

Projet n°3 : DevSecOps

Pour faire suite à notre formation chez Guardia Cybersecurity School, nous avons eu comme tâche de désigner, coder et sécuriser un site en groupe de 4. Notre choix s'est orienté vers un site de gestion d'événements que nous hébergerons sur github. Par la suite, nous avons assigné à chacun des membres du groupe un rôle de "lead" parmi 4 (Front-end, Back-end, DevOps, Manager).

Premièrement, nous avons structurer et lister les fonctionnalités souhaités du site le site selon la manière suivante :

- Une page de garde qui permet de présenter le site aux utilisateurs
- Une page de connexion
- Une page d'inscription
- Une page principale pour visualiser nos événements
- Une page (fenêtre) d'ajout d'événements
- Une page de présentation des fondateurs

Ainsi, nous avons commencé à coder les différentes pages du site en utilisant du HTML et du CSS (Front-end), ces deux langages nous permettant de mettre en page le contenu du site selon nos envies, c'est-à-dire la structure (textes, tableaux, éléments...) et la forme du site (charte graphique, organisation des éléments...)

```
<header class="hero-section">
  <div class="hero-content">
    <div class="hero-text-box">
      <a href="index.html" class="titre-barres">MonAgendaPro</a>
      <div class="bar-deco top"></div>
      <div class="bar-deco bottom"></div>
    </div>
  </div>
```

```
.navbar {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 80px;
```

Nous avions donc au départ prévu de créer 6 fichiers HTML, or par la suite, nous nous sommes rendus compte que certains de ces fichiers nous étaient inutiles car des fichiers PHP les remplaçaient.

De ce fait, nous avons décidé de garder uniquement les fichiers index.html, login.html et register.html, founder.html (pages de garde, de connexion, d'inscription et des fondateurs). Nous voilà alors avec 4 fichiers HTML, chacun avec un rôle précis :

index.html :

C'est le corps du site, la page de garde. Lorsque l'on arrive sur le site, c'est la 1ere page sur laquelle l'utilisateur arrive. Dans ce fichier une fonts API de google est utilisé pour charger du css, de même pour quelque autres fichier css :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css2?family=Orbitron:wght@400;700&display=swap">
    <title>MonAgendaPro</title> <!--Nom du site dans l'onglet de navigation-->
    <link rel="stylesheet" href="style.css"> <!--Liens pour relier l'HTML au CSS-->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/all.min.css">
</head>
```

Le fichier charge en plus une navbar (barre de liens en haut du site avec les liens MonAgendaPro, Présentation, Les Fondateurs, Connexion), des boutons avec des liens vers les pages de connexions et d'inscription et un bouton pour descendre vers la présentation du site et enfin, un footer de bas de page avec des copyrights

login.html :

Cette page sert uniquement lors de la connexion de l'utilisateur à son compte client. Un formulaire permettant à l'utilisateur de saisir son nom d'utilisateur et son mot de passe s'affiche ; lorsqu'il valide, les informations sont envoyées au script "login.php" pour l'authentification sinon une erreur s'affiche (en cas d'identifiants incorrects), et un lien est également proposé pour accéder à la page d'inscription. Un bouton permettant de choisir entre le mode jour et nuit

register.html :

De même, la page register.html ne sert uniquement lors de la création du compte de l'utilisateur. un bouton permettant d'activer ou désactiver le mode sombre, et un formulaire complet demandant un nom d'utilisateur, une adresse email, un mot de passe et sa confirmation ; il gère aussi l'affichage automatique d'erreurs via les paramètres de l'URL, et propose un lien vers la page de connexion pour les utilisateurs déjà inscrits.

founder.html :

Enfin, la page la dernière page, founder.html, est une page de présentation des fondateurs comprenant :

- une animation de fond,
- un lien vers la page d'accueil,
- quatre cartes décrivant les rôles de chaque membres

En parallèle, nous nous sommes renseignés et avons effectué des tests sur une iframe JavaScript (langage côté client utilisé sur les sites web pour l'interactivité avec les utilisateurs) qui nous permet d'afficher directement une carte Google Maps avec une adresse depuis notre site, et nous l'avons donc intégrée.

<pre><body> <div class="map-container"> <div class="info-box">📍 Chargement... <iframe src="" allowfullscreen="" loading="lazy" referrerpolicy="no-referrer-when-downgrade"> </iframe> </div> </body></pre>	<pre><script> const ADDRESS = "22 Terr. Bellini, 92800 Puteaux"; document.addEventListener('DOMContentLoaded', () => { const iframe = document.querySelector('.map-container iframe'); const encodedAddress = encodeURIComponent(ADDRESS); iframe.src = `https://www.google.com/maps?q=\${encodedAddress}&output=embed`; const infoBox = document.querySelector('.info-box'); infoBox.textContent = `📍 \${ADDRESS}`; }); </script></pre>
HTML	JavaScript

Le script ci-dessus va donc s'exécuter quand la page est finie, et va par la suite chercher l'iframe à l'intérieur de ".map-container". C'est là que l'URL Google Maps va être injectée. Puis le script encode l'adresse pour qu'elle soit compatible avec une URL (espaces, accents, etc.). L'iframe est ensuite configurée pour afficher Google Maps centré sur l'adresse. Finalement, le script sélectionne ".info-box" et y met l'adresse en texte, avec un petit emoji de localisation.

Puis nous avons décidé de créer une base de données pour la gestion des différents événements avec MySQL (choisis pour sa simplicité d'utilisation et d'installation) :

En détails:

```
CREATE DATABASE IF NOT EXISTS guardia_app CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;  
USE guardia_app;
```

Le code commence par créer une base de données nommée `guardia_app` si elle n'existe pas déjà, en utilisant l'encodage “UTF-8” (`utf8mb4`) pour supporter tous les caractères, y compris les emojis, et la collation `utf8mb4_unicode_ci` pour des comparaisons insensibles à la casse et compatibles Unicode. La commande `USE guardia_app;` permet ensuite de sélectionner cette base pour que toutes les commandes suivantes s'appliquent sur elle.

```
-- Table des utilisateurs  
CREATE TABLE IF NOT EXISTS users (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    username VARCHAR(255) NOT NULL UNIQUE,  
    email VARCHAR(255) NOT NULL UNIQUE,  
    password VARCHAR(255) NOT NULL,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Ensuite, la table `users` est créée pour stocker les informations des utilisateurs. Elle contient un identifiant unique auto-incrémenté (`id`), un nom d'utilisateur (`username`) et une adresse e-mail (`email`) qui doivent être uniques, un mot de passe (`password`) obligatoire, et un champ `created_at` qui enregistre automatiquement la date et l'heure de création. Cette table utilise le moteur **InnoDB**, qui supporte les transactions et les clés étrangères, et reprend l'encodage UTF-8 de la base de données.

```
-- Table des événements
CREATE TABLE IF NOT EXISTS events (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    theme VARCHAR(25) NULL,
    description TEXT NULL,
    date DATE NOT NULL,
    start_time TIME NOT NULL,
    end_time TIME NOT NULL,
    end_date DATE NULL,
    address VARCHAR(255) NOT NULL,
    city VARCHAR(50) NOT NULL,
    postal_code VARCHAR(10) NOT NULL,
    country VARCHAR(100) DEFAULT 'France',
    image_path VARCHAR(255) NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Enfin, la table `events` est définie pour stocker les événements. Elle comprend un identifiant unique (`id`), le nom de l'événement (`name`), un thème (`theme`) et une description (`description`) optionnels, ainsi que la date et les horaires de début et de fin (`date`, `start_time`, `end_time`). Pour les événements sur plusieurs jours, un champ `end_date` est prévu. Les informations de localisation sont stockées dans `address`, `city`, `postal_code` et `country` (par défaut "France"). Un chemin vers une image (`image_path`) et la date de création (`created_at`) complètent la table. Là encore, **InnoDB** et UTF-8 sont utilisés pour garantir compatibilité et fiabilité.

Après cela, nous avons changé et adapté le fichier `index.html` et rajouter du php, langage (côté serveur) obligatoire pour produire un site web avec des pages dynamiques via un serveur Web. Pour ce faire, il a fallu créer plusieurs fichiers :

admin.php :

Ce fichier PHP constitue le “dashboard d’administration” d’une application de gestion d’événements appelée MonAgendaPro. Il commence par vérifier que l’utilisateur est connecté et qu’il a les droits d’administrateur, puis gère différentes actions via des formulaires sécurisés par un token CSRF, comme promouvoir/dégrader un utilisateur, supprimer un utilisateur ou un événement, tout en empêchant l’auto-suppression ou la perte de ses propres droits. Le script récupère également des statistiques globales (nombre d’utilisateurs, d’admins, d’événements passés et à venir) et les listes complètes d’utilisateurs et d’événements depuis la base de données, qu’il affiche dans un tableau interactif avec onglets. Enfin, il intègre des fonctionnalités front-end comme le mode nuit, la navigation par onglets, et des badges visuels pour distinguer les rôles et statuts, le tout dans une interface responsive et sécurisée.

events.php :

Ce fichier PHP génère la “page principale des événements” de MonAgendaPro pour un utilisateur connecté. Il commence par vérifier que l’utilisateur est connecté, puis récupère tous les événements depuis la base de données, triés par date et heure. Chaque événement est affiché sous forme de carte avec son nom, ses dates et heures, son adresse, sa description, son thème et éventuellement une image. La page intègre un volet “Google Maps” qui affiche la localisation d’un événement sélectionné ainsi qu’un widget météo qui récupère dynamiquement les prévisions via `get_weather.php` pour la ville et la date de l’événement. Des fonctionnalités front-end supplémentaires incluent la sélection d’un événement pour afficher ses détails sur la carte et la météo, un mode nuit personnalisable et une navigation conditionnelle pour les administrateurs, leur permettant de créer des événements ou accéder à l’administration. L’ensemble combine donc visualisation, interaction et informations pratiques pour les utilisateurs.

form.php :

Ce fichier PHP est le “formulaire sécurisé de création d’événements” pour MonAgendaPro, accessible uniquement aux administrateurs. Il permet d’ajouter des événements d’une journée ou sur plusieurs jours avec des champs pour le nom, le thème, la description, l’adresse complète, les dates/horaires et une image optionnelle, tout en validant les données côté serveur et client (y compris la taille et le type de l’image, et la cohérence des dates/heures). La carte Google Maps intégrée se met à jour en temps réel selon l’adresse saisie, et le formulaire propose une prévisualisation de l’image, un basculement entre mode jour/nuit sauvegardé dans `localStorage`, ainsi que des alertes UX pour guider l’utilisateur. Un token CSRF et la vérification de l’accès administrateur assurent la sécurité, et après soumission l’événement est inséré dans la base de données avant de rediriger vers la liste des événements.

index.php :

Ce fichier PHP est le **point d’entrée principal de l’application**, qui ne contient aucun traitement côté serveur autre qu’une redirection immédiate vers la page d’accueil `views/index.html`, assurant que toute requête arrivant à la racine du site est envoyée vers l’interface utilisateur.

login.php :

Ce fichier PHP gère “l’authentification des utilisateurs” : il vérifie que la requête est bien en POST, valide le token CSRF, récupère et sécurise le nom d’utilisateur et le mot de passe, puis compare le mot de passe soumis avec le hash stocké en base de données. Si la vérification réussit, il initialise la session avec l’ID, le nom d’utilisateur et le statut admin, et redirige vers `events.php` ; sinon, il renvoie vers la page de login avec une erreur, tout en protégeant la session contre la fixation.

logout.php :

Ce script PHP “déconnecte un utilisateur” en vidant toutes les variables de session, détruisant la session en cours, puis en redirigeant automatiquement vers la page de login (`views/login.php`).

register.php :

Ce script PHP gère l’inscription d’un nouvel utilisateur : il vérifie que la requête est bien en POST, valide le token CSRF, nettoie et contrôle les champs (nom d’utilisateur, email, mot de passe), empêche la création d’un compte existant, hache le mot de passe, insère l’utilisateur dans la base de données, régénère l’ID de session pour la sécurité, connecte automatiquement l’utilisateur et le redirige vers `events.php` , tout en gérant les erreurs avec des redirections appropriées.

Mis à part ces fichiers, nous avons apporté quelques modifications au HTML et aux JavaScript pour bien intégrer le PHP au projet.

Ensuite, une couche de sécurité a été mise en place grâce à la CI/CD et à SonarQube. La CI/CD est un ensemble de pratiques d’automatisation qui permettent de développer, tester et déployer un logiciel de manière rapide et fiable :

