1-How to Get Help?

Hello and welcome.

We're going to get started with course in just a moment and I'm really excited to teach you everything I know about Vue js, but before we do, I want to tell you how you can get help if you get stuck at any point during this course.

So, of course, in this tutorial, we are going to learn all the significant aspects Vue js. So, as you start to go through the videos and write out our code, if you have any questions or need any help troubleshooting, you can reach out to me on Udemy's Q&A section down below.

Now, I always recommend that you start there, because very frequently, if you are running into an issue, the chances are someone else has as well. Don't think you were alone. And there might already be some posts describing how to solve the issue that you've come up with.

You can also reach out to me via direct message on Udemy as well.

OK, so let's take a quick pause right here and start discussing TypeScript in the next video.

2-What is Vue JS?

In the series we will start from scratch and learn the different concepts in Vue. In this introductory video, I will briefly talk about what and why of vue.js and also the prerequisites to get started with Vue.

All right. Let's begin with what is vue.js. Vue.js is a popular JavaScript framework for building user interfaces. The last three words are really important.

The core Vue library is focused on doing one thing and doing that one thing really well which is building user interfaces. Vue does not focus on other aspects of your application like routing or http requests.

Now if that raises a concern as to how would you handle routing http and other functionality. Let me tell you that Vue has a rich ecosystem of other powerful libraries that you can integrate based on your app requirement.

For example, there is Vuex (Vue X) npm package for complex state management, Vue router for routing, Veautify for UI elements and a lot more libraries for every requirement you might have for your application.

All right now that we know what Vue is and for what purpose it is used. Let's see why you might want to learn vue.

I'm going to start with a comforting fact. Vue currently has 190.000 github stars on its repo which makes it the third most star github repository in the world. Does that mean it is better than react or other JavaScript frameworks? Absolutely, No?

It simply means that thousands of developers around the world enjoy working with Vue, because of this huge community you're going to find solutions to most of the problems that you face when building complex web applications.

This for me personally is a huge plus point. Knowing that I’m going to find resources online for any problem that I might face during development is always comforting, but let's focus on the technical aspects of Vue which makes it a wonderful framework to use.

The Vue docs categorize this into three core reasons. The first one is that Vue is approachable. Getting started with Vue is really simple. All you need is familiarity with HTML, CSS, and JavaScript.

You can add a script tag with a reference to vue.js and start building Vue applications in no time at all. The developer experience is also great with tools like the Vue dev tools which gives insight into your apps and also tools like the Vue cli with which you can quickly scaffold and manage projects.

Vue also has a component-based architecture. This lets you break down your application into small, encapsulated parts which can then be composed to make more complex UIs. Components make it possible to write reusable code which is always great for any project.

Another important point is that Vue is declarative. Now what does that mean? Well basically we just have to tell Vue what we want the ui to look like and Vue will do the hard work of ensuring the DOM is modified to reflect the UI.

This is in contrast with the imperative programming model where we usually list down the steps explicitly and ask the library to do one thing after the other.

The bottom line is that Vue will make it painless for you to create complex user interfaces by abstracting away the difficult parts.

The second reason is that Vue is versatile you have the choice of creating powerful single page applications from scratch using build tools like webpack or you can choose to incorporate Vue into your existing legacy projects and make progressive enhancement by making changes only to certain parts of your application.

The third reason is that Vue is performant. Vue measures just 20 kilobytes minified and gzipped at one time. Also because of virtual DOM only the necessary pieces in the DOM are updated which results in better performance.

Vue is a framework built by taking the good parts from a lot of other libraries and frameworks and then improving where necessary. So, if you work with react for example you might actually see a lot of similarities. Let me just tell that Vue is going to be a great addition to your skill set.

Now then what are the prerequisites to get started with Vue. As I’ve already mentioned HTML, CSS and JavaScript fundamentals are absolutely necessary. For this course I will also be making use of es6 plus features. So, our knowledge of modern JavaScript is also essential.

You don't have to be an expert by any means but there are a few concepts that make it so much easier to write Vue code.

My goal is to make sure we all advance from a complete beginner to being an expert with Vue. All right then with this introduction let's get started with a simple image generator application in the next video.

See you guys in the next one.

3-First App Preview

In the last section, we talked about What is Vue JS?

So, now in this section, let's talk about the app that we're going to build and get a better idea of how we're going to put it together.

All right.

So first off, a quick little mockup.

So, we're going to build something that we're going to call an image generator. In this little generator, our users are going to enter in a little bit of text into this input right here.

We're going to take that text and we're going to generate what is called robohash. Robohash is an easy web service that makes it easy to provide unique, robot/alien/monster/whatever images for any text. Shortly we will build an image generator.

And I can almost guarantee you that even if you've never heard of a robohash before, you probably more or less have seen one in action already.

So as a very quick example, let me show you a couple of generated images over on robohash dot org.

So, these are generated images right here.

They are created by taking some input string.

We take that string and then we turn it into a robot, alien picture like, so. These pictures are traditionally used whenever we need some type of user picture where the user doesn't otherwise provide some profile image or something like that.

Now, might sound like it's really intimidating to make it this, but in truth, you and I are going to use a library that's going to kind of automate the process for us and make it a little bit more straightforward.

You and I are going to be much more concerned with how we do stuff like, taking some user input and how we call some function inside of our Vue code and eventually get some type of image to display inside of our application as well.

Again, you and I don't really have to worry too much about generating this image.

So, with this in mind, I want to give you just a little bit more background on app and how robohash work.

So, the idea behind an generating an image line is that we'll have some type of input string.

The string input can be any string you might possibly imagine, like QWERTY, 1234 or just about anything else. A little function that you and I are going to use and again, this is going to be a function that we're going to use is going to take that input string and then turn it into a picture of sorts.

And the picture is going to look vaguely like the ones you see over here.

I'm just using these pictures right here as kind of a stand in of sorts.

One of the most important aspects of robohash is that if you put in the same input twice, you'll always get back the exact same image.

So, in this diagram, you'll notice that I put in QWERT, and I got out this identical one right here and then at the bottom, if I put in QWERTY again like a second time, I will always get back the exact same icon again.

So same input, same output.

OK, so I think that we've got a kind of reasonable idea of what we're trying to build here.

So, let's continue the next section and we're going to get started working on this application.

4-Online Code Editor Setup

In the last section, we spoke about how we're going to be making an image generator. We’ll now begin working on this application by creating a new JS Fiddle instance.

Now, just to be clear, we're not going to use JS Fiddle throughout this entire course.

In fact, this is probably the last time we're going to use it after this application, will then go on to using some actual code editors on our local machine to work on our apps, which is much more realistic of how you're going to really work on new applications in your own professional and personal projects.

So, again, last time we use JS Fiddle. Now I have another preconfigured JS Fiddle instance, that we're going to use as a starting point.

Here's a link to it on the screen.

So, I'm going to copy this link.

I want to also remind you that this is a case sensitive link.

So, I'm going to open a new window and I'll go to that link and of course, I see JS Fiddle immediately appear on the screen.

Now, this JS Fiddle instance I just opened, has just a tiny bit of configuration already in it and I want to show you exactly where that is.

Just so you really understand the starting point that we are kind of arriving at right here.

I'm going to find the panel and I'll click on this dropdown menu.

So that lands me on my hand JavaScript settings.

If you click language dropdown menu, you will see the languages that you can work with in here. I make sure that I choose Vue.

You'll also notice that Vue has already been inserted into this project. So, we don't need to import it or add it in any other way or anything like that.

OK, so close the menu and now we're pretty much ready to get started on our application but this time around, as we start working on this app, I don't want to just throw code on the screen like we did last time.

Instead, I want to give you a very well detailed and well-thought-out thought-out explanation of exactly what Vue is going to do for us and how we're going to use it.

So, let's come back the next section and we're going to talk about how we're going to use Vue to build our little robohash application.

So, see you in just a minute.

5-Vue Template and Instance

Now that we've got our JS Fiddle instance put together, it's time to start thinking about how we're going to actually use Vue to build this application.

I took the liberty of taking our mockup right here and I added a couple of notes to it. Just so we can better understand some of the different things that we're going to have to do.

So, here, you'll see my two notes. Pretty straightforward notes for right now.

The first one says that, yeah, we're probably gonna need some, like starting text and some form input to appear on the screen any time a user comes to application. By this note right here, I mean, to say that any time a user comes to our application, we want them to instantly see the text.

My Robohash title appear on the screen. Along with the description, text input and a form input and the output thing right here.

So, all those elements are supposed to appear on the screen of our browser.

The other aspect of our application that we want to be aware of is the fact that any time a user types into that text input, we want to run some JavaScript code that's going to create the robot image itself and then somehow put that onto the screen as well.

So, if you really think about these two elements right here, they're kind of describing two different aspects of our application.

On the one hand, we've got the aspect of somehow rendering or getting content to appear on the screen.

And then on the second item here, we're talking about somehow interacting with user input or responding to user input.

In the Vue JS world, we map up both different tasks to two different elements or two different parts of the application.

The first part is the first item we spoke about. Somehow describing the structure or the content of our application, like what appears on the screen when a user first comes to our application.

That task is handled by creating what is called a Vue template.

So, we create new templates to show information to users.

Vue templates are created in HTML inside of our application. When I say created in HTML, I mean to say that in JS Fiddle, we're going to create our Vue template inside of this HTML panel right here.

The first template that you and I create is going to be almost indistinguishable from normal HTML that you might create in any typical application but over time we're going to add in some advanced syntax to that HTML, which is going to turn it into our actual Vue template. Thing I want to remind you about here, and I know I've said it two times, but I'm going to repeat these many times throughout the course.

The Vue template is in charge of showing content to our users.

Now, the second aspect of our application, like handling user input, is going to be handled by creating what is called a Vue instance.

A Vue instance is created by writing out some amount of JavaScript code and the JavaScript side of our application.

And so, relating that back to our code and window, our Vue instance is going to be created down this JS panel here.

This Vue instance is what is responsible for dealing with user input through some fashion.

So, any time user types out some text into that text input, we're going to use our Vue instance to take that text, generate an image out of it, and then take that image and pass it back over to the template.

And the template will then present it to the user on the screen.

Now, one thing I want to mention here in particular is that this is just one possible way of many of structuring of Vue application.

So, you might be immediately curious, well, how else would we structure a Vue application?

To give you a very quick example, I just told you that we're going to create our Vue template on a HTML file or the HTML section of JS Fiddle.

But in some other uses of Vue, you'll sometimes see this template created over in the JavaScript side instead. In this first application that you and I are going to work on, we're going to first start by creating the template on the HTML side.

But once we put our application together, I'll very quickly show you how we might define our template over on the JavaScript side of our application as well.

Don't let this little item confuse you.

However, it's a real straightforward thing to kind of redesign your application and move where these different elements are defined.

OK, so just one more time just to make sure it's really crystal clear.

We use our template to somehow define the presentation layer of our application and we use the Vue instance to somehow respond to user feedback.

So, with that in mind, let's continue in the next section where we are going to create our first Vue template.

So quick break and I'll see you in just a minute.

6-Creating A Template

In the last section, we spoke about the differences between a Vue template, an a Vue instance. We're now going to get started on creating our Vue template.

And remember, this template is going to be the sort of presentation layer for our application. For the first iteration of our template.

We're going to write out essentially just plain HTML and you're going to look at this and you're going to say, hey, this is HTML.

What do you mean template?

What's template about this?

Well, don't worry.

Over time, we're going to come back to this template thing, and we'll add in some additional elements to it, to really turn it from being just kind of plain each HTML into an actual template.

So, without further ado, let's flip back over to JS Fiddle and we're going to write out our first template. In the HTML, I'm going to begin by writing out a simple div.

Then inside of the div, I'm going to put down the HTML that is going to the same structure of the mockup that we were just looking at.

So, inside that div, I'm going to put down an H2 tag to represent the header and then put down a div to hold this text and then a text input to represent the form input right here.

So that's all going to go directly into my Vue template, which is going to be located inside of this div.

So inside of here I'll place in h2 that says Robohash Unique Image Generator.

I'll then place a div that says input and then I'll put down my actual input tag as well.

Then underneath that div I'm going to put one other div in that's going to have the text output just like that.

All right, so this is our first Vue template right here and again, I know you're thinking, still this is a plain HTML.

What do you say in template?

Don't worry, we're going to come back to this little snippet right here and we're going to add in some additional syntax that's going to make it very clear that we're not quite working with HTML here.

Instead, we're really working with the templates.

So, with that in mind, let's take a quick pause.

We'll come back in the next section and we're going to create our Vue instance on the JavaScript side of our application.

7-Linking Templates and Instances

In the last section, we created our Vue template on the HTML side of our application.

We're now going to flip on over to the JavaScript side of our application where we're going to define our Vue instance.

And remember, this Vue instance is going to be responsible for watching user input and somehow reacting to that in some fashion.

So back over inside of JS Fiddle, I'm going to open the JS panel down here. Inside the jazz panel, I'll write out new Vue. Notice I've got one set of parentheses and then I've got one set of curly braces inside of this object right here or inside of the curly braces, we're going to add in a bunch of different properties over time.

All these different properties are going to serve to customize how this Vue instance behaves.

All the different properties are going to be stuffed inside of here is really where a lot of the complexity of Vue itself comes into play and it's understanding these different properties and using them correctly, that's going to turn you into a professional Vue engineer.

So, with that in mind, we're going to introduce the first property that we're going to make use of inside of you instances right now.

So, the first property that we're going to define in here is going to be a property simply called E L.

So, E L is short for the word element.

That element property right here serves to tie one Vue instance to one Vue template that has been defined inside of our application.

I'm going to pull up a diagram to make that a little bit clearer.

All right.

Here we go.

So, here's the HTML side of our application.

And then on the left-hand side, we have the JavaScript side.

We just defined our Vue instance in.

On that Vue instance, we defined a L property.

The purpose of this EL property is to tell Vue where our template sits inside of our HTML structure.

So, we defined our Vue template inside the HTML structure.

We have to tell you to go and find that template and we do so by using that Vue property to do so.

So you really quickly how it's going to work.

We're going to write out the code for it and then I'll tell you very clearly how the two really get tied together.

So first, I'm going to go back up to the HTML section here and on the top-level div.

So, my very root div right here.

I'm going to add in an ID of root like some then down on the Vue instance at the E-L property, I'm going to provide a string that has hash root.

OK, so now you'll notice that we've got to div with ID of root right here.

Let's go back to the diagram and I'll tell you exactly what this little hash root thing is doing.

So, our Vue template now exists inside of that div with Id of root.

By providing hash root we are providing a DOM query selector to our Vue instance that tells it exactly where its template exists inside the DOM.

So, by providing the string hash root, that means go look for some element inside the DOM or the HTML structure that has an Id of root.

Specifically, that hash or pound means ID. By providing that property, we've now instructed our Vue instance where its respective template sits inside of our DOM. Now, that tells Vue to somehow take that template and it tells vue that it is responsible for dealing with all the content of that template.

This is what really links together the JavaScript and HTML sides of our application.

OK, so now we've got a better idea of how to link together our Vue instance in our Vue template.

Let's continue in the next section and we're going to continue talking about how to build this application.

8-Step by Step Implementation

In the last section, we defined our Vue instance and then provided an E-L property. This told our Vue instance, what area the DOM was going to be responsible for.

So now this Vue instance right here is tied to all the content placed inside of this div.

You'll notice, however, that there's really apparently no change to our application at this point in time.

There's no difference.

There's no different content that's being rendered on the screen.

It's only as we start to add some more code to our Vue instance that we're going to start to see any change inside of our HTML output over here.

So, with that in mind, let's get a better idea of exactly how we're going to implement our application.

Remember, we've said several times that we want to make sure that any time user types inside of here, we generate a brand-new image and then show it next to this output section right here.

So, let's walk through the flow that we're going to use to implement that behavior.

All right.

So first off, we're going to walk through the exact series of steps that are going to occur inside of our application.

The first thing we need to do is detect any time a user has entered some new text into this input right here.

And there's a very small clarification I want to make or very small detail, I want to point out. I want to make sure that any time a user types one character, we generate a brand-new image and then show it on the screen.

We're not going to wait for a user to type out a bunch of text and then press enter.

So, it's with every single key press that we're going to generate a new image.

So that means that we really have to detect every time a user enters a new character into that input.

Once that occurs, we then need to somehow get the text that the user enters.

Remember, we're using that text to generate the image.

So, it definitely makes sense that we need to know exactly what the user just entered.

Once we get that text, we're then going to use it to generate the new image itself.

Now, this step right here, a little bit scary.

We don't really know how to do that just yet outside of using that API we took a quick glance at a couple of videos ago.

So, we're going to somehow use that web service to generate the new image.

But I think it makes sense that we're going to have to probably run some custom JavaScript code to do so.

So, for this step right here, we're probably going to have to have the ability to, like, run some function or something like that, any time a user enters some new text.

Then finally, after we generate that image, we're going to take it and we're going to show it right next to that output section right there.

Now, everything I just listed out, like these four steps right here might seem like they're very obvious.

You know, it might seem like well, Stephen, of course, we want to, you know, detect any time a user type something.

And of course, we want to generate the new image and then somehow show it on the screen.

But the reason I'm showing you this diagram right here is that for every one of these steps right here, Vue has a different feature built in to help you implement each step.

So, every one of these different steps that we're going to go through, we're going to learn about a different piece of The Vue API or the layer that we use to interact with that Vue instance that we just created.

So, with that in mind, let's take a quick pause.

We're going to come back in the next section and we're going to learn how to implement every single one of these steps bit-by-bit.

So quick break and we'll see you in just a second.

9-Defining Instance Methods

In the last section, we spoke about the four different steps that we're going to implement to get our application working. In this section, we're going to start off with step number one.

So, we need to make sure that we can somehow detect any time the user enters some new text and whenever they do, we're probably going to want to run some custom JavaScript code.

So, we're going to focus on this first step right here inside the section. To implement the step, here's what we're going to do.

You and I are going to add what is called a method to our Vue instance.

And we're going to make sure that that method gets called any time our user enters some new text.

So first, let's get a definition.

What is a method?

A method is a function that is going to be tied to our Vue instance.

That function can then be called at any point in time to somehow implement or somehow update our user interface.

So, we primarily make use of methods to somehow respond to user events, which is exactly what we're trying to do right now.

Let's first begin by implementing the method and then we'll figure out how to call it any time a user enters some text.

So first, the method. To add a new method to our component instance or give me our Vue instance right here.

