PHPTAL

Template Attribute Language for PHP

2008

I’m here to talk about PHPTAL, which is an implementation of a very interesting language - TAL.

XML template engine

TAL is a template language built around XML.

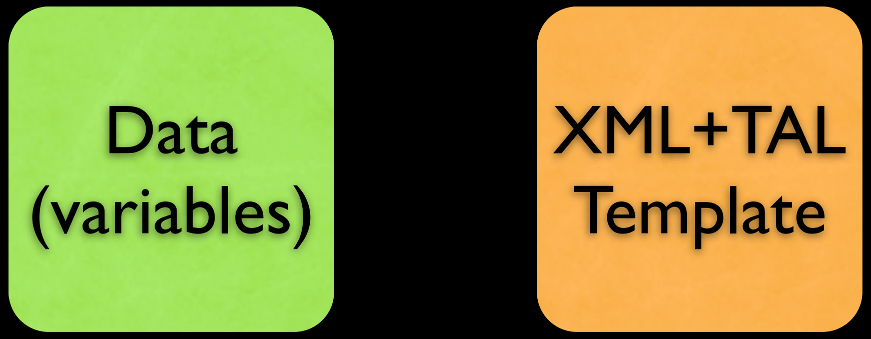
It takes TAL template written in XML, mixes it with data (variables in PHP implementation) and produces neat well-formed XHTML, or

XML template engine

TAL is a template language built around XML.

It takes TAL template written in XML, mixes it with data (variables in PHP implementation) and produces neat well-formed XHTML, or

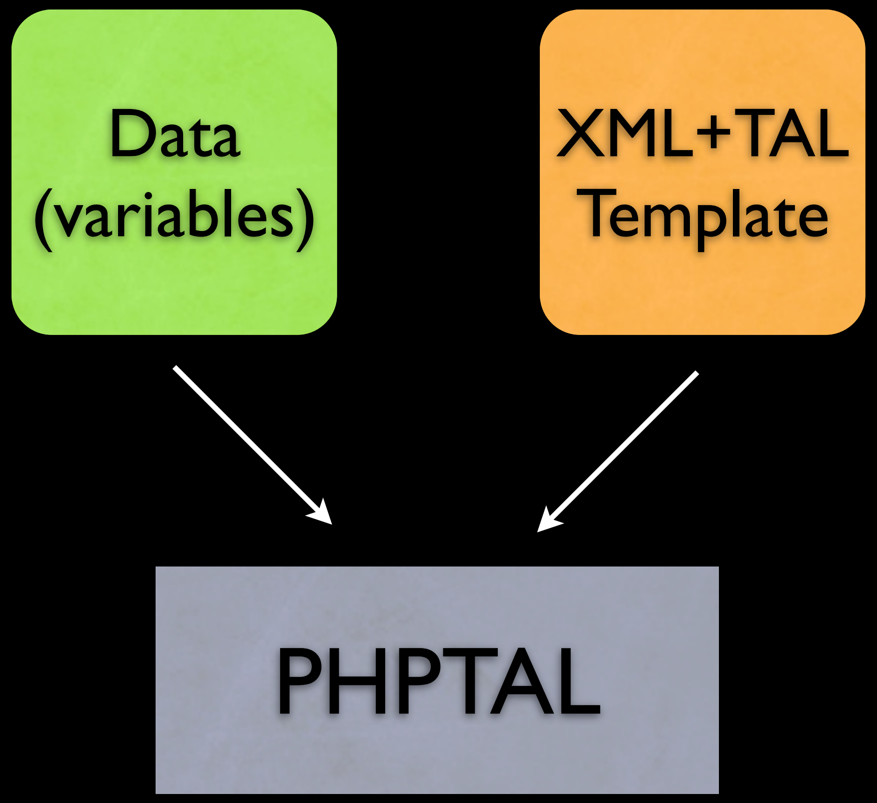
XML template engine



TAL is a template language built around XML.

It takes TAL template written in XML, mixes it with data (variables in PHP implementation) and produces neat well-formed XHTML, or

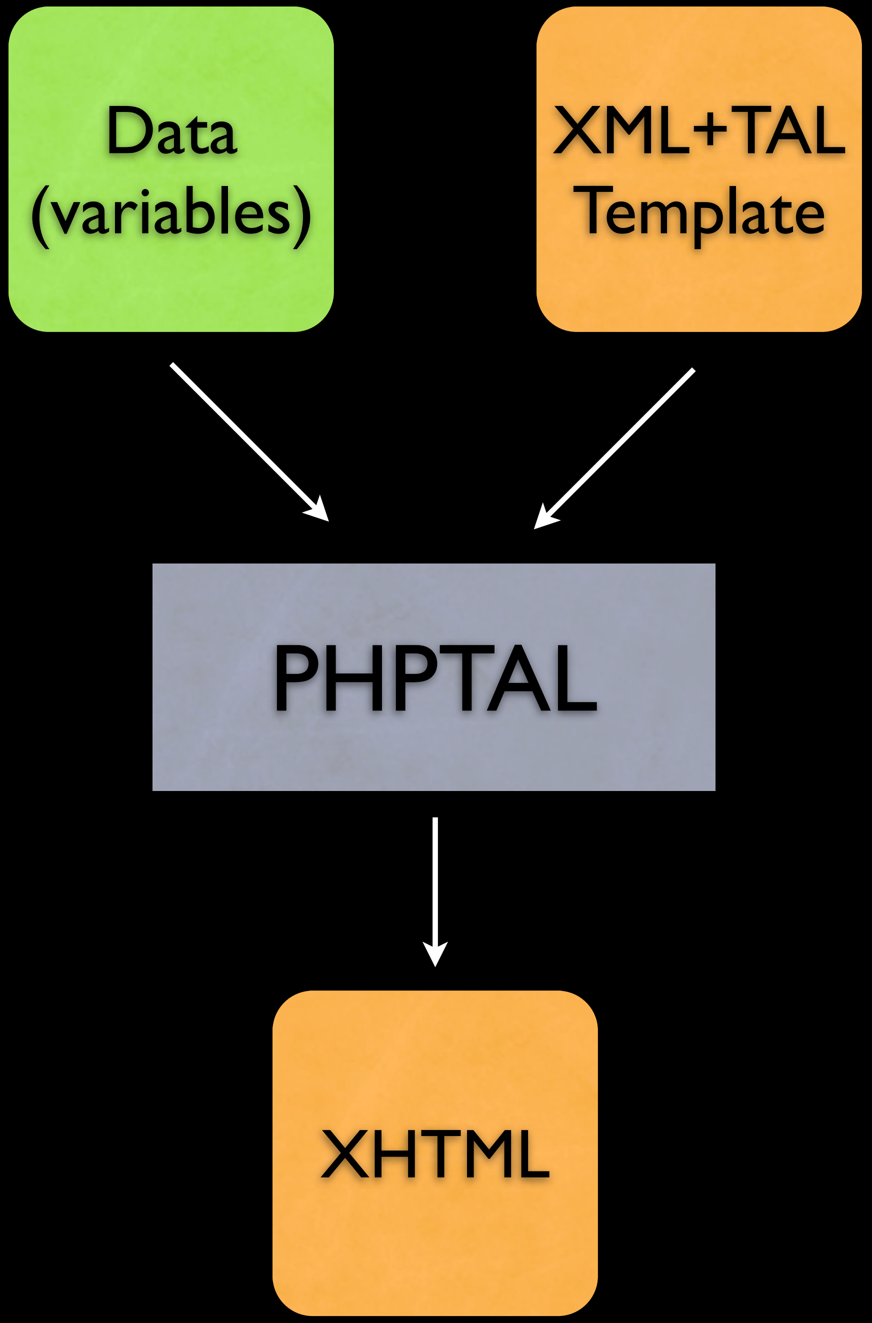
XML template engine



TAL is a template language built around XML.

