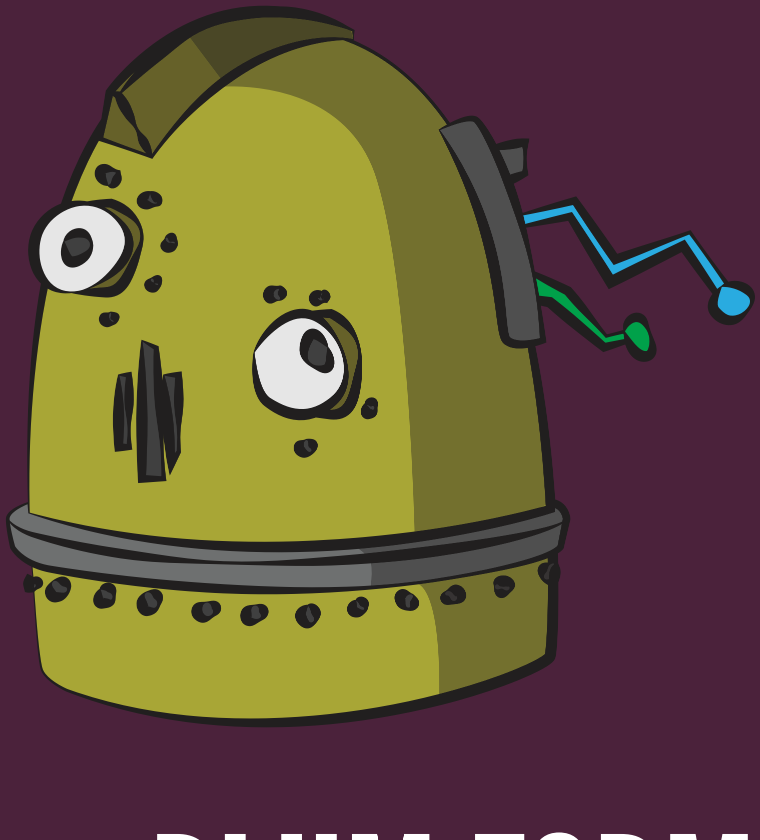

PLUM.FORM

BEAUTIFYING WEB FORMS

Hello! Thanks so much for purchasing plum.Form! After going through this documentation, you'll be well on your way to creating beautiful web forms that don't skimp on features and style.

Plum.Form is designed to allow you complete control over forms in web design, and aspect that has long been a burder on front-end designers and developers.

1 THE PLUGIN

1.1 INTRODUCTION

For sake of simplicity, plum.Form will be refered to as “Plum” inside this documentation. You should know how to use jQuery plugins; you can get a crash-course on jQuery by reading How jQuery Works (http://docs.jquery.com/Tutorials:How_jQuery_Works).

1.2 REQUIREMENTS

- jQuery 1.4.3 or newer

- Mozilla Firefox 2 or newer
- Apple Safari 3 or newer
- Google Chrome
- Opera 9 or newer
- Microsoft Internet Explorer 7 or newer

1.3 CHANGE LOG

- version 1.5.2 (15 January, 2012)
    - **Changed in 1.5.1**: checkbox groups no longer use two classes for identification. See 7.5
    - Fixed binding “submit” and “complete” function options to the form
    - Various demo page and documentation fixes
- version 1.5.1 (8 January, 2012)
    - Fixed auto-upload for file fields
    - Fixed queue removal for file fields
    - Fixed multi-select in Mac OS
- version 1.5 (6 January, 2012)
    - Fixed validation on submission
    - Better support for multiple file fields
    - Added “speed” option for animations
    - Added “invalid” class name for invalid fields

- version 1.4.1 (25 October, 2011)
    - Better width control
    - Fix for elements not initially in the DOM
    - Fix for dots in email validation
- version 1.4 (13 October, 2011)
    - Added “live” support for on-the-fly form elements
    - Added “shake” option
    - CSS tweaks
- version 1.3 (6 August, 2011)
    - Added classes for HTML5 input types
    - Added “active” class to indentify when an element is active
    - Added option to reset a form after a successful submission
    - Changed file configuration options (see 4.2)
    - Improved validation expressions
    - Added “url” check
    - Added “method” validation option
    - Added support to verify a value
    - Changed “phone” validation to “tel” to match with the HTML5 element of the same type
    - Fixed “onchange” event on select menus
    - Added “rebuild,” “open,” and “close” events for select-one menus
    - Added the ability to extend plum.Form with custom methods
- version 1.2.3 (15 July, 2011)
    - Numerous Internet Explorer fixes
    - Fixed a bug that didn’t allow the same file to be selected more than once
- version 1.2 (2 July, 2011)
    - Modularized Plum’s methods to allow for easier expansion
    - Better support for multiple forms
    - Added support for field validation
    - Added “validate” and “success” class options for field validation
    - Added support for in-field labels
    - Added “label” class option for in-field labels
    - Added support for file upload progress in HTML5 browsers
    - Added progress bar to the fileItem option
    - Added support for styling individual elements

