**Projet de Sécurité et Cryptographie**

**Etienne Dupriez – Selim Morjane (AIR3)**

**Introduction :**

L’objectif de ce projet est d’implémenter l’algorithme DES complet. Il s’agit d’un algorithme de chiffrement symétrique (par blocs) qui transforme un message en clair **P** avec une clé secrète **K**. Le résultat est un message chiffré **C.** La fonction de chiffrement doit également être inversible

K

K

C

P

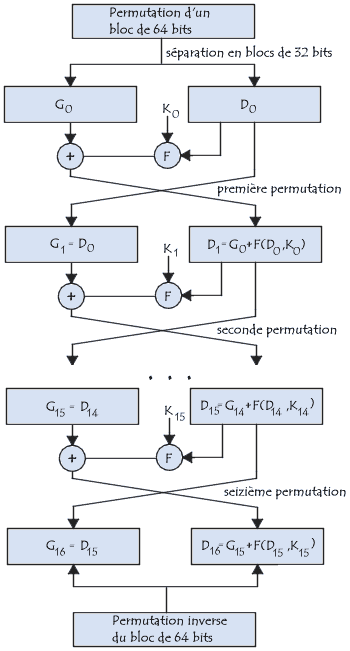
P

**Chiffrement**

**Déchiffrement**

**Implémentation de l’algorithme DES**

Pour cette première partie, nous nous sommes basés sur le schéma ci-dessous et nous avons choisi le langage JAVA pour implémenter l’algorithme.



**Etapes de l’implémentation :**

D’abord, il faut procéder à une permutation initiale qui consiste à échanger les bits du tableau selon une table bien précise (la table *IP* ci-dessous).

***final******static******int******IP****[] = {*

*58, 50, 42, 34, 26, 18, 10, 2,*

*60, 52, 44, 36, 28, 20, 12, 4,*

*62, 54, 46, 38, 30, 22, 14, 6,*

*64, 56, 48, 40, 32, 24, 16, 8,*

*57, 49, 41, 33, 25, 17, 9, 1,*

*59, 51, 43, 35, 27, 19, 11, 3,*

*61, 53, 45, 37, 29, 21, 13, 5,*

*63, 55, 47, 39, 31, 23, 15, 7*

*};*

***public******static*** *BitSet permuter(BitSet tab,* ***int****[] cle)*

*{*

*BitSet tmp =* ***new*** *BitSet(10);*

***int*** *j=0;*

***for*** *(****int*** *i:cle) {*

*tmp.set(j,tab.get(i-1));*

*j++;*

*}*

***return*** *tmp;*

*}*

Le tableau obtenu (bloc de 64 bits en format bitset) est alors coupé en deux blocs de 32 bits chacun : Celui de gauche contient les bits possédant une position paire et celui de droite les bits possédant une position impaire.

***public******static*** *BitSet left\_shift(BitSet tab,* ***int*** *taille)*

*{*

*BitSet left =* ***new*** *BitSet(taille/2);*

***for****(****int*** *i=0; i<taille/2; i++) {*

*left.set(i, tab.get(i));*

*}*

***return*** *left;*

*}*

***public******static*** *BitSet right\_shift(BitSet tab,* ***int*** *taille)*

*{*

*BitSet right =* ***new*** *BitSet(taille/2);*

***for*** *(****int*** *i=taille/2; i<taille; i++) {*

*right.set(i-taille/2, tab.get(i));*

*}*

***return*** *right;*

*}*

Les blocs *left\_shift* et *right\_shift*obtenus sont soumis à un ensemble de transformations itératives appelées *rondes.*

**public** **static** BitSet f(BitSet right\_tab, BitSet cle, **int** taille) {

BitSet rn\_1 = *permuter*(right\_tab, ***E***);

rn\_1.xor(cle);

String concat = "";

**for**(**int** i=0; i<8; i++) {

BitSet B = *bitSetdivider*(rn\_1,i);

BitSet S = *sBox*(B,i+1);

concat += *bitSetTo4String*(S);

}

BitSet concatBit = *bitSetFromString*(concat);

BitSet f = *permuter*(concatBit, ***P***);

**return** f;

}

**Explication :**

D’abord nous allons superposer les bits de la table *right\_shift* en fonction de la table *E* pour l’étendre de 32 bits à 48 bits grâce à la méthode ‘permuter’ vue précédemment : les 48 bits sont mélangés et 16 d'entre eux sont dupliqués.

