Understanding Jinja

Jinja is a templating engine written in Python designed to help the rendering process of API responses. In every templating language, there are variables that get replaced with the actual values passed to them when the template is rendered, and there are tags that control the logic of the template.

The Jinja templating engine makes use of curly brackets { } to distinguish its expressions and syntax from regular HTML, text and any other variable in the template file.

The {{ }} syntax is called a **variable block**. The {% %} syntax houses control structures such as **if/else**, **loops**, and **macros**.

The three common syntax blocks used in the Jinja templating language include the following:

- {% ... %} – This syntax is used for statements such as control structures.
- {{ todo.item }} – This syntax is used to print out the values of the expressions passed to it.
- {# This is a great API book! #} – This syntax is used when writing comments and is not displayed on the web page.

Jinja template variables can be of any Python type or object if they can be converted into strings. A model, list, or dictionary type can be passed to the template and have its attributes displayed by placing these attributes in the second block listed previously.

In the next section, we'll be looking at filters. Filters are an important part of every templating engine and in Jinja, filters enable us to execute certain functions such as joining values from a list and retrieving the length of an object, among others.

In the following subsections, we'll be looking at some common features used in Jinja: filters, if statements, loops, macros and template inheritance.

Filters

Despite the similarity between Python and Jinja's syntax, modifications such as joining strings, setting the first character of a string to uppercase, and so on cannot be done using Python's syntax in Jinja. Therefore, to perform such modifications, we have filters in Jinja.

A filter is separated from the variable by a pipe symbol (|) and may entertain optional arguments in parentheses. A filter is defined in this format:

```
{{ variable | filter_name(*args) }}
```

If there are no arguments, the definition becomes the following:

```
{{ variable | filter_name }}
```

Let's take a look at some common filters in the following subsections.

The default filter

The `default` filter variable is used to replace the output of the passed value if it turns out to be `None`:

```
{{ todo.item | default('This is a default todo item') }}
This is a default todo item
```

The escape filter

This filter is used to render raw HTML output:

```
{{ "<title>Todo Application</title>" | escape }}
<title>Todo Application</title>
```

The conversion filters

These filters include `int` and `float` filters used to convert from one data type to another:

```
{{ 3.142 | int }}
3

{{ 31 | float }}
31.0
```

The join filter

This filter is used to join elements in a list into a string as in Python:

```
{{ ['Packt', 'produces', 'great', 'books!'] | join(' ') }}
Packt produces great books!
```

The length filter

This filter is used to return the length of the object passed. It fulfills the same role as `len()` in Python:

```
Todo count: {{ todos | length }}
Todo count: 4
```

> **Note**
> For a full list of filters and to learn more about filters in Jinja, visit `https://jinja.palletsprojects.com/en/3.0.x/templates/#builtin-filters`.

Using if statements

The usage of `if` statements in Jinja is similar to their usage in Python. `if` statements are used in the `{% %}` control blocks. Let's look at an example:

```
{% if todo | length < 5 %}
    You don't have much items on your todo list!
{% else %}
    You have a busy day it seems!
{% endif %}
```

Loops

We can also iterate through variables in Jinja. This could be a list or a general function, such as the following, for example:

```
{% for todo in todos %}
         <b> {{ todo.item }} </b>
{% endfor %}
```

You can access special variables inside a `for` loop, such as `loop.index`, which gives the index of the current iteration. The following is a list of the special variables and their descriptions:

Variable	Description
`loop.index`	The current iteration of the loop (1 indexed)
`loop.index0`	The current iteration of the loop (0 indexed)
`loop.revindex`	The number of iterations from the end of the loop (1 indexed)
`loop.revindex0`	The number of iterations from the end of the loop (0 indexed)
`loop.first`	True if first iteration

Variable	Description
`loop.last`	True if last iteration
`loop.length`	The number of items in the sequence
`loop.cycle`	A helper function to cycle between a list of sequences
`loop.depth`	Indicates how deep in a recursive loop the rendering currently is; starts at level 1
`loop.depth0`	Indicates how deep in a recursive loop the rendering currently is; starts at level 0
`loop.previtem`	The item from the previous iteration of the loop; undefined during the first iteration
`loop.nextitem`	The item from the following iteration of the loop; undefined during the last iteration
`loop.changed(*val)`	True if previously called with a different value (or not called at all)

Macros

A macro in Jinja is a function that return an HTML string. The main use case for macros is to avoid the repetition of code and instead use a single function call. For example, an input macro is defined to reduce the continuous definition of input tags in an HTML form:

```
{% macro input(name, value='', type='text', size=20 %}
    <div class="form">
        <input type="{{ type }}" name="{{ name }}"
         value="{{ value|escape }}" size="{{ size }}">
    </div>
{% endmacro %}
```

Now, to quickly create an input in your form, the macro is called:

```
{{ input('item') }}
```

This will return the following:

```
<div class="form">
    <input type="text" name="item" value="" size="20">
</div>
```

Now that we have learned what macros are, we will proceed to learn what template inheritance is and how it works in FastAPI.

Template inheritance

Jinja's most powerful feature is the inheritance of templates. This feature advances the **don't repeat yourself** (**DRY**) principle and comes in handy in large web applications. Template inheritance is a situation where a base template is defined and child templates can interact, inherit, and replace defined sections of the base template.

> **Note**
>
> You can learn more about Jinja's template inheritance at `https://jinja.palletsprojects.com/en/3.0.x/templates/#template-inheritance`.

Now that you've learned the basics of Jinja's syntax, let's learn how to use templates in FastAPI in the next section.

Using Jinja templates in FastAPI

To get started, we need to install the Jinja package and create a new folder, `templates`, in our project directory. This folder will store all our Jinja files, which are HTML files mixed with Jinja's syntax. Since this book does not focus on user interface design, we will be making use of the CSS `Bootstrap` library and avoid writing our own styles.

The Bootstrap library will be downloaded from the CDN upon page load. However, extra assets can be stored in a different folder. We will look into serving static files in the next chapter.

We'll start by creating the homepage template, which will house the section for creating new todos. The following is a mockup of how we want our homepage template to look: