Rapport de stage chez BeSport

Contribution au projet libre Ocsigen

Simplification, amélioration et extension d'un compilateur wikicréole → HTML existant. Planification et exécution d'un plan de déploiement de celui-ci pour tous les projets Ocsigen. Mise en place d'un processus d'intégration continue à l'aide de Travis CI.

Léo Valais (valais_l)

12 septembre 2018 — 15 janvier 2019

Table des matières

1 Introduction **1**

2 Présentation de BeSport **3**
2.1 L'entreprise 4
2.1.1 Équipe technique 4
2.1.2 Équipe sport 4
2.2 Le service 4
2.2.1 Le réseau social BeSport 4
2.2.2 Le projet Ocsigen 5
2.3 Le positionnement du stage dans les travaux de l'entreprise 5

3 Le travail effectué — one_html_of_wiki **5**
3.1 But général 6
3.1.1 Le wikicréole 6
3.1.2 Ocsimore et Ocsiforge 6
3.1.3 html_of_wiki 7
3.1.4 one_html_of_wiki 8
3.2 Cahier des charges 10
3.2.1 Le langage à supporter : le wikicréole extensible 10
3.2.2 L'outil principal : one_html_of_wiki 13
3.2.3 L'outil auxiliaire de ohow : wiki_in_template 23
3.2.4 doc_of_project 26
3.2.5 quickdop 27
3.2.6 Gestion de la documentation du project Ocsigen 28
3.3 Réalisation 29
3.3.1 Développement 29
3.3.2 Déploiement avec Travis CI 31
3.3.3 Documentation 35
3.4 Conformité au cahier des charges 35

4 Conclusion du stage **35**

5 Annexes **39**
5.1 Le Wikicréole 39
5.1.1 Standard 39
5.1.2 Additions du projet Ocsigen 39
5.2 GitHub Pages 41
5.3 Documentation de html_of_wiki 41

1 Introduction

Ayant travaillé sur un projet en Common Lisp au LRDE[1], je souhaitais trouver un stage centré sur la programmation fonctionnelle, de préférence avec un langage statiquement typé. Mon choix s'est porté sur l'offre de stage de la startup *BeSport* essentiellement pour deux raisons :

1. leur langage de prédilection est OCaml. C'est un langage fondamentalement fonctionnel, statiquement typé développé à l'INRIA[2]. Cela correspond donc parfaitement à mes attentes en termes de technologies.
2. leurs applicatifs (*backend*, *frontend*, *importation de données*, etc.) sont entièrement développés en OCaml, notamment grâce au projet Ocsigen, permettant le développement d'applications dites *multi-tiers*. Habituellement développées en JavaScript (NodeJS pour le backend et ReactJS, Angular ou VueJS pour le frontend le plus souvent), l'utilisation de OCaml, de sa *rigueur*, de son *système de types fort*, sur *l'intégralité de la stack*, permet entres autres :
   — de partager les structures de données et fonctions entre le code client et le code serveur,
   — de vérifier statiquement les pages web générées pour être conforme à la norme W3C,
   — toujours bénéficier des technologies JavaScript de pointe grâce au compilateur Js_of_ocaml, et
   — pouvoir exporter l'application autant vers le web que le mobile (iOS et Android).

* * *

Le sujet proposé sur la convention est :

> « *[…] Le but du stage est de développer de nouveaux modules pour les applications Web et mobiles Be Sport : outils d'importation de données, amélioration des outils de développement (dans le cadre du projet libre Ocsigen), développement de widgets spécifiques, interaction avec la base de données, mise en place de tests de l'application. Le stagiaire s'occupera des spécifications en collaboration avec les membres de l'équipe, puis travaillera sur un prototype et ira jusqu'à la mise en production.* »

Ce sujet, vaste, permet d'avoir un aperçu des différentes fonctionnalités de BeSport tout en exploitant au maximum les possibilités du langage OCaml et du projet Ocsigen.

Cependant, en arrivant, il m'a été confié une mission plus urgente que le développement de l'application BeSport : *la création d'un outil ergonomique de gestion de la documentation du projet Ocsigen pour GitHub Pages*. En effet, la documentation est écrite dans un format particulier (le wikicréole) muni d'extensions propriétaires, qu'il est nécessaire de compiler vers de l'HTML à chaque modification de celle-ci. C'était le but de Ocsimore et Ocsiforge, outils du projet Ocsigen, aujourd'hui dépréciés. Les remplacent un outil interne, lourd, monolithique et peu flexible, qu'il m'a été demandé de remplacer. Ma mission consiste donc à développer cet outil, simple, ergonomique et flexible, générant les pages de documentation au format HTML, afin de les télé-verser vers GitHub Pages. Il me faut également déployer cet outil vers tous les projets Ocsigen afin de permettre, à l'aide de Travis CI, de générer et déployer leur documentation en continu, à l'échelle du commit. La mise en place d'un environnement d'intégration continue, ou CI/CD, fait donc partie mon sujet de stage.

1. Laboratoire de Recherche et Développement de l'ÉPITA
2. Institut National de Recherche en Informatique et en Automatique

Mes attentes par rapport à ce stage sont multiples.

— Découvrir l'utilisation professionnelle de langages fonctionnels et de OCaml en particulier.
— Découvrir la programmation multi-tiers au travers du projet Ocsigen.
— Une introduction à l'univers de la startup et les différentes méthodologies de travail telles qu'Agile et Scrum et les pratiques associées.
— Une introduction aux tâches de *DevOps*, en particulier l'intégration continue (CI) en milieu professionnel.
— Une première expérience de la collaboration à plus grande échelle que les projets scolaires, avec des enjeux réels.

2 Présentation de BeSport

2.1 L'entreprise

BeSport est une *startup* développant un *réseau social* du même nom. Ce dernier, disponible sur toutes les plate-formes majeures, c'est-à-dire *le web, iOS et Android*, est destinée, comme son nom l'indique, au milieu du sport.

Travaillent chez BeSport les personnes listées ci-dessous, dans deux « équipes » distinctes : une équipe *technique*, chargée du développement de la plate-forme et une équipe *sport* qui est chargée de fournir du contenu sportif ainsi que de garantir que le produit soit au plus proche des besoins du sport possible.

S'ajoutent à cela quatre stagiaires, moi compris.

2.1.1 Équipe technique

Phillipe Robert CEO

Vincent Balat CTO

Jérôme Vouillon Ingénieur de recherche OCaml et Ocsigen

Jan Rochel Ingénieur de recherche

Rémy El Sibaïe Développeur

Idir Lankri Développeur

Baptiste Strazzula Développeur

Kolia Iakovlev Data scientist/importation de données

Maya El Sibai UI/UX designer

Victor Girla UI/UX designer

Léa Hubsch Ressources humaines & stratégie

2.1.2 Équipe sport

Benoit Guyot

Jean Marc Gillet

Léopold Rucher

Hugo Bruchet

François Trillo

Raphaelle Steg

2.2 Le service

2.2.1 Le réseau social BeSport

Le réseau social BeSport est né du constat que le milieu du sport avait encore peu bénéficié de cette « transformation numérique » censée toucher toutes les activités. En effet, le sport reste très hiérarchisé et presque exclusivement relayé par les « médias traditionnels » tels que la télévision et la radio.

Le projet BeSport, ambitieux, vise donc à mettre à disposition une plate-forme qui mettrait tout cela « à plat ». Elle est conçue pour un panel d'utilisateurs très large, allant des *professionnels* du sport aux *pratiquants* (athlètes) en passant par les simples « *fans* » de sport.

BeSport veut aussi se distinguer sur deux dimensions : la « *largeur* » et la « *profondeur* ». La largeur est le panel de sport supportés et la profondeur est l'écart de niveau supporté pour un sport donné. BeSport veut donc supporter le maximum de sports possible — c'est-à-dire pas juste le football et le rugby, mais aussi des sports moins populaires (ou moins médiatisés) tels que le surf ou le hockey sur glace. De plus, BeSport veut avoir le plus de profondeur possible dans chaque sport : chacun d'eux doit être représenté, que ce soit au niveau professionnel ou olympique qu'au niveau amateur ou des clubs locaux.

BeSport a donc pour objectif de toucher le monde du sport au sens large : **tous les sports, à tous niveaux**. Pour cela, BeSport doit d'abord être techniquement disponible sur toutes les plate-formes (web et mobile) : il a donc été choisi que le réseau social serait développé avec l'aide du projet Ocsigen.

2.2.2 Le projet Ocsigen

Le projet Ocsigen est un projet *open-source* développé à l'IRILL [3], en OCaml, servant à développer des applications dites *multi-tiers* dans ce même langage. Ce projet est composé de différents sous-projets indépendants mais permettant, lorsqu'utilisés conjointement, de développer *l'intégralité d'une application web en OCaml*, *backend* et *frontend* compris. Parmi ceux-ci l'on peut citer les plus populaires :

Js_of_ocaml un compilateur de *bytecode* OCaml → JavaScript,

Eliom *framework* pour la programmation multi-tiers,

Tyxml bibliothèque permettant de construire des pages HTML en OCaml.

* * *

Il y a de nombreux avantages à utiliser un langage comme OCaml pour développer des applications web complexes, par exemple :

- c'est un langage fonctionnel et rigoureux moins sensible aux erreurs une fois déployé,
- son système de types fort et poussé permet (à l'aide de Tyxml), de ne générer que des pages HTML conformes à la norme W3C,
- c'est un langage pensé pour être fonctionnel en son coeur, paradigme qui ne se rajoute qu'aujourd'hui dans le milieu du développement web (comme une « sur-couche » sur des langages pas vraiment pensé pour cela au départ), ou encore,
- avoir *un seul* programme, *cohérent* puis-qu'écrit dans un seul et même langage, au lieu d'un programme client plus un programme serveur plus un middleware dans le cloud plus etc. tous écrits dans des langages différents.

2.3 Le positionnement du stage dans les travaux de l'entreprise

C'est d'abord dans le cadre d'une *contribution au projet Ocsigen* que mon stage se porte. En effet, BeSport utilisant un projet de recherche open-source comme *stack*, l'entreprise dispose d'un financement recherche afin de continuer à maintenir et développer le projet Ocsigen.

Ma mission est donc de reprendre et transformer un outil interne de gestion de la documentation du projet afin de le rendre plus ergonomique, flexible et plus complet que le précédent. L'objectif final étant de l'inclure dans la liste des projets « publics » du projet Ocsigen comme outil générique de gestion de documentation.

3. L'Initiative pour la Recherche et l'Innovation sur le Logiciel Libre. C'est un laboratoire de recherche français. Ce centre a été créé en septembre 2010 par l'INRIA avec l'Université Pierre-et-Marie-Curie et l'Université Paris VII - Diderot.

Wikicréole

```
Un paragraphe de wikicréole.

Le **gras** entre deux astérisques, //l'italique// entre deux barres obliques

= Listes (en-tête de premier niveau préfixée par un =)
== Listes non-numérotées

* le wikicréole
** permet
** de faire
* bien plus

== Listes numérotées

# des tableaux
# des liens: [[www.google.fr|lien vers google]]
# etc.
```

Listing 1: Exemple de texte formatté en wikicréole.

3 Le travail effectué — one_html_of_wiki

3.1 But général

3.1.1 Le wikicréole

Le projet Ocsigen est vieux de plus de dix ans. Les formats de documentation actuellement répandus, tels que le *Markdown* ou le *reStructured Text*, n'existaient alors pas ou commençaient à émerger. Le seul format alors en vigueur était le format *wiki* de Wikipédia, et toutes ses déclinaisons. C'est l'une d'entre elles qui a été choisie pour la documentation du projet Ocsigen : le *wikicréole*. Le wikicréole, qui a l'avantage d'être arrivé plus tard que ses concurrents, est pensé pour corriger les erreurs de conception que ces derniers ont pu commettre, tout en conservant l'esprit et la syntaxe générale de ceux-ci.

Le listing 1 montre un texte formatté en wikicréole. C'est donc un format léger et relativement pratique à utiliser, mais il est nécessaire de le convertir en HTML pour pouvoir le publier (sur GitHub Pages dans le cas de BeSport).

3.1.2 Ocsimore et Ocsiforge

C'était le but des outils Ocsimore et Ocsiforge, autrefois partie du projet Ocsigen, aujourd'hui déprécies et plus maintenus.

Leur rôle était de fournir une infrastructure permettant de gérer la documentation d'un ensemble de projets. Ils servaient, entres autres, à :

— convertir la documentation de chaque projet du wikicréole vers de l'HTML,
— insérer les pages web dans des *templates* (de patrons) de page web,
— gérer les différentes version de la documentation de chaque projet,
— gérer les permissions sur l'écriture de la documentation,
— déployer la documentation générée sur internet,

— proposer un blog, un newsfeed, etc., et
— gérer un base d'utilisateurs s'étant inscrits sur le site déployé.

De plus, Ocsimore a ajouté au wikicréole un système *d'extensions* afin d'étendre les possibilités du langage. Schématiquement, les extensions fonctionnent de la manière suivante :

— Chaque extension s'enregistre auprès du compilateur (elle donne son nom — par exemple `"myext"` — et une fonction OCaml — `f`).
— Lorsque l'extension est rencontrée dans fichier — `<<myext arg="val" | content>>` — le compilateur appelle la fonction associée — `f [("arg", "val")] content`.
— La fonction retourne alors le code HTML qui sera mis à la place de l'extension le fichier final.

Ce puissant système d'extensions naturellement *Turing-complet* permet d'étendre considérablement le wikicréole.

Cependant ces projets devenant trop lourds et difficile à maintenir, leur support et leur développement a été arrêté et une majorité de leur fonctionnalités abandonnées. Mais parmi celles-ci, *l'exportation des wikis vers des pages HTML se devait d'être conservée* pour garantir la disponibilité de la documentation aux utilisateurs du projet Ocsigen.

3.1.3 html_of_wiki

C'est dans cette optique qu'est né le projet html_of_wiki[4] (abrégé **how**). Cet outil reprend une partie du code d'Ocsimore, l'analyseur syntaxique (*lexer*) et l'analyseur grammatical (*parser*), mais est *complètement dédié au projet Ocsigen*. C'est donc un outil *interne*. Son objectif initial devait être simplement de convertir le wikicréole en HTML. Cependant, comme ce projet est en réalité une *simplification d'Ocsimore*, il est à cause de cela soumis à un certain nombres de contraintes, en particulier en termes de condition d'utilisation. Pour comprendre en quoi la présence de ces contraintes d'utilisation est indésirable, il faut d'abord s'intéresser au fonctionnement de **how**.

1. **how** clone le dépôt git dédié nommé `ocsigen.org-data`, qui contient la documentation de chaque version de chaque projet.
2. **how** compile ensuite chaque fichier `*.wiki` en un fichier HTML, tout en s'assurant que chaque lien et valide (signale les liens morts) et que les *extensions* sont bien exécutées.
3. Les pages HTML sont ensuite placées dans un dossier `ocsigen.org-repositories/`.
4. Ces pages sont enfin copiées dans la branche `gh-pages` du projet auquel elles appartiennent et sont poussées vers GitHub.

Plusieurs inconvénients majeurs se distinguent déjà.

— Ou bien l'on génère *toute* la documentation, ou bien l'on ne génère *rien*. Il n'est pas possible de ne compiler que quelques fichiers choisis, ou qu'une version donnée d'un projet donné.
— *Toute* la documentation de *toutes* les versions de *tous* les projets se trouve dans *un seul* dépôt `ocsigen.org-data`. Celui-ci est donc trop lourd (plusieurs gigaoctets).
— Il faut penser manuellement à ajouter les nouvelles versions de chaque projet à ce dépôt.
— La nécessité de `ocsigen.org-data` induit une forte désorganisation au sein des projets :

4. Il existe en OCaml une convention de nommage pour les fonctions de conversion. Par exemple, la fonction standard qui convertit un `int` en `string` s'appelle `string_of_int`. C'est en suivant cette convention que l'outil qui transforme du wikicréole en HTML s'appelle html_of_wiki.

