Wrapping JavaScript Libraries with C#

So far, we've learned how to call JavaScript code from C# and how to call C# code from JavaScript. This is very useful when we need to write some custom JavaScript code. But JavaScript is huge, and there are already many useful and popular libraries out there. Some of them have been around for years and have been thoroughly tested by the community. It would be crazy to rewrite everything in Blazor WebAssembly, so we're going to learn a way to wrap the existing JavaScript libraries and use them in our Blazor applications. For this purpose, we're going to use the Toastr JavaScript library. Toastr is an excellent Javascript library we can use to show non-blocking notifications in our application. We'll leave a link below the video. We've already used Blazored.Toast for this purpose, but let's say we've been using Toastr for years, and we don't really want to switch now. Let's see how we can register it, pass the configuration parameters and use it in our C# code. We're going to separate the logic to another class library. So let's start by adding a new class library project to our solution. Let's go to add a new project… and then select Razor Class Library… click next… give it a name... "BlazorProducts.Client.Toastr" and then click Create… check quickly that we're using the right SDK version… and click Create once again... After the creation, let's remove all the files from the wwwroot folder, and ExampleJsInterop.cs and Component1.razor files too. To include the Toastr JavaScript library in our Razor class library, we have to add some JavaScript and style files to the wwwroot of our BlazorProducts.Client.Toastr project. You can find the files below the video or in the resources folder on this module branch. So let's add a jquery library which is needed for the toastr to work, and also styles and the main toastr library. Now, let's reference the BlazorProducts.Client.Toastr project from our main project... and let's import these files inside the index.html file... Let's import the styles first… and then jQuery and toastr scripts... To continue, we are going to create ToastrWrapper.razor and ToastrWrapper.razor.cs files in the Pages folder. For now, we are just going to add a route to the .razor file. Now let's add the navigation menu item so we can navigate to this page… Excellent. Now, let's go back to the BlazorProducts.Client.Toastr project and let's create a new JavaScript file in the wwwroot folder… and name it toastrWrapper.js… In this file, we're going to expose the toastrWrapper object globally and create a single showToastrInfo function inside it. In this function, we’re just going to call the info function of the toastr library to show the info notification with some dummy text. Okay, let’s get back to the index.html file and import this Javascript file too... Good. Now let’s get down to business. We need to be able to call this function from our C# code, so let’s put the knowledge we have to use. Let’s create a new folder called “Services'' in BlazorProducts.Client.Toastr project… and then add a new service to it… and name it Toastr service. We want this service to call the JavaScript function that we’ve defined in the toastrWrapper. So let’s add a private field of the IJSRuntime type… a constructor to initialize it… and then let’s create a wrapper method that just calls the showToastrInfo function from our toastrWrapper JavaScript code. There’s nothing new in this piece of code, we’re just using the mechanism that we’ve already covered in this module. Next on, we can create a ServiceCollection extension class… in order to be able to add our newly created service to the services collection in the main project. This way we can easily enable or disable our library directly in the main project. So let’s add a new class… name it ServiceCollectionExtensions… make it static… and then add one public static method AddBlazorToastr… which just adds ToastrService as a scoped service to the collection of our services. As easy as that. Now we can go to the Program class and simply enable our service by calling the AddBlazorToastr method… That’s all it takes. Currently, we have a Toastr library wrapped inside a C# service, and the library users may use it in their C# code. Let’s see how they can do that. Let’s start by modifying the ToastrWrapper.razor.cs file. First, we’re going to inject the ToastrService we’ve made… and then create a private method called ShowToastrInfo… that calls the ShowInfoMessage from our ToastrService. And let’s add some markup to our ToastrWrapper razor template file as well… We’re going to add a row with two columns as always… Add a header text… and then add a button that calls ShowToastrInfo method on the click event… Let’s give it some text… Excellent. Now we can run our application to test out what we’ve made so far. Let’s navigate to our toastr wrapper menu item… and then let’s click on the button… and what do you know... our info message toast shows up. Everything’s working as expected so far. But we can do more to improve on this solution. The first thing we want to do is to extend

