Université Sciences et Technologies - Bordeaux1

Master 2 Informatique : Genie logiciel parcours conduite de projet

Rapport :

Visualisation interactive de topologie de plates-formes parallèles avec lstopo et HTML

Réalisé par : GreenScrum

Encadré par : Philippe Narbel , David Auber.

Client : Brice Goglin

Table des matières

1 Introduction

1.1 Présentation du projet

Le but du projet est de proposer un logiciel de visualisation au format HTML d'une topologie de machines. Cette topologie est constituée d'informations matérielles de chaque machine. Ces données servent à àméliorer les calculs parallèles en optimisant l'utilisation du matériel. Les données à utiliser sont extraites du module lstopo de la bibliothèque logicielle hwloc

1.2 Réalisation du projet

Le projet devait être fait en 2 parties avec une partie en C et une partie Web. Le but de la 1ère partie devait être de créer une exportation des données à visualiser. Mais, après discussion avec le client, nous avons compris que ce n'est pas nécessaire. En effet, une exportation au format XML existe déjà, et elle est suffisante pour fournir les données nécessaires à la création de l'application de visualisation.

2 Architecture

2.1 MVC

L'image ci-dessous réalisée avec le logiciel Enterprise Architect représente un schéma de l'architecture de l'application. Il est important de comprendre que ce schéma n'utilise pas une norme ni même un langage de modélisation en particulier. Il a simplement été conçu pour simplifier la compréhension et les liens existants entre les différentes entités de l'application, il ne peut être considéré comme exactement conforme à la réalité.

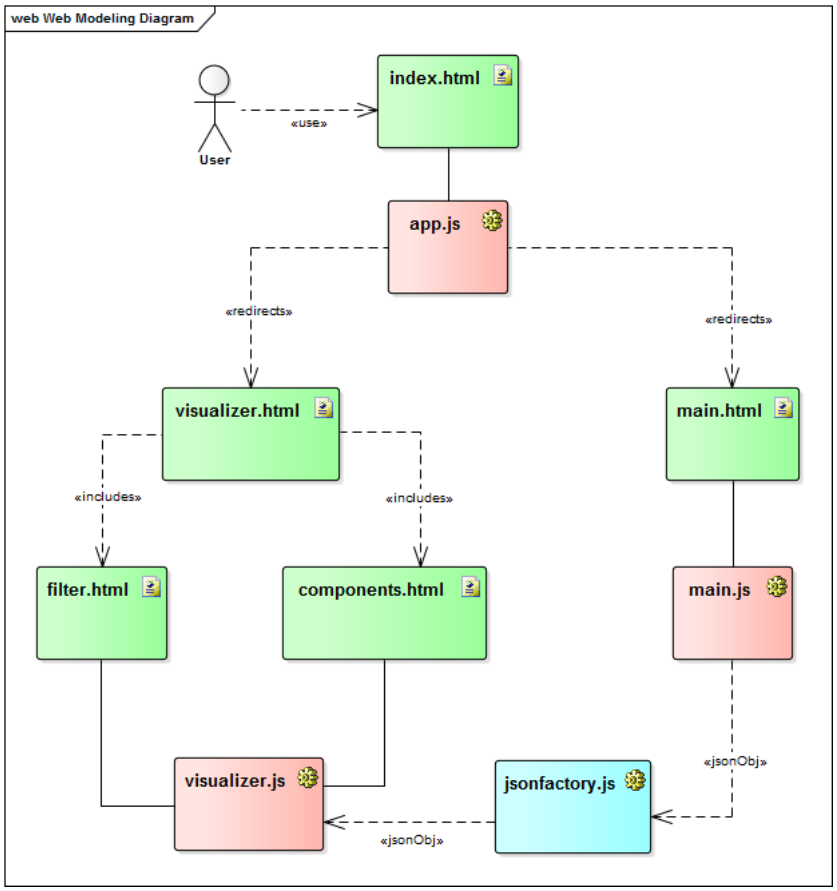


FIGURE 1 – Architecture simplifiée de l'application

D'une certaine façon l'application est organisée suivant une architecture modèle-vue-contrôleur avec quelques particularités. Dans un souci de lisibilité, le schéma utilise un système de couleur. En vert sont représentées les vues qui, dans notre cas, correspondent à des fichiers HTML et en rouge les différents contrôleurs.

Lors de la création du site web, nous avions utilisé une architecture **monopage** (single-page application) avec Angular Js, c'est à dire une application web accessible via une page web unique, contrairement à un site web classique composés de plusieurs pages et donc de plusieurs codes sources.
Cette architecture nous permet de naviguer sur le site de la même manière qu'un site web classique mais sans avoir à recharger notre page à chaque action demandée, plus précisément les liens ne rechargent pas la page mais le contenu est modifié au fur et à mesure selon les requêtes.
Pour ce faire il existe deux manières differentes : soit on charge tout les éléments dans un fichier html (template), soit on récupere et affiche dynamiquement les ressources nécessaires selon les requêtes envoyées par l'utilisateur.

Avant de rentrer plus en détails, il est nécessaire de faire le point sur les objectifs des différentes entités présents sur le schéma :

- **index.html** et **app.js** sont en charge du routage.
- **main.html** et son contrôleur **main.js** sélectionnent, récupèrent et traduisent les données du fichier XML.
- **visualizer.html**, **filter.html**, **components.html** et leur contrôleur visualizer.js affichent et permettent d'interagir avec les données issues du fichier XML.
- **jsonfactory.js** en bleu est un service qui transmet les données issues du fichier XML entre les deux contrôleurs décrits précédemment.

L'utilisateur accède à l'application via index.html qui est le layout général. Cette page associée au script app.js gère le système de routage de l'application et va insérer le bon template (main.html ou visualizer.html) en fonction de l'URL demandé et en particulier des ancres.

Dans notre cas, il n'existe pas de choix à proprement parler. Lorsque l'utilisateur lance l'application il est forcément dirigé sur main.html qui est la vue en charge de sélectionner le fichier XML à analyser. Le contrôleur associé main.js va traduire ce fichier XML au format JSON et insérer toutes les informations dans une variable javascript qu'il va transmettre au service jsonfactory.js en vue d'être exploitée par le contrôleur visualizer.js.

La vue visualizer.html inclus deux fragments externes HTML. Le premier, filter.html décrit l'interface graphique avec les différents boutons et autres éléments sur lesquels l'utilisateur peut interagir. Le second n'est autre que components.html qui est la vue pour l'affichage des informations provenant indirectement du fichier XML.

Ces vues sont en relation avec le script visualizer.js qui contient deux contrôleurs. Le premier contient plusieurs types de fonctions :

— Les fonctions permettant d'extraire les données considérées par l'application et contenues jusqu'alors dans la variable javascript du service jsonfactory.js décrit avant. Ces données dites « scopes » (AngularJS) serviront donc de modèles.
— Les fonctions déclenchées par des évènements utilisateurs et celles appelées en réponse.

Ce contrôleur est donc en lien avec filtre.html et une partie de component.html.

Le second contrôleur est entièrement dédié à la partie concernant les cartes d'extension PCI et qui seront visuellement représenté sous la forme d'un arbre. Nous verrons dans une prochaine partie les technologies utilisées pour une telle représentation.

Une demonstration de notre application web est disponible sur ce lien.[3]

2.2 Technologies

AngularJS, le framework javascript de Google est au cœur des relations entres les vues et les contrôleurs décrits précédemment. L'intérêt principal de cet outil est bien entendue la mise en place d'un architecture MVC et ce, même si l'on ne considère que le côté client d'une application.

AngularJS étend le HTML classique via l'apport de directives natives qui ont la forme d'attributs HTML et qui permettent de maîtriser le DOM. Par exemple, à l'aide de la directive ng-repeat dans components.html on va pouvoir itérer indirectement sur les éléments parents du fichier XML de base (des packages ou des groupes par exemple) et appliquer de façon récursive un même template HTML sur les éléments enfants.

La visualisation concernant la partie des caches et des cœurs s'appuie sur du HTML via des balises div et span aidé par du feuilles de style.

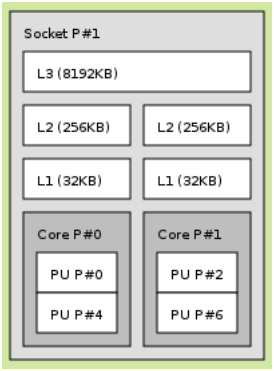


FIGURE 2 – Exemple d'une représentation des caches et des coeurs produit par lstopo

Pour la partie concernant les PCI il a fallut essayer plusieurs outils avec des résultats

très variables.

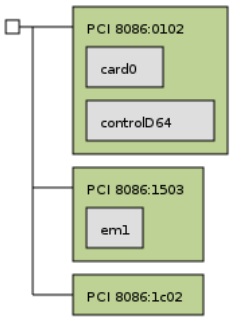


FIGURE 3 – Exemple d'un arbre PCI obtenu avec lstopo

Voici les principales technologies qui ont été testé pour se rapprocher au maximum d'un résultat satisfaisant :

— La bibliothèque graphique javascript **D3.js** spécialisée dans la représentation de données sous forme graphique et dynamique.
— La bibliothèque javascript **JointJS** qui permet la création de diagramme interactif.
— Le composant HTML **canvas** qui permet la réalisation de dessin.

Dans le cas de D3, en utilisant un modèle nativement destiné à la représentation d'arborescence de fichiers (collapsible indented tree) nous sommes parvenus à un résultat proche de celui attendu. La principale différence résidait dans la représentation des entités PCI qui n'était pas exactement la même.

Avec JointJS nous étions également parvenus à un résultat satisfaisant. Cette fois la différence se faisait dans la représentation des arêtes ou des branches qui n'étaient pas tout à fait conforme aux attentes.

Si les solutions avec ces deux outils ont été abandonnées, malgré le fait qu'elles permettaient une interaction, c'est parce que les deux différentes représentations que l'on obtenait était gravement déformées lors de l'exportation en PNG (une fonctionnalité importante de l'application). Tandis que les branches étaient transformées en polygone pleins, les entités étaient dépourvues de contour.

Ces deux bibliothèques utilisent la technologie SVG (scalable vector graphics) pour la représentation des données. Or cela implique une conversion avec canvas avant de pouvoir être exporté sous un format d'image. C'est sans doute cette conversion qui posait problème mais n'ayant pas trouvé de véritable solution nous nous sommes tournée vers la troisième technologie : canvas.

Canvas est un composant qui fait partie de la spécification HTML5 et qui correspond à une zone de dessin. Du code javascript permet ensuite d'accéder à cette zone et d'utiliser toute une série de fonction de dessin comme une API classique.

L'interface graphique de l'application utilise l'outil **Bootstrap** et plusieurs directives AngularJS particulières :

— **angular-bootstrap-colorpicker** pour une sélection poussée des couleurs sur chaque élément.
— **angular-multi-select** pour une liste déroulante des éléments à afficher ou non.

La conversion du fichier XML en JSON pour une manipulation des données en JavaScript est possible grâce à la bibliothèque **x2js** qui à l'intérêt de ne pas avoir de dépendances.

2.3 Fonctionnalités

Pendant l'ensemble du projet nous avons réalisé les deux principales fonctionnalités demandées par le client qui sont :

— La visualisation du fichier xml généré par hwloc pour tous les exemples possibles avec les éléments de base. Elle reprend exactement la représentation que fournissait Hwloc avec l'image produite par celui-ci. Effectivement le but n'était pas d'avoir une représentation différente de celle produite par hwloc mais d'avoir la même en web. Elle est réalisée essentiellement en html/css/Javascript avec utilisation de bordure pour faire les contours de chaque objet.
— La seconde partie consistait à implémenter les options permettant de personnalisé l'affichage du fichier xml généré au besoin du client. Il contient :
  - la possibilité de cacher ou rendre visible certains éléments de la représentation comme les différents caches, la partie PCI ou des plus grosses entités comme des packages, groupes ou autres pour avoir une représentation spécifique ou plus générale.
  - une personnalisation des couleurs pour l'ensemble des objets selon les besoins des clients ou des standards mis en place.
  - changer la taille de police de caractère pour avoir un affichage plus ou moins gros selon la visibilité de base.
  - exporter au format PDF ou PNG l'ensemble des représentations personnalisées par le client. - pour finir la possibilité d'enregistré sur notre machine la configuration des couleurs pour pouvoir la réutiliser plus tard et bien sur la possibilité de charger une configuration existante.

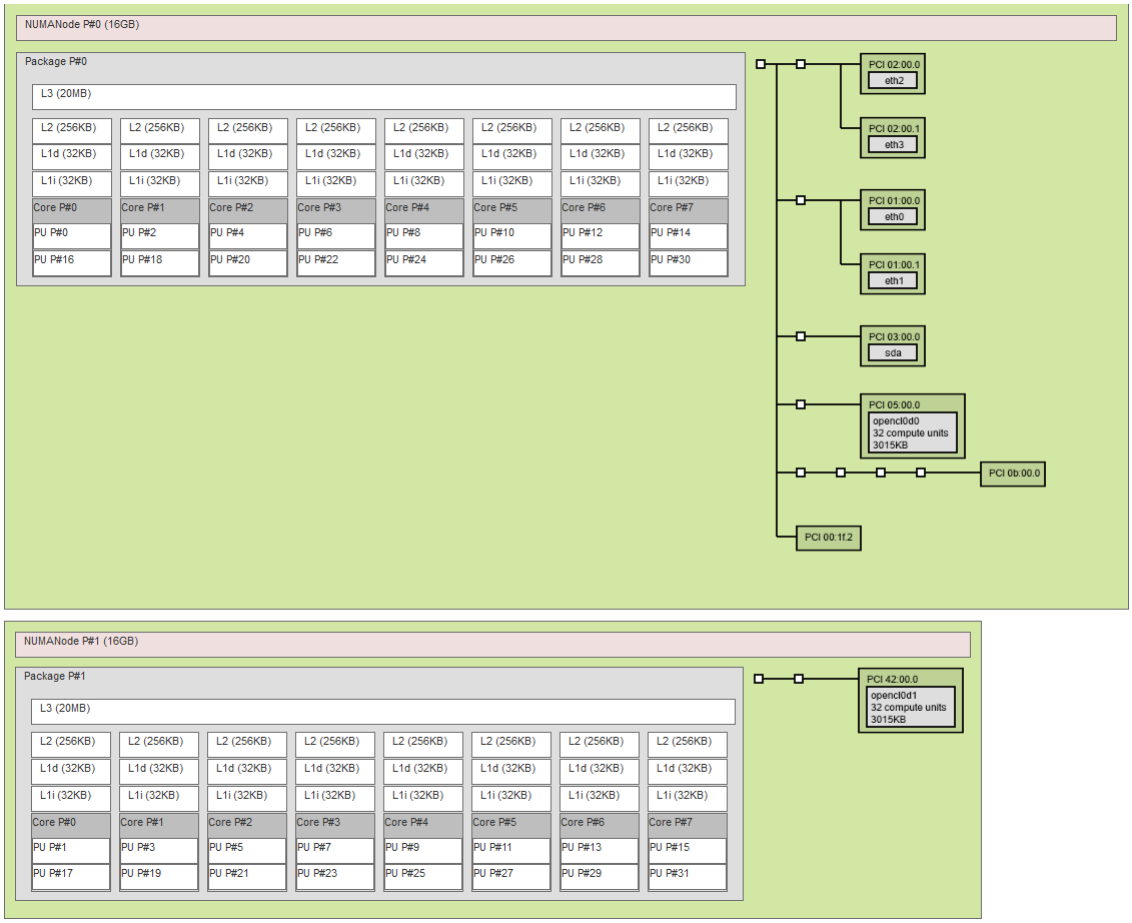


FIGURE 4 – Exemple de représentation pour le xml Alaric

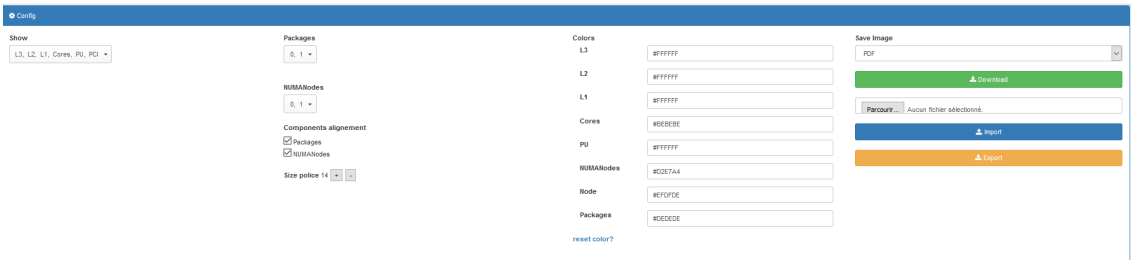


FIGURE 5 – Interface graphique du menu d'option

3 Intégration Continue

Intégration continue est une étape importante à mettre en place dans le processus de développement logiciel. Nous parlerons donc ici de cette pratique et de l'outil utilisé au cours de ce projet.

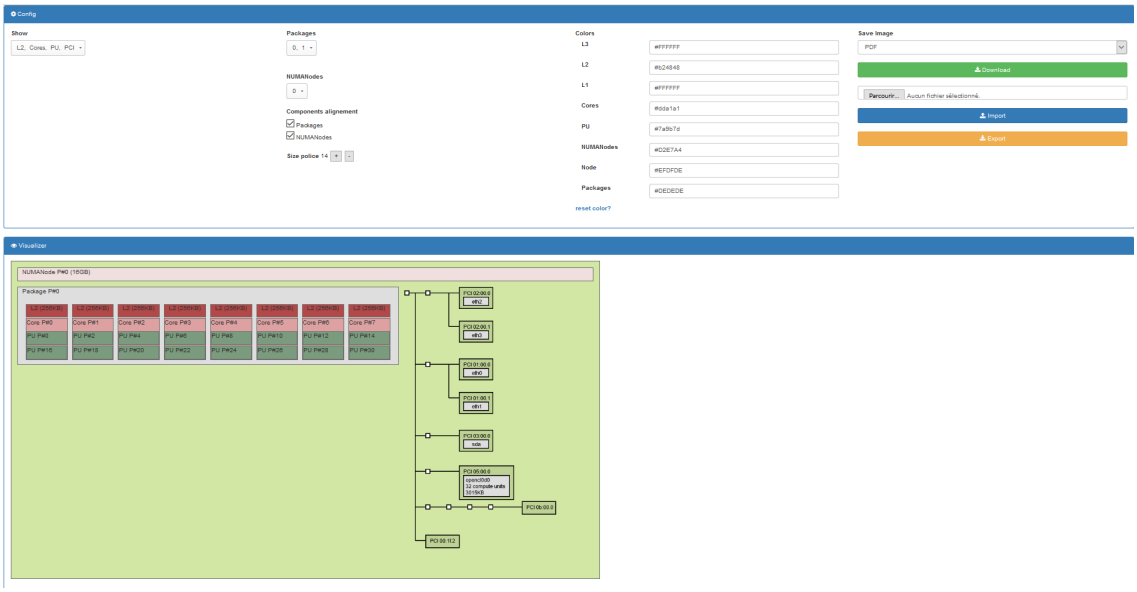
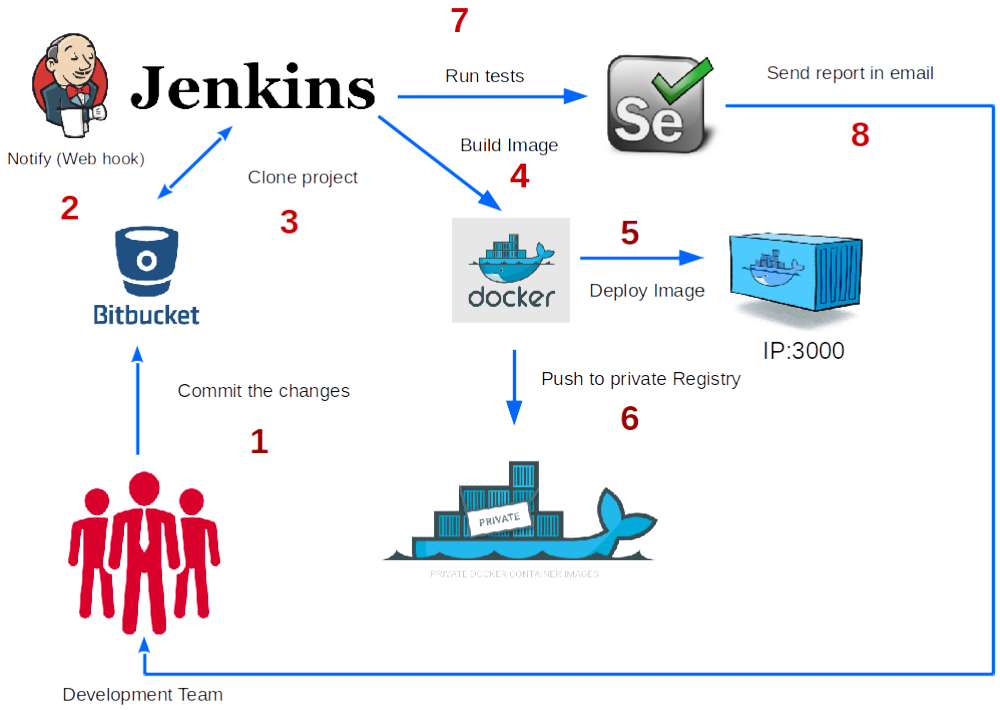


