

Documentation et modélisation pour les architectures orientée service

AXEL MOUSSET

Université de Technologie de Troyes

Printemps 2016

Résumé

Modéliser et documenter un logiciel constituent un travail essentiel pour la maintenance et l'évolution de projet logiciel sur le long terme. Au delà des document de conception initiaux qui constituent un pré-requis de tout projet d'ingénierie, un cas extrême de ce besoin de documentation est notamment fourni par le développement de logiciel open-source qui ajoute la contrainte de rendre disponible le code source et d'ouvrir à la participation d'autres développeurs au delà de l'équipe projet.

La modélisation UML centrale pour le paradigme objet, les wiki, et framework de test de logiciel sont des ressources importantes pour développer une documentation claire et exploitable. Cependant, avec le développement d'application logiciels toujours plus complexes et distribuées désormais permises par l'Internet, aucune stratégie de documentation claire et spécifique n'émerge pour le développement des applications orientées services.

L'enjeu de ce document est de proposer et de mettre en oeuvre une stratégie de documentation pour la maintenance et l'évolution d'applications distribuées et orientées services. Ce travail sappuiera sur le cas de Buckless, une application de paiement électronique dématérialisé destinée principalement à la simplification des transactions dans le cadre des associations étudiantes. Une version majeure de cette solution open source sera bientôt mise en production par une équipe de développeurs et l'enjeu est de constituer une documentation pertinente pour assurer la maintenance et l'évolution du projet sur le long terme.

Plusieurs outils sont considérés (modèles UML, outils de documentation d'API, wiki) afin de documenter les aspects statiques, les principales interfaces et dépendances, et les aspects comportementaux du système. A partir de l'exemples d'autres cas de projets open-source, une stratégie de documentation sera ainsi formalisée et mise en oeuvre. Cette dernière est mise à l'épreuve en interrogeant son utilité lors de l'ajout de nouveaux composants et fonctionnalité au projet Buckless.

TABLE DES MATIÈRES

I	Etat de l'art	3
I	Documentation du code	3
I.1	Commentaires	3
I.2	UML - Diagrammes de classe	4
II	Documentation de service	5
II.1	Swagger	5
III	Documentation de système	6
III.1	Wiki	6
IV	Intégration dans les processus de développement	7
IV.1	Modélisation avant production	7
IV.2	Linters	8
IV.3	Git hooks	8
IV.4	Intégration continue	8
IV.5	Déploiement continu	9
II	Cas d'études open-source	10
I	Nylas N1	10
I.1	Présentation	10
I.2	Stratégie de documentation	10
II	Ghost	12
II.1	Présentation	12
II.2	Stratégie de documentation	12
III	Application au projet Buckless	15
I	Présentation du projet	15
I.1	Contexte	15
I.2	Technologies utilisées	15
II	Stratégie de documentation	15
II.1	UML - Diagrammes de composants	16
II.2	Représentation des modèles	17
II.3	Description de l'API	17
II.4	Commentaires et documentation	17
II.5	Automatisation	18
III	Application pratique	18
III.1	Architecture globale	18
III.2	Architecture serveur	20
III.3	Architecture du client d'administration	21
III.4	Documentation dynamique de l'API	22
IV	Conclusion	23

I. ETAT DE L'ART

Alors qu'un état de l'art classique se vaut exhaustif, cette partie place volontairement un filtre sur les éléments proposés. En effet, la diversité des outils est telle que seuls ceux poussés de l'avant par la communauté de développeurs seront abordés. D'autres outils moins populaires (comme les diagrammes UML de composants), mais témoignant d'une pertinence particulière pour les architectures orientées services, seront également étudiés.

I. Documentation du code

I.1 Commentaires

La documentation du code source au travers de commentaires est probablement l'étape la plus importante dans un projet faisant appel à une équipe de plusieurs développeurs. Certaines bonnes pratiques se sont démarquées avec le temps, jusqu'à créer une formalisation de ces commentaires, qui donnera plus tard la célèbre **JavaDoc** pour le langage Java. Ce standard est décliné pour la plupart des langages de programmation : on retrouve ainsi jsDoc pour javascript, doxygen pour C/C++, etc..

```
/**
 * Create an array of all the right files in the source dir
 * @param {String} source path
 * @param {Object} options
 * @param {Function} callback
 * @jsFiddle A jsFiddle embed URL
 * @return {Array} an array of string path
 */
function collectFiles(source, options, callback) {
    foo()
}
```

Listing 1 – Exemple de documentation de fonction avec jsDoc

Au delà de l'intérêt immédiat des commentaires sur la lecture du code, ce formalisme vient avec un ensemble d'outil permettant l'extraction des commentaires, puis la génération d'une documentation statique de la structure du code.

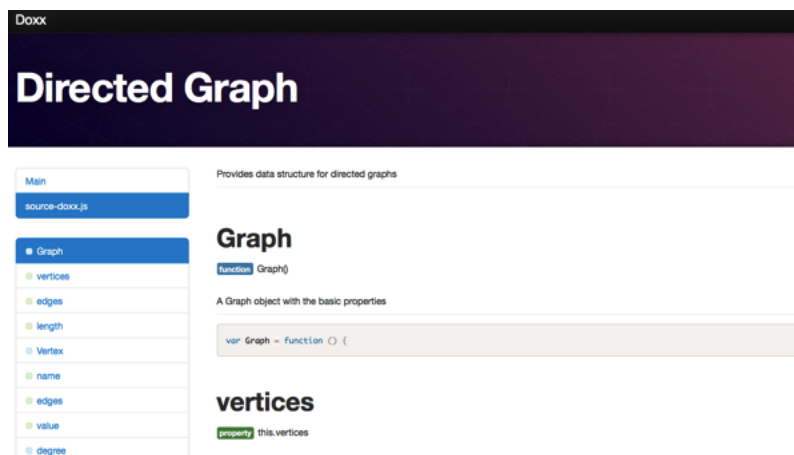


Figure 1 – Une documentation HTML générée par Doxx et jsDoc

On voit donc que les formalismes de commentaires de code sont une première étape essentielle à la documentation d'un projet logiciel. Ces documentations sont cependant sous forme de texte. Ainsi, sur des sources de plusieurs

centaines de fonctions / classes, il peut être intéressant de proposer une vue d'ensemble du code. C'est dans cette optique que nous allons étudier l'utilisation des diagrammes UML de classe.

I.2 UML - Diagrammes de classe

UML¹ est un langage de description, né de collaboration de la communauté orienté-objet, devenu un standard dans la description de systèmes (informatiques ou non). Ses spécifications décrivent un grand nombre de diagrammes. Parmi les plus polulaires, le diagramme de classe pour la modélisation orientée-objet. De plus, le paradigme orienté-objet étant quasiment systématiquement utilisé dans le développement logiciel moderne, il paraît naturel de le présenter ici.

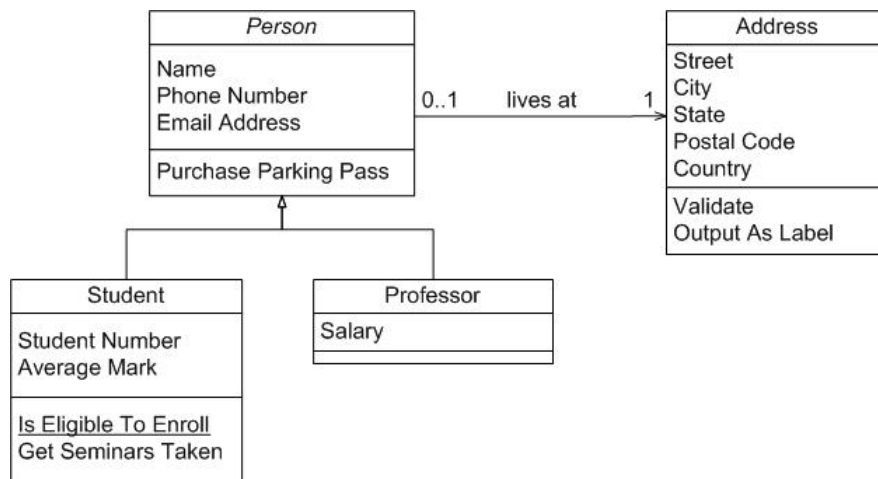


Figure 2 – Un diagramme de classe.

Le diagramme de classe peut être utilisé en amont comme outil de réflexion sur la structure d'un projet logiciel. Lors de la production du code, cette structure est amenée à être altérée. Ainsi, une deuxième utilisation émerge, celle de la description de la structure à un instant du code à un instant t . Il existe plusieurs logiciels permettant d'adopter cette "démarche inverse", à savoir pouvoir produire des diagrammes à partir de code existant.

Cette utilisation est d'autant plus intéressante qu'elle permet aux côtés des documentations statiques vues précédemment, de rendre compte de l'état du code à travers différents canaux (textes, visuels...). De plus, contrairement au formalismes de documentation par commentaire, la syntaxe UML des diagrammes de classes permet de visualiser de manière intuitive la nature des liens entre les différentes entités.

1. Unified Modeling Language

II. Documentation de service

À l'image de la documentation normalisée visant à rendre le code accessible à d'autres développeurs, le besoin de décrire des WebServices est vite apparu. Nous allons présenter ici les trois formalismes de description d'API REST les plus populaires.

II.1 Swagger

Swagger est le premier essai de formalisme de description d'API REST. Son but, définir un standard *language-agnostic*² permettant aux humains comme à des outils informatiques d'explorer une API sans avoir à plonger dans le code. Ses créateurs définissent Swagger[3] par analogie comme l'équivalent des interfaces en programmation orienté-objet, appliqué aux services.

```
paths :
  /pets :
    get :
      description : "Returns all pets from the system that the user has access to"
      produces :
        - "application/json"
      responses :
        "200" :
          description : "A list of pets."
          schema :
            type : "array"
            items :
              $ref : "#/definitions/Pet"
definitions :
  Pet :
    type : "object"
    required :
      - "id"
      - "name"
    properties :
      id :
        type : "integer"
        format : "int64"
      name :
        type : "string"
      tag :
        type : "string"
```

Listing 2 – Exemple de définition d'API Swagger

L'ensemble des outils dérivés incluent : un générateur de code client / serveur, un éditeur de définition en ligne, et un générateur de documentation dynamique. La génération de documentation dynamique passe tout d'abord par la description du service par le formalisme de Swagger. Cette description s'est fait avec le langage de description YAML³, ou JSON.

De nombreuses alternatives à Swagger existent : RAML, Apiary, API blueprint... Elles reposent globalement sur les même concepts, il n'est pas pertinent de détailler leur spécificités ici.

2. Qui ne dépend pas du langage de programmation

3. Superset de JSON en plus lisible

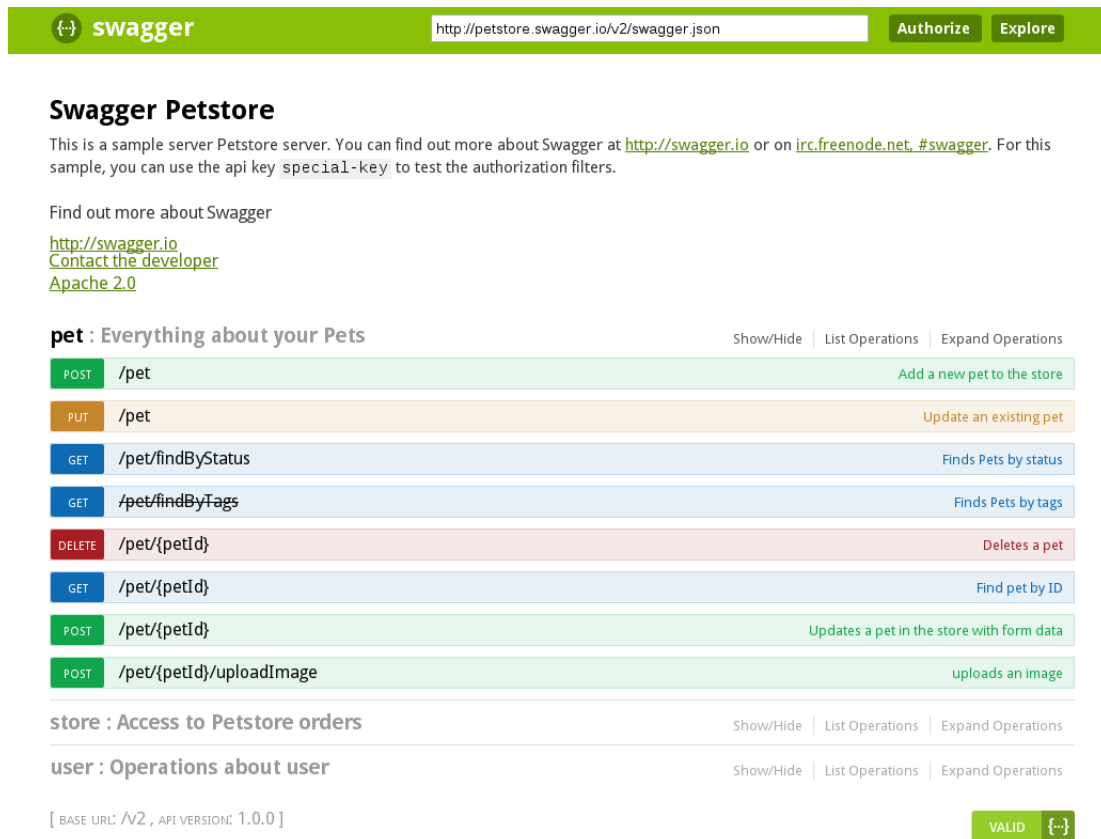


Figure 3 – Documentation dynamique générée par Swagger

III. Documentation de système

III.1 Wiki

Un wiki est une application web qui permet de la gestion collaborative de contenu. Sa structure flexible permet de pouvoir centraliser, trier et distribuer les différentes strates de documentations discutées ci-dessus. De plus, il peut servir pour ajouter du contenu décrivant le système et ce qui gravite autour, à un niveau de description qui n'avait pas sa place dans les contextes vu précédemment. Un rassemblement des wiki les mieux tenus dans le milieu de l'open source est proposé par GitHub[5].

Netflix / **Hystrix**

Watch 777
Star 6,148
Fork 1,136

Code
Issues 47
Pull requests 7
Wiki
Pulse
Graphs

How it Works

Matt Jacobs edited this page on Aug 7, 2015 · 27 revisions

Contents

- Flow Chart
- Circuit Breaker
- Isolation
- Threads & Thread Pools
- Request Collapsing
- Request Caching

Flow Chart

The following diagram shows what happens when you make a request to a service dependency by means of Hystrix:

for larger view) [\(Click](#)

Pages 14

- Home
- Getting Started
- How it Works
- How To Use
- Operations
- Configuration
- Metrics
- Plugins
- Dashboard
- End-to-End Examples
- Migration Guide
- FAQ : General
- FAQ : Operational

Clone this wiki locally
<https://github.com/Netflix/Hystrix>

Figure 4 – Un exemple de wiki alliant plusieurs moyens de documentations

IV. Intégration dans les processus de développement

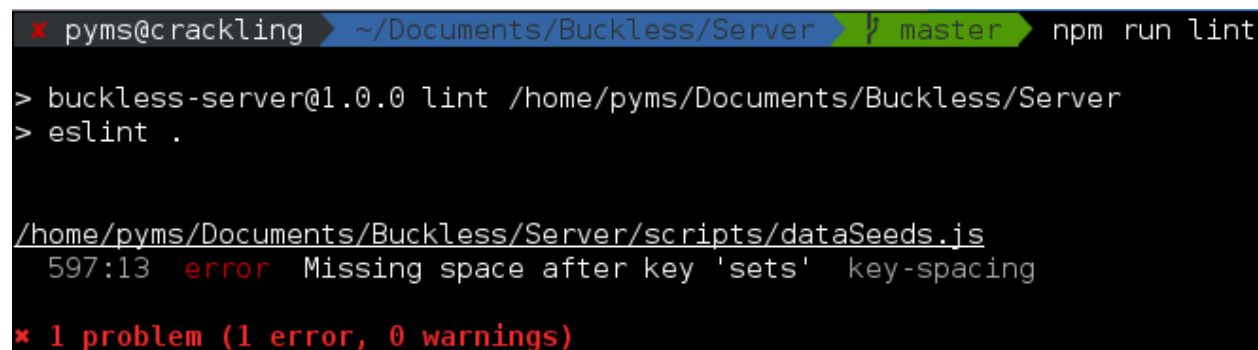
Une grande problématique du développement logiciel est la maintenance de la documentation associée à jour. Nous allons étudier les différents leviers qui permettent d'intégrer la documentation à la production de code.

IV.1 Modélisation avant production

La première étape avant la production du code est de spécifier et modéliser les systèmes et leurs interactions. On pourra alors commencer par modéliser les ressources : diagrammes de classes, diagrammes entité / association. Vient ensuite la description des services avec des diagrammes de composants et des outils type Swagger, RAML, etc. Au moment de la production du code, et durant toute son évolution, l'intégration de la documentation s'appuiera sur un outil incontournable, le linter.

IV.2 Linters

Un linter désigne un logiciel d'analyse statique de code. Configuré pour, un linter peut vérifier la présence de commentaires de type javadoc. On utilisera cette fonctionnalité de vérification pour intégrer la documentation au développement logiciel.



```

x pyms@crackling ~/Documents/Buckless/Server master npm run lint
> buckless-server@1.0.0 lint /home/pyms/Documents/Buckless/Server
> eslint .

/home/pyms/Documents/Buckless/Server/scripts/dataSeeds.js
  597:13  error  Missing space after key 'sets'  key-spacing

* 1 problem (1 error, 0 warnings)

```

Figure 5 – Exemple de sortie d'eslint

Pour ce qui est de la maintenance à jour des diagrammes UML, certains outils proposent de générer les diagrammes à partir du code. Cependant les résultats ne sont pas toujours satisfaisant, on ne les présentera donc pas ici.

IV.3 Git hooks

Git est aujourd'hui l'outil de versionning de code le plus largement utilisé. Parmi ses possibilités, celle de définir des hooks⁴ lorsqu'une nouvelle version est propagée. Ainsi, il est possible d'utiliser le linter de code avec un hook "pré-commit", c'est à dire avant la propagation d'une nouvelle version, pour n'autoriser la propagation qu'une fois que le linter valide l'état du code et de ses commentaires.

Cette méthode est néanmoins un peu brutale, car parfois incompatible avec certaines méthodes de développement. Il est parfois critique de devoir déployer sur la production un bugfix, ce qui peut être gênant lorsque l'on travaille avec des hooks qui ne sont pas forcément triviaux à désactiver.

Elle n'est cependant pas à oublier : certaines grosses entreprises avec des exigences de qualité de code très strictes utilisent des mécanismes similaires.

Pour créer un hook de pré-commit, il faut créer un fichier qui contient des instructions bash, qui sera exécuté avant chaque commit. En cas d'erreur, le commit n'est pas effectué.

```
touch .git/hooks/pre-commit
```

Ce fichier doit avoir les droits d'exécution :

```
chmod +x .git/hooks/pre-commit
```

Enfin, il ne reste plus qu'à appeler le linter :

Ainsi, lors de l'appel d'un commit, avec un code non conforme, on obtient la figure suivante.

IV.4 Intégration continue

L'intégration continue désigne un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application déve-

4. Action déclenchée automatiquement


```
#!/bin/sh

eslint .
```

Listing 3 – Exemple de script de hook

```
* pyms@crackling ~/Documents/Buckless/Server master git commit -m "Commit test"
> buckless-server@1.0.0 lint /home/pyms/Documents/Buckless/Server
> eslint .

/home/pyms/Documents/Buckless/Server/scripts/dataSeeds.js
  597:13  error  Missing space after key 'sets'  key-spacing

* 1 problem (1 error, 0 warnings)
```

Figure 6 – Resultat du hook : le commit est annulé

loppée. Il assure la bonne compilation du code, mais aussi le jeu des tests unitaires, le packaging, le déploiement et l'exécution des tests dans un environnement d'intégration. Intégrer le lint au coeur de ce processus permet de déclarer le code *únon saintz*, et ainsi pousser le développement à se faire en même temps que celui de sa documentation. Parmi les outils d'intégration continue les plus populaires, on pourra citer : Travis CI, Circle CI, Jenkins, Gitlab CI. L'écosystème est cependant en pleine évolution, et il existe des dizaines d'alternatives.

IV.5 Déploiement continu

On appelle déploiement continu le fait d'automatiser les mécanismes de déploiement du code en production à la suite de l'intégration continue. Un cas d'utilisation immédiat pour la documentation est alors la mise en ligne automatique des documentations générées par Doxx ou Swagger après chaque nouvelle version du code. Cette pratique est d'autant plus facile d'accès maintenant que GitHub propose d'héberger des sites statiques gratuitement pour les projets open-source.

II. CAS D'ÉTUDES OPEN-SOURCE

Avant de définir une stratégie de documentation pour le projet Buckless, nous allons nous intéresser à la manière dont les différents outils présentés précédemment ont été intégrés par de gros projets open source. Ces projets ont été choisis pour leur similitude avec le projet Buckless, que ce soit par les technologies utilisées, le type de logiciel produit, l'architecture logicielle semblable, etc.

I. Nylas N1

I.1 Présentation

Nylas est une organisation réalisant le projet N1, un client mail se basant sur des technologies web. En effet, le projet utilise un chromium⁵ embarqué pour distribuer une application web en tant qu'application native. Cela permet d'avoir une interface facilement stylisable et extensible, en plus d'être multiplateforme et portable. Cette interface utilise une API REST, ouverte, dont la documentation est générée à partir du code. Malheureusement, le code source de cette API n'est pas disponible et il n'y a pas moyen de connaître la méthode de génération. On peut toutefois observer que la génération de la documentation est faite en interne via un script, et non automatiquement à l'aide d'intégration continue.

I.2 Stratégie de documentation

Une documentation est aussi disponible pour l'application en elle-même. Vu que le projet est basé sur le paradigme de Programmation Orienté Objet, les classes sont assez facile à identifier avec "l'API Reference". De plus, N1 utilise les Web Components qui apportent la modularité aux sites webs en découpant chaque partie en composants autonomes, et ils sont eux aussi facilement identifiables.

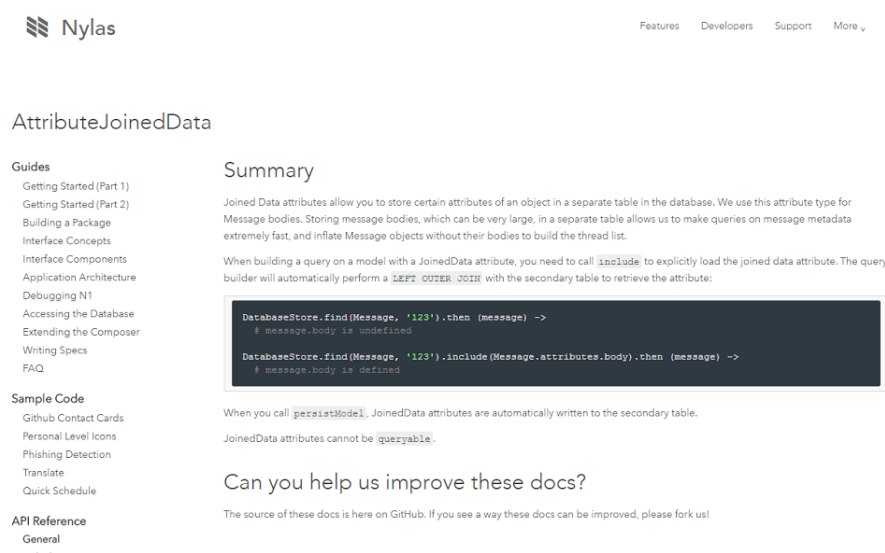


Figure 7 – Accueil de la documentation

L'intérêt majeur de cette documentation est qu'elle est générée via le code source de l'application. En utilisant les commentaires, leur script de génération de documentation va créer une arbre syntaxique (AST⁶), aussi utilisé dans la compilation de programmes.

5. Navigateur web

6. Abstract Syntax Tree

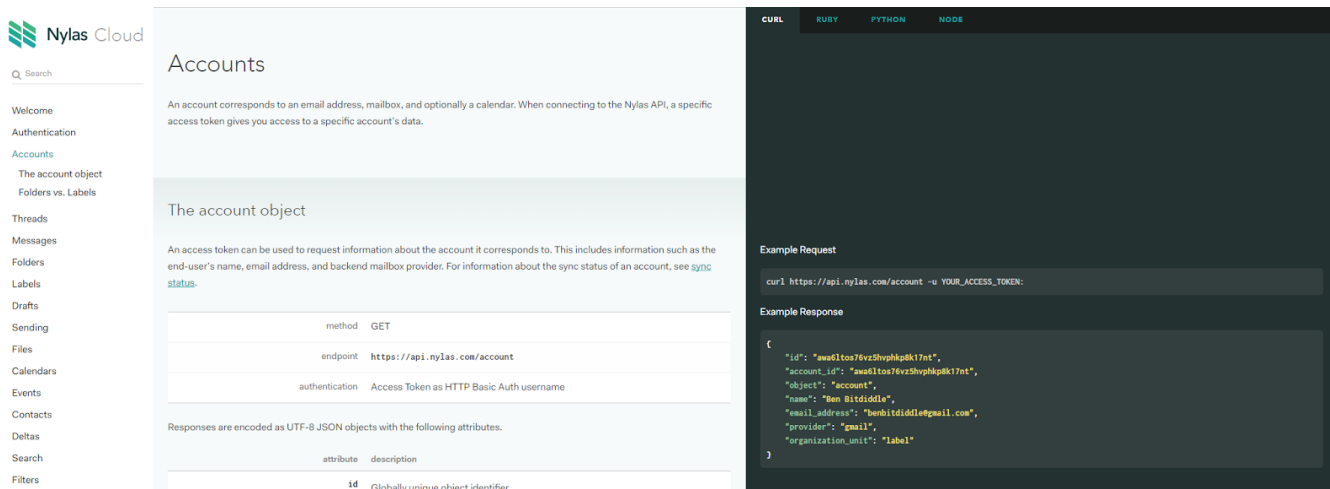


Figure 8 – La référence de l'API

```
# Public: A mutable text container with undo/redo support
class TextBuffer
  @prop2 : "bar"

  # Public: Takes an argument and does some stuff.
  #
  # a - A {String}
  # Returns {Boolean}.
  @method2 : (a) ->
```

Listing 4 – Exemple de commentaire TomDoc

Au niveau des commentaires, un dérivé de jsDoc (TomDoc[6]) est utilisé, plus adapté pour des grosses parties de documentation (là où jsDoc est plus adapté pour l'auto complétion et la documentation rapide). La génération de l'AST est faite à l'aide d'une librairie spécialisée[7].

```
"files" : {
  "spec/metadata_templates/classes/class_with_prototype_properties.coffee" : {
    "objects" : [
      {
        "type" : "class",
        "name" : "TextBuffer",
        "bindingType" : null,
        "classProperties" : [],
        "prototypeProperties" : [11, 11],
        "doc" : " Public: A mutable text container with undo/redo support\n\n "
      }
    ]
  }
}
```

L'AST généré est ensuite transformé en code quasiment exportable en HTML, en utilisant une deuxième librairie[8]. Enfin, Nylas possède un script qui va automatiquement réaliser l'AST puis le contenu et générer le code HTML de la documentation. Le code est ensuite publié sur une branche spécifique à la documentation de leur projet. Cette branche est automatiquement mise en ligne par GitHub[9] ce qui permet uniquement en modifiant le code, de mettre à jour la documentation en ligne directement.

II. Ghost

II.1 Présentation

L'idée de Ghost est née lorsque Jon O’Nolan, fondateur de Wordpress, se rendit compte que Wordpress était devenu beaucoup trop complexe pour une simple plateforme de blogging. C’est à partir de ce constat qu’il décida de créer une plateforme de blogging dédiée uniquement au contenu. Débutant de projet de zéro, Ghost a été construit sur du javascript côté client et serveur.

Aujourd’hui, Ghost est une plateforme massivement utilisée. C’est d’ailleurs un projet open-source, qui base son modèle économique sur du Software As A Service (SAAS). C’est donc autant pour les similitudes technologiques que celles du business model que Ghost a été retenu en tant qu’exemple.

II.2 Stratégie de documentation

Contrairement à Nylas N1, Ghost a pris le parti de ne pas intégrer la documentation à sons processus de développement. La documentation est quelque chose de complètement indépendant du code, et possède son propre versionning. Ghost propose deux grands types de documentation : une documentation orientée utilisateur, et une orientée développeur. C’est sur cette dernière que va se concentrer notre analyse.

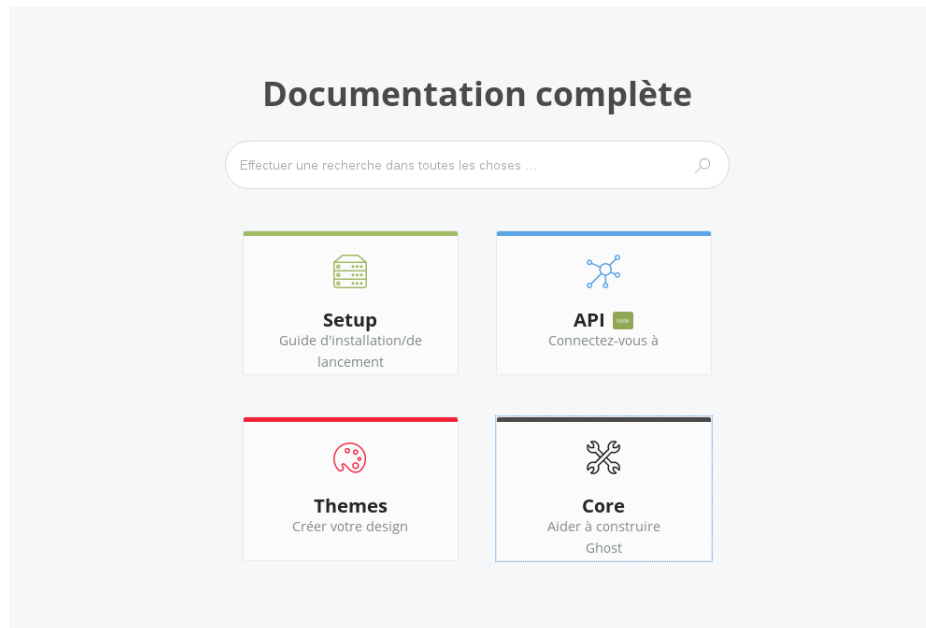


Figure 9 – Accueil de la documentation

- ▶ INTRODUCTION
- ▶ GENERAL CONCEPTS
- ▼ PUBLIC ENDPOINTS
 - GET /posts/
 - GET /posts/id
 - GET /posts/slug/slug
 - GET /tags/
 - GET /tags/id
 - GET /tags/slug/slug
 - GET /users/
 - GET /users/id
 - GET /users/slug/slug
 - GET /users/email/email
- ▶ RESOURCES
- ▶ PARAMETERS
- ▶ COOKBOOK

GET /users/

limit	string 15
How many users to retrieve, use all to retrieve all users	
page	string 1
Which page of paginated results to retrieve	
order	string last_login (desc)
Choose which field to order by and direction of ordering (asc or desc)	
include	string
count.posts	
fields	string
Choose which fields you want to include in collection	
filter	string
Use Ghost Query Language to build complex query	

The users endpoint allows you to browse all active users on a particular blog. By default it returns a paginated set of 15 active users in descending order of last_login.

HTTP · jQuery.ajax · Internal · Handlebars

GET /users/

You can further order/filter your collection using the allowed parameters above. These are discussed in more detail in the [parameters section](#).

Specifying *id*, *slug* or *email* changes the query to a **read** request, in this case only the **include** parameter is valid.

Definition

http://demo.ghost.io/ghost/api/v0.1/users/

Examples

jQuery.ajax · Internal · Handlebars

```
$.get(ghost.url.api('users')).done(function (data){
  console.log('users', data.users);
}).fail(function (err){
  console.log(err);
});
```

Result Format

200 OK · 400 Bad Request

```
{
  "users": [
    {
      "accessibility": null,
      "bio": null,
      "cover": null,
      "created_at": "2014-10-11T19:02:27.147Z",
      "created_by": 1,
      "id": 1,
      "image": null,
      "language": "en_US",
      "last_login": "2014-11-17T19:02:27.147Z",
      "location": null,
      "meta_description": null,
      "meta_title": null,
      "name": "Eric Almeida",
      "slug": "eric-almeida",
      "status": "active",
      "tour": null,
      "updated_at": "2014-10-11T19:02:27.147Z",
      "updated_by": 1,
      "uuid": "fs4a0021-b22a-33a1-531c-424e2caba3c3",
      "website": null
    }, {
      "accessibility": null,
      "bio": "Hude Developer"
```

Figure 10 – API Reference

Parmi la documentation développeur, deux grands axes sont développés. Le premier, est une documentation de l'API RESTful de Ghost. Un formalisme de description d'API tel que Swagger (probablement API Blueprint) est utilisé pour générer une documentation dynamique. Il n'est malheureusement pas possible d'en avoir la certitude, puisque la solution⁷ utilisée est payante.

7. readme.io

Le deuxième est une documentation de la structure interne du serveur. Celle-ci aborde moins le côté "fonctionnel" du serveur, à savoir les différentes routes et ce qu'elles retournent, mais plus une approche structurale qui permet de former les potentiels contributeurs. Elle décrit notamment le style de code à utiliser lors d'une contribution, l'architecture globale de l'application et ses composants, mais aussi des indications quant au déploiement de Ghost. Cette documentation se présente sous la forme d'un Wiki, hébergé directement sur le dépôt du projet sur GitHub. Différents types de documents sont proposés : du texte, des diagrammes et des schémas ASCII. On peut remarquer qu'aucun formalisme n'est utilisé pour les différents diagrammes.

TryGhost / Ghost Watch 1,022 Star 19,787 Fork 4,820

Code Issues 185 Pull requests 28 **Wiki** Pulse Graphs

Home

JT Turner edited this page on Feb 20 · 62 revisions

Welcome to the Ghost Github wiki!

The wiki contains all kinds of documentation about the development of Ghost, and is also the place where you'll find early documentation for working with the various APIs. These documents are usually aimed at the people developing features, so they're rough around the edges, but they're here nonetheless.

Here's a quick list of the most useful pages:

General:

- [Roadmap](#) (mostly a link to the [Public Roadmap](#))
- [Planned Features](#)
- [Core Team](#)

For contributors:

- [Contributing Guide](#)
- [Code standards](#)
- [Git workflow](#)
- [Grunt Toolkit](#)

Pages 40

Find a Page...

- [Home](#)
- [\[WIP\] API Documentation](#)
- [Accounts API](#)
- [App Ideas](#)
- [App Ideas Permissions](#)
- [Apps Getting Started for Ghost Devs](#)
- [Channels 101](#)
- [Code standards](#)
- [Codebase Overview](#)
- [Context aware Filters and Helpers](#)
- [Core Team](#)
- [Deploy Ghost to EC2](#)

Figure 11 – *Accueil du wiki*

Il est intéressant de constater les différences de stratégie de documentation des deux projets. D'un côté, Nylas N1 pousse la principe de la fusion des processus au maximum, en intégrant des paragraphes entiers de documentation au sein des commentaires de la source pour générer sa documentation. De l'autre, Ghost fait preuve d'une politique plus traditionnelle, en gardant une complètement séparé l'action de développer et celle de documenter. Il est possible de voir ici des raisons économiques, puisque Ghost est un projet actuellement beaucoup plus gros. Ainsi, les ressources humaines sont également plus importantes et permettent d'avoir un pouvoir de développement et la capacité de pouvoir documenter en même temps sans avoir à se poser de question. Mais cela peut aussi être le résultat d'un choix, où l'automatisation de la documentation demande une grande part de configuration et l'utilisation d'une multitude d'outils, ce qui complexifie le développement. C'est sur base de ces observations que nous allons maintenant essayer de définir une stratégie de documentation pour le projet Buckless.

III. APPLICATION AU PROJET BUCKLESS

I. Présentation du projet

I.1 Contexte

L'idée a vu le jour au sein de notre campus, en se basant sur des situations qui nous posaient problème. Vendre des produits ou des services peut être quelque chose de complexe si l'on considère le temps qu'il faut à un client pour accéder à un vendeur. De plus, ces derniers sont le plus souvent des étudiants volontaires qui n'ont pas l'expérience qu'un professionnel pourrait avoir, spécialement lors d'un rush. C'est pour cela que nous avons décidé de développer une solution de paiement entièrement dématérialisée pour optimiser les flux monétaires et logistiques lors des événements de notre école.

Buckless se démarque de certains projets similaires par le fait qu'il ne compte pas devenir un système de paiement global, intermédiaire de la banque. L'idée est de créer des monnaies locales, indépendantes entre chaque structure. Les clients gardent ainsi la main mise sur leur trésorerie et leur infrastructure informatique. La solution se présente sous la forme de bornes physiques (téléphones, tablettes, ordinateurs tactiles), utilisées par des vendeurs. Le client, lui, peut recharger un compte virtuel via une application en ligne, ou via des points de rechargement physique directement sur site. Son moyen de paiement est défini par l'entreprise : un bracelet, un tatouage pratique dans le monde événementiel ou une carte monde étudiant, ou encore son smartphone. Ce moyen de paiement permet d'identifier le client et de réaliser des achats instantanément.

I.2 Technologies utilisées

Le client de vente présent sur les bornes est une application web. En plus de l'avantage d'être multi-plateforme, procéder ainsi permet le déploiement automatique de chaque nouvelle version du client sur les bornes. L'application web est entièrement écrite en javascript/css/html sous la forme d'une *single page application*. Il a été choisi de construire le client à l'aide de *Web Components*, une approche moderne et modulaire de développement. L'idée est de définir des composants indépendants, composés de leurs propres fichiers javascript/css/html, puis de les agencer afin d'obtenir l'application finale. Pour ce faire, le framework Vue.js a été choisi, bien que d'autres puissent faire de même (Angular.js, react, ..).

Le serveur est développé en javascript sur la plateforme Node.js. Il est composé d'un service RESTful, de services d'authentification, et d'un serveur WebSocket pour la communication en temps réel.

II. Stratégie de documentation

À la vue des différents projets étudiés, de leurs approches de la documentation, et des spécificités du projet Buckless, il a été possible de définir une stratégie de documentation. L'équipe de développement étant très restreinte, il a été trouvé plus judicieux d'avoir une approche la plus automatisée possible de la documentation, afin de pouvoir se forcer à écrire cette dernière en même temps que le code. Cette automatisation passe par les mécanismes et les outils décrits dans l'état de l'art : commentaires normalisés, javadoc, formalisme de description d'API, et déploiement continu.

La stratégie proposée est relativement opposée à celle de Ghost, et donc proche de celle de Nylas, avec la majeure différence de ne pas inclure de larges portions de textes dans les commentaires en vue de générer des pages. Ces larges portions de textes seront rédigées séparément, sur une plateforme de type wiki. De même, la documentation du code et du système par les diagrammes ne pouvant être générée (avec un résultat convenable) à partir du code, ceux-ci devront être tenus à jour à la main. Afin d'éviter d'avoir un travail important de mise à jour des diagrammes, seuls des éléments jugés stables (architecture globale, modèles) sont représentés par ces derniers.

II.1 UML - Diagrammes de composants

Parmi la spécification d'UML, un type de diagramme sous-utilisé, le diagramme de composants, se révèle être un bon moyen de décrire les architectures utilisant des services.

En effet, un service peut être vu comme un composant d'un système global, et les interfaces de ce composant comme les ressources exposées par le service.

L'approche "composant" au niveau logiciel est intéressante, car elle permet de rendre compte de l'architecture logicielle à une échelle plus grande que celle du diagramme de classe. Il est ainsi possible de modéliser les interactions entre des grandes parties du logiciel, parties découpées de manière "fonctionnelle", et ce malgré d'éventuels changements au niveau du code.

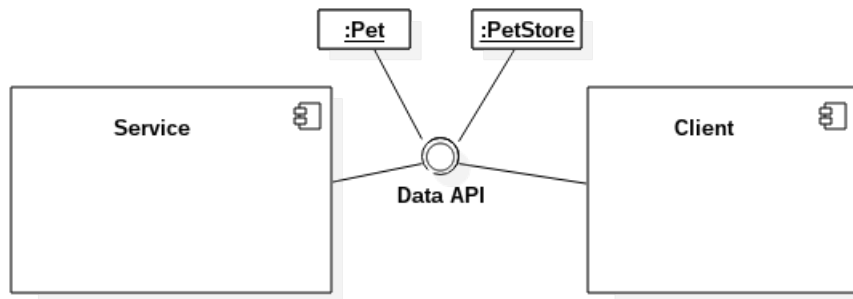


Figure 12 – Représentation avec des objets d'une API et ses ressources

Un premier essai de modélisation est donné ci-dessus. La première caractéristique d'une telle modélisation est l'impossibilité de voir où les ressources sont sollicitées au sein même du composant. De plus, à mesure que le nombre de ressource augmente, l'encombrement visuel dû à la représentation des instances passées rend la représentation difficile.

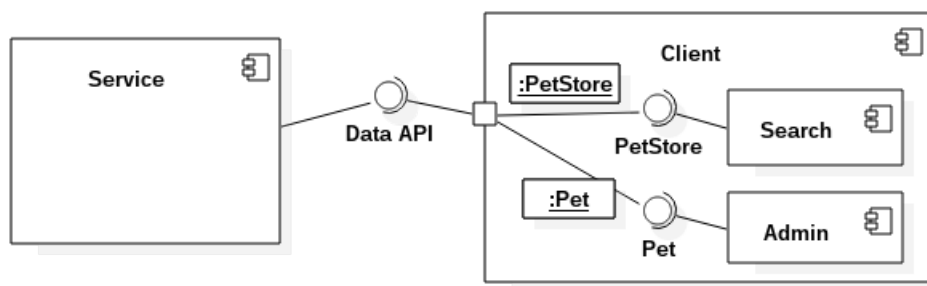


Figure 13 – Représentation d'une API et de ses ressources avec des interfaces

La représentation de l'interaction avec les sous composants (des WebComponent par exemple) est rendue possible par l'utilisation d'interfaces. De plus, pour éviter des diagrammes trop verbeux, on proposera la **convention** suivante pour les diagrammes de composants orientés service : **Une interface est homonyme au ressource qu'elle expose.**

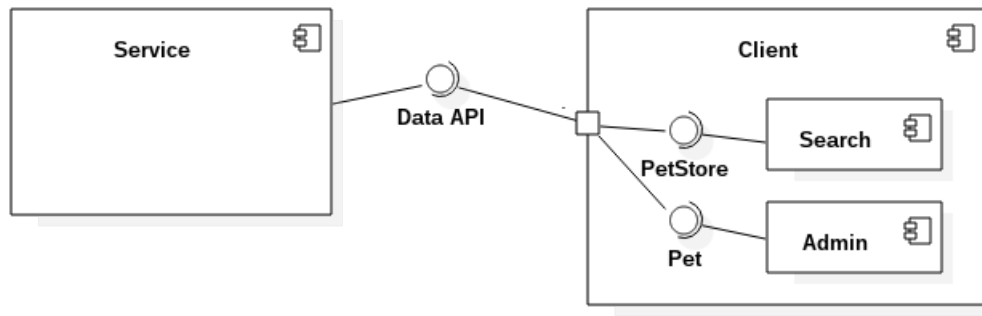


Figure 14 – Représentation d'une API et de ses ressources avec des interfaces homonymes aux instances

Cette utilisation des interfaces renvoie directement à la définition (cf II.1) du formalisme de description d'API Swagger. On remarque alors que le diagramme de composant pour les architectures orientées services, est aux formalismes de type Swagger ce que les diagrammes de classes sont aux commentaires de type Javadoc : une représentation visuelle alternative.

Il est intéressant de constater qu'il devient aisé d'avoir une vue globale de l'état d'utilisation des services (et donc des ressources) par les différentes parties de ou des logiciels. Un bénéfice immédiat de cette modélisation est donc de pouvoir anticiper l'impact qu'aurait la modification du fonctionnement d'un service, ou de la modification des modèles qui lui sont associés.

II.2 Représentation des modèles

La représentation au niveau composant de l'architecture logicielle a été jugée suffisamment détaillée pour ne pas descendre d'échelle et proposer des diagrammes de classes.

En revanche, la question de pose de la représentation des différents modèles annotés en tant qu'interfaces sur le diagramme de composant. Pour cela, deux représentations possible : le diagramme entité / association (en notation UML ou notation de Chen), ou le diagramme de classes. Le premier offre une description d'un point de vue théorique, là où le second se soucie des détails d'implémentations (description des tables de jointures, par exemple). Dans la mesure où l'implémentation de la base de donnée n'a pas été prévue pour changer de technologie, il a été choisi de décrire cette dernière à l'aide de diagrammes de classes, complémentaires aux diagrammes de composants.

II.3 Description de l'API

Parallèlement au diagrammes de composants qui offrent une vue globale et visuelle des services (mais pas forcément exhaustive), nous avons du choisir un formalisme de type Swagger pour générer une documentation dynamique de l'API.

Le problème que nous avons eu avec Swagger est son principe déterministe[11]. Il n'est pas possible de définir plusieurs blocs de réponses possible pour un même code http. Ainsi, nous avons du nous tourner vers une alternative : APIAry et la syntaxe Blueprint..

II.4 Commentaires et documentation

Nous avons vu au travers l'exemple de Nylas N1 qu'il était possible d'intégrer des larges portions de texte de documentation, dans l'optique de réunir code et documentation dans les même fichiers. Pour buckless, nous avons jugé cette pratique de vouloir réunir documentation et code un peu excessive, car parfois peu lisible au niveau du code source. C'est pourquoi nous nous sommes limités à des commentaires de types jsDoc simples.

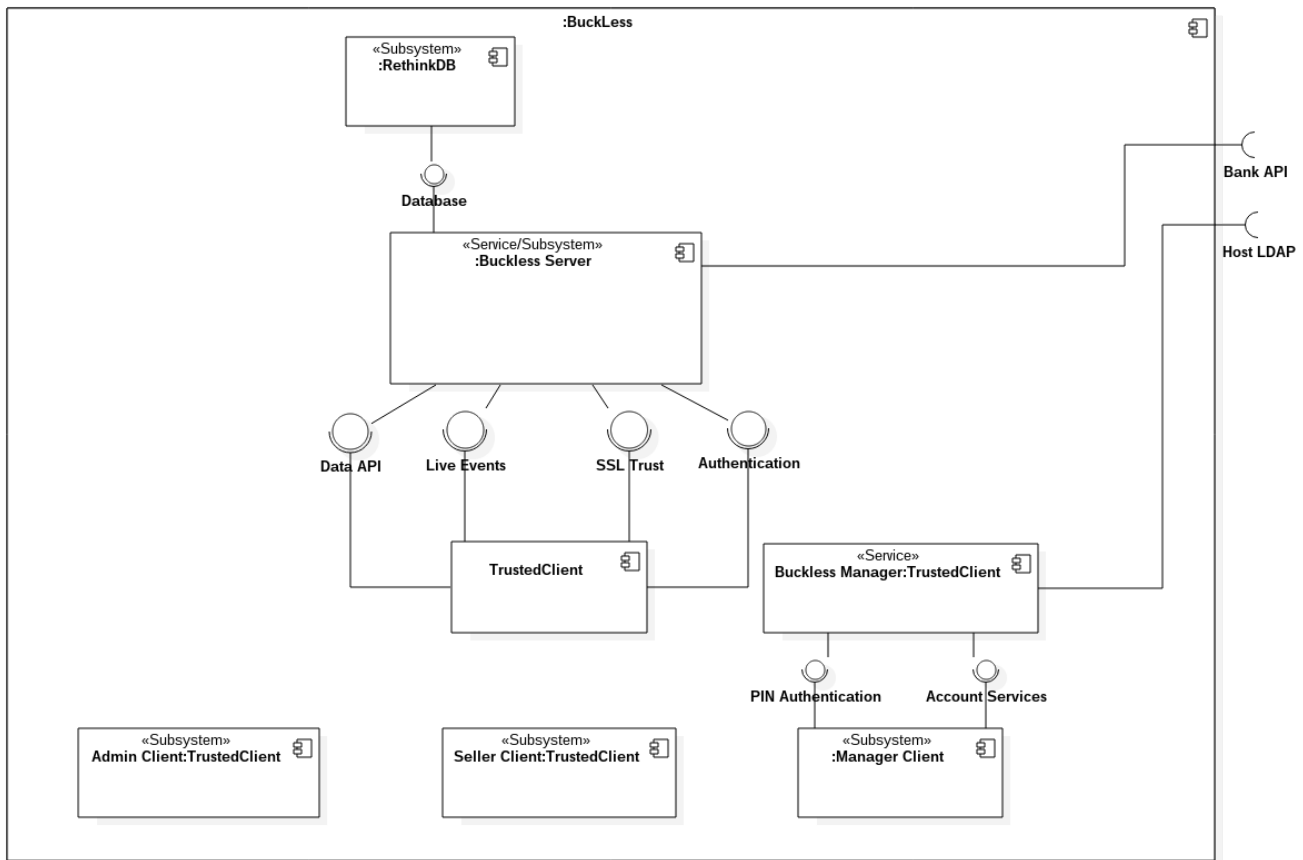
II.5 Automatisation

Au niveau de l'automatisation de la production de la documentation, le duo linter et déploiement continu a été retenu. Pour des raisons de workflow incompatible (évoqué en IV.3), l'utilisation des hooks n'a pas été retenue.

III. Application pratique

Dans cette partie, nous essaierons de nous appuyer sur des éléments de documentations pratiques qui ont été produits pour le projet, afin de présenter certains mécanismes et choix techniques. Buckless étant un projet complexe, seul les points les plus intéressants, ou les mieux représentés seront présentés ici.

III.1 Architecture globale

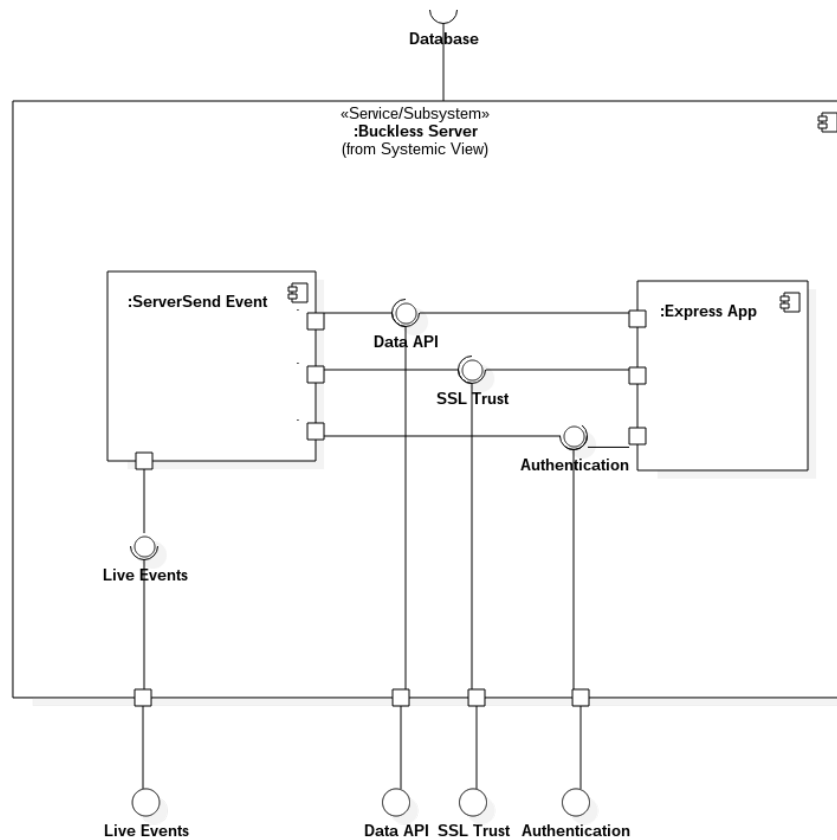


Buckless est composé d'une base de données (**database**) et de quatre grandes briques logicielles : une API centrale (**Buckless Server**), un client de de vente (**Seller client**), un client d'administration (**Admin client**), et une application utilisateur mobile (**Manager client**).

Tous les clients de l'API partagent une structure abstraite commune (**Trusted client**), qui consiste en un client disposant des différentes interfaces de l'API sous réserve d'une authentification SSL.

Le système global est dépendant de deux services tiers, l'API de la banque avec laquelle les transactions sont faites, et un potentiel annuaire du système d'information hôte (le caractère optionnel d'une interface ne peut pas être représenté avec le formalisme du diagramme de composant).

