INDIAN INSTITUTE OF TECHNOLOGY MADRAS

सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Modern Application Development -II
Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Vue-Managing Components

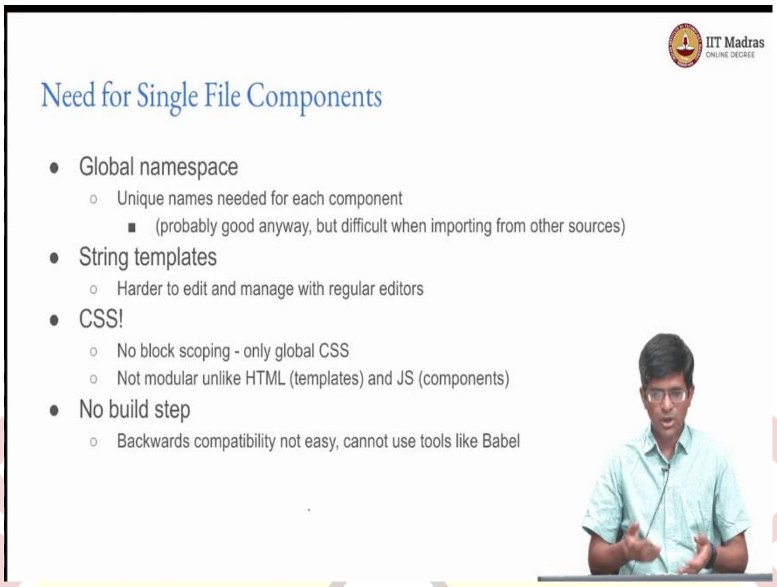
Hello, everyone, welcome to Modern Application Development Part 2.

(Refer Slide Time: 00:14)

Managing Components

So far, we have been looking at fairly simple implementations of view applications where everything could be done just by editing some HTML code and by providing the view components in JavaScript. And in fact, we kept it sufficiently simple for a simple for the main reason that we wanted to focus on the functionality provided by view, rather than the extra tooling that is required in order to get a complex view application working.

(Refer Slide Time: 00:43)



Need for Single File Components

- Global namespace
  - Unique names needed for each component
    - (probably good anyway, but difficult when importing from other sources)
- String templates
  - Harder to edit and manage with regular editors
- CSS!
  - No block scoping - only global CSS
  - Not modular unlike HTML (templates) and JS (components)
- No build step
  - Backwards compatibility not easy, cannot use tools like Babel

But there are some problems that come up there. And in particular, some of these issues such as for example the fact that when I am creating components, I need to just add them in as view dot components. And they go into something called the global namespace which means that I need to have unique names for each component, which is probably a good thing. But, can be problematic when I am trying to import components from libraries written by other people.

Because for one thing, I might have decided to name a component something and then I find later on that somebody else decided to name it the same, which makes it difficult for me to use their libraries. Or even worse, I need to use two different libraries, where the components internally have been named the same.

Now, the top level components may be different, but they may in turn have instantiated some components, which cause problems. So, that is one major issue. The next is the fact that we can use string templates, while defining components, use the backticks and so on. But they are sort of hard to edit. And managing them properly is not straightforward. You need some extra editor support in order to manage them cleanly.

There is a big problem with the CSS, the styling that is used in such components. Because as you would have noticed, in all the view application that we had the styling had to be applied globally, it had to be applied at the top level HTML file. You cannot do what is called block scoping where you basically say that the CSS that I am doing is only for one particular component. It is not straightforward to do that at least.

So this part, suddenly, unlike the HTML, and the JavaScript, which can be separated out into files of their own is not modular. Now, there is no build step, what I mean by a build step is a compiler. So when you write a C program, for example, you write the program in an editor, save it, you first compile it using GCC or C plus plus, and then you run it. Instead, in a Python program, you write the Python program, and you just call Python and the file name and it runs.

So Python effectively what it is doing is, it is interpreting the code on the fly. It reads it in line by line as it is going through and runs it. Of course, it does it fast enough that, for most purposes, it looks like it is instantaneous. But there are plenty of cases where a compiled program in C would be significantly faster orders of magnitude faster than what you can get in Python.

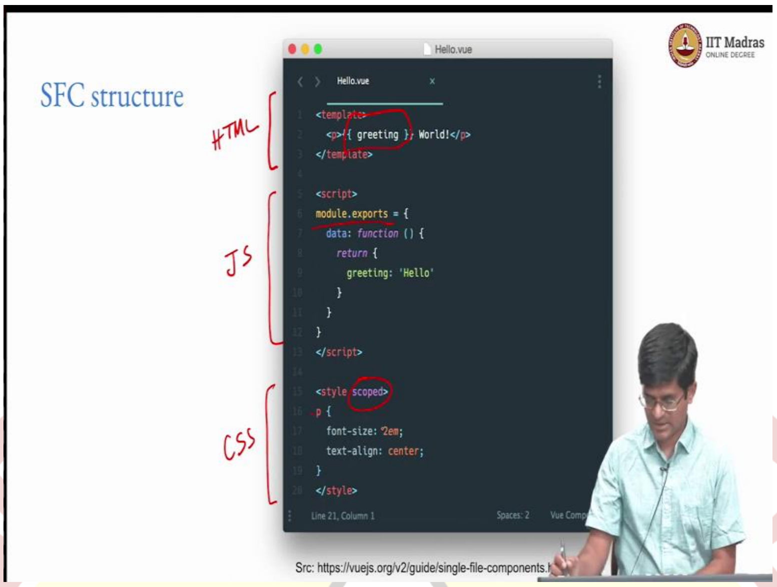
Now, the problem is that C program that builds step over there can get complicated for large programs. Which means that as far as possible, what we were trying to do so far in using view was to avoid any kind of build step. We just want to be able to use view as a library in JavaScript without necessarily getting into complicated compilation steps. So that is good, that is one good thing about view it can be used that way.

