

New Page (will be separated)

Table of Contents

- [System Overview](#)
- [Architecture Layers](#)
- [Database Architecture](#)
- [Authentication & Authorization](#)
- [API Design](#)
- [WebSocket Architecture](#)
- [File Management](#)
- [Scheduled Tasks](#)
- [Security Architecture](#)
- [Scalability Considerations](#)

System Overview

ON Internal API is a multi-tenant, microservices-style backend system built on Node.js and Express.js. It manages multiple business domains through a unified API gateway while maintaining separation of concerns through modular architecture.

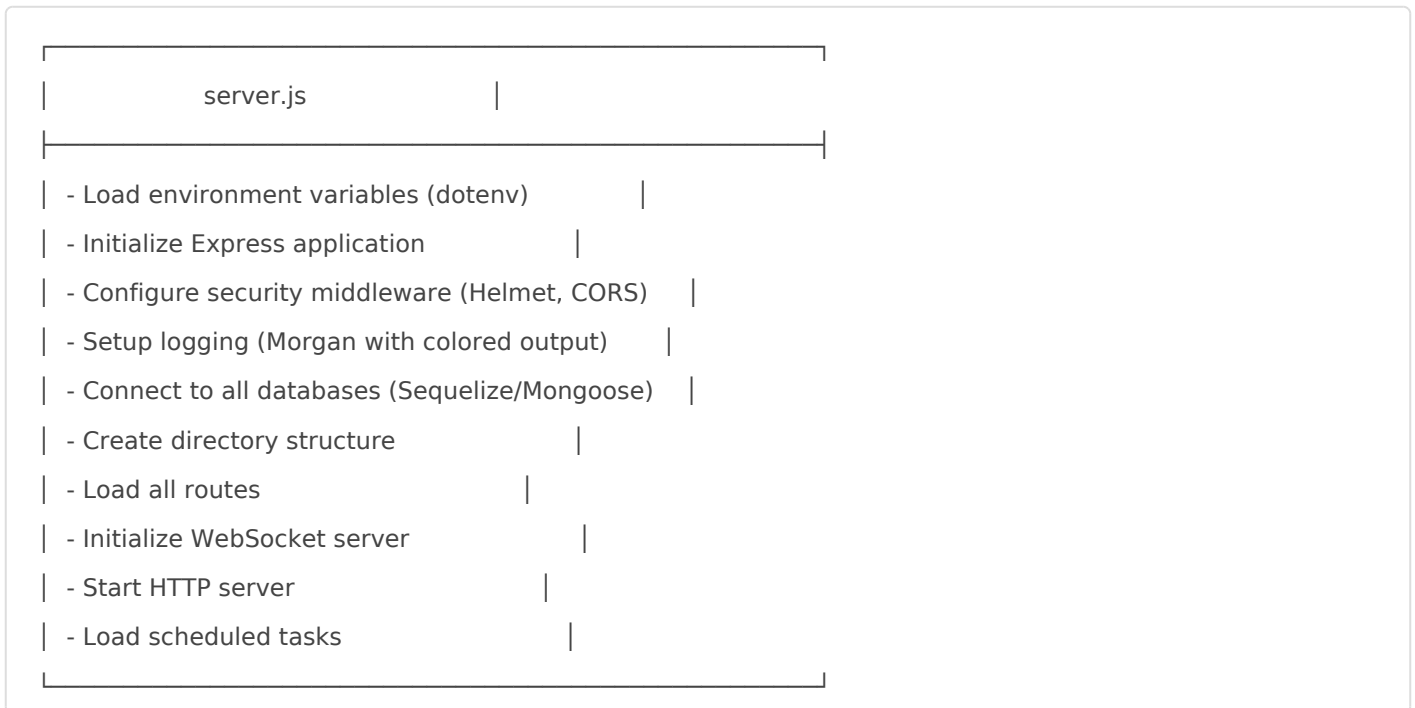
Design Principles

1. **Modularity:** Each business domain (employee, asset, fleet, etc.) is isolated in its own module
2. **Separation of Concerns:** Clear separation between routes, controllers, models, and utilities
3. **Database Segregation:** Different databases for different domains to ensure data isolation
4. **Security First:** Multiple layers of security (Helmet, CORS, JWT, validation)
5. **Real-time Capable:** WebSocket support for live updates

