Using Blazored Toast to Improve the UX

Now that we have a working form, the only thing left to do is to improve the user experience. Nobody likes a mute form, and we would like to see if the form submit action succeeded or failed. In order to implement the notifications, we'll use a component called Toast. Bootstrap has its own toast component, but this time we'll go with the Toast that is implemented specifically with the Blazor in mind. This component comes as a NuGet package and we can install it through the package manager console with Install-Package Blazored.Toast… we'll make sure to select the main project and then just install the package. Okay, the first thing we need to do is to register the Blazor Toast Service in the Program class… so let's add it to the services with the AddBlazoredToast() extension method… And let's make the service available through the _Imports.razor file. Great. Now we can proceed to actually create and configure the component. To do that, we need to go to the MainLayout.razor file and add the <BlazoredToasts> component… This is also the place to do our initial configuration and we'll add a few default values. First, we want to add the position… with Position="ToastPosition.TopRight"... and since the compiler is getting angry at us, we'll import the Blazored.Toast.Configuration in the Imports file… Then, we want to set the timeout to 10 seconds, that's the delay after which the toast will disappear… and the default icon type to FontAwesome... We also want to set the SuccessIcon class… and ErrorIcon class… to make our toasts a bit easier on the eyes. That's about it. The default value for the Position attribute is actually the top right, but we'll leave it since you might want to change the position of your toasts. There are many more options available for configuring the toast component, but we're using the default styles, so this works for us. We'll link the documentation below the video so you can check out what else you can configure at this point. The classes we've used for the icons are actually from the open iconic, which is the open-source set of icons, and we're using them because they come as a part of the blazored toasts styles. We'll link the icons right below the video. Speaking of stylesheets, we still haven't added a blazored toast stylesheet, so let's go to the index.html and add it… Okay, now we're all set, so let's go to the CreateProduct class… and modify the Create method… First, we need to inject the toast service… so let's add a new property of the type IToastService… and then import the missing dependencies… and let's mark it with the Inject attribute. Then, let's go to the Create method… and call the success toast since we're sure we'll get the success result at this point. We can do that by calling the ShowSuccess method of the toast service… and add a simple message to inform the user about the action success… After that, we want to clear the form, in case the user wants to add more products. To do that, we can just set the product to the new Product instance. Remember, our interceptor handles the bad request response, so we can go to the interceptor to add the error toast. Okay, since the interceptor is a service, we need to inject our toast service through the constructor… so let's add one more parameter of the IToastService… and one more private readonly field … and then just assign it in the constructor… Now we can go to the BadRequest case, and add the error toast with the _toastService.ShowError method, and pass the message as the argument. Excellent. Let's run the application and test this out… Navigate to the create product form… and then fill in the data quickly… Okay, let's submit the form… and there's our success toast… As you can see the form has been cleared immediately like we've wanted to… That's great! Now let's head back to the ProductController on the server-side… and let's force the BadRequest result… by commenting out everything but the BadResult…and adding a simple message. Now we can run the server again… And try filling in the form one more time… This time we can use the random data since we'll get a bad result anyways… And there we go. This time, we've left the data in the form, if the user wants to try and correct the fields before resubmitting the form again. After this, we can head back to the server app and uncomment the code in the action… Before we finish this module, we have to fix one thing. With the current implementation, as soon as we successfully create a product, our form gets cleared out, but the Create button is still enabled and our form is not validating input fields. Well, we should fix that. So, let's navigate to the CreateProduct class… and modify the Create method by adding a new event to the OnValidationStateChanged event handler… and calling the NotifyValidationStateChanged() method, to explicitly call the ValidationChanged event. Then let's generate our missing method… Inside it, we want to disable the button by setting the formInvalid field to true. Also, we remove the HandleFieldChanged event from the handler, create a new instance of EditContext, add again the HanldeFieldChanged event to the handler, and finally remove this method from the OnValidationStateChanged handler. Lastly, we have to extend the Dispose method and remove both events from their handlers. Excellent. With this in place, our button is going to be disabled as soon as we create a new product, and also if we start creating a new one, our validation will work as it is supposed to. That's it, we have a form for creating new products in our application. In the next module, we'll see how we can replace the image URL field with the real file upload field and upload it directly.