'm still getting my Top Secret Data being

passed all the way down to my level 3 stateless widget.

And that comes from right here. As you can see let's say I decide to change this data to, I don't know, let's

say a top secret code.

And now if I rerun my app, you can see that data is actually being passed over to level 3 without

needing to drill through level 1 and level 2. Now

that's all very well and good.

We were able to provide our data at the very top level and then we were able to read the data all the

way down here without needing to go through any of the in-between parts in our tree.

So now when the state of that property changes up here, because level 3 is listening to those changes

well it can receive it immediately and rebuild itself without needing to rebuild level 1 or level

2,

so faster more efficient code.

Now let's make this a little bit more complex because at the moment all that we're doing with provider

is we're providing and we're reading. What if we also had to update the state from somewhere down within

our tree?

So let's update our simple app to still have level 1, 2 and 3

but now we also have a text widget over here called MyText in a separate class

and we also have a MyTextField widget down here

that's a child of Level 2.

So the idea is how can we get an update that happens down here all the way over to MyText which is

higher up in the tree and also in level 3 using the power provider?

So here I have my separate class MyText and the other class which is MyTextField and I'm going to

slot them in to my tree.

So my text is going to go as the text widget here in the AppBar.

So let's change that to MyText and then MyTextField is going to be a child of my level 2 widget.

So instead of this empty container here, I'm going to add MyTextField right here.

So now we basically have this exact structure of widget tree and I have to be able to get this data

all the way over here as well as over here without needing to rebuild any of the intervening widgets.

So how do we do this?

Well we need the use of a class called a ChangeNotifier.

So let's create another class here which is going to be our data class and make this data class extend

the ChangeNotifier class.

Now this ChangeNotifier class actually comes from Flutter itself.

This is not something that came with provider.

It already exists and it exists for the purpose of notifying listeners to this particular class of any

updates in the class.

So let's move our data down into our data class and instead of having it being final we need to be able

to change it

right?

We want to we have to make updates to our data.

So let's start it off with just some dummy data, let's call it 'Some data'. And let's start providing this

data object instead of just the simple string that we had from before.

So instead of returning data which now no longer exists, we're going to return a object of the data class.

And this object is going to no longer be using a normal provider which just provides a string data type,

well instead it's going to be a ChangeNotifier provider and it's going to be providing a Data data

type.

So now in order to get access to our data right here, we will still use our provider.of

and instead of using the string data type we actually have to provide the data type that we want to

provide which is data. And instead of providing this entire object, this data object that gets created

from the class, we're going to be providing the data object's data property because that is the thing

that is a string.

And that string is then going to be put into our text widget and be displayed.

So now I can also copy that and put it into the MyText widget as well so that it's rendered in my App

Bar as well as my level 3 text widget.

So let's first run our app and check to make sure that we're actually able to both provide our data

as well as read our data in both the MyText widget in the app bar as well as the text widget in level

3.

So that's working pretty well.

So now we have to achieve our next goal which is to update the data and we're going to update it within

MyTextField.

So we all know that MyTextField has a property called onChanged which takes a callback that has access

to the new value or the new text that gets typed into the text field.

And we now need to be able to move this text into our data class to update this string.

So let's create a method inside our data class. Let's create a void method so it doesn't return anything.

And it's going to change our string.

So it's the changeString method. And we'll pass in a new string which is going to take the place of

our data and I'm going to set the data to equal the new string. And then most crucially, I'm going to

use a method that's available to us because we are inheriting from the ChangeNotifier.

And that method is the notifyListeners method.

So what this does is as soon as we update our data to the new value, we call notify listeners so that

everybody who's listening to our provider can rebuild their widgets as well because they're notified

of that state change.

So now if inside our onChanged we tap into our provider.of and we specify the type of data we're interested

in which is our data class,

Well then we can tap into that method called changeString and we're gonna pass in the newText as the

newString

that's gonna be put in to this data property.

So now if we rerun our app and I go into here and I start typing, then you can see that all the listeners

to this data, namely the myText widget and the text widget in level 3, well they're going to be rebuilding

because the state is now updating every time I change the text in my text field.

So let's just review the code once more.

So I created this class called Data which extends something called a ChangeNotifier that allows pieces

of the data inside this class to notify any listeners which are trying to track the state of a piece

of data namely this property called data. And that property is being read both inside this text widget

in the AppBar as well as this text widget inside level 3.

So the appearance of these two text widgets are tied to the state of this data property.

And when that data property updates, then this text widget is going to rebuild itself to reflect that

change.

Now that data property comes from right here and we update it inside our change

String method where we pass in a new string to replace the old string inside the data property. And then

and crucially we call notifyListeners so that everybody who's listening to that data property gets

notified and they can rebuild themselves to update the latest state.

Now because we need to call this notifyListeners

this is also why we can't just simply tap into provider of data which we know is basically this data

object and simply tap into that data property and set it equal to the new text.

Because in this case when we're just updating this string from the outside, we're not actually calling

notifyListeners.

We're just updating this property's value. So we must use a method to change the string so that we actually

call notifyListeners.

Now another important thing to remember is that this provider.of method actually has a optional

parameter called listen.

So let's say that MyText which is the text widget inside the AppBar

well let's say that we don't actually want it to update with every single update of MyTextField.

Let's say that we want it to get the original value of the data that's stored but we don't want it to

listen for updates.

We don't want it to rebuild

every time this data changes.

Well then all we have to do is to add the listen property and we can set it to false.

Now by default this property is set to true which is why every time you use provider.of you're getting

a listener which is going to update itself based on changes in that thing that you're listening for.

But now that we've said listen to false let's rerun our app and you can see that both text properties

start out with some data which comes from here

when data gets initialized that's going to be the value of data.

But when I start typing in the text field, because my text widget

this one here is not listening to the changes,

it doesn't update and it doesn't redraw.

So this gives you quite a fine and granular control over what are the things where you want to listen

for updates to the data and redraw and what are the things where you just want an initial value of the

data

and to never change again even if it changes. Now that you've seen the provider package in action, you

might be wondering to yourself, 'Well how exactly does it work?'

So in order to understand that let's pop up in the hood and see what's going on behind the scenes. Out

of all of these different ways of managing state

there is one where we use an inherited widget. Now an inherited widget is basically a widget that allows

us to pass data down the tree without having to rebuild the intervening part of the tree.

So that sounds kind of familiar right?

That's kind of how provider works.

Essentially the provider package it's just a wrapper around the inherited widget. But because the inherit

widget class has some specific requirements in order to construct it and you have to construct it in

a certain way that ensures that it's still being efficient and it's running as expected.

So there's quite a bit of boilerplate code and you have to be quite mindful of how it works when you're

using it.

So I recommend taking a look at the inherited widget class and having a listen through the explanation

of this widget by Philip with his very cool Google bicycle shirt and learning a little bit more about

how it actually works behind the scenes. Because essentially our provider package is just leveraging

the power of this class but it's doing it in a way that makes it easier to work with it and to force

you to use it in a safe and effective way.

And once you're done with that, I recommend you to build your very own version of this widget tree. So

have separate classes

well in this case I've got about six different classes which are organized in this way and to use the

provider package yourself to see how you can provide data at the very top level, how you can read data

at a lower level and to update data using the ChangeNotifier provider.

So have a play around with that and if you want to see the code that we've been using in this lesson

then you'll be able to download it in the course resources too.

But in the next lesson, we're actually going to plug in provider into our actual Todoey app and see

it in a much more complex and more real life scenario.

So for all that and more, I'll see on the next lesson.

