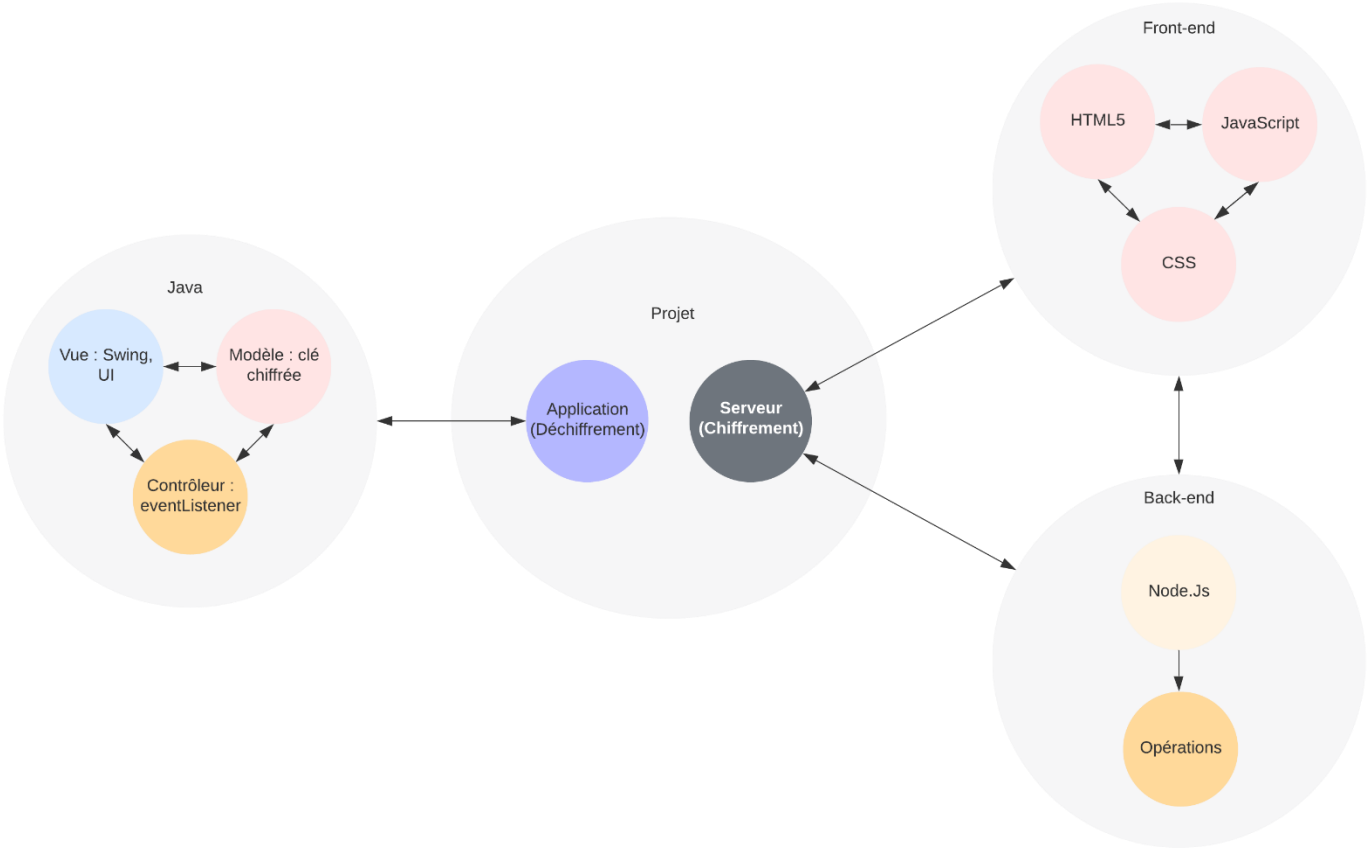
Table des matières

Introduction

Description

Le projet présenté ici comporte 2 modules distincts, une partie destinée aux utilisateurs (étudiants) et l'autre au client (université), le premier module sert principalement à ajouter un formulaire et ses fonctions sur une page html contenue par un serveur existant où les étudiants réaliseront leurs devoirs en distanciel afin de permettre le chiffrement de données sensibles, dont la note obtenue au devoir.
Le deuxième module sert quant à lui aux professeurs et a pour vocation de déchiffrer les clés cryptées fournit par les étudiants sur Moodle par l'intermédiaire d'une application Java.



