# **Learning Vue.js**

### **Is Vue for you?**

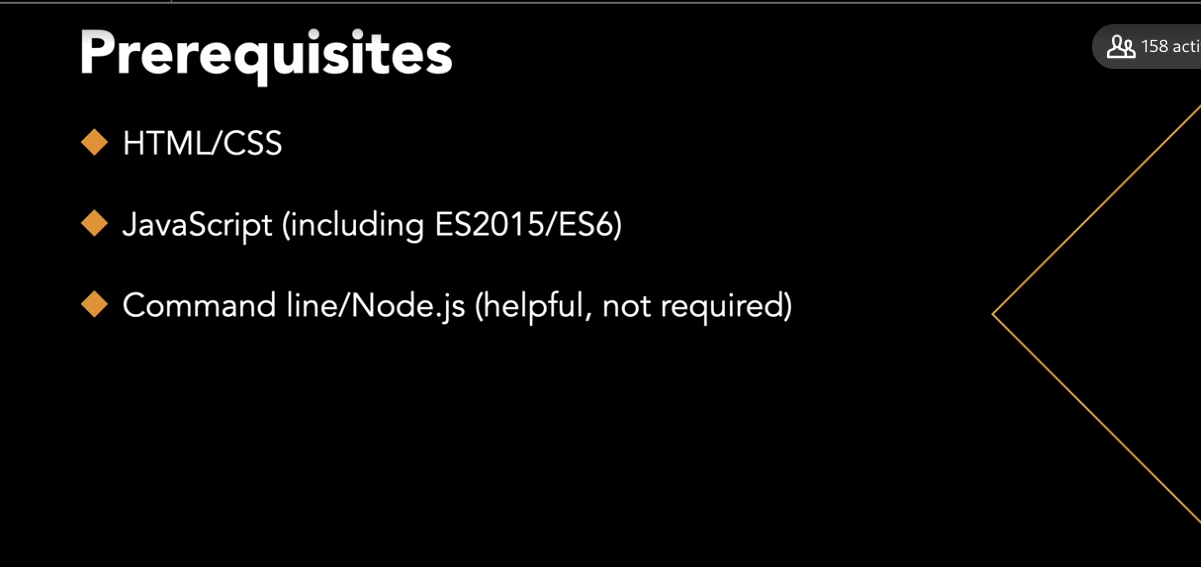
Selecting transcript lines in this section will navigate to timestamp in the video

- [Michael] Welcome to Learning Vue.js. I'm Mike Sullivan and if you're looking to become a more efficient and versatile front end developer, then you're in the right place. In this course, I'll talk about the reasons that we use frameworks like Vue.js in the first place and how to quickly get up and running with nothing more than a browser and a text editor. We'll take a detailed look at how to manage form controls and dynamic attributes of HTML elements. You'll learn how to take advantage of Vue's powerful reactivity system that allows us to keep our UI in sync with our data model and how to use Vue's flexible computed properties that are cached for optimal performance. There are many different reasons that developers choose one framework over another or end up loving a particular framework. If you're not a regular user of any framework yet, or if you're looking to switch to one that you're more comfortable with, then by the end of this course, you'll have a solid understanding of whether Vue can play a role in helping you get your work done or build your next passion project.

### **What you should know**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] To get the most possible out of this course, you should already be comfortable working with basic HTML and CSS. You should also have some experience with JavaScript including ECMAScript2015 syntax or ECMAScript 6 as it was formerly known. Some familiarity with the command line, Node.JS, and the npm package manager will be helpful, but the overwhelming majority of videos involve only the use of a text editor and a browser.

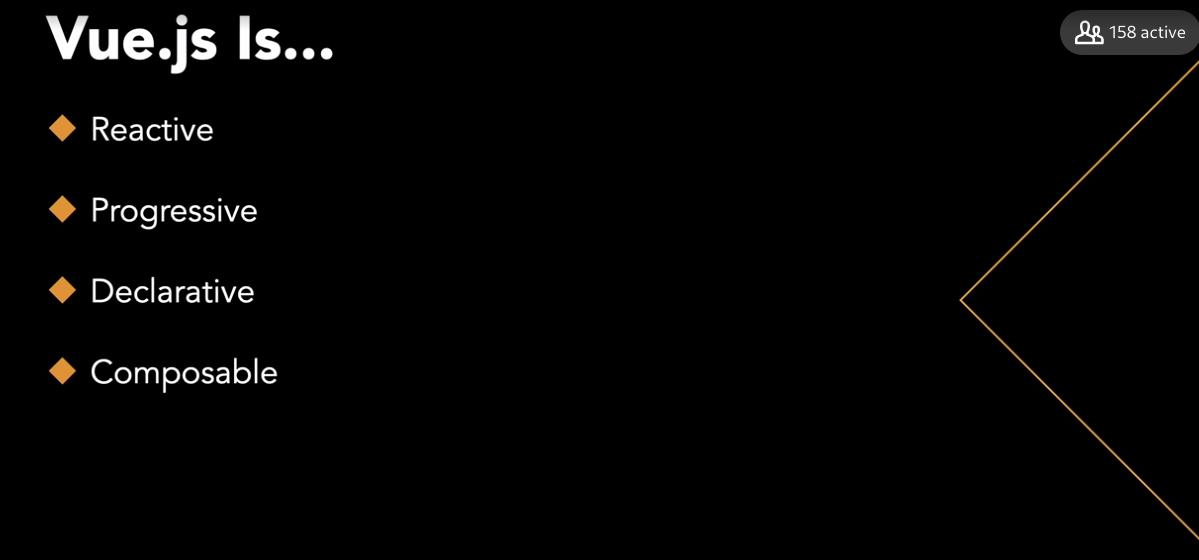


### **What problems does Vue.js solve?**

Selecting transcript lines in this section will navigate to timestamp in the video

- Before writing any code. I want to spend a few minutes talking about our motivation for using Vue.js and the problems that it aims to solve as a framework for JavaScript applications. In the days when libraries like jQuery were first widely used, or even earlier, it was common to write JavaScript code that looked like this. The purpose of this function is to add an element to the Dom or document object model. Our in memory representation of the webpage. In turn, updating the onscreen representation for the user. But after the call to appendElement on the first line this function is almost entirely concerned with updating the rest of the UI to make sure everything stays in a consistent state. While we could extract some of this logic to another reusable function, the point is that we need to explicitly do this additional work to keep the UI in sync, whenever we make a small change. And the more complex the UI becomes the more code it takes to manage all of that. Managing complexity is something I often think about when using Vue, the concept is self-explanatory but I probably first came across the term in Steve McConnell's classic book, Code Complete. He calls it Software's Primary Technical Imperative. If we can accept that managing complexity is at least one of our chief concerns as web developers whether we're talking about the code itself or just how our software can help a user complete a complex task, then it makes sense to use tools that are designed to manage or reduce that complexity that go beyond what's available out of the box. The native Dom API that you see used here on this MDN web docs page, was really designed with just making dynamic updates to the Dom even possible and not with managing the complexity that's involved with doing that in a large application. A better approach for our hypothetical example, if we could achieve it somehow, would be to set up triggers, when we initialize the app for those cascading updates to happen automatically, whenever we add or even remove or edit a row in our table. Then we could just call appendElement without having to wrap it in the function that also handles keeping the UI in sync. This makeSureEverythingStaysInSyncFromNowOn concept is where frameworks like Vue.js really shine. And it's a large part of why we use them to build our applications. We say that Vue is reactive. Its easy to set things up so that the UI is kept in sync with the underlying data, by automatically reacting to changes in that data. It's also progressive. It's easy to get started with nothing more than a script tag in order to add a small amount of dynamic behavior to your app. And then you can progressively add more features and tools as needed, as your app grows and becomes more sophisticated. It's declarative, we can set up the relationships and calculations required to keep our UI in sync when our application starts and then not have to worry about them later when managing user interactions and other updates. Finally, it's composable. We'll see that you can structure your application using Vue's component system for easier code reuse and easier reasoning about your applications logic and state. In the next video, we'll jump right in and talk about how to add Vue.js to a new or existing project.





