Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE CHARGÉ
DE L'EMPLOI

DOSSIER PROFESSIONNEL (DP)

Nom de naissance ▸ Lucbert
Nom d'usage ▸ Lucbert
Prénom ▸ Baptiste
Adresse ▸ 8bis rue de la marne 95220 Herblay-sur-Seine

Titre professionnel visé

TITRE PROFESSIONNEL CONCEPTEUR DEVELOPPEUR D'APPLICATIONS

MODALITE D'ACCES :

☒ Parcours de formation
☐ Validation des Acquis de l'Expérience (VAE)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel. **Ce titre est délivré par le Ministère chargé de l'emploi.**

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen**.

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

1. Des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
2. Du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
3. Des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
4. De l'entretien final (dans le cadre de la session titre).

[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]

Ce dossier comporte :

- Pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- Un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- Une déclaration sur l'honneur à compléter et à signer ;
- Des documents illustrant la pratique professionnelle du candidat (facultatif)
- Des annexes, si nécessaire.

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.

http://travail-emploi.gouv.fr/titres-professionnels

Sommaire

Exemples de pratique professionnelle

PRATIQUE PROFESSIONNELLE

Activité-type 1 Développer une application sécurisée

Exemple n°1 ▸ **Développement de l'API Symfony sécurisée pour la réception des données IoT**

Dans le cadre du projet de supervision IoT LoRaWAN développé pour Homeeguard, j'ai été chargé de concevoir et d'implémenter une API REST sécurisée sous Symfony, jouant un rôle central dans la communication entre les bridges ESP32, les capteurs LoRa et le tableau de bord client. Mon rôle consistait à permettre aux bridges d'envoyer des trames cryptées contenant les mesures environnementales (température, humidité, mouvements MPU), les états des interrupteurs, ainsi que toutes les métadonnées nécessaires à la supervision en temps réel.

Pour cela, j'ai mis en place un système d'authentification basé sur une clé API unique attribuée à chaque client Homeeguard. À chaque requête entrante, l'API vérifie la présence et la validité du champ *clientId* avant d'autoriser tout traitement. J'ai également implémenté un contrôle approfondi de l'intégrité des trames : vérification du *nodeId*, *bridgeId*, cohérence du timestamp global, conformité au format JSON attendu et détection des valeurs anormales. Les données valides sont ensuite traitées, normalisées, puis enregistrées dans la base MySQL via Doctrine ORM, notamment dans les tables liées aux équipements, aux logs des mesures et aux métadonnées de configuration. J'ai également intégré une gestion automatisée des "deltas" : lorsque le client modifie un paramètre dans l'interface, l'API enregistre ces nouvelles valeurs afin qu'elles soient renvoyées automatiquement au prochain échange avec le node.

L'ensemble de ce travail a été réalisé sur mon environnement de développement local, sous macOS. J'ai utilisé Symfony CLI, Composer, MySQL et Postman pour les tests unitaires et fonctionnels. Le Symfony Profiler m'a permis de surveiller en détail les requêtes, les performances, les éventuelles erreurs et la structure des réponses. J'ai travaillé à partir d'une base documentaire existante tout en effectuant des recherches complémentaires pour garantir conformité, robustesse et sécurité, en particulier vis-à-vis des contraintes du milieu IoT : trames très courtes, bande passante limitée, cryptage AES côté microcontrôleur et faible puissance de calcul des nodes.

Le développement s'est appuyé sur différentes technologies : PHP 8, Symfony 6, Doctrine ORM pour la gestion de la base MySQL, Symfony Validator pour la validation stricte des données JSON reçues, ainsi que les journaux de logs produits par le bridge afin d'analyser les trames dans leur format brut. Le matériel utilisé incluait un Mac pour le développement, ainsi que des bridges ESP32 et des nodes LoRaWAN afin de tester des trames réelles, en conditions proches du terrain. Pour la partie sécurité, j'ai mis en place un ensemble de protections permettant de garantir la résistance de l'API face aux entrées non conformes : filtrage systématique, validation des types, contrôle des valeurs limites, gestion d'erreurs normalisées, blocage des requêtes mal formées et authentification obligatoire par clé API.

Bien que le développement de l'API ait été mené en autonomie, plusieurs échanges réguliers avec mon maître d'alternance ont permis de valider certains points structurants : la définition

précise des endpoints, la normalisation du format JSON des trames LoRaWAN, les règles de sécurité attendues pour l'environnement IoT ainsi que les exigences métier concernant le stockage des données, la logique d'alertes et la supervision temps réel.

Ce travail s'est inscrit dans un contexte professionnel concret, au sein de l'entreprise Homeeguard, dans le service Développement logiciel & Supervision IoT. Il s'est déroulé du 10/03/2025 au 28/10/2025 et s'intégrait dans une démarche globale de conception d'un écosystème IoT complet, sécurisé et pilotable depuis une interface web moderne.

Cette API constitue aujourd'hui un élément fondamental du projet. Elle garantit la sécurité du système grâce à l'authentification et à la validation stricte de toutes les trames entrantes. Elle assure également la fiabilité du stockage et de la supervision grâce aux logs, aux métadonnées et aux entités structurées qui permettent d'historiser les événements. Elle est compatible avec les bridges actuels et futurs, ce qui ouvre la voie à une évolution progressive du parc de capteurs. Enfin, elle représente la base technique indispensable au bon fonctionnement du tableau de bord client, qui repose entièrement sur ses données pour afficher l'état des équipements, les alertes, les cartes et les statistiques.

Sur le plan professionnel, cette mission m'a permis d'approfondir mes compétences en sécurité backend, en architecture logicielle IoT et en bonnes pratiques OWASP, tout en développant une compréhension avancée du fonctionnement des échanges entre microcontrôleurs embarqués et API web modernes. Elle m'a également conforté dans ma capacité à mener un développement complet : conception, sécurité, implémentation, tests et validation métier.

Exemple n°2 ▸ **Implémentation de l'authentification par clé API (clientId / apiKey)**

Dans le cadre de la sécurisation de l'API IoT du projet Homeeguard, j'ai développé un mécanisme d'authentification reposant sur une clé API unique pour chaque client. Ce système a pour objectif d'identifier formellement l'origine de chaque requête envoyée par un bridge LoRaWAN, d'empêcher toute tentative d'accès non autorisé à l'API, de protéger la plateforme contre les injections de données ou les imitations de capteurs, et plus globalement d'assurer l'intégrité et l'authenticité des trames reçues.

La première étape a consisté à analyser les contraintes spécifiques au domaine IoT. Les capteurs et bridges LoRaWAN disposent d'une puissance très limitée, ce qui rend impossible l'utilisation de protocoles d'authentification complexes. Après étude, j'ai défini un modèle simple, robuste et adapté au matériel : une clé unique, générée automatiquement, transmise dans chaque requête envoyée vers l'API. Cette clé est ensuite contrôlée systématiquement par le serveur.

Sur le plan technique, j'ai ajouté un champ *apiKey* dans l'entité Comptes et Utilisateur, afin qu'un client Homeeguard puisse disposer d'une clé unique associée à son matériel. Une génération sécurisée du token a été mise en œuvre, puis j'ai intégré dans l'API un contrôle strict à chaque réception de trame JSON. L'API vérifie que la requête contient bien un champ *clientId*, que ce champ n'est pas vide et qu'il correspond réellement à un compte enregistré en base. Si la clé est absente, mal formatée ou erronée, la requête est rejetée. Voici un extrait du code réel illustrant cette logique :

```
$data = json_decode($request->getContent(), true);
if (!$data || !isset($data['clientId'])) {
    return new JsonResponse(['error' => 'Format JSON invalide'], 400);
}

$client = $em->getRepository(Comptes::class)->findOneBy(['apiKey' => $data['clientId']]);

if (!$client) {
    return new JsonResponse(['error' => 'Clé API invalide'], 401);
}
```

Pour valider le fonctionnement de ce mécanisme, j'ai réalisé une série de tests. Avec Postman, j'ai simulé plusieurs scénarios : requêtes sans clé API, requêtes avec une clé incorrecte, et requêtes correctement formées. J'ai ensuite testé la même logique avec un ESP32 Bridge en conditions quasi réelles sur mon réseau Wi-Fi local. Les résultats ont confirmé que toute trame dépourvue de clé API était immédiatement rejetée, qu'une clé erronée entraînait une réponse *401 Unauthorized*, et qu'un bridge authentifié possédant la clé valide pouvait transmettre des données en toute sécurité.

Pour mener à bien ce travail, j'ai utilisé Symfony 6, Doctrine ORM pour la persistance, PHP 8.2, une base MySQL pour stocker les comptes clients, ainsi que Postman pour les tests fonctionnels. J'ai également intégré plusieurs mécanismes de sécurité complémentaires : validation systématique des JSON reçus, filtrage des requêtes anonymes, gestion propre des erreurs HTTP et durcissement global des points d'entrée de l'API. Du côté matériel, j'ai utilisé un bridge ESP32 configuré pour envoyer des trames JSON simulées, ainsi que des nodes LoRaWAN produisant de vraies données chiffrées AES.

Bien que le développement ait été mené principalement en autonomie, j'ai travaillé en coordination ponctuelle avec mon tuteur chez Homeeguard pour valider les règles de sécurité, confirmer la structure exacte des trames envoyées par le firmware ESP32 et garantir une compatibilité parfaite entre le code backend et la logique implantée dans les microcontrôleurs.

Ce travail s'est déroulé dans le cadre professionnel de l'entreprise Homeeguard, au sein du service Développement logiciel & Supervision IoT, entre le 10/03/2025 et le 28/10/2025.

L'implémentation de ce mécanisme d'authentification représente une pierre angulaire de la sécurité du système. Elle empêche l'injection de fausses données dans la plateforme, garantit que seul le matériel autorisé peut communiquer avec l'API, protège les utilisateurs et contribue directement au respect des bonnes pratiques OWASP IoT. Elle renforce la fiabilité globale de la supervision Homeeguard tout en consolidant la confiance entre le client et l'infrastructure.

La mise en place de cette fonctionnalité m'a permis d'approfondir la sécurisation des API, la gestion des accès machine-to-machine et les problématiques liées à la manipulation de données issues d'objets connectés dans un environnement professionnel exigeant.

Exemple n°3 ▸ **Mise en place du chiffrement AES entre Node → Bridge → API**

Dans le cadre de la sécurisation des communications IoT du projet Homeeguard, j'ai conçu et mis en place un système complet de chiffrement de bout en bout basé sur l'algorithme AES 128 bits en mode CBC, couvrant toute la chaîne technique : des Nodes LoRaWAN (capteurs) jusqu'à l'API Symfony, en passant par le Bridge ESP32. L'objectif était de garantir la confidentialité des trames, de rendre très difficile toute injection ou falsification de messages, et d'assurer une cohérence parfaite entre le chiffrement côté microcontrôleurs, le routage LoRa et le traitement côté serveur.

La première étape a consisté à analyser les différentes options de chiffrement adaptées à un environnement IoT contraint. J'ai comparé plusieurs modes d'AES (ECB, CBC, GCM) en tenant compte des limites matérielles des ESP32 et de la simplicité d'implémentation avec les bibliothèques Arduino. L'AES en mode ECB a été écarté car trop faible sur le plan de la sécurité (absence de randomisation par bloc), et AES-GCM, bien que très robuste, était plus complexe et plus lourd à intégrer sur microcontrôleur. J'ai donc retenu AES-CBC 128 bits, qui offre un bon équilibre entre niveau de sécurité, performances et compatibilité Arduino, avec l'utilisation d'un vecteur d'initialisation (IV) pour éviter la répétition des blocs.

Côté Node, j'ai implémenté la logique de génération et de chiffrement des trames. Le Node construit d'abord une chaîne de caractères contenant les données (identifiant du node, température, humidité, état de l'interrupteur, mesures de l'accéléromètre/gyroscope…), puis cette chaîne est chiffrée avec AES-CBC avant d'être envoyée en LoRa. Comme le canal radio ne transporte que des octets, j'ai ajouté une couche d'encodage Base64 pour transformer le résultat du chiffrement en texte transmissible. Le code suivant illustre la fonction de chiffrement réellement utilisée sur le Node :

```
String encryptMessage(const String& plaintext) {
  byte plainBytes[plaintext.length() + 1];
  plaintext.getBytes(plainBytes, sizeof(plainBytes));

  int paddedSize = ((plaintext.length() / 16) + 1) * 16;
  byte padded[paddedSize] = {0};
  memcpy(padded, plainBytes, plaintext.length());

  byte encrypted[paddedSize];

  byte iv[16];
  memcpy(iv, AES_IV, 16);

  aes.set_key(AES_KEY, sizeof(AES_KEY));
  aes.cbc_encrypt(padded, encrypted, paddedSize / 16, iv);
```

```
  char output[base64_enc_len(paddedSize)];
  base64_encode(output, (char*)encrypted, paddedSize);

  return String(output);
}
```

Chaque trame envoyée par le Node est donc chiffrée, paddée correctement, puis encodée, ce qui permet de garantir que même en cas de sniffing radio, les données brutes restent illisibles pour un attaquant.

