**SOMMAIRE**

[**I. Chiffrement AES: 1**](#_pufgdct4zfec)

[**II. Fonctionnement de l’algorithme: 1**](#_v9jv7g1n98ly)

[1. Organisation des données 2](#_hhuu4d726cqq)

[2. Étapes de l'AES 2](#_xye5uaa5j1pz)

[- Étape Initiale : AddRoundKey 2](#_6iteywvr72rj)

[- Tours intermédiaires (répétés pour chaque tour, sauf le dernier) : 2](#_xdiwip8bkcmk)

[- Dernier tour : 2](#_whd06r2zjif9)

[3. Routine d'expansion de clé 2](#_xkww7x8kdtvg)

[4. Déchiffrement 2](#_3hz2qbbs92qa)

[5. Methodes de l’AES: 3](#_q51xb81iwmvd)

[- AES-CBC (Cipher Block Chaining) : 3](#_5afgr4gh47e8)

[- AES-CTR (Counter Mode) : 3](#_674rf5qirlki)

[- AES-GCM (Galois/Counter Mode) : 3](#_qzfw2og7zu2f)

[6. Defit en Big Data: 3](#_iiyvtk292htq)

[**III. Choix de la méthode de chiffrement: 3**](#_gi2jny71auzf)

[**IV. Implementation de l’AES en JavaScript: 3**](#_rwvcp6v6cegk)

[1. Propriétaire : 3](#_jbd7mvgiq0br)

[2. Version et utilisation : 4](#_xas9ooh0ofe4)

[**V. Fonctionnement du mode CTR (Counter Mode) 4**](#_54ml2po1pxqe)

[1. Principe du mode CTR 4](#_ncwtguotr9x0)

[2. Étapes du chiffrement avec le mode CTR 4](#_zb9np1ktc0d6)

[- Génération de la clé : 4](#_scddgdxl44fp)

[- Construction du bloc de comptage : 4](#_elqxbunxuyhh)

[- Chiffrement du bloc de comptage : 4](#_jb2h6gzyglz)

[- Opération XOR : 4](#_on1ehgdknudp)

[- Encodage final : 5](#_slyur5gx2p0o)

[3. Déchiffrement avec le mode CTR 5](#_2dk74r3zr98v)

[**VI. Explication du code 5**](#_kig4gdlae03r)

[1. Chiffrement dans AES-CTR: 5](#_pjoiqnagyi7h)

[2. Dechiffrement dans AES-CTR: 7](#_ied2km6g7d18)

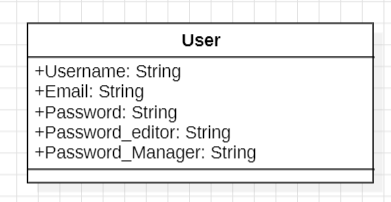
# **Réalisation de l’application de gestion des dossiers médicaux dans une base de donnée distribué et encryptant les données :**

1. *La Conception :*

Notre application viser a encrypté les données d’une base des données distribuée localement appliquant ce concept dans la gestion des dossiers médicaux, garantissant ainsi la confidentialité, l'intégrité et la disponibilité des informations sensibles tout en répondant aux exigences de sécurité des données dans le domaine de la santé.

On a ajouté aussi une gestion des utilisateurs authentifiant à l’interface de notre application avec les paramètres d’accès nécessaire.

* 1. *Structure des tables de base de donnée :*
* Pour le table de l’utilisateur on a proposé la structure suivant :

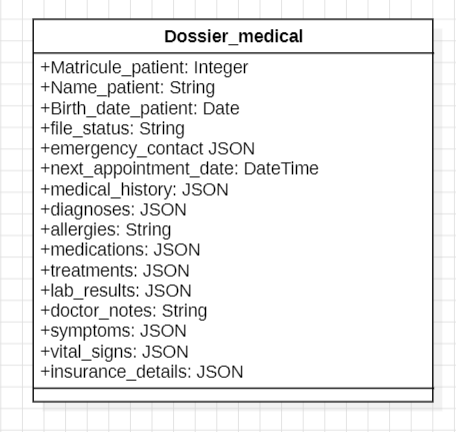


Tel que chaque utilisateur a 3 mots de passe :

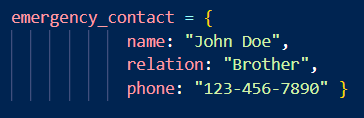
* **Password :** pour authentifier à l’interface de l’application.
* **Password\_editor :** pour ajouter ou supprimer des dossiers médicaux.
* **Password\_manager :** pour consulter et modifier des dossiers médicaux.

Attribuer trois mots de passe distincts à chaque utilisateur améliore considérablement la sécurité, en limitant l'accès aux fonctionnalités critiques selon les autorisations spécifiques et en réduisant le risque d'accès non autorisé.

* Pour le table de dossier médical on a proposé la structure suivant :



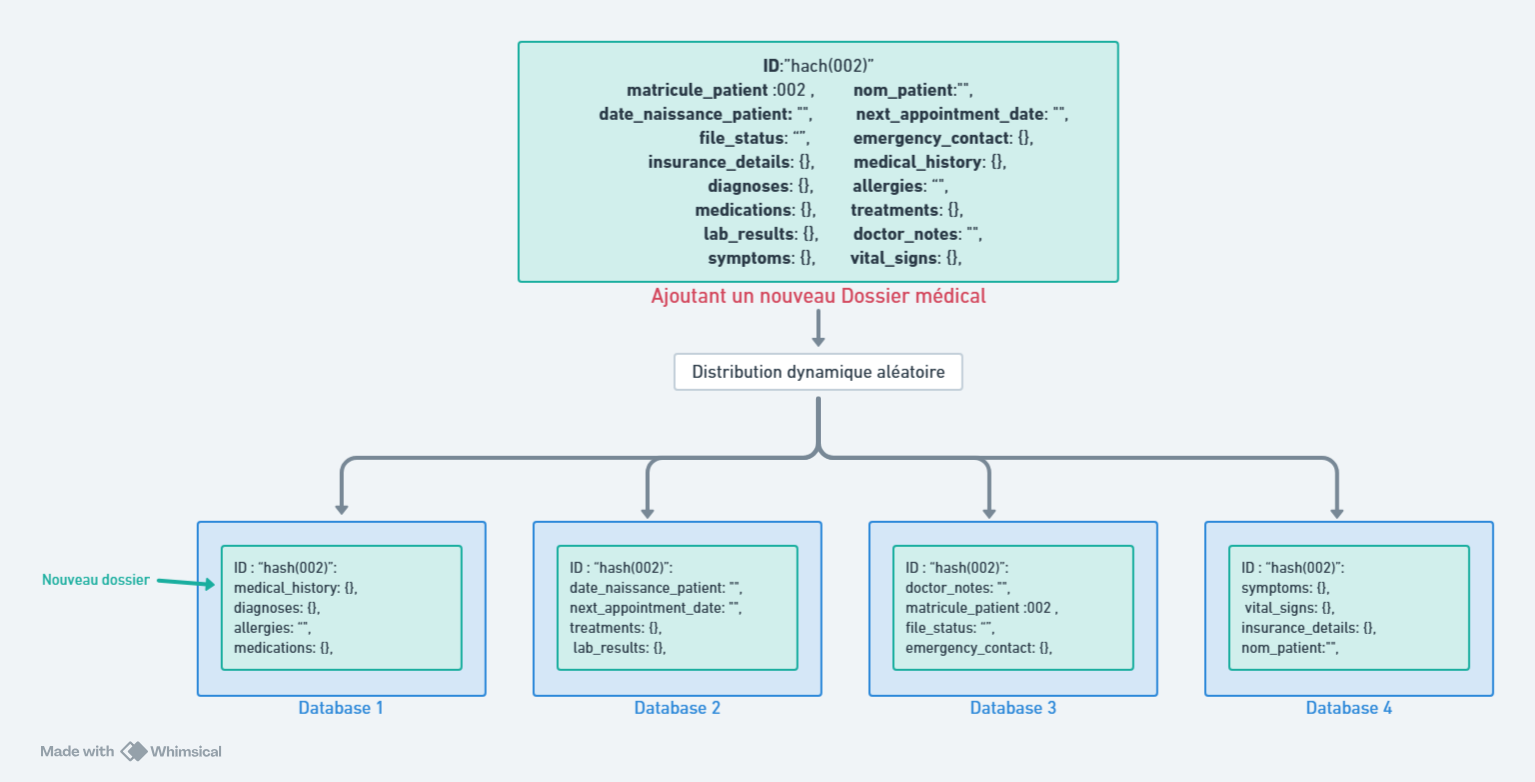
Le dossier médical contient des attributs structurés (String, int et date) et des attributs semi-structuré (JSON) tel que chaque attribut de type JSON contient les éléments associés pour lui, par exemple l’attribut ‘emergency\_contact’ :



* 1. *La distribution des données :*

Nous avons proposé une distribution **dynamique** des données dans une base de données distribuée contenant 4 bases de données distinctes. Étant donné que nous avons 16 attributs, à chaque ajout d’un objet, 4 attributs aléatoires seront assignés de manière aléatoire à l’une des bases de données. De plus, chaque partie d’un objet contenant 4 attributs dans sa base de données possède le même identifiant (ID) que les autres parties. Cette approche permet de répartir les données de manière équilibrée tout en garantissant une gestion efficace et flexible.

*Exemple :*

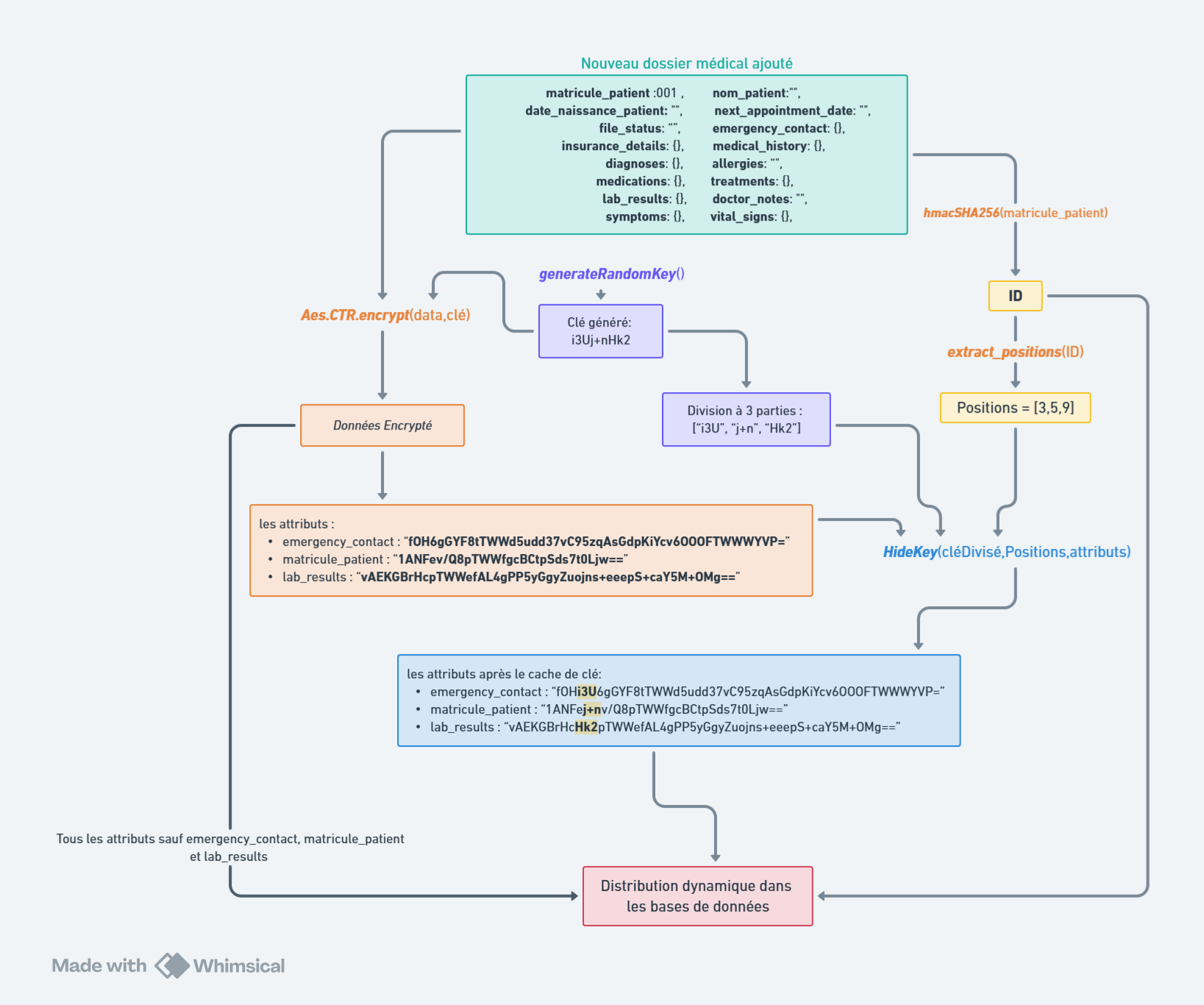


* 1. *Le chiffrement des données :*

Pour l’encryptage et décryptage des données on a utilisé une fonction AES (expliqué dans l’implémentation), tel que chaque objet (dossier médical) est encrypté et décrypté par une seule clé, et la clé est différente pour chaque dossier médical, suivant les étapes suivantes :

* Une clé de 9 caractères est générée **aléatoirement** pour chaque objet.
* Un ID de dossier est créé par le hachage de matricule de patient utilisant une fonction de hachage SH256 (expliqué dans l’implémentation).
* Tous les données de l’objet sont encryptées par cette clé et la fonction AES.
* La clé alors est divisée en 3 partie chaque partie contient 3 caractères.
* Les 3 parties de la clé sont caché dans 3 attributs fixe qui sont : ‘emergency\_contact’, ‘patient\_matricule’ et ‘lab\_results’ à des positions dynamiques calculés utilisant le ID de dossier et une fonction ***extract\_positions(hashedID)*** qui entrer l’identifiant et le divise en 3 parties et après utilisant ***parseInt*** pour avoir 3 positions d’après l’identifiant.
* Les données alors sont divisés en 4 parties et tous les parties ont le même ID qui est le ID de dossier médical (matricule hachée)

*Exemple :*



**Remarque :**

* Dans l’ajout des objets User tous les attributs sont haché avec la fonction ***hmacSHA256*** (username, email et les mots de passe), cette approche est utilisée pour garantir la sécurité des données et car les données de l’utilisateur sont juste utilisées dans l’authentification et alors on les besoin juste pour comparaison.

1. *L’implémentation :*

Pour l’implémentation de notre application on a utilisé les Framework et plateformes suivants :

* ***NodeJS :***

Node.js est une plateforme d'exécution JavaScript côté serveur. Elle permet de créer des applications performantes et évolutives grâce à son modèle non bloquant et basé sur les événements. Dans ce projet, Node.js est utilisé pour gérer la logique côté serveur et traiter les requêtes des utilisateurs.

* ***ReactJS :***

ReactJS est une bibliothèque JavaScript pour créer des interfaces utilisateur interactives. Elle permet de développer des composants réutilisables et dynamiques, facilitant la gestion de l'interface de l'application. ReactJS est utilisé ici pour créer l'interface utilisateur de manière rapide et efficace.

* ***ExpressJS :***

ExpressJS est un framework minimaliste pour Node.js, conçu pour simplifier la création d'API et de routes HTTP. Il fournit des fonctionnalités robustes pour développer des applications web et des services RESTful. Dans ce projet, il est utilisé pour structurer et gérer les API côté serveur.

* ***MongoDB :***

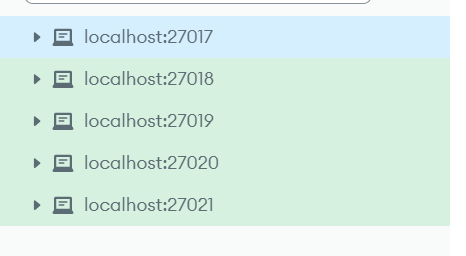
On a utilisé MongoDB qui est une base de données NoSQL qui stocke les données sous forme de documents JSON. Elle est idéale pour la distribution des données et alors on a l’utiliser ici pour stocker et organiser les données de l'application.

* ***Docker :***

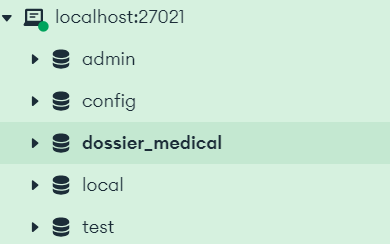
Docker est une plateforme conteneurisation. Il permet de regrouper une application et ses dépendances dans des conteneurs isolés, et efficace pour exécuter les services de base de données distribué.

* 1. *Distribution de base de donnée :*

Pour la distribution des données on a créé 5 base de données dans MongoDB tel que chaque base de donnée nécessite un service pour être exécuté :



Tel que les 4 base de données de 27018 à 27021 contient la collection dossier\_medical ou les données sont stockés :



La base de données de port 27017 est le **port interne** utilisé par les conteneurs MongoDB pour les communications au sein de l'environnement des autres bases de données et aussi les conteneurs de Docker (tel que chaque conteneur exécute une base de donné), comme Docker nécessite pour chaque service un port de Host (le port de base de donné) et un port de conteneur (le port 27017), donc le mapping des ports est comme suit :

* DataBase1 (localhost 27018) :  ‘27018 : 27017’
* DataBase2 (localhost 27019) :  ‘27019 : 27017’
* DataBase3 (localhost 27020) :  ‘27020 : 27017’
* DataBase4 (localhost 27021) :  ‘27021 : 27017’
  1. *L’algorithme de chiffrement :*

# *Chiffrement AES:*

Il s’agit d’un algorithme de chiffrement symétrique qui fonctionne en utilisant une clé unique pour le chiffrement et le déchiffrement, avec des tailles de clé de 128, 192 ou 256 bits, offrant des niveaux croissants de sécurité. AES chiffre les données par blocs de 128 bits.

# *Fonctionnement de l’algorithme:*

L'algorithme AES repose sur des transformations successives appliquées à des blocs de données de 128 bits en plusieurs tours d’opérations (10, 12 ou 14 selon la taille de la clé). L'algorithme comporte les étapes suivantes (référence [FIPS 197, Advanced Encryption Standard (AES)](https://csrc.nist.gov/files/pubs/fips/197/final/docs/fips-197.pdf)

):

## 1. Organisation des données

* Les données d'entrée (plaintext) sont divisées en blocs de 128 bits et organisées en une matrice de 4 lignes et 4 colonnes appelée État.
* La clé de chiffrement est utilisée pour générer des clés de tour (Round Keys) via une routine d'expansion de clé.

## 2. Étapes de l'AES

L'algorithme AES applique 10, 12 ou 14 tours (selon la taille de la clé), chacun comprenant plusieurs transformations :

### **Étape Initiale : AddRoundKey**

* + La clé initiale est appliquée à l'entrée par une opération XOR sur chaque octet du bloc avec les octets de la clé.

### **Tours intermédiaires (répétés pour chaque tour, sauf le dernier) :**

* + **SubBytes** : Chaque octet de la matrice est remplacé par un octet équivalent via une table de substitution non linéaire appelée **S-Box**.
  + **ShiftRows** : Les lignes de la matrice sont décalées de manière cyclique :
* Ligne 0 : Pas de décalage.
* Ligne 1 : Décalée d’un octet à gauche.
* Ligne 2 : Décalée de deux octets.
* Ligne 3 : Décalée de trois octets.
  + **MixColumns** : Les colonnes de la matrice sont transformées via des multiplications dans un champ fini (**GF(2⁸)**) avec un polynôme fixe. Cela mélange les données pour une meilleure diffusion.
  + **AddRoundKey** : Une clé de tour est appliquée par un XOR à chaque colonne de la matrice.

### **Dernier tour :**

* + Similaire aux tours intermédiaires, mais **sans MixColumns**.

## 3. Routine d'expansion de clé

Pour chaque tour, une clé dérivée (Round Key) est générée à partir de la clé initiale :

* **RotWord** : Permute les octets d’un mot de 32 bits.
* **SubWord** : Applique la substitution de la table S-Box.
* **Rcon** : Ajoute une constante de tour pour introduire de l’unicité.

## 4. Déchiffrement

Le processus de déchiffrement inverse les transformations dans l'ordre suivant :

1. **AddRoundKey**
2. **InvShiftRows** : Décalage inverse des lignes.
3. **InvSubBytes** : Substitution inverse via une table **Inverse S-Box**.
4. **InvMixColumns** : Multiplication inverse sur les colonnes.

## 5. Quelques méthodes de l’AES:

### **AES-CBC (Cipher Block Chaining)** :

Ce mode est couramment utilisé pour chiffrer les fichiers en blocs successifs, chaque bloc dépendant du bloc précédent. Cependant, CBC est plus vulnérable aux attaques de manipulation de bits et peut ne pas être idéal pour des systèmes Big Data modernes.

### **AES-CTR (Counter Mode)** :

AES en mode CTR est souvent préféré pour le Big Data, car il convertit le chiffrement en flux, rendant chaque bloc indépendant. Cela permet d’accélérer le processus de chiffrement/déchiffrement en parallèle, ce qui est avantageux dans un environnement distribué. CTR est souvent le meilleur choix pour HDFS en raison de sa capacité à traiter les données de manière rapide et évolutive.

### **AES-GCM (Galois/Counter Mode)** :

GCM offre non seulement la confidentialité mais aussi l’intégrité des données, ce qui le rend très sécurisé. Il est rapide comme CTR, mais ajoute des données d’authentification pour garantir que les données n’ont pas été altérées. Bien que légèrement plus gourmand en ressources, AES-GCM est souvent préféré dans les systèmes nécessitant une très haute sécurité.

## 6. Defit en Big Data:

* Puissance de calcul.
* Gestion des cles →KMS(Key Management Service)

# *Choix de la méthode de chiffrement:*

→ Dans notre projet on a utilisé AES-CTR pour les raisons suivantes:

**Propriétés cryptographiques fortes** : Le mode CTR garantit un haut niveau de sécurité en utilisant un **nonce unique** pour chaque session ou fichier.

**Conformité réglementaire** : AES-CTR respecte les normes de sécurité (par exemple, FIPS-197, NIST) exigées dans les secteurs sensibles comme la finance, la santé, et la gestion des données personnelles.

# *Implementation de l’AES-CTR en JavaScript:*

## 1. Propriétaire :

Développé par **Chris Veness** entre 2005 et 2014 et est sous licence **MIT**.

**Lien de référence pour AES** :<http://www.movable-type.co.uk/scripts/aes.html>​​.

## 2. Version et utilisation :

* Le fichier *aes.js* implémente les fonctions de base pour l'algorithme AES (cryptage/décryptage) suivant les spécifications du **FIPS-197**. Il prend en charge les tailles de clé AES : 128, 192 et 256 bits.
* Le fichier *aesCTR.js* étend cette implémentation pour prendre en charge le **mode de comptage (CTR)**, basé sur la publication **NIST SP800-38A​**.

# *Fonctionnement du mode CTR (Counter Mode) :*

Le mode CTR (Counter) transforme AES, un algorithme de chiffrement par blocs, en un algorithme de chiffrement par flux.

## 1. Principe du mode CTR

* **Bloc de comptage** :
  + Chaque bloc de données est combiné avec un **bloc de comptage** unique via une opération XOR.
  + Le **bloc de comptage** est initialisé avec un nonce (valeur arbitraire unique) et un compteur incrémenté à chaque bloc.
* **Avantages** :
  + **Parallélisation** : Chaque bloc peut être chiffré/déchiffré indépendamment.
  + **Conversion en flux** : Le mode CTR permet de chiffrer des données de longueur arbitraire sans rembourrage.

## 2. Étapes du chiffrement avec le mode CTR

L'implémentation de aesCTR.js suit ces étapes pour le chiffrement :

### **Génération de la clé** :

* + La clé est dérivée d'un mot de passe en utilisant AES lui-même.
  + La clé générée est étendue à la taille nécessaire (16, 24 ou 32 octets selon les bits).

### **Construction du bloc de comptage** :

* + Le **nonce** (8 octets) est généré à partir de l'horodatage et d'une valeur aléatoire.
  + Le reste du bloc est utilisé comme compteur.

### **Chiffrement du bloc de comptage** :

* + Chaque bloc de comptage est chiffré avec AES pour produire un flux chiffrant.

### **Opération XOR** :

* + Les blocs du texte en clair sont XORés avec les blocs chiffrés du compteur pour produire le texte chiffré.

### **Encodage final** :

* + Les blocs chiffrés sont concaténés et encodés en Base64 pour le stockage ou la transmission.

## 3. Déchiffrement avec le mode CTR

Le déchiffrement est symétrique au chiffrement :

* + Le nonce est extrait des 8 premiers octets du texte chiffré.
  + Les blocs de comptage sont reconstruits et chiffrés.
  + Le texte chiffré est XORé avec les blocs du compteur chiffré pour retrouver le texte en clair.

# *Explication du code:*

## 1. Chiffrement dans AES-CTR:

Le chiffrement dans AES-CTR fonctionne en générant un flux chiffrant à partir du **bloc de comptage (counter block)**, puis en XORant ce flux avec les données en clair.

* **Génération de la clé de chiffrement à partir du mot de passe:**

Le mot de passe est transformé en une clé chiffrante à l'aide de l'algorithme AES lui-même.

var blockSize = 16;

if (!(nBits==128 || nBits==192 || nBits==256)) return '';

plaintext = String(plaintext).utf8Encode();

password = String(password).utf8Encode();

var nBytes = nBits/8;

var pwBytes = new Array(nBytes);

for (var i=0; i<nBytes; i++) {

pwBytes[i] = isNaN(password.charCodeAt(i)) ? 0 : password.charCodeAt(i);

}

var key = Aes.cipher(pwBytes, Aes.keyExpansion(pwBytes));

key = key.concat(key.slice(0, nBytes-16));

* **Initialisation du bloc de comptage et le nonce:**

Le **nonce** (nombre arbitraire unique) est généré à partir d'un horodatage et d'une valeur aléatoire. Ce nonce est utilisé pour initialiser les 8 premiers octets du **bloc de comptage**, puis converti en texte pour être ajouté en préfixe au texte chiffré.

var nonce = (new Date()).getTime();

var nonceMs = nonce%1000;

var nonceSec = Math.floor(nonce/1000);

var nonceRnd = Math.floor(Math.random()\*0xffff);

for (var i=0; i<2; i++) counterBlock[i] = (nonceMs >>> i\*8) & 0xff;

for (var i=0; i<2; i++) counterBlock[i+2] = (nonceRnd >>> i\*8) & 0xff;

for (var i=0; i<4; i++) counterBlock[i+4] = (nonceSec >>> i\*8) & 0xff;

// convertir en text

var ctrTxt = '';

for (var i=0; i<8; i++) ctrTxt += String.fromCharCode(counterBlock[i]);

* **Génération des clés pour chaque tour (Key Schedule):**

La méthode keyExpansion génère les clés de tour nécessaires pour l'AES.

Cette méthode est utilisée pour préparer les clés pour chaque étape de chiffrement.

var keySchedule = Aes.keyExpansion(key);

* **Traitement bloc par bloc:**

Le texte en clair est divisé en blocs de 16 octets (taille fixe pour AES).

Chaque bloc est traité séparément.

var blockCount = Math.ceil(plaintext.length/blockSize);

var ciphertxt = new Array(blockCount);

* **Chiffrement du bloc de comptage:**

Chaque bloc de comptage est chiffré pour générer un flux chiffrant.

Les 8 derniers octets du **bloc de comptage** sont mis à jour pour inclure le compteur.

Le flux chiffrant est combiné avec le texte en clair à l'aide de l'opération XOR.

for (var b=0; b<blockCount; b++) {

for (var c=0; c<4; c++) counterBlock[15-c] = (b >>> c\*8) & 0xff;

for (var c=0; c<4; c++) counterBlock[15-c-4] = (b/0x100000000 >>> c\*8);

var cipherCntr = Aes.cipher(counterBlock, keySchedule);

// block size is reduced on final block

var blockLength = b<blockCount-1 ? blockSize : (plaintext.length-1)%blockSize+1;

var cipherChar = new Array(blockLength);

for (var i=0; i<blockLength; i++) { // -- xor plaintext with ciphered counter char-by-char --

cipherChar[i] = cipherCntr[i] ^ plaintext.charCodeAt(b\*blockSize+i);

cipherChar[i] = String.fromCharCode(cipherChar[i]);

}

ciphertxt[b] = cipherChar.join('');

}

* **Assemblage final et encodage Base64:**

Tous les blocs chiffrés sont concaténés avec le nonce en préfixe.

Le texte résultant est encodé en Base64 pour faciliter son stockage ou sa transmission.

var ciphertext = ctrTxt + ciphertxt.join('');

ciphertext = ciphertext.base64Encode();

return ciphertext;

## 2. Déchiffrement dans AES-CTR:

Le déchiffrement suit exactement le même processus que le chiffrement en mode CTR. Le flux chiffrant est XORé avec le texte chiffré pour récupérer le texte en clair.

* **Récupération du nonce:**

Le nonce est extrait des 8 premiers octets du texte chiffré.

Cela permet de reconstruire le **bloc de comptage** pour générer le flux chiffrant.

var counterBlock = new Array(8);

var ctrTxt = ciphertext.slice(0, 8);

for (var i=0; i<8; i++) counterBlock[i] = ctrTxt.charCodeAt(i);

* **Traitements des blocks chiffres:**

Le texte chiffré est divisé en blocs de 16 octets, en excluant les 8 octets du nonce.

Chaque bloc sera XORé avec le flux chiffrant généré à partir du **bloc de comptage**.

var nBlocks = Math.ceil((ciphertext.length-8) / blockSize);

var ct = new Array(nBlocks);

for (var b=0; b<nBlocks; b++) ct[b] = ciphertext.slice(8+b\*blockSize, 8+b\*blockSize+blockSize);

ciphertext = ct;

* **Reconstitution du text clair:**

Chaque bloc de texte chiffré est XORé avec le flux chiffrant pour reconstituer le texte en clair.

Le processus est identique au chiffrement, mais cette fois sur le texte chiffré.

var plaintxt = new Array(ciphertext.length);

for (var b=0; b<nBlocks; b++) {

for (var c=0; c<4; c++) counterBlock[15-c] = ((b) >>> c\*8) & 0xff;

for (var c=0; c<4; c++) counterBlock[15-c-4] = (((b+1)/0x100000000-1) >>> c\*8) & 0xff;

var cipherCntr = Aes.cipher(counterBlock, keySchedule);

var plaintxtByte = new Array(ciphertext[b].length);

for (var i=0; i<ciphertext[b].length; i++) {

// -- xor plaintxt with ciphered counter byte-by-byte --

plaintxtByte[i] = cipherCntr[i] ^ ciphertext[b].charCodeAt(i);

plaintxtByte[i] = String.fromCharCode(plaintxtByte[i]);

}

plaintxt[b] = plaintxtByte.join('');

}

* **Decodage finale:**

Les blocs de texte en clair sont concaténés, puis décodés en UTF-8 pour retrouver le texte d'origine.

var plaintext = plaintxt.join('');

plaintext = plaintext.utf8Decode();

return plaintext;

* 1. *L’algorithme d’hachage HMAC-SHA256 :*

L'HMAC est un mécanisme de vérification de l'intégrité des données et de l'authenticité d'un message, utilisé dans de nombreux protocoles cryptographiques comme TLS/SSL, IPsec et dans les API REST pour garantir que le message n'a pas été modifié.

**HMAC-SHA256**

L'HMAC est basé sur une fonction de hachage cryptographique (ici, SHA-256), et il applique deux étapes de hachage à un message en utilisant une clé secrète. Le processus suit ces étapes générales :

1. **Préparation de la clé :** Si la clé est trop longue, elle est hachée pour être réduite à une taille appropriée. Si la clé est trop courte, elle est remplie avec des zéros jusqu'à la longueur requise.
2. **Génération des pads :** Deux pads (ou clés internes et externes) sont générés par XOR avec des valeurs fixes.
3. **Calcul des hachages :** Le message est d'abord haché avec la clé interne (inner hash), puis ce résultat est haché avec la clé externe pour générer le HMAC final.

**Structure de l'implémentation**

**1. Fonction Key.sha256**

La fonction sha256 implémente un hachage SHA-256 simplifié. En réalité, l'algorithme complet de SHA-256 comprend plusieurs étapes complexes (comme le padding, les fonctions de compression et les constantes spécifiques). Cette fonction a pour but de simuler le comportement du hachage SHA-256 en effectuant une transformation simple des données d'entrée.

**Étapes principales :**

* **Encodage des données** : La donnée d'entrée (le message) est convertie en un tableau de bytes (avec TextEncoder).
* **Initialisation du tableau de hachage** : Un tableau de 8 mots de 32 bits est initialisé avec des valeurs arbitraires pour simuler l'état initial du hachage.
* **Transformation simplifiée** : Chaque byte du message modifie le tableau de hachage en fonction de sa position.
* **Retour du résultat** : Le tableau de hachage est converti en un tableau de bytes et retourné comme résultat du hachage.

**2. Fonction Key.hmacSHA256**

La fonction hmacSHA256 implémente le processus complet de l'HMAC en utilisant la fonction de hachage sha256 comme brique de base.

**Étapes principales :**

* **Préparation de la clé** : Si la clé est trop longue, elle est hachée avec sha256. Si elle est trop courte, elle est remplie avec des zéros pour atteindre la taille de 64 octets (taille de bloc pour SHA-256).
* **Génération des pads** : Deux pads sont générés en effectuant un XOR avec les valeurs fixes 0x36 pour le pad interne (iKeyPad) et 0x5c pour le pad externe (oKeyPad).
* **Calcul de l'inner hash** : Le pad interne est concaténé avec le message et haché avec SHA-256.
* **Calcul de l'outer hash** : Le pad externe est concaténé avec le résultat de l'inner hash et haché à nouveau pour produire le HMAC final.
* **Retour du HMAC** : Le HMAC est renvoyé sous forme d'un tableau de bytes.

**Exemple d'utilisation**

const key = "my-secure-password";

const message = "Hello";

const hmacResult = Key.hmacSHA256(key, message);

// Affichage du résultat HMAC en hexadécimal

console.log("HMAC-SHA256 (hex):", Array.from(hmacResult).map(byte => byte.toString(16).padStart(2, "0")).join(""));

L'exemple ci-dessus montre comment utiliser la fonction hmacSHA256 pour générer un HMAC pour un message donné avec une clé secrète. Le résultat est un HMAC en format hexadécimal.

**Analyse du Code**

**1. Séparation des responsabilités :**

* La fonction sha256 est dédiée à effectuer le hachage SHA-256 de base.
* La fonction hmacSHA256 s'occupe spécifiquement du calcul de l'HMAC, en utilisant sha256 comme base pour le hachage.

**2. Réutilisabilité :**

* La fonction sha256 peut être utilisée indépendamment pour hacher des données, ce qui rend l'implémentation modulaire.
* La fonction hmacSHA256 dépend de sha256 pour effectuer son travail, permettant une réutilisation du même hachage.

**3. Simplification du processus SHA-256 :**

* Bien que la fonction sha256 soit simplifiée, elle illustre les concepts clés derrière l'algorithme de hachage.
* Une implémentation complète de SHA-256 nécessiterait de suivre l'algorithme exact, avec des étapes comme le padding, la compression, et des constantes précises, ce qui dépasse le cadre de cette démonstration.

**Conclusion**

L'implémentation manuelle de HMAC-SHA256 permet de comprendre les principes sous-jacents de l'HMAC et du hachage SHA-256 sans dépendre des bibliothèques cryptographiques standard. Cette approche permet d'explorer les concepts de hachage et de validation de l'intégrité des données de manière détaillée. Cependant, dans un environnement de production, il est toujours recommandé d'utiliser des bibliothèques cryptographiques éprouvées et optimisées pour garantir la sécurité et la performance.

* 1. ***La manipulation des données :***

Nous avons développé cinq scripts pour manipuler des données dans la base de données MongoDB **Manga** via des API Express. Voici une brève description de chaque script :

1. **insertData.js** : Ce script reçoit des données via une API POST, les insère dans la base de données MongoDB et renvoie une confirmation à l'utilisateur, ce script utilise les fonctions ***AES.CTR.encrypt*** et ***hmacSHA256*** pour crypté les données et haché le matricule pour créer un identifiant, et insérer les données d’après la logique expliquée.
2. **deleteData.js** : Via une API DELETE, ce script supprime un document spécifique dans MongoDB en utilisant un identifiant unique et renvoie une réponse sur le succès ou l'échec de l'opération, ce script reçoit les matricules des objets pour supprimer et utilise la fonction ***hmacSHA256*** pour trouver ces objets et après les supprimer.
3. **modifyData.js** : Ce script permet de modifier les informations d'un objet spécifique dans MongoDB et trouvant l’identifiant par le hachage de matricule utilisant la fonction ***hmacSHA256***, puis renvoie une confirmation de la mise à jour.
4. **searchData.js** : Ce script reçoit des critères de recherche via une API GET, effectue une requête dans la base de données et renvoie les résultats correspondant aux critères spécifiés, utilisant ***AES.CTR.decrypt*** pour décrypter tous les données de base de donnée et comparer avec les éléments entrés.
5. **findData.js** : Ce script reçoit une matricule par un API GET pour la consultation de dossier complet, elle utilise ***hmacSHA256*** pour trouver l’identifiant de l’objet et après ***AES.CTR.decrypt*** pour décrypter l’objet et envoyer les données claires par la réponse.

Chaque script interagit avec la base de données MongoDB avec une connexion de client et en fonction de la logique définie (les bases de données distribués) et utilise des API Express pour recevoir des requêtes et envoyer des réponses appropriées.

* 1. ***Schéma pour représenter la structure de l’application :***

1. 