Propositions d'Intégration Communication Temps Réel et ESP32

Projet FAJMA

Table des Matières

Résumé Exécutif

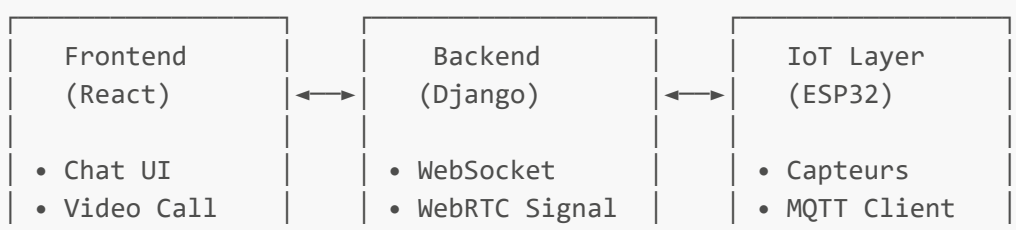
Ce document présente une architecture complète pour l'intégration de la communication temps réel (chat, vidéo) et des dispositifs ESP32 dans le système de télémédecine FAJMA. L'objectif est de créer une plateforme unifiée permettant aux médecins de communiquer avec leurs patients tout en surveillant leurs données biométriques en temps réel.

Objectifs Principaux

- **Communication bidirectionnelle** : Chat texte et appels vidéo sécurisés
- **Surveillance IoT** : Intégration des capteurs ESP32 pour le monitoring biométrique
- **Temps réel** : Transmission instantanée des données critiques
- **Sécurité** : Chiffrement bout-en-bout et conformité RGPD
- **Scalabilité** : Architecture microservices pour la montée en charge

Architecture Proposée

Vue d'Ensemble



```
│ • IoT Dashboard │    │ • MQTT Broker    │    │ • WiFi/BLE      │
└─────────────────┘    └──────────────────┘    └─────────────────┘
```

Composants Principaux

1. Couche Frontend (React)

- **Interface Chat** : Composants React pour messagerie instantanée
- **Appels Vidéo** : Intégration WebRTC pour communication audio/vidéo
- **Dashboard IoT** : Visualisation temps réel des données biométriques
- **Notifications** : Alertes critiques et notifications push

2. Couche Backend (Django)

- **API REST** : Endpoints pour gestion des utilisateurs et consultations
- **WebSocket Server** : Communication bidirectionnelle temps réel
- **MQTT Broker** : Réception et traitement des données IoT
- **Signaling Server** : Coordination des connexions WebRTC

3. Couche IoT (ESP32)

- **Capteurs Biométriques** : Fréquence cardiaque, SpO2, tension artérielle
- **Communication** : WiFi, MQTT, Bluetooth LE
- **Sécurité** : Chiffrement matériel et authentification par certificat

Solutions de Communication Temps Réel

1. Chat Texte (WebSocket)

Implémentation Backend (Django Channels)

```
# consumers.py
class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.consultation_id = self.scope['url_route']['kwargs']
['consultation_id']
        self.room_group_name = f'chat_{self.consultation_id}'

        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )
        await self.accept()

    async def receive(self, text_data):
        data = json.loads(text_data)
        message = data['message']
        user_id = self.scope['user'].id
```

```
        # Sauvegarder le message
        await self.save_message(user_id, message)

        # Diffuser à tous les participants
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
                'user_id': user_id,
                'timestamp': timezone.now().isoformat()
            }
        )
```

Implémentation Frontend (React)

```
// ChatComponent.jsx
const ChatComponent = ({ consultationId }) => {
    const [socket, setSocket] = useState(null);
    const [messages, setMessages] = useState([]);
    const [newMessage, setNewMessage] = useState('');

    useEffect(() => {
        const ws = new WebSocket(
            `ws://localhost:8000/ws/chat/${consultationId}/`
        );

        ws.onmessage = (event) => {
            const data = JSON.parse(event.data);
            setMessages(prev => [...prev, data]);
        };

        setSocket(ws);
        return () => ws.close();
    }, [consultationId]);

    const sendMessage = () => {
        if (socket && newMessage.trim()) {
            socket.send(JSON.stringify({
                'message': newMessage
            }));
            setNewMessage('');
        }
    };
};
```

