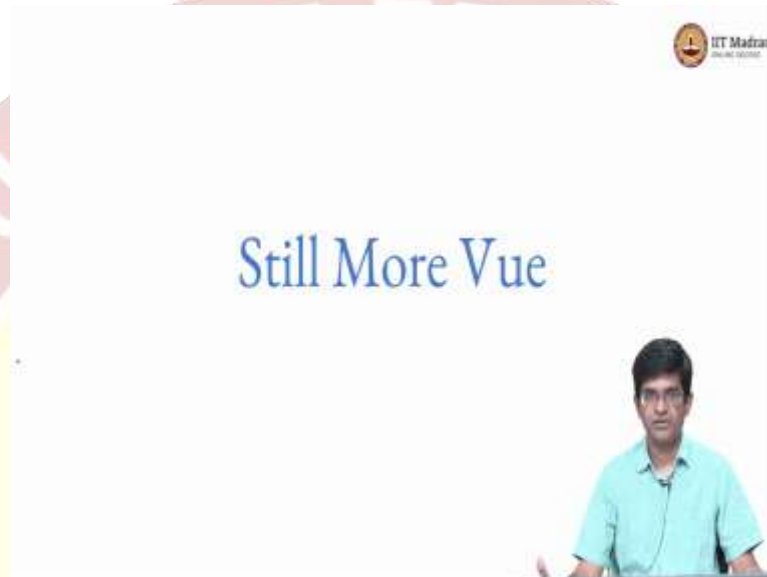




IIT Madras
ONLINE DEGREE

Modern Application Development - II
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Advanced State Management

(Refer Slide Time: 00:14)



Hello, everyone. Now, we are going to look at some more aspects of the vue framework, primarily towards sort of wrapping up what are the capabilities of vue. But having said that, in order to really make use of all the facilities that (you) vue offers to developer, you probably need to be using it in many more contexts. What we are doing over here is looking at some of the more interesting concepts, and to look in a little bit more detail into those to see which of those might be applicable to various different kinds of situations that you might come across.

(Refer Slide Time: 00:52)



Today, we are going to look at three topics, specifically; one is state management, then the use of routes in frameworks like Vue, and finally, look at a topic known as single page applications and a related topic called progressive web applications that we will look at briefly in order to understand what they are and what they are capable of doing.

(Refer Slide Time: 01:16)



Now, we are going to look at the topic of state management. We will first start by looking at it in a general context and then we will get into a little bit more detail on how it can be implemented within a framework such as Vue.

(Refer Slide Time: 01:29)



UI State

Core idea of declarative programming:

$$UI = f(State)$$

So, first things first, what do we mean by state management. We have already understood the concept of state in a couple of different contexts so far. One, of course, is the entire system state which is maintained at the server and which is responsible or other which is used by the server in order to decide how to respond to a given request from a user, from a client.

On the other hand, we also have this notion of what is called the application state as seen by the user, which means that there is some notion of variables or values that need to be stored at the front end. And as we saw earlier, this, the core idea of declarative programming is that the user interface UI is presumed to be a pure function of state.

What I mean by a pure function is, given a particular state of the system, the UI should be just programmatically derived from there. It should not depend on other factors that are unknown in the state. So, in other words, the state of the system is something that completely captures what the system looks like. And given a complete state of the system, you should be able to reproduce the UI perfectly.

So, this state that we are talking about could be different from the system level state, which could include the backend, such as the inventories, what are the different numbers of items of various types that are present in the inventory, and so on. You do not necessarily need to know that in order to generate a particular user interface. So, when we say state over here, we are talking primarily about state at the front end.

(Refer Slide Time: 03:08)



So, states need to be managed, meaning that, obviously, the idea of a state is something that changes with time. And over here, we come up with something called the state management pattern. And this figure on the right is something that I have taken from the documentation of a library called Vuex, which we will get to in a little bit more detail later. But you can see that essentially, we are trying to define some kind of pattern. Ultimately, that is what most of software engineering comes down to identifying design patterns that can be reused.

Now, the state management pattern essentially says we start with a state, which is what we call the source of truth about the internals of the app. So, in other words, you know the state, which means that you know the internal structure of the or rather the internal state of the app, and you should be able to derive other parts of what the app looks like, based on that state. Now, the view is a function of the state, which is why we have an arrow from the state to the view. Essentially, what it is saying is, given the state we should be able to compute the view.