**final** **static** **int** ***E***[] = {

32, 1, 2, 3, 4, 5,

4, 5, 6, 7, 8, 9,

8, 9, 10, 11, 12, 13,

12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21,

20, 21, 22, 23, 24, 25,

24, 25, 26, 27, 28, 29,

28, 29, 30, 31, 32, 1

};

La matrice résultante (rn\_1) subit un *OU exclusif* avec la première clé. Le résultat de ce *OU exclusif* est ensuite scindé en 8 blocs de 6 bits (grâce à la méthode bitSetdivider que nous avons implémenté).

***public******static*** *BitSet bitSetdivider(BitSet tab,* ***int*** *i) {*

*BitSet divided =* ***new*** *BitSet(6);*

***for****(****int*** *j=0; j<6; j++) {*

*divided.set(j, tab.get((i\*6)+j));*

*}*

***return*** *divided;*

*}*

Chaque bloc passe ensuite par des **fonctions de sélection (sbox)**

Les premiers et derniers bits de chaque nouveau bloc détermine (en binaire) la ligne de la fonction de sélection, les autres bits (respectivement 2, 3, 4 et 5) déterminent la colonne.

StrRange représentera donc les lignes et strColonne représentera les colonnes. Ces valeurs seront converties en base 2. La sélection de la ligne se faisant sur deux bits, il y a 4 possibilités (0,1,2,3). La sélection de la colonne se faisant sur 4 bits, il y a 16 possibilités (0 à 15).

Chacun des 8 blocs de 6 bits doit passer dans la fonction de sélection correspondante, ce qui donne en sortie 8 valeurs de 4 bits chacune.

Chaque bloc de 6 bits est ainsi substitué en un bloc de 4 bits. Ces bits sont regroupés pour former un bloc de 32 bits.

***public******static*** *BitSet sBox(BitSet tab,* ***int*** *i) {*

*String strRange = String.valueOf(booleantoInt(tab.get(0))) + String.valueOf(booleantoInt(tab.get(5)));*

***int*** *range = Integer.parseInt(strRange, 2);*

*String strColonne = String.valueOf(booleantoInt(tab.get(1))) + String.valueOf(booleantoInt(tab.get(2)))*

*+ String.valueOf(booleantoInt(tab.get(3))) + String.valueOf(booleantoInt(tab.get(4)));*

***int*** *colonne = Integer.parseInt(strColonne, 2);*

***int*** *sboxValue = 0;*

***switch****(i) {*

***case*** *1:*

*sboxValue =* ***S1****[range][colonne];*

***break****;*

***case*** *2:*

*sboxValue =* ***S2****[range][colonne];*

***break****;*

***case*** *3:*

*sboxValue =* ***S3****[range][colonne];*

***break****;*

***case*** *4:*

*sboxValue =* ***S4****[range][colonne];*

***break****;*

***case*** *5:*

*sboxValue =* ***S5****[range][colonne];*

***break****;*

***case*** *6:*

*sboxValue =* ***S6****[range][colonne];*

***break****;*

***case*** *7:*

*sboxValue =* ***S7****[range][colonne];*

***break****;*

***case*** *8:*

*sboxValue =* ***S8****[range][colonne];*

***break****;*

*}*

***return*** *BitSet.valueOf(****new******long****[] {sboxValue});*

*}*

Revenons à la fonction f(), une fois le bloc de 32 bits obtenu et concaténé, il subit une permutation avec la table P (ci-dessous)

***final******static******int******P****[] = {*

*16, 7, 20, 21, 29, 12, 28, 17,*

*1, 15, 23, 26, 5, 18, 31, 10,*

*2, 8, 24, 14, 32, 27, 3, 9,*

*19, 13, 30, 6, 22, 11, 4, 25*

*};*

Le résultat final donné par la méthode f() est soumis à un OU Exclusif avec le tableau *left\_shift* de départ

L'ensemble des étapes précédentes (rondes) est réitéré 16 fois.

A la fin des itérations, les deux blocs *left\_shift* et *right\_shift* sont "recollés, puis soumis à la permutation initiale inverse (*IP\_1*)

***final******static******int******IP\_1****[] = {*

*40, 8, 48, 16, 56, 24, 64, 32,*

*39, 7, 47, 15, 55, 23, 63, 31,*

*38, 6, 46, 14, 54, 22, 62, 30,*

*37, 5, 45, 13, 53, 21, 61, 29,*

*36, 4, 44, 12, 52, 20, 60, 28,*

*35, 3, 43, 11, 51, 19, 59, 27,*

*34, 2, 42, 10, 50, 18, 58, 26,*

*33, 1, 41, 9, 49, 17, 57, 25*

*};*

A présent, nous allons voir à quoi ressemble notre méthode de chiffrement qui contient toutes les méthodes vues précédemment.

