Djula HTML templating system Documentation

Release 0.2

Nick Allen

Dec 23, 2020

CONTENTS

Djula is an HTML templating system similar to Django templates for Common Lisp.

Contents:

CHAPTER

ONE

INTRODUCTION

Djula is an HTML templating system similar to Django templates for Common Lisp.

Djula's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML.

Philosophy

If you have a background in programming, or if you're used to languages which mix programming code directly into HTML, you'll want to bear in mind that the Djula template system is not simply Common Lisp code embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.

The Djula template system provides tags which function similarly to some programming constructs – an **:ttag:`if`** tag for boolean tests, a **:ttag:`for`** tag for looping, etc. – but these are not simply executed as the corresponding Lisp code, and the template system will not execute arbitrary Lisp expressions. Only the tags, filters and syntax listed below are supported by default (although you can add your own extensions to the template language as needed).

1.1 Prerequisites

TODO: list of Common Lisp compilers Djula works on.

1.2 Installation

Djula is available on Quicklisp:

```
(ql:quickload :djula)
```

CHAPTER

TWO

BASICS

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.

```
{% extends "base.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>

{% for story in story_list %}
<h2>
  <a href="{{ story.url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:100 }}</p>
{% endfor %}
{% endblock %}
```

CHAPTER

THREE

USAGE

To render our templates, they need to be compiled first. We do that with the *COMPILE-TEMPLATE** function. For inheritance to work, we need to put all the templates in the same directory so that Djula can find them when resolving templates inheritance.

Djula looks for templates in the **CURRENT-STORE**. For our templates to be found, we have to add the template folder path the templates store. We can do that with the *add-template-directory* function.

Here is an example:

```
(add-template-directory (asdf:system-relative-pathname "webapp" "templates/"))

(defparameter +base.html+ (djula:compile-template* "base.html"))

(defparameter +welcome.html+ (djula:compile-template* "welcome.html"))

(defparameter +contact.html+ (djula:compile-template* "contact.html"))
```

Then we can render our compiled templates using the *RENDER-TEMPLATE** function:

```
(djula:render-template* +welcome.html+ s
                        :title "Ukeleles"
                        :project-name "Ukeleles"
                        :mode "welcome")
```

3.1 Auto-reload

By default, Djula automatically recompiles the templates when they change.

If you want to disable this, use the *:djula-prod feature*:

```
(push :djula-prod *features*)
```

3.2 API

function (**add-template-directory***directory &optional*) *template-store *current-store**
Adds `DIRECTORY` to the search path of the `TEMPLATE-STORE`

generic (**compile-template***compiler name &optional error-p*)
Provides a hook to customize template compilation.

Specializers

- `(toplevel-compiler common-lisp:t)`
- `(compiler common-lisp:t)`

function (**compile-template***_name_)
Compiles template `NAME` with compiler in *CURRENT-COMPILER*

function (**render-template***_template &optional stream &rest *template-arguments*_)
Render `TEMPLATE` into `STREAM` passing *TEMPLATE-ARGUMENTS*

CHAPTER

FOUR

VARIABLES

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result. Variable names consist of any combination of alphanumeric characters and the underscore (`"_"`). The dot (`"."`) also appears in variable sections, although that has a special meaning, as indicated below. Importantly, *you cannot have spaces or punctuation characters in variable names.*

Use a dot (`.`) to access attributes of a variable.

Behind the scenes

For accessing variables the `ACCESS` Common Lisp library is used: https://github.com/AccelerationNet/access

---

In the above example, `{{ section.title }}` will be replaced with the `title` attribute of the `section` object.

Note that "bar" in a template expression like `{{ foo.bar }}` will be interpreted as a literal string and not using the value of the variable "bar", if one exists in the template context.

4.1 Default template variables

You can use the `*default-template-arguments*` variable to store arguments that will be available for all templates. It is a plist, so use getf to add arguments, like this:

```
(setf (getf djula:*default-template-arguments* :foo) 'some-value)
```

And now, you can access `{{ foo }}` in your template.

