Синтаксические плагины

Syntax Plugins are plugins to extend DokuWiki's syntax. To be able to understand what is needed to register new Syntax within DokuWiki you should read how the Parser works.

Synopsis

A Syntax Plugin *Example* needs:

- class name `syntax_plugin_example`
- which extends SyntaxPlugin[1].
- to be stored in a file `lib/plugins/example/syntax.php`.

Moreover, a plugin.info.txt file is needed. For full details of plugins and their files and how to create more syntax components refer to plugin file structure.

The class needs to implement at least the following functions:

- **`getType()`** Should return the type of syntax this plugin defines (see below)
- **`getSort()`** Returns a number used to determine in which order modes are added, also see parser, order of adding modes and getSort list.
- **`connectTo($mode)`** This function is inherited from dokuwiki\Parsing\ParserMode\AbstractMode [2]. Here is the place to register the regular expressions needed to match your syntax.
- **`handle($match, $state, $pos, Doku_Handler $handler)`** to prepare the matched syntax for use in the renderer
- **`render($format, Doku_Renderer $renderer, $data)`** to render the content

The following additional methods can be overridden when required:

- **`getPType()`** Defines how this syntax is handled regarding paragraphs[3]. Return:
  - `normal` — (default value, will be used if the method is not overridden) The plugin output will be inside a paragraph (or another block element), no paragraphs will be inside
  - `block` — Open paragraphs will be closed before plugin output, the plugin output will not start with a paragraph
  - `stack` — Open paragraphs will be closed before plugin output, the plugin output wraps other paragraphs
- **`getAllowedTypes()`** (default value: `array()`) Should return an array of mode types that may be nested within the plugin's own markup.
- **`accepts($mode)`** This function is used to tell the parser if the plugin accepts syntax mode $mode within its own markup. The default behaviour is to test $mode against the array of modes held by the inherited property `allowedModes`. This array is also filled with modes from the mode types given in `getAllowedTypes()`.

Additional functions can be defined as needed.

Inherited Properties

- **`allowedModes`** — initial value, an empty array, inherited from AbstractMode [4]. Contains a list of other syntax modes which are allowed to occur within the plugin's own syntax mode (ie. the modes which belong to any other DokuWiki markup that can be nested inside the plugin's own markup). Normally, it is automatically populated by the `accepts()` function using the results of `getAllowedTypes()`.

Inherited Functions

- See common plugin functions for inherited functions available to all plugins. e.g. localisation, configuration and introspection.

Syntax Types

DokuWiki uses different syntax types to determine which syntax may be nested. Eg. you can have text formatting inside of tables. To integrate your plugin into this system it needs to specify which type it is and which types can be nested within it. The following types are currently available:

Modetype	Used in mode…	Description
container	listblock, table, quote, hr	containers are complex modes that can contain many other modes – hr breaks the principle but they shouldn't be used in tables / lists so they are put here
baseonly	header	some modes are allowed inside the base mode only
formatting	strong, emphasis, underline, monospace, subscript, superscript, deleted, footnote	modes for styling text – footnote behaves similar to styling
substition[5]	'acronym', 'smiley', 'wordblock', 'entity', 'camelcaselink', 'internallink', 'media', 'externallink', 'linebreak', 'emaillink', 'windowssharelink', 'filelink', 'notoc', 'nocache', 'multiplyentity', 'quotes', 'rss'	modes where the token is simply replaced – they can not contain any other modes
protected	'preformatted', 'code', 'file', 'php', 'html'	modes which have a start and end token but inside which no other modes should be applied
disabled	unformatted	inside this mode no wiki markup should be applied but lineendings and whitespace isn't preserved
paragraphs	eol[6]	used to mark paragraph boundaries

For a description what each type means and which other formatting classes are registered in them read the comments in `inc/parser/parser.php`.

Tutorial: Syntax Plugins Explained

The goal of this tutorial is to explain the concepts involved in a DokuWiki syntax plugin and to go through the steps involved in writing your own plugin.