I'm going to add a comma after the E-L property and then I'll write out methods like, so. Methods is going to have an object tied to it and inside this object, we're going to define all the different methods that are tied to our Vue instance.

Right now, there's just one method we care about. The method that's going to be called any time a user enters some text into this input.

But over time, if we had a more complicated template with like more text inputs or more buttons or just more ways of interacting with it, we might end up having many different methods defined inside this object.

So, let's begin by implementing this method that's going to be called any time a user enters some text.

I'm going to call this new method handle input change like so. handle input change is going to be a function, and it's functionally called any time a user enters some text.

Now, one thing I want to make sure is clear here that the word handle input change was just somewhat randomly picked by me.

We did not have to call this function handle input change.

We could have called it on User Typing, or it could have been ‘when user type’. The anything we want.

But in general, I find that using the codification of handle and in the event that occurred to be a really straightforward and simple to understand. So, handle input change makes it really clear to other engineers in the future that this function right here might be called any time someone types in some new text into our text input.

OK, so that's kind of step one of step one.

We just added a new method, which is a function that will be called any time a user enters some text.

Now, just to make sure that we know when we're kind of having some success here. Let's add a console log into this function and it can say anything inside the console log.

I just want to somehow see the console log appear any time user types.

So, I'm going to add in console log, ‘User is typing now!’ and then I'm going to open my console so I can see this console log appear to open the Chrome console.

You can right click anywhere on the screen and then click on inspect and then find the console section up here at the top.

Now, you might already see a couple of errors or something here inside the console.

If you do, that's totally fine. You can click on clear console to clear them out for now.

If you do see errors, chances are they are appearing from when we were first working on our application.

For right now, you should probably just ignore them unless they start to reappear, in which case then you're going to want to figure out what's going on.

OK, so now we'll try typing inside of here and just see what happens.

So, if I type inside this input, you'll notice that we do not see any console log appearing on the screen, which is totally fine.

You see, even though we've defined a method called handle input change, we have not instructed our Vue instance or our Vue template that we want this method right here to be called any time user types inside this input. We have to very directly wire up typing in this input right here or just wire up this input to calling this method.

They do not automatically get tied together.

So, let's take a quick pause and figure out how we will implement that step inside the next section.

So quick break and we'll see you in just a minute.

10-Defining Vue Directives

In the last section, we defined our first method inside of this methods object on our Vue instance, we defined a method called input, and it's a function that hopefully will eventually be ran any time a user enters some text into the input right here.

However, as we just saw, that's not the case right now.

We have to make sure that we somehow tell this input tag right here that any time a user enters text into it, we want to run this handle input change function.

To do so, we're going to start to change the HTML structure of our template, and this is where our template is no longer going to look like plain HTML. It's going to look like something a little bit more complicated.

So, let's add in the code that's going to somehow tie this input right here to our handle input change method and we'll talk about exactly what it's doing.

So, on the input element, I'm going to add something that says, ‘v dash on colon input’ and then I'll say equals and then inside of double quotes, make sure these are double quotes right here.

I'll say handle input change like so.

Now, I'm not going to let that run, I definitely don't see any errors over my console over here.

I do see some warnings, but those are totally OK to ignore for right now.

So, it looks like, you know, maybe something good happened. I don't know.

Let's try entering some text into the input and seeing what happens.

So, if I type inside of here, you'll see that I'm now getting a bunch of console logs. Let me zoom this in.

I get a bunch of console logs that say, ‘User is typing’.

So clearly, any time someone is typing inside this input right here, it is causing my handle input change function to run.

So, what happened?

Well, with this right here, we defined something called a directive.

A directive is a piece of template syntax inside a Vue that somehow enhances the behavior of otherwise normal HTML code.

So, behind the scenes, when our Vue instance first boots up, it looks at the EL property right here and as we said previously, it tries to find some element inside the DOM with an ID of root.

The Vue instant's finds are div with of evaporate here, and it scans over all the HTML that we have input inside of here.

When Vue scans over all this HTML, it looks for any directives like the one we just added in right here that have been added to our HTML.

If it finds any of these directives, it then does a couple of extra little steps to parse this directive and figure out how to correctly process it.

In this case, we have to find a directive that sets up an event handler.

The left-hand side of this that says V dash on specifically means we are trying to define an event handler.

We then place a colon and then the name of the event that we want to watch for.

In this case, we are watching for an event of type input.

So, any time a user enters input into this input tag, Vue is going to attempt to do something for us.

The thing that it actually does is defined by us on the right-hand side of the equals sign.

So, we said equals and then in big double quotes handle input change.

So, Vue says, OK, any time someone enter some new input into this text box right here, I'm going to try to run a method tied to my Vue instance called Handle Input Change.

So, it's specifically because we call this thing Handle Input Change that it matches up with a method that we defined down here, called Handle Input Change and again, we didn't have to call this Handle Input Change.

We could have called it handle input change and if we called it that, then we would have to make sure that we updated the directive as well.

So, we would have to say handle input change, when I update that, I can then still type inside of here and see the council logs appear.

We're going to be talking about a collection of different directives over time and I'm going to be completely honest with you in my opinion, I think that probably the most complicated part of you in general is understanding these directives.

So, I'm going to make sure that I put a lot of extra emphasis on making sure that you really understand how these directives work.

The event handler directive is one of the more straightforward ones, but over time, we'll start to see some more complicated directives.

OK, so in this section, we took the method that we had previously defined.

We added a directive to our input tag and that directive linked our input to the method we had previously made.

Before we move on, one thing I want to do very quickly, I just want to revert to the change that I just made there.

So, I'm going to revert this to saying Handle Input Change and I'm going to change my method back to being Handle Input Change as well.

OK, so it looks like some good progress.

Let's take a pause right here and we'll continue in the next section.

11-Retrieving Event Information

In the last section, we used a Vue directive to tie an event that occurred to this text input to the Handle Input Change method that we define inside of our Vue instance.

Like I said previously, viewed directives are one of the more complicated parts of Vue and we're going to go through too many times in this course review exactly how they work.

Now that we can run a function any time user types in this input, we need to now move on to our next step, which is to somehow get the text that the user entered.

To do so, we're going to use that Handle Input Change function.

You see, the Handle Input Change function gets called with a single argument that we usually refer to as the event object.

This is an object that contains a bunch of different information about the event that just occurred and, in this case, the event that occurred is an input event.

The event object has the actual text that the user entered.

And so, we can make use of that argument to figure out exactly what text the user typed.

So, let's go back over to our JS Fiddle instance and we'll figure out exactly how to do that.

I'm going to find my methods, object inside of my Vue instance.

Inside there, I'll find the Handle Input Change function and to that function signature, I'm going to reference the single argument that is provided to it.

Like I just said, we usually refer to this as the event object.

This event object has a property called target value and that will be a reference to the text that the user entered inside of here.

So, for right now, let's just console log that out and make sure that the text that was entered appears inside this function.

So, I'm going to do console log event dot target dot value like so.

The first event right here is a reference to the event that just occurred, Target is a reference to the HTML element that just had the event applied to it.

In this case, target will be this text input right here. Then value is a reference to the current value of that text input.

So, for us, that's the actual text that we care about.

So now we're logging that out. Let's try typing some text into this input and then just verifying that it somehow appears inside of our console over here.

So, if I enter some text, you'll very quickly see it appearing over here on the right-hand side.

OK, that's not too bad.

So, I think you can agree with me that now any time a user enters some text, we can retrieve the text that the user has entered.

So now that we've got this step two put together, let's continue in the next section where we're going to start moving through the rest of these steps.

12-Imperative vs Declarative Programming

In the last section, we figured out how to use that event, object to get access to the text that a user entered into our text input. You might be thinking that we're now going to move on to step number three, but instead we're going to do a quick little detour, OK?

We're going to do a little side topic quickly that is not super connected with our application as it stands right now.

So, in this detour, we're going to get a better idea of how to structure and build vue applications and give me a lot of fun, because we're not going to have to obsess over a little APIs or anything like that and we'll get a better idea of some of the big picture stuff around how to design the applications.

So, let's get started.

All right.

So, here's what we're going to talk about.

We're going to discuss the differences between imperative and declarative programming styles. Especially as applied to vue applications. In the world of Web development from maybe 2005 all the way to twenty thirteen or so, we practiced a programming style referred to as imperative programming.

With imperative programming, we write up code that lists exactly what our application should do step by step.

So, we say just like the diagram I had over here, whenever someone enter some text, take the text, generate new image, and then put that image on the screen.

This right here is an example of an imperative design flow where we list out exactly what to do step by step. In the world of declarative programming, we instead list out some rules that our application should follow.

We then provide what we refer to as some initial state to our application, and we let those rules kind of define how our application behaves.

Now, these very quick descriptions, I'm giving you probably don't make a lot of sense.

So, let's walk through the differences between imperative and declarative programming by walking through a little analogy.

OK, so I want you to imagine for just a second that maybe you and I are not building software anymore.

Maybe instead you and I bake cakes for a living.

So, we go into work every single day and we have to bake a cake.

Now, unfortunately, you and I maybe are very poor cooks, and we don't really know how to bake a cake.

So, our boss sits down and gives us a list of directions, essentially a recipe.

So, this right here is a recipe that might tell us exactly how to make a cake and I've got one recipe that is written in an imperative style.

And then we've got one recipe that is written in a declarative style.

So, let's walk through both of these recipes and get a better idea of the differences between imperative and declarative approaches.

Imperative approaches are characterized by step-by-step directions.

They tell you, step one, you start with step one.

You then go directly to step two.

You do step two, then you go to step three, step for step five and so on.

And then after you go through this entire list of steps, you eventually end up with some end product.

So, in the context of baking a cake, maybe our imperative recipe tells us to take these raw ingredients, mix these ingredients together, pour the ingredients into a pan, and then put that pan into an oven.

And then after you go through these four steps right here today, you end up with a cake or your end product.

In a declarative world, it's a little bit different, as you might imagine.

So, with a declarative recipe, we would separate out our recipe into many, maybe two separate parts.

We would start out with one part that lists what we might refer to as the initial state of our recipe.

The initial state would be like the variables, and in the case of a recipe, our variables would be maybe our ingredient quantities in the ingredient types.

We then take this initial state or this initial listing of ingredients, and then we apply a set of rules to them.

So, you can kind of imagine that we take these ingredients right here or this initial recipe.

We pass it through the rules one time, and then maybe after that first time, we then stick it into the rules a second time.

And then maybe after we go through the second time, we go through then a third time, and we keep repeating this process over and over until we eventually come out the other side with a baked cake.

So, it starts to get really important to understand how we might structurally structure these rules in a declarative approach.

So, for the rules in a declarative recipe, we might say, OK, let's examine our ingredients or our state.

If our ingredients have been mixed, then put them in a pan.

Well, clearly right now, with our initial state that doesn't quite match. We have not yet mixed our ingredients.

So, let's go down to rule number two here.

So rule number two says if the ingredients are mixed, mixed them together in a bowl.

OK, well, that's good.

We have unmixed ingredients, so mix them all together in a bowl.

So maybe now instead of having one egg, cup flour and sugar, we would instead have one bowl, mixed ingredients. We then take that state and apply it to our rules again. We've now got our mixed ingredients. So, we're going to look at rule number one.

We say if ingredients have been mixed, put them in a pan.

OK, well, that applies to us.

So, let's do that.

So, we now have one batch ingredients in a pan.

We now take that, and we go look at our rules again.

OK, that doesn't apply.

Our ingredients are now mixed in in the pans.

That doesn't apply.

Well, rule number three here says if we're in a pan, put it in oven for 30 minutes at three fifty.

OK, well, that definitely applies to us.

So, we take this, and we put it in the oven at for 30 minutes at 350 degrees, and then we come out with that with a baked cake.

So, in the world of declarative programming or declarative cooking, I suppose, we have some state that we start off with and then we take that state and apply the set of rules to it.

Now, I know this might sound like it's a real complicated approach for what we're trying to build right now.

So, I took the liberty of writing out our applications kind of rules of sorts in an imperative and declarative approach.

So, let's consider how we might build our application in an imperative approach and a declarative approach.

So, in an imperative approach to our current program, which is kind of what we're following right now, we might say that whenever a user enters some text, we want to do step number one right here, which is to get the new value out of that input.

Then we'll move directly on the step number two, which is to turn that input value into an image and then we'll go directly to step number three, which is to take that image and put it on the screen.

Now, this might seem like a very direct and easy to understand flow and you might be thinking, hey, imperative programming like this meshes with me.

I understand this.

This is what I'm used to.

Well, maybe for very simple flows like this, that makes sense.

But most Web applications that we spend any amount of time building are flows of what occurs inside of our application is much more complicated than this.

So even though I'm giving you a very simple example here, I hope that you can kind of extrapolate this example and imagine a more complicated flow.

We're going step by step in code.

Might be a little bit more challenging.

OK, so this is the imperative approach to our current application.

So, let's now look at a declarative approach.

OK, so here's our declarative approach.

So, if we took a declarative idea to our current application, then maybe we would say that we've got some like initial state or some like initial ingredients of sorts of a single variable called something like text input and maybe text input starts off as an empty string. So, this is supposed to be empty string.

Well then take this kind of initial state of sorts and apply it to this set of rules that we've defined inside of our Vue instance.

So, we'll say, OK, maybe we'll say, well, if a user enters text update, text input, well, that doesn't really apply to us.

We'll say if template is rendered, calculate the image.

Well, that doesn't really apply to us, and we'll say we'll have text input is updated, render the template.

OK, well, no, these quite seem to apply to us, so we'll just kind of wait around and wait for something to happen.

So maybe then at some point in time a user updates our text input and when that happens, we might apply rule number one right here.

So, if a user enters some text, we want to update the value of text input.

So, I'll now update that to whatever our new value is, which maybe is like, you know, VUE JS and then we also look at the other rules that are listed in here.

So, we just updated our text in that caused text input to update as well. So now we'll look at rule number two that doesn't apply. We'll look at rule number three.

It says, if text input is updated that just occurred for us, then we want to rerender the template. OK, so we'll render the template.

And if we look back at rule number two right here, if template is updated, calculate the image and then.

OK, well, let's calculate the image and then put that on the screen.

So, I know that these rules right here don't quite translate from the idea of, like baking to what we're doing right here.

But I think you can kind of start to get the idea where inside of our Vue instance will declare some type of data or some type of variable to sit around.

And then inside of that Vue instance will declare the set of different rules.

That tells our Vue instance how it should behave whenever something occurs inside of our application.

So, again, in general, you and I inside of Vue want to strive for this more declarative programming style as opposed to a more declarative approach.

Again, I know that imperative programming might seem like a little bit clearer and more obvious, and it might be more challenging to understand.

Why we would take this approach right here, but in general, a declarative approach scales very nicely for larger applications.

OK, so now we've got this idea of declarative programming in mind.

Let's continue in the next section where we'll talk a little bit more about how we can kind of implement these rules right here into our Vue instance and get our application working.

So quick break and we'll see you in just a minute.

13-Declarative Apps with the Vue API

In the last section, we spoke about some of the differences between imperative and declarative programming.

We're now going to take this idea of declarative programming and apply it to our actual application with some real terminology that is used in Vue instances.

So, the first thing we're going to do is take a look at each of these steps right here.

And I want you to consider like the actual purpose of each of these rules or maybe a better way to phrase it would be to imagine when each of these rules would be applied.

Let's look at what I mean. Here we go.

So, at the very top, we start out with that initial state of sorts or that like initial list of ingredients.

We might also refer to that as our data inside of our application.

The first rule right here where we say watch for a user to enter some text and whenever that happens, update, text input.

We'll try to characterize the purpose of this step right here.

I would say that the purpose of this step is to update our data or to update our state.

So, this is really an update action or kind of like an action that changes things inside of our application.

The second step right here, which says that whenever the template gets re rendered, we need to somehow calculate that image to show on the screen is a step that somehow uses our data or uses our state to show stuff on the screen.

Another way to put it would be to say that this step right here kind of consumes data to present it to users inside of our application.

The last step down here, which is to say that if the text input is ever updated, rerender the template, this one is a step that just kind of happens automatically with a Vue behind the scenes.

So, any time you and I change our state or change our data inside of our application, our Vue instance is going to automatically update everything on the screen.

So, this step right here isn't super important to us. Right now, I'm a little bit more curious about how you and I can represent our initial state and these two sets of rules right here inside of our application.

So, again, just one more time, this step right here where we say, OK, let's take that new text and update, text input.

That's a step that really updates our data and then turning the text into an image is a step that kind of consumes our data, so to speak.

So, with that in mind, let's now apply some more precise terminology to each of these steps.

So, these are terms right here on the right-hand side of the different properties that you and I are going to eventually define on our Vue instances to implement each of these different steps.

So first off, at the very top, we've been referring to this kind of initial list of ingredients or our initial state of sorts.

In a Vue instance, we refer to this as a property defined as data.

So, you and I are going to define a data property inside of our Vue instance, and that's going to tell our instance what the kind of like starting ingredients of sorts are inside of our application.

Next, you and I have kind of already taken care of this step right here.

But to really break it out, you and I are going to define a couple of different methods eventually. One side of this application, we're just going to define the one, and that's the handle input change method that we've got right here, but the purpose of that method's property is to define a bunch of functions that are supposed to somehow update our data.

So, all the functions you're going to see on methods is going to describe how we might update the data inside of our application and thinking about what it's being used for inside of our application right now, yeah, that definitely matches up.

We're saying that any time a user instigates a change, you and I want to update our data and so to do that, we're going to use that method's property inside of our Vue instance.

Now, the last one right here or second to last is a little bit more complicated.

So, any time you and I want to kind of consume that data and get it into our actual template or show it on the screen in some fashion, we're going to use a computed property.

OK, so again, these are three pieces of terminology or three properties that you and I are going to define inside of our Vue instance eventually.

So, we've got data to kind of initialize our data inside of our application.

We've got methods to change it and then we've got computed to somehow consume it and get it to be visible on the screen.

OK, so I think, you know, that terminology may be starting to make a little bit of sense. Don't worry, we'll see a lot of examples around it, but just to really drill home this terminology right here, we're going to look at kind of two more slightly repetitive diagrams.

So here we go.

So just to be clear, data methods and computed, these are all parts of the API. Data defines the initial state. Methods defines the ways that we can change that state. Finally, computed defines how we're going to take that data and turn them into viewable values that can be displayed on the screen.

