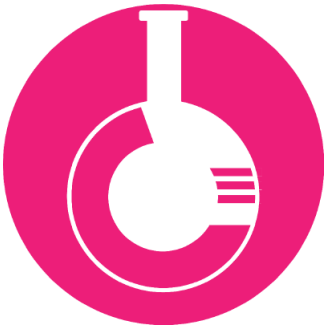
COC.JS Nuxt-iView



Visit Us On Github

What is COC.JS Nuxt-iView

VUE.JS assistant based on COC core engine and designed to work with (Nuxt, iView) environment

- Coc is providing the developer many helpers for data structures, validation, events, and filters.
- You will also have extra 20 components designed to work with iView UI.
- Coc Forms components are designed to build a complex forms with minimal code and more features

Install with npm

```
npm install coc-nuxt-iview
```

Get Started

After installing from npm, create `coc.js` file in `/plugins` directory. in `coc.js` add the following

```
import COC from 'coc-nuxt-iview'
import moment from 'moment'
import 'moment-timezone'
import lodash from 'lodash'
import pkg from '~/package'
import Vue from 'vue'
export default ({ app: { router, $axios }, env }, inject) => {
  // Init COC
  COC.Init({
    Vue,
    lodash,
    moment,
    axios: $axios
  })
  // Config app data
  COC.Config.Meta({
    name: pkg.name,
    brandName: 'Techno Service',
    version: pkg.version,
    repository: pkg.repository,
    author: pkg.author,
    mode: env.mode,
    logo: {
      primary: '/logo.jpg',
      invert: '/invert.jpg'
    }
  })
  inject('coc', COC)
}
```

Afer adding cocjs file in your plugins, add the file in your plugins in `nuxt.config.js` Then you can add the default COC theme by adding `coc-nuxt-iview/dist/index.css` in your `nuxt.config.js` under the `css` attribute. You can also use COC-Theme-Builder just clone the repo and start customizing your own

COC theme.

UI Design Concepts

The plugin is using iView as a UI framework, so mainly we are implementing Ant-Design concept, but also we are using some parts from other packages such as Materialize so you can have better Grid System, Color Palates, and Shadows, you will also find a Color Palates and Button Classes from Bulma. **External resources** - MaterializeCss - Grid System - MaterializeCss - Colors - MaterializeCss - Shadow - AnimateCss

Resources By COC - Typography Styles - Borders Styling and Colors - Bounds and House Keepers

COC Custom Components

Based on iView. - Forms - Input - Forms - Select - Forms - Radio/Checkbox - Forms - Datepicker - Forms - Button

COC Pure - Coc-Axios - Coc-ShowKeys - Coc-Collapse - Coc-Form Item - Coc-Form - Coc-Watch My Window - Coc-Option - Coc-Select - Coc-Input - Coc-Input Field - Coc-Layout Master Splited - Coc-Main Master - Coc-Master Footer - Coc-Master Nav - Coc-Docker -

Check our full template

COCJS Ready Template for Nuxt-iView-COC

CSMA (Coc Standard model API)

Coc Components are using a powerful standard for modeling the components every `v-model` will give you an object that includes three main sections

- val
- control
- meta

Every CSMA has a `control` section where you can access all the CSMA controllers from. Accessing this controllers locally is being done by a very simple calls using CSMA API, for example here's how to use update controller
`myModel.control.update('myVal')`.
`myModel.control.reset()`
`myModel.control.copy()`
`myModel.control.clear()`
`myModel.control.select()`
`myModel.control.focus()`
`myModel.control.blur()`
`myModel.control.meta('isFired')`

You can also add a callback, eg :
`myModel.control.reset().then(myCallback)` or `myModel.control.update('foo').then(myCallback)`

For calling CSMA controllers remotely we're using VUE custom events.

