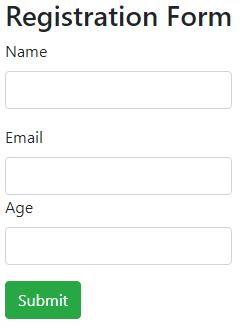
# Lab: Vuelidate Lab

## Html Form

We are provided with html registration form, which consist of 3 fields: Name, Age and Email. We must use Vuelidate to validate the form. You are free to implement whatever logic you want.



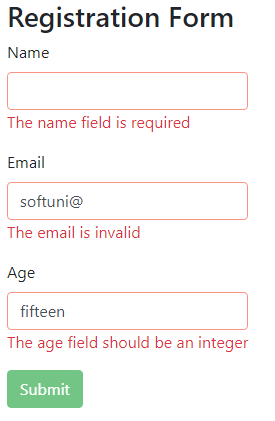
That's how our form should look like.  
The html is pretty simple you have 3 input fields and the labels for them. At the end we have submit button to submit the form.

### Validation rules

Let's start with basic validation. The validation should be executed upon submitting and when a field is selected.  
First it's important to set the rules that we are going to follow.

* All the fields are required.
* Name: Name should be at least 3 characters long.
* Email: Should be valid email.
* Age: Age must be in range of 12 to 120 and should be numeric.

The errors will be rendered under the input fields, at the end of the lab this would look something like this.

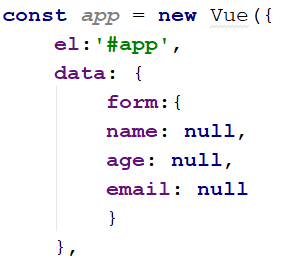


## The Vue instance

We need the **el:"{element id}"** to bind the data of the form   
after that declare **data: {}** with the given properties

* **name, age, email:** all these properties will be null by default and they will hold the information of the input fields
* **form:** this object will encapsulate all of our properties to make the whole form validation a bit more easy

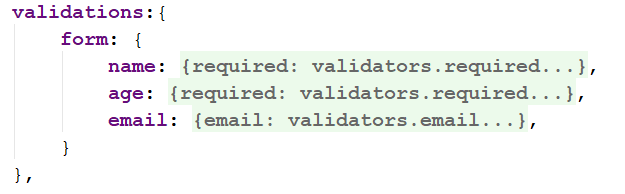
It should look something like this:



## Implement validations

Now we must implement the rules using **Vuelidate** **validations,** we are going to use mostly the build in validators.

Let start with the form, it should have 3 properties name, age, email so when one of these properties is not valid the whole form will be invalid as well.

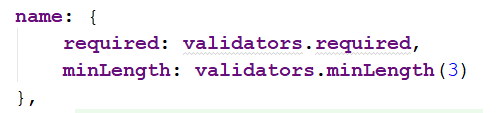


Now lets get step by step through the properties.

### Name

The name is required and at least 3 characters long so using built in validators would be pretty easy task for us.  
Use the following validators

* **validators.required:** this validator will ensure that our field is filled
* **validators.minLength(int):** this validator will secure the length of the name



### **Age**

The age should be required, integer and in the range of 12 to 120.

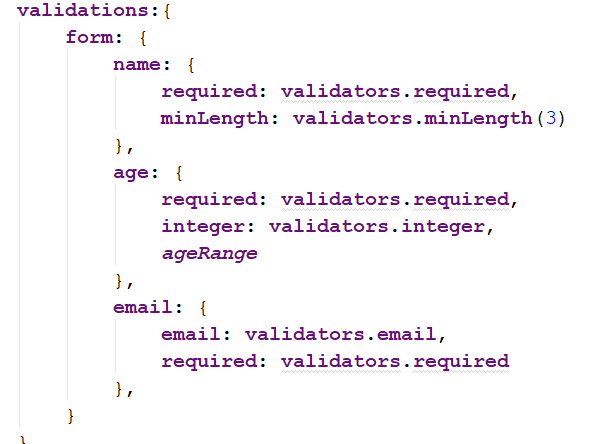
First let us make our task a bit easier, declare a constant named for example **ageRange** and set **validators.between(12, 120).**Now whenever we want to use the age range we will use the constant and here is the code:



You probably already know how to implement the validators for the age so try it by yourself, if you meet any difficulties the example of the validations will be provided at the end of this task.

