Eskimon

Zeste de savoir

Créer un site web statique avec Pelican

24 septembre 2019

Table des matières

I. Présentation de Pelican 5

1. Pelican : pourquoi ? 7
1.1. Un générateur de site statique, qu'est ce que c'est ? . . . 7
1.1.1. Rapide explication . . . 7
1.1.2. Pourquoi ? . . . 7
1.2. Pourquoi utiliser pelican ? . . . 8
1.2.1. Parce que python . . . 8
1.2.2. Parce que Pelican lui-même ! . . . 9
1.3. D'autres générateurs de sites statiques . . . 9
1.3.1. En Python . . . 9
1.3.2. En javascript . . . 10
1.3.3. D'autres langages, Go et Ruby . . . 10

2. Pelican : comment ? 11
2.1. Pip install . . . 11
2.2. Bonjour le monde . . . 12
Contenu masqué . . . 13

II. Générons nos premières pages 14

3. Hello World ! 16
3.1. Quick Start . . . 16
3.1.1. Rappel sur l'installation . . . 16
3.1.2. L'outil de démarrage rapide « *quickstart* » . . . 17
3.2. Qui va là ? . . . 18
3.2.1. L'arborescence en détail . . . 18
3.3. Premiers articles . . . 18
3.3.1. Notre première recette . . . 18
3.3.2. Ajouter des images . . . 20

4. Les métadonnées 21
4.1. Dans l'entête du fichier . . . 21
4.1.1. Données simples . . . 21
4.1.2. Des listes . . . 21
4.1.3. Des dates . . . 22
4.2. Via le contexte du fichier . . . 23
4.2.1. Date de l'article . . . 23
4.2.2. La catégorie . . . 23

4.2.3. Slug et autres . . . 23
4.2.4. Summary . . . 24

5. Différents formats de contenu, un seul rendu 25
5.1. Markdown . . . 25
5.2. reStructuredText - .rst . . . 26
5.3. Hypertext Markup Language - .html . . . 26

6. Changer le style des pages - les thèmes 28
6.1. Activer un thème . . . 28
6.2. Petit aparté sur le serveur de développement intégré . . . 29
6.3. Ajouter un thème . . . 29
6.3.1. Télécharger un thème . . . 30
6.3.2. Installer le thème . . . 31
6.3.3. Paramétrer le thème . . . 31

III. Customisons le rendu de notre site 34

7. Principe 36
7.1. Le moteur de template, kézaco ? . . . 36
7.1.1. Jinja2 . . . 36
7.1.2. Un ensemble de fichiers . . . 36
7.2. Un monde d'embriquement . . . 37
7.2.1. Diviser pour mieux régner . . . 37
7.2.2. Étendre les templates . . . 37
7.2.3. Des blocs . . . 37
7.3. Un exemple . . . 38
Contenu masqué . . . 41

8. La base 44
8.1. Inventaire des fichiers nécessaires . . . 44
8.2. base.html, la colonne vertébrale du site . . . 45
8.3. La section head . . . 46
8.4. La section body . . . 48
8.5. L'en-tête et le pied de page du site . . . 49
8.5.1. L'en-tête (header.part.html) . . . 50
8.5.2. Le pied de page (footer.part.html) . . . 50
8.5.3. Allure globale . . . 51
8.6. La barre latérale . . . 52
8.6.1. La liste des catégories . . . 53
8.6.2. La liste des liens sociaux . . . 54
8.6.3. Rendu final de base.html . . . 54
8.7. One more thing : extends . . . 55
Contenu masqué . . . 56

9. Page de « listing » 59
9.1. Les pages de listes . . . 59
9.1.1. Les pages de haut niveau . . . 59

9.1.2. Les pages de listes d'articles . . . 59
9.2. Les listes de « haut niveau » . . . 60
9.2.1. Modèle générale . . . 60
9.2.2. Un peu d'optimisation . . . 62
9.3. Les listes d'articles . . . 63
9.3.1. Factorisation . . . 65
9.4. Annexe : La pagination . . . 66
9.4.1. L'objet `Page` . . . 67
9.4.2. L'objet `Paginator` . . . 67
9.4.3. Mettre tout cela en œuvre . . . 67
9.4.4. Améliorer le paginateur . . . 69
Contenu masqué . . . 72

10. Rendu d'un article **75**
10.1. Principe . . . 76
10.1.1. Structure . . . 76
10.1.2. Ajout de données dans le `<head>` de la page . . . 76
10.2. L'en-tête de notre article . . . 77
10.3. Le corps et le pied de l'article . . . 78
10.3.1. Le corps de l'article . . . 78
10.3.2. Le pied de page . . . 79
10.3.3. Intégralité de la page . . . 80
10.4. Annexe : Ajouter une section de commentaires avec Disqus . . . 80
10.4.1. Créer un compte Disqus . . . 80
10.4.2. Intégrer Disqus aux pages . . . 82
10.4.3. Afficher un compteur de commentaires . . . 83
Contenu masqué . . . 85

11. [TP] Page d'accueil **89**
11.1. Consigne et indices . . . 89
11.1.1. Consigne . . . 89
11.1.2. Quelques indices et rappels . . . 89
11.1.3. Bon courage ! . . . 90

IV. Aller plus loin **91**

12. Utiliser des plugins **93**
12.1. Généralités sur les plugins . . . 93
12.1.1. *reader* ou *generator*, deux types de plugins . . . 93
12.1.2. Comment installer un plugin . . . 93
12.2. Une extension pour générer un sitemap . . . 94
12.2.1. Installer sitemap . . . 95
12.2.2. Paramétrer le plugin . . . 95
12.3. Une extension pour créer des sous-catégories . . . 96

13. Quelques paramètres de configuration **98**
13.1. Quelques bonnes pratiques . . . 98
13.1.1. Sélection de la config . . . 98

13.1.2. Héritage de configuration 98
13.2. Gestion des fichiers statiques 99
13.2.1. Pour le thème 100
13.2.2. Les contenus statiques 100
13.3. Quelques conseils supplémentaires 101
13.3.1. Pour les flemmards 101
13.3.2. Les der des ders 101

Vous débutez dans le développement web et aimeriez apprendre de nouvelles choses ? Vous savez faire des pages mais en avez marre de retaper le code HTML sans arrêt et aimeriez que votre projet soit plus intelligent et/ou mieux organisé ? Pourquoi dans ce cas ne pas utiliser un générateur de site web statique ? C'est ce que je vous propose ici dans ce tutoriel parlant du générateur nommé « Pelican ».

Ces outils de création offre un bon compromis entre simplicité de rédaction et performance, mais nous verrons cela plus en profondeur dans la première partie de ce tutoriel.

Nous allons ensuite tout de suite rentrer dans le vif du sujet en créant les premières pages de notre site web. Pour cela, je vous propose un fil rouge tout au long de ce cours qui consistera à créer notre propre site de recettes de cookies et de smoothies !

La partie suivante sera consacrée à la customisation de notre site web. Nous commencerons par voir comment utiliser un thème préexistant pour avoir rapidement un rendu correct puis ensuite comment créer de A à Z un design qui nous ressemble.

Enfin, une partie finale facultative présentera quelques notions avancées.

i

Afin de bien suivre le contenu de ce cours, des connaissances de niveau débutant en Python, ainsi qu'en HTML et CSS, sont souhaitables.

Prêt pour l'aventure ? Alors en avant !

Première partie

Présentation de Pelican

Découvrons tout de suite notre nouvel outil : Pélican.

Pour cela, cette partie sera découpée en deux chapitres qui vont vous présenter les générateurs de site statique ainsi que Pelican (et pourquoi ce choix), puis ensuite un court chapitre guidera l'installation de ce dernier.

1. Pelican : pourquoi ?

Ce petit chapitre va vous présenter brièvement le principe des générateurs de sites statiques, ainsi que leurs avantages et inconvénients. Dans un second temps, nous verrons pourquoi j'ai fait le choix de vous parler de Pelican plutôt qu'un autre, tout en restant ouvert à la "concurrence" .

1.1. Un générateur de site statique, qu'est ce que c'est ?

Commençons par le début, qu'est ce qu'un site internet statique ?

S'il y a des sites statiques, c'est qu'il y en a sûrement des dynamiques .

1.1.1. Rapide explication

Les sites que l'on pourrait qualifier de "dynamique" se basent sur l'utilisation d'un logiciel sur un serveur qui va générer les pages à chaque requête. Autrement dit, à chaque fois que vous allez demander une page au site en question, votre requête sera analysée, traitée, puis une page spécifique sera construite juste pour vous et vous sera envoyée. Si Bob et Alice demandent la page A, ils auront *peut-être* une page différente chacun.

Un site statique ne bénéficie pas de cela. Toutes les pages sont avant tout pré-générées grâce à un générateur. On obtient alors une collection de pages HTML. Ce sont ces dernières qui seront alors directement envoyées au lecteur, sans avoir d'autres traitements entre temps. Si Bob et Alice demandent la page A, ils obtiendront tous les deux exactement la même.

1.1.2. Pourquoi ?

Une question vous vient peut-être à l'esprit :

? Pourquoi faire des sites statiques, si le contenu est gravé dans le marbre et qu'on ne peut pas interagir avec ?

Et bien pour plusieurs raisons. Tout d'abord, effectivement cela peut sembler un peu triste, un site qui ne change pas. Mais en y réfléchissant, il existe plein de cas d'usage. Le plus fréquent est sûrement le format blog. L'auteur publie des articles, qui vont créer un ensemble de pages qui fera un site internet. Un lecteur d'un blog de son côté se contente juste de lire les articles, il n'y a pas plus d'interactions que cela (oublions les cas des commentaires et/ou liens sociaux,

qui sont gérables aussi sur des sites statiques). On peut aussi imaginer le cas d'une photothèque. Je veux partager mes photos de vacances à ma famille, pas besoin d'avoir plus d'interactions que ça pour eux, juste de quoi naviguer entre les pages.

Et tout cela a un énorme avantage : **la vitesse**.

En effet, puisque toutes les pages sont pré-générées, le serveur n'a rien d'autre à faire que de les envoyer. Aucun travail de traitement n'est effectué. Si Bob demande la page A, le serveur ne se pose pas de question et renvoie la page A.html. C'est tout. C'est rapide. Et ce genre de mécanisme peut être encore plus rapide grâce à la mise en place de cache qui sera très efficace car les ressources ne changent que très peu.

Un autre atout : **La maintenance**. Une fois votre site en ligne, rien d'autre à faire si ce n'est de le faire vivre en ajoutant de nouveaux articles / nouvelles pages de temps en temps. Pas de base de données à gérer, peu de risque de sécurité, voire même pas de serveur tout court à gérer si vous optez pour une offre d'hébergement mutualisée (nous y reviendrons).

Enfin, **c'est facile** !! Pas besoin de compétences de développeur pour faire tout cela, juste un peu de bidouille pour savoir utiliser le générateur. Ensuite, soit vous utilisez un thème pré-existant, soit vous vous faites le vôtre si vous maîtrisez un peu HTML/CSS.

Tous ces atout permettent ainsi aux créateurs de contenu de se focaliser sur la création de nouveaux articles plutôt que du développement technique et de la maintenance chronophage.

C'est ce qui m'a fait craquer pour cette technologie pour mon blog, eskimon.fr .

1.2. Pourquoi utiliser pelican ?

Pourquoi Pelican et pas un autre? C'est une question légitime, à laquelle je vais essayer de répondre du mieux possible. Bien entendu, tout les goûts et les couleurs sont dans la nature. Bien que je vais essayer de rester le plus impartial possible, certains aspects seront propres à mes préférences pour certains langages et à mon historique en tant que développeur…

1.2.1. Parce que python

Python est un langage que j'aime beaucoup. Du coup lors de mes premiers pas avec les générateurs de sites statiques, c'est évidemment vers cet écosystème que je me suis tourné.

Plus objectivement, python a l'avantage d'avoir une communauté très importante. C'est un langage qui fait maintenant partie des leaders du marché et qui bénéficie d'une très grande communauté. Il est donc souvent très facile d'avoir des réponses à ses questions rapidement via une simple recherche internet ou en posant sa question sur des forums comme ceux de Zeste de Savoir.

L'avantage d'une grande communauté est aussi de pouvoir trouver de (très) nombreux exemples de code pour pouvoir s'inspirer et progresser. De la même façon, de nombreux plugins pour tout les systèmes existent bien souvent, faisant ainsi gagner du temps plutôt que de réinventer la roue. Et si ces derniers fonctionnent mal ou sont incomplets, il sera bien souvent possible de proposer des corrections ou améliorations à l'auteur initial si le code est open-source.

1.2.2. Parce que Pelican lui-même !

Pelican tire selon moi son épingle du jeu pour plusieurs aspects.

Tout d'abord, ce n'est pas un projet à l'abandon. À l'heure d'écriture de ces lignes, les derniers commits dataient de moins d'un mois ! Les commits les plus vieux ont quant à eux plusieurs années derrière eux. C'est plutôt une bonne chose, le projet semble mûr.

Autre aspect, c'est open-source. C'est toujours cool de pouvoir aller lire les sources quand on a une incompréhension ou par simple curiosité.

Après lecture de la documentation, la rédaction m'a semblé simple et intuitive (notamment pour la rédaction des métadonnées des articles). Le support natif du markdown a été pour moi un gros plus étant très familier avec de langage de rédaction.

Enfin, de manière plus pragmatique, Pelican est facile à customiser (j'en parle dans ce billet) et surtout le rendu est simple à customiser. En effet, le moteur de template utilisé est jinja2, qui ressemble fortement à celui de base de Django et est bien connu dans le monde python (et donc offre de nombreuses réponses à toutes les questions que l'on peut se poser). Ayant pas mal utilisé ce dernier durant mes développements sur ZdS, je suis bien content d'avoir affaire à quelque chose de familier.

1.3. D'autres générateurs de sites statiques

Ne soyons pas sectaires, voyons quelles autres solutions existent, que ce soit en python ou dans d'autres langages. Ce chapitre sera rédigé grâce notamment au site staticgen qui liste les différents générateurs de site statiques open-source existant. Il est toujours bon de voir un peu plus loin que le bout de son nez. Savoir ce qu'offre chez la "concurrence" ne fait donc pas de mal, qui sait, peut-être un jour vous vous inspirerez d'une fonctionnalité existante chez un autre pour la porter sur Pelican!

Par souci de concision, je ne serai bien entendu pas exhaustif concernant tous les outils existants. Je ne ferai donc qu'une brève présentation de ceux dont j'ai le plus entendu parler ou même testé au moment de l'écriture de ce tutoriel.

1.3.1. En Python

Au moment d'écriture de ces lignes, le service staticgen liste un peu plus de 25 outils en python. Je vais couper sévèrement dans la liste pour vous en présenter deux, Sphinx et Lektor.

En python il existe un générateur que vous connaissez sûrement si vous lisez des documentations techniques d'outils open-source : Sphinx . En effet, sphinx est utilisé par exemple pour le site de la documentation officielle du langage python. Un autre grand acteur de la documentation open-source, ReadTheDocs, utilise aussi Sphinx pour la génération de ses contenus. Sphinx est donc un grand acteur du monde python.

Un autre outil populaire méritant d'être cité est Lektor . Je le trouve cependant un peu plus compliqué d'utilisation et de paramètrage, c'est notamment pourquoi je préfère vous présenter Pelican dans ce tutoriel. En revanche, Lektor propose un petit serveur web local permettant d'offrir un petit gestionnaire de contenu à la volée et une interface de rédaction qui est un petit plus plutôt sympa.

1.3.2. En javascript

En rédigeant cette section, j'ai découvert Hexo . Je ne connaissais pas avant, mais quand on voit le premier argument "générer des dizaines de pages en une seconde" forcément ça fait rêver. Cet outil utilise du Node.js, qui devient un grand classique du monde javascript. Comme partout, la rédaction se fait en markdown et la génération des pages se fait en une commande. Je ne peux pas en dire beaucoup plus, ne connaissant pas vraiment l'outil.

Un autre outil (plus connu ?) est GitBook . Il est assez connu des personnes voulant rédiger des ouvrages techniques je pense et possède notamment une interface de rédaction et publication en ligne. L'idée derrière ce générateur est de rapprocher le monde de l'édition conventionnel papier au monde de l'édition numérique, notamment en proposant par défaut une publication au format pdf. Les ouvrages sont versionnés via le compte de l'auteur sur GitHub.

1.3.3. D'autres langages, Go et Ruby

Tout comme pour Hexo, j'ai découvert Hugo lors de la rédaction et donc ne vous le présenterais que pour le principe. Hugo est codé en Go, langage édité par Google et se voulant très efficace. Ce qui fait que là encore, on retrouve la vitesse de génération comme argument principal d'Hugo. Si vous êtes familier avec le Go, allez donc y jeter un œil

En dernier lieu, je souhaiterais vous présenter Jekyll non pas parce que je suis fan de ruby (je n'en ai jamais touché), mais surtout car il fait figure d'étoile montante des générateurs de site statique. En effet, Jekyll est embarqué comme outil de rédaction par plusieurs grands services d'hébergements de pages web statiques, comme Github ou Netlify.

J'espère que les choses sont maintenant clairs sur les tenants et aboutissants des différentes technologies. Bien entendu, aucune solution n'est parfaite, mais vous avez maintenant des moyens de comparaison.

La prochaine partie va nous amener doucement vers la pratique en voyant comment installer Pelican.

2. Pelican : comment ?

Dans ce court chapitre, je vous présente la manière la plus simple pour installer Pelican.

> ! Étant donné que c'est un logiciel en Python, les manipulations décrites ci-dessous sont facilement transposables à n'importe quel OS. Je décrirai ici comment le faire sur un système Linux, Ubuntu en particulier.

> *i* Cette partie part du principe que vous connaissez l'outil de gestion de paquets python `pip` et que vous savez l'utiliser de manière simple (installer des paquets, les supprimer...)

2.1. Pip install

> Contenu masqué n°1

Installer Pelican est vraiment trivial, puisqu'il suffit d'installer seulement deux paquets, `pelican` et `Markdown`. Comme nous ne sommes pas des gros bourrins, on va toutefois travailler proprement dans un environnement virtuel.

Je vais donc commencer par créer ce dernier puis l'activer. Je travaillerai dans le dossier "mon-super-site".

```
$ mkdir mon-super-site
$ cd mon-super-site
$ virtualenv venv
$ source venv/bin/activate
$ pip install pelican Markdown
```

Et voilà, Pelican est maintenant installé !!

2.2. Bonjour le monde

Maintenant que les outils sont là, vérifions que tout marche avant de nous lancer à l'assaut de notre propre site !

Pour cela, Pelican met à disposition un outil de démarrage sous la forme d'une commande : `pelican-quickstart`.

Lorsque vous lancez cette dernière, une série de questions vous sera posée. Ignorez celles où un choix par défaut entre crochet est proposé en vous contentant d'appuyer sur entrée et répondez n'importe quoi aux autres, nous supprimerons tout puis reviendrons sur tout cela dès le chapitre suivant.

Un exemple de réponse peut-être le suivant :

```
```

Enfin exécutez la commande `pelican`, le terminal vous répondra qu'aucun article n'a été généré (c'est normal) avec les lignes suivantes :

```
1 WARNING: No valid files found in content.
2 Done: Processed 0 articles, 0 drafts, 0 pages and 0 hidden pages in
      0.05 seconds.
```

Vous devriez voir cependant qu'un dossier output est rempli de fichiers. Ouvrez le fichier `index.html` avec votre navigateur favori et Tadddaaa, votre premier site web statique sous vos yeux !

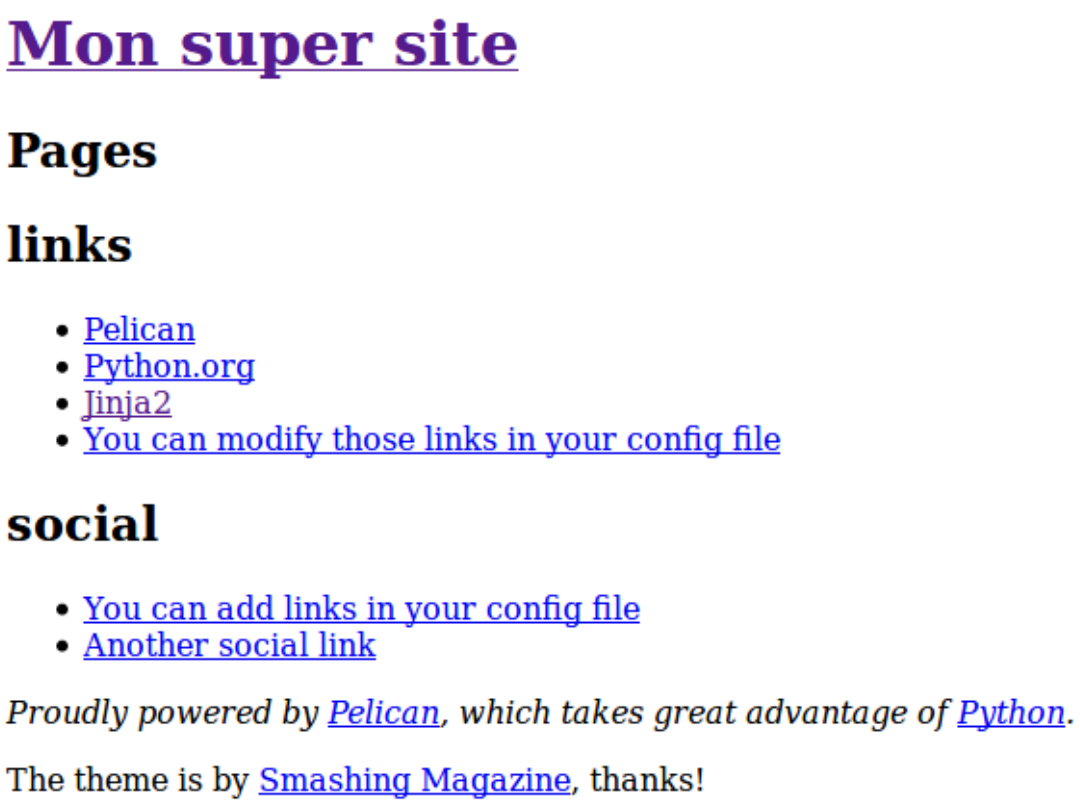


FIGURE 2.1. – premier site

Bon ok, c'est moche et inutile. Remédions à cela en attaquant le chapitre suivant qui va vous permettre de faire VOTRE site .

Vous en savez maintenant un peu plus sur les raisons d'être des outils de générations de sites statiques et vous avez aussi une vue un peu globale de l'écosystème.

Sachez que si python n'est pas votre langage favori, les concepts que nous allons voir dans les parties suivantes sont assez souvent transposables dans les autres outils vus ici.

Mais ne perdons pas plus de temps et rentrons tout de suite dans le cœur du sujet en générant notre site web dans une version simplifiée dans la partie suivante !

Contenu masqué

Contenu masqué n°1

TL ;DR

`pip install pelican Markdown` Retourner au texte.

Deuxième partie

Générons nos premières pages

Maintenant que les présentations sont faites, passons aux choses sérieuses et réalisons nos premières pages.

Dans cette partie, vous allez découvrir comment doivent être formatés les articles que nous allons rédigés. Rien de bien compliqué, simplement un modèle à suivre. Nous verrons aussi la souplesse que nous propose pélican pour pouvoir rédiger selon la convention qui nous plaît le plus (markdown, asciidoc, rst...).

En avant !

3. Hello World !

Le "Hello World !", le grand classique des tutos d'informatique, pas de raison que l'on y échappe ici .

Nous allons voir ici comment bien démarrer un projet dans l'univers Pelican, rien de bien méchant mais c'est important de bien comprendre les bases !

