

# Liquid Templating Language

Oliver Clarke - [ollie@clarketus.net](mailto:ollie@clarketus.net)

# What is Liquid?

- Templating language that can be used as an alternative to ERB.
- Uses its own “liquid” syntax inside dynamic statements.
- Templates are parsed via a native ruby plugin or via a stand-alone gem.
- Separates template logic from rest of application
- Implementations for other languages EG: PHP

<http://www.liquidmarkup.org/>

Liquid is an extraction from the e-commerce system Shopify. Shopify powers many thousands of e-commerce stores which all call for unique designs. For this we developed Liquid which allows our customers complete design freedom while maintaining the integrity of our servers.

Liquid has been in production use since June 2006 and is now used by many other hosted web applications.

It was developed to for usage in Ruby on Rails web applications and integrates seamlessly as a plugin but it also works excellently as a stand alone library.

# Template Code

```
<div class="posts">
  {% for post in posts %}
    <li>
      <h2>{{ post.title }}</h2>
      <p>{{ post.content | truncate: 160 }}</p>
    </li>
  {% endfor %}
</div>
```

# Variables

- Primitive types
- Drops
- Filters
- Tags

# Primitive types

- Null
- Boolean
- Integers
- Strings
- Arrays
- No hashes!
- All converted automatically from ruby objects

# Drops

- Represent external objects
- Basically a big “Filter” for the object you want referenced within the template.
- define `to_liquid` on the ruby object
- Usage: `{{ post.title }}`

# Define a drop

```
class PostDrop < Liquid::Drop

  def initialize(post)
    @post = post
  end

  # access controller inside drop
  def invoke_drop(method)
    @controller = @context.registers[:controller]

    variables = @controller.instance_variable_names
    variables -= @controller.protected_instance_variables if self.respond_to?(:protected_instance_variables)
    variables.each { |name| self.instance_variable_set name, @controller.instance_variable_get(name) }

    super(method)
  end
  alias :[] :invoke_drop

  # otherwise PostDrop#inspect is used inside {{ }}
  def to_s
    ""
  end

  def id
    @post.id
  end

  def name
    @post.name
  end

  def url
    @controller.send(:post_url, @post)
  end

end
```

# Filters

- Basic string helpers
- Usage: `{{ post.content | truncate: 160 }}`

- Code:

```
module LiquidFilters

  def image_tag(url)
    # implementation etc
  end

end
```

- And then:

```
Liquid::Template.register_filter(LiquidFilters)
```

# Tags

- Liquid logic commands EG: if, else, for
- Define your own!
- `{% %}` executes a tag
- Usage: `{% if posts.size > 1 %}{% endif %}`

# Define a tag

```
class Tags::Capture < Liquid::Block
  Syntax = /(\w+)/

  def initialize(tag_name, markup, tokens)
    if markup =~ Syntax
      @to = $1
    else
      raise SyntaxError.new("Syntax Error in 'capture' - Valid syntax: capture [var]")
    end

    super
  end

  def render(context)
    output = super
    context[@to] = output.join
    ''
  end
end
```

## Must also register tag:

```
Template.register_tag('capture', Tags::Capture)
```

# Rendering

- Render manually:

```
assigns = {  
  "post" => @post  
}
```

```
liquid = Liquid::Template.parse(File.read("/path/to/template"))  
result = liquid.render!(assigns, :registers => {:controller => self})
```

- Code in the plugin available for .liquid files to be automatically targeted in view folder
- <http://github.com/madadam/liquidizer> for rendering from the DB

# Pros

- Separates logic from templates, so that templates can be generated by third parties with security guaranteed
- Simple syntax, easy for designers!
- Existing documentation for default tags etc
- Does not appear to slow the application down

# Cons

- Less flexibility within templates
- Loose control of frontend
- Lots of documentation to write!

# Beyond

- Vision - Shopifys desktop theme editor
  - Local webrick server
  - packaged with .app and .exe
  - <http://github.com/Shopify/vision>
- Other “secure” options to liquid?

Thanks!