This is useful when you have many templates that rely on the same set of variables. Use this variable to refactor your code when appropriate.

Note that you could also write a function that wraps `render-template*` and uses a default list of variables plus other ones given as arguments:

```
(defun my-render-template (template stream &rest args)
  (apply #'djula:render-template* template stream (list* :foo 'some-value args)))
```

CHAPTER

FIVE

TAGS

5.1 Overview

Tags look like this: `{% tag %}`. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. `{% tag %} ... tag contents ... {% endtag %}`).

Here are some of the more commonly used tags:

:ttag:`for` Loop over each item in an array. For example, to display a list of athletes provided in `athlete-list`:

```
<ul>
{% for athlete in athlete-list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

:ttag:`if`, `else` Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete-list %}
    Number of athletes: {{ athlete-list|length }}
{% else %}
    No athletes.
{% endif %}
```

:ttag:`block` and :ttag:`extends` Set up **`template inheritance`_** (see below), a powerful way of cutting down on "boilerplate" in templates.

5.2 List of tags

Tags

- *block*
- *extends*
- *super*
- *comment*
- *cycle*

- *debug*
- *filter*
- *firstof*
- *for*
- *if*
- *Boolean operators*
- *ifchanged*
- *ifequal*
- *ifnotequal*
- *include*

5.2.1 block

Defines a block that can be overridden by child templates.

Sample usage:

```
{% block stylesheets %}
   ...
{% endblock %}
```

See Template inheritance for more information.

5.2.2 extends

Extends a template

Sample usage:

```
{% extends "base.html" %}
```

5.2.3 super

Gets the content of the block from the parent template. You can pass the name of the block of the parent block you want to access. If no name is passed, then the current block's parent is used.

Sample usage:

```
{% super "stylesheets" %}

{% block stylesheets %}
  {% super %}
{% endblock %}
```

5.2.4 comment

Ignores everything between `{% comment %}` and `{% endcomment %}`. An optional note may be inserted in the first tag. For example, this is useful when commenting out code for documenting why the code was disabled.

Sample usage:

```
<p>Rendered text with {{ pub-date|date }}</p>
{% comment "Optional note" %}
    <p>Commented out text with {{ create-date|date }}</p>
{% endcomment %}
```

`comment` tags cannot be nested.

5.2.5 cycle

Produces one of its arguments each time this tag is encountered. The first argument is produced on the first encounter, the second argument on the second encounter, and so forth. Once all arguments are exhausted, the tag cycles to the first argument and produces it again.

This tag is particularly useful in a loop:

```
{% for o in some-list %}
    <tr class="{% cycle "row1" "row2" %}">
        ...
    </tr>
{% endfor %}
```

The first iteration produces HTML that refers to class `row1`, the second to `row2`, the third to `row1` again, and so on for each iteration of the loop.

You can use variables, too. For example, if you have two template variables, `rowvalue1` and `rowvalue2`, you can alternate between their values like this:

```
{% for o in some-list %}
    <tr class="{% cycle rowvalue1 rowvalue2 %}">
        ...
    </tr>
{% endfor %}
```

You can mix variables and strings:

```
{% for o in some-list %}
    <tr class="{% cycle "row1" rowvalue2 "row3" %}">
        ...
    </tr>
{% endfor %}
```

You can use any number of values in a `cycle` tag, separated by spaces. Values enclosed in double quotes (`"`) are treated as string literals, while values without quotes are treated as template variables.

5.2.6 debug

Outputs a whole load of debugging information

5.2.7 filter

Filters the contents of the block through one or more filters. Multiple filters can be specified with pipes and filters can have arguments, just as in variable syntax.

Note that the block includes *all* the text between the `filter` and `endfilter` tags.

Sample usage:

```
{% filter force-escape|lower %}
    This text will be HTML-escaped, and will appear in all lowercase.
{% endfilter %}
```

Note: The **:tfilter:`escape`** and **:tfilter:`safe`** filters are not acceptable arguments. Instead, use the **:ttag:`autoescape`** tag to manage autoescaping for blocks of template code.