- La CI (intégration continue) consiste à fusionner fréquemment du code dans un dépôt central où des tests automatisés vérifient la qualité à chaque changement

- Tandis que la CD (livraison ou déploiement continu) automatise la préparation et éventuellement le déploiement en production du logiciel, réduisant ainsi les erreurs, les délais et les interventions manuelles.

La CI/CD est donc déployé dans le répertoire github .github/workflows afin de le mettre en place dans github action. De ce fait, nous avons créé 2 fichiers .yml (yaml, langage semblable en json qui permet de structurer des données de manière simple et plus compréhensible que le json). Un fichier CI.yml et un fichier CD.yml ont donc été initialiser :

Explication pour la CI :

Partie 1 — Build & Test :

La première partie correspond au job “Build & Test”, exécuté automatiquement à chaque push sur la branche `main`. Il récupère le code, installe l’environnement PHP requis et vérifie la syntaxe de l’ensemble des fichiers PHP afin de garantir que le projet est valide avant d’autoriser l’exécution des étapes suivantes du pipeline.

Partie 2 — SonarCloud Scan :

La seconde partie correspond au job “SonarCloud Scan”, exécuté uniquement après la réussite du “Build & Test”. Il récupère le code et lance une analyse automatisée via SonarCloud afin d’évaluer la qualité du code, détecter bugs, vulnérabilités, duplications et mauvaises pratiques, en utilisant les secrets GitHub pour l’authentification. Cette étape génère un rapport complet de qualité logicielle.

Partie 3 — Deploy to Server :

La dernière partie correspond au job “Deploy to Server”, exécuté uniquement après la réussite du “Build & Test”. Il récupère le code puis réalise les opérations nécessaires au déploiement, généralement via une connexion au serveur pour transférer les fichiers et mettre en ligne la nouvelle version de l’application. Cette étape clôture le pipeline en assurant la mise en production du projet.

Explication pour la CD :

Partie 1 — Configuration générale :

La configuration générale du workflow, nommé "Continuous Deployment", définit ses conditions de déclenchement : il s'exécute automatiquement à chaque push sur la branche “main”, garantissant que les vérifications et le déploiement ne concernent que la version principale du projet et évitent toute action accidentelle sur d'autres branches.

Partie 2 — Jobs Build & verify :

Le workflow automatise le processus de validation et de déploiement : il vérifie d'abord la syntaxe et la configuration du code PHP via le job “Build & Verify”, puis, après succès de cette étape, prépare le projet pour sa mise en production avec le job “Deploy to Production”, garantissant que seules des versions de code correctes et valides sont publiées.

Exemple de code CI/CD :

<pre>1 name: CI 2 3 on: 4 push: 5 branches: 6 - main 7 8 jobs: 9 build-and-test: 10 name: Build & Test 11 runs-on: ubuntu-latest</pre>	<pre>1 name: Continuous Deployment 2 3 on: 4 push: 5 branches: 6 - main 7 8 jobs: 9 build: 10 name: Build & Verify 11 runs-on: ubuntu-latest</pre>
CI	CD

SonarQube, lui, est un outil d'analyse automatique de la qualité du code qui détecte les bugs, vulnérabilités de sécurité et mauvaises pratiques dans un projet logiciel. Il fonctionne en analysant le code source (JavaScript, PHP, etc.) et en produisant un tableau de bord clair avec des indicateurs comme la dette technique, la couverture de tests ou les duplications. Intégré dans la pipeline CI/CD, SonarQube aide à maintenir un code propre, fiable et conforme aux standards. Ainsi, grâce à SonarQube nous avons pu corriger des erreurs de sécurité rapidement et très simplement juste en suivant ses indications.

Par la suite, nous avons ajouté une API. Pour cela, nous avons créé des fichiers PHP supplémentaires :

config.php (ci-dessous l'exemple des lignes 4 à 8) :

