Now in the last lesson we looked at a really simple out-of-the-box animation that comes with Flutter which is the hero animation and we saw how easy it was to implement something that would have actually taken us a lot more code had we done it in a different platform. But in this lesson, we're going to take it a step further and we're going to be looking at how we implement super custom animations, things that we want to happen on screen, be it changing the background colour or changing the size of something or moving things around.

But we're going to do it to the beat of our clock to say how long we want it to go on for whether if we want it to loop around and what we want to actually happen with those animations. But to be able to do this, we have to learn a little bit more about how custom animations work in Flutter. And again we're coming back to our smoothie analogy.

Well in this case most of the animations that we're gonna be building with Flutter require just three things and it's three important concepts that you need to understand and to be able to really do what it is that you want it to do.

And the three components are a ticker, an animation controller and an animation value.

Now a ticker is exactly what it sounds like, it's a bit like the ticking of a clock.

So we need something to be able to count and take our animation along so that at each tick of the clock, our animation changes in value and changes in shape or color whatever it may be. When you think about the simplest type of animation that you could create which is just by drawing different pictures onto a flip book and then as you flip through it, that's how the animation effect comes about.

Well that's exactly the same with our animations in Flutter.

So for every time the ticker ticks, it triggers a new set state so that we can render something different on screen, and it's just like flipping through the pages of a flip book. Now the other component is the animation controller and this is sort of the manager of the animation if you will.

It's the animation controller class which is going to tell the animation to start, to stop, to go forwards, to loop back, how long to animate for, all of these things are determined when we create an animation controller and set its properties.

And finally there's the animation value and this is the thing that actually does the animating.

So usually the animation value will go from 0 to 1.

And by using that value, we're able to change things such as the height or the size of a component or the color or the alpha or the opacity of a component.

So let's try it out in actual code and apply our theory to practice.

So I'm currently inside the welcome\_screen.dart, and the first thing I'm gonna do in order to create my animation is to build a animation controller.

We're first going to just build a variable which is of type AnimationController. And make sure that you've built an animation controller and not a AnimatedController.

And we're just gonna give it a name of controller.

So we haven't initialized anything yet.

We're just saying that this is a variable that's going to hold data of type AnimationController. And the time when we actually want to initialize it is when our state object gets initialized.

So we're going to override the init state method.

So we know that this method gets called at the moment in time when this welcome screen state gets created.

And it's right at that moment in time that we also want to create a animation controller and save it into this controller property.

So we're going to create it using the animation controller's constructor, so we're going to say animation controller and you can see that it takes a number of properties.

So one of those properties is the duration, so how long do you want this animation to go on for?

And this property takes a duration object, so we can specify how long in days, hours, minutes, seconds etc..

And so I'm gonna say that my animation is just going to last for 1 second.

Now after I've specified the duration, the next property that is a required property for any animation controller you can see at the moment, I'm still getting this warning here,this little yellow triangle because my animation controller has a required parameter which is 'vsync' and I haven't provided it yet.

So let's go ahead and do that. Now vsync, what exactly is this thing? Well this is where we provide the ticker provider.

So what is the thing that can act as a ticker for our animation controller?

Now usually, the ticker provider is going to be our state object,so in this case, it's our welcome screen state. And you can see that the welcome screen state inherits or extends its parent class which is a state widget. And to be able to turn this welcome screen state objects into something that can act as a ticker, we have to use the keyword 'with' and specify that this class welcome screen state can act as a single ticker provider.

So it's like we're upskilling our welcome screen state with a new ability. The ability to act as a ticker for a single animation.

So if we had multiple animations then we would use the TickerProviderStateMixin.

But in this case because we only want to animate one thing, so we're going to use the single ticket provider as the add-on ability. Now when we're doing this, essentially what we're using is a concept called Mix in. And mix ins enable your class with different types of capability.

And unlike inheriting from a class, you can add multiple mix ins to add many capabilities.

Now in the next lesson of the course, we're going to do a deep dive on how mix ins work in Dart and what exactly they are and how they actually enable our class to be able to do things such as act as a ticker.

