

## ÉVALUATION FINALE

Security by Design

Spécialité : EISI

---

## formulaire de contact sécurisé en local

---

*Réalisé par :*

M. MOKRANE Rabah

*Projet dirigé par :*

Mme. MARIE Amina  
(EPSI)

*Le 12 Mai 2025*

# Table des matières

<b>Introduction générale</b> . . . . .	<b>1</b>
<b>1 Mise en œuvre d'un formulaire de contact sécurisé : approche Security by Design</b> . . . . .	<b>2</b>
1.1 Introduction . . . . .	2
1.2 Architecture du projet . . . . .	2
1.2.1 Structure des dossiers . . . . .	3
1.2.2 Modularité et séparation des responsabilités . . . . .	4
1.3 Environnement de développement . . . . .	4
1.4 Objectifs de sécurité mis en œuvre . . . . .	5
1.5 Conclusion du chapitre . . . . .	7
<b>2 Implémentation des mécanismes de sécurité dans le backend</b> . . . . .	<b>8</b>
2.1 Introduction . . . . .	8
2.2 Authentification et gestion des sessions . . . . .	8
2.2.1 Création de compte et hachage des mots de passe . . . . .	8
2.2.2 Connexion et vérification des identifiants . . . . .	9
2.2.3 Middleware de protection des routes . . . . .	9
2.3 Vérification du reCAPTCHA côté serveur . . . . .	9
2.4 Chiffrement des messages de contact . . . . .	10
2.5 Sécurisation des en-têtes HTTP . . . . .	11
2.6 Journalisation et suivi des activités . . . . .	11
2.7 Conclusion du chapitre . . . . .	12
<b>3 Interface utilisateur, validation côté client et tests de sécurité</b> . . . . .	<b>13</b>
3.1 Introduction . . . . .	13
3.2 Développement du frontend . . . . .	13
3.2.1 Structure HTML et composants dynamiques . . . . .	13
3.2.2 Validation des champs côté client . . . . .	14
3.2.3 Intégration du reCAPTCHA dans l'interface . . . . .	14
3.3 Analyse de sécurité avec des outils spécialisés . . . . .	15
3.3.1 Tests de requêtes avec Postman . . . . .	15
3.3.2 OWASP ZAP – Analyse automatisée des vulnérabilités . . . . .	15
3.3.3 Inspecteur navigateur (DevTools) . . . . .	16
3.4 Mesures de renforcement mises en place . . . . .	16
3.5 Conclusion générale . . . . .	16
<b>Conclusion générale</b> . . . . .	<b>18</b>

# Table des figures

1.1	Arborescence du projet <code>formulaire_securise</code> . . . . .	3
1.2	Contenu sécurisé du fichier <code>.env</code> (informations sensibles masquées) . . . .	5
1.3	Génération des certificats SSL avec <code>OpenSSL</code> pour un serveur HTTPS local	6
2.1	Middleware <code>verifyCaptcha.js</code> – vérification du reCAPTCHA côté serveur	10
2.2	Démarrage du serveur Express en HTTPS sur <code>https://localhost:3000</code> .	12
3.1	Formulaire de Connexion avec <code>reCAPTCHA</code> actif côté frontend . . . . .	14
3.2	Formulaire de contact avec <code>reCAPTCHA</code> actif côté frontend . . . . .	15
3.3	Vérification des cookies avec les attributs <code>HttpOnly</code> et <code>Secure</code> dans DevTools	16

# Introduction générale

Dans un contexte où les applications web deviennent omniprésentes, la sécurité des données personnelles et la protection des utilisateurs constituent des enjeux majeurs. Chaque jour, des millions d'informations transitent via des formulaires en ligne : noms, adresses électroniques, messages confidentiels, voire données sensibles. Cette réalité rend impératif le renforcement des mesures de sécurité dès les premières étapes du développement.

Ce rapport présente la mise en œuvre d'un formulaire web sécurisé permettant la saisie de trois champs : nom, adresse e-mail et message. Afin de garantir la sécurité des données, plusieurs contraintes techniques ont été intégrées, telles que le chiffrement des données, l'utilisation de HTTPS, l'intégration d'un système Captcha, et la protection des cookies.

Le projet a été structuré en plusieurs étapes allant de l'initialisation du projet au déploiement sécurisé de l'application. Une phase finale de tests a permis de vérifier la robustesse du formulaire face aux attaques courantes : injections SQL, attaques XSS, et failles liées à la configuration HTTPS ou aux cookies.

