Université Sciences et Technologies - Bordeaux1

Master 2 Informatique : Genie logiciel parcours conduite de projet

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

Réalisé par : Youssef Chagtab, Gregory Emirian, Bertrand Guillozet, Mohamed Labouardy, Clovis Nobile, Ouadii Tabich et Guillaume Verdugo.

Responsables : Philippe Narbel , David Auber.

Client : Brice Goglin

Table des matières

1 Introduction

Le projet décrit dans ce rapport s'inscrit dans le cadre du projet d'étude et développement du second semestre du Master 2 Informatique.

Ce projet est lié à **hwloc** (Hardware Locality) qui est une bibliothèque logicielle exposant de manière portable et abstraite la topologie des machines, en terme de coeurs, caches partagés, threads, sockets, noeuds NUMA, etc [**?**].

Hwloc est notamment utilisée par la plupart des bibliothèques de calcul parallèle de type **MPI** et **OpenMP** afin de maîtriser le matériel et ainsi mieux l'exploiter en tenant compte des affinités entre différents composants et entre composants et tâches de calcul.

L'un des constituants le plus célèbre de hwloc est l'outil **lstopo** qui permet de visualiser dans de nombreux formats la topologie de la plate-forme telle que analysée par hwloc.

Le projet était initialement composé de deux parties avec une partie en C et une autre qui utilise les technologies Web. Le but de la première partie était de créer une exportation des données à visualiser tandis que la seconde consistait à implémenter des fonctions d'intéraction.

Après une première rencontre avec le client, nous avons convenu que la première partie n'était pas nécessaire étant donné que le fichier XML qui peut être généré par hwloc suffisait pour obtenir les données nécessaires à la création de l'application de visualisation. La seconde partie était donc la visualisation des données avec l'ajout d'options pour une personnalisation du rendu final.

1.1 Fonctionnalités

Les objectifs principaux de ce projet étaient les suivants :

— extraction de données du fichier XML.
— visualisation de ces données.
— implémentation de capacités d'intéraction.

La représentation des données issues du fichier XML est sensiblement la même que celle fournie par hwloc. Effectivement le but n'était pas d'avoir une représentation différente de celle produite par la bibliothèque mais d'obtenir la même et de pouvoir l'exploiter dans un navigateur.

Cette représentation est essentiellement réalisée en HTML, CSS et Javascript avec notamment l'utilisation de bordures pour définir les contours de chaque objet. Nous reviendrons en détail sur les technologies utilisées dans l'une des parties suivantes.