    - Changed AJAX file upload methods
    - Main wrapper class changed from “plum” to “plum-form”
    - Small bug fixes & tweaks to the default skin

- version 1.1 (14 June, 2011)

    - Reset buttons fixed
    - Multiple file support added
    - Minor bug fixes

- version 1.0 (6 June, 2011)

    - The first release of plum.Form

2 THE JAVASCRIPT FILES

2.1 JQUERY.JS

Plum.Form needs a minimum of jQuery 1.4.3 to support all of its features. It includes jQuery 1.6.2, the latest at the time of this writing.

2.2 JQUERY.PLUM.JS

This file is included with all jQuery plugins by RoboCréatif, as they will all use it to initiate the plugin. There is a small difference between Plum plugins for jQuery and “normal” plugins. Usually, a plugin is used with:

```
$('#selector').pluginName(options);
```

A Plum plugin is called by using:

```
$('#selector').plum('plugin_name', options);
```

Where “plugin_name” is any plugin in the Plum series. It’s done this way so Plum’s plugins use only one namespace, which will allow other plugins to use what Plum may have used (for example, plum.Form allows another plugin to use `$('#selector').form();`).

2.3 PLUM.FORM.JS

This file defines plum.Form. It contains all of the methods that allow you to style and validate forms.

2.4 JQUERY.PLUM.FORM.JS

This file contains both jquery.plum.js and plum.form.js. If you’re not using any other Plum plugins, it’s recommended to use this file as it will save on HTTP requests to your server.

3 INSTALLATION

3.1 INCLUDE THE JAVASCRIPT FILES

As with any other jQuery plugin, you need to include each file on every page you want to style a form. You will need jQuery, plum.Form, and of course, a style sheet to make your forms beautiful. The skins that come with Plum need a couple extra tweaks to work properly in Internet Explorer 7. You can include the IE7 CSS with a conditional comment.

```
<!DOCTYPE html>
<html lang="en">
<head>
      <title>My Awesome Form</title>
      <script src="/js/jquery.js"></script>
      <script src="/js/jquery.plum.form.js"></script>
      <link rel="stylesheet" href="/skins/default/default.css">
      <!--[if IE7]>
            <link rel="stylesheet" href="/skins/default/default.ie7.css">
      <![endif]-->
</head>
<body>
```

3.2 CREATE A FORM

Naturally, the only way to style a form is to create one. You don't need to do anything special to your forms to have them working with Plum. Basic HTML is fine.

```
<form action="/process-form.php" method="post" enctype="multipart/form-data">
      <p><input name="email" type="email"></p>
      <p><input name="file" type="file"></p>
      <p><input type="submit" value="Send">
</form>
```

3.3 TARGET YOUR FORMS

Finally, you will need to tell Plum that you want it to style the form you just created. A basic installation can include nothing more than the plugin; a more advanced set up might contain a list of configuration options. The configuration is a comma-separated list of "key: value" pairs passed as the second argument in the `.plum()` function.

Plum can not access your forms until the document has finished loading. This can be accomplished by calling it inside jQuery's `$(document).ready(function () { });` method, or by including the configuration at the end of the page. Plum's demos include it at the end of the page, so that's what we'll do here.

```
<script>
$('form').plum('form', {
      submit: function () {
            alert('Form is being submitted.');
      }
});
</script>
</body>
</html>
```

You can also target individual elements. For example, if you wanted to style only the file upload field in the form above, you can tell Plum to look for file fields.

```
<script>
$(':file').plum('form', { ajax: true });
</script>
```

4 CONFIGURATION OPTIONS

4.1 LIST OF OPTIONS

Now that you know how to get Plum to pay attention to forms, you're ready to dive into the lovely list of options that are available. If you have multiple forms, you can define different options for each form.

```
$('#form-1').plum('shop', {
      ajax: true,
      json: true,
      complete: function (result) {
            alert(result);
      }
});
$('#form-2').plum('form');
```

Below is a full list of available options, their default values, and the description of each option.

