Form Validation

CodeIgniter provides a comprehensive form validation and data prepping class that helps minimize the amount of code you'll write.

Page Contents

Overview

Before explaining CodeIgniter's approach to data validation, let's describe the ideal scenario:

1. A form is displayed.
2. You fill it in and submit it.
3. If you submitted something invalid, or perhaps missed a required item, the form is redisplayed containing your data along with an error message describing the problem.
4. This process continues until you have submitted a valid form.

On the receiving end, the script must:

1. Check for required data.
2. Verify that the data is of the correct type, and meets the correct criteria. For example, if a username is submitted it must be validated to contain only permitted characters. It must be of a minimum length, and not exceed a maximum length. The username can't be someone else's existing username, or perhaps even a reserved word. Etc.
3. Sanitize the data for security.
4. Pre-format the data if needed (Does the data need to be trimmed? HTML encoded? Etc.)
5. Prep the data for insertion in the database.

Although there is nothing terribly complex about the above process, it usually requires a significant amount of code, and to display error messages, various control structures are usually placed within the form HTML. Form validation, while simple to create, is generally very messy and tedious to implement.

Form Validation Tutorial

What follows is a "hands on" tutorial for implementing CodeIgniters Form Validation.

In order to implement form validation you'll need three things:

1. A View file containing a form.
2. A View file containing a "success" message to be displayed upon successful submission.
3. A controller method to receive and process the submitted data.

Let's create those three things, using a member sign-up form as the example.

The Form

Using a text editor, create a form called myform.php. In it, place this code and save it to your application/views/ folder:

```
<html>
<head>
<title>My Form</title>
</head>
<body>

<?php echo validation_errors(); ?>

<?php echo form_open('form'); ?>

<h5>Username</h5>
<input type="text" name="username" value="" size="50" />

<h5>Password</h5>
<input type="text" name="password" value="" size="50" />

<h5>Password Confirm</h5>
<input type="text" name="passconf" value="" size="50" />

<h5>Email Address</h5>
<input type="text" name="email" value="" size="50" />

<div><input type="submit" value="Submit" /></div>

</form>

</body>
</html>
```

The Success Page

Using a text editor, create a form called formsuccess.php. In it, place this code and save it to your application/views/ folder:

```
<html>
<head>
<title>My Form</title>
</head>
<body>

<h3>Your form was successfully submitted!</h3>

<p><?php echo anchor('form', 'Try it again!'); ?></p>

</body>
</html>
```

The Controller

Using a text editor, create a controller called Form.php. In it, place this code and save it to your application/controllers/ folder:

```
<?php

class Form extends CI_Controller {

        public function index()
        {
                $this->load->helper(array('form', 'url'));

                $this->load->library('form_validation');

                if ($this->form_validation->run() ==
FALSE)
                {
                        $this->load->view('myform');
                }
                else
                {
                        $this->load->view('formsuccess');
                }
        }
}
```

Try it!

To try your form, visit your site using a URL similar to this one:

```
example.com/index.php/form/
```

If you submit the form you should simply see the form reload. That's because you haven't set up any validation rules yet.

Since you haven't told the Form Validation class to validate anything yet, it returns FALSE (boolean false) by default. ``The run()`` method only returns TRUE if it has successfully applied your rules without any of them failing.

Explanation

You'll notice several things about the above pages:

The form (myform.php) is a standard web form with a couple exceptions:

1. It uses a form helper to create the form opening. Technically, this isn't necessary. You could create the form using standard HTML. However, the benefit of using the helper is that it generates the action URL for you, based on the URL in your config file. This makes your application more portable in the event your URLs change.
2. At the top of the form you'll notice the following function call:

   ```
   <?php echo validation_errors(); ?>
   ```

   This function will return any error messages sent back by the validator. If there are no messages it returns an empty string.

The controller (Form.php) has one method: `index()`. This method initializes the validation class and loads the form helper and URL helper used by your view files. It also runs the validation routine. Based on whether the validation was successful it either presents the form or the success page.

