Below is a step-by-step plan to develop the **backend** first. We’ll use Node.js/Express and MongoDB (with Mongoose or a similar ODM). The plan integrates **Agile** best practices and prepares a solid foundation for future real-time and AI integrations.

## ****1. Project Initialization and Basic Structure****

**Initialize the Project**

* 1. Create a new folder for your backend, e.g., firefighter-backend.
  2. Initialize a Node.js project with:

npm init -y

* 1. Install **Express** and **Mongoose** (or Prisma if you prefer), plus any other essential libraries:

npm install express mongoose cors dotenv

**Set Up Folder Structure**

* 1. A typical structure might look like this:

firefighter-backend/

├── src/

│ ├── config/

│ │ └── db.js

│ ├── controllers/

│ ├── models/

│ ├── routes/

│ ├── middleware/

│ ├── utils/

│ └── app.js

├── tests/

├── .env

├── package.json

└── README.md

* 1. app.js: Entry point where you initialize Express, middleware, routes, etc.
  2. config/db.js: MongoDB connection logic.
  3. models/: Mongoose schemas (User, Vehicle, Intervention, etc.).
  4. controllers/: Controller functions that implement business logic.
  5. routes/: Route definitions (REST endpoints).
  6. middleware/: Authentication, authorization, error-handling middlewares.
  7. utils/: Helper functions, such as date/time utilities, logging, etc.
  8. tests/: Unit and integration tests.

**Set Up Environment Variables**

* 1. Create a .env file to store secrets (like DB\_URI, JWT\_SECRET, etc.).
  2. Add .env to your .gitignore to avoid committing secrets to version control.

**Basic Server Start**

* 1. In app.js, set up a minimal Express server:

const express = require('express');

const cors = require('cors');

require('dotenv').config();

const app = express();

app.use(cors());

app.use(express.json());

// Placeholder route

app.get('/', (req, res) => {

res.send('Firefighter backend API is running');

});

module.exports = app;

* 1. In a separate file (e.g., server.js):

const app = require('./src/app');

const PORT = process.env.PORT || 4000;

app.listen(PORT, () => {

console.log(`Server is running on port ${PORT}`);

});

## ****2. Database Connection****

**Create** config/db.js

const mongoose = require('mongoose');

require('dotenv').config();

const connectDB = async () => {

try {

await mongoose.connect(process.env.DB\_URI, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

console.log('MongoDB connected');

} catch (err) {

console.error('Error connecting to MongoDB:', err);

process.exit(1);

}

};

module.exports = connectDB;

**Initialize in** server.js

const connectDB = require('./src/config/db');

// Connect to DB before starting server

connectDB();

## ****3. Define Mongoose Models****

Based on the previously outlined entities, create Mongoose schemas in models/:

### ****3.1 User Model****

// models/User.js

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema(

{

firstName: { type: String, required: true },

lastName: { type: String, required: true },

email: { type: String, unique: true, required: true },

password: { type: String, required: true },

role: {

type: String,

enum: [

'firefighter',

'coordinator',

'logistic\_officer',

// add other roles as needed

],

default: 'firefighter',

},

assignedRegion: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Region',

},

location: {

// e.g., { type: 'Point', coordinates: [lon, lat] }

type: { type: String, default: 'Point' },

coordinates: [Number],

},

status: {

type: String,

enum: ['active', 'inactive', 'on-duty'],

default: 'inactive',

},

},

{ timestamps: true }

);

const User = mongoose.model('User', userSchema);

module.exports = User;

### ****3.2 Region Model****

// models/Region.js

const mongoose = require('mongoose');

const regionSchema = new mongoose.Schema(

{

name: { type: String, required: true },

coordinates: {

// Could store a polygon or array of lat/lon points

type: Array,

default: [],

},

},

{ timestamps: true }

);

const Region = mongoose.model('Region', regionSchema);

module.exports = Region;