It takes TAL template written in XML, mixes it with data (variables in PHP implementation) and produces neat well-formed XHTML, or

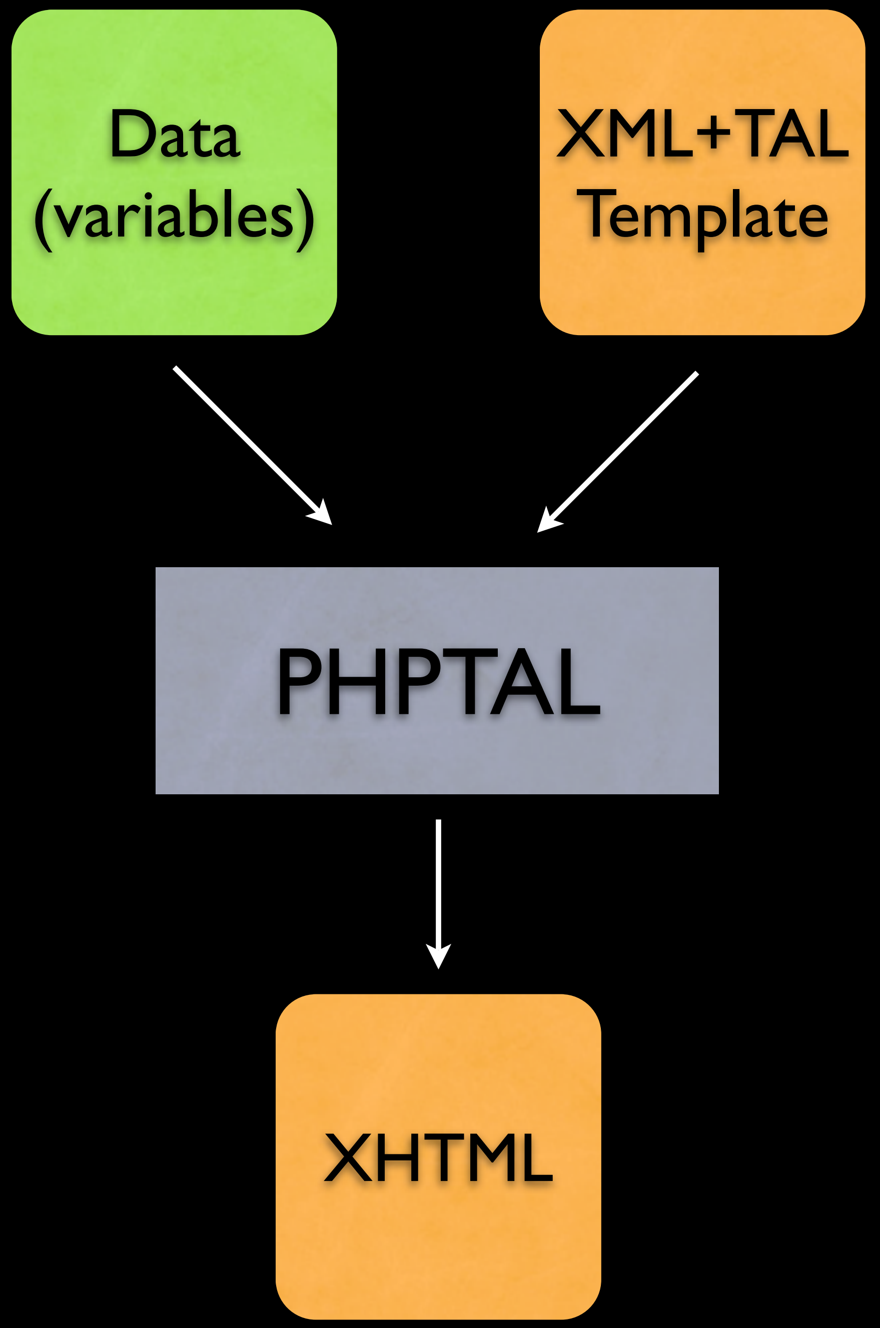
XML template engine



TAL is a template language built around XML.

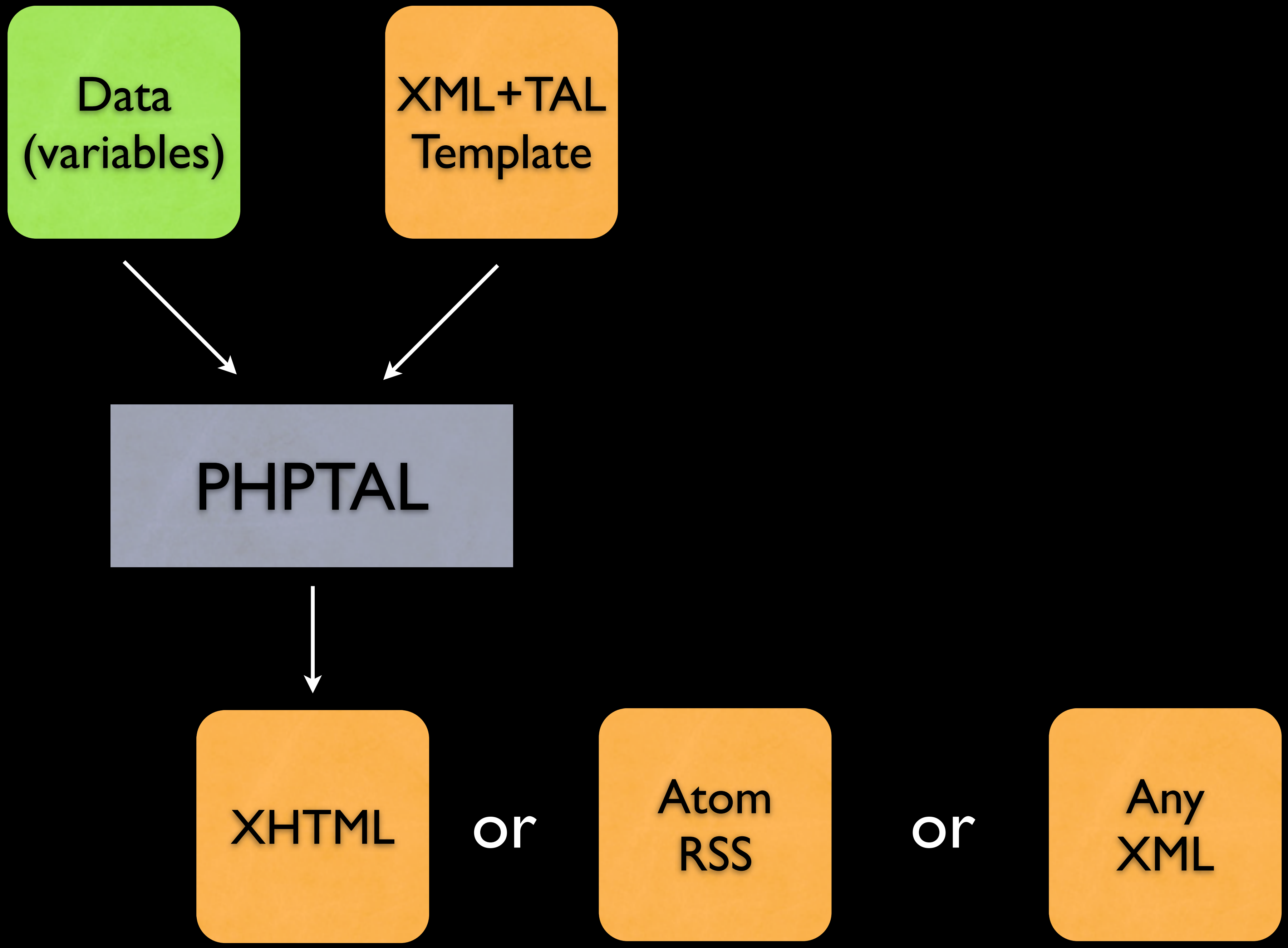
It takes TAL template written in XML, mixes it with data (variables in PHP implementation) and produces neat well-formed XHTML, or