— le code d'un projet X se trouve sur `master` dans le dépôt X,
— les wikis de X se trouvent se trouvent dans le dossier X de `ocsigen.org-data`, et
— les pages HTML générées se trouvent sur la branche `gh-pages` du dépôt X.

D'autre part, how souffre également des problèmes suivants :

— La page d'accueil https://ocsigen.github.io doit être traitée comme un cas particulier [5].
— Les liens générés sont des liens *absolus*, ce qui proscrit toute modification de l'architecture des projets ou du site au risque de casser la plupart des liens.
— Puisque la gestion des versions est manuelle, la gestion des redirections l'est aussi [6].

> **Exemple**
>
> La page d'accueil du projet Eliom doit être `eliom/2.8/intro.html`, en supposant que `2.8` est la dernière version en date. Il faut alors créer le lien symbolique `index.html` à la racine du projet sur la branche `gh-pages` et le faire pointer vers `intro.html`. À la sortie de la version `2.9`, il faut alors penser à placer les wikis dans `ocsigen.org-data/eliom`, mais aussi à mettre à jour le lien `eliom/index.html` décrit ci-dessus.

— Un certain nombre d'inconvénients mineurs (par exemple, devoir mettre les wikis ayant le rôle de manuel d'utilisation dans un dossier `src/`).

La figure 1 illustre et résume la (lourde) architecture nécessaire pour faire fonctionner html_of_wiki.

how est donc un outil qui fonctionne, mais lourd, monolithique, peu flexible et peu ergonomique. La mission de créer un outil pour le remplacer m'a donc été confiée.

3.1.4 one_html_of_wiki

L'idée était donc de simplifier drastiquement how mais en le privant du minimum de fonctionnalités possible. Ce nouvel outil devait être capable de compiler les wikis *un par un* et **sans examen de dépendances** pour la construction des liens. C'est pour cela que cet outil a été nommé one_html_of_wiki (abrégé ohow). Cet outil se devait d'être *simple* et *composable* (plus en accord avec la philosophie Unix). Cependant, cela implique la nécessité d'un outil supplémentaire, capable d'utiliser ohow pour générer la documentation *d'une version entière* d'un projet donné : doc_of_project (abrégé dop). Ces deux outils ont donc principalement pour but de :

— remplacer définitivement how,
— avoir la possibilité de placer la documentation des projets ailleurs que dans `ocsigen.org-data` et de s'en débarrasser,
— garder la syntaxe wikicréole extensible, et surtout,
— *reproduire le site existant au minimum à l'identique* (au mieux avec quelques améliorations).

Le listing 2 montre le code HTML généré par ohow à partir du wikicréole du listing 1.

L'objectif principal est de remplacer l'utilisation de how en *interne*. Cependant, dans un second temps et selon mon choix, il est aussi possible de pousser le développement de ces outils plus loin, afin de les *distribuer* et de les rendre utilisables par la communauté OCaml

5. Cela est dû au fonctionnement de GitHub Pages. Pour plus d'information consulter l'annexe 5.2 sur GitHub Pages.
6. GitHub Pages permet de définir des redirections sous forme de liens symboliques. Consulter l'annexe 5.2 pour plus d'informations.

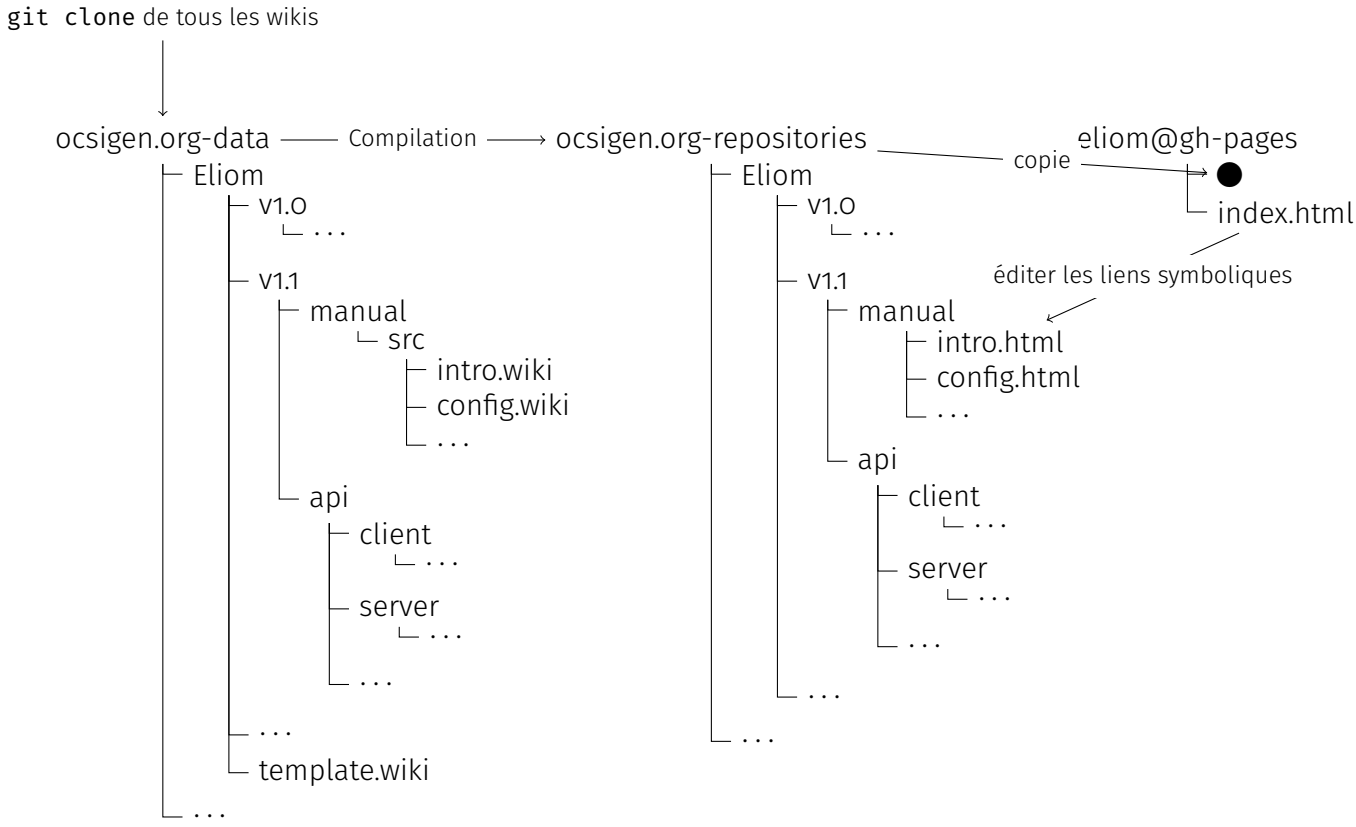


FIGURE 1 – Architecture nécessaire pour faire fonctionner html_of_wiki.

HTML code

```
<p>Un paragraphe de wikicréole.
</p>
<p>Le <strong>gras</strong> entre deux astérisques,
 ↪  <em>l'italique</em> entre deux barres obliques
</p>
<h1> Listes (en-tête de premier niveau préfixée par un =)</h1>
<h2> Listes non-numérotées</h2>
<ul><li> le wikicréole
<ul><li> permet
</li><li> de faire
</li></ul></li><li> bien plus
</li></ul>
<h2> Listes numérotées</h2>
<ol><li> des tableaux
</li><li> des liens: <a href="www.google.fr"
 ↪  class="ocsimore_phrasing_link">lien vers google</a>
</li><li> etc.</li></ol>
```

Listing 2: Le code HTML du wiki du listing 1 produit par one_html_of_wiki.

(en réalité par tous ceux ayant ou souhaitant avoir de la documentation écrite en wikicréole extensible).

3.2 Cahier des charges

Comme expliqué dans la section « But général », le cahier des charges pour ce projet dépasse la simple réalisation de l'outil one_html_of_wiki. Les principaux objectifs de cette mission sont les suivants.

- Remplacer html_of_wiki par la combinaison de deux outils : one_html_of_wiki, le compilateur wikicréole → HTML qui opère fichier par fichier et doc_of_project, qui opère version de projet par version de projet,
- conserver le même format de wikicréole et les même extensions (100% rétrocompatible avec how),
- être capable, à l'aide de ces outils, de régénérer le site https://ocsigen.org à l'identique,
- l'améliorer et régler ses différents problèmes techniques (barre de recherche, design, etc.),
- trouver une solution pour se passer du dépôt `ocsigen.org-data`, et,
- éventuellement rendre ohow suffisamment indépendant de l'infrastructure du projet Ocsigen pour pouvoir le diffuser et qu'il puisse être utilisé sur d'autres projets.

L'on commence par spécifier le langage que doit supporter ohow, puis l'on dresse son cahier des charges détaillé. Nous continuons par la description de celui de dop et finissions par la liste des objectifs à atteindre quant à l'organisation du projet Ocsigen.

3.2.1 Le langage à supporter : le wikicréole extensible

Le wikicréole La spécification du langage wikicréole utilisée par le projet Ocsigen est décrite dans l'annexe 5.1.

Les extensions La documentation du projet Ocsigen s'appuie fortement sur une construction syntaxique particulière : *les extensions*. Ce sont des constructions non-standard du wikicréole développées afin d'augmenter les capacités du langage en permettant l'exécution de code OCaml *pendant la compilation* du fichier.

Concrètement, les extensions ont la syntaxe suivante :

```
1  <<extension_name attr1="val1" attr2="val2" ... attrN="valN" |
   ↪  content>>
```

Note

- Les attributs sont conçus pour avoir des valeurs (`val1`, `val2`, etc.) contenant du *texte brut* à utiliser comme paramètre pour la fonction OCaml.
- En revanche, le contenu `content` de l'extension peut contenir du wikicréole, *arbitrairement complexe*, s'étendant possiblement sur plusieurs lignes, pouvant lui aussi *contenir des extensions* et est donc conçu pour être *compilé récursivement*.

L'auteur de chaque extension doit alors *l'enregistrer auprès du compilateur* de wikicréole en lui fournissant le **nom** de l'extension (ici `"extension_name"`) ainsi qu'une **fonction OCaml** qui sera appelée à chaque fois que l'extension est rencontrée dans le fichier en train d'être compilé. Cette fonction doit retourner du code HTML qui sera positionné à l'endroit où se situe l'extension dans la page web produite.

λ OCaml code

```ocaml
(*
   Fournit une extension qui colore le texte du contenu.
   Exemple d'utilisation:
     <<colored color="blue" | **I'm bold and blue.**>>
   produit:
     <div style="color: blue"><strong>I'm bold and
    ↪  blue.</strong></div>
*)

(** Fonction produisant le code HTML de l'extension "colored". *)
let create_colored_text _ args content =
  let color = List.assoc "color" in
  Tyxml.Html.(div ~a:[a_style ("color: " ^ color)]
                content)

(* Enregistement de l'extension auprès du compilateur. *)
let () =
  Wiki_syntax.register_simple_extension
    Wiki_syntax.wikicreole_parser (* analyseur syntaxique du
     ↪  contenu *)
    "colored" (* nom de l'extension *)
    create_colored_text (* fonction à appeler *)
```

Listing 3: Code OCaml nécessaire pour ajouter l'extension `colored`.

Le listing 3 illustre comment ajouter une extension au compilateur décrit dans le paragraphe suivant et permet de comprendre aisément en quoi ce système est très puissant de par sa capacité à exécuter du code arbitrairement complexe au coeur du processus de compilation.

Le tableau 1 liste et décrit les extensions (et leur attributs [7]) définies pour le projet Ocsigen que ohow doit pourvoir supporter [8].

Le compilateur wikicréole → HTML Comme expliqué dans la section 3.1, ohow doit être construit par dessus how, lui-même écrit à partir de Ocsimore et Ocsiforge. Ces derniers avaient implanté un compilateur wikicréole → HTML écrit en OCaml afin de pouvoir remplir leur fonction. Ce compilateur a été repris par how et *doit également être réutilisé par* ohow.

Cependant celui-ci doit d'être *modifié*, notamment ce qui concerne la génération de liens entre les pages web. En effet, les liens générés sont des *liens absolus*, trop rigides (il est impossible de tester localement le site web généré puisque les liens pointent vers des URLs comme https://ocsigen.org/project/page). Il faut donc modifier les procédures concernées afin de générer les *liens relatifs*, plus flexibles, mais plus contraignants à implanter.

7. Toutes les extensions acceptent les attributs `class` et `id` correspondant aux attributs HTML de même nom.
8. Les descriptions données sont simplement indicatives. Les extensions complexes seront décrites avec plus de précision lors de l'explication des problèmes techniques qu'elles causent.

Nom	Attributs	Courte description
`a_api_code`	`project`, `subproject`, `version`	Insère un lien vers l'API (implantation).
`a_api_type`	`project`, `subproject`, `version`	Insère un lien vers l'API (déclarations).
`a_api`	`project`, `subproject`, `version`	Insère un lien vers l'API du projet donné.
`a_file`	`src`	Insère un lien vers un fichier quelconque.
`a_img`	`src`	Insère une image.
`a_manual`	`project`, `chapter`, `version`	Insère un lien vers le chapitre donné.
`client-server-switch`		Insère un lien API client ↔ API serveur.
`code-inline`	`language`	Insère du code à l'intérieur d'un paragraphe.
`code`	`language`	Insère un bloc de code.
`div`		Insère un élément HTML `<div>`.
`doctree`		Insère le menu du projet.
`docversion`		Insère une liste de versions à afficher.
`drawer`		Insère un menu latéral déroulant.
`googlesearch`		Insère une barre de Google Search.
`nav`		Insère un élément HTML `<nav>`.
`pre`		Insère un élément HTML `<pre>`.
`reason-switch`		Convertit le OCaml en ReasonML.
`script`	`src`	Insère un script JavaScript.
`span`		Insère un élément HTML `<span>`.
`wip-inline`		De même mais à l'intérieur d'un paragraphe.
`wip`		Marque un contenu « Work In Progress ».
extension sans nom		N'insère rien : a fonction de commentaire.

TABLE 1 – Extensions Ocsigen que **ohow** doit impérativement supporter.

3.2.2 L'outil principal : one_html_of_wiki

La fonction de one_html_of_wiki est *simple* et *unique*[9] :

> « *Pouvoir compiler les fichiers wikicréole* ***individuellement*** *et* ***sans examen de dépendances*** *(compilation* ***isolée****).* »

C'est-à-dire que **ohow** peut être appelé à compiler un fichier `mydoc.wiki` et doit produire `mydoc.html`, document HTML consultable depuis un navigateur web et conforme à la norme W3C. L'outil doit avoir généré des liens *relatifs* et *valides* sans vérifier que la page de destination existe.

Objectifs

- C'est un outil en *ligne de commande*, écrit en OCaml qui doit réutiliser le compilateur wikicréole → HTML de **how**.
- Le fichier wikicréole à compiler ne doit pas être modifié.
- Les pages HTML produites doivent être complètes (avec un en-tête `<head>` correct).
- Il doit être possible de ne pas générer cet en-tête avec l'option `--headless`.
- Les pages HTML produites doivent être conformes à la norme W3C.
- Les liens générés doivent être tous[10] relatifs.
- Il ne doit pas y avoir d'examen de dépendances.
- Il doit être possible de tester les pages localement avec des liens redirigeant vers des adresses existantes (option `--local`).
- Il doit gérer (au minimum) les extensions listées par le tableau 1.
- Les URLs produites par **ohow** doivent être *compatibles* avec celles générées par **how**. Cela a pour effet de ne pas casser les liens que des sites tiers font avec la documentation.