Now, obviously, the text that a user enters our application is itself like a viewable value but for you and me, we're trying to create an image in here.

So, a viewable value would be taking that text and turning it into image. Now, just in case all this terminology is still just way too complicated, I took even this diagram, and I made a more simple version of it, one that hopefully will even stick in your brain more easily.

OK, so here's here's as simple as I can put it. It's a really just put it down as simple as I possibly can.

We use data to describe how everything starts, like how all our data starts.

We use that method thing to change our data and we use the computed thing to turn that data into a viewable value.

So, this is as concise as I can possibly make it.

OK, so we've now got a good idea of some of these different parts of the API.

I'm sure you're still kind of unsure of where these terminology things right here apply.

So, let's continue the next section and we'll start to apply these data methods and computer things and trust me, you'll very quickly get a very good understanding of what's going on.

So quick break and let's take care of this in the next section.

14-Data, Computed, and Methods

In the last section, we took the idea of declarative programming and applied it to our application and more specifically took a look at The Vue API. In some of the different properties that were going to define on our Vue instance.

So, let's now go back over to JS Fiddle.

We're going to start defining these different properties inside of the JavaScript side of our application.

OK, so here's JS Fiddle.

I'm going to give myself a lot of space inside this JavaScript panel right here. Just so I can very easily see everything.

Now, you recall that we already defined that method's property, that methods object.

So, I'm going to add a comment inside of here just to remind myself, use these functions to change data like so.

Then I’m going to immediately define that data property and the computed property as well, and I'll add in some comments on those to remind myself of how they are used as well.

So, I'm going to say data is going to be an object.

I'm going to add a comment in to say, initialize our list of ingredients and we don't call this our list of ingredients.

I'm just putting that on here on here to remind you that you kind of imagine this as being like a recipe of sorts and then finally, we'll also add in computed and I'll add in a comment that says, turn data into viewable values like so.

OK, now one thing that might be a little bit misleading here with methods, we have clearly defined a function right here. Inside of this computer object, we are also going to define functions.

So, these are going to be functions that are executed to turn our data into viewable values but with a data property right here, we are not defining functions. Instead, we are defining simple properties and property names and property values.

So, for example, if we want to call the text, whatever kind of like piece of state we refer to right here as our text input, we would initialize that ingredient of sorts by writing out Input Content is empty string like so.

So, with data we define simple key value pairs that have, say, strings or numbers or arrays or objects.

But with computed and methods, we define functions on both these objects and those functions are executed to either update our data or turn that data into a viewable value.

OK, so let's take a pause right here.

Now that we've added in some comments to further guide us, we'll continue in the next section and start implementing data and computed as well.

15-Updated Data Values

In the last section, we added in our data property, in our computed property. We also added in some comments to describe their purpose inside of this Vue instance. In the section, we're going to go back to our handle input change function inside in the methods object right here.

Remember, all the functions inside of methods are intended to somehow update our data and any time we update our data, it's going to cause our Vue instance to update the HTML that is displayed inside the browser over here.

At present, we're just taking the value that the user types in which is event target dot value and we're logging it out.

So that's not incredibly useful right now.

I think we should probably remove that control log and replace it with something that's going to update our data's input content property right here.

Remember, input content is supposed to reflect the current value of the text input right here.

We can kind of imagine this data property of input content as being like our list of ingredients.

And based on what the value of input content is inside of this data object, we will do different things to render different stuff out onto the page.

OK, so I'm going to take that console log.

I'm going to remove it and then to update the value of this input content data property right here, we will write out this dot input content equals event dot target value.

Now, one thing that's very important here that I want to point out is that we simply said this dot text input.

We did not say anything like this dot data dot input content.

We only use the word data when we are first initializing this data property.

But after that, to refer to any of the properties that are initialized inside of here, we simply say this dot and then the property name.

OK, now what do you want to be really crystal clear about?

Because I mentioned this in passing one or two videos ago, and I also just mentioned it in passing inside of this video as well, is that any time we update the value of one of our data properties, it causes our Vue instance to automatically render inside of the browser window.

At present, our template is not making use of this text input property at all, so you don't see any changes over here.

But as soon as we start updating or making use of text input inside of the actual template, you're going to see that any time we run this handle input change function, everything inside the window over here is going to automatically update as well, which is a very nice feature.

We don't have to forcibly say, please rerender my application or anything like that.

OK, so just to bring everything full circle now, I want to go back to the diagrams we are just looking at and kind of walk through the process of what we've done now with the data property and the handle input change function right here.

So, I'm going to go back to this diagram where we had said that we have some initial starting data of input content equal to empty string.

So that's how our application first starts off.

So, we start off with input content being empty string.

We then sit around and wait and then at some point in time, that method called handle input change is executed.

So, the user enters some text and inside there we update the value of input content.

So, let's imagine that maybe the user enters in text of Vue JS.

So, after they enter that text input in or we just enter that text, we're going to update the text input value.

So now data is instead Vue JS when that gets updated.

So, if input content is updated, we then rerender the template.

So that happens automatically.

And then when that template is updated, we look at we say whenever the template is rendered or updated, we calculate the image by using one of our computed functions which we've not yet defined, but we will in just a second.

So, I know that some of this stuff is awfully confusing and honestly, I really wish I could present it to you in a much more linear fashion.

But with a Vue world, you kind of have to kind of take all these different topics in all at the same time and understand them at the same time.

It's kind of challenging to to study them piece by piece, one at a time.

Nonetheless, we are going to talk about how this data stuff works and methods and computed, non-stop throughout this course.

So, if it seems confusing right now, don't sweat it because we're going to get a lot of practice with it.

So, we've made some good progress.

Let's continue the next section where we're going to start working on our computed property and hopefully start to bring everything full circle.

So quick break and we'll see you in just a minute.

16-Computed Properties

We've now got our data property being initialized here, and we've got some way to update our data over time whenever a user does something inside of our application.

It's now time to turn to the last step here, which is to take our data and turn it into an actual viewable value.

Now, to be entirely clear, our data is already in kind of like a viewable value, right?

Like it's a plain text string.

Obviously, we can show it inside of our template and have our users kind of view the output.

However, that's not what we want our users to view.

We don't care about showing them the text they just entered right here.

We want to show them an image generated by input.

So, we need to do some processing of our data before it gets displayed on the screen.

And any time that's the case, any time we want to do some calculation on a value before it gets showed on the screen, that's when we make use of a computed function.

So inside of this computed object right here, we're going to define a new function and I'm going to call it set image like so.

So, any time set image gets called, you and I are going to return something that's going to get the image to be available or viewable on the screen right here.

Now, as a quick reminder, we will use an API called robohash. I showed you before start codding on JS Fiddle.

So, we're going to write some code that's going to make use of this API right here to produce the actual image based on the text input that the user has provided to us.

So, inside the set Image function, we're going to make use of that robohash API.

So, to do so, I'll use backticks because I will right out a string include an html line.

And then we copy and paste this image htm over here.

Now, we'll pass in the input Content because that's kind of like the seed of sorts from which are image and gets calculated.

Remember that we can reference any of these data properties by simply saying this and then the property name.

So, I will delete YOUR\_TEXT and replace with string interpolation to pass in the input content. I will simply say this dot input content.

Now the very last thing we need to do is return this calculated, or this computed value from this function so I can make sure I get the return keyword inside there as well.

OK, so I know that at this point, we still don't really see any change inside of our output over here or out inside of the HTML that represents our application.

So, to get this identical function running and to displayed on the screen, we have to actually reference this function from within our template, which is inside of the HTML panel over here.

To call a computed function from within our template or to somehow get that information to appear on the screen, we can use a little bit of advanced Vue syntax.

So, underneath where we have output right here, I'm going to reference this computed set Image function right here, by writing out curly brace, curly brace, so notice that I've got like sets on both sides right here and then inside I'll say simply set Image like so.

All right, so you'll see a bunch of kind of like crazy strange text right here, and it might look like this is not what we want, but in fact, it's very close to what we really, truly do want.

What you see right here is an image tag.

So, in this case, whenever we make use of that set function right here, called to image, as you might imagine it returns an image element.

But when we try to show that inside of our template, we just see the actual raw HTML that represents the image tag. To get this to show up as a real piece of rendered image, we're going to use another Vue directive.

So rather than putting out the curly brace, curly brace, set image which simply prints out all that raw HTML, I'm going to instead replace this with another directive very similar to the one we did up here.

So right underneath the output I'm going to write v dash html equals then double quotes and set image and then I'll close off the div like so. Then eventually you see that robot image appear on the screen.

Now I know that that last step there where I said, OK, well we don't really want the image raw text to appear here.

We want to instead show the HTML.

This might seem a little bit confusing.

So, we're going to come back to how this works right here in just a second but before we do, let's try entering in some text here.

As you start to type stuff in, you'll notice that the image is changing every single time and we get some new image appearing on the screen.

So that looks pretty cool.

But I think that we've still got a lot to understand about how this application is truly working behind the scenes.

So, let's come back to the next section and we're going to do a very big review to describe exactly how

this application is working right now.

17-Review from Start to Finish

In the last section, we got our application working, but I think you'll agree with me that there's still some confusion over how everything works right now.

So, in this section, we're going to walk through a big timeline diagram that's going to give you a better idea of exactly what is occurring inside of our Vue application and it's going to really tie everything together.

So, let's get started.

At the very top we first start off by creating our Vue instance.

When that instance is created, the data property is evaluated.

So that data property is this right here.

Vue sees that we are providing an object that has a property named input content with a value of empty string because we are initializing this data property right here.

Vue is going to take that and it's going to assign it to the value of this accessible inside of our computed functions and our methods functions.

After that value is initialised, our template inside the DOM is then evaluated by Vue and eventually rendered onto the screen.

So that's when we see some starting content appear.

We then wait for some user to type into our text input.

The instant they do, our handle input change method is executed and inside of that, handle input change method, which is right here, we update the value of input content variable.

That's when some interesting stuff starts to happen.

So, it's when you start to update these data properties that have been assigned to this that you start to see some interesting behavior inside of your application.

So, in that value gets updated our template gets automatically rendered to the screen.

So, Vue says, hey, someone just updated that value.

I need to automatically rerender the template and update content that is visible on the screen. During that process Vue looks at our template and it sees that our template references a computed property when we put together that directive right underneath output right here.

So, we added in the directive of V HTML, which we're going to expand upon in just a moment and inside there we reference the set Image computed function.

So set image and right here is a reference to the computed function, set image right here.

So, because we add this reference right here, Vue says, OK, we need to go find that function and execute it and then whatever is returned from that function will be provided to this v HTML directive.

So, it seems that set Image property, it gets executed and that set image function returns some raw HTML as a string.

So that was just a raw string that had an image tag inside of it.

But as you and I very well know, we don't want to show raw HTML to the user. That's not very useful.

We want to show an actual image.

So, to get Vue to interpret that string as being actual HTML and not a plain old string, we used the V HTML directive.

So, this directive right here, V Dash HTML says we're going to give you a snippet of HTML.

It's going to be a string, but it's going to look like HTML.

You need to interpret that as HTML and render it as HTML as the output. Don't try to just print out the string.

What we did before where we use those curly braces, like that.

This is how we just print out a very simple string right here, and that's not what we wanted to do.

We wanted to somehow get our actual image to appear as some rendered HTML.

And you'll notice that this is not quite showing up properly right here for a comment out the div. Let's cut that out for just a second. And when I do, we'll instead see. OK, that's better.

There we go.

So, there's that actual output text right there.

So again, we don't want to see that HTML.

We want this to be interpreted as being actual HTML that needs to be rendered and that's why we use that directive right there.

Now, what do I want to mention very quickly is that we don't traditionally use this HTML directive right here very frequently unless you are working on an application where you need to make use of like some generated HTML.

So, it's much more frequently that we use other methods for displaying content on the screen.

It's just in this one very particular case that we're using this directive.

OK, so hopefully that's a little bit more explanation and makes what's happening inside of our application a little bit more clear.

There's still some side topics I want to address about this application.

So, let's take a quick pause right here and we'll continue in the next video.

18-Template Placement

In the last section, we reviewed our entire application and got a better idea of how it's working. In this section, I want to start going over a couple of quick odds and ends around on the application we just put together.

These odds and ends are going to be a couple of side topics that are going to help flesh out your knowledge of some of these very basic Vue topics and make some ideas a little bit clearer before we start working on our next application, which is going to be much more complex than this one.

So, the first odd and end of sorts I want to tell you about is the placement of our template.

Remember that many videos ago we had said that we have our Vue template in our Vue instance.

And I'd said that this was one possible way of structuring a Vue application.

I had said that our Vue template does not have to be created on the HTML side of our application and that we could instead declare that Vue template in the JavaScript side instead.

So, I want to show you a very quick example of that, and it's a very quick and easy refactor to do.

So back inside of my code, I'm going to take all the HTML that we had placed inside of our div with ID root.

I'm going to highlight all of it and I'm going to cut it all out.

So, notice that we still have the wrapping div inside of here that still exists. We have not deleted it.

I'm then going to take the JS panel and expand it and then down at the bottom I'm going to add a comma, I'll add on a new property called Template and I'm going to use a pair of back ticks here.

So back ticks are not normal quotes. So, this is not a single quote. It's not a double quote. It's a back tick.

It's a character on your keyboard or the key that's on your keyboard left after the one. Then inside of here I'm going to paste all that HTML that we just copied.

One other thing, I'm going to make sure that I also wrap this template or all the HTML inside of it with a div, just like we had previously.

The reason that we did not copy that div itself was that we need both divs inside of here.

So, when we make use of a template that is attached directly to a Vue instance, we provide a string that contains some amount of HTML.

Inside that string, we have to have exactly one root element.

So that's why we just added this div inside of here.

If I remove that div, so the opening tag and the closing tag, you'll see that we only see the H two right here.

That's the only thing visible. So that's why we wrap it with this root div.

We are expected to only return one element or one root element inside this template string.

So, you'll see very plainly that everything works the exact same way that it did before.

I can still type some text inside of here and I see some content appear on the screen.

So, like I said, different ways to structure your Vue application.

You can either create your template inside of your HTML or you can create the template and attach it directly to your Vue instance.

Now, in reality, the vast majority of applications that you're going to be working on is going to take this type of approach right here. Where we define our template very closely tied to our Vue instance rather than creating it over inside of our HTML.

The reason I showed you this approach at all is that all the documentation that you're going to see on The Vue official docs is going to take this type of approach. Where they show you one snippet of HTML, and it really is like HTML and then separate from that, they'll show you some code for The Vue instance.

So again, that's why I showed you this approach, even though traditionally we usually attach the template directly to the Vue instance.

OK, so that was the first little odd and end I wanted to tell you about.

Let's now continue in the next section and talk about one or two other quick topics.

19-Referencing Data in the Template

In the last section, we spoke about one quick odd and end. In this section we're going to cover another small topic that I just want you to be aware of.

So, in this one, I want to focus in on the fact that we used a computed function in putting our application together.

And what they want to be very clear about is that our template can show data that has been assigned to our data property.

We don't have to always use computed functions. In general, we only use a computed function any time that we want to somehow mess around with some piece of data before it gets shown on the screen.

So let me show you a good example of this. Back inside of our application. Let's say that we want to dramatically change how our app behaves.

Let's say that maybe instead of entering in some input and then getting back in image, we want to just echo back our text directly to the user.

So just so like, if I enter in Hi there’, the output should be ‘Hi there’. No change whatsoever.

If you want to print out some value that is assigned to data without making any change to it whatsoever, you don't have to use a computed function.

Instead, you can reference your property or your data name directly from within the template.

So, to do so, I'm going to scroll on down to where my template is now defined.

Inside of here, you'll notice that I still have that div with the directive of the V HTML.

So, I'm going to say I don't want to show the image anymore, so I'm going to delete that entirely. Now instead, I will directly reference the text content property and just try to print it out directly into the template.

To print out a direct property without trying to do any fancy directives or anything like that, we place those double curly braces and then inside there we just write out the property name that we want to display.

The name that we put inside of here can be either the name of one of our computed functions or the name of one of our data properties.

So, in this case, I just want to print out input Content directly and not do anything else fancy with it.

So, inside the double curly braces, I'll write out simply input content.

Now, if I start to type inside of my input again, you'll see that it gets directly mirrored right underneath it because I'm just taking one of my input value is in printing out directly inside the template.

One thing I want to mention inside of here very explicitly is that when we reference this input content property, we did not have to do anything like, this dot input content and we do not have to do this dot data input content or anything like that.

We simply say the name of the property, which is input content.

Now, one thing that that kind of like one issue that kind of brings up. You want to make sure that whenever you are defining computed functions right here, you don't accidentally duplicate the name of one of your data properties.

So, I would not want to make a computed function name something like input content, because then it would be really ambiguous as to whether inside my template, I'm referring to this data property right here or one of the computed function names.

OK, so I'm going to take that back to be set Image instead.

All right.

So hopefully that clears up a little bit of confusion there.

So, we don't have to use a computed property if we don't want to.

We can just reference the data name directly and that will be printed out straightforward inside of our template.

Now, I am going to revert this change right here because I do personally like the image that gets displayed.

So, I'm going to go back to the Div with the HTML directive.

OK, so it looks better.

Let's continue in the next section.

And there's just one last thing I want to mention about this application before we move on.

20-Expressions in Templates

In the last section, we spoke about one additional odd and end. In this video, we're going to close off with one very closely related topic.

So here it is, odd and end number three.

So, when we were talking the last section about using those curly braces to display some value inside of a template that made use of a technique called string interpolation.

So, whenever we use string interpolation, that means that we are kind of injecting a value directly into our template by using those curly braces any time that we use those curly braces, we are not only limited to referring to a simple data property or one of those computed properties.

If we want to, we can put in some tiny amount of JavaScript logic into those curly braces.

To be more precise, we can put exactly one JavaScript expression so we can write out, say, like a full if statement.

We can't write out a for loop or anything like that. Let's go back over to code editor and I'll show you a good example of this.

So, I'll go back over to code editor and I'm going to go back down to the very bottom of my template again, where we've got that set image being printed out inside the div by using the HTML directive.

I'm going to again delete that, and I'll replace it with my double curly braces and then I will reference inside of here about the input content piece of data again.

Now, if I type something out inside the input, just as before, I see some text appear right here. So right now, we're simply referencing input content but if we want to, we can add a very limited amount of JavaScript logic inside of these curly braces if we want to.

So, for example, we could do a little bit of string concatenation. So, I could say text input plus Vue JS and then when my app first renders text input is an empty string.