XML template engine



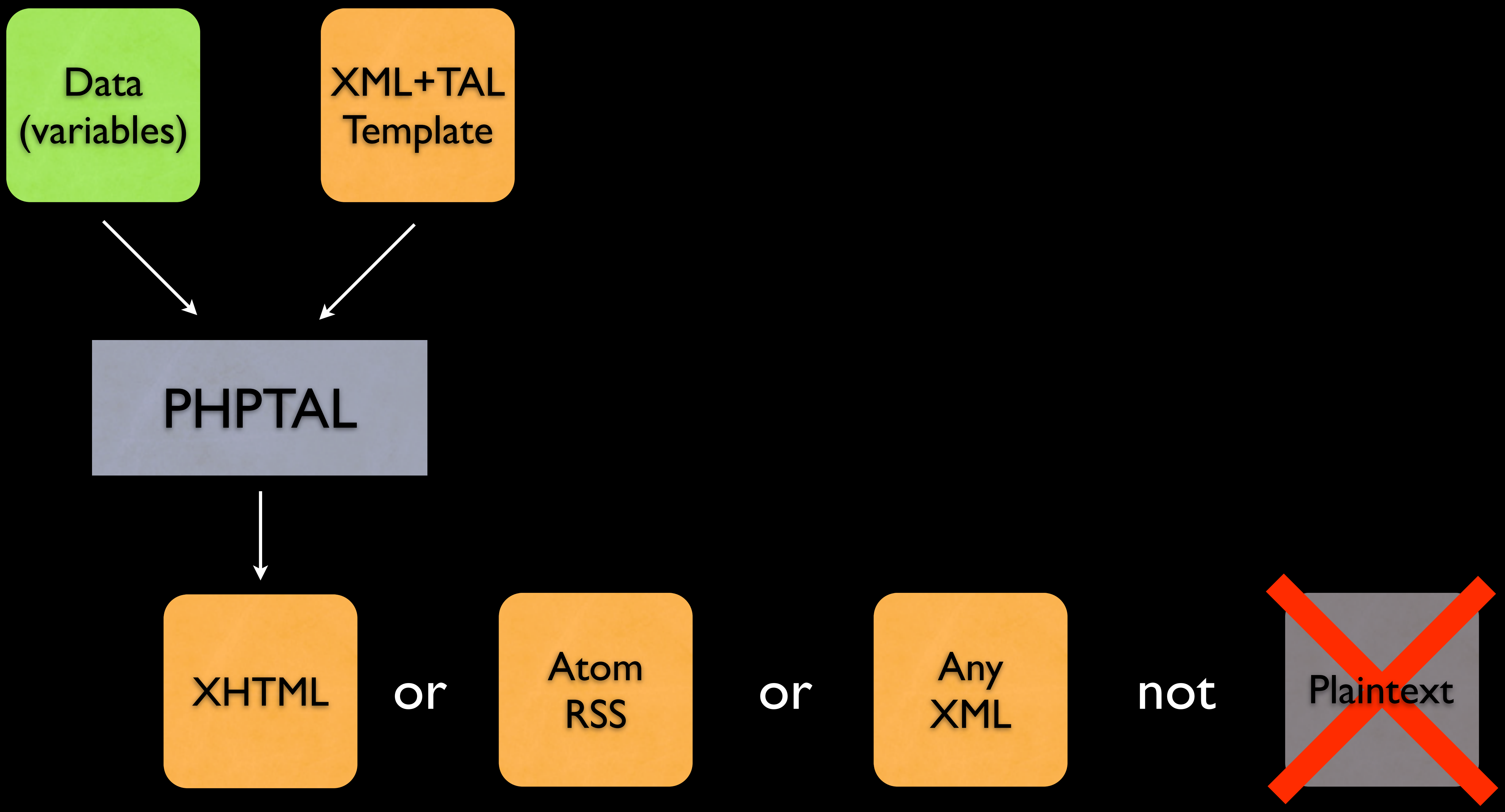
or Atom/RSS feeds, SVG or anything that’s XML. It’s all built around XML, and one thing it fails at is plain text.

XML template engine



or Atom/RSS feeds, SVG or anything that’s XML. It’s all built around XML, and one thing it fails at is plain text.

XML template engine



or Atom/RSS feeds, SVG or anything that's XML. It's all built around XML, and one thing it fails at is plain text.

There are countless template engines. Everybody invents their own, however most remain as one man’s toy, and some become de-facto standards.

What’s remarkable about TAL is pretty close to being a true web standard:
- there is an open, free specification written by the Zope community (Zope is a CMS in Python)
- there are multiple independent implementations: a few rewrites in Python, Java, two in Perl, PHP version and there’s even a XSLT spreadsheet that transforms TAL into XSLT (BTW: TAL feels like imperative language and it’s interesting that it can be translated into declarative functional language).

TAL is a web standard
(sort of)

There are countless template engines. Everybody invents their own, however most remain as one man's toy, and some become de-facto standards.

What's remarkable about TAL is pretty close to being a true web standard:
- there is an open, free specification written by the Zope community (Zope is a CMS in Python)
- there are multiple independent implementations: a few rewrites in Python, Java, two in Perl, PHP version and there's even a XSLT spreadsheet that transforms TAL into XSLT (BTW: TAL feels like imperative language and it's interesting that it can be translated into declarative functional language).

TAL is a web standard
(sort of)

Official **specification**

http://wiki.**zope.org**/ZPT/TAL

There are countless template engines. Everybody invents their own, however most remain as one man's toy, and some become de-facto standards.