***public******static*** *BitSet chiffrer(BitSet message, BitSet cle,* ***int*** *taille) {*

*BitSet shifted = cle;*

*BitSet Kn =* ***new*** *BitSet(taille);*

*BitSet IP\_msg = permuter(message,* ***IP****);*

*BitSet L = left\_shift(IP\_msg, 64);*

*BitSet R = right\_shift(IP\_msg, 64);*

*BitSet temp;*

***for****(****int*** *i=0; i<16; i++) {*

***for****(****int*** *j=0; j<****Shift****[i]; j++) {*

*shifted = shift(shifted, taille);*

*}*

*Kn = permuter(shifted,* ***PC2****);*

*BitSet f = f(R, Kn, taille);*

*L.xor(f);*

*temp = L;*

*L = R;*

*R = temp;*

*}*

*BitSet RL = concatBitSet(R, L, 64);*

*BitSet fin = permuter(RL,* ***IP\_1****);*

***return*** *fin;*

*}*

Nous avons également utilisé certaines méthodes au cours de cette implémentation que nous allons expliquer :

La fonction shift() : cette méthode permet de décaller d’un bit vers la gauche les deux moitiés du BitSet en entrée.

***public******static*** *BitSet shift(BitSet tab,* ***int*** *taille)*

*{*

*BitSet left\_tab = left\_shift(tab, taille);*

*BitSet right\_tab = right\_shift(tab, taille);*

*BitSet tmp\_left =* ***new*** *BitSet();*

*BitSet tmp\_right =* ***new*** *BitSet();*

***for*** *(****int*** *i = 0 ; i<taille/2 -1 ; i++) {*

*tmp\_left.set(i, left\_tab.get(i+1));*

*tmp\_right.set(i, right\_tab.get(i+1));*

*}*

*tmp\_left.set(taille/2 - 1, left\_tab.get(0));*

*tmp\_right.set(taille/2 - 1, right\_tab.get(0));*

*BitSet shift = concatBitSet(tmp\_left, tmp\_right, taille);*

***return*** *shift;*

*}*

Cette méthode est utilisée lors du chiffrement, notamment dans la partie génération de clé et repose sur ce tableau

***final******static******int******Shift****[] = {*

*1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1*

*};*

Elle nous permet ensuite de faire la permutation avec le tableau PC2 fournissant en sortie un bloc de 48 bits, représentant la clé.

***final******static******int******PC2****[] = {*

*14, 17, 11, 24, 1, 5, 3, 28,*

*15, 6, 21, 10, 23, 19, 12, 4,*

*26, 8, 16, 7, 27, 20, 13, 2,*

*41, 52, 31, 37, 47, 55, 30, 40,*

*51, 45, 33, 48, 44, 49, 39, 56,*

*34, 53, 46, 42, 50, 36, 29, 32*

*};*

Nous avons également implémenté quelques méthodes de conversions pour pouvoir manipuler les BITSET : fromStringToBinaryString , stringFromBitSet , stringFromBinaryString , bitSetToInt, bitSetToString…

Enfin, la méthode déchiffrer() permet de faire l’inverse du chiffrement et se présente comme suit :

***public******static*** *BitSet dechiffrer(BitSet message, BitSet cle,* ***int*** *taille) {*

*BitSet shifted = cle;*

*BitSet Kn[] =* ***new*** *BitSet[16];*

*BitSet IP\_msg = permuter(message,* ***IP****);*

*BitSet L = left\_shift(IP\_msg, 64);*

*BitSet R = right\_shift(IP\_msg, 64);*

*BitSet temp;*

***for****(****int*** *i=0; i<16; i++) {*

***for****(****int*** *j=0; j<****Shift****[i]; j++) {*

*shifted = shift(shifted, taille);*

*}*

*Kn[i] = permuter(shifted,* ***PC2****);*

*}*

***for****(****int*** *i=16; i>0; i--) {*

*BitSet f = f(R, Kn[i-1], taille);*

*L.xor(f);*

*temp = L;*

*L = R;*

*R = temp;*

*}*

*BitSet RL = concatBitSet(R, L, 64);*

*BitSet fin = permuter(RL,* ***IP\_1****);*

***return*** *fin;*

*}*

**Deuxième partie : Réaliser un programme client-serveur qui implémente un protocole d’échange sécurisé**

**Introduction**

Dans cette partie, nous allons créer une application client/serveur qui permet à un utilisateur de chiffrer/déchiffrer des messages en toute sécurité.

La sécurité est assurée par un protocole d’échange sécurisé (HTTPS)

Nous avons choisi de développer cette application en utilisant le langage JAVA, le framework SPRING BOOT et le JAVASCRIPT.