Sur le Bridge ESP32, j'ai implémenté la partie symétrique de la chaîne, avec une fonction de déchiffrement AES-CBC 128 bits. Le bridge reçoit la trame LoRa encodée en Base64, la décode en binaire, puis applique le déchiffrement avec la même clé et le même IV. À la fin du processus, on récupère la chaîne de texte en clair, prête à être analysée, validée et envoyée à l'API Symfony via HTTP. Voici un extrait du code réel utilisé côté bridge :

```
String decryptMessage(const String& encodedCiphertext) {
  int decodedLen = base64_dec_len(encodedCiphertext.c_str(), encodedCiphertext.length());
  byte ciphertext[decodedLen];
  base64_decode((char*)ciphertext, encodedCiphertext.c_str(), encodedCiphertext.length());

  byte decrypted[decodedLen];

  byte iv[16];
  memcpy(iv, AES_IV, 16);

  aes.set_key(AES_KEY, sizeof(AES_KEY));
  aes.cbc_decrypt(ciphertext, decrypted, decodedLen / 16, iv);

  return String((char*)decrypted);
}
```

Une fois le message déchiffré, le Bridge vérifie sa cohérence (présence des champs, format attendu), puis ne transmet à l'API Symfony que des données claires validées. Cela permet de filtrer les trames corrompues, altérées ou mal formées, et réduit les risques d'injection de données malveillantes dans la couche backend. L'API, de son côté, applique ensuite ses propres contrôles : validation JSON, vérification de l'API Key, contrôle de l'équipement et enregistrement en base.

J'ai réalisé plusieurs batteries de tests de bout en bout pour valider ce pipeline sécurisé Node → Bridge → API. D'abord en environnement de développement avec le Serial Monitor, pour vérifier visuellement le contenu des trames avant et après chiffrement/déchiffrement, puis en conditions plus réalistes avec le Bridge connecté en Wi-Fi et l'API Symfony exécutée en local sur mon Mac. J'ai testé des cas normaux (trames valides), mais aussi des cas d'erreur : messages

altérés volontairement, tronqués, avec des caractères modifiés, checksum implicite incorrect, ou encore duplication de trames pour simuler des tentatives de replay. Ces tests ont confirmé que les trames altérées provoquaient des problèmes au moment du déchiffrement ou de la validation, et qu'elles étaient correctement rejetées par la chaîne.

Pour mener à bien ce travail, j'ai utilisé Arduino IDE pour le développement et le flash des firmwares ESP32/LoRa, la bibliothèque AES adaptée à Arduino pour le chiffrement/déchiffrement, des fonctions d'encodage/décodage Base64, ainsi que la bibliothèque RadioHead Mesh pour la gestion des trames LoRa au niveau réseau. Côté serveur, j'ai utilisé Symfony 6 pour l'API et MySQL pour le stockage des données, en complément de l'API key et de l'horodatage pour sécuriser la validité des trames. Du point de vue matériel, le dispositif reposait sur un Node ESP32 équipé d'un module LoRa RA-02, un Bridge ESP32, et des capteurs tels que DHT22 (température/humidité) et MPU6050 (accéléromètre/gyroscope).

Ce travail a été principalement réalisé en autonomie dans le cadre du développement des couches sécurisées IoT. J'ai toutefois eu des échanges réguliers avec mon tuteur Homeeguard afin de valider les contraintes techniques du hardware (mémoire, fréquence, gestion énergétique), de s'assurer que les trames reçues par l'API étaient bien dans le format attendu et de définir ensemble les limites de sécurité pertinentes dans un contexte IoT (risques de sniffing, injection, falsification de capteurs).

L'ensemble de ces développements s'inscrit dans le contexte de l'entreprise Homeeguard, au sein du service Développement logiciel & Supervision IoT, sur la période du 10/03/2025 au 28/10/2025.

La mise en place de ce chiffrement AES de bout en bout m'a permis de construire un véritable pipeline de sécurité IoT : chiffrement matériel sur les nodes, déchiffrement sécurisé sur le bridge, authentification et validation côté serveur, puis communication bidirectionnelle fiable (par exemple pour les deltas de configuration envoyés du backend vers les nodes). C'est l'un des aspects les plus représentatifs du projet, car il nécessite à la fois une bonne compréhension des protocoles IoT, de la cryptographie symétrique, du développement bas niveau sur microcontrôleur et de la sécurité backend.

Ce module de chiffrement a significativement renforcé la résilience de la solution Homeeguard face aux attaques classiques dans le monde IoT (sniffing radio, replay, injection ou modification de trames) et constitue un argument fort pour une mise en production future dans un contexte réel.

Exemple n°4 ▸ **Sécurisation des formulaires Symfony (CSRF, validation, filtrage)**

Dans le cadre du projet de supervision IoT Homeeguard, j'ai été amené à sécuriser plusieurs formulaires Symfony utilisés par les clients sur le dashboard, en particulier le formulaire de connexion, le formulaire de configuration des seuils / deltas des capteurs (page *delta*), ainsi que certains formulaires liés à la gestion des équipements et aux préférences utilisateur. Ces formulaires sont au cœur de l'application, car ils permettent à l'utilisateur d'agir directement sur le comportement des capteurs et sur son compte ; ils constituaient donc un point sensible du point de vue de la sécurité.

L'objectif principal était de protéger l'application contre les attaques classiques ciblant les formulaires web (CSRF, XSS, inputs invalides ou malveillants), mais aussi de garantir que seules des données valides, cohérentes et conformes aux règles métier soient enregistrées en base de données. Il fallait également éviter qu'un utilisateur malveillant puisse modifier le HTML côté navigateur, contourner les vérifications front-end et injecter des valeurs inattendues pouvant casser le fonctionnement métier ou dégrader la qualité des données supervisées.

Concrètement, j'ai commencé par m'appuyer sur le système de formulaires de Symfony 6 pour encadrer strictement la structure des données entrantes. Chaque formulaire sensible (authentification, configuration des deltas, gestion d'équipement…) a été défini via des Form Types Symfony, ce qui permet de contrôler les champs attendus, leur type, ainsi que les contraintes qui s'y appliquent. J'ai activé et vérifié la protection CSRF sur ces formulaires : Symfony génère automatiquement un token CSRF pour chaque formulaire et vérifie sa validité lors de la soumission, ce qui empêche un attaquant externe de forcer l'envoi d'un formulaire à la place de l'utilisateur.

Sur le plan métier, j'ai ajouté des contraintes de validation sur les entités et/ou directement dans les formulaires, comme NotBlank, Range, Type, Email, etc., afin d'encadrer les champs critiques : par exemple, les deltas de température ou d'humidité devaient rester dans des plages raisonnables, et certains champs ne pouvaient jamais être vides. Les données envoyées par l'utilisateur sont ainsi d'abord validées par Symfony côté serveur, avant toute tentative d'enregistrement via Doctrine ORM dans MySQL.

Un exemple représentatif de ce travail concerne la page de configuration des *deltas* des équipements. L'utilisateur y renseigne des seuils qui sont ensuite appliqués côté Node pour décider si une variation est significative ou non. Dans le contrôleur, j'ai mis en place une boucle qui parcourt toutes les données du Request, filtre uniquement les clés pertinentes (celles qui se terminent par _DELTA) et persiste les valeurs dans l'entité EquipementAchetesMetadata de façon contrôlée :

```
foreach ($request->request as $cle => $valeur) {
    if (str_contains($cle, '_DELTA')) {
```

```
        $meta = $repoMeta->findOneBy(['equipement' => $equipement, 'cle' => $cle]);
        if (!$meta) {
            $meta = new EquipementAchetesMetadata($equipement, $cle);
        }
        // Ici, on ne stocke que des valeurs filtrées / contrôlées
        $meta->setValeur($valeur);
        $em->persist($meta);
    }
}
$em->flush();
$this->addFlash('success', 'Configuration sauvegardée');
```

Ce traitement est couplé aux validations Symfony : seules des valeurs considérées comme valides par le système (type, format, plages) arrivent réellement à ce stade. En parallèle, les templates Twig gèrent l'affichage des erreurs de validation au niveau des champs et utilisent les mécanismes intégrés pour inclure les tokens CSRF dans les formulaires. Les messages d'erreur et les *flash messages* permettent au client de comprendre pourquoi une soumission a échoué (champs manquants, valeurs hors plage, etc.), ce qui améliore à la fois l'expérience utilisateur et la sécurité.

Pour vérifier la robustesse de ces protections, j'ai réalisé des tests manuels en conditions proches du réel : en utilisant le navigateur, j'ai soumis des formulaires avec des champs vides, des valeurs hors limites, mais aussi en modifiant volontairement le HTML dans les DevTools (suppression de required, modification de min/max, injection de texte dans des champs numériques…). L'objectif était de s'assurer que même si la validation front-end est contournée, la validation serveur et les contraintes Symfony bloquent toujours les entrées invalides. J'ai également utilisé le Symfony Profiler pour suivre les requêtes, visualiser les éventuelles erreurs 4xx/5xx générées par la validation et m'assurer qu'aucune donnée incohérente ne parvenait jusqu'à la base.

Pour mener à bien cette tâche, je me suis appuyé sur : Symfony 6 pour la gestion des formulaires et de la sécurité, PHP 8 pour le développement backend, le composant Validator pour les contraintes, le composant Security pour les tokens CSRF, Twig pour le rendu des formulaires et des messages d'erreur, Doctrine ORM pour la persistance MySQL, ainsi que les outils de test classiques (navigateur, DevTools, Symfony Profiler). L'ensemble de ces développements a été réalisé en environnement local, dans le cadre du service Développement logiciel & Supervision IoT de l'entreprise Homeeguard, sur la période du 10/03/2025 au 28/10/2025.

Le travail a été mené principalement en autonomie, mais j'ai eu des échanges ponctuels avec mon encadrant afin de valider le comportement fonctionnel des formulaires, d'identifier les champs considérés comme critiques et de vérifier la cohérence entre les règles métier (par exemple les plages de valeurs des deltas configurables par le client) et les règles techniques (contraintes de validation côté serveur).

La sécurisation de ces formulaires Symfony a été un point clé du projet, car ces interfaces permettent à l'utilisateur de modifier le comportement des capteurs, d'accéder aux données sensibles de supervision et de gérer son compte client. Mettre en place ces protections m'a permis de mettre en œuvre concrètement des mesures de sécurité combinant à la fois le front et le back, de mieux comprendre les failles classiques (CSRF, XSS, injections, falsification de champs) et de renforcer significativement la robustesse globale de l'application. Au final, l'ensemble des saisies utilisateur passe désormais par une validation serveur stricte, ce qui contribue directement à la fiabilité, à la sécurité et à la qualité des données au cœur de la solution Homeeguard.

Exemple n°5 ▸ **Protection contre les attaques XSS sur le dashboard utilisateur**

Dans le cadre du dashboard client Homeeguard, j'ai dû traiter un enjeu de sécurité important : certaines données affichées à l'écran ne proviennent pas uniquement du backend maîtrisé, mais aussi de sources potentiellement risquées, comme des valeurs saisies ou modifiées par les utilisateurs (noms d'équipements, paramètres, deltas…), et des données venant de l'IoT (capteurs LoRaWAN, bridge ESP32). Même si le matériel est sous contrôle, un attaquant pourrait tenter de simuler un node ou d'injecter des valeurs malveillantes pour exécuter du JavaScript dans le navigateur, via une faille XSS. Mon travail a donc consisté à verrouiller l'ensemble de la chaîne d'affichage afin que, quoi qu'il arrive, aucune donnée reçue ne puisse se transformer en script exécutable côté client.

La première couche de protection que j'ai mise en place se situe dans Twig, le moteur de templates utilisé par Symfony. Par défaut, Twig applique un auto-escaping sur toutes les variables, ce qui signifie que les caractères HTML potentiellement dangereux (<, >, ", '…) sont automatiquement échappés avant d'être envoyés au navigateur. J'ai vérifié que cet auto-escaping restait bien activé sur l'ensemble des templates du dashboard et que les données issues de l'utilisateur ou de l'IoT n'étaient jamais rendues avec le filtre |raw. Dans les vues de type listes d'équipements ou d'alertes, je me suis assuré d'utiliser systématiquement des expressions du type :

```
<td>{{ equipement.nom }}</td>
<td>{{ alerte.message }}</td>
```

et jamais {{ alerte.message|raw }}. De cette manière, même si un utilisateur essayait d'enregistrer comme nom d'équipement ou comme message d'alerte une valeur du type :

```
<script>alert('XSS')</script>
```

le navigateur recevrait en réalité :

```
&lt;script&gt;alert('XSS')&lt;/script&gt;
```

La chaîne est affichée telle quelle au lieu d'être interprétée comme du code JavaScript, et aucun script ne s'exécute.