5.2.8 firstof

Outputs the first argument variable that is not `False`. Outputs nothing if all the passed variables are `False`.

Sample usage:

```
{% firstof var1 var2 var3 %}
```

You can also use a literal string as a fallback value in case all passed variables are False:

```
{% firstof var1 var2 var3 "fallback value" %}
```

5.2.9 for

Loops over each item in an array, making the item available in a context variable. For example, to display a list of athletes provided in `athlete-list`:

```
<ul>
{% for athlete in athlete-list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

If you need to loop over an association list, you can unpack the values in each sublist into individual variables. For example, if your context contains a list of (x . y) coordinates called `points`, you could use the following to output the list of points:

```
{% for (x . y) in points %}
    There is a point at {{ x }},{{ y }}
{% endfor %}
```

This can also be useful if you need to access the items in a hash-table. For example, if your context contained a hash-table named `data`, the following would display the keys and values of the hash-table:

```
{% for (key . value) in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

The for loop sets a number of variables available within the loop:

Variable	Description
`forloop.counter`	The current iteration of the loop (1-indexed)
`forloop.counter0`	The current iteration of the loop (0-indexed)
`forloop.revcounter`	The number of iterations from the end of the loop (1-indexed)
`forloop.revcounter0`	The number of iterations from the end of the loop (0-indexed)
`forloop.first`	True if this is the first time through the loop
`forloop.last`	True if this is the last time through the loop
`forloop.parentloop`	For nested loops, this is the loop surrounding the current one

5.2.10 if

The `{% if %}` tag evaluates a variable, and if that variable is "true" (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output:

```
{% if athlete-list %}
    Number of athletes: {{ athlete-list|length }}
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete-list` is not empty, the number of athletes will be displayed by the `{{ athlete-list|length }}` variable.

5.2.11 Boolean operators

:ttag:`if` tags may use `and`, `or` or `not` to test a number of variables or to negate a given variable:

```
{% if athlete-list and coach-list %}
    Both athletes and coaches are available.
{% endif %}

{% if not athlete-list %}
    There are no athletes.
{% endif %}

{% if athlete-list or coach-list %}
    There are some athletes or some coaches.
{% endif %}

{% if not athlete-list or coach-list %}
    There are no athletes or there are some coaches (OK, so
    writing English translations of boolean logic sounds
    stupid; it's not our fault).
{% endif %}

{% if athlete-list and not coach-list %}
```

(continues on next page)

(continued from previous page)

```
    There are some athletes and absolutely no coaches.
{% endif %}
```

Use of both `and` and `or` clauses within the same tag is allowed, with `and` having higher precedence than `or` e.g.:

```
{% if athlete-list and coach-list or cheerleader-list %}
```

will be interpreted like:

```
(if (or (athlete-list and coach-list) cheerleader-list) ..)
```

Use of actual parentheses in the **:ttag:`if`** tag is invalid syntax. If you need them to indicate precedence, you should use nested **:ttag:`if`** tags.

5.2.12 ifchanged

Check if a value has changed from the last iteration of a loop.

The `{% ifchanged %}` block tag is used within a loop.

If given one or more variables, check whether any variable has changed.

For example, the following shows the date every time it changes, while showing the hour if either the hour or the date has changed:

```
{% for date in days %}
    {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
    {% ifchanged date.hour date.date %}
        {{ date.hour }}
    {% endifchanged %}
{% endfor %}
```

The `ifchanged` tag can also take an optional `{% else %}` clause that will be displayed if the value has not changed:

```
{% for match in matches %}
    <div style="background-color:
        {% ifchanged match.ballot-id %}
            {% cycle "red" "blue" %}
        {% else %}
            gray
        {% endifchanged %}
    ">{{ match }}</div>
{% endfor %}
```

5.2.13 ifequal

Output the contents of the block if the two arguments equal each other.

Example:

```
{% ifequal user.pk comment.user-id %}
    ...
{% endifequal %}
```

As in the **:ttag:`if`** tag, an `{% else %}` clause is optional.

The arguments can be hard-coded strings, so the following is valid:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

An alternative to the `ifequal` tag is to use the **:ttag:`if`** tag and the == operator.

5.2.14 ifnotequal

Just like **:ttag:`ifequal`**, except it tests that the two arguments are not equal.

An alternative to the `ifnotequal` tag is to use the **:ttag:`if`** tag and the `!=` operator.

5.2.15 include

Loads a template and renders it with the current context. This is a way of "including" other templates within a template.

The template name can either be a variable or a hard-coded (quoted) string, in either single or double quotes.

This example includes the contents of the template `"foo/bar.html"`:

```
{% include "foo/bar.html" %}
```

A set of parameters can also be added, which become available as context variables when the included template is rendered:

```
{% include "user.html" :user record.creator %}
{% include "user.html" :user record.updater %}
```

5.3 Custom tags

TODO

CHAPTER

SIX

COMMENTS

To comment-out part of a line in a template, use the comment syntax: `{# #}`.

For example, this template would render as `'hello'`:

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

If you need to comment out a multiline portion of the template, see the **:ttag:`comment`** tag.

CHAPTER

SEVEN

VERBATIM

If you want to leave some portion of text unprocessed by Djula, use the verbatim syntax: `{$ $}`.

For example, this template would render as `'this is {{verbatim}}'`:

```
{$ this is {{verbatim}} $}
```

CHAPTER

EIGHT

FILTERS

8.1 Overview

You can modify variables for display by using **filters**.

Filters look like this: `{{ name|lower }}`. This displays the value of the `{{ name }}` variable after being filtered through the **:tfilter:`lower`** filter, which converts text to lowercase. Use a pipe (`|`) to apply a filter.

Filters can be "chained." The output of one filter is applied to the next. `{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

Some filters take arguments. A filter argument looks like this: `{{ bio|truncatewords:30 }}`. This will display the first 30 words of the `bio` variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaced you'd use `{{ list|join:", " }}`.

Djula provides about thirty built-in template filters. You can read all about them in the built-in filter reference. To give you a taste of what's available, here are some of the more commonly used template filters:

8.2 List of filters

Filters

- *add*
- *addslashes*
- *capfirst*
- *cut*
- *date*
- *time*
- *datetime*
- *default*
- *reverse*
- *divisibleby*
- *sort*

- *first*
- *join*
- *last*
- *length*
- *length_is*
- *linebreaks*
- *linebreaksbr*
- *lower*
- *make_list*
- *safe, escape*
- *slice*
- *force-escape*
- *format*
- *replace ... with*
- *rest*
- *scan*
- *time*
- *truncatechars*
- *upper*
- *urlencode*

8.2.1 add

Adds the argument to the value.

For example:

```
{{ value|add:2 }}
```

If `value` is `4`, then the output will be `6`.

8.2.2 addslashes

Adds slashes before quotes. Useful for escaping strings in CSV, for example.

For example:

```
{{ value|addslashes }}
```

If `value` is `"I'm using Djula"`, the output will be `"I\'m using Djula"`.

8.2.3 capfirst

Capitalizes the first character of the value. If the first character is not a letter, this filter has no effect.

For example:

```
{{ value|capfirst }}
```

If `value` is `"djula"`, the output will be `"Djula"`.

8.2.4 cut

Removes all values of arg from the given string.

For example:

```
{{ value|cut:" " }}
```

If `value` is `"String with spaces"`, the output will be `"Stringwithspaces"`.

8.2.5 date

Formats a date

Example:: {{ date-today | date }}

A LOCAL-TIME format spec can be provided:

```
(defvar timestamp 3752179200)
{{ timestamp | date:(:year "/" (:month 2) "/" (:day 2)) }} ;; shows 2018/11/26
```

8.2.6 time

Formats a time

Example:

```
{{ time-now | time }}
```

8.2.7 datetime

Formats a date and time

Example:

```
{{ time-now | datetime }}
```

8.2.8 default

If value evaluates to `False`, uses the given default. Otherwise, uses the value.

For example:

```
{{ value|default "nothing" }}
```

If `value` is `""` (the empty string), the output will be `nothing`.

8.2.9 reverse

Takes a list and returns that list reversed.

For example:

```
{{ list | reverse }}
```

8.2.10 divisibleby

Returns `True` if the value is divisible by the argument.

For example:

```
{{ value|divisibleby:"3" }}
```

If `value` is `21`, the output would be `True`.

8.2.11 sort

Takes a list and returns that list sorted.

For example:

```
{{ list | sort }}
```

8.2.12 first

Returns the first item in a list.

For example:

```
{{ value|first }}
```

If `value` is the list `("a" "b" "c")`, the output will be `"a"`.

8.2.13 join

Joins a list with a string.

For example:

```
{{ value|join:" // " }}
```

If `value` is the list `("a" "b" "c")`, the output will be the string `"a // b // c"`.

8.2.14 last

Returns the last item in a list.

For example:

```
{{ value|last }}
```

If `value` is the list `("a" "b" "c" "d")`, the output will be the string `"d"`.

8.2.15 length

Returns the length of the value. This works for both strings and lists.

For example:

```
{{ value|length }}
```

If `value` is `("a" "b" "c" "d")` or `"abcd"`, the output will be `4`.

8.2.16 length_is

Returns `True` if the value's length is the argument, or `False` otherwise.

For example:

```
{{ value|length_is:"4" }}
```

If `value` is `['a', 'b', 'c', 'd']` or `"abcd"`, the output will be `True`.

8.2.17 linebreaks

Replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (`<br />`) and a new line followed by a blank line becomes a paragraph break (`</p>`).

For example:

```
{{ value|linebreaks }}
```

If `value` is `Joel\nis a slug`, the output will be `<p>Joel<br />is a slug</p>`.

8.2.18 linebreaksbr

Converts all newlines in a piece of plain text to HTML line breaks (`<br />`).

For example:

```
{{ value|linebreaksbr }}
```

If `value` is `Joel\nis a slug`, the output will be `Joel<br />is a slug`.

8.2.19 lower

Converts a string into all lowercase.

For example:

```
{{ value|lower }}
```

If `value` is `Still MAD At Yoko`, the output will be `still mad at yoko`.

8.2.20 make_list

Returns the value turned into a list. For a string, it's a list of characters. For an integer, the argument is cast into an unicode string before creating a list.

For example:

```
{{ value|make_list }}
```

If `value` is the string `"Joel"`, the output would be the list `['J', 'o', 'e', 'l']`. If `value` is `123`, the output will be the list `['1', '2', '3']`.

8.2.21 safe, escape

Marks a string as not requiring further HTML escaping prior to output. When autoescaping is off, this filter has no effect.

Note: If you are chaining filters, a filter applied after `safe` can make the contents unsafe again. For example, the following code prints the variable as is, unescaped:

```
{{ var|safe|escape }}
```

8.2.22 slice

Returns a slice of a sequence (i.e. lists, vectors, strings)

Uses the Common Lisp `cl-slice` library.

Syntax:

```
{{ seq | slice: slices }}
```

Each `slice` selects a subset of subscripts along the corresponding axis.

- A nonnegative integer selects the corresponding index, while a negative integer selects an index counting backwards from the last index:

  ```
  {{ list | slice: 4 }}
  ```

if the list is `(1 2 3 4 5 6)` it will output `(5)`

- `(start . end)` to select a range. When `end` is `NIL`, the last index is included.

Each boundary is resolved according to the other rules if applicable, so you can use negative integers:

```
{{ string | slice: (0 . 5) }}
{{ string | slice: (5 . nil) }}
```

if the string is `"Hello world"` is will output `Hello` and `world`.

8.2.23 force-escape

Forces escaping HTML characters (`<`, `>`, `'`, `\`, `&`):