### 

### **Adding Vue.js to a project**

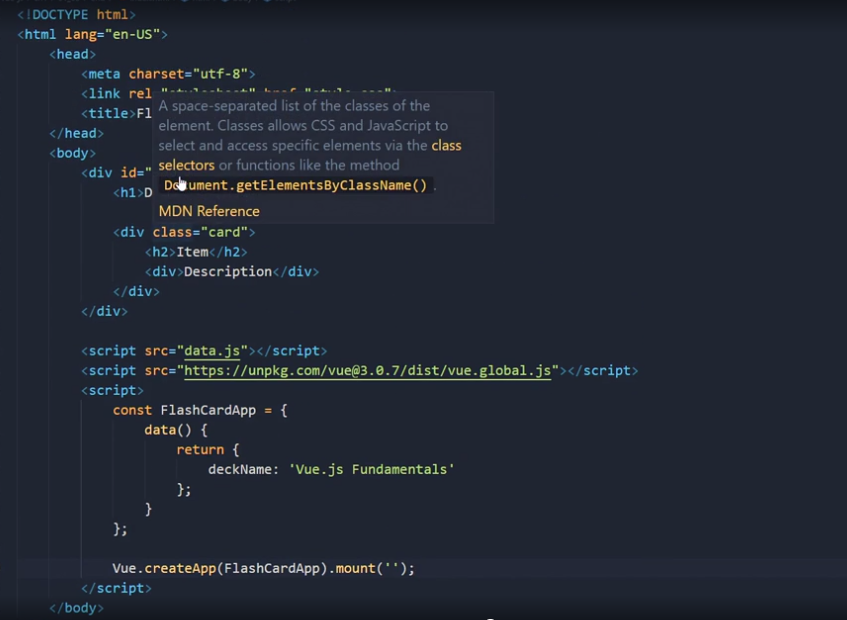
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Whether you're starting a new project or you have an existing project that you're considering using Vue for, getting set up is simple. At the time I'm recording this course, the Get Started button on the vuejs.org homepage still points to the version 2 guide. So instead, we'll click up here to get to the version 3 guide and then click Get Started. So, if I click this Installation button here I'm presented with several different ways to get set up. The first, Import from a CDN, really just means adding a script tag to an HTML page. A**nd I could do that with the CDN URL, or I could download it and host the script locally. If you're familiar with the command line package manager, NPM then you probably know that most JavaScript libraries and frameworks that are still used today are available as downloadable packages that can be automatically installed with those tools.** Vue CLI, or command line interface, is just a more sophisticated way of installing Vue with a package manager along with some other useful development tools and some skeleton code to help you get started. To keep things simple for now, we'll grab the URL from the CDN page, but instead of using this script tag I'm going to copy this URL and then use the URL of the specific version that that redirects to. So, in case, you are following along, you can ensure that you're using the same version that I'm using in the video, which is 3.0.7. In the exercise files for this video you'll find index.hmtl. This will be the starting point for our example project, which will be a flashcard app. This page pulls in a style sheet and a data file, none of which is specific to vue.js. The CSS just helps us see what's changing a little bit better as we build out the app and makes it a little prettier. So don't worry about its content. And the data file just contains an array of JavaScript objects that represent flashcards, each with a front and a back property. So, the front is the name of an item, or a concept and the back is its description, sort of like questions and answers. So, we'll add the script tag down here under the data file. And then, in the next video, we'll talk about how Vue can manage this section of the document, this div, to dynamically render it using some of the flashcard attributes from our data file.

### **Configuring dynamic elements**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In order to configure our Vue app so that it can manage part of our page, we'll add another script element down here making sure that it appears after the one that loads the Vue framework. Next I'm going to declare a new constant called FlashcardApp. And by the way, you'll see const used in this way in the Vue 3 guide as well. A discussion of why that makes sense as opposed to var or let is beyond the scope of this course, but feel free to ask in the Q&A tab if you want to know more about that. So this is not really the app itself, but an object that describes it by representing some options that we're going to pass to a method that will create the app. One of those options is the data that our app is going to work with, and we declare that with a function called data that returns an object. The object that the data function returns will contain, for now, just a property called deckName, the name of our flashcard deck. And this is going to be Vue.js Fundamentals. This is going to be a deck of flashcards related to Vue.js concepts, but this app could actually be used to present any deck of flashcards. And this is all we need to get going for now. So on a new line, I'm going to call Vue.createApp, passing in our options object. And then chain onto that a call to mount, which will take a selector for the DOM element that we want Vue to manage for us by mounting the app on it. In this case, we can use this div up here that already has an ID of app. So just like a CSS selector, I'm going to do #app. And now what does this allow us to do? Well, instead of having to write code that, say, selects this H1 element and imperatively sets the inner text, I can just declaratively say that I want the inner text of this element to be the value of our deck name data property, and I can do that using the mustache or double curly brace syntax. The other benefit to this is that it's very easy now for someone to come along later on, and that someone may be you in the future, and just quickly scan this HTML template, and have a good idea of what's going on as compared to when you have some code in a script somewhere that's explicitly accessing this element's inner text value. So if I go over to the browser now and refresh, I can see that deck name has now been replaced by Vue.js Fundamentals. One problem with this is that you might notice a brief flicker of the curly braces when the page is loading before Vue has a chance to take over and replace the div that it's mounted on with its own version. In the next video, we'll look at a simple solution for this, and also explore some other ways to bind data to DOM elements.



### **Binding to inner text and HTML**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Narrator] In this video, we'll start to get a sense of Vue's reactivity system and observe how the UI changes when the underlying data changes. First, to get rid of that flash of curly braces when the page loads. One we can do that is just to not put anything between the "h1" tags, to not use the curly braces, and instead opt to use the "v-text" directive which is a special attribute that we can use to bind the inner text to our "deckName" data property. So now when we refresh, there's no sign of curly braces because we aren't using them anymore. Now let's take a step toward displaying some flashcard data by adding our cards array to our view data. So, we'll add a new property in our data object called "cards" that will initialize with the value "cards" from the data file. And, this type of property declaration, where the name and the value are the same is so common that we've had a shortcut for it since ES 2015 and in this case it would just be "cards". So let's see if we can just display the content of the first card by setting the inner text of this "h2", I'll remove "item" here, to "cards", the first element at index zero, ".front" And, the inner text of this "div" to "cards[0].back". If we refresh, we can see that content but one problem we have is that you'll notice that there's a little bit of HTML in the description here. Since we're using "v-text" to set the inner text, that HTML is going to be escaped and if we want to render it as HTML, we should use the "v-html" directive instead. Another problem is that we don't want to show both the front and the back of the card at the same time. We'll see some cleaner ways to handle this later but one way to approach this is to add a new property called "flipped" which will keep track of whether this card has been flipped over or not and it'll be "false" initially. It's worth mentioning that you can use any single expression for the value provided to "v-text" or "v-html" even a ternary expression like this. So we're saying that if the value of "flipped" is "true", we want an empty string here 'cause the card's been flipped over, otherwise the content of the front like we had it before and then the opposite for the back. Sometimes when I start putting ternary expressions in my templates, it will feel a little bit messy to me and that's a signal that I might need to structure something differently. But, for now, it's good to know that we can do this if we need to. Let's use this as an opportunity to observe the reactive behavior that's so important to all modern JavaScript frameworks which will instantly bring our UI in sync with the underlying data whenever it changes. So first we'll save a reference to our created app in a new variable named "app" and now, having done that, we can refresh the browser and then if we open up the dev tools in the console, I can type "app.flipped" and notice that it shows the default value of "false". If I set "app.flipped" to "true", the DOM is updated immediately. This is the power of a declarative, reactive UI. Next we'll look at one way to set up a mechanism to allow the user to flip the card themselves.