### ****3.3 Vehicle Model****

// models/Vehicle.js

const mongoose = require('mongoose');

const vehicleSchema = new mongoose.Schema(

{

type: {

type: String,

enum: ['ambulance', 'fire\_truck', 'other'],

required: true,

},

status: {

type: String,

enum: ['available', 'in\_use', 'maintenance'],

default: 'available',

},

location: {

type: { type: String, default: 'Point' },

coordinates: [Number],

},

assignedRegion: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Region',

},

},

{ timestamps: true }

);

const Vehicle = mongoose.model('Vehicle', vehicleSchema);

module.exports = Vehicle;

### ****3.4 Intervention Model****

// models/Intervention.js

const mongoose = require('mongoose');

const interventionSchema = new mongoose.Schema(

{

location: {

// Could be GPS or a string address

type: String,

required: true,

},

status: {

type: String,

enum: ['en\_route', 'on\_site', 'finished'],

default: 'en\_route',

},

incidentType: {

type: String,

enum: ['fire', 'accident', 'medical', 'other'],

required: true,

},

startTime: { type: Date, default: Date.now },

endTime: { type: Date },

assignedResources: [

{

resourceId: { type: mongoose.Schema.Types.ObjectId },

resourceType: { type: String, enum: ['User', 'Vehicle'] },

},

],

},

{ timestamps: true }

);

const Intervention = mongoose.model('Intervention', interventionSchema);

module.exports = Intervention;

### ****3.5 Transcription Model****

// models/Transcription.js

const mongoose = require('mongoose');

const transcriptionSchema = new mongoose.Schema(

{

interventionId: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Intervention',

required: true,

},

audioUrl: { type: String },

transcriptText: { type: String },

language: { type: String, default: 'fr' },

},

{ timestamps: true }

);

const Transcription = mongoose.model('Transcription', transcriptionSchema);

module.exports = Transcription;

### ****3.6 ResourceAlert Model****

// models/ResourceAlert.js

const mongoose = require('mongoose');

const resourceAlertSchema = new mongoose.Schema(

{

resourceType: { type: String, required: true }, // e.g. 'fuel', 'water'

level: { type: Number },

alertThreshold: { type: Number, default: 20 }, // example threshold

triggeredAt: { type: Date, default: Date.now },

regionId: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Region',

required: true,

},

},

{ timestamps: true }

);

const ResourceAlert = mongoose.model('ResourceAlert', resourceAlertSchema);

module.exports = ResourceAlert;

### ****3.7 PredictiveAlert Model****

// models/PredictiveAlert.js

const mongoose = require('mongoose');

const predictiveAlertSchema = new mongoose.Schema(

{

interventionId: {

type: mongoose.Schema.Types.ObjectId,

ref: 'Intervention',

required: true,

},

predictedResourceNeeds: {

type: Object, // e.g. { vehicles: 2, staff: 5 }

default: {},

},

},

{ timestamps: true }

);

const PredictiveAlert = mongoose.model('PredictiveAlert', predictiveAlertSchema);

module.exports = PredictiveAlert;

## ****4. Authentication & Role-Based Access****

**Install Additional Dependencies**

npm install bcrypt jsonwebtoken

**Create Authentication Helpers**

* 1. **Password hashing** using **bcrypt**.
  2. **JWT token** generation and validation.

**Middleware for Auth & Role Checking**

* 1. In middleware/auth.js:

const jwt = require('jsonwebtoken');

const User = require('../models/User');

exports.authenticate = async (req, res, next) => {

try {

const token = req.headers.authorization?.split(' ')[1];

if (!token) {

return res.status(401).json({ message: 'No token provided' });

}

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

req.user = await User.findById(decoded.userId);

next();

} catch (error) {

return res.status(401).json({ message: 'Invalid token' });

}

};

exports.authorizeRoles = (roles) => {

return (req, res, next) => {

if (!roles.includes(req.user.role)) {

return res.status(403).json({ message: 'Access denied' });

}

next();

};

};

**Auth Routes**

* 1. In controllers/authController.js, define **register** and **login** endpoints:

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

const User = require('../models/User');

exports.register = async (req, res) => {

try {

const { email, password, firstName, lastName, role } = req.body;

const hashedPassword = await bcrypt.hash(password, 10);

const newUser = await User.create({

email,

password: hashedPassword,

firstName,

lastName,

role

});

return res.status(201).json(newUser);

} catch (error) {

return res.status(400).json({ message: error.message });

}

};

exports.login = async (req, res) => {

try {

const { email, password } = req.body;

const user = await User.findOne({ email });

if (!user) return res.status(404).json({ message: 'User not found' });

const isValid = await bcrypt.compare(password, user.password);

if (!isValid) return res.status(401).json({ message: 'Invalid credentials' });

const token = jwt.sign({ userId: user.\_id }, process.env.JWT\_SECRET, { expiresIn: '8h' });

return res.json({ token, user });

} catch (error) {

return res.status(500).json({ message: error.message });

}

};

* 1. Create routes/authRoutes.js:

const express = require('express');

const router = express.Router();

const { register, login } = require('../controllers/authController');

router.post('/register', register);

router.post('/login', login);

module.exports = router;

* 1. In app.js, mount these routes:

const authRoutes = require('./routes/authRoutes');

app.use('/api/auth', authRoutes);

## ****5. CRUD Endpoints (Core Resources)****

Create **controllers** and **routes** for each of the core models: User, Vehicle, Intervention, etc.

### ****5.1 Example: User Controller & Routes****

**Controller (**controllers/userController.js**)**:

const User = require('../models/User');

exports.getAllUsers = async (req, res) => {

try {

const users = await User.find();

res.json(users);

} catch (error) {

res.status(500).json({ message: error.message });

}

};

exports.getUserById = async (req, res) => {

try {

const user = await User.findById(req.params.id);

if (!user) return res.status(404).json({ message: 'User not found' });

res.json(user);

} catch (error) {

res.status(500).json({ message: error.message });

}

};

exports.updateUser = async (req, res) => {

try {

const { firstName, lastName, role, status } = req.body;

const user = await User.findByIdAndUpdate(

req.params.id,

{ firstName, lastName, role, status },

{ new: true }

);

res.json(user);

} catch (error) {

res.status(500).json({ message: error.message });

}

};

exports.deleteUser = async (req, res) => {

try {

await User.findByIdAndDelete(req.params.id);

res.json({ message: 'User deleted' });

} catch (error) {

res.status(500).json({ message: error.message });

}

};

**Routes (**routes/userRoutes.js**)**:

const express = require('express');

const router = express.Router();

const { authenticate, authorizeRoles } = require('../middleware/auth');

const {

getAllUsers,

getUserById,

updateUser,

deleteUser,

} = require('../controllers/userController');

// Only coordinator or logistic\_officer (example) can see all users:

router.get('/', authenticate, authorizeRoles(['coordinator', 'logistic\_officer']), getAllUsers);

router.get('/:id', authenticate, getUserById);

router.put('/:id', authenticate, updateUser);

router.delete('/:id', authenticate, authorizeRoles(['coordinator']), deleteUser);

module.exports = router;

**Mount routes in** app.js**:**

const userRoutes = require('./routes/userRoutes');

app.use('/api/users', userRoutes);

### ****5.2 Repeat for Vehicles, Interventions, etc.****

* Follow a similar pattern: define CRUD methods in controllers, link them in routes, and mount them in app.js.

## ****6. Real-Time Communication (Socket.io)****

Once basic CRUD is working:

1. **Install Socket.io**:

npm install socket.io

1. **Initialize Socket.io** in server.js:

const http = require('http');

const socketIo = require('socket.io');

const app = require('./src/app');

const connectDB = require('./src/config/db');

connectDB();

const server = http.createServer(app);

const io = socketIo(server, {

cors: {

origin: '\*',

},

});

io.on('connection', (socket) => {

console.log('New client connected: ', socket.id);

// Example: Listen for vehicle status changes

socket.on('vehicle-status-change', (data) => {

// broadcast to all clients

io.emit('vehicle-updated', data);

});

socket.on('disconnect', () => {

console.log('Client disconnected:', socket.id);

});