Alors dépoussiérez vos claviers, on y va !

3.1. Quick Start

Si vous lisez ce tutoriel en continu, les dernières actions que nous avons faites pour créer un début de site sont encore fraîche dans votre mémoire. Pour les autres, remettons nous au jus.

3.1.1. Rappel sur l'installation

Pour créer notre site de recettes de cookies (que j'appellerais *coolcookies*), nous allons créer un nouveau dossier de travail dans lequel nous allons tout de suite nous placer.

```
$ mkdir coolcookies
$ cd coolcookies
```

Maintenant, nous créeons un nouvel environnement virtuel que nous activons tout de suite.

```
$ virtualenv venv
$ source venv/bin/activate
```

Enfin, on installe dans cet environnement virtuel les différents paquets dont nous avons besoin (Pelican et markdown).

```
$ pip install pelican Markdown
```

3.1.2. L'outil de démarrage rapide « *quickstart* »

Nous voilà avec notre base de travail. Nous allons maintenant utiliser l'outil « *quickstart* » de Pelican pour préparer le terrain. Cet outil interactif va vous poser quelques questions pour pré-remplir des fichiers de configurations. Pour le lancer, tapez simplement `pelican-quickstart`.

Voici un exemple de réponse possible à fournir, je vous explique juste après à quoi servent les différentes étapes.

```
> Where do you want to create your new web site? [.]
> What will be the title of this web site? Cool Cookies
> Who will be the author of this web site? Eskimon
> What will be the default language of this web site? [fr]
> Do you want to specify a URL prefix? e.g., http://example.com
    (Y/n) n
> Do you want to enable article pagination? (Y/n)
> How many articles per page do you want? [10] 5
> What is your time zone? [Europe/Paris]
> Do you want to generate a Fabfile/Makefile to automate generation
    and publishing? (Y/n) n
> Do you want an auto-reload & simpleHTTP script to assist with
    theme and site development? (Y/n)
Done. Your new project is available at
    /home/eskimon/Documents/tuto-pelican
```

Voyons voir... (pour rappel, si les réponses par défaut (en majuscules) vous conviennent vous n'êtes pas obligés de spécifier quoi que ce soit, appuyez juste sur entrée).

1. Cette première ligne demande où doit être créer le projet. Étant déjà dans un dossier créé pour l'occasion, ce dernier me convient très bien, je laisse donc le choix par défaut (entre crochets).
2. Quel sera le titre de mon site. *Cool Cookies, pour des recettes de cookies qui sont cools.* (Vous remarquerez qu'aucun choix par défaut n'est proposé, l'outil ne lit pas encore dans les pensées).
3. Qui sera le créateur du site. Moi, Eskimon.
4. Quelle sera la langue par défaut des articles. `fr` pour Français.
5. Voulez-vous spécifier une URL de préfixe ? Si vous savez déjà quel domaine utilisera votre site alors spécifiez le. Sinon les URLS seront *relatives*, de la forme `/ma-super-recette` plutôt que `https://coolcookies.fr/ma-super-recette`. Cela ne perturbera pas le développement du site, pas d'inquiétude.
6. Souhaitez vous que les articles soient paginés ? Autrement dit "dans une page contenant une liste d'articles (page de catégorie par exemple), les articles doivent-ils tous être balancés dans une grande liste ou faut-il créer plusieurs pages listant qu'une partie des articles à chaque fois. On choisit de garder la pagination pour l'exemple.

7. Du coup, quelle sera le nombre d'articles maximum par page. 5, c'est bien pour illustrer le phénomène .
8. Quelle est votre fuseau horaire ? Histoire que les articles soient horodatés correctement
9. Voulez-vous générer un fabfile. Si vous ne savez pas ce que c'est, ce n'est pas grave (c'est utilisé pour les déploiements automatisés notamment). Nous nous en passerons dans ce tutoriel, donc répondez **non**.
10. Enfin, voulez-vous un outil pour aider au développement, répondez oui ! (nous en reparlerons plus tard).
11. Bravo, votre squelette de site est prêt !

3.2. Qui va là ?

3.2.1. L'arborescence en détail

Si tout s'est bien passé, voici l'arborescence que vous devriez obtenir (en omettant le dossier de l'environnement virtuel) :

```
```

Comme on le voit dans cette arborescence, il y a plusieurs fichiers et deux dossiers. Voyons brièvement leur rôle.

1.
2. Le dossier `content`. En français "contenu", c'est ici que nous placerons nos rédactions au format markdown.
3. `develop_server.sh`. L'outil d'aide au développement, j'y reviendrais.
4. Le dossier `output`. C'est ici que va se retrouver notre site internet final mis en forme.
5. Le fichier de configuration de Pelican, pour régler certains paramètres permettant de passer du contenu au rendu final.
6. Idem que ci-dessus, mais pour reparamètrer certains aspects.

(Les points 5 et 6 seront vu en détail dans une des parties annexes sur les paramètres de configuration . En effet, ces derniers sont déjà normalement plutôt bien réglés grâce à l'outil quickstart que nous venons d'utiliser.

Mais trêve de blabla, Créons notre premier article !

3.3. Premiers articles

3.3.1. Notre première recette

Sans perdre plus de temps, allez tout de suite créer un fichier dans le dossier `content`. Nommez le `cookies-pepites-chocolat.md`.

Attention à l'extension du fichier. Ici j'ai bien marqué `.md`, qui fera comprendre à Pelican que ce fichier est rédigé en markdown.

Dans ce fichier, collez le contenu suivant.

```

```

Vous pouvez remarquer que le fichier est divisé en deux parties. Tout d'abord, une liste de données du type "Information : Valeur". C'est ce que l'on appelle des *metadata*, métadonnées en français, et nous verrons leur rôle dès le chapitre suivant. Suite à cela, on trouve le corps de notre article, rédigé au format markdown.

Maintenant, à la racine de votre projet (donc pas dans le dossier content ou output), lancer la commande `pelican` (tout simplement). Si tout se passe normalement, vous devriez avoir la réponse suivante :

```
$ pelican
Done: Processed 1 article, 0 drafts, 0 pages and 0 hidden pages in
    0.11 seconds.
```

Vous êtes alors informé qu'un article a bien été traité. Voyons voir ce qui se passe dans notre arborescence dorénavant...

```

```

Vous remarquez que le dossier `output` a maintenant tout un tas de dossiers et de fichiers ? Ceux qui attirent l'attention sont notamment :

- `authors`, un dossier qui propose une page HTML par auteur identifié. Il est accompagné d'un fichier `authors.html`qui va lister tous les auteurs sur le site ;
- `category` et `categories.html`, similaires à auteurs mais pour les categories de contenu sur votre site ;
- `tag` et `tags.html`, vous avez deviné ? ;
- Un dossier `theme` qui contiendra tous les fichiers *statiques* (feuille de style CSS, images, script, etc). Je vous le présenterait en détail dans la partie sur la création de votre propre thème ;
- Enfin, deux fichiers HTML sont à la racine de ce dossier output. `index.html` et `cookies-aux-pepites-de-chocolat.html`. Le fichier index sera l'accueil de votre site, la page qui sera présentée quand un visiteur arrive directement sur la racine du site. La page `cookies-aux-pepites-de-chocolat.html` quant à elle sera la page contenant votre fameuse recette !

Vous pouvez déjà essayé d'aller voir votre travail en ouvrant ces deux pages avec votre navigateur préféré.

Vous devriez alors voir votre recette (oui je sais, c'est moche pour le moment) !

Mais les deux pages sont identiques ?

Effectivement, pour l'instant nous n'avons qu'une seule recette et le rendu par défaut fait que la page index a la même allure que la page de la recette.
Ajoutons une deuxième recette pour voir la différence en créant le fichier suivant dans `content/cookies-bananes.md`

Régénérez le site avec la commande `pelican` et observez de nouveau le dossier output. Un nouvel article a du faire son apparition.

Si vous ouvrez de nouveau la page `index.html`, cette dernière devrait présenter les deux recettes à la suite .

3.3.2. Ajouter des images

Le texte c'est bien, mais avec des illustrations, c'est mieux !

Pour ajouter des illustrations, il va falloir faire plusieurs choses. Tout d'abord, nous allons créer un dossier `images` dans le dossier `content` où nous avons écrit nos articles jusqu'à présent.

Ensuite, il faut informer Pelican que ce dossier est un dossier "statique" et qu'il faut copier le contenu tel quel dans le dossier de sortie "output". Pour cela, il faut ajouter une ligne de configuration dans le fichier `pelicanconf.py`. Cette ligne sera `STATIC_PATHS = ['images']`. Si jamais vous avez d'autres dossiers à exporter (comme un dossier contenant des exports pdf des articles par exemple), il suffit des les rajouter dans la liste.

Enfin, il ne reste plus qu'à utiliser la bonne syntaxe pour insérer l'image. Par exemple, en markdown on écrira `![Texte alternatif de ma belle image](/images/ma-belle-image.png)`.

Et voilà, vous devriez maintenant être familier avec l'arborescence du projet que nous allons manipuler ainsi que les fichiers de base le composant. Nous allons maintenant voir comment enrichir nos articles en ajoutant des informations contextuels via les *metadata*.

Et sinon, avez-vous eu la curiosité d'aller voir les pages supplémentaires dans les dossiers tags ou categories ?

4. Les métadonnées

Comme nous venons de le voir en créant nos premiers articles, un ensemble de données appelé métadonnées servent à définir certains paramètres concernant l'article, comme par exemple son titre ou encore sa date de parution. Nous allons voir dans cette partie les différents moyens utilisables pour définir ces données.

4.1. Dans l'entête du fichier

Le premier moyen mis à notre disposition (et le plus flexible) pour ajouter des données de contexte à notre article est de les placer avant le corps de notre contenu. Pour cela, il suffit de présenter les informations sous la forme d'une liste « clé :valeur », avec une donnée par ligne.

4.1.1. Données simples

En premier lieu, on trouve des données « simples », composées juste d'un texte. On trouve par exemple le titre, la description ou la catégorie de l'article :

```
1 Title: Mon superbe article
2 Summary: Voici un article absolument génial pour présenter les
      choses !
3 Category: Cookie
```

On trouve aussi des données qui doivent uniquement être en « un seul mot », donc avec les espaces représentés par des tirets (`-`) ou des *underscores* (`_`). C'est le cas notamment du *slug* cette chaîne représentant l'identifiant unique de l'article (qui est souvent extrapolé à partir du titre).

```
1 Title: Mon superbe article
2 Slug: mon-superbe-article
```

4.1.2. Des listes

On peut aussi rencontrer des listes, comme par exemple une liste de *tags* pour thématiser notre article ou encore une liste d'auteurs. Dans ce cas, le caractère de séparation sera la virgule

```
Tags: cuisine, dessert, cookies
Authors: Eskimon, Clementine Sanspepins, Jean Bonbeurre
```

Si jamais vos listes contiennent des textes avec des virgules, alors vous devez utiliser le `;` comme séparateur.

4.1.3. Des dates

Afin d'informer les lecteurs de la fraîcheur de nos contenus, la date de publication est essentielle ! La date de modification peut elle aussi être utile. Les dates sont des contenus particuliers puisqu'elles doivent être rédigées sous une forme compatible avec le sous-ensemble ISO 8601 . Les différents formats sont :

- Année uniquement : `AAAA` (exemple : `2018`)
- Année et mois : `AAAA-MM` (exemple : `2018-06`)
- Date complète : `AAAA-MM-JJ` (exemple : `2018-06-04`)
- Date complète + heures et minutes : `AAAA-MM-JJThh:mmTZD` (exemple : `2018-06-04T15:28+01:00`) (Attention au `T` séparant date et heure ainsi qu'à l'indicateur de timezone)
- Date complète + heures, minutes, secondes : `AAAA-MM-JJThh:mm:ssTZD` (exemple : `2018-06-04T15:30:25+01:00`)
- Date complète + heures, minutes, secondes et une fraction de seconde : `AAAA-MM-JJThh:mm:ss.sTZD` (exemple : `2018-06-04T15:31:26.42+01:00`)

Par exemple pour un article paru le 6 mai 2018 et mis à jour le 25 décembre 2018 :

```
Date: 2018-05-06
Modified: 2018-12-25
```

Le format horaire peut aussi être exprimé en UTC , dans ce cas il faut supprimer la partie ajoutant l'heure et mettre un Z à la place. Ainsi, `2018-06-04T15:28+01:00` devient `2018-06-04T14:28Z`

Voici un aperçu des métadonnées les plus usuelles. Vous aurez remarqué qu'elles sont en anglais . Si jamais vous souhaitez ajouter des données supplémentaires, vous être libre de le faire ! La seule condition est que la clé soit en un seul mot (par exemple une clé "lien canonique" pourrait être notée "lien_canonique").

Seul le titre est obligatoire. Toutes les autres données seront extrapolées des données du fichier lui-même si jamais elles sont absentes. Voyons cela.

4.2. Via le contexte du fichier

Comme expliqué un peu plus tôt, seul le titre (*Title*) est obligatoire. Le reste est alors lu à partir des données du fichier et sous couvert que le fichier de configuration `pelicanconf.py` est bien paramêtré. Voyons cela.

4.2.1. Date de l'article

Si aucune date n'est présente dans les metadonnées, alors ce sera la date de création du fichier qui sera prise en compte. Il faut pour cela que le paramètre de configuration `DEFAULT_DATE` ai pour valeur `fs` (signifiant *filesystem*).

```
1 'DEFAULT_DATE' = 'fs'
```

Le champ `Modified` n'exploite pas les données du fichier. Il aura la même valeur que `Date` s'il n'est pas précisé.

4.2.2. La catégorie

La catégorie de l'article (*Category*) peut-être devinée via le chemin du fichier. Un document "choco-banane.md" situé dans le dossier "cookie" (`content/cookie/choco-banane.md`) sera alors rangé dans la catégorie "cookie".

Si jamais le paramètre de configuration `USE_FOLDER_AS_CATEGORY` est à `False` alors le champ `DEFAULT_CATEGORY` prend le relais.

4.2.3. Slug et autres

Le nom du fichier peut-être paramétré pour retirer des données. Par exemple on pourrait vouloir que le nom de notre fichier soit aussi le slug de l'article. Pour cela, il faut régler le paramètre `FILENAME_METADATA` de façon à décrire "où" sont les données dans le fichier. Si par exemple le fichier s'appelle `2018-05-06_de-bons-cookies.md` et que `FILENAME_METADATA = '(?P<date>\d{4}-\d{2}-\d{2})_(?P<slug>.*)'`, pelican automatiquement pourra sortir la date au format AAAA-MM-JJ puis le slug.

4.2.4. Summary

Le champ *Summary* quant à lui exploitera directement le début de l'article pour se former automatiquement s'il n'est pas spécifié. Un paramètre de configuration, `SUMMARY_MAX_LENGTH`, permet de régler le nombre de mots du début de l'article seront utilisé pour construire le résumé.

Dernière information importante, les données précisé dans l'entête de l'article ont toujours la priorité.

Vous avez maintenant toutes les cartes en mains pour paramétrer vos articles, il ne restent plus qu'à les écrire ! Pour cela, le prochain chapitre vous propose d'aller à la découverte des différents formats vous permettant de rédiger.

5. Différents formats de contenu, un seul rendu

Pelican étant sympa, l'outil propose différentes méthodes de rédaction pour correspondre au goût du plus grand nombre. Parmi les plus habituels, on retrouve le markdown, le rst ou encore le HTML. Je vais vous présenter leur utilisation dans Pelican dans ce chapitre, mais je ne rentrerais pas dans le détail concernant leur mécanisme de rédaction. Aussi, ormis les différences dans la syntaxe de la rédaction, les metadonnées ne seront pas écrites pareil, et c'est là surtout l'objet de ce court chapitre.

5.1. Markdown

Le premier moyen de rédaction que je souhaite vous présenter est le markdown. Pour l'utiliser il faudra que vos fichiers aient l'extension `.md` (les extensions `.markdown`, `.mkd` ou `.mdown` sont aussi acceptées). Aussi, le *parseur* markdown doit-être installé via la commande `pip install markdown`.

Markdown est un format de rédaction très agréable à utiliser car son *markup* est très léger, ce qui fait qu'un document rédigé mais non mis en forme reste tout de même très lisible.

La syntaxe des métadonnées lorsque l'on utilise le format markdown est celle que l'on a vu dans le chapitre précédent, à savoir une liste de clé :valeur.

```
1  Title: Ma super recette de cookies
2  Date: 2018-06-06 09:00
3  Modified: 2018-06-07 14:42
4  Category: Cookie
5  Tags: cookie, chocolat, recette
6  Slug: ma-super-recette-de-cookies
7  Authors: Eskimon
8  Summary: Dans cette article je vous présente une recette pour des
       cookies délicieux !
9
10 Voici le corps de l'article, blablabla...
```

5.2. reStructuredText - .rst

Le format reStructuredText possède quant à lui l'extension `.rst`. Il ne nécessite pas l'installation d'un package supplémentaire particulier. Pour ceux qui l'ignore, c'est le langage utilisé pour la rédaction de la documentation de python !

Là encore, au delà des sucres syntaxiques, tout se joue dans l'organisation des métadonnées. On retrouve de nouveau une liste de clé :valeur au format `:clé:valeur` (`:` avant la clé). Regardez bien l'exemple suivant, le titre lui aussi change d'aspect.

```
Ma super recette de cookies
###########################

:Date: 2018-06-06 09:00
:Modified: 2018-06-07 14:42
:Category: Cookie
:Tags: cookie, chocolat, recette
:Slug: ma-super-recette-de-cookies
:Authors: Eskimon
:Summary: Dans cette article je vous présente une recette pour des
    cookies délicieux !

Voici le corps de l'article, blablabla...
```

5.3. Hypertext Markup Language - .html

Le dernier langage utilisable lui aussi sans extension est le HTML. Indispensable pour faire des pages web, il permet aussi de rédiger directement vos articles. Il est cependant bien plus verbeux que les deux précédents que nous venons de voir.

Pour l'utiliser, il faudra que les fichiers des articles aient l'extension `.html` ou `.htm`. Ensuite, il suffit d'imaginer rédiger une page web classique, en se contentant du strict minimum. On trouve alors les metadonnées dans la balise `<head>` et le corps de l'article dans `<body>`. Le tout étant englobé d'une balise `<html>`.

Voici notre exemple de nouveau :

```
<html>
    <head>
        <title>Ma super recette de cookies</title>
        <meta name="tags" content="cookie, chocolat, recette" />
        <meta name="date" content="2018-06-06 09:00" />
        <meta name="modified" content="2018-06-07 14:42" />
        <meta name="category" content="Cookie" />
        <meta name="authors" content="Eskimon" />
```

```
        <meta name="summary" content="Dans cette article je vous
            présente une recette pour des cookies délicieux !" />
    </head>
    <body>
        Voici le corps de l'article, blablabla...
    </body>
</html>
```

Voilà cette rapide présentation faite. Comme vous pouvez vous en doutez, bien que la rédaction aura un aspect différent dans le document brut, le rendu sera quant à lui le même.

Si jamais vous souhaitez utiliser un autre format de rédaction, il existe de nombreux plugins ajoutant ces fonctionnalités. Veillez à bien consulter leur documentation, la mise en forme des métadonnées peut changer d'un article à l'autre. Pour ceux voulant aller plus loin, j'avais même fait un billet traitant de l'ajout du zmarkdown dans Pelican pour pouvoir rédiger avec le même markdown que celui de Zeste de Savoir !

6. Changer le style des pages - les thèmes

Maintenant que nous savons rédiger, il va être temps de rendre notre site un peu plus joli ! En effet, les textes noirs sur fond blanc avec zéro mise en forme ça va bien 2 minutes...

Avant de nous attaquer à la réalisation de notre propre thème, nous allons passer par l'utilisation de thème crée par la communauté Pelican.

6.1. Activer un thème

Commencons par voir ce qui existe par défaut avant de nous lancer dans des choses plus compliquées.

À l'heure d'écriture de ces lignes, Pelican propose deux thèmes par défaut : « notmyidea » et « simple ». On peut retrouver cette information en tapant la commande `pelican-themes -l` qui donnera en retour la liste des thèmes installés dans l'environnement actuel.

Le thème « simple » porte plutôt bien son nom puisqu'il n'applique (presque) aucun style à la page. En revanche, le thème « notmyidea » est un tout petit plus évolué et propose une vraie mise en forme.

Pour changer de thème, il suffit simplement de changer le paramètre de configuration `THEME` dans le fichier de configuration `pelicanconf.py` et lui inscrire le nom du thème que l'on souhaite. Par exemple, pour utiliser le thème « notmyidea » on écrit : `THEME = 'notmyidea'`. Si ensuite vous regénerez le site, vous pourrez voir la différence de rendu dans les pages.

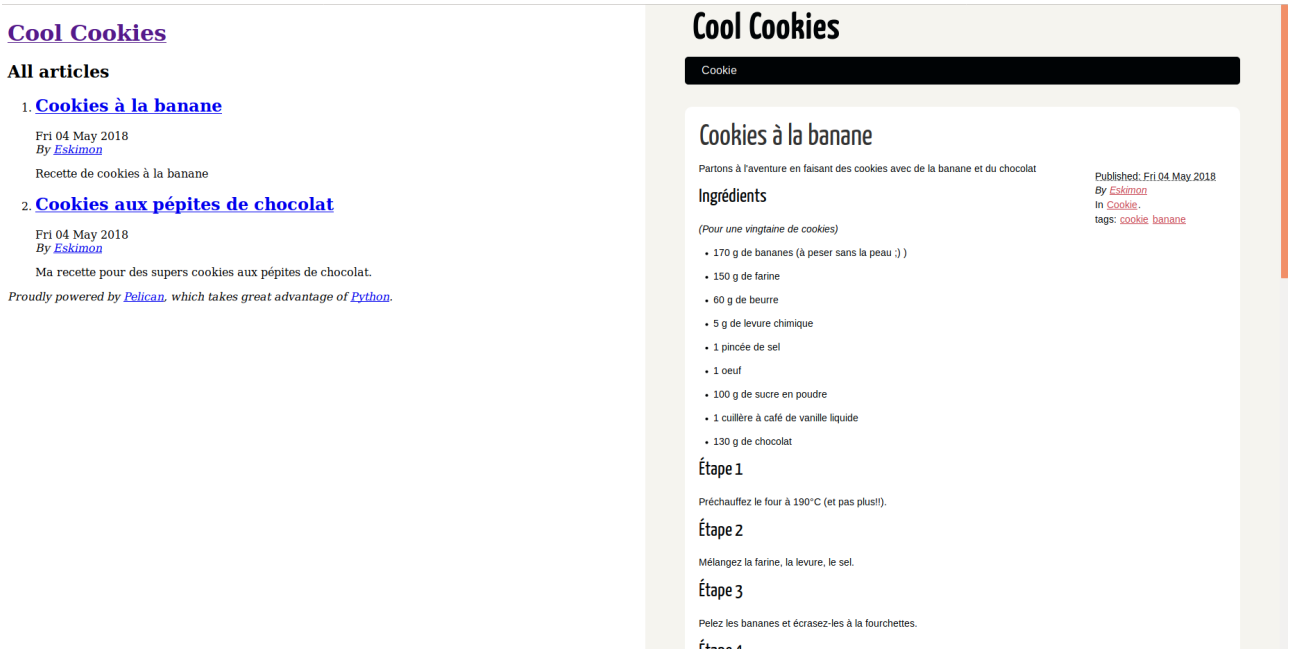


FIGURE 6.1. – Thème « simple » à gauche, « notmyidea » à droite

6.2. Petit aparté sur le serveur de développement intégré

Dans le premier chapitre de cette partie, je vous ai parlé d'un outil intégré à Pelican : le serveur de développement. Cet outil surveille les fichiers de votre dossier de travail et relance une génération à chaque fois qu'un élément change (nouvelle élément dans le CSS, changement d'un paramètre de configuration, ajout/édition d'un article, etc). Le but bien entendu est de gagner du temps en évitant d'oublier de générer les fichiers lors d'un changement et pour ne pas s'embêter à sans arrêt retourner taper la commande pour lancer la génération.

