Let's Talk About Templates

Armin Ronacher
@mitsuhiko
why are we discussing templates in 2014?
In 2011 we all thought single page applications are the future
“Cloud” Performance > Phone Performance
It's really hard to make a nice, JS heavy UI
Server Side Rendering is Fancy Again
(at least for us)
“Talk about Performance”

ev ery i n v i t a t i o n a b o u t a t e m p l a t e e n g i n e t a l k e v e r
History Lessons

History of Python Template Engines
Django and Jinja and the Greater Picture

❖ 2000: mod_python
❖ 2003: Webware for Python (→ wsgikit → paste → webob)
❖ 2003: WSGI spec
❖ 2005: Django
❖ 2006: Jinja
❖ 2008: Jinja2
❖ 2014: haven't touched templates in years!

(story not continued)
Personal Growth

Why I have a hard time talking about Jinja today
Armin and Jinja

- Armin learning programming: 2003
- Armin learning Python: 2004
- Django’s first public release: July 2005
- Jinja’s first public release: January 2006
- Jinja2: June 2008
Jinja2 has bugs, bug fixing some of them would probably break people’s templates
Jinja’s Problems

- Hand written lexer with problematic operator priorities
- Slightly incorrect identifier tracking
- Non ideal semantics for included templates
- Slow parsing and compilation step
not broken enough for a rewrite

(there won’t be a Jinja 3)
How do they work?

What makes a template engine work
Django and Jinja2 differ greatly on the internal design.

- Django is an AST interpreter with made up semantics.
- Jinja is a transpiler with restricted semantics to aid compilation.
General Preprocessing Pipeline

❖ Load template source
❖ Feed source to lexer for tokenization
   ❖ Parser converts tokens into an AST (Abstract Syntax Tree)
     ❖ -> Compile to Bytecode
     ❖ -> Keep AST for later evaluation
渲染管道

- 创建一个包含模板所需所有数据的上下文对象
- 拿取AST/字节码
  - 将上下文和AST/字节码传递给渲染系统
  - 获取结果
The Differences

How do Jinja2 and Django differ?
Execution Model

What they do when they render

- Evaluates Bytecode

- Evaluates AST
From Source to Node Tree

- Overarching Grammar
- As the lexer encounters a block opener tag it will switch its parsing state
- Allows arbitrary nesting of lexical constructs
- Two stage grammar
- Lexer splits template into tokens in the form “block”, “variable”, “comment” and “template data”
- Second stage lexer splits tokens into smaller ones
- No nesting
Tokens after Lexing

{% if expr %}...
{% endif %}

- BLOCK_START
- NAME "if"
- IDENT "expr"
- BLOCK_END
- DATA "...
- BLOCK_START
- NAME "endif"
- BLOCK_END

- BLOCK "if expr"
- DATA "...
- BLOCK "endif"
Nodes in Jinja act as AST
The AST gets processed and compiled into Python code
Nodes are thrown away post compilation

Nodes in Django are kept in memory
Upon evaluation their callbacks are invoked
Callbacks render the template recursively into strings
Overarching Grammar
As the lexer encounters a block opening tag it will switch its parsing state
Allows arbitrary nesting of lexical constructs

Two stage grammar
Lexer splits template into tokens in the form “block”, “variable”, “comment” and “template data”
Second stage lexer splits tokens into smaller ones
No nesting
Extensions

- heavily discouraged
- syntax consistent with Jinja core
- need to generate Jinja nodes
- tricky to debug due to compiled nature

- encouraged and ubiquitous
- can and do have custom syntax
- easy to implement due to the render method and context object
- debugging possible within Django due to the debug middleware
Rendering

- compiles into a generator yielding string chunks.
- proper recursive calls will buffer
- syntax supported recursion will forward iterators
- each render function yields a string
- any form of recursive calls will need to assemble a new string
Error Handling

- keeps source information
- integrates into Python traceback, supports full recursion including calls to Python and back to Jinja
- Customizable behavior for missing variables
- keeps simplified source location on nodes
- uses its own error rendering and for performance reasons cannot provide more accurate information
- Missing var = empty string
The Context

- Source of data
- Only holds top-level variables
- Two-layer dictionary, optionally linked to a parent scope but not resolved through

- Store of data
- Holds all variables
- Stack of dictionaries
Autoescaping

- uses markupsafe
- escaping is “standardized”
- lives in Python
- the only integration in the template engine is:
  - awareness in the optimizer
  - enables calls to escape() for all printed expressions

- Django specific
- lives largely only in the template engine with limited support in Python
- Django one-directionally supports the markupsafe standard
class Foo(object):
    def __html__(self):
        return Markup(u'This object in HTML context')
    def __unicode__(self):
        return u'This object in text context'