For those who are really impatient to get started, grab a copy of the syntax plugin skeleton. It's a bare bones plugin which outputs «*Hello World!*» when it encounters «<TEST>» on a wiki page.

Quick Summary

modes

- each individual piece of DokuWiki syntax, including your plugin, has its own mode.
- similar modes are grouped together into mode types.
- a mode's «allowedTypes» govern which other DokuWiki syntax is recognised when nested within the mode's own syntax. All the modes which belong to the allowedTypes will be permitted.
- a mode's «type» lets other modes know if they can permit this mode within their syntax.

handle

- the `handle()` method is called when the parser encounters wiki page content that it decides belongs to your syntax mode.
- the `$state` parameter says which type of pattern registered to your mode was triggered. If it's just ordinary text the state parameter will be set to `DOKU_LEXER_UNMATCHED`
- do as much processing and decision making as possible here, leaving as little as possible to be carried out in the `render()` method because the output of handle is cached. This also means that you shouldn't do any stuff here that mustn't be cached.

render

- The `render()` method processes the renderer instructions that apply to the plugin's syntax mode - and which were created by the plugin's `handle()` method.
- add content to the output document with `$renderer->doc .= 'content';`
- access the return value of handle() using the *$data* parameter of render($format, Doku_Renderer $renderer, $data).
- ensure any content output by the plugin is **safe** - run raw wiki data through an entity conversion function.
- do the minimum possible processing and decision making here, it should all have been done in the `handle()` method.

There is no guarantee the `render()` method will be called at the same time as the `handle()` method. The instructions generated by the handler are cached and can be used by the renderer at a future time. The only sure way to pass data from `handle()` to `render()` is using the array it returns - which is passed to `render()` as the `$data` parameter.

Key Concepts

modes

Modes (or more properly syntax modes) are the foundation on which the DokuWiki parser is based. Every different bit of DokuWiki markup has its own syntax mode. E.g. there is a strong mode for handling **strong**, a superscript mode for handling ^{superscript}, a table mode for processing tables and many more.

When the parser encounters some markup it enters the syntax mode for that markup. The properties and methods of that particular syntax mode govern how the parser behaves while it is within that mode, including:

- what other syntax modes are allowed to occur
- what instructions to prepare for the renderer

Your plugin will add its own syntax mode to the parser - that is automatically handled by DokuWiki when the plugin is first loaded, the name assigned is `plugin_`+ the name of the plugin's directory (which must also be the plugin's class name without the prefix «`syntax_`»). Then, when the parser encounters the markup used for your plugin, the parser will enter into that syntax mode. While it is in that mode your plugin controls what the parser can do.

mode types

To simplify things, syntax modes which behave in a similar manner have been grouped together into several mode types - a complete list can be found on the syntax plugin page.

Each mode type corresponds to a key in the `$PARSER_MODES` array. The entry for each mode type is itself an array which holds all the syntax modes which belong to that type. e.g. In vanilla DokuWiki with no plugins installed, `$PARSER_MODES['formatting']` holds an array containing: 'strong', 'emphasis', 'underline', 'superscript', 'subscript', 'monospace', 'deleted' & 'footnote'.

When each plugin is loaded into the parser it is queried, via `getType()`, to discover which mode type it will belong to. The syntax mode associated with the plugin is then added to the appropriate `$PARSER_MODES` array.


The mode type your plugin reports governs where in a DokuWiki page the parser will recognise your plugin's markup. Other DokuWiki (and plugin) syntax modes won't know about your plugin, but they do know about the different mode types. If they allow a particular mode type, they will allow all the modes which belong to that type, including any plugins that have returned that mode type.

Select the mode type for your plugin by comparing the behaviour of your plugin to that of the standard DokuWiki syntax modes. Choose the type that the most similar modes belong to.

allowed modes

These are the other modes that can occur nested within the current mode's own markup.