Cette application répondra aux fonctionnalités suivantes :

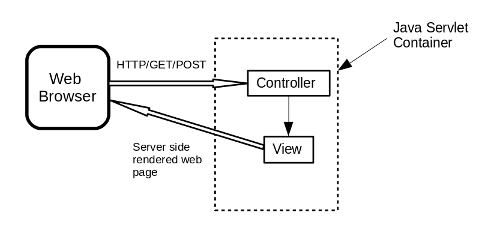
- Les échanges doivent être sécurisés dans les deux sens

- Les messages échangés doivent être supérieurs à la taille du bloc

- Utiliser le mode ECB

**Architecture de l’application :**

Notre application suit l’architecture suivante :



Au niveau du serveur, nous avons nos méthodes construites lors de la partie 1 du projet.

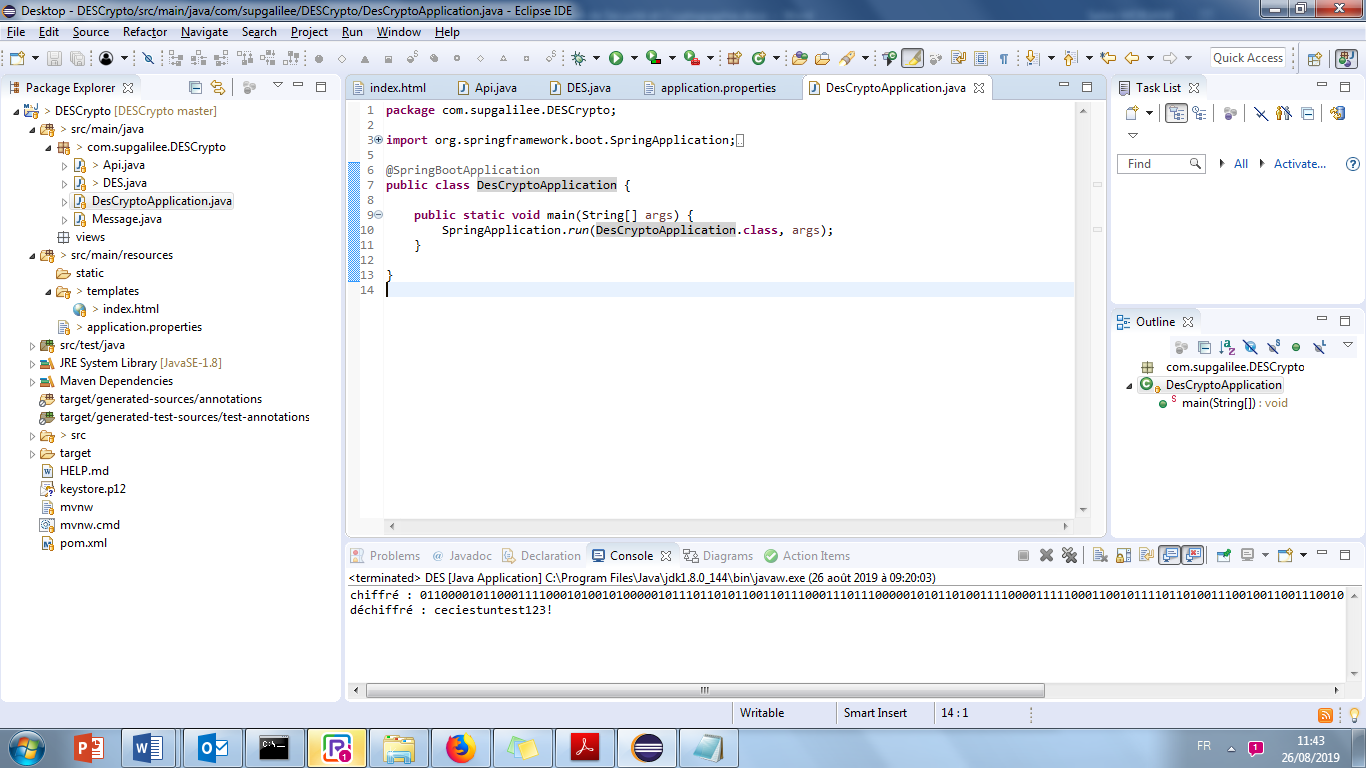
Nous avons adapté notre projet au framework SPRING boot qui suit le modèle MVC :

Le modèle sera le message transmis par l’utilisateur.

Le contrôleur sera un contrôleur REST *(@restController)* qui nous permet d’envoyer des requêtes POST et de récupérer des informations saisies par l’utilisateur via l’annotation @ResponseBody.

Les vues seront en HTML, Javascript, JQUERY, et CSS.