En complément de cette sécurisation à l'affichage, j'ai également renforcé le contrôle des entrées côté API / Symfony. Pour les champs configurables depuis l'interface (deltas, libellés, paramètres), j'ai mis en place plusieurs niveaux de protection : contraintes de validation via le composant Validator (Regex, Length, NotBlank, Type, etc.), et filtrage explicite avant l'enregistrement en base. Un exemple concret se trouve dans la mise à jour des deltas de

configuration d'un équipement. Lorsqu'un formulaire POST est soumis, je parcours toutes les données reçues, je filtre les clés qui concernent les seuils (*_DELTA) et j'applique un nettoyage sur les valeurs avant de les persister :

```
foreach ($request->request as $cle => $valeur) {
   if (str_contains($cle, '_DELTA')) {

      // Nettoyage des valeurs envoyées
      $valeur = strip_tags($valeur);
      $valeur = htmlspecialchars($valeur, ENT_QUOTES);

      $meta = $repoMeta->findOneBy(['equipement' => $equipement, 'cle' => $cle])
          ?? new EquipementAchetesMetadata($equipement, $cle);

      $meta->setValeur($valeur);
      $em->persist($meta);
   }
}
$em->flush();
$this->addFlash('success', 'Configuration sauvegardée');
```

L'utilisation combinée de strip_tags() et htmlspecialchars() permet de supprimer les balises HTML et d'échapper les caractères critiques. Ainsi, même si quelqu'un tente de forcer une balise <script> dans un champ de seuil, la valeur sera neutralisée avant de se retrouver en base de données, et a fortiori avant d'être renvoyée dans les vues Twig.

J'ai également pris en compte le cas où les données ne viennent pas directement d'un utilisateur web, mais des capteurs ou du bridge. Même si les nodes LoRaWAN et le bridge ESP32 sont gérés par la solution, un attaquant avancé pourrait tenter de simuler un node et d'envoyer des valeurs formatées en HTML dans les trames LoRa. Ces données pourraient ensuite être affichées dans les pages "alertes" ou "statistiques". Le fait d'avoir une politique systématique d'échappement dans Twig garantit que même ces données externes – considérées comme non fiables – ne peuvent pas être interprétées comme du code par le navigateur. En d'autres termes, l'origine de la donnée n'a pas d'importance : tout est traité comme du texte, jamais comme du code.

Pour valider l'efficacité de ces protections, j'ai mené une série de tests manuels XSS. J'ai essayé différentes charges malveillantes directement dans les formulaires et, dans certains cas, via des requêtes artificielles envoyées à l'API :

- <script>alert('XSS')</script>
- <img src=x onerror=alert(1)>
- "><script>alert(1)</script>
- variantes encodées (<script>, tentatives base64, etc.)

Après soumission, j'ai vérifié à la fois dans le navigateur et dans la base de données que :

- aucun code JavaScript ne s'exécutait,
- les valeurs affichées sur le dashboard étaient sanitized (échappées / nettoyées),
- aucune balise active ou événement JavaScript ne subsistait dans les données stockées.

Pour réaliser ce travail, je me suis appuyé sur Twig (avec l'auto-escaping activé), sur les filtres PHP htmlspecialchars() et strip_tags() côté API, sur le composant Validator de Symfony (contraintes Regex, Length, NotBlank, etc.), ainsi que sur Doctrine et les entités métier pour centraliser les règles de validation. Les tests ont été effectués avec différents navigateurs, les DevTools pour observer le HTML final, et des outils comme Burp Suite Community pour injecter des payloads XSS dans les requêtes et vérifier la résistance de l'application.

Ce travail a été réalisé en autonomie dans le cadre de la sécurisation du dashboard utilisateur, avec des échanges réguliers avec mon encadrant pour identifier les champs les plus sensibles (noms, descriptions, paramètres), définir les valeurs autorisées pour les deltas et choisir les scénarios de test à exécuter. Le tout s'inscrit dans le contexte de l'entreprise Homeeguard, au sein du service Développement logiciel & Supervision IoT, sur la période du 10/03/2025 au 28/10/2025.

La mise en place de ces protections contre les attaques XSS a été essentielle pour garantir l'intégrité visuelle de l'interface, la sécurité du navigateur du client, la confidentialité des données affichées et la robustesse globale de l'application. Ce travail m'a permis de consolider ma compréhension de la sécurité applicative Web et des bonnes pratiques OWASP côté front-end et back-end, tout en intégrant ces principes dans un contexte concret de supervision IoT où les données sont multiples, dynamiques et parfois issues de sources externes peu fiables.

Activité-type 2 — Concevoir et développer une application sécurisée organisée en couches

Exemple n°1 ▸ **Conception UML complète (use-case, séquences, architecture)**

Dans le cadre du projet de supervision IoT LoRaWAN pour Homeeguard, j'ai réalisé une conception UML complète de l'application avant et pendant le développement, afin de structurer l'architecture logicielle, clarifier les échanges entre les différentes couches (IoT, API, base de données, dashboard) et faciliter la compréhension du fonctionnement global par les parties prenantes techniques et métier. Cette phase de conception a été une étape clé pour transformer le besoin métier en un système cohérent, organisé et sécurisé.

J'ai commencé par une analyse détaillée des besoins et l'identification des acteurs impliqués. Côté métier, Homeeguard avait pour objectif de superviser des capteurs (température, humidité, mouvements, états d'interrupteur), de détecter des anomalies, de générer des alertes pertinentes et de fournir un dashboard clair et accessible aux clients (particuliers ou professionnels). À partir de ces besoins, j'ai formalisé les principaux acteurs : le client (utilisateur final du dashboard), l'administrateur Homeeguard, le bridge LoRaWAN, les nodes / capteurs, l'API Symfony et la base de données. Cette étape m'a permis de poser clairement « qui fait quoi » et de préparer les diagrammes suivants.

J'ai ensuite réalisé des diagrammes de cas d'utilisation (use-case) pour couvrir l'ensemble des fonctionnalités majeures. Un premier use-case global de supervision décrit le parcours standard d'un client : se connecter, consulter ses équipements, afficher les alertes récentes, visualiser la cartographie, consulter les statistiques et modifier la configuration des capteurs (seuils, deltas, paramètres). J'ai décliné ce use-case général en cas d'utilisation plus détaillés : configuration des seuils d'alerte d'un équipement via la page *Delta*, consultation de l'historique des alertes, visualisation des capteurs sur une carte, supervision de l'état en ligne / hors ligne des équipements. Pour améliorer la lisibilité, j'ai scindé cette vue fonctionnelle en deux blocs : d'un côté les use-cases UI / Dashboard (ce que voit et fait le client dans l'interface web), de l'autre les use-cases API / Sécurité / IoT (réception des trames, validation, stockage, gestion des deltas). Ces diagrammes m'ont servi de check-list fonctionnelle : vérifier que chaque besoin métier disposait bien d'au moins un cas d'utilisation.

Sur cette base, j'ai conçu plusieurs diagrammes de séquence pour détailler la dynamique exacte des échanges. Le premier scénario important décrit la chaîne Node → Bridge → API : le node LoRaWAN prépare une trame chiffrée (AES), l'envoie au bridge via LoRa, le bridge déchiffre et reconstruit un payload JSON, puis appelle l'endpoint HTTP POST /api/lora-data de l'API Symfony. Dans ce diagramme, on voit la validation du clientId (clé API), la vérification du bridgeId / serialNumber, la recherche de l'équipement associé, puis la création du log en base (température, humidité, MPU, état interrupteur, horodatage). Un second diagramme de séquence important modélise les interactions Dashboard → API : chargement de la page d'accueil (récupération du compte client, liste des équipements, alertes récentes, nombre

d'équipements en ligne), consultation filtrée des alertes, et mise à jour des deltas de configuration via formulaire (soumission du formulaire, traitement dans le contrôleur, mise à jour des entités Doctrine, persistance en base, retour de confirmation et rafraîchissement de l'interface). Ces séquences UML m'ont permis de visualiser précisément la circulation des données entre le front, l'API, la base et la couche IoT, et de vérifier la cohérence des enchaînements.

En parallèle, j'ai travaillé sur la modélisation de l'architecture logicielle sous forme de diagramme de classes et d'architecture en couches. J'ai structuré l'application autour de plusieurs couches bien séparées :

- une couche présentation, composée des templates Twig (accueil.html.twig, alertes.html.twig, delta.html.twig, cartographie.html.twig, statistiques.html.twig, etc.) et du JavaScript associé (gestion des tableaux, filtres d'alertes, cartes, notifications) ;
- une couche métier, construite autour des entités Symfony/Doctrine : Comptes (clé API, client), Utilisateur, EquipementAchetes, EquipementAchetesLogs (historique des mesures / alertes), EquipementAchetesMetadata (deltas, paramètres), Product, avec les règles métier associées (gestion des deltas, statut des équipements, agrégation d'historique, filtrage d'alertes) ;
- une couche d'accès aux données, basée sur les repositories Doctrine et la base MySQL (requêtes spécifiques : logs récents, équipements d'un client, métadonnées par équipement…) ;
- une couche IoT / infrastructure, qui englobe le firmware Node/Bridge (ESP32 + LoRa), le protocole LoRa, le chiffrement AES, et la couche HTTP qui relie le bridge à l'API Symfony.

Le diagramme de classes UML que j'ai construit met en avant les entités principales, leurs attributs essentiels (ex. : apiKey dans Comptes, serialNumber dans EquipementAchetes, temperature, humidity, accX/accY/accZ, horodatage dans EquipementAchetesLogs, cle / valeur dans EquipementAchetesMetadata) ainsi que leurs relations (OneToMany, ManyToOne, etc.). Cette vue structurée a été très utile pour concevoir la base de données, générer les entités et penser les requêtes d'exploitation (statistiques, alertes, supervision temps réel).

Tout ce travail de conception UML a été réalisé en amont et en parallèle du développement, à partir du cahier des charges Homeeguard et enrichi au fil des itérations : ajout de la notion de métadonnées configurables, intégration des deltas de capteurs, logs horodatés, états en ligne / hors ligne… J'ai utilisé principalement Draw.io pour produire les différents diagrammes (use-case, séquences, classes, architecture), en commençant parfois par des esquisses papier/crayon pour clarifier les flux avant de les formaliser.

Le travail a été réalisé principalement en autonomie, en tant que concepteur-développeur de la solution, puis présenté à mon encadrant technique Homeeguard pour validation. Ces échanges ont permis d'ajuster certains aspects, notamment la séparation claire entre les use-cases orientés UI (dashboard client) et ceux orientés IoT / API / sécurité, afin que chacun puisse lire les diagrammes selon son angle (métier, technique, réseau).

Ce travail de conception UML m'a permis de sécuriser la compréhension fonctionnelle du projet avant de coder, d'organiser proprement l'architecture en couches pour faciliter la maintenance, d'anticiper les impacts des choix techniques (radio LoRa, chiffrement, API, base de données) et de disposer de supports visuels réutilisables dans le dossier de projet comme lors de l'oral. Il a véritablement servi de colonne vertébrale au développement ultérieur, en alignant l'IoT, l'API Symfony, la base MySQL et le dashboard client autour d'un même modèle de référence.

Exemple n° 2 ▸ ***Architecture logicielle en couches — Nodes LoRaWAN → Bridge ESP32 → API Symfony → Dashboard Web***

Dans le cadre du projet IoT LoRaWAN Homeeguard, j'ai conçu une architecture logicielle complète, structurée en plusieurs couches, afin de garantir la maintenabilité, la sécurité et la robustesse de l'application tout au long de son cycle de vie. Cette architecture devait répondre à des exigences fortes : gérer des données provenant de capteurs LoRaWAN, assurer un traitement sécurisé côté passerelle et côté API, stocker et organiser les informations dans une base de données fiable, et mettre à disposition des utilisateurs un dashboard clair et performant.

La première étape a consisté en une analyse approfondie des besoins métier et techniques. J'ai étudié le fonctionnement global du système de supervision IoT : les nodes LoRaWAN émettent des données cryptées via radio, le bridge ESP32 collecte et déchiffre ces trames avant de les envoyer en HTTP à l'API Symfony, qui doit ensuite traiter, valider et stocker les informations. À partir de cette analyse, j'ai défini une architecture fonctionnelle organisée en couches : une couche capteurs (Nodes), une couche passerelle (Bridge ESP32), une couche backend (API REST sous Symfony), une couche de stockage (Doctrine + MySQL), et enfin une couche présentation (dashboard web, alertes, cartographie).

J'ai ensuite produit une conception UML complète, permettant de visualiser et valider l'architecture avant le développement. Plusieurs diagrammes ont été réalisés avec précision :

- des diagrammes de cas d'utilisation couvrant l'ensemble du parcours utilisateur (connexion, supervision, alertes, statistiques, configuration des équipements) ;
- des diagrammes de séquence IoT détaillant la chaîne Node → Bridge → API (émission chiffrée, réception, décryptage, envoi des données, validation du clientId et enregistrement en base) ;
- des diagrammes de séquence Dashboard → API montrant comment les données du client sont chargées, filtrées ou modifiées dans l'interface ;
- un modèle conceptuel de données (MCD) représentant les entités essentielles : comptes, utilisateurs, équipements achetés, logs horodatés, métadonnées configurables (deltas), et catalogue produit ;
- un diagramme de classes UML décrivant de manière structurée les relations OneToMany / ManyToOne entre les entités métier.

Cette architecture UML m'a servi de base pour passer au développement. J'ai créé l'ensemble des entités Symfony : Comptes, Utilisateur, EquipementAchetes, EquipementAchetesLogs, EquipementAchetesMetadata, et Product. J'ai également mis en place leurs repositories Doctrine, permettant d'effectuer des requêtes optimisées comme la récupération des alertes récentes, des équipements du client ou des métadonnées configurables pour un appareil donné. Sur cette base, j'ai structuré la couche métier autour de concepts clés tels que l'équipement, les alertes, les logs et les paramètres modifiables, tout en respectant les bonnes pratiques MVC de

Symfony. L'organisation en couches a permis de séparer clairement la logique front-end (Twig, JavaScript, cartographie, tableaux), la logique métier (calculs, règles d'alerte, gestion des seuils), et la logique de persistance (Doctrine + MySQL).

Ce travail a été mené dans un environnement local sur Mac, en toute autonomie, mais avec des échanges techniques réguliers pour valider les choix fonctionnels. J'ai utilisé divers outils pour concevoir et mettre en place cette architecture : Draw.io pour les diagrammes UML, Composer et Symfony CLI pour la gestion du backend, phpMyAdmin pour la base de données, Git/GitHub pour le versionnement, ainsi que du matériel IoT réel (ESP32 LILYGO T-Display avec gyroscope, DHT et interrupteur) pour développer et tester la couche IoT.

Ce chantier s'inscrit dans le contexte de l'entreprise Homeeguard, au sein du service Développement logiciel et Supervision IoT, entre le 10/03/2025 et le 28/10/2025. Il représente l'un des fondements les plus importants du projet, car il a conditionné la réussite de toute la chaîne IoT → Bridge → API → Dashboard.

En réalisant cette architecture en couches, j'ai pu structurer un système cohérent, évolutif et sécurisé, capable de traiter des données provenant de microcontrôleurs, de respecter les bonnes pratiques d'ingénierie logicielle, et de faciliter les développements futurs (CCP1, CCP3). Cette phase de conception a été déterminante pour anticiper les problématiques techniques, sécuriser le fonctionnement global et assurer la qualité et la pérennité de l'application.

Exemple n°3 ▸ **Modélisation et création des entités métier (Comptes, Utilisateur, Équipements, Logs, Metadata)**

Dans le cadre du projet de supervision IoT LoRaWAN pour Homeeguard, j'ai conçu et implémenté l'ensemble des entités métier représentant la structure logique et fonctionnelle du système. Cette étape a été essentielle pour traduire les besoins métier en objets logiciels concrets, reflétant les clients, les équipements IoT, les données remontées par les capteurs, les alertes et les paramètres configurables. L'objectif était de disposer d'un modèle clair, scalable et sécurisé, capable de soutenir le fonctionnement complet du dashboard Homeeguard.

J'ai commencé par une analyse approfondie du domaine métier, afin d'identifier les éléments fondamentaux à représenter. Plusieurs entités centrales se sont dégagées :

- Comptes, représentant les clients Homeeguard et intégrant notamment la *clé API* permettant l'authentification des bridges IoT ;
- Utilisateur, représentant les comptes utilisateurs rattachés à un compte client, avec leurs identifiants, rôles et accès au dashboard ;
- EquipementAchetes, représentant les équipements IoT installés chez le client (bridge, capteurs), identifiés par leur *serialNumber* ;
- EquipementAchetesLogs, représentant l'historique des données remontées par les capteurs (température, humidité, accéléromètre, horodatage, etc.) ;
- EquipementAchetesMetadata, représentant les paramètres configurables du capteur (deltas, seuils, paramètres), modifiables depuis le dashboard ;
- Product, représentant le catalogue de produits Homeeguard avec leurs caractéristiques.

À partir de cette analyse, j'ai défini les relations principales entre ces objets : un Compte possède plusieurs Utilisateurs et plusieurs EquipementsAchetes, et chaque équipement possède plusieurs Logs ainsi que plusieurs Metadata. Ces liens ont été modélisés à travers des relations Doctrine OneToMany et ManyToOne, permettant de garantir l'intégrité de la base de données et la cohérence des échanges entre l'API et le dashboard.

J'ai ensuite conçu l'ensemble des entités Symfony/Doctrine en définissant leurs attributs (types, tailles, contraintes), leurs relations, et leurs règles métier. Par exemple, la relation entre un équipement et ses logs est représentée de manière explicite dans les deux entités via une relation bidirectionnelle Doctrine :

```
// Dans EquipementAchetes
/**
 * @ORM\OneToMany(targetEntity=EquipementAchetesLogs::class,
mappedBy="equipement")
 */
private Collection $logs;
```

```
// Dans EquipementAchetesLogs
/**
 * @ORM\ManyToOne(targetEntity=EquipementAchetes::class,  inversedBy="logs")
 * @ORM\JoinColumn(nullable=false)
 */
private ?EquipementAchetes $equipement = null;
```

J'ai intégré dans ces entités des attributs en lien direct avec la sécurité et le métier. Par exemple, le champ apiKey dans l'entité *Comptes* permet d'authentifier les requêtes venant des bridges IoT; l'attribut serialNumber dans *EquipementAchetes* permet d'identifier précisément un équipement à partir des données reçues par l'API ; et les clés de métadonnées telles que TEMP_DELTA ou HUM_DELTA permettent de configurer les seuils d'alerte directement depuis l'interface client.

Lors de la réception des données IoT, l'API crée et alimente l'entité *EquipementAchetesLogs* avec les valeurs reçues :

```
$log = new EquipementAchetesLogs();
$log->setEquipement($equipement);
$log->setTemperature($data['temperature']);
$log->setHumidity($data['humidity']);
$log->setAccX($data['accX']);
$log->setAccY($data['accY']);
$log->setAccZ($data['accZ']);
$log->setHorodatage(new \DateTime());
$em->persist($log);
$em->flush();
```

Une fois les entités finalisées, j'ai généré les migrations Doctrine, puis procédé à la création et à la mise à jour du schéma de base de données MySQL via la commande :
php bin/console doctrine:migrations:migrate.
J'ai vérifié ensuite l'intégrité des relations, des index et du modèle global directement dans phpMyAdmin.

Ce travail a été réalisé dans un environnement local Symfony/MySQL, en étroite cohérence avec les besoins réels de Homeeguard, notamment en matière de supervision en temps réel, d'alertes et de configuration dynamique.

Pour mener ce développement, j'ai utilisé Symfony 6, Doctrine ORM pour le mapping objet-relationnel, MySQL pour le stockage des données, ainsi que les outils de développement courants comme VS Code/PHPStorm, Composer et la console Symfony. L'ensemble du travail a été mené en autonomie, avec des échanges réguliers avec mon encadrant pour valider la pertinence de la modélisation, ajuster certains attributs ou étendre le modèle en fonction de nouveaux besoins (ex. : ajout de nouvelles métadonnées ou de logs spécifiques).

Cette modélisation métier, réalisée au sein de l'entreprise Homeeguard, service Développement logiciel & Supervision IoT, entre le 10/03/2025 et le 28/10/2025, a joué un rôle essentiel dans la réussite du projet. Elle a permis de garantir un schéma de base de données cohérent, de faciliter la création des contrôleurs et du dashboard client, d'assurer l'évolutivité du système pour accueillir de nouveaux capteurs, et de fournir une traçabilité complète des données remontées par les équipements IoT.

Par cette mission, j'ai consolidé mes compétences en modélisation objet, en architecture logicielle en couches, en structuration des données et en maîtrise avancée de Doctrine au sein de Symfony.

Activité-type 3 Préparer le déploiement d'une application sécurisée

Exemple n° 1 ▸ ***Installation et exécution de l'API Symfony en environnement local sous Mac***

Dans le cadre du développement de la plateforme Homeeguard, j'ai pris en charge l'installation, la configuration et l'exécution de l'API Symfony en environnement local sur mon Mac. Cette API constitue le point central de communication entre les bridges LoRaWAN, les capteurs et le dashboard web, il était donc indispensable qu'elle soit correctement installée, stable et facilement testable en conditions proches du réel.

J'ai d'abord mis en place tout l'environnement technique nécessaire sous macOS. Pour cela, j'ai installé Homebrew, le gestionnaire de paquets, afin de disposer d'une base propre et maintenable. À partir de Homebrew, j'ai installé PHP et Composer (brew install php, brew install composer), puis l'outil Symfony CLI (brew install symfony-cli/tap/symfony-cli). Cette étape m'a permis d'avoir une version récente de PHP, un gestionnaire de dépendances fonctionnel et un outil dédié au lancement et au suivi des projets Symfony.

Une fois l'environnement prêt, j'ai récupéré le projet backend Symfony. Selon le contexte, cela passait soit par un git clone du dépôt, soit par la récupération du code source dans un dossier de travail local. J'ai ensuite exécuté composer install pour télécharger toutes les dépendances PHP du projet (Symfony, Doctrine, composants Security, Validator, Serializer, etc.). Après cette étape, l'arborescence du projet était opérationnelle et prête à être configurée.

J'ai ensuite créé un fichier .env.local spécifique à mon environnement Mac, afin d'isoler la configuration locale de la configuration générique du projet. Dans ce fichier, j'ai paramétré la connexion à la base MySQL locale, les variables d'environnement nécessaires au mode développement (APP_ENV=dev) et les paramètres propres au projet, comme l'URL de la base Homeeguard. Par exemple :

```
APP_ENV=dev
DATABASE_URL="mysql://root:root@127.0.0.1:8889/homeeguard"
```

En parallèle, j'ai démarré le serveur MySQL (via Homebrew, MAMP ou un autre gestionnaire) et vérifié que la base de données était accessible. J'ai ensuite utilisé les commandes Doctrine pour créer et mettre à jour le schéma :

```
php bin/console doctrine:database:create puis php bin/console doctrine:migrations:migrate.
```

Ces commandes ont permis de générer toutes les tables nécessaires : comptes, utilisateurs, équipements achetés, logs, métadonnées, produits, etc. J'ai contrôlé la bonne création de la base via phpMyAdmin ou un autre outil (TablePlus), en vérifiant la présence des clés étrangères, des types de colonnes et de la cohérence globale.

Une fois la base opérationnelle, j'ai lancé l'API Symfony en local en utilisant principalement la commande :
symfony server:start
Cette commande démarre un serveur HTTP local spécialement adapté aux projets Symfony, avec rafraîchissement automatique, gestion simplifiée des logs, et possibilité d'activer HTTPS si nécessaire. L'API était alors accessible via une URL du type :

```
http://127.0.0.1:8000/api/lora-data
```

Ce point d'entrée est celui utilisé par le bridge ESP32 pour envoyer les données IoT au format JSON.

J'ai ensuite réalisé une série de tests fonctionnels et techniques pour valider le bon fonctionnement de l'API. Avec Postman, j'ai simulé des requêtes HTTP POST vers les endpoints de l'API en envoyant des JSON similaires à ceux du bridge (avec clientId, bridgeId, valeurs de capteurs, etc.). Cela m'a permis de vérifier :

- que le JSON était bien reçu et décodé,
- que la vérification du clientId (clé API) fonctionnait,
- que l'équipement correspondant au bridgeId était bien retrouvé dans la base,
- que les logs étaient correctement créés et enregistrés via Doctrine,
- que l'API renvoyait une réponse JSON cohérente au bridge (par exemple, status, timestamp, éventuelles configurations/deltas à renvoyer).

En complément, j'ai effectué des tests en conditions IoT réelles en connectant le bridge ESP32 au réseau local de mon Mac. Le bridge envoyait alors de vraies trames issues des capteurs, ce qui m'a permis de valider bout à bout le flux Node → Bridge → API locale → base MySQL.