```
{{ value | force-escape }}
```

It calls `djula::escape-for-html`.

8.2.24 format

Formats the variable according to the argument, a string formatting specifier. This specifier uses Common Lisp string formatting syntax

For example:

```
{{ value | format:"~:d" }}
```

If `value` is `1000000`, the output will be `1,000,000`.

8.2.25 replace ... with

The `replace` and the `with` filters work together:

```
{{ value | replace:regexp | with:string }}
```

This will replace all occurences of the regexp in "value" with a new string, using `ppcre:regex-replace-all`.

8.2.26 rest

Returns the `rest` of a list (aka `cdr`).

For example:

```
{{ values|rest }}
```

If `values` is the list `("a" "b" "c")`, the output will be `("b" "c")`.

8.2.27 scan

Extracts and displays a regexp from the value:

```
{{ value | scan:regexp }}
```

This will display only the text that matches the regexp (using `ppcre:scan-to-strings`).

8.2.28 time

Formats a time according to the given format.

For example:

```
{{ value | time }}
```

8.2.29 truncatechars

Truncates a string if it is longer than the specified number of characters. Truncated strings will end with the :cl:symbol:*ELLISION-STRING*, which defaults to "...".

Argument: Number of characters to truncate to

For example:

```
{{ value|truncatechars:9 }}
```

If `value` is `"Joel is a slug"`, the output will be `"Joel i..."`.

8.2.30 upper

Converts a string into all uppercase.

For example:

```
{{ value|upper }}
```

If `value` is `"Joel is a slug"`, the output will be `"JOEL IS A SLUG"`.

8.2.31 urlencode

Escapes a value for use in a URL.

For example:

```
{{ value|urlencode }}
```

If `value` is `"http://www.example.org/foo?a=b&c=d"`, the output will be `"http%3A//www.example.org/foo%3Fa%3Db%26c%3Dd"`.

An optional argument containing the characters which should not be escaped can be provided.

If not provided, the '/' character is assumed safe. An empty string can be provided when *all* characters should be escaped. For example:

```
{{ value|urlencode:"" }}
```

If `value` is `"http://www.example.org/"`, the output will be `"http%3A%2F%2Fwww.example.org%2F"`.

8.3 Custom filters

Use the `def-filter` macro. Its general form is:

```
(def-filter :myfilter-name (value arg)
   (body))
```

It always takes the variable's value as argument, and it can have one required or optional argument. For example, this is how those built-in filters are defined:

```
(def-filter :capfirst (val)
  (string-capitalize (princ-to-string val)))
```

This is all there is to it. Once written, you can use it in your templates. You can define a filter wherever you want and there is no need to register it or to import it in your templates.

Here's a filter with a required argument:

```
(def-filter :add (it n)
  (+ it (parse-integer n)))
```

and with an optional one:

```
(def-filter :datetime (it &optional format)
  (let ((timestamp ...))))
```

When you need to pass a second argument, make your filter return a lambda function and chain it with the `with` filter:

```
(def-filter :replace (it regex)
   (lambda (replace)
     (ppcre:regex-replace-all regex it replace)))

(def-filter :with (it replace)
   (funcall it replace))
```

Now we can write:

```
{{ value | replace:foo | with:bar }}
```

Errors are handled by the macro, but you can handle them and return a `template-error` condition:

```
(def-filter :handle-error-filter (it)
   (handler-case
         (do-something)
     (condition (e)
       (template-error "There was an error executing this filter: ~A" e))))
```

CHAPTER

NINE

TEMPLATE INHERITANCE

The most powerful – and thus the most complex – part of Djula's template engine is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

It's easiest to understand template inheritance by starting with an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

This template, which we'll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content.

In this example, the **:ttag:`block`** tag defines three blocks that child templates can fill in. All the **:ttag:`block`** tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
```

(continues on next page)

(continued from previous page)

```
{% endfor %}
{% endblock %}
```

The **:ttag:`extends`** tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent – in this case, "base.html".