Setting Validation Rules

CodeIgniter lets you set as many validation rules as you need for a given field, cascading them in order, and it even lets you prep and pre-process the field data at the same time. To set validation rules you will use the `set_rules()` method:

```
$this->form_validation->set_rules();
```

The above method takes **three** parameters as input:

1. The field name - the exact name you've given the form field.
2. A "human" name for this field, which will be inserted into the error message. For example, if your field is named "user" you might give it a human name of "Username".
3. The validation rules for this form field.
4. (optional) Set custom error messages on any rules given for current field. If not provided will use the default one.

Note

If you would like the field name to be stored in a language file, please see Translating Field Names.

Here is an example. In your controller (Form.php), add this code just below the validation initialization method:

```
$this->form_validation->set_rules('username', 'Username',
'required');
$this->form_validation->set_rules('password', 'Password',
'required');
$this->form_validation->set_rules('passconf', 'Password
Confirmation', 'required');
$this->form_validation->set_rules('email', 'Email',
'required');
```

Your controller should now look like this:

```
<?php

class Form extends CI_Controller {

        public function index()
        {
                $this->load->helper(array('form', 'url'));

                $this->load->library('form_validation');

                $this->form_validation->set_rules('username', 'Username', 'required');
                $this->form_validation->set_rules('password', 'Password', 'required',
                                array('required' => 'You must provide a %s.')
                );
                $this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
                $this->form_validation->set_rules('email', 'Email', 'required');

                if ($this->form_validation->run() == FALSE)
                {
                        $this->load->view('myform');
                }
                else
                {
                        $this->load->view('formsuccess');
                }
        }
}
```

Now submit the form with the fields blank and you should see the error messages. If you submit the form with all the fields populated you'll see your success page.

> **Note**
>
> The form fields are not yet being re-populated with the data when there is an error. We'll get to that shortly.

Setting Rules Using an Array

Before moving on it should be noted that the rule setting method can be passed an array if you prefer to set all your rules in one action. If you use this approach, you must name your array keys as indicated:

```
$config = array(
        array(
                'field' => 'username',
                'label' => 'Username',
                'rules' => 'required'
        ),
        array(
                'field' => 'password',
                'label' => 'Password',
                'rules' => 'required',
                'errors' => array(
                        'required' => 'You must provide a
%s.',
                ),
        ),
        array(
                'field' => 'passconf',
                'label' => 'Password Confirmation',
                'rules' => 'required'
        ),
        array(
                'field' => 'email',
                'label' => 'Email',
                'rules' => 'required'
        )
);

$this->form_validation->set_rules($config);
```

Cascading Rules

CodeIgniter lets you pipe multiple rules together. Let's try it. Change your rules in the third parameter of rule setting method, like this:

```
$this->form_validation->set_rules(
        'username', 'Username',

'required|min_length[5]|max_length[12]|is_unique[users.usernam

        array(
                'required'      => 'You have not provided
%s.',
                'is_unique'     => 'This %s already
exists.'
        )
);
$this->form_validation->set_rules('password', 'Password',
'required');
$this->form_validation->set_rules('passconf', 'Password
Confirmation', 'required|matches[password]');
$this->form_validation->set_rules('email', 'Email',
'required|valid_email|is_unique[users.email]');
```

The above code sets the following rules:

1. The username field be no shorter than 5 characters and no longer than 12.
2. The password field must match the password confirmation field.
3. The email field must contain a valid email address.

Give it a try! Submit your form without the proper data and you'll see new error messages that correspond to your new rules. There are numerous rules available which you can read about in the validation reference.

Note

You can also pass an array of rules to `set_rules()`, instead of a string. Example:

```
$this->form_validation->set_rules('username',
'Username', array('required', 'min_length[5]'));
```

Prepping Data

In addition to the validation method like the ones we used above, you can also prep your data in various ways. For example, you can set up rules like this:

```
$this->form_validation->set_rules('username', 'Username',
'trim|required|min_length[5]|max_length[12]');
$this->form_validation->set_rules('password', 'Password',
'trim|required|min_length[8]');
$this->form_validation->set_rules('passconf', 'Password
Confirmation', 'trim|required|matches[password]');
$this->form_validation->set_rules('email', 'Email',
'trim|required|valid_email');
```

In the above example, we are “trimming” the fields, checking for length where necessary and making sure that both password fields match.

Any native PHP function that accepts one parameter can be used as a rule, like ``htmlspecialchars()``, ``trim()``, etc.