6. **Stateless:** JWT-based authentication allows horizontal scaling

Architecture Layers

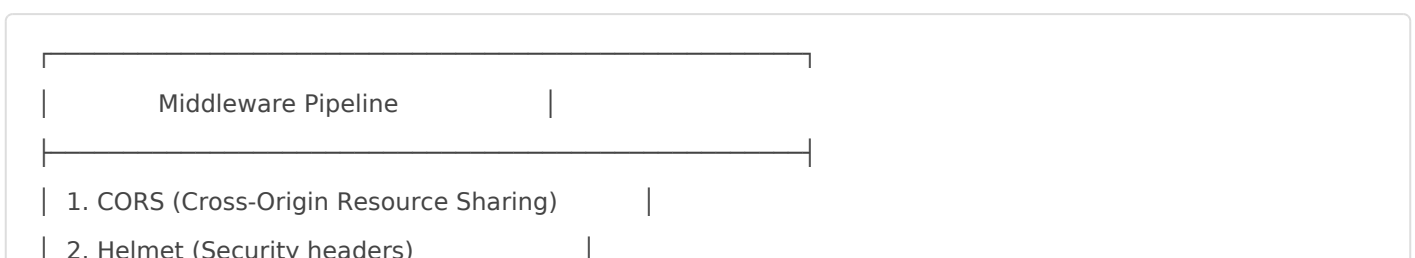
1. Entry Point Layer (server.js)



Key Responsibilities:

- Application bootstrap
- Global configuration
- Middleware stack setup
- Database connection management
- Route registration
- Server initialization

2. Middleware Layer



- | 3. Morgan (Request logging with colors) |
- | 4. Express JSON/URL-encoded parsers (10MB limit) |
- | 5. Custom middleware (per route): |
 - | - authJwt.verifyToken (JWT validation) |
 - | - contentTypeValid (Content-Type check) |
 - | - validator.validate (Schema validation) |
 - | - checkEmployee/checkFleet (Role validation) |
- | 6. Error handler (errorParamHandler) |

Middleware Components:

Security Middleware (`app/middleware/authJwt.js`):

```
verifyToken(req, res, next) {  
  // Extract token from header  
  // Verify JWT signature  
  // Decode and attach user to request  
  // Continue or reject  
}
```

Validation Middleware (`app/schemas/`):

- Uses AJV for JSON Schema validation
- Validates request body, query params, and URL params
- Returns structured error messages

Error Handler (`app/middleware/errorParamHandler.js`):

- Catches all errors
- Formats error responses
- Logs errors
- Returns consistent JSON error format

3. Route Layer

Routes define API endpoints and map them to controllers:

```
// Example: app/routes/auth.routes.js  
module.exports = function (app) {  
  app.post(  
    "/api/auth/login",
```

```

[
  contentTypeValid("application/json"),
  validator.validate({ body: auth.login }),
],
authController.login
);

app.get(
  "/api/auth/account",
  authJwt.verifyToken,
  authController.loadAccount
);
};

```

Route Organization:

- One route file per module
- Middleware applied declaratively
- Clear HTTP method usage
- RESTful design patterns

4. Controller Layer

Controllers contain business logic:

Controller Responsibilities
1. Extract data from request (body, query, params)
2. Call database operations via models
3. Process business logic
4. Format response data
5. Handle errors
6. Return HTTP response

Controller Pattern:

```

exports.methodName = async (req, res) => {
  try {

```

```
// 1. Extract data
const { param } = req.body;

// 2. Validate business rules
if (!param) throw new Error("Missing parameter");

// 3. Database operations
const result = await Model.findOne({ where: { id } });

// 4. Process data
const processed = processData(result);

// 5. Return response
res.status(200).json({
  success: true,
  data: processed
});
} catch (error) {
  // 6. Error handling
  res.status(400).json({
    success: false,
    message: error.message
  });
}
};
```