FIGURE 6 – Exemple de représentation pour le xml Alaric avec options



Au moment d'un push d'un ou plusieurs commit au niveau de Bitbucket (sur la

banche **release**) on déclenche via un hook la tâche de build qui permet la construction de l'image Docker à partir de Dockerfile, le push de cette image dans un dépot privé, le déploiement de cette image dans un container exposer sur le port 80 et enfin de passage des TUs au niveau de Jenkins. Dès qu'une modification du code est donc poussée sur le repository, il y a vérification que l'ensemble fonctionne toujours.

```
FROM  mlabouardy/apache
MAINTAINER mlabouardy <mohamed@labouardy.com>

# Copy app
COPY . /var/www/html/ped

# Bundle app source
COPY . /src

EXPOSE  80
```

3.1 Bitbucket

Un système de versionnement, il a été choisi pour héberger les sources des projets internes et clients.[1] Pour notre projet on a utilisé 3 branches :

— **master** utilisé pour la phase de développement
— **release** utilisé pour la phase de production
— **docs** utilisé pour les documents (rapport, maquettes, cadrage ...)

3.2 Jenkins

Serveur d'intégration continue, il est capable d'aller se connecter à un outil de gestion de sources (Dans ce cas Git) et de voir si des modifications ont été effectuées. S'il en détecte, il peut lancer un **build** qui va pouvoir lancer un certain nombre d'actions(déploiement automatisé, tests unitaires ...)

3.3 Docker

Docker est un système de container linux ultra léger basé sur les cgroups, lxc et aufs.

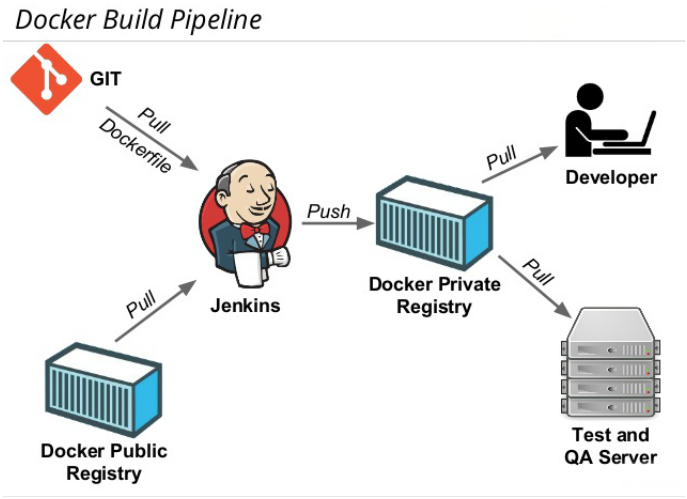
L'idée est la suivante pour un build :

— Jenkins crée une image docker à partir de Dockerfile
— Jenkins push l'image dans un dépot privé
— Jenkins crée un container docker à partir de cette image

Les containers sont running de manière continue, permettant ainsi aux responsables du projet et au client de tester la dernière version du produit poussé manuellement à partir de Jenkins.

3.4 Private Registry

Depot pour stocker l'ensemble des versions de l'application tout au long de la phase de développement

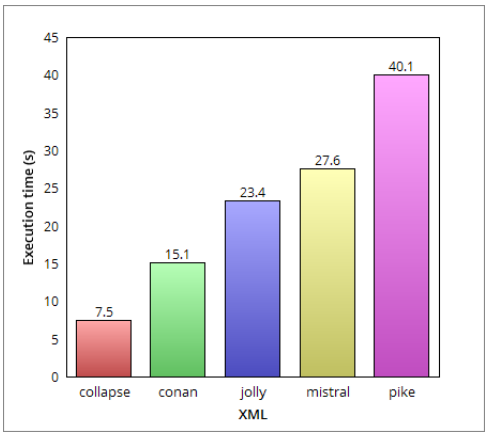


L'avatange d'utilisation d'un docker private registry c'est avoir un systéme de versionning pour l'application. et rendre disponible facilement le dernier exécutable.