So, we add empty string to Vue JS and the output is simply Vue JS.

If I now start adding in some text, it takes Hello and adds Vue JS onto that.

We're not only limited to say adding text on like this, but we could also do something as complicated as, say, reversing the string.

So, a very simple way of reversing a string with JavaScript would be to split this into an array by every character, to then reverse that array and then to join the result back together like so.

So now if I enter some text in, all these transformations will be applied to that string.

It's going to split it into an array, reverse the array and then join the array back together into a string.

And as you can see up here, the output is completely reversed from what I entered.

So if I put in ABCDE, so on, it gets reversed into EDCBA.

Now, one thing I want to point out here is that while we can add some amount of logic directly into the string interpolation, the reason I showed you this idea of computed properties so early in the course is to direct you towards using these computed functions inside of your templates.

I only mention the fact that we can put some little bit of logic into these little interpolation locations just to be complete and just to tell you, hey, you can do this.

But in general, if you need to play around with any value or format it in any way before you print it out, then I highly recommend you instead make use of one of these computed functions instead.

So as a very quick example, we could add in another function here called something like reverse, then inside of here, we could return this dot input content, split, dot, reverse dot join.

So that's doing the same reversing operation as we were before but now inside the template, I'll reference this reverse function rather than trying to stuff all this reversing logic directly into the template itself.

So now back down inside the template, inside the curly braces, which we always use to interpolate values unless we wanted to use that voice HTML directive, which, like I said, is very rare.

So now inside the curly braces, we can say simply reverse like so and now if I enter some text, you'll see that it gets automatically reversed for us.

I highly recommend using those computed functions any time you need to play around with some data before it gets printed out, just because it keeps your templates very, very clean and very easy to read.

It also makes these computed functions much easier to understand because you can more easily document these functions up here by adding in comments, whereas adding in comments is a lot more laborious inside of your template.

OK, so there's just a little bit more on how we can kind of play around values with values inside of these little string interpolation locations.

So, I think that's pretty much it for stuff we need to talk about with this example.

I'm going to revert all my code here and go back to the actual image generator implementation.

So that's pretty much it for this app.

We're going to continue the next video by discussing the first big application we're going to create and of course.

Currently, if anything seems a little complicated or confusing. Honestly, don't sweat it just yet, because all the topics that we covered in this first application, we're going to cover again and again throughout the rest of this course, because everything we've done on this app, frankly, is stuff that you repeat all the time.

Yeah, initialize some data.

You handle user input, and then you somehow manipulate that data before printing it out inside of your application.

We're going to do this over and over again.

So let's take a quick break and we'll continue with our next application in the next section.

21-App Overview

In the last section, we wrapped up our robohash project and in this video, we're going to start talking about the next application that we're going to be working on inside this course.

This next application is a lot more ambitious and it's going have a lot more features and complexity to it.

I think that we're going to learn a lot while building it. So, let's get started.

We'll first take a look at the mockup of what we're going to built.

So, we're going to make a sort of stock media search engine.

This application is going to allow users to enter a search term up at the very top here.

So, this is going to be a little text input where a user is going to enter in some text.

They might search for a term like, say, flowers.

Once they submit that, well, then use the Pixabay API to search for the term they just entered.

Well, then get back a list of videos that match that search term.

We'll print that list of videos on beneath of the screen over here.

Then any time a user clicks on one of these videos, we'll show a big detail up here where the user can then play the video and they'll also see a quick description like how many likes it has and download maybe tags of the video on the right side.

This is going to test our knowledge of you, and it's going to make sure that we learn many different aspects of how working with Vue occurs inside of a normal application.

So now that we understand what we're going to make, let's continue in the next section where we're going to start talking about some of the big challenges and solutions that we might come up against inside this application.

22-App Challenges

In the last section, we took a look at a mockup that describes the next application that we're going to be working on. Now We will continue by talking about some of the big problems that we're going to have to address as we start working on this app and also, of course, talk about some possible solutions as well.

So, I think that there's probably four maybe more big issues that we're probably going to run into overtime.

So, we'll talk about each of these.

And I'm going to propose a different solution for each.

And the first big problem is perhaps one of the more obvious ones.

You and I need to somehow search for videos on Pixabay, given a search term. To do so, we're going to sign up for the free Pixabay API.

We can use this API to take some search term provided by the user and do a search on all the videos that are hosted on Pixabay.

This API will then return a list of videos that match that search term, along with thumbnail images, titles, descriptions, all that kind of good stuff.

The next big problem that we're going to have to address is the fact that we want to somehow show a collection of different elements on the screen.

In the last stop you and I put together, we had like that one simple text input and then we had the image and appear on the screen.

And so that was straightforward in nature.

But in the next step that we're going to build the Pixabay app, there's a lot more stuff going on, on the screen at any given time.

As we start to make more complex Vue applications, we start to isolate different parts of our app and create them in separate components inside of our Vue code.

Each of these components will be responsible for a different area of the screen.

So, you and I might make one component to implement the search input up here at the top.

We might make one component to show the list of videos over here on the right-hand side.

And then one component to show the big show media in the middle.

We'll talk a lot more about what components are and how we're going to use them to implement this application.

The next big challenge is the fact that we need a place to write our code and then somehow run it.

So so far in this course, we've only made use of JS Fiddle, which is only intended for very small projects and just kind of experimenting with little snippets of code.

So, to give us a more full featured boilerplate of sorts or a project that can be more easily expanded to handle large Vue projects, you and I are going to use a tool called Vue cli, and we're going to use this to generate a new project.

The project that gets generated is going to have a lot of preconfigured stuff already added to it, that's going to make working with new projects much easier and more straightforward than if we tried to do all this from scratch.

Now, the last big challenge that we're going to run up against is the fact that we have to somehow handle user input, like when a user enters in some search string into the search input at the top or when they click on a video with the attempt to play it.

In the last application we worked on, we did look at how we somehow handled these events or handle input from users. So, we're going to take all those same ideas around handling events with Vue directives, around adding methods and around making use of computed functions to get all this stuff to work properly and handle user interaction with our application.

OK, so that's pretty much some of the big four issues that I can foresee in the app that we're going to be working on.

So, let's continue the next section where we're going to start by generating a new project by making use of Vue cli.

So quick break and we'll take care of that in the next video.

23-Generating a New Project

In the last section, we took a look at a mockup that describes the next application that we're going to be working on. Now, we're going to make use of a tool called Vue cli to generate a new project boilerplate.

We're using this tool because setting up your own Vue project from scratch takes a little bit of time and it's not the best thing to get started with when you're first trying to start learning Vue.

So, we're going to install the VUE CLI at our terminal and then we'll use it to generate a new project.

So, let's do that right now.

I'm going to change on over to my terminal and then I will install of VUE CLI by running the command NPM install dash G @ Vue slash cli like so and then I'll go ahead and run that command.

Now while that's being installed, I want to see the documentation for Vue cli very quickly.

So, here's the documentation at GitHub.

It's that GitHub dot com slash Vue js dash Vue dash cli.

There's not a tremendous amount of documentation on here because VUE CLI is still in active development right now.

You can find a link to the full documentation link in read me section and once you're over here, you'll see some information about how to do further configuration of the project.

OK, so feel free to take a look at the documentation there if you wish.

Otherwise, let's go back over to our terminal where you'll notice that the installation is now complete.

We can now use this tool to generate a new project.

So, I'm going to make sure that I'm inside of some directory where I feel like I can have kind of a workspace folder of sorts.

Make sure that you are in a similar folder as well.

Then inside of this directory, I'm going to run the command Vue create and then we're going to put down the name of our project, which in this case, I'm going to use a name of video-portal.

So that's what it's really doing here.

We're using this application to browse Pixabay stock videos and then run that command, and that's going to automatically generate the new project and start to install dependencies.

When you run the command, you might see a couple of quick questions like this right here.

And if you do just select any of the default options that are represented.

So, I'm just going to say give me the Vue 2 default with babel and e s lint.

OK, and there goes the installation.

Let's take a quick pause right here, and when we come back, we'll talk about what we are getting when we install this default project here.

You know, what's this really doing for us?

So, let's talk about that in the next video.

24-Why Use Vue CLI?

In the last section, we made use of VUE CLI to generate a new project. It looks like my installation is now complete and I'm given the directions to run this command, to change into that newly created directory that contains our project and then to run the command and npm run serve to start our project up.

Before we do that, I won't have a quick discussion on exactly why we are making use of this Vue cli project generator.

Like why do we need all this infrastructure and all this preconfigured stuff anyways, let's take a look at a diagram to get a better sense of why.

OK, so this is an illustration of how you're just running code a little bit ago by making use of JS Fiddle.

When we used JS Fiddle, we automatically loaded up the Vue js file by adding it as a dependency to the project.

We then also wrote a little bit of code directly on JS Fiddle and these two together formed a usable application.

The key thing there to keep in mind is that all the code that you and I wrote, like the JavaScript code in the HTML, we're all inside of essentially a single file authored directly on JS Fiddle.

But when we start working on real projects, we don't really get to enjoy the fact of only having to create one single file. With real Vue projects, we much more frequently make a collection of files like many different files that form one single application.

And that's where this added complexity starts to come in.

That makes us want to use a tool like Vue cli.

So, the Vue cli, we get the ability to instead make many different files, each of which might have one individual or distinct purpose inside of our application.

So, we might have one file that has some code for fetching videos and then maybe another one that's in charge of that search input at the top of our application, another one for playing a video and then one more for maybe showing a list of videos to the user.

So, all these different files are going to come together to form our one usable application.

Here's the thing about JavaScript, though.

With JavaScript out of the box, with the JavaScript language, we don't really have a great way to take all these different files and combine them down to one single file that we can then send to the user's browser and have our application start up and be executed.

So, take all these different files and combine them together, we make use of a tool called Webpack. You may have heard of Webpack before.

Essentially, its purpose is to look at a big collection of files and possibly dependencies as well like Vue js, and combine them all down to one single file, which can then be sent down to your user's browser when they try to make use of your application.

So, this Webpack tool is one of the big reasons that we're making use of Vue cli. Webpack is somewhat notorious for being a little difficult to set up out of the box.

And so, by making use of VUE CLI, it automatically sets up Webpack for us and chooses a lot of very sensible configuration defaults. Along with VUE CLI, we also get access to babel pre-configured for us right out of the box.

You may have heard of Babel before as well.

Babel allows us to write 20, 15, 16, 17 and so on, code that can be safely executed inside of a user's browser, even if their browser does not have support for a particular dialect of JavaScript that we might want to use inside of the code that you and I write.

So just like Webpack, babel is also included automatically inside of VUE CLI, and it has a lot of very sensible configuration defaults already set up for us.

So that's why we are making use of VUE CLI in general.

It does all of a set up for us and we don't have to waste any time kind of laboring over documentation and figuring out how to set that stuff up.

OK, so now we've got a better idea of why we're making use of VUE CLI at all.

Let's continue in the next section where we're going to start up our application and get a better sense of all the different files and folders that were created for us automatically when we just generated the project.

So quick break and we'll see you in a minute.

25-Project Walkthrough

In the last section, we made use of VUE CLI to generate a new project. It looks like my installation is now complete and I'm given the directions to run this command, to change into that newly created directory that contains our project and then to run the command and npm run serve to start our project up.

So, I'm going to change into video portal and then I will start up the project by running the command NPM run serve.

Throughout the rest of this course, we're going to be using this command to start our project up.

So, if you ever stop the course, you ever take a pause or go to sleep. When you want to start the project back, you can just run npm run serve again.

Now, when we run that command, you'll see some information about starting up the development server.

So, when we run NPM Run Serve, it starts up a local server that starts up Babel and Webpack takes all of our project files and bundles them together into one single JavaScript file where it can then be served up into the browser.

You'll notice that you might get an automatically open browser window and it's automatically open window is pointing at localhost eighty eighty, which is where our project is hosted.

So now I could say refresh the page and you'll see the exact same app appear on the screen.

You'll also notice that there's a little bit of default content already visible on the screen here.

So, this is a little bit of Vue code that has generated the content you see on the screen, and this is code that is given for us automatically when we generate our new project.

Let's now open our code editor inside the project directory and look at some of the different files and folders that were generated for us.

So back over at my terminal, I'm going to start my code editor.

I'm going to make use of the code editor, vs code, throughout this course but you're free to use any editor you want.

And I would say that every editor can pretty much handle Vue as well as any other. So don’t feel like you have to use any one particular editor to have a good time with Vue. There is also a special code editor build for Vue js called HBuilder. Maybe you can setup it up your pc and work with it during the course. Like I said, It is totally up to you.

OK, so now on the left hand of my screen, you'll see some of the different files and folders that were created for us automatically when we generated our project. Inside the node modules directory is a list of all the different dependencies that our project depends upon. Underneath that, you'll find the public directory.

Inside this folder is a very important file. It's our index dot HTML file.

Any time someone tries to visit our application by coming to our server in their web browser, even as we just did right here, this index dot HTML file is always going to be loaded up automatically.

Inside this file you might notice a div with id app, just like we wrote a little bit ago inside of JS Fiddle.

This div right here represents the root location of where our app is going to be booted up.

Then inside the src directory, we'll find a bunch of different files and folders that are related to the actual implementation of our Vue app.

We're going to be spending most of our time inside this src directory.

I want to talk about some of the different files and folders that are located inside of here, because you'll very quickly notice that there's a very big difference between some of the code that you and I were writing earlier on, JS Fiddle and some of the code that you're going to find inside these files.

Let's first start off by looking at the main dot js file.

Inside of here, you'll find some very plain JavaScript code, and towards the bottom, you'll find a function call to new Vue, which we use a little bit ago, to create a new Vue instance over on JS Fiddle instance.

Then you'll notice that there's also this render option in here and also a dollar sign mount function call as well.

We'll just ignore those two little pieces of code for right now.

They will come back to those very shortly and talk about what their purposes are.

OK, so now here's the weird part.

Here's where things start to get really interesting.

You'll notice that also inside the src directory is an app dot Vue file.

Notice the extension there, its dot Vue, rather than being like dot JS or anything else like that.

Let's open that file and see what's inside of it.

At the top of this file, you'll notice that there's what looks like a little bit of HTML.

All this HTML is nested inside of a template tag and then below all that HTML, you'll find what looks like another piece of HTML, a script tag, and inside of there is some JavaScript code.

So, I don't know about you, but this looks like one weird little file.

So, let's talk about what's going on here in the next section.

A quick break and we'll cover this Vue file and it's kind of strange syntax.

See you in just a minute.

26-Vue Files

In the last section, we started looking at some the different files and folders that were generated first automatically when we made our new project. We opened the app dot Vue file. Inside of there, we saw some very interesting looking syntax.

So, at the top we see a template tag and below that we see a script tag. So, let's talk about what's going on here.

First off, quick reminder of what we were doing previously over Inside of JS Fiddle.

So back over here, we had said that this was one possible way of structuring of Vue app.

We might write out our Vue template inside of some HTML, like an actual HTML document and then we also might create a Vue instance over in some JavaScript code.

We then later said that we could optionally also create this Vue template inside of our JavaScript code as well and we saw an example of that over in JS Fiddle when we use those backtick characters to insert our template directly attached to our Vue instance.

So again, these are both two possible ways of structuring a Vue application but what you're seeing over here inside of this app dot Vue file is yet another way of structuring your Vue code.

So, this takes use of a kind of paradigm called Vue files.

The idea behind a Vue file is that you'll have exactly one file that is responsible for creating a single component inside of your application.

A single component is created as a reusable piece of code that can be used all over your application many times. Inside of this single Vue file, you'll find not only the template that your Vue file or your Vue component is going to use, but you'll also find all the JavaScript code related to it as well.

So, in other words, inside of one single file, we get access to all our HTML for this Vue component.

We get all the JavaScript for it, we'll also locate all the CSS for this component as well.

So, one file that contains all the code related to one component or one piece of our application. As we start to implement our Pixabay video portal application, We are going to end up with a couple of different Vue files, each of which are responsible for implementing one distinct part of this application.

So, we might make one Vue file that is responsible for the search input up here at the top.

We might make another single Vue file that is responsible for this list and then a single Vue file that is responsible for the big show media file here in the middle that's going to eventually play a video.

So, again, the idea is that with one single file, we have all the HTML JavaScript and CSS all directly placed inside there.

Now, you might think that this violates one of the core principles of writing Web applications.

The idea of separate concerns. The idea of separate concerns means that we try to not mix and match all our HTML, JavaScript, and CSS together because that could possibly make a real big mess of our application.

But one thing I want to point out here is that even when we start making use of Vue files, even though we've got our template and script and your CSS right here inside the style tag, even though this is all inside a single file. Well, technically it is still somewhat separated because inside this file, we've got one area responsible for HTML, one area responsible for our JavaScript and one area responsible for our CSS as well.

So, yes, they are all located inside of a single file, there's still at least somewhat isolated inside this file.

One of the big benefits to making use of these Vue files is that if you ever need to change your application in some fashion, it'll be a lot easier to find all the related code to one area of your app.

For example, if we wanted to make a change to the way that the show media file over here.

We know that we could always open the Vue file that contains some implementation and inside there will find the HTML, JavaScript, and CSS.

And that means that we don't have to go hunting around different directories inside of our application to find all the different aspects of code that are related to the show media component.

So, it might take a little bit of time to get used to this Vue file syntax, but over time, I'm very confident that you'll probably come to enjoy it.

OK, so we've spoken a little bit about the purpose of The Vue file at this point, but we haven't really spoken about how this strange syntax is taken and used to build an actual application.

So, let's take a quick pause.

In the next video, we're going to examine some of the behind-the-scenes stuff that occurs to eventually get this code to run inside of your browser.

27-Behind the Scenes of Vue Files

In the last section, we started talking about Vue files. Inside of a single Vue file, we will locate all the code related to one distinct portion of our application.

We're going to use Vue files in place of making a separate Vue instance that are already created in the DOM.

So, again, I showed you this methodology right here using JS Fiddle because this is how a lot of the documentation is put together.

But the vast majority of professional Vue projects that you'll probably work on in industry are usually going to make use of these more complicated Vue files. Just because they are easier to expand and scale into large applications.

Rather than placing all your Vue templates directly into some HTML.

In the section I want to answer a question that you might have right now, and I question that I'm assuming that you might be thinking about is ‘How is our browser going to execute or somehow understand this code right here?’

So, in fact, what you see on the screen right here is not what gets sent down to your user's browser.

