# Jinja

Jinja is a web template engine for the Python programming language. It was created by Armin Ronacher and is licensed under a BSD License. Jinja is similar to the Django template engine but provides Python-like expressions while ensuring that the templates are evaluated in a sandbox. It is a text-based template language and thus can be used to generate any markup as well as source code.

The Jinja template engine allows customization of tags, filters, tests, and globals. Also, unlike the Django template engine, Jinja allows the template designer to call functions with arguments on objects. Jinja is Flask's default template engine and it is also used by Ansible, Trac, and Salt.

Jinja is therefore a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax. Then the template is passed data to render the final document. It includes:

* Template inheritance and inclusion.
* Definition and importation of macros within templates.
* HTML templates can use autoescaping to prevent XSS from untrusted user input.
* A sandboxed environment can safely render untrusted templates.
* Async support for generating templates that automatically handle sync and async functions without extra syntax.
* I18N support with Babel.
* Templates are compiled to optimized Python code just-in-time and cached, or can be compiled ahead-of-time.
* Exceptions point to the correct line in templates to make debugging easier.
* Extensible filters, tests, functions, and even syntax.

Jinja’s philosophy is that while application logic belongs in Python if possible, it shouldn’t make the template designer’s job difficult by restricting functionality too much.

## Template Designer

### Template File Extension

A Jinja template is simply a text file. Jinja can generate any text-based format (HTML, XML, CSV, LaTeX, etc.). A Jinja template doesn’t need to have a specific extension: .html, .xml, or any other extension is just fine.

A template contains variables and/or expressions, which get replaced with values when a template is rendered; and tags, which control the logic of the template. The template syntax is heavily inspired by Django and Python. Interestingly enough, any file can be loaded as a template, regardless of file extension. Adding a .jinja extension, like user.html.jinja may make it easier for some IDEs or editor plugins, but is not required. Autoescaping, introduced later, can be applied based on file extension, so the developer will need to take the extra suffix into account in that case. Another good heuristic for identifying templates is that they are in a templates folder, regardless of extension. This is a common layout for projects that is also followed in the case of this thesis.

### Variables

Template variables are defined by the context dictionary passed to the template. Variables can be tweaked and overall ‘messed-with’, as long as they are passed in by the application. They may have attributes or elements on them that the developer can access. Indeed, those attributes have proven particularly useful at handling vital information regarding the application of this thesis and Jinja has certainly helped with simplifying the task required. A variable can be written in one of the following manners:

|  |
| --- |
| {{ row.breakfast.index.PROCTN }} |
| {{ row.breakfast[index][‘PROCTN’] }} |

Both depict the protein count of a food (row) during breakfast, taken from the database array. It is important to notice that the outer double-curly braces are not part of the variable, but the print statement. If the need to access variables inside tags arise, it is advised to not put the braces around them.

Lastly, if a variable or attribute does not exist, an undefined value will be returned. What can be done with that kind of value depends on the application configuration: the default behavior is to evaluate to an empty string if printed or iterated over, and to fail for every other operation.

## API Basics

Jinja uses a central object called the template Environment. Instances of this class are used to store the configuration and global objects, and are used to load templates from the file system or other locations. Even if the developer is in need of creating templates from strings by using the constructor of the Template class, an environment is created automatically for them, albeit a shared one.

Most applications will create one Environment object on application initialization and use that to load templates. In some cases, however, it’s useful to have multiple environments side by side, if different configurations are in use. The simplest way to configure Jinja to load templates for an application is to use PackageLoader.

This will create a template environment with a loader that looks up templates in the templates folder inside the Python package of the application. It also enables autoescaping for HTML files. This loader only requires that application is importable, it figures out the absolute path to the folder for you.

# Bibliography

[1] Jinja Documentation (https://jinja.palletsprojects.com/en/3.0.x/)