Un autre avantage du serveur de développement est qu'il permet de "servir" les fichiers statiques. En effet, si vous développez sans et que vous essayer de visiter votre site, vous avez peut-être remarqué que le CSS ou les images ne se chargent pas. Le boulot du serveur de développement est de palier à cela. Il réceptionne les demandes de chargement de fichier et les envoi à votre navigateur pour que le site se charge correctement.

Bref, c'est un outil bien utile qu'il ne faut pas hésiter à utiliser !

Pour l'utiliser, il suffit de taper la commande `./develop_server.sh start`. L'outil va alors directement générer votre site et continuer à tourner en tâche de fond. À chaque changement de fichier le site sera automatiquement régénéré et le statut s'affichera dans le terminal sans le bloquer pour autant. Dorénavant, pour visiter votre site vous pouvez directement aller à l'adresse http ://127.0.0.1 :8000 . Notez bien la présence de `:8000` qui précise au navigateur le port à utiliser pour accéder au serveur.

i Si pour une raison ou une autre vous souhaitez changer le port utilisé par le serveur de développement, vous pouvez le préciser en dernier paramètre de la commande. Par exemple pour utiliser le port 8080, il suffira de taper `./develop_server.sh start 8080`.

Pour arrêter le serveur de développement, retaper le nom du script suivi de l'argument "stop" : `./develop_server.sh stop`.

6.3. Ajouter un thème

Dans cette dernière section, nous allons voir comment utiliser un thème pré-existant, qui aura été fait par un tiers de la communauté Pelican. Afin de télécharger le thème qui nous plaît, nous allons devoir distinguer plusieurs cas de figures. En effet, selon que le thème possède son propre dépôt git ou est hébergé de manière plus rudimentaire va influencer la manière de le récupérer. Mais à la fin, tout marchera de la même façon !

Pour cela, je vais vous faire utiliser deux thèmes répondants à ces deux cas de figures : attila et bootstrap.

6.3.1. Télécharger un thème

Tout d'abord, il est bon de savoir que les thèmes open-source crée par la communauté Pelican sont regroupés dans le dépôt Github pelican-themes . C'est donc tout naturellement là-bas que nous allons récupérer nos exemples.

> Si vous souhaitez avoir un aperçu du rendu d'un thème avant de le télécharger, vous pouvez aller faire un tour sur le site pelicanthemes.com qui recense tout les thèmes publiés sur le dépôt https ://github.com/getpelican/pelican-themes avec leur captures d'écrans respectives si disponibles.

6.3.1.1. ...ayant un dépôt git

Le premier exemple correspondra à l'utilisation d'un thème qui possède son propre dépôt git, géré par son propre mainteneur. On identifie ses derniers facilement en voyant qu'ils ont un « hash" » dans leur nom :

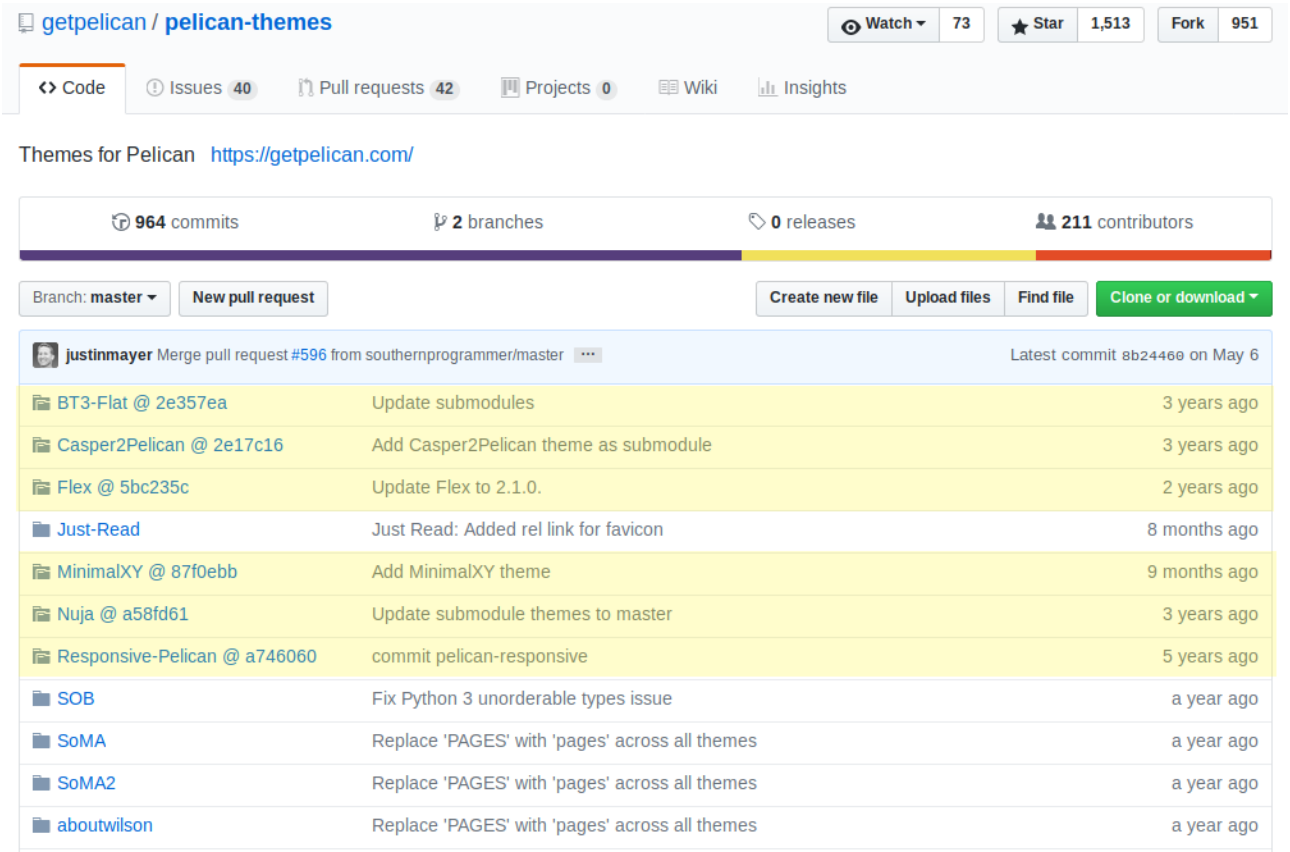


FIGURE 6.2. – En jaune quelques thèmes hébergés dans un sous-dépôt. Notez la présence du « @hash » le signifiant.

Pour l'exemple, récupérons le thème nommé « attila ». Pour cela, trouvez le dans la liste puis cliquez dessus pour vous rendre dans son dépôt . À partir de là, deux options s'offrent à nous. Soit vous connaissez l'utilisation de git, et dans ce cas clonez le dépôt dans un dossier qui vous

plaira sur votre machine. Sinon, cliquez sur le bouton vert à droite « *clone or download* » puis sélectionner l'option « Download zip ». Vous téléchargerez alors un zip que vous pourrez extraire ou vous le souhaitez sur votre ordinateur. C'est tout pour le moment.

6.3.1.2. ...qui n'est qu'un sous-dossier git

Si le thème n'est pas dans un sous-dépôt git, alors il nous faudra récupérer uniquement le dossier qui va bien. Malheureusement, il n'y a pas de solution magique intégrée à Github pour récupérer uniquement un dossier. Nous allons donc passer par un service en ligne externe, DownGit , qui permet cela.

Pour l'utiliser, il suffit d'identifier le chemin du dossier qui nous intéresse. Pour l'exemple, prenons le thème dans le dossier « bootstrap » en cliquant sur ce dernier. La barre d'adresse nous donne alors le chemin https ://github.com/getpelican/pelican-themes/tree/master/bootstrap . Copions ce dernier dans le champ texte de DownGit et cliquons sur « *Download* ». L'outil va alors créer une archive qui va lancer un téléchargement. Vous pouvez alors décompresser cette archive où bon vous semble.

6.3.2. Installer le thème

Maintenant que nous avons récupéré les thèmes, il ne nous reste plus qu'à les installer. Cette étape est très simple puisqu'elle consiste à appeler simplement la commande `pelican-themes` avec l'option `--install`. Par exemple, pour installer le thème bootstrap je ferais : `pelican-themes --install /chemin/vers/le/dossier/du/theme/bootstrap/`. Une fois cela fait, le thème devrait normalement apparaître dans la liste des thèmes installés avec la commande `pelican-themes --list`

6.3.3. Paramétrer le thème

Vous avez peut-être remarqué, certains thèmes proposent des options pour customiser le rendu. Par exemple, dans l'affichage du site avec le thème boostrap on voit sur la gauche une section « Blogroll » et une autre « Social ».

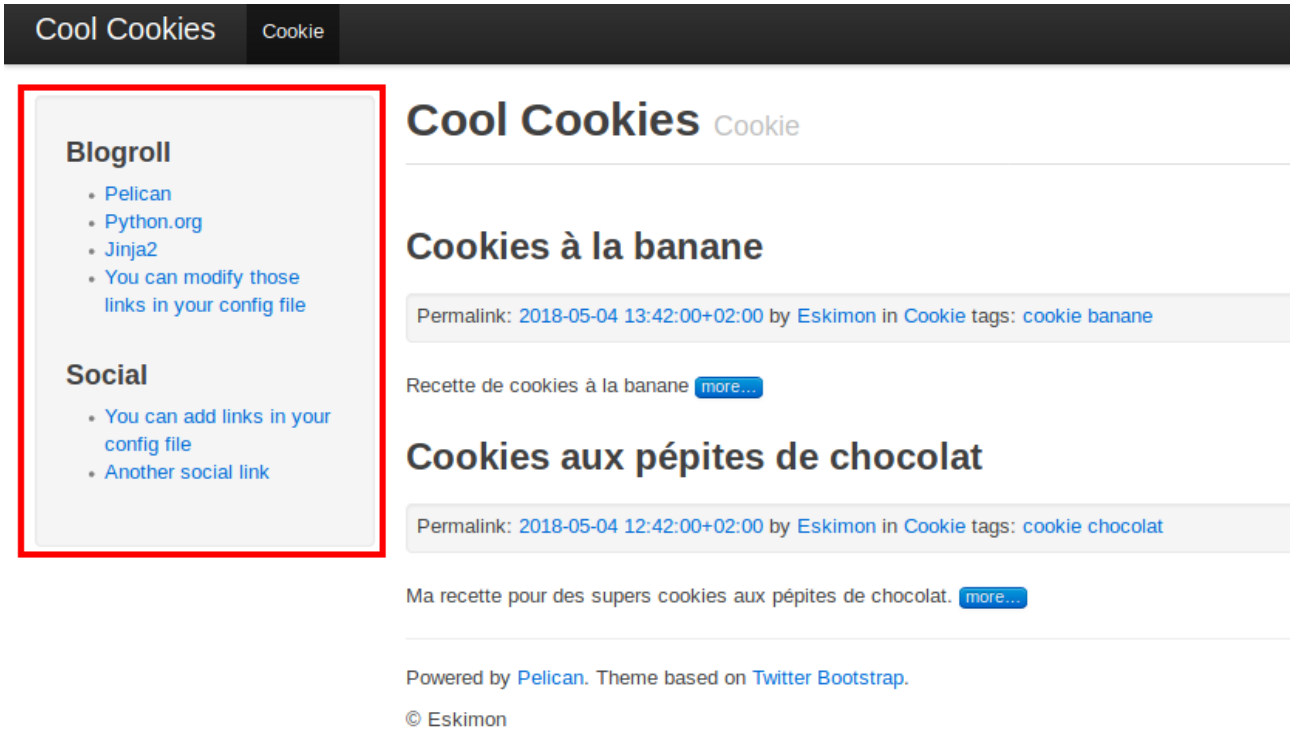


FIGURE 6.3. – Liens externes et sociaux

Le contenu de ces deux sections est assez souvent présentes dans les thèmes et est configurable directement dans le fichier `pelicanconf.py`. Si vous ouvrez ce dernier, vous devriez y trouver les listes suivantes :

```
1 # Blogroll
2 LINKS = (('Pelican', 'http://getpelican.com/'),
3         ('Python.org', 'http://python.org/'),
4         ('Jinja2', 'http://jinja.pocoo.org/'),
5         ('You can modify those links in your config file', '#'),)
6
7 # Social widget
8 SOCIAL = (('You can add links in your config file', '#'),
9          ('Another social link', '#'),)
```

Essayez d'éditer ce contenu et regénerez le site, vous devriez voir vos changements apparaître sans avoir eu à modifier la structure du thème !

i

Il est assez fréquent qu'un thème propose des options de personnalisation. Il est donc important de bien consulter le fichier « Readme » qui est normalement toujours fourni avec ce dernier. Si le travail est bien fait, il vous indiquera les options possibles, si elles

sont facultatives ou non et les valeurs attendues.

Et voilà, cette seconde partie nous aura donnée le plaisir d'avoir un premier résultat. Mais malheureusement, son design reste somme toute sommaire !

Dans la partie suivante, nous allons nous attaquer à rendre tout cela plus attrayant, tout d'abord en explorant des designs (thèmes) déjà existant et produit par d'autres utilisateurs, puis en réalisant le notre de A à Z.

Troisième partie

Customisons le rendu de notre site

L'esprit littéraire a pu s'exprimer dans la partie précédente. Laissons dorénavant place à notre fibre technico-créatrice pour générer un site qui nous ressemble graphiquement.

Étant un piètre designer, je vous accompagnerais ici dans la réalisation d'un thème relativement simple. Vos connaissances en HTML / CSS et recherche sur internet devrons prendre le relais ! L'objectif ici est surtout de vous présenter le fonctionnement de la réalisation d'un thème, ce n'est pas un cours de CSS. Je passerais donc sur ce dernier en me contentant de vous coller les morceaux de style utilisé.

7. Principe

Découvrons rapidement le fonctionnement de la personnalisation des thèmes. Ce chapitre est surtout là pour vous présenter le fonctionnement général et les principes du moteur de template que nous allons utiliser.

7.1. Le moteur de template, kézaco ?

7.1.1. Jinja2

Comme je l'ai rapidement évoqué dans la section « Pourquoi j'ai choisi Pelican », un moteur de *template* (« modèles ») nommé Jinja2 est utilisé par Pelican. Le fait de pouvoir passer par des templates est primordial, puisqu'il permet à Pelican de savoir où mettre chaque élément d'un article, en utilisant un système de variables et de marqueurs pour placer les éléments. Vous verrez, ce n'est pas si compliqué.

Par exemple, notre article possède un titre. Ce dernier sera stocké comme élément `title` dans la variable `article` et on y accédera sous la forme `{{ article.title }}`. Il existe de nombreuses variables, pour chaque élément que l'on peut afficher dans une page. Nous les découvrirons petits à petit dans les prochains chapitres.

7.1.2. Un ensemble de fichiers

Afin de rendre tout ce travail plus confortable, les *templates* se divisent en différentes pages HTML qu'il faudra créer. Ainsi, on aura une page correspondant au rendu d'un article, une autre qui correspond aux rendus d'une liste d'articles, une autre pour l'affichage d'un auteur ou d'une liste d'auteurs etc...

Il nous faudra donc créer chacune des ces pages, afin de gérer tout les cas possibles et garantir une expérience de lecture la plus cohérente et agréable possible pour le visiteur. Tout un programme !

Voici la liste des pages que nous allons créer :

— `archives.html` : Pour gérer l'affichage des archives
— `period_archives.html` : Idem que ci-dessus, mais filtré par période
— `article.html` : Rendu d'un article
— `author.html` : Rendu pour un auteur
— `authors.html` : Liste de tout les auteurs
— `categories.html` : Liste de toutes les catégories
— `category.html` : Rendu pour une catégorie

— `index.html` : Pas d'accueil du site
— `page.html` : Rendu d'une page (format d'article particulier)
— `tag.html` : Rendu pour un tag
— `tags.html` : Liste de tout les tags

Certains templates seront assez similaires, c'est le cas notamment des pages de listes (auteurs, catégories, tags).

7.2. Un monde d'embriquement

7.2.1. Diviser pour mieux régner

Un des autres gros avantages d'un moteur de template est que nous pouvons ainsi diviser notre page en différents morceaux pour *factoriser* les éléments qui se répètent. Éviter les répétitions est primordial, ça fait gagner du temps et surtout évite de faire des oublis de corrections de certains morceaux.

Ainsi par exemple, inutile de copier/coller le bout de code qui gère une barre de titre. Nous allons plutôt le mettre dans son propre fichier et il nous suffira simplement de l'inclure dans notre page pour ajouter tout son contenu via le marqueur `{% include 'chemin/du/morceau.html' %}`.

7.2.2. Étendre les templates

Comme je le disais plus tôt, il vaut mieux éviter autant que possible les répétitions. Pour cela, on peut « étendre » un template. Autrement dit, on va créer un squelette servant de base à notre page, et nous nous contenteront de personnaliser uniquement des blocs de ce dernier. On parle alors d'héritage. Les pages que nous allons créer vont "hériter" d'une page mère, et se contenterons uniquement de modifier des morceaux des blocs de cette dernière.

7.2.3. Des blocs

Les blocs parlons-en justement ! C'est un peu le même principe que l'include, mais dans l'autre sens. Au lieu d'aller chercher un morceau de code pour le rajouter, on déclare un emplacement comme pouvant être personnalisé. Si ce dernier n'est jamais édité, alors aucun morceau de code n'est rajouté et le bloc ne s'affichera pas dans la page finale. Leur syntaxe est la suivante :

```
1 {% block content %}
2 {% endblock %}
```

7.3. Un exemple

Voici un petit exemple d'un design très simple pour mettre en valeur les avantages liés au moteur de template.

Imaginons que nous souhaitons réaliser la page suivante :

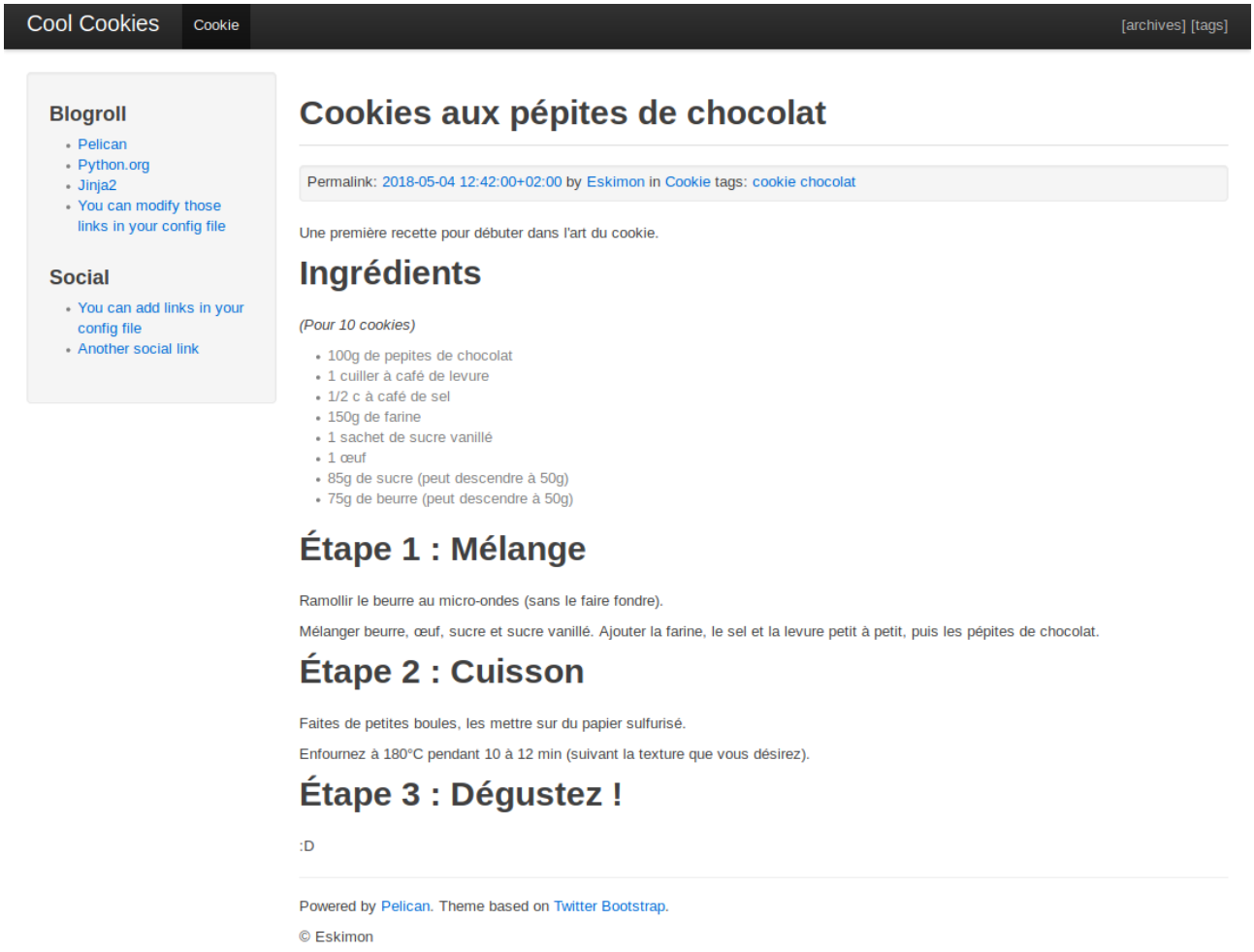


FIGURE 7.1. – Page d'exemple

Nous pouvons identifier différentes parties :

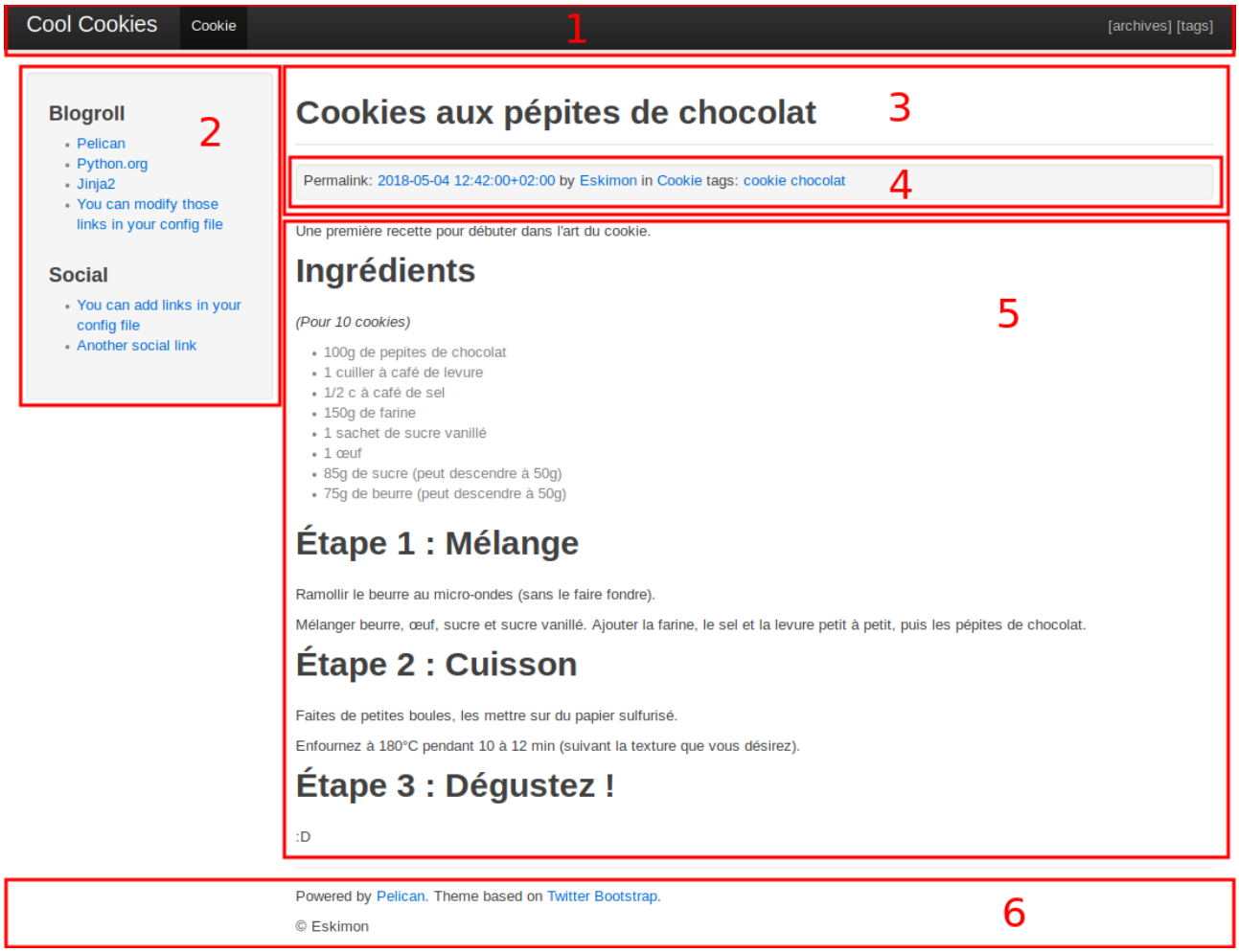


FIGURE 7.2. – Exemple annoté

1. Le *header* (bandeau d'en-tête) de notre site, avec un logo/titre à gauche et des liens à droite ;
2. Une *sidebar* (barre latérale) à gauche pour afficher des catégories, des liens, des infos... ;
3. L'en-tête de l'article, avec son titre et ...
4. ... des métadonnées comme la date de parution ;
5. Le corps de l'article lui-même ;
6. Le *footer* (pied-de-page) de notre site.

