Transportation Management System - Backend Architecture Documentation

Last Updated: January 18, 2026
Version: 1.0
Tech Stack: Node.js + Express.js + Prisma + PostgreSQL

📁 Project Structure Overview

```
backend/
├── src/
│   ├── config/          # Application configuration
│   ├── controllers/     # Request handlers (HTTP layer)
│   ├── jobs/           # Background tasks & cron jobs
│   ├── middleware/     # Express middleware functions
│   ├── routes/         # API route definitions
│   ├── services/       # Business logic layer
│   ├── utils/          # Helper functions & utilities
│   ├── validators/     # Request validation schemas
│   └── app.js          # Express application setup
├── prisma/
│   └── schema.prisma   # Database schema
├── server.js           # Application entry point
├── .env                # Environment variables (not in git)
├── .env.example        # Environment template
└── package.json        # Dependencies & scripts
```

🗂️ Folder-by-Folder Documentation

📂 `src/config/` - Configuration Files

Purpose: Centralize all application configuration and external service connections.

File	Purpose	Key Responsibilities

`database.js`	Prisma client initialization	Singleton pattern for database connection, exports configured Prisma client
`env.js`	Environment variable validation	Validates required env vars on startup, provides typed config object

Why separate config?

- Single source of truth for settings
- Easy to mock in tests
- Prevents scattered configuration across codebase
- Validates environment on startup (fail-fast principle)

Example Usage:

```
// database.js
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();
module.exports = prisma;

// env.js
module.exports = {
  PORT: process.env.PORT || 3000,
  DATABASE_URL: process.env.DATABASE_URL,
  JWT_SECRET: process.env.JWT_SECRET,
  // ... validates all required vars
};
```

📂 `src/controllers/` - HTTP Request Handlers

Purpose: Handle HTTP requests, validate input, call services, return responses.

Rule: Controllers should be thin - they orchestrate, they don't contain business logic.

File	Handles	Typical Operations
`auth.controller.js`	Authentication	Login, register, token refresh, logout
`loads.controller.js`	Load management	CRUD operations for loads, status updates

`drivers.controller.js`	Driver management	Driver CRUD, availability, assignments
`vehicles.controller.js`	Fleet management	Vehicle CRUD, assignments, maintenance
`shippers.controller.js`	Shipper clients	Shipper CRUD, users, statistics
`documents.controller.js`	Load documents	Upload, approval, OCR processing
`pod.controller.js`	Proof of delivery	POD submission, verification, photos
`invoices.controller.js`	Billing	Invoice generation, payments, line items
`settlement.controller.js`	Driver payments	Settlement creation, approval, disputes
`notification.controller.js`	Notifications	Get, mark read, preferences

Controller Pattern:

```
class LoadController {
  async getAllLoads(req, res, next) {
    try {
      // 1. Extract & validate params
      const { page, limit, status } = req.query;

      // 2. Call service layer
      const result = await loadService.getLoads({ page, limit, status });

      // 3. Return formatted response
      return ApiResponse.paginated(res, result.data, result.pagination);
    } catch (error) {
      next(error); // Pass to error handler
    }
  }
}
```

Why separate controllers?

- Thin HTTP layer, easy to test
- Reusable business logic in services
- Consistent error handling
- Easy to swap transport layer (REST → GraphQL)

📂 `src/jobs/` - Background Tasks & Cron Jobs

Purpose: Scheduled tasks that run automatically without user interaction.

File	Schedule	Purpose
`index.js`	-	Initializes all cron jobs, exports `startJobs()`
`cleanupNotifications.job.js`	Daily 2 AM	Delete expired & old read notifications
`checkOverdueInvoices.job.js`	Hourly	Find overdue invoices, send notifications
`checkExpiringDocuments.job.js`	Daily 8 AM	Alert for expiring licenses, medical certs, insurance

Why background jobs?