Instead, babel in Webpack, work together to somehow convert all the contents of this file into some very plain JavaScript code that your browser can safely run.

So, in this video, I'm going to give you a quick demonstration of what this file turns into when it gets sent down to your browser.

Now, for this demonstration, I really recommend that you just watch what I'm doing because I'm going to click through a couple of steps very quickly, so you don't have to watch me very laboriously click around a whole bunch.

So, let's get to it.

I'm going to show you what this file ends up like when it gets shipped down to your browser.

I'm going to first go back over to our application and I'm going to open the request log.

Then I'm going to go back over to my code editor and I'm going to make one very small change to this file and then save it.

Once I do that, I'll go back over to the request log.

So, here's a whole bunch of JavaScript code that represents that change that we just made.

I'm going to copy it back over to my code editor.

I'm going to find and replace newline characters in here very quickly, and then I'm just going to delete a tiny little bit of code.

All right, let's do that delete. Here we go.

So, I'm going to delete that and I'm going to delete all that as well.

OK, so like I said, I just very quickly want it to go through that change just so we can very quickly see what that file gets turned into.

So, this render function right here is what our App Vue file was converted into before it got sent down to our browser.

This right here is the actual code that is running inside your browser that represents that Vue file.

Now, before we walk through the code inside of here too deeply, let's go back over to the App dot Vue file very quickly.

Inside the template up here, you'll notice that we are creating a div with an idea of app.

And then inside there we've got an image with a src logo dot png.

And then we've got this hello world tag thing right here that has a message of welcome to Your Vue JS app.

Now, with that template, now that we've got a better understanding of it, let's go back over to the file that I just created.

So inside of here, you'll notice a couple of strange variable declarations at the top. We’ll ignore those for just a second. Instead, I want you to look at the function call right here to underscore C.

Inside this, we have a first argument of div.

So, this word right here, div represents this top level div tag right here inside of our template.

Then the second argument is an object that says that div should have an ID of app and then the third argument is a list of other elements that are contained within that div.

So, the first element that exists inside that div is an image tag that has a source of logo dot png and then the second element is that ‘hello world’ tag that has a message property of ‘Welcome to your Vue JS App’.

So even though the syntax still might look very strange and might not look like any Vue code that we've looked at or written so far, I hope that you can at least get the sense that, yes, everything inside this file is somehow converted into other structure before it gets sent down to our browser.

So inside of our browser, no, we're not running this code inside of here.

It all gets translated into this other form and then execute it.

So really, the idea of making use of these Vue files that has the template tag and the script tag and the style tag down here is just to make your life and my life as developers a little bit easier and more straightforward.

That's the only purpose of it. It is just to help us organize our code inside these Vue files.

OK, so now that we've got a better idea of what's happening inside of a Vue file, let's continue in the next section and do a little bit more of a walkthrough through some of the code that we have inside of here.

So quick break and I'll see you in just a minute.

28-Vue Components vs Vue Instances

In the last section, we started talking about Vue files. Inside of a single Vue file, we will locate all the code related to one distinct portion of our application.

Now, we're going to move on to working on our video portal application in just a second but before we do, there's one last important topic I want to share with you.

So, in the last couple of videos, I've been using the term vue component a couple of different times.

Earlier on in the course back when we were working inside of JS Fiddle, I was using the term Vue instance.

So, I want to make a distinction very quickly between Vue components and Vue instances. All right.

So, you can think of a Vue component as like a blueprint or a set of rules on how to create something that can be inserted into the DOM, in our browser that the user can interact with.

So, whenever we make a Vue file, we are going to define exactly one component inside of it.

And that component is going to have this set of rules that tells Vue how to display some content on the screen and how a user can interact with it.

So again, think of a Vue component as being like a blueprint of sorts.

On the other hand, is a Vue instance, which is what we were making back inside of JS Fiddle. Vue instance is really an instance of a Vue component.

It represents something that has been inserted into the DOM and is something that a user can interact with.

So, I know that the relation between these two things is very tenuous.

If you've got an understanding or a background, an object-oriented programming at all, you can think of a Vue component as being like a class and a Vue instance as being like an instance of that class.

That's the real relationship that's going on here.

So, like I said, we are always going to create one Vue file for every Vue component.

So, one of the very critical steps that we're going to undergo in every Vue application we put together is to plan out the different components that we might want to make.

In general, we like to look at mockups of the applications that are going to make and do some quick brainstorming on how we might assemble a set of different components to make up that application.

So, with that in mind, I took the liberty of taking our mockup for the video portal application and thinking about how I might divide this up into a set of different components.

You'll notice I've added some color boxes on this mockup and on every box of attached a label.

The label that you see here is the name of a component that we're going to make to implement the video portal application.

So, you and I are going to make one Vue file called Search input.

This is going to have a single Vue component that is going to contain a text input.

And any time a user types inside there, we're going to somehow trigger some searching operation on the Pixabay API.

We will also make a single Vue file that houses a component called videos.

This will this videos component will know how to take a list of videos and render them out onto the screen.

In term will also have a item, which will be a single component that represents a single video.

So, the videos component will contain many items inside of it because a item represents one video and the videos represents the entire list. We’ll also have a Vue file for the show media, which is responsible for showing details about one single video.

And then all these components will be assembled underneath one component that we'll call the app component.

This app component is kind of like the central brains of our entire application. It represents the single point that kind of organizes these different components and controls how data flows between them.

It's extremely common to always have a single component called App in a Vue application. It's always going to serve that same kind of purpose of serving like the brains of your app.

It's the last thing I want to show you with all these component names in mind is a quick diagram of how they are all related.

OK, so this is the overall structure that you and I are going to eventually head towards as we start to build our video portal application.

At the very top, we've got our main JS file, which is sort of responsible for kind of like booting up our entire application and rendering onto the screen.

That may not just file is then going to show our app component and internally the app component will show the show media, the search input in the videos.

And then in turn, the videos component will show a couple of copies of the item component.

So that's how all these different pieces are going to be wired together.

Again, we're going to make one separate Vue file for each of these different components.

And each of those Vue files are going to contain all the HTML, CSS, and JavaScript related to that single component.

OK, so that's pretty much it for getting a better idea.

Some of the high-level architecture.

If any of this kind of component stuff still seems a little bit strange, this is another one of those topics where we're going to be doing this breakdown of talking about components throughout the course.

This is another topic that we're just going to come back to again and again.

We'll get a lot of practice and understanding how components are really working.

So, with that in mind, let's continue in the next section and we're going to start working on our app.

So, we'll see you in just a minute.

29-Starting from Scratch

At this point, we've spoken a lot about all the code that's placed inside the src directory, and it's now time to actually start working on our video portal application.

But I don't really want to use any of the code that is already placed inside of here.

You see me, try to start from scratch where possible.

I don't really like having to go through the Webpack or Babel setup, which is why we're using Vue cli, but most of the code inside the src directory that we were given is super helpful for us.

So, the first thing we're going to do in working on our video portal application is to delete everything inside the src directory and we're going to start over from scratch.

So, you get a really good idea of exactly how we build out of your application.

So, to get started, I'm going to highlight the src directory and I'm going to delete all of them, and then I'm going to immediately recreate src directory by making a new folder called src.

Then inside of the Src directory, we're going to make that main JS file again.

So inside of Src, I'll make main dot js.

Now, I got encourage you right now, please make sure that you name this file main dot. js. M is lowercase as well.

You see, whenever our application first boots up the VEU CLI and Webpack and all the tooling that exists inside this project is going to look inside the Src directory for a file called main do js.

This name right here is a special name. It's the entry point of our application.

Now, inside this file, we are going to do the same exact steps that we're already inside the main JS file.

But this time around we'll talk about what every line of code is doing, so you'll get a better idea of how all that stuff was structured.

So, inside the main dot js file, we're going to first start off by importing the Vue JS Library.

To do so at the top will write out import Vue from Vue. As a quick reminder of how import statements work.

Webpack is going to interpret this line of code right here when it processes all of our code. It's going to see that we're trying to import something called Vue.

Now, we don't have any files inside of our SRC directory right now called Vue. So instead, Webpack is going to automatically look into the node modules directory, and you'll recall that that is where all of our different dependencies for our project are placed.

If you scroll down to the very bottom of this list, you'll find a file called Vue.

So, this right here is the actual Vue JS library.

When you and I write out import Vue from Vue, Webpack is going to look into that folder and pull in some amount of JavaScript code to support our project.

All right.

Now, the next thing we're going to do is to create a new file called the App dot Vue File.

And this is going to function very similarly to the App dot vue file that was here just a moment ago.

Inside that file, you and I are going to make a single Vue component, and this component is going to serve as the kind of brains of our application.

It's going to be responsible for coordinating all the other components and making sure that the correct content is visible on the screen at any given time.

Once we create this app dot Vue file and create a component inside of it, well, then import it into the main dot js file and make sure that we show that on the screen.

So, inside the source Src directory will now create the app Vue file.

As a quick reminder.

It's very common inside of any Vue application to always have a root component called App dot Vue.

So, between the two files right here, I would expect you to see these two files in most projects that you're going to work on.

OK, so inside of here, we're going to put together our first Vue file.

Every single file that we put together is going to look very similar.

At the very top, we're going to place that template tag and this thing is going to contain all the HTML or really the template for the component that we are creating.

Inside the template tag itself right here.

We are only allowed to have one root HTML element.

In other words, we cannot do something like div and another div.

These are two sibling divs. So, in this case we would have two root elements and that is not allowed.

We have to always have exactly one root element, so I cannot do div div like so, but I could do a div with two divs inside of it if I wanted to.

We don't really want to, but just mean to say, make sure you've always got one root element.

I think that at this point we just want to get some content on the screen.

So, inside this div I'm just going to say ‘App, then underneath the template tag, we're going to add the script tag.

So, this is going to contain all the JavaScript codes that defines this component and tells you how it should behave any time user interacts with it in some fashion.

Inside of the script tag, we're going to first write out export default and then a set of curly braces.

Now we haven't really spoken a lot about this export default syntax so far, even though we saw it in the previous implementation of The Vue file.

So let's take a quick pause right here.

We're going to come back the next video.

We'll talk about what the purpose of this export defaulting is.

So quick break and I'll see you in just a minute.

30-The App Component

In the last section, we started working on our main app component. Remember, this is going to serve as a sort of brains of our application and it's going to coordinate all the other components that we create inside of our app.

We left off by working on the script tag right here where we wrote out export default and then placed a set of curly braces.

So, let's talk about what the purpose of export default in the curly braces are right here.

First, I want to give you a quick reminder back to some of the JS Fiddle code that we were working on earlier.

So, I've still got the image generetor application here from code up on my screen.

You'll recall that inside of the JavaScript that we added for this project, we had a couple of different properties that we were providing to this Vue instance.

So, we had things like data, computed methods.

And at some point, we also added the template directly into this thing as well.

So, this object that we passed to New Vue had some number of properties that customized the Vue instance.

The same exact thing is true of this object right here.

So, this object that you see is going to contain a couple of different properties that customize this Vue component in how it behaves.

So, for example, we might eventually have a data property on here.

We might eventually have some computed functions and we might eventually have some different methods tied to it as well.

Right now, there's only one property that we're going to assign inside of this object.

And this property is one that we've not discussed previously when we were working back inside of JS Fiddle.

So, the one property that we're going to assign in here for right now is called simply name.

And we're going to name this app like so.

The name property right here is not strictly required for you to work properly.

We put in the name properly or the name property just to make our lives as developers a little bit easier.

You see, there's a couple of debugging tools out there that you can download for free to help you debug the Vue applications inside the browser.

Many of these debug tools that we use with Vue, we'll look for different components that have specific names tied to them.

So simply by providing this name property right here, it's going to make those debugging tools work a little bit better by allowing you, the developer, to more easily identify different components that are in use in your application.

OK, so that's really all we need for right now to get this component started.

We don't have to provide anything like data. We don't have to provide methods or computed just to get some content on the screen.

We're only going to start to add those extra properties once our component needs their behavior.

So, like, once our component needs data or needs methods or anything else like that.

Now, one other thing I'll mention here very quickly.

You'll recall that back on our Vue instance that we created over an here, we provided this E-L argument.

The E-L argument was short for element and it told you where we wanted to insert this instance. It told where our Vue template was in the DOM.

So, in the case of a Vue component that we're putting together right here, Vue is going to automatically understand that our template is created in the same file.

So, you and I do not have to specify any property to tell this component where template is or where should render itself.

So, no need for the L inside of here.

OK, so that's pretty much it for the app component for right now.

We're going to come back to this thing very shortly and adding a bunch more code to further develop our application.

But for right now, we're going to take a quick pause and then go back over to the main dot js file and try to get that app component to show up on the screen.

So quick break and I'll see you in just a minute.

31-Another Way of Specifying Content

In the last section, we were finished working on App dot Vue file, we're not going to go back over to our main dot js file and we're going to use that app component to get it to display on the screen when we load up our app inside the browser.

So back inside of main dot js, we're going to do a couple of different steps very quickly here.

The first thing we're going to do is to import that app component into this file.

So, we're going to import App dot Vue into main dot js.

We can do so by writing out import App from dot slash app like.

Then underneath that you and I are going to create a new Vue instance and we are going to attempt to show this app component.

So, you and I already know how to show a or how to create a Vue instance.

We write out new Vue and then pass in an object with some number of configuration options.

Now, the first option that we're going to put in here is going to be one that's going to look a little bit strange.

So, we're going to say render and then this is going to be a function. It's going to be called with another function called create element, then inside of the function, we're going to return, create elements and we'll pass the app into that thing like so.

So, at this point, you and I have seen more than one way to specify the template for a given Vue instance.

We've seen how we can still get the diagram over here, so we've seen how we can stick our template into our HTML.

We've also seen how we can attach the template directly to our Vue instance.

So, what you're seeing right here is another way.

Yes, yet another way to tell a Vue instance how to show some content on the screen.

So, this is yet another way to tell this Vue instance right here that we want to show something on the screen. So, we can define a render function that gets called with another function called create element.

So, we call discrete element thing and we are passing the component that you and I just made and then we return the result to that.

So, you can kind of imagine that this takes the app component, turns it into a Vue instance, which can then be turned into HTML and we take that HTML and we put it on the screen of our application.

So that's what's going on here with this function.

Now, one quick refactor that we're going to do here to make this look much more similar to how you'll see this in real applications.

Usually, we do not write out the entire function name of elements here.

Instead, we usually abbreviate the create element function name as simply H.

Don't ask me why we call it H.

That's just what we call it.

I honestly don't know why we came up with that convention.

Then the other refactor that you'll see very commonly around this render function is to not use the entire function keyword, but instead to use an arrow function like an ES6 arrow function.

So, what you see right here is completely equivalent to writing out simply h, h app like so.

So, what we had just a moment ago and this right here, one hundred percent equivalent.

So now any time this file is executed, we will try to create a new Vue instance.

When that Vue instance is created, the render function will be called, it will be called with this create element thing.

We then pass in our app component to create element and the entire result is return from the render function.

And so our app is created or app component is created and then the result of that is stuck into the DOM.

OK, so let's take a quick pause right here.

We're going to come back in the next section and make sure that this entire file right here actually gets some content on the screen of our browser.

So quick break and I'll see you in just a minute.

32-Multiple Ways of DOM Attachment

In the last section, we started working on our main dot js file, so we've now created a Vue instance, and we've told this instance what we wanted to do to get some content on the screen.

So when we eventually render this thing to our actual DOM, this render function will be called And we are telling that render function that we want to display an instance of our app component.

And the very last thing we have to do inside of here is to make sure that we actually try to take this instance of app that is created and stick it into the DOM.

You'll recall that back inside of this image generator app, we had created that L property right here and we discussed this property many times.

The L property not only tells you where it can find its template, but it also tells you where it should stick the rendered HTML into inside of our HTML structure.

And so hash root right here means go and find some element inside of our DOM with an ID of root.

The same thing has to be provided to our Vue instance over here as well.

So, if you ran this code right now, we would create our application, but it would not show up on the screen because we've not told Vue where to put this HTML.

You'll recall that inside the public directory we've got that index DOT HTML file and inside there is a div with ID app.

So, when we generated Vue cli, we already got a place right out of the box created for us, for us to render our application to.

So, to get our main dot JS files Vue instance right here to actually show up inside the browser, we need to provide a L property right here and we're going to specify that div with ID of app or root whatever you want but remember, the root div in HTML must have an id as same as L property. These two things must be same.

So just like before I'll say hash root.

So now one very quick thing I want to mention here.

Back when we first generated our Vue cli project, like just a moment ago, you might recall that when we looked at the main dot js file, we did not have an object or a property in here called L.

So, when we first generate this project main dot that just looked a little bit different and previously like how it came to us right on the box, it had a different way of specifying the element that we wanted to render our application too.

Make the new Vue instance right here and then it called a function on that instance that was created, and they had something like Dollar Sign, Mount hash root just like that.

So, this line of code right here or this chained on function also tells you where it should render itself to.

Now, let me be very clear.

If you haven't really noticed yet in the Vue JS World, there's like two or three or four or five ways of doing any given operation.

We've already seen many ways in which we can define templates that are tied to our application and this entire idea of mounting or somehow rendering our components in our Vue, instances to the DOM is no different.

So, dollar sign mounts and then passing pound root right here is completely identical to writing out L app like.

So, these two things totally identical. It's just two different ways of doing the exact same operation.

And like I said, this is a pattern that you're going to notice in the Vue world repeatedly.

There's just more than one way to do just about everything.

So, is there a good reason to use this approach or this approach?

Well, not really, technically, just so you know, a little bit of behind-the-scenes stuff. If we did not provide the property right there and we also did not provide a specify dollar sign dot mount function.

Well, then if we just ran this code right here, it would create an instance of our application, but it would not actually render it to the DOM.

We could later render it to the DOM if we wanted to.

But technically you can use the pound mountain right here to do some other interesting stuff.

But it's not super relevant for what we're talking about right now.

So, I think just for the sake of convention and sticking with what Vue cli does by default, we'll stick with this mount option rather than using E-L.

We're going to call Dollar Sign Mount and then we'll tell us where we want it to be rendered to in our DOM.

OK, so that's pretty much it to get some content on the screen.

Now, last thing we have to do is go back over to the browser and just test our application and make sure that we can see this text of ‘App’ appear on the screen.

So, I'm going to go back over to my browser.

I'm going to go back to localhost eighty eighty.