Le HTML du rendu de cette page ressemble à ceci :

Contenu masqué n°2

Dans une version « découpée », cela ressemblerait plutôt à un ensemble de morceau s'imbriquant ensemble. On commencerait par une page « globale » qui définirait les emplacements des choses :

```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="utf-8">
    <title>{% block title %}{% endblock %}</title>
    <meta name="description"
        content="{% block description %}{% endblock %}">
    <meta name="author"
        content="{% block author_name %}{% endblock %}">
    <link href="style.css" rel="stylesheet">
</head>

<body>
    {% include 'parts/header.html' %}
    <div class="container-fluid">
        {% include 'parts/sidebar.html' %}
        <div class="content">
            <div class='article'>
                {% block content %}
                {% endblock %}
            </div>
        </div>
    </div>
    {% include 'parts/footer.html' %}
</body>
</html>
```

Avez-vous remarqué les changements ? La balise de titre et de description sont devenues des « blocs » qui seront remplis automatiquement. L'en-tête, le pied-de-page et la barre latérale sont aussi passés dans des fichiers externes, afin de découper notre design et rendre le fichier plus léger et du coup plus facile à lire.

Je conçois que tout cela peut paraître un peu abstrait. Mais vous allez voir en pratiquant que tout cela est en fait très simple à mettre en œuvre tout en étant très puissant pour personnaliser efficacement les pages.

Contenu masqué

Contenu masqué n°2

```
<!DOCTYPE html>
<html lang="fr">

<head>
    <meta charset="utf-8">
    <title>Cool Cookies</title>
    <meta name="description"
        content="Une description concernant cette article">
    <meta name="author" content="Eskimon">
    <link href="style.css" rel="stylesheet">
</head>

<body>
    <header>
        <div class="topbar-inner">
            <div class="container-fluid">
                <a class="brand" href="/">Cool Cookies</a>
                <ul class="nav">
                    <li class="active">
                        <a href="/category/cookie.html">Cookie</a>
                    </li>
                </ul>
                <p class="pull-right">
                    <a href="/archives.html">[archives]</a>
                    <a href="/tags.html">[tags]</a>
                </p>
            </div>
        </div>
    </header>
    <div class="container-fluid">
        <div class="sidebar">
            <div class="well">
                <h3>Blogroll</h3>
                <ul>
                    <li><a
                        href="http://getpelican.com/">Pelican</a></li>
                    <li><a
                        href="http://python.org/">Python.org</a></li>
                    <li><a
                        href="http://jinja.pocoo.org/">Jinja2</a></li>
                    <li><a href="#">You can modify those links in
                        your config file</a></li>
                </ul>
                <div class="social">
```

```
40                      <h3>Social</h3>
41                      <ul>
42                          <li><a href="#">You can add links in your
                                config file</a></li>
43                          <li><a href="#">Another social
                                link</a></li>
44                      </ul>
45                  </div>
46              </div>
47          </div>
48          <div class="content">
49              <div class='article'>
50                  <div class="page-header">
51                      <h1>Cookies aux pépites de chocolat</h1>
52                  </div>
53                  <div class="well small">Permalink:
54                      <a class="more"
                            href="/cookies-aux-pepites-de-chocolat.html">2018-05-04
                            12:42:00+02:00</a>
55                      by
56                      <a class="url fn"
                            href="/author/eskimon.html">Eskimon </a>
57                      in
58                      <a href="/category/cookie.html">Cookie</a>
59                      tags:
60                      <a href="/tag/cookie.html">cookie</a>
61                      <a href="/tag/chocolat.html">chocolat</a>
62                  </div>
63                  <div>
64                      <p>Une première recette pour débuter dans l'art
                            du cookie.</p>
65                      <h1>Ingrédients</h1>
66                      <p>
67                          <em>(Pour 10 cookies)</em>
68                      </p>
69                      <ul>
70                          <li>100g de pepites de chocolat</li>
71                          <li>1 cuiller à café de levure</li>
72                          <li>1/2 c à café de sel</li>
73                          <li>150g de farine</li>
74                          <li>1 sachet de sucre vanillé</li>
75                          <li>1 œuf</li>
76                          <li>85g de sucre (peut descendre à
                                50g)</li>
77                          <li>75g de beurre (peut descendre à
                                50g)</li>
78                      </ul>
79                      <h1>Étape 1 : Mélange</h1>
80                      <p>Ramollir le beurre au micro-ondes (sans le
                            faire fondre).</p>
```

```
81                      <p>Mélanger beurre, œuf, sucre et sucre vanillé.
                            Ajouter la farine, le sel et la levure
                            petit à petit, puis
82                          les pépites de chocolat.</p>
83                      <h1>Étape 2 : Cuisson</h1>
84                      <p>Faites de petites boules, les mettre sur du
                            papier sulfurisé.</p>
85                      <p>Enfournez à 180°C pendant 10 à 12 min
                            (suivant la texture que vous désirez).</p>
86                      <h1>Étape 3 : Dégustez !</h1>
87                      <p>:D</p>
88                      <p>
89                          <img alt="Des cookies"
                                src="/images/cookies.jpg">
90                      </p>
91                  </div>
92              </div>
93              <footer>
94                  <p>Powered by
95                      <a href="http://getpelican.com/">Pelican</a>.
                            Theme based on
96                      <a
                            href="http://twitter.github.com/bootstrap/">Twitter
                            Bootstrap</a>.</p>
97                  <p>&copy; Eskimon</p>
98              </footer>
99          </div>
100     </div>
101 </body>
102 </html>
```

Retourner au texte.

8. La base

Maintenant que le concept est vu, passons à la pratique en nous attaquant tout de suite à la création du squelette de notre site.

> ! Cette partie va être très dense et longue, mais au profit des suivantes qui seront quant à elles plus légères. Je vous recommande donc d'y aller tranquillement, en prenant votre temps pour chaque morceau.

8.1. Inventaire des fichiers nécessaires

Comme je vous l'expliquais dans le chapitre précédent, un certain nombre de fichiers est nécessaire pour créer le design du site. En voici l'arborescence.

```
mon-theme/
├── static
│   ├── css
│   │   └── style.css
│   ├── images
│   │   └── [...]
│   └── js
│       └── [...]
└── templates
    ├── archives.html
    ├── article.html
    ├── author.html
    ├── authors.html
    ├── base.html
    ├── categories.html
    ├── category.html
    ├── index.html
    ├── page.html
    ├── parts
    │   └── [...]
    ├── period_archives.html
    ├── subcategory.html
```

```
23      ├── tag.html
24      └── tags.html
```

Je vous invite donc à créer tout de suite un dossier « mon-theme » directement à la racine de votre site (donc à côté du dossier `content` par exemple. Nous allons ensuite informer Pelican que c'est ce dossier qui servira de thème dorénavant en modifiant simplement la constante `THEME` du fichier `pelicanconf.py` par `THEME = 'mon-theme'`. Profitez-en aussi pour ajouter la ligne `THEME_STATIC_DIR = 'static'` à la suite. Cette information permet de prévenir l'outil que des fichiers *statiques* sont présents dans ce dossier.

Les fichiers statiques ne manipulent pas le contenu des articles, mais servent à la mise en page. Comme par exemple le fichier `style.css` pour le CSS ou encore des scripts javascript ou des images comme votre logo ou une favicon par exemple.

8.2. base.html, la colonne vertébrale du site

Une fois que tout est là, nous pouvons commencer à créer !

Nous allons commencer en éditant le fichier `base.html`n qui se trouve dans le dossier `tem plates/`. Celui-ci va contenir tout le code HTML le plus commun à toutes les pages, tout en exposant des blocs qui seront personnalisés plus tard.

Sans plus de détour, voici le fichier. Je vais vous l'expliquer en détail ensuite.

```
1  <!DOCTYPE html>
2  <html lang="{{ DEFAULT_LANG }}">
3  <head>
4      <meta charset="utf-8" />
5      <meta http-equiv="Content-Type"
           content="text/html; charset=utf-8">
6
7      <title>{% block titre %}{{ SITENAME }}{% endblock %}</title>
8      <link rel="stylesheet"
           href="/{{ THEME_STATIC_DIR }}/css/style.css" />
9      <link rel="icon" type="image/x-icon"
           href="/{{ THEME_STATIC_DIR }}/images/favicon.ico">
10
11     {% block extra_head %}
12     {% endblock %}
13 </head>
14
15 <body>
16     {% include 'parts/header.part.html' %}
17
```

```
18      <div class="main">
19          {% include 'parts/sidebar.part.html' %}
20          <div class="content">
21              {% block content %}
22              {% endblock %}
23          </div>
24      </div>
25
26      {% include 'parts/footer.part.html' %}
27
28      <script src="/{{ THEME_STATIC_DIR }}/js/script.js"></script>
29      {% block extra_js %}
30      {% endblock %}
31  </body>
32  </html>
```

Vous avez remarqué comme il est court ? C'est parce que je fais usage des directives que nous avons vu plus tôt, notamment la fonction `include` permettant d'inclure des morceaux de templates plutôt que de les écrire directement.

Voyons tout cela en détail.

8.3. La section head

Sans surprise, notre fichier reprend la même structure que n'importe quel autre. On trouve dons un *DOCTYPE* tout en haut, puis la balise `<html>` qui englobe le tout. Ensuite vient la balise `<head>` pour définir la section de métadonnée concernant la page puis on trouve le `<body>` qui contiendra le contenu affiché à l'écran.

Revenons sur la section `<head>`.

```
1   <head>
2       <meta charset="utf-8" />
3       <meta http-equiv="Content-Type"
            content="text/html; charset=utf-8">
4       <title>{% block title %}{{ SITENAME }}{%endblock%}</title>
5       <link rel="stylesheet"
            href="/{{ THEME_STATIC_DIR }}/css/style.css" />
6
7       <!-- Favicon -->
8       <link rel="icon" type="image/x-icon"
            href="/{{ THEME_STATIC_DIR }}/images/favicon.ico">
9
10      {% block extra_head %}
11      {% endblock %}
12  </head>
```

Les deux premières lignes ne nous intéressent pas trop ici, elles servent surtout à informer le navigateur pour qu'il interprète correctement le contenu.

En revanche, la ligne suivante est très intéressante car elle fait appel à plusieurs nouveautés.

```
<title>{% block titre %}{{ SITENAME }}{% endblock %}</title>
```

D'un point de vue HTML, c'est simplement la balise `<title>`, qui définira le titre de la page (qui s'affichera dans l'onglet du navigateur). En revanche, son contenu est `{% block titre %}{{ SITENAME }}{% endblock %}`, ça c'est original !

De manière plus clair, nous avons affaire ici à notre premier bloc personnalisable ! Ce bloc s'appelle « *title* » et a pour l'instant comme valeur `{{ SITENAME }}`. J'attire votre attention sur la construction d'un bloc. On commence tout d'abord par créer un marqueur à base d'accolades et de symbole de pourcentage contenant le mot-clé bloc puis un nom qui référence ce bloc . En l'occurrence "titre". On obtient ainsi le marqueur `{% block titre %}`. Ensuite, il faut indiquer quand ce termine le bloc. Là encore avec des accolades et des symboles pourcentage et le mot-clé `endblock` pour former la chaîne `{% endblock %}`.

Ici, notre bloc n'est pas vide, il possède déjà une valeur qui est `{{ SITENAME }}`. C'est une valeur curieuse n'est ce pas ? En fait, c'est carrément l'appel d'une variable globale du site. Cette dernière est renseignée dans le fichier de configuration `pelicanconf.py` et sert à porter le nom de votre site. Par exemple "Cool Cookies". Ainsi, lorsque Jinja rencontre un marqueur composé de double accolades `{{ ... }}` il sait qu'il ne doit pas imprimer l'information telle quelle mais plutôt le contenu de la variable indiquée. Il existe plein de variables globales, nous les verrons au fur et à mesure de nos besoins.

Une fois que le générateur sera passé dessus, si aucune autre page ne vient modifier le bloc "titre", cette ligne se transformera simplement en :

```
<title>Cool Cookies</title>
```

Wahoo. Une seule ligne et on a découvert déjà tant de choses !!

Voyons la suivante :

```
    <link rel="stylesheet"
        href="/{{ THEME_STATIC_DIR }}/css/style.css" />
```

Cette dernière nous sert à inclure une feuille de style qui servira à mettre en forme notre page. J'espère que vous avez remarqué l'utilisation de nouveau d'une variable (globale) `THEME_STATIC_DIR`. Souvenez-vous, au tout début du chapitre je vous ai demandé de la créer pour indiquer à Pelican où se trouvait les fichiers statiques, notamment le CSS. Eh bien là nous y faisons appel ! Concrètement, une fois que le générateur sera passé sur cette ligne elle deviendra :

```
    <link rel="stylesheet" href="/static/css/style.css" />
```

Les plus pointus d'entre vous auront peut-être remarqué que l'adresse est **relative**, en commençant directement par un `/`. Cela nous permet ainsi d'être indépendant du nom de domaine utilisé.

Allez on continue, ligne suivante s'il vous plaît !

```
    <link rel="icon" type="image/x-icon"
        href="/{{ THEME_STATIC_DIR }}/images/favicon.ico">
```

Ici rien de super original, on retrouve ce que l'on vient de voir avec la ligne précédente. La variable `THEME_STATIC_DIR` sera remplacée pour obtenir un chemin valide et ainsi pouvoir afficher notre favicon (qu'il faudra placer dans le dossier `mon-theme/static/images/` avec le nom `favicon.ico` pour que cela fonctionne).

Enfin, le dernier bout de code de cette section <head> est le suivante.

```
    {% block extra_head %}
    {% endblock %}
```

C'est la déclaration d'un bloc nommé « extra_head » et vide pour le moment. Ce dernier sera rempli par le template servant à générer le contenu d'un article, afin de rajouter des métadonnées utiles pour le référencement entre autres. Nous verrons donc cela plus tard .

8.4. La section body

Nous venons déjà de voir un sacré morceau avec la section `<head>`, mais ne nous reposons pas sur nos lauriers, nous avons *presque* fini avec cette première page.

Voyons maintenant la section `<body>`.

```
    {% include 'parts/header.part.html' %}

    <div class="main">
        {% include 'parts/sidebar.part.html' %}
        <div class="content">
            {% block content %}
            {% endblock %}
        </div>
```

```
9     </div>
10
11    {% include 'parts/footer.part.html' %}
12
13    <script src="/{{ THEME_STATIC_DIR }}/js/script.js"></script>
14    {% block extra_js %}
15    {% endblock %}
```

Il commence fort avec une nouvelle instruction, la fonction `include`. Comme brièvement expliqué plus tôt, cette dernière va nous permettre d'inclure un morceau de code qui sera écrit dans un autre fichier. On peut ainsi garder notre fichier plus léger et donc plus facile à lire et à créer. Un autre avantage est pour les morceaux de code qui se répètent (dans une boucle la plupart du temps). On peut ainsi les isoler et les ajouter de manière procédurale assez facilement. Nous verrons ce cas plus tard, lorsque nous étudierons la réalisation des pages de listing.

La fonction `include` a besoin du chemin du fichier HTML à inclure. En l'occurrence, le fichier se trouve dans le dossiers `parts` et se nomme `header.part.html`. Ce dernier contient tout le code de la barre d'en-tête de notre site. Nous verrons ce contenu à la fin de ce chapitre.

Je vous invite à essayer de respecter la convention de nommage `*.part.html` pour les fichiers contenant uniquement des morceaux de code destinés à être inclus. Vous savez ainsi en un clin d'œil qu'il ne se suffise pas à eux-mêmes mais sont destinés à être inclus dans un autre fichier.

Lorsque Jinja va croiser cette ligne, il va tout simplement copier et interpréter tout le contenu du fichier visé.

Le morceau suivant ne fait pas grand chose. Il se contente simplement de déclarer une `<div>` avec la classe CSS `main` qui contiendra de nouveau un include pour le code de la barre latérale et un bloc qui sera personnalisé plus tard (vide pour le moment).

Ensuite, on retrouve de nouveau un `include` mais cette fois-ci se sera pour ajouter le pied-de-page de notre site. Les mêmes remarques que pour l'en-tête s'appliquent.

Enfin, le dernier morceau de code contient à la fois l'inclusion d'un fichier javascript provenant du dossier `static` (remplacement d'une variable globale), et ensuite la déclaration d'un bloc "extra_js", vide pour le moment, qui sera personnalisé par d'autre template plus tard.

8.5. L'en-tête et le pied de page du site

Bon c'est bien sympa tout ça mais pour l'instant on a rien fait du tout ! Notre site n'affiche qu'une simple page blanche.

Je vous propose donc de compléter les fichiers `header.part.html` ainsi que `footer.part.html` pour donner un peu de corps (ahah) à notre page.

8.5.1. L'en-tête (header.part.html)

Commençons en toute logique par l'en-tête du site, c'est à dire la « barre de titre » tout en haut de notre page.

Cette dernière aura l'allure suivante :

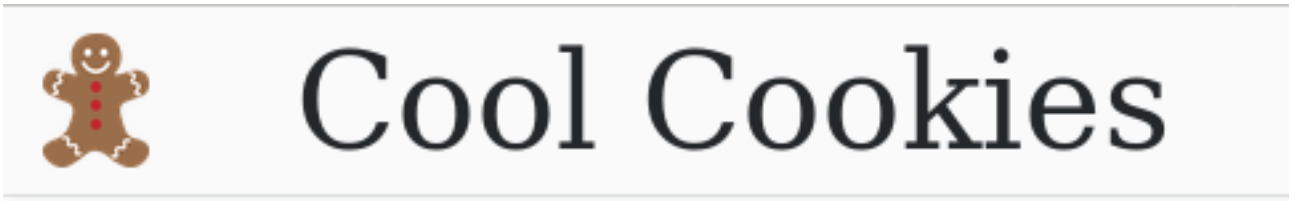


FIGURE 8.1. – En-tête

Son code sera tout simple, et composé d'une balise `<header>` contenant un lien avec une image (redirigeant vers la racine du site) et un span contenant le nom du site.

```
<header class="header-bar">
    <a class="header-nav" href="/">
        <img src="/{{ THEME_STATIC_DIR }}/images/logo.png"
            alt="{{ SITENAME }} logo">
    </a>
    <span class="header-title">{{ SITENAME }}</span>
</header>
```

Rien de bien transcendant, j'attire toutefois votre attention sur l'utilisation une nouvelle fois des variables pour paramétrer le titre du site et aller chercher les images (le logo en l'occurrence) au bon endroit.

Voici le CSS associé :

Contenu masqué n°3

Maintenant place au pied de page !

8.5.2. Le pied de page (footer.part.html)

Le pied de page sera lui aussi assez simple à mettre en œuvre. Il ne possédera qu'une petite indication de copyright sur la gauche puis une liste de liens (tous bidon pour l'exemple) sur la droite.

Voici le HTML, le CSS et un rendu. Là encore l'utilisation de la variable `SITENAME` est faite pour le copyright.

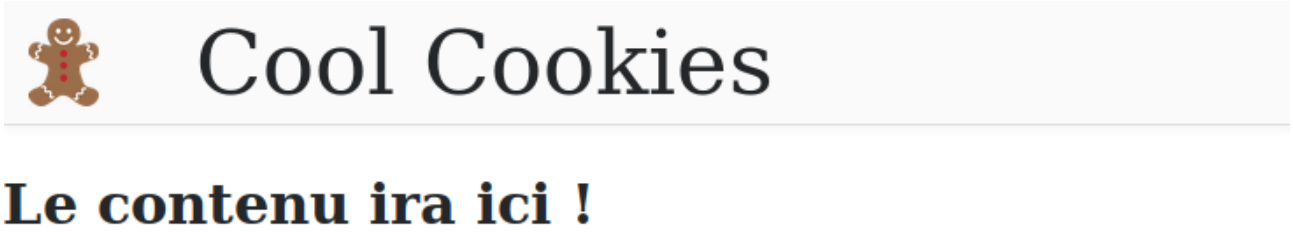
```html
<footer class="footer-bar">
    <span class="footer-copyright">© {{ SITENAME }}</span>
    <ul class="footer-list">
        <li><a href="#">À propos</a></li>
        <li><a href="#">Contact</a></li>
        <li><a href="#">Mentions Légales</a></li>
    </ul>
</footer>
```

Contenu masqué n°4

FIGURE 8.2. – Le pied de page

8.5.3. Allure globale

Voici le rendu que vous devriez avoir :



Le contenu ira ici !

FIGURE 8.3. – Rendu global - base.html

Un peu de CSS a été utilisé pour étirer correctement le conteneur principale, voici-donc tout ce que j'ai passé sous silence :

Contenu masqué n°5

8.6. La barre latérale

Dernière étape, rajouter la barre latérale (*sidebar*) qui recensera les catégories de notre blog ainsi que quelques liens « sociaux ».

Comme d'habitude, je vais tout d'abord vous copier le code puis je vous l'expliquerais ensuite :

```
<nav class="sidebar-nav">
    <ul class="categories">
        {% for cat, articles in categories|sort %}
        <li>
            <a href="/{{ cat.url }}">{{ cat }} ({{ articles|count
                }})</a>
        </li>
        {% endfor %}
    </ul>
    <hr>
    <ul class="links">
        <ul>
            {% for name, link in SOCIAL %}
            <li>
                <a href="{{ link }}">{{ name }}</a>
            </li>
            {% endfor %}
        </ul>
    </ul>
</nav>
```

Contenu masqué n°6

Vous l'avez sûrement vu, encore de nouvelles choses sont au programme : les boucles !