2. Appels Vidéo (WebRTC)

Signaling Server (Django)

```
# webrtc_consumers.py
class WebRTCSignalingConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['room_name']
        self.room_group_name = f'webrtc_{self.room_name}'

        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )
        await self.accept()

    async def receive(self, text_data):
        data = json.loads(text_data)
        message_type = data['type']

        # Relayer les messages de signaling
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'webrtc_message',
                'data': data
            }
        )
```

Client WebRTC (React)

```
// VideoCallComponent.jsx
const VideoCallComponent = ({ roomId }) => {
    const [localStream, setLocalStream] = useState(null);
    const [remoteStream, setRemoteStream] = useState(null);
    const [peerConnection, setPeerConnection] = useState(null);

    const initializeWebRTC = async () => {
        const pc = new RTCPeerConnection({
            iceServers: [
                { urls: 'stun:stun.l.google.com:19302' }
            ]
        });

        // Obtenir le flux local
        const stream = await navigator.mediaDevices.getUserMedia({
            video: true,
            audio: true
        });

        setLocalStream(stream);
        stream.getTracks().forEach(track => {
            pc.addTrack(track, stream);
        });
```

```
        // Gérer le flux distant
        pc.ontrack = (event) => {
            setRemoteStream(event.streams[0]);
        };

        setPeerConnection(pc);
    };
};
```

Intégration ESP32 et IoT

1. Configuration ESP32

Code Arduino pour ESP32

```
// esp32_health_monitor.ino
#include <WiFi.h>
#include <PubSubClient.h>
#include <ArduinoJson.h>
#include <WiFiClientSecure.h>

// Configuration WiFi et MQTT
const char* ssid = "FAJMA_NETWORK";
const char* password = "secure_password";
const char* mqtt_server = "mqtt.fajma.com";
const int mqtt_port = 8883;

// Certificats SSL/TLS
const char* ca_cert = "-----BEGIN CERTIFICATE-----\n...";
const char* client_cert = "-----BEGIN CERTIFICATE-----\n...";
const char* client_key = "-----BEGIN PRIVATE KEY-----\n...";

WiFiClientSecure espClient;
PubSubClient client(espClient);

// Capteurs
const int HEART_RATE_PIN = A0;
const int SPO2_PIN = A1;
const int TEMP_PIN = A2;

void setup() {
    Serial.begin(115200);

    // Configuration SSL
    espClient.setCACert(ca_cert);
    espClient.setCertificate(client_cert);
    espClient.setPrivateKey(client_key);

    // Connexion WiFi
    WiFi.begin(ssid, password);
```

```
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }

    // Configuration MQTT
    client.setServer(mqtt_server, mqtt_port);
    client.setCallback(callback);
}

void loop() {
    if (!client.connected()) {
        reconnect();
    }
    client.loop();

    // Lecture des capteurs
    float heartRate = readHeartRate();
    float spo2 = readSpO2();
    float temperature = readTemperature();

    // Création du payload JSON
    StaticJsonDocument<200> doc;
    doc["device_id"] = WiFi.macAddress();
    doc["timestamp"] = millis();
    doc["heart_rate"] = heartRate;
    doc["spo2"] = spo2;
    doc["temperature"] = temperature;

    String payload;
    serializeJson(doc, payload);

    // Publication MQTT
    String topic = "fajma/sensors/" + WiFi.macAddress();
    client.publish(topic.c_str(), payload.c_str());

    delay(5000); // Envoi toutes les 5 secondes
}
```