Interface à respecter

- Il doit y avoir une option `--print` pour afficher le code HTML résultant dans la console (sortie standard) afin de pouvoir être composé[11].
- Il doit y avoir une option `--output FILE` pour sélectionner le nom du fichier HTML de sortie. En conséquence, les deux commandes du listing 4 doivent produire *exactement* le même résultat.
- Pour des raisons de commodité, **ohow** doit pouvoir accepter plusieurs fichiers en entrée mais les résultats dans ce cas-là doivent être *strictement les mêmes* que si **ohow** avait été exécuté n fois pour chacun des n fichiers donnés.
- Il doit y avoir une option `--help` qui affiche la liste des options disponibles et qui fournit une description succinte de l'outil.
- La version de l'outil doit pouvoir être affichée avec l'option `--version`.

Les Liens Le cas des liens est complexe, déjà car il existe deux manières d'écrire un lien en wikicréole étendu :

9. Cela permet de rendre le programme plus simple à maintenir mais aussi plus en accord avec la philosophie Unix. « Make each program do one thing well. »
10. Ceci ne concerne que les liens entre les pages du projet. Les liens vers d'autres sites web sont nécéssairement absolus.
11. Une utilisation idomatique de la ligne de commande des systèmes Unix est de composer différents programmes simples pour réaliser une action complexe. « Expect the output of every program to become the input to another, as yet unknown, program. »

```
>_ Shell script
1 $ ohow --output out.html mydoc.wiki # has the same effet as
2 $ ohow --print mydoc.wiki > out.html
```

Listing 4: Relation entre les options `--print` et `--output` de ohow.

```
λ OCaml code
1 (* Pour le lien [[wiki(eliom):intro|goto intro]], la fonction
  ↪ [href_of_link] est appelée ainsi : *)
2 href_of_link "wiki(eliom):intro" [Tyxml.Html.pcdata "goto intro"]
3 (* Elle retourne alors un élément Tyxml créé comme suit : *)
4 Tyxml.Html.(a ~a:[a_href "../../the/path/to/eliom/intro"] [pcdata
  ↪ "goto intro"])
5 (* Cet élément correspond à la construction HTML suivante : *)
6 (* <a href="../../the/path/to/eliom/intro">goto intro</a> *)
```

Listing 5: Compilation du lien `[[wiki(eliom):intro|goto intro]]`.

— avec la syntaxe wikicréole standard, par exemple `[[wiki(name):page]]` ou encore `[[site:address]]` [12], et,

— avec les extensions Ocsigen `a_manual`, `a_api`, `a_img`, etc.

Le problème est que ces deux types de liens sont traités dans deux parties bien distinctes du compilateur.

Pour la syntaxe standard `[[link]]`, la déduction de l'URL (c'est-à-dire le chemin vers la page de destination) est faite lors de la génération des éléments HTML. Concrètement, il existe une fonction `href_of_link : string -> 'a Html5.elt list -> [> 'a Html_types.a ] Html5.elt` qui, pour chaque lien (syntaxe standard) du wiki, associe l'élément `<a/>` correspondant, c'est-à-dire la balise HTML utilisée pour créer un lien sur une page web (dont le type avec `Tyxml` est `[> 'a Html_types.a ] Html5.elt`). Cette fonction est appelée par le compilateur *directement* et est au coeur du processus de compilation.

Exemple

Le listing 5 montre en quoi la fonction `href_of_link` est responsable d'une déduction de liens correcte et aussi, plus généralement, comment l'on peut générer des morceaux d'HTML en OCaml à l'aide de la bibliothèque Tyxml (appartenant au projet Ocsigen).

En revanche, pour les extensions, le fonctionnement est différent et un exemple d'implantation d'une extension simple est donné par le listing 3. Les extensions qui ont fonction de lien dans la documentation du projet Ocsigen sont les suivantes :

— `a_manual` qui décrit un lien vers un chapitre [13] du manuel,

12. Il existe d'autres syntaxes standard pour les liens, ainsi que de nombreuses abréviations. Pour plus d'informations, consulter l'annexe 5.1 sur le wikicréole.

13. L'on parle de « chapitre » pour désigner les pages web du manuel et de « page » pour désigner les pages web de(s) API(s), mais les deux termes désignent bien tous deux des pages HTML classique.

Attribut	Extensions concernées	Description	Valeur par défaut
`project`	Tous sauf `a_img` et `a_file`	Le projet contenant la page	Projet courant
`chapter`	`a_manual`	Le chapitre du manuel contenant la page	Aucun
`subproject`	`a_api`, `a_api_type`, `a_api_code`	Le sous-projet contenant le symbole	Aucun
`text`	`a_api`, `a_api_type`, `a_api_code`	Texte du lien produit	Le symbole
`version`	Tous sauf `a_img` et `a_file`	La version du projet contenant la page	Version courante
`fragment`	Tous sauf `a_img` et `a_file`	*Fragment* (ou *ancre*) HTML	Aucune
`src`	`a_img`, `a_file`	Le chemin vers la ressource à lier	*Requis*

TABLE 2 – Description des attributs des extensions Ocsigen ayant fonction de lien.

Wikicréole

```
<<a_manual project="eliom" chapter="intro" | introduction of
 ↪  Eliom's docs>>
<<a_manual project="eliom" subproject="ppx" | index.html of the
 ↪  PPX subproject of Eliom>>
<<a_manual version="2.1" | index.html of the current project
 ↪  version 2.1>>
<<a_api project="eliom" subproject="server" text="some text" |
 ↪  type Eliom_client.server_function>>
<<a_api_type project="eliom" subproject="server" | type
 ↪  Eliom_client.server_function>>
<<a_file src="mindmap.pdf">>
<<a_img src="logo.png">>
```

Listing 6: Exemple d'utilisation des liens sous forme d'extensions Ocsigen.

— `a_api`, `a_api_type`, `a_api_code` qui décrivent un lien vers la page de documentation d'un symbole d'une API [14],
— `a_img` qui insère une image (balise HTML `<img/>`) à l'endroit donné, et,
— `a_file` qui décrit un lien vers un fichier quelconque à télécharger, stocké au même endroit que le site généré (GitHub Pages dans le cas du projet Ocsigen).

Le tableau 2 décrit les attributs que ces extensions doivent supporter. Le listing 6 donne un exemple d'utilisation de ces extensions.

Cependant, il existe une différence fondamentale entre les liens syntaxiques `[[link]]` et ces extensions : ces premiers indiquent *où* trouver la ressource liée alors que les extensions indiquent seulement *quelle* ressource lier *sans préciser sa localisation*. Autrement dit, `[[other/page|text]]` peut être lu comme « dans le dossier de la page courante, aller dans le dossier `other`, s'y trouve une page `page` : c'est elle qu'il faut lier » et `<<a_manual project="P" chapter="C"|text>>` comme « *trouver* la page `C` *quelque part* dans le projet `P` (un dossier, lui aussi *quelque part*) et créer un lien vers celle-ci ».

14. *Application Programming Interface* ou « interface de programmation applicative », désigne l'ensemble des *fonctionnalités* (fonctions, classes, types, etc.) que le concepteur d'un service fournit à son utilisateur. L'API constitue le *sous-ensemble* de l'implantation du service qui est *ouvert* à ses utilisateurs et auquel ces derniers sont censés se limiter.

Dans le cas des extensions, le logiciel ne pouvant pas « deviner » où se situe la ressource demandée, il est nécessaire *d'imposer une architecture* que l'utilisateur de celles-ci doit *respecter* afin que leur fonctionnement soit garanti. Concrètement, l'architecture ci-dessous a été choisie, autant pour des raisons techniques imposées par GitHub Pages que pour rendre la compatibilité des URLs plus simple à maintenir.

- Tous les projets (dossiers) concernés par les extensions se situent dans le même dossier (ils sont tous au même niveau). Dans la terminologie employée par ohow, c'est le **projects directory**.
- À la racine de chaque projet se trouve les dossiers de versions, dont le nom correspond à la version concernée, par exemple `1.1`, `2.0.3`, etc. Plus formellement, leur nomenclature est décrite par *l'expression régulière* `([0-9](.[0-9])+|dev)`, où `dev` correspond à la documentation de la version en développement. Dans la terminologie employée par ohow, c'est le **versions directory**.
- Dans ce versions directory doit se trouver un lien symbolique nommé `latest` qui pointe vers le dossier de la version considérée comme « la plus récente ».
- À la racine de chaque version doit se trouver un lien symbolique nommé `index.html` [15] qui pointe vers la page d'introduction ou de présentation du projet (pour la version en question).
- Dans le versions directory doit également se trouver un lien symbolique nommé aussi `index.html` qui pointe vers la page d'accueil du projet par défaut lorsque la version n'est pas spécifiée. `latest/index.html` est une bonne valeur pour ce lien.
- Pour rappel, ohow compile les wikis un par un : il est donc possible de connaître la version de la documentation à laquelle le wiki appartient. Dans la terminologie employée par ohow, c'est le **root directory**.
- Dans l'idée de rester le plus générique possible, ohow accepte les options en ligne de commande suivantes :
  - `--root` désigne le *root directory*,
  - `--manual` désigne le dossier contenant les wikis du manuel (`a_manual`),
  - `--api` désigne le dossier contenant les wikis de l'API (`a_api*` [16]),
  - `--images` désigne le dossier contenant les images (`a_img`), et,
  - `--assets` désigne le dossier contenant les fichiers (`a_file`).

> **Note**
>
> Il n'est pas nécessaire que les valeurs de ces options correspondent à des dossiers réels *au moment de la compilation*. En effet, ces valeur sont seulement utilisées pour déduire les liens relatifs produits par les extensions `a_*`. Naturellement, ces dossiers devront exister une fois le site déployé afin de ne pas avoir de liens morts.

> **Exemple**
>
> Voici une invocation possible de `ohow` :

15. Il y a également un mécanisme (désirable) intégré à GitHub Pages concernant `index.html`. Consulter l'annexe 5.2 sur GitHub Pages pour plus de détail.

16. La notation `foo*` est tirée de la syntaxe shell (plus spécifiquement du *globbing*) afin de désigner tous les fichiers préfixés par `foo`. Cette notation est utilisée ici pour dénoter toutes les extensions préfixées par `a_api`, c'est-à-dire `a_api`, `a_api_code` et `a_api_type`. De la même manière `a_*` désigne toutes les extensions ayant fonction de lien implantées pour le projet Ocsigen.

```
$ ohow --root eliom/4.0.1 \
       --manual eliom/4.0.1/manual/src \
       --api eliom/4.0.1/api \
       --images eliom/4.0.1/manual/src/illustrations \
       --assets eliom/4.0.1/manual/src/illustrations
```

Avec ces informations et contraintes supplémentaires, les extensions `a_*` peuvent déduire correctement chaque lien à générer. Si une de ces extensions est utilisée dans un wiki mais que les options correspondantes ne sont pas renseignées (par exemple si `a_manual` est rencontrée mais que `--manual` ou `--root` n'est pas renseigné), alors **ohow** devra s'arrêter sur une erreur.

L'extension `<<doctree>>` Cette extension affiche le menu de la figure 2. C'est l'une des extensions les plus complexes à gérer pour **ohow** même si son fonctionnement reste relativement simple. Cette extension cherche des fichiers nommés `menu.wiki` dans le manuel et dans l'API (sous-projets inclus), les compile, et les inclus dans un élément HTML `<nav/>` (élément ayant pour sémantique la navigation à travers le site). Le listing 7 montre le fichier `menu.wiki` du manuel du projet Eliom.

Il faut donc maintenir cette extension en état de marche, sans toucher aux fichiers `menu.wiki`. Cette extension n'accepte pas d'attributs ni de contenu.

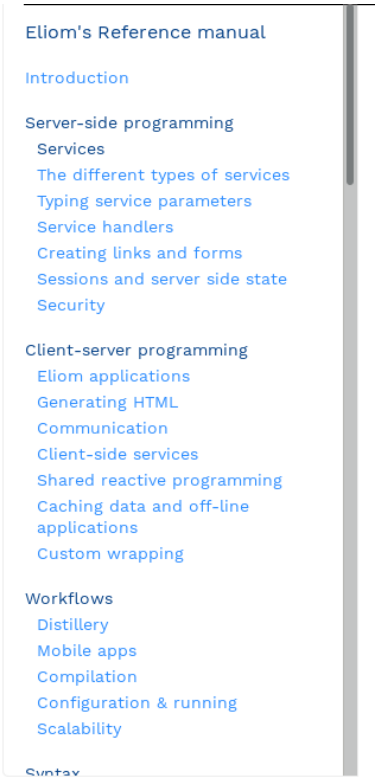


FIGURE 2 – Menu affiché par l'extension `<<doctree>>`.

L'extension `<<docversion>>` Cette extension affiche la liste déroulante de la figure 3. Elle permet de choisir la version de la documentation à afficher. Lorsque l'utilisateur choisit une ver-

Wikicréole

```
= Eliom's Reference manual

==[[intro|Introduction]]

==Server-side programming
===Services
====[[server-services|The different types of services]]
====[[server-params|Typing service parameters]]
====[[server-outputs|Service handlers]]
====[[server-links|Creating links and forms]]
===[[server-state|Sessions and server side state]]
===[[server-security|Security]]

==Client-server programming
===[[clientserver-applications|Eliom applications]]
===[[clientserver-html|Generating HTML]]
===[[clientserver-communication|Communication]]
===[[clientserver-services|Client-side services]]
===[[clientserver-react|Shared reactive programming]]
===[[clientserver-cache|Caching data and off-line applications]]
===[[clientserver-wrapping|Custom wrapping]]

==Workflows
===[[workflow-distillery|Distillery]]
===[[mobile-apps|Mobile apps]]
===[[workflow-compilation|Compilation]]
===[[workflow-configuration|Configuration & running]]
===[[scalability|Scalability]]

==Syntax
===[[ppx-syntax|PPX-based syntax]]
===[[clientserver-language|Camlp4-based syntax]]
===[[ppx-migration|Migration to PPX]]
```

Listing 7: Fichier `menu.wiki` du manuel du projet Eliom.

sion X dans la liste, il doit être redirigé vers la page `X/index.html`.

Exemple

Par exemple, si l'utilisateur est sur la page https://ocsigen.org/eliom/6.3/manual/intro.html et qu'il sélectionne la version **2.1** dans la liste déroulante, il doit être redirigé vers https://ocsigen.org/eliom/2.1/index.html.

Il faut donc maintenir cette extension en état de marche. Cette extension n'accepte pas d'attributs ni de contenu.

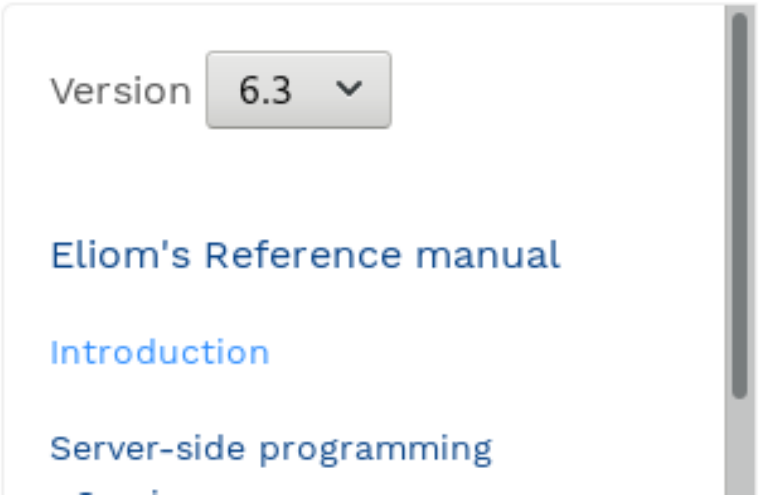


FIGURE 3 – Aperçu de l'extension `<<docversion>>`.

L'extension `<<googlesearch>>` Cette extension affiche une barre de recherche accompagnée d'un bouton permettant d'effectuer une recherche Google sur les pages du site https://ocsigen.org uniquement. Cela permet de naviguer dans la documentation plus rapidement.

Cependant cette barre de recherche n'étant pas stylisée, elle jure avec le reste du site et il a été décidé de la masquer avec le code CSS `display: none` (elle se situe normalement entre le `<<docversion>>` et le `<<doctree>>`).

Il m'a donc été demandé de la styliser et de la placer à un endroit plus approprié. Cette extension n'accepte pas d'attributs ni de contenu.

L'extension `<<client-server-switch>>` Certaines parties de l'API (modules, types, etc.) de certains projets existent en deux versions : une version *client* et une version *serveur*. C'est par exemple le cas pour le module `Eliom_lib` du projet Eliom.

Résumé

Grâce au outils du projet Ocsigen, il est possible d'annoter quel morceau de code est du code client, du code serveur, ou les deux. C'est ensuite à la compilation que la séparation se fait et que les deux binaires (un pour le client, un pour le serveur) sont générés. Certains modules (comme `Eliom_lib`) existent donc en deux versions (puisque les fonctionnalités qu'ils offrent ont une utilité des deux côtés).

C'est pourquoi, sur la page de documentation de ces modules, l'on trouve le composant illustré par la figure 4 et produit par l'extension `<<client-server-switch>>`. Cependant celui-ci est considéré comme peu intuitif, mal stylisé et mal positionné (entre le `<<docversion>>` et le `<<doctree>>`).

Il m'a été demandé de le remplacer, en le rendant plus esthétique, en améliorant sa position dans la page et tout en conservant sa fonction. Cette extension n'accepte pas d'attributs ni de contenu.

FIGURE 4 – L'extension `<<client-server-switch>>` telle qu'elle était à mon arrivée.

L'extension `<<content>>` Cette extension ne se situe pas dans les wikis contenant la documentation mais dans des fichiers wiki spéciaux : les *templates*. Dans ces derniers, elle permet de spécifier l'endroit où insérer le contenu. Plus de détails sur ce mécanisme et sur cette extension sont exposés dans section 3.2.3, où est détaillé le cahier des charges d'un outil auxiliaire de ohow dédié à la gestion de ces templates.

Puisque ce n'est donc pas la responsabilité de ohow que de gérer les templates, l'outil n'a donc pas à gérer cette extension.

> **En bref**
>
> Un template est un fichier wikicréole classique contenant une occurrence de l'extension `<<content>>`. Lorsqu'un wiki classique est compilé, son contenu est « copié-collé à la place » de cette extension dans le template, et le tout est ensuite compilé vers de l'HTML. Cela a pour effet de « rajouter » du code wikicréole avant et après le contenu de chaque wiki, notamment pour des questions de mise en page. C'est aussi dans le template que l'on spécifie l'agencement de la page — la position relative du menu par rapport au contenu.
> Ce mécanisme permet de ne pas avoir à recopier (et maintenir) les parties communes à chaque page de la documentation.

Les autres extensions Il n'est pas nécessaire de réécrire ni même modifier les autres extensions du tableau 1.

Gestion des liens vers les scripts et les feuilles de style Le site https://ocsigen.org utilise des feuilles de style en cascade (Css) ainsi qu'un certain nombre de scripts JavaScript

pour, entre autres, colorer les morceaux de code OCaml que l'on peut trouver dans la documentation.

Le lien vers ces ressources est codé en dur dans **how** (« *hardcoding* »), ce qui constitue une *mauvaise pratique de programmation* puisqu'elle n'est ni générique, ni portable, ni maintenable aisément.

Il me faut donc trouver une solution pour corriger cela. Coder une extension supplémentaire ayant ce rôle qui serait placée dans le template de chaque projet est une solution satisfaisante qui fonctionne.

L'insertion conditionnelle de contenu Les figures 5 et 6 donnent un aperçu global de ce à quoi ressemble (et doit continuer à ressembler) le site https://ocsigen.org, avec deux wikis (contenus) différents. L'on y distingue les différents composants de la page :

- le menu (`<<doctree>>`) de la documentation du projet,
- l'en-tête de la page (balise HTML `<header/>`) avec le logo et les liens vers les principaux projets, identique pour toutes les pages du site,
- le menu latéral rétractable (extension `<<drawer>>`), aussi identique pour toutes les pages du site [17], et enfin,
- la partie centrale, où le contenu du wiki consulté est affiché.

À l'exception du *contenu* de la partie centrale, tout le reste est quasiment identique, *à quelques détails près*. Par exemple dans l'en-tête et le menu latéral, le projet courant est souligné en bleu (coloré en orange dans le menu). Cet effet est accompli en ajoutant une classe Css — `mainmenu-current` et `drawermainmenu-current` — au lien vers le projet courant dans ces menus.

Exemple

Dans le fichier `template.wiki` du projet Eliom, l'on a [a] :

```
*@@class="mainmenu-current"@@[[wiki("eliom"):|Eliom]]
*[[wiki("js_of_ocaml"):|Js_of_ocaml]]
*[[wiki("lwt"):|Lwt]]

[...]

*@@class="drawermainmenu-current
 ↪ drawermainmenu-project"@@[[wiki("eliom"):|Eliom]]
*@@class="drawermainmenu-
 ↪ project"@@[[wiki("js_of_ocaml"):|Js_of_ocaml]]
*@@class="drawermainmenu-
 ↪ project"@@[[wiki("ocsigenserver"):|Server]]
*@@class="drawermainmenu-project"@@[[wiki("lwt"):|Lwt]]
```

alors que dans le fichier `template.wiki` du projet Lwt, l'on a le code suivant, *presque identique* à l'exception de la position des classes `mainmenu-current` et `drawermainmenu-current` :

17. Le site web est dit *responsive*, c'est-à-dire qu'il s'adapte aux différentes résolutions d'appareils (laptop, tablette, smartphone, etc.) et lorsque la largeur est trop faible, le menu de gauche « se déplace » dans le menu droite. Il n'y a en réalité aucun travail supplémentaire à fournir du côté du compilateur de wikicréole, c'est un travail de webdesign essentiellement en Css.

```
*[[wiki("eliom"):|Eliom]]
*[[wiki("js_of_ocaml"):|Js_of_ocaml]]
*@@class="mainmenu-current"@@[[wiki("lwt"):|Lwt]]

[...]

*@@class="drawermainmenu-project"@@[[wiki("eliom"):|Eliom]]
*@@class="drawermainmenu-
 ↪  project"@@[[wiki("js_of_ocaml"):|Js_of_ocaml]]
*@@class="drawermainmenu-
 ↪  project"@@[[wiki("ocsigenserver"):|Server]]
*@@class="drawermainmenu-current
 ↪  drawermainmenu-project"@@[[wiki("lwt"):|Lwt]]
```

a. La syntaxe `@@class="cls"@@` permet d'ajouter une classe HTML à l'élément suivant. Voir l'annexe 5.1 sur le wikicréole pour plus d'informations.

La majorité du code est donc *dupliquée* dans les différents fichiers `template.wiki` de tous les projets et les seules différences se situent au niveau de *détails* comme les classes Css évoquées précédemment. La duplication de code étant une mauvaise pratique de programmation, il m'a été demandé de trouver une solution afin de n'avoir *qu'un seul fichier* `template.wiki` *pour tous les projets* et de trouver un moyen d'ajouter cette classe automatiquement lors de la compilation.

Ceci peut être accompli en implantant une extension `when-project`, qui n'insère son contenu que si le wiki en train d'être compilé appartient au projet spécifié.

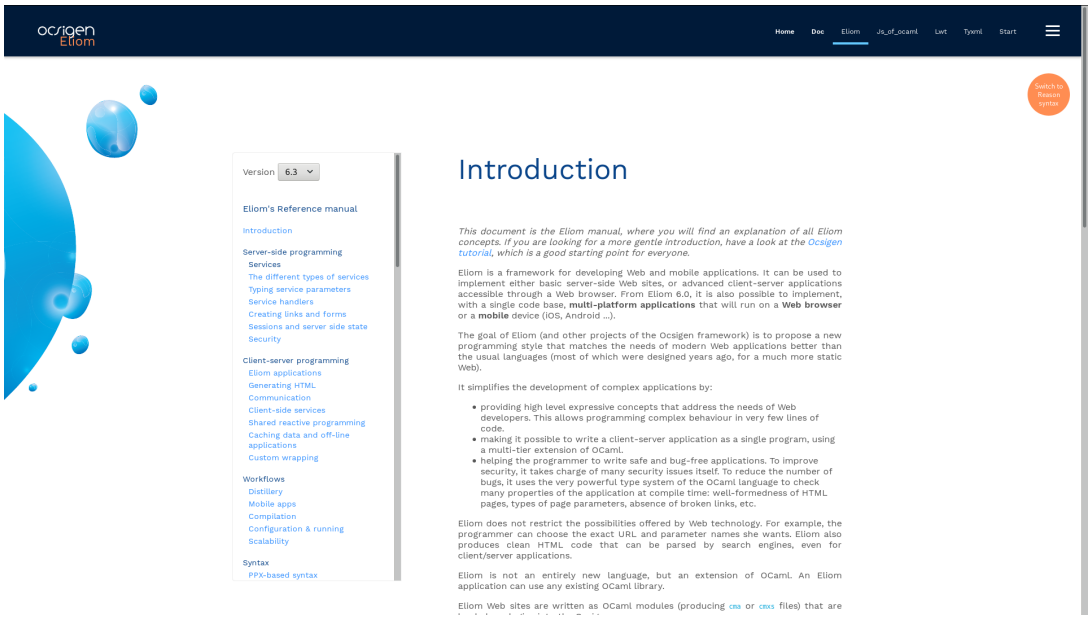


FIGURE 5 – Aperçu du site https://ocsigen.org (page d'accueil du projet Eliom).

Documentation Que ohow soit destiné à un usage interne uniquement ou à être publié, il est impératif de le documenter. De plus, étant un outil de gestion de documentation, il est

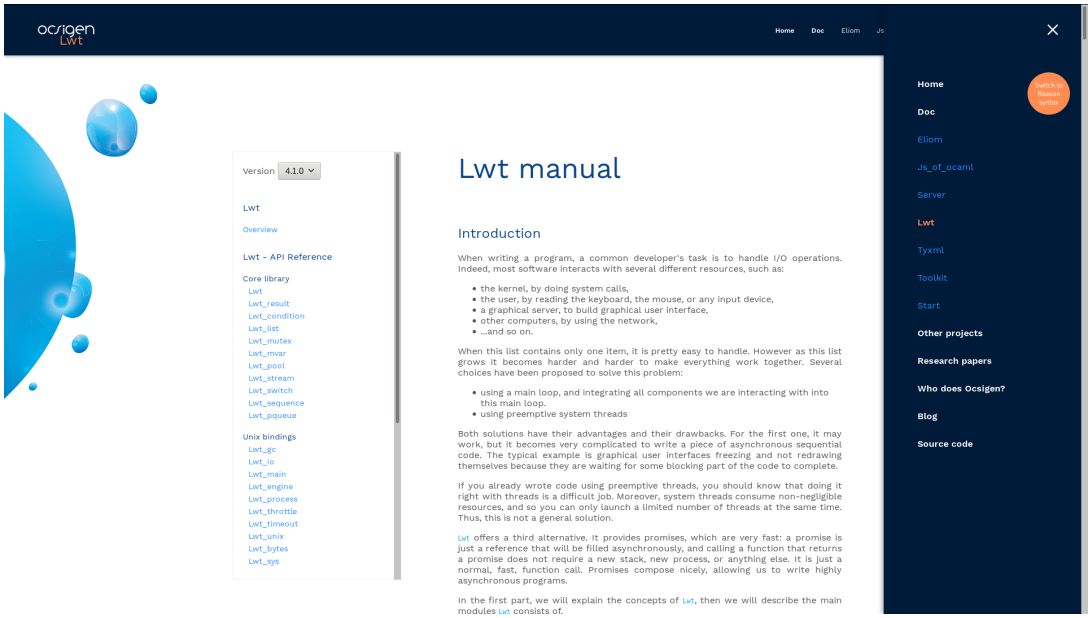


FIGURE 6 – Aperçu du site https://ocsigen.org (page d'accueil du projet Lwt).

nécessaire que sa documentation soit parfaitement claire, précise et soignée. Cela permet aussi de démontrer l'efficacité de l'outil s'il est capable générer sa propre documentation.

Il faut donc écrire la documentation de **ohow** en wikicréole étendu et la compiler avec ohow.

3.2.3 L'outil auxiliaire de ohow : wiki_in_template

Comme expliqué dans le paragraphe sur l'extension `<<content>>` de la section 3.2.2, il doit être possible d'inclure le contenu de chaque wiki dans un fichier spécial : le *template* (ou *patron*). Ces fichiers sont différents dans le sens où ils contiennent une (ou plusieurs) occurrence(s) de cette extension.

Cependant, toujours afin de rester dans l'idée que chaque programme ne doit faire qu'une seule chose[9], il a été décidé que la fonction d'insertion de *contenu* dans un template doit être le but d'un outil distinct. En effet, **ohow** est un *compilateur wikicréole → HTML* : celui-ci prend donc en entrée « du code wikicréole » et sort « du code HTML équivalent », alors que la fonction qui « *inline* » (insère) un wiki dans son template est une fonction qui prend en entrée « deux codes wikicréole » et sort « un code wikicréole (le second dans le premier) ».

Ce sont donc deux fonctionnalités *complètement orthogonales* (c'est-à-dire que l'on peut très bien utiliser l'une sans l'autre) pour deux outils *bien distincts*. L'outil donc chargé *d'inline* un wiki dans un template est donc *wiki_in_template*[18], abrégé `wit`.

Objectifs

- `wit` doit être un outil en ligne de commande Unix.
- Il doit permettre d'insérer le contenu *d'un* wiki dans *un* template.
- Aucun de ces deux fichiers ne doit être modifié.
- Le contenu du wiki est inséré à la place de la *première balise* `<<content>>`.

18. Une légère adaptation de la convention de nommage `A_of_B` d'OCaml (« wiki_of_template » n'aurait pas eu beaucoup de sens puisque les deux sont paramètres de l'outil).

```shell
>_ Shell script
cat content.wiki | wit t1.wiki | wit t2.wiki
```

Listing 8: Commande d'insertion d'un wiki dans deux templates.

```shell
>_ Shell script
wit <(wit template.wiki < c1.wiki) < c2.wiki
```

Listing 9: Commande d'insertion de deux contenus dans un seul template.

— Cette extension n'accepte pas d'attributs ni de contenu.

Interface

— La commande **wit** prend en seul argument (requis) un *fichier* : le template.
— Elle lit le contenu du wiki à insérer sur *l'entrée standard* `stdin`.
— Le résultat de l'insertion est envoyé vers *la sortie standard* `stdout`.

* * *

Cette interface simple faisant une utilisation massive des flux d'entrée et de sortie standard permet à **wit** d'être *extrêmement composable* et ainsi permettre l'utilisation de *plusieurs templates* pouvant contenir chacun *plusieurs balises* `<<content>>`.

En effet, pour deux templates `t1.wiki` et `t2.wiki` (contenant chacun une seule balise `<<content>>`) et un wiki `content.wiki`, insérer ce dernier dans le premier template puis dans le second revient à exécuter le code shell du listing 8. Cette commande produit le résultat escompté sur `stdout` (que l'on peut éventuellement rediriger facilement vers un fichier de sortie).

C'est avec la même simplicité que l'on peut gérer le cas de l'insertion de *deux contenus* `c1.wiki` et `c2.wiki` dans *un seul* template `template.wiki` (contenant au moins deux balises `<<content>>`), comme l'illustre le listing 9 [19].

Enfin, les deux cas traités précédemment fonctionnent une fois généralisés à plus de deux templates et deux wikis, et peuvent être combinés, puisque la sortie de **wit** peut à la fois avoir fonction de template que de contenu.

> **Note**
> Il est important de comprendre que cette puissance provient de la combinaison de la simplicité de la conception de **wit**, de son adéquation avec la philosophie d'Unix et de la puissance du langage shell.

Exemples

— Le listing 10 montre le résultat de l'insertion d'un wiki dans deux templates.
— Le listing 11 montre le résultat de l'insertion de deux wikis dans un seul template.

19. Afin de simplifier l'écriture de celle-ci la commande du listing 9 utilise une fonctionnalité *non-Posix* : les *process substitutions*. Il est néanmoins possible d'obtenir le même effet avec plusieurs commandes et l'utilisation de fichiers temporaires.

```
>_ Shell script

bash$ ls
bottom.wiki hello.wiki top.wiki
bash$ cat hello.wiki
= Hello world!
This is the content wiki.
bash$ cat top.wiki
Template above.
<<content>>
bash$ cat bottom.wiki
<<content>>
Template below.
bash$
bash$ cat hello.wiki | wit top.wiki | wit bottom.wiki
Template above.
= Hello world!
This is the content wiki.
Template below.
bash$
```

Listing 10: Exemple d'insertion d'un wiki dans deux templates.

```
>_ Shell script

bash$ ls
title.wiki text.wiki template.wiki
bash$ cat title.wiki
= Hello world!
bash$ cat text.wiki
Some text.
bash$ cat template.wiki
Before first.
<<content>>
In between.
<<content>>
After.
bash$
bash$ wit <(wit template.wiki < title.wiki) < text.wiki
Before first.
= Hello world!
In between.
Some text.
After.
bash$
```

Listing 11: Exemple d'insertion de deux wikis dans un template.

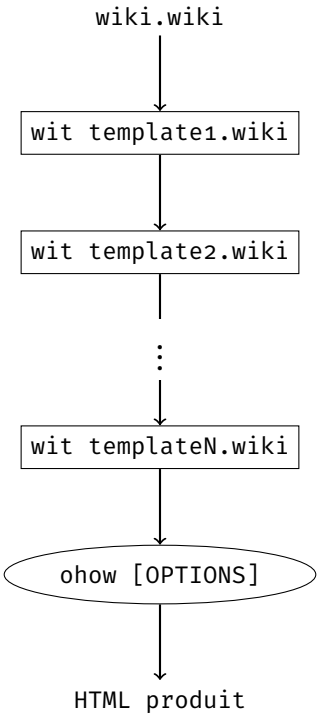


FIGURE 7 – *Pipeline* de compilation d'un wiki avec templates.

La figure 7 schématise la *pipeline* (chaîne de processus) mise en place pour compiler un contenu wikicréole avec ses différents templates (encore une fois, leur nombre n'est évidemment pas limité à 2).

3.2.4 doc_of_project

Si `ohow` permet bien de compiler un fichier wikicréole en HTML, il est, par design, incapable de générer la documentation d'un projet en entier. Il est donc nécessaire d'écrire un outil supplémentaire, formant une couche d'abstraction autour de one_html_of_wiki, chargé de cela. Nous avons nommé cet outil *dop*, pour « doc_of_project ».

`dop` n'apporte pas de nouvelles fonctionnalités mais permet d'utiliser `ohow` plus simplement. Cet outil permet d'améliorer *l'ergonomie* et le *confort d'utilisation*.

Interface

- Outil en ligne de commande.
- Accepte un *fichier de configuration*, au format JSON, supportant les paramètres décrits par le tableau 3.
- En particulier, `dop` doit être capable de *déduire*, en analysant la structure du projet, un maximum de paramètres de configuration.
- Accepte les options en ligne de commandes décrites par le tableau 4.

> **Note**
>
> Pour des raison techniques, le format JSON requiert qu'un outil externe, `jq`, soit installé. Comme il n'est pas toujours possible de l'installer et afin d'assurer une portabilité maximale, `dop` supporte

Entrée JSON	Type	Option de `ohow` correspondante
`project`	*string*	`--project`
`manual`	*string*	`--manual`
`api`	*string*	`--api`
`client`	*string*	—
`server`	*string*	—
`assets`	*string*	`--assets`
`images`	*string*	`--images`
`csw`	*boolean*	`--csw`
`menu`	*boolean*	`--doctree`
`templates`	*string array*	`--template`
`default_subproject`	*string*	`--default-subproject`

TABLE 3 – Entrées supportées par le fichier de configuration géré par `dop`.

JavaScript

```
{
    "project": "html_of_wiki",
    "manual": "man",
    "api": "api",
    "assets": "files",
    "images": "files/images",
    "csw": false,
    "menu": true,
    "templates": ["template1.wiki", "template2.wiki"]
}
```

Listing 12: Exemple de fichier de configuration de `dop`.

aussi un autre format pour le fichier de configuration, nommé **`plain`**. Voir la documentation correspondante, disponible à l'adresse suivante : https://ocsigen.org/html_of_wiki/2.0/manual/intro#conf.

Exemples Le listing 12 donne un exemple de fichier de configuration. Supposons que ce fichier est nommé `how.json` et qu'il se situe au même niveau qu'un dossier `2.0`, contenant le manuel et l'API de la version correspondante du projet `html_of_wiki` (cf. `"project"`). Pour le compiler dans son intégralité, l'on invoque `dop` de la manière suivante :

```
$ dop -r 2.0-compiled -t json -c how.json -viulf 2.0/
```

3.2.5 quickdop

Si `dop` permet de simplifier considérablement l'utilisation de `ohow`, il reste néanmoins un paramètre à fournir manuellement et qui pourrait être automatisé : `--docversion`. Ce para-

Nom(anglais)	Option	Valeur	Valeur par défaut	Description
Config	`-c`	Fichier	—	Fichier de configuration
Config type	`-t`	`json` ou `plain`	`plain`	Format de ce fichier
Clean	`-k`	Présence	—	Conserver les fichiers `*.wiki`
Inferred	`-i`	Présence	—	Afficher la configuration déduite
Used	`-u`	Présence	—	Afficher la configuration utilisée
No run	`-n`	Présence	—	Ne pas compiler (*dry run*)
Root dir	`-r`	Nom	`_dop`	*Root directory*
Force	`-f`	Présence	—	Réécrit ce dossier s'il existe
Local	`-l`	Présence	—	Liens locaux (cf. `--local`)
Docversion	`-d`	Fichier	—	Fichier `docversion` (cf. `--docversion`)
Verbose	`-v`	Présence	—	Mode verbeux
Help	`-h`	Présence	—	Affiche l'aide et termine

TABLE 4 – Options de ligne de commande que doit supporter `dop`.

mètre, comme expliqué précédemment, prend comme valeur le chemin vers un fichier contenant les noms des versions de la documentation à compiler, une par ligne.

C'est l'objectif de `quickdop` (« quick-doc_of_project »). Cet outil en ligne de commande s'invoque de la manière suivante :

```
$ quickdop [-f] PROJECT OUTDIR [DOP_OPTIONS]
```

`PROJECT` le dossier contenant les versions à compiler

`OUTDIR` le dossier devant contenir le résultat de la compilation (utiliser l'option `-f` pour autoriser la réécriture)

`DOP_OPTIONS` les options à donner à `dop`, à l'exception de `-d` et `-r`

Lorsque `quickdop` est ainsi invoqué, l'outil va se charger d'analyser le contenu de `PROJECT` et appeler `dop` pour tous les dossier dont le nom est décrit par l'expression régulière `([0-9.]+){,3}|dev`. Il se charge également de générer le fichier passé à `--docversion` avec le nom des versions qu'il compile, ainsi que de les ordonner (versions par ordre numérique et `dev`, si elle existe, à la fin).

3.2.6 Gestion de la documentation du project Ocsigen

Comme indiqué dans la section 3.1, il m'a été demandé de régler deux problèmes distincts :
— celui de remplacer how par un outil plus souple et,
— *simplifier* et *automatiser* la gestion de la documentation des projets.

Simplification
— fusion de tous les fichiers `template.wiki` (voir la section 3.2.2),
— suppression du dépôt `ocsigen.org-data` (voir la section 3.1), et,
— gestion des liens symboliques faisant office de redirections[20] automatique.

20. Voir l'annexe 5.2 pour plus d'informations.

Automatisation Il m'a également été demandé de mettre en place un *job d'intégration continue* avec Travis CI. Ce job doit être composé des *phases* suivantes :

Build il faut tout d'abord compiler les wikis :
- insérer tous les wikis dans leur(s) template(s) (dans le bon ordre),
- compiler toutes les versions du projet,
- lancer un *outil de détection de liens morts* sur les pages HTML ainsi générées.

Post-build tests après la phase précédente, il faut vérifier que :
- toutes les compilations ont terminées avec succès (pas d'erreur de syntaxe, d'utilisation de fonctionnalités dépréciées, etc.),
- l'outil de détection de liens morts n'en a trouvé aucun.

Deploy enfin, si les tests post-compilation sont tous passés, il faut déployer le site généré sur GitHub Pages, c'est-à-dire en poussant (`git push`) les fichiers HTML sur la branche `gh-pages` du projet[20].

3.3 Réalisation

Dans cette section est détaillé l'ensemble de mes travaux durant le stage, ainsi que l'étude de leur adéquation avec le cahier des charges décrit dans la section précédente. La figure 8 schématise le déroulé de mon stage.

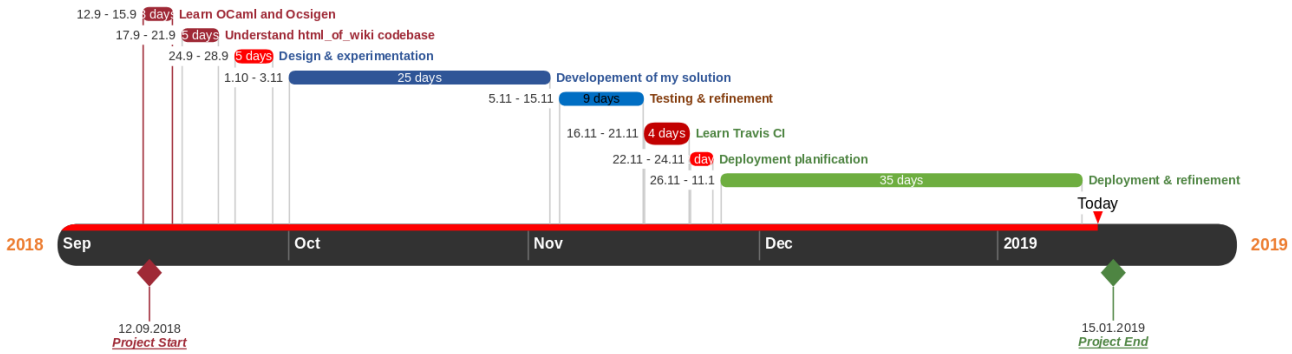


FIGURE 8 – Diagramme de Gantt du déroulé du stage

3.3.1 Développement

Une fois le besoin établi comme précédemment, le développement s'est réalisé sans réels problèmes et assez rapidement. Il serait trop long et fastidieux de détailler dans le détail les problèmes techniques et algorithmiques rencontrés durant cette phase. Il est cependant important de noter certains points et de tirer les conclusions suivantes de cette période.

Premièrement, le développement en OCaml est extrêmement confortable (en comparaison avec des langages plus classiques comme le C ou le Java). Son système de types fort et très expressif permet, grâce à la vérification de types intégrée au compilateur, de vérifier la cohérence du programme et ainsi détecter de nombreuses erreurs dès la compilation, au lieu de

ne les subir qu'à l'exécution, comme dans d'autres langages, moins fortement typés (Python, JavaScript, PHP, etc.)[21].

Deuxièmement, ayant travaillé avant ce stage avec le langage Haskell, un autre langage fonctionnel populaire, de nombreuses différences se sont vite fait voir.

— *La gestion des effets de bord.* En effet, Haskell n'autorise aucun effet de bord (affectation de variable). Pour palier à cela, le programmeur peut utiliser une structure mathématique appelée *monade* afin de simuler ces effets de bord dans l'écriture du code, mais que l'implantation de ces effets restent fonctionnels purs. Cependant, ce concept est difficile à comprendre et à expliquer, et repousse chez beaucoup la volonté d'apprendre le Haskell. OCaml autorise les effets de bords et les monades sont beaucoup moins présentes, ce qui le rend *nettement plus accessible*.

— *Le pragmatisme.* OCaml est un langage beaucoup plus pragmatique que Haskell (qui lui est en revanche mathématiquement, théoriquement plus élégant). OCaml supporte la mutabilité (mot clef `mutable`), les arguments par défaut, les arguments optionnels, la programmation orientée objet dans son sens le plus populaire (Haskell a des *classes de types*, moins répandues), les modules de premier ordre, des opérations d'entrée-sortie classique (contrairement à Haskell qui requiert l'utilisation d'une monade, `IO`), etc.

J'ai pu utiliser certains projets du projet Ocsigen, notamment Lwt et Tyxml. Lwt permet l'écriture de code *asynchrone* grâce à l'utilisation de *promesses* (promises en anglais) et Tyxml permet d'écrire du HTML directement en OCaml. Les différents éléments HTML sont notamment transposés dans le système de types d'OCaml afin de vérifier statiquement à la compilation la conformité des pages web ainsi construites avec le standard W3C. Par exemple, mettre deux `<body/>` dans un `<html/>` ne passe pas la vérification de type car le standard le ne permet pas. Le paragraphe suivant explique succinctement comment utiliser Lwt pour écrire du code OCaml asynchrone.

Par exemple, supposons que la fonction `get_users` retourne une liste d'utilisateurs (de type `user list`) qu'elle cherche dans une base de données sur le réseau. Cette fonction risque donc de prendre beaucoup de temps pour finir son exécution. Grâce à Lwt, au lieu de retourner une liste d'utilisateurs au bout de plusieurs secondes, elle retourne une *promesse de liste d'utilisateurs* (de type `user list Lwt.t`) *immédiatement*. À ce moment-là, la liste d'utilisateurs n'est pas encore construite, donc sa valeur n'est pas accessible. En revanche, l'on peut indiquer à la promesse quoi faire avec cette liste lorsqu'elle sera construite (quelque secondes plus tard), grâce à la fonction `Lwt.bind`. Cette fonction retourne elle-même une promesse, qui ne sera tenue que lorsque la promesse de `get_users` sera elle-même tenue. Le code du listing 13 décrit comment récupérer la promesse de liste d'utilisateurs, `users_p`, récupère leur noms lorsqu'elle est tenue et en fait une autre promesse, `names_p`, qui est immédiatement tenue (grâce à `Lwt.return`). Enfin les noms sont tous affichés. Seule la première promesse (`users_p`, retournée par `get_users`) met du temps à être tenue. Les promesses sont un excellent moyen d'organiser du code concurrent par nature. En effet, pendant que le processus associé à `get_users` travaille, le programme OCaml peut continuer à s'exécuter, et chaîner (avec `Lwt.bind`) les exécutions du programme. Les deux processus s'exécutent donc en même temps. Ce type de paradigme de programmation est particulièrement efficace pour le web, qui requiert par nature beaucoup d'acquisition de ressources. Cela permet à la fois au serveur et au client d'exécuter le maximum de code possible, jusqu'à ce que l'acquisition de la ressource devienne bloquante. (Le

21. Ces langages disposent souvent d'outils d'analyse de code (les *linters*), effectuant des vérification de types et de bonnes pratiques. Néanmoins, aussi aboutis soient-ils, ces outils ne remplaceront jamais les fortes garanties intégrées à OCaml, au coeur du langage par conception.

λ OCaml code

```
val fetch_and_print_users : unit -> unit Lwt.t
let fetch_and_print_users () =
  let users_p = get_users () in
  let names_p = Lwt.bind users_p (fun users ->
      let names = List.map user_name users in
      Lwt.return names)
  in
  Lwt.bind names_p (fun names ->
      Lwt.return (List.iter print names))
```

Listing 13: Exemple de code asynchrone avec Lwt

λ OCaml code

```
let fetch_and_print_users () =
  let%lwt users = get_users () in
  let%lwt names = Lwt.return (List.map user_name users) in
  (* Ou plus simplement:
     let names = List.map user_name users in
  *)
  Lwt.return (List.iter print names))
```

Listing 14: Listing 13 réécrit grâce aux extensions de syntaxe `ppx`

code du listing 13 peut également s'écrire comme dans le listing 14 grâce aux extensions de syntaxe `ppx`.)

Finalement, il m'a été possible d'utiliser certaines fonctionnalités avancées d'OCaml tels que les foncteurs de modules afin d'écrire du code générique facilement et élégamment.

La partie développement de la solution a été assez rapide, intéressante et amusante (par rapport à la partie déploiement). Il m'a cependant fallu l'écourter un peu artificiellement. En effet, une fois le développement de `ohow` quasiment terminé, il restait à faire `dop` et `quickdop`. Or vu qu'il m'aurait fallu encore un certain temps pour les faire correctement en OCaml, ces outils ont été codés en shell. Ces scripts restent temporaires (dans l'idée). (En ce qui concerne la détection de lien morts, j'ai également développé un outil en Common Lisp pour réécrire les logs de `linkchecker` [22] au format JSON, afin de les lire facilement sur un afficheur de tableaux, tel que http://json2table.com/.)

3.3.2 Déploiement avec Travis CI

Comme me le répétait quelques fois mon maître de stage :

> « *En informatique, on fait les 80% du travail les premiers 20% du temps. Pour 80% des 20% de travail restants, cela nous prend 20% des 80% de temps restants. Et ainsi de suite pour les 20% restants des 20%. On ne finit jamais un travail, on ne fait que converger…* »

Cela correspond très exactement à mon expérience avec le déploiement de html_of_wiki. Il ne m'a fallu qu'autour d'une semaine pour apprendre et expérimenter avec Travis CI et

22. https://wummel.github.io/linkchecker/

```yaml
language: python
python:
    - "2.7"
    - "3.5"
    - "3.6"
script:
    - pytest
```

Listing 15: Exemple de fichier `.travis.yml`

pour concevoir et planifier le déploiement de la nouvelle infrastructure de documentation. En revanche, il m'a fallu environ un mois et demi pour régler les différents détails propres à chaque projet et pour que cela convienne à peu près aux différents mainteneurs.

L'intégration continue avec Travis CI Travis CI est un service d'intégration continue (CI/CD) pour GitHub. Pour chaque dépôt pour lequel Travis CI est activé, il faut y installer, à la racine, un fichier nommé `.travis.yml`, au format YAML, contenant la configuration pour la CI pour cette branche git. Le listing 15 contient un exemple de configuration de Travis CI pour un projet python. À chaque commit, Travis CI démarrera trois containers, un pour Python 2.7, un pour Python 3.5 et un pour Python 3.6. Il effectuera ensuite dans chacun d'eux la même action, c'est-à-dire exécuter le script `pytest` qui lance les tests unitaires. Si l'un d'entre eux, dans n'importe laquelle des trois versions de Python utilisées, échoue, le script `pytest` échoue aussi faisant ainsi échouer à son tour la CI. Le commit est alors marqué comme contenant une erreur et un mail est envoyé à son auteur pour l'informer de l'erreur. Dans le cas contraire où tous les tests passent pour toutes les versions, la CI passe et le commit est marqué comme valide.

Reconduire à chaque commit ces tests permet à la fois de s'assurer continuellement de la *qualité* du code, mais aussi de prévenir la *régression* en la détectant le plus tôt possible.

Travis CI pour html_of_wiki Dans un premier temps, il avait été convenu avec mon maître de stage de déplacer la documentation de chaque projet sur la branche `master`. Or, après discussion avec le mainteneur d'un des projets (Tyxml), l'on a du changer de modèle. En effet ce dernier ne voulait pas imposer à ses utilisateurs voulant basculer sur la branche `master` la copie de plusieurs dizaines de kilo octets de documentation (pour la plupart générée automatiquement qui plus est).

Le second modèle adopté a été de faire partir du branche `wikidoc` à partir de `master` pour y entreposer la documentation. Cela a plusieurs avantages :

1. La documentation n'est plus sur `master`.
2. Il n'y a pas besoin de modifier le `.travis.yml` de chaque projet pour y intégrer la génération de la documentation. Une seule configuration générique peut être copiée telle quelle (quasiment) dans chaque projet.
3. Il n'y a pas le surcoût de compilation des wikis (parfois atteignant une vingtaine de minutes) à chaque commit du projet. Cela permet de réduire considérablement le temps d'exécution de chaque *build*.

Il m'a alors fallu recommencer ce travail de déploiement pour les projets traités avant Tyxml.

Enfin, il m'a été suggéré de ne pas faire partir la branche **wikidoc** de **master** mais d'en faire une *branche orpheline*, c'est-à-dire une branche sans historique (comme **master** après un `git init`). Cela se fait avec la commande suivante :

```
1 | git checkout --orphan wikidoc
```

Deux phases de déploiement Le déploiement a dû être réalisé en deux étapes à cause de problèmes techniques. En effet, il n'est pas possible de déployer html_of_wiki sur chaque projet en même temps car il faut que chaque mainteneur *review* le code. Et cela peut prendre du temps… De plus, en cas de problème mineur, il vaut mieux éviter que le site https://ocsigen.org soit inaccessible.

Cependant, le code Css du site a été modifié par mes soins, le rendant incompatible avec le site généré par html_of_wiki 1.0. J'ai en effet ajouté et refait le design de certains composants de la page :

— la barre de recherche qui se déploie lorsque sélectionnée (figure 9) ;
— le *switch* client/serveur, à la iOS (figure 10).

De plus les templates ont été centralisés dans le dépôt **ocsigen.github.io**.

Ainsi, histoire de conserver l'état de l'ancien site, mais de commencer le déploiement d'une partie du nouveau, il a été nécessaire de passer par l'utilisation de scripts et feuilles de style temporaires. C'est dans un second temps seulement, que ces fichiers temporaires ont étés supprimés. Le listing 16 donne un exemple de fichier `.travis.yml` dans sa version finale (projet js_of_ocaml). Le script temporaire y a été remplacé par le script `how.sh`, factorisé et téléchargé par chaque *build* pour éviter la duplication de code. La seule réelle nouveauté par rapport à l'exemple du listing 15 est la partie `deploy` en fin de fichier. Lorsque le `script` a fonctionné (compilation des wikis sans erreurs), Travis CI déploie les fichiers générés sur GitHub Pages (`provider`) et commit automatiquement sur la branche **gh-pages** du projet. Il ne fait cela que lorsque le commit traité se situe directement sur la branche wikidoc (`on: branch: wikidoc`) afin d'éviter de déployer des wikis encore en développement (dans une *Pull Request* par exemple).

> **Note**
> À cause d'un bug dans Travis CI, celui-ci ne peut pas déployer les liens symboliques crées par un des processus de la CI. Cela pose un problèmes pour la génération automatique de redirections. Afin de contourner le problème, des redirections HTTP ont été utilisées avec l'aide de la balise HTML : `<meta http-equiv="refresh" content="0; URL=X.html" />`. Cf. https://github.com/travis-ci/dpl/issues/912.

FIGURE 9 – La barre de recherche après mes modifications

Conclusion du déploiement Cette phase de mon stage a été fastidieuse et répétitive. J'ai néanmoins pu découvrir comment fonctionne et mettre en place un environnement d'intégration continue (l'une de mes attentes du stage).

Tous les mainteneurs n'étaient pas satisfaits de l'infrastructure mise en place. Néanmoins, cela constitue un net progrès en termes de flexibilité et d'utilisation par rapport à l'infrastructure existante à mon arrivée.

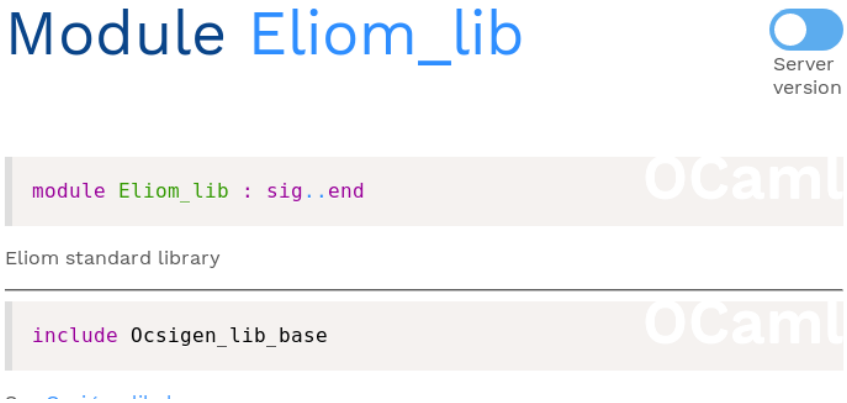


FIGURE 10 – Le *switch* client/serveur après mes modifications

YAML code

```yaml
language: c

env: HOW_DOC=doc HOW_CONFIG=how.json HOW_OUT=_doc

install:
    - wget https://ocsigen.org/how.sh
    - . how.sh
    - how-install

script:
    - how-generate
    - how-redirect "$HOW_LATEST/manual/overview"
    ↪  $HOW_OUT/index.html
    - how-redirect-manual 1.4 1.0.9
    - how-redirect-manual 1.4 1.1.1
    - how-redirect-manual 1.4 1.2
    - how-redirect-manual 1.4 1.3
    - how-redirect-manual 2.4.1 2.0
    - how-redirect-manual 2.4.1 2.1
    - how-redirect-manual 2.4.1 2.2
    - how-redirect-manual 2.4.1 2.3
    - how-redirect-manual 2.4.1 2.4

deploy:
  provider: pages
  keep-history: true
  skip-cleanup: true
  github-token: $GHP_TOKEN
  local-dir: _doc
  on:
    branch: wikidoc
```

Listing 16: Fichier `.travis.yml` final pour js_of_ocaml

3.3.3 Documentation

Il a fallu écrire un certain nombre de documentations, disponibles sur le site https://ocsigen.org :

- la documentation de html_of_wiki (voir l'annexe 5.3);
- un wiki pour l'installation du projet Ocsigen (voir la figure 11);
- un wiki pour le processus de contribution vis-à-vis de la documentation (voir la figure 12).

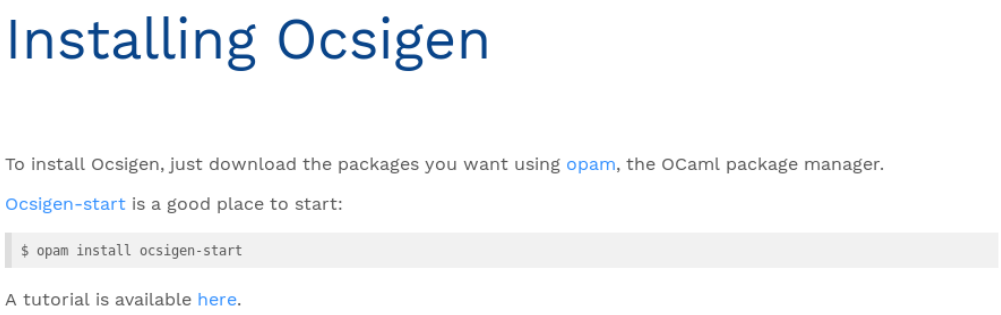


FIGURE 11 – https://ocsigen.org/install

3.4 Conformité au cahier des charges

Ma solution est conforme au cahier des charges et répond au besoin exprimé. Le seul point non implanté est la vérification automatique de lien morts dans la CI, par manque de temps. Heureusement, l'absence de cette fonctionnalité n'impacte pas la validité de l'infrastructure mise en place et la validité des résultats.

Contributing to Ocsigen

For individual contributers

To contribute to an Ocsigen project, first see the guidelines it provides. It will give you the directives on how to make your contribution. Once ready, create a *Pull Request* on the repository and wait for feedback!

Every contribution has to be documented! The documentation is stored on a branch `wikidoc`. For your contribution to be accepted, you also need to make a PR on that branch for your documentation. On your forked branch, feel free to push and the CI will tell you if your documentation has errors. It will only be deployed to https://ocsigen.org when merged on `wikidoc`.

The documentation format is the wikicréole. Please find the reference here. You can generate the documentation of your API directly in Ocsigen's wikicréole using `ocamldoc`. Do not hesitate to ask for help if you experience any trouble.

Remember, you have to issue **2** pull requests—one for your code on `master` and one for its documentation on `wikidoc`—for your contribution to be accepted!

Create a release (for package maintainers)

1. Make sure the documentation on `wikidoc` is **up to date** with the code on `master`.
2. On `wikidoc`, `cp -r dev X.Y.Z`, where `X.Y.Z` is the new version's number.
3. On `master`, update opam configuration file.
4. `git tag`
5. Create a release on GitHub.
6. `opam publish` and wait for the package to be accepted.

Also do not forget to push on `wikidoc` as it will update the website automatically for the new version.

Local documentation generation

html_of_wiki can (to some extent) locally generate your documentation. Please find its manual here. Each project, on the `wikidoc` branch, contains a file `how.json`. Please ensure that its configuration is correct for your changes. The following steps require you to be at *the root of the project*.

- Install html_of_wiki on your computer. Please follow the instructions provided on the project's homepage.
- Copy the `template` directory from the repository ocsigen.github.io and rename it `how_template`. You can use the following command:

```
tmp=$(mktemp -d) && \
git clone --depth 1 https://github.com/ocsigen/ocsigen.github.io.git $tmp && \
mv $tmp/{template,how_template} && \
mv $tmp/how_template .
```

- Use the following command to generate the documentation of the project, where `DOC` is the directory containing the documentation:

```
quickdop -f DOCS _doc -t json -c how.json -viul
```

If you want to avoid generating the whole documentation—just some parts of it—read the documentation of the tools **dop** and **ohow** in the manual of html_of_wiki.

FIGURE 12 – https://ocsigen.org/contributing

4 Conclusion du stage

Pour résumer en quelques phrases, à mon arrivée chez BeSport, la refonte de l'outil de documentation du projet Ocsigen m'a été confiée. L'outil précédemment utilisé (html_of_wiki 1.0) était lourd, monolithique, peu flexible et ne permettait pas au différents mainteneurs de gérer leur documentation comme ils le souhaitent.

Les premiers jours étaient dédiés à la prise en main le site https://ocsigen.org et à l'expérimentation avec les différents projets composant le projet Ocsigen. Par la suite, il a fallu comprendre la codebase de html_of_wiki, dont certaines parties ont plus de dix ans. La première attente de mon maître de stage était la compréhension rapide et globale de ce code.

Une fois cela terminé, il a été nécessaire de réfléchir pour concevoir correctement ce que serait *html_of_wiki 2.0*, ma contribution au projet Ocsigen. Il a fallu correctement détecter, analyser et comprendre l'architecture déjà en place et concevoir une solution aux différents problèmes, parmi lesquels :

— la rigidité ;
— la complexité ;
— le manque de généricité ;
— le manque d'automatisation.

* * *

La solution trouvée se divise en trois outils, complémentaires. `ohow`, un compilateur de wikicréole (format utilisé pour la documentation) générique, sans artifices, en accord avec la philosophie Unix. Cet outil, développé en OCaml, un langage français puissant, expressif, abouti et élégant, constitue le coeur de la solution. Ce compilateur peut, entre autres, compiler chaque wiki de la documentation d'un projet indépendamment, sans connaissance des autres pages. Cela permet d'isoler la génération de la documentation de chaque version de chaque projet, ce qui est essentiel pour garantir la modularité de la solution.

Autour de `ohow` s'articulent deux autres outils : `dop` et `quickdop`, tout deux écrits en shell (principalement pour accélérer leur développement). Ce premier permet de générer la documentation pour une version d'un projet, faisant le lien entre le manuel, la référence de l'API et les fichiers statiques automatiquement. `quickdop`, très similaire à `dop`, permet lui de faire le lien entre les différentes versions d'un projet donné.

Le développement de ces outils s'est fait assez rapidement et sans réel heurt. Cette partie de mon stage a correspondu parfaitement à mes attentes : l'utilisation d'OCaml et la possibilité d'approfondir son usage par l'exploitation de fonctionnalités avancées.