Prérequis

Les applications Java ont été réalisées en local avec l'ide Visual Studio Code utilisant la version OpenJDK 17 (LTS) de Java Runtime, les librairies principales utilisées sont :

- javax.swing.*, java.awt, java.io.Fil, java.util.Date, java.text.SimpleDateFormat, java.awt, javax.crypto, java.util.Base64, java.security, Les fonctions liées aux conversions d'Apache.

Le formulaire a été simulé en local en utilisant Node.Js combiné aux diverses technologies du Web tel que HTML5, CSS, JavaScript.
Les principales librairies utilisées par Node.js sont les suivantes :

- Crypto
- Express

Serveur web contenant le formulaire de chiffrement

Intégration du pop-up à un serveur web existant

Dans ce projet la technologie utilisée pour simuler un serveur web local est le module express de node.js :

```
let express = require("express")
let app = express()
```

Afin d'ajouter le pop-up qui contient le formulaire à remplir par l'étudiant, il suffit de se rendre dans la page du serious game simulée, nommée index.ejs et d'importer tout ce qui se trouve dans une balise commentaire nommée TO IMPORT dans la page du vrai serious game.

```
<!-- ----------------------------------TO IMPORT---------------------------------- -->
<!-- ----------------------------------------------------------------------------- -->
...
<!-- ----------------------------------------------------------------------------- -->
<!-- ----------------------------------TO IMPORT---------------------------------- -->
```

Une fois cela fait, nous aurons besoin d'importer les dépendances fonctionnelles de ce même pop-up qui se trouvent dans des fichiers bien définis selon leur fonction :

- Nous avons besoin de définir dans notre serveur le dossier qui doit contenir les fichiers statiques tels que les feuilles de style (CSS), le JavaScript (JS) etc.
  Cela se fait par la commande suivante directement dans notre fichier serveur.js :

```
app.use(express.static("public"))
```

  Puis on organise nos répertoires comme suit :
  - public/css : pour les feuilles de style.
  - public/js : pour nos fichiers JavaScript.
  - Etc.

Exemples de chemin d'accès, une fois nos répertoires organisés et le fichier public défini comme répertoire contenant des éléments statiques :

```
<link rel="stylesheet" href="css/style.css">
<script type="text/javascript" src="js/main.js"></script>
```

Description des fonctions du serveur

À la racine, nous pouvons retrouver un fichier nommé **serveur.js** qui nous sert à démarrer le serveur web simulé et qui réalise les traitements suivants :

- Une méthode **get** qui nous permet d'envoyer un utilisateur sur notre page index.ejs lorsqu'un utilisateur entre l'url de connexion localhost sur le port d'écoute 8080 (localhost :8080), où index.ejs est la page qui simule un serious game.
- Une méthode **post** qui nous permet ici de récupérer les informations entrées par l'utilisateur (Num-étudiant, Nom, Prénom, Date de naissance) depuis le formulaire ajouté à la page du serious game.
  Par la suite, il traite ces données et effectue le chiffrement.

Description du protocole de chiffrement

Le module utilisé pour réaliser le chiffrement de nos données se nomme **Crypto** :

```
const crypto = require("crypto")
```

Dans ce projet, nous utilisons le protocole de chiffrement **AES CBC 256** qui nécessite 2 paramètres, la **clé secrète** (rend le chiffrement unique) et **le vecteur d'initialisation** (sert de sel). Modifier un de ces 2 paramètres, revient à devoir changer dans toutes les applications utilisant le chiffrement/déchiffrement, leurs valeurs pour qu'elles soient identiques (ces valeurs devant rester hors de portée des utilisateurs) :

```
const key = "12345678926457893563672635876353"
const iv = "1234567891123456";
```

Dans notre cas, la chaîne à crypter se trouve dans la variable **data** et s'organise comme suit :

```
data = num+","+nom+","+prenom+ ","+dateN+","+note + ","+ currentDate
+ ","+ currentHour;
```

Ou :

- Num = Numéro étudiant
- Nom = Nom de l'étudiant
- Prénom = Prénom de l'étudiant
- DateN = Date de naissance de l'étudiant
- Note = Note obtenue lors du serious game
- CurrentDate = la date du jour (Date du serveur web)
- CurrentHour = l'heure à laquelle la clé a été délivrée (Heure du serveur web)

On va créer un **cipher (crypteur)** qui va permettre de chiffrer les données :

```
let cipher = crypto.createCipheriv("aes-256-cbc", key, iv)
```

Puis on initialise une variable **encrypted** qui contiendra la clé chiffrée :

```
let encrypted = cipher.update(data, "utf-8", "hex")
```

Enfin, on ajoute le vecteur d'initialisation à la variable définie plus tôt :

```
encrypted += cipher.final("hex")
```

Génération d'un fichier texte contenant la clé chiffrée

Une fois la clé chiffrée générée, on souhaite que l'utilisateur puisse la télécharger, on va donc utiliser le module **fs** de node.js :

```
const fs = require('fs');
```

On utilise **unlinkSync,** car il vérifie l'existence du fichier à créer et le supprime s'il existe pour en recréer un nouveau, s'il n'existe pas, il crée simplement le fichier voulu :

```
fs.unlinkSync('public/files/clé.txt')
```

Puis on ajoute la clé au contenu du fichier texte :

```
fs.appendFile('public/files/clé.txt', encrypted, function (err)
```

Enfin, l'utilisateur peut télécharger la clé chiffrée par le biais du bouton **download** qui va récupérer le fichier généré dans le répertoire statique **public/files/clé.txt** :

```
<a href="files/clé.txt" download="CLé crypté">Download</a>
```

Application java permettant le déchiffrement

Les interfaces graphiques

Les classes représentant les diverses interfaces graphiques sont les suivantes :

- Windows.java
- WindowsP.java
- WindowsM.java

Ces classes sont toutes construites de la même manière, ce sont des classes qui étendent **JFrame** où l'on a rajouté des **JPanel** pour organiser nos divers éléments (JButton, JTextarea, …).

La classe qui contient le main permettant d'ouvrir les différentes fenêtres est **WindowsP.java**, elle permet de sélectionner le mode de déchiffrement voulu par le biais de 2 **JButton** (« mono », « multi ») auquel on a ajouté des **eventListener** permettant d'ouvrir les fenêtres correspondantes au click :

```
JButton multi = new JButton("Par liste");
JButton mono = new JButton("Par étudiant");
```

La classe **WindowsM.java** est l'interface graphique permettant de déchiffrer un ensemble de clés contenues dans un fichier zip (format Moodle) et exportant les résultats dans un fichier csv.

```
JButton selection = new JButton("Sélectionner");
```

Le bouton **« selection »** permet d'ouvrir une première fois le **JFileChooser** afin de sélectionner le fichier zip contenant la liste des clés chiffrées (format Moodle), cela nous permet de récupérer le Path (chemin d'accès) du fichier à exploité sur la machine exécutant le programme.

```
pathsrc = dialogue.getSelectedFile().getAbsolutePath();
```

Puis il le programme va directement décompresser le fichier zip grâce à la classe **Unzip** à la racine du Path obtenu dans un fichier nommé dst :

```
pathdst = dialogue.getSelectedFile().getParentFile().getAbsolutePath() + File.separator + "dst";
UnzipFile.unZip(pathsrc, pathdst);
```

Si ce fichier existe déjà, il est automatiquement supprimé grâce à un contrôle d'existence et de la classe **RemoveDir** :

```
if (dir.exists())
RemoveDir.delete(dir);
```

Enfin, on rend visible le bouton permettant de décrypter :

```
decrypte.setVisible(true);
```

```
JButton decrypte = new JButton("Décrypter");
```

Le bouton **« Décrypter »** permet de déclencher la méthode **parse** de la classe **Parse** qui ici va nous permettre de parcourir le fichier dst créé et de lire le contenu de chaque fichier texte :

```
for (File file : folder.listFiles()) {
    if (!file.isDirectory()) {
        System.out.println(file.getAbsolutePath());
        Scanner sc = new Scanner(file);
        while (sc.hasNextLine())
```

Et ainsi de déchiffrer leurs contenus et d'ajouter le résultat dans la zone de texte :

```
WindowsM.zoneOutput.append(Decrypt.decrypt(sc.nextLine()));
WindowsM.zoneOutput.append("\n");
```

```
JButton exporter = new JButton("Exporter");
```

Le bouton **« Exporter »** sert à générer un fichier csv contenant les informations de la clé déchiffrée en ouvrant une boite de dialogue **JFileChooser** où l'on doit sélectionner l'endroit dans lequel sauvegarder le fichier et entrer son nom :

```
JFileChooser dialogue = new JFileChooser();
dialogue.showSaveDialog(null);
String pathsrc = dialogue.getSelectedFile().getAbsolutePath();
```

Puis on lance la méthode **createCSV** de la classe **CreateCSV** :

```
CreateCSV.createCSV(pathsrc);
```

```
JFileChooser dialogue = new JFileChooser();
```

La boite de dialogue sert à récupérer les différents Path (chemin d'accès) des fichiers à exploiter ou des fichiers à créer.

La classe **Windiows.java** est l'interface graphique permettant de déchiffrer clé par clé en affichant le résultat dans un **JTextarea**.

```
JTextArea zoneInput = new JTextArea();
```

Cette zone de saisie sert à entrer une clé chiffrée afin de la déchiffrer simplement en stockant la chaîne entrée dans une variable :

```
String crypted = zoneInput.getText();
```

Puis, au click du bouton **« decrypter »** par un **eventListener,** on fait appel à la méthode **decrypt** de la classe **Decrypt** sur cette même variable :

```
String decode = Decrypt.decrypt(crypted);
```

Enfin, on ajoute le résultat à la zone de texte prévue à cet effet dont l'édition est désactivée :

```
zoneOutput.setText(decode);
```

Le déchiffrement

Pour le déchiffrement, on a utilisé la librairie java crypto :

```
import javax.crypto
```

La classe **Decrypt** nécessite une **clé secrète** et un **vecteur d'initialisation** identique à ceux utilisés pour le chiffrement, on a donc :

```
private static final String ENCRYPTION_KEY = "12345678926457893563672635876353";
private static final String ENCRYPTION_IV = "1234567891123456";
```

Pour commencer, on passe en argument la chaîne cryptée à la fonction **decrypt** :

```
public static String decrypt(String src)
```

Puis, on initialise le **cipher** au format voulu, soit **AES CBC 256** :

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.DECRYPT_MODE, makeKey(), makeIv());
```

Ensuite, on se sert des fonctionnalités de conversion fournies par Apache pour retourner le résultat en Base64 :

```
byte[] encodeBase = Hex.decodeHex(src.toCharArray());
String result = Base64.getEncoder().encodeToString((encodeBase));
```

Enfin, on ajoute la dernière partie des données (sel) puis on retourne le résultat complet :

```
return new String(cipher.doFinal(Base64.getDecoder().decode(result)));
```

Les méthodes **makeKey** et **makeIv** servent juste à former une chaîne de byte à partir de la String originale.

Le générateur de csv

La classe **CreateCSV** permet de générer un fichier csv comme suit :

- Pour commencer, on va créer un fichier grâce au Path pris en argument par la fonction qui finira par « .csv » :

```
PrintWriter printWriter = new PrintWriter(new File(path + ".csv"));
```

- Puis, on initialise une string grâce à **StringBuilder**, qui nous permettra de structurer notre csv :

```
StringBuilder stringBuilder = new StringBuilder();
```

- Puis, on crée la représentation de la 1re ligne du fichier qui servira de titre pour les différentes colonnes :

```
stringBuilder.append("Numero Etudiant");
stringBuilder.append(";");
stringBuilder.append("Nom");
stringBuilder.append(";");
stringBuilder.append("Prenom");
…
stringBuilder.append("\r\n");
```

- Enfin, on ajoute toutes les lignes contenues dans la zone de texte de l'interface graphique en y ajoutant un « ; » pour respecter le format csv (colonnes) * on remplit les colonnes de gauche à droite en splittant la chaîne grâce au caractère « , » séparant les informations dans la chaîne originale :

```
String split[] = total.split(",", 0);
for (String s : split) {
stringBuilder.append(s);
stringBuilder.append(";");
}
```

Version locale en java du formulaire de chiffrement

L'interface graphique

La classe **LocalCrypto** est définie comme les autres interfaces graphiques vues précédemment et utilise les méthodes suivantes : **encrypt** de la classe **Decrypt** et **createTxt** de la classe **CreateTxt**.
Les champs permettant de rentrer les informations nécessaires au chiffrement :

```
JTextField zoneInputNum = new JTextField();
JTextField zoneInputNom = new JTextField();
JTextField zoneInputPrenom = new JTextField();
JTextField zoneInputDate = new JTextField();
```

Le bouton **« encrypter »** est associé à un **eventListener** qui récupère la date et l'heure de l'ordinateur sur lequel le programme est exécuté et déclenche la méthode **encrypt** de la classe **Decrypt** :

```
JButton encrypter = new JButton("Crypter");
```

```
date = new SimpleDateFormat("dd/MM/yyyy").format(aujourdhui);
heure = new SimpleDateFormat("hh:mm:ss").format(aujourdhui);
String crypted = zoneInputNum.getText() + "," + zoneInputNom.getText() + "," + zoneInputPrenom.getText() + ","
+ zoneInputDate.getText()
        + "," + NOTE + "," + date + "," + heure;
String encode = Decrypt.encrypt(crypted);
```

Le bouton **« Exporter »** permet de générer un fichier texte contenant la clé chiffrée grâce à la méthode **createTXT** de la classe **CreateTXT** :

```
JButton exporter = new JButton("Exporter");
```

La zone de texte **« zoneOutput »** affiche simplement la clé chiffrée :

```
static JTextArea zoneOutput = new JTextArea();
```

Conversions des chaînes

On retire les informations non nécessaire {espaces vides, tabulations, retour chariot} dans les champs remplis par l'utilisateur, qui peuvent perturber le chiffrement :

```
String num = zoneInputNum.getText().replaceAll(" ", "").replaceAll("\t", "").replaceAll("\r", "")
        .replaceAll("\n", "");
String nom = zoneInputNom.getText().replaceAll(" ", "").replaceAll("\t", "").replaceAll("\r", "")
        .replaceAll("\n", "");
String prenom = zoneInputPrenom.getText().replaceAll(" ", "").replaceAll("\t", "").replaceAll("\r", "")
        .replaceAll("\n", "");
String dateN = zoneInputDate.getText().replaceAll(" ", "").replaceAll("\t", "").replaceAll("\r", "")
        .replaceAll("\n", "");
String crypted = num + "," + nom + "," + prenom + "," + dateN + "," + NOTE + "," + date + "," + heure;
```

Le chiffrement

Il fonctionne exactement comme pour le déchiffrement et se trouve dans la même classe (Decrypt) avec ces exceptions :
On doit mettre le paramètre **ENCRYPT_MODE** lors de l'initialisation du **cipher** :

```
cipher.init(Cipher.ENCRYPT_MODE, makeKey(), makeIv());
```

Puis, on doit ajouter le vecteur d'initialisation (sel) :

```
byte[] encrypted = cipher.doFinal(src.getBytes());
```

Enfin, on se sert des fonctionnalités fournies par Apache pour convertir le format Base64 en Hexadécimal :

```
String toHex = Hex.encodeHexString(encrypted);
return toHex;
```

Le générateur de fichier texte

La classe **CreateTXT** possède une méthode **createTXT** qui prend en argument la destination (Path) suivie d'un « .txt », cette méthode crée un fichier texte au chemin fournit.

```
public static void createTXT(String path)
```

On initialise le fichier à l'emplacement et au nom voulu :

```
PrintWriter printWriter = new PrintWriter(new File(path + ".txt"));
```

Puis, on initialise la String qui sera ajoutée au fichier texte.

```
StringBuilder stringBuilder = new StringBuilder();
```

On récupère le résultat contenu dans la zone de texte de l'interface graphique prévue à cet effet :

```
String result = LocalCrypto.zoneOutput.getText();
```

Enfin, on ajoute le résultat à la String définit plus tôt et on l'ajoute au fichier texte généré.

```
stringBuilder.append(result);
printWriter.write(stringBuilder.toString());
```