### **Dynamic attributes and v-model**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Next I want to take a quick look at how to bind data properties, to an elements attributes. Let's say that we wanted to add a link for more information down here. The link text will be read more and, it will point to the Vue guide. But if we want to our app to be a generic solution for any kind of flashcard deck, we would probably have our URL as part of the data, and we can just call it readMoreUrl and initialize it to that same value. All we have to do now, to use our data property in place of this hard-coded href value, is replace it with readMoreUrl, and change href to v-bind:href. So now if I refresh, I have a link down here, with a dynamic href value. One way to simplify your template or your markup, is just to remove the v-bind part, leaving only :href, this is just a shorthand version of the v-bind syntax, and it's usage is very common. In fact for the rest of the course we'll be using the shorthand syntax, instead of the v-bind syntax. Next, I'm going to add a text field up here just to experiment a little with. (keyboard crunching) And if I wanted to pre-populate this with an initial value, I could do the same thing with v-bind value or just :value. Let's create a new data property called user input, and down in our data object we'll give it a default value. (keyboard crunching) And we'll just say default for now. So going back up here again I'm just applying this markup and CSS to make it all a little bit easier to see. And we'll invite the user to type something. So now when I refresh, I see the default value there, and if I go to the console, and do app.userInput to access our data property, I see that it has the value default, and, I can also reassign it to some other value, and once I hit enter, I see that reflected on the page immediately. But one limitation with this is that the data is only flowing in one direction, from Vue to the DOM. So if I go up here and I type, something else, it's not going to be reflected, when I evaluate app.userInput again it still says some other value. And I'm certainly going to want my app to have access to that updated value. To solve this problem, we can just use instead of :value, the v-model directive to set up that two way binding. So now when the user edits this field, it will automatically update our user input data property. So let's test this by checking for the word flip, and using user input, as a mechanism for flipping the card. So instead of flipped here, I'm going to ask if user input is equal to flip, and we'll do the same thing right below, instead of, flipped we have that conditional expression. So now when I refresh, and type the word flip, notice I don't even have to submit the form, I don't even have a form actually, and in fact, this field isn't even named, but as soon as I type the P, we got the card to flip, and then if I change it or just hit backspace, it unflips it, that's because the reactivity mechanism is based on the native input event that's being raised every time I type a character. So here we have a very declarative very reactive setup to handle this user input, but of course a simple click to flip the card would be better so we'll look at that next.

### **Handling events with v-on**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now we'll see just how easy it is to respond to a click event in order to flip our card. I've removed the text fields that we were using in the last video, and also reintroduced our previous logic where we were checking the value of the flipped data property. If we want to allow the user to click anywhere on a card and have it flipped to show the back, all we have to do is set up a click handler on this div, using the v-on directive. So v-on:click. And then for the value, I can add a statement for what I want to happen. In this case, I want to toggle the Boolean value flipped. So we just set it to its opposite. So now when I refresh, I can click to flip the card as many times as I want. v-on also has a shorthand syntax. And it's an @ symbol instead of the v-on:. Again, this is very common so we'll be using this form throughout the rest of the course. Another thing that's overdue to be addressed is the fact that we're only loading the first card in the deck at index zero. Let's change this to Index and we'll add a new data property called Index to keep track of the current index in the array. So down here, I'll add it after cards. Index, and we'll initialize it to zero to show the first card. Next, let's add a button to advance to the next slide. This HTML entity that I'm typing is just a forward arrow. And I'm going to wrap this in a div with class navigation. That's just something that's already in my style sheet that'll help me center this. Now we'll add a click handler with @click. And this time, instead of using an inline statement, we're going to encapsulate our logic in a method named move. So then down in my options object, I'm going to add a key for methods which takes an object containing the methods. And we'll add the move method. So this will just increment the value of Index. And while we're inside a method, we could access Index through our app variable down here. Or we could just say this.Index. And this will just increment the value again. One reason that it's nice to use a method is that it's a lot cleaner than putting something in the template if we need to add additional logic here. For example, it would be a good idea to reset the value of flipped. So we see the front of the next card and not the back as I'm cycling through. Now, when I refresh, I'm able to move forward in the deck and cycle through the cards. So I named this method move instead of next or advance because I knew I wanted to use it to move in either direction and not just forward. So up in the template, notice that I don't need parentheses here if I'm not passing a parameter. This is actually calling the method. But if I created another button with a left arrow, then I could pass negative one or one to indicate the direction I want to move. Back down here in the method, I could call this parameter change. And instead of doing Index++, I can now do Index plus equals the change. We should be careful to not go outside the bounds of our array. So let's just test, this.cards and then this.Index plus change for a truthy value just to see if that index exists. Otherwise we won't try to move. So either way, it will keep our HTML template a lot cleaner if we keep all of this logic in a method. So now I have my two arrows. And I can move both ways. In the next chapter, we'll take a deeper dive into how to work with various form controls in Vue.

### **Text field bindings**

Selecting transcript lines in this section will navigate to timestamp in the video

- In this video, we'll take a look at a very simple HTML form, and talk about how to manage text fields using Vue. In my code editor, you can see that this is indeed a form element with an input of type text and then a text area for lengthier input. And then below we have the submit button. Our view app is created the same way as before and I'm starting with a data object that has only the deck name property for now. So we've already seen that we can use v-bind value, or the shorthand version, just colon value to pre-populate a text field with a given value. And remember that value could be a data property, or a JavaScript expression. Or even something like math dot random. So let's use deck name instead. And if this were an existing flashcard deck that we were just editing, it might be perfectly fine to preload that value, and then just use a traditional form submission to update it. But if we did that, why would we bother using Vue at all in that case? Because it's still might help us manage some dynamic behavior that we want to add. For instance, I might want to display the length of this value here. So inside double curly braces I could say deck name dot length, and then the word characters after. So this shows us that the current value is 19 characters long. The only problem is that it doesn't change when I edit the fields. And that's because we haven't set up that two way data binding. We've used Vue to render a text field with a dynamic value, but we haven't told it to update the data property when the input changes. To do that remember, we need to use the v-model directive, instead of a dynamic value. So now that we see, as I remove characters or as I type more, we're writing the new value to our view instance with every keystroke and updating the UI accordingly. V-model and colon value work the same way with the text area. So if you're familiar with text areas, you know that a pre-populated value usually goes between the text area tags. You should avoid doing that when using Vue.js, even though it might seem to work in some cases. But usually we want that two way data binding, so we should use v-model anyway. Let's add a property called deck description. And then I'll add it to my data object initialized to an empty string. Of course, a more typical example of counting the length of an input would be to indicate a limit. So if our limit for deck description were say, 200 characters, then we could add the curly braces here and say, 200 minus deck description dot length. And we could say that that many characters are still available, or that many characters are left. So now if I refresh again, as I type you can see that number being updated on the fly. So that should give you a good introduction to managing text fields with Vue. Next, we'll take a look at check boxes.