`ajax`	`false`	This is a boolean value (`true` or `false`) directing Plum to send the form without loading the page, or naturally by submitting the data to a new page. In older browsers and Microsoft Internet Explorer, a hidden iframe is used for file uploads if `ajax` is set to `true`.
`action`	`null`	If `action` is null, Plum uses the "action" attribute defined in the nearest `<form>` tag. If an action isn't found, the current URL is used.
`classes`	`[ object ]`	The classes option is an object which contains class names that you'd like to use while styling your form. Because there are so many, they are discussed in chapter 4.3.
`complete`	`function () { }`	After the form has finished submitting, Plum calls this function. If your server returns a response, that response is passed as a single argument to the `complete` function. This is only applicable to forms with `ajax: true`.
`file`	`[ object ]`	An object containing options specific to file uploads. See 4.2 for the list of options.
`json`	`false`	Setting this option to true will attempt to parse a JSON string that the server returns after a form submission.
`labels`	`false`	In-field labels can be enabled by setting this option to `true`.
`reset`	`true`	A value of `true` will reset the form after the form has successfully submitted.
`shake`	`true`	If a user attempts to submit a form with invalid fields, those fields will run a quick "shake" method. Setting this to `false` will prevent invalid fields from shaking.
`speed`	`150`	The speed of all animations, including opening select menus, in-field label fading and removing files.
`submit`	`function () { }`	A function that is called prior to submitting the form. If this function returns `false` (or equivalent), the form will not be submitted. Any other value will submit the form.

4.2 FILE OPTIONS

To configure file uploads, file options are contained within a nested object.

```
$('form').plum('form', {
      ajax: true,
      file: {
            html: '{filename} {filesize}',
            size: 10240
      }
});
```

Below is a list of options available for file uploads.

`button`	`'Choose a file...'`	The text you would like to appear on the button that triggers the file dialogue window.
`complete`	`function () { }`	When submitting an AJAX-enabled form, files are uploaded individually and separately from the main form. This function is called after each file has finished uploading.
`errorsize`	`'Please choose a file smaller than {filesize}.'`	The error message that is displayed when a user attempts to upload a file that's too large.
`errortype`	`'Allowed file types: {filetype}.'`	The error message that is displayed when a user attempts to upload an unsupported file.

`files`	`0`	The maximum number of files that can be uploaded in a single submission. 0 denotes unlimited files.
`html`	`string`	An HTML string that is used to format each file in the file upload queue. See below for more info on `file.html`.
`progress`	`function () { }`	Called during the upload progress of a file. Requires HTML5's `XMLHttpRequestUpload`.
`size`	`0`	The maximum size per file, in bytes. 0 denotes a file of any size. Requires HTML5's File API.
`start`	`function () { }`	This function is called just prior to uploading each file. Requires HTML5's `XMLHttpRequestUpload`.
`types`	`[ ]`	An array of allowed file types. For example, `[ 'image/jpeg', 'image/png' ]` will allow only JPEG and PNG images. Requires HTML5's File API. Please note that MIME types are determined by the browser, not by Plum, and not all browsers will recognize all MIME types.

Alternatively to using the `button` option to creating the button text, you can also add a `title` attribute to the field. For example,

```
<input type="file" name="files[]">
<script> $(':file').plum('form', { file: { button: 'Browse files' }});
```

...is the same as using this:

```
<input type="file" name="files[]" title="Browse files">
<script> $(':file').plum('form'); </script>
```

You can alter the HTML used in each list item in the file upload queue by using the `file.html` configuration option. Within this HTML string, you can also use some special short codes to identify unknown values. The default HTML used for each file is:

```
<span class="filename">{filename}</span>
<span class="remove">&times;</span>
<span class="filesize">{filesize}</span>
<div class="progress"><div></div></div>
```

Two of these classes ("remove" and "progress") can be changed with the `classes` configuration option, as they have a special purpose. "filename" and "filesize," however, are nothing more than identifiers that

the skins included with Plum use for styling. By no means are you required to use them.

Plum includes three shortcodes that you can use in the `html`, `errorsize` and `errortype` options.

`{filename}`	This is replaced with the name of the file being uploaded.
`{filesize}`	In supporting browsers, the size of the file is added here. Depending on the size, Plum will convert it to GB, MB, KB, or bytes.
`{filetype}`	Replaced with the file's MIME type as reported by the browser.

Remember, displaying an error message is meant to be a convenience to the user, rather than an end-all solution to validation. Old browsers and visitors with JavaScript disabled will not have problems reported to them if they are attempting to upload an invalid file.

4.3 CLASS NAMES

All Plum wrappers are given a universal class name called "plum-form" (you can use `.plum-form { }` to style this). Additionally, they are given class names that apply to the element's current state and type. For example, Plum will "wrap" a text field with a div, and replace the original field with:

```
<div class="plum-form input text">
      <input type="text">
</div>
```

To prevent conflicts with other HTML elements on your page that may be using the "input" or "text" classes, you can redefine Plum's class names by using the `class` configuration option.

```
$('form').plum('form', {
      classes: {
            input: 'custom-input-class',
            text: 'text-class-name'
      }
});
```

Plum will then use the above to create:

```
<div class="plum-form custom-input-class text-class-name">
      <input type="text">
</div>
```

Below is a list of all classes that can be changed, their default values, and the description.