```
4  class Config {
5      // Clé API OpenWeatherMap depuis les variables d'environnement
6      public static function getApiKey(): string {
7          return getenv('OPENWEATHER_API_KEY') ?: '';
8      }
9      // Lien vers des résultats météo
```

Ce fichier PHP définit une classe `Config` dans l'espace de noms `Privee`, destinée à centraliser les paramètres liés à l'utilisation de l'API OpenWeatherMap : il contient une méthode statique qui récupère la clé API depuis les variables d'environnement pour éviter de l'exposer directement dans le code, ainsi que deux constantes indiquant la langue des données météo (français) et les unités de mesure utilisées (système métrique).

get_weather.php :

Ce fichier PHP sert de “point d’entrée API” pour retourner des données météo au format JSON. Il inclut la classe `WeatherService`, récupère les paramètres depuis l’URL (`\$_GET`), et vérifie que `city` et `date` sont fournis. Si l’un de ces paramètres est manquant, il renvoie immédiatement un message d’erreur JSON avec un message d’erreur.

```
try {
    $weather = WeatherService::getWeatherForDate($city, $date, $country);

    if ($weather === null) {
        echo json_encode(['error' => 'Ville non trouvée ou API indisponible']);
        exit;
    }

    // Ajouter l'icône Font Awesome
    if (isset($weather['main'])) {
        $weather['icon_fa'] = WeatherService::getWeatherIcon($weather['main']);
    }

    echo json_encode($weather);
} catch (Exception $e) {
    echo json_encode(['error' => 'Erreur: ' . $e->getMessage()]);
}
```

Ce bloc récupère la météo pour une ville et une date via `WeatherService`, ajoute l’icône Font Awesome correspondante, renvoie les données en JSON, et gère les erreurs ou exceptions en retournant un message JSON approprié (ci-dessus extrait du code des lignes 16 à 27).

WeatherService.php (ci-dessous extrait des lignes 90 à 99) :

```
90     public static function getWeatherForDate($city, $date, $country = 'FR') {
91         // Nettoyer la ville (enlever espaces et caractères spéciaux)
92         $city = trim($city);
93         if (empty($city)) {
94             return [
95                 'available' => false,
96                 'reason' => 'no_city',
97                 'message' => 'Ville non spécifiée'
98             ];
99         }

```

Enfin, le dernier fichier, aussi le plus important fournit un service PHP pour récupérer et traiter les prévisions météorologiques d'une ville via l'API OpenWeatherMap, en renvoyant des données détaillées ou simplifiées selon la date demandée.

Ce fichier définit une classe WeatherService dans le namespace Privee. L'objectif principal de cette classe est de fournir des prévisions météorologiques à partir de l'API OpenWeatherMap. Le fichier commence par inclure une configuration externe via `require_once __DIR__ . '/config.php';`.

Cette configuration est censée contenir au minimum une clé API (`Config::getApiKey()`) et probablement d'autres constantes comme l'unité de mesure et la langue (`Config::WEATHER_UNITS` et `Config::WEATHER_LANG`).

La classe WeatherService utilise des méthodes statiques, ce qui signifie qu'on peut appeler ses fonctions directement sans créer une instance de la classe, par exemple:

```
WeatherService::getForecast('Paris');
```

La méthode `init()` est très simple : elle initialise la clé API en la récupérant depuis la configuration. Cette clé sera utilisée pour authentifier toutes les requêtes vers OpenWeatherMap.

La méthode privée `fetchUrl($url)` est le moteur qui effectue les requêtes HTTP vers l'API. Elle utilise un des deux méthodes suivantes :

- cURL pour faire une requête HTTP GET sécurisée avec des options comme le timeout, le suivi des redirections et un user-agent personnalisé.
- `file_get_contents` en fallback si cURL n'est pas disponible, avec un contexte HTTP et SSL désactivant la vérification des certificats (ce qui peut poser des problèmes de sécurité en production, mais permet de contourner des erreurs SSL courantes).

Cette double méthode garantit que le service peut fonctionner même sur des environnements PHP limités.

La méthode `getForecast($city, $country = 'FR')` est responsable de récupérer les prévisions sur 5 jours pour une ville donnée. Elle construit l'URL de l'API avec la ville, le pays, la clé API, les unités et la langue, puis appelle `fetchUrl()` pour récupérer les données JSON. Les données sont ensuite décodées en tableau PHP via `json_decode()`.

Si l'API retourne une erreur (par exemple si la ville n'existe pas), la méthode renvoie null. Sinon, elle renvoie le tableau complet des prévisions, comprenant les prévisions toutes les 3 heures sur 5 jours, ainsi que des informations sur la ville.

La méthode `getWeatherForDate($city, $date, $country = 'FR')` elle, permet de récupérer la météo pour un jour précis. Son fonctionnement est le suivant :

- Vérifie que la ville est renseignée.
- Récupère les prévisions via `getForecast()`.
- Compare la date demandée à la date actuelle et à la date maximale disponible

Selon le cas, plusieurs situations sont gérées :

- Date passée → renvoie un message indiquant que l'événement est passé.
- Date trop lointaine → renvoie la dernière prévision disponible, avec un indicateur partiel pour signaler que ce n'est pas exact.
- Date dans la plage disponible → cherche la prévision la plus proche de midi pour ce jour afin d'avoir une donnée représentative. Si aucune prévision exacte n'est trouvée, elle prend la première disponible ce jour-là ou après.

La méthode renvoie alors un tableau contenant les informations principales : température, humidité, description météo, vent, nuages, et l'icône correspondante.

Enfin, des méthodes utilitaires sont définies :

- `getDaysUntil($date)` : calcule le nombre de jours entre aujourd'hui et la date cible.
- `getWeatherIcon($weatherMain)` : retourne l'icône Font Awesome correspondante au type de météo (Clear, Clouds, Rain, etc.). Si le type n'est pas reconnu, une icône par défaut est utilisée

Après cela, un fichier gitignore a été créé dans le but d'ignorer certains fichiers lors du push de la branche main sur github :

```
1  sonar-project.properties
2
3  # IDE
4  .vscode
5  .vscode/settings.json
```

Et finalement, afin de clôturer ce projet, nous avons mis en place des dockers compose afin de visualiser notre projet en rendu final. La aussi, afin de mieux optimiser notre site, nous avons mis en place un fichier dockerignore en plus des fichiers de docker compose (fichier permettant la création, la gestion et en somme, l'implémentation de notre site dans un environnement viable) afin de ne pas démarrer des fichiers inutilisés.