Obviously, it is not that just given the state alone we can directly know what the application is going to look like. We also need to know this transformation function. So that will usually be some kind of a template which needs to be rendered with the state information. So, that is what that f of that function corresponded to. It is a template or some kind of a function that programmatically takes the variables, the state information and converts it into HTML or

something else that can be rendered on screen, but it is a fixed deterministic function, that is the whole point. So, given state and given that function, we can derive the view.

Now, what can the view do? It is basically a function on the state. It is a declarative mapping between the variables in the state and what appears on display, whether it be a screen or whether it be some other mechanism of interacting with the user. And now we have this third element over here called actions.


So, an action is something where a view is able to take or initiate certain actions. In the case of a web app, those actions primarily are clicking on links or filling out a form and clicking on a button. So, any one of those, the input to the action comes from the view. And the action in turn has some ability to change the state. So, the state changes, in other words, in response to the action.

So, what you can see over here is, you have what we can call a one way or a unidirectional data flow. We never have the view directly changing the state. The view has to go through an action. It has to specify a certain action. And that action has a specific way by which it can change the state. A state by itself does not invoke any actions.

So, in other words, just the internal state of the system left to itself is not supposed to invoke any actions that modify it. It has to go through a view. Now, of course, there are certain things like timers and so on, but we include those in how we essentially need to go through a part of the view, because after all, timer also can be thought of as something which takes a certain amount of time and then present something either to the user or to some consumer, the timer interrupt.

So, in that way, we have this sort of unidirectional data flow. States do not directly invoke actions, actions do not control what a view looks like, a view cannot directly change the state. But on the other hand, given a state, you can go forward to the view. You know what the view looks like. Given a view, you have certain actions that you can take. And given an action, it can change the state in certain ways. So, this is what we call the state management pattern.

(Refer Slide Time: 07:10)



The slide is titled "Contrast with MVC" in blue text. It features a list of three bullet points: "Here we are only looking at UI state: not system state", "MVC can still be used on server to update system state", and "Not either/or". In the bottom right corner, there is a small inset video of a man in a light blue shirt speaking. The top right corner has a logo for "IIT Madras ONLINE LECTURES".

- Here we are only looking at UI state: not system state
- MVC can still be used on server to update system state
- Not either/or

Now, you need to understand that we usually talk about the MVC model, a model view controller. Here, we are primarily looking only at the user interface state, the, not the entire system state. And MVC can still be used at the server end. So you still have the controllers that can go and change the underlying data models.

And in fact you would find that there are also different variants where people talk about things like controller views and view models and various other kinds of combinations of those terms. And say that, yes, even when you are talking about some kind of a state management pattern, you might have some notion of a controller. But rather than being too rigid about whether this is a controller or whether it is just an action update, it is probably better to think in terms of what is being accomplished over here.

The idea is that in the state management pattern, you want that one way data flow, you want to keep that separation clean, so that states always influence views, views in trigger actions, and those actions affect states. As long as you have that flow, you can keep the sort of management of the state relatively clean and under control.

And the important thing to keep in mind is MVC and the other state management it is not an either or proposition. It is not that you need to choose either MVC or the action state view model that I spoke about just before this. They refer to different parts of the system and they are both applicable in different contexts.

(Refer Slide Time: 08:43)



The slide is titled "Hierarchy - multiple components" in blue text. It features a list of bullet points: "Parent -> child" (with a sub-bullet "pass information through props"), "Child -> parent" (with sub-bullets "pass information through events" and "can directly invoke parent functions or modify parent data"), and two sub-bullets under the last point: "not desirable: breaks clean separation of code" and "harder to debug". In the bottom right corner, there is a small video inset of a man in a light blue shirt speaking. The slide is overlaid on a large, faint watermark of the IIT Madras logo.

- Parent -> child
 - pass information through props
- Child -> parent
 - pass information through events
 - can directly invoke parent functions or modify parent data
 - not desirable: breaks clean separation of code
 - harder to debug

Now, we already know that, in view, you have components that can be used in order to, essentially, specify what the view looks like given a specific state. So, each component has certain data associated with it. And the overall view app itself has certain state associated with it. And usually what happens is inside the high level vue app, I instantiate certain components and I can pass properties into those components from the parents.