At that point, the template engine will notice the three **:ttag:`block`** tags in `base.html` and replace those blocks with the contents of the child template. Depending on the value of `blog_entries`, the output might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>
```

Note that since the child template didn't define the `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site.
- Create a `base_SECTIONNAME.html` template for each "section" of your site. For example, `base_news.html`, `base_sports.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use **:ttag:`{% extends %}<extends>`** in a template, it must be the first template tag in that template. Template inheritance won't work, otherwise.
- More **:ttag:`{% block %}<block>`** tags in your base templates are better. Remember, child templates don't have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It's better to have more hooks than fewer hooks.

- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the **`next section`**_), since it was already escaped, if necessary, in the parent template.
- For extra readability, you can optionally give a *name* to your `{% endblock %}` tag. For example:

```
{% block content %}
...
{% endblock content %}
```

  In larger templates, this technique helps you see which `{% block %}` tags are being closed.

Finally, note that you can't define multiple **:ttag:`block`** tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill – it also defines the content that fills the hole in the *parent*. If there were two similarly-named **:ttag:`block`** tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

CHAPTER

TEN

INTERNATIONALIZATION

10.1 Syntax

The easiest way to translate a string or variable is to enclose it between {_ and _}:

```
{_ var _}
{_ "hello" _}
```

10.2 Tags

10.2.1 trans

Translates a variable or string

Example:

```
{% trans var %}
{% trans "hello" %}
```

10.3 Filters

10.3.1 trans

Translates a variable or string.

For example:

```
{{ var | trans }}
{{ "my string" |  trans }}
```

10.4 Choosing language

To choose the language to use, set the **CURRENT-LANGUAGE** variable.

For example:

```
(let ((djula:*current-language* :es))
   (djula:render-template* +translation.html+))
```

10.5 Backends

Djula supports two backends for doing translations: cl-locale and gettext

Please have a look at the demo and the documentation of those packages to figure out how to use them.

CHAPTER

ELEVEN

ERROR HANDLING

Djula catches errors and barfs them to the template output by default.

That is controlled via the **CATCH-TEMPLATE-ERRORS-P**. If changed to `NIL`, then errors are not caught anymore and are debuggable from the lisp listener.

Djula provides more or less verbosity in template errors. Verbosity is controlled via the variable **VERBOSE-ERRORS-P**.

Also, there's a fancy page to display errors, which can be disabled if desired. That is controlled via the variable **FANCY-ERROR-TEMPLATE-P**

11.1 API

variable *catch-template-errors-p*

variable *fancy-error-template-p*

variable *verbose-errors-p*

CHAPTER

TWELVE

API

Djula external symbols documentation

function (**compile-template***name*)
Compiles template NAME with compiler in *CURRENT-COMPILER*

function (**url-encode***string*)
URL-encodes a string using the external format EXTERNAL-FORMAT.

variable *current-store*
The currently in-use template store. Defaults to a FILE-STORE.

variable *allow-include-roots*

variable *current-compiler*

variable *fancy-debug-p*
When enabled, displays fancy html based debugging information for the {% debug %} tag

variable *djula-execute-package*

function (**fetch-template***key*)
Return the text of a template fetched from the *CURRENT-STORE*.

function (**url-encode-path***path*)

function (**url-decode**)

macro (**def-tag-compiler***name args &body body*)

function (**find-template***name &optional*) *error-p t*

variable *fancy-error-template-p*
When enabled, show a fancy template when an error ocurrs

variable *default-language*

function (**render-template***template &optional stream &rest *template-arguments**)
Render TEMPLATE into STREAM passing *TEMPLATE-ARGUMENTS*

variable *default-template-arguments*

function (**add-template-directory***directory &optional*) *template-store *current-store**
Adds DIRECTORY to the search path of the TEMPLATE-STORE

variable *catch-template-errors-p*
When enabled, caught errors during the rendering of the template are written to the output instead of being handled by the lisp listener

variable *current-language*

variable *verbose-errors-p*

When enabled, errors are displayed more verbosely. Good for debugging

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

A

C

D

F

R

U