5. Model Layer

Models define database schemas using Sequelize (PostgreSQL) or Mongoose (MongoDB):

```
// Example Sequelize model
module.exports = (sequelize, Sequelize) => {
  const Model = sequelize.define("table_name", {
    id: {
      type: Sequelize.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
  },
```

```

name: {
  type: Sequelize.STRING,
  allowNull: false
},
// ... other fields
}, {
  timestamps: true,
  tableName: "table_name"
});

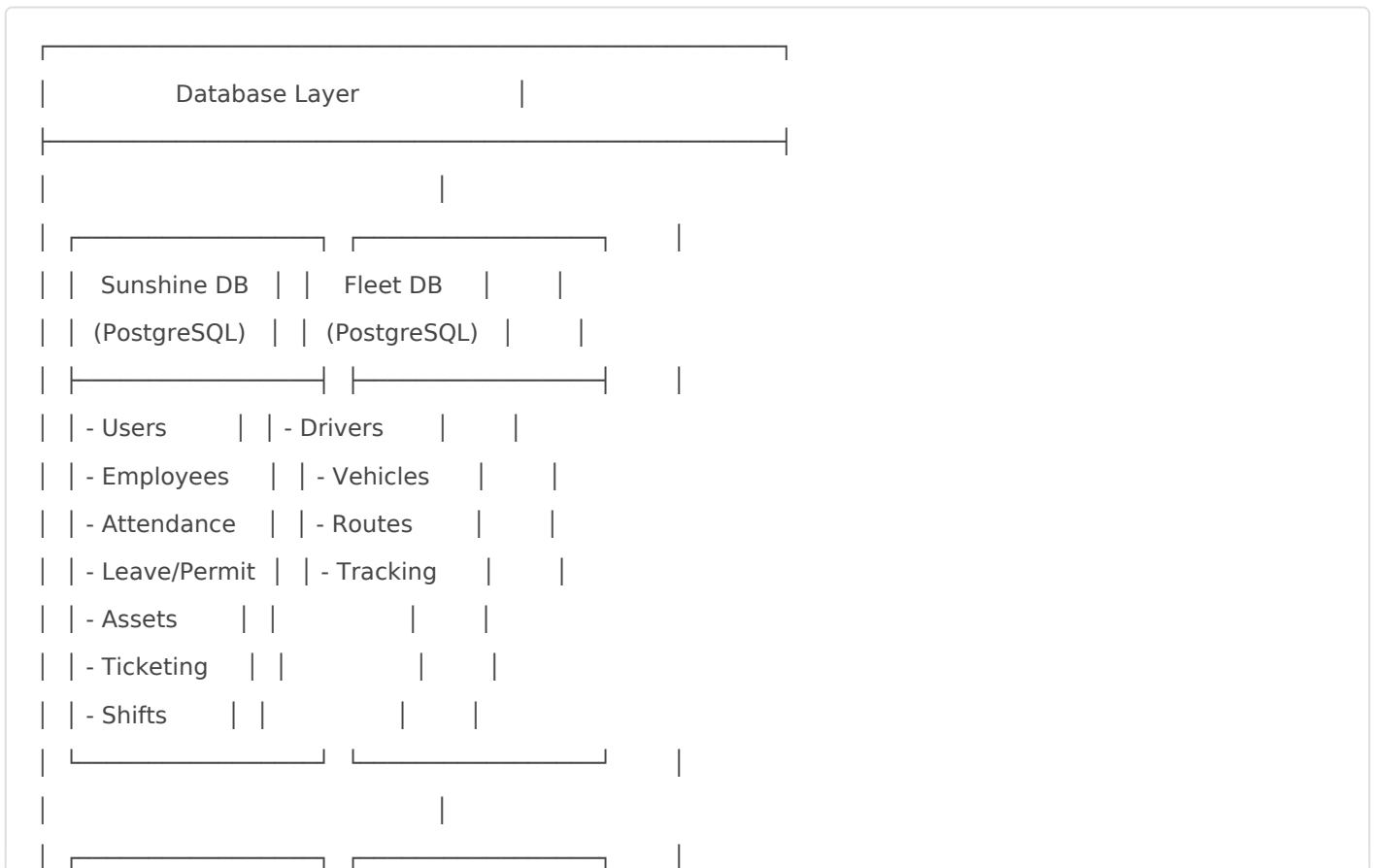
return Model;
};

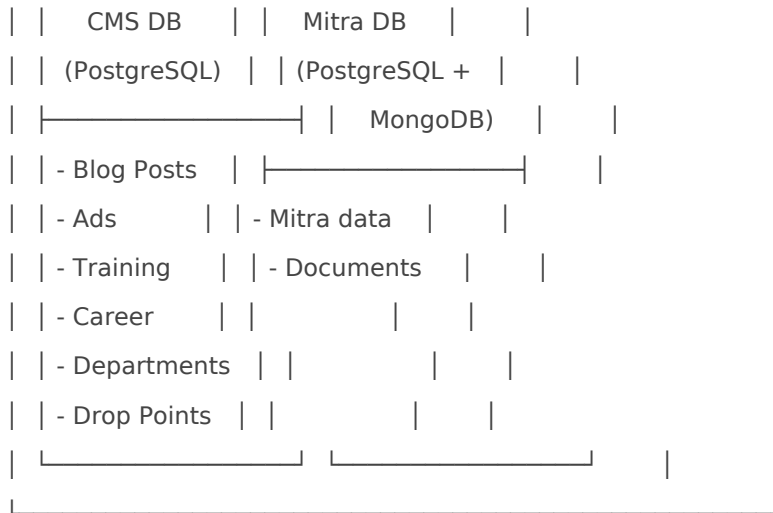
```