I'll do a forceable refresh of the application and you'll see the text of ‘App’ appear on the screen, just as you might expect.

OK, so that's great.

We figured out how to just get very simple content on the screen.

So, let's continue the next section and we're going to start really diving into our application and building out some more of the guts and different components that we had spoken about earlier, like, say, the search input.

And the videos and so on, so let's take a quick break right here.

We'll continue in the next section.

33-Making the SearchBar

Now, we've spoken a lot about all the code that's in the src directory, and it's now time to start working on our video portal application.

We're not going to create search input component.

Remember, the idea here is that this search input component is going to show an input. Any time user types in there will then trigger a search on the Pixabay API to attempt to go find some list of videos related to that search term.

So, let's flip back over to our code editor, where we're going to create a new Vue file to contain the search input component but first I am going to App dot Vue file and delete everything inside template, script and style tag except export default name App. Then I will delete this Helloworld dot vue file.

Inside the template tag itself right here.

We are only allowed to have one root HTML element.

In other words, we cannot do something like div and another div.

These are two sibling divs. So, in this case we would have two root elements and that is not allowed.

We have to always have exactly one root element, so I cannot do div and div like so, but I could do a div with two divs inside of it if I wanted to.

We don't really want to, but just mean to say, make sure you've always got one root element.

I think that at this point we just want to get some content on the screen.

So, inside this div I'm just going to say, ‘Video Portal App’. I am going to save that and go back over to my browser localhost eighty eighty.

I'll refresh the page and you'll see the text of ‘Video Portal App’ appear on the screen, just as you might expect.

OK, so that's great. App is working perfectly.

Then, I'll find the SRC directory. Usually, we create a components directory inside of here and inside of that directory, we'll locate all the different components we have.

So, let's do that here as well. Inside of SRC, I'll make a new folder called Components and then inside there I'm going to make a new file called Search input dot Vue.

Like so inside the search input Vue file.

We're going to put down some different boilerplate tags inside of here that are going to become very familiar over time.

So, we'll first begin by placing a template tag and then underneath that we'll put a script tag.

And then finally, we'll also put a style tag which will eventually contain some CSS that is solely related to this one component and no other component inside of our application.

Back up inside the templates will immediately create an input element and we'll wrap it with a div.

So, I'm going to place a div and then inside there we'll put an input very similar to the one we had worked with previously back on JS Fiddle.

Now you might be a little bit curious why we are wrapping this with a div.

Honestly, that's just for styling.

By default, a div has CSS display property of block, which will make sure that this input element appears in the browser on its own individual line.

So, the div here is just kind of for layout purposes, not necessarily because this component requires it from any functionality standpoint.

Next, we're going to start working inside the script tag. Inside the script tag, we'll put down export default and then provide an object that's going to specify a couple of different options that we're going to use.

OK, so this looks good.

Let's take a quick break here and then define a couple of different parameters inside this object.

See you in the next one.

Nesting Components

We just put together our search input file and now ready to start adding a couple of options inside of here.

Let's first get started by adding one parameter. The name parameter, which we spoke about a couple of videos ago.

So, I'm going to provide a name of simply search input.

So now, before we go any further, I think that we should try to get this search input component right here to be displayed inside of our application.

So, if you go back over to the browser right now and refresh the page, you'll notice that the search input doesn't appear at all, which kind of makes sense if you want to get any component to show up inside of a Vue application, you have to manually wired up to another component.

So, to get that search input to appear, we have to wire it up to the app component. To do so, I'm going to go back over to my code editor and open the App Vue file.

Inside this file, we're going to add an import statement into the script tag right here.

So right above the export default, we'll add in import search input from and then we're going to provide a relative path to that search input file right here.

So, we'll say dot slash in the components directory, look for the search input file.

Then to get this thing to render inside the app component, we could add a search input tag inside the template.

So, I'm going to remove the text ‘Video Portal App’ because we don't really need that text anymore and I will replace it with search input just like that.

By convention we can do a self-closing tag right here if we do not expect this component to contain any elements of its own.

But in this case, we will do. So, I'm going to use a full tag.

OK, so let's save this right here and we'll go back over the browser and just see what happens.

So now here, if you open your console, you might notice a little error message.

So, this is maybe a little bit of a surprise.

You see, whenever you want to nest a component like this or show one component inside of another, it's not quite enough to just import the components and then place it into the template.

There's one other step that we have to go through as well.

Now, might seem like it's kind of silly of me to skip this extra last step, but I specifically want you to see this error message right here, because the last thing we have to do is something that's very easy to forget.

All right.

So back over inside of our code editor still inside the APP Vue file, we imported search input right here and then we used it inside the template. So that steps one and two.

Step number three, this is the one that you might forget kind of frequently. Inside of our options object that represents the app component.

We're going to add on one additional property called components.

This is going to be an object that lists all the different components that are going to be used inside of the app's template.

So, we have to place search input inside of here so that our app component knows that it might see a search input tag inside of its template.

To do so, we'll add in search input and we're going to have a key of search input and a value of the actual component itself and that component is Search input. We can use a little bit of ES6 syntax to shorten this up because the key and the value are identical.

So, in here we can simply search input. let's now save this, I'll flip back over to the browser and now you'll notice that we get our input on the screen.

If I zoom back out, you probably see something a little bit more like this.

I just usually have my browser a little bit zoomed in so you can see everything more easily, OK.

So again, to show one component inside of another, we add the import statement. We use a tag inside of the template.

And then the last step that I don't want you to forget, and therefore I showed you that error message.

Don't forget to add the property to your components property into the scripts.

That looks good.

Let's continue in the next section and we'll start thinking about how we're going to get our search input file to recognize the user type in some text in here and then do a search on the Pixabay API.

Event Directives

I'm inside of our search input vue dot file, which is now visible inside of our application. If we go back over to the browser, you'll see our input element appearing right here.

So, our goal is to now make sure that any time a user types inside this input, we trigger some type of search for the Pixabay API.

Let's first get started by just making sure that we can get some event or some notification any timeuser types inside of here.

We already did this previously back over on one of our image generator exercises.

You'll recall that we're going to add a directive to this input tag right here.

And then we will also add a method object to our component definition inside of here.

So, let's get to it.

I'll first begin by adding in the methods object to our component definition.

Remember, methods give us the ability to change data inside of our application.

So, I'm going to make a method called on change.

So, let's define the function will be called with an event and that event right there is going to contain the text that the user just entered into the input.

So, for right now, let's just do a quick console log of events dot target dot value.

We'll also wire that method up to our input element right here.

You'll recall that we can do so by adding a custom directive.

So, we'll save v dash on, then a colon and then the name of the event that we want to watch for. In this case, an input event. Then we'll do an equals double quotes and then we'll list out the name of the function or the method that we want to call any time this event is triggered.

So, we want to run the on change method any time an input event occurs.

So, we'll write on change like so.

All right.

So, let's save this.

We'll go back over to the browser.

I'm going to do a refresh of the page and then I'll enter some text in and you'll see the console logs start to appear. Great.

There's one little refactor I want to do to this directive right here.

The refactor that we're going to do is going to have absolutely no effect on what's going on with the on change directive.

It's only a syntactic change.

So, the change we're going to make is just to make this code a little bit more legible. It does absolutely nothing else.

So, we're going to say that rather than putting in the dash on and then colon, we're going to replace all that.

So, everything from V to the colon with simply an add sign. So you can read this right here as meaning any time someone triggers an input event on this element, run the on input function.

Again, this symbol right here is one hundred percent equivalent to writing V Dashon and then a colon.

So why is there this different syntax?

Well, honestly, I think people just got tired of writing V dush on colon again and again as the exact same effect behind the scenes on setting up an event listener on this input element.

In general, I do recommend that you try using this shortened syntax where possible just because it's going to clean up your templates a little bit.

All right.

So, let's take a quick pause right here and we'll continue the next section and start thinking about how we can use this search term right here to do a search on the Pixabay API.

Data Sharing Between Components

The search input component is now aware of what a user is typing into our application by on change method we just created last video. So, we've now got some search term that we can use to search the Pixabay API with but we need to ask ourselves a very important question.

Where inside of our application should we locate the code to execute that search?

Should we put the code for that inside of the on change function right here?

Well, maybe there's a better place to put it.

Let's look at a couple of diagrams that will help us make a better decision on where to place the logic to do a search on the Pixabay API.

So, let's first begin by reviewing our component hierarchy or how all our components are going to be wired together at the very top.

We've got our app component right now, the app components showing the search input component.

Eventually the app will also show the show media and the videos components as well. There's one very important element that I want to remind you about here.

In our application, we are trying to display a list of videos to our user, and we're also trying to allow them to select one individual video as well.

So, it seems to me like there are two areas in our application where we need to know about the current list of videos that the user has search for.

In particular, the show media component probably needs to know about our list of videos, and the videos does as well.

But the search input component doesn't really care about the list of videos that is being fetched from Pixabay API.

The search input doesn't show any videos.

All it does is accept some user text and that's it.

So, when we start thinking about what components should execute the search for these Pixabay videos. Well, I think that maybe it would be most appropriate to put the code for that inside the app component.

You might be thinking, well, wait a minute, if the show media and the videos are what needs access to the list of videos, why don't we put the code for doing that search in one of those?

Let me tell you why.

Here's a kind of simplified diagram.

And I took out a couple of components that were being displayed underneath the videos component.

So, any time that we want to communicate information throughout our app, it is very easy for us to communicate data or just arbitrary information between components that are parent in children to each other.

So, in other words, the show media component can easily communicate data with the app component and likewise, the app can easily communicate with a show media.

The app can also usually communicate with the search input or the videos.

However, it is not very easy for us to communicate across sibling components.

So, show media, search input and videos are all being displayed by the app component, which means these three are all sibling components.

Communicating directly from the show media over component to the videos component or from the search input over to the show media is not very easy.

So that means that if there is some common data that needs to be used between both the show media and the videos, it would be easiest to locate all that data inside the app and then the app can easily share that information down to show media or share it down to the videos component.

So essentially, it's really going to be ideal for us to locate all the code, to execute the search and then store the list of videos that have been found inside of this app component and then communicate that data down to the show media and down to the videos.

So, with that in mind, let's continue in the next section and we're going to figure out how we're going to make sure that the search input can somehow tell the app component that it needs to execute a search of the Pixabay API.

So quick break and I'll see you in just a minute.

Communication with Props and Events

Now, we have to do is figure out how to get the search input to somehow tell the app component that it needs to execute a search in the first place.

At present, the search input component is just reading in input that is provided by the user.

So, the search input needs to somehow say to the app, hey, we got some new input. It's time for you to execute a search.

So, let's now talk about how we can communicate from a child component up to a parent one.

So, we essentially want to have the search input component right here, communicate to the app that it needs to run a search with a term.

To communicate from a parent component to a child component is slightly different, depending upon which way we're trying to communicate.

So, if we want the app component to somehow provide some information or provide some data or otherwise, just communicate something down to the search input, then we do so by going through a process called passing props.

Props is short for the term properties.

So any time we want to communicate from the app down to the search input, we will pass a prop.

A prop can be anything from an array to an object, to a string, to a function, all types of values and we'll see many examples of this over time. However, communicating from the search input up to the app.

A search uses a slightly different system.

So, to communicate upwards, we make use of the event system that is included in Vue.

So, if the search input wants to say something to the app, then the search input needs to emit an event.

The app component will then listen for that event and whenever that event is triggered, the app can run some custom code.

Like in our case, it would have the ability to run a search on the Pixabay API.

We're going to see many examples of communication from top to bottom or from bottom to top throughout this course.

So, let's take a quick pause right now.

We'll come back to the next video and we're going to start to wire up some events between the search input and the app component.

Emitting Events

In the last section, we spoke about how a parent component can communicate to the child by passing props and how a child can communicate with the parent by emitting events.

So, let's now go back over to our code editor and add some code to the search input to make sure that it emits an event any time someone changes that input element.

I'm going to open my code editor. I've got my search input Vue file open still and then I will find out on-change function.

So, every single time that this function right here is called, we are going to want to emit an event to inform the app that there is a new term to use to search the Pixabay API with.

To do so, I'm going to delete the console log that we have inside of here and then I will replace it with a function call that is going to emit an event. To do so will write out this dot dollar sign emit.

The first argument to this function is the name of the event that we want to emit.

In this case, I'll use the name of new term to indicate that, hey, our search term just changed, we got a new term to handle.

As a quick aside, I want to mention that back up here, when we were listening for an input element or an input event, this is a very special event name.

In other words, input elements, emit events called input and input event, any time someone add some input.

However, when you and I are emitting events, we have complete control over the name that we choose.

So, this right here can be ‘new term’ or it could be anything whatever you want. You can name it as you wish.

But in general, I recommend using the name of what is about to occur, like the thing that is about to have something happen to it, and then a verb that indicates what just we need to do. In this case handling a new term is appropriate.

As a second argument, we can provide some additional information about the event that just occurred.

So, in this case, we're probably going to want to inform the AB component about the new search term, which is available on event target value.

But we can write it a bit cleaner. So, I am going to define a variable called term and this term equals to event dot target dot value. Now we can pass this term as a second argument into emit event just like that.

Let's say I want to mention here is the dollar sign in this emit function name.

Just you know, there's nothing special about using a dollar sign with a function name.

This is one hundred percent valid JavaScript.

So, I should say the Vue authors could have just as easily called this emit, but they decided to put the dollar sign in there just in case you ever decided to add in some additional property to your component called emit as well.

So, the dollar sign here really means nothing.

It could have been any other character.

It does not make the function special or anything like that.

OK, so now every single time someone type something into our search input, we are emitting an event called new term.

Let's take a pause right here.

When we come back, we will work on our app component and make sure that it listens for the search input to emit an event and each time that occurs will then trigger a search on the Pixabay API.

So quick break and I'll see you in a minute.

Listening for Custom Events

Our search input component is now emitting an event, any time someone enters some new text. We're now going to open our app component and make sure that it listens to the search input.

Any time the search input emits this new term event, we'll make sure that the app has the ability to run some code that will do a search on the Pixabay API.

As a quick aside, we have not yet added in any functionality to do a search on Pixabay.

So, we will need to make sure that we add that in at some point as well.

I'll first begin by opening the App dot Vue file. Inside of here will first add a little bit of code to make sure that the app component listens for events coming from the search input.

To do so, we will add in a directive just like we have previously, to listen for input events.

So, on the search input, we could add in the v dash on then the name of the event that we want to listen for, which is new term.

Then we'll add an equals double quotes and the name of a method that we want to run any time this event right here is triggered.

So, I think that a method name of something like handle new term would be completely appropriate.

Let's now add in that method quickly and then we'll talk a little bit about the syntax right here again. Down inside of my component definition, I'll add in another key value pair of methods and then I will define a function called handle new term.

So, handle new term is going to be a function.

OK, so now once again, any time this event right here is triggered, we will run that handle new term function.

So, here's handle new term. You'll remember that back inside of search input, whenever we emit new term, we also pass along the new search term that the user just entered, because we added this as a second argument right here. It will show up as the first argument to our callback function or the event handler right here.

So, the first argument to this thing will be the new search input that we get.

So, this is going to be a string, the same string that the user just entered that search input for.

Right now, let's add in the council log of search terms.

You might mix this search input term and search input component. I know it confusing, but these are two different things. This search input starts with lowercase ‘s’.

OK, now, before we go test this out, I want to draw your attention to the listener that we added right here.

So, chances are this look very familiar.

It's almost identical to the same syntax. We used to listen to an input event on the input element.

So, the way that we listen to events triggered by normal HTML elements like, say, a button or an input or a text area is completely identical to how we listen to events on components that you and I author.

So, we just wrote out right here V on and then the event name that we're listening for, you'll recall that we can shorten the syntax right here to be simply @, which means the exact same thing.

Like I said previously, we generally want to make sure that we use the same form of syntax throughout our entire project.

And I said, like I said, we'll be favoring this shorter @ syntax throughout all the projects that we work on.

OK, so let's save this.

We'll go back over to the browser, and we'll test out our code to see how it's doing.

So back inside the browser, I'll open our application.

I'm going to do a quick refresh the page and then if I type some text in, you'll again see all the text logged out at our console.

Now, note that this is not the same console log that we had previously before we had our search input component console login, that search term.

But that console log no longer exists over here.

We moved the console log over to the app component.

So that means that every single time a user types the search input emits an event, the app component listens for that event.

And then the app components handle new term method is executed, which is where the console log takes place.

So, seems to me like the search input component is successfully communicating up to the app, which is great.

So now we need to move forward a little bit and figure out how we can use this search term right here to run a search on the Pixabay API.

So, let's take care of that in the next video.

Pixabay API Signup

Our app component is now aware of any time the user enters in a new search term, so inside this handle new term function right here is probably where we will want to locate a little bit of code to somehow initiate a search for some videos.

Let's first look at a diagram that's going to give us a better idea of how we're going to use the Pixabay API and then we'll go look at the API itself.

OK, so every single time a user type something in, we are going to use a library called Axios to make Ajax requests over to the Pixabay API. In the Ajax request we will provide this term that we are looking for.

The Pixabay API will then execute a search on its own servers and then respond to us with a list of videos that match that search term.

In order to make use of the Pixabay API, we first have to sign up for it and get an API key.

Let's take care of that right now to sign up for the Pixabay API.

I'll open up a new tab inside my browser and then I'll navigate to pixabay dot com.

At this page, at the top right corner we click join and then we will see a sign-up modal. You can sign up with any options here. You can use your Google or Facebook account already or create new pixabay account with your email.

It doesn't have to be a real account and we're only going to use it in the context of using this Pixabay API.

When you first come to the profile dashboard, you'll see menu icon on the top right corner. Click that and a menu opens up the choose API link. Now we can go to API documentation.

So either way, we just want to open up the wizard to create a new project.

We now have to get an API key so that we can actually access the API itself.

OK, very good.

I'm going to select all API key over here and copy the key.

Then I'll go back over to my code editor.

I'm inside of my laptop Vue component and directly underneath our import statement right here.

I'll add in a new variable declaration called API Underscore Key and I'll paste my application inside of a string.

You'll notice that I use capital letters with the underscore right here to declare the variable name.

That's because this is a constant variable and I never expect this to be changed.

OK, so that's pretty much all we have to do.

Well, we'll take a quick pause.

So, I'll see you in just a minute.

Searching Pixabay

In the last section, we successfully signed up for a Pixabay API key.

Remember, We will use fetch method to actually make a AJAX request over to the Pixabay API.