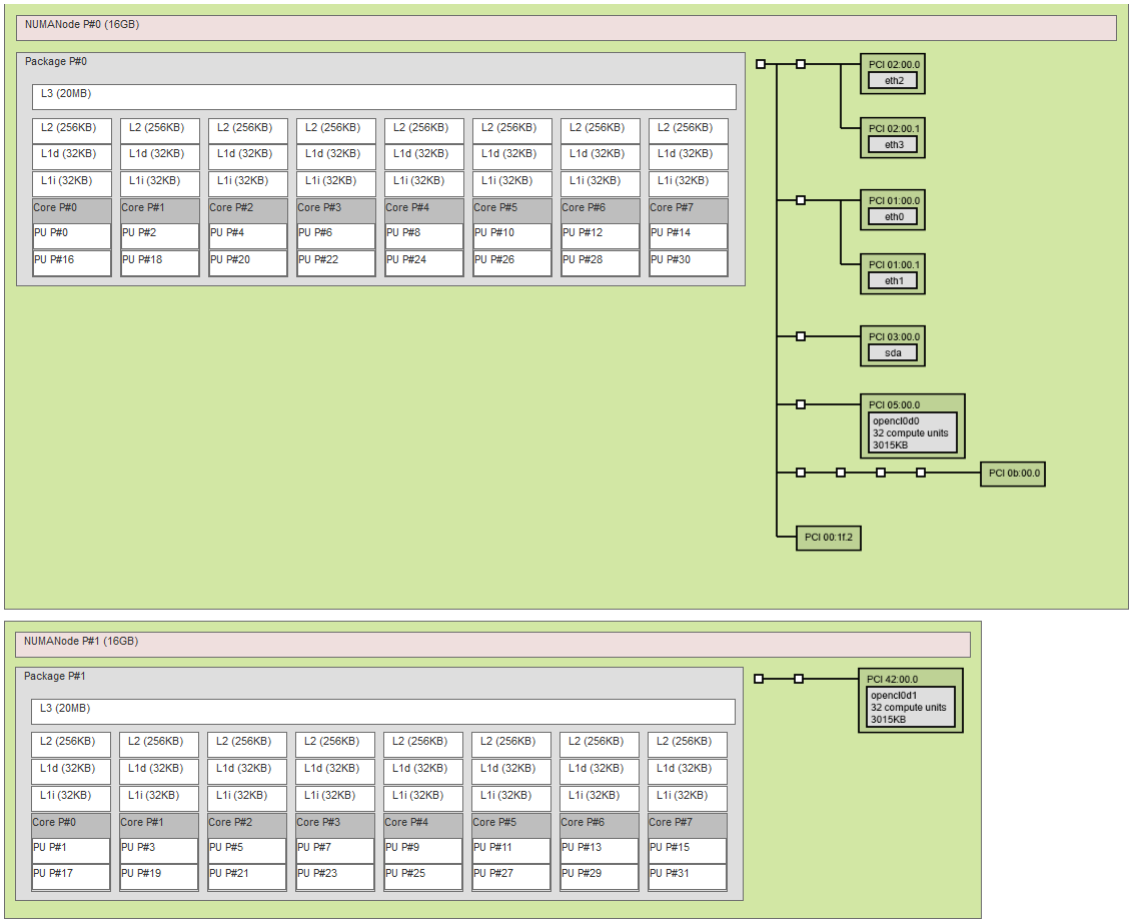


FIGURE 1 – Exemple de représentation pour le XML Alaric

Le second objectif consistait à implémenter des fonctions permettant de personnaliser l'affichage des données issues du fichier XML généré, selon les besoins du client. Voici une liste des principales fonctionnalités d'intéraction :

— la possibilité de cacher ou de rendre visible certains éléments de la représentation comme les différents caches, la partie PCI ou des entités parentes comme des packages, groupes ou autres afin d'avoir une représentation spécifique ou plus générale.
— une personnalisation des couleurs pour l'ensemble des objets selon les besoins de l'utilisateur ou des standards mis en place.
— changer la taille de police pour avoir un affichage plus ou moins important selon la visibilité initiale.
— exporter au format PDF ou PNG l'ensemble des représentations personnalisées par le client.
— la possibilité d'enregistrer sur la machine la configuration des couleurs pour pouvoir la réutiliser plus tard et bien sur la possibilité de charger une configuration préexistante.

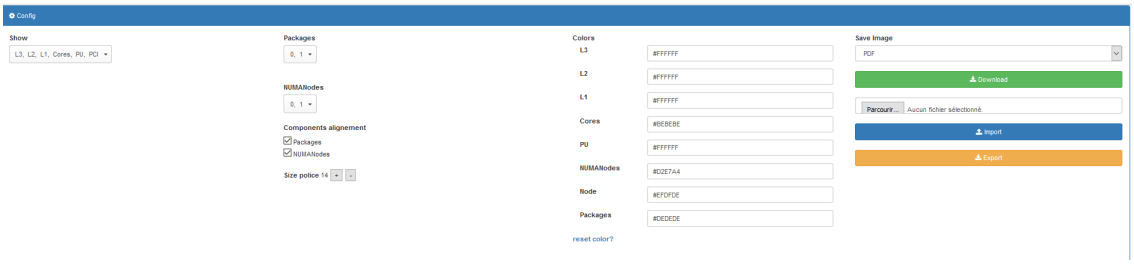


FIGURE 2 – Interface graphique du menu d'option

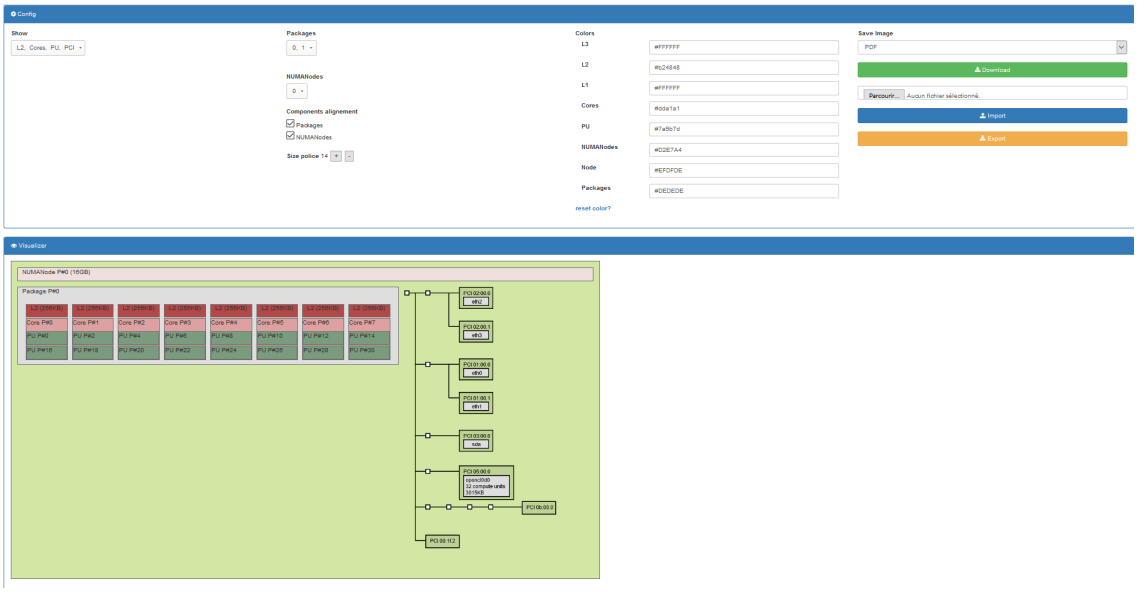


FIGURE 3 – Exemple de représentation pour le xml Alaric avec options

2 Architecture

2.1 Technologies utilisées

AngularJS [**?**], le framework javascript de Google est au cœur des relations entre les vues et les contrôleurs décrits précédemment. L'intérêt principal de cet outil est bien entendu 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 manipuler 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** aidées par des feuilles de style **CSS**.

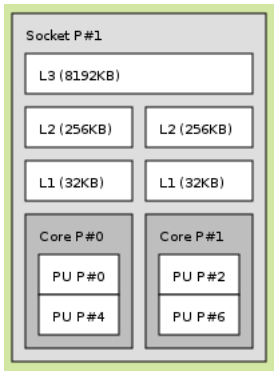


FIGURE 4 – Exemple d'une représentation des caches et des coeurs obtenue avec lstopo

Pour la partie concernant les PCI il a fallu essayer plusieurs outils avec des résultats très variables.

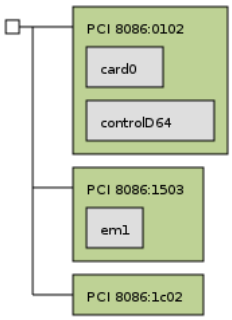


FIGURE 5 – Exemple d'un arbre PCI obtenu avec lstopo

Voici les principales technologies qui ont été testées 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 dessins [**?**].

Dans le cas de D3, en utilisant un modèle surtout destiné à la représentation d'arborescence de fichiers [**?**] 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 que celle de référence.

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 tourné 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 fonctions 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 présente l'intérêt de ne pas avoir de dépendances [**?**].

2.2 Système 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.

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 couleurs. En vert sont représentées les vues qui, dans notre cas, correspondent

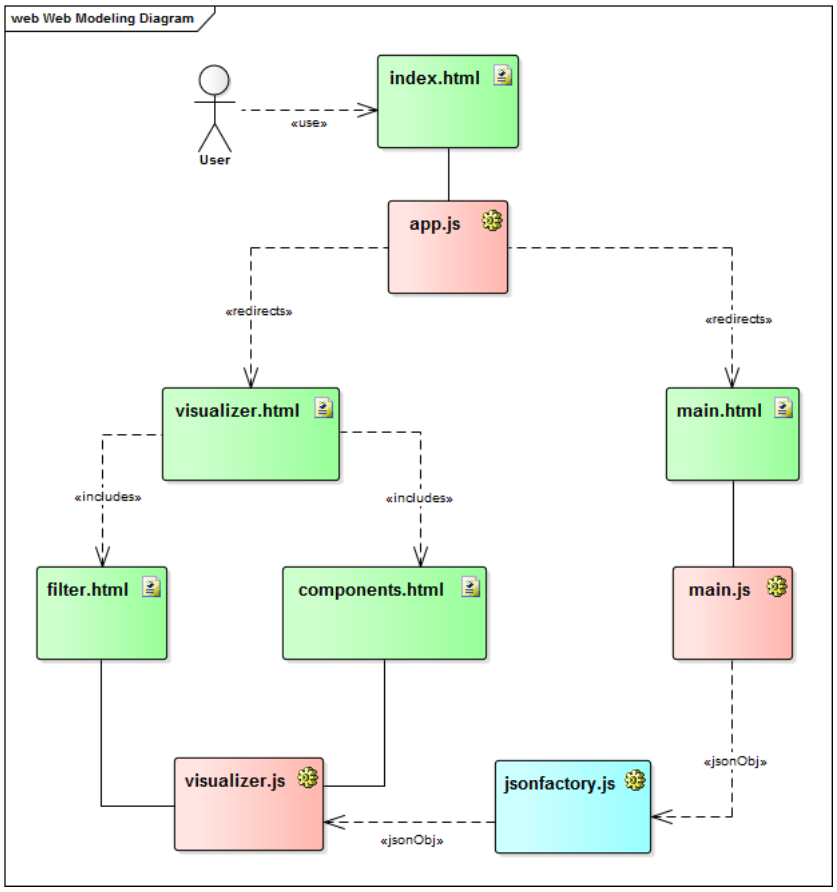


FIGURE 6 – Architecture simplifiée de l'application

à des fichiers HTML et en rouge les différents contrôleurs.

Avant de rentrer plus en détail, il est nécessaire de faire le point sur les objectifs des différentes entités présentes 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 demandée 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 inclu 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.

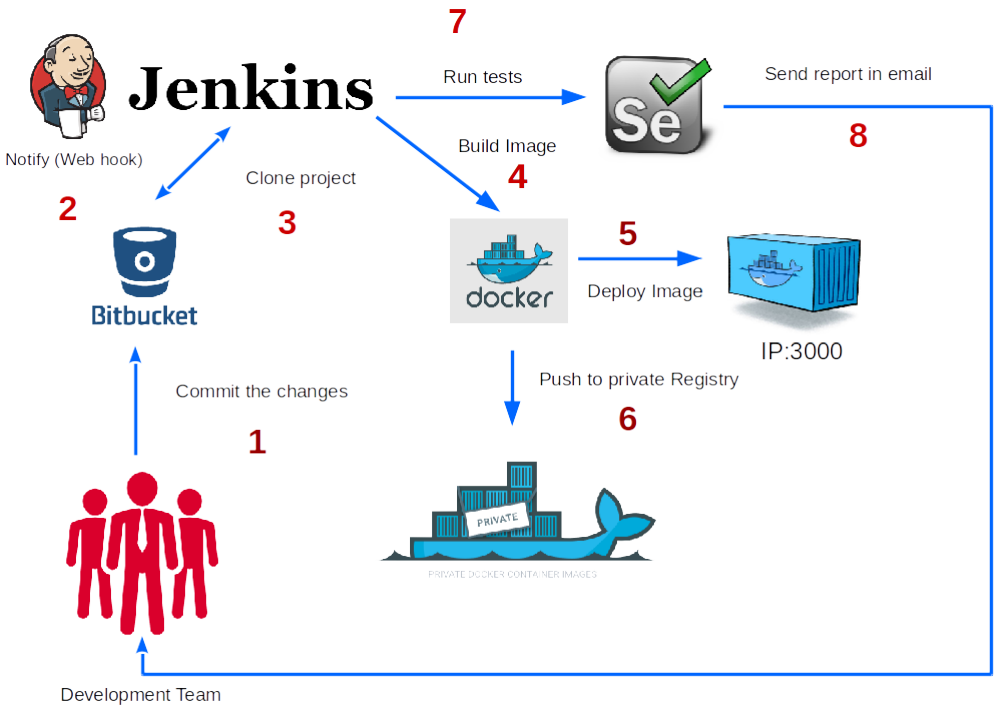
Ce contrôleur est donc en lien avec filtre.html et une partie de components.html.

Le second contrôleur est entièrement dédié à la partie concernant les cartes d'extension PCI qui seront visuellement représentées sous la forme d'un arbre.

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

3 Intégration Continue

L'intégration continue est une étape importante à mettre en place dans le processus de développement logiciel. Nous parlerons donc ici de cette pratique et des outils utilisés au cours de ce projet.