Les boucles sont un moyen très pratique pour construire des listes d'éléments. Ici, nous faisons deux boucles, une pour générer la liste des catégories (et le nombre d'articles présents dans chacune) et une autre qui parcourt tout les éléments présents dans la liste globale `SOCIAL` qui est dans le fichier `pelicanconf.py`.

8.6.1. La liste des catégories

La construction de cette nouvelle instruction n'est pas très compliquée vous allez voir. Voici comment elle se décompose :

```
{% for element-de-la-liste in liste %}
    [ ... Instructions executer à chaque passage dans la boucle ...
        ]
{% endfor %}
```

On voit apparaître plusieurs choses. Tout d'abord le mot-clé `for`, qui indique que l'on veut faire une boucle. Ensuite, on donne un nom symbolique pour une variable que l'on souhaite utiliser pour stocker un élément de la liste à parcourir. Enfin, le mot-clé `in` suivi du nom de la variable contenant la liste à parcourir.

Notre cas d'études est un peu particulier :

```
{% for cat, articles in categories|sort %}
```

Déjà, on voit que l'on stocke dans `cat, articles`. C'est dû à la nature de la liste étudiée `categories`. Cette dernière est en fait une liste de couple *(catégorie, liste d'articles de la catégorie)*. À chaque passage dans la boucle, à chaque élément récupéré, on récupère donc une catégorie (stockée dans `cat`) et la liste des articles de cette catégorie (stockée dans ... `articles`). Petite facétie en plus, on applique le modificateur `sort` sur la liste `categories` initial pour récupérer ses éléments en ordre alphabétiques.

Ok, on a donc maintenant à chaque tour de boucle une catégorie et sa liste d'articles, il est temps de s'en servir.

```
<li>
    <a href="/{{ cat.url }}">{{ cat }} ({{ articles|count }})</a>
</li>
```

On va donc placer l'élément `url` de la catégorie `cat` dans le lien de la balise `<a>` et on utilisera directement `cat` dans le texte du lien (pour dire vrai, c'est alors sa représentation textuelle qui sera appelée). Enfin, on mettra entre parenthèses le nombre d'articles de la catégorie en cours. Pour ce faire, on utilise la fonction `count` sur la variables `articles` (qui est une liste je vous rappelle).

Et après toutes ces aventures, on obtient une jolie liste à puces avec toutes nos catégories, des liens vers ces dernières et le nombre d'articles qu'elles contiennent !

8.6.2. La liste des liens sociaux

Cette liste est un peu plus simple que la précédente puisque ce sont "juste" des tuples mis dans une liste. Voici un exemple de ce qu'elle peut contenir :

```
SOCIAL = (('Facebook', 'https://lien-vers-profil-facebook.com/'),
          ('Twitter', 'https://lien-vers-profil-twitter.com/'),
          ('Google+', 'https://lien-vers-profil-googleplus.com/'),
          ('ZdS', 'https://lien-vers-profil-zestedesavoir.com/'))
```

Ainsi, en la parcourant on va récupérer deux éléments, un "titre" et un lien vers une page web. Pour la parcourir, on fait donc de nouveau appel à la boucle `for` sur la variable globale `SOCIAL`, en précisant deux variables pour stocker les éléments :

```
{% for name, link in SOCIAL %}
```

Ensuite, il suffit de mettre ces éléments dans des balises `<li>` et `<a>` pour construire notre joli liste :

```
{% for name, link in SOCIAL %}
<li>
    <a href="{{ link }}">{{ name }}</a>
</li>
{% endfor %}
```

Notre liste va ainsi se construire auto-magiquement pour notre plus grand plaisir. Les deux variables étant de simples chaînes de caractères, il n'y a pas à accéder à un sous-élément.

8.6.3. Rendu final de base.html

FIGURE 8.4. – Rendu final - base.html

8.7. One more thing : extends

Une petite dernière chose (promis c'est la dernière du chapitre !). Maintenant que nous avons une fondation solide, il faut pouvoir l'utiliser !

C'est ce à quoi nous servira le mot-clé `extends`, qui va informer Jinja que l'on souhaite "étendre", "compléter" un modèle. Ainsi, en tapant `{% extends "base.html" %}` **au début** de n'import lequel de vos fichier `.html`, Jinja utilisera automatiquement le fichier `base.html` et complètera les blocs personnalisables avec ce que vous souhaitez.

Par exemple, le rendu final que l'on a obtenu à la section précédente pourra être fait en tapant simplement :

```
1 {% extends "base.html" %}
2
3
4 {% block content %}
5 <h1>Le contenu ira ici !</h1>
6 {% endblock %}
```

Automatiquement Jinja saura que vous voulez travailler avec le squelette `base.html` et que vous souhaitez modifier le bloc `content`.

Eh bien, sacré morceau !

Ce chapitre fût assez dense, je le reconnais, mais il a posé les bases de plusieurs choses, comme l'utilisation de variables ainsi que celles des blocs. Tout ceci est essentiel pour la suite et il était donc important de bien les introduire.

Les chapitres suivants vont rentrer dans le vif du sujet en vous permettant de découvrir la personnalisation des différents types de pages que votre site proposera.

Contenu masqué

Contenu masqué n°3

```
.header-bar {
  padding: 5px;
  margin-bottom: 5px;
  background-color: #fafafb;
  box-shadow: 0 1px 0 rgba(12,13,14,0.1),0 1px 6px
      rgba(59,64,69,0.1);
}

a.header-nav {
  text-decoration: none;
}

a.header-nav img {
  height: 50px;
  vertical-align: middle;
  margin-left: 10px;
}

.header-title {
  font-size: 50px;
  margin-left: 50px;
  display: inline-block;
  vertical-align: middle
}
```

Retourner au texte.

Contenu masqué n°4

```css
.footer-bar {
  padding: 5px 20px;
  margin-top: 5px;
  background-color: #242729;
  color: #848d95;
  display: flex;
  justify-content: space-between;
}

.footer-bar span {
  margin-top: 4px;
}

.footer-bar ul {
  margin: 0;
  list-style-type: none;
}

.footer-bar li {
  margin: 4px 0;
}

.footer-bar a {
  color: #848d95;
  text-decoration: none;
}

.footer-bar a:hover {
  color: #bbc0c4;
}
```

Retourner au texte.

Contenu masqué n°5

```css
html, body {
  height: 100%;
}

body {
  margin: 0;
  color: #242729;
  display: flex;
```

```
  flex-direction: column;
}

.main {
  flex-grow: 1;
}
```

Retourner au texte.

Contenu masqué n°6

```
.sidebar-nav {
  background-color: #f2f2f2;
  padding: 10px 20px;
  border: solid 1px grey;
  border-radius: 5px;
}

.sidebar-nav hr {
  color: #fff;
}

.sidebar-nav ul {
  padding: 0;
  margin: 0;
  list-style-type: none;
}

.sidebar-nav li {
  margin: 4px 0;
}

.sidebar-nav a {
  text-decoration: none;
  color: #43474b;
}

.sidebar-nav a:hover {
  color: #000;
}
```

Retourner au texte.

9. Page de « listing »

Avec le chapitre précédent c'est un bon gros morceau qui est abattu ! Ce chapitre ici fera un peu office de pause Il sera plus léger et les pages que nous allons réaliser sont moins primordiales pour les visiteurs. J'espère cependant que vous le trouverez tout aussi intéressant !

Nous allons ici créer les pages de « *listings* ». Ce sont les pages qui liste les articles correspondants à une catégorie, un auteur ou un tag. Il existe plein de manières de présenter ces différentes pages (comme par exemple faire un nuage de mots pour une page qui recense tous les tags), mais nous nous contenterons de faire simple ici et toutes les pages auront donc la même présentation, des listes !

9.1. Les pages de listes

Comme dit dans l'intro, plusieurs pages de "listings" existent et on peut les classer en deux catégories :

— Les pages de "Haut niveau" qui listent les auteurs, les catégories ou encore les tags ;
— Les pages de "listes d'articles" qui vont lister tout les articles d'une page de haut niveau, par exemple tout les articles écrits par Eskimon.

9.1.1. Les pages de haut niveau

On trouve 3 pages de haut niveau, que nous avons déjà très brièvement vu dans l'arborescence des fichiers du thème. Il s'agit des fichiers suivants :

— `authors.html` : Qui liste tous les auteurs ayant au moins écrit un article sur votre site ;
— `categories.html` : Qui liste toutes les catégories contenant au moins un article ;
— `tags.html` : Qui liste tous les tags contenant au moins un article.

Vous remarquez que ces noms de fichiers sont en anglais, afin de bien illustrer qu'ils listent les éléments eux mêmes (les tag**s**) par exemple) et non pas le contenu d'un élément (les articles **d'un** tag). En effet, ces derniers sont gérés par les pages de listes d'articles.

9.1.2. Les pages de listes d'articles

Et c'est avec cette superbe transition que je vais vous présenter les pages qui vont servir à donner le détail du contenu d'une page de haut niveau. Par exemple, tous les articles écrits par Eskimon et personne d'autres.

Ces pages ont le même nom que leur pendant de haut-niveau, mais au singulier. Pratique non ? Logiquement, on trouve donc :

— `author.html` : Qui liste tous les articles d'un auteur ;
— `category.html` : Qui liste tous les articles d'une catégorie ;
— `tag.html` : Qui liste tous les articles d'un tag.

Maintenant que les présentations sont faites, voyons comment nous allons construire tout cela.

Ces listes ne sont pas immuables. En effet, certains *plugins* (comme le fameux *subcategories*) peuvent rajouter de nouveaux fichiers de listings qu'il faudra alors créer.

9.2. Les listes de « haut niveau »

Nous allons dès à présent voir le principe de réalisation de nos différentes pages. Là encore nous allons le faire en deux étapes, les pages de haut-niveau et celle listant des articles. Deux astuces vont être utiles ici, la généralisation de codes et l'utilisation de la balise `include`. Commençons par les pages de haut niveau.

9.2.1. Modèle générale

Prenons comme exemple la page listant les catégories et partons du principe que nos articles se divisent pour le moment en deux catégories : "Cookies" et "Gateaux". L'objectif de la page `categories.html` sera donc de nous lister ces deux catégories, avec en bonus le nombre d'articles que chacune possèdent.

Lorsque l'on va faire un tour sur la documentation de création de thème de Pelican , on voit que toutes les pages recoivent les variables contenant tout les auteurs (`authors`), les catégories (`categories`) ou encore les (`tags`). Il nous faut donc utiliser la bonne sur la bonne page. Ensuite, une simple boucle `for` permet de générer la liste et hop, le tour est joué !

Par exemple, voilà ce que l'on pourrait faire :

```
1  {% extends "base.html" %}
2
3
4  {% block content %}
5
6  <h1 class="list-title">Liste des catégories</h1>
7  {% for cat, articles in categories|sort %}
8  <section class="list-element">
9      <h2 class="list-element-title">
10         <a href="/{{ cat.url }}">{{ cat }}</a>
11     </h2>
12     <div class="list-element-count">
```

```
13          ({{ articles|count }} article{% if articles|length > 1
                %}s{% endif %})
14      </div>
15  </section>
16  {% endfor %}
17
18  {% endblock %}
```

C'est somme toute assez simple. Tout d'abord, on "étend" le squelette `base.html`, sans quoi on n'aurait pas la base même de la page. Ensuite, on personnalise le *block* "content". Dans ce dernier, on placera tout d'abord un titre ("Liste des catégories") puis ensuite, une boucle `for` va itérer sur la variable `categories` pour en resortir à chaque tour une catégorie (variable `cat`) et la liste d'articles contenu dans cette dernière (variables `articles`). Enfin, on construit une section dans la boucle for pour placer tout ses éléments.

Voici un exemple de rendu que l'on obtient en visitant la page http ://localhost :8000 (rappel, il faut que le serveur de développement soit en marche) :

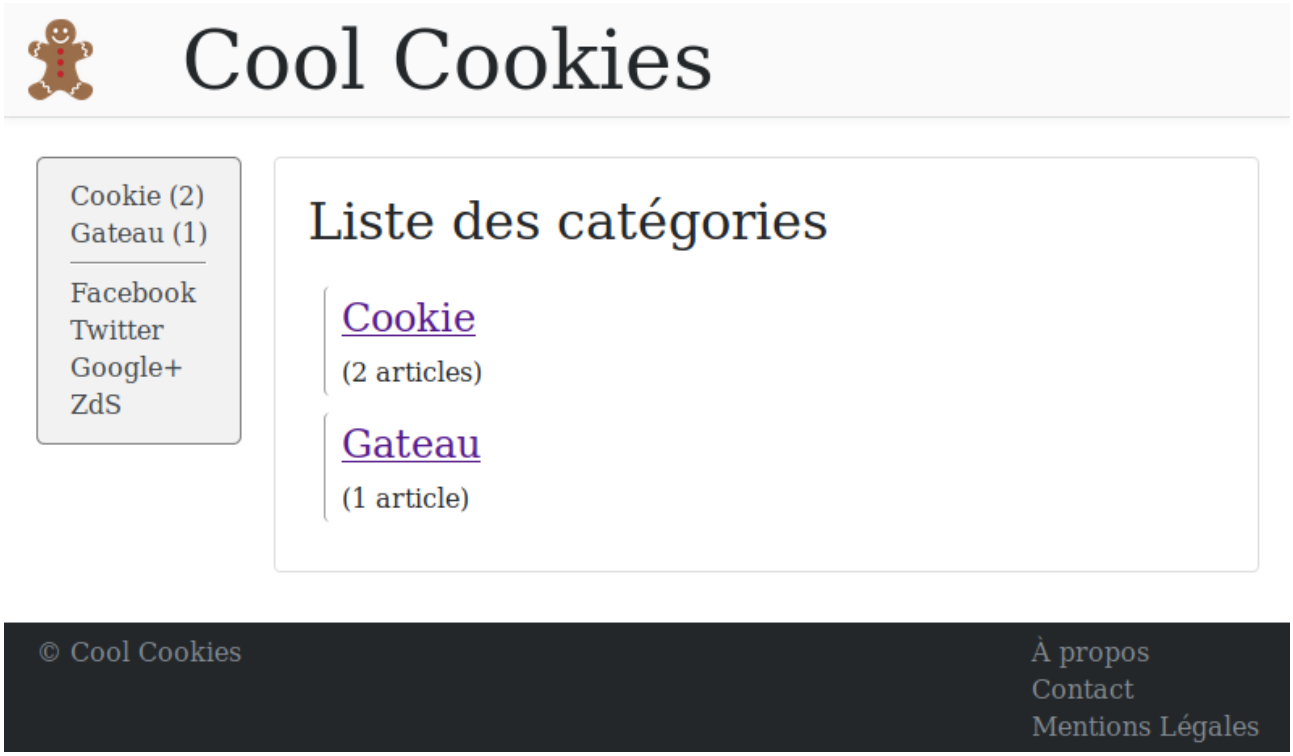


FIGURE 9.1. – Rendu de la liste des catégories

(CSS ci-dessous)

Contenu masqué n°7

?

Avez-vous remarqué l'utilisation de la structure `{% if ... %} ... {% endif %}` pour mettre au pluriel le mot *article* ?

9.2.2. Un peu d'optimisation

Bon c'est chouette, on a un truc qui tourne et donne un rendu. Maintenant il suffit de copier/coller ça pour les tags et les auteurs, modifier le titre et en avant Guingamp n'est ce pas ?!

Mauvais réflexe ! Le copier-coller n'est pas toujours la solution ! Imaginons nous faisons une modification sur le nom d'une classe par exemple, eh bien il ne faudrait surtout pas oublier de la refaire dans les autres fichiers, sinon on aurait un site qui serait propre par endroit, mais avec des vieux morceaux à d'autres !

Bon ok je chipote un peu pour 3 fichiers, mais quand même, pour le principe (et pour l'exercice) on va faire en sorte de rendre notre code le plus générique possible. Ainsi, en donnant un bon gros coup de hache dedans, on va se retrouver avec ça :

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

{% block content %}
<h1 class="list-title">Liste des catégories</h1>
{% set _object = categories %}
{% include 'parts/high-level-list.part.html' %}
{% endblock %}
```

Léger non ? Que s'est-il passé ?

Tout d'abord, on a gardé le titre du contenu, puisque ce dernier ne peut pas être deviné. Ensuite, on crée une variable avec le mot-clé `set` et la syntaxe `{% set ma_variable = ma_valeur %}`. Ici la variable s'appellera `_object` et aura pour valeur la variable globales de Pelican `categories` (l'underscore devant le nom servant par convention à symboliser une variable privée dans de nombreux langages).

Ensuite, on insère un fragment de code situé dans le dossier `parts` et se nommant `high-level-list.part.html`. C'est ce dernier qui ira construire notre liste de catégories (en fait quelque soit la liste dans `_object`) comme précédemment, seule la variable utilisée change.

```
{% for obj, articles in _object|sort %}
<section class="list-element">
    <h2 class="list-element-title">
        <a href="/{{ obj.url }}">{{ obj }}</a>
    </h2>
    <div class="list-element-count">
```

```
7           ({{ articles|count }} article{% if articles|length > 1
                %}s{% endif %})
8       </div>
9   </section>
10  {% endfor %}
```

Vous cernez la flexibilité du truc ? Maintenant pour les pages `tags.html` et `authors.html` il y a vraiment un minimum de choses à reprendre et modifier ! On gagne du temps et si jamais on veut bidouiller la liste on a un seul fichier à modifier.

Par exemple, pour la liste des auteurs, dans `authors.html` on aura :

```
1  {% extends "base.html" %}
2
3
4  {% block content %}
5  <h1 class="list-title">Liste des auteurs</h1>
6  {% set _object = authors %}
7  {% include 'parts/high-level-list.part.html' %}
8  {% endblock %}
```

On a juste eu besoin de remplacer le titre et la variable à utiliser et tout le reste s'est fait auto-magiquement !

Je vous laisse faire la page `tags.html`, ça ne devrait pas vous poser trop de soucis !

Pour visiter ces trois pages et en voir le rendu, rendez-vous aux adresses suivantes :

— `authors.html` : http ://localhost :8000/authors
— `categories.html` : http ://localhost :8000/categories
— `tags.html` : http ://localhost :8000/tags

Bien entendu cette astuce de factorisation marche bien si vous voulez avoir un rendu homogène sur ces 3 pages. En revanche, si vous souhaitez faire un style différent à chaque fois dans ce cas il n'y a évidemment pas d'astuce particulière, il faudra personnaliser le template idoine de la bonne façon selon vos rêves et désirs !

9.3. Les listes d'articles

Il ne nous reste plus qu'à faire les listes d'articles et nous aurons fait le tour de ce type de page !

Si vous avez bien compris la section précédente, alors celle-ci sera du gâteau ! À vrai dire vous pourriez presque déjà la faire sans moi, il suffit juste de trouver quelles variables utiliser pour générer la liste.

Comme vu précédemment, les listes d'articles sont les détails des listes de haut niveau. Ainsi, on va là encore trouver trois pages à créer / éditer : `author.html`, `category.html` et `tag.html`. Comme l'explique la documentation , chacune de ses pages proposes une variable correspondante au contenu de haut niveau en cours d'affichage, donc respectivement `author`, `category` et `tag`. On y trouvera aussi une variable `articles` qui contiendra les articles de la page étudiée.

Ainsi, avec toutes ses informations on peut faire un premier jet de réalisation similaire à la précédente. Cette fois-ci nous afficherons un peu plus d'informations, en effet les articles sont riches en métadonnées, profitons-en pour aider le lecteur à choisir un contenu qui lui plaît ! Voici pour une page présentant une catégorie. Je pense qu'avec tout ce que nous avons vu jusqu'à présent, ce code ne devrait pas vous poser trop de souci .

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


{% block content %}
<h1 class="list-title">Article de la catégorie "{{ category
    }}"</h1>

{% for article in articles %}
<section class="list-element">
    <h2 class="list-element-title">
        <a href="/{{ article.url }}">{{ article.title }}</a>
    </h2>
    <div class="list-element-content">
        <p>
            <em>Écrit par <a href="/{{ article.author.url }}">{{
                article.author }}</a> le {{ article.locale_date
                }}.</em>
        </p>
        <p>
            {{ article.summary }}
        </p>
    </div>
</section>
{% endfor %}

{% endblock %}
```

i

Si vous souhaitez utiliser une liste d'articles triés par date et non pas par titre, vous pouvez utiliser la variable `dates` au lieu de `articles`.

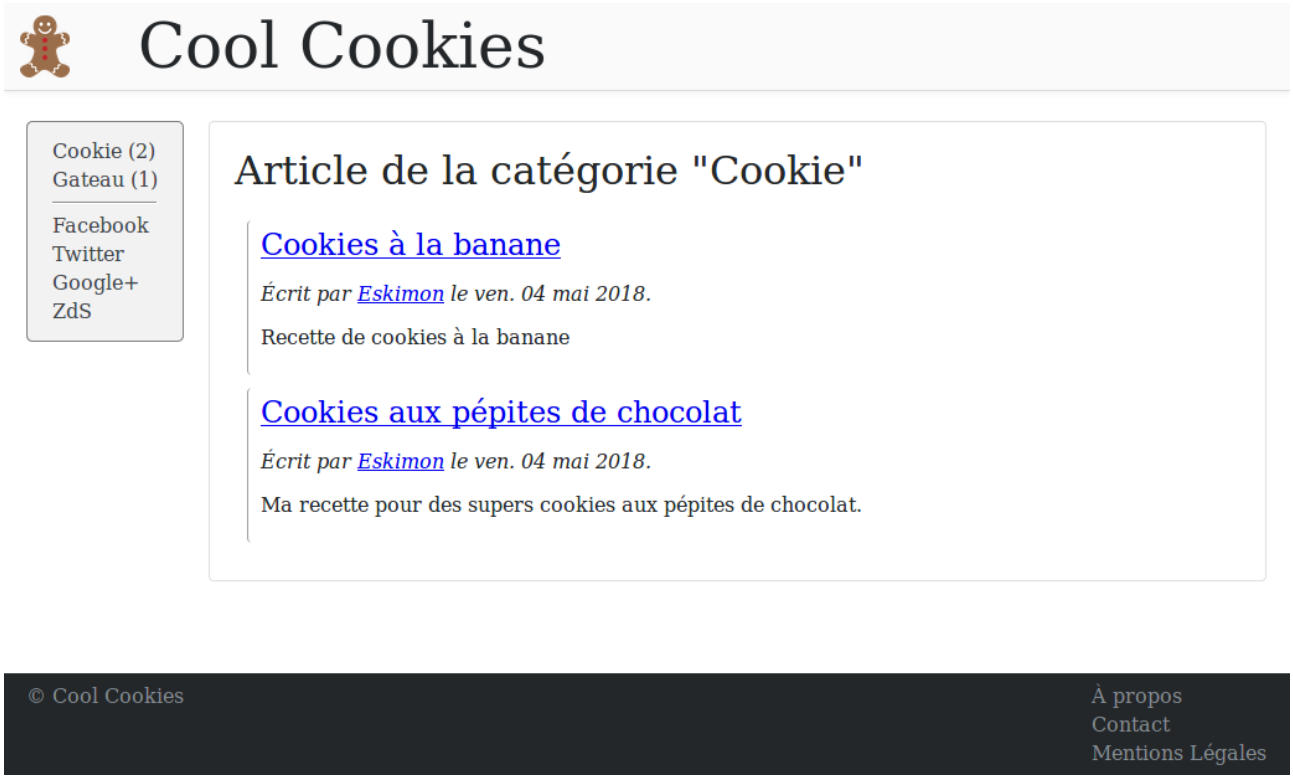


FIGURE 9.2. – Rendu d'une liste d'articles

Avez-vous remarqué que les variables pour parcourir tout les contenus sont `articles` (au pluriel, contenant la liste des articles de cette catégorie) et `article` (au singulier, l'article actuellement étudié à chaque tour dans la boucle `for`). C'est une convention d'écriture assez habituelle, alors n'ayez pas peur de l'utiliser : `for singulier in pluriel`

9.3.1. Factorisation

Comme pour les listes de haut-niveau, là encore nous pouvons déplacer une bonne partie du code dans un fichier `part` et ainsi réutiliser sans peine le code pour toutes nos listes d'articles. Voici ma version allégée, le code ne changeant pas énormément et ne voulant pas répéter ce que vous connaissez déjà je vous laisse l'analyser vous-même. Aussi, il ne faut pas oublier de reproduire ce schéma pour les pages `tag.html` et `author.html`.

```
1 {% extends "base.html" %}
2
3
4 {% block content %}
5
6 <h1 class="list-title">Article de la catégorie "{{ category
      }}"</h1>
7
```

```
{% include 'parts/articles-list.part.html' %}

{% endblock %}
```

```
{% for article in articles %}
<section class="list-element">
    <h2 class="list-element-title">
        <a href="/{{ article.url }}">{{ article.title }}</a>
    </h2>
    <div class="list-element-content">
        <p>
            <em>Écrit par <a href="/{{ article.author.url }}">{{
                article.author }}</a> le {{ article.locale_date
                }}.</em>
        </p>
        <p>
            {{ article.summary }}
        </p>
    </div>
</section>
{% endfor %}
```

9.4. Annexe : La pagination

Qui dit liste, dit souvent pages. En effet, notre exemple ici ne paie pas de mine, mais on est pas à l'abri qu'un jour on ait beaucoup d'articles. On risquerait alors de se retrouver avec une liste longue comme le bras, avec une page mettant du temps à charger et pénible à naviguer.