>>> Markup('<em>%s</em>') % '<script>alert(document.cookie)</script>'
Markup(u'<em>&lt;script&gt;alert(document.cookie)&lt;/script&gt;</em>')
Django's Templates

How it renders and does things
Parsing after “Tokenizing”

- look at first name
- load “parsing callback for name”
  - parsing callback might or might not use “token splitting function”
  - parsing callback creates a node
Templates are really old

❖ whoever wrote it, learned what an AST interpreter is
❖ someone else changed it afterwards and forgot that the idea is, that it's not mutating the state of nodes while rendering
❖ only after Jinja2's release could Django cache templates because rendering stopped mutating state :)

How it Represents

Hello {{ variable|escape }}

NodeList([
   TextNode("Hello "),
    VariableNode(FilterExpression(
        var=Variable("variable"),
        filters=[("escape", ())])
])
How it Renders

Hello {{ variable|escape }}

```python
class NodeList(list):
    def render(self, context):
        bits = []
        for node in self:
            if isinstance(node, Node):
                bit = node.render(context)
            else:
                bit = node
            bits.append(force_text(bit))
        return mark_safe(''.join(bits))
```
class IfNode(Node):
    def __init__(self, conditions_nodelists):
        self.conditions_nodelists = conditions_nodelists

    def render(self, context):
        for condition, nodelist in self.conditions_nodelists:
            if condition is not None:
                try:
                    match = condition.eval(context)
                except VariableDoesNotExist:
                    match = None
                else:
                    match = True
                if match:
                    return nodelist.render(context)
        return ''
Jinja is Complex

Jinja does things because it can
def root(context):
    l_variable = context.resolve('variable')
    t_1 = environment.filters['escape']
    yield u'Hello ' 
    yield escape(t_1(l_variable))
Knowledge Allows Optimizations

def root(context):
    yield u'Hello &lt;World&gt;!'
Different Transformations

{% for item in seq %}<li>{{ item }}{% endfor %}
def root(context):
    l_seq = context.resolve('seq')
    l_item = missing
    l_loop = missing
    for l_item, l_loop in LoopContext(l_seq):
        yield u'<li>%s: %s' % (escape(environment.getattr(l_loop, 'index')), escape(l_item)),
    l_item = missing
def root(context):
    yield u'<title>'
    for event in context.blocks['title'][0](context):
        yield event
    yield u'</title>'

def block_title(context):
    yield u'Default Title'

blocks = {'title': block_title}
Super Calls

{% extends "layout" %}{% block title %}{{ super() }}{% endblock %}

def root(context):
    parent_template = None
    parent_template = environment.get_template('layout', None)
    for name, parent_block in parent_template.blocks.iteritems():
        context.blocks.setdefault(name, []).append(parent_block)
    for event in parent_template.root_render_func(context):
        yield event

def block_title(context):
    l_super = context.super('title', block_title)
    yield escape(context.call(l_super))

blocks = {'title': block_title}
Errors

{% macro may_break(item) -%}  
  [{% if item == 0 %}0{% endif %}]
{%- endmacro %}

Traceback (most recent call last):
  File "example.py", line 7, in <module>
    print tmpl.render(seq=[3, 2, 4, 5, 3, 2, 0, 2, 1])
  File "jinja2/environment.py", line 969, in render
    return self.environment.handle_exception(exc_info, True)
  File "jinja2/environment.py", line 742, in handle_exception
    reraise(exc_type, exc_value, tb)
  File "templates/broken.html", line 4, in top-level template code
    <li>{{ may_break(item) }}</li>
  File "templates/subbroken.html", line 2, in template
    [{% if item == 0 %}0{% endif %}]
ZeroDivisionError: division by zero
Make one like the other

About the many attempts of making Django like Jinja
Why make one like the other?

❖ People like Jinja because of
  ❖ expressions
  ❖ performance

❖ People like Django because of
  ❖ extensibility
The Performance Problem

- Jinja is largely fast because it choses to “not do things”:
  - it does not have a context
  - it does not have loadable extensions
  - if it can do nothing over doing something, it choses nothing
  - it tracks identifier usage to optimize code paths
Why can't Django do that?

- Jinja needed to sacrifice certain functionality
- Doing the same in Django would break everybody's code
Why not make a Jinja Inspired Django?

- Making the Django templates like Jinja2 would be a Python 3 moment
- There would have to be a migration path (allow both to be used)
- Cost / Benefit relationship is not quite clear
Pluggable Template Engines?

- Most likely success
- Could start switching defaults over at one point
- Pluggable apps might not like it :(
Questions and Answers

Slides will be at lumur.pocoo.org/talks

Contact via armin.ronacher@active-4.com

Twitter: @mitsuhiko

If you have interesting problems, you can hire me :)}