Each syntax mode has its own array of allowed modes which tells the parser what other syntax modes will be recognised whilst it is processing the mode. That is, if you want your plugin to be able to occur nested within «**strong**» markup, then the strong mode must include your plugin's mode in its allowedModes array. And if you want to allow strong markup nested within your plugin's markup then your plugin must have `'strong'` in its allowModes array.


Your plugin gets in the allowedModes array of other syntax modes through the mode type it reports using the `getType()` method.

Your plugin tells the parser which other syntax modes it permits by reporting the mode types it allows via the `getAllowedTypes()` method.

PType

PType governs how the parser handles html elements when dealing with your syntax mode.

Generally, when the parser encounters some markup, there will be a currently open HTML paragraph tag. The parser needs to know if it should close that tag before entering your syntax mode and then open another paragraph when exiting, that is `PType='block'` and `PType='stack'`, or whether it should leave the paragraphs alone, `PType='normal'`.

The PType also decides how and if paragraphs are created **inside** the syntax mode. With `PType='normal'` no paragraphs are created at all. `PType='stack'` opens a paragraph when inside the syntax mode (and closes it later, parsing paragraphs like usual). And `PType='block'` starts with no paragraph, but creates them as usual as soon as there are more than two newlines.

For those that know CSS, returning `PType='block'` and `PType='stack'` means the html generated by your plugin will be similar to `display:block` and returning `PType='normal'`means the HTML generated will be similar to `display:inline`.

Example

Suppose we have a fairly standard syntax plugin with the ENTRY ⇒ UNMATCHED ⇒ EXIT pattern. Depending on the PType setting, and will be inserted by the renderer automatically at various points outside, or even interspersed, with the plugin text. That means your plugin doesn't need to take care of those tags.

wikisyntax	PType=normal	PType=block	PType=stack

`foo`
`<plugin>text</plugin>`
 	
`bar`
	`<p>foo`
	`ENTRY("<plugin>")`
	`UNMATCHED("text")`
	`EXIT("</plugin>")`
	`</p>`
	`<p>bar</p>`
		`<p>foo</p>`
		`ENTRY("<plugin>")`
		`UNMATCHED("text")`
		`EXIT("</plugin>")`
		`<p>bar</p>`
			`<p>foo</p>`
			`ENTRY("<plugin>")`
			`<p>`
			`UNMATCHED("text")`
			`</p>`
			`EXIT("</plugin>")`
			`<p>bar</p>`

Sort Number

This number is used by the lexer[7] to control the order it tests the syntax mode patterns against raw wiki data. It is only important if the patterns belonging to two or more modes match the same raw data - where the pattern belonging to the mode with the lowest sort number will win out.

You can make use of this behaviour to write a plugin which will replace or extend a native DokuWiki handler for the same syntax. An example is the code plugin.

Details of existing sort numbers are available for both the parser (sort list).

Patterns

The parser uses PHP's preg[8)] compatible functions. A detailed explanation of regular expressions and their syntax is beyond the scope of this tutorial. There are many good sources on the web.

The complete preg syntax is not available for use in constructing syntax plugin patterns. Below is a list of the known differences:

- don't surround the pattern with delimiters
- to use a pipe «|» for multiple alternatives, make them a non-captured group, e.g. «`(?:cat|dog)`»
- be very wary of look behind assertions. The parser only attempts to match patterns on the next piece of «not yet matched» data. If you need to look behind to characters that have been involved in a previous pattern match, those characters will never be there.
- option flags can only be included as inline options, e.g. `(?i)`, `(?-i)`
- back references do not work, e.g. «`(\w)\1\w+`» (finding a word with a doubled first characters), due to the way the lexer functions internally.

The parser provides four functions for a plugin to register the patterns it needs. Each function corresponds to a pattern with a different meaning.

- **special patterns** — `addSpecialPattern()` — these are the patterns used when one pattern is all that is required. In the parser's terms, these patterns represent entry in the the plugin's syntax mode and exit from that syntax mode all in the one match. Typically these are used by `substition` plugins.
- **entry patterns** — `addEntryPattern()` — the pattern which indicates the start of data to be handled by the plugin. Typically these patterns should include a look-ahead to ensure there is also an exit pattern. Any plugin which registers an entry pattern should also register an exit pattern.
- **exit patterns** — `addExitPattern()` — the pattern which indicates the end of the data to be handled by the plugin. This pattern can only be matched if text matching the entry pattern has been found.
- **internal patterns** — `addPattern()` — these represent special syntax applicable to the plugin that may occur between the entry and exit patterns. Generally these are only required by the more complex structures, e.g. lists and tables.

One plugin may add several patterns to the parser, including more than one pattern of the same type.

Tips

- use non-greedy quantifiers, e.g. +? or *? instead of + or *.
- be wary of using multiple exit patterns. The first exit pattern encountered will most likely trigger the parser to exit your syntax mode - even if that wasn't the pattern the entry pattern looked ahead for. Needing multiple exit patterns probably indicates a need for multiple plugins.
- In the Plugin Survey of 2011 it looks like a majority of special patterns are either {{…}} (160 cases) or ~~…~~ (80 cases). A very common entry/exit pattern (231 plugins) is something like an XML tag even if some use upper case letters.
- early versions of the DokuWiki lexer had a bug which prevented use of «<» or «>» in look

ahead patterns. This bug has been fixed and angle brackets can now be used. Some plugins will still contain the hex codes for angle brackets («\x3C», «\x3E») which was the workaround to overcome the effects of this bug.
- Use this for a example of correct regular expression: Use correct regular expressions

handle() method

This is the part of your plugin which should do all the work. Before DokuWiki renders the wiki page it creates a list of instructions for the renderer. The plugin's `handle()` method generates the render instructions for the plugin's own syntax mode. At some later time, these will be interpreted by the plugin's `render()` method. The instruction list is cached and can be used many times, making it sensible to maximize the work done once by this function and minimize the work done many times by `render()`.

The complete signature is: `public function handle($match, $state, $pos, Doku_Handler $handler)` with the arguments:

$match parameter — The text matched by the patterns, or in the case of **DOKU_LEXER_UNMATCHED** the contiguous piece of ordinary text which didn't match any pattern.

$state parameter — The lexer state for the match, representing the type of pattern which triggered this call to handle():

- **DOKU_LEXER_ENTER** — a pattern set by addEntryPattern()
- **DOKU_LEXER_MATCHED** — a pattern set by addPattern()
- **DOKU_LEXER_EXIT** — a pattern set by addExitPattern()
- **DOKU_LEXER_SPECIAL** — a pattern set by addSpecialPattern()
- **DOKU_LEXER_UNMATCHED** — ordinary text encountered within the plugin's syntax mode which doesn't match any pattern.

$pos parameter — The character position of the matched text.

$handler parameter — Object Reference to the Doku_Handler object.

return — The instructions for the `render()` method. These instructions are cached. The return value can be everything you require for your needs. Often, it is an array in which the different values are collected that are founded or determined in handle() and which are useful in `render()`.

render() method

The part of the plugin that provides the output for the final web page - or whatever other output format is supported. It is here that the plugin adds its output to that already generated by other parts of the renderer - e.g. by concatenating its output to the renderer's `doc` property.

```
$renderer->doc .= "some plugin output...";
```

Any raw wiki data that passes through `render()` should have all special characters converted to HTML entities. You can use DokuWiki's hsc() or the PHP functions, htmlspecialchars(),

htmlentities() or the renderer's own _xmlEntities() method. e.g.

```
$renderer->doc .= $renderer->_xmlEntities($text);
```

The complete signature is: `public function render($format, Doku_Renderer $renderer, $data)` with the arguments:

$format parameter — Name for the format mode of the final output produced by the renderer. At present DokuWiki only supports one output format XHTML and a special (internal) format `metadata` [9]. New modes can be introduced by renderer plugins. The plugin should only produce output for those formats which it supports - which means this function should be structured ...

```
if ($format == 'xhtml') {  // supported mode
  // code to generate XHTML output from instruction $data
}
```

$renderer parameter — Give access to the object Doku_Renderer, which contains useful functions and values. Above you saw already the usage of `$renderer->doc` for storing the render output.