- Automated maintenance (cleanup, archiving)
- Proactive alerts (expiring documents, overdue payments)
- Reduce manual administrative work
- Keep database clean and performant

Pattern:

```
// index.js
const cron = require('node-cron');

const startJobs = () => {
  cron.schedule('0 2 * * *', cleanupNotifications);
  cron.schedule('0 * * * *', checkOverdueInvoices);
  console.log('Background jobs started');
};
```

```
module.exports = { startJobs };
```

Integration:

```
// server.js
const { startJobs } = require('./src/jobs');
if (process.env.NODE_ENV === 'production') {
  startJobs();
}
```

📂 `src/middleware/` - Express Middleware

Purpose: Functions that intercept requests before they reach controllers.

File	Purpose	When to Use
`auth.js`	JWT authentication	Verify user identity, attach `req.user`
`roleCheck.js`	Authorization	Check user has required role/permissions
`validation.js`	Input validation	Validate request body/params against schemas
`errorHandler.js`	Error handling	Global error handler, formats error responses
`auditLog.js`	Audit logging	Log all state-changing operations (CREATE/UPDATE/DELETE)
`upload.js`	File uploads	Handle multipart/form-data, validate files
`rateLimit.js`	Rate limiting	Prevent abuse, limit requests per IP/user

Middleware Execution Order:

```
// app.js
app.use(rateLimit);           // 1. Rate limit first
app.use(auth);                // 2. Authenticate user
app.use(roleCheck(['ADMIN'])); // 3. Check permissions
app.use(validation);          // 4. Validate input
// → Controller executes
app.use(auditLog);            // 5. Log the action
app.use(errorHandler);        // 6. Catch any errors
```

Why middleware?

- DRY: Write once, apply everywhere
- Separation of concerns: Auth ≠ business logic
- Easy to test independently
- Composable: Stack multiple middleware

Example:

```
// routes/loads.routes.js
router.post(
  '/',
  auth,                           // Must be logged in
  roleCheck(['SHIPPER_USER']),    // Must be shipper
  validate(loadValidator.create), // Validate request body
  loadController.createLoad       // Finally, create load
);
```

📂 `src/routes/` - API Route Definitions

Purpose: Define URL endpoints and map them to controller functions.

File	Base Path	Purpose
`index.js`	`/api`	Main router, combines all sub-routers
`auth.routes.js`	`/api/auth`	Login, register, refresh token
`loads.routes.js`	`/api/loads`	Load CRUD, status updates
`drivers.routes.js`	`/api/drivers`	Driver management, HOS records
`vehicles.routes.js`	`/api/vehicles`	Fleet management, maintenance

`shippers.routes.js`	`/api/shippers`	Shipper client management
`documents.routes.js`	`/api/documents`	Document upload/approval
`pod.routes.js`	`/api/pod`	POD submission/verification
`invoices.routes.js`	`/api/invoices`	Invoice generation/payment
`settlements.routes.js`	`/api/settlements`	Driver settlements
`notifications.routes.js`	`/api/notifications`	User notifications
`reports.routes.js`	`/api/reports`	Analytics & reporting

Route Pattern:

```
// loads.routes.js
const express = require('express');
const router = express.Router();
const loadController = require('../controllers/loads.controller');
const { auth, roleCheck, validate } = require('../middleware');

// GET /api/loads - List all loads
router.get('/', auth, loadController.getAllLoads);

// POST /api/loads - Create new load
router.post(
  '/',
  auth,
  roleCheck(['SHIPPER_USER']),
  validate(loadValidator.create),
  loadController.createLoad
);

// GET /api/loads/:id - Get specific load
router.get('/:id', auth, loadController.getLoadById);

module.exports = router;
```

Index Router Pattern:

```
// index.js
const express = require('express');
const router = express.Router();

router.use('/auth', require('./auth.routes'));
router.use('/loads', require('./loads.routes'));
router.use('/drivers', require('./drivers.routes'));
// ... etc

module.exports = router;
```

📂 `src/services/` - Business Logic Layer

Purpose: Contains all business logic, calculations, and complex operations.

Rule: Services are pure business logic - no HTTP, no req/res objects.

File	Responsibility
`auth.service.js`	Password hashing, token generation, user authentication logic
`loads.service.js`	Complex load operations, status transitions, validation rules
`notification.service.js`	Create/send notifications, bulk operations, cleanup
`email.service.js`	Send emails (SMTP, templates, attachments)
`storage.service.js`	File upload to S3, signed URLs, file management
`audit.service.js`	Create audit logs, track changes, compliance
`geocoding.service.js`	Validate addresses, geocode locations (Google Maps API)
`pdf.service.js`	Generate PDFs (invoices, BOLs, PODs)

Why services?

- **Reusability:** Use from controllers, jobs, or CLI scripts
- **Testability:** No HTTP dependencies, pure functions
- **Single Responsibility:** Each service has one job
- **Business Logic Centralization:** Not scattered in controllers

Service Pattern:

```
// notification.service.js
class NotificationService {
  async createNotification(data) {
    // Validation
    if (!data.recipientId) throw new Error('Recipient required');

    // Business logic
    const notification = await prisma.notification.create({ data });

    // Side effects (email, SMS, push)
    await this.sendViaChannels(notification);

    return notification;
  }

  async sendViaChannels(notification) {
    if (notification.sentViaEmail) {
      await emailService.send(/* ... */);
    }
    // ... SMS, push, etc
  }
}
```

Controller uses Service:

```
// notification.controller.js
async createNotification(req, res, next) {
  try {
    const notification = await
notificationService.createNotification(req.body);
    return ApiResponse.success(res, notification, 201);
  } catch (error) {
    next(error);
  }
}
```

📂 `src/utils/` - Helper Functions & Utilities

Purpose: Reusable utility functions used across the application.

File	Purpose	Example Functions
`errors.js`	Custom error classes	`ValidationError`, `NotFoundError`, `UnauthorizedError`
`constants.js`	App-wide constants	User types, roles, status enums, limits
`formatters.js`	Data formatting	Format phone numbers, currency, dates
`jwt.js`	JWT operations	Sign tokens, verify tokens, decode
`logger.js`	Logging utility	Winston logger, log levels, file/console output
`response.js`	Standard API responses	Success, error, paginated response helpers

Why utils?

- DRY: Reuse common operations
- Consistency: Same formatting everywhere
- Easy to update: Change in one place
- Testable: Pure functions

Examples:

```
// errors.js
class NotFoundError extends Error {
  constructor(message = 'Resource not found') {
    super(message);
    this.statusCode = 404;
  }
}

// constants.js
```

```
module.exports = {
  USER_TYPES: {
    INTERNAL: 'INTERNAL_USER',
    SHIPPER: 'SHIPPER_USER',
    DRIVER: 'DRIVER'
  },
  PAGINATION: {
    DEFAULT_LIMIT: 20,
    MAX_LIMIT: 100
  }
};

// formatters.js
function formatCurrency(amount) {
  return `$${Number(amount).toFixed(2)}`;
}

function formatPhoneE164(phone) {
  // Convert to +12345678900 format
}

// response.js
class ApiResponse {
  static success(res, data, message, statusCode = 200) {
    return res.status(statusCode).json({
      success: true,
      message,
      data
    });
  }

  static error(res, message, statusCode = 400) {
    return res.status(statusCode).json({
      success: false,
      message
    });
  }

  static paginated(res, data, pagination) {
    return res.json({
      success: true,
      data,
      pagination
    });
  }
}
```

📂 `src/validators/` - Input Validation Schemas

Purpose: Define validation rules for incoming request data using Joi.

File	Validates
`auth.validator.js`	Login, register, password reset
`load.validator.js`	Load creation, updates, status changes
`driver.validator.js`	Driver registration, profile updates
`vehicle.validator.js`	Vehicle registration, maintenance records
`invoice.validator.js`	Invoice creation, line items, payments
`settlement.validator.js`	Settlement creation, deductions, approval

Why separate validators?

- Centralized validation rules
- Reusable across routes
- Clear error messages
- Easy to maintain

Pattern:

```
// load.validator.js
const Joi = require('joi');

const loadValidator = {
  create: Joi.object({
    origin: Joi.string().required(),
    destination: Joi.string().required(),
    equipmentType: Joi.string()
      .valid('DRY_VAN', 'REEFER', 'FLATBED', 'STEP_DECK', 'LOWBOY')
      .required(),
    weightLbs: Joi.number().integer().min(1).required(),
    pickupDate: Joi.date().iso().required(),
    deliveryDate: Joi.date().iso().min(Joi.ref('pickupDate')).required(),
    commodity: Joi.string().required(),
    specialInstructions: Joi.string().allow('').optional()
  }),

  update: Joi.object({
    status: Joi.string().valid('DRAFT', 'PENDING_REVIEW', /* ... */),
    // ... partial update fields
  }),

  statusUpdate: Joi.object({
    status: Joi.string().required(),
    notes: Joi.string().optional()
  })
};

module.exports = loadValidator;
```

Usage in routes:

```
// routes/loads.routes.js
const validate = require('../middleware/validation');
const loadValidator = require('../validators/load.validator');

router.post(
  '/',
  auth,
  validate(loadValidator.create),
  loadController.createLoad
);
```

🔄 Request Flow Diagram

```
┌─────────────────┐
│   Client        │
│  (Browser/      │
│  Mobile App)    │
└────────┬────────┘
         │ HTTP Request
         ▼
┌──────────────────────────────────────────────────────┐
│         Express Middleware Stack                     │
├──────────────────────────────────────────────────────┤
│  1. rateLimit.js   → Check rate limits               │
│  2. auth.js        → Verify JWT                      │
│  3. roleCheck.js   → Check permissions               │
│  4. validation.js  → Validate input                  │
└───────────────────┬──────────────────────────────────┘
                    │
                    ▼
┌──────────────────────────────────────────────────────┐
│           Routes (Routing)                           │
│  Match URL → Call Controller                         │
└───────────────────┬──────────────────────────────────┘
                    │
                    ▼
┌──────────────────────────────────────────────────────┐
│        Controllers (HTTP Layer)                      │
│  - Extract params                                    │
│  - Call Service                                      │
│  - Format response                                   │
└───────────────────┬──────────────────────────────────┘
                    │
                    ▼
┌──────────────────────────────────────────────────────┐
│      Services (Business Logic)                       │
│  - Validate business rules                           │
│  - Perform calculations                              │
│  - Call other services                               │
│  - Interact with database                            │
└───────────────────┬──────────────────────────────────┘
                    │
                    ▼
┌──────────────────────────────────────────────────────┐
│      Prisma ORM → PostgreSQL                         │
│  Database operations                                 │
```

```
└───────────────────┬──────────────────────────────┘
                 │
                 ▼  (Response bubbles back up)
┌──────────────────────────────────────────────────┐
│       auditLog.js Middleware                │
│  Log the operation                          │
└───────────────────┬──────────────────────────────┘
                 │
                 ▼

┌──────────────────────────────────────────────────┐
│         Return JSON Response                │
│  { success: true, data: {...} }             │
└───────────────────┬──────────────────────────────┘
                 │
                 ▼
            Client receives response
```

🛠️ Key Design Patterns

1. Layered Architecture

- **Presentation Layer:** Routes + Controllers
- **Business Layer:** Services
- **Data Layer:** Prisma + PostgreSQL

2. Dependency Injection

- Services are singletons (exported instances)
- Controllers import services, not vice versa
- Easy to mock in tests