### **Using v-model with related checkboxes**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Let's take a look at some ways that we can work with checkboxes in Vue. So, here you can see that I've added some options to the Deck Settings page, so far, I've only added some basic HTML, nothing related to Vue or any kind of dynamic behavior. Let's take a look at this first checkbox, include reverse cards just means that, the user would be presented with both sides of the card as they go through the deck. So, they may be shown a description and they'll have to come up with the name of the item and vice versa. One way we might deal with managing the state of this particular checkbox within our Vue app is to set up a data property that's associated with it, which we could call, reverse. This will be a Boolean value because it's just going to keep track of whether this checkbox is checked or unchecked. So, we would initialize to false. And then we would connect that to the checkbox using v-model, so, v-model = reverse and note that the value attribute here really isn't serving any purpose, but we'll leave it for now because we'll use it in another way in a moment. To help us see this in action, let's add another div down here where we just display the value of reverse. So, you can say the value of reverse is, and I'll make it bold and we can just display it as a string. So, now when I refresh, you can see the default value is false. And as I toggled the state of this checkbox, you can see a change there. Let's say you're dealing with a legacy system that's expecting a parameter named reverse, that set to either on or off when we post this form. To handle that, we could use the true-value and false-value directives. So, I could say, when this is checked, I want its value to be on and when it's unchecked, I want the value to be off. So, now when I toggle the checkbox, you see, on and off, instead of true and false, note that I didn't change the default value of false, so we still saw that one first. In this case, you'd probably want to just change that to off. Another way we can handle these checkboxes is as a group, in Vue, it's typical to use the same v-model property on a group of related checkboxes. So, instead of being named individually, we could use the name options and then copy that to the other two checkboxes. And then down in data, instead of reverse, we could have options initialized to an empty array. So, what that will do is store all of the values of the checked boxes in an array for us. And again, to see this in action, let's just display the array as a string up here. So, instead of value of reversed, I'll show selected options and that will just cast it to a string and show us what we've selected. So now, as I'm checking and unchecking these boxes, you can see the values are added to our options array and they're kept in sync with the state of the checkboxes. Next, we'll take a look at radio buttons and select elements which usually behave in a similar fashion.

### **Configuring radio buttons and select elements**

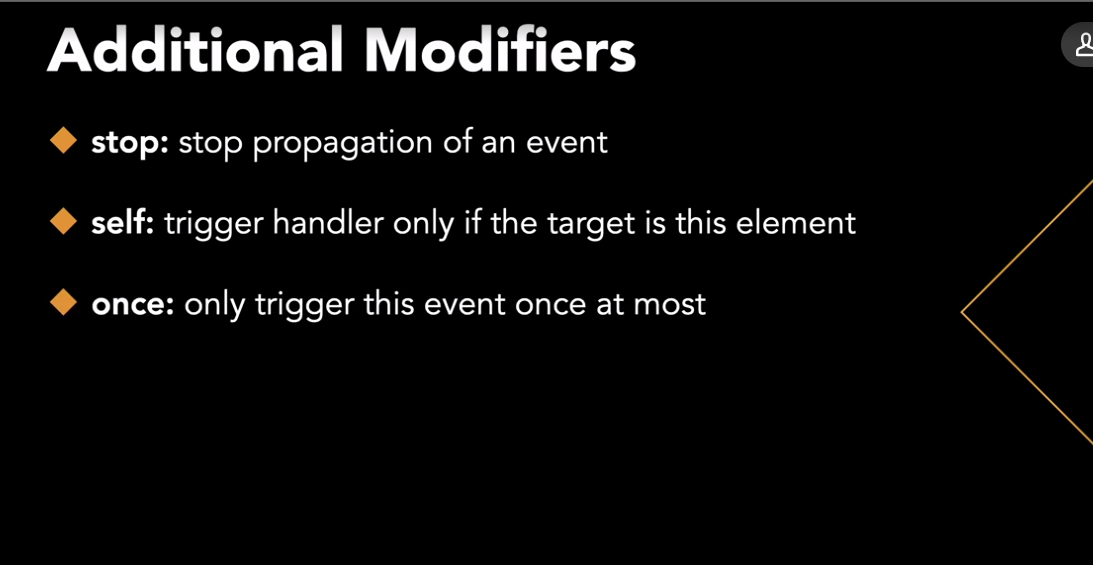
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now we'll find out how to bind data to radio buttons, and then we'll do the same for select elements, also known as drop-downs. So here we have a couple of related radio buttons in a field set for an additional option called leech action, with one of two values, tag or suspend. Leech is a term that I'm borrowing from a popular flashcard program called Anki, which gets his name from the Japanese word for memorization. Anki defines leeches as cards that you tend to have trouble with repeatedly. So you'll have the choice to either temporarily suspend them or just tag them as leeches for reference. Much like a group of related check boxes, I'll use the same V-model property on each radio button element. And we'll call it leech. So I'll add V-model equals leech, and then copy that to the next radio button. (computer keys clicking) And then in our data object down here, we'll add leech and we can let it default to tag because that's the first one that appears. Instead of displaying the selected value on the page or looking at it in the JavaScript console, let's track this value using the Vue JS devtools extension, which is available for Chrome and Firefox, or as a standalone app. I happen to be using Firefox at the moment. So I'm just going to search for Firefox Vue devtools. At the time I'm recording this course, the first value here will still install the Vue two version, although it might be different for you by the time you watch the course. But for me, the Vue three version is still in beta, so I'm going to look at this third result here and visit the GitHub repository, scroll down to installation, and click on beta channel next to the Firefox add-on. I'll click on the first release in the list and then click on this XPI file down here. Continue to installation, and allow it to be added to Firefox. And now you should see this little V in the upper right. If we go back to our page and click on that V, we should see a message telling us that Vue JS was detected on this page and suggesting that we opened the devtools and look for the Vue panel under the Vue tab. So I usually do that with F12, and if you don't see it right away, try refreshing the page, closing and opening the dev tools again. And there it is. So over on the right, I have the Vue tab, and if I click that, then I can click route which corresponds to my route Vue instance. Then I should be able to see all of my data over here on the right, including the value for leech, which I can confirm changes from tag to suspend as I toggle those radio buttons. Next, let's take a look at this select element here which offers some possible categories to assign to our flashcard deck. Again, if I want to manage the selected value with Vue, perhaps to post the options as JSON, to an API with an Ajax request, I can simply add V-model (computer keys clicking) and I'll call it category ID. (computer keys clicking) And then let's make sure that we add category ID to our data down here. (computer keys clicking) And we can initialize it with the same value as the default that just invites the user to select, which is an empty string. And to ensure that we've set this up correctly, (computer mouse clicks) I can refresh and take a look at the Vue dev tools. You can see this default category ID empty string and watch it change as we select different options from the dropdown. So seeing that ID changed there makes me confident that I can access the selected value in the dropdown, when necessary, from my Vue logic.

### **Adding modifiers**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] View includes several directive modifiers that provide shorthand syntax for adding certain behaviors. For example, if we look at the deck name property in view DevTools, we can see that the bound value increases in length if I just hit the Space bar a few times. We might want to write some code in a submit event handler that trims that string before sending it to the server. But we have a convenient alternative in the trim modifier which is basically as easy as just typing the word trim in our template. So we'll do that right after the V model for deck name. So this becomes V-model.trim. If I go back and refresh, and if you watch deck name now in the view DevTools, you can see that no matter how many times I press the Space bar, the bound value in our data object doesn't change. But of course this only handles leading or trailing white space. So if I start typing out here, I'll get a long string with that space in the middle. Again, it's nice that View keeps our form fields and our data in sync with every keystroke. It does that by handling the native input event. But what if we didn't want that? What if we only wanted to update our data property after the user has finished typing and then moved the focus away from that field? We can tell you to handle the change event instead of the input event, by adding the lazy modifier. So if we do V-model.lazy equals deck description then you can see that typing in this text area, has no effect on the value we see in the DevTools until I hit Tab. Next, the category ID values are treated as strings by the browser. You can see that if I select biology, say the value for category ID is now the string three. But it may be helpful at times to automatically cast those to integers. That can be accomplished with the number modifier. So if I change this to the V-model.number, equals category ID, when I refresh and select biology again, I now have three as a number type. Other modifiers exist for use with event handlers. The one that I use the most is probably the prevent modifier. So if I were to add a V on click or the shorthand version, just @click to this submit button and have it call a method named submit. Recall that I would need to add a key down here in our options for methods, and then define the submit method. For now let's just have it pop an alert that says that we're submitting the form. So now if I make some changes and then I click the create button, it pops the alert, but once I hit, okay the browser submits the form and refreshes the page. If I want it to prevent that default behavior I could just do, @click.prevent. Now we can handle this submission however we want, again, perhaps with an AJAX call and a client site update. There are a few other modifiers that will come in handy at times, including stop for stopping event propagation, self for detecting whether the target was this element or a child element. And once for limiting the handler to a single execution. For now, we'll wrap up this chapter with a discussion of View's computed properties feature which is a great example of how powerful its reactivity system is.