Database Architecture

Multi-Database Strategy

The system uses four separate databases:





Database Configuration

Connection Pooling (for performance):

```

pool: {
  max: 5,    // Maximum connections
  min: 0,    // Minimum connections
  acquire: 30000, // Max time to get connection (ms)
  idle: 10000 // Max idle time before closing (ms)
}
  
```

Database Selection Pattern

```

// In models/index.js
const sequelizeSunshine = new Sequelize(sunshineDB.DB, ...);
const sequelizeFleet = new Sequelize(fleetDB.DB, ...);
const sequelizeCms = new Sequelize(cmsDB.DB, ...);

// Models are registered to specific connections
db.employee = require("./employee.model")(sequelizeSunshine, Sequelize);
db.driver = require("./driver.model")(sequelizeFleet, Sequelize);
  
```

Data Relationships

Cross-Database Relationships:

- Avoided when possible for better independence
- When needed, handled at application layer (not DB constraints)
- Use foreign keys within same database

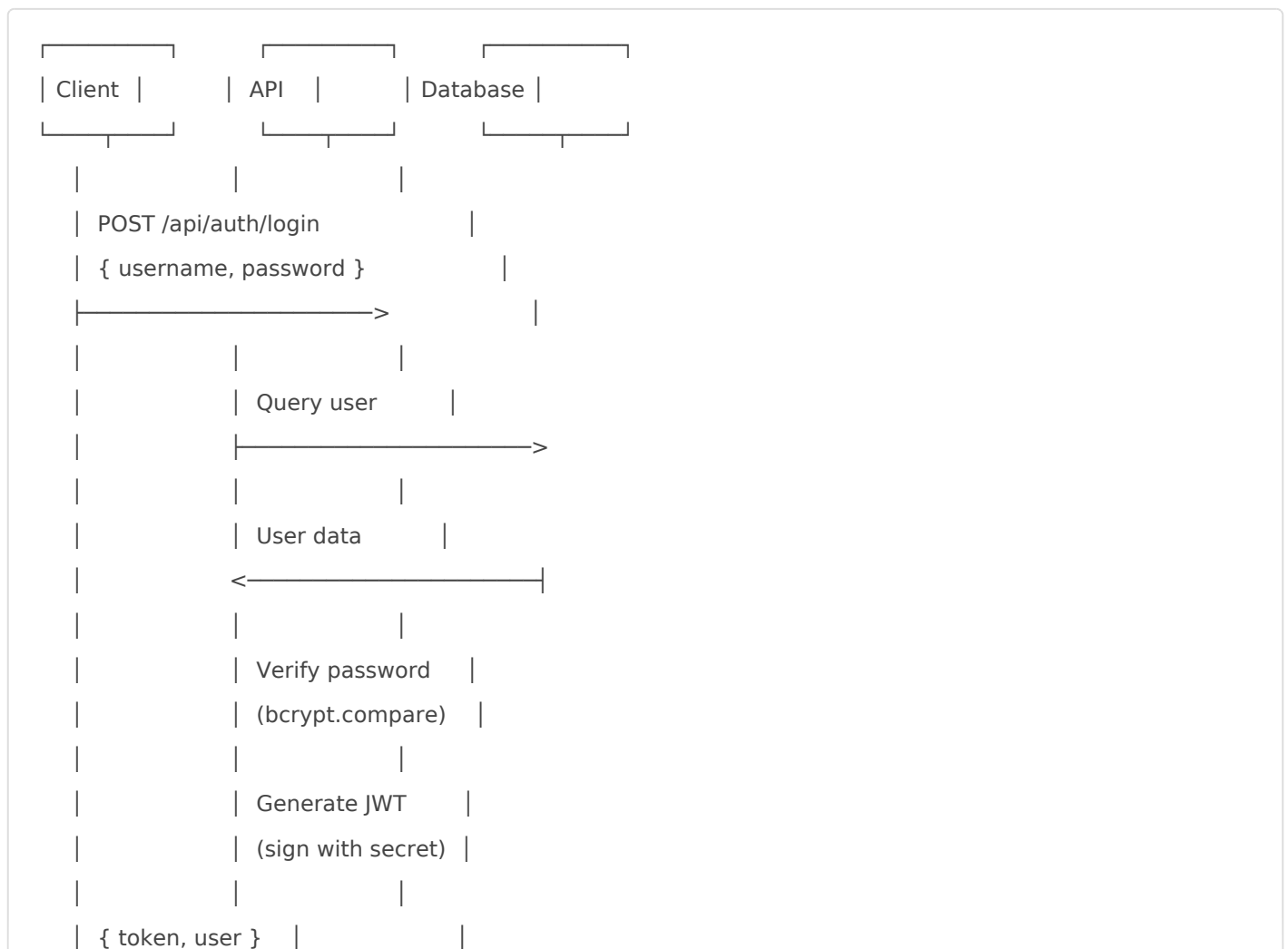
Example:

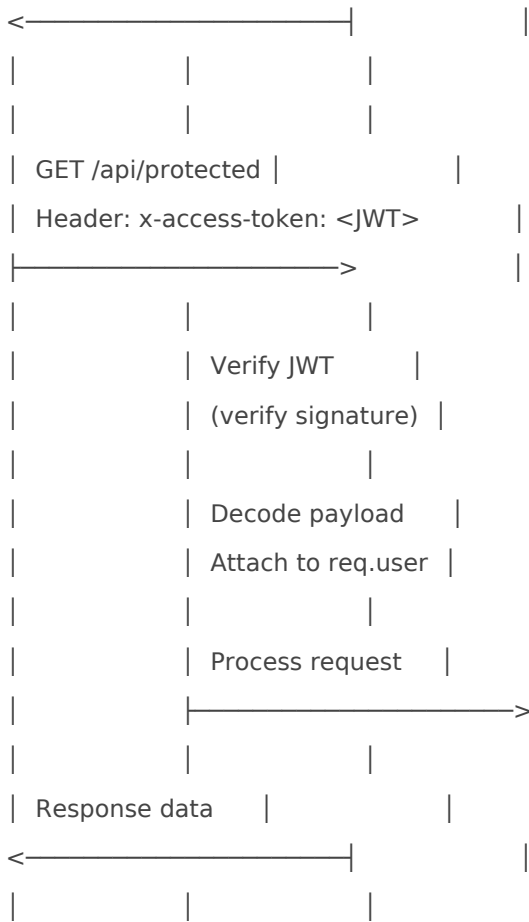
```
// Employee (Sunshine DB) -> Asset Assignment (Sunshine DB) ✓
// Employee has Assets relationship

// Employee (Sunshine DB) -> Driver (Fleet DB) ✗
// Handled via employee_id field and application logic
```

Authentication & Authorization

JWT Authentication Flow





Authentication Components

Login Process (`app/controllers/auth.controller.js`):

1. Receive username/password
2. Query database for user
3. Verify password with bcrypt
4. Generate JWT token with user data
5. Return token and user info

Token Verification (`app/middleware/authJwt.js`):

1. Extract token from header (`x-access-token`)
2. Verify token signature with `JWT_SECRET`
3. Decode payload (user ID, roles, etc.)
4. Attach decoded data to `req.user`
5. Continue to next middleware/controller

Token Structure:

```
{
  id: userId,
  username: "user@example.com",
  role: "admin",
  iat: 1234567890, // Issued at
  exp: 1234654290 // Expires at
}
```

Authorization Patterns

Role-Based Access Control:

```
// In middleware
if (req.user.role !== 'admin') {
  return res.status(403).json({
    message: "Unauthorized access"
  });
}
```

Resource-Based Authorization:

```
// Verify user owns resource
const resource = await Model.findOne({
  where: { id: resourceId, userId: req.user.id }
});

if (!resource) {
  return res.status(403).json({
    message: "Access denied"
  });
}
```

API Design

RESTful Principles

Resource Naming:

- Plural nouns: `/api/employees`, `/api/assets`
- Nested resources: `/api/employees/:id/shifts`
- Actions as sub-resources: `/api/tickets/:id/assign`

HTTP Methods:

- `GET`: Read/retrieve data
- `POST`: Create new resource
- `PUT`: Update entire resource
- `PATCH`: Partial update
- `DELETE`: Remove resource

Status Codes:

- `200`: Success
- `201`: Created
- `400`: Bad request (validation error)
- `401`: Unauthorized (not authenticated)
- `403`: Forbidden (not authorized)
- `404`: Not found
- `500`: Internal server error

Response Format

Success Response:

```
{
  "success": true,
  "data": {
    "id": 1,
    "name": "Example"
  },
  "message": "Operation successful"
}
```

Error Response:

```
{
  "success": false,
  "message": "Validation failed",
  "errors": [
    {
      "field": "email",
```

```
  "message": "Invalid email format"
}
]
}
```

Pagination (when applicable):

```
{
  "success": true,
  "data": [...],
  "pagination": {
    "page": 1,
    "perPage": 20,
    "total": 150,
    "totalPages": 8
  }
}
```

Input Validation

Using AJV JSON Schema:

```
// app/schemas/example.schema.js
module.exports = {
  create: {
    type: "object",
    properties: {
      name: { type: "string", minLength: 1 },
      email: { type: "string", format: "email" },
      age: { type: "integer", minimum: 18 }
    },
    required: ["name", "email"],
    additionalProperties: false
  }
};
```

Applied in routes:

```
app.post(
  "/api/example",
  validator.validate({ body: exampleSchema.create }),
  controller.create
);
```

WebSocket Architecture

WebSocket Server Setup

```
// app/routes/ws.routes.js
const WebSocket = require("ws");

module.exports = (server, app) => {
  const wss = new WebSocket.Server({ noServer: true });

  server.on("upgrade", (request, socket, head) => {
    // Parse URL to route to correct WebSocket handler
    const pathname = url.parse(request.url).pathname;

    if (pathname === "/ws/branding-approval") {
      wss.handleUpgrade(request, socket, head, (ws) => {
        wss.emit("connection", ws, request);
      });
    }
  });

  wss.on("connection", (ws, request) => {
    // Handle connection
    ws.on("message", (message) => {
      // Handle message
    });

    ws.on("close", () => {
      // Handle disconnect
    });
  });
};
```

```
});  
};
```

WebSocket Endpoints

1. **Branding Approval** (`/ws/branding-approval`)
 - Real-time approval status updates
 - Mobile and web clients
 - Notification delivery
2. **Ticketing** (`/ws/ticketing`)
 - Ticket status updates
 - Assignment notifications
 - Real-time chat (if applicable)

Connection Management

Heartbeat (Ping/Pong):

```
// interval.js  
setInterval(function ping() {  
  wss.clients.forEach(function each(ws) {  
    if (ws.isAlive === false) {  
      return ws.terminate();  
    }  
    ws.isAlive = false;  
    ws.ping();  
  });  
}, 30000); // Every 30 seconds
```

Client Tracking:

```
// Track connected clients  
const clients = new Map();  
  
wss.on("connection", (ws, request) => {  
  const userId = extractUserId(request);  
  clients.set(userId, ws);  
  
  ws.on("close", () => {  
    clients.delete(userId);  
  });  
});
```