Processus d'intégration continue

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 dépôt privé, le déploiement de cette image dans un container exposé sur le port 80 et enfin de passage des tests unitaires au niveau de Jenkins. Dès qu'une modification du code est donc poussée sur le repository, il y a une 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
```

FIGURE 7 – Le Dockerfile utilisé pour construire l'image

3.1 Bitbucket

Un système de versionnement, il a été choisi pour héberger les sources des projets internes.[**?**] Pour ce projet trois branches ont été utilisées :

— **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 **Bitbucket**) 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**.

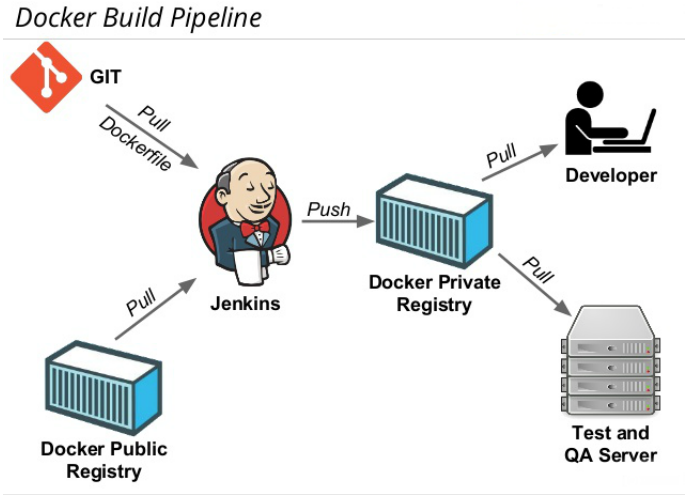
L'idée est la suivante pour un build :

— Jenkins crée une image docker à partir de Dockerfile
— Jenkins push l'image dans un dépôt 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 crée à partir de Jenkins.

3.4 Private Registry

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

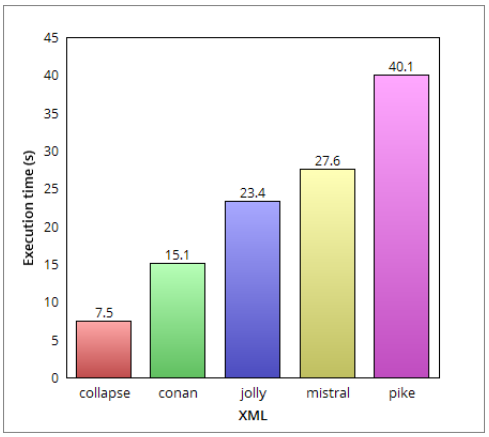


L'avatange d'utilisation d'un **Docker Private Registry** c'est avoir un systéme de versionning pour l'application. et rendre disponible facilement le dernier exécutable.

3.5 Selenium Webdriver

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

Un exemple de test effectué avec l'API Selenium c'est la mesure du temps d'exécution pour un ensemble d'architectures.

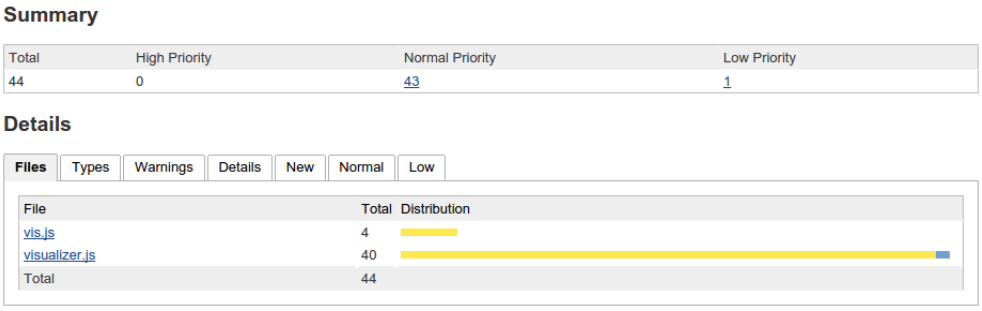


3.6 JSLint/CSSLint

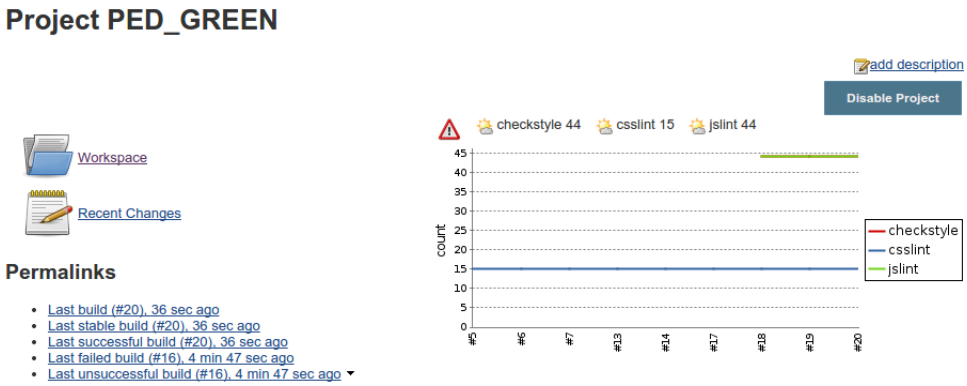
La qualité du code est vitale pour qu'un projet soit pérenne sur le moyen et long terme. De nombreux outils existent pour automatiser les contrôles et générer des rap-

ports statistiques :

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



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 sept personnes.

Pour appliquer la méthode Scrum nous avons 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 réaliser pendant la durée du projet, définir différents sprints avec ces User Stories et les différentes tâches les constituant.

Dans chaque sprint on peut définir le kanban pour les différentes tâches et connaitre leur état. Nous avons choisi de réaliser des sprints de deux semaines comme c'était prévu à la base.