`active`	`'active'`	When a mousedown event occurs on an element, this class is added to its wrapper.
`arrow`	`'select-arrow'`	Identifies the drop-down arrow in a select-one menu.
`button`	`'button'`	A class to identify the wrapper for `<input type="button">` and `<button></button>`.
`checkbox`	`'checkbox'`	Used for `<input type="checkbox">`.
`checked`	`'checked'`	Added to the wrapper when a checkbox or radio button is checked.
`closed`	`'closed'`	Used when a select menu is closed.
`container`	`'select-container'`	Used for the option list in select menus.
`date`	`'date'`	Used for `<input type="date">`.
`datetime`	`'datetime'`	Used for `<input type="datetime">`.
`disabled`	`'disabled'`	Added to any field or option that is disabled.
`file`	`'file'`	Used for `<input type="file">`.
`filelist`	`'filelist'`	The list of files in the upload queue.
`focus`	`'focus'`	Added to an element's wrapper when the element is focused.
`email`	`'email'`	Used for `<input type="email">`.
`error`	`'error'`	Added to `<div class="info"></div>` for field validation, or to the `<li>` tag in the file queue list if an error occurs.
`hover`	`'hover'`	Used when hovering over an element.
`info`	`'info'`	Used for the status icon in fields that need to be validated.
`input`	`'input'`	Used for all `<input>` wrappers.
`invalid`	`'invalid'`	Added to the wrapper of all fields that have failed validation.
`label`	`'label'`	Used for the in-field labels.
`loading`	`'loading'`	Added to the `<li>` tag in the file list while that file is being uploaded.
`mixed`	`'mixed'`	Added to checkbox group handlers if some boxes in the group are checked.
`month`	`'month'`	Used for `<input type="month">`.
`multiple`	`'multiple'`	Identifies a select-multiple field.
`number`	`'number'`	Used for `<input type="number">`.
`open`	`'open'`	When a select menu is open (this is always added to select-multiple fields).
`optgroup`	`'optgroup'`	Identifies the `<optgroup>` in a menu.
`option`	`'option'`	Identifies the `<option>` tags in a menu.
`password`	`'password'`	Used for `<input type="password">`.
`progress`	`'progress'`	Used for the progress bar in file uploads.
`radio`	`'radio'`	Used for `<input type="radio">`.
`range`	`'range'`	Used for `<input type="range">`.
`remove`	`'remove'`	You can add this class to any tag in your `file.html` option. Clicking it will remove the file from the upload queue.
`reset`	`'reset'`	Used for `<input type="reset">`.
`text`	`'text'`	Used for `<input type="text">` and `<input>` (a field without a type).
`textarea`	`'textarea'`	Used for `<textarea></textarea>`.
`search`	`'search'`	Used for `<input type="search">`.
`select`	`'select'`	Used for the outer wrapper of `<select>` menus.
`selected`	`'selected'`	Added to the currently selected option(s).
`single`	`'single'`	Used for select-one menus.
`tel`	`'tel'`	Used for `<input type="tel">`.
`url`	`'url'`	Used for `<input type="url">`.
`value`	`'select-value'`	In select-one menus, this class is added to the container that holds the text of the currently selected option.
`waiting`	`'waiting'`	Added to the `<li>` tags in the file list for files that are waiting to be uploaded.
`wrapper`	`'select-wrapper'`	Select menus have two different wrappers to allow for overflow and scrollbars. This is the class name of the inner wrapper.

4.4 IN-FIELD LABELS

You can enable in-field labels simply by setting the `labels` option to `true`. In-field labels will take a field's existing label and lay it over the original field. When a user clicks on the field, the label will fade to 30% opacity. Typing in the field will completely remove the label, which will become visible again when the field is blurred and no text exists. If a field has a default value, the in-field label will not be displayed.

Labels can be triggered in one of two ways: by adding a "for" attribute to a label and "id" to the field, or by wrapping the field in a label.

```
<label for="text-field">Text field label</label>
<input id="text-field" type="text">

<label>Text field label <input type="text"></label>

<label for="textarea-label">Textarea label</label>
<textarea id="textarea-label" cols="50" rows="6"></textarea>
```

5 FILE UPLOADS

5.1 USING CALLBACK FUNCTIONS

If you are using an AJAX-enabled form, there are three callback functions that you can use when dealing with file uploads: `start`, `progress` and `complete`. All are defined within the `file` configuration option. In modern browsers that support HTML5's `XMLHttpRequestUpload` object, you can take advantage of its progress event to let the user know what percentage of the file has been uploaded.