Ce travail vise à illustrer, de manière pratique, l'importance de la sécurité by design dans le développement web. Il met en lumière les bonnes pratiques essentielles à adopter pour réduire la surface d'attaque des applications et renforcer la confiance des utilisateurs.

# Chapitre 1

## Mise en œuvre d'un formulaire de contact sécurisé : approche Security by Design

### 1.1 Introduction

Dans un contexte où les attaques informatiques sont de plus en plus fréquentes et sophistiquées, la sécurité des applications web devient une priorité incontournable. Le projet présenté ici a pour objectif de développer une application web complète permettant à des utilisateurs d'envoyer des messages via un formulaire de contact sécurisé. Ce formulaire n'est pas simplement fonctionnel, il est également conçu pour résister aux principales menaces du web (XSS, injections, bots, vol de session, etc.).

L'approche adoptée dans ce projet repose sur le principe de *Security by Design*, une méthodologie qui consiste à intégrer la sécurité dès la phase de conception de l'application, et non comme un ajout ultérieur. Cela permet de prévenir plutôt que de corriger, et d'assurer un niveau de confiance élevé dès la mise en production.

Le projet utilise une architecture full-stack moderne, avec un backend en Node.js/Express, une base de données MongoDB, un frontend HTML/CSS/JS natif, et plusieurs outils de sécurité comme Helmet, reCAPTCHA, le chiffrement AES, les sessions sécurisées et HTTPS. Il s'agit d'une solution pédagogique, mais réaliste et extensible, conforme aux bonnes pratiques recommandées par l'OWASP.

### 1.2 Architecture du projet

Le projet est organisé selon une structure modulaire, suivant le modèle MVC (Modèle-Vue-Contrôleur). Chaque composant a une fonction claire et indépendante, facilitant la maintenance, la lecture du code et les évolutions futures.

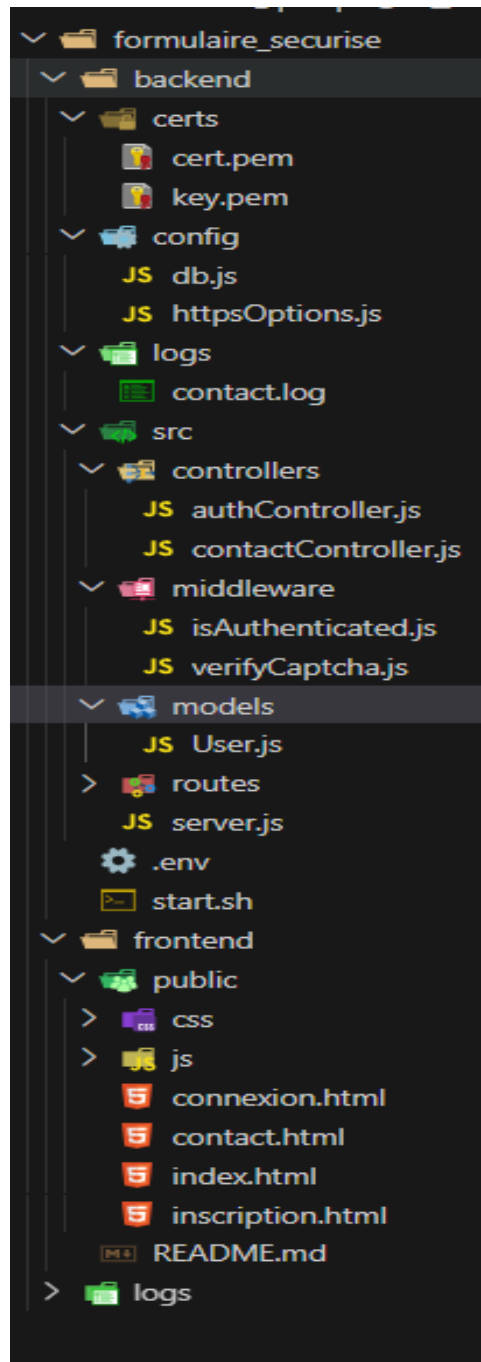


FIG. 1.1 : Arborescence du projet formulaire\_securise

### 1.2.1 Structure des dossiers

L'arborescence générale est la suivante :

```
formulaire_securise/  
  backend/           → Partie serveur  
    certs/          → Certificats SSL pour HTTPS  
    config/         → Paramètres de connexion et sécurité  
    logs/           → Enregistrement des événements (messages, accès)
```

```
src/  
  controllers/ → Gestion de la logique métier  
  middleware/  → Vérifications intermédiaires (session, captcha)  
  models/      → Schémas de données (Mongoose)  
  routes/      → Définition des endpoints API  
server.js      → Serveur principal Node.js (Express, HTTPS)  
.env          → Variables d'environnement sensibles  
  
frontend/  
  public/  
    css/       → Feuilles de style  
    js/        → Scripts JavaScript (validation, requêtes)  
    index.html → Page d'accueil ou redirection  
    contact.html → Formulaire de contact avec reCAPTCHA  
    inscription.html → Formulaire de création de compte  
    connexion.html → Formulaire de connexion utilisateur  
  
logs/          → Répertoire centralisé des journaux  
README.md      → Documentation détaillée du projet
```