The API structure is very simple, but also you need to be explicit about your scopes and the controllers you need, you need to tell `COC` what scope do you need to respond, the controller you need to access, and the arguments that you need to pass it 'credentials'.
Here's the general rule for using CSMA events for this component.

```
$nuxt.$emit('COCFormController', {
scope : ['myScope'] ,
controller : 'foo' ,
credentials : 'bar'
})
```

The previous API is the general case for any CSMA, its recommended to be explicit about your remote calls anyway. for example :

```
$nuxt.$emit('COCFormController',{
type : 'input' ,
scope : ['myScope'] ,
controller : 'foo' ,
credentials : 'bar'
})
```

If you didn't mention a specific type, all the components will react to this event, which is not recommended, as it will be more load and also you may call an undefined controllers for some components, in this case `COC` will handle it for you.
Passing Callbacks in remote events is the same as local usage, all you need to do is to add the callback attribute to your call, and you're all set!

```
$nuxt.$emit('COCFormController' ,{
type : 'input' ,
scope : ['myScope'] ,
controller : 'foo' ,
credentials : 'bar' ,
callback : myCallback
})
```

OR

```
$nuxt.$emit('COCFormController', {
type : 'input' ,
scope : ['myScope'] ,
controller : 'foo' ,
credentials : 'bar' ,
callback(){ //some code }
})
```

Note : COC is validating the events arguments for you, so incase you have orderd a controller that does not exist COC will inform you using console warnings.

Here's a real example that you can try right now, in the next lines we'll be calling update controller to change the value of the upper COC EL INPUT, just copy it and paste it in your console to check it out!

```
$nuxt.$emit('COCFormController', { type : 'input' ,
scope : ['form'] ,
controller : 'update' ,
credentials : 'foo' ,
callback(){ alert('Hello World') }
})`
```

Ask For Meta API is a kind of conversations between components, or even between the root and a child component, if you need to get some meta of a specific component remotly, you can ask it, and it will reply with required meta if exists, otherwise `COC` will handle it for you and also warn you about it in your console.
Setup
Asking for meta is a regular controller, all you need to do is to select `meta` controller and pass your meant attribute as a `credentials` for example

```
$nuxt.$emit('COCFormController' ,  {
type : 'input' ,
scope : ['form'] ,
controller : 'meta' ,
credentials : 'fired'
})
```

After this, the component will emit an event back, so you need to recieve the response and handle it to get your info.
For Example

```
$nuxt.$on('COCFormMeta' ,(payloads)=>{
  console.log(payloads)
})
```

So here's how to recieve it, now but this code in your console, and then ask for meta using the previous code in your console, and check the result.
, simply the field name will be under `meta` attribute, and the value will be under `credentials` .
the payloads will also carry the meant scope as you can handle when to react on those components using them.

Validator

What is COC.JS Validator

Javascript Validator based on COC core engine.

- Built with 29 rules ready.
- Able to work with your own custom validator.
- Easy and powerful Status Delivery

Install individually with npm

```
npm install coc-validator
```

`coc-nuxt-iview` comes with coc-validator shipped and integrated already.

Get Started

After installing from npm, all you need to do then is importing validator class and you are ready!

```
import Validator from 'coc-validator'
```

Validator

Coc Validator is a validation class that was made to validate user input with a ready made validators that will cover most of common cases, you can also add a custom rules and custom error delivery messages.

Rules

```
[
   "HasValue",                 //Arguments: none
   "SameAs",                   //Arguments: none
   "IsString",                 //Arguments: none
   "IsEmail",                  //Arguments: none
   "IsNumeric",                //Arguments: none
   "IsNumericString",          //Arguments: none
   "IsDateString",             //Arguments: none
   "IsArray",                  //Arguments: none
   "IsObject",                 //Arguments: none
   "IsEvenNumber",             //Arguments: none
   "IsOddNumber",              //Arguments: none
   "NumberGreaterThan",        //Arguments: Number min
   "NumberLessThan",           //Arguments: Number max
   "NumberBetween",            //Arguments: Object { min: Number, Max: Number }
   "MaxDate",                  //Arguments: MomentInstance min
   "MinDate",                  //Arguments: MomentInstance max
   "DateBetween",              //Arguments: Object { MomentInstance min, MomentInstance Max }
   "MatchesRegex",             //Arguments: Regex regex
   "MinLength",                //Arguments: Number min
   "MaxLength",                //Arguments: Number max
   "LengthBetween",            //Arguments: Object { min: Number, Max: Number }
   "MinArrayLength",           //Arguments: Number min
   "MaxArrayLength",           //Arguments: Number max
   "Each",                     //Arguments: Rules will be applied on each element in the array
   "Keys",                     //Arguments: Object includes the desired keys to be validated, each of them has set of rules
   "Remote",                   //Arguments: Object args than includes:
                                 //Object options (axios args eg: { url: '/foo', method: 'get'})
                                 //Function callback to be excuted on resolvation, consider res as a parameter
                                 //AxiosInstance agent --optional
                                 // Function catch to be excuted on fail --optional
   "PreConditions",           //Arguments: Array validators
                                 //Each of them should return true , or error messagee or false
]
```

