Now that we've learned all about the provider package, it's time to actually put it into action in our app so that we can read the tasks that are stored,we can add new tasks,we can check tasks off and we can delete tasks when the user desires it.

So in this first part of putting provider into practice, we're going to tackle the first goal which is to be able to get the tasks and to be able to use it both on the task screen as well as the task list.

Essentially we're going to keep the state of our task provider high so that we can use it all the way down the tree and we're going to be providing that data both to the TaskScreen as well as the Tasks List and they're going to listen for changes in the tasks data. On the task screen the reason why we need it is because we have this little part where it says 12 tasks.

That number has to be dynamic and reflect all the tasks that are currently inside our To Do list. And for the task list well that's pretty self-explanatory.

We need the task data available there so that we can actually render this list of tasks.

Now in the coming lessons where we're implementing provider, I'm going to explain exactly what behavior we need first upfront and then you have the choice of either completing it as a challenge or simply following along with me.

But I recommend that you really give it a go yourself and even if you get stuck and come back, you will learn a lot more than simply just following along.

So in this case, what we need to be to do is to use our change notifier provider to provide our tasks data right at the top of our app at the very root of our widget tree.

And then we can listen for it inside the task screen to update that number as well as inside our tasks list to update the list of task tiles.

So once you already, open up your current Todoey app project and try and achieve this goal.

So I hope you've given that a go and you're here just to check the solution.

Now remember there's a lot of ways of doing the same thing. You can achieve exactly the same behavior using all sorts of different ways and my solution is just one way of doing that.

So the first thing we need to do is to of course go into our pubspec.yaml and to add our provider

as a dependency.

I'm gonna do that right here and then I'm gonna get all my provider related files and now I can use it.

So the first thing that I'm gonna do is I'm going to create a new Dart file inside my models folder

and I'm going to call this task\_data.

So this is going to be the object that's going to be provided throughout my tree so that they can tap into the data of our tasks.

This class is simply going to be called TaskData and it's again going to extend our change notifier class.

Now the reason why it's erroring out at the moment is because we need to be able to import the foundation.dart package or we could import the material.dart package.

But the material.dart package actually contain the foundation.dart package so if you wanted to keep it light then you can simply just use foundation.dart.

And now we will be able to find the change notifier inside foundation.dart and we can use it to extend our task data class so that we can turn this class into something that other objects can listen to for changes and update when we tell it to.

So the first thing I'm gonna do is I'm going to import my other models class which is my task.dart and that just a quick reminder is basically the structure of a task object. And now if I go into my tasks screen.dart, I can find where currently my task data exists. It's inside this list of task inside my TasksScreenState.

So I'm going to lift that right out, so I'm going to cut it out of here, and I'm going to move it over to my task data object.

So it's going to be pasted inside here.

It's going to be created exactly the same way. Tasks have a name and by default they're set to 'not done'. So now I'm ready to provide this task property across my app. And I'm going to do that at the very top of my widget tree.

So I'm going to do it inside the MyApp class and I'm going to wrap everything that's inside my material app with this change notifier provider class. And of course to be able to use it I need to import my provider.dart package.

So remember that provider has to be at a place in the tree that's high enough to be seen by the rest of the widgets that branch off later on.

So in our case, that means putting it right at the top of MyApp class.

Now if you only needed the data say inside the task tile and nowhere else, then you might be able to go up one level to its parent and then wrap its parent which is this task list which of course lives right here, then you could wrap that with a provider widget to be able to provide to all of its children.

But if you get this wrong and you put this too low in the tree then the children might not have access to it then you'll get errors when you run the app and try to do things telling you that it's not placed at the correct position in the widget tree.

So take care where this goes.

In our case it needs to be above our material app.

Now I'm getting a little warning here because there's a parameter called builder for the change notifier provider that's needed.

So we saw this before. The builder takes the current context,so the current position inside the widget tree, and it returns the object that needs be provided to all of the children in the tree. And that object is of course on newly created task data object.

So let's go ahead and provide the task data object and also tell it about this task data object which comes from the models folder and it's called task\_data.dart.

So now we have our change notifier provider at the very top of our widget tree providing the task data object to all children below the tree that wants to listen for the data in there. Now we can actually read the data from here namely the tasks list.