But all we need to know for now is that by adding this single take a provider state Mix in, we enable our welcome screen state to act as the ticker provider.

So when we go into our vsync property, when it's looking for a ticker, this is where we provide the current welcome screen state object as the value for the vsync.

So this is what the ticker is going to be.

And when we want to reference the object made from the class in the class' own code, we use the keyword 'this'. So this line of code says that who's going to provide the ticker for my animation controller.

It's going to be this current welcome screen state object.

It's going to be the very object that gets created from this class which now has the capability to act as a ticker because we've added that mix in to the class declaration.

So now that we have a ticker, we have a controller.

Well all we need to do is to actually use the animation and get the animation to start. To do that on the next line inside our init state.

We're going to tap in to our controller and we're going to say, 'forward', so this is going to proceed our animation forwards. Now by default, animation controllers will animate a number.

So for every tick of the ticker, it will increase that number and the number usually goes from 0 to 1.

So we'll go from 0.01 to 0.02.

So in one second we might get 60 ticks on our ticker in which case our controller will animate from 0 to 1 in 60 steps.

So if we wanted to be able to see what the controller is doing, well then we have to add a listener to it.

So we say controller.addListener and the listener takes a callback.

So we're going to add a callback in here.

And now we can listen to the value of the controller which is the actual animation right?

So let's print the value of the controller.value.

And now let's go ahead and hit hot restart because remember, when our code is in the init state it only gets called the first time that this state gets created.

And that doesn't happen when we reload.

So let's click hot restart and let's take a look within that console to see the values that were getting printed.

So you can see that we're getting zero for a while and then we're getting 0.32,

0.36, 0.39 etc.

And it ticks along and increases that value along a steady interval. So this number we can now use for a number of things right?

So we could for example say, well what if we apply that number to our background color?

If instead of having a boring white color, let's instead have a red color and we specify the opacity, so a red with a degree of opacity.

Now the opacity takes a value between 0 and 1, 1 being fully opaque 0 being completely transparent.

So this is a perfect application for our controller value which we already know to go from 0 to 1 within the space of one second.

So let's provide that as the opacity.

So the controller.value which is also a double and this expected double so that's perfectly fine.

So now if we hit hot restart again and we check out our screen, you can see that it starts out being completely transparent.

So pretty much just showing what's behind it which is the material app which is black.

But it doesn't actually change anything right?

Why is it not changing, why is it not animating even though I can see that my values are in fact changing and animating?

Well that of course comes back to how Flutter works and if you remind yourself about how Flutter is able to react to values such as the control.value, then you remember we're going to need set state to tell our app that this opacity is going to be dirty.

And we have to rebuild our screen by calling the build method.

So that means inside add listener, we can call set state.

And we don't have to do anything inside set state because our values are already changing with the animation controller.

So now if we hit run you can see our animation happens right?

It goes from completely transparent red, which is basically showing what's behind, to completely fully opaque red.

And it animates that across one second.

Now the only property for our animation controller that's required is the vsync, to specify what is going to act as the ticker.

But in this case we've also specified a duration,so how long the animation is going to run for.

But there's also other properties that we can change such as the lower bound and the upper bound. If instead of animating our value from 0 to 1, we can change those values by changing the properties.

So let's change the upper bound from 1 to 100.

And now instead of changing the capacity of the background color, so I'm going to change that back to white, well instead let's go ahead and change our text. Instead of saying 'Flash chat', let's make it actually print out that value from 0 to 1.

So let's add in our string interpolation symbol, our dollar sign, and then we're going to tap into that controller.value and because it's a double we should change it to an integer, a whole number, so that we don't have to look at 0.0etc. And then I'm going to add a percentage sign as a string, so it should be outside of the curly braces.

And now if I run my app then you can see that instead of having flash chat show up on screen, I get a loading indicator right?

It animates almost like a loading functionality, like it's going from 0 to 100% which looks pretty neat.

So that's another way that we could use the controllers value to animate something on the screen.

Now we can also use that value in order to maybe animate size. So let's restore this text back to flash chat and let's go into our logo container again. And instead of having a height of 60, let's make it go from 0 to 100.

So let's apply that controller.value here.