Note

You will generally want to use the prepping functions **after** the validation rules so if there is an error, the original data will be shown in the form.

Re-populating the form

Thus far we have only been dealing with errors. It’s time to repopulate the form field with the submitted data. CodeIgniter offers several helper functions that permit you to do this. The one you will use most commonly is:

```
set_value('field name')
```

Open your myform.php view file and update the **value** in each field using the `set_value()` function:

Don’t forget to include each field name in the :php:func:`set_value()` function calls!

```
<html>
<head>
<title>My Form</title>
</head>
<body>

<?php echo validation_errors(); ?>

<?php echo form_open('form'); ?>

<h5>Username</h5>
<input type="text" name="username" value="<?php echo
set_value('username'); ?>" size="50" />

<h5>Password</h5>
<input type="text" name="password" value="<?php echo
set_value('password'); ?>" size="50" />

<h5>Password Confirm</h5>
<input type="text" name="passconf" value="<?php echo
set_value('passconf'); ?>" size="50" />

<h5>Email Address</h5>
<input type="text" name="email" value="<?php echo
set_value('email'); ?>" size="50" />

<div><input type="submit" value="Submit" /></div>

</form>

</body>
</html>
```

Now reload your page and submit the form so that it triggers an error. Your form fields should now be re-populated

Note

The Class Reference section below contains methods that permit you to re-populate <select> menus, radio buttons, and checkboxes.

Important

If you use an array as the name of a form field, you must supply it as an array to the function. Example:

```
<input type="text" name="colors[]" value="<?php echo
set_value('colors[]'); ?>" size="50" />
```

For more info please see the Using Arrays as Field Names section below.

Callbacks: Your own Validation Methods

The validation system supports callbacks to your own validation methods. This permits you to extend the validation class to meet your needs. For example, if you need to run a database query to see if the user is choosing a unique username, you can create a callback method that does that. Let's create an example of this.

In your controller, change the "username" rule to this:

```
$this->form_validation->set_rules('username', 'Username',
'callback_username_check');
```

Then add a new method called `username_check()` to your controller. Here's how your controller should now look:

```
<?php

class Form extends CI_Controller {

	public function index()
	{
		$this->load->helper(array('form', 'url'));

		$this->load->library('form_validation');

		$this->form_validation->set_rules('username', 'Username', 'callback_username_check');
		$this->form_validation->set_rules('password', 'Password', 'required');
		$this->form_validation->set_rules('passconf', 'Password Confirmation', 'required');
		$this->form_validation->set_rules('email', 'Email', 'required|is_unique[users.email]');

		if ($this->form_validation->run() == FALSE)
		{
			$this->load->view('myform');
		}
		else
		{
			$this->load->view('formsuccess');
		}
	}

	public function username_check($str)
	{
		if ($str == 'test')
		{
			$this->form_validation->set_message('username_check', 'The {field} field can not be the word "test"');
			return FALSE;
		}
		else
		{
			return TRUE;
		}
	}

}
```

Reload your form and submit it with the word “test” as the username. You can see that the form field data was passed to your callback method for you to process.

To invoke a callback just put the method name in a rule, with "callback_" as the rule **prefix**. If you need to receive an extra parameter in your callback method, just add it normally after the method name between square brackets, as in: `callback_foo[bar]`, then it will be passed as the second argument of your callback method.

> **Note**
>
> You can also process the form data that is passed to your callback and return it. If your callback returns anything other than a boolean TRUE/FALSE it is assumed that the data is your newly processed form data.

Callable: Use anything as a rule

If callback rules aren't good enough for you (for example, because they are limited to your controller), don't get disappointed, there's one more way to create custom rules: anything that `is_callable()` would return TRUE for.

Consider the following example:

```
$this->form_validation->set_rules(
        'username', 'Username',
        array(
                'required',
                array($this->users_model,
'valid_username')
        )
);
```

The above code would use the `valid_username()` method from your `Users_model` object.