The components can themselves contain sub-components inside them. And in all of those cases, the way that I usually communicate between a parent and a child component or between the top level and the child components of the top level, is that we pass information through props or properties. So, parent or child information can be passed by means of the props. What about child to parent? Usually, we pass that information back by triggering events. We can sort of trigger an event that is passed up the hierarchy.

We emit an event so to say. And when the event is emitted it goes back to the parent, it can then be cascaded upwards all the way to the top level. So, we can pass information from the parent to the child and from the child to the parent. There are ways by which you can also in a child component directly invoke parent functions, or even directly go and modify parent data.

Now, although that is possible, it is not advisable. And the reason why we say it is not advisable is not that it is fundamentally wrong from the code point of view. The software will work perfectly fine. The problem in many of these cases is more in terms of the maintainability of the

software. When you have a child that can directly go in and modify something belonging to a parent class or something that it is not supposed to have direct access to, the reason why people generally object to it is that it becomes, it makes the code harder to read and understand.

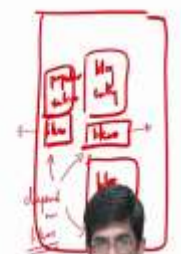
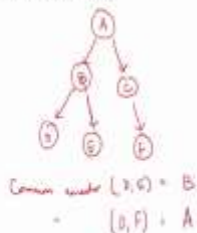
Because when I am looking at the parent class, I know that it is instantiating child classes or child components, but I do not expect those child components to directly have any impact on the parents' data, without having something very clear that tells me what it is that is modifying this. And by directly allowing a child to go and modify the parents data or to invoke a function over there we can, we are breaking the separation of concerns, which means that the debugging and maintainability of the code becomes harder.

So, please keep that in mind. The whole idea of object orientation and the whole idea of separation of concerns is not to say that a particular style of writing code is wrong or that it will give you the wrong results. It will probably give you the correct results. But it is harder to maintain, which is why it is usually strongly discouraged.

(Refer Slide Time: 11:47)

Problem: multiple components

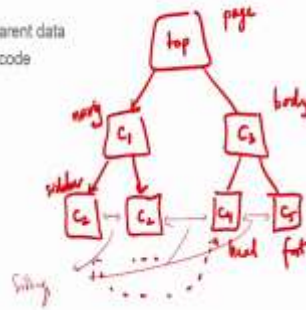
- Multiple views may depend on same piece of state
- Actions from different views may try to modify state
- "Sibling" components
 - At same or similar levels of hierarchy
 - Pass events up from source until common parent
 - Pass props back down to destination



Common ancestor (D, F) = B
= (D, F) = A

Hierarchy - multiple components

- Parent -> child
 - pass information through props
- Child -> parent
 - pass information through events
 - can directly invoke parent functions or modify parent data
 - not desirable: breaks clean separation of code
 - harder to debug



So, now, we saw that there is a hierarchy. And by hierarchy what I mean here is, I could have the top level component. And under it, I have a component 1. This intern instantiates, let us say, 2 copies of component two. And I have some other component 3 out here, which in turn could have its own components inside it.

What could these components be maybe the, this top level component corresponds to an entire page. This might correspond to, let us say, navigation, this might correspond to the main body, this might be a header or footer, inside the navigation there could be a sidebar. So, these are just options that I have over here.

My point is that you could construct a page, a top level page, by breaking it up into components that make sense to you, giving a template for each one of those components, and then sort of specifying how those components are aligned relative to each other. And as you can see over here, naturally, this makes sense.

The navigation, I would expect it to be, the navigation and body I would both expect to be parts of the top level page. Inside the navigation, I might have certain kinds of links. Inside the body I might have certain kinds of headers, footers and other kinds of structures. And that whole overall logical structure does make sense.

But you can see that there is a problem here. How does C2, for example, communicate with C4? Let us say that there is something in the sidebar that needs to modify something that appears in a header. And that is essentially where the problem with multiple components arises. Multiple

views in the system, each of those components define some part of the view. And multiple views may depend on the same piece of state.

An example is that if we take a top level page for a blog post for or even Facebook or something like that. I might have, let us say, a blog entry over here and it has a subcomponent or here that says the number of likes that it got. There is another blog entry and the number of likes that it got. Now, over here I have something else that basically says popular entries and these have some likes of their own.