Une solution : La pagination.

J'ai brièvement évoqué cet aspect dans le début du tutoriel, il est temps de le mettre en pratique.

Tout d'abord, il faut repérer le paramètre `DEFAULT_PAGINATION` dans le fichier `pelicanconf.py`. La valeur de ce dernier va servir à régler combien d'articles par page seront affichés. Une valeur de 5 par exemple permettera d'afficher 5 articles sur chaque page. Donc si on a 20 articles dans une catégorie, l'affichage du contenu de cette dernière se fera sur 4 pages distinctes.

Puisque nous parlons de l'exemple des catégories, retournons voir la documentation de ces dernières. Tout d'abord, on apprend que si plusieurs pages sont nécessaires pour afficher la liste des articles, alors elles seront formatées selon la convention `category/{category_name}{number}.html`. La page 1 contiendra les articles 1 à 5, la n°2 les 6 à 10 etc.

Ensuite, le tableau nous affiche le nom et l'utilité des variables supplémentaires spécifiques à cette page. Cinq attirent notre attention :

— `articles_paginator` : Un objet de type *paginator*, utile pour gérer la pagination ;
— `articles_page` : Un objet de type *Page* représentant la page courante. Je reviens tout de suite là dessus ! ;
— `articles_previous_page` : Un objet de type *Page* représentant la page précédente ;
— `articles_next_page` : Un objet de type *Page* représentant la page suivante ;
— `page_name` : Nous servira à contruire les liens pour accéder aux différentes pages. Ce dernier représente le lien de "base" de la page (par exemple `category/cookie` pour n'importe quel numéro de page de la catégorie cookie).

9.4.1. L'objet `Page`

Ce type de variable transporte plein d'informations intéressantes, notamment pour créer notre joli pagination. Je ne vais pas rentrer dans le détail de tout ce qu'il propose, cependant sachez que son *code-source-pas-si-compliqué* est disponible ici : https ://github.com/getpelican/pelican/blob/master/pelican/paginator.py .

Le premier élément nous qui nous intéresse sera la variable `object_list` de cet objet. Cette dernière nous met à disposition la liste des articles contenus dans uniquement cette page (donc au plus 5 articles si la pagination est paramétré à 5 articles par page). Via les variables mentionnées précédemment, on utilisera `articles_page.object_list` pour les articles de la page courante.

Un autre élément est l'accès aux fonctions `has_previous` et `has_next`. Ces deux fonctions renvoient un booléen si il y a effectivement une page précédente ou suivante par rapport à la courante. On pourra alors utiliser `previous_page_number` et `next_page_number` pour avoir le numéro de la page précédente et celui de la suivante.

Enfin, on peut aussi obtenir l'url de la page via la variable `url` (`category/cookie2.html` pour la deuxième page de la catégorie *cookie* par exemple).

9.4.2. L'objet `Paginator`

Cet objet est lui aussi assez simple à utiliser. Son code se trouve au même endroit que celui de `Page`. Seules certaines variables contenues dans `Paginator` vont vraiment nous intéresser :

— `count` : Le nombre total d'articles dans cette pagination (par exemple 20 pour 20 articles, avec 5 articles par page) ;
— `num_pages` : Le nombre total de pages composant la pagination (par exemple '4' pour 20 articles, avec 5 articles par page) ;
— `page_range` : Un générateur allant de 1 au nombre de pages nécessaire pour tout parcourir (par exemple [1..4] pour afficher 20 articles, avec 5 articles par page).

9.4.3. Mettre tout cela en œuvre

Maintenant que nous savons tout cela, il va falloir modifier notre code pour afficher la liste d'articles de la meilleure manière qu'il soit. Et bonne nouvelle, il ne faudra le faire qu'à un seul

endroit puisque nous avons factorisé tout le code d'affichages des listes d'articles dans le fichier `articles-list.part.html` !

Pour commencer, on ne va afficher que les articles concernant la page courante. Pour cela on va remplacer le for :

```
{% for article in articles %}
```

devient

```
{% for article in articles_page.object_list %}
```

Donc au lieu d'afficher 20 articles on en affiche plus que 5.

Maintenant, il va falloir créer l'affichage qui nous permettra de sélectionner la page que nous souhaitons afficher.

Pour cela, on va utiliser le générateur `page_range` de `articles_paginator` dans un for pour créer une liste de toutes les pages (en dehors de la boucle `for` de la liste d'articles) :

```
{% if articles_paginator.num_pages > 1 %}
<div class="paginator">
    {% for cpt in articles_paginator.page_range %}
    <a href="/{{ page_name }}{{ cpt if cpt > 1 else '' }}.html">{{
        cpt }}</a>
    {% endfor %}
</div>
{% endif %}
```

Comme vous pouvez le remarquer, j'ai ajouté un `if` sur le nombre de pages. En effet, inutile de construire le sélecteur de pages s'il y en a qu'une seule !

On trouve aussi une condition pour construire le lien des pages. En effet, la toute première page ne se nomme pas `categorie/cookie1.html` mais juste `category/cookie.html` (sans le 1 final).

Avec un peu de CSS, voici ce que l'on peut déjà obtenir :

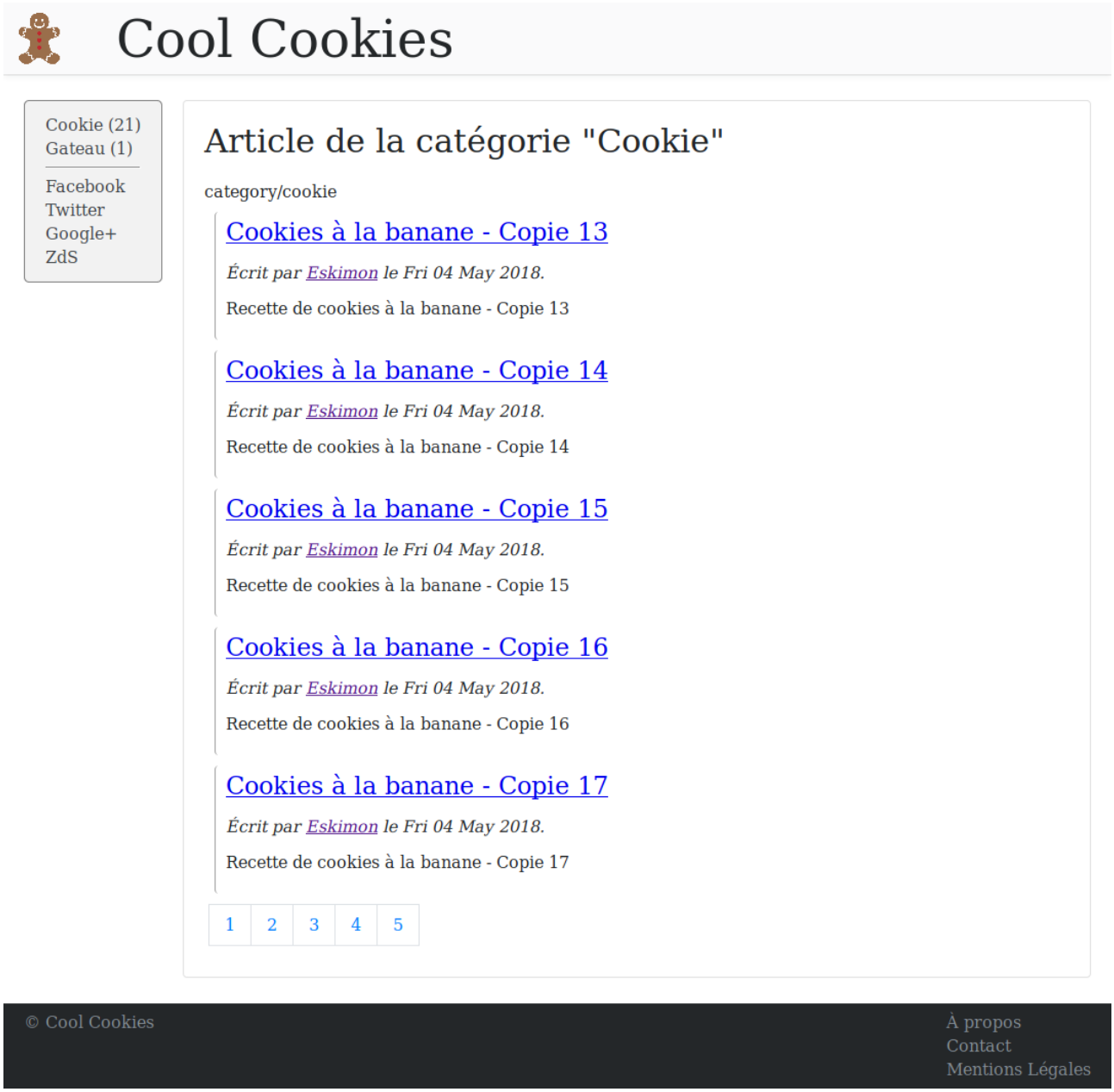


FIGURE 9.3. – Paginateur simple

Contenu masqué n°8

C'est un bon début, mais on peut faire un peu mieux...

9.4.4. Améliorer le paginateur

En utilisant à bon escient les différentes variables, on peut obtenir un résultat fort sympathique. Voici par exemple ce qui est possible :

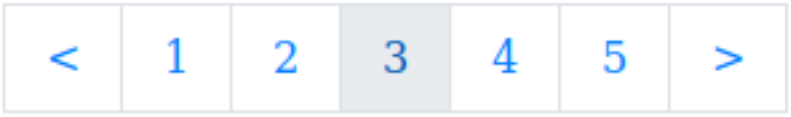


FIGURE 9.4. – Zoom sur le paginateur

On voit ainsi apparaitre des chevrons à droite et gauche permettant de naviguer en avant/arrière d'une page à la fois, et aussi un indicateur visuel permettant de savoir sur quelle page nous nous trouvons. Les chevrons ainsi que la page courante sont rendus non cliquables dans les cas où c'est utile (si nous sommes sur la première ou dernière page par exemple).

Pour obtenir ce résultat, voici le code utilisé :

```
{% if articles_paginator.num_pages > 1 %}
<div class="paginator">
    {% if articles_previous_page %}
    {% set num = articles_page.previous_page_number() %}
    <a
        href="/{{ page_name }}{{ num if num > 1 else '' }}.html">&lt;</a>
    {% else %}
    <span>&lt;</span>
    {% endif %}

    {% for cpt in articles_paginator.page_range %}
        {% if cpt == articles_page.number %}
        <span class="active">{{ cpt }}</span>
        {% else %}
        <a
            href="{{ SITEURL }}/{{ page_name }}{{ cpt if cpt > 1 else '' }}.htm
            cpt }}</a>
        {% endif %}
    {% endfor %}

    {% if articles_next_page %}
    <a
        href="/{{ page_name }}{{ articles_page.next_page_number() }}.html">&gt;
    {% else %}
    <span>&gt;</span>
    {% endif %}
</div>
{% endif %}
```

Pour construire ce beau paginateur, on fait appel à plusieurs conditions `if`. La première et dernière condition servent toutes deux à afficher les chevrons. Ils sont soit affichés dans un span (si nous sommes à la première/dernière page), soit dans un lien pour se rendre à la page précédente/suivante lorsqu'elle existe (il faut là encore ruser pour créer l'adresse de la toute première page).

Au milieu, on retrouve notre boucle `for`. Son contenu a cependant été un peu amélioré pour pouvoir afficher un span au lieu d'un lien lorsque il faut afficher le numéro de la page courante.

Et voici le CSS mise à jour :

Contenu masqué n°9

Je reconnais que ce code est un peu plus compliqué, mais finalement, avec un peu de réflexion et pas mal d'essais on finit par obtenir des trucs sympas non ?

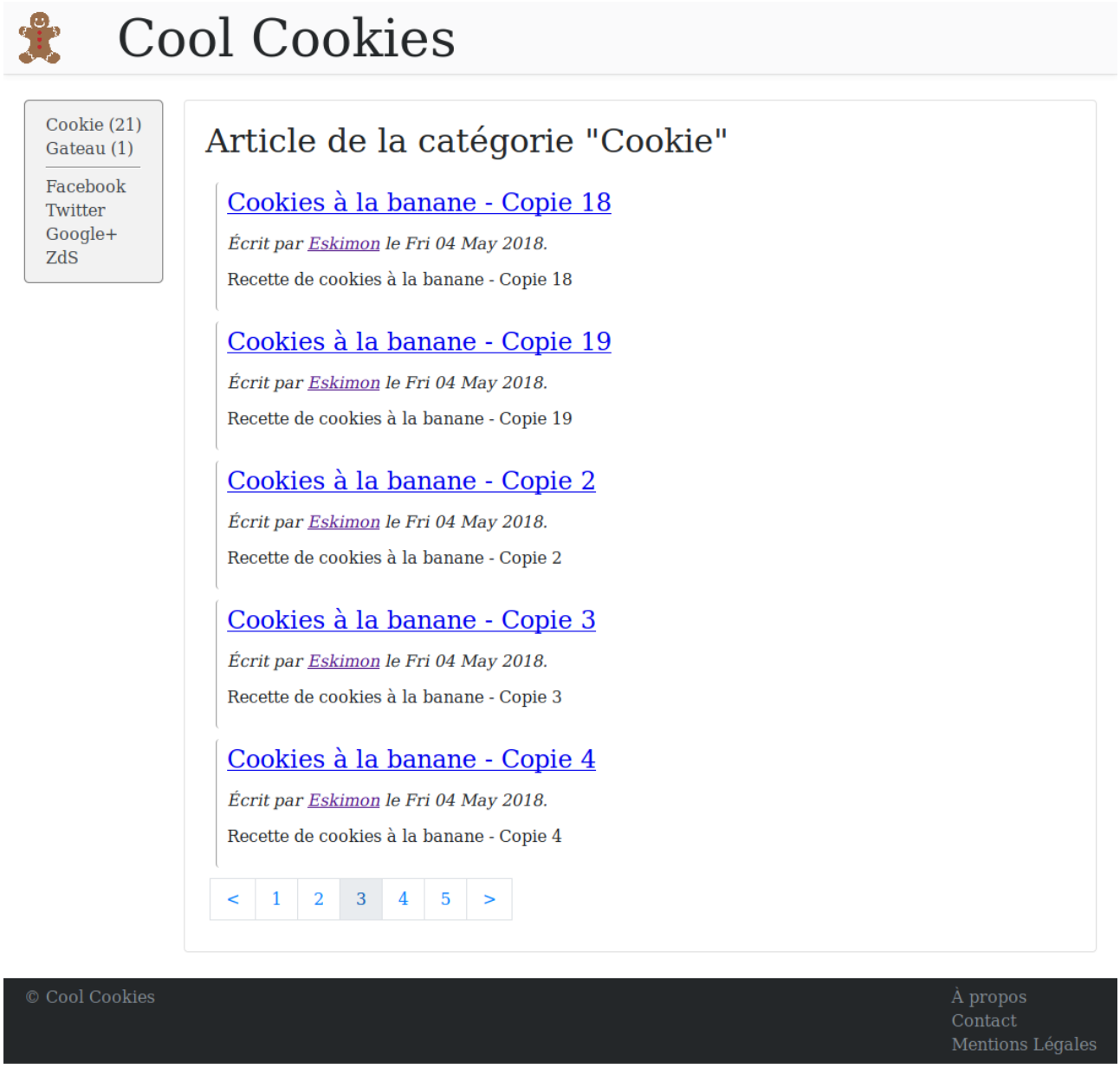


FIGURE 9.5. – Paginateur amélioré

Et voilà encore un beau morceau d'abattu, avec quelques nouveautés comme les boucle `for` mais surtout l'occasion de (re)travailler tout ce qui avait pu être vu pendant le chapitre précédent.

Maintenant nous pouvons passer au cœur du site en créant le design d'un article lui-même.

Contenu masqué

Contenu masqué n°7

```
.content {
  flex-grow: 1;
  margin-left: 20px;
  border: 1px solid rgba(0,0,0,.125);
  border-radius: .25rem;
  padding: 20px;
}

h1, h2, h3, h4, h5 {
  font-weight: 500;
}

.list-title {
  margin-top: 0;
}

.list-element {
  padding: 5px;
  border-left: 1px solid rgba(0,0,0,.375);
  border-radius: .25rem;
  margin: 10px;
  padding-left: 10px;
}

.list-element-title {
  margin-top: 0;
  margin-bottom: 10px;
}
```

Retourner au texte.

Contenu masqué n°8

```
.paginator {
  margin: 10px;
}

.paginator a {
  display: inline-block;
  padding: 10px 15px;
  margin-left: -6px;
  color: #007bff;
  background-color: #fff;
  border: 1px solid #dee2e6;
  text-decoration: none;
}

.paginator a:hover {
  color: #0056b3;
  text-decoration: none;
  background-color: #e9ecef;
  border-color: #dee2e6;
}
```

Retourner au texte.

Contenu masqué n°9

```
.paginator {
  margin: 10px;
}

.paginator a, .paginator span {
  display: inline-block;
  padding: 10px 15px;
  margin-left: -6px;
  color: #007bff;
  background-color: #fff;
  border: 1px solid #dee2e6;
  text-decoration: none;
}

.paginator a:hover {
  color: #0056b3;
  text-decoration: none;
  background-color: #e9ecef;
```

```
19    border-color: #dee2e6;
20  }
21
22  .paginator span {
23    color: rgba(0,0,0,.3);
24  }
25
26  .paginator span.active {
27    color: #0056b3;
28    background-color: #e9ecef;
29  }
```

Retourner au texte.

10. Rendu d'un article

Nous y voila, une des dernières pages à créer, celle qui représentera un article !

Avec tout ce que nous avons vu, vous allez voir que c'est vraiment un jeu d'enfant !

Tout le code que nous allons écrire ici sera à mettre dans le fichier `article.html`.

Voici ce vers quoi nous allons tendre :

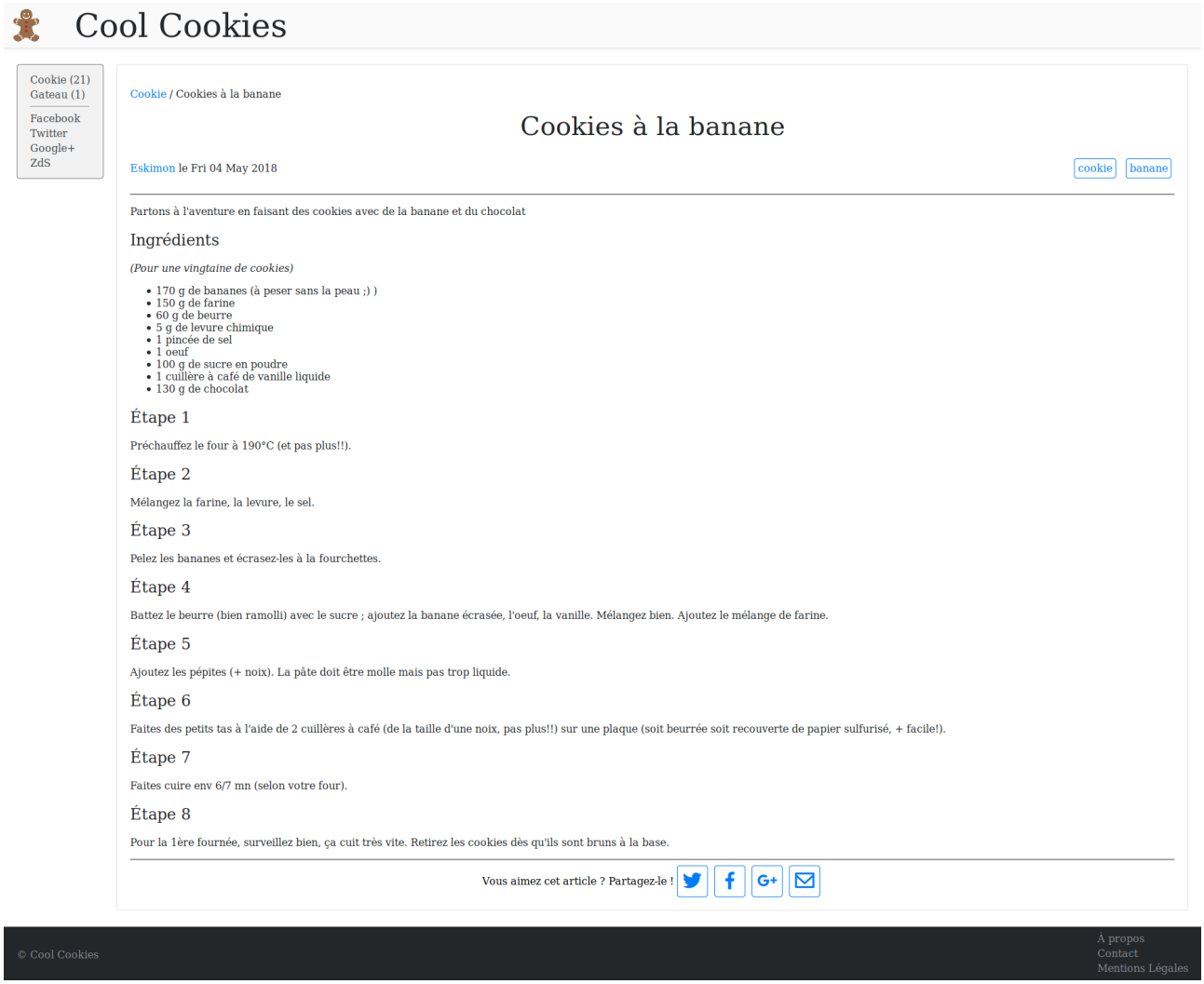


FIGURE 10.1. – Rendu d'un article

10.1. Principe

10.1.1. Structure

Afin de réaliser ce morceau de la page, nous allons diviser le travail en trois parties. Tout d'abord l'entête de l'article, qui sera dans une balise `<header>`. Ensuite viendra le corps de l'article, puis enfin un pied de page dans une balise `<footer>`. Le tout sera encapsulé dans une balise `<section>` et viendra personnaliser le bloc `content`. Le tout est basé sur le squelette habituel, `base.html`.