3.5 Selenium Webdriver

WebDriver est un framework de tests fonctionnels issu du projet Selenium, célèbre outil d'automatisation de tests pour navigateurs.

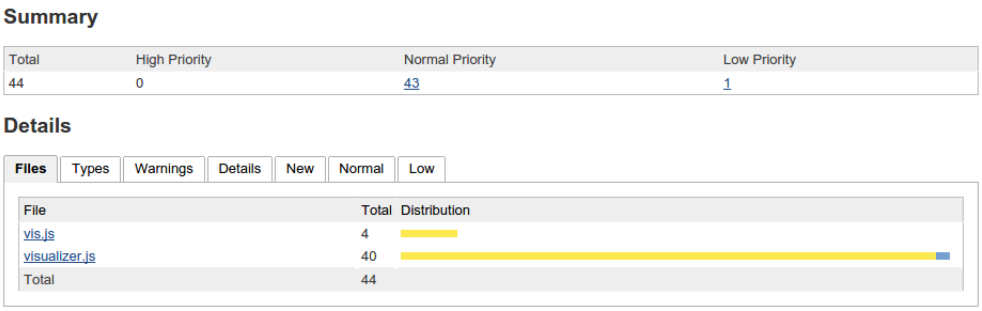
Un exemple des test faites avec l'API Selenium c'est la mesure du temps d'execution pour un ensemble des exemples d'Architecture.



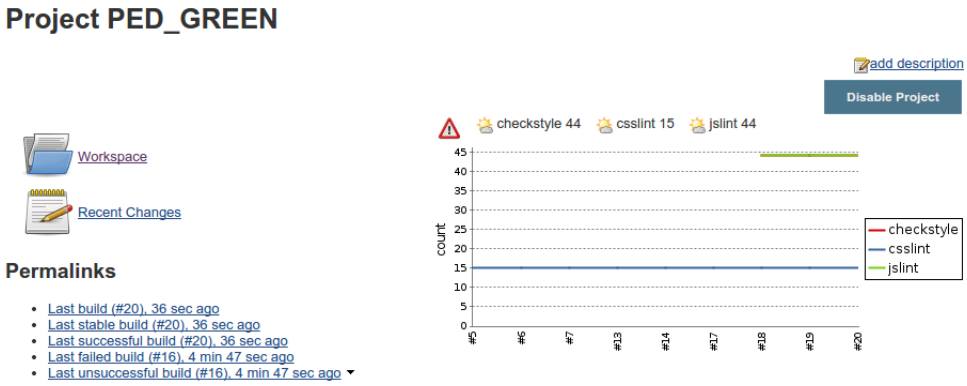
3.6 JSLint/CSSLint

La qualité du code est vitale pour qu'un projet soit pérenne sur le moyen et long terme. De nombreux outils existent pour automatiser les contrôles et générer des rapports statistiques :

JSLint est un analyseur de code Javascript. Son but est de parser le code Javascript pour vérifier que vous respecter les règles de coding Javascript.



CSSLint détecte les problèmes d'une feuille de style CSS.



3.7 AngularJS Batarang

Afin de développer tester, déboguer et surveiller notre application AngularJS, on a installé un plugin sur le navigateur :

Une extension de Google Chrome. Il permet d'observer le code en action, de faire des benchmarks sur les fonctions, modules, etc...