$data parameter — An array containing the instructions previously prepared and returned by the plugin's own `handle()` method. The `render()` must interpret the instruction and generate the appropriate output.

XHTML renderer

When your plugin needs to extend the content of a wiki page, you need the output format mode `xhtml`. Because `render()` is called for all the format modes, you need to filter by the desired modes.

```
if ($format == 'xhtml') {  // when the format mode is xhtml
    /** @var Doku_Renderer_xhtml $renderer */
    // code to generate XHTML output from instruction $data
    $renderer->doc .= '<div>Adds your div</div>';
}
```

Detail: the variable `$renderer` is now the Doku_Renderer_xhtml object.

Metadata renderer

A special render format `metadata` is for rendering metadata. Metadata are the extra properties kept for your wiki page, which you can also extend or modify in your plugin.

In the metadata rendering format you extracts metadata from the page. This is particularly important if you manually handle certain kinds of links. If you don't register these, they will not show up as backlinks on the pages that they refer to. Here is an example of how to register these backlinks:

```
public function render($format, Doku_Renderer $renderer, $data) {
    if($format == 'xhtml') {
        /** @var Doku_Renderer_xhtml $renderer */
```

```
        // this is where you put all the rendering that will be displayed in
the
        // web browser
    return true;
    }
    if($format == 'metadata') {
        /** @var Doku_Renderer_metadata $renderer */
        $renderer->internallink($data[0]);
        // I am assuming that when processing in handle(), you have stored
        // the link destination in $data[0]
        return true;
    }
    return false;
}
```

This example uses the internallink() function from `inc/parser/metadata.php`. You can also access the metadata directly in the renderer with `$renderer->meta` and `$renderer->persistent`, because `$renderer` is now the Doku_Renderer_metadata object. Here is a snippet from the tag plugin:

```
public function render($format, Doku_Renderer $renderer, $data) {
    if ($data === false) return false;

    // XHTML output
    if ($format == 'xhtml') {
        /** @var Doku_Renderer_xhtml $renderer */
        ...

    // for metadata renderer
    } elseif ($format == 'metadata') {
        /** @var Doku_Renderer_metadata $renderer */
        // erase tags on persistent metadata no more used
        if (isset($renderer->persistent['subject'])) {
            unset($renderer->persistent['subject']);
            $renderer->meta['subject'] = [];
        }

        // merge with previous tags and make the values unique
        if (!isset($renderer->meta['subject'])) {
            $renderer->meta['subject'] = [];
        }
        $renderer->meta['subject'] =
array_unique(array_merge($renderer->meta['subject'], $data));

        // create raw text summary for the page abstract
        if ($renderer->capture) {
            $renderer->doc .= implode(' ', $data);
        }

        ...
        return true;
```

```
    }
    return false;
}
```

First it handles old persistent metadata no longer used by this plugin. This persistent metadata is always kept, thus when you change your mind and use current metadata instead, you need to remove it explicitly.

When handling persistent data in the metadata renderer, take care you update also the current metadata, when you update persistent metadata.

The tag plugin stores here 'subject' data by `$renderer->meta['subject'] = …`. Be aware that when you use `p_set_metadata` to set current metadata somewhere, that the next time the metadata is rendered it will overwrite this data. Using p_get_metadata($ID, $key) gives access to stored metadata. For details see metadata.

When some raw text from your syntax should be included in the abstract you can append it to `$renderer->doc`. When the abstract is long enough, `$renderer->capture` becomes false.

The xhtml mode is called when DokuWiki is in need of a new xhtml version of the wikipage. The metadata is a bit different. In general, the metadata of the page is rendered on demand when p_get_metadata() is called somewhere.

When someone edit a page and use the preview function, the metadata renderer is not called. So the metadata is not yet updated! This is done when the page is saved.