### 1.2.2 Modularité et séparation des responsabilités

Chaque sous-dossier a une fonction précise :

- **controllers/** contient la logique applicative (comme le chiffrement des messages ou la création d'un utilisateur).
- **middleware/** intègre les vérifications de sécurité (session active, validation de Captcha, contrôle d'accès).
- **models/** définit les schémas MongoDB pour la persistance des données.
- **routes/** configure les routes HTTP (POST /contact, POST /login, etc.).

Cette approche permet de réduire les dépendances internes et de rendre chaque composant réutilisable.

## 1.3 Environnement de développement

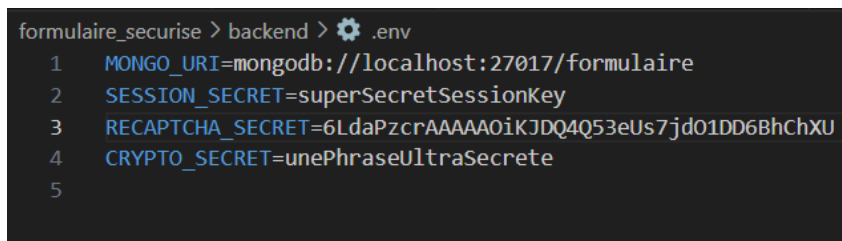
Le projet a été entièrement développé en local, avec un écosystème d'outils orienté développement sécurisé :

- **Node.js** (v18+) et **Express.js** : pour gérer les routes, middlewares, sessions et HTTPS.
- **MongoDB** : base NoSQL utilisée pour stocker les utilisateurs et les messages chiffrés.

- **Visual Studio Code** : éditeur principal avec extensions pour linting et debug.
- **OpenSSL** : pour la génération de certificats auto-signés.
- **OWASP ZAP** : scanner de vulnérabilités pour tester la résistance aux attaques.
- **Postman** : outil de test des routes API (POST /contact, /login...).
- **Browser DevTools** : vérification des cookies, des headers HTTP et du comportement JS.

Les dépendances principales du projet incluent :

- **express-session** pour la gestion sécurisée des sessions utilisateur.
- **bcrypt** pour le hachage des mots de passe.
- **crypto** (natif Node.js) pour le chiffrement AES-256 des messages.
- **dotenv** pour la gestion des variables sensibles (clé AES, URI MongoDB, etc.).
- **helmet** pour ajouter des headers de sécurité HTTP (dont une CSP personnalisée).
- **morgan** pour la journalisation dans les fichiers de logs.



```
formulaire_securise > backend > .env
1 MONGO_URI=mongodb://localhost:27017/formulaire
2 SESSION_SECRET=superSecretSessionKey
3 RECAPTCHA_SECRET=6LdaPzcrAAAAAoiKJDQ4Q53eUs7jd01DD6BhChXU
4 CRYPTO_SECRET=unePhraseUltraSecrete
5
```

FIG. 1.2 : Contenu sécurisé du fichier `.env` (informations sensibles masquées)

Le fichier `.env` contient les secrets utilisés pour la session, le chiffrement et la connexion à la base de données, comme illustré dans la figure 1.2.

## 1.4 Objectifs de sécurité mis en œuvre

La mise en place du projet s'est appuyée sur un ensemble de bonnes pratiques de sécurité, toutes orientées vers la réduction des vulnérabilités dès la conception :

- **Sécurisation des sessions utilisateur** :
  - Utilisation des attributs `Secure`, `HttpOnly` et `SameSite=strict` sur les cookies.
  - Expiration des sessions et destruction manuelle après déconnexion.



- **Validation côté client et serveur :**
  - JavaScript natif pour la vérification du format des champs.
  - Middleware de validation pour éviter les données malicieuses côté serveur.
- **Protection contre les bots :**
  - Intégration de Google reCAPTCHA v2 avec double vérification côté client et côté serveur.
- **Chiffrement des messages :**
  - Utilisation de l'algorithme AES-256-CBC.
  - Clé de chiffrement générée depuis l'environnement (.env) et IV aléatoire.
- **Communication sécurisée :**
  - Serveur Express configuré en HTTPS avec certificat SSL généré via OpenSSL.