So, clearly, what we can see over here is that this, this and this all depend on the number of likes. So, how do I handle something of this sort? I basically have something where there are multiple different components, all of which potentially depend on one variable. In other words, they depend on the same piece of state. And that when I say a piece of state essentially what I mean is some variable, some value that is present inside the system. Now, actions from different views may try to modify the state.

For example, over here I might have something which increases the number of likes. There is a button that allows me to click on it and then it increases the number of likes for that blog post. That in turn has to then go and update something else, some other component. Or on the other hand, I might have some way by which I can cancel or delete some number of likes from another view.

So, especially in this case, we have this notion of what are called sibling components, things that are at the same or similar levels of hierarchy. It is not a direct parent child relationship in other words. Going back to this previous one, these two are what we will call siblings. These two would also be siblings, for that matter, even C2 and C4 over here could be considered as siblings. Why, because they are not parents of each other. What they do have is some kind of a common ancestor.

So, then if I have some kind of a structure where I have a tree that is building up, I could talk about a common ancestor of D and E and this would basically be B, because B is the first point at which D and E both meet, but the common ancestor of D and F would be A. To be more precise, this is something called the least common ancestor, the closest one that is a common

ancestor, because I could argue that A is also a common ancestor of D and E, but it is not the closest to them.

So, finding this common ancestor is probably important in terms of being able to communicate data between the two places. If I want to send a message from D to E, I do not have a direct way of sending something from D to E. I probably need to send it from D back to the parent, which is B. And in this case, B knows how to get to E.

So, B can then transfer the data on to E. Things start getting complicated. So, clearly, this is not something that you would ideally want to have when you are trying to scale up your application into a large user interface with lots of different interacting pieces of data out there.

(Refer Slide Time: 17:16)

Solution - Global variables?

- Directly accessible from all components
- All components can modify a state variable
- All components can read a state variable for updating views

Problem:

- Keeping track of who modified what is difficult!
- Harder to debug/maintain



Problem: multiple components

- Multiple views may depend on same piece of state
- Actions from different views may try to modify state
- "Sibling" components
 - At same or similar levels of hierarchy
 - Pass events up from source until common parent
 - Pass props back down to destination



Another solution that sort of presents itself at this point is, why not simplify and just use global variables? So, what is a global variable? It is something that could be accessed in any part of the code. So, whichever one of these components, A, B, C, D, E, F, will have access to the global variable. By the way, one small digression over here, you will notice that in all these pictures, I am essentially drawing trees.

So, what is a tree? It has sort of a graph structure. It has these nodes corresponding to the components. It then has sub-elements. And naturally, when you look at a page and you start decomposing it into components, you will find that it has a tree structure. The simple reason being that, unlike a general graph, so I cannot, for example, have something where I also have E being a child of D, as well as being a child of P.

That does not really make sense, because I do not have such a notion inside an actual page. I cannot have one component over there being a member of two different components. So, the tree structure is natural, especially when we talk about document layouts. It is a natural way of decomposing a document layout into component parts.

So, now, the point is, we would ideally like to avoid this kind of going up and down the tree in order to transfer data between siblings. Instead, can we use global variables? They are directly accessible from all components. Any component can modify a state variable and any component can read a state variable in order to update its own view.

Now, the problem is, just like global variables in any programming language that you would have worked with, they are a maintenance nightmare. Once again, is it wrong to use them? Of course, not, because when you go down and look at it, ultimately, at the assembly language level, everything is a global variable.

The system has access to a pool of memory, which is then carved up into different storage areas. They are called variables for the humans to understand. But as far as the computer is concerned, it can go and access any one of those locations. So, all of these things are once again, even do not use global variables, is basically a directive to humans, saying that if you do use global variables, it makes your code harder to understand.

In particular, in the present context, keeping track of which component modified which part of the state becomes very difficult. That is similar in any global variable. Ultimately, if I have global variables in some C code, my problem is that when I go and look inside a function, I do not know where the variable was declared. I do not know what value it had to start with. I do not know how it came to me. And I do not know who last modified it.

And if I modify it, I do not know who else is going to see it next. So, that is the kind of sense in which global variables are bad. From the readability of the code and understanding what exactly it does, it becomes very hard to reason about any safety guarantees in the code. So, it becomes hard to debug, it becomes hard to maintain. So, we would like to avoid having such global variables.