### **Computed properties**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Vue.js uses a reactivity system that tracks changes in the data that we declare when we create a view instance. It can tell when changes to the data necessitate re-rendering parts of the DOM. To improve the performance of this process, as well as the readability of our templates, we can make use of something called computed properties. A computed property is something that's derived or calculated based on existing data properties or even other computed properties. In practice, they work a bit like methods, but they have some advantages for organization and performance. Take a look at our mustache expression here that shows how many characters are still available for our description. As JavaScript expressions go, this one's fairly readable, but there are a lot of front-end developers who would say that even this is too much code for what we're claiming as a declarative template. What a computed property allows us to do is move this logic into our Vue instance and replace it with a simple identifier. In this case, we could use something like charsLeft. This certainly makes our template cleaner. Let's see how it works in our Vue options. After data, I'm going to add another key called computed which is an object containing my computed properties. for now, I'll add a single property called charsLeft which we declare as a function. And all that has to do is return the expression that was previously in our template which I had cut to the clipboard. Except, I also have to remember to change deckDescription to this.deckDescription, because now we're inside the Vue instance and not in the template where we can just use the bare property name. So if I refresh, I can see that this should be working the same way as before. This number is updated on the fly. And it's actually possible to set this up, using a method instead. You might be wondering about that. I could comment this out and move it down to methods. And the only difference is, I would need to invoke the method with parentheses up here, instead of just the computed property name. The reason that we prefer a computed property when possible is that they're cached and only recalculated as necessary. Whereas, the method version would be executed every time that part of the DOM is re rendered. At this point, we've covered a lot of basic concepts in Vue, but we still haven't talked about using conditional rendering or loops within our templates. So in the next chapter, we'll tackle those concepts and discuss Vue's built-in support for managing CSS. And that will open up a lot of possibilities for a dynamic user interfaces.

### **Conditional rendering with v-if**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, we'll turn our attention back to our flashcard presenter and clean up our template using some additional directives for conditionally displaying elements. If we look at this first inner div here inside the div that our app is mounted on, we have this click handler here and for the value of that directive we have a bit of JavaScript code that will be executed when a user clicks. And that's instead of perhaps a method call to handle the click event. Again, some Vue developers would prefer not to have any statements like this in their templates. Personally, I think a simple assignment like this is okay, but that's about as far as I would go with actually including imperative code in your template. And moving inside where we display either the front or the back of the card in the level two heading or div element here, depending on the value of flipped, this is definitely more logic than I would usually like to see in a template. And this type of toggling behavior can be more elegantly implemented with some simple if and else directives. Since we're using Vue, it will be v-if. So we add v-if here and I might've swapped these elements if I want it to start with v-if flipped, but I can also just say v-if. If the card is not flipped, then I want to display the front here, and I can take out this ternary statement and just change it to cards index.front. And then for the div, I'll just have v-else. No value needed, it's just the else condition, so if flipped is true. And again, I can remove part of this ternary and just leave cards index.back. And while we're cleaning this up, let's go ahead and create computed properties for the front text and the back HTML. So we could just use front and back in our template. So I'm going to cut this and replace it with just front. And then I'm going to go down to our Vue options, add back that computed key, and...we're going to define a computed property called front, and we'll need to remember to add, not only returned, but this in front of cards and this in front of index. So I'm just going to copy this for back and change front to back on the end there. And now if I go back up here, I already have front for v-text, and I can clean this up a little bit. And for v-html, we have back. So now, if I refresh, I see the same behavior as before, but it's implemented in a cleaner, more readable way. And by the way, because that H2 element is no longer there to push the description down, I did sneak some CSS into the stylesheet to add some padding to that div there, so it wasn't all the way at the top. By the way, there's also v-elseif, which can be pretty useful at times. For example, I might want to make sure that back actually exists before rendering it. So I might have a check for that there. And you can have as many elements with v-elseif as you want, followed by an optional v-else at the end. One thing that's important to understand about v-if and v-else is that they determine whether or not elements actually exist in the DOM and not just whether they are visible. So if we pull up the developer tools and let me flip this back, if we zero in on this H2 here, you can see that there's no sign of that sibling div tag until we click. So it's actually yanking these elements out of the DOM and then recreating them as we keep clicking and flipping the card instead of hiding them with CSS. But sometimes it's preferable to hide and reveal elements with CSS, and we'll talk about how to implement that in the next video.

### **Conditional rendering with v-show**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Next, we'll take a look at a directive that's very similar to v-if called v-show, and we'll discuss some of the reasons you might want to choose one over the other in certain situations. Remember what I showed you, at the end of the last video, when we use v-if and v-else, the element that's toggled off is actually removed from the DOM, but sometimes it's beneficial to not have you create and destroy these elements on demand, and instead just hide or reveal them with CSS. And that's especially true if it's resource intensive or time consuming to create the elements, and that's where v-show comes in, to demonstrate, we're going to introduce a short delay before returning the computed property back, which is referenced here. So before the return statement, we'll just let an empty loop run, so for zero, all the way to three times 10 to the ninth. We'll let that loop run before returning the back content, and this will just simulate an expensive operation that fetches or generates this content. So now that we have that delay, let's refresh the page and we'll see how the card flip behaves. So I'm clicking the card now, and you can see it takes a few seconds for it to show me the back, that's because the div element that was just created, it didn't exist until we clicked, and the value of "back" isn't computed until it's needed, but if I keep clicking, you can see that the delay has gone because the computed value of "back" is cached, remember, that's one of the advantages of setting this up as a computed property, of course, when I go to the next card, I've now changed the value of "index", which is a dependency of "back" and "back" must be recomputed sending it through that artificial delay loop once again. So I'm clicking now, and I have to wait again. Finally, we'll get to v-show. So back up here, we'll change v-if to v-show, they behave very similarly, except one drawback to v-show, is that you can't use it with v-else. But we can just do v-show="flipped". So refreshing again, once the page loads, the back content has already been rendered, although as a hidden element, and when we get around to flipping the card, which I'll do now, that value has already been computed, of course, if you noticed we did block the initial rendering of the page with that JavaScript loop, but in a real application, we might be loading that data behind the scenes with Ajax, which would avoid that completely, since we're on the subject of hiding and revealing things, it's a good time to talk about v-cloak, remember that if we wanted to, we could rewrite this with mustache syntax here instead of using v-text. Refresh again, now with that delay, those curly braces are especially visible, but if we wanted to hide that element until Vue has loaded, we can do it with v-cloak. So we're just going to add the v-cloak directive here on the h2, and v-cloak is a special directive that's always removed as soon as our view instance has finished compiling, by itself, it has no effect, but while it's present we can use it to hide the h2 element with CSS. So in my style sheet, I have v-cloak display: none, and v-cloak is in square brackets because it's like an attribute. Now, when we refresh again, we've hidden the mustache and the variable name from the user. In the next video, we'll talk about rendering a collection of elements using a for-loop.