A file upload list may look something similar to:

```
<ul class="filelist">
      <li class="success">
            <div>Filename.jpg</div>
            <div class="progress"><div></div></div>
      </li>
      <li class="loading">
            <div>Filename.jpg</div>
            <div class="progress"><div></div></div>
      </li>
      <li class="waiting">
            <div>Filename.jpg</div>
            <div class="progress"><div></div></div>
      </li>
</ul>
```

This has three files in the queue: one which has already been uploaded ("success"), one which is currently being uploaded ("loading"), and a third which is waiting to be uploaded ("waiting").

Each of Plum's file callback functions uses the current `<li>` object as its `this` value. That means you can manipulate the list item of the current file being uploaded. They also have the default event object passed as an argument, extended to included the progress bar and percentage uploaded.

To better explain, here are some examples of how to use the file callback functions. Remember, we're working inside the `file` configuration option with AJAX enabled:

```
$('form').plum('form', {
      ajax: true,
      file: {
            // The file callback functions go here
      }
});
```

Example 1: adding a 5-pixel green border around a file when it begins to upload.

```
start: function () {
      $(this).css('border', '5px solid green');
}
```

Example 2: adjusting the progress bar to the current upload percentage.

This is working under the assumption that you are using the default `file.html` (see 4.2).

```
progress: function (event) {
      $('div.progress div', this).css('width', event.percent + '%');
}
```

Example 3: alerting the user that the file has finished uploading.

```
complete: function () {
      alert('Upload complete.');
}
```

The `complete` callback function has a second argument passed through it. This argument contains the printed result from your server after the upload.

Example 4: checking the success of an upload.

Let's assume your form is uploading to "process-form.php". Inside process-form.php, you have a script to upload files (examples in the following sections). You may need to check file size or type, but for whatever reason, the file may fail to be saved on the server.

```
<?php
if ($upload_success) {
      echo 'File uploaded.';
} else {
      echo 'File upload failed.';
}
exit;
?>
```

You can listen for the result of that upload inside your `complete` function.

```
complete: function (event, result) {
      if (result === 'File uploaded.') {
            alert('File has been successfully uploaded.');
      } else {
            alert('File failed to upload.');
      }
}
```

It should be noted that while the `this` value of the `complete` function is the current list item, Plum will automatically hide the item when a file has finished uploading.

5.2 HANDLING FILE UPLOADS

Due to the nature of AJAX file uploads, files are uploaded separately from the rest of the form data. Unless a user is working on an unsupported browser, which uploads the entire form via a hidden iframe, Plum will first upload the form's text data, followed by individual images.

Because of this, it can sometimes be slightly confusing if you're uploading files in an application that directly relate to the text data (for example, a blog entry that uses images). You can work around such limitations by storing the text data in a session, then process the text and images together. It's also very important to realize that AJAX file uploads are not sent to the server in the traditional form. In PHP, for example, modern browsers will not send the file in a way that stores it in the `$_FILES` array. Older browsers will still be sent in the traditional method.

Example 1: using modern and traditional upload techniques.

Note: the following code snippets are done in PHP 5. They may cause problems in older versions

Modern browsers will upload a file through HTTP headers. We can begin by checking that the relevant headers exist. If they do, we can assume we need to use the modern method. Let's call this "upload.php".

```
<?php
// upload.php
if (
      array_key_exists('HTTP_X_REQUESTED_WITH', $_SERVER)
      && array_key_exists('HTTP_X_FILE_NAME', $_SERVER)
) {

      // The php://input stream holds the file contents.
      // $_SERVER['HTTP_X_FILE_NAME'] may be something like 'Filename.jpg'
```

```
		$file = file_get_contents('php://input');
		file_put_contents($_SERVER['HTTP_X_FILE_NAME'], $file);

}
```

Otherwise, if the proper headers do not exist, we can assume files are being uploading in a more traditional method.

```
elseif (!empty($_FILES)) {

	foreach ($_FILES as $name => $files) {
		foreach ($files['error'] as $i => $error) {
			// If an error occurred, continue to the next file.
			if ($error) {
				continue;
			}
			// "tmp_name" contains the file contents.
			// "name" contains the file name, like 'Filename.jpg'
			copy($files['tmp_name'][$i], $files['name'][$i]);
		}
	}

}
?>
```

Example 2: storing form data in a session for later use.

```
<form action="process.php" method="post" enctype="multipart/form-data">
	<p><input name="text-field"></p>
	<p><input name="files" type="file"></p>
	<p><input type="submit" value="Submit"></p>
</form>
```