And because it's already expecting a double, then we can leave it just as that.

So now when I run my app and we see the animation show up, we should see our logo go from 0 height to 100 pixels in height over a second through that animation.

So you can see it's extremely flexible to use that controller's value to animate a whole bunch of things.

Now at the moment, our animation happens linearly. So as it grows in size it grows linearly in size.

And when we look at the value of the controller, you can see that the increments go from 0 through to 100 pretty evenly right?

And it's pretty much doing this about 60 times a second.

Now what if we wanted this to look a little bit different? Well then we could use a class called Curved Animation.

And this way we can change the animation value along a curve. And the types of curves that we can use include things such as bouncing for example. And Flutter has got this great documentation where it shows you what each type of curve looks like on a graph, so going from 0 to 1 and going along one second for example. And you can see what the animation as a result of applying each of these curves will look like depending on what you're trying to change.

So let's try and do something really simple. Let's use a decelerate curve and let's see how we would do that in our code.

So in order to use curves, we have to create another variable and this variable is going to be of type Animation. And we're going to call our animation just animation and then inside the init state, we're going to initialize that animation to a new curved animation. And a curved animation expects two required parameters.

One is the parent.

And notice the parent has to be an animation controller.

Now the parent is what will we apply this curve to, so in this case this is going to be our controller, so our existing animation comes from this controller.

The second property is what kind of curve we want to apply to our animation.

And previously we picked out one from our curves, which was the decelerate curve.

So that's the one that looks something like this.

So if it was scale, it would be fast at first then it would decelerate scaling up towards the end.

So now that we've created our curved animation and we specified what animation we want to apply this curve to and what type of curve we're going to apply to it, then the last thing that you have to check is make sure that when you're applying a curved animation to your controller that we can't actually have an upper bound that's greater than 1.

These curves have to draw from 0 to 1.

So if we leave this upper bound in then we'll actually get an exception thrown and our app or crash.

Well let me show you firstly how that looks.

So instead of using the controller value we're now going to use the animation value because the animation is almost like the layer that's applied on top of the control animation.

So let's print the animation.value and inside our height property, let's change that to animation.value as well so that we use this new value that's generated based on the curve that we selected.

So now if we hit run, our app will crash.

And the reason that we'll get is because the upper bound can't be greater than 1, it has to be less than equal to 1.

So always remember that if you're using curves to check and make sure that the upper bound is not larger than one. And if you just want one as the upper bound then you don't actually have to change anything.

But having an animation that goes from 0 to 1 for the size of our image is not very useful right?

We won't be able to see a change between 0 and 1.

So let's multiply that number by 100 to exaggerate the effect.

And I'm going to keep the animation value as it is to be printed so that we can see what it looks like.

So now here's how our animation looks with that decelerate curve applied. It first starts out increasing in very large amounts, so 15 goes to 23 goes 29 goes to 33, but then towards the end of the second, it starts increasing very slowly,0.98 0.99 0.99 etc..

And the effect is that our image will increase in size very quickly and then it'll decelerate to stop expanding towards the end.

Now feel free to check out what some of the other curves look like.

For example we can have 'ease in' as one of the curves and this one will actually be kind of like the opposite.

It'll be slow to start with but then it'll be a bit faster towards the end. Now what if I wanted my animation to go the other way,what if I wanted to go from large to small?

Well instead of getting our controller to go forwards, I can actually make it reverse. And I can say my animation should reverse and when we're reversing, it's a good idea to specify a from property.

So this is going to be the new starting point.

So let's say that we start off at 1 times 100 and then we go all the way down to 0.

So now if we hit run and we take a look at our animation, it starts out big but then goes back down to small again, so from 100 to 1. What if we wanted our animation to loop, what we wanted it to go large then go small and large then go small?

Well in order to do that we need to know when the animation has actually completed, when has a reverse animation completed wnd when has a forward animation completed.

So let's go ahead and change this back to forward.

And we're going to use a method that we can attach to our animation which is called addStatusListener.

And here we can pass a callback that listens for the status of the animation.

So let's just print it out and see what the status actually is.

And if we take a look at the console along with our animation, you can see that once our animation has completed, we get the status animation status completed.