4 Gestion de projet

4.1 Scrum

Durant l'ensemble du projet nous avons suivi la méthode Scrum qui est une méthode agile. Le but était d'être en conditions réelles de projet avec une équipe de 7 personnes. Pour appliquer la méthode Scrum on a utilisé le logiciel Icescrum qui permet de répondre aux besoins de ce genre de projet. Il nous a permis notamment de définir un backlog qui contenait l'ensemble des User Stories à faire pendant la durée du projet, définir différents sprints avec les User Stories à réaliser et les différentes taches de celles-ci. Dans chaque sprint on peut définir le kanban pour les différentes tâches et connaitre l'évolution de leurs taches. Nous avons choisi de réaliser des sprints de deux semaines comme c'était prévu à la base. Pour les taches nous avons choisi différentes couleurs afin de différencier les différents groupes de taches.[2] Il y a :

— Taches jaune : Elles représentent le code produit pour le projet.
— Taches bleu : Elles représentent les tests de validations pour notre code.
— Taches grise : Elles représentent les tests d'intégrations.
— Taches rouges : Elles représentent les recherches concernant certaines bibliothèques non mis en place mais qui peuvent améliorer le projet et le rapport.

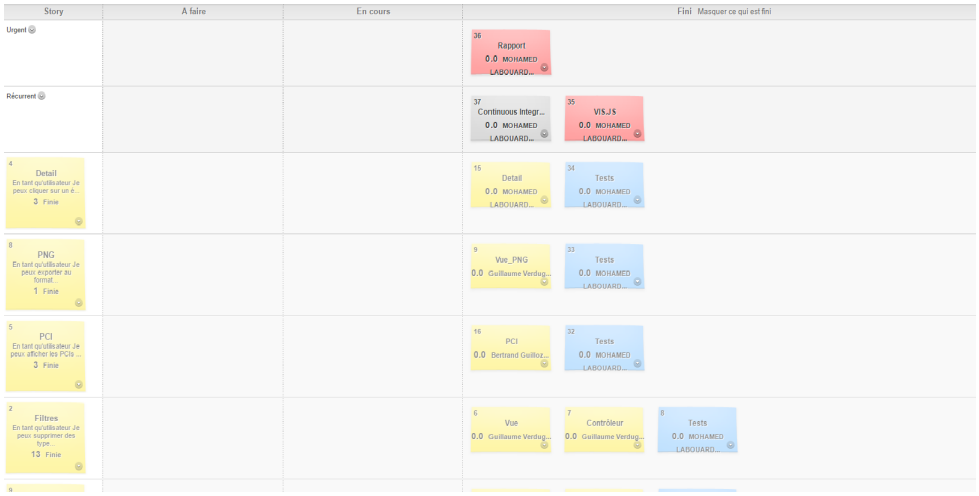


FIGURE 7 – Sprint 2 avec les différentes taches

Par la suite on peut regarder l'évolution du projet avec le burn down chart qui est disponible pour chaque projet et via différent onglet voir les US fini et l'état des sprints.

4.2 Outils complémentaires

5 Critiques

5.1 Technologies

En ce qui concerne les technologies utilisées dans ce projet, on aurait pu utiliser reactjs qui est une bibliothèque développée par Facebook et permet de manipuler un DOM viruel. Et puisqu'on n'avait pas d'expériences avec cette bibliothèque, on a choisi de travailler avec angularjs.

5.2 IceScrum

Ce logiciel qui nous a été proposé durant le cycle de développement du projet et qu'on a due l'implémenter pour la gestion des taches au niveau des équipes était plus ou moins efficace à cause de sa lenteur et le temps mis pour la compréhension et l'adaptation avec notre besoin.

5.3 Performances

On peut trouver plusieurs pratiques qui peuvent nous permettre d'améliorer les performances de l'application, par exemple :
- L'utilisation du cache du navigateur qui consiste à garder en mémoire des copies des pages servies. Cela permettra d'éviter la répétition des traitements à chaque fois et gagner au niveau du temps du chargement de la page web.
- Le One time binding (: :value) qui permet d'indiquer que le binding ne se fasse qu'une seule fois afin de gagner au niveau performance.

5.4 Améliorations possibles

En plus des améliorations citées bien avant, on peut aussi ajouter d'autres propositions possibles, notamment :
- Avoir une application responsive, c'est-à-dire offrir plus de réactivité à l'application afin qu'elle s'adapte aux différentes résolutions d'écrans disponibles.

Références

[1] https://bitbucket.org/berguizzz/ped_green.

[2] http://vps166100.ovh.net:54321/m2scrum/p/LSTOPO.

[3] GreenScrum. lstopo - show the topology of the system provides a detailed explanation of the hwloc system. http://vps259935.ovh.net:3000/ped.