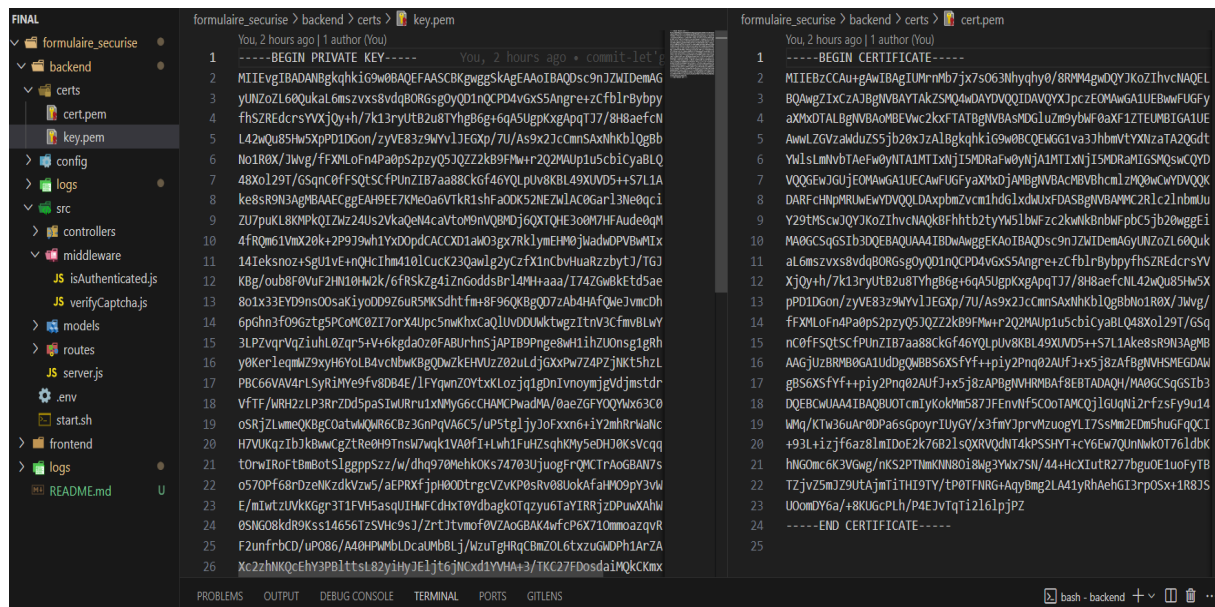


FIG. 1.3 : Génération des certificats SSL avec OpenSSL pour un serveur HTTPS local

Comme montré dans la figure ??, la commande OpenSSL a été utilisée pour générer les certificats SSL autosignés nécessaires au chiffrement HTTPS local.

- **Journalisation des événements :**
  - Chaque message envoyé via le formulaire est loggé dans un fichier `contact.log`.
  - Les requêtes HTTP sont journalisées automatiquement avec Morgan.
- **Protection contre les attaques XSS et injections :**
  - Header CSP (Content-Security-Policy) strict via Helmet.
  - Nettoyage et validation des champs côté backend.

## 1.5 Conclusion du chapitre

Ce premier chapitre a permis d'introduire le contexte du projet, son objectif pédagogique et technique, ainsi que son organisation globale. Le projet s'inscrit dans une logique de développement sécurisé dès les premières étapes. L'environnement de développement local a été configuré pour simuler des conditions réalistes de sécurité, avec la mise en place d'outils professionnels de validation.

La base est ainsi posée pour les prochaines étapes, qui consisteront à détailler les mécanismes de sécurité mis en œuvre dans le backend, l'intégration du reCAPTCHA, le chiffrement des messages et la sécurisation des communications HTTPS.

# Chapitre 2

## Implémentation des mécanismes de sécurité dans le backend

### 2.1 Introduction

Ce chapitre présente en détail l'implémentation des fonctionnalités critiques du backend, notamment l'authentification, la gestion des sessions, le chiffrement des messages et l'intégration du système de vérification reCAPTCHA. L'objectif est de démontrer comment les bonnes pratiques de sécurité ont été traduites en code, en s'appuyant sur l'écosystème Node.js, MongoDB et divers modules de sécurité.

Toutes les données sensibles (mots de passe, messages) sont soit hachées, soit chiffrées, et les routes critiques sont protégées par des middlewares. De plus, l'application ne se contente pas de vérifier les entrées côté client, mais renforce les contrôles côté serveur pour éviter toute injection ou tentative d'usurpation.

### 2.2 Authentification et gestion des sessions

#### 2.2.1 Création de compte et hachage des mots de passe

Lors de l'inscription, les utilisateurs saisissent un nom, une adresse e-mail et un mot de passe. Le mot de passe est immédiatement haché à l'aide de la bibliothèque `bcrypt`, avec un sel généré dynamiquement. Ce hachage rend tout vol de base de données inexploitable sans force brute coûteuse.

```
const bcrypt = require('bcrypt');
const saltRounds = 12;

const hashedPassword = await bcrypt.hash(req.body.password, saltRounds);
```

Le mot de passe haché est ensuite stocké dans MongoDB via un modèle Mongoose (`User.js`).

### 2.2.2 Connexion et vérification des identifiants

Lors de la connexion, le backend compare le mot de passe saisi avec le hachage stocké :

```
const isMatch = await bcrypt.compare(password, user.hashPassword);
```

Si la vérification est réussie, une session est créée à l'aide du module `express-session`. Cette session utilise des cookies protégés par les flags suivants :

- `HttpOnly` : empêche l'accès aux cookies depuis JavaScript.
- `Secure` : assure que les cookies ne transitent que via HTTPS.
- `SameSite: 'strict'` : bloque les envois de cookies inter-domaines.

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    httpOnly: true,
    secure: true,
    sameSite: 'strict'
  }
}));
```

### 2.2.3 Middleware de protection des routes

Certaines routes, comme celle du formulaire de contact, nécessitent que l'utilisateur soit connecté. Un middleware personnalisé `isAuthenticated.js` a été mis en place pour vérifier l'état de la session :

```
function isAuthenticated(req, res, next) {
  if (req.session.user) {
    next();
  } else {
    res.status(401).send("Accès refusé.");
  }
}
```

## 2.3 Vérification du reCAPTCHA côté serveur

Pour éviter les soumissions automatisées de formulaire (bots), l'application utilise Google reCAPTCHA v2. Ce composant est affiché côté client dans le formulaire, puis validé côté serveur via l'API Google :

```
try {
  const response = await fetch(
    `https://www.google.com/recaptcha/api/siteverify`,
    {
      method: "POST",
      headers: { "Content-Type": "application/x-www-form-urlencoded" },
      body: `secret=${secret}&response=${token}`,
    }
  );

  const data = await response.json();
  console.log("Réponse reCAPTCHA:", data);

  if (!data.success) {
    return res.status(400).send("Captcha invalide");
  }

  next();
} catch (err) {
  console.error("Erreur lors de la vérification du captcha:", err);
  res.status(500).send("Erreur lors de la vérification du captcha");
}
};
```

FIG. 2.1 : Middleware `verifyCaptcha.js` – vérification du reCAPTCHA côté serveur

La figure 2.1 montre le middleware `verifyCaptcha.js` qui intercepte les requêtes entrantes pour vérifier le token reCAPTCHA, limitant ainsi les soumissions automatisées.

```
const response = await axios.post(
  `https://www.google.com/recaptcha/api/siteverify`,
  null,
  {
    params: {
      secret: process.env.RECAPTCHA_SECRET,
      response: req.body.token
    }
  }
);
```

Si la vérification échoue, le serveur retourne une erreur, interrompant la soumission du formulaire. Ce processus renforce la protection contre les attaques de type DDoS léger, brute force ou spam.

## 2.4 Chiffrement des messages de contact

Les messages soumis via le formulaire (nom, e-mail, contenu du message) sont considérés comme sensibles. Un chiffrement AES-256-CBC est appliqué à ces données avant stockage :

- Une clé de chiffrement de 256 bits est stockée dans le fichier `.env`.
- Un vecteur d'initialisation (IV) aléatoire est généré pour chaque message.
- Les données chiffrées et le IV sont stockés dans la base de données.

```
const iv = crypto.randomBytes(16);
const cipher = crypto.createCipheriv('aes-256-cbc', Buffer.from(key), iv);
let encrypted = cipher.update(text, 'utf8', 'hex');
encrypted += cipher.final('hex');
```

La déchiffrement est uniquement possible via le backend, avec la clé et l'IV associés. Cela permet de garantir la confidentialité même en cas d'accès non autorisé à la base.

## 2.5 Sécurisation des en-têtes HTTP

L'outil `Helmet` est intégré comme middleware Express pour renforcer les en-têtes HTTP, bloquer certaines attaques et forcer des politiques de sécurité strictes :

```
const helmet = require('helmet');
app.use(helmet());
```

Une politique CSP (Content Security Policy) personnalisée a été définie afin de restreindre les sources de scripts, de styles et de connexions :

```
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", "https://www.google.com", "https://www.gstatic.com"],
    styleSrc: ["'self'", "'unsafe-inline'"],
    frameSrc: ["https://www.google.com"]
  }
}));
```

Cette configuration empêche l'injection de scripts externes malveillants (attaque XSS).

## 2.6 Journalisation et suivi des activités

L'ensemble des requêtes HTTP entrantes est enregistré à l'aide du module `morgan`, en complément d'un fichier de logs spécifique (`contact.log`) pour enregistrer tous les messages reçus :

```
const fs = require('fs');
const morgan = require('morgan');
const accessLogStream = fs.createWriteStream('./logs/contact.log', { flags: 'a' });

app.use(morgan('combined', { stream: accessLogStream }));
```

Cette journalisation est essentielle pour :

- Tracer les actions des utilisateurs.
- Identifier les tentatives d'abus ou d'injection.
- Apporter des preuves en cas d'incident de sécurité.

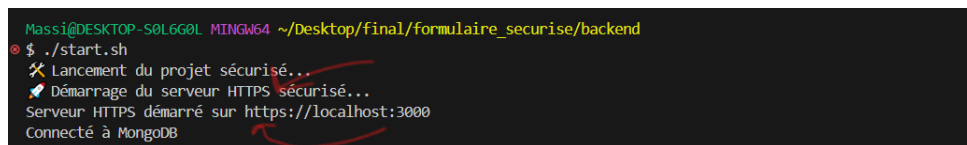


FIG. 2.2 : Démarrage du serveur Express en HTTPS sur `https://localhost:3000`

Comme montré dans la figure 2.2, le serveur Express a été démarré avec succès en mode HTTPS, confirmant que le projet fonctionne en local avec une connexion sécurisée.

## 2.7 Conclusion du chapitre

L'ensemble des mécanismes décrits dans ce chapitre montrent que le backend de l'application ne se limite pas à une logique fonctionnelle : chaque action utilisateur, chaque interaction, chaque donnée transmise est soumise à un contrôle rigoureux. Le système combine plusieurs couches de protection – hachage, chiffrement, sessions, Captcha, entêtes HTTP – pour créer une défense robuste.

Ce niveau de sécurisation est conforme aux recommandations de l'OWASP et constitue un socle fiable pour des extensions futures du projet (ex : système de rôles, gestion de fichiers sécurisés, authentification à deux facteurs). Dans le chapitre suivant, nous détaillerons le fonctionnement côté client et l'intégration des outils de test pour l'évaluation de la sécurité.

# Chapitre 3

## Interface utilisateur, validation côté client et tests de sécurité

### 3.1 Introduction

Ce dernier chapitre se concentre sur la couche visible de l'application : le frontend. Bien que la majorité des mécanismes de sécurité soient gérés côté serveur, la validation et la robustesse côté client jouent un rôle crucial dans l'expérience utilisateur et la prévention des erreurs. De plus, nous abordons les outils utilisés pour tester la résilience de l'application face aux attaques courantes (XSS, injections, interception de données, etc.).

### 3.2 Développement du frontend

#### 3.2.1 Structure HTML et composants dynamiques

Le frontend a été conçu de manière simple, épurée, mais efficace. Les pages principales incluent :

- **index.html** : page d'accueil ou de redirection.
- **connexion.html** : formulaire de connexion sécurisé.
- **inscription.html** : formulaire d'inscription avec vérification.
- **contact.html** : formulaire de contact avec Google reCAPTCHA.

Ces pages utilisent du HTML5, stylisé via CSS natif (et éventuellement SCSS), et animées avec JavaScript natif. Les interactions (soumission de formulaire, redirection) se font en utilisant `fetch()` pour une communication fluide avec le backend.



### 3.2.2 Validation des champs côté client

Chaque champ du formulaire (nom, e-mail, mot de passe, message) est soumis à une validation stricte avant envoi :

- **Nom** : lettres uniquement, longueur minimale.
- **Email** : regex standard de vérification.
- **Message** : taille maximale et vérification anti-injection.
- **Mot de passe** : longueur minimale, caractères spéciaux.

Voici un exemple de validation JavaScript :

```
const emailRegex = /^[\\w-\\.]+@([\\w-]+\\.){2,4}$\n/;\nif (!emailRegex.test(email)) {\n    alert("Adresse email invalide");\n    return;\n}
```

### 3.2.3 Intégration du reCAPTCHA dans l'interface

La page `contact.html` contient un composant Google reCAPTCHA v2 :