So we know that if we're doing a forward animation, we can detect when it's actually done animating by checking for that status of .complete.

But if we use a reverse animation, so let's again make it go from 1 to 0.

Now in this case, what we get is we get animation status dismissed called.

So the end of the reversed animation is dismissed and the end of the forward animation is complete.

So that means we can now quite easily check to see if the status is equal to AnimationStatus.completed, then that means the forward animation is done and we can make it go backwards.

So we can say controller.reverse from 1. And then we can check or else if the status is equal to AnimationStatus.dismissed, then that's the end of that reverse animation and we can make it go forwards again. So we start out by telling our controller to animate forwards, so it goes from small to large.

Once that's done then our status will trigger as completed and we make it go reverse.

And then once the reverse animation is done then we'll get a status of dismissed and we make it go forwards again.

So now if I hit run you'll see my animation bounce from large to small to large to small until eternity basically.

And this is a good point to mention that this animation will continue forever unless we actually trash the animation controller. At the moment, even if this screen is dismissed that controller still lives on and it's costing resources.

So whenever you're using animation controllers, it's really important that you tap into the dispose method and override it so that when the screen is going to be disposed or when this welcome screen state is going to be destroyed, we have to make sure that we also dispose our controller.

So this way it doesn't end up staying in memory and hogging all the resources.

So now the final type of animation I want to show you is what's called a tween animation.

It's basically a set of predefined tween animations that essentially go in between values.

So for example we have a starting color and we have an ending color, then our tween is going to go from the beginning to the end in a smooth transition.

So firstly let's go ahead and delete the addStatusListener.

We don't want our animation to be bouncing back and forth anymore.

And I'm also gonna delete my curved animation but I'm going to keep that animation variable, but instead I'm going to sign it to have a different value and I'm going to set it to be a ColorTween animation instead.

Now this color tween animation only takes two properties, a begin and an end.

So the begin is the starting color, so let's say we'll choose colors.red maybe and the end we could tween over to a colors.blue let's say. This is going to look really ugly, but nonetheless that works just fine for us.

So here we've created a color tween and once we're done defining the tween, we're going to call animate on it which is going to return an animation that uses this color tween.

And that's what we're saving into this animation variable here. And we're going to apply this color tween to our animation controller. So now we're building a color tween that's going to transition from red to blue and it's going to apply that animation to our controller to build an entirely new animation.

Now we can use that animation, so I'm going to change this height back down to 60 now and we're going to keep it non-animated.

I don't like the logo bouncing around. And instead we're going to apply that color right here.

So we're going to change the background color to animation.value. And now when I run the app you can see that the color changes from red to blue within a second.

So I can probably make that a little bit longer than one second just to make it easier to see.

So let's change that to maybe let's say over 3 seconds and we can see that transition happen a little bit slower as it goes from red to blue.

And in the console, you can see because we're logging the animation.value it's actually building a color that goes through each of the steps from red all the way down to blue.

And it builds out small increments in changes in order to go from the red to the blue.

So this is what tween or in-between animations are.

Now there's a whole bunch of other tween animations that you can use, such as the border radius tween or the alignment tween.

But what they all do is take a beginning value and an end value and you can structure the code in the same way where we create the tween, we animated it and apply it to an animation controller. And then we use the value of that animation inside our build method.

So the animation.value for the color tween is obviously going to be a color but the animation of a border radius tween will be a border radius that we could maybe apply to say our buttons.

So in this lesson we've looked at a lot of different types of animations and the ones that I'm going to actually keep is probably just the color tween animation where I'm going to begin with a maybe a grey or blue-grey color, and then we're going to end on a white color background, so colors.white. And we're going to change it back down to one second.

So this way when our app first loads up and the user sees our welcome screen, it's going to be a quick animation that just reveals our welcome screen. In the next lesson, we're going to do a deep dive on Dart mix ins and learn what this 'with' keyword does and what these mixins actually do to our class.

So if you already know a lot about mix ins and you have no problems understanding this code, then feel free to skip the next lesson. But if you want to know what this is all about, how it works and why we would use it, then I'll see you on the next lesson.