### **Looping and list rendering with v-for**

Selecting transcript lines in this section will navigate to timestamp in the video

- In this video, we'll talk about how to use the v-for directive to iterate over a collection of items and render a series of related DOM elements. Our example for this exercise will be a card manager page, which shows all of the cards in a list and allows the user to edit them. So I've created a skeleton page that looks similar to where we started in chapter one. For each card I want to show the front and the back side by side. So that the user can see the whole deck at a glance. To start, I'm going to add a div with the class card pair. And that will allow me to place two child divs inside, each with class card. And these will be lined up nicely with Flexbox with the help of some CSS that's already in our style sheet. So again, we have our cards array in our data down here. And what we want to do is iterate over that array, and render a card pair div for every element. So we're going to add v-for to this card pair div. In the syntax is going to be V dash for and then a looping variable that we can name whatever we want. I'll call it card. And then just in cards. So we might say for every card in cards render one of these divs. And what that will allow us to do, is to reference this card looping variable anywhere inside this div that has the v-for directive. So for the front, we can add V-text is card.front. And then for the back we have V-html is card.back. So now when we refresh the page, we see all of the cards on one screen. But how can we edit them? Well, in this course we're not using any persistent data storage. So the updates will last only until the page is refreshed. But we can simulate, editing and saving. So for this exercise, we'll just make the front editable. And this card pair div is a repeating element. So, if we're going to turn one of these into a text field to edit it in line, we'll need to know the index in the array of the card we want to edit. To access the current index as we're looping, all we have to do is change card to card comma index. Or you can even use I or whatever you want to call it. And then surround these two variables with parentheses. So now we can add a click handler on the front card. And all that needs to do, is record the index of the card that we want to edit. So I'm going to add a new data property called editIndex, and we'll just set that to the current index. We could also have written a method that took index as a parameter if we wanted to. So then, let's go ahead and define editIndex in data. And we're going to initialize it to an invalid index so that we're not editing anything by default. So next, I'm going to add another div and using v-if. We can say that we should render another card div if the editIndex is the current index. So this will have class card also, but instead of card.front, we'll have a text area with v-model card.front to allow us to edit it. And then we're going to add v-else on the original front card. And we'll render that if we're not editing. So let's see what we have so far. I should be able to click any of these cards on the left, and it's transformed into a text area that I could use to edit this content. To finish this off, we can allow a user to press enter on the text area to simulate saving their changes. So a shortcut for doing this would be to use the keydown or keyup event. So we can say, on keydown, the same as V on keydown. So remember this is the shorthand syntax @keydown just like @click. And we can actually add a modifier to target a specific key, in this case enter. So keydown enter, and what do we want to do? All we need to do actually is just reset the edit index back to negative one. So now if I refresh, let's say I want it to change mustache syntax to add an additional hint. We'll add the double curly braces. And when I press enter, my value is updated, and I'm back to the static version of the card. In the next two videos. We'll take a look at some techniques for managing class and style attributes.

### **Binding style attributes**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, we'll take a look at how we can use dynamic style attributes in view. Whether we're using view or just writing static HTML. The use of style attributes is generally something we should avoid because it has a tendency to litter our markup with CSS properties that would be better off in an external style sheet but there are exceptions to that rule of thumb. And also it can sometimes be helpful for quick prototyping. A classic example for a dynamic style attribute is wanting to change a numeric value at a very granular level, such as a font size or a width which would be difficult to set up with CSS classes. Let's see if we can add an option to dynamically change the font size on our card manager page. So first I'm going to wrap this repeating div in another div where we'll be adding the dynamic style attribute. And that's just so that I'm not repeating it over and over again in the card pair div and also so that it doesn't affect the h1 up here. So we can set up a dynamic style attribute using the shorthand for v bind style, colon style. And one way to set the value is as a JavaScript object with CSS property names and values. So we can use the actual property name, font size. If we put it in quotes, or we could use camel case font size without the quotes. Then we can either put a static value in quotes like 15px or we would make more sense for our purposes is to build that string using a data property that we could call font size and then just add the px to it. Now let's go ahead and add font size to data and we'll initialize it to 15. And the reason we just use the numeric portion is that makes it a lot easier to change programmatically. So next, we'll go back up here and set up some controls for the user. So I'm going to add a static style attribute but we'll make it dynamic in a moment. And I'm just going to add text align center and cursor pointer so that the user knows that they can click here. They won't be able to click on text size, that's okay for now, but then we'll add a span that they can click on, which will increment font size. So just fontsize++ here. And that text is going to be bigger. We'll add a vertical bar with another span and you can probably guess this is going to decrement font size, and it's going to set smaller. So now if I refresh, you can see, I can click on bigger or smaller and can make the text way too big or way too small. In a real app we would probably want to use a method instead of the inline statement so that we could add some limits on either end and some additional logic there. This next example won't really be dynamic but just to show you another syntax I could take these two properties here. I'll cut those to the clipboard and I could move them into an object in data that we'll call text size control style. And again since text-align is not a valid JavaScript identifier, I would want to put quotes around it or I have the option of changing it to camel case. But since center is a value, I do want quotes around that and pointer and cursor we can leave as this. So having done that, I can change this style up here to a dynamic style attribute, colon style, and for the value, I can just give it the name of that object. So if I refresh, I see the same style here. I have my cursor and everything is aligned in the center. It's also possible to use an erase syntax here in case you have multiple objects that you wanted to apply. So if another style object were defined in data, I could do that and have both sets of CSS properties assigned. In the next video, we'll take a look at CSS class bindings which offer a little more flexibility in terms of being able to conditionally apply them.

### **Using component props**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Next, we'll talk about passing data to child components using props. So we might try changing this href in our read-more component, back to a dynamic attribute and then we could change the value here to readMoreurl. referencing our data property. But if we refresh the page now and look in our dev tools in the console if we have warnings enabled, we'll probably see one that says the readMoreurl. was accessed, but not defined. Like I mentioned before, the read more component can't just reach up into the parent instances data. What we want to do instead is have the parent pass this data down to the child component through the use of what's called a prop. So down in the component options I can add props to declare the data that this component is expecting to be provided with. So this can take several forms and one is just an array of string names and we'll include URL as one of the props that this component is expecting. And then instead of referring to readMoreurl. here I can just use its own prop URL. And then up in our main template on the read-more tag we'll pass URL just like any other dynamic property colon URL and the value is readMoreurl.. It's important to stop and note that we don't always have to use a colon here and make our props dynamic. And that's true for any attribute of any element in the template. For example, I could just pass a hard coded URL here and then remove the colon. But with the colon, we're signaling that readMoreurl. is the name of a data property and not a string value. So, notice that this is no longer underline and I can't click it there's no href there, but if I refresh I have my link working again. And if we can do that then it should be relatively simple to create a component out of the card itself. So this entire div here. So again, I'm going to cut this markup to the clipboard and use it to create a new component that I'm going to call flash-card. So next, after we register read-more we can register our new component flash-card. So we give it the tag and then the options. And we declare the template inside single quotes. I'm going to paste what I had on the clipboard. And remember, now that this is a string that's spread out among multiple lines. I do have to escape the new lines with a backslash. And let's also give it a name like the other one. We'll call it flash-card. Now let's think about what props this component needs to accept. Look at the data properties that are referenced here. We have flipped isRepeat, front and back. So again we'll add props and then an array with those names. Flipped isRepeat, front and back. Let's go back up to the flash-card tag and set it up. So it's passing those properties down to the card component because we set up those computer properties we can just pass flipped for flipped isRepeat for isRepeat and so on. One thing we need to be careful of though is that HTML attributes are not case sensitive. So instead of camel casing isRepeat we'll want to use a kebab case meaning a hyphen between is, and repeat. We could go ahead and change that in the props array in the component options as well if we wanted to, but that's optional. So now if we go back to the browser and refresh look at the view dev tools, I have a flash-card component and a read more component as child components of the route. And what's more, I can see all the props for this component on the right here. Notice that isRepeat is undefined because that's not in our data file, but just something we add on the fly to flash-card. A child component also can't access the methods of it's parent component. So got-it and again are not going to work yet but before we even get that far, we have to deal with another problem, which is that we're not allowed to just reassign the value of flipped when we click since flipped is data that's owned by the parent and only shared with this flash-card component you'll see a message in the console about that if you do click. Saying that props are read only. So in the next video we'll talk about how to handle that issue correctly.