This is just an example of course, and callbacks aren't limited to models. You can use any object/method that accepts the field value as its' first parameter. You can also use an anonymous function:

```
$this->form_validation->set_rules(
        'username', 'Username',
        array(
                'required',
                function($value)
                {
                        // Check $value
                }
        )
);
```

Of course, since a Callable rule by itself is not a string, it isn't a rule name either. That is a problem when you want to set error messages for them. In order to get around that problem, you can put such rules as the second element of an array, with the first one being the rule name:

```
$this->form_validation->set_rules(
        'username', 'Username',
        array(
                'required',
                array('username_callable', array($this->users_model, 'valid_username'))
        )
);
```

Anonymous function version:

```
$this->form_validation->set_rules(
        'username', 'Username',
        array(
                'required',
                array(
                        'username_callable',
                        function($str)
                        {
                                // Check validity of $str and return TRUE or FALSE
                        }
                )
        )
);
```

Setting Error Messages

All of the native error messages are located in the following language file:
system/language/english/form_validation_lang.php

To set your own global custom message for a rule, you can either extend/override the language file by creating your own in
application/language/english/form_validation_lang.php
(read more about this in the Language Class documentation), or use the following method:

```
$this->form_validation->set_message('rule', 'Error
Message');
```

If you need to set a custom error message for a particular field on some particular rule, use the set_rules() method:

```
$this->form_validation->set_rules('field_name', 'Field
Label', 'rule1|rule2|rule3',
        array('rule2' => 'Error Message on rule2 for this
field_name')
);
```

Where rule corresponds to the name of a particular rule, and Error Message is the text you would like displayed.

If you'd like to include a field's "human" name, or the optional parameter some rules allow for (such as max_length), you can add the **{field}** and **{param}** tags to your message, respectively:

```
$this->form_validation->set_message('min_length', '{field}
must have at least {param} characters.');
```

On a field with the human name Username and a rule of min_length[5], an error would display: "Username must have at least 5 characters."

Note

The old *sprintf()* method of using **%s** in your error messages will still work, however it will override the tags above. You should use one or the other.

In the callback rule example above, the error message was set by passing the name of the method (without the "callback_" prefix):

```
$this->form_validation->set_message('username_check')
```

Translating Field Names

If you would like to store the "human" name you passed to the `set_rules()` method in a language file, and therefore make the name able to be translated, here's how:

First, prefix your "human" name with **lang:**, as in this example:

```
$this->form_validation->set_rules('first_name',
'lang:first_name', 'required');
```

Then, store the name in one of your language file arrays (without the prefix):

```
$lang['first_name'] = 'First Name';
```

Note

If you store your array item in a language file that is not loaded automatically by CI, you'll need to remember to load it in your controller using:

```
$this->lang->load('file_name');
```

See the Language Class page for more info regarding language files.

Changing the Error Delimiters

By default, the Form Validation class adds a paragraph tag (<p>) around each error message shown. You can either change these delimiters globally, individually, or change the defaults in a config file.

1. **Changing delimiters Globally** To globally change the error delimiters, in your controller method, just after loading the Form Validation class, add this:

```
$this->form_validation->set_error_delimiters('<div
class="error">', '</div>');
```

   In this example, we've switched to using div tags.
2. **Changing delimiters Individually** Each of the two error generating functions shown in this tutorial can be supplied their own delimiters as follows:

```
<?php echo form_error('field name', '<div
class="error">', '</div>'); ?>
```

   Or:

```
<?php echo validation_errors('<div class="error">',
'</div>'); ?>
```

3. **Set delimiters in a config file** You can add your error delimiters in application/config/form_validation.php as follows:

```
$config['error_prefix'] = '<div
class="error_prefix">';
$config['error_suffix'] = '</div>';
```

Showing Errors Individually

If you prefer to show an error message next to each form field, rather than as a list, you can use the `form_error()` function.

Try it! Change your form so that it looks like this:

```
<h5>Username</h5>
<?php echo form_error('username'); ?>
<input type="text" name="username" value="<?php echo
set_value('username'); ?>" size="50" />

<h5>Password</h5>
<?php echo form_error('password'); ?>
<input type="text" name="password" value="<?php echo
set_value('password'); ?>" size="50" />

<h5>Password Confirm</h5>
<?php echo form_error('passconf'); ?>
<input type="text" name="passconf" value="<?php echo
set_value('passconf'); ?>" size="50" />

<h5>Email Address</h5>
<?php echo form_error('email'); ?>
<input type="text" name="email" value="<?php echo
set_value('email'); ?>" size="50" />
```

If there are no errors, nothing will be shown. If there is an error, the message will appear.