What's remarkable about TAL is pretty close to being a true web standard:
- there is an open, free specification written by the Zope community (Zope is a CMS in Python)
- there are multiple independent implementations: a few rewrites in Python, Java, two in Perl, PHP version and there's even a XSLT spreadsheet that transforms TAL into XSLT (BTW: TAL feels like imperative language and it's interesting that it can be translated into declarative functional language).

TAL is a web standard
(sort of)

Official **specification**

http://wiki.**zope.org**/ZPT/TAL

Multiple **implementations**

http://en.wikipedia.org/wiki/**Template_Attribute_Language**

There are countless template engines. Everybody invents their own, however most remain as one man's toy, and some become de-facto standards.

What's remarkable about TAL is pretty close to being a true web standard:
- there is an open, free specification written by the Zope community (Zope is a CMS in Python)
- there are multiple independent implementations: a few rewrites in Python, Java, two in Perl, PHP version and there's even a XSLT spreadsheet that transforms TAL into XSLT (BTW: TAL feels like imperative language and it's interesting that it can be translated into declarative functional language).

Why use templates at all?

If you’ve ever had to change layout of osCommerce store than you know what I’m talking about – if you start moving HTML around you may accidentally remove some security check or break SQL code. You can introduce critical bugs when just changing how it looks!

The solution to this mess is of course separation of HTML and code.

If you do that, you get extra bonus – you can have different presentations of the data: XHTML, feeds, AJAX and XML RPC APIs. These all happen to be XML, so you can implement that by just swapping one PHPTAL template with another! (PHP doesn’t need help with JSON)

- No messy code like osCommerce

If you’ve ever had to change layout of osCommerce store than you know what I’m talking about – if you start moving HTML around you may accidentally remove some security check or break SQL code. You can introduce critical bugs when just changing how it looks!

The solution to this mess is of course separation of HTML and code.

If you do that, you get extra bonus – you can have different presentations of the data: XHTML, feeds, AJAX and XML RPC APIs. These all happen to be XML, so you can implement that by just swapping one PHPTAL template with another! (PHP doesn’t need help with JSON)

- No messy code like osCommerce
- Presentation of data is less dependent on rest of the program

If you’ve ever had to change layout of osCommerce store than you know what I’m talking about – if you start moving HTML around you may accidentally remove some security check or break SQL code. You can introduce critical bugs when just changing how it looks!

The solution to this mess is of course separation of HTML and code.

If you do that, you get extra bonus – you can have different presentations of the data: XHTML, feeds, AJAX and XML RPC APIs. These all happen to be XML, so you can implement that by just swapping one PHPTAL template with another! (PHP doesn’t need help with JSON)

- No messy code like osCommerce
- Presentation of data is less dependent on rest of the program
- You can easily have different presentations of the data (XHTML, feeds, AJAX, REST API)

If you've ever had to change layout of osCommerce store than you know what I'm talking about – if you start moving HTML around you may accidentally remove some security check or break SQL code. You can introduce critical bugs when just changing how it looks!

The solution to this mess is of course separation of HTML and code.

If you do that, you get extra bonus – you can have different presentations of the data: XHTML, feeds, AJAX and XML RPC APIs. These all happen to be XML, so you can implement that by just swapping one PHPTAL template with another! (PHP doesn't need help with JSON)

“but PHP *is* a template language!”

PHP started as a template language, so why not use it then? Well,

There's something wrong with it.

PHP has old, simplistic approach to generation of HTML – just glue together bits of text, everything is just a flat array of characters. But HTML isn't really a text, it's a tree of nodes, with clear separation between elements, attributes and text nodes. And PHP is completely unaware of that. In fact it turns out to be optimized for plain text rather than HTML. In PHP the easiest thing to output is a messy tag soup that's barely an HTML. Valid, well-formed HTML is something you have to specially care about.

and even if you were to glue HTML from bits of text, the syntax could be better. You have to keep jumping back and forth between HTML and PHP mode, and PHP's angle brackets look awkward inside HTML attributes.

PHP as a programming language improved a lot over the years, but the templating part haven't changed much. It actually became harder to use, because short open tag syntax became deprecated!

Simple example:

- PHP is based around old concept

There's something wrong with it.

PHP has old, simplistic approach to generation of HTML – just glue together bits of text, everything is just a flat array of characters. But HTML isn't really a text, it's a tree of nodes, with clear separation between elements, attributes and text nodes. And PHP is completely unaware of that. In fact it turns out to be optimized for plain text rather than HTML. In PHP the easiest thing to output is a messy tag soup that's barely an HTML. Valid, well-formed HTML is something you have to specially care about.

and even if you were to glue HTML from bits of text, the syntax could be better. You have to keep jumping back and forth between HTML and PHP mode, and PHP's angle brackets look awkward inside HTML attributes.

PHP as a programming language improved a lot over the years, but the templating part haven't changed much. It actually became harder to use, because short open tag syntax became deprecated!

Simple example:

- PHP is based around old concept
- PHP's implementation and design aren't great

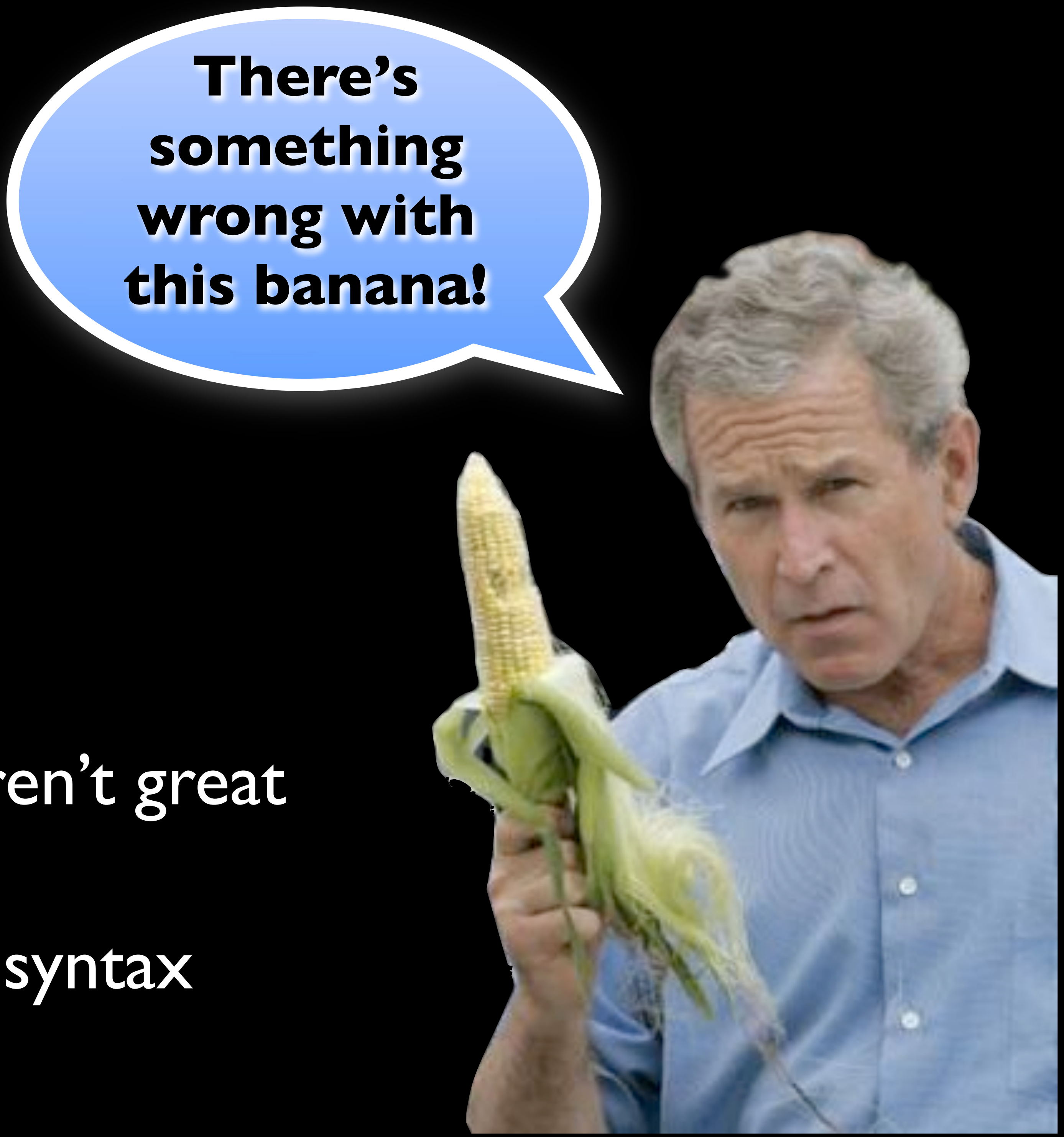
There's something wrong with it.