2. Traitement Backend des Données IoT

MQTT Consumer (Django)

```
# mqtt_handler.py
import paho.mqtt.client as mqtt
import json
from channels.layers import get_channel_layer
from asgiref.sync import async_to_sync
from .models import CapteurIoT, DonneesBiometriques

class MQTTHandler:
```

```
    def __init__(self):
        self.client = mqtt.Client()
        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.channel_layer = get_channel_layer()

    def on_connect(self, client, userdata, flags, rc):
        print(f"Connecté au broker MQTT avec le code {rc}")
        client.subscribe("fajma/sensors/+")

    def on_message(self, client, userdata, msg):
        try:
            # Décoder le message
            payload = json.loads(msg.payload.decode())
            device_id = payload['device_id']

            # Sauvegarder en base
            capteur = CapteurIoT.objects.get(identifiant_unique=device_id)
            donnees = DonneesBiometriques.objects.create(
                capteur=capteur,
                frequence_cardiaque=payload['heart_rate'],
                saturation_oxygene=payload['spo2'],
                temperature=payload['temperature']
            )

            # Vérifier les seuils critiques
            if self.is_critical_alert(payload):
                self.send_critical_alert(capteur, payload)

            # Diffuser via WebSocket
            async_to_sync(self.channel_layer.group_send)(
                f"patient_{capteur.patient.id}",
                {
                    'type': 'iot_data',
                    'data': payload
                }
            )

        except Exception as e:
            print(f"Erreur traitement MQTT: {e}")

    def is_critical_alert(self, data):
        return (
            data['heart_rate'] > 120 or data['heart_rate'] < 50 or
            data['spo2'] < 90 or
            data['temperature'] > 38.5
        )
```

Sécurité et Conformité

1. Chiffrement des Communications

Configuration TLS/SSL

```
# settings.py
SECURE_SSL_REDIRECT = True
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True

# Configuration WebSocket sécurisé
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            "hosts": [("redis://localhost:6379/0")],
            "symmetric_encryption_keys": [SECRET_KEY],
        },
    },
}
```

2. Authentification et Autorisation

JWT pour API et WebSocket

```
# authentication.py
from channels.middleware import BaseMiddleware
from django.contrib.auth.models import AnonymousUser
from rest_framework_simplejwt.tokens import AccessToken

class JWTAuthMiddleware(BaseMiddleware):
    async def __call__(self, scope, receive, send):
        token = self.get_token_from_scope(scope)
        if token:
            try:
                access_token = AccessToken(token)
                user = await self.get_user(access_token['user_id'])
                scope['user'] = user
            except Exception:
                scope['user'] = AnonymousUser()
        else:
            scope['user'] = AnonymousUser()

        return await super().__call__(scope, receive, send)
```

3. Chiffrement des Données IoT

Modèle de Données Sécurisé

```python
# models.py
from cryptography.fernet import Fernet
from django.conf import settings

class SecureCapteurIoT(models.Model):
    identifiant_unique = models.CharField(max_length=100, unique=True)
    cle_chiffrement = models.BinaryField()  # Clé AES-256
    certificat_x509 = models.TextField()    # Certificat pour authentification

    def encrypt_data(self, data):
        f = Fernet(self.cle_chiffrement)
        return f.encrypt(json.dumps(data).encode())

    def decrypt_data(self, encrypted_data):
        f = Fernet(self.cle_chiffrement)
        return json.loads(f.decrypt(encrypted_data).decode())
```

Prérequis Techniques

1. Infrastructure Serveur

Spécifications Minimales

- **CPU** : 4 cœurs (8 recommandés)
- **RAM** : 8 GB (16 GB recommandés)
- **Stockage** : 100 GB SSD
- **Bande passante** : 100 Mbps symétrique

Services Requis

- **Base de données** : PostgreSQL 14+
- **Cache/Message Broker** : Redis 7+
- **Reverse Proxy** : Nginx avec SSL
- **MQTT Broker** : Mosquitto avec TLS

2. Dépendances Backend (requirements.txt)

```
# Core Django
Django==4.2.7
djangorestframework==3.14.0
djangorestframework-simplejwt==5.3.0
django-cors-headers==4.3.1

# Communication Temps Réel
channels==4.0.0
channels-redis==4.1.0
django-channels-presence==1.0.1
aiortc==1.6.0
websockets==12.0
```

```
# IoT et MQTT
paho-mqtt==1.6.1
mqtt-client==1.0.0
pyserial==3.5
bleak==0.21.1

# Sécurité IoT
cryptography==41.0.8
PyJWT==2.8.0
certifi==2023.11.17

# Base de données et cache
psycopg2-binary==2.9.9
redis==5.0.1
django-redis==5.4.0

# Traitement données temps réel
numpy==1.24.4
pandas==2.0.3
scipy==1.11.4

# Monitoring et logging
django-health-check==3.17.0
django-prometheus==2.3.1
sentry-sdk==1.38.0

# Tests
pytest==7.4.3
pytest-django==4.7.0
pytest-asyncio==0.21.1
factory-boy==3.3.0
```