});

const PORT = process.env.PORT || 4000;

server.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

1. **Broadcast Updates**
   * In your vehicle controller, after updating a vehicle, emit an event:

exports.updateVehicle = async (req, res) => {

try {

const vehicle = await Vehicle.findByIdAndUpdate(req.params.id, req.body, { new: true });

// Emitting an update to Socket.io

req.app.get('socketio').emit('vehicle-updated', vehicle);

res.json(vehicle);

} catch (error) {

res.status(500).json({ message: error.message });

}

};

* + In app.js, attach io to app:

// In server.js, after creating io:

app.set('socketio', io);

## ****7. Additional Integrations & Features****

**Transcription (Speech-to-Text)**

* + If using an external service (e.g., Google Cloud Speech-to-Text), create a microservice or a route that:
    1. Receives an audio file or stream.
    2. Sends it to the speech-to-text provider.
    3. Saves the transcript in Transcription collection.
  + **Security**: Ensure only authorized roles or background processes can upload audio.

**Predictive Alerts**

* + You could implement a scheduled job (e.g., using **node-cron**) to run a predictive model:

npm install node-cron

const cron = require('node-cron');

const PredictiveAlert = require('./models/PredictiveAlert');

// ...

cron.schedule('\* \* \* \* \*', async () => {

// every minute as example

console.log('Running predictive analysis job...');

// fetch active interventions, run model, store results in PredictiveAlert

});

**Resource Alerts**

* + Similar approach: monitor resource levels in background tasks or real-time streams.

## ****8. Error Handling & Logging****

* Create a global error handling middleware to capture unexpected errors.

// middleware/errorHandler.js

module.exports = (err, req, res, next) => {

console.error(err.stack);

res.status(500).json({ message: 'Something went wrong' });

};

* In app.js, mount this middleware last:

const errorHandler = require('./middleware/errorHandler');

app.use(errorHandler);

* Use a logging library (e.g., **Winston**, **Morgan**) for structured logs.

## ****9. Testing Strategy****

1. **Set Up Testing Framework**
   * Install Jest, Supertest (for HTTP testing):

npm install --save-dev jest supertest

* + Configure scripts in package.json:

{

"scripts": {

"test": "jest --coverage"

}

}

1. **Write Unit Tests**
   * For each controller method, test the logic in isolation.
2. **Write Integration Tests**
   * Use **Supertest** to spin up the Express app and hit the endpoints directly.

## ****10. CI/CD & Deployment****

1. **Continuous Integration**
   * Use GitHub Actions or GitLab CI to run tests and lint checks on push.
2. **Dockerize**
   * Create a Dockerfile:

FROM node:14

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 4000

CMD ["node", "server.js"]

1. **Deployment**
   * Deploy on a cloud platform (AWS, GCP, Azure, or Docker-based services like ECS or Kubernetes).

## ****Sprint Breakdown Example****

* **Sprint 1**:
  + Setup project structure, environment, database connection, minimal routes (User CRUD).
  + Implement user authentication (register/login) + basic role-based access.
* **Sprint 2**:
  + CRUD for Vehicles, Interventions, Regions.
  + Socket.io setup for real-time updates (e.g., vehicle status).
  + Automated tests for core endpoints.
* **Sprint 3**:
  + Integrate Transcription service (mock first, then real).
  + Resource alerts & predictive alerts (minimal logic).
  + Enhanced error handling, logging, and coverage tests.
* **Sprint 4+**:
  + Advanced AI/ML integration (predictive resource needs).
  + Monitoring dashboards, performance optimization, load tests.
  + Finalize deployment pipeline, security audits, and refine documentation.

## ****Conclusion****

Following this plan will give you a **robust, modular backend**. It starts with a clear folder structure, sets up **MongoDB** models, implements **authentication** and **role-based authorization**, introduces **real-time features** with Socket.io, and outlines further **AI integrations** and background tasks. Once you’re confident in the backend, you can proceed to develop the frontend (React/Next.js) to consume these APIs, ensuring a clean separation of concerns and a maintainable codebase.