So the first place where we might want to do that is actually on the task screen. Here where we have all tasks.length which shows the user how many tasks they have remaining in their list,well this needs to be based on our task list.

We need to count how many there are in here and provide it as a interpolated string right here.

And we're getting an error right now because we removed tasks from our task screen state.

Let's go ahead and remove tasks.length and let's use what we learned previously where we tap into to provider and we call the .of method.

Now I'm going to get Android Studio to help me out with typing out the code.

But first that means I have to import the provider.dart.

And now I can tap into the .of method of provider and of course always specify the date type in the of method and it's going to be my task data object which again we need to import.

So we're going to tap into the models and we're going to tap into the task\_data.dart.

Eventually we're going to remove the need for the task.dart and to manage all of that related stuff inside task data.

But for now let's go ahead and tap in to this object which is going to be equivalent to our task data object now and let's go into it to get hold of the tasks property. So let's write a .tasks and then we're going to get the count or the length of that object which currently should be equal to 3.

So let's go ahead and hit save and try to get rid of the part where we have a Dart analysis error.

So the first one is this part where we're adding tasks to our previous task list which now doesn't exist so of course it's undefined.

So I'm actually going to go ahead and just comment out our set state and in the next lesson when we tackle how to add new tasks to our tasks data provider, then we'll figure this part out.

So the other part is of course down here where we have our tasks being passed to our task list.

Now this is of course the second place where we need to read from our task data.task property but instead of passing it over and drilling through our tree, instead I'm simply going to go into our task list and get a hold of a reference to that task data object over here instead.

So we actually don't need to pass anything over anymore so we don't need these two things and we don't need this reference to our task.dart either.

But instead I'm going to be importing our provider.dart and also our model's folder the task\_data.dart as well. And we're going to be using it inside our task list to build our list view.

Now right here instead of using the widget.tasks which taps into those properties passed into the

task list stateful widget, instead I'm simply just going to tap into the provider.of method. So instead of the widget.tasks.[index].name, I'm going to replace widget.tasks with provider.of and specifying the data type which is our task data. And then it's going to be provider.of.tasks[index].name.

So I can copy this over and replace our widget in all of these places.

And for now I'm simply actually going to comment out our set state to handle what should happen when our checkbox gets checked.

We're going to deal with that in a later lesson.

But right now all we want is just to be able to read the data that's inside the tasks inside our tasks data which is being provided using our provider package.

So now notice how we're not using that widget anymore which comes from the stateful widget.

So that's our way of accessing the properties that we deleted.

And so we actually don't need this to be stateful even.

We can actually turn it into a simpler stateless widget and we can still get hold up with the provider and get the data that we need because we're no longer passing it over from our task screen via the constructor.

So now notice how we also don't need states inside our task screen so we can convert this to a stateless widget as well by deleting all of these lines and changing this to a stateless widget and we can delete the unused imort right up here as well.

Now our app is actually a lot simpler because we have these memoryless stateless widgets which simply get constructed when they're needed and destroyed and replaced with new ones when they update.

So now we're tapping into the provider of our task data and getting hold of that list right here in order to populate our task titles and also we're using it inside our task screen to show how many tasks we have remaining.

And this is just simply reading the tasks.

Let's go ahead and hit save and run our app from scratch. And you can see that hopefully everything going well and all your code in the right places we should have exactly the same user interface as we had before. But this time this data is no longer being passed around,drilled down through the tree and instead it's coming from our provider.

So did you manage to get to this point when you tried to tackle it yourself?

Well there's a couple of things that we did including creating our task data class that extends change notifier and wrapping our material app within a change notifier provider which provides that task data object and then finally using it by using the provider.of task data to tap into it both inside our task screen as well as our task list. That works right now but we can actually make it a lot simpler. And we can do this through the use of something called a consumer widget that comes from our provider.dart as well. Because notice how we're using our provider.of task data.tasks in three places here.

It's getting a little bit repetitive right?

So instead of calling provider.of in three places, we can simply just wrap all of the downstream widgets that need to be updated when this tasks property changes inside what's called a consumer widget.

So notice how we're using our provider.of inside our tasks tile but we're also using it inside the item count.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 stateful 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.