* * *

La seconde partie du stage était le déploiement de ma solution pour remplacer l'infrastructure de documentation existante. Ce déploiement avait deux objectifs.

1. Rendre chaque projet Ocsigen responsable de sa documentation.
2. Automatiser la génération de la documentation au travers de l'utilisation de Travis CI. Ce service d'intégration continue (CI/CD) permet d'exécuter du code, automatiquement, dans des containers sur des machines distantes, à chaque commit sur le projet. Cela permet par exemple de prévenir la régression en testant continuellement.

Il a donc fallu concevoir et appliquer un plan de déploiement, en tenant compte des différentes contraintes techniques (liées à GitHub par exemple), des détails propres à chaque projet et à leur documentation et des préférences des différents mainteneurs. À chaque commit sur une branche spéciale, crée et dédiée à la documentation, html_of_wiki est utilisé dans la *build* de Travis CI pour compiler la documentation. Si cette étape est un succès, alors

la documentation compilée est déployée sur GitHub Pages, mettant de ce fait à jour le site https://ocsigen.org, mais seulement la partie propre au projet en question, tout cela automatiquement.

Cette seconde partie du stage était fastidieuse et répétitive, car l'installation de Travis CI devait être répétée manuellement pour chaque projet. Néanmoins j'ai pu apprendre comment mettre en place un processus d'intégration continue et avoir une micro-introduction au métier de DevOp grâce aux différentes ressources en ligne que j'ai pu lire.

* * *

Au final, le cahier des charges a quasiment été respecté, autant pour la partie développement que déploiement. En conséquence, l'infrastructure de gestion de la documentation du projet Ocsigen a été mise à jour à 100% et est fonctionnelle. Ainsi, les différents mainteneurs et développeurs qui travaillent pour le projet Ocsigen pourront plus simplement déployer leur documentation en ligne pour les utilisateurs. BeSport étant un utilisateur massif du projet Ocsigen, cela devrait indirectement mais positivement impacter la confort des développeurs de la startup. De plus, l'amélioration de certaines fonctionnalités du site, demandées par ses utilisateurs, comme une barre de recherche, ont été ajoutées.

Pour conclure, je suis satisfait de mon expérience chez BeSport, qui répondait à toutes les attentes que j'avais de ce stage.

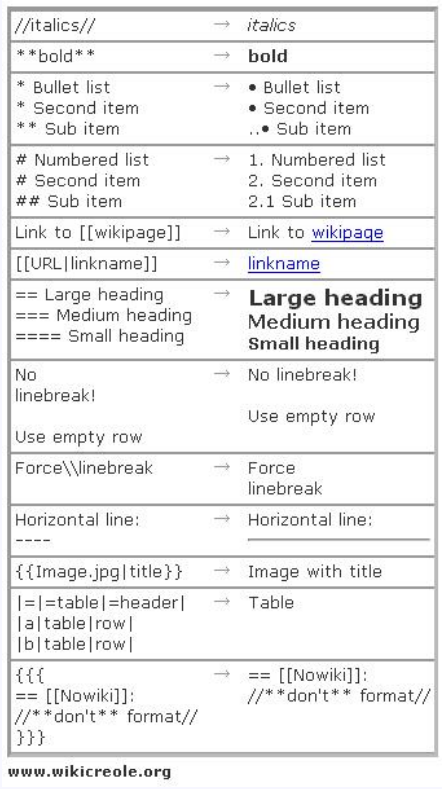
5 Annexes

5.1 Le Wikicréole

Le wikicréole est un format de mise en forme de texte de la famille des formats wiki. Sa dernière version en date est la `1.0`.

5.1.1 Standard

La figure 13 résume le standard wikicréole. Noter que cette image omet de parler du caractère d'échappement `~` qui permet de ne pas interpréter l'élément qu'il précède.



//italics//	→	*italics*
bold	→	**bold**
* Bullet list * Second item ** Sub item	→	• Bullet list • Second item ..• Sub item
# Numbered list # Second item ## Sub item	→	1. Numbered list 2. Second item 2.1 Sub item
Link to [[wikipage]]	→	Link to wikipage
[[URL|linkname]]	→	linkname
== Large heading === Medium heading ==== Small heading	→	**Large heading** **Medium heading** **Small heading**
No linebreak! Use empty row	→	No linebreak! Use empty row
Force\\linebreak	→	Force linebreak
Horizontal line: ----	→	Horizontal line: ———
{{Image.jpg|title}}	→	Image with title
|=|=table|=header| |a|table|row| |b|table|row|	→	Table
{{{ == [[Nowiki]]: //**don't** format// }}}	→	== [[Nowiki]]: //**don't** format//

www.wikicreole.org

FIGURE 13 – Résumé du standard wikicréole `1.0`

> **Note**
> Dans le wikicréole supporté par le projet Ocsigen, les titres de premier niveau sont préfixés par un seul caractère `=`. Un tel titre s'écrit alors `= Premier niveau (h1)` et un titre de niveau inférieur s'écrit `== Second niveau (h2)`, `=== Troisième niveau (h3)`, etc.

5.1.2 Additions du projet Ocsigen

Liens Le tableau 5 liste les syntaxes de liens ajoutées et le tableau 6 les abréviations disponibles.

Syntaxe	Description
`[[wiki("name"):page]]`	Lien vers la `page` du wiki `name`.
`[[wiki:page]]`	Lien vers la `page` du wiki courant.
`[[site:href]]`	Lien relatif à la racine du site.
`[[href:path]]`	Valeur de l'attribut `href` du lien explicite.

TABLE 5 – Syntaxes de liens ajoutées par le projet Ocsigen.

Abréviation	Équivalence	Description
`[[]]`	`[[href:.]]`	Page courante.
`[[#anchor]]`	`[[href:#anchor]]`	Page courante (avec ancre).
`[[/path]]`	`[[site:path]]`	Lien relatif à la racine du site.
`[[path]]`	`[[href:path]]`	Lien relatif.

TABLE 6 – Syntaxes d'abréviations de liens ajoutées par le projet Ocsigen.

Note

Une ancienne syntaxe dont BeSport souhaitait se débarrasser permettait d'identifier un wiki avec un identifiant unique : `[[wiki(id):page]]`. Cette syntaxe étant assez obscure pour le lecteur (`wiki(12)` est moins explicite que `wiki("eliom")`), il a été décidé qu'elle ne serait plus maintenue.

Décorations Le tableau 7 liste les décorations ajoutées par le projet Ocsigen. Est aussi ajouté un autre type de listes : les *listes de définitions*. Le code wikicréole du listing 17 peut ainsi générer la liste de définitions suivante :

A définition de A
B définition de B
C définition de C

Extensions Comme expliqué dans la section 3.2.2, le wikicréole Ocsigen inclut un puissant système d'extensions avec la syntaxe suivante : `<<ext attr1="val1" attr2="val2" | content>>`,

Wikicréole

```
;A
:définition de A
;B
:définition de B
;C
:définition de C
```

Listing 17: Une liste de définitions en wikicréole (non-standard).

Syntaxe	Description	Exemple
`--`	*en-dash*	–
`---`	*em-dash*	—
`##text##`	Police à chasse fixe	`text`
`^^super^^`	Exposant	super
`„sub„`	Indice	sub
`__text__`	Soulignement	text
`/-text-/`	Barrage	~~text~~

TABLE 7 – Décorations ajoutées au wikicréole par le projet Ocsigen.

où le contenu `content` est analysé récursivement et les valeurs `valN` non.

Classes HTML *inline* Il est enfin possible d'ajouter des classes HTML à des éléments individuellement. Par exemple, `@@class="title"@@== Level 2 heading` génère `<h2 class="title">Level 2 heading</h2>`.

5.2 GitHub Pages

Consulter la documentation de GitHub Pages à l'adresse https://pages.github.com/ pour plus d'informations sur ce mécanisme.

5.3 Documentation de html_of_wiki

Les pages suivantes, jusqu'à la fin de ce rapport, sont une copie de la documentation de html_of_wiki, en ligne à l'adresse https://ocsigen.org/html_of_wiki, que j'ai écrite.

Switch to Reason syntax

html_of_wiki

> *`html_of_wiki` is a versatile, minimalist yet powerful **static website generator**. It is designed with simplicity in mind: no special directories, almost no configuration and everything working out of the box. No web server required, just the compilation of your content to HTML and you're free to choose which static files server service fits you: GitHub Pages, GitLab Pages, etc.*
>
> *It allows writing website content in a feature-rich and extensible language: the **wikicréole**. This language, thanks to its clever design, allows to write rich content as much expressively as with plain HTML.*
>
> *`html_of_wiki` is part of the Ocsigen project and is used to generate its documentation—including this page! It has no problem dealing with large websites, it can handle several versions of several projects.*
>
> *`html_of_wiki` is also composable, it can be integrated into a CI/CD process so you can automate the deployment of your website. Rebuild and deploy on each commit!*

Installation

Clone this repository:

```
$ git clone https://github.com/ocsigen/html_of_wiki.git
```

then, use `opam pin` to install it:

```
$ opam pin add -y html_of_wiki html_of_wiki
```

What you will learn here

The following sections describe:

1. The wikicréole format
2. How to use `ohow`, a wikicréole compiler to HTML
3. How to use
    1. `dop` for generating the documentation of a whole project's version, and
    2. `quickdop` for generating the documentation of a whole project's.
4. The available extensions

The wikicréole

The standard

The following picture summarises the wikicréole 1.0 format.

//italics//	→	*italics*
bold	→	**bold**
* Bullet list * Second item ** Sub item	→	• Bullet list • Second item ..• Sub item
# Numbered list # Second item ## Sub item	→	1. Numbered list 2. Second item 2.1 Sub item
Link to [[wikipage]]	→	Link to wikipage
[[URL|linkname]]	→	linkname
== Large heading === Medium heading ==== Small heading	→	**Large heading** **Medium heading** **Small heading**
No linebreak! Use empty row	→	No linebreak! Use empty row
Force\\linebreak	→	Force linebreak
Horizontal line: ----	→	Horizontal line: ----
{{Image.jpg|title}}	→	Image with title
|=|=table|=header| |a|table|row| |b|table|row|	→	Table
{{{ == [[Nowiki]]: //**don't** format// }}}	→	== [[Nowiki]]: //**don't** format//

www.wikicreole.org

Notes:

- The wikicréole supported by `html_of_wiki` uses only one `=` sign for toplevel headings.
- There is also the tilde character `~` for not interpreting the character it prefixes.

Syntax additions

Supported links syntaxes

The following table lists additional link syntaxes supported by the tool.

Syntax	Description
`[[wiki("name"):page]]`	Link to the `page` of the project `name`
`[[wiki:page]]`	Link to the `page` of the current project
`[[site:page]]`	Website root-relative link
`[[href:path]]`	Raw `href` value

The following table lists available links abbreviations.

Abbreviation	Equivalent syntax	Description
`[[]]`	`[[href:.]]`	Current page
`[[#anchor]]`	`[[href:#anchor]]`	Current page with anchor
`[[/path]]`	`[[site:path]]`	Website root-relative link
`[[path]]`	`[[href:path]]`	Relative link

Decorations

The following table lists the additional available decorations.

Syntax	Description	Example
`--`	*en-dash*	–
`---`	*em-dash*	—
`##text##`	Mono-spaced font	`text`
`^^super^^`	Superscript	text^{super}
`,,sub,,`	Subscript	text$_{sub}$
`__text__`	Underline	text
`/-text-/`	Strike-through	~~text~~

Definition lists

The following code produces a definition list.

```
;title1
:definition1
;title2
:definition2
```

title1
: definition1

title2
: definition2

Inline HTML classes

It is possible to inline some HTML classes to the following element using the syntax `@@class="title"@@`.

Extensions

The wikicréole supported by `html_of_wiki` support **extensions**, a powerful mechanism that allows executing arbitrary OCaml code registered to the parser. More detail about this mechanism can be found in the last section of this document.

Extensions syntax: `<<extension attr1="val1" attr2="val2" ... argN="valN"|content>>`

Here are some common, widely used by Ocsigen's documentation extensions:

`<<a_manual>>`
: Link to a page of the manual of a project.

`<<span>>`
: Inserts an HTML `<span/>` element allowing a fine control over the DOM of the page.

`<<doctree>>`
: Displays a menu like the one on the left of this text.

How to use ohow (one_html_of_wiki)

one_html_of_wiki is a CLI tool for generating a *single* HTML document from a *single* wikicréole file.

Basic usage

```
$ ohow file.wiki # generates file.html
$ ohow -o somename.html file.wiki # generates somename.html
$ ohow --help # shows help
$ ohow --version # shows version
$ ohow --print file.wiki # prints the HTML to stdout instead of writing a file
$ ohow --headless file.wiki # do not include HTML head nor wrapping body tag
$ ohow --local file.wiki # generate local-navigation compatible links (REMOVE THAT FOR DEPLOYMENT)
```

Advanced usage

Templates

one_html_of_wiki supports the use of *templates*. A template is a classic wikicréole file which contains a (unique occurrence of) `<<content>>` somewhere. It is where the content will be inserted. Use the `--template` option to provide a path to the template.

Consider the following example for a deeper understanding on how to use templates:

```
$ cat template.wiki
<<head-css|
    .red {color: red}
    .blue {color: blue}
>>
@@class="red"@@Before.
<<div class="around"|
    <<content>>
>>
@@class="red"@@After.
$
$ cat page.wiki
I'm a wiki page, providing some <<span class="blue"|**useful**>> content!
$
$ ohow --template template.wiki --print page.wiki
<html><head><title></title><meta charset="utf8"/><style>
    .red {color: red}
    .blue {color: blue}
</style></head><body><p class="red">Before.
</p><div class="around"><p>I'm a wiki page, providing some
<span class="blue"><strong>useful</strong></span> content!</p></div>
<p class="red">After.</p></body></html>
$
```

There is also an extension, `<<include>>`, that allows a programmatic content insertion. This extension accepts two, *mutually exclusive*, attributes:

`template="PATH"`
: inserts the content of the file `$(dirname T)/PATH`, where `T` is the value of the *required* option `--template`.

`wiki="PATH"`

inserts the content of the file `$(dirname CF)/PATH`, where `CF` is the path to the currently compiled wiki file.

Notes:

- The option `--template` is only required if the `template` is used somewhere.
- This extension ignores its content.

wiki_in_template

If the template has to contain more than one `<<content>>` tag or if several templates have to be used, the integration of `wit` to your workflow has to be considered. *wiki_in_template* (`wit`) is a simple CLI tool for inserting content inside templates without converting them into HTML: it only deals with wikicréole.

Here's an example of how to use it to insert one page inside several templates:

```
$ ls
bottom.wiki hello.wiki top.wiki
$ cat hello.wiki
=Hello world!
This is the content wiki.
$ cat top.wiki
Template above.
<<content>>
$ cat bottom.wiki
<<content>>
Template below.
$
$ cat hello.wiki | wit top.wiki | wit bottom.wiki
Template above.
=Hello world!
This is the content wiki.
Template below.
$
```

And an example of how to insert several pages inside one template:

```
$ ls
title.wiki text.wiki template.wiki
$ cat title.wiki
=Hello world!
$ cat text.wiki
Some text.
$ cat template.wiki
Before first.
<<content>>
In between.
<<content>>
After.
$
$ wit <(wit template.wiki < title.wiki) < text.wiki
Before first.
=Hello world!
In between.
Some text.
After.
$
```

Finally, as far as `wit` is concerned, a page is a wiki file without `<<content>>` and a template is a wiki file with at least one occurrence of `<<content>>`—so there's nothing wrong in saying that `wit` can output a template.

A few notes:

- You cannot escape the string `<<content>>` or `<<content|>>` using the three curly braces syntax.
- You cannot write a comment with the string `<<content>>` or `<<content|>>` inside.
- The `<<include>>` extension is currently not supported by `wit`. (PR welcomed!)

Link extensions

There are several link extensions that ship with `ohow`. They differ from the classic link syntax because they do not explicitly state where is located the resource to link. For example, `<<a_manual project="eliom" chapter="intro" | introduction of Eliom's docs>>` leads to the chapter `intro` of the `eliom` project *without explicitly giving the location of that page*.