### **Creating and registering components**

Selecting transcript lines in this section will navigate to timestamp in the video

- We've talked a little bit along the way about strategies for cleaning up our templates. One of the most effective tools that Vue offers us in that regard is its component system, which allows us to encapsulate sections of our template and make them easier to reason about and debug. Let's target a section of our template for component extraction. We'll start with something simple like the "read more" link here and to make it even simpler, we're going to temporarily change this to a static href attribute and just put the URL for the guide there instead of "read more" URL. But, we'll leave the "read more" data property down there so that we can reintroduce it soon. What using components will enable us to do is remove this element from the template. I'm going to cut it to the clipboard and then we can replace it with a custom element that I'll call "read more" that will look something like this with open and closed tags. You'll want to keep the tag names of your custom components lowercase for now, and include a hyphen if possible to help ensure that it's valid HTML and also not a tag name that already exists. And then to define, "read more", we can go all the way down here and use app, which points to our view instance and with the component method, we can say that this app uses a component with the tag "read more" and next we have an options object. It's very similar to the options object that we use when we declare the main instance, but because we're not mounting it on a certain element, but just making it available through the use of a custom tag, we need to define its template. So I'm just going to paste what I cut before inside these single quotes and note that if I were to break this up into multiple lines, maybe something like this, I would have to escape each new line with a backslash so that, that would still be valid JavaScript. Finally, we need to make sure that when we mount our view app already knows about the read more component. So we'll change this a little bit. First, we're going to just create the app then register the component and then call app.mt Now, if this is starting to feel like jumping through hoops just to encapsulate some markup and a custom tag, and if the idea of putting the component template in a string and having to escape new lines all feels a bit messy to you, then let me just say that I completely agree, but in a larger, more complex app, using components is an absolute must. And, there's a much simpler and more powerful way to use components that doesn't involve expressing the template as a string, but as plain HTML in a separate file. It does require setting up a build tool, such as web pack, so for now we're going to stick with this approach as our introduction to components. So now let's refresh and we see the same thing as before our read more link that we can click to get to the Vue.js guide and let's take a moment to look at this in view dev tools so I'm going to click the view tab over here again and I can see my route instance with its data and computer properties over here on the right. And I also see that I now have a child component that's called "anonymous component". You might be expecting to see the "read more" name here after all, it's surrounded by angle brackets, but when using components defined like this if we want to see a different name in the view dev tools, we'll need to add a name property. So I can just add name and here, the casing doesn't matter. I could use Pascal case like that, "read more". And then when I refresh now you see the "read more" component. So the pane on the right is blank because this component doesn't use any dynamic data. And in fact, components don't have direct access to a parent component's data. So in the next video, we'll talk about how to share data with child components.

### **Using component slots**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] One way to make our components more flexible, is through the use of slots and slot content. Again, we'll use the read more component, for a simple example. We're already allowing the consumer of this component, to provide a URL through that prop. But, what if we wanted to allow someone to provide custom content for the link text? We could certainly use an additional prop, but, what if we want it to be even more flexible, and allow arbitrary HTML content there? That's where slots come in handy. So down in the read more template, if I just change this link text, to a slot element, with open and close tags, that becomes a placeholder for any content that's between the read more tags in the main template, meaning anything that's between, the open and closed tags there where the cursor is now. So I could put the read more text there, and the main template now, but, let's try something a little different, and I'll wrap it in EM tags, just to make sure that this is working when I refresh, I'll say click here to read more. So now I have my custom link text, provided through a slot and I can see that, the link is still working. You might be wondering what if, you want to make this content optional, and perhaps fall back to the read more texts that we had before, if nothing is provided here. That's pretty easy. Just put the read more text, between the slot tags here. So if I refresh again, I still see the same link text, but, if I go back up to my main template and, remove this, then I see the default content. Sometimes you want to include multiple slots in a component. Maybe we want a copyright line, below the link here. To use multiple slots, give the other ones a name. So we could say, slot, name equals, copyright. Then, up in my main template, to provide content for that slot, I can use a template element, with the V slot directive, and the slot name like this. So I could say, copyright, 2021, flashcard out. If I still want to provide content for the default slot, for the link text, I can do that the same way as before, or I can use another template element with V slot default. So let's make sure this much is working. That looks good. And, next, it's important to understand that, if for some reason I wanted to use the URL, in the slot content here, I would have to do that, by referencing read more URL, and not, just URL. That's because the scope that we're working with here is still the route scope, because we're in the root template. And URL is only defined inside the read more component, after it's passed in as a prop. But what if there were some data in the read more component, that we wanted to reference here? And what I mean by that is, what if read more had its own data? For example, if this component, somehow kept track of the year, if we had, year 2021 here. In that case, we could make it available, to the copyright slot, by passing it as a prop to the slot. So I would have here, year equals year. And we might have multiple props like this, and we would access them by adding a name for an object that contains all of them as a value for the V slot directive, backup in our main template. So V slot copyright equals, we could call it props. And then instead of 2021 here, we could do, props.year. Let's check to see if we did that correctly. Still looks good. Now I would say that, this convention and its syntax here, is pretty far from intuitive. So don't worry too much about this right now if you find it confusing. The main takeaway here should be that, if we need to, we can access data from the child component, when providing slot content, from the parent component. We'll wrap up this chapter with a look at a more powerful way to work with components, that have proper templates as normal markup in separate files instead of specified as strings.

Give feedbac

### **Configuring custom events**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] In this video, we'll solve the problem of not being able to flip the card now that it's in its own separate component. One solution could be to set it up as a data property, just like we do with our route instance. Child components can also have their own data and it can be copied from props. So, we could set up a data property. We wouldn't use the name Flipped because that's already being used for the prop, but we can call it isFlipped, and then we could use Flipped as the initial value. And then, we could update our template to use isFlipped and just manipulate that value as necessary. But we'll be better off if we continue to let the parent have complete control over this property and that's so that we can make sure we're always resetting Flipped to false as we cycle through the cards in the parent component. We don't want both components to be able to change this value as that will get confusing to debug. So instead of changing the value when we click, as we're doing here, we're instead going to just raise or emit an event, alerting the parent component that the user has clicked on the card. So to do that, we just call dollar sign emit, and pass an event name, which I'll call it Flip. - And the dollar sign is used there just to make it less likely that you'll write a method with a name that collides with this one. And this is where these string templates get a little bit messier because we now have to escape the quotes around Flip, if you noticed. But what this allows us to do is to handle an event called Flip in our root instance now. So, in the flashcard tag, just like we're using at click elsewhere, we can now use at flip and put our flip logic there. And all we have to do is say flipped is the negation of flipped. So now if we go back to the browser and refresh, that warning is gone and we can flip our cards again. But remember, I mentioned that a child component can't access methods on the parent, so we'll see an error if we try to click Got It or Again and actually, we already see errors about Got It and Again being accessed but not defined. So let's see if we can fix that now. First, when a tag and all its attributes starts to grow like this, it's usually better to put each attribute on a separate line. So, maybe something like this. And some people actually like to put the closing angle bracket on its own line, or maybe do that. It's all really a matter of personal preference. Okay, so back to Got It and Again. Same as with the card click, we'll emit events instead of trying to call methods or manipulate any parent data. So I'm going to just call these events yes and no, because those names are short and sweet and it will also allow us to avoid any possible issues with case sensitivity. So then, back in our flashcard tag we can handle a yes event by calling our Got It method, and handle the no event by just calling Again. So finally, when I refresh, I don't see any errors. I can click Got It, Got It, Again, and I can cycle through the cards and everything seems to be working. In the next video, we'll look at a feature of components called slots which we use for managing dynamic content of a component.