Safety & Security

Raw wiki page data which reaches your plugin has not been processed at all. No further processing is done on the output after it leaves your plugin. At an absolute minimum the plugin should ensure any raw data output has all HTML special characters converted to HTML entities. Also any wiki data extracted and used internally should be treated with suspicion. See also security.

Common plugin functions

Some function are shared between the plugins, refer to next sections for info:

- Plugin configuration settings
- Localization
- Using styles and javascript

Adding a Toolbar Button

To make it easy on the users of wikis which install your plugin, you should add a button for its syntax to the editor toolbar.

- The toolbar page explains how you can extend by PHP or javascript.

- Another example is available at the Action Plugin page.

Writing Your Own Plugin

Ok, so you have decided you want to extend DokuWiki's syntax with your own plugin. You have worked out what that syntax will be and how it should be rendered on the user's browser. Now you need to write the plugin.

1. Decide on a name for the plugin. You may want to check the list of available plugins to make sure you aren't choosing a name that is already in use.
2. In your own DokuWiki installation, create a new sub directory in the `lib/plugins/` directory. That directory will have the same name as your plugin.
3. Create the file `syntax.php` in the new directory. As a starting point, use a copy of the skeleton plugin.
4. Edit that file to make it yours.
    - change the class name to be `syntax_plugin_<your plugin name>`[10].
    - change the `getType()` method to report the mode type your plugin will belong to.
    - add a `getAllowedTypes()` method to report any mode types your plugin will allow to be nested within its own syntax. If your plugin won't allow any other mode then this can be left out.
    - change the `getPType()` method to report the PType that will apply for your plugin. If its `'normal'` you can remove this method.
    - change the `getSort()` method to report a unique number after checking the getsorted list and
    - alter the `connectTo()` method to register the pattern to match your syntax.
    - add a `postConnect()` method if your syntax has an second pattern to say when the parser is leaving your syntax mode.
5. That's the easy part done, you now have a plugin that will say «Hello World!» when it encounters your syntax pattern. Time to test it and make sure the pattern works as expected - visit your wiki and make up a page with the syntax for your plugin, save it and make sure «Hello World!» shows up.
6. Write your `handle()` & `render()` methods.
    - if you have entry and exit patterns remember to handle the unmatched data.
    - treat raw wiki data with suspicion and remember to ensure all special characters go to an entity converter.
7. Add a plugin.info.txt file in your plugin directory (see for example, the sample plugin below)
8. Test and post your completed plugin on the DokuWiki plugin page.

Read also

- The Plugin Wizard can create a basic skeleton or with the dev plugin.
- Plugin file structure
- Common plugin functions
- Plugin programming tips
- Plugin Development

Sample Plugin 1 - Now

When its syntax, `[NOW]`, is encountered in a wiki page the current date and time will be inserted in RFC2822 format.

- type is `'substition'`. We are substituting a time stamp for the `[NOW]` token, similar to the way smileys and acronyms are handled. They belong to the mode type `'substition'` so we will too.
- allowedTypes are not required, no other DokuWiki syntax can occur within our `[NOW]` syntax. Therefore we don't need the `getAllowedTypes()` method.
- PType is `normal`, that's the default value, so we don't need the `getPType()` method.
- there is no need for an entry and exit pattern, just a special pattern to detect `[NOW]`. The only thing we need to be careful of is «[» and «]» have special meanings in regular expressions, so we will need to *escape* them, making our pattern - `'\[NOW\]'`.
- in this case the `handler()` method doesn't need to do anything. We have no special states to take care of or extra parameters in our syntax. We just return an empty array to ensure a render instruction for our plugin is stored.
- all the `render()` method needs to do is add the time stamp to the current wiki page — `$renderer->doc .= date('r');`

And that's our plugin finished.