Here are the available extensions:

- `<<a_manual>>`
- `<<a_api>>`, `<<a_api_type>>` and `<<a_api_code>>`
- `<<a_img>>`
- `<<a_file>>`

and the attributes they accept:

Attribute	Extensions	Description	Default value
`project`	All except `a_img` and `a_file`	The project of the page	Current project
`chapter`	`a_manual`	The manual chapter to link	None
`subproject`	`a_api`, `a_api_type`, `a_api_code`	The targeted sub-project	None
`text`	`a_api`, `a_api_type`, `a_api_code`	Text of the produced link	What's documented
`version`	All except `a_img` and `a_file`	The version of the project containing the page	`latest`
`fragment`			

	All except `a_img` and `a_file`	HTML fragment	None
`src`	`a_img`, `a_file`	The path to the resource to link	*Required*

However, even if the extensions doesn't require the *writer* to explicitly give the linked resource's location, the compiler still needs to know where to find it. It is why, whenever any of these extensions is used inside a wiki document, the writer must call `ohow` with the correct values for these options: `--manual`, `--api`, `--images`, `--assets` and `--root` (the latter defaults to the current working directory).

The *root* directory is the directory containing all wiki content for a specific project's version. For example, a project `proj/`, with two versions, `1.0/` and `2.0/`, each one containing the `man/`, `api/` and `assets/` directories. `ohow` must then be called with the following options (`pwd = ~/user/proj/`):

- `--root 2.0/`
- `--manual man/` — The path is relative to the *root* directory (not the current directory!). You could still have used `--manual 2.0/man/` (the root prefix is automatically stripped away).
- `--api 2.0/api/` — or equivalently, `--api api`
- `--assets assets`
- (`--images assets` — Only if you used `a_img` and that the images are in the `assets/` directory.)

For inter-project links, `ohow` **supposes that every projects are inside the same directory**. For example, `<<a_manual project="eliom" chapter="intro">>` will produce the following link: `rewinding to the root directory/../../eliom/path to the manual directory given with --manual/intro`. The first `../` rewinds from the root directory into the project directory (containing all versions). The second `../` rewinds inside the directory containing all the projects. It could, in practice, look like `../../../../eliom/latest/man/intro`.

html_of_wiki imposes these constraints for links because these are the constraints GitHub Pages imposes. To conclude, setting up these extensions is not as complicated as it sounds (it's roughly passing some paths to `ohow` through command line arguments) and the benefits are huge: this way you abstract away any hierarchy between wiki pages and projects and let the compiler work out the links by itself. Thus, any structural change of your projects **will not** force you to rewrite your documentation!

The `<<doctree>>` extension

`<<doctree>>` is an extension without parameters that inserts a menu. That menu is built by concatenating the content of all the files named `menu.wiki` inside the *root* directory. The order is:

1. the `menu.wiki` of the manual first, if any
2. the `menu.wiki` files of the API, if any, alphabetically sorted by sub-project's name, if any

For example, for a project with a manual and two API sub-projects, the extension will look for the following files, in that order:

1. `manual/menu.wiki`
2. `api/menu.wiki`
3. `api/sub-project1/menu.wiki`
4. `api/sub-project2/menu.wiki`

The menu on the left of this text is generated using that extension. Here is an example of what one can put inside a `menu.wiki` file:

```
=##html_of_wiki##
==[[#title|Introduction]]
=The wikicréole
==[[#wikistd|Standard]]
==[[#wikiadd|Additions]]
=[[#ohow|##ohow##]]
```

Again, that extension requires some extra parameters—`--root`, `--manual` and `--api`—which are described above. The generated element is a `<nav/>` with `class="how-doctree"`.

The `<<docversion>>` extension

`<<docversion>>` is an extension which inserts a dropdown list that lets the visitor select the version of the documentation to see. An example of such a widget on the left of this text.

To use it, place the versions to display in a file, say `versions.txt`, one per line, in the expected order. Then, pass that file as an argument of `ohow` using the option `--docversion`. If `--root`'s directory name is inside `versions.txt`, that entry will be automatically selected.

The extension places a text—`"Version"`—and a `<select/>` element with `class="how-versions"`. Required options are `--root` and `--docversion`.

The `<<client-server-switch>>` extension

WARNING: This extension is specific to the Ocsigen project.

To include a client/server switch, include `<<client-server-switch>>` without attributes somewhere in the template. The data of this extension is automatically collected by either dop or quickdop. At a lower level, this extension requires a parameter—`--csw`—which expects a file. Inside must figure the name a all wiki files (with extension `.wiki`), one per line, that should have a client/server switch. dop uses the Unix builtin `comm -12` to extract the files the sub-projects `client` and `server` have in common.

Legacy support

- In previous versions of `html_of_wiki`, projects used to have a file—`config.js`—which declaratively described these projects. Amongst that information, a field `default_subproject` declared the `<<a_api* subproject` attribute's default value. Since that file no longer exist, an option `--default-subproject` exists but is **highly deprecated**.
- That previous version also supported a special link syntax—`[[wiki(id):page]]`—in which projects were given an ID (in `config.js`). Considered unclear, that syntax is now **deprecated**, any wiki using it will *not* compile.

Higher level documentation generators

one_html_of_wiki works great for compiling one wiki page, but a project's documentation is often composed of many wiki pages for each version. However, `ohow` is designed with the Unix philosophy in mind: **simplicity** and **composability**. This, it is very easy to integrate it in an existing workflow.

To make things even easier, we provide out of the box two higher level documentation generators:

- *doc_of_project* (`dop`) — for generating a single version of a project
- *quick-doc_of_project* (`quickdop`) — for generating the documentation of a project with many versions

These tools are described in detail in the following sections.

dop

It is a wrapper on `ohow` designed to generate the document of a single version of a project with a one-line command.

Configuration file

Even if all configuration data could be passed using CLI options—as `ohow` does—it is often more convenient to have a configuration file. Two formats are accepted: *JSON* and *plain text*. Here is an example of the same configuration file, written in these two formats:

config.json:

```
{
  "project": "html_of_wiki",
  "manual": "man",
  "api": "api",
  "assets": "files",
  "images": "files/images",
  "csw": false,
  "menu": true,
  "templates": ["template1.wiki", "template2.wiki"]
}
```

config.txt:

```
project   html_of_wiki
manual    man
api       api
assets    files
images    files/images
csw       false
menu      true
templates template1.wiki:template2.wiki
```

Note that the JSON format needs **jq** (https://stedolan.github.io/jq/) installed and in the `PATH`. Use the plain format if `jq` cannot be installed. It also can be useful (thanks to its minimalism) if it needs to be algorithmically generated. It is read by `awk` with default settings: `$0` should contain the key, and the other records its value. Arrays have their value separated by a colon `:` (in a `PATH`-like fashion).

Before reading the configuration file you provide (if any), `dop` will try to **infer it** by analyzing the *root* directory. If the option `-i` is given, it will print that inferred configuration (either in JSON or in plain text). Then, if a configuration file is explicitly provided, its keys replaces the inferred ones and the ones *not* present keep their inferred values. First check what `dop` can infer for your organization's architecture from the doc's version its given (use the `-n` option for not generating the docs and just do the inferring work).

The following table describes the entries recognized by `dop`:

Key name	Type	`ohow` option	Short description
`project`	string	`--project`	Project's name
`manual`	string	`--manual`	Manual directory (root-relative path)
`api`	string	`--api`	API directory (root-relative path)
`client`	string	—	API's client directory (real path)
`server`	string	—	API's server directory (real path)
`assets`	string	`--assets`	Assets directory (root-relative path)
`images`	string	`--images`	Images directory (root-relative path)
`csw`	boolean	`--csw`	Contains a client-server-switch?

`menu`	boolean	`--doctree`	Contains a doctree?
`templates`	string array	`--template`	List of templates
`default_subproject`	string	`--default-subproject`	*Deprecated*

The `templates` key is an array because it would be possible (thanks to `wit`) to have several templates but this feature is not currently implemented in `dop`. PR welcomed! ;-)

Command-line interface

The command `dop` accepts the following short options:

Name	Option	Value	Default value	Description
Config	`-c`	File	—	Configuration file
Config type	`-t`	`json` or `plain`	`plain`	Configuration file type (format)
Clean	`-k`	Flag	—	Do not clean wikis after compilation
Inferred	`-i`	Flag	—	Print inferred configuration
Used	`-u`	Flag	—	Print used (final) configuration
No run	`-n`	Flag	—	Dry run (no compilation)
Root dir	`-r`	Name	`_dop`	Root directory (output directory)
Force	`-f`	Flag	—	Removes root dir if exists
Local	`-l`	Flag	—	Generate local links (`--local`)
Docversion	`-d`	File	—	`docversion` files (see `--docversion` option)
Verbose	`-v`	Flag	—	Verbose
Help	`-h`	Flag	—	Show help and exit

Examples

Minimal example

```
$ tree
.
├── 1.0
│   └── manual
│       └── intro.wiki
└── 2.0
    ├── api
    └── manual
        ├── intro.wiki
        └── other.wiki
$ dop 2.0
$ tree
.
├── 1.0
│   └── manual
│       └── intro.wiki
├── 2.0
│   ├── api
│   └── manual
│       ├── intro.wiki
│       └── other.wiki
└── _dop
    ├── api
    └── manual
        ├── intro.html
        └── other.html
```

More examples

You can look at the files called `.howdocgen` inside the `wikidoc` branch of each documented Ocsigen project for a real demonstration.

`quickdop`

To make things easier for projects with multiple versions of the documentation, html_of_wiki provides `quickdop`. It takes care of generating the docs for version found (directory name matching `[0-9]+|dev`) and provides automatically a value for the `--docversion` option.

Usage

```
quickdop [-f] PROJECT OUTDIR [DOP_OPTIONS]
```

`PROJECT`
: The directory containing the versions.

`OUTDIR`
: The build directory. Automatically created and replaced when the `-f` option is provided.

`DOP_OPTIONS`
: The options usually passed to `dop`. Obviously, the options `-r`, `-d` and the target are provided by `quickdop` and must not be explicitly given.

Built-in wikicréole extensions

`html_of_wiki` is shipped with a number of built-in, general purpose, wikicréole extensions. The documentation of the extensions `doctree`, `docversion`, `client-server-switch`, `a_manual`, `a_api`, `a_api_type`, `a_api_code`, `a_img` and `a_file` can be found in earlier sections of this document.

Each extension documented below accept the attributes `class` and `id` for explicitly giving an HTML class and an HTML id to the generated code.

Nameless extension

Acts as a comment, i.e., inserts no HTML. For example: `<<|I'm a comment: I will not appear inside the generated HTML>>`.

`code` and `code-inline`

Attributes
`language="X"`, `translated="translated"` (optional, for `X` = `"ocaml"` only)
Description
Inserts a code block.
HTML element
`<pre class="manually-translated"><code class="language-X"></code></pre>` or `<pre><code class="language-ocaml translatable"></code></pre>`
Example

```
<<code language="ocaml"|int_of_string "12" + 3>>
```

produces

```
int_of_string "12" + 3
```

The `code-inline` extension is similar but inserts a code snippet inside the current paragraph and not in a block of its own.

`div`, `span`, `nav`, `pre`

Inserts the corresponding HTML elements.

`googlesearch`

Attributes
`domain="DOMAIN"`, `icon="ICON"`
Description
Inserts a search bar with icon `ICON` performing a Google search restricted to the specified domain `DOMAIN`.
Tyxml element

```
Html.[form ~a:[a_id "googlesearch";
                a_action "https://google.com/search"]
          [input ~a:[a_name "q";
                     a_id "gsearch-box";
                     a_placeholder "Search using Google"]
            ();
           label ~a:[a_label_for "gsearch-box"]
             [img ~src:ICON ~alt:"" ~a:[a_id "gsearch-icon"] ()];
           input ~a:[a_input_type `Submit;
                     a_id "gsearch-submit";
                     a_onclick @@ "document.getElementById('gsearch-box').value +=
' site:" ^ DOMAIN ^ "';"]
             ()]]
```

Example

```
<<googlesearch domain="ocsigen.org" icon="search.svg">>
```

`wip` and `wip-inline`

Inserts the content of extension in a `<div/>` element with the class `wip`. You may want to add in your CSS the following rule:

```
.wip {display: none;}
```

`wip-inline` is the inline counterpart of `wip`.

`when-local` and `unless-local`

`when-local` inserts its content when `ohow` (or any higher level documentation generator) is called with the `--local` (or `-l`) option. `unless-local` inserts its content when not. This extension is useful for addressing complex resource linking issues.

`when-project`

Attributes
: `when="P"` or (**exclusive**) `unless="P"`

Description
: Inserts the content when (or unless) the project the wiki file belongs to is `P` (see the configuration key `project`).

`script`

Attributes
: `src="JS_SRC"`

Description
: Inserts a `<script>` tag importing the given `JS_SRC` JavaScript source file *inside the `<body>` of the document*.

`head-script`

Attributes
: `src="JS_SRC"`

Description
: Inserts a `<script>` tag importing the given `JS_SRC` JavaScript source file *inside the `<head>` of the document*.

`head-css`

Attributes
: `href="CSS_SRC"`

Description
: Inserts a `<link rel="stylesheet">` tag importing the given `CSS_SRC` CSS stylesheet *inside the `<head>` of the document*.

Table des figures

1 Architecture nécessaire pour faire fonctionner html_of_wiki. 9
2 Menu affiché par l'extension `<<doctree>>`. 17
3 Aperçu de l'extension `<<docversion>>`. 19
4 L'extension `<<client-server-switch>>` telle qu'elle était à mon arrivée. 20
5 Aperçu du site https://ocsigen.org (page d'accueil du projet Eliom). 22
6 Aperçu du site https://ocsigen.org (page d'accueil du projet Lwt). 23
7 *Pipeline* de compilation d'un wiki avec templates. 26
8 Diagramme de Gantt du déroulé du stage 29
9 La barre de recherche après mes modifications 33
10 Le *switch* client/serveur après mes modifications 34
11 https://ocsigen.org/install 35
12 https://ocsigen.org/contributing 36
13 Résumé du standard wikicréole `1.0` 39

Liste des tableaux

1 Extensions Ocsigen que **ohow** doit impérativement supporter. 12
2 Description des attributs des extensions Ocsigen ayant fonction de lien. 15
3 Entrées supportées par le fichier de configuration géré par `dop`. 27
4 Options de ligne de commande que doit supporter `dop`. 28
5 Syntaxes de liens ajoutées par le projet Ocsigen. 40
6 Syntaxes d'abréviations de liens ajoutées par le projet Ocsigen. 40
7 Décorations ajoutées au wikicréole par le projet Ocsigen. 41

List of Listings

1 Exemple de texte formatté en wikicréole. 6
2 Le code HTML du wiki du listing 1 produit par one_html_of_wiki. 9
3 Code OCaml nécessaire pour ajouter l'extension `colored`. 11
4 Relation entre les options `--print` et `--output` de **ohow**. 14
5 Compilation du lien `[[wiki(eliom):intro|goto intro]]`. 14
6 Exemple d'utilisation des liens sous forme d'extensions Ocsigen. 15
7 Fichier `menu.wiki` du manuel du projet Eliom. 18
8 Commande d'insertion d'un wiki dans deux templates. 24
9 Commande d'insertion de deux contenus dans un seul template. 24
10 Exemple d'insertion d'un wiki dans deux templates. 25
11 Exemple d'insertion de deux wikis dans un template. 25
12 Exemple de fichier de configuration de `dop`. 27
13 Exemple de code asynchrone avec Lwt 31
14 Listing 13 réécrit grâce aux extensions de syntaxe `ppx` 31
15 Exemple de fichier `.travis.yml` 32
16 Fichier `.travis.yml` final pour js_of_ocaml 34
17 Une liste de définitions en wikicréole (non-standard). 40