The problem is, there are certain cases where it becomes very hard to really do something useful with that. And an example is when you want to maintain backwards compatibility, because you might have written your JavaScript code in some kind of specific way that supports all the ES 2015 features. ESX features or ES 2021 features. And you find that the browser's are not up to date.

So, what do you do? If you had a compiler and there are some very nice compilers that take care of this for you. They basically create extra output files that say that if the browser does not have support for these kinds of features, like they will check the browser to see what kind of features it has. And if it does not have support for specific kinds of features, it loads something called a polyfill.

And a polyfill is essentially some extra piece of JavaScript code that provides functionality that was not natively there in the browser. So, not having a build step is a good thing but, it can also have its drawbacks when you are trying to build more complicated applications.

(Refer Slide Time: 05:00)



SFC structure

Hello.vue

```
<template>
  <p>{{ greeting }} World!</p>
</template>

<script>
module.exports = {
  data: function () {
    return {
      greeting: 'Hello'
    }
  }
}
</script>

<style scoped>
p {
  font-size: 2em;
  text-align: center;
}
</style>
```

Line 21, Column 1 Spaces: 2 Vue Comp

Src: https://vuejs.org/v2/guide/single-file-components.h

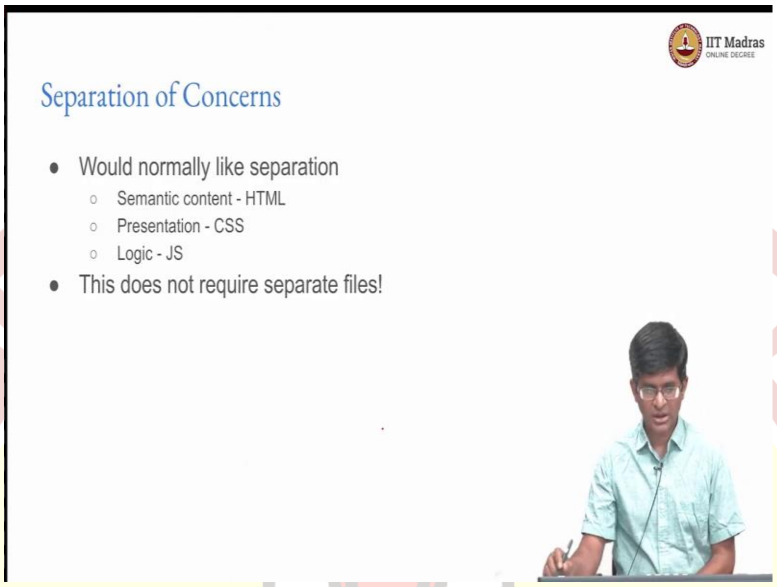
So, instead view define something called a single file component. And the SFC has the structure as shown over here. What you can see is that there, the entire code has sort of been broken up into segments. There is a template out here, which if you look at it is pure HTML, except for the fact that it also has the interpolated variables, the double curly brackets. So, you can still use those. But the template part of it by itself can mostly be treated as HTML, which means that you can have your component where the overall structure what it is rendered, like what it looks like, can be handled in fairly normal looking HTML.

There is another part of it, which is, so this first part is the HTML. The second part is basically the JavaScript; the logic of the code. And over here, what we do is we basically define the explicit exports. And this entire thing is sort of, given as a default export of this entire module.

Now, we are using modules, we are using exports, we are going to be using inputs, all of these are the ESX functions, ECMAScript 6 functions, which means that they do not really work with old versions of JavaScript. Having said that, in 2022, pretty much any browser, or Node JS supports all of these things. So for the most part, we should not be worrying too much about, whether the browser supports it or not.

And in case you have a browser that does not support it, there are compilers, like Babel, that will usually help you with it. And of course the last part out here is the styling CSS. And as you can see over here, that styling is scoped. Meaning that it essentially says the scope of the styling is only to this component that we have here. Which means that the paragraph that is present in this component will have this font size of two M and the text alignment of center, it will be center aligned. But, other paragraphs in other parts that are not inside this component will not look the same. Now, this is a nice way of sort of representing all the different parts of a component.

(Refer Slide Time: 07:25)



Separation of Concerns

- Would normally like separation
  - Semantic content - HTML
  - Presentation - CSS
  - Logic - JS
- This does not require separate files!