We're going to add some code inside of the handle new term function right here to execute a search on the Pixabay API.

Now, before adding that code in, I do want to point you very quickly to the Pixabay API documentation.

So, if you do a search for Pixabay API, one of the first results that you're going to see is the Pixabay API Documentation right here.

So, this is the actual documentation on how we use the API.

Now, I'll tell you everything you need to know about this document.

I just want to point it out in case you want to read over some of the stuff on your own.

So, let's go back over to our app component and we'll add a little bit of code into handle new term to execute that search.

Now down inside of handle new term method we’ll add some code to fetch method make a HTTP request to the Pixabay API.

To make the request itself, we'll say fetch then we'll enter the Pixabay API specifically for the search and point the URL for this is https colon slash slash pixabay dot com slash api slash videos and question mark.

Why we put a question mark here because after this pixabay api end point we want to make a request with some queries.

API Query parameters can be defined as the optional key-value pairs that appear after the question mark in the URL.

Basically, they are extensions of the URL that are utilized to help determine specific content or action based on the data being delivered.

Query parameters are appended to the end of the URL, using a ‘?’. The question mark sign is used to separate path and query parameters.

If you want to add multiple query parameters, an ‘&’ sign is placed in between them to form what is known as a query string. It can feature various object types with distinct lengths such as arrays, strings, and numbers.

Then as a second argument, we're going to add in a couple of different parameters to configure this request and make sure that it's clear exactly what search term we want to execute the search with.

After that we say plus and create a new URL Search Parameters. Inside this parameter we are going to add an object that take some queries that include our API key inside of here as well.

Let's first take care of the API key. To attach the API key to this request.

Now we will say Key is the API key that we defined just a moment ago.

So, our key is API key then the second thing we're going to add search query. To indicate the actual term that we want to use to search the Pixabay API with will add in Q which is short for query. In our query is for our search input. So, say Q of search input just like that.

OK, so this code block right here is going to execute the actual request over to the Pixabay API.

Now I will assign this request to a variable called response.

Any time we make a request over the network, it takes some amount of time to complete that request.

So fetch method returns a promise which allows us to get a little notification when the request is complete. So we can handle that promise by using async await syntax.

It is a special syntax to work with promises in a more comfortable fashion. It’s easy to understand and use.

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically. In this case we will put async keyword in front of handle new term method.

So, async ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There’s another keyword, await, that works only inside async functions.

The keyword await makes JavaScript wait until that promise settles and returns its result.

Await literally suspends the function execution until the promise settles, and then resumes it with the promise result. That doesn’t cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events.

So in this case we will put await key word before fetch method just like that.

Now we need to turn this request into a JSON object and get the data. To do so, I will define variable called data and it is going to be equal to response dot json method and we call it

Then for right now maybe we just console log out the response like so.

OK, so I'm going to save this file and we'll fly back over to the browser, and we'll give this a shot, I'm going to do a refresh of the page and then I'll do a search for, say, sky and in the console we got some results.

So, let's take a quick pause.

We'll come back to the next video.

Investigating Video Responses

So as soon as I enter that sky you'll see a collection of console logs over here.

So, remember, right now, every single time a user enters some text right here, we immediately trigger an event.

And that event instantly triggers an HTTP request over to the Pixabay API.

So, these are all results of a search on the Pixabay API.

This seems like overkill right now to do this many searches, but we'll address that in just a moment.

For now, let's investigate one of these responses and see what type of data that we get back from the Pixabay API.

So inside of here, you'll see a couple of different properties.

The data that came back these three properties

You can then look at items under hits property which will list all the different Pixabay videos that were found from our search request.

Inside of one of these videos, you'll find stuff like an comments, id likes and so on.

There are the information about that we get about this particular video.

So, this case, the tags are stars, long exposure, starry sky, which is definitely a video about sky, I'm sure.

OK, so this looks pretty good, it looks like we have some information about this list of videos coming back from the Pixabay API.

So, let's take a quick pause right here.

And when we come back, we'll talk about how we can use this list of videos to somehow display them inside of our application.

So quick pause and I'll catch you in just a minute.

Rendering a List of Videos

In the last section, we were able to execute a search on the Pixabay API for some given search term.

We then got back a list of objects that described all the different videos that were found during that search.

So, we now need to take this list of videos right here and somehow display them on the screen.

Remember, if we go back to our original mockup, we had said that at the very top we've got that search input component.

And then to render that list of videos bottom here, we would create a second component called videos.

The videos would be responsible for maintaining the overall list of videos, but we were also going to make a separate component called Item to represent one individual video.

Now, one thing I want to expand on just a little bit is that we are kind of breaking up this list into two separate components.

Just so you see a very good example of how we can easily take advantage of Vue to nest components in each other.

But if this was a real project, we could have just as easily decided to render the entire list using just one individual component.

Really, it's kind of an engineering decision.

If you have individual items inside the list that have a lot of complexity or a lot of events tied to them, well, then that might be a signal that you would want to break out a second component inside there rather than trying to stuff all that logic directly into the component that renders the overall list.

So, with that in mind, in this section, we're going to make this videos component and then wire it up to our app component, will then take a quick break and then figure out how we can communicate the list of videos down into the videos component.

So back inside my code editor, I'm going to create a new file inside of my components directory, and I will call it videos dot Vue

Remember, we always make one separate Vue file for every component that we create.

Inside of here we'll write out our template tag, our script tag and our style tag, and you'll notice that we haven't really been using our style tags that much so far, but we are going to come back through and do a pass on styling inside of our application in just a little bit and that's going to make sure that our application looks nicer.

Inside of here, I think that we want to render a list of videos and to render that list, we probably need some appropriate element inside the template that deals with lists appropriately.

To render the list of elements, I think that will put a UL inside of here. For right now, we'll just put the text videos inside there, but we'll come back in a little bit and make sure that we get that list of items or list of videos correctly rendered inside of here.

Then inside of the script tag, we'll add in our typical boilerplate, which is export default, and then an object that has a name, property of videos.

Before we go any further, let's make sure that we wire this component up to our app component just to get it visible on the screen.

So I'm going to go back into our app component and then I'll go through that three step process that we have to go through every time that we want to show one component inside of another. We will first add in an import statement. So, I will import videos from components videos.

Inside of my template itself, I'll add in a tag for the videos component and then step number three is to add it to the components property on the component configuration itself.

So right after search input, I'll put a comma and then videos like so.

OK, so let's save this file.

Well, then go back over to our browser and we see the text videos appear.

OK, so I think that the component is showing up on the screen.

Let's now continue in the next section and we'll figure out how we can communicate our list of videos that was just retrieved from the Pixabay API down to that component and then render them out as a list.

Updating Data Causes Rerenders

Our handle new term method is now making a request to the Pixabay API to fetch a list of videos, and then we've also got that videos component set up to take a list of videos and display them somehow on the screen.

The next thing we have to do is to figure out some how to communicate the list of videos that we fetch down to that videos component.

So, we have to communicate from the parent down to the child component.

In this case, the videos component is the child because it is being displayed from within the app component.

So, let's look at a diagram that's going to give us a better idea of how we're going to take that list of videos that came back in the Pixabay API response and communicate it down to the videos.

All right.

This is a timeline of sorts where we're going to start off at the very top and end up at the bottom.

So, in our flow for getting a list of videos to appear on the screen, everything starts with that handle new term method being called.

When we call that method, we're going to make a request to the Pixabay API.

The Pixabay API will then respond with that list of videos or give me that list of objects that represents individual videos.

We looked at that list just a moment ago inside the console of our browser, and we saw that we had an array of objects and each of those objects represented one video.

They had properties like the videos ID.

They had the like and download count, tags and some other properties as well.

So now the next step is where things start to get interesting.

Remember that any time that we have some type of variable, or some type of information tied to a Vue instance or a component, we store that information on that data property.

So, we use methods to update our data.

Whenever we update our data, the component automatically renders and then we can optionally make use of that computed property to somehow twist those values into something that can be displayed on the screen.

So, we're going to take that list of videos that are returned and we're going to assign it to a property on our data object inside of the app component.

Because we are updating the data property, we are going to cause the app component to automatically rerender its template.

Now, the important thing here to realize is that when we show one component inside of another, as we are right here with the videos component, whenever the app component renders its template with some new data that causes all the child components to be rendered as well.

So, whenever we update data in the app component, videos will be updated at the same time and that all happens automatically.

So, we need to take that data property that contains the videos.

We're going to take that data and we're going to stuff it down into the videos component.

The videos will then receive that list of videos and then once they are inside there, we can figure out exactly how to get the videos to render them out on the screen.

So that's the big idea here. That's what we're going to do.

We're going to take the list of videos. We're going to store it on data. When we do that causes everything to render and then we can get our list of videos inside a videos component and render them out inside of that things template.

So, with that in mind, let's continue with the next section where we're going to take our list of videos that are being retrieved inside of our handle new Term and make them make sure that we are storing them on our data object.

So quick break.

We'll take care of that in the next video.

Data in Components vs Instances

In the last section, we spoke about how we're going to take our list of videos and store them on the data property of our app component. When we store the list of videos on that data property, it will cause the app component to rerender and automatically rerender the videos at the same time. So that's how we're going to communicate the list of videos down to the videos component.

Before we start adding code to actually implement this data thing, I want to do a quick review on how we use data on the previous app we worked on over in Code Pen.

So here's our Copan implementation over here.

You'll recall that we made that data property that had text input and we defined the data property as being an object.

Now, this is going to be a tough little thing in the world of you.

As I mentioned several times before, there's more than one way to do just about everything in Vue.

And the same is true of how we define the state of property right here.

So depending upon whether or not we are working with a Vue instance or a Vue component, we are going to define this data property slightly differently.

So first, let's take a look at a diagram that's going to put out the difference between these two in very simple terms.

So if we are working with a Vue instance, then we can define that data property as an object or a function that returns an object back inside of our code to an app over here.

We were working with a Vue instance.

So this is a signature to make a Vue instance right here.

So because we were working with a Vue instance, we could define the data property as a object.

But once we start working with Vue components, that data property must be a function that returns an object.

So in other words, we have to do something that looks like function and then inside of this function.

There we go, we can return an object like soap.

So this is what we have to do whenever we are working with a Vue component.

And right now, inside of our Apter Vue file, yes, this is a component right here.

So if we want to make use of data inside this thing, we have to make a function that returns an object.

Now, at this point, I'm sure you're very curious about why there's this distinction.

Why can we do an object simply with a Vue instance, but not with a component?

Well, let's expand on that very briefly, just to give you a good sense of why this requirement is in place.

OK, so this is a diagram here where we're going to kind of investigate what happens when we create that data property as an object.

So I want you to imagine for just a second that we break the rules.

Maybe we break the rules by saying, you know what, we're going to make an app component and we're going to define our data property as an object rather than as a function.

When we define that data property as an object, you can imagine that it essentially gets created inside of our computers memory up here.

Let's make sure that's actually visible.

Here we go.

So inside of our computers memory, we've got that object just sitting around when we then make a instant's out of this app component, like let's say we make three instances over here, every instance will be referring to the exact same object inside of memory.

So if any of these instances start to modify this object right here, then all the other instances will think that the data object is being modified as well, because they're all sharing the exact same object that has been created.

The easiest way to get around this is to make a function that returns an object.

When we do that, every single time we create a component instance, they will each have their own separate copy of that data object.

And so each of them will refer to the their own separate copy of data sitting inside of memory like so.

So essentially because of this data sharing issue, we have to create functions that return objects when we're working with data inside of a component.

So with that in mind, let's take a quick pause.

When we come back, we're going to update our app component so that it has a data property that returns an object.

Updating Data

In the last section, we spoke about how we're going to take our list of videos and store them on the data property of our app component. When we store the list of videos on that data property, it will cause the app component to rerender and automatically rerender the videos at the same time. So that's how we're going to communicate the list of videos down to the videos component.

So inside of my app component, right between our list of components and our methods, will add a new property that will return our object that represents our data.

So, we'll start off by defining the data property.

This is going to be a function that's going to return an object.

Now, remember, we can use a little bit of ES6 syntax to simplify this, to be simply data and then a set of parentheses. That inside of here we will return an object. In this object right here is going to initialize our data.

This is going to be like the starting state where the initial ingredients for our component.

So, we have to start to think about what different properties our data is going to contain.

Well, we know that we need to have a list of videos because that's what we are retrieving from our handle new term function right here.

So, I think that I will add a property to my data object simply called videos and because this is supposed to be an array of objects where every object represents one video, I'm going to initialize this to be an empty array like so.

Now, down inside of handle new term method, where to wait for a response from the Pixabay API, we can replace the console log with some logic that will take the list of videos that are contained inside this response and update our data properties, specifically the videos property with that new list of videos.

So, to capture that list of videos and assign them to our videos, piece of data right here, I will replace the console log statement that we have.

I'm going will say this videos is data dot hit just like that.

So, again, remember this hits property right here?

Not at all related to our component instance.

This is the data property that is tied to our response object that comes back from Pixabay and then on that hits property is the list of videos that is contained within items right here.

OK, so that's pretty much it.

So now any time we make a request, well, then get back our response.

We take the list of videos that were found, and we update our data piece of videos right here inside of our component instance.

When this line of code right here is executed, that is going to cause our entire template inside this app component to automatically rerender.

So maybe as a quick test just to make sure that everything is working correctly, let's render all array videos that have been found on the Pixabay API on browser. To do so, I'm going to add a little space underneath our videos component and then remember to print out any value from our component, we can add in those double curly braces. Then inside of here we could put down a very limited amount of JavaScript code, kind of like just do a little quick calculation.

Usually, we would use a computed function to do this.

But because this is a very temporary little thing that we're going to print out, we won't bother with making a computed function.

So inside of here, we can say videos dot valueOf just like that. So this is going to be the videos that are tied to the data property and then we're going to retrieve the array and print it out.

OK, so let's save this and we'll go back over to our browser and you'll notice that now we've got that zero empty array appearing on the screen.

But if I type in, say, car, then I whole array printed out right here.

OK, so that's great.

Not only are we taking a list of videos and assigning it to the data property on our component, but we can very easily see that every time we update that data property, it causes our component template to automatically rerender and update the content that we see on the screen.

So now we've got the list of videos stored on data.

We can move on to the next step here, which is to make sure that we take that list of videos on the data and pass it down to that videos component where they will then be rendered out as a list.

So quick break and we'll take care of that in the next video.

Communicating from Parent to Child

In the last section, we updated our data property of videos whenever we got a response back from the Pixabay API, because we updated a property that was defined on our data object that caused our app component to automatically render.

That's why we saw this videos dot valueOf thing right here, automatically change inside the browser any time we search for a list of videos.

So, remember, any time you want to update your component or cause it to render, you want to update a data property.

Now that we've got this in place, we want to somehow communicate this list of videos down to the videos component, because the videos component is responsible for rendering out information about each particular video on the screen.

If we want to communicate from a parent down to the child that we make use of the props system, and you'll recall that props is short for properties.

So, let's talk a little bit about how we communicate properties from the app down to some child component.

So, this is going to be a two-step process.

The first thing we're going to do is add a little bit of code to the parent component. Inside of the parent components template, we are going to add another directive to the child tag.

So as a quick example of that, here is our app component. It is the parent to the videos component. Here's the videos tag right here.

So, we are going to add a directive to this tag.

That directive is called the V Bind Directive, and we'll talk about what v bind means in just a moment.

Right now, all we really need to understand is that the parent components template needs to be updated. To make sure that this data flows down as a prop to the child.

Then inside the child, we're going to add a little bit of configuration to tell it that it should expect to receive some property from its parent.

So, in other words, communicating data from a parent down to a child requires some configuration not only in the parent, but inside the child as well.

Both parties must be aware that they need to exchange some amount of information.

So, let's now begin by opening our parent component and we'll take care of step number one right here, which is to add that v bind directive to the child component’s template tag.

OK, so here's my app component, it's the parent in this case. I'm going to first begin by deleting that videos dot valueOf tag that we had just a moment ago, because we don't really need to display array of videos that have been returned. Then find the tag that represents the child component that I'm trying to communicate some information to. It is right here.

So, to tag, we are going to add a V bind directive.

To do so, we will write out V Dash Bind a colon, then we will write out the name of the property that we want this to show up as inside the child.

I'm going to use the name videos. Because we are providing this name right here are videos.

That means that inside of the videos component we are going to have access to a property called videos.

If we had called this something else, like if we had this video list, then inside the videos component we would have access to a property called video list instead.

But in this case, it makes sense to simply call it videos.

Well, then place an equal sign, a pair of double quotes, and then inside the double quotes, we're going to write out the name of the data property that we want to share from the app component.

So, the app component wants to share its videos property right here down to that child.

So, we will write out videos just like that.

OK, so I know that seeing the word videos here twice is a little bit confusing, so just remember, the left-hand side says this is the name of the property that we want to have shown up inside the child and this is the name of the property that we want to share as it exists inside the parent.

Now, you might be a bit curious about what the purpose of this v bind thing right here is at all. V bind means that any time the videos property is updated inside of the parent, it should automatically try to render the videos component and provide that new list of videos down to the videos.

So, you can kind of think of v bind right here as meaning like, any time videos is updated, try to rerender the videos and provide it the new list of videos.

That's the purpose of v bind.

It kind of joins these two components together and ties them or binds them together, so to speak.

Now one other quick thing I'm going to throw in here.

Now, there is a shorthand form for defining a v bind directive as well.

This is another case yet again where there's more than one way to do things with Vue. So, we can shorten this directive right here from instead being the dash blind to instead simply saying colon videos.

So, the two are identical. The v dash blind means the same thing as simply colon videos.

So, this is another instance where you will want to be very consistent throughout your application and always either use this shorthand form of just colon and then the property name or everywhere use the dash find. I will keep use v dash bind.

OK, so we've now set up our parent component to tell it that any time its videos piece of data which is defined right here is updated, it needs to pass that list of videos down to the videos.

So, let's now continue in the next section where we're going to add some configuration to our child component to tell it about the props that it should expect to receive from its parent.

So quick break and we'll take care of step number two in the next section.

Prop Validation

In the last section, we added some configuration to our app component to take our list of videos and pass it down to the videos component.

We're now going to move on to step two of passing props, which is to add some configuration to our child component, to tell a bit about what data it should expect to receive from its parent.

So, I'm now going to open up my videos dot Vue file inside of our JavaScript section right here, we're going to add a new property called Props.