Important

If you use an array as the name of a form field, you must supply it as an array to the function. Example:

```
<?php echo form_error('options[size]'); ?>
<input type="text" name="options[size]" value="<?php
echo set_value("options[size]"); ?>" size="50" />
```

For more info please see the Using Arrays as Field Names section below.

Validating an Array (other than $_POST)

Sometimes you may want to validate an array that does not originate from `$_POST` data.

In this case, you can specify the array to be validated:

```
$data = array(
        'username' => 'johndoe',
        'password' => 'mypassword',
        'passconf' => 'mypassword'
);

$this->form_validation->set_data($data);
```

Creating validation rules, running the validation, and retrieving error messages works the same whether you are validating `$_POST` data or another array of your choice.

Important

You have to call the `set_data()` method *before* defining any validation rules.

Important

If you want to validate more than one array during a single execution, then you should call the `reset_validation()` method before setting up rules and validating the new array.

For more info please see the Class Reference section below.

Saving Sets of Validation Rules to a Config File

A nice feature of the Form Validation class is that it permits you to store all your validation rules for your entire application in a config file. You can organize these rules into "groups". These groups can either be loaded automatically when a matching controller/method is called, or you can manually call each set as needed.

How to save your rules

To store your validation rules, simply create a file named form_validation.php in your application/config/ folder. In that file you will place an array named $config with your rules. As shown earlier, the validation array will have this prototype:

```
$config = array(
        array(
                'field' => 'username',
                'label' => 'Username',
                'rules' => 'required'
        ),
        array(
                'field' => 'password',
                'label' => 'Password',
                'rules' => 'required'
        ),
        array(
                'field' => 'passconf',
                'label' => 'Password Confirmation',
                'rules' => 'required'
        ),
        array(
                'field' => 'email',
                'label' => 'Email',
                'rules' => 'required'
        )
);
```

Your validation rule file will be loaded automatically and used when you call the `run()` method.

Please note that you MUST name your `$config` array.

Creating Sets of Rules

In order to organize your rules into "sets" requires that you place them into "sub arrays". Consider the following example, showing two sets of rules. We've arbitrarily called these two rules "signup" and "email". You can name your rules anything you want:

```
$config = array(
        'signup' => array(
                array(
                        'field' => 'username',
                        'label' => 'Username',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'password',
                        'label' => 'Password',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'passconf',
                        'label' => 'Password
Confirmation',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'email',
                        'label' => 'Email',
                        'rules' => 'required'
                )
        ),
        'email' => array(
                array(
                        'field' => 'emailaddress',
                        'label' => 'EmailAddress',
                        'rules' => 'required|valid_email'
                ),
                array(
                        'field' => 'name',
                        'label' => 'Name',
                        'rules' => 'required|alpha'
                ),
                array(
                        'field' => 'title',
                        'label' => 'Title',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'message',
                        'label' => 'MessageBody',
                        'rules' => 'required'
                )
        )
);
```

Calling a Specific Rule Group

In order to call a specific group, you will pass its name to the `run()` method. For example, to call the signup rule you will do this:

```
if ($this->form_validation->run('signup') == FALSE)
{
        $this->load->view('myform');
}
else
{
        $this->load->view('formsuccess');
}
```

Associating a Controller Method with a Rule Group

An alternate (and more automatic) method of calling a rule group is to name it according to the controller class/method you intend to use it with. For example, let's say you have a controller named Member and a method named signup. Here's what your class might look like:

```
<?php

class Member extends CI_Controller {

        public function signup()
        {
                $this->load->library('form_validation');

                if ($this->form_validation->run() ==
FALSE)
                {
                        $this->load->view('myform');
                }
                else
                {
                        $this->load->view('formsuccess');
                }
        }
}
```

In your validation config file, you will name your rule group member/signup:

```
$config = array(
        'member/signup' => array(
                array(
                        'field' => 'username',
                        'label' => 'Username',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'password',
                        'label' => 'Password',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'passconf',
                        'label' => 'PasswordConfirmation',
                        'rules' => 'required'
                ),
                array(
                        'field' => 'email',
                        'label' => 'Email',
                        'rules' => 'required'
                )
        )
);
```

When a rule group is named identically to a controller class/method it will be used automatically when the `run()` method is invoked from that class/method.