3. Error Handling

- Custom error classes with status codes
- Try-catch in controllers
- Global error handler middleware
- Consistent error format

4. Middleware Chain

- Composable, reusable

- Order matters (auth before roleCheck)
- Each middleware has single responsibility

5. Service Pattern

- Business logic separate from HTTP
- Reusable from controllers, jobs, CLI
- Pure functions where possible

📊 File Relationship Map

```
Routes
  ↓ (defines endpoints)
Controllers
  ↓ (calls)
Services ←────────┐
  ↓             |
Database      Utils (helpers)
  ↑             |
  └─────────────────┘


Validators ──→ Middleware ──→ Routes
                   ↓
               (protects)
                   ↓
              Controllers
```

🧪 Testing Strategy

Unit Tests

- **Test:** Services, Utils
- **Why:** Pure business logic, no dependencies
- **Tools:** Jest, isolated from DB

Integration Tests

- **Test:** Controllers + Services + Database
- **Why:** Verify full request-response cycle

- **Tools:** Jest + Supertest + Test DB

E2E Tests

- **Test:** Full user flows
- **Why:** Ensure features work end-to-end
- **Tools:** Playwright/Cypress

🚀 Development Workflow

1. Add New Feature

1. Define Prisma schema → Run migration
2. Create validator (if new entity)
3. Create service (business logic)
4. Create controller (HTTP handler)
5. Define routes
6. Add middleware (auth, validation)
7. Test

2. Debug Request

1. Check routes → Is URL correct?
2. Check middleware → Is auth/validation passing?
3. Check controller → Is service called correctly?
4. Check service → Is business logic correct?
5. Check logs → Any errors?

3. Performance Issue

1. Check database queries (Prisma logs)
2. Add indexes if needed
3. Optimize N+1 queries (use include/select)
4. Add caching if appropriate
5. Use pagination

📚 Best Practices

Controllers

- Thin controllers, fat services
- Always use try-catch with next(error)
- Use ApiResponse helper for consistency
- No business logic in controllers
- No direct database calls (use services)

Services

- Single responsibility
- Reusable, testable functions
- Return data, throw errors
- No req/res objects
- No HTTP status codes (that's controller's job)

Middleware

- One job per middleware
- Call next() or next(error)
- Attach data to req object if needed
- Don't do complex business logic

Routes

- RESTful URL design
- Proper HTTP verbs (GET, POST, PUT, DELETE)
- Version API (/api/v1/...)
- Use middleware for protection

🔐 Security Checklist

- [x] JWT authentication on protected routes
- [x] Role-based access control
- [x] Input validation on all endpoints
- [x] Rate limiting to prevent abuse
- [x] SQL injection protection (Prisma ORM)
- [x] XSS protection (sanitize inputs)
- [x] CORS configuration
- [x] Helmet.js for security headers
- [x] Audit logging for compliance
- [x] Environment variables for secrets

📖 Common Questions

Q: Where do I add new API endpoint?

A:

1. Create/update controller
2. Add route in routes file
3. Add to routes/index.js if new resource

Q: Where do I add validation?

A: Create schema in validators/, use in route with validate() middleware

Q: Where do I add complex logic?

A: Services folder, not in controllers

Q: Where do I add scheduled tasks?

A: jobs/ folder, register in jobs/index.js

Q: How do I handle errors?

A: Throw custom errors (from utils/errors.js), errorHandler middleware catches them

Q: Where do I log operations?

A: Use auditLog middleware for state changes, logger for debugging

🎯 Quick Reference

Need to...	File/Folder
Add API endpoint	routes/ + controllers/
Add business logic	services/
Validate input	validators/

Add middleware	middleware/
Add helper function	utils/
Add cron job	jobs/
Change database	prisma/schema.prisma
Add constants	utils/constants.js

End of Documentation

Keep this document updated as the architecture evolves.