J.C.D.I

JavaScript Clean Data Initiative

ISAAC PARKER

UPDATED: 8/1/2018

Contents

Information 3
- Background 3
- Purpose 3
- Requirements 3
- Assumptions 3

Installation 4
- location 4
- Inclusion 4

Usage 5
- Initialization 5
- Example 6
- Explanation 7
- Callback function 7
- Data Rules 8

Breaking Down Data Rules 9
- List of Rules 9
- displayName 9
- min 10
- max 10
- required 10
- dataType 10
- patternType 10
- myPattern 11
- matchField 11

CSS Help 12

Conclusion 12

Information

Background

Forms are an important means of communication between an application and a client. However, giving users this form of gateway to an application's lower level data can quickly become catastrophic without guidelines. One of the primary ways that giving users forms to completed without restriction harms an application is through the input of bad data. Bad data can range from something as simple as leaving a field blank when it should not be, to inputting numbers in a form field that should only be letters. Fortunately, software developers have created a multitude of methods for handling data sanitization; each with their benefits and drawbacks. For this project, **JCDI (JavaScript Clean Data Initiative)**, we implemented data sanitization by means of Vanilla JavaScript. To ensure a universal generic scope, the plugin was written with no dependencies and written in vanilla JavaScript. Additionally, because of the generic nature of the script, it only performs sanitization on the client side

Purpose

The purpose of JCDI is to deliver an effective and efficient system for sanitizing data as well as apply strict data rules through input validation.

Requirements

Since the plugin does not levy any external libraries, the only requirement is that standard inclusion in the application via script tags.

Assumptions

- All data is bad data and need to be cleaned.
- The individual(s) using the plugin have basic knowledge of JSON objects, Objects, and able to create a JavaScript function.

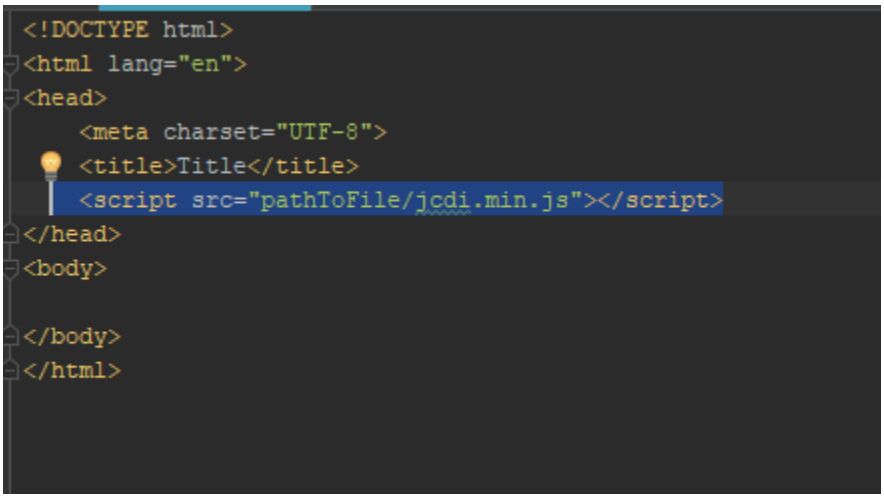
Installation

location

The file currently resides at: https://github.com/iceman5508/JCDI , once cloned or downloaded you will find two files jcdi.js and jcdi.min.js. The recommendation is to include the jcdi.min to your project.

Inclusion

To include the plugin in your project, simply add it to the head tag of your



project or in the body tag before the closing body tag.

Once you have included the plugin in your project, installation is completed.

Usage

Initialization

The plugin was designed to be very easy to use. Initialization as is simple as creating an **n_observer** object and passing in the required information.

The **n_observer** is the class that users will directly interact with. However, this interaction will be done through the **hijackForm** method. The **hijackForm** method has the following signature:

hijackForm(id, dataRules, Callback)

breaking down the params:

id: The id of the form to run this instance of n_observer against.

dataRules: The data rules to validate for.

Callback: The callback function to run after validation.

Important note: hijackForm method must be called after the form has been created or else JavaScript will throw an error saying that the form does not exist.

Example

```
    <form action="" id="sample">
        Email: <br>
        <input type="text" name="email" >
        <br>
        Username: <br>
        <input type="text" name="uname" >
        <br>
        Password:<br>
        <input type="text" name="pw" >
        <br>
        Password2:<br>
        <input type="password" name="pw2" >
        <br><br>
        <input type="submit" value="Submit">
    </form>

<script type="text/javascript">
    rules = {
        'email': {
            'displayName' : 'Email'
            ,'max' : 20
            ,'min' : 7
            ,'required':true
            ,'dataType': 'string'
            ,'patternType': 'email'
        }
        ,
        'uname': {
            'displayName' : 'Username'
            ,'max' : 20
            ,'min' : 7
            ,'required':true
            ,'dataType': 'string'
            ,'myPattern':'An-an-nn'
        }
        ,
        'pw': {
            'displayName' : 'Password'
            ,'max' : 20
            ,'min' : 7
            ,'required':true

        },
        'pw2': {
            'displayName' : 'Password2'
            ,'matchField': 'pw'

        }

    }
    new n_observer().hijackForm('sample', rules);
</script>
```

In the given example, I have a form with the id of ‘sample’. Below that we have the JavaScript code for the plugin. All the code presented is within the body tag of the application.

Note: I fully expect that your JavaScript will be in an external .js file. However, for this simple example we will us inline JavaScript.