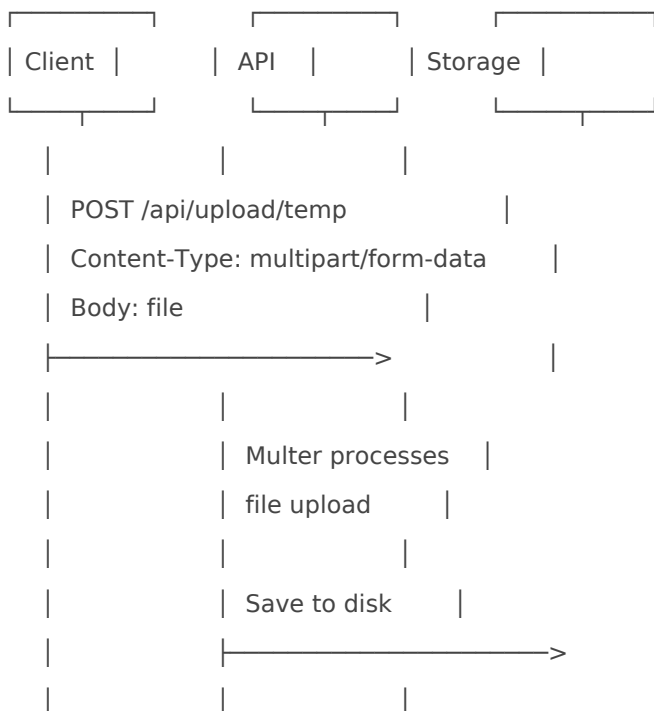
```
});  
});
```

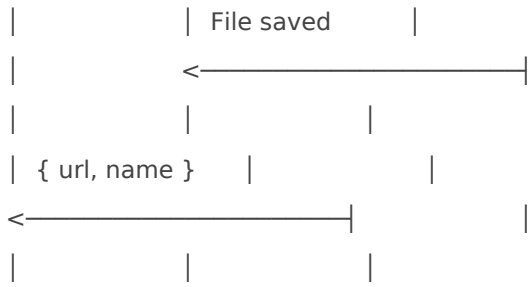
Broadcasting:

```
// Send to all clients  
wss.clients.forEach((client) => {  
  if (client.readyState === WebSocket.OPEN) {  
    client.send(JSON.stringify(data));  
  }  
});  
  
// Send to specific user  
const userWs = clients.get(userId);  
if (userWs && userWs.readyState === WebSocket.OPEN) {  
  userWs.send(JSON.stringify(data));  
}
```

File Management

Upload Flow





Storage Structure

```
resources/
├── static/      # Permanent files
│   ├── employee/  # Employee photos, documents
│   ├── ticketing/ # Ticket attachments
│   ├── cms/
│       ├── ad/    # Advertisement images
│       ├── blog/  # Blog post images
│       ├── training/ # Training materials
│       └── department/ # Department logos
│   ├── branding-approval/ # Branding assets
│   └── onapps/
│       └── voucher/ # Voucher images
└── temp/        # Temporary files (auto-deleted)
```

File Upload Middleware

```
// app/middleware/uploadTemp.js
const multer = require("multer");
const path = require("path");

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, path.join(__remoteDir, __stageEnv, "temp"));
  },
  filename: (req, file, cb) => {
    const uniqueName = `${Date.now()}-${file.originalname}`;
    cb(null, uniqueName);
  }
});
```

```
});

const upload = multer({
  storage: storage,
  limits: { fileSize: 10 * 1024 * 1024 }, // 10MB
  fileFilter: (req, file, cb) => {
    // Validate file type if needed
    cb(null, true);
  }
});
```

File Serving

Temporary Files:

- Served via `/api/temp/:filename`
- Automatically deleted after first download
- Used for reports, exports, etc.

Static Files:

- Served via `/api/static/:filename`
- Permanent storage
- Used for user uploads, images, etc.

Scheduled Tasks

Task Scheduler Architecture

```
// app/schedulers/file-temporary.scheduler.js
const cron = require("node-cron");

exports.cleanTempFile = () => {
  // Run every hour
  cron.schedule("0 * * * *", async () => {
    const tempDir = path.join(__remoteDir, __stageEnv, "temp");
    const files = fs.readdirSync(tempDir);
```

```
files.forEach(file => {
  const filePath = path.join(tempDir, file);
  const stats = fs.statSync(filePath);
  const now = Date.now();
  const age = now - stats.mtimeMs;

  // Delete files older than 24 hours
  if (age > 24 * 60 * 60 * 1000) {
    fs.unlinkSync(filePath);
  }
});
};
```