Sur le principe on aura donc quelque chose comme ceci :

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


{% block content %}
<section>
    <header>
        <!-- L'entête de l'article -->
    </header>
    <hr>
    <div>
        <!-- Le contenu de l'article lui-même -->
    </div>
    <hr>
    <footer>
        <!-- Le pied de page de l'article -->
    </footer>
</section>
{% endblock %}
```

Bien entendu, ce n'est qu'une proposition parmi tant d'autres !

10.1.2. Ajout de données dans le `<head>` de la page

Nos articles ayant plein d'informations à faire passer, nous allons rajouter des choses dans la section `<head>` de la page elle-même, comme par exemple une balise `<meta>` avec le résumé de l'article ou encore avec les tags qui définissent ce dernier. Pour cela, on va simplement personnaliser le bloc `extra_head` qui est justement prévu pour ça.

```
{% block extra_head %}
    <link
        href='https://cdn.jsdelivr.net/npm/boxicons@1.5.1/css/boxicons.min.css
        rel='stylesheet'>

```

```
4       {% if article.description %}
5       <meta name="description" content="{{ article.summary }}" />
6       {% endif %}
7
8       {% for tag in article.tags %}
9       <meta name="tags" content="{{ tag }}" />
10      {% endfor %}
11  {% endblock %}
```

Comme vous pouvez le voir, j'ai surtout ajouté trois choses.

Tout d'abord, j'importe le CSS de boxicons , qui me permettra d'afficher des icônes assez simplement.

Ensuite, je crée une balise `<meta> description`, qui contiendra la variable `article.summary`. Cela permet au moteur de recherche de savoir quoi afficher dans leur page de résultat en dessous du titre de votre article (le texte sous l'url sur cette image issue du moteur de recherche qwant).

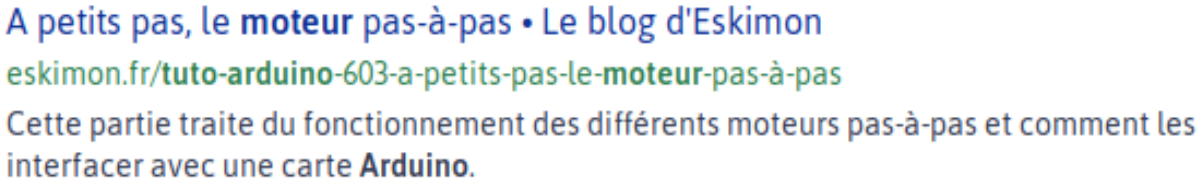


FIGURE 10.2. – Description dans un moteur de recherche

Enfin, j'ajoute autant de balises `<meta> tag` que nécessaire grâce à une boucle `for` sur la variable `article.tags`. Ceci est notamment fait pour optimiser le SEO de notre article.

Une fois tout cela fait, attaquons l'article en lui-même !

10.2. L'en-tête de notre article

Pour commencer, nous allons composer l'en-tête de notre article qui servira à afficher un fil d'Ariane (pour voir (et accéder) facilement la catégorie de l'article), puis viendra le titre de l'article et quelques métadonnées qui le compose.

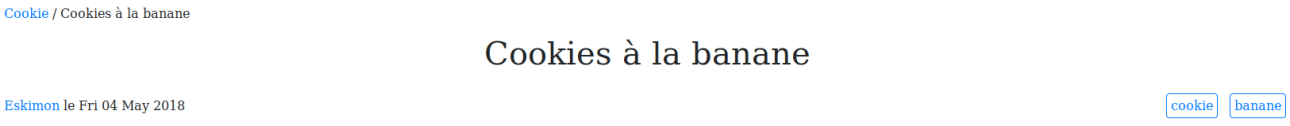


FIGURE 10.3. – En-tête article

C'est relativement trivial à faire, en voici le code que je vous explique ensuite :

```
<header>
    <p class="breadcrumb">
        <a href="/{{ article.category.url }}">{{
            article.category}}</a>
        /
        <span>{{ article.title}}</span>
    </p>
    <h1>{{ article.title}}</h1>
    <div class="metadata">
        <p>
            {% for author in article.authors %}
            <a href="/{{ author.url }}" rel="author">{{ author
                }}</a>
            {% endfor %}
            <time datetime="{{ article.date.isoformat() }}"
                pubdate>
                le {{ article.locale_date }}
            </time>
        </p>
        <p>
            {% for tag in article.tags %}
            <a class="link-item" href="/{{ tag.url }}">{{ tag
                }}</a></li>
            {% endfor %}
        </p>
    </div>
</header>
```

Comme vous pouvez le constater, il n'y a rien de très compliqué ici. On fait appel aux différentes variables de `article` pour composer les différentes éléments. Ensuite, via des boucles on peut afficher des listes d'éléments comme tout les auteurs ou tout les tags.

Voici le CSS utilisé (qui sera aussi repris pour d'autres morceaux de cette page) :

Contenu masqué n°10

10.3. Le corps et le pied de l'article

Ces deux morceaux aussi seront assez simple, voyons cela...

10.3.1. Le corps de l'article

Voici la partie qui représente tout le contenu de votre article et pourtant, ce sera la plus courte de ce tuto !

En effet, tout le corps de l'article se trouve dans la variable `article.content`. Il nous faut donc juste l'afficher et hop, c'est réglé !

```
<div class="article-content">
    {{ article.content }}
</div>
```

Bien entendu, tout cela est personnalisable à grand coup de CSS !

10.3.2. Le pied de page

Il ne reste plus qu'à faire un morceau contenant le pied de page de notre article. En l'occurrence, j'ai choisi d'y afficher une invitation au partage, en mettant des liens vers les réseaux sociaux les plus populaires. Ces liens permettent de pré-remplir un message à poster sur les réseaux.

FIGURE 10.4. – Pied de page de l'article

```
<footer>
    <span>Vous aimez cet article ? Partagez-le !</span>
    <a class="link-item"
        href="https://twitter.com/share?url={{ SITEURL }}/{{ article.url }}&amp
        rel="nofollow" title="Partager cet article sur Twitter">
        <i class="bx bxl-twitter bx-md"></i>
    </a>
    <a class="link-item"
        href="https://www.facebook.com/sharer.php?u={{ SITEURL }}/{{ article.u
        rel="nofollow" title="Partager cet article sur Facebook">
        <i class="bx bxl-facebook bx-md"></i>
    </a>
    <a class="link-item"
        href="https://plus.google.com/share?url={{ SITEURL }}/{{ article.url }}
        rel="nofollow" title="Partager cet article sur Google +">
        <i class="bx bxl-google-plus bx-md"></i>
    </a>
    <a class="link-item"
        href="mailto:?subject={{ article.title }}&amp;body={{ SITEURL }}/{{ ar
        title="Partager cet article par email">
        <i class="bx bx-envelope bx-md"></i>
    </a>
</footer>
```

C'est ici que nous ferons appel aux icônes "boxicon" dont on a intégré le CSS dans le `<head>` du site.

Et un petit bout de CSS pour la forme !

Contenu masqué n°11

10.3.3. Intégralité de la page

En résumé, voici le code de tout mon fichier `article.html` :

Contenu masqué n°12

10.4. Annexe : Ajouter une section de commentaires avec Disqus

Vous vous souvenez, tout au début de ce tuto je vous expliquais qu'un site statique était gravé dans le marbre etc MAIS que en fait ce n'était pas complètement exact ? Et bien voilà un excellent exemple de comment rajouter de l'interactivité à votre site : En rajoutant une section de commentaires !

Nous n'allons bien entendu pas gérer les commentaires nous-même, sinon on perdrait tout l'intérêt d'utiliser un site statique (avec 0 base de données etc). Nous allons plutôt déléguer cela à un service externe, en l'occurrence Disqus . Ce service est très répandu, vous l'avez sûrement déjà croisé lors de vos voyages sur internet.

Ainsi, en déléguant la gestion des commentaires à ce service, il nous suffira juste de rajouter un peu de javascript sur notre site pour intégrer ces derniers. Le chargement se fera alors en tâche de fond lors du chargement de la page, de manière transparente pour l'utilisateur.

Info pour les libristes qui me lisent : Disqus est un logiciel propriétaire géré par une entreprise. Je n'ai pas connaissance d'équivalent libre.

10.4.1. Créer un compte Disqus

Pour commencer il va falloir un compte sur le site Disqus. Nous pourrons ensuite y créer un nouveau site à gérer.

Pour créer un compte, aller sur la page de *signup* et remplissez-y le formulaire de création de compte. Si vous voulez un minimum d'utilisation de vos données, laissez la dernière case décochée.

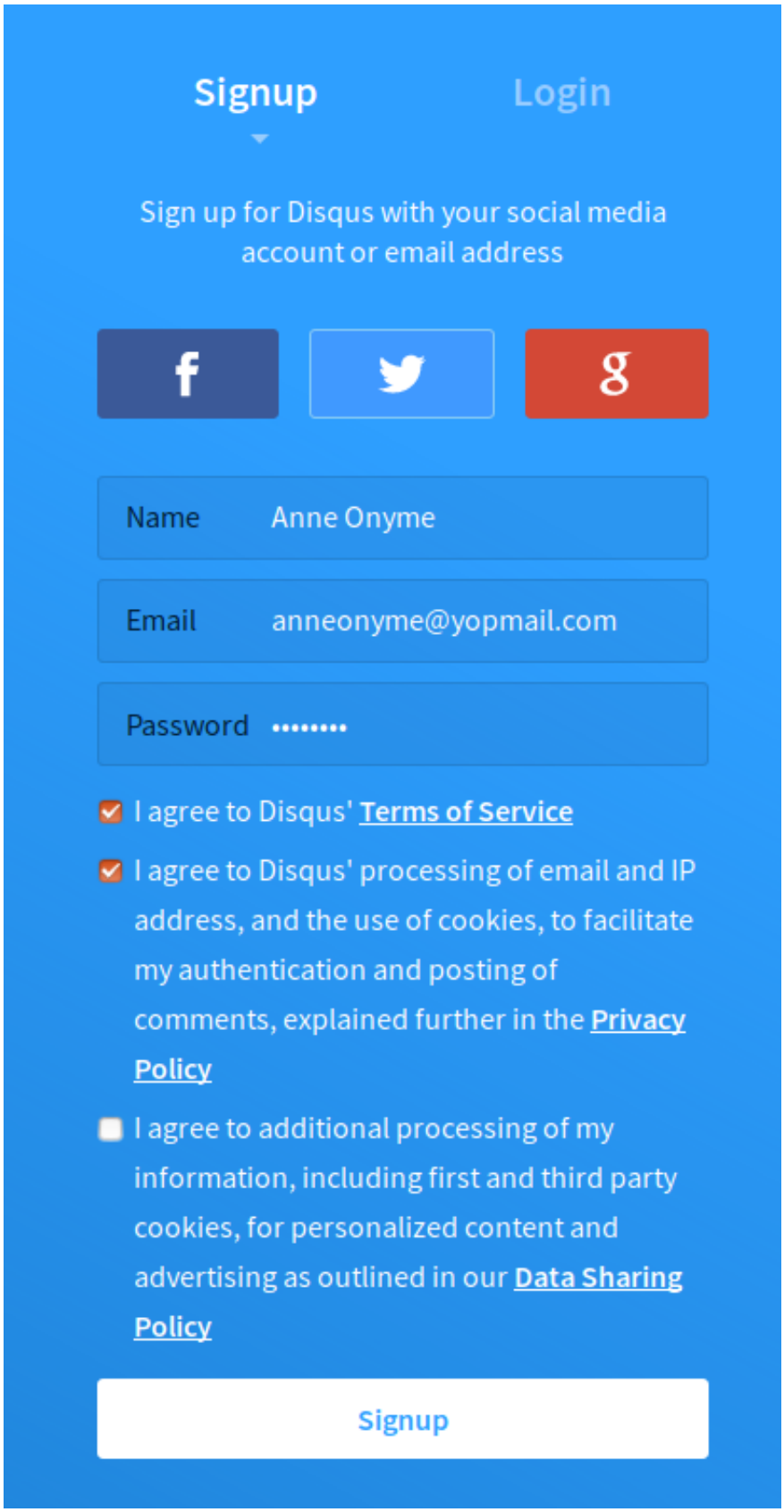


FIGURE 10.5. – Formulaire de création de compte Disqus

Une page vous demandant ce que vous souhaitez faire apparaît, sélectionnez alors "I want to install Disqus on my site". On vous demandera alors sur une nouvelle page le nom du site à

créer, saisissez-y vos informations.

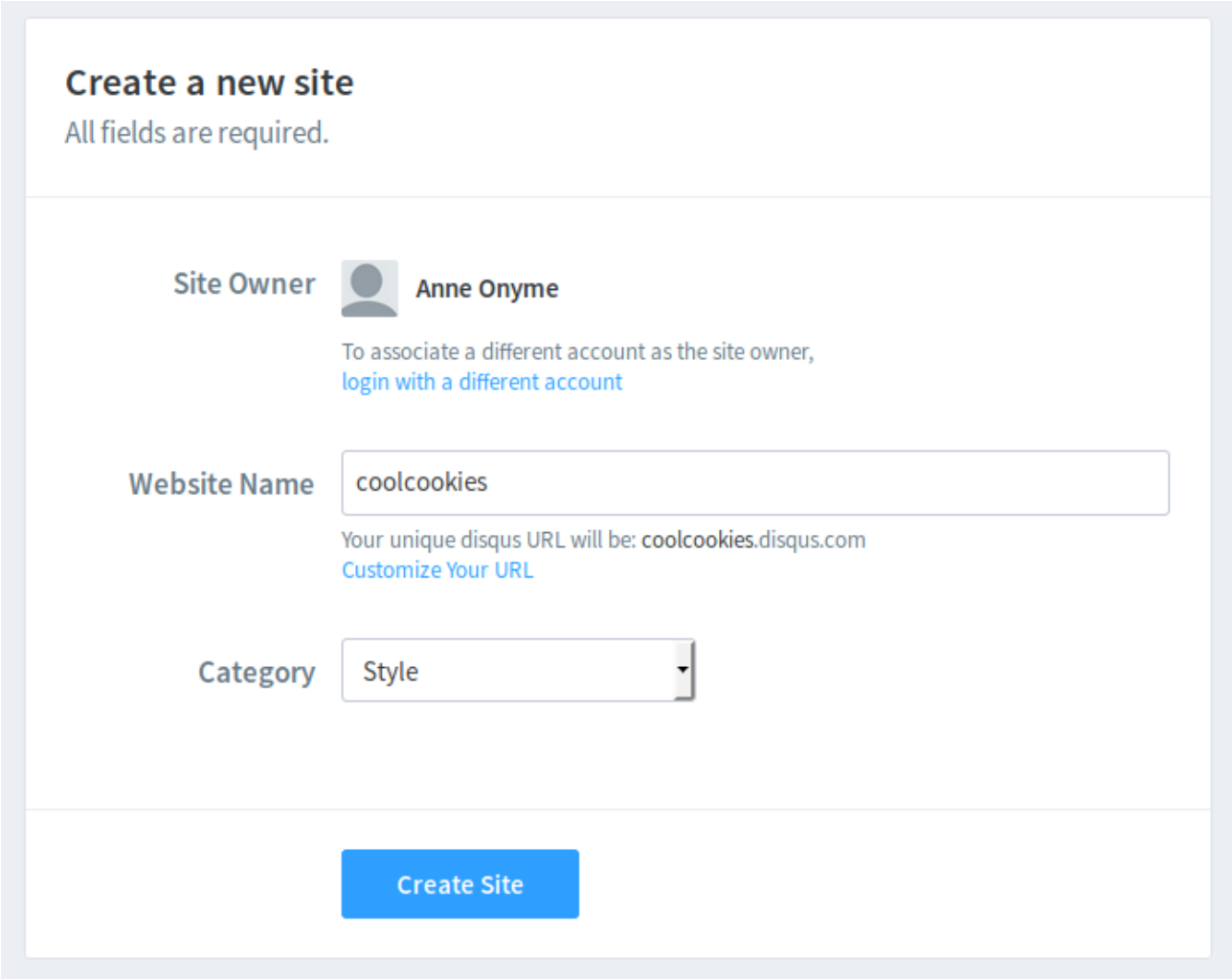


FIGURE 10.6. – Ajouter un site dans Disqus

Il faut maintenant sélectionner un tarif, nous prendrons l'option offensive gratuite, "Free". Puis sur la page suivante, sélectionnez la toute dernière option "I don't see my platform listed, install manually with Universal Code".

Le site vous montre alors un laïus sur comment intégrer Disqus à vos page web, voici la même chose mais en français !

10.4.2. Intégrer Disqus aux pages

Pour que la zone de commentaires puisse se charger, il va falloir rajouter un morceau de javascript. Nous allons mettre ce dernier juste après le `<footer>` de notre article, afin que les commentaires s'affichent à cet endroit.

Le morceau à rajouter aura cet allure :

```
<!-- disqus start -->
<div id="disqus_thread"></div>
<script>
    var disqus_config = function () {
        this.page.url = '{{ SITEURL }}/{{ article.url }}';
        this.page.identifier = '{{ article.slug }}';
        this.page.title = '{{ article.title }}';
    };
    (function () { // DON'T EDIT BELOW THIS LINE
        var d = document, s = d.createElement('script');
        s.src =
            'https://<le-nom-de-votre-site-dans-disqus>.disqus.com/embed.js';
        s.setAttribute('data-timestamp', +new Date());
        (d.head || d.body).appendChild(s);
    })();
</script>
<noscript>Please enable JavaScript to view the
    <a href="https://disqus.com/?ref_noscript">comments powered by
        Disqus.</a>
</noscript>
<!-- disqus end -->
```

Faites bien attention à bien remplacer le nom de votre site à la ligne 11 du code précédent. C'est grâce à cela que le javascript fera le lien avec votre site !

Et voilà, c'est en fait tout ce qu'il y avait à faire ! Maintenant le service de commentaires se chargera automagiquement sous chacun des articles !

Si vous retournez sur la page Disqus que nous avons laissée, vous verrez que le site vous propose d'aller configurer des options supplémentaires, n'hésitez pas à aller le faire !

10.4.3. Afficher un compteur de commentaires

Afin de voir en un clin d'œil combien de commentaires ont été posté sous un article, nous allons afficher un compteur de commentaires directement dans le `<header>` de l'article.

FIGURE 10.7. – En-tête de l'article avec nombre de commentaires

Pour ce faire, on commence par rajouter un lien qui servira à accueillir le texte contenant le nombre de commentaires :

```
(<a href="{{ SITEURL }}/{{ article.url }}#disqus_thread"
    data-disqus-identifier="{{ article.slug }}"></a>)
```

(Dans l'exemple ci-dessus j'ai mis ce lien juste après la balise `<time>` du header de l'article)

Peu importe le texte du lien, Disqus le modifiera pour afficher la bonne information.

Ensuite, il faut de nouveau ajouter un bout de javascript qui fera une requête chez Disqus pour obtenir le comptage :

```
{% block extra_script %}
<script id="dsq-count-scr"
    src="//<le-nom-de-votre-site-dans-disqus>.disqus.com/count.js"
    async></script>
{% endblock %}
```

De nouveau il faut bien penser à changer le nom du site disqus à charger dans le lien.

Et avez-vous remarqué ? J'ai rajouté ce morceau de javascript dans le bloc `extra_head`, qui est juste avant la balise fermante `<body>`. Ainsi ce javascript ne se chargera qu'en dernier, sans bloquer le reste de la page pour les connexions les plus lentes.

Enfin, retournons chez Disqus pour modifier le texte à afficher. En effet, Disqus est un service anglais avant tout, et donc les textes sont en anglais par défaut. Donc le texte « 0 commentaire » afficherait par défaut « *0 Comment* ». Pas top pour les sites francophones. Pour modifier ce paramètre, allez sur la page d'administration « *community* » de votre site (https ://<le-nom-de-votre-site-dans-disqus>.disqus.com/admin/settings/community/). Vous verrez alors la première section qui permet de modifier les textes et les pluriels. De mon côté, je les ai modifié de la manière suivante :

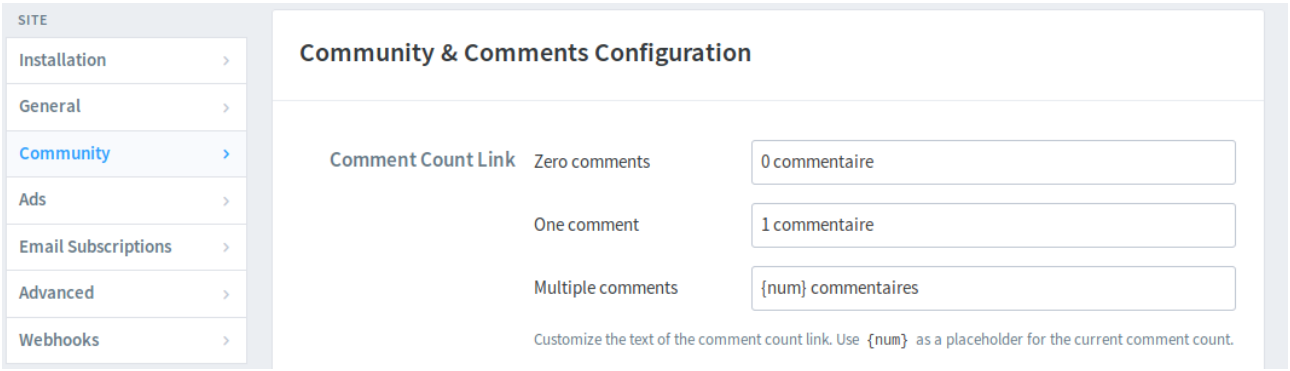


FIGURE 10.8. – Texte des commentaires

Une fois cela fait, vous avez tous les paramètres standards pour avoir des commentaires aux petits oignons, et un site déjà plein de supers fonctionnalités !

C'est fini, notre site est maintenant prêt à l'emploi. Enfin presque ! Il reste tout de même à faire une page d'accueil, celle que verrons en premier vos lecteurs avide de lecture de votre prose. C'est le sujet du chapitre suivant, qui ne sera rien d'autre qu'un exercice de mise en pratique de tout ce que vous avez pu apprendre dans cette partie.

Contenu masqué

Contenu masqué n°10

```css
section.article header a {
  color: #007bff;
  text-decoration: none;
}

section.article header h1 {
  font-size: 40px;
  margin: 10px;
  text-align: center;
}

.metadata {
  display: flex;
  align-items: center;
  justify-content: space-between;
}

.metadata p {
  display: inline-block;
}

.link-item {
  display: inline-block;
  color: #007bff;
  border: solid 1px #007bff;
  text-decoration: none;
  border-radius: .25rem;
  padding: 5px;
  margin: 0 5px;
}

.link-item:hover {
  color: #fff;
```

```
34    background-color: #007bff;
35  }
```

Retourner au texte.

Contenu masqué n°11

```
1  footer {
2    display: flex;
3    align-items: center;
4    justify-content: center;
5    color: #000;
6  }
```

Retourner au texte.

Contenu masqué n°12