Explanation

The line that activates the plugin, in this example is:

new n_observer().hijackForm('sample', rules);

For most uses of the plugin, in conjunction with the data rules, this signature is all you will ever need.

new n_observer().hijackForm(formId, dataRule, Callback function);

Callback function

In the example you will see that there is no Callback function provided. This was done intentionally. The callback function is optional. By default, the callback function will log all errors in the console if there was an error and log nothing if there was none.

In the example if you leave all fields blank and hit submit. You will see the errors printed in the console.

For custom, Callback functions, they must have the following signature:

Callback(status, data);

The Callback function will be passed the status of the validation process. If all was valid, it will have a status value of true. If not then it will have a status value of false as well as a data value of an array of the errors. The manipulation of that data is left to the user’s discretion.

If we were to add a custom CallBack function it may look something like this:

```
function callBack(status, data){
    if(status){ //run on success }
    else { alert(data.join(‘\n’)) ;}
}

new n_observer().hijackForm(formId, dataRule, callBack);
```

Data Rules

Perhaps the most complicated aspect of the plugin to master are the data rules. These rules are the guidelines that the plugin will use to validate the field provided.

Structurally, the data rules are **nested json objects**. The rules take the following form:

{

fieldname : { properties }

}

In the provided example our data rule is structured in the following manner:

```
        rules = {
    'email': {
        'displayName' : 'Email'
        ,'max' : 20
        ,'min' : 7
        ,'required':true
        ,'dataType': 'string'
        ,'patternType': 'email'
    }
    ,
    'uname': {
        'displayName' : 'Username'
        ,'max' : 20
        ,'min' : 7
        ,'required':true
        ,'dataType': 'string'
        ,'myPattern':'An-an-nn'
    }
    ,
    'pw': {
        'displayName' : 'Password'
        ,'max' : 20
        ,'min' : 7
        ,'required':true

    },
    'pw2': {
        'displayName' : 'Password2'
        ,'matchField': 'pw'

    }

}
```

Breaking Down Data Rules

From the code above, the first field that we are targeting is the email field.

Note: Fieldnames must match the name of the field as presented in the targeted form.

Once we have selected our field, we then begin to target the rules we want to validate for that field.

Note: While a specific order was used for targeting these rules, you can target the rules in any order you want. We simply prefer to start with displayName.

A complete list of the rules are as follows, currently there are only 8:

List of Rules

- displayName
- min
- max
- required
- dataType
- patternType
- myPattern
- matchField

displayName

All field rules must have a displayName. **This rule is required, and it is the only required rule.** It accepts data of type string and can be named anything the user desires. However, it is recommended it be named the same as the field label. Since this is what the user will see as the field name.

For example, in the code provided we have a field named ‘uname’ with a displayName of ‘username’. If an error was to occur on that field, the plugin will log an error of the form.

Username is required and cannot be blank.

min

This rule is to indicate the minimum number of characters that is required for the field. It accepts integer values and is **not** a required field rule.

max

This rule is to indicate the maximum number of characters that is required for the field. It accepts integer values and is **not** a required field rule.

required

This rule is to indicate the if the field is required or not. It accepts boolean values and is **not** a required field rule.

Note: a required property on the field in the html code also accomplishes the same goal as this rule. Additionally, having a **min** rule automatically sets the field as required.

dataType

This rule is to indicate the data type that is expected in the field. It accepts string values of the **valid** data type and is **not** a required field rule. Currently **valid data types are as follows:**

- **string** - this data type is alphanumeric
- **number** – this datatype is strictly numbers
- **boolean** – true or false values

patternType

This rule is to indicate the pattern the field should have. **Think of this rule as a more specific extension of the dataType rule**. It accepts string values of the **valid** pattern type and is **not** a required field rule. Currently **valid pattern types are as follows:**

- **email** – expects the data to be an email
- **ssn** – expects the data to be a social security number
- **phone** – expects the data to be a phone number. Primarily programmed for U.S phone numbers, but it does allow for some international patterns.

myPattern

Perhaps our favorite rule, this rule is to indicate the custom pattern the field should have. **Think of this rule as a user defined extension of the patternType rule**. It accepts a string value of the **token** pattern and is **not** a required field rule. Currently **valid tokens are as follows:**

- **A** – expects a capitalized letter
- **a** – expects a lowercased letter
- **n** – expects a numerical value.

All characters outside of alphanumeric values will be matched directly.

In the example provided, the uname field has a **myPattern** rule of 'An-an-nn'. This means that the username must.

1. Start with a capital letter
2. Then a number
3. Then a dash
4. Then a lowercase letter
5. Then a number
6. Then a dash
7. Followed by two numbers.

With this pattern of 'An-an-nn' the following username would be valid: B9-v8-12. If we wanted a 'mm/dd/yyyy' formatted date pattern, then our pattern would read: **'nn/nn/nnnn'**. Or, if we wanted a basic U.S phone number pattern we could set the pattern to: '**(nnn)-nnn-nnnn'**

matchField

This rule is to indicate the other form field to compare the current field's data to. It accepts a string value of the form fieldName to check and is **not** a required field rule.

In the example code we have this rule on the 'pw2' field. We have it set to match the pw field. Meaning that the data in both pw and pw2 must match.

CSS Help

The plugin is css friendly as it add a special ‘n_err’ class to any field that causes an error. Those fields can then be targeted with user defined css, personally we just go with the simple:

```
.n_err {
    border: 1px solid red;
}
```

Which causes the border of what ever field caused an error to be red. In some cases, depending on your markup you may have to target the parent and the child n_err as follows:

```
#formId .n_err {
    border: 1px solid red;
}
```

Conclusion

Upon completion of your data rules, simple pass it into the hijack function as shown in the example code and let the plugin handle the rest.

Best Wishes and Happy Coding!