PHP has old, simplistic approach to generation of HTML – just glue together bits of text, everything is just a flat array of characters. But HTML isn't really a text, it's a tree of nodes, with clear separation between elements, attributes and text nodes. And PHP is completely unaware of that. In fact it turns out to be optimized for plain text rather than HTML. In PHP the easiest thing to output is a messy tag soup that's barely an HTML. Valid, well-formed HTML is something you have to specially care about.

and even if you were to glue HTML from bits of text, the syntax could be better. You have to keep jumping back and forth between HTML and PHP mode, and PHP's angle brackets look awkward inside HTML attributes.

PHP as a programming language improved a lot over the years, but the templating part haven't changed much. It actually became harder to use, because short open tag syntax became deprecated!

Simple example:



- PHP is based around old concept
- PHP's implementation and design aren't great
- Unlike other features, the template syntax hadn't improved

There's something wrong with it.

PHP has old, simplistic approach to generation of HTML – just glue together bits of text, everything is just a flat array of characters. But HTML isn't really a text, it's a tree of nodes, with clear separation between elements, attributes and text nodes. And PHP is completely unaware of that. In fact it turns out to be optimized for plain text rather than HTML. In PHP the easiest thing to output is a messy tag soup that's barely an HTML. Valid, well-formed HTML is something you have to specially care about.

and even if you were to glue HTML from bits of text, the syntax could be better. You have to keep jumping back and forth between HTML and PHP mode, and PHP's angle brackets look awkward inside HTML attributes.

PHP as a programming language improved a lot over the years, but the templating part haven't changed much. It actually became harder to use, because short open tag syntax became deprecated!

Simple example:

Super-basic use-case, just output variable. And that's easy to get wrong! If you do it like that, chances are you'll generate an ill-formed document (if $world contains &) or worse, make site vulnerable to XSS (if someone puts <script> in there)

So you have to escape every single bit of text and name of the function that does that couldn't be longer. It's ridiculous!

Filtering of all HTML on input isn't a solution - PHP went that path with magic_quotes. We don't want magic_entities!

```
<h1>Hello <?php echo $world; ?></h1>
```

Super-basic use-case, just output variable. And that's easy to get wrong! If you do it like that, chances are you'll generate an ill-formed document (if $world contains &) or worse, make site vulnerable to XSS (if someone puts <script> in there)

So you have to escape every single bit of text and name of the function that does that couldn't be longer. It's ridiculous!

Filtering of all HTML on input isn't a solution - PHP went that path with magic_quotes. We don't want magic_entities!

ill-formed
& vulnerable

```
<h1>Hello <?php echo $world; ?></h1>
```

Super-basic use-case, just output variable. And that’s easy to get wrong! If you do it like that, chances are you’ll generate an ill-formed document (if $world contains &) or worse, make site vulnerable to XSS (if someone puts <script> in there)

So you have to escape every single bit of text and name of the function that does that couldn’t be longer. It’s ridiculous!

Filtering of all HTML on input isn’t a solution – PHP went that path with magic_quotes. We don’t want magic_entities!

ill-formed
& vulnerable

```
<h1>Hello <?php echo $world; ?></h1>
<h1>Hello <?php echo htmlspecialchars($world); ?></h1>
```

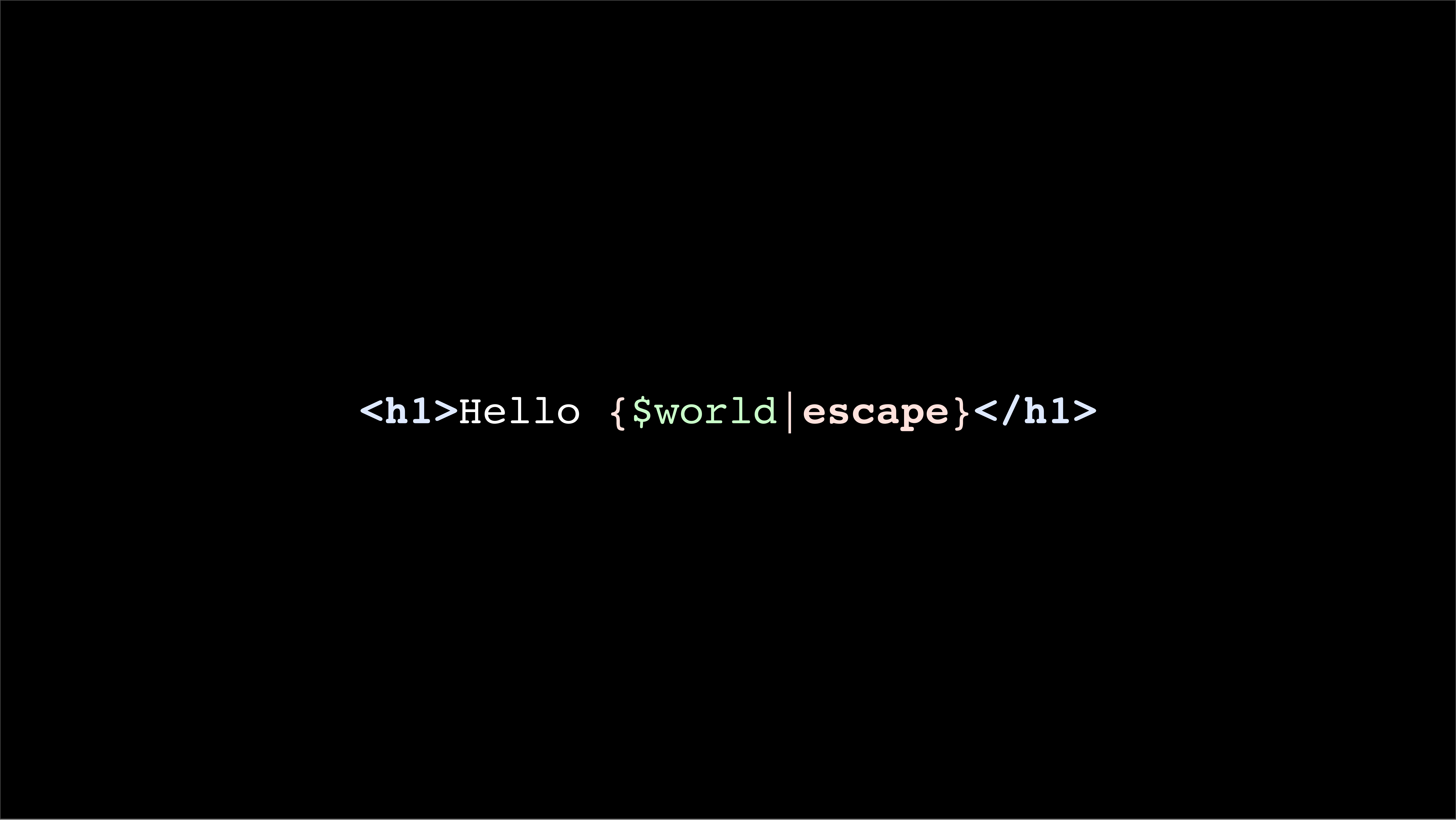
Super-basic use-case, just output variable. And that's easy to get wrong! If you do it like that, chances are you'll generate an ill-formed document (if $world contains &) or worse, make site vulnerable to XSS (if someone puts <script> in there)