```
1  {% extends "base.html" %}
2
3
4  {% block extra_head %}
5      <link
           href='https://cdn.jsdelivr.net/npm/boxicons@1.5.1/css/boxicons.min.css
           rel='stylesheet'>
6
7      {% if article.description %}
8      <meta name="description" content="{{ article.summary }}" />
9      {% endif %}
10
11     {% for tag in article.tags %}
12     <meta name="tags" content="{{ tag }}" />
13     {% endfor %}
14 {% endblock %}
15
16
17 {% block content %}
18 <section class="article">
19     <header>
20         <p class="breadcrumb">
21             <a href="/{{ article.category.url }}">{{
                   article.category}}</a>
22             /
```

```
23              <span>{{ article.title}}</span>
24          </p>
25          <h1>{{ article.title}}</h1>
26          <div class="metadata">
27              <p>
28                  {% for author in article.authors %}
29                  <a href="/{{ author.url }}" rel="author">{{ author
                        }}</a>
30                  {% endfor %}
31                  <time datetime="{{ article.date.isoformat() }}"
                        pubdate>
32                      le {{ article.locale_date }}
33                  </time>
34              </p>
35              <p>
36                  {% for tag in article.tags %}
37                  <a class="link-item" href="/{{ tag.url }}">{{ tag
                        }}</a></li>
38                  {% endfor %}
39              </p>
40          </div>
41      </header>
42      <hr>
43      <div class="article-content">
44          {{ article.content }}
45      </div>
46      <hr>
47      <footer>
48          <span>Vous aimez cet article ? Partagez-le !</span>
49          <a class="link-item"
               href="https://twitter.com/share?url={{ SITEURL }}/{{ article.url }}
               rel="nofollow"
               title="Partager cet article sur Twitter">
50              <i class="bx bxl-twitter bx-md"></i>
51          </a>
52          <a class="link-item"
               href="https://www.facebook.com/sharer.php?u={{ SITEURL }}/{{ artic
               rel="nofollow"
               title="Partager cet article sur Facebook">
53              <i class="bx bxl-facebook bx-md"></i>
54          </a>
55          <a class="link-item"
               href="https://plus.google.com/share?url={{ SITEURL }}/{{ article.u
               rel="nofollow"
               title="Partager cet article sur Google +">
56              <i class="bx bxl-google-plus bx-md"></i>
57          </a>
58          <a class="link-item"
               href="mailto:?subject={{ article.title }}&amp;body={{ SITEURL }}/{
               title="Partager cet article par email">
```

```
59              <i class="bx bx-envelope bx-md"></i>
60          </a>
61      </footer>
62  </section>
63  {% endblock %}
```

Retourner au texte.

11. [TP] Page d'accueil

Vous savez maintenant pratiquement tout ce qu'il faut savoir pour créer les différentes pages de votre site. Il reste néanmoins la page `index.html` à réaliser. C'est la page *racine* de votre site, celle que les visiteurs verront lorsqu'ils arrivent sur l'accueil du site.

Afin de laisser librement s'exprimer votre créativité, je ne vais rien imposer ici, mais laissez votre imagination et votre talent s'exprimer. En effet, ce chapitre sera un TP !

11.1. Consigne et indices

11.1.1. Consigne

Vous l'avez compris, le but est de réaliser le contenu de la page `index.html`. Il n'existe pas une réponse universelle sur ce à quoi elle doit ressembler. À chaque site son identité !

Une seule consigne : Amusez-vous ! Faites vous plaisir et essayez de découvrir ce qu'il est possible de faire, et si possible intelligemment en faisant appel aux différentes structures que nous avons pu voir (`include`, `for`, `if/else`). Il n'y a cependant pas d'obligations, l'important est d'avoir un résultat qui vous convient.

11.1.2. Quelques indices et rappels

Une fois n'est pas coutume, revenons brièvement à la documentation, et notamment à la section « *Common variables* ». Cette dernière vous liste les différentes variables accessibles sur toutes les pages, donc aussi sur `index.html`. Voici celle que je juge les plus utiles (en français) :

— `articles` : La liste des tout les articles du site
— `dates` : La liste des tout les articles du site triés par date de publication
— `tags` : La liste des tags
— `categories` : La liste des catégories

Souvenez vous aussi que vous pouvez accéder à toutes les données d'un article lorsque vous les explorées. Par exemple `article.title` sera le titre de l'article en train d'être examiné (dans le cas de l'utilisation d'une boucle `for article in articles` par exemple).

L'index propose aussi, comme toutes les pages pouvant avoir des listes, un mécanisme de pagination pour les articles (la doc).

11.1.3. Bon courage !

Je n'ai pas grand chose de plus à vous dire. Amusez-vous bien, n'hésitez pas à aller voir comment font vos sites internet favoris pour vous inspirer à droite à gauche. Vous pouvez aussi allez vous inspirer des thèmes Pelican existant si vous êtes en panne de code !

Enfin, n'hésitez pas à venir sur le forum "Dev Web " pour partager vos créations ou poser vos questions !

Bon courage !

Qu'il est beau notre site ! Il est temps maintenant de le montrer à tout le monde !

Quatrième partie

Aller plus loin

"Quand y en a plus, y en a encore !". Vous savez tout pour faire un site web de base, qui tourne bien. Maintenant voyons ensemble quelques éléments supplémentaires pour aller toujours plus loin, grâce entre autres aux joie de l'open source et du partage de code.

12. Utiliser des plugins

De base, Pelican propose déjà pas mal de choses. Mais grâce à son système de *plugins*, il est possible de rajouter de nouvelles fonctionnalités à ce générateur de site en un clin d'œil.

Dans cette partie, je vous présenterais tout d'abord le principe de fonctionnement des plugins. Ensuite, je vous mentionnerais quelques plugins que je juge presque indispensable.

12.1. Généralités sur les plugins

12.1.1. *reader* ou *generator*, deux types de plugins

Dans écosystème Pelican, on trouve surtout deux types de plugins. L'un regroupe les *readers* (les *lecteurs*), et l'autre les *generators* (*générateurs*). Voyons les différences.

12.1.1.1. Les readers

Un reader est un plugin qui aura la charge de prendre un document en entrée pour le transformer en un nouveau format. Par exemple, un plugin qui permet de rédiger en LaTeX tombera dans cette catégorie. En effet, il aura la tâche de transformer un nouveau format d'entrée (le LaTeX) en html, le format des pages web. Le reader a deux tâches principales. La première, identifier et regrouper les métadonnées du contenu afin qu'elles soient utilisable dans les templates. La seconde est bien entendu la transformation du contenu réelle de l'article en html.

12.1.1.2. Les generators

Les générateurs ont aussi pour but de créer de nouvelles pages, mais pour cela il ne partent pas d'un seul fichier source mais plutôt de l'ensemble de votre site. Par exemple, le plugin "sitemap" que nous allons voir dans ce chapitre permet, à partir de l'ensemble des pages générées, de créer un fichier *sitemap.xml* qui regroupe le plan de votre site, bien utile pour les outils d'indexation des moteurs de recherche et autres robots.

12.1.2. Comment installer un plugin

Installer un plugin est relativement trivial. Il suffit tout simplement de télécharger ce dernier, le mettre au bon endroit puis régler sa configuration. Voyons cela.

12.1.2.1. Trouver un plugin

Pour cela rien de plus simple, la majeur partie des plugins (si ce n'est la totalité) se trouve sur le dépôt git suivant : https ://github.com/getpelican/pelican-plugins .

Une fois le plugin qui vous intéresse identifié, il ne reste plus qu'à le récupérer. Deux cas de figures peuvent alors se présenter. Soit le plugin est dans un sous-module git, et dans ce cas il suffit de cloner uniquement ce sous-module. Soit le plugin est directement dans le dossier. Dans ce cas, je vous conseille de passer par l'outil DownGit pour récupérer juste le dossier qui vous intéresse.

12.1.2.2. Installer le plugin

Passons maintenant à l'installation d'un plugin.

Tout d'abord, je vous propose de créer un dossier nommé `plugins` à la racine de votre dossier de travail (donc au même niveau hiérarchique que le dossier `content`).

Ensuite, il va falloir informer Pelican que nos plugins vont se trouver dans ce dossier. Il faut pour cela ajouter une ligne dans le fichier de configuration `pelicanconf.py` :

```
1 PLUGIN_PATHS = ["plugins"]
```

Cette ligne contient simplement un tableau de tout les dossiers que Pelican va explorer pour trouver des plugins.

Enfin, vous l'avez peut-être deviné, il va falloir copier/déplacer les plugins précédemment téléchargé pour les mettre dans ce fameux dossier `plugins/` que nous venons de créer.

Il ne reste plus qu'à activer ou non les plugins pour les voir rentrer en action. Une simple ligne dans le fichier de configuration fait ce travail :

```
1 PLUGINS = ["sitemap", "subcategory", "permalink"]
```

Là encore, c'est un tableau contenant les noms des dossiers des plugins à utiliser tels qu'ils sont nommés dans le dossier `plugins/`.

De manière général, pour obtenir plus d'informations sur les plugins la page de référence se trouve ici : http ://docs.getpelican.com/en/stable/plugins.html .

12.2. Une extension pour générer un sitemap

Afin de voir comment fonctionne les plugins, je vous propose d'en étudier deux : `sitemap` et `subcategory`. Commencons par `sitemap` dans cette partie.

12.2.1. Installer sitemap

Ce premier plugin permet de générer une page de type *sitemap*. c'est le type de page qui permet d'inventorier les pages accessibles pour les moteurs de recherche notamment. C'est donc plutôt utile si vous êtes sensible à la visibilité de votre site sur internet. En général, cela consiste en un fichier au format `xml` regroupant les pages ainsi que leur date de dernière mise à jour.

Trêve de bavardage, récupérons tout de suite le plugin (via DownGit) à l'adresse suivante : https ://github.com/getpelican/pelican-plugins/tree/master/sitemap .

Une fois que vous l'avez sur votre disque dur, il vous faudra le déplacer dans le dossier plugin de votre site internet comme vu précédemment. Enfin, il ne reste plus qu'à l'activer en l'ajoutant à la constante `PLUGINS` du fichier de configuration `pelicanconf.py`.

```
1 PLUGINS = ["sitemap"]
```

12.2.2. Paramétrer le plugin

Comme de très nombreux plugins, sitemap propose des options de configuration. Si vous ne faites rien, il fonctionnera de base très bien. Mais pour des utilisateurs ayant des besoins plus spécifiques, quelques options sont réglables. Ces dernières sont résumé dans le fichier `Readme` du plugin.

La première option est le format du fichier généré. Par défaut, ce sera du `xml`, mais un format texte brut (`txt`) est aussi possible. Pour choisir, il faut ajouter une nouvelle ligne dans le fichier de configuration `pelicanconf.py`. Cette nouvelle ligne sera une constante du nom de `SITEMAP` et contiendra un dictionnaire des différents paramètres à modifier. Donc pour régler le format de sortie, on obtiendra la ligne suivante :

```
1 # Pour passer au format txt plutôt que xml
2 SITEMAP = {
3     'format': 'txt'
4 }
```

Sitemap propose aussi d'exclure des urls pour limiter leur indexation. Cela se fait avec le paramètre `exclude` suivi d'un tableau des adresses à supprimer. Par exemple pour ne pas proposer les pages de tags et de catégories, on modifierai la donnée de la manière suivante :

```
1 SITEMAP = {
2     'format': 'txt',
3     'exclude': ['tag/', 'category/']
4 }
```

Un dernier paramètre intéressant à paramétrer est la fréquence de changement des contenus. Si vos contenus sont susceptibles de changer fréquemment, autant prévenir les moteurs de recherches pour qu'ils reviennent plus souvent voir les différences. Les fréquences possibles sont always (toujours en changement), hourly (toutes les heures), daily (tous les jours), weekly (hebdomadaire), monthly (mensuel), yearly (annuel) et never (ne changera jamais). Ces paramètres peuvent être appliqués aux articles, aux pages ou encore aux indexes (les pages de listes par exemple). Pour cela, on ajoute un dictionnaire à nos données :

```
SITEMAP = {
    'format': 'txt',
    'exclude': ['tag/', 'category/'],
    'changefreqs': {
        'articles': 'daily',
        'pages': 'monthly',
        'indexes': 'daily'
    }
}
```

Comme dis plus haut, toutes ces configurations sont facultatives, celle par défaut feront sûrement très bien le travail dans la plupart des cas.

Si jamais vous voulez en savoir plus ou voir les quelques paramètres restants, n'hésitez pas à consulter le Readme dans votre dossier d'installation ou en ligne .

12.3. Une extension pour créer des sous-catégories

Afin de clôturer ce chapitre avec de bonne idées, voici une deuxième extension que j'affectionne : subcategory. Comme son nom l'indique, elle permet de regrouper les contenus en catégories *et* sous-catégories. Par exemple, on pourrait avoir une catégorie "Dessert" qui regrouperait des sous-catégories "gâteaux", "cookies", "crèmes" ...

Comme pour le plugin précédent, ce dernier est disponible dans le dépôt principal de pelican-plugins à l'adresse suivante : https ://github.com/getpelican/pelican-plugins/tree/master/subcategory . Là encore, il faudra le déplacer dans le dossier `plugins/` de votre site web puis l'ajouter à la ligne `PLUGINS` de la configuration :

```
PLUGINS = ["sitemap", "subcategory"]
```

Maintenant, pour l'utiliser c'est très simple. Il suffit simple d'éditer la métadonnée `Category` de vos contenus. Par exemple, pour qu'un article soit automagiquement classé dans la catégorie "Dessert/Gateaux" on fera simplement `Category: Dessert/Gateaux` (le caractère `/` sert de séparateur). Vous pourrez maintenant assez simplement faire un fil d'ariane dans vos template en utilisant nouvelle variable `subcategories`. Par exemple :

```
<ol>
    <li><a href="/{{ article.category.url }}">{{
        article.category}}</a></li>
    {% for subcategory in article.subcategories %}
        <li class="breadcrumb-item"><a
            href="/{{ subcategory.url }}">{{ subcategory.shortname
            }}</a></li>
    {% endfor %}
    <li class="breadcrumb-item active" aria-current="page">{{
        article.title }}</li>
</ol>
```

Là encore, quelques paramètres sont disponibles, surtout lié aux chemins d'enregistrement des fichiers. Ceux par défaut sont de nouveaux très bien. Je vous laisse les consulter dans le README, à la section *settings* .

Vous savez maintenant comment ajouter des nouvelles fonctionnalités à votre générateur de site. Dorénavant, si une fonctionnalité vous manque, vous pourrez carrément la coder vous même afin d'enrichir votre contenu facilement. A ce sujet, j'avais de mon côté fait un plugin pour générer des pages à partir du format markdown utilisé sur Zeste de Savoir, le zmarkdown. Voici l'article à ce sujet .

13. Quelques paramètres de configuration

Ce dernier chapitre viendra conclure ce tutoriel. Il vous permettra quelques paramètres de configuration, afin de pouvoir peaufiner votre utilisation de Pelican et rendre votre installation la plus pratique pour vos usages. Les différents paramètres présentés ici peuvent tous être retrouvés sur cette page de la documentation de Pelican .

> Afin de garder le chapitre digeste, je ne vais pas présenter tous les paramètres utilisables dans le fichier de configuration, mais seulement ceux que j'estime les plus utiles pour commencer. Libre à vous de consulter la page de documentation ci-dessus pour en savoir plus.

13.1. Quelques bonnes pratiques

Commencons tout d'abord par quelques bonnes pratiques.

13.1.1. Sélection de la config

Saviez-vous qu'il est possible de sélectionner un fichier de configuration au moment de l'appel à Pelican ? En effet, une option à passer en paramètre de la commande `Pelican` permet de spécifier le chemin vers un fichier spécifique, plutôt que le fichier de base `pelicanconf.py` dont nous parlons depuis le début de ce tutoriel.

Cette option est activable via le paramètre `-s <chemin/du/fichier/de/conf.py>` (*s* pour *settings*). Il suffit donc simplement de rajouter ce paramètre à l'appel de la commande. Par exemple :

```
1 pelican content -s /home/eskimon/coolcookies/ma_conf_de_prod.py
```

13.1.2. Héritage de configuration

Il est toujours bon de séparer proprement les paramètres de votre environnement de développement et ceux servant à produire les pages finales. En effet, ces derniers peuvent varier entre les deux cas.

Cependant, il est pénible de recopier à la main tout les paramètres d'un fichier de configuration vers l'autre. De plus, faire ainsi risque d'engendrer des oublis si une modification doit être faite dans les deux fichiers à la fois.

Pour palier à cela, on utilise une méthode que l'on pourrait qualifier «d'héritage». Nous allons créer un fichier de base (en l'occurrence notre fichier habituelle `pelicanconf.py`) qui servira à définir tout nos paramètres par défaut commun à tout nos environnements (développement, préprod', prod' etc...). Ensuite, on créera autant de fichier de configuration propre aux différents environnements. Prenons par exemple le cas ou l'on veut une configuration de développement dans laquelle la pagination sera réglée à 5 articles pour s'assurer que le paginateur fonctionne bien, et une configuration de production avec une pagination à 20 articles. Dans notre fichier standard `pelicanconf.py` on aura alors `DEFAULT_PAGINATION = 5`.

Nous allons ensuite créer un fichier `prod_conf.py`. Ce dernier devra reprendre toute la configuration de base, mais devra mettre à jour le paramètre de pagination. Pour cela, on commencera par importer dans notre fichier toutes les valeurs du fichier «parent», `pelicanconf.py`. Ensuite, il ne restera qu'à modifier les paramètres nécessaires. Tout les autres seront automatiquement importé, mais garderont leur valeur d'origine.

```
# On importe toute la configuration de pelicanconf
import os
import sys
sys.path.append(os.curdir)
from pelicanconf import *

# Puis on modifie les paramètres qu'il faut mettre à jour
DEFAULT_PAGINATION = 20
```

Ainsi, quelque soit les modifications faites dans `pelicanconf.py`, on les retrouvera forcément dans `prod_conf.py`.

Il ne reste plus qu'à appeler ce fichier lors de la génération via la commande

```
pelican content -s prod_conf.py
```

13.2. Gestion des fichiers statiques

Tout d'abord, clarifions la notion de fichiers «statiques» étant donnée que depuis le début on travaille sur un site qui est lui-même statique. Dans le contexte du développement web, un fichier statique est un fichier qui doit-être servi tel quel, sans manipulation du serveur. Par exemple, le fichier contenant le CSS ou encore une image sont tout deux des fichiers que l'on qualifient de statiques.

Dans Pelican, on trouve ce type de fichiers dans deux endroits bien différents : pour le thème et dans les contenus.

13.2.1. Pour le thème

Comme on vient de le voir, le thème a besoin de fournir des fichiers statiques, notamment pour le CSS. Lorsque l'on a créé notre thème, nous les avons mis dans le dossier `static/css`. Ce dossier peut-être personnalisé via le paramètre `THEME_STATIC_PATHS`. Par défaut, on a nommé notre dossier `static`, mais on aurait très bien pu changer ce chemin si besoin. Vous remarquerez que cette variable est un tableau. En effet, on peut fournir plusieurs chemins à utiliser.

```
1 THEME_STATIC_PATHS = ['static']
```

13.2.2. Les contenus statiques

Sur notre site, on peut vouloir imaginer avoir des fichiers que l'on a réaliser nous-même et ne sont donc pas généré via le Pelican. Ce peut-être par exemple des fichiers «de gestion techniques» du site (comme le fichier `robots.txt`) ou bien des documents que l'on souhaiterais mettre à disposition du public (un fichier pdf par exemple).

Là encore, il va falloir informer Pelican où trouver ces fichiers, et où les placer dans le dossier `output` final.

Pour cela, on utilise plusieurs paramètres. Tout d'abord, `STATIC_PATHS` qui va lister les emplacements à copier. Dans l'exemple suivant, l'ensemble du dossier `pdfs` et le fichier `extra/robots.txt` seront copiés. Ces deux éléments seront considérés comme étant dans le dossier de contenu `content`.

```
1 STATIC_PATHS = ['pdfs', 'extra/robots.txt']
```

Si l'on ne fait rien de plus, les éléments seront copiés tel quel ainsi que leur arborescence directement dans le dossier `output`. Cependant, certains fichiers peuvent nécessiter d'être placé à un endroit particulier. C'est là que rentre en jeu `EXTRA_PATH_METADATA`. Ce dictionaire va permettre de paramètrer certaines métadonnée de différents objets, en l'occurrence nos fichiers statiques. Pour cela, on lui fournir comme clé le fichier à modifier (`extra/robots.txt` par exemple) et la donnée à modifier (en l'occurence son chemin, `path`).

Voici par exemple ce que nous pourrions faire pour déplacer le fichier `extra/robots.txt` à l'emplacement `robots.txt` (sans le préfixe `extra/`) :

```
1 EXTRA_PATH_METADATA = {
2     'extra/robots.txt': {'path': 'robots.txt'}
3 }
```

(Vous remarquerez que je n'ai pas touché au dossier `pdfs`).

13.3. Quelques conseils supplémentaires

Voici en vrac quelques paramètres qui peuvent toujours être utile.

13.3.1. Pour les flemmards

Si vous en avez marre de spécifier `content` dans l'appel de votre commande Pelican, vous pouvez spécifier le nom du dossier qui contient le contenu (articles, pages, etc) dans la configuration. C'est le paramètre `PATH` qui gère cela.

```
PATH = 'content'
```

De la même facon, si vous êtes seul à rédiger tout le contenu, la variable author dans toutes les métadonnées des articles est un peu monotone. Elle peut-être paramétrer directement via la variable `AUTHOR` dans la configuration.

```
AUTHOR = 'Eskimon'
```

13.3.2. Les der des ders

13.3.2.1. Enlarge your summary

Si vous trouver que la variable contenant le résumé (`article.summary`) est trop courte, vous pouvez l'augmenter via le paramètre `SUMMARY_MAX_LENGTH` avec un entier représentant le nombre de caractère.

```
SUMMARY_MAX_LENGTH = 100
```

13.3.2.2. Altération du dossier output

Enfin, je vous propose pour finir deux astuces liés au dossier qui va contenir le site généré. Premièrement, la constante `OUTPUT_PATH` permet de donner le nom de ce dossier. Par défaut c'est `output`, mais vous pouvez mettre ce que vous voulez. Par exemple, si vous souhaitez faire de l'hébergement sur l'outil Pages de GitLab , alors il faudra mettre ce nom de dossier à `public`.

```
OUTPUT_PATH = 'public/'
```

Enfin, si vous souhaitez conservez des fichiers/dossiers dans votre dossier de sortie (`output`), comme par exemple les fichiers de gestion de version si vous gérer votre déploiement via un outil de ce genre, c'est la donnée `OUTPUT_RETENTION` qu'il faut éditer. Il faut lui fournir un tableau contenant tout les fichiers à conserver. Ces derniers ne seront alors plus effacés si vous avez activez le paramètre `DELETE_OUTPUT_DIRECTORY`.

```
# Par exemple, pour conserver mon suivi de version via git
OUTPUT_RETENTION = ['.git', '.gitignore']
```

Ce tutoriel touche maintenant à sa fin. Bien sûr, comme souvent en informatique, "fin" est un bien grand mot. Mais vous devriez dorénavant avoir toutes les informations pour poursuivre par vous même votre apprentissage et continuer à améliorer les sites que vous souhaiter produire. Bonne continuation à vous !

Liste des abréviations

CSS Cascading Style Sheets. 4, 8, 19, 29, 35, 45, 47, 49, 50, 52, 61, 68, 71, 77–80, 99, 100

HTML HyperText Markup Language. 4, 7, 8, 19, 25, 26, 35, 36, 39, 45, 47, 49, 50

SEO Search Engine Optimization. 77