Give feed

### **Installing Vue CLI**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Tutor] The best way to work with Vue components is by defining them in files with the dot Vue extension and these are known as single file components. Here's a component called hello, defined in its own file as shown in the Vue guide. Browsers can't natively support such files so they need to be compiled to JavaScript with a build tool like Webpack or Vue CLI or command-line interface which can configure Webpack for us and make the whole build process a lot simpler. Vue CLI is a mature system for rapid Vue.js development and it makes it incredibly fast to scaffold and configure a project that can make use of single file components. From the Vue guide, you'll find links to the Vue CLI docs in a few different places. One being the ecosystem menu up here. If I go down and click on Vue CLI and then click get started, I can click installation in the left-hand nav here. I'm going to be using the NPM package manager provided by Node.js. So I'm just going to copy this first command here. And if you don't have Node.js already installed on your machine, head over to node.js.org and download and install the latest LTS or long-term support version for your platform. Once that's taken care of, you should be able to install UCLI on your command line of choice using the NPM package manager, that's included with Node. I'm on a Windows machine using vs code. So, I'm going to open up a terminal with Control + backtick and then I'm just going to paste that command that I copied from the Vue CLI docs. I'm not going to talk about any of the initial output here because it's likely different on your machine. But once this is finished installing, we can tell that we've successfully installed the correct version if I go back to the Vue CLI docs and click on GitHub in the upper right. I can see on the right side that the latest release is four dot five dot 11. So again, on the command line if I do Vue dash dash version, I should see that same version four dot five dot 11 and I do. So next we'll go ahead and create a project with Vue create and the name I'll give it is flashcard dash app. There's a lot of options here especially if you look at manually select features, but for now we'll use the Vue three preview preset. And then when this is finished scaffolding the project we're given some helpful instructions for starting a lightweight development server. So let's change to that directory. CD flashcard app and then we'll execute NPM run serve. And as long as you don't have another web server or some other process that's using port 8080, you should see these links. Also note that if we wanted to deploy this to a production server, we would first want to execute NPM run build as it says here, and that would create a dist directory containing an optimized version that would work on any web server. But for now, if we just click on this local host one, we should see that we're up and running on the Vue dev server. So to get a sense of how all of this works with the local development server, we can start in our flashcard app directory with index dot HTML in the public directory. You see here, the div with ID app, just like we had in the other version, you won't see any script tags on this HTML page because the development server will just automatically inject them as it says in this comment here. Obviously that's not going to work if you plan on deploying this to a server that's running Nginx or Apache and serve up this HTML file. So you would want to run that NPM run build command as was already suggested in the console output. So to understand the JavaScript, our entry point is going to be in the source directory, main dot js. This is similar to the way we mounted our app before but instead of defining an options object here, we're importing it from app dot Vue. So let's go there next. In the same source directory, app dot Vue is our first look at a single file component. So we can call this the root component of our Vue app. It's sort of like the route Vue instance that we defined in our options object previously, only this time, it defines its own template within these template tags here, instead of with those messy strings. And that allows us to take full advantage of IDE syntax highlighting. And then we also have an optional style element down here. Notice it's also importing a child component, hello world, which it uses in its template up here. We'll talk a little bit more about what's going on in this file in the next video, but for now I want to show you one nice feature of the development server, which is automatic reloading. So if I were to line up my IDE and my browser side-by-side and then just change this message prop here, instead of welcome to your Vue.js app I'm going to just say flashcard app. You can see that the browser refreshes automatically so I can see the change right away. This is a really nice way to work and iterate on your projects. In the next video, we'll go through the exercise of creating a simple component and integrating it into our app.

### **Creating single file components**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Now we'll create our own simple component and add it to this page. Back in our single file component named app, let's take a closer look at what's going on here. This components option here is just another way to register components. Previously, we were using the component method to globally register them, making view aware of them so that they could be used anywhere in the app. Using the components option is a way to locally register components as child components of the app component. Let's go ahead and remove all references to this hello world component and then we'll create a new component for our deck name. So in the components folder, I'm going to add a new file decname.view and now we'll create a single file component from scratch. Most single file components have a template element, a script element and a style element. The script element will be used for defining our options object and then exporting it as a JavaScript module. So it's common to see here, export default instead of a named export, and then the options object. By the way, you'll notice that the view team tends not to use semi-colons at the end of JavaScript statements but I still find them sort of reassuring. So we can give our component a name. This will be deck name and this will take the name of the deck as a prop. So we'll add props here and remember we'll just have the name, name in an array. There are other ways to declare props and we can even include the type that we're expecting. And you can read about that in the view guide, but for now this will do just fine. So now that we have that set up, let's talk about our template. This is going to be incredibly simple just an H1 tag with our name prop interpolated there with the mustache syntax like so. We don't have to use style if we don't want to but this is a convenient place to add components specific CSS and it can make it really easy to update this component later because we'll have all the markup, logic and relevant CSS, all in one place. Let's just add a little bit of styling to the H1 tag to make sure this is all working as expected. So we'll just italicize it. Another nice feature of single file components is that we can make sure that this CSS doesn't affect anything outside of this component by adding the scoped attribute to the style element and I'll show you how that will change the rendered element in a moment. So now back in app let's import this component deck.name from dot the current directory slash components slash deck name. And that will pick up the deckname.view for us. We'll register it as a local component by just adding deck name here and then we'll add it to the template. So up here now that we're using Vue CLI and compiled templates, I can use the Pascal casing here. I don't have to use the kebab casing as we were doing before which would be deck dash name all lowercase. And remember, this is expecting a name prop. So we'll just pass in hard-coded Vue JS fundamentals. So we're not passing a data property or a JavaScript expression. So we don't use colon name, just name and we can have the open and closed tags or we can just use a self-closing tag like so. So now if I go over to the browser which has automatically been refreshed, we see that our simple component is being rendered correctly. And if you take a look at the dev tools and just zero in again on this H1 element you'll see that this data attribute has been inserted here. And the CSS rule that goes along with it has been updated as well. This pretty much guarantees that this CSS would never affect anything outside of this component. Even if we had another H1 tag. We've really only scratched the surface of what's possible with single file components and Vue CLI. And with view itself. We'll wrap up the course in the next video with some tips and ideas about where to allocate your learning time moving forward to get the most out of UJS.

### **Next steps**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Michael] I'll leave you now with a few ideas for where to focus next if you're interested in learning more and growing as a Vue developer. First, even if you're the type of learner who doesn't always find value in reading through guides and documentation I would suggest at least giving the Vue Guide a try. It's known for being very well-written and user-friendly and it will reinforce the concepts covered in this course while exposing you to the ones that we didn't have time to get to. One thing you'll see in there that was, perhaps, conspicuously absent from this course is a discussion of the Vue lifecycle. This describes how Vue instances are created, mounted and the process of updating data and re-rendering the DOM based on Vue's in memory representation of elements known as the Virtual DOM. If we had been using an API to fetch data for our example app, I would likely have used the created hook shown here. This is fired when the instance is first created and it's quite common to tap into that event to make an AJAX call since you can't just declare data that's fetched asynchronously from an API in the data object, like we did with our cards array that was already available. You'll also find out about other advanced features, like Vue Router, for building single page apps with client side routing and Vuex whose name is based on the flux pattern made popular by React. This can be a great solution for managing state and sharing data between components in more complex Vue apps. If you're on Twitter, definitely check out the Team page and start following the core members. Maybe even create a Twitter list to see them all in one place and find out what they're up to periodically. You'll be surprised what a tweet can lead to for your personal growth. I, myself, came to Vue years ago after seeing a tweet praising Vue for being innovative while also effectively borrowing things that already work well from existing frameworks. For more video content, please check out Ray Villalobos' Vue.js 3 Essential Training here on LinkedIn Learning. I hope, by now, that you have enough of a feel for Vue to at least be able to determine whether you want to go deeper and perhaps even adopt it as your JavaScript framework of choice moving forward. That's what I did over five years ago now. And I, honestly, haven't looked back since. Thanks for watching and best of luck with whatever this path or another may lead to.