Le point d'entrée de l'application sera



**Configurations :**

Pour assurer un échange sécurisé, Spring boot nous permet d’activer HTTPS dans notre application en générant un certificat SSL manuellement.

Pour ce faire, nous allons d’abord exécuter la commande suivante :

keytool -genkeypair -alias cle -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore keystore.p12 -validity 3650

Et suivre les étapes de la configuration manuelle. Ces configurations seront ensuite implémentées dans le fichier application.properties de l’application sous la forme :

server.port: 8443

server.ssl.key-store: keystore.p12

server.ssl.key-store-password: selim123

server.ssl.keyStoreType: PKCS12

server.ssl.keyAlias: cle

Une fois que nous avons choisi, le numéro de port, nous pouvons générer une action de telle sorte que lorsqu’on se connecte via l’adresse sécurisée <https://localhost:8443/> nous arriverons à établir une connexion sécurisée entre le client et le serveur et ce dernier pourra renseigner sa clé publique. (Nous avons choisi de saisir la clé manuellement)

La clé:

**static** String *cle* = "0001001100110100010101110111100110011011101111001101111111110001";

Au niveau du contrôleur :

@RequestMapping(method = RequestMethod.***POST***, value = "/hello")

**public** String showHello(@RequestBody String m) {

**return** DES.*cle*;

}

Au niveau de la vue :

(function(){

$("#user-form").hide();

// Envoi du message HELLO

$.ajax({

url: "/hello",

method: "post",

data: {

message: "HELLO"

}

}).done(function(data){

$("#cle-publique").append(data);

$("#user-form").show();

});

Voici à quoi ressemblent les échanges réseau :

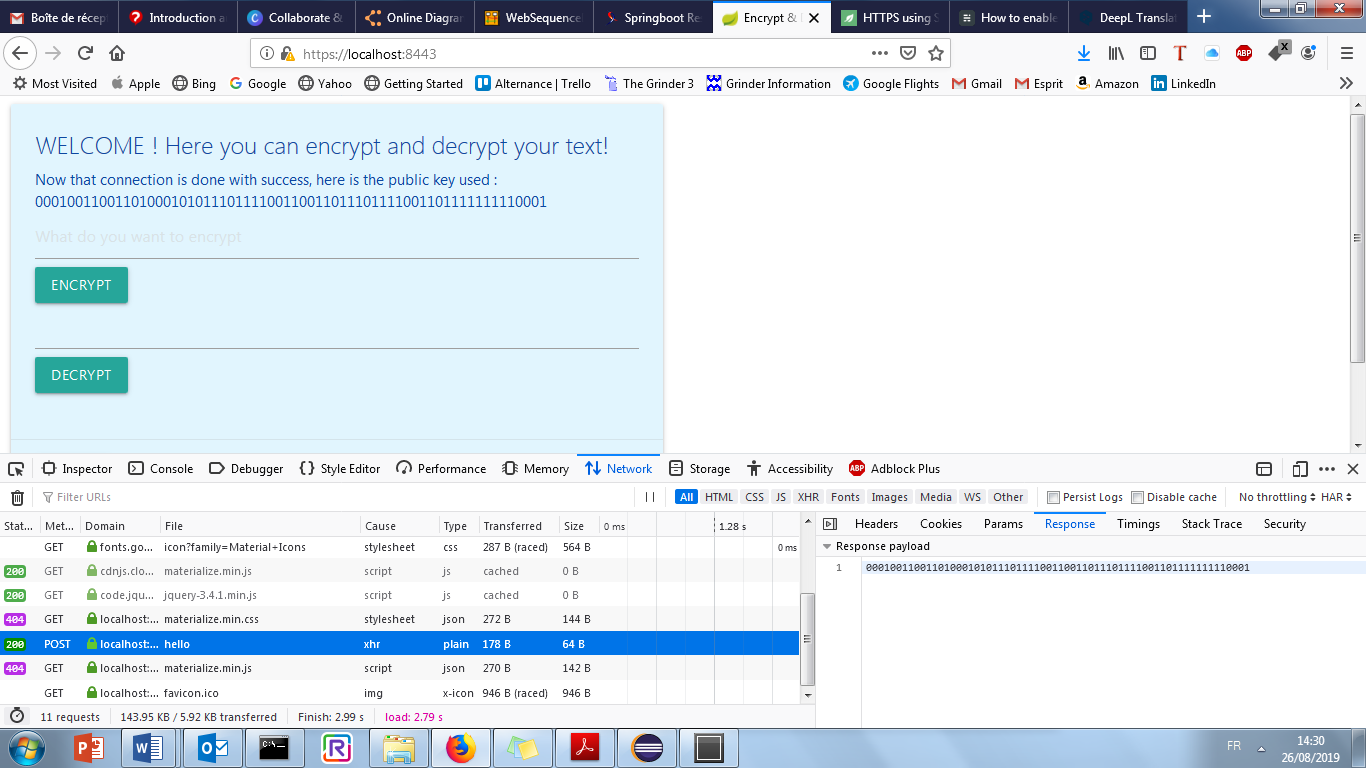
*Request URL:https://localhost:8443/hello*