So you have to escape every single bit of text and name of the function that does that couldn't be longer. It's ridiculous!

Filtering of all HTML on input isn't a solution – PHP went that path with magic_quotes. We don't want magic_entities!

Smarty makes it a bit cleaner, dropping long opening tags and shorteing function name, but fundmamentally it’s the same thing. You still have to remember to escape every bit of text every time. It does nothing to prevent from generating invalid HTML.

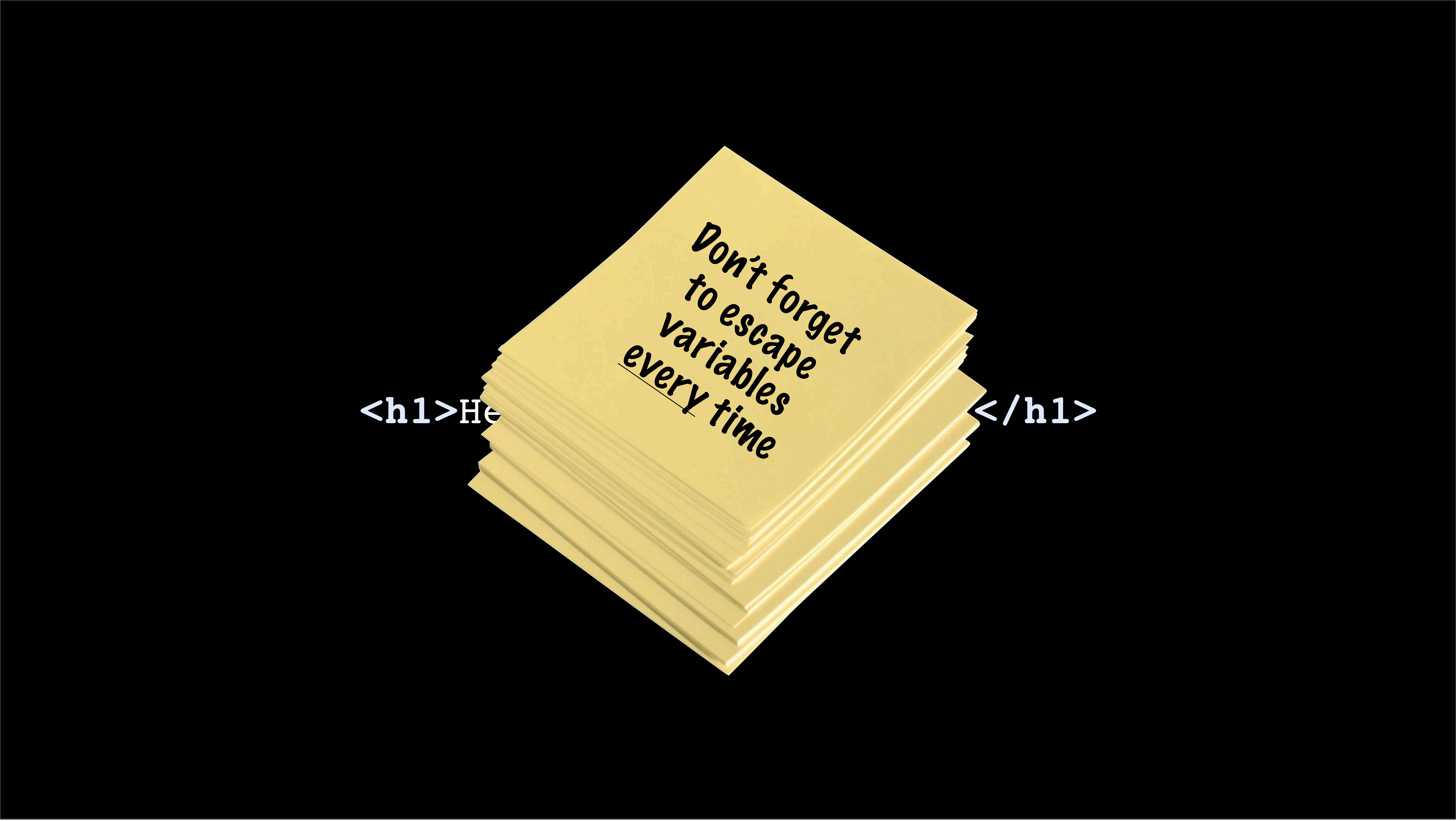
(BTW: according to hixie 80-90% of pages on the web have syntax errors: http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2007-February/009517.html)



```
<h1>Hello {$world|escape}</h1>
```

Smarty makes it a bit cleaner, dropping long opening tags and shorteing function name, but fundmamentally it’s the same thing. You still have to remember to escape every bit of text every time. It does nothing to prevent from generating invalid HTML.

(BTW: according to hixie 80-90% of pages on the web have syntax errors: http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2007-February/009517.html)



`<h1>He` … `</h1>`

Smarty makes it a bit cleaner, dropping long opening tags and shorteing function name, but fundmamentally it's the same thing. You still have to remember to escape every bit of text every time. It does nothing to prevent from generating invalid HTML.

(BTW: according to hixie 80-90% of pages on the web have syntax errors: http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2007-February/009517.html)

Little change and suddenly handling of escapes is sooo much easier, because that's the 80% case.

You can still easily output XHTML from the database or tagsoup, if you have to – just add "I know what I'm doing" keyword.