Tous ces travaux ont été réalisés entièrement en autonomie, dans un contexte de développement individuel, avec des échanges ponctuels avec mon encadrant Homeeguard pour valider la configuration, les endpoints critiques et les données nécessaires aux tests. Le contexte était celui de l'entreprise Homeeguard, au sein du service *Développement logiciel – Supervision IoT*, sur la période du 10/03/2025 au 28/10/2025.

Cette mise en place de l'API Symfony en environnement local a été une étape essentielle du projet : elle m'a permis de disposer d'un environnement de travail complet et réaliste, de tester la communication en temps réel avec le bridge ESP32, d'intégrer progressivement les fonctionnalités (gestion des logs, métadonnées, alertes, filtrage par client) et de déboguer efficacement. Grâce à ce setup, j'ai pu garantir que l'API serait suffisamment robuste et stable avant d'envisager un déploiement vers un environnement plus proche de la production.

Exemple n°2 ▸ **Gestion du réseau et configuration sécurisée du Bridge (Wi-Fi, DNS, redondance)**

Dans le cadre du développement du système Homeeguard, j'ai pris en charge la configuration réseau et la sécurisation du bridge LoRaWAN (ESP32) afin d'assurer une communication fiable entre les capteurs (Nodes), l'API Symfony et, en cas de problème, un bridge secondaire de secours. L'objectif était de garantir à la fois la continuité de service, la résilience en cas de panne et la sécurité des échanges dans un contexte IoT contraint.

J'ai d'abord mis en place une connexion Wi-Fi sécurisée et robuste au niveau du firmware du bridge. Pour cela, j'ai développé une fonction dédiée connectToWiFi() qui vérifie en permanence l'état de la connexion réseau et tente une reconnexion automatique en cas de déconnexion. Cette fonction commence par contrôler si le module est déjà connecté (WiFi.status() == WL_CONNECTED), puis, le cas échéant, lance la procédure de connexion avec WiFi.begin(ssid, password). Une boucle avec un timeout de 10 secondes surveille l'avancement de la connexion, avec un affichage progressif dans la console série pour faciliter le diagnostic :

```
void connectToWiFi() {
  if(WiFi.status() == WL_CONNECTED) return;

  WiFi.begin(ssid, password);
  Serial.println("Connexion au WiFi...");

  unsigned long startAttemptTime = millis();
  while(WiFi.status() != WL_CONNECTED && millis() - startAttemptTime < 10000) {
    delay(500);
    Serial.print(".");
  }

  if(WiFi.status() != WL_CONNECTED) {
    Serial.println("\nÉchec de la connexion WiFi");
  } else {
    Serial.println("\nConnecté au WiFi");
  }
}
```

Cette logique est appelée régulièrement dans la boucle principale, ce qui permet au bridge de rester connecté de manière quasi permanente, même en cas de micro-coupures ou de pertes temporaires du réseau Wi-Fi. Cela est indispensable pour un système de supervision temps réel : les capteurs continuent de transmettre, et le bridge doit être capable de relayer les données dès que la connexion revient.

Pour la communication avec l'API, j'ai ensuite choisi une configuration basée sur une adresse IP locale fixe, par exemple :

```
const char* nodeDataUrl  = "http://192.168.1.138:8000/api/lora-data";
const char* bridgeDataUrl = "http://192.168.1.138:8000/api/lora-data";
```

Ce choix permet de s'affranchir des problèmes liés au DHCP ou au DNS local pendant la phase de développement : le bridge sait exactement où joindre l'API Symfony, sans dépendre d'un nom de domaine ou d'une résolution DNS potentiellement instable en environnement de test. Cela simplifie également l'analyse réseau (Wireshark, logs) et facilite la reproductibilité des scénarios de tests IoT. En production, cette logique pourrait évoluer vers un nom de domaine ou un serveur exposé, mais pour le projet, cette IP fixe apporte un contrôle total sur le flux réseau et les échanges entre l'ESP32 et la machine de développement.

Un élément important du dispositif est la mise en place d'un mécanisme de redondance via un bridge secondaire. L'idée est la suivante : si le bridge principal n'est pas en mesure d'envoyer les données à l'API (coupure réseau prolongée, problème côté serveur), il ne doit pas perdre les trames. J'ai donc implémenté une fonction transferToSecondaryBridge(...) qui permet de transférer les messages vers un deuxième bridge via le réseau LoRa Mesh (RHMesh).

Cette fonction :

- construit une trame spéciale de type "TRANSFER:<nodeId>,<message>",
- chiffre cette trame avec la fonction encryptMessage() (AES + Base64),
- tente de l'envoyer au bridge secondaire (adresse mesh 2) avec plusieurs tentatives et délais croissants,
- sauvegarde les données en local en cas de succès pour garder une trace.

Extrait de code :

```
void transferToSecondaryBridge(uint8_t from, const String& message, uint32_t timestamp) {
  Serial.println("Transfert des données vers le bridge secondaire...");

  String transferMessage = "TRANSFER:" + String(from) + "," + message;
  String encryptedTransfer = encryptMessage(transferMessage);

  uint8_t retries = 0;
  bool sent = false;

  while (retries < 3 && !sent) {
    res = manager.sendtoWait((uint8_t*)encryptedTransfer.c_str(),
encryptedTransfer.length(), 2);

    if (res == RH_ROUTER_ERROR_NONE) {
      Serial.println("Transfert réussi vers le bridge secondaire");
```

```
            sent = true;
            saveToSPIFFS(from, message, timestamp);
        } else {
            retries++;
            delay(1000 * (retries + 1));
        }
    }
}
```

Grâce à ce mécanisme, le système n'est plus dépendant d'un seul point de défaillance : si l'API Symfony est indisponible, les données peuvent être relayées vers un autre bridge ou au minimum sauvegardées localement. Cela renforce fortement la résilience globale du dispositif IoT.

En complément de cette redondance réseau, j'ai mis en place un système de logs local basé sur SPIFFS, la mémoire de fichiers intégrée de l'ESP32. À chaque message reçu d'un Node et traité par le bridge, une entrée de log est enregistrée dans un fichier texte (/bridge1_log.txt) avec des informations telles que :

- le numéro d'événement,
- l'identifiant du Node (from),
- le timestamp global synchronisé,
- le contenu du message décrypté.

Pour éviter de saturer la mémoire, j'ai implémenté une rotation automatique des logs : lorsqu'un seuil d'occupation est atteint (par exemple 90 % de l'espace SPIFFS), le firmware supprime les entrées les plus anciennes pour faire de la place aux nouvelles. Les logs peuvent être relus ou vidés via des commandes série comme SHOW_LOGS ou CLEAR_LOGS. Ce système permet d'avoir:

- une traçabilité complète des événements IoT,
- une capacité de diagnostic hors ligne (même si l'API est inaccessible),
- une sécurité opérationnelle : aucune donnée n'est perdue immédiatement en cas de coupure réseau.

L'ensemble de ce travail a été réalisé sur un ESP32 réel, connecté en Wi-Fi au même LAN que la machine exécutant l'API Symfony. J'ai mené de nombreux tests en conditions dégradées : coupure volontaire du Wi-Fi, arrêt du serveur API, lenteurs réseau, pertes de paquets. Ces scénarios m'ont permis de vérifier que :

- le bridge tente de se reconnecter automatiquement au Wi-Fi,
- les erreurs d'envoi à l'API sont gérées proprement,
- le transfert vers le bridge secondaire se déclenche si nécessaire,
- les données sont bien sauvegardées dans SPIFFS même en cas de problème côté réseau.

Pour réaliser ce travail, j'ai utilisé un ensemble de moyens matériels et logiciels. Côté IoT : un ESP32 LoRa (bridge principal), un éventuel second ESP32 en bridge secondaire et des Nodes LoRa jouant le rôle de capteurs (température, humidité, mouvements…). Côté réseau : un Wi-Fi 2.4 GHz local, le routage LoRa Mesh via RHMesh entre bridges et l'API REST Symfony accessible sur le réseau LAN. Au niveau du code, j'ai exploité plusieurs bibliothèques C++ / Arduino : WiFi.h pour la connectivité, HTTPClient.h pour les requêtes HTTP, AES.h pour le chiffrement, RHMesh.h / RH_RF95.h pour la couche radio LoRa, SPIFFS.h pour le stockage persistant et ArduinoJson pour la sérialisation/désérialisation JSON. Pour les tests et le diagnostic, j'ai utilisé le Serial Monitor, des captures réseau (Wireshark), des tests de déconnexion forcée et des scénarios manuels de charge et de panne.

Ce travail a été réalisé en autonomie, avec des échanges techniques ponctuels avec mon encadrant Homeeguard pour :

- définir les priorités réseau (ce qui est critique, ce qui peut être mis en file d'attente),
- valider le comportement du système en cas de perte Wi-Fi ou d'API indisponible,
- faire des choix éclairés entre différentes stratégies : simple stockage local, redondance via un deuxième bridge, ou combinaison des deux.

Le contexte de réalisation est celui de l'entreprise Homeeguard, au sein de l'activité *Développement logiciel & Supervision IoT*, sur la période du 10/03/2025 au 28/10/2025.

Ce travail m'a permis de mieux comprendre les problématiques réseau spécifiques à l'IoT : gestion de la connectivité sur microcontrôleur, latence, pertes de paquets, dépendance à des infrastructures locales (Wi-Fi, API). J'ai pu mettre en place des mécanismes concrets de résilience (reconnexion automatique, redondance LoRa, logs SPIFFS), renforcer la sécurité des communications entre équipements (Wi-Fi + chiffrement + contrôle API) et garantir une continuité de service indispensable pour un système de supervision comme Homeeguard, où chaque donnée de capteur peut être critique pour la surveillance d'un environnement réel.

Exemple n°3 ▸ **Mise en place des logs SPIFFS + rotation automatique sur l'ESP32**

Dans le cadre du système Homeeguard, le bridge ESP32 occupe une position centrale : il reçoit les données chiffrées en provenance des capteurs LoRa, les déchiffre, les analyse puis les transmet à l'API Symfony. Pour garantir la fiabilité, la traçabilité et la résilience de l'ensemble, j'ai conçu et implémenté un système de journalisation complet reposant sur SPIFFS, la mémoire persistante interne de l'ESP32. L'objectif était double : d'une part, ne jamais perdre les données issues des capteurs, même si l'API ou le réseau sont momentanément indisponibles ; d'autre part, disposer d'un outil de diagnostic très précis permettant d'analyser le comportement du bridge (radio, crypto, Wi-Fi, redémarrages, erreurs diverses) en conditions réelles.

J'ai d'abord mis en place et initialisé l'infrastructure SPIFFS au démarrage du bridge. Le code vérifie que la partition SPIFFS est bien montable et, en cas d'échec, tente automatiquement un formatage avant de redémarrer le microcontrôleur. Typiquement, la séquence ressemble à :

```
if (!SPIFFS.begin(FORMAT_SPIFFS_IF_FAILED)) {
   Serial.println("Erreur SPIFFS - Tentative de formatage...");
   if (!SPIFFS.format()) {
      ESP.restart();
   }
}
```

Cette approche garantit que la mémoire de stockage persistante est dans un état sain avant toute opération de log, ce qui est indispensable pour éviter des comportements aléatoires en production ou en phase de tests intensifs.

Une fois SPIFFS opérationnel, j'ai implémenté la fonction de sauvegarde des événements saveToSPIFFS(). À chaque fois que le bridge reçoit une trame LoRa décryptée en provenance d'un node, il écrit une ligne structurée dans un fichier de log (par exemple /bridge1_log.txt). Chaque entrée contient : un numéro d'événement incrémental, l'identifiant du node (FROM), le timestamp global synchronisé (TIME) et le message brut reçu (MSG).
Concrètement, une ligne ressemble à :

```
String record = "EVENT:" + String(eventNumber) + ",";
record += "FROM:" + String(from) + ",";
record += "TIME:" + String(timestamp) + ",";
record += "MSG:" + message + "\n";

file.print(record);
```

Pour maintenir une chronologie parfaite même après redémarrage du bridge, j'ai ajouté une logique de récupération du dernier numéro d'événement via une fonction getLastEventNumber() qui relit le fichier au lancement, trouve le plus grand identifiant

d'événement et reprend la numérotation à partir de là. De cette manière, chaque événement est unique et ordonné, ce qui simplifie énormément l'analyse a posteriori.

Pour éviter que SPIFFS ne se remplisse et ne bloque le système, j'ai mis en place une rotation automatique des logs. Le principe est le suivant : si l'espace utilisé dépasse un certain seuil (par exemple 90 % de la capacité totale), le firmware déclenche une procédure de "nettoyage" des anciennes lignes. Concrètement, le programme lit le fichier, ignore un certain nombre de premières lignes (par exemple 10 % des entrées les plus anciennes), puis réécrit le reste dans le même fichier. Cela peut se traduire par une logique du type :

```
if (SPIFFS.usedBytes() > (SPIFFS.totalBytes() * 0.9)) {
   int linesToRemove = MAX_LOG_ENTRIES * 0.1;
   // Lecture du fichier, suppression des n premières lignes,
   // réécriture du reste.
}
```

Ce mécanisme permet d'obtenir une journalisation "sans fin" : les logs les plus anciens sont supprimés en priorité, mais le bridge ne se retrouve jamais bloqué par manque d'espace. C'est un point crucial sur un microcontrôleur où la capacité mémoire est très limitée.

Afin de rendre ce système exploitable par un développeur ou un technicien, j'ai également ajouté plusieurs commandes série administratives. En connectant l'ESP32 à un PC via USB, il est possible d'envoyer des commandes dans la console série pour interagir avec le système de logs :

- SHOW_LOGS : affiche tout l'historique des événements stockés (utile pour analyser ce qui s'est passé en cas de bug ou de panne réseau),
- CLEAR_LOGS : supprime l'intégralité des logs SPIFFS,
- CHECK_MEMORY : affiche l'état de la mémoire (espace utilisé / total),
- BRIDGE_STATUS : présente un résumé de l'état du bridge (nombre d'erreurs consécutives, statut Wi-Fi, mémoire heap disponible, heure courante, etc.).

Ces outils facilitent énormément le diagnostic sur le terrain sans avoir besoin de flasher un nouveau firmware ou de disposer d'outils complexes : un simple câble USB et un terminal série suffisent pour récupérer les informations nécessaires.

L'ensemble de ces développements a été testé sur un ESP32 réel, avec de vrais capteurs LoRa et dans des conditions réseau variées : coupures fréquentes du Wi-Fi, arrêt volontaire de l'API Symfony, redémarrages forcés du microcontrôleur, envoi massif de trames pour simuler une charge importante. Les tests ont montré que le système de logs :

- continue à enregistrer les événements même lorsque l'API n'est plus joignable,
- survive aux redémarrages tout en conservant l'historique,
- ne sature pas la mémoire grâce à la rotation automatique,
- offre une vue complète du flux IoT et des erreurs rencontrées.

Pour réaliser ce travail, j'ai utilisé un ESP32 LoRa 868 MHz comme bridge principal, des nodes LoRa pour générer des trames, et un PC relié en USB pour la lecture des logs série. Sur le plan logiciel, je me suis appuyé sur les bibliothèques SPIFFS.h et FS.h pour la gestion des fichiers, RHMesh pour la couche radio, AES.h pour le chiffrement des trames, ArduinoJson pour la structuration éventuelle des données, ainsi que l'Arduino IDE / VS Code + PlatformIO pour le développement, la compilation et le déploiement du firmware. Des outils comme Serial Monitor, Serial Plotter et parfois Wireshark ont été utilisés pour inspecter le comportement réseau et la stabilité générale du système.

Ce travail a été réalisé en autonomie, avec une coordination régulière avec mon encadrant Homeeguard pour définir la stratégie de rotation des logs, la quantité d'informations à conserver, les limites mémoire à ne pas dépasser et les informations utiles pour le support technique. Le contexte est celui de l'entreprise Homeeguard, au sein de l'activité *Développement logiciel & Supervision IoT*, sur la période du 10/03/2025 au 28/10/2025.

Au final, cette fonctionnalité de journalisation est devenue un élément clé du système Homeeguard. Elle assure la traçabilité complète du flux IoT, contribue à la sécurité des données (pas de perte silencieuse), améliore la résilience en cas d'incident réseau ou matériel et facilite grandement le débogage et le support. Elle renforce la qualité globale de la solution et apporte un vrai plus en termes de professionnalisme et de maintenabilité du système.

Exemple n°4 ▸ **Mise en production du dashboard client**

Dans le cadre du projet Homeeguard, j'ai pris en charge la mise en production du dashboard client, l'interface web qui permet aux utilisateurs particuliers et professionnels de superviser leurs équipements IoT (température, humidité, alertes, statistiques, cartographie). Même si la cible finale est un déploiement sur serveur Linux, toute cette mise en production a été réalisée dans un environnement local de type "production" sur mon Mac, afin de pouvoir tester le fonctionnement réel avec le bridge ESP32 et les nodes LoRa avant une mise en ligne ultérieure.

J'ai commencé par préparer l'application Symfony pour un usage en production. Côté backend, j'ai nettoyé les dépendances et optimisé l'autoloader via composer install --no-dev --optimize-autoloader, puis j'ai généré un cache optimisé en exécutant les commandes : php bin/console cache:clear --env=prod puis php bin/console cache:warmup --env=prod. En parallèle, j'ai vérifié la configuration PHP (version, extensions activées, limite mémoire) et j'ai compilé les assets front-end (CSS, JS) pour obtenir un dashboard plus léger et performant. J'ai également ajusté les variables d'environnement dans un fichier .env.local avec, entre autres, APP_ENV=prod et APP_DEBUG=0, de manière à reproduire un comportement de production (pas d'affichage de stack trace, erreurs gérées proprement, logs adaptés).

Sur le plan serveur, même si je n'utilisais pas encore un Apache ou un Nginx Linux, j'ai configuré un environnement simulant une production classique : j'ai utilisé le serveur web intégré de Symfony (symfony server:start -d), en m'assurant que le document root pointe bien vers le répertoire public/, et que l'API Symfony tournait également en mode production. L'objectif était d'avoir un environnement cohérent : même code, même configuration que sur un futur serveur Linux, mais hébergé localement. Tous les accès au dashboard passaient donc par cette instance Symfony sécurisée, avec la base de données MySQL locale déjà remplie avec les entités métier (Comptes, Utilisateur, EquipementAchetes, Logs, Metadata, Product).

J'ai ensuite travaillé sur la connexion entre le dashboard, l'API et les équipements IoT. Le contrôleur Symfony récupère les données via différents repositories, par exemple :

```
$equipements = $repoEquipement->findBy(['client' => $user]);
$alertesRecentes = $repoLogs->findRecent($user);
$equipementsOnline = $repoMeta->countOnline($user);
```

Grâce à ces requêtes, le dashboard affiche : le nombre d'équipements, le nombre d'équipements en ligne, les alertes récentes, etc. J'ai vérifié que les données affichées dans les pages accueil.html.twig, alertes.html.twig, cartographie.html.twig, statistiques.html.twig correspondaient bien aux données réellement générées par le bridge ESP32 et les nodes LoRa passant par l'API /api/lora-data. Des tests bout en bout ont été effectués : un node envoie une trame chiffrée → le bridge la déchiffre et la poste à l'API → l'API enregistre les logs et métadonnées → le dashboard les affiche sous forme de listes, cartes et graphiques.

Avant de considérer le dashboard comme “prêt pour la production”, j’ai effectué un ensemble de tests d’interface et de performance fonctionnelle. Sur la page des alertes, j’ai vérifié le bon comportement du tableau DataTables : pagination, recherche, tri (par exemple par date décroissante), capacité à afficher un grand nombre d’alertes sans ralentissement visible. Sur la partie cartographie, j’ai testé le chargement des positions d’équipements et de leurs statuts sur la carte. Sur la page statistiques, j’ai testé le rendu des courbes et des agrégations de données. J’ai également contrôlé le comportement responsive du dashboard (mobile / tablette / desktop) et le fonctionnement des notifications sonores et visuelles gérées par les scripts JavaScript (alert-notification-manager.js, dashboard-manager.js, etc.), afin de m’assurer que l’utilisateur est effectivement alerté lorsqu’un événement critique survient.

En parallèle, j’ai renforcé la sécurité du dashboard en mode production. Seuls les utilisateurs authentifiés peuvent accéder à leurs pages grâce au système de connexion Symfony (sécurité par firewall, login/password, rôles). L’API REST côté backend reste protégée par la clé API (clientId / apiKey) pour les bridges IoT. Tous les formulaires sensibles (connexion, configuration des deltas, etc.) sont protégés par des tokens CSRF, et les routes sont filtrées via la configuration de sécurité de Symfony. Côté front, Twig joue pleinement son rôle d’échappement automatique des variables pour contrer les attaques XSS. L’ensemble garantit qu’un utilisateur ne peut accéder qu’à ses propres données, et qu’aucun script malveillant ne peut être exécuté dans le navigateur à partir des données affichées.

Enfin, j’ai réalisé une série de tests finaux “comme en production” : scénarios complets de connexion / déconnexion, navigation entre toutes les pages (accueil, alertes, statistiques, cartographie, delta, FAQ), modification des deltas de configuration, vérification que les valeurs modifiées sont bien prises en compte par l’API et propagées vers les équipements, simulation d’alertes générées par le bridge ESP32, puis consultation de ces alertes en temps réel sur le dashboard. J’ai testé sous plusieurs navigateurs (Chrome, Firefox) pour m’assurer de la compatibilité et de la stabilité.

Pour réaliser tout cela, j’ai utilisé un environnement local de type production sur Mac, le serveur Symfony**,** PHP 8**,** Composer, le couple Twig + JavaScript/jQuery/DataTables/Bootstrap pour l’interface, l’API Symfony configurée en mode prod, ainsi que l’ESP32 Bridge + nodes LoRaWAN pour les tests en conditions réelles. Le Symfony Profiler a été utilisé dans les phases pré-production (en mode dev), puis désactivé une fois le mode production activé.

Ce travail a été effectué en autonomie, avec des échanges ponctuels avec mon encadrant Homeeguard pour définir les pages prioritaires à finaliser, valider l’ergonomie et l’apparence générale, et vérifier que les données IoT affichées correspondaient bien aux attentes fonctionnelles. Le contexte est celui de l’entreprise Homeeguard, sur le chantier *Développement logiciel & Supervision IoT*, durant la période du 10/03/2025 au 28/10/2025.

Cette mise en production locale du dashboard m’a permis de valider l’ensemble de la chaîne UI + API + IoT avant un déploiement sur un véritable serveur Linux : j’ai pu tester le comportement métier, la sécurité, la performance et l’expérience utilisateur dans des conditions proches du réel. Elle m’a apporté une expérience concrète en déploiement web, en optimisation Symfony,

en configuration d'un environnement de production et en tests finaux multi-couches, tout en livrant une interface stable, sécurisée et prête à être présentée lors de la soutenance.

Exemple n°5 ▸ **Mise en œuvre du plan de tests fonctionnels / IoT / API**

Dans le cadre du projet Homeeguard, j'ai conçu et exécuté un plan de tests complet, couvrant l'intégralité de la chaîne technique : des capteurs LoRa (Nodes), en passant par le Bridge ESP32, puis l'API Symfony, jusqu'au dashboard client. L'objectif était de garantir la fiabilité des communications IoT, la cohérence des données, la robustesse de l'API, la sécurité des échanges et une expérience utilisateur fluide. Tous les tests ont été réalisés en environnement local de type production, sur mon Mac, en interaction réelle avec un ESP32 Bridge et plusieurs Nodes LoRaWAN.

J'ai d'abord structuré le plan de tests en plusieurs volets : tests IoT, tests API, tests base de données, tests dashboard, tests en conditions dégradées et tests de sécurité. Cette approche m'a permis de tester chaque couche indépendamment, puis ensemble en bout en bout. Côté IoT, j'ai validé la transmission cryptée AES depuis les Nodes, la réception LoRa via RHMesh par le Bridge, puis le déchiffrement sur l'ESP32. Les messages radio ont été testés dans différents scénarios : transmission normale, pertes de paquets, trames corrompues et envois massifs pour éprouver les buffers. Lors de ces tests, j'ai observé des traces concrètes sur le moniteur série, comme :

```
[LoRa] Message reçu du Node 1
[AES] Déchiffrement : OK
[DATA] Temp=21.3°C / Hum=47%
```

Ces résultats ont confirmé le bon fonctionnement du chiffrement/déchiffrement et de la communication IoT. J'ai également vérifié la gestion automatique des réessais LoRa, ainsi que la détection de l'absence temporaire d'un Node.

J'ai ensuite testé l'API Symfony en simulant tous les cas possibles via Postman et via le Bridge lui-même : envoi de données valides, JSON mal formé, clientId invalide, absence de clé API, équipement inconnu ou timestamps incorrects. Les résultats ont été conformes aux attentes :
Exemple de test réussi :

```
POST /api/lora-data → 200 OK
{ "status": "success", "message": "Données enregistrées" }
```

Exemple d'échec géré :

```
POST /api/lora-data avec clientId: "mauvaiseCle" → 401 Unauthorized
```

L'API renvoyait systématiquement des erreurs cohérentes, ce qui démontre la solidité de la validation des données et de la sécurité par clé API.
J'ai également vérifié le comportement de Doctrine : création correcte des logs

(EquipementAchetesLogs), association avec le bon équipement, insertion des métadonnées (EquipementAchetesMetadata) et gestion des timestamps. J'ai contrôlé l'intégrité référentielle sur MySQL et l'absence de doublons ou incohérences après de longues sessions de tests IoT.