*Request method:POST*

*Remote address:127.0.0.1:8443*

*Status code: 200*

*Version:HTTP/1.1*



**Cryptage/Décryptage :**

Concernant la partie cryptage et décryptage, nous avons d’abord créé nos méthodes de type POST au niveau du contrôleur et nous faisons appel à nos méthodes implémentées lors de la partie 1 du projet.

@RequestMapping(method = RequestMethod.***POST***, value = "/encrypt")

**public** String encryptMessage(@RequestBody String message) {

**return** DES.*crypt*(message);

}

@RequestMapping(method = RequestMethod.***POST***, value = "/decrypt")

**public** String decryptMessage(@RequestBody String crypted) {

**return** DES.*decrypt*(crypted);

}

Et c’est au niveau de JQUERY que les affichages se passent : Nous récupérons le message saisit par l’utilisateur, nous l’envoyons au contrôleur pour qu’il puisse le crypter/décrypter et nous l’affichons dans le bon endroit.

$("#user-form").submit(function(e){

e.preventDefault();

$.ajax({

url: "/encrypt",

method: "post",

data: {

message: $("#message").val()

}

}).done(function(data){

$("#message-crypted").val(data);

});

});

$("#decrypt-it").submit(function(e){

$.ajax({

url: "/decrypt",

method: "post",

data: {

crypted: $("#message").val()

}

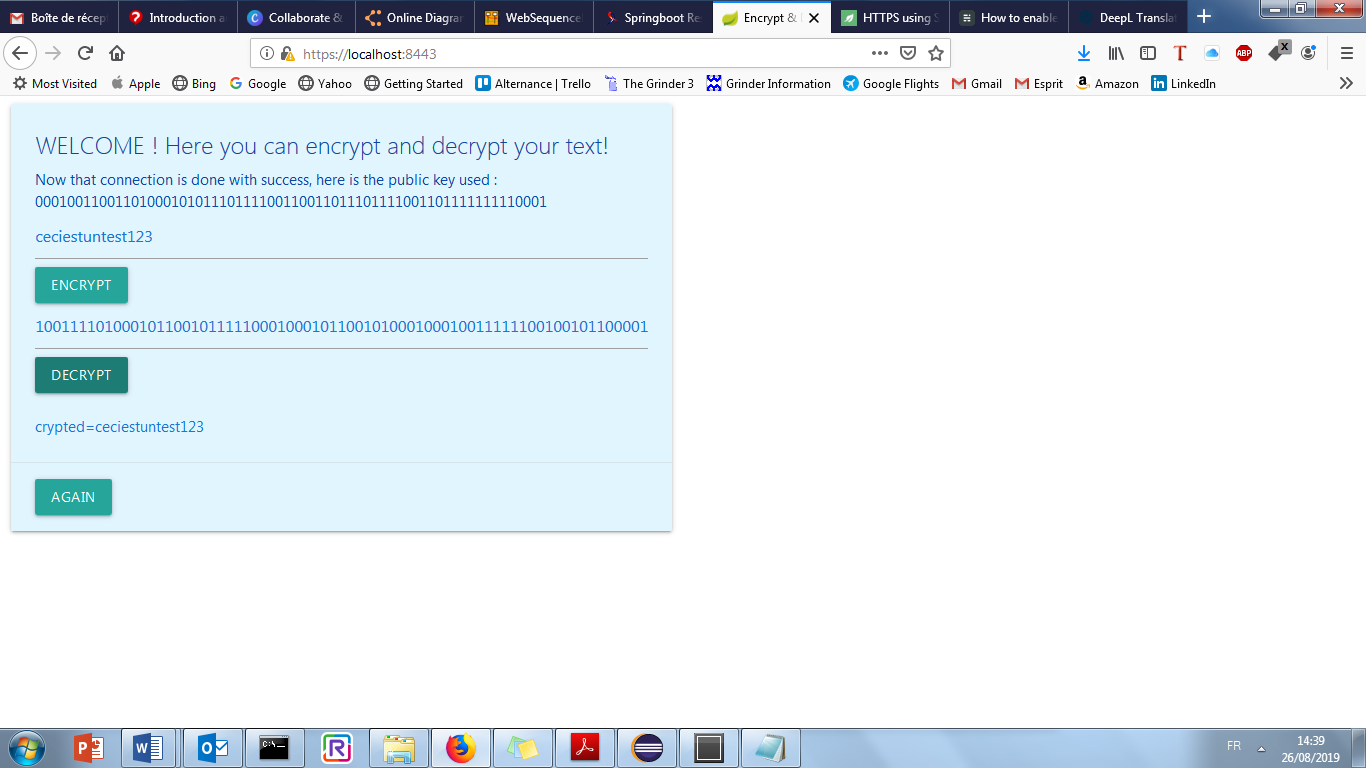
}).done(function(decrypteddata){

$("#final").append(decrypteddata);

