Error Checking

Blesta uses the Input component to perform automatic error checking and error message handling. The way it works is by defining a set of rules and validating input based on those rules. The Input component is a simple, yet powerful way to validate input. As we'll see, input can even be formatted during validation, either immediately before or after the rule is executed.

Boolean Rules

Boolean rules are the simplest rules of all. Setting the rule to **true** will always pass validation. Similarly, setting the rule to **false** will always fail validation.

Boolean Rules

```
<?php
        ...
        $rules = array(
                'field_1' => array(
                        'empty' => array(
                                'rule' => true,
                                'message' => "This error will never be displayed"
                        )
                ),
                'field_2' => array(
                        'valid' => array(
                                'rule' => false,
                                'message' => "This error will always be displayed"
                        )
                )
        );
        $this->Input->setRules($rules);
?>
```

if_set Option

It is often useful to evaluate an input field only if it is submitted (such as when an edit is being performed). To do so, we can use the 'if_set' option.

Boolean Rules

```
<?php
        ...
        $rules = array(
                'field_1' => array(
                        'empty' => array(
                                'if_set' => true,
                                'rule' => "isEmpty",
                                'negate' => true,
                                'message' => "This rule is only evaluated if field_1 is set and not null"
                        )
                )
        );
        $this->Input->setRules($rules);
?>
```

Built-in Rules

The Input component has a set of built in rules. The syntax for invoking these rules is a string or a single dimensional array.

Built-in Rules

```
<?php
        ...
        $rules = array(
                'field_1' => array(
                        'empty' => array(
                                'rule' => "isEmpty",
                                'negate' => true,
                                'message' => "Field 1 may not be empty."
                        )
                ),
                'field_2' => array(
                        'valid' => array(
                                'rule' => array("maxLength",32),
                                'message' => "Field 2 must be 32 characters or less."
                        )
                )
        );
        $this->Input->setRules($rules);
?>
```

isEmpty

This built-in rule determines whether or not content passed to it is empty. This may be boolean false, an integer 0, an empty string (i.e. ""), null, etc. It is common to set the "negate" field with this rule, as often the validation rule should only accept data that is **not** empty, rather than what **is** empty. The negate field simply tells the rule to validate against the negation of the rule's return value.

```
<?php
        ...
        $rules = array(
                'field_1' => array(
                        'empty' => array(
                                'rule' => "isEmpty",
                                'negate' => true,
                                'message' => "Field 1 may not be empty."
                        )
                )
        );
        ...
?>
```

isPassword

This built-in rule determines whether a given string matches a regular expression for validating passwords. It may accept additional parameters identifying the minimum length of the password; the type of regular expression to validate against, in the case you choose to use an existing expression already supported; or your own custom regular expression.

This rule accepts multiple parameters:

1. The minimum length required for the password
2. The type of regular expression to use, which must be one of the following:
    a. **any** - Validates anything that meets the minimum length requirement
    b. **any_no_space** - Validates anything that meets the minimum length requirement and does not contain spaces
    c. **alpha_num** - Validates anything that meets the minimum length requirement and is alpha-numeric
    d. **alpha** - Validates anything that meets the minimum length requirement and contains only alpha characters
    e. **num** - Validates anything that meets the minimum length requirement and contains only integers
    f. **custom** - Specifies that you want to set your own custom regular expression as the next parameter
3. The custom regular expression you would like to use, if the type of regular expression is set to "custom"

```
<?php
        ...
        $rules = array(
                'password1' => array(
                        'format' => array(
                                'rule' => array("isPassword", 8, "alpha_num"),
                                'message' => "Password1 must be at least 8 characters in length, and alpha-
numeric."
                        )
                ),
                'password2' => array(
                        'format' => array(
                                'rule' => array("isPassword", 0, "custom", "/^[a-z][0-9]{11,}$/Di"),
                                'message' => "Password2 must begin with a letter, followed by only numbers, no
less than 12 characters in length."
                        )
                )
        );
        ...
?>
```

isDate

This built-in rule determines whether the given string is in a valid date format. It may also accept minimum and maximum date values as parameters to be used in determining whether the date to be validated is within the given range. The dates may be string-formatted dates, or UNIX timestamps.

The rule accepts multiple parameters:

1. The minimum date
2. The maximum date

```
<?php
        ...
        $rules = array(
                'date_begins' => array(
                        'format' => array(
                                'rule' => "isDate",
                                'message' => "The date_begins is not in a valid date format."
                        )
                ),
                'date_ends' => array(
                        'format' => array(
                                'rule' => array("isDate", "2013-05-01", "2013-06-01"),
                                'message' => "The date_ends must be a valid date between May 1, 2013 and June
1, 2013."
                        )
                )
        );
        ...
?>
```

matches

This built-in rule determines whether a given string matches a custom regular expression.

This rule accepts multiple parameters

1. The regular expression

```
<?php
        ...
        $rules = array(
                'phone_number' => array(
                        'format' => array(
                                'rule' => array("matches", "/^([0-9]{3}\.){2}[0-9]{4}$/"),
                                'message' => "The phone number must be of the format ###.###.####"
                        )
                )
        );
        ...
?>
```

compares

This built-in rule satisfies a logical comparison between the input and your custom comparator.