Concernant le dashboard, j'ai validé le bon affichage des équipements, le calcul des équipements en ligne, la liste des alertes récentes, les statistiques (avec Chart.js), la cartographie (Leaflet.js), les filtres DataTables et la mise à jour des deltas côté client. Les tests incluaient la navigation sur toutes les pages, la modification de paramètres, l'affichage des alertes générées en temps réel par le Bridge et le rendu responsive sur différentes tailles d'écran. Chaque action devait se refléter correctement dans la base et dans l'interface.

J'ai aussi testé des scénarios dégradés pour reproduire des conditions réelles : coupure Wi-Fi au niveau du Bridge, API arrêtée, Node qui disparaît temporairement, trames volontairement corrompues, mémoire SPIFFS saturée. Le Bridge réagissait correctement : réessais successifs, stockage local SPIFFS en cas d'API indisponible, reprise automatique une fois la connexion restaurée, et possibilité de basculer vers un bridge secondaire. Ces tests ont démontré la résilience du système IoT.

L'ensemble des tests a été réalisé avec un ESP32 réel, plusieurs Nodes LoRa, un environnement Symfony configuré en production locale, MySQL, phpMyAdmin, Postman, les outils Arduino IDE / PlatformIO, le Serial Monitor, Wireshark, Chrome DevTools, ainsi que toutes les bibliothèques techniques utilisées dans le projet (AESLib, RHMesh/RH_RF95, SPIFFS, Doctrine, DataTables, jQuery, Bootstrap, Leaflet).

Le travail s'est déroulé principalement en autonomie, avec un suivi ponctuel de mon encadrant pour valider les scénarios critiques, les valeurs des tests IoT et la cohérence du plan de tests. Le contexte professionnel était celui de l'entreprise Homeeguard, sur le chantier *Développement logiciel & Supervision IoT*, entre le 10/03/2025 et le 28/10/2025.

Ce plan de tests m'a permis d'identifier plusieurs faiblesses potentielles (comme la gestion des trames invalides, les coupures réseau, les erreurs JSON), de renforcer la robustesse de l'API, de fiabiliser la chaîne IoT, et d'assurer que le dashboard affiche toujours des données correctes, même dans des conditions extrêmes. C'est aujourd'hui l'un des éléments les plus importants du projet, car il prouve que toute la chaîne — du capteur LoRa au navigateur web — fonctionne correctement, de manière sécurisée, stable et reproductible.

Exemple n°6 ▸ **Vérification de conformité OWASP IoT Top 10**

Dans le cadre du développement du système Homeeguard, j'ai mené un travail spécifique de vérification de la sécurité du système en l'alignant avec les recommandations OWASP IoT Top 10, qui sert de référence pour l'analyse des risques dans les environnements d'objets connectés. L'objectif n'était pas seulement de "cocher des cases", mais réellement d'identifier les failles potentielles, de vérifier que les mécanismes déjà en place allaient dans le bon sens, et de proposer des pistes d'amélioration concrètes, tout en documentant clairement les risques résiduels.

J'ai commencé par cartographier l'ensemble de la surface d'attaque du système :
les Nodes LoRaWAN qui collectent les données (température, humidité, mouvements, états), le Bridge ESP32 qui fait le lien entre le monde radio et le réseau IP, l'API Symfony qui traite, valide et stocke les données, et enfin le dashboard web accessible aux clients. Pour chaque maillon – radio LoRa, Wi-Fi, HTTP, stockage SPIFFS, base MySQL, interface utilisateur – j'ai identifié les vecteurs d'attaque possibles : injections dans les trames, interception réseau, XSS, CSRF, accès physique au matériel, etc.

À partir de cette cartographie, j'ai passé le projet en revue point par point au regard des 10 grands risques OWASP IoT Top 10. Pour chaque item, j'ai vérifié l'état réel du système :

- Sur les mots de passe et identifiants par défaut (I1), la solution Homeeguard ne repose pas sur des identifiants statiques côté device : chaque client dispose d'une API key unique, utilisée via le champ clientId, et aucun mot de passe "usine" n'est exposé sur les équipements.
- Pour les interfaces non sécurisées (I2), j'ai examiné les routes Symfony, la gestion des contrôleurs, l'utilisation des protections CSRF sur les formulaires sensibles, ainsi que la validation systématique des données JSON côté API.
- Concernant les communications non chiffrées (I3), j'ai vérifié que la couche IoT utilisait bien du chiffrement AES entre Node et Bridge, que les trames étaient déchiffrées et vérifiées avant d'être envoyées à l'API, et que l'API ne traite jamais de données brutes LoRa non maîtrisées.
- Pour le manque de mise à jour sécurisée (I4), j'ai constaté qu'aucun mécanisme OTA n'était encore en place : le firmware est mis à jour manuellement. Ce point a été identifié comme un risque résiduel, avec une recommandation future : mise en place d'un canal de mise à jour sécurisé signé.
- Sur l'écosystème global (I5), j'ai vérifié la façon dont sont gérés les logs, le stockage SPIFFS et la base Doctrine/MySQL, en m'assurant que les données critiques ne sont pas exposées et que l'accès aux services est maîtrisé.
- Pour les accès physiques (I6), le bridge ESP32 est destiné à être enfermé dans un boîtier fermé, limitant l'accès direct au port USB ou à la puce, même si, comme pour tout IoT, un accès physique complet reste toujours un vecteur possible à documenter.

- Sur la confidentialité des données (I7), j'ai vérifié que les données sensibles ne sont jamais transmises en clair sur le réseau radio, et que l'API ne retourne pas d'informations techniques inutiles dans ses réponses qui pourraient aider un attaquant.
- Pour les services réseau exposés inutilement (I8), le bridge n'expose aucun port réseau accessible depuis l'extérieur : il initie des connexions HTTP vers l'API, mais ne joue pas le rôle de serveur. L'API Symfony est, dans le cadre du projet, uniquement disponible en local.
- Côté gestion du cycle de vie (I9), j'ai vérifié que le système est stable et que le modèle de données permet des évolutions (nouveaux capteurs, nouveaux types d'alertes). J'ai cependant noté que la gestion longue durée (politiques d'update, rotation de clés, etc.) devra être formalisée pour une vraie production.
- Enfin, pour les attaques logiques (I10), j'ai rapproché le projet des recommandations OWASP API/Web : tests de XSS, validation d'input côté Symfony, protections contre les injections SQL via Doctrine, contrôle strict du clientId, filtrage des formulaires, etc.

À partir de cette analyse, j'ai distingué plusieurs catégories de résultats.
Les points conformes et robustes incluent notamment :

- le chiffrement AES 128 bits entre Node et Bridge,
- l'authentification par API key unique côté API,
- la validation systématique des payloads JSON,
- les protections CSRF sur les formulaires,
- la protection XSS via Twig (auto-escaping) combinée au filtrage backend,
- l'absence de ports ouverts côté ESP32, qui limite fortement l'exposition réseau.

J'ai aussi identifié des axes d'amélioration réalistes :

- l'absence actuelle de mise à jour OTA sécurisée pour le firmware, potentiellement problématique à long terme (firmware obsolète, correctifs de sécurité difficiles à déployer),
- l'utilisation de HTTP en local pour l'API Symfony, acceptable en environnement de développement mais à remplacer par HTTPS (avec certificat) pour un déploiement en production,
- le non-chiffrement des logs SPIFFS, qui n'est pas critique dans le contexte (il faut un accès physique au bridge), mais qui reste à mentionner comme risque résiduel.

Tout ce travail a été structuré dans un rapport de conformité : pour chaque risque OWASP IoT, j'ai rédigé une fiche comprenant la description du risque, l'état dans le projet Homeeguard, les protections mises en place, les risques restants et des recommandations techniques claires (par exemple : ajout futur d'HTTPS, mise en place d'un processus d'update OTA signé, chiffrement des logs si exposition physique jugée sensible).

Pour mener cette analyse, je me suis appuyé sur la documentation officielle OWASP IoT Top 10 et OWASP API Top 10, un audit manuel du code côté IoT (C++/Arduino) et côté API Symfony/PHP, ainsi que des tests concrets d'injection et de comportement : requêtes forgées

dans Postman ou curl, tentatives de XSS dans le dashboard via le navigateur, simulation de trafic réseau avec Wireshark, observation des traces via le Serial Monitor et vérifications directes dans la base MySQL via phpMyAdmin. J'ai également utilisé des outils comme OWASP Threat Dragon pour modéliser les menaces et documenter les flux sensibles.

Le travail a été réalisé principalement en autonomie, avec une revue technique et une validation finale par mon encadrant, ainsi que quelques échanges avec un développeur senior orienté API pour confronter mes conclusions aux bonnes pratiques DevSecOps. Le tout s'est déroulé dans le contexte de l'entreprise Homeeguard, au sein du chantier *Développement logiciel & Supervision IoT*, entre le 10/03/2025 et le 28/10/2025.

Cette démarche de mise en conformité avec l'OWASP a permis au projet de gagner en robustesse, de corriger ou d'encadrer plusieurs risques significatifs, de renforcer les protections CSRF/XSS/validation API, et d'aboutir à une architecture où les données sont protégées de bout en bout : chiffrement AES sur la couche IoT, authentification par API key côté backend, validation et filtrage strict avant l'enregistrement en base. C'est un volet majeur de la préparation à la mise en production, et un véritable atout pour la qualité et la crédibilité de la solution Homeeguard.

Exemple n°7 ▸ **Documentation technique et maintenance du système**

Dans le cadre du projet Homeeguard, j'ai rédigé et structuré une documentation technique complète couvrant l'ensemble du système IoT : capteurs LoRa, bridge ESP32, API Symfony, base de données et dashboard utilisateur. Cette documentation avait plusieurs objectifs : faciliter la maintenance future, permettre à un autre développeur de reprendre le projet sans repartir de zéro, assurer une cohérence avec les bonnes pratiques professionnelles et préparer un futur déploiement dans un contexte plus proche de la production.

J'ai d'abord travaillé sur une documentation d'architecture globale. J'y ai décrit, sous forme de schémas et de textes, le cheminement complet des données : des Nodes LoRa jusqu'au dashboard web, en passant par le bridge ESP32 et l'API Symfony. J'ai formalisé un diagramme d'architecture système (Nodes → Bridges → API → Dashboard), détaillé les flux de données, expliqué le fonctionnement du chiffrement AES et de la transmission LoRa → Wi-Fi → HTTP, et intégré des schémas UML (diagrammes de cas d'utilisation, de séquence et de classes) pour rendre le fonctionnement plus lisible pour un développeur ou un décideur technique.

En parallèle, j'ai rédigé un guide d'installation et de mise en route pour l'environnement technique, côté IoT comme côté backend. Pour la partie embarquée, j'ai listé les étapes pour compiler et flasher les firmwares Node et Bridge (dépendances Arduino, bibliothèques AES, RHMesh, SPIFFS, WiFi, ArduinoJson, configuration des paramètres radio, des serial numbers et des clés partagées). Pour la partie API Symfony, j'ai détaillé la procédure d'installation locale : utilisation de Composer, configuration du fichier .env ou .env.local, paramétrage de la connexion MySQL, exécution des migrations Doctrine, lancement du serveur Symfony. Ce guide est pensé pour qu'un développeur puisse reproduire mon environnement de travail macOS et retrouver rapidement un système fonctionnel.

J'ai également produit une documentation de la base de données, en décrivant les tables principales (Comptes, Utilisateur, EquipementAchetes, EquipementAchetesLogs, EquipementAchetesMetadata, Product), leurs rôles, leurs principaux champs, ainsi que leurs relations (OneToMany, ManyToOne, jointures). J'y ai intégré un schéma physique (MPD) annoté, afin de rendre visibles les liens entre les comptes, les utilisateurs, les équipements, les logs et les métadonnées de configuration. Ce travail permet de comprendre rapidement comment les données IoT sont stockées, historisées et exploitées par le dashboard.

Une autre partie importante de la documentation concerne la maintenance du bridge et des nodes. J'y ai formalisé des procédures précises : comment effectuer un reset matériel, comment mettre à jour le firmware, comment lire et interpréter les logs SPIFFS, comment fonctionne la rotation automatique des logs, et comment analyser les messages affichés en série (par exemple les états "WARN", "ERROR" ou les erreurs réseau récurrentes). J'y ai aussi décrit les mécanismes internes du bridge : gestion du timestamp global synchronisé avec l'API, redondance entre bridge principal et éventuel bridge secondaire, incrément des erreurs consécutives et conditions de redémarrage automatique.