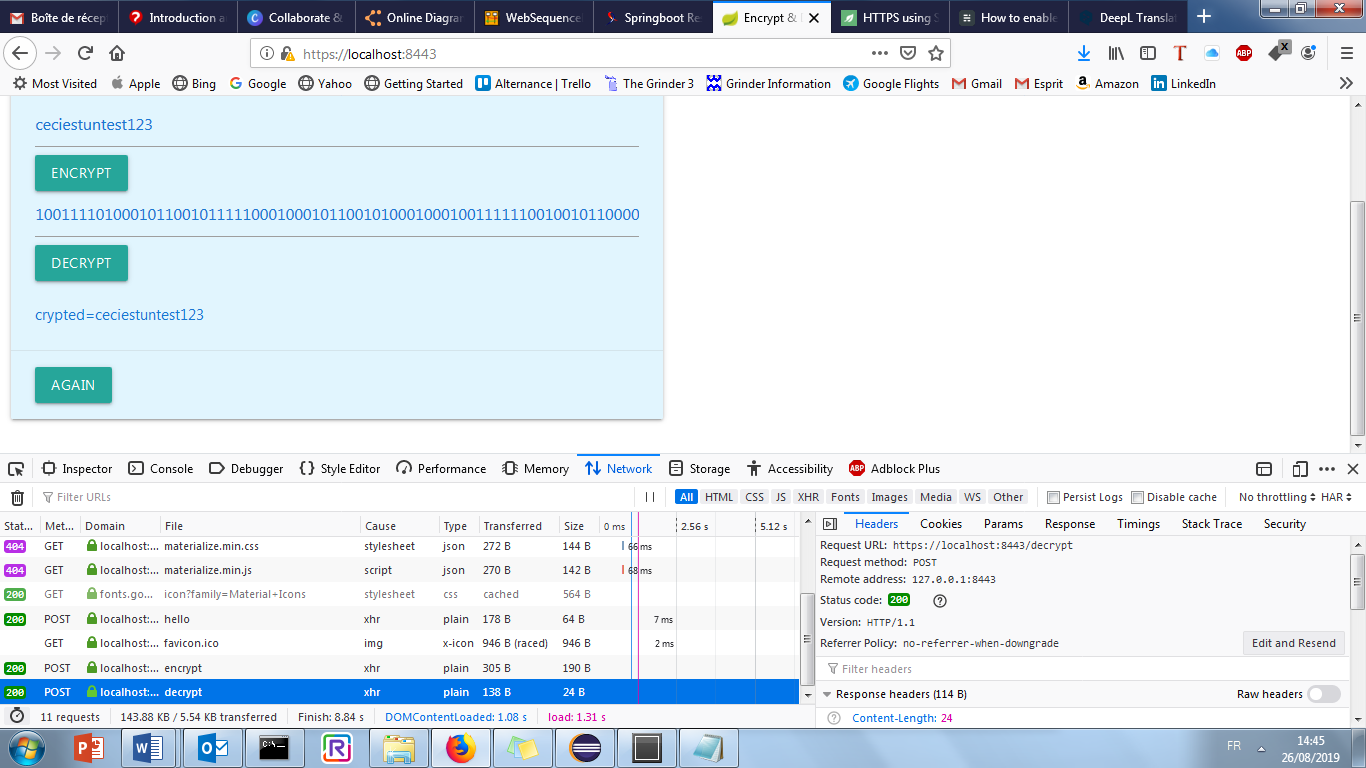
});

e.preventDefault();

})

Ce processus, intégré avec du CSS (notamment la bibliothèque Materialize) nous donne l’affichage suivant : 

Pour suivre les échanges réseaux :