Scheduled Tasks

1. Temporary File Cleanup

- Schedule: Every hour
- Action: Delete temp files older than 24 hours
- Purpose: Prevent disk space issues

2. Leave Auto-Rejection

- Schedule: Daily
- Action: Auto-reject pending leave requests past deadline
- Purpose: Workflow automation

Adding New Scheduled Tasks

```
// 1. Create scheduler file
// app/schedulers/your-task.scheduler.js
const cron = require("node-cron");

exports.yourTask = () => {
  cron.schedule("0 0 * * *", async () => { // Daily at midnight
    // Your task logic
  });
};

// 2. Load in server.js
```

```
const yourTaskScheduler = require("../app/schedulers/your-task.scheduler");
yourTaskScheduler.yourTask();
```

Security Architecture

Security Layers

Security Layers
1. Transport Layer (HTTPS in production)
2. CORS (Cross-Origin Resource Sharing)
3. Helmet.js (Security headers)
4. Request Size Limits (10MB)
5. JWT Authentication
6. Input Validation (AJV JSON Schema)
7. SQL Injection Protection (Sequelize ORM)
8. Password Hashing (bcryptjs)
9. Error Sanitization (no stack traces in prod)

Security Headers (Helmet.js)

Automatically applied headers:

- X-DNS-Prefetch-Control
- X-Frame-Options
- X-Content-Type-Options
- X-XSS-Protection
- Strict-Transport-Security (HTTPS only)

CORS Configuration

```
app.use(cors()); // Allow all origins (configure for production)
```

```
// Custom CORS for specific resources
```

```
app.use((req, res, next) => {
  res.setHeader("Cross-Origin-Resource-Policy", "cross-origin");
  next();
});
```

Production CORS (recommended):

```
const corsOptions = {
  origin: [
    "https://yourdomain.com",
    "https://app.yourdomain.com"
  ],
  credentials: true
};
app.use(cors(corsOptions));
```

SQL Injection Prevention

Using Sequelize ORM with parameterized queries:

```
// ✓ Safe - parameterized
User.findOne({
  where: { email: userInput }
});

// ✗ Unsafe - raw query without parameters
sequelize.query(`SELECT * FROM users WHERE email = '${userInput}'`);

// ✓ Safe - raw query with parameters
sequelize.query(
  "SELECT * FROM users WHERE email = :email",
  {
    replacements: { email: userInput },
    type: QueryTypes.SELECT
  }
);
```

Password Security

```
const bcrypt = require("bcryptjs");

// Hashing (on user creation)
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);

// Verification (on login)
const isValid = await bcrypt.compare(password, user.password);
```

Scalability Considerations

Horizontal Scaling

Stateless Design:

- JWT tokens (no server-side sessions)
- No in-memory state (use database/cache)
- Load balancer compatible

Database Connection Pooling:

- Limited connections per instance
- Reuse existing connections
- Automatic cleanup of idle connections

Performance Optimization

Database Queries:

- Use indexes on frequently queried fields
- Limit result sets (pagination)
- Use `select` to fetch only needed fields
- Avoid N+1 queries (use `include` for joins)

Caching (future consideration):

```
// Example with Redis
const redis = require("redis");
const client = redis.createClient();
```

```
// Cache frequently accessed data
const cached = await client.get(key);
if (cached) return JSON.parse(cached);

const data = await database.query();
await client.setex(key, 3600, JSON.stringify(data));
```

Request Rate Limiting (future consideration):

```
const rateLimit = require("express-rate-limit");

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // Limit each IP to 100 requests per windowMs
});

app.use("/api/", limiter);
```

Monitoring & Logging

Current Logging:

- Morgan for HTTP request logging
- Colored console output for readability
- Sequelize query logging

Production Logging (recommended):

- Use Winston or Bunyan
- Log to files/external service
- Structured logging (JSON format)
- Different log levels (error, warn, info, debug)

Metrics (future consideration):

- Response times
 - Error rates
 - Database query performance
 - Active connections
 - Memory/CPU usage
-

Document Version: 1.0.0
Last Updated: 2026-02-04

Revision #1

Created 5 February 2026 03:37:26 by ondelivelooper

Updated 5 February 2026 03:38:11 by ondelivelooper