Using Arrays as Field Names

The Form Validation class supports the use of arrays as field names. Consider this example:

```
<input type="text" name="options[]" value="" size="50" />
```

If you do use an array as a field name, you must use the EXACT array name in the Helper Functions that require the field name, and as your Validation Rule field name.

For example, to set a rule for the above field you would use:

```
$this->form_validation->set_rules('options[]', 'Options',
'required');
```

Or, to show an error for the above field you would use:

```
<?php echo form_error('options[]'); ?>
```

Or to re-populate the field you would use:

```
<input type="text" name="options[]" value="<?php echo
set_value('options[]'); ?>" size="50" />
```

You can use multidimensional arrays as field names as well. For example:

```
<input type="text" name="options[size]" value="" size="50"
/>
```

Or even:

```
<input type="text" name="sports[nba][basketball]" value=""
size="50" />
```

As with our first example, you must use the exact array name in the helper functions:

```
<?php echo form_error('sports[nba][basketball]'); ?>
```

If you are using checkboxes (or other fields) that have multiple options, don't forget to leave an empty bracket after each option, so that all selections will be added to the POST array:

```
<input type="checkbox" name="options[]" value="red" />
<input type="checkbox" name="options[]" value="blue" />
<input type="checkbox" name="options[]" value="green" />
```

Or if you use a multidimensional array:

```
<input type="checkbox" name="options[color][]" value="red"
/>
<input type="checkbox" name="options[color][]"
value="blue" />
<input type="checkbox" name="options[color][]"
value="green" />
```

When you use a helper function you'll include the bracket as well:

```
<?php echo form_error('options[color][]'); ?>
```

Rule Reference

The following is a list of all the native rules that are available to use:

Rule	Parameter	Description
required	No	Returns FALSE if the form element is empty.
matches	Yes	Returns FALSE if the form element does not match the one in the parameter.
regex_match	Yes	Returns FALSE if the form element does not match the regular expression.
differs	Yes	Returns FALSE if the form element does not differ from the one in the parameter.

Rule	Parameter	Description
is_unique	Yes	Returns FALSE if the form element is not unique to the table and field name in the parameter. Note: This rule requires Query Builder to be enabled in order to work.
min_length	Yes	Returns FALSE if the form element is shorter than the parameter value.
max_length	Yes	Returns FALSE if the form element is longer than the parameter value.
exact_length	Yes	Returns FALSE if the form element is not exactly the parameter value.
greater_than	Yes	Returns FALSE if the form element is less than or equal to the parameter value or not numeric.
greater_than_equal_to	Yes	Returns FALSE if the form element is less than the parameter value, or not numeric.

Rule	Parameter	Description
less_than	Yes	Returns FALSE if the form element is greater than or equal to the parameter value or not numeric.
less_than_equal_to	Yes	Returns FALSE if the form element is greater than the parameter value, or not numeric.
in_list	Yes	Returns FALSE if the form element is not within a predetermined list.
alpha	No	Returns FALSE if the form element contains anything other than alphabetical characters.
alpha_numeric	No	Returns FALSE if the form element contains anything other than alpha-numeric characters.
alpha_numeric_spaces	No	Returns FALSE if the form element contains anything other than alpha-numeric characters or spaces. Should be used after trim to avoid spaces at the beginning or end.

Rule	Parameter	Description
alpha_dash	No	Returns FALSE if the form element contains anything other than alpha-numeric characters, underscores or dashes.
numeric	No	Returns FALSE if the form element contains anything other than numeric characters.
integer	No	Returns FALSE if the form element contains anything other than an integer.
decimal	No	Returns FALSE if the form element contains anything other than a decimal number.
is_natural	No	Returns FALSE if the form element contains anything other than a natural number: 0, 1, 2, 3, etc.
is_natural_no_zero	No	Returns FALSE if the form element contains anything other than a natural number, but not zero: 1, 2, 3, etc.
valid_url	No	Returns FALSE if the form element does not contain a valid URL.

Rule	Parameter	Description
valid_email	No	Returns FALSE if the form element does not contain a valid email address.
valid_emails	No	Returns FALSE if any value provided in a comma separated list is not a valid email.
valid_ip	Yes	Returns FALSE if the supplied IP address is not valid. Accepts an optional parameter of 'ipv4' or 'ipv6' to specify an IP format.
valid_base64	No	Returns FALSE if the supplied string contains anything other than valid Base64 characters.