```
<div class="g-recaptcha" data-sitekey="clé_publicque_recaptcha"></div>
```

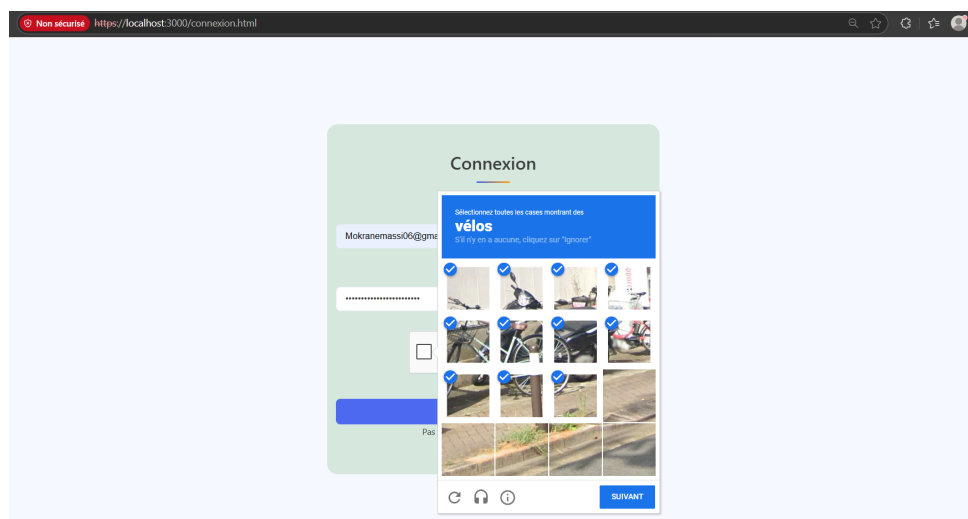


FIG. 3.1 : Formulaire de Connexion avec reCAPTCHA actif côté frontend

A screenshot of a web browser showing a contact form titled "Contactez-nous". The form is centered on a light blue background. It has a title "Contactez-nous" with a small orange and blue line underneath. Below the title are three input fields: "Nom :" with the value "massi", "Email :" with the value "Mokranemassi06@gmail.com", and "Message :" with the value "Bonjour espi". Below the message field is a reCAPTCHA widget showing a green checkmark and the text "Je ne suis pas un robot". At the bottom of the form is a blue button labeled "Envoyer". A link "Retour à l'accueil" is also visible above the button. The browser's address bar shows "https://localhost:3000/contact.html" and a security warning "Non sécurisé".

FIG. 3.2 : Formulaire de contact avec reCAPTCHA actif côté frontend

Avant d'envoyer le formulaire, le script JS récupère le token généré, et le transmet au serveur pour validation.

### 3.3 Analyse de sécurité avec des outils spécialisés

#### 3.3.1 Tests de requêtes avec Postman

Postman a été utilisé pour envoyer des requêtes HTTP directement à l'API. Cela a permis de :

- Vérifier les réponses attendues (succès, erreurs, statut HTTP).
- Tester les routes en contournant l'interface frontend.
- Tenter d'envoyer des messages sans reCAPTCHA (rejetés avec erreur 403).

#### 3.3.2 OWASP ZAP – Analyse automatisée des vulnérabilités

Le scanner OWASP ZAP a été utilisé pour analyser l'application en mode proxy. Les résultats ont révélé :

- **Aucune injection détectée** : grâce aux validations et à l'absence de SQL.
- **Aucune faille XSS active** : grâce à Helmet et à la CSP configurée.
- **HTTPS détecté avec certificats auto-signés** : malgré l'alerte, cela est normal en local.
- **Cookies sécurisés** : tous les cookies affichent les flags `HttpOnly` et `Secure`.

### 3.3.3 Inspecteur navigateur (DevTools)

Les DevTools ont permis de confirmer que :

- Les cookies ne sont pas accessibles via `document.cookie`.
- Les connexions se font uniquement via `https://localhost`.
- Le contenu des formulaires n'est pas stocké en clair dans les réponses.
- Les requêtes JavaScript ne génèrent pas d'erreurs de politique CSP.

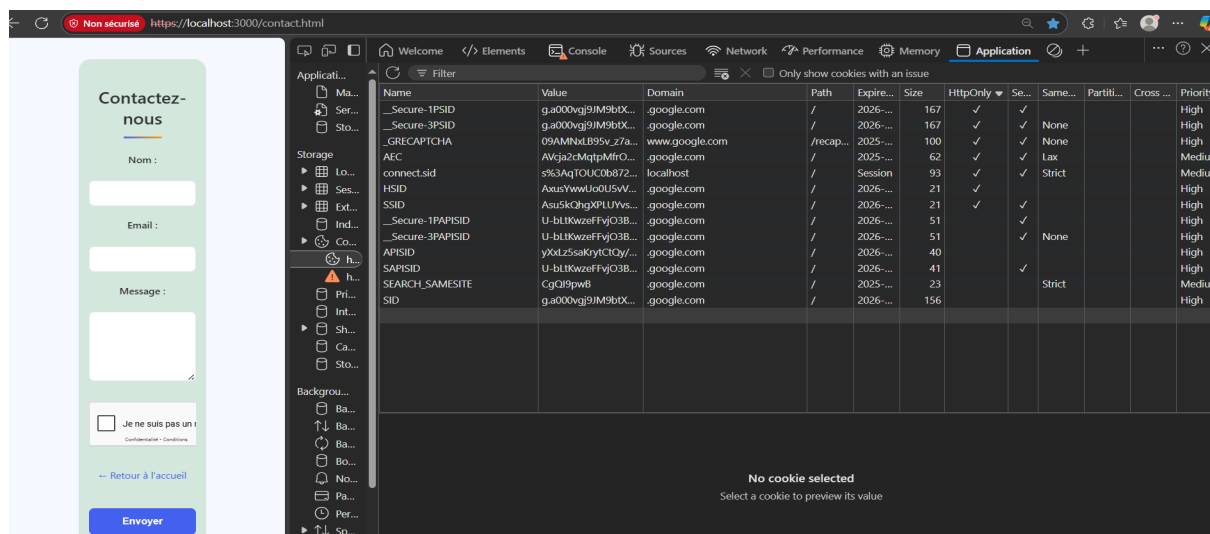


FIG. 3.3 : Vérification des cookies avec les attributs `HttpOnly` et `Secure` dans DevTools

## 3.4 Mesures de renforcement mises en place

Outre les tests, des mesures supplémentaires ont été appliquées :

- Interdiction d'accès aux routes sensibles sans authentification.
