Contents

List of Abbreviations

HRC HTTP Request Checker

HTTP Hypertext Transfer Protocol

MVC Model-Viewer-Controller

PHP PHP: Hypertext Preprocessor; originally Personal Home Page

1 Introduction

PHP is a widely used server-side scripting language, utilized to generate web pages [3]. Validating user inputs in PHP, for instance data in forms, is a common practice. As you might guess, numerous validators exist, especially as a part of a PHP MVC framework, such as Zend [4], Nette [2] or CodeIgnitor [1]. They are able to validate custom strings and verify if the string represents, for instance, a float value. However, those validators do not quite suit our needs.

In this Section, we outline the main problems within the existing validators (section 1.2), and set goals we want to achieve with an implementation of a new validator (section 1.3).

1.1 Terminology

In higher programming languages, an Array data type is considered to be a homogenous structure, i.e. each element in the Array is of the same data type. In PHP, the array could contain various values with various indexes. That makes the PHP array a heterogeneous structure with indexes similar to a hash array. In this document, we will use both, homogenous and heterogeneous, types of arrays. In order to avoid confusion and make the writing and reading of this document easier we use the following terms:

Array which is considered to be a **true-array**, i.e. homogenous structure.

array which is simply a **PHP array.**

array attribute usually refers to a value being validated. The value is an attribute in some array, such as $_POST.

object that represents an **object-like array**. In PHP, it is still an associative array. In JavaScript context, an object contains various attributes with various data types, i.e. a heterogeneous structure.

1.2 Problem

The client, usually a web-browser, sends the user inputs using HTTP to the server, i.e. all the data is sent within a text representation. It could be simply a name, an integer value, etc. Basically, there are only two data types in PHP that could be retrieved from the HTTP request :

1. **String** which is able to represent any value submitted by the client. For instance, “John”, “john@serve.com”, “1971”, or “0.25”. This type of submitted values is easy to validate with the existing validators.
2. **Array** which contains other values, either strings or other arrays. It could possibly be both: homogenous or heterogeneous structure. The common PHP validators lack the ability to easily validate Arrays, objects, or more complex structures.

There is a possible way how to avoid using the complex structures, and to use just the simple string data type. We could concatenate more string values using a string separator. However, this approach has several drawbacks:

Extra processing in the web-browser (concatenation) and also on the server (separation/parsing).

Escaping the separator to avoid problems on the server when parsing the string. Let's say, we separate the values with a comma and the concatenated string might look like this: "value1,other value,not good?". In case that the user writes a comma in one of the values and it is not somehow escaped, the process will most probably end up with an unexpected result.

1.3 Motivation

The possibility to use arrays in the HTTP request has numerous advantages. Especially, if we use JavaScript, many parts of the submitted data are created dynamically, and we do not exactly know how many parameters will be sent to the server.

Nevertheless at the moment, we do not have a straightforward way how to validate the inputs contained in an array. The main goal is to create a validator which is easy to learn to use, and has the ability to easily set up validation rules especially for the arrays and more complex structures.

For example, we might obtain an Array of integers similar to the example below. The task is to create a validator which able to validate such an Array of integers possibly with a single line, without the usage of any additional loop. Similar when talking to a human, you are supposed to tell the validator: "validate an array of integers" - simple as that.

```
array(
        0 => "1",
        1 => "0",
        2 => "5"
);
```

2 HTTP Request Checker – New Validator

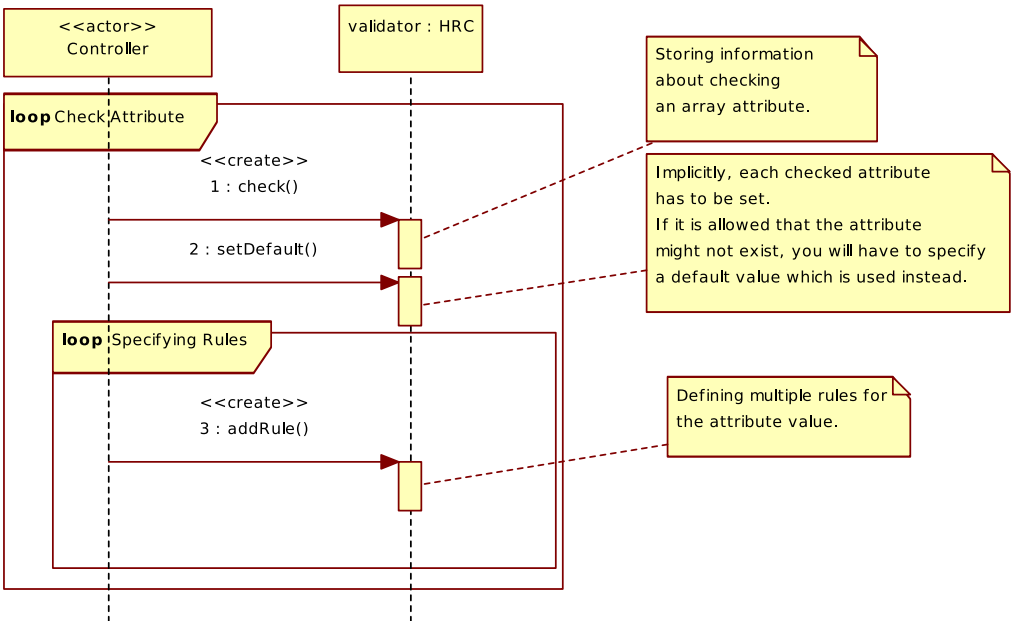
In this section, the main idea about how the new validators works is explained. In section 2.1, you can see the basic usage of the validator. The validation itself is explained in section 2.2. In section 2.3, you can find out how the validation rules are built and what kind of variables and attributes are you able to validate.