3. Dépendances Frontend (package.json)

```
{
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "@mui/material": "^5.14.0",
    "@mui/icons-material": "^5.14.0",
    "socket.io-client": "^4.7.0",
    "simple-peer": "^9.11.1",
    "react-router-dom": "^6.8.0",
    "axios": "^1.6.0",
    "recharts": "^2.8.0",
    "react-notifications-component": "^4.0.1"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.0.0",
    "vite": "^4.4.0",
```

```
    "eslint": "^8.45.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/jest-dom": "^5.16.4"
  }
}
```

4. Configuration ESP32

Bibliothèques Arduino

- **WiFi** : ESP32 WiFi Library
- **MQTT** : PubSubClient 2.8+
- **JSON** : ArduinoJson 6.21+
- **Crypto** : ESP32 Crypto Library
- **Capteurs** : MAX30102 (SpO2), DS18B20 (Température)

Spécifications Matérielles

- **Microcontrôleur** : ESP32-WROOM-32
- **Mémoire** : 4 MB Flash, 520 KB RAM
- **Connectivité** : WiFi 802.11 b/g/n, Bluetooth 4.2
- **Alimentation** : 3.3V, consommation < 100mA

Plan d'Implémentation

Phase 1 : Infrastructure de Base (2-3 semaines)

1. **Configuration serveur** : Installation PostgreSQL, Redis, Nginx
2. **Setup Django Channels** : Configuration WebSocket
3. **MQTT Broker** : Installation et configuration Mosquitto
4. **Certificats SSL** : Génération et déploiement

Phase 2 : Communication Temps Réel (3-4 semaines)

1. **Chat WebSocket** : Implémentation backend et frontend
2. **WebRTC Signaling** : Serveur de signalisation
3. **Interface utilisateur** : Composants React pour chat et vidéo
4. **Tests d'intégration** : Validation des fonctionnalités

Phase 3 : Intégration IoT (4-5 semaines)

1. **Développement ESP32** : Code capteurs et communication MQTT
2. **Backend IoT** : Traitement des données biométriques
3. **Dashboard temps réel** : Visualisation des données
4. **Alertes critiques** : Système de notifications

Phase 4 : Sécurité et Optimisation (2-3 semaines)

1. **Chiffrement bout-en-bout** : Implémentation complète

2. **Tests de sécurité** : Audit et pentesting
3. **Optimisation performances** : Cache et compression
4. **Documentation** : Guide utilisateur et technique

Recommandations

1. Sécurité

- **Chiffrement obligatoire** : TLS 1.3 pour toutes les communications
- **Authentification forte** : JWT avec refresh tokens
- **Audit trail** : Logging complet des accès et actions
- **Conformité RGPD** : Anonymisation et droit à l'oubli

2. Performance

- **CDN** : Distribution de contenu pour les assets statiques
- **Load balancing** : Répartition de charge pour la scalabilité
- **Monitoring** : Surveillance proactive avec Prometheus/Grafana
- **Cache intelligent** : Redis pour les données fréquemment accédées

3. Maintenance

- **CI/CD** : Pipeline automatisé de déploiement
- **Tests automatisés** : Couverture > 80%
- **Backup automatique** : Sauvegarde quotidienne des données
- **Documentation** : Maintien à jour de la documentation technique

4. Évolutivité

- **Architecture microservices** : Séparation des responsabilités
- **API versioning** : Gestion des versions d'API
- **Containerisation** : Docker pour le déploiement
- **Orchestration** : Kubernetes pour la production

Version : 1.0
Auteur : FENKU-IT
Statut : Proposition Technique