J'ai ensuite documenté le dashboard utilisateur côté front-end. Cette partie décrit le rôle des principaux modules JavaScript du projet :

- alert-notification-manager.js pour la gestion des notifications d'alertes,
- dashboard-manager.js pour la mise à jour des compteurs et de la page d'accueil,
- delta-manager.js pour la gestion des formulaires de seuils/deltas,
- map-manager.js pour l'affichage des équipements sur la carte.

J'y ai expliqué quelles routes Symfony sont consommées par ces scripts, comment les données sont récupérées et injectées dans les templates Twig, comment fonctionne le filtrage des alertes via DataTables, et comment le client peut modifier les deltas de configuration depuis l'interface (et comment ces modifications sont ensuite propagées vers la base et, au besoin, vers les capteurs).

Enfin, j'ai centralisé l'ensemble de cette documentation dans un seul ensemble cohérent : un dossier structuré comprenant l'architecture, le fonctionnement technique détaillé, les procédures de déploiement, le plan de tests, les schémas UML, les spécifications fonctionnelles et techniques, ainsi qu'un mini guide utilisateur. Ce corpus est conçu pour être fourni à un nouveau développeur, à un examinateur ou à une équipe de maintenance.

Pour produire tout cela, je me suis appuyé sur plusieurs outils : un éditeur de texte (Markdown/Word) pour la rédaction, la console Symfony et Composer pour vérifier les commandes et les étapes d'installation, PhpMyAdmin pour visualiser et documenter le schéma de base de données, Arduino IDE et les bibliothèques ESP32 pour la partie IoT, ainsi que différents logiciels de schématisation (Draw.io pour les UML, éventuellement Fritzing pour le schéma électronique, Excalidraw pour les flux). J'ai illustré la documentation avec des captures d'écran du dashboard (pages d'accueil, alertes, statistiques, cartographie, configuration des deltas) afin de rendre l'ensemble plus pédagogique.

Ce travail a été réalisé principalement en autonomie, avec une relecture et une validation technique de la part de mon encadrant Homeeguard, ainsi que quelques échanges avec un développeur backend pour vérifier la cohérence des entités Doctrine et du modèle de données. Il s'inscrit dans le contexte de l'entreprise Homeeguard, dans le service de développement logiciel et supervision IoT, sur la période du 10/03/2025 au 28/10/2025.

Cette documentation technique est aujourd'hui un élément clé du projet : elle garantit la continuité du système, améliore la qualité et la sécurité globale de la solution IoT, rend le projet transférable à une autre personne et prépare l'avenir, que ce soit pour un déploiement sur serveur Linux, une montée en charge, ou de nouvelles évolutions fonctionnelles. Elle marque aussi une vraie démarche professionnelle, où le développement n'est pas seulement du code, mais s'accompagne d'une vision de maintenance et de pérennité à long terme.

Titres, diplômes, CQP, attestations de formation

Intitulé	Autorité ou organisme	Date
Bac Professionnel Système numérique option Réseaux Informatique et Système Communicant	Éducation Nationale	2024
BTS SIO Option SLAM (Solution Logiciel et Application Métier)	Éducation Nationale	2025

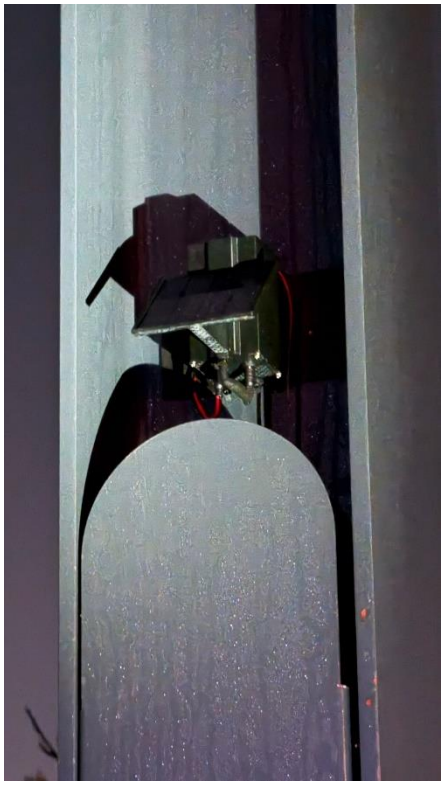
Déclaration sur l'honneur

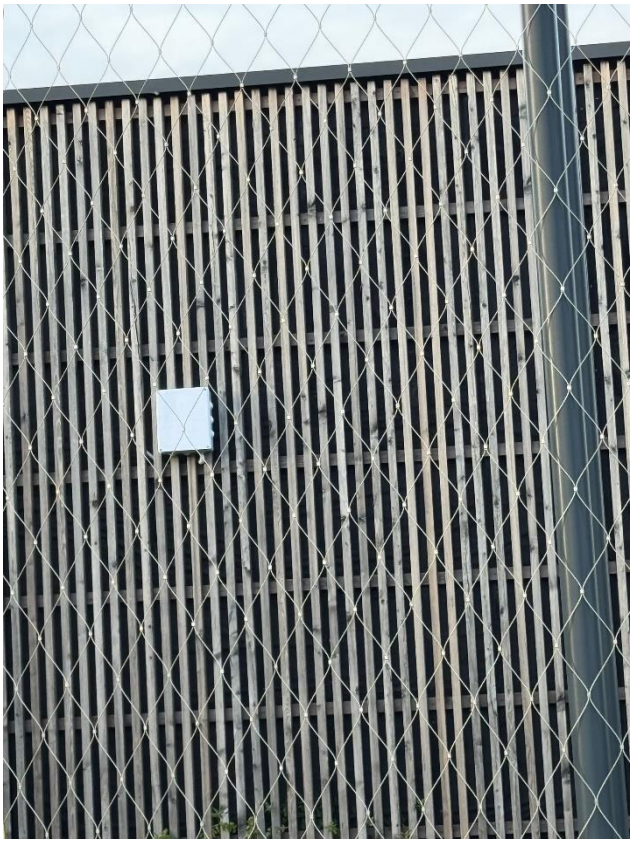
Je soussigné Baptiste LUCBERT, déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis l'auteur(e) des réalisations jointes.

Fait à Herblay-Sur-Seine le 19/11/2025 pour faire valoir ce que de droit.

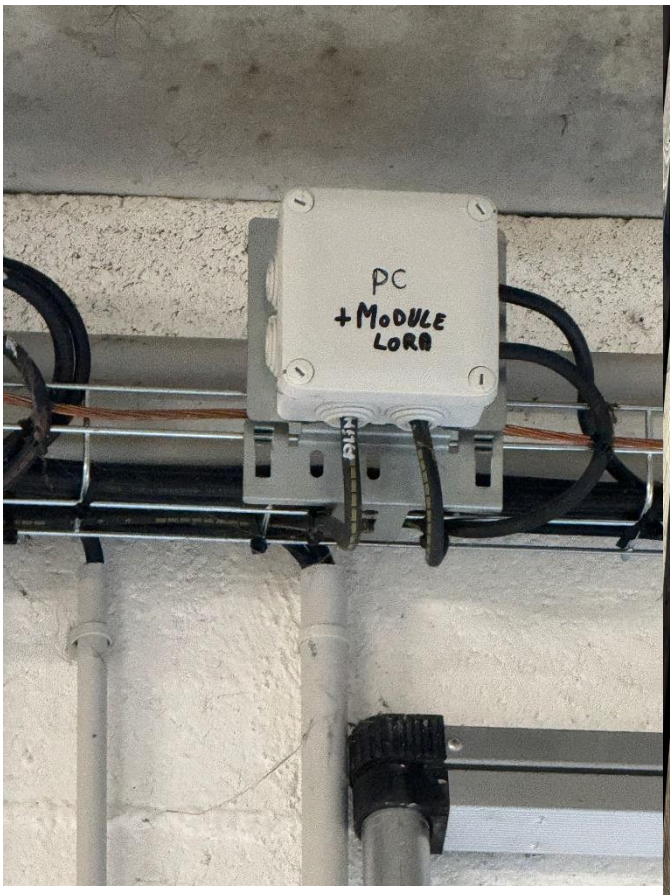
Signature : Baptiste Lucbert

ANNEXES

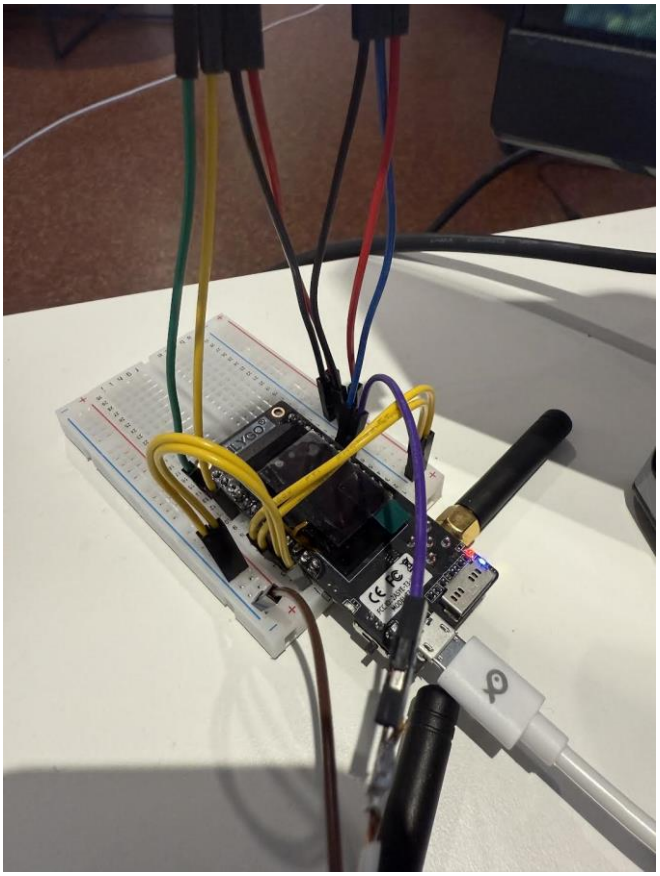


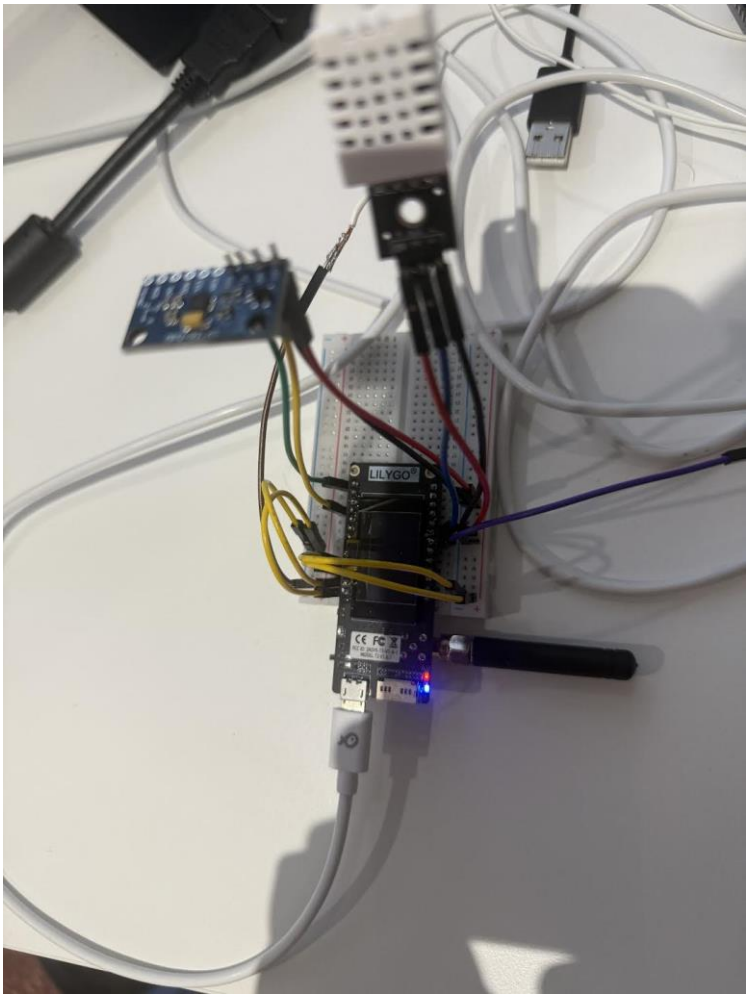



PC
+MODULE
LORA
IP55
legrand


PC
+MODULE
LORA


FLUKE 175 TRUE RMS MULTIMETER
237.5
V AC
Auto Range
AutoHOLD
HOLD
MINMAX
RANGE
OFF
400 mA
10A
FUSED
COM




LILYGO