2.1 Basic Usage

The basic usage of *HTTP Request Checker* (HRC) is similar to other known PHP user input validators. The process consists of several steps:

1. Specify which array variable to evaluate. For instance, it could be a superglobal like `$_POST` or a local variable such as `$myArray`.

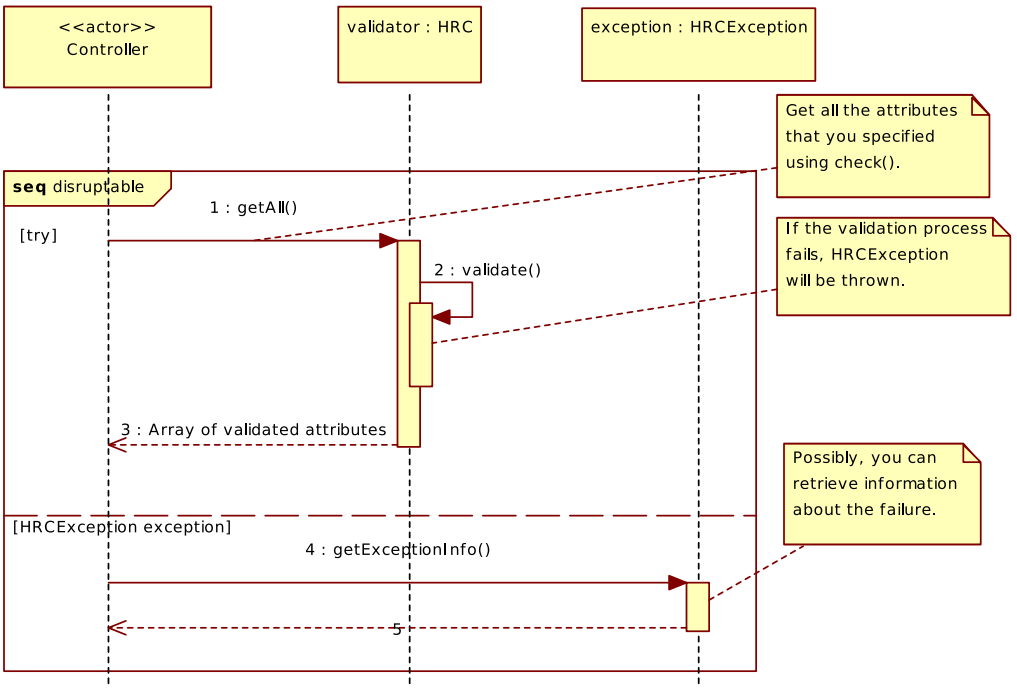
Figure 1: UML sequence diagram illustrating preparation of HRC object to check and validate certain array attributes.



2. Store the information about which array attributes to check and which validation rules to apply. For instance, some attributes are integer values, some are e-mail addresses. Additionally, you can specify a default value for the attribute in case it does not exist. This might be helpful when working, for instance, with checkboxes because the web browser usually does not send any information about the checkbox when it is not checked.
   This step is illustrated in Figure 1.

3. Retrieve the previously specified attributes. The validation of the attributes takes place in this step. In case any of the attributes does not comply with the specified rules, the standard execution flow is interrupted, and an exception is thrown [5].
   This step is illustrated in Figure 2.

Figure 2: UML sequence diagram demonstrating validation and retrieval of specified array attributes.



2.2 Validation

The validation occurs when we use a getter to retrieve a desired attribute. All the specifications and validation rules need to be set beforehand. There are a few types of getters:

`get(sAttributeName)` retrieves a single attribute that was previously specified. The validation process takes place in this method, and it is illustrated in Figure 3.