**Gestion de session :**

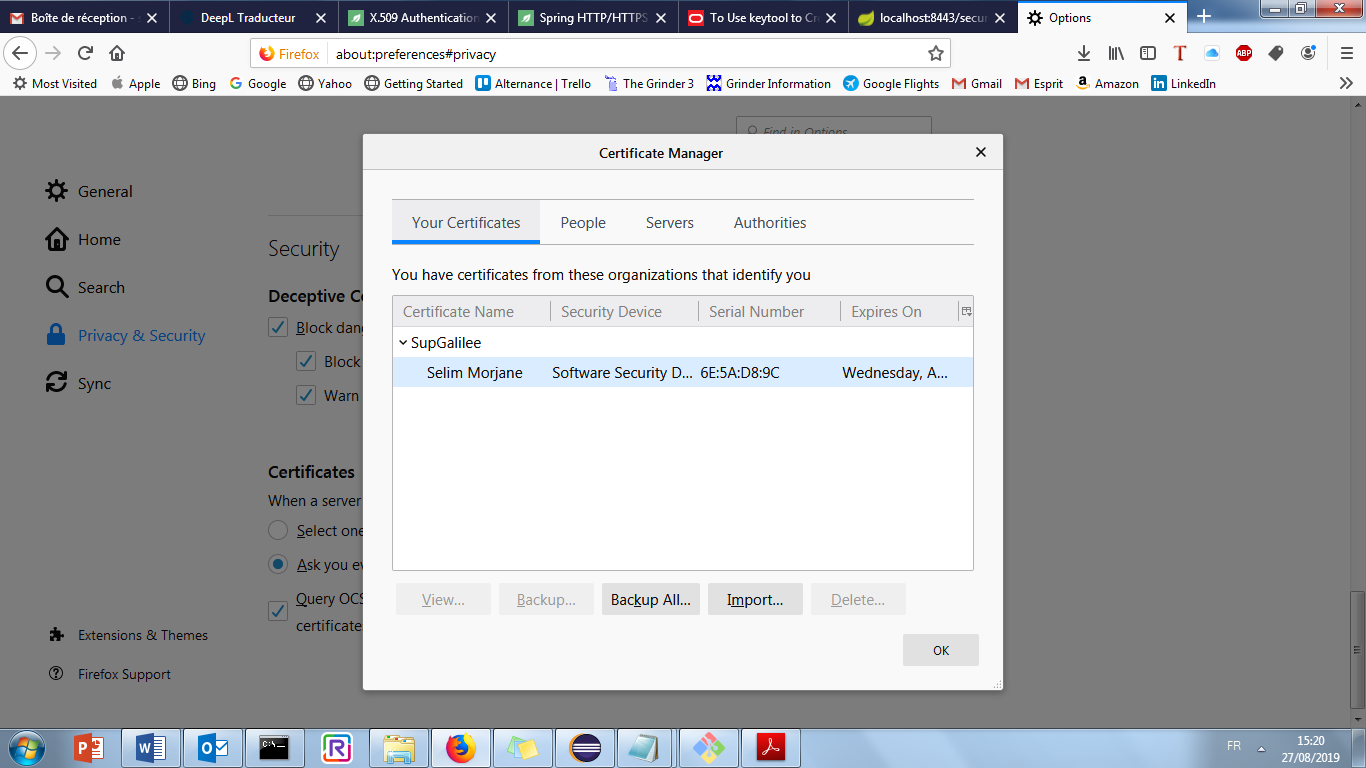
Parmi les spécifications du projet nous devons nous penser à vérifier l'identité d'un pair de communication lors de l'utilisation du protocole HTTPS

Lorsqu'une connexion sécurisée est établie, le client vérifie le serveur en fonction de son certificat (émis par une autorité de certification de confiance).

Mais au-delà de cela, X.509 dans Spring Security peut être utilisé pour vérifier l'identité d'un client via le serveur lors de la connexion. C'est ce qu'on appelle "l'authentification mutuelle"

Pour ce faire, nous allons commencer par exécuter la commande suivante tout en configurant les propriétés :

keytool -genkey -alias server-alias -keyalg RSA -keypass password123 -storepass pass123 -keystore keystore.jks



**https://www.baeldung.com/x-509-authentication-in-spring-security**

**Conclusion et webographie :**

Nous avons réalisé ce projet en deux parties : la première partie consiste à implémenter l’algorithme DES en java.

La deuxième partie consiste à créer une application web client/serveur qui soit à la fois ergonomique et pratique tout en utilisant les méthodes implémentées dans la première partie et en respectant les contraintes imposées dans l’énoncé.

Webographie :

https://www.commentcamarche.net/contents/204-introduction-au-chiffrement-avec-des

<https://www.baeldung.com/spring-boot-https-self-signed-certificate>

<https://openclassrooms.com/fr/courses/1631636-simplifiez-vos-developpements-javascript-avec-jquery>

https://www.baeldung.com/x-509-authentication-in-spring-security