our toast with some custom messages and options. We don't have much use for the hardcoded toast. So let's go to our toastrWrapper JavaScript file and extend the existing showToastrInfo message with some parameters… a message parameter and the options parameter… We'll assign the options to the toastr.options so we can configure it differently… and we'll show a message as we did before... This way we can have our custom toast messages. Now, since we've changed the JavaScript function, we need to change the invocation in our service as well… so let's extend the ShowInfoMessage method with the message and the options parameters… and use them in our invocation. Now we can go to our main project and modify our ShowToastInfo method in the ToastrWrapper class to reflect these changes. We want to add a message first… and then add some options… We're going to configure our toast a bit… give it a close button… configure duration… and show/hide animations… and we want to show it at the bottom right… Now we can send the message and the options to our service. Good. Now, let's start the app... navigate to the component... and press the button... And there we go. We can see our notification on the bottom-right side, with the close button and also with some slideUp and slideDown animations while opening and closing. Awesome. But we can make it even better. Even though our solution looks good on the surface… we can do some things to improve it even more. We want to avoid hardcoded strings as much as we can, so we'll create some enumerations to help the library users. We can make our options parameter strongly typed, thus providing IntelliSense for all kinds of options for the library users.
So, let's start by creating a new Enumerations folder in the BlazorProducts.Client.Toastr project, and add a new ToastrShowMethod enumeration inside…We'll add the FadeIn, and the SlideDown options… and we'll decorate them with the description attribute... By using the [Description] attribute we can map our enum names to those that the toastr library expects... For example, toastr expects the fadeIn with the lower capital letter… and we want to call it FadeIn with the upper capital letter. We want to use enumerations because that way we get some nice IntelliSense and we avoid potential mistakes that can happen when using the strings all over the place. Now, let's create two more enumerations. The ToastrHideMethod enumeration... And another one for the position:

That’s it. Of course, there are many other options for the Toastr library, but these will be enough for now. You can check which options are available on the toastr library website. There is a small catch here. Currently, System.Text.Json, the library in charge of the serialization can’t convert the Description attribute out of the box, so we have to provide a custom converter to help with the process. The converter is not that trivial since it uses some not so intuitive code and some reflection to get the job done. But stay with us… it’s a one time thing you can reuse it later on in your projects. That said, let’s create a new CustomConverters folder in the BlazorProducts.Client.Toastr project... and a new class inside that folder… let’s name it CustomEnumDescriptionConverter... We’ll immediately make it generic, and inherit from JsonConverter class, and restrict it to the Enum type. Once we’ve done that, let’s implement it… and we can see that we need to override two methods… We’re going to leave the Read method as is, since we don’t need to implement it in our case. In the Write method, we are going to use reflection to extract the type of the T parameter, and the GetField method to extract the field with a specified name of type FieldInfo. Then, we use the reflection some more with the GetCustomAttribute method to extract the attribute of the provided type and cast it into that type – in this case, the DescritpionAttribute type. Finally, we are going to use the WriteStringValue method to write the value of the Description attribute. We use the null conditional operator (?) to prevent any errors if the description is null, or in other words, if the Description attribute is not provided. Now, let’s create one more class in the BlazorProducts.Client.Toastr project… And let’s add some properties to it like… position… hide method… show method… close button… and hide duration… This covers all of our options we need. Now we have to help the serializer with the property names by using the JsonPropertyName attribute… and then we need to add our custom converter to the position… hide method… and show method… this will help us properly convert our description attributes. With this in place, we can modify the ShowToastrInfo method and use the strongly-typed options… we don’t need to use anonymous objects anymore… we can use ToastrOptions class… and then we can replace the hardcoded strings with enums… for our HideMethod… ShowMethod… and Position... Excellent. Now let’s test our library once more… and it works as expected. While nothing has changed visually, we’ve made our library much more user-friendly, and we can easily extend the options if we need to. This is only one example of creating the JavaScript library wrappers. There are plenty of nice JavaScript libraries out there, so make sure to play around a bit and try to wrap your library of choice. You have all the tools you need now, so make sure to use them well. This wraps it up for this module. Let’s recap what we’ve learned so far.