syntax.php

```php
<?php
/**
 * Plugin Now: Inserts a timestamp.
 *
 * @license    GPL 2 (http://www.gnu.org/licenses/gpl.html)
 * @author     Christopher Smith <chris@jalakai.co.uk>
 */

// must be run within DokuWiki
if(!defined('DOKU_INC')) die();

/**
 * All DokuWiki plugins to extend the parser/rendering mechanism
 * need to inherit from this class
 */
class syntax_plugin_now extends DokuWiki_Syntax_Plugin {

    public function getType() { return 'substition'; }
    public function getSort() { return 32; }

    public function connectTo($mode) {
        $this->Lexer->addSpecialPattern('\[NOW\]',$mode,'plugin_now');
    }

    public function handle($match, $state, $pos, Doku_Handler $handler)
{
```

```
        return array($match, $state, $pos);
    }

    public function render($format, Doku_Renderer $renderer, $data) {
    // $data is what the function handle return'ed.
        if($format == 'xhtml'){
            /** @var Doku_Renderer_xhtml $renderer */
            $renderer->doc .= date('r');
            return true;
        }
        return false;
    }
}
```

You also need the plugin.info.txt file:

plugin.info.txt

```
base now
author me
email me@someplace.com
date 2005-07-28
name Now Plugin
desc Include the current date and time
url https://www.dokuwiki.org/devel:syntax_plugins
```

Note: due to the way DokuWiki caches pages this plugin will report the date/time at which the cached version was created. You would need to add ~~NOCACHE~~ to the page to ensure the date was current every time the page was requested.

Sample Plugin 2 - Color

When its syntax, `<color somecolour/somebackgroundcolour>`, is encountered in a wiki page the text colour will be changed to somecolour, the background will be changed to somebackgroundcolour and both will remain that way until `</color>` is encountered.

- what we are doing is similar to the strong mode, its type is 'formatting' so we should use that type too.
- allowedTypes should be the inline modes - `substition`, `formatting` & `disabled`.
- PType is `normal`, that's the default value, so again we don't need a `getPType()` method.
- we need to use an entry and exit pattern. The entry pattern should check to make sure there is an exit pattern, which means `'<color.*>(?=.*?</color>)'`. The exit pattern is simpler, `</color>`.
- the `handle()` method will need to deal with three states matching our entry and exit patterns and unmatched for the text which occurs between them.
    - DOKU_LEXER_ENTER state requires some processing to extract the colour and background colour values, they make up our render instruction.

    - DOKU_LEXER_UNMATCHED state doesn't require any processing, but we have to pass the unmatched text (in `$match`) to `render()` so that goes into our render instruction.
    - DOKU_LEXER_EXIT state doesn't require any processing or have any special data, we simply need to generate an exit instruction for `render()`.
- the `render()` method will need to deal with the same three states as `handle()`.
    - DOKU_LEXER_ENTER, open a span with a style using the colour and/or background colour values.
    - DOKU_LEXER_UNMATCHED, add the unmatched text to the output document.
    - DOKU_LEXER_EXIT, close the span

Put the file syntax.php from below into a folder named «color» directly below your plugins folder, e.g. /srv/www/htdocs/dokuwiki/lib/plugins. If you do not name this folder «color», the plugin will not work:

syntax.php

```
<?php
/**
 * Plugin Color: Sets new colors for text and background.
 *
 * @license    GPL 2 (http://www.gnu.org/licenses/gpl.html)
 * @author     Christopher Smith <chris@jalakai.co.uk>
 */

// must be run within Dokuwiki
if(!defined('DOKU_INC')) die();

/**
 * All DokuWiki plugins to extend the parser/rendering mechanism
 * need to inherit from this class
 */
class syntax_plugin_color extends DokuWiki_Syntax_Plugin {

    public function getType(){ return 'formatting'; }
    public function getAllowedTypes() { return array('formatting',
'substition', 'disabled'); }
    public function getSort(){ return 158; }
    public function connectTo($mode) {
$this->Lexer->addEntryPattern('<color.*?>(?=.*?</color>)',$mode,'plugin
_color'); }
    public function postConnect() {
$this->Lexer->addExitPattern('</color>','plugin_color'); }


    /**
     * Handle the match
     */
    public function handle($match, $state, $pos, Doku_Handler
$handler){
        switch ($state) {
          case DOKU_LEXER_ENTER :
                list($color, $background) = preg_split("/\//u",
```

```
substr($match, 6, -1), 2);
                if ($color = $this->_isValid($color)) $color =
"color:$color;";
                if ($background = $this->_isValid($background))
$background = "background-color:$background;";
                return array($state, array($color, $background));

          case DOKU_LEXER_UNMATCHED :  return array($state, $match);
          case DOKU_LEXER_EXIT :       return array($state, '');
        }
        return array();
    }

    /**
     * Create output
     */
    public function render($format, Doku_Renderer $renderer, $data) {
        // $data is what the function handle() return'ed.
        if($format == 'xhtml'){
            /** @var Doku_Renderer_xhtml $renderer */
            list($state,$match) = $data;
            switch ($state) {
                case DOKU_LEXER_ENTER :
                    list($color, $background) = $match;
                    $renderer->doc .= "<span style='$color
$background'>";
                    break;

                case DOKU_LEXER_UNMATCHED :
                    $renderer->doc .= $renderer->_xmlEntities($match);
                    break;
                case DOKU_LEXER_EXIT :
                    $renderer->doc .= "</span>";
                    break;
            }
            return true;
        }
        return false;
    }

    /**
     * Validate color value $c
     * this is cut price validation - only to ensure the basic format
is correct and there is nothing harmful
     * three basic formats  "colorname", "#fff[fff]",
"rgb(255[%],255[%],255[%])"
     */
    private function _isValid($c) {
        $c = trim($c);

        $pattern = "/^\s*(
```

```
            ([a-zA-z]+)|                                    #colorname -
not verified
            (\#([0-9a-fA-F]{3}|[0-9a-fA-F]{6}))|            #colorvalue
            (rgb\(([0-9]{1,3}%?,){2}[0-9]{1,3}%?\))         #rgb triplet
            )\s*$/x";

        if (preg_match($pattern, $c)) return trim($c);

        return "";
    }
}
```

Note: No checking is done to ensure colour names are valid or RGB values are within correct ranges.

Unit Testing Syntax Plugins

For a general introduction about Unit Testing in DokuWiki please see unittesting. For a syntax plugin a common test goal will be to ensure that a certain wiki code produces the expected XHTML code or other destination language code.

The following example function shows a simple way to do this:

```
    public function test_superscript() {
        $info = [];
        $expected = "\n<p>\nThis is <sup>superscripted</sup> text.<br
/>\n</p>\n";

        $instructions = p_get_instructions('This is ^^superscripted^^
text.');
        $xhtml = p_render('xhtml', $instructions, $info);

        $this->assertEquals($expected, $xhtml);
    }
```

Here we strongly benefit from DokuWiki's good design. The two function calls to `p_get_instructions()` and `p_render()` are enough to render the example code `//'This is ^^superscripted^^ text.'//` and store the result in the variable **$xhtml**. Finally we only need a simple assert to check if the result is what we **$expected**.

Дополнения и Файлы

Ссылка на оригинал статьи

1)
defined in `lib/Extension/SyntaxPlugin.php`, before called `DokuWiki_Syntax_Plugin` which is still available as alias

2)

defined in `inc/Parsing/ParserMode/AbstractMode.php`, inherited via dokuwiki\Parsing\ParserMode\Plugin

3)

See `Doku_Handler_Block`

4)

defined in `inc/Parsing/ParserMode/AbstractMode.php`

5)

Yes this is spelled wrong, but we won't change it to avoid breaking existing plugins. Sometimes a typo becomes a standard - see the HTTP «referer» header for an example

6)

This is actually a class, it does not mean «end of life», but «end of line».

7)

the part of the parser which analyses the raw wiki page

8)

perl compatible regular expressions
ref: www.php.net/manual/en/ref.pcre.php

9)

The special mode `metadata` does not output anything but collects metadata for the page. Plugin can add other formats such as the ODT format. Use it to insert values into the metadata array. See the translation plugin for an example.

10)

The name may not contain underscores and needs to match your class name