(Refer Slide Time: 20:40)



The slide features a title 'Solution - Restricted Global access' in blue text. Below it, a bulleted list states: 'Global still required so all components can update their views easily', 'But changes should be constrained', and 'No direct modification of state variable' and 'Only through special mutation actions'. At the bottom left, it says 'Vuex - state management library for Vue.js'. On the right, there is a video inset of a man in a light blue shirt speaking. The IIT Madras logo is in the top right corner.

- Global still required so all components can update their views easily
- But changes should be constrained
 - No direct modification of state variable
 - Only through special mutation actions

Vuex - state management library for Vue.js

Having said that, they also do look like one of the most easy solutions to this problem, which is why the other solution that is proposed for the state management problem is to have a global variable, but with some kind of restricted access. So, the global variable is still required in the sense that the components can read the value from the global variable and update their internals easily.

And any component, for example, could register that it is dependent on a certain global variable, which means that reactively anytime that that variable changes this particular function in this or this view in this component will get updated. So, that makes it sort of easy. If there is a global read only store so to say, it could potentially make it such that it allows you to control who gets updated.

But the important thing is, if any component is allowed to also change the values of those global variables. That is what we want to avoid. So, instead, what we say is, let us try and constrain who is allowed to change a given state variable. And in particular, we will specify something called a mutation, a mutation action, which can be triggered by various components under certain kinds of conditions, but only in very specific restricted ways.

So, what we have is, we have a notion of how we could solve this global access problem. We provide global variables, but with restricted kind of access so that updates can be done only

under certain controlled conditions. And what we have is, Vuex is a state management library for the vue dot js platform that allows you to pretty much achieve this.

(Refer Slide Time: 22:23)



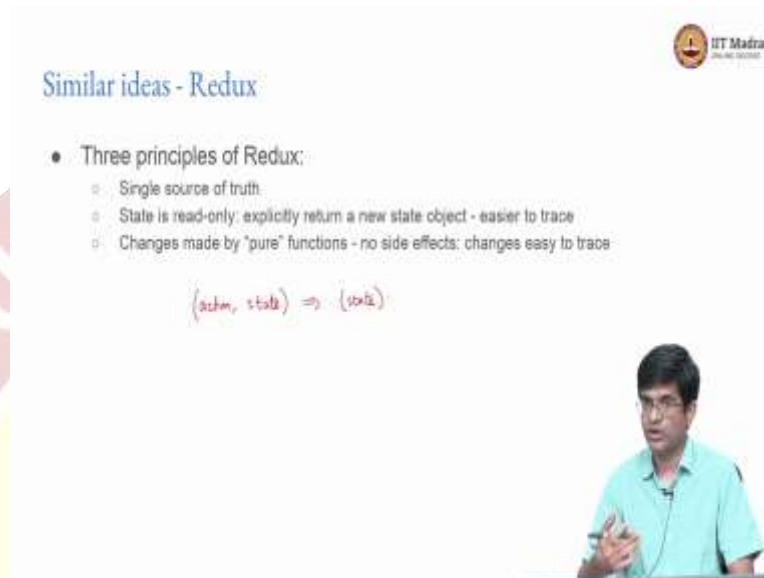
Now, there are similar ideas from various other languages and frameworks. And in fact, Vuex explicitly says that it is inspired by ideas from three particular sources. Now, those in turn have other inspirations of their own. So, clearly, Vuex is something that has a good history behind it. A lot of people have thought about various things and come up with good solutions. And in some sense, Vuex is trying to distill the best out of those.

Now, Flux is a similar state management framework primarily meant for React, originally developed by Facebook. It once again has this notion of unidirectional data flow. You have a store that maintains the state variables, a dispatcher that sends action messages. So, once again, like I told you, the actions have to be triggered. It is not that I can directly go and change a particular value somewhere. I need to trigger an action. And then of course, there is the vue that takes the values from the state and updates component views, very similar notionally.

So, understanding Flux would help you in understanding Vuex, understanding Vuex in turn would also help you understanding Flux, provided that you are careful not to get bogged down in the syntax of the different approaches or even the terminology. I mean, so there is a dispatcher over here and a store, these are terms that are not directly used in Vuex. But if you look behind it

and think, what exactly is it trying to do, you should be able to find connections with the equivalents in whichever framework you are trying to use.