First, we need to listen for the “text-field” POST data.

```
<?php
// process.php
session_start();

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
	if (!array_key_exists('HTTP_X_FILE_NAME')) {
```

```
		// We can save the entire $_POST array to a "post" index in
		// the session
		$_SESSION['post'] = $_POST;

		// Exit the script after saving the text data
		exit;
	}
}
```

Once the script ends and returns back to the client, Plum will continue and upload files. At this point, we can also work with the session data.

```
if (isset($_SESSION['post'])) {
	// Process data from $_SESSION['post']
	// For example, save it in a database.

	// Upload the file. You can reference contents in $_SESSION['post'] from
	// within "upload.php"
	include 'upload.php';
}
?>
```

6 FORM VALIDATION

6.1 INTRODUCTION

Validation in Plum works as an additional plugin. To use it, simply target the plugin on each field that you need validated.

```
$('#selector').plum('form.verify', options);
```

`options` is a list of validation techniques that you need applied to that specific field. When a form is submitted, Plum runs each of the verification processes. If any process fails, an `:invalid` pseudo-selector is added to that element. If any elements are marked with `:invalid`, the form will not be submitted.

6.2 PRE-DEFINED MATCHING METHODS

Plum includes three different methods that check for value matches. These are "email," "tel," and "url."

Example 1: check that a field contains a valid email address.

```
$('#selector').plum('form.verify', { method: 'email' });
```

The same works with "tel" and "url" to verify a telephone number and URL. Plum's telephone verification will match 7- or 10-digit numbers with delimeters being spaces, dots or dashes, and extensions.

These are considered to be valid telephone numbers (and by no means limited to this set):

```
555 5555
1-555-555-5555
1.555.555.5555
15555555555 extension 555
15555555555 ext. 555
15555555555 # 555
15555555555 x. 555
```

6.3 FIELD LENGTH

You can also dictate how long you wish a field's value to be. This is accomplished through the "min" and "max" options.

Example 1: determine a field to be a minimum of 5 characters.

```
$('#selector').plum('form.verify', { min: 5 });
```

Example 2: determine a field to be a maximum of 100 characters.

```
$('#selector').plum('form.verify', { max: 100 });
```

6.4 METHOD COMBINATIONS

You can combine methods to create a more detailed check. Let's say you have a blog comment form with an optional URL field. Setting `method: url` will require the field to be populated, but a populated field is not optional.

Example 1: validate an optional URL field.

```
$('#selector').plum('form.verify', { min: 0, method: 'url' });
```

By setting `min` to 0, you are essentially telling Plum that the field is valid if it has a length of 0 or is a valid URL. You could also use the `max` option to dictate a maximum URL length.

6.5 MATCH AN EXACT STRING

The simplest validation method involves directly matching a string. This is done by passing the string as the second argument, in place of the options list.

Example 1: ensure a field is identical to "Hello world".

```
$('#selector').plum('form.verify', 'Hello world');
```

6.6 CUSTOM VALIDATION

If the supplied validation techniques don't meet your requirements, you can customize it with a callback function. Like strings, the callback is used in place of the options list. This allows you to match multiple regular expressions, or even verify the contents of other fields in relation to the current field.

The return value dictates the validity. If the funtion returns false, the field is invalid. If it returns true, it is valid.

Example 1: determine if a field has at least one letter and one number.

```
$('#selector').plum('form.verify', function () {
      return /[a-zA-Z]/.test(this.value) && /[0-9]/.test(this.value);
});
```

Example 2: verify that a “retype password” field matches a “password” field

```
$('#selector').plum('form.verify', function () {
      // Get the value of the "password" field
      var value = $('#password-selector').val();

      // Check that the "password" field is populated and that it matches the
      // value of this field.
      return value && this.value === value;
});
```

7 THE INNER WORKINGS OF PLUM

7.1 WRAPPERS AND CONTAINERS

What makes Plum so powerful is the simplicity in how fields are constructed. For every element you need Plum to access, it will wrap the original element with a `<div>` tag. That tag is referred to as a wrapper, and it contains all of the classes that define the element’s type and state.

By using easy-to-remember class names, you can very easily access and style the wrappers and their contents. All wrappers have a global class, called “plum-form.” You can style it using:

```
.plum-form { /** wrapper properties */ }
.plum-form * { /** inner properties */ }
```

The following examples are given as a means for you to have an understanding of the HTML that Plum creates. You can use the classes in chapter 4.3 for a more thorough documentation, and this chapter’s general structural outline so you can style your forms. Remember that you don’t need extra markup beyond basic form elements.

