All right.

So we've managed to get our app to work and the functionality is pretty much there.

But let's think more deeply about the problem.

Let's look at the bigger picture.

So in this lesson, I want to talk about design patterns, app architecture and dive deeper into Fluttter　state management.

So once you've started programming for a while, you'll hear these terms thrown around a lot,　design patterns, architecture, state management,　and the question really is why do we really care?

Why does it matter?

Well it's all down to complexity.

Complexity is the root of all evil really　in software.

And here's something to illustrate that.

Let's say that we decided to try and harness some energy.

So we build a windmill.

And every time the wind blows in UK that's pretty much every day,　then you capture the wind and you could turn that energy into milling flower or if you put a generator　on it then you can create electricity.

Now that's actually relatively easy to manage.

And if you check YouTube and you read some books, you could probably also build one of these.

But what if we wanted to scale up our production and instead we wanted to build a nuclear power plant.

Now for anybody who's watched a novel, then you'll know that it is a very complex task and there's a　lot of things that you have to think about, the design of the plant, fail safes, the architecture, how things　are going to be run.

So when complexity increases, the opportunities for things to go wrong also increase.

And the other way to think about it is it's a bit like say if you decided to rent a shop space and you　wanted to start a restaurant right? It's a mom-and-pop shop. It's just you and a couple of cooks and a　few waiters.

But let's say that we told you that you had to scale up your production overnight.

So instead of running a small startup, you had to run one of the largest corporations there are.

So you've given a whole bunch of money to recruit and organize your staff and turn it into an Amazon　or a Google overnight.

You can imagine that the same rules or the same things that worked for a small company,　it cannot work for a large corporation with you know thousands of people working for it.

So that's why different companies have different architectures, some better than others.

And it's because once you get to a certain scale once you have a certain level of complexity, then you　need a good architecture to ensure that everything works smoothly and that your staff are working as　efficiently as possible.

So if you look at the big technology companies, you have various architectures that people have created　to manage the company like a　for example a top down pyramid where you have the CEO who delegates to managers and then middle managers　and then employees or you have other types of structures where you have say an individual group of employees　and they all, you know, inter-communicate.

So we have the same kind of thing in software.

And one of the most popular architectures for mobile at least is something called the Model View Controller.

And this was first popularized by Apple in developing iOS apps.

So the concept is really simple for this architecture.

You have the model which manages the data, you have the view which manages what goes onto the screen　and you have the controller who is kind of the middleman between all of these components.

And so you end up with the MVC design pattern where there's the model that handles the data and logic,　the view which handles the user interface and the controller which is the mediator.

So you might have a case where the user taps on something on the on the app then the code in the view　will then send a message about those input events to the controller.

The Controller will then use those input events to decide what it should do next　and it could be that it will make a request to the model to request some data　right?

Say if the user taps in a contact app and they wanted to view Bob's phone address, then the controller　is going to make that request to the model class.

Now then the model class deals with the database or the data storage and it eventually sends back the data　to the controller.

The controller then uses that data to modify the view.

And this is a very simplified way of looking at how the MVC pattern works and it works well because　it separates out the entire large app into separate categories where we can have specialized skills.

So you'll have a department for welding,　you'll have a department for laser cutting and this way everybody knows what they're doing and they　don't tread on each other's toes.

Now when we think about Flutter though, a really important thing to consider is the difference between　imperative programming and reactive programming. Because traditionally native apps are built with imperative　code.

So it's kind of like you have a boss who tells you what to do.

'Today you have to', I don't know, 'write this report and you have to write it now.

Where's that report?

What's happening?'

But there's also the other alternative right?

It's kind of utopia idea of, what if the employee just knew what they had to do?

What if they had a list of tasks or they had an in-and-out tray and they would just get on with their　work not being commanded by any particular middle manager,　wouldn't life be so much better?

Well that's kind of how reactive programming is supposed to work.

And when we think about it in that way, it's kind of like we have these objects right?

Say a list view or a button and we tell them in the beginning as soon as we create them,　well this is your role in life. You　the button is meant to respond to touch events and when a user taps on you well this is the state that　you're going to change. Or in another case where you have a text widget, well the text widget is maybe　meant to track the state of a particular variable. When that variable changes, then the text widget has　to redraw right?

Reactive.

So in this case then instead of thinking about design patterns, we're thinking more about state management.

How do we manage the state of the app so that we can increase the complexity of our app　meanwhile still keeping it modular and easy to manage? Now an important thing I should say before we　dive in to design pattern for Flutter or architecture for Flutter or state management if you will, is　that different developers have different ideas about what is the best way of managing states and design　patterns and they feel really strongly about these things.

So whenever you come across any sort of blog article or videos about state management and design patterns,you'll see people usually having a very strong idea about what everyone should use.

And the important thing to remember is that there is no one way that covers all that should be used　every single time.

Because even if we're just talking about Flutter, there's a lot of ways that you could manage the state.

And you might hear these words brandished about a Scoped Model, Provider, BloC, Redux, MobX and similar　to how people who start learning programming always ask well what's the best programming language to　learn?

Well there is actually no best programming language.

It's kind of like if you learn how to use a hammer, well a hammer is a great tool but you can't do everything　with a hammer right?

You can try and you know cut a piece of wood with a hammer, but the end result is not going to look very　pretty.

And that's the same for programming languages and state management.

So at the end of the day, these different ways of managing state are just simply architectural blueprints　and there can be different architectures depending on say your requirements.

So you might have certain design patterns or state management methods that are better because they　fulfill a certain requirement of your app.

So similar to architectural patterns for creating an igloo versus a tree house versus a house that needs　to be on top of water, based on the different need you have　you might use a different architectural pattern to build your house　right?

Something that needs to be elevated in a tree is going to have very different architectural requirements　versus something that needs to stand on a lake.

Now there's also other things that you might think about when choosing your architectural pattern and　that's style based.

And this is probably the biggest source of contention because just as, you know, we might all have different　opinions about which of these houses look the best,　similarly programmers will have different opinions based on what is the best style for creating an architecture　for an app or for a program.

And the final thing to remember is that there's also the maturity side to consider because Flutter is　still quite new and it's growing and evolving.

People are still working out what is the best method for managing state.

So for example if you look at the Google IO talk this year in Flutter state management, for most people　they recommend to use the provider package. And in order to understand how the provider package works　at least the theory of it, head over to the next lesson where we're going to talk more on that.