To tell this component about what data it should expect to receive from its parent, we will place an array and inside this array we will place a number of strings.

These strings are going to be the exact property names of the data that it should expect to receive from the parent.

So back inside of the app, we had said that we want to pass in our list of videos down to the videos component and inside the videos component.

We want that property to be called videos. So back inside videos component.

When I list out all the property names that we should expect to see. I will put videos right here.

So, there's a distinct tie between this name right here and how we called it inside of the parent component right here.

So, on both sides, we'll call it videos.

Now, defining our prop list right here can be as simple as listing out the names of all the different properties that we expect to receive.

But we can also add in some amount of configuration or validation to make sure that the child component right here is receiving the correct type of property as well.

So, we might want to validate to make sure that the parent is passing down, say, an array or to validate to make sure it's passing down a number or a string or an object.

So, if we want to and this is optional, the Vue documentation recommends you do this if you can but this is an optional step.

We can instead of providing an array of strings right here, provide an object where the keys are the names of the properties that we expect to receive, and then the values are the type of the property that we expect.

So, in the case of our videos property right here, we expect it to be an array of objects or more specifically, an array. So, we can say array.

This tells this component that it's going to expect to receive a property called videos and that property better be an array and if it's not an array, then this component is going to throw an error.

So again, you can either specify an object to do a little bit of simple type checking right here if you want to, or you can simply list out the names of the properties that you receive.

It's up to you in your project.

Again, I want to mention that The Vue documentation always recommends that you add and validation if you can, but it is your decision.

OK, so we've now taken care of step number two for passing props for from a parent down to a child.

So now inside of our child component, we can get take access to that prop that has been provided and make use of it anywhere inside of our component. So, we can create some computed functions, or we can create some methods, or we can directly access it inside of our template as well.

Let's do that last one for right now just to make sure that this data is showing up.

So, inside my template, I'm going to try to, again, print out the number of videos that we have been passed from the parent.

To do so, I'll put down those double curly braces again, which is how we do string interpolation inside of our template to print out some simple value.

And then inside of here, I can refer to the prop that has been passed down by writing out simply videos.

So, I should say it is identical to what we did previously when we accessed some data property inside of our template. Back inside of app, here is my app template right here.

We got access to the data property videos right here by simply writing out curly brace, curly brace videos.

So, this access to our data property that was defined inside of our component, we do the exact same thing when we are accessing a prop as well.

So, the very common theme here is any time we want to access any property, any data, or any computed function from within a template, we always write out just that things name. We don't have to write out this. We don't have to call it. We don't have to do anything like that. We just write out the name.

So, in our case, we specifically want to print out the array of videos that have been passed down as props. So, we'll do videos valueOf.

All right.

So, let's now flip over to our browser and make sure this thing is working. I'll go back to my application. You'll see that we now see a videos and then a empty array after it.

But if I search for, say, car, I very quickly see videos and array of videos. Great.

So, it looks like we have successfully shared some data from the app component down to the Child videos component.

So now we've got that data showing up inside of here as props, we can now start to add some logic to the videos to render those out as an actual list.

So, let's start taking care of that in the next section.

Separate List Components

Our videos component now knows about the list of videos that it needs to somehow render to the screen. As a quick reminder, remember that the entire list itself is going to be comprised of two separate components.

The first is the videos, which is what we are working on right now.

The purpose of the videos component is really very simple and straightforward. It's going to take that list of videos that we passed down to it, and then it's going to try to render one item component for each video that it received.

As a quick reminder, we did not have to strictly divide this out into two separate components. I just wanted to give you a very good example of how you would render out kind of a multi component list. Using this technique of having two separate components to render the list is most useful when you have list items or like individual rendered items inside of here that are rather complicated or complex in nature, because it allows you to more simply isolate code that is related to rendering a list from code that is related to simply rendering a single item.

So, it's a little bit of arbitrary complexity that we threw in here, and we could have used one single component to do both steps. Both render the list and each individual item, but this is the path we're going to go down. So, let's continue.

I'm back inside of my videos component right here.

We can't really do a whole lot inside a videos component until we create the item component as well.

Once we create that, then we can use the video item to actually put the list together inside a videos.

So, let's first get started by creating a new file for the item component.

Inside my components directory, I'll make a new file called Item dot Vue. Inside of here we will put together some of our usual boilerplate.

So, we'll say template, script and then style.

Inside the script tag, we will immediately place some of the boilerplate that we need whenever we are defining a component.

So, I'm going to place inside of here export default and then our lists are curly braces, and I will provide a name for this component, which is going to be ‘item’ like so.

Then inside of the template, we're going to put down just a little bit of stand in HTML for right now.

i think that since we have a ul being displayed inside a videos, we will make each individual list item L-I (li).

So inside of here, I'll say li and we'll say List Item right now.

So, we've now created the item component, let's make sure that we import this into the videos component and wire it up as a component that the videos component can display.

So, here's my videos, I will go down to the script tag, I'm going to add an import statement, so I'm going to import item from item like so.

And then remember, any time that we want to tell this parent component that it has access to this child, one, we must add the components property to the parent component itself.

So inside of the videos configuration object right here in between name and props, I'm going to add in components. That's going to be an object and then I'll add in the name of the component that this thing might need to display, which is item.

So now inside of our videos component, we have access to the videos item component. So, all we really need to do now is figure out how we can create one item component for every video that is contained within our videos prop right here.

So, to build out that actual list, we'll take care of it in the next video. We're going to start looking at a directive that you're going to be using many times inside of your projects any time you want to build a list.

So quick break and we'll take care of that in just a moment.

Lists with V-For

In the last section, we created the item component and we wired it up to its parent of videos component.

We now need to add some code to the videos to make sure that it shows exactly one instance of the item for every video that we have fetched.

So far, it looks like every single time we get a response back from the Pixabay API, we are always getting a list of 20 videos.

So, in total, I would expect to see 20 item components.

To render out this list, we are going to use another directive inside of our videos component.

So inside of the videos template right here, I'm going to delete some of the stand in text we have right now, and then we are going to replace it with an instance of item component. So, let's say videos item.

Now, as I've written this component out right now, this would only create exactly one instance or one copy of the item component. But that's not the behavior that we want.

We want to have one instance of this component right here created for every single video that has been loaded up.

So, to tell our component that that is what we want to have happen here, we are going to use a directive called the V for Directive. So, let's write out the directive itself and we'll talk about how it works.

Inside of item I'm going to write the dash for then it equals and then video in videos just like that.

So, again, we refer to this as the V for directive and it's a directive that you're going to use every single time inside of the vue application that you want to build a list of components out.

We can interpret this directive right here as meaning for every single object or for every single video inside of the list of videos that was provided as a prop to our videos component, make exactly one item component.

So, this statement right here is going to take this item component right here and duplicate it down 20 times because we have 20 separate videos usually inside of this array.

Notice how we also declared this word right here video. So, we said video in videos, because we have video right here that means that inside of this component tag, we can access an individual video and print out some information about it. We'll see an example of that in just a second but for right now, let's save this and go back over to the browser and see what happens.

Now, as a quick reminder, our item component at present, this is a item component is simply printing out an li of videos item.

So, if this V directive works appropriately, then I would expect to see the term videos item printed out 20 times on the screen.

As you we see an error right here before we test out our app. It says ‘Custom elements in iteration require 'v-bind key' directives’.

So, you need the v-bind:key in a v-for because it needs to differentiate each component rendered, in case of data changing. You need to use the both, v-bind:key and v-bind:value in your component option, to it work properly. This key value must be unique in iteration.

ID is the property that has unique value in this all iteration. So, I will use id property as key. I am saying v bind key equals video dot id.

Now, I'm going to go back over to my application inside the browser and you'll see that we have 20 items listed out here. I can, of course, refresh the application.

When I refer to the application, you'll notice that that list goes away because when we first load our application up, we have not yet attempted to retrieve any videos whatsoever from the Pixabay API.

So, at present we have zero videos and that's why we see no items listed out here.

But if we do some search that retrieves 20 videos from the Pixabay API and now we can see 20 instances of the item component appear on the screen.

OK, so that is step one of getting a list of components rendered onto the screen.

Let's continue in the next section and we're going to figure out how we can customize the item component to show some information about the particular video that is supposed to represent.

Handling Props with V-For

In the last section, we were able to get a collection of item components to print out on the screen by using the V for directive. The only issue now is that it appears that every item is printing out the hard coded text of simply video item.

So, we need to figure out some way to say that for every video in this list of videos, we want to communicate this video thing right here down to this component so that this component instance knows what video it is supposed to be displaying information about on the screen.

So, in other words, this first list item right here or this first videos item needs to communicate information about the first video that was returned from Pixabay API and then the second one needs to communicate information about these second video and so on down the list.

So, this is another instance where we want to communicate some data from the parent component of videos down to the child component of item.

So, we are going to, again, make use of the props system in Vue to communicate some information from the parent down to the child, a item.

So, we just went through this process but as a quick reminder, to pass some props for the parent down to the child, we first have to add the bind expression to the parent components template.

Now, this time around, that might seem like it's a little bit more challenging here. It's not the same type of setup that we had back inside of our app component right here.

And, how do we get access to, like one individual video to pass down to the item component?

Well, when we write out v-for and then video in videos, this declares a temporary variable that we can use inside of our template of video.

Video right here will represent one video or one object in our array.

So, this represents one video that has been retrieved from Pixabay.

So, I'm going to give myself a little bit of space inside this tag here, because we're going to add in a pretty good amount of configuration to it.

And then underneath the V for Directive, we're going to set up our V bind an expression, by writing out the dash spined colon and then the name of the prop as we want it to show up inside of the child component.

So, the name of the prop that we want to have shown up inside an item component will make simply video just to stay consistent.

So now item component is going to have access to a variable called video.

Then we need to put in our equals in our double quotes and then inside the double quotes, we will specify the piece of data or the variable of sorts from this videos component that needs to be passed down to the child.

So, this is where we're going to get access to this video variable right here.

And again, you can kind of think of this as being like a temporary variable that is available only inside of our template right here. So, we're going to passing video.

At this point, notice that we have absolutely no property tied to our component called video. It does not exist inside of here. There's nothing simply called video.

So, this value of video right here is referencing the current video that we are iterating over inside of our list of videos.

So, this thing right here is basically being shoved to right here.

If we had instead called this thing right here, like ‘item’, we would also have to update this name right here because they are the exact same thing, these two things, the exact same thing.

But again, I want to stick with simply video to have some consistent terminology.

OK, so now that we are passing from some information as a prop from the parent down to the child. We need to make sure that we tell the child component about the props that it should expect to receive as well.

So I'm going to open up my videos item component again and I'll add in our props property.

Remember that this can be either an array of strings that has the names of the properties that this thing should expect to receive, or we can make it an object if we want to do some validation on the prop when it comes in but again, totally up to you.

So, we expect to receive a property from the parent simply called video.

Now, to make sure that this thing is actually working, we're going to go into our item component template will delete the hardcoded text, a video item, and we will replace it with a reference to this video prop that just got passed down to us from the parent.

We will try to print out the tags of the video that we just retrieved. To get the tags of the video, we'll put in our curly braces like so to indicate we want to do some string interpolation and then the tags of the video will be available on video dot tags. So, you have seen this property already when we looked at the request response that came back from Pixabay inside of our browser console.

The video object right here represents a reference to the property that we were passed down into this child.

That video object had that tags property and that tags property had the tags for the video that we just retrieved.

And so, in this case, we are trying to print out just the tags right here.

OK, so this is looking good, let's save this and then we'll test our application side the browser and we'll make sure that we can see the tags up here.

All right.

So back over here.

Well, you'll see right away that, yep, we've got a list of video tags appearing on the screen inside of our list.

And if we wanted to, we could delete the search term and then we could instead search for something else like, say, pen, and then we get a list of video tags appearing related with pen.

So, this is looking pretty good.

Let's continue in the next section.

And we're going to talk about one last thing around this v-for directive that we put together inside the videos component.

So quick break.

We'll take care of that in the next video.

Keys with V-For

In the last section, we made use of the V for directive to render a list of videos items.

You'll notice that we also made a little mistake, or at least I should say that I made a mistake in leaving in the long form syntax for the Vine directive right here.

So, as I said earlier, we want to be very consistent in our project as to whether or not we are using shorthand or long form syntax for all these different directives.

So rather than writing out dash spined, I'm going to shorten this to the shortened syntax of simply colon video like so.

OK, that looks good.

Now, there's one last thing I want to talk about around the V Directive.

You see, any time we use this TV4 directive to build out a list of items, there is one other property that we have to provide to the element where the component that we are making the list out of, and that is a property called the key property.

We define the key property on the component that we are making the list out of by writing Kullen key equals and then double quotes like so this key property is used whenever our list is updated.

In particular, it's used to update our list in a performant fashion.

If you go back over to the list that we've built already, you'll notice that there's really no warning or anything like that inside of our console complaining about a key not being present.

So you don't strictly have to provide a key, but in general, we always should provide a key if possible, because it enhances the performance of rendering our list of items.

When we define this key property right here, we have to provide some value inside these double quotes that is unique and consistent to the individual record that we are rendering.

Usually whenever we are rendering out a list of records of sorts, like in the case of this list of videos, we will use the ID of the record that we are trying to render as the key property.

So the video object right here that we got back from Pixabay has a property called e-TAG associated with it.

This e-TAG property is an ID of sorts and it is unique to every video inside the collection of videos.

So for the Kolinsky property right here, I will reference video dot e-TAG.

So again, this is a property provided by Pixabay.

It is a unique identifier.

And any time we use this key property right here, along with the Vine directive, we want to make sure that the key that we provide is unique and consistent for the record of rendering.

So if I save this file now and then go back over to the browser and refresh the app, you'll notice that everything still works out just fine.

We don't see any additional errors here.

But again, we definitely want to make sure we always provide the key property if we can.

OK, so this is looking pretty good now.

That's pretty much all we have to do for the V for DirecTV.

Let's now move on to our videos item component, where we're going to work on some of the styling of this thing and making sure that it actually renders out, say, not only the title of every video, but maybe a little image as well.

So let's start taking care of some styling in the next section.

Including Bootstrap Styling

We've now done everything we need to do to get our list of videos to appear on the screen, however, the list that is appearing is just the title right here, and it definitely does not look very much like the original mockup that we were looking at for our application.

So in our mockup, we definitely show having a image thumbnail for the video on the left hand side and the actual video title to the right, right next to it.

So in this section, we're going to start doing a little bit of styling throughout our application to make sure that it looks much more like the mockup that we were looking at originally to help us out with styling.

Inside this application, we're going to use the very popular Bootstrap Access Library in this section will first begin by installing the bootstrap library into our project.

And then after that, we can start making use of it to style the project to install bootstrap.

We're going to include a success tag directly from the bootstrap documentation.

So inside of a new browser tab, I'm going to navigate to get Bootstrap Dotcom, which is the homepage for the Bootstrap Project.

Down here on the bottom left hand side, you'll find links for get started or download, we'll click on Get Started and then we are immediately presented with a success or link tag right here that we can include inside of our project.

So I'm going to copy this link tag.

I'll then go back over to my code editor and I need to find the HTML file that is rendered any time we load up inside of our meeting time and load up our application inside the browser.

So inside the public directory we'll find the indexed HTML.

As a reminder, this HTML document right here is loaded up into our browser.

Any time we visit our application inside the head tag, I'm going to paste that link that we just copied like.

So I'll then go back over to the browser.

I'm going to go back to my application and if I refresh the page and then do a search, you'll notice that the font here looks just a little bit different, which is definitely a sign that we just correctly loaded up the bootstrap library.

So now that we've got bootstrap included, let's continue the next section where we're going to start to take advantage of bootstrap to style some different elements of our application.

Styling the Search input

In the last section, we installed the Bootstrap Access Library into our project, we're now going to use some of the access that is included with that to stihler our project and make it look just a little bit more professional than it is right now.

Well, first, begin by working on the search input at the very top.

So we want to make sure that the search input extends the entire length of the screen or at least a significant portion.

And we also want it to be centered on the screen as well.

At present, it's definitely just off to the left hand side.

Doesn't look that great right now.

OK, let's go back over to our code editor, and we're going to start to add in a couple of different styles to make the search input look a little bit better.

The first thing we're going to do is open up our app component right here.

Any time we make use of the bootstrap library, Bootstrap always assumes that our root element has a class of container.

This root class of container is ser is used throughout bootstrap to better style and align different items inside of our project.

So the div right here inside of our app component is one of the absolute root elements inside of our application.

And so I'm going to add in a class of container right here.

I'll save this.

And then if we go back over to our application, you'll immediately see that all the content on our page is more aligned towards the center of the screen.

So just by adding in that class of container, we automatically got a little bit of margin added to the right hand side and left hand sides of our document, which better centers our content on the screen.

So now that our content is a little bit better centered, we'll open up the search input and start adding

some styling for that.

So I'll find my search input component right here.

Then inside of Search input, we're going to make use of that style tag that we had added at the very bottom but never actually used.

And there's one change that we're going to make to the style tag.

Any time that we add some styling inside of a component file, we want to make sure that the success that we add only modifies this template right here.

So if we had some success inside this file, we do not want to accidentally change the style of some other component inside of our application to make sure that this the success that we had inside the style tag right here only affects this individual component.

We're going to say style and then we're going to add on the term scoped to it like so.

So by adding on the word scoped right here, that makes sure that any styling we add inside the style tag is only going to affect this single component.

Then inside of here, we'll start to add a little bit of styling.

So the first thing I'm going to do is to add a rule for the input tag itself.

I'm going to give it a width of seventy five percent so that it stretches across more the screen than we had before.

So we want to have a look of this right here, something where the search input is really looking across the screen in that width rule will take care of that.

Now, we also want to make sure that the search input is centered as well.

So to center that are going to add in a rule for div that's going to select this div right here inside of our template.

And on that div, I'm going to add in a rule of text, Dasch Align Center, which is going to center that input.

And then I'm also going to add in a margin of twenty pixels, which is going to cause that input to kind of stand off against the top address bar up here and also put some spacing against the bottom towards the list down here as well.

So if I now save this file and then flip back over to the browser, I'm going to refresh the page here really quick.

And then when I do, you'll notice that the search input is much better centered on the screen.

And there's also a little bit more margin between the address bar in the search input itself.

So I now search for archery.

You'll also notice that there some margin down here at the bottom as well.

OK, so that's a good place to get started.

Let's now continue in the next section and we'll take care of styling on the actual videos as well.