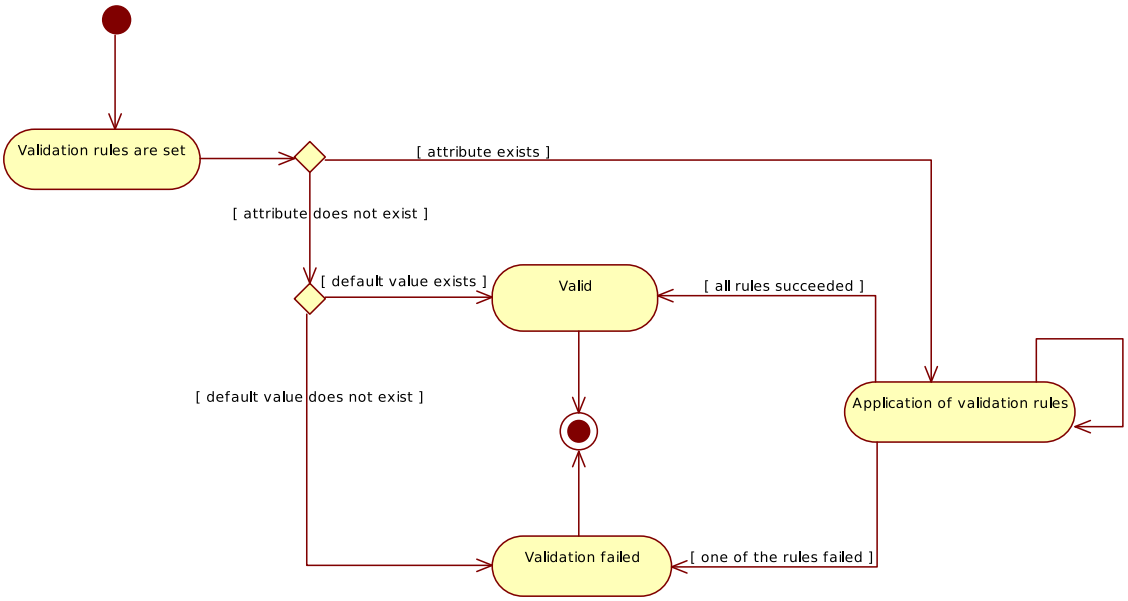
`getAll()` retrieves and validates each previously specified attribute using the method `get()`. The returned result is an array containing the validated attributes.

`take(sAttributeName)` retrieves a single value using the method `get()`. Additionally, the attribute specification is deleted from the HRC object. This might be convenient when you want to have a few attributes obtained separately and then retrieve the remaining subset using `getAll()`.

2.3 Validation Rules

A validation rule evaluates an array attribute that we specified. We could build a lot of custom validation rules templates that would probably call other functions in order to decide if an array attribute is valid. That would take a lot of time in the implementation process as well as learning how to use the validator. A more straightforward way would be to use just any existing function and based on its result make a decision about validity of the attribute. PHP provides a mechanism

Figure 3: UML state machine diagram demonstrating the validation process of a single array attribute.



to call existing functions, e.g. based on a string name of the function. In order to create a flexible rule that can be basically used to call just any function or method, the rule contains five attributes; the first three attributes define what function with what parameters to call, the other two indicate how to handle the returned result:

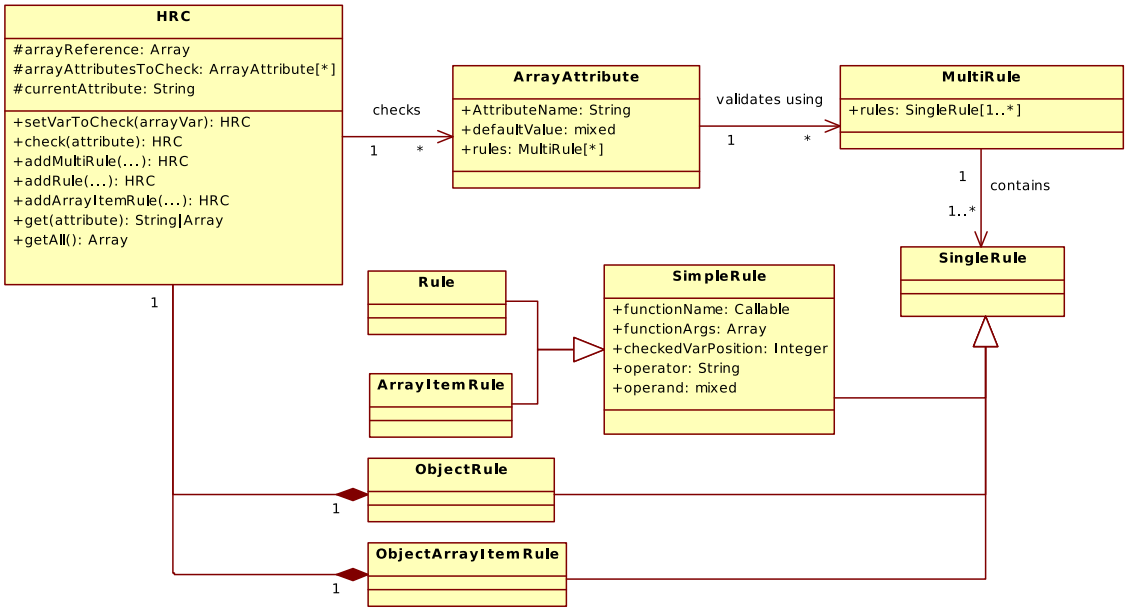
1. **PHP callable** which could be, for instance, a string name of a function or a pair of an object and its method.
2. **Arguments** to be passed to the callable. Most of the callables used for validation usually accept just one argument which is, in our case, the array attribute we want to validate. However in some cases, you need to pass other arguments to the callable as well.
3. When you pass more arguments to the callable, the validated attribute is appended to the argument list by default. Nevertheless, sometimes you need to insert the validated attribute at another **position** in the argument list.
4. **Operator** used to compare the result returned from the callable. The operator is stored as a string.
5. **Value** which is compared with the returned result from a callable using the specified operator. The outcome of the comparison indicates if the validation rule succeeded or failed. For example, `false == true`, where `false` is the result of a callable and `true` is the specified value, indicates a failure.

2.3.1 Array Validation Rule

In order to enable the functionality to validate an Array, we include an additional attribute in the rule. This attribute indicates if we should validate a single value or treat it as an Array and

validate every single element in that Array. In Figure 4, the ArrayItemRule is shown as a different class for a better illustration. However as mentioned in this paragraph, the only difference between a Rule and ArrayItemRule is a value of a single attribute.

Figure 4: UML class diagram showing the possibilities of validation rules to map any array attributes.



2.3.2 Multi Rule Using OR Logic

You can specify several rules to validate an array attribute. Those rules use the AND logic - all the rules have to succeed. In some cases it might be useful to apply the OR logic. For example, a user should write a number but if he leaves a field blank, i.e. sends an empty string instead of a number to the server, it is not supposed to cause any trouble.

As shown in Figure 4, you can mix different types of rules, e.g. Rules and ArrayItemRules, into the MultiRule. For example, you could expect to receive an Array of integers OR an empty string.

In order to avoid pointless complexity in the implementation, each single rule which is not using the OR logic is wrapped into a MultiRule, anyway. That means that the MultiRule would contain exactly one rule.

2.3.3 Object Rule

Until now, it was explained how to validate a string and an Array of strings. It is also possible to validate more complex structures. In the following code, you can see an object representing a person.

```
array(
        "name"   => "George",
        "age"    => "33",
```

```
        "email" => "george@someserver.org"
);
```

If the client sends only these three attributes to the server, we can easily set a few rules for them. However, if we receive more of these objects or possibly an Array of objects, we should rather apply a different technique.

As you can see in Figure 4, a rule can contain another HRC object instead of a callable with additional attributes. This results in validating the array attribute as an object (with other attributes) rather than a single value. And again, the HRC object contains information indicating whether to check a single object or an Array of objects. For example, we could obtain an Array of objects as shown below:

```
array(
        0 => array(
                "name"  => "George",
                "age"   => "33",
                "email" => "george@someserver.org"
        ),
        1 => array(
                "name"  => "Emma",
                "age"   => "21",
                "email" => "emma@otherserver.org"
        ),
        .....
        .....
);
```

To validate such a structure with the HRC validator is an easy task:

1. Prepare a HRC object to validate a single object, such as in the previous example.
2. Create a new HRC object and specify to validate an Array of objects where the validation rule is actually the first HRC object that was specified to validate a single object.

To see an actual example, look at the generated API documentation located in the Appendix A on the following page.

Appendix A

The generated API comes here...

References

[1] CodeIgniter User Guide Version 2.1.3: Form Validation. http://ellislab.com/codeigniter/user-guide/libraries/form_validation.html. [Accessed: 2013.02.11].

[2] Nette\Utils\Validators. http://doc.nette.org/en/validators. [Accessed: 2013.02.11].

[3] PHP. http://www.php.net. [Accessed: 2013.02.11].

[4] ZEND: Standard Validation Classes. http://framework.zend.com/manual/1.12/en/zend.validate.set.html. [Accessed: 2013.02.11].

[5] IBM. How to visualize and model Exception handling in Sequence Diagrams. http://www-01.ibm.com/support/docview.wss?uid=swg21221178. [Accessed: 2013.02.11].