Note

These rules can also be called as discrete methods. For example:

```
$this->form_validation->required($string);
```

Note

You can also use any native PHP functions that permit up to two parameters, where at least one is required (to pass the field data).

Prepping Reference

The following is a list of all the prepping methods that are available to use:

Name	Parameter	Description
prep_for_form	No	DEPRECATED: Converts special characters so that HTML data can be shown in a form field without breaking it.
prep_url	No	Adds "http://" to URLs if missing.
strip_image_tags	No	Strips the HTML from image tags leaving the raw URL.
encode_php_tags	No	Converts PHP tags to entities.

Note

You can also use any native PHP functions that permits one parameter, like `trim()`, `htmlspecialchars()`, `urldecode()`, etc.

Class Reference

class `CI_Form_validation`

`set_rules`(*$field* [, *$label* = '' [, *$rules* = '' [, *$errors* = *array()*]]])

Parameters:
- **$field** (*string*) – Field name
- **$label** (*string*) – Field label
- **$rules** (*mixed*) – Validation rules, as a string list separated by a pipe "|", or as an array or rules
- **$errors** (*array*) – A list of custom error messages

Returns: CI_Form_validation instance (method chaining)

Return type: CI_Form_validation

Permits you to set validation rules, as described in the tutorial sections above:

- Setting Validation Rules
- Saving Sets of Validation Rules to a Config File

`run` ([*$group* = ''])

Parameters:
- **$group** (*string*) – The name of the validation group to run

Returns: TRUE on success, FALSE if validation failed

Return type: bool

Runs the validation routines. Returns boolean TRUE on success and FALSE on failure. You can optionally pass the name of the validation group via the method, as described in: Saving Sets of Validation Rules to a Config File

`set_message` (*$lang* [, *$val* = ''])

Parameters:
- **$lang** (*string*) – The rule the message is for
- **$val** (*string*) – The message

Returns: CI_Form_validation instance (method chaining)

Return type: CI_Form_validation

Permits you to set custom error messages. See Setting Error Messages

`set_error_delimiters` ([*$prefix* = ' ' [, *$suffix* = ' ']])

Parameters:
- **$prefix** (*string*) – Error message prefix
- **$suffix** (*string*) – Error message suffix

Returns: CI_Form_validation instance (method chaining)

Return type: CI_Form_validation

Sets the default prefix and suffix for error messages.

`set_data` (*$data*)

Parameters:
- **$data** (*array*) – Array of data validate

Returns: CI_Form_validation instance (method chaining)

Return type: CI_Form_validation

Permits you to set an array for validation, instead of using the default `$_POST` array.

`reset_validation` ()

Returns: CI_Form_validation instance (method chaining)

Return type: CI_Form_validation

Permits you to reset the validation when you validate more than one array. This method should be called before validating each new array.

`error_array` ()

Returns: Array of error messages

Return type: array

Returns the error messages as an array.

`error_string` ([*$prefix* = '' [, *$suffix* = '']])

Parameters:
- **$prefix** (*string*) – Error message prefix
- **$suffix** (*string*) – Error message suffix

Returns: Error messages as a string

Return type: string

Returns all error messages (as returned from error_array()) formatted as a string and separated by a newline character.

`error` (*$field* [, *$prefix* = '' [, *$suffix* = '']])

Parameters:
- **$field** (*string*) – Field name
- **$prefix** (*string*) – Optional prefix
- **$suffix** (*string*) – Optional suffix

Returns: Error message string

Return type: string

Returns the error message for a specific field, optionally adding a prefix and/or suffix to it (usually HTML tags).

`has_rule` (*$field*)

Parameters:
- **$field** (*string*) – Field name

Returns: TRUE if the field has rules set, FALSE if not

Return type: bool

Checks to see if there is a rule set for the specified field.

Helper Reference

Please refer to the Form Helper manual for the following functions:

- `form_error()`
- `validation_errors()`
- `set_value()`
- `set_select()`
- `set_checkbox()`
- `set_radio()`

Note that these are procedural functions, so they **do not** require you to prepend them with `$this->form_validation`.