### **Email**

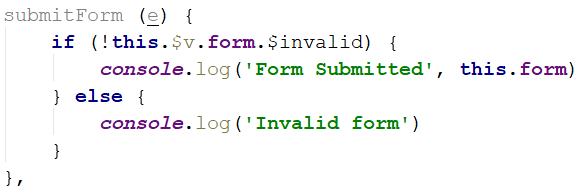
The email should be required and valid. You can use **validators.email** to ensure that the email is valid.  
Nothing special here so try to implement the code by yourself.

As promised here is the code of the validations ☺   


## Methods

Now we will need to implement a submit handler using Vuelidate validations is making this extremely easy.  
All we need to do is to check if the form is valid, but how do we do that, that's how:  
**"$v.form.$invalid"** this property will be true if the form is invalid and false if the form is valid.

This time we will use the console just to be sure if the form is submitted or not. So the validation looks like this

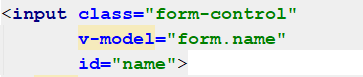


The form has **@submit.prevent** so this will prevent the page of reloading.

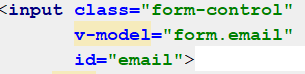
## Changes to the page

Now to bind the data we should include the **v-model** directive to bind the input with the data.

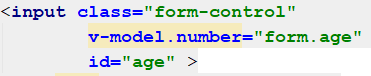
* Name:



* Email:



* Age: Include **.number** to parse age to number



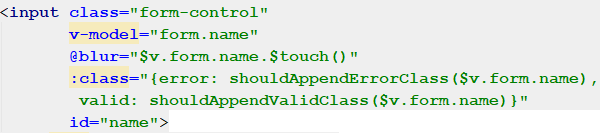
Now when trying to submit the form it will be only submitted when our rules are satisfied, you can see this in the console but we want the errors  
to be displayed under every field.

That’s our next task.

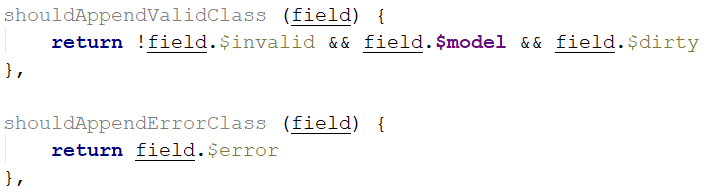
## Warning fields

Now we are going to make our fields to shine in red when they are not matching our validation.

First let's implement a way to check if the field was touched.  
Here we are going to provide you the code for one of the field and explain it step by step what it is doing.



When the field is selected **@blur** this will invoke **$touch()** method and this will mark the name as dirty.   
Using :**class**  the class **error** will be added when the method **shouldAppendErrorClass** with the passed on field returns true,  
and it ind its all the same ue, and its the passed on field returns true, and the validep by step what it is doing.en we can render ous all the same with the **valid** class.

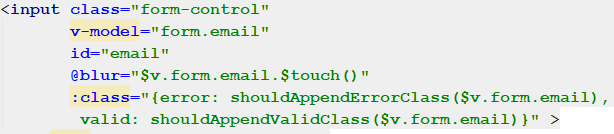
Now let's head back to the **methods** and implement the new logic that is required for our view to work.  


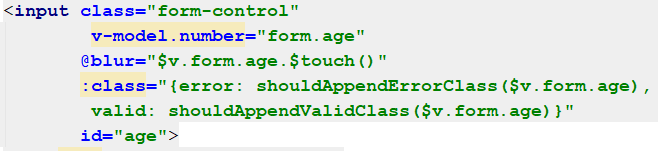
Both methods are fairly simple.  
The **shouldAppendValidClass()** method checks if the field is valid, not null and if it's dirty (has been selected once).

The **shouldAppendErrorClass ()** method on the other hand returns $error value which is

* true if the field is invalid and is dirty
* false if the field is not dirty
* false if the field is valid and dirty

Implement the same logic for every single input tag and test the form.

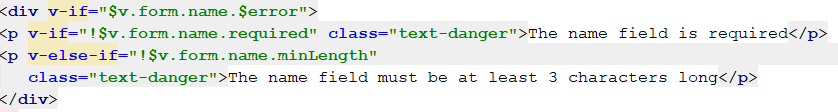




## Errors Rendering

Now to fully implement the errors rendering we might think for a second, how we want to render them.  
It would be a cool feature if the errors are rendered only if the field was already selected. And we don’t need to render all the errors at once.

So to do this we can have a div which will be rendered based on the field **$error.** After the div is rendered we can use **v-if** and **v-else-if** combined with the validators to control how the messages will be rendered.



It looks very simple and familiar, doesn’t it?   
The logic is exactly the same with all of the fields.  
You probably are already pissed off all these hints so you might try to do it by yourself.