But this isn't how PHPTAL usually looks like. Let's take another PHP fragment as an example:

```
<h1>Hello ${world}</h1>
```

Little change and suddenly handling of escapes is sooo much easier, because that’s the 80% case.

You can still easily output XHTML from the database or tagsoup, if you have to – just add “I know what I’m doing” keyword.

But this isn’t how PHPTAL usually looks like. Let’s take another PHP fragment as an example:

```
<h1>Hello ${structure world}</h1>
```

Little change and suddenly handling of escapes is sooo much easier, because that's the 80% case.

You can still easily output XHTML from the database or tagsoup, if you have to – just add "I know what I'm doing" keyword.

But this isn't how PHPTAL usually looks like. Let's take another PHP fragment as an example:

```
<?php if (count($products)) { ?>
 <ul>
  <?php foreach($products as $p) { ?>
   <li>
    <?php echo htmlspecialchars($p->name); ?>
   </li>
  <?php } ?>
 </ul>
<?php } ?>
```

Pretty basic case - output a list, and yet so verbose and with plenty of places to go wrong (you can replace easy to lose brace with easy to forget colon and verbose end keyword).

In PHPTAL...

```
<ul tal:condition=”products”>
<li tal:repeat=”p products”
    tal:content=”p/name” />
</ul>
```

It’s that simple.

It’s an template attribute language, so you do everything using attributes.

tal:condition attribute checks if there are any products.

tal:repeat causes <li> element to repeat. There aren’t any special tags for looping – you just add an attribute to any element. In this case it assigns each product to variable p, and on the same element tal:content attribute outputs it.

Note the slash in tal:content – like in XPath, you can traverse data structures in PHP – objects, arrays. With this syntax you can call functions and browse their result – it uses PHP’s reflection capabilities to make syntax easier and hide implementation details – just use slash for everything.

<li> is empty here – you can save some typing.

```
<ul tal:condition=”products”>
 <li tal:repeat=”p products” tal:content=”p/name”>
  example product
 </li>
</ul>
```

I’ve added example text here. When template is run it will be completely replaced by the actual content, so why bother?

Notice that...

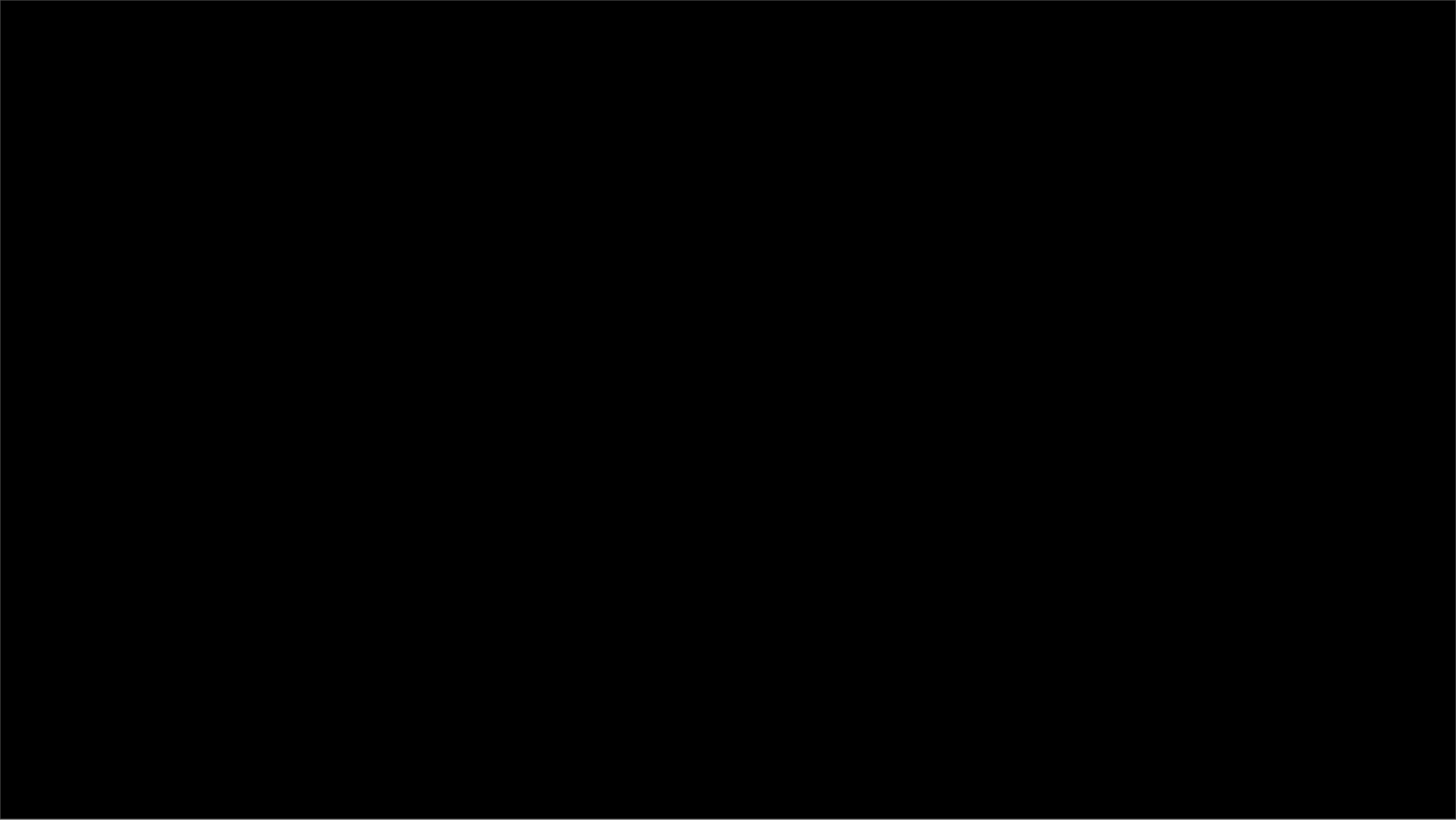
```
<ul tal:condition=”products”>
 <li tal:repeat=”p products” tal:content=”p/name”>
  example product
 </li>
</ul>
```

...when you ignore the attributes, all that’s left is proper HTML!

Syntax and structure remains valid. This is awesome, because your editor’s syntax highlighting works. Your HTML or XML processing tools work, and of course it can be displayed in a browser, so you can edit CSS even after you’ve chopped up your initial XHTML, but before you’ve got server-side application running.

Even WYISYWG editors can open such template without completely destroying it.

but what matters the most is that...



it makes it **easy** to produce valid and well-formed XHTML. With PHPTAL it's easier to produce well-formed document than ill-formed one.

TAL syntax makes it trivial to check if template's XML is well-formed (and PHPTAL does that), and since running the template won't make it ill-formed (with the one exception when you explicitly tell it to), you can be pretty sure that if it works for you, it will work for everyone else.

It isn't like that with PHP or Smarty, where bits of code are hidden in conditional expressions. Unless you check every combination of ifs and elses, you're risking that some of your users will get YSOD.

