Journal de bord   
Dessin collaboratif   
SVG Edit réparti

# Jour 1

Paint collaboratif: <http://miageprojet2.unice.fr/Intranet_de_Michel_Buffa/HTML5_Tutorial>

Cours JavaScript, JQuery, Web Sockets: <http://miageprojet2.unice.fr/Intranet_de_Michel_Buffa/M1_informatique_-_M1_MBDS_-_Option_Web_2.0_2012-2013>

TP Chat Web Socket: [http://miageprojet2.unice.fr/Intranet\_de\_Michel\_Buffa/M1\_informatique\_-\_M1\_MBDS\_-\_Option\_Web\_2.0\_2012-2013/TP\_3\_M1\_IFI-RIF%2c\_2012-2013](http://miageprojet2.unice.fr/Intranet_de_Michel_Buffa/M1_informatique_-_M1_MBDS_-_Option_Web_2.0_2012-2013/TP_3_M1_IFI-RIF%2C_2012-2013)

Travail fourni :

* Prise en main du sujet
* Lecture des tutoriels
* Entretien avec Mr. Buffa

Après rencontre avec M Buffa et réflexion sur le sujet, voici notre première approche du sujet :

Nous avons eu l’idée de créer un web application permettant l’édition simultanée par plusieurs utilisateurs d’un document de dessins. Par exemple, il existe un outil : SVG Edit, codé en JavaScript permettant l’édition de documents dessins. Notre projet consisterait en l’implémentation de cet outil en répartis. C’est-à-dire qu’un groupe de personnes puissent utiliser SVG Edit et répercuter les actions qu’ils font sur un document commun à leurs collaborateurs. De plus, il leur sera offert de pouvoir communiquer via un chat (commun et privé, notion de salles ou sessions).

Cet application tournera sur un serveur web adapté : nodeJS puisque notre application est essentiellement JavaScript.

Le côté réseau sera géré par les web sockets, à l’aide des cours d’introduction de M. Buffa. Cette technologie permet de développer simplement et efficacement les communications full duplex (communication bidirectionnelle).

Afin de pouvoir gérer des pages HTML et CSS sur notre serveur, l’utilisation du module « express » est indispensable.

Le module socket.io est aussi indispensable pour pouvoir travailler avec les web sockets en JavaScript.

Le tout sera utilisé à l’aide de JQuery, le couteau suisse du développeur.

Répartition des tâches:

* Tous : lire les tutoriels de Buffa, apprendre comment fonctionnent les web sockets, JavaScript, JQuery, et SVG Edit.
* Zhou :
* Jean-Sébastien :
* Guillaume : Création du serveur nodeJS avec les modules sockets et express, et installation du serveur sur redhat.
* Adrien : Journal de bord

# Jour 2

Lien décrivant les fonctionnalités de svg-edit

<http://ehmdunque.altervista.org/i-informatica/manuali/SVG-edit/SVG-Edit_2.6/Short_intro_SVG-edit.html>

Nous avons tous regardé les tutoriels de M. Buffa sur JavaScript, JQuery et les Web sockets.

Après nous être familiarise avec ces concepts, nous avons commencé à réfléchir à l’hébergement de l’application.

Nous avons donc ouvert un serveur Openshift proposé par Redhat. <https://www.openshift.com/> à l’URL suivante: <http://svgwebsocket-polytech.rhcloud.com/>

Ce serveur fonctionne sous nodeJS.

Pour pouvoir utiliser ce serveur, il faut uploader dessus le .js implémentant le serveur, ainsi que la page html contenant notre chat et le SVG Edit. Une fois ces fichiers dans le serveur, il faut installer les modules express et socket.io (expliqué au jour 1) via la commande

$ npm install xxx

On note qu’il faut donc utiliser une console avec un accès ssh sur le serveur afin de pouvoir installer ces modules dessus.

Pour lancer le serveur nodeJS il suffit de lancer la commande

$ node xxx.js

Où xxx.js est le fichier JavaScript correspondant au serveur.

Travail effectué :

* Adrien et Guillaume :

Afin de pouvoir tous travailler sur un code de façon collaborative, nous avons cherché un IDE permettant directement de déployer notre code sur notre serveur Openshift.

Nous avons essayé de faire fonctionner cloud9, qui travaille en collaboration avec Redhat, mais depuis quelques mois, la fonction permettant de travailler de concert entre Openshift et Cloud9 ne fonctionne plus.

Nous cherchons donc actuellement une solution.

Nous avons aussi essayé de faire fonctionner les solutions basiques du chat sur le serveur Openshift.

* Zhou :

Intégration du module SVG Edit avec le chat. C’est-à-dire créer un .html contenant ces deux parties là. Pour cela, Zhou doit apprendre comment manipuler SVG Edit, et comment le configurer (taille, intégration dans un iframe, canvas…)

* Jean-Sébastien :

Jean-Sébastien se concentre sur le fonctionnement des web sockets. Il a créé un serveur JavaScript minimal contenant les fonctions dont nous aurons besoin afin de faire communiquer nos clients.

Nous disposons donc de web sockets permettant de communiquer avec le serveur nodeJS en communiquant avec du JSON.

# Jour 3

Création du fichier annexe utilisation des socket.io, expliquant le fonctionnement des web sockets dans notre serveur.

Mise en ligne du site avec le chat et la version de Svg Edit.

Etude du code de Svg Editor.

# Jour 4

Adrien va s’occuper de compléter le chat de façon à ajouter les fonctionnalités de sélection de documents (qui remplacent les rooms), et la liste des participants.

Ces fonctionnalités devront être implémentées avec l’aide des web sockets.

Ajout d’une configuration pour express afin de pouvoir utiliser les link des css et les src pour l’iframe contenant l’editor.

//configure everything, just basic setup

app.configure(**function**(){

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'jade');

app.use(express.favicon());

app.use(express.logger('dev'));

app.use(express.bodyParser());

app.use(express.methodOverride());

app.use(app.router);

app.use(express.**static**(\_\_dirname));

});

# Jour 5

Adrien et Zhou ont modifié le code de la ChatRoom pour lui rajouter des fonctionnalités. Ils ont modifié l’interface avec HTML et CSS.

Réunion avec Michel Buffa :

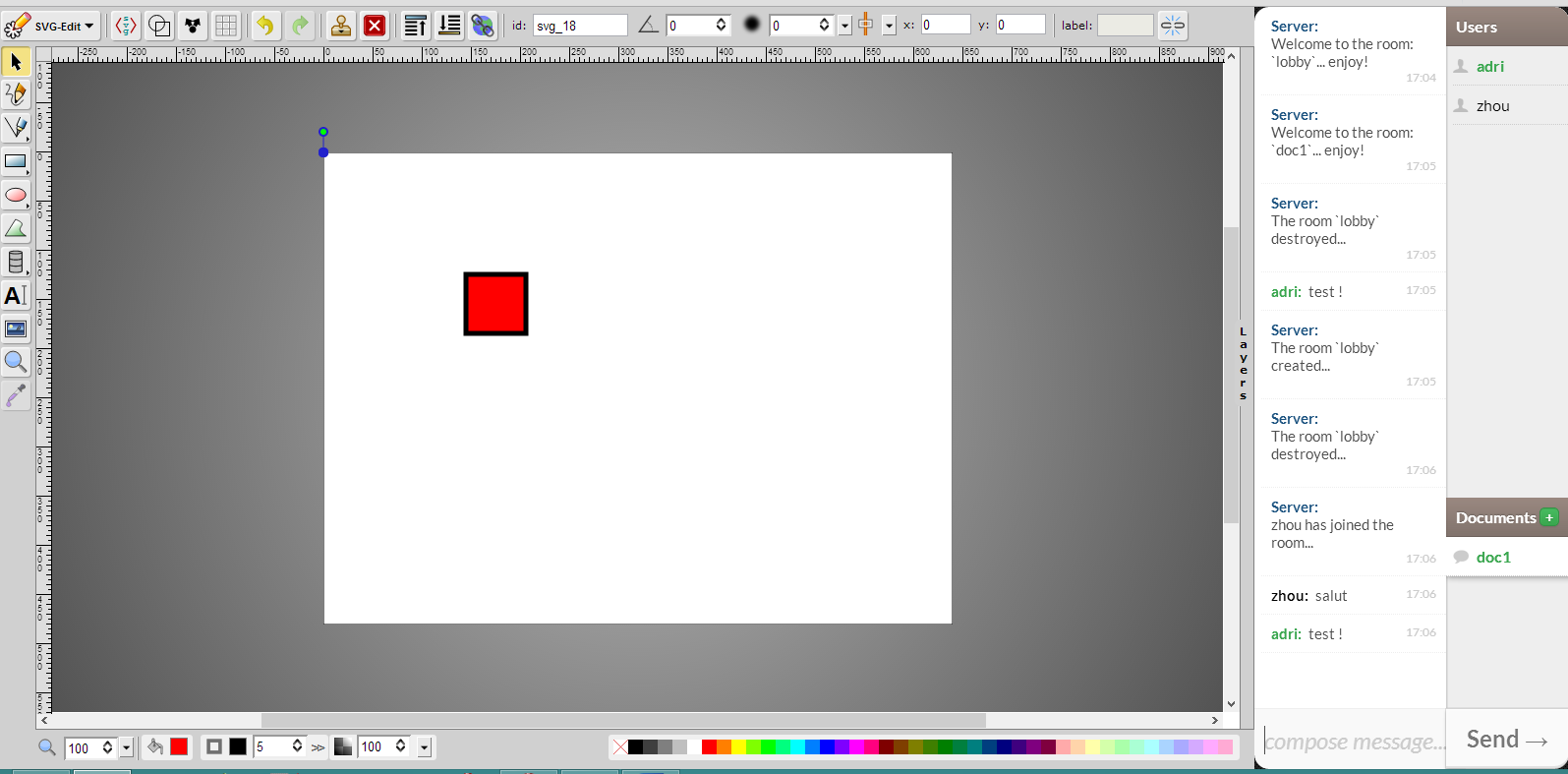
On a étudié le code de SVG Edit pour chercher où est-ce que les objets SVG sont créés. Une fois qu’on les aura trouvés, il faudra rajouter du code pour les envoyer avec les web sockets.

# Jour 6 – Lundi

Pendant le week end, nous avons terminé les fonctionnalités du Chat.

Nous avons mis en ligne la version finale (niveau IHM) de l’application avec les web sockets.

Voici à quoi cela ressemble :



Les fonctionnalités du Chat sont :

* Lobby : quand on n’a pas créé ou rejoint de documents existant, on se retrouve dans cette salle d’attente
* Création de document, ajout dans la liste, et on peut sélectionner n’importe quel document
* Liste des utilisateurs connectés sur le même document

Pour l’instant, le chat et les documents ne sont pas reliés. L’objectif maintenant, est de faire en sorte qu’à la connexion à un document, les SVG se synchronisent, puis que lors d’une modification, elle soit répercutée.

Pour pouvoir synchroniser les documents lorsqu’un nouvel utilisateur entre dans le document, il faut utiliser

// Function: setSvgString

// This function sets the current drawing as the input SVG XML.

//

// Parameters:

// xmlString - The SVG as XML text.

//

// Returns:

// This function returns false if the set was unsuccessful, true otherwise.

Le serveur utilisera la fonction inverse pour lui donner le SVG en texte :

// Function: importSvgString

// This function imports the input SVG XML as a <symbol> in the <defs>, then adds a

// <use> to the current layer.

//

// Parameters:

// xmlString - The SVG as XML text.

//

// Returns:

// This function returns false if the import was unsuccessful, true otherwise.

# Jour 9 – Jeudi

L’API fournie pour les extensions de SVG Edit déclenche un évènement « elementChanged » quand un élément du SVG a changé. Cependant, cet évènement peut correspondre indifféremment à la modification d’un élément (redimensionnement, déplacement, etc.) ou à sa suppression. Dans les deux cas, c’est le même évènement qui est appelé et qui renvoie le même résultat (un tableau contenant les éléments qui ont changé).

Ceci est problématique pour l’implémentation de l’outil collaboratif car le code SVG de chaque client doit être synchronisé pour que tout le monde voie la même chose. Il faut donc être capable de faire la différence entre une modification et une suppression.

Pour résoudre le problème, nous avons modifié le code source de SVG Edit pour introduire cette différence en fabriquant un évènement « elementRemoved » qui se déclenche à chaque suppression d’élément.

La première étape a été de créer une fonction elementRemoved. Comme son fonctionnement est proche de celui de la fonction elementChanged, nous avons repris le code de cette dernière que nous avons légèrement adapté.

La seconde étape a été de lier cet évènement à la liste des évènements reconnus par SVG Edit. Il existe pour cela une fonction bind(name, function) prenant en paramètre le nom de l’évènement et la fonction de callback à exécuter quand cet évènement est déclenché :

svgCanvas.bind("removed", elementRemoved);

Nous avons ensuite modifié les méthodes qui appellent cet évènement. Dans le cadre de la suppression d’un élément, il a suffit de trouver la méthode appelée lorsqu’on appuie le bouton Suppr du clavier ou qu’on clique la croix rouge de suppression d’élément dans SVG Edit. Cette méthode contenait une ligne call("changed", selectedCopy); qui déclenchait l’évènement « elementChanged ». Nous l’avons donc changée par call("removed", selectedCopy); pour appeler l’évènement nouvellement créé « elementRemoved ».

Suite à ces modifications, SVG Edit est devenu capable de distinguer une modification d’élément d’une suppression. Pour terminer il a donc suffit donc d’ajouter des actions différentes pour les évènements « elementChanged » et « elementRemoved » dans notre extension web-socket.