III.2 Architecture serveur

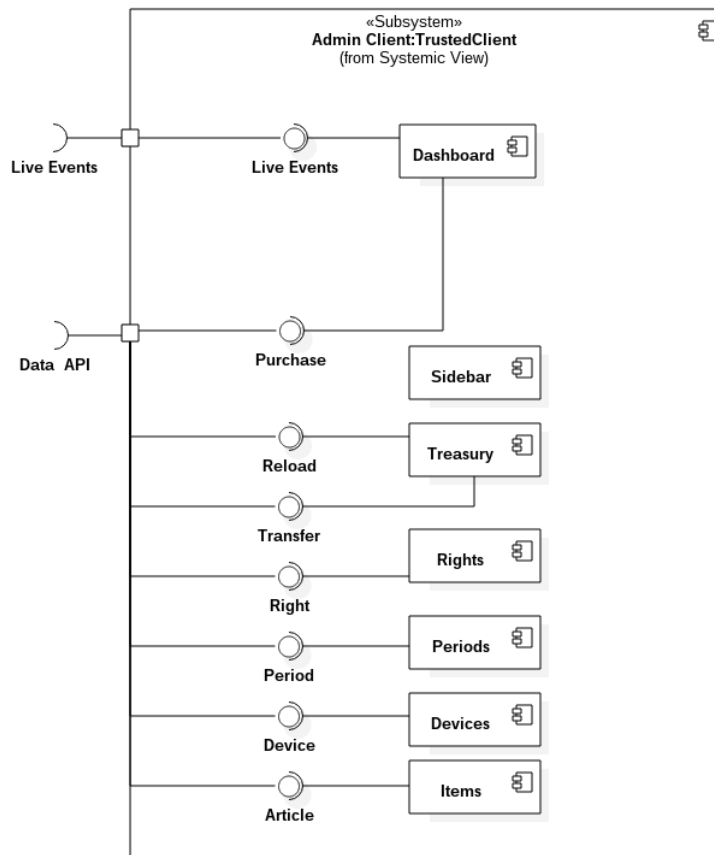


Le serveur a comme unique dépendance la base de donnée. Il est composé d'une API REST et d'un composant permettant l'envoi de données depuis le serveur en temps réel. L'API REST est développée avec le framework express (**Express App**). À côté, l'envoi des données depuis le serveur vers le client est fait avec une librairie de Server Send Event (**SSE**). Les deux sont intimement liés, puisque pour récupérer des données, le SSE passe par les différentes méthodes exposées par l'application express.

Les mécanismes de connexion et d'authentification SSL sont faits au travers de *middlewares*⁸ qui sont encore une fois décrits dans une logique commune aux deux composants. Ces mises en commun sont représentées par des délégations d'interface au sein du composant global **Buckless Server**.

8. Routines logicielles exécutées entre la connexion et l'applicatif

III.3 Architecture du client d'administration



Ce diagramme manque cruellement de détail, car cette partie est encore en développement. Seule l'idée de base est représentée ici. Chaque sous composant interne représente un WebComponent (alliance d'un document html, de son style CSS et de sa logique javascript). L'intérêt ici est d'associer chaque WebComponent avec les ressources dont il dépend au niveau de l'API. Les dépendances envers l'API REST ainsi que vers les Server Send Events sont représentées par des interfaces requises. Au niveau de la **Data API**, un port est utilisé pour découpler l'interface requises en des interfaces fournies. Celles ci respectent la convention de nommage des interfaces, qui reflètent les ressources qu'elles délivrent.

III.4 Documentation dynamique de l'API

The screenshot displays the Apiary API documentation interface for a project named 'Buckless' by Gabriel JUCHAULT. The interface is divided into a sidebar on the left and a main content area on the right. The sidebar contains links for 'INTRODUCTION', 'REFERENCE', 'Documents', and 'Services'. The 'Services' section is expanded, showing three services: 'Login', 'Submitting a basket', and 'Transfer credit'. The 'Login' service is selected, and its details are shown in the main content area. The 'Login' service is described as 'Log a user in.' and has a 'POST' method. The endpoint is 'https://localhost:3000/services/login'. The 'Request' section shows a JSON body with fields: 'meanOfLogin: 'cardId'', 'data: '22000000353423'', 'pin: '123'', and 'password: 'abc''. The 'Response' section shows a status of '200' and headers: 'Content-Type: application/json' and 'point: The point id calculated from the ssl client certificate'.

La documentation dynamique de l'API a été générée par Apiary, un outil semblable à celui utilisé par Ghost. Il propose de générer des documentations à partir des plus gros formalismes de description d'API (Swagger, API blueprint, RAML).

IV. CONCLUSION

Aujourd'hui, il existe une grande diversité d'outils permettant de faciliter la production d'une documentation claire et complète. Nous avons vu qu'il était possible de faire en sorte d'automatiser un maximum l'utilisation de ces outils, afin de les intégrer au coeur du processus de développement logiciel.

Le monde du logiciel libre est intimement lié avec la mise en oeuvre de stratégie de documentation efficace. En effet, l'attractivité du logiciel envers ses contributeurs est directement corrélé avec la présence d'une documentation complète minimisant le temps de formation. Au travers de l'exemple de Nylas, il a été force de constater que la documentation était au coeur de leur processus de développement. L'approche utilisée, consistant à l'automatisation à l'extrême par l'incorporation de la documentation au sein même du code, permet de lier le développeur inévitablement à une production de documentation.

La définition d'une stratégie de documentation pour le projet Buckless a été faite en se basant sur les différents outils présentés dans l'état de l'art, mais aussi en prenant comme exemple des projets open source similaires comme Nylas N1. Il est cependant important de noter que malgré la similitude des projets au niveau technologique, certaines spécificités ont fait que le modèle n'a pas pu être tout simplement calqué. Ainsi, il a été choisi d'explorer l'utilisation du diagramme de composants en tant qu'outil de représentation de système orienté service. L'intérêt de ce diagramme est double, puisqu'il permet de représenter à la fois les services web, mais aussi l'interaction des internes de chaque clients avec ces derniers. En effet, les technologies web se tournent de plus en plus vers l'utilisation de WebComponents réutilisables, appelant des services de manière indépendante. Il devient alors possible de représenter deux échelles d'un système sur un même diagramme. On regrette malheureusement l'absence d'outils permettant une intégration de la production de diagramme avec celle du code.

A titre plus personnel, ce travail de recherche m'a permis de me rendre compte de l'importance de la documentation d'un projet. J'ai eu l'occasion d'avoir des discussions techniques avec des tiers sur le fonctionnement de Buckless, et la présence d'une documentation forte, aussi bien textuelle que schématique a grandement facilité la communication. De plus, la grande diversité des outils et leur utilisation pas forcément automatique m'a fait prendre conscience de l'actualité du sujet. Il est important de faire de la veille sur les différentes stratégies de documentation, car elles sont la porte d'entrée à la production d'un logiciel de bonne qualité. Enfin, la formalisation d'une ligne directrice en terme de documentation, et donc de développement, a donné un vrai coup de fouet au projet, le rendant plus rigoureux et attractif. Je suis désormais confiant quant au temps de formation que devrait suivre un nouveau collaborateur.

RÉFÉRENCES

- [1] UML 2 Component Diagrams : An Agile Introduction, Agile Modeling, *Scott Ambler and Associates*
<http://agilemodeling.com/artifacts/componentDiagram.htm>
- [2] The component diagram
<http://www.ibm.com/developerworks/rational/library/dec04/bell/>
- [3] Swagger, getting started
<http://swagger.io/getting-started/>
- [4] Swagger, specifications
<http://swagger.io/specification/>
- [5] Projects with great wikis, GitHub
<https://github.com/showcases/projects-with-great-wikis>
- [6] TomDoc RFC
<http://tomdoc.org/>
- [7] Donna, A CoffeScript documentation generator, npmjs
<https://www.npmjs.com/package/donna>
- [8] Tello, Digests biscotto metadata, npmjs
<https://www.npmjs.com/package/tello> <https://www.npmjs.com/package/donna>
- [9] GitHub Pages, Websites for you and your projects, GitHub
<https://pages.github.com/>
- [10] Swagger, Allow for multiple response schemas for each response code, GitHub
<https://github.com/OAI/OpenAPI-Specification/issues/270>
- [11] Project Ghost, John O'nolan
<http://john.onolan.org/project-ghost/>