7.2 FILE INPUT FIELDS

In addition to a basic wrapper, file fields are also given a custom button and an unordered list, where files queued for upload are displayed.

If your form has:

```
<input type="file" name="files[]">
```

... Plum will convert it to:

```
<div class="plum-form input file">
      <div class="plum-form button">
            <button type="button">Choose a file...</button>
            <input type="file" name="files[]">
      </div>
      <ul class="filelist">
            <li>File in queue</li>
      </ul>
</div>
```

7.3 SELECT MENUS (AND CUSTOM EVENTS)

Select menus are a little more complicated, as they need support for scroll bars, multiple and single selection. To accomplish this, select menus are given two wrappers. The original select menus are hidden from view, but activity on the menu created by Plum is still recognized by the original menu.

Example 1: a select-one menu.

```
<select name="menu">
      <option>Option 1</option>
      <option value="2">Option 2</option>
      <option value="3" disabled>Disabled option</option>
      <optgroup label="Option group">
            <option value="4" selected>Option 4
      </optgroup>
</select>
```

... is translated into:

```
<div class="plum-form select single closed">
      <div class="select-wrapper">
            <div class="selected-value">
                  <div>Selected option 4</div>
                  <div class="select-arrow"></div>
            </div>
            <ul class="select-container">
                  <li class="option">Option 1</li>
                  <li class="option">Option 2</li>
                  <li class="option disabled">Disabled option</li>
                  <li class="optgroup">
                        <div>Option group label</div>
                        <ul>
                              <li class="option selected">Selected option 4</li>
                        </ul>
                  </li>
            </ul>
      </div>
      <select name="menu">
            <option>Option 1</option>
            <option value="2">Option 2</option>
            <option value="3" disabled>Disabled option</option>
            <optgroup label="Option group label">
                  <option value="4" selected>Selected option 4</option>
            </optgroup>
      </select>
</div>
```

Example 2: a select-multiple menu.

```
<select name="menu" multiple disabled>
      <option value="1">Option 1</option>
      <option value="2">Option 2</option>
      <option value="3">Option 3</option>
</select>
```

... is translated into:

```
<div class="plum-form select multiple disabled open">
      <div class="select-wrapper">
            <ul class="select-container">
                  <li class="option">Option 1</li>
                  <li class="option">Option 2</li>
                  <li class="option">Option 3</li>
            </ul>
      </div>
      <select name="menu" multiple disabled>
            <option value="1">Option 1</option>
            <option value="2">Option 2</option>
            <option value="3">Option 3</option>
      </select>
</div>
```

Plum select menus have three unique events: “rebuild,” “open” and “close.” You can listen for these events to create very customizable forms.

Cascading menus are a series of menus. When the value of one menu changes, the contents of another are updated. Plum needs to rebuild the HTML for custom menus, so you can trigger the “rebuild” event when a menu’s value is changed.

Example 3: buiding a cascading menu with camera manufacturers and models.

First, build your menu.

```
<select id="camera-make" name="camera_make">
      <option value="Select a manufacturer">Select a manufacturer</option>
      <option value="Canon">Canon</option>
      <option value="Nikon">Nikon</option>
      <option value="Pentax">Pentax</option>
</select>
```

```
<select id="camera-model" name="camera_model">
      <option value="Select a model">Select a model</option>
</select>
```

Then you can write some basic JavaScript using jQuery to listen for changes and make the necessary updates on the "camera-model" menu.

```
<script>
$('#camera-make').bind('change', function () {
      // We need to determine what models are associated with what manufacturer.
      switch (this.value) {
            case 'Canon':
                  var options = ['60D ($1,170)', '7D ($1,700)'];
                  break;
            case 'Nikon':
                  var options = ['D90 ($1,200)', 'D7000 ($1,500)'];
                  break;
            case 'Pentax':
                  var options = ['K-5 ($1,600)'];
                  break;
            default:
                  var options = [];
                  break;
      }
      // Next, we add the "Select a model" option to the front of the list
      options.unshift('Select a model');
      $('#camera-model')
            // Add the options to the "camera-model" menu
            .html('<option>' + options.join('</option><option>') + '</option>')
            // Finally, tell Plum to rebuild the menu
            .trigger('rebuild');
});
</script>
```

Example 2: build a cascading menu using AJAX.