This rule accepts multiple parameters:

1. The logical comparison operator, one of:
    a. >
    b. <
    c. >=
    d. <=
    e. ==
    f. ===
    g. !=
    h. !==
2. The value to compare with

```
<?php
        ...
        $rules = array(
                'status' => array(
                        'format' => array(
                                'rule' => array("compares", "==", "active"),
                                'message' => "The status must be 'active'."
                        )
                )
        );
        ...
?>
```

between

This built-in method determines whether the input value is between a minimum and maximum value.

This rule accepts multiple parameters:

1. The minimum value to compare the value against
2. The maximum value to compare the value against
3. A boolean value of false if the rule must validate strictly within the min/max range, or true if the range may include the min and max values themselves

```
<?php
        ...
        $rules = array(
                'score' => array(
                        'format' => array(
                                'rule' => array("between", "1", "100", false),
                                'message' => "The score must be a number between 2 and 99."
                        )
                )
        );
        ...
?>
```

minLength

This built-in method determines whether the given value is at least a given character length.

This rule accepts multiple parameters:

1. The minimum length to require

```
<?php
        ...
        $rules = array(
                'last_name' => array(
                        'format' => array(
                                'rule' => array("minLength", 3),
                                'message' => "Your last name must be at least 3 characters in length."
                        )
                )
        );
        ...
?>
```

maxLength

This built-in method determines whether the given value is at most a given character length.

This rule accepts multiple parameters:

1. The maximum length to require

```
<?php
        ...
        $rules = array(
                'last_name' => array(
                        'format' => array(
                                'rule' => array("maxLength", 32),
                                'message' => "Your last name must not exceed 32 characters in length."
                        )
                )
        );
        ...
?>
```

betweenLength

This built-in method determines whether the given value is between both the minimum and maximum character lengths.

This rule accepts multiple parameters:

1. The minimum length to require
2. The maximum length to require

```
<?php
        ...
        $rules = array(
                'last_name' => array(
                        'format' => array(
                                'rule' => array("betweenLength", 3, 32),
                                'message' => "Your last name must be between 3 and 32 characters in length."
                        )
                )
        );
        ...
?>
```

PHP Rules

In additional to the built-in Input rules, you can also invoke any PHP function as a rule as well.

> ⚠ **PHP language constructs are not functions**
>
> It's important to note that PHP has a number of language constructs that *look* like functions but are not. You can not use language constructs as rules.

Consider the example below that validates whether an input value is in a predefined array using PHP's "in_array" function:

```
<?php
        ...
        $rules = array(
                'status' => array(
                        'format' => array(
                                'rule' => array("in_array", array("active", "pending", "in_review")),
                                'message' => "The given status is invalid. Valid statuses are 'active',
'pending', or 'in_review'."
                        )
                )
        );
        ...
?>
```

Custom Rules

You may also define your own custom rules using a callback function to perform input validation. All callback validation methods must return boolean true if the input value validates successfully and boolean false if the validation fails. They must also be **public** methods. The syntax for the rule is a two-dimensional array containing the reference to the class the validation method resides, and to the method itself, followed by any additional parameters for that method.

Consider the example below which validates whether a invoice record exists in the database, assuming that the validation method is apart of the same class where the validation takes place.

```
<?php
class MyClass {
        ...
        private function getRules() {
                return array(
                        'invoice_id' => array(
                                'exists' => array(
                                        'rule' => array(array($this, "validateExists"), 256),
                                        'message' => "The invoice ID does not exist."
                                )
                        )
                );
        }

        public function validateExists($invoice_id, $client_id=null) {
                Loader::loadComponents($this, array("Record"));

                // Query the database to check whether the invoice ID exists
                $this->Record->select()->from("invoices")->where("invoices.id", "=", $invoice_id);

                // If a client ID is given, require the invoice belong to that client
                if ($client_id)
                        $this->Record->where("invoices.client_id", "=", $client_id);

                return ($this->Record->numResults() > 0);
        }
        ...
}
?>
```

Formatting

There are two attributes that handle data formatting during validation. They are:

- pre_format
- post_format

As their names suggest they allow formatting of data before, or after, validation. These callbacks work just like the callbacks we've seen for rule validation.

The following example pre-formats the given input number into only digits and validates that it is at least 4 digits in length:

```
<?php
        ...
        $rules = array(
                'number' => array(
                        'format' => array(
                                'pre_format' => array(array($this, "formatNumber")),
                                'rule' => array("minLength", 4),
                                'message' => "Please enter a number at least 4 digits in length."
                        )
                )
        );
        ...

        public function formatNumber($number) {
                return preg_replace("/[^0-9]*/", "", $number);
        }
?>
```

The following example validates that an input name is given, and then performs PHP's trim function on the input to remove white-space before and after the string:

```php
<?php
        ...
        $rules = array(
                'name' => array(
                        'format' => array(
                                'rule' => "isEmpty",
                                'negate' => true,
                                'message' => "Please enter your name",
                                'post_format' => array("trim")
                        )
                )
        );
        ...
?>
```