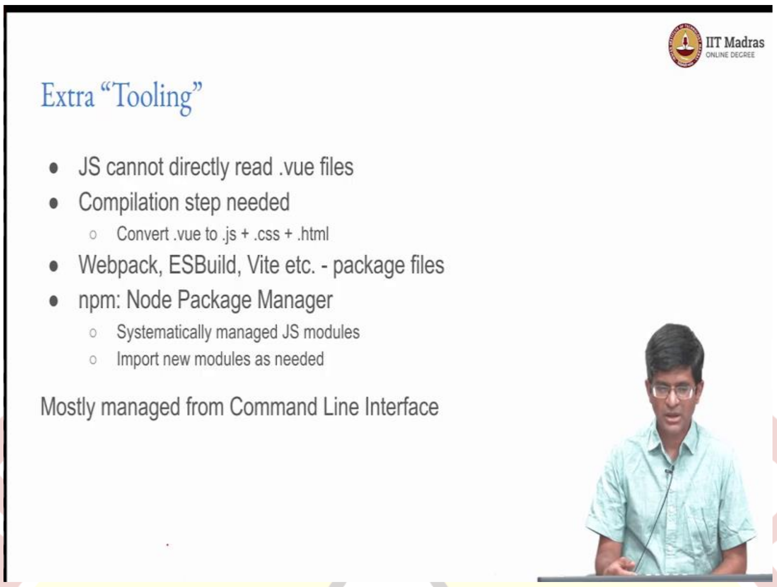
There is HTML, which is responsible for the so called semantic content. Then there is the presentation, which is handled by CSS, the styling. And there is the logic which is handled by the JavaScript. So, one question that could arise over here is what happened to separation of concerns.

Because that is a term we have been sort of repeatedly saying, from the beginning of application development, we want frontend versus backend, we want HTML versus CSS versus JavaScript versus the back end Python. We want to sort of keep, you know, each of these components, or each of these languages should do one thing well.

Now, the point is that a single file component does not necessarily mean that we are trying to bundle everything together. You could if you wanted to separate out the HTML, CSS and JavaScript of a component into separate files, but it is really unnecessary, because if you look at it from a logical point of view, they belong together.

And inside that file, they have been very cleanly separated out. The template, the script and the style are three parts of the file that are very clearly visible and each one performs its own task. So, as long as we use that approach, it means that you know, this entire component definition within view can be done in a very modular manner. You can create modules that can be reused nicely.

(Refer Slide Time: 08:50)



Extra "Tooling"

- JS cannot directly read .vue files
- Compilation step needed
  - Convert .vue to .js + .css + .html
- Webpack, ESBuild, Vite etc. - package files
- npm: Node Package Manager
  - Systematically managed JS modules
  - Import new modules as needed

Mostly managed from Command Line Interface

The problem is that JavaScript cannot directly read these dot vue files. So usually, they are given the extension dot vue, because they are not directly HTML, they are not directly JavaScript and they are not directly CSS, so what do you do? In order to really make sense of them, a compilation step is needed which explicitly goes through separate or separates out these components and converts a dot vue file into separate JS, CSS and HTML files.

And there are a number of different sort of build tools or packaging tools, write them many names, Webpack ESB to roll up with, etcetera, which basically take care of packing or packaging the files in such a way that you know, they can be presented for downloading and are present in, put on a server so that people can access it as an application.

Now, in addition to this, all of this means that, you also need to bring in something called NPM, the node package manager. Now, is this absolutely necessary? It is very difficult to work without it. And what happens in NPM is that it systematically manages JavaScript modules and allows you to import new modules as needed. And, for example, you could have a test module or something else that just gets imported.

One issue over here is it is mostly managed from the command line interface. And the second issue that can sometimes you know that some people find objectionable is adding one module in NPM, usually pulls in a whole bunch of different other modules as well, because there are dependencies.

And before you know what is happening, the node modules that you have, suddenly there are something like 800 packages occupying 100 megabytes of disk space, all for something which looks pretty much like a HelloWorld application. Of course, the final application may not consume that much. Because what gets seen by the end user is only the HTML and JavaScript that needs to be explicitly loaded. And those can be optimized fairly well.

But in the meantime, the temporary step is that, you end up taking quite a hit on the time required for building as well as the disk space required for building. There is one additional, fairly serious issue that off late has become seen more and more common with NPM and the approach that it takes for package management, which is what is called a supply chain attack.

There have been multiple instances where one particular JavaScript package somewhere managed on GitHub or something of that sort has been replaced by malicious code. In a couple of instances recently, the code by itself was not malicious, but it broke functionality and ended up, it was just something which basically said, you need to pay for this or something like that.

And the result of that is that there are some potentially very serious security issues involved with how this entire package management works. It is not that such things do not exist with other languages or with other build systems, but others deal with it separately. Node in particular, has come in for a lot of criticism because of this aspect that it has not been taking it sort of seriously enough in some sense.

But all that is later, it comes at a point where you are much more comfortable with working with the command line. The point is that if you want to write single file components and these dot View vue files, you will need to get some familiarity with the command line interface and how to build such systems.