Using the same menu markup as example 1, you can listen for changes on the menu and grab data from your server to populate the "camera-model" menu.

```
<script>
$('#camera-make').bind('change', function () {
      $.post('options-list.php', { value: this.value }, function (options) {
            $('#camera-model').html(options).trigger('rebuild');
      });
});
</script>
```

```
<?php
// options-list.php
switch ($_POST['value']) {
      case 'Canon':
            $options = array('60D ($1,170)', '7D ($1,700)');
            break;
      case 'Nikon':
            $options = array('D90 ($1,200)', 'D7000 ($1,500)');
            break;
      case 'Pentax':
            $options = array('K-5 ($1,600)');
            break;
      default:
            $options = array();
            break;
}
array_unshift($options, 'Select a model');
foreach ($options as $option) {
      echo '<option value="' . $option . '">' . $option . '</option>';
}
?>
```

We can also have fun with the "open" and "close" events, which are triggered when a menu is opened or closed, respectively.

Example 3: changing the width of a menu when opened and closed.

```
<select id="open-close" name="open_close">
```

```
	<option value="1">Option 1</option>
	<option value="2">Option 2</option>
	<option value="3">Option 3</option>
</select>
```

```
<script>
var open_close_width;
$('#open-close').bind({
	open: function () {
		var wrapper = $(this).parent();
		// We need the width of the outer wrapper so we know what to go back
		// to after the menu is closed.
		// "this.wrapper" is a cached jQuery object equivalent to
		// $(this).closest('.plum-form');
		open_close_width = wrapper.width();
		// Then change the width of the wrapper and the options container
		$('ul.select-container', wrapper).add(wrapper)
			.animate({ width: 300 }, 150);
	},
	close: function () {
		var wrapper = $(this).parent();
		// Now we need to adjust the containers back to their original widths
		$('ul.select-container', wrapper).add(wrapper)
			.animate({ width: open_close_width }, 150);
	}
});
</script>
```

7.4 ALL OTHER ELEMENTS

File inputs and select menus are really the only elements with specially designed markup. All other form elements are given a very simple wrapper that defines the type and its current state. The sheer number of possible class combinations is too great to document here, so these are some examples of what Plum might create for your forms.

Example 1: submit, reset and button fields.

```
<div class="plum-form input submit">
	<input type="submit">
```

```
</div>
<div class="plum-form input reset">

	<input type="reset">
</div>
<div class="plum-form button">
	<button type="button">Button</button>
</div>
```

Example 2: hovering over a checked radio button.

```
<div class="plum-form radio checked hover">
	<input type="radio" checked>
</div>
```

Example 3: mouse-down on a basic text field.

```
<div class="plum-form input text active">
	<input>
</div>
```

7.5 CHECKBOX GROUPS

Plum has a special feature to further enhance your experience with checkboxes. Group handlers (boxes which either check all or none in a group) are created by adding a “check-all-groupname” class, where “groupname” is the name of each box in the group.

```
<input class="check-all-boxes[]" type="checkbox> Check all
<input type="checkbox" name="boxes[]"> Box 1
<input type="checkbox" name="boxes[]"> Box 2
```

By clicking on the “Check all” box, both “Box 1” and “Box 2” will be checked. If they are both checked and the “Check all” box is clicked, all boxes are emptied. If one box in the group is checked, the “Check all” box’s wrapper is given a “mixed” class.

Example 1: Plum’s HTML markup for all boxes checked.

```
<div class="plum-form input checkbox checked">
	<input class="check-all-boxes[]" type="checkbox" checked>
</div> Check all
<div class="plum-form input checkbox checked">
```

```
      <input type="checkbox" name="boxes[]" checked>
</div> Box 1

<div class="plum-form input checkbox checked">
      <input type="checkbox" name="boxes[]" checked>
</div> Box 2
```

Example 2: Plum's HTML markup with one box checked.

```
<div class="plum-form input checkbox checked mixed">
      <input class="check-all group-boxes[]" type="checkbox" checked>
</div> Check all
<div class="plum-form input checkbox">
      <input type="checkbox" name="boxes[]">
</div> Box 1
<div class="plum-form input checkbox checked">
      <input type="checkbox" name="boxes[]" checked>
</div> Box 2
```

7.6 STYLING TIPS

The underlying feature in Plum is basic CSS. There are tons of class combinations that you can use to style specific parts of your form, far too many in fact to cover in this documentation. To take control, you just need to understand what classes are applied where.

By default, Plum uses the name of the tag (e.g., `.input { }` for `<input type="text">`), the type of the field (e.g., `.text { }` for `type="text"`) and any applicable states (e.g., `.hover { }` when the mouse hovers over the field). Chapter 4.3 has a full list of the default class names and how Plum uses them.

Example 1: target the wrapper of a disabled checkbox.

```
.plum-form.input.checkbox.disabled { }
.plum-form.checkbox.disabled { }
.input.checkbox.disabled { }
.checkbox.disabled { }
```

All of the above will accomplish the same task, so you only need one. This is just an example showing possible combinations should you need to further refine your design.