Rules, Args and Options

```
// Standard
const standardOptions = {
  HasValue: {
    args: true,
    active: true,
    message: 'This field is required',
    icon: 'fa fa-error',
  }
}

// Only the needed attributes
const onlyIfNeeded = {
  HasValue: { icon: 'fa fa-error'},
  MaxLength: {
      args: 3,
      message: 'Whoops!, length cant pass |*args*|'
    }
}

// Quick
const quickOptions = { HasValue: true, MaxLength: 3 }
```

Examples

```
import Validator from 'coc-validator'

const person = { name: 'Jhon Doe', age: 24, sports: [ 'basket', 'ping-pong' ] }

const rules = {
   HasValue: true,
   IsObject: true,
   Keys: {
     name: { HasValue : true },
     age: { IsNumeric: true, NumberLessThan: 20 }
   }
}

const v = new Validator( person, rules )
const result = await v.Run()
```

The expected result is

```
{
  attemp:  0
  attemps:  0
  code:  12
  error:  "NumberLessThan"
  icon:  "ivu-icon ivu-icon-ios-alert-outline"
  instance:  "coc-validator"
  message:  "This number must be less than 20"
  path:  (2) ["root",  "age"]
  val:  24
  valid:  false
 }
```

The same demo with custom error messages

```
import Validator from 'coc-validator'

const person = { name: 'Jhon Doe', age: 24, sports: [ 'basket', 'ping-pong' ] }

const rules = {
   HasValue: true,
   IsObject: true,
   Keys: {
     name: { HasValue : true },
     age: {
            IsNumeric: true,
            NumberLessThan: {
              args: 20,
              message: 'Age cant be greater than 20'
            }
       }
   }
}

const v = new Validator( person, rules )
const result = await v.Run()
```

Or you can event set a dynamic error message by passing the message like this

```
{args: 20, message: 'Age cant be greater than |*args*|'}
```

in this way, Coc will replace `|*args*|` by the value in your args.

Another case is when our args is an object, such as NumberBetween rule, it expects args to be `{ max: 20, min: 10 }` for example, in this case dynamic error message will be something like this.

```
{
  args: { max: 20, min: 10 },
  message: 'Age cant be greater than |*args.max*| and less than |*args.min*|'
}
```

Another case also is validating using user custom validators, which could happen in `PreConditions` Validator for example. The args right there is an array of functions, so what if i want a special message for each of them. moreover, what if i want each of them to append an error message to some fixed prefix.

The answer is, simply let the validators returns true or error message

eg

```
const custom = val => val > 20 : 'can not be less than 20'

const rules = {
  Keys: {
    age: {
      PreConditions: {
        args: [ custom ],
        message: 'age |*args*|'
      }
    }
  }
}
```

so if custom returns the string "can not be less th.." then coc will replace `|*args*|` with this and the result will be `age can not be less than 20`

Form Components

The concept behind Coc Form Components is simply making complex stuff quickly and easily, validation is being done by **Coc Validator**, contacting api is being done by **Coc Button** component, here's a little sample of a form using Coc Form Components.

```
 <coc-input
   v-model = "xdata.email"
   :rules = "{ HasValue: true, IsEmail: true }"
   :scope = "['my-form']"
   placeholder = "Email.."
   light-model
   labeled />
<coc-select
   v-model = "xdata.gender"
   :rules = "{ HasValue: true }"
   :scope = "['my-form']"
   :data = "['male', 'female']"
   placeholder = "Gender.."
   light-model
   labeled />
<coc-button
   :scope = "['my-form']"
   :request = "{url: '/api/register', method: 'post', xdata}"
   placeholder = "Submit"
   @coc-submit-accepted = "myCallback" />
```

This simple code can do the following:

- Validate user Input.
- Show reactive error messages for invalid fields.
- Notify when the user submit with missing fields.
- Contact the API with user data.
- Notify if the request did not success.
- Notify if the request succeeded.
- Reset Input fields on success.
- Execute a callback function on both success and error.