(Refer Slide Time: 23:59)



The slide is titled "Similar ideas - Redux" and features the IIT Madras logo in the top right corner. It lists three principles of Redux:

- Three principles of Redux:
 - Single source of truth
 - State is read-only; explicitly return a new state object - easier to trace
 - Changes made by "pure" functions - no side effects: changes easy to trace

Below the list, a handwritten code snippet shows a function `(action, state) => {state}` in red ink.

Similarly, Redux, Redux is also primarily meant for React. It once again has three principles that they sort of enumerate on their main webpage. One is that the store acts as the single source of truth. Single source of truth meaning that ultimately everything about the system, about the UI state, and therefore, the UI, can be derived from the information stored in that state variable. Now, they also have one interesting take on this, which is to say that state is read only, meaning that you actually are, nobody is allowed to change the state.

What they mean by that is that if you are changing the state, you essentially need to create a new object which takes the old value of the state, updates whatever parts it needs to and return something new, which then needs to get updated. This might sound a little bit like unnecessary overhead, but it just sort of helps with reasoning about how this thing works in practice.

And the good part about that is it means that changes over here can be made by what are called pure functions. Now, a pure function is sort of a computer science construct or a mathematical construct which says that you have certain inputs, some computation and an output is generated, and output depends only on the inputs. In particular, there is nothing else that can get modified as a side effect of whatever function you called.

An example of a function with a side effect is something as simple as a `printf`, or a `print` statement in python or a `printf` in C. Think about what is happening. You are calling a `printf` with an argument which is a set of characters. But what is happening? It is not like it is computing a value and giving it back to you.

The value, `printf` may return a value, which is basically probably the string that was given to or `print` may return a value which is a string given to it or it may not. It depends on the implementation, depends on the language. But important side effect is that something got displayed on the screen that was not there before.

So, there are functions with side effects and there are functions with no side effects they are called pure functions. A pure function only returns one value explicitly, which means that you can effectively have this notion of something that says that I give it an action or a function to call, I give it the present value of the state, and this in turn returns the new value of the state. This is essentially how the computation can be expressed mathematically or in terms of computer science function.

Now, why are we going to all of these efforts over here, it is primarily because by having these kinds of restrictions, it allows you to reason a bit more clearly about what can and cannot happen within such a framework. So, Redux is also another very popular framework. Once again used primarily for react, but there are similar things for other kinds of languages as well.

(Refer Slide Time: 27:03)

The slide features a large, faint watermark of the IIT Madras logo in the background. The title 'Similar ideas - Elm architecture' is at the top left. Below it, a definition states 'Elm: Functional language designed for web application development'. A bulleted list defines the three components of the Elm architecture: Model (state), View (HTML generation), and Update (state updates from messages). To the right, a diagram shows a cycle: a blue circle labeled 'Elm' connects to a computer icon labeled 'HTML', which connects to a person icon labeled 'Msg', which then connects back to the 'Elm' circle. In the bottom right corner, there is a video inset showing a man in a light blue shirt speaking.

Similar ideas - Elm architecture

Elm: Functional language designed for web application development

- **Model** — the state of your application
- **View** — a way to turn your state into HTML
- **Update** — a way to update your state based on messages

Diagram illustrating the Elm architecture cycle: Elm (Model) → HTML (View) → Msg (Update) → Elm.

And a lot of these ultimately actually say that, in some sense, they are inspired by an architecture used in a programming language called Elm. Now, Elm is a functional language, which is designed for web application development. And literally, the Elm architecture basically comes down to this diagram on the right from their webpage.

What it says is the Elm language processor, so to say, it generates HTML output, which gets displayed on a screen. And that, in turn, triggers events, which are sent us messages back to Elm, which once again, takes in those messages, sees what its present state was, generates a new state, and therefore, new output, which is once again sent out as HTML.

In other words, it is what we have been drawing before, state, view, action, but simplifying it even further, essentially, you just have a view and a state, and they just communicate between each other by messages. So, this is, in some sense, probably the simplest that you can actually do for getting this entire structure in place.

So, you have a model, which has a state of your application, a view that essentially, it is a way to turn your state into HTML, which is what is going out over here, and a message which is an update, a way to update your state based on sending messages back from the view to the model. So, keep this in mind. Why are all of these things important, because ultimately they are forming a mental model for you to think about your application?

And by thinking about your application in as simple a form as possible by reducing it to things where you sent messages between different parts of the system and get back updates over there. It cleans up your code to the point where it becomes much easier to understand and maintain.