Pour les tâches nous avons choisi différentes couleurs afin de différencier les différents types de tâches :

— Tâches jaunes : elles représentent le code produit pour le projet.
— Tâches bleues : elles représentent les tests de validations pour notre code.
— Tâches grises et rouges : elles représentent les recherches concernant certaines bibliothèques non mis en place, la rédaction de rapport et d'analyse comlémentaire.

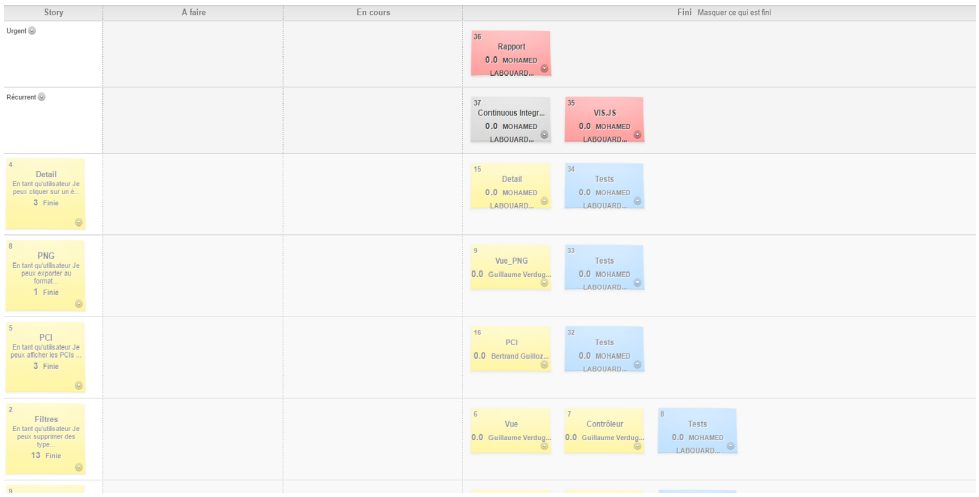


FIGURE 8 – Sprint 2 avec les différentes tâches

Il est important de noter que les noms apparaissant en dessous des tâches ne correspondent pas forcément au responsable de l'entité. Des rencontres régulière ont eu lieux et une seule personne était chargée de modifier le kanban.

4.2 Outils complémentaires

En raison de nombreux problèmes de bug et de ralentissement rencontrés durant tout le projet, nous avons parfois délaissé IceScrum pour créer des kanbans sur des outils différents (Excel).

5 Critiques

5.1 Technologies

En ce qui concerne les technologies utilisées dans ce projet, nous aurions pu nous tourner vers ReactJS qui est une bibliothèque développée par Facebook et qui permet de manipuler un DOM virtuel.

Par manque d'expérience avec cette bibliothèque, nous avons choisi de travailler avec AngularJS.

5.2 Critiques d'IceScrum

Ce logiciel qui nous a été imposé durant le cycle de développement du projet n'était pas assez d'intuitif et comme nous l'avons évoqué précédemment,des ralentissements liés au serveur nous ont parfois empêchés d'intéragir avec nos sprints.

5.3 Performances

On peut trouver plusieurs pratiques qui nous ont permis ou qui pourrait permettre d'améliorer les performances de l'application :

— 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** qui permet d'indiquer que le binding ne se fasse qu'une seule fois afin de gagner au niveau performance.

5.4 Améliorations possibles

Certaines fonctionnalités (non prioritaires) n'ont pas été totalement implémenté tandis que d'autres auraient pu présenter un intérêt :

— l'exportation en PDF.
— amélioration des feuilles de style
— amélioration des arbres PCI qui ne prennent pas en compte toutes les configurations possibles.