And there's sad XHTML reality - IE. Not only you have to worry about XML well-formedness, but also how badly IE will misinterpret the code. Fortunately PHPTAL is aware of XHTML compatibility guidelines and will stick to them.

Re image:

Makes *well-formed* XHTML easy

No bozos!

it makes it **easy** to produce valid and well-formed XHTML. With PHPTAL it's easier to produce well-formed document than ill-formed one.

TAL syntax makes it trivial to check if template's XML is well-formed (and PHPTAL does that), and since running the template won't make it ill-formed (with the one exception when you explicitly tell it to), you can be pretty sure that if it works for you, it will work for everyone else.

It isn't like that with PHP or Smarty, where bits of code are hidden in conditional expressions. Unless you check every combination of ifs and elses, you're risking that some of your users will get YSOD.

And there's sad XHTML reality – IE. Not only you have to worry about XML well-formedness, but also how badly IE will misinterpret the code. Fortunately PHPTAL is aware of XHTML compatibility guidelines and will stick to them.

Re image:

Makes *well-formed* XHTML easy

- Ensures that you close all elements and quote attributes

No bozos!

it makes it **easy** to produce valid and well-formed XHTML. With PHPTAL it's easier to produce well-formed document than ill-formed one.

TAL syntax makes it trivial to check if template's XML is well-formed (and PHPTAL does that), and since running the template won't make it ill-formed (with the one exception when you explicitly tell it to), you can be pretty sure that if it works for you, it will work for everyone else.

It isn't like that with PHP or Smarty, where bits of code are hidden in conditional expressions. Unless you check every combination of ifs and elses, you're risking that some of your users will get YSOD.

And there's sad XHTML reality - IE. Not only you have to worry about XML well-formedness, but also how badly IE will misinterpret the code. Fortunately PHPTAL is aware of XHTML compatibility guidelines and will stick to them.

Re image:

Makes *well-formed* XHTML easy

- Ensures that you close all elements and quote attributes
- Escapes all ampersands by default
  `&` → `&amp;`

No bozos!

it makes it **easy** to produce valid and well-formed XHTML. With PHPTAL it's easier to produce well-formed document than ill-formed one.

TAL syntax makes it trivial to check if template's XML is well-formed (and PHPTAL does that), and since running the template won't make it ill-formed (with the one exception when you explicitly tell it to), you can be pretty sure that if it works for you, it will work for everyone else.

It isn't like that with PHP or Smarty, where bits of code are hidden in conditional expressions. Unless you check every combination of ifs and elses, you're risking that some of your users will get YSOD.

And there's sad XHTML reality – IE. Not only you have to worry about XML well-formedness, but also how badly IE will misinterpret the code. Fortunately PHPTAL is aware of XHTML compatibility guidelines and will stick to them.

Re image:

Makes *well-formed* XHTML easy

- Ensures that you close all elements and quote attributes
- Escapes all ampersands by default
  `&` → `&amp;`
- Is aware of XHTML compatibility guidelines
  `<img></img>` → `<img/>`

No bozos!

it makes it **easy** to produce valid and well-formed XHTML. With PHPTAL it's easier to produce well-formed document than ill-formed one.

TAL syntax makes it trivial to check if template's XML is well-formed (and PHPTAL does that), and since running the template won't make it ill-formed (with the one exception when you explicitly tell it to), you can be pretty sure that if it works for you, it will work for everyone else.

It isn't like that with PHP or Smarty, where bits of code are hidden in conditional expressions. Unless you check every combination of ifs and elses, you're risking that some of your users will get YSOD.

And there's sad XHTML reality – IE. Not only you have to worry about XML well-formedness, but also how badly IE will misinterpret the code. Fortunately PHPTAL is aware of XHTML compatibility guidelines and will stick to them.

Re image:

```
<a href=”${href}” tal:omit-tag=”not:href”>
  optionally linked text
</a>
```

Let’s say I’ve got a bit of text that sometimes is a link and sometimes isn’t – like a current menu entry.

With templates like Smarty, if you want to omit start and end tags, you have to add if twice – once for opening tag and once for closing and of course if you don’t put identical condition in both cases, you’ll be called a bozo!

And this case is particularly evil in XSLT, where you just can’t simply omit an element. You have to create two versions of code – one with tag and one without, and you’ll need to create an additional template for each case!

In TAL it’s super-simple. tal:omit-tag makes tag disappear, but not its content. One attribute and problem solved. TAL takes care of ending tag for you.

```
<title tal:content=”page/title | site/title | default”>
  My Website
</title>
```

The idea of mixing examples and working code can have practial use too – you can give alternatives for content attribute. If one doesn’t exist, it will try another or leave the default text in.

```
<textarea tal:content=”text | nothing”/>
```

or it might just keep it empty.

PHPTAL will complain if you use non-existant variable. This prevents errors.

```
<option tal:repeat=”c countries”
        tal:content=”c”
        tal:attributes=”selected php:c==’UK’” />
```

You can optionally add attributes – list them in tal:attributes.

Notice the php: part of expression. This switches expression to PHP mode where you can write more powerful expressions.

TALES (TAL expression syntax) is intentionally very basic, to keep application logic away from the templates. It could have been `tal:attributes=”selected c/isSelected”` in pure TALES.

PHPTAL is smart enough to expand XHTML boolean attributes properly.

```
<option tal:repeat=”c countries”
        tal:content=”c”
        tal:attributes=”selected php:c==’UK’” />

<option selected=”selected”>UK</option>
```

You can optionally add attributes – list them in tal:attributes.

Notice the php: part of expression. This switches expression to PHP mode where you can write more powerful expressions.

TALES (TAL expression syntax) is intentionally very basic, to keep application logic away from the templates. It could have been `tal:attributes=”selected c/isSelected”` in pure TALES.

PHPTAL is smart enough to expand XHTML boolean attributes properly.

```
<tr tal:repeat=”row rows”
    tal:attributes=”class php:repeat.row.odd?‘odd’:NULL”>
```

There’s magic variable repeat, similar to Smarty’s section, that contains all kinds of information about loops like keys, even/odd indexes, even roman numerals.

TAL’s php: mode uses dots instead of ->. It is a bit weird, but it’s supposed to make transition between PHP and Python (where TAL comes from) easier – simplest cases won’t need code changes.

Other features

- ✓ Compiles templates into PHP code
- ✓ Caching of output
- ✓ i18n
- ✓ METAL macros
- ✓ Advanced TALES paths and custom modifiers
- ✓ Nice integration with PHP and SPL

Open Source LGPL

There’s plenty more.

I’ts quite fast. Compiles templates into straightforward PHP code that isn’t very different from what you’d write yourself in “old style” PHP (there’s no DOM heavy lifting at run time).

Compiled templates are cached and you can even cache final output of individual elements (e.g. cache some divs in a sidebar, but have rest of the page dynamic).

Supports internationalization, works with gettext.

METAL (macro extension) can include content from other files or output data recursively.

You can easily create your own TALES modifiers – e.g. for formatting of dates, sums of money, etc.

PHPTAL plays nice with PHP5 and supports SPL interfaces and SimpleXML.

Thank you!

Questions? http://phptal.motion-twin.com/contact.html