- Redirection automatique vers HTTPS.
- Contrôle du type MIME des fichiers statiques.
- Déconnexion manuelle et expiration automatique des sessions.

## 3.5 Conclusion générale

Grâce à une stratégie *Security by Design*, le projet couvre tous les aspects de sécurité applicative moderne : chiffrement, authentification sécurisée, validation double (client + serveur), cookies protégés, et tests de robustesse. Le résultat est un formulaire de contact

fiable, résistant aux attaques de type XSS, CSRF, brute-force, et à la compromission de données.

Ce projet constitue une base solide pour tout développement web sécurisé et pourrait être étendu vers un système plus complet (portail d'administration, API REST sécurisée, etc.). Il démontre également l'importance de combiner frontend ergonomique et backend rigoureux pour garantir la sécurité globale d'un système web.

# Conclusion générale

Ce projet de création d'un formulaire de contact sécurisé, avec une approche *Security by Design*, illustre l'importance d'intégrer des pratiques de sécurité dès les premières étapes du développement web. En mettant en œuvre des techniques telles que le chiffrement des données sensibles, l'utilisation de HTTPS, la validation des entrées côté client et serveur, ainsi que la protection contre les bots avec reCAPTCHA, ce système assure la confidentialité, l'intégrité et la disponibilité des informations des utilisateurs.

La modularité de l'architecture, basée sur des technologies comme `Node.js`, `MongoDB`, et `Express`, permet une gestion efficace des sessions sécurisées, des logs, ainsi qu'une évolutivité facile du projet. De plus, l'utilisation de `bcrypt` pour le hachage des mots de passe et le chiffrement AES-256 pour la protection des données utilisateurs garantit une sécurité renforcée, même en cas d'attaque.

Les tests de sécurité, réalisés avec des outils comme `OWASP ZAP` et `Postman`, ont permis de s'assurer que le système est résistant aux attaques courantes telles que l'injection SQL, les failles XSS, et les vulnérabilités liées à la gestion des cookies. L'approche *Security by Design* a donc permis d'implémenter une architecture fiable et robuste, tout en respectant les meilleures pratiques en matière de protection des données.

Ainsi, ce projet offre une bonne base pour tout formulaire web nécessitant une gestion sécurisée des données utilisateurs, tout en démontrant l'importance d'intégrer la sécurité tout au long du cycle de développement d'une application web. Ce travail est un pas vers des applications plus sûres et plus résilientes face aux menaces actuelles.