Development Guidelines

Code Structure Conventions

Controller Pattern

```
// controllers/<feature>/<feature>.controller.js

exports.functionName = async (req, res) => {
  try {
    // 1. Extract parameters
    const { param1, param2 } = req.body;

    // 2. Validate input (AJV schema)
    const validate = ajv.compile(schema);
    if (!validate(req.body)) {
      return res.status(400).json({
        message: "Validation error",
        errors: validate.errors
      });
    }

    // 3. Business logic
    const result = await performOperation(param1, param2);

    // 4. Return response
    return res.status(200).json({
      message: "Success",
      data: result
    });

  } catch (error) {
    console.error(error);
    return res.status(500).json({
      message: "Internal server error"
```

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

Route Pattern

```
// routes/<feature>.routes.js

const controller = require('../controllers/<feature>/<feature>.controller');
const { authJwt } = require('../middleware');

module.exports = function(app) {
  app.post(
    '/api/<feature>/<action>',
    [authJwt.verifyToken],
    controller.functionName
  );
};
```

Model Pattern

```
// models/<model>.model.js

module.exports = (sequelize, Sequelize) => {
  const Model = sequelize.define("model_name", {
    id: {
      type: Sequelize.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    field1: {
      type: Sequelize.STRING,
      allowNull: false
    },
    created_at: {
      type: Sequelize.DATE,
      defaultValue: Sequelize.NOW
    }
  }, {
```

```
    tableName: 'table_name',
    timestamps: false
  });

  return Model;
};
```

Naming Conventions

Files:

- Controllers: `<feature>.controller.js`
- Models: `<model>.model.js`
- Routes: `<feature>.routes.js`
- Utilities: `<purpose>.util.js`

Functions:

- camelCase for function names
- Descriptive names: `createWaybill`, `calculateFee`, `updateScan`

Variables:

- camelCase for variables
- Constants: UPPER_SNAKE_CASE

Database:

- Table names: snake_case, plural
- Column names: snake_case

Error Handling

Standard Error Response:

```
{
  "message": "Error description",
  "error": "Error code or type",
  "details": { /* additional context */ }
}
```

HTTP Status Codes:

- 200: Success
- 201: Created
- 400: Bad Request (validation errors)
- 401: Unauthorized (authentication failed)
- 403: Forbidden (insufficient permissions)
- 404: Not Found
- 500: Internal Server Error

Error Handler Middleware:

Location: `app/middleware/error_param_handler.js`

Usage: Automatically applied to all routes

Input Validation

AJV Schema Example:

```
// schemas/<feature>.schema.js

const createWaybillSchema = {
  type: "object",
  properties: {
    origin_code: { type: "string", minLength: 3 },
    destination_code: { type: "string", minLength: 3 },
    weight: { type: "number", minimum: 0 },
    service_type: { type: "string", enum: ["REG", "EXP", "CARGO"] }
  },
  required: ["origin_code", "destination_code", "weight", "service_type"],
  additionalProperties: false
};

module.exports = { createWaybillSchema };
```

Usage in Controller:

```
const ajv = new Ajv();
const { createWaybillSchema } = require('../schemas/waybill.schema');

const validate = ajv.compile(createWaybillSchema);
if (!validate(req.body)) {
```

```
  return res.status(400).json({
    message: "Validation failed",
    errors: validate.errors
  });
}
```

Database Best Practices

Transactions:

```
const t = await db.sequelize.transaction();

try {
  // Multiple operations
  await Waybill.create(data, { transaction: t });
  await Scan.create(scanData, { transaction: t });

  await t.commit();
} catch (error) {
  await t.rollback();
  throw error;
}
```

Query Optimization:

```
// Use specific fields instead of SELECT *
await Waybill.findAll({
  attributes: ['id', 'awb_number', 'status'],
  where: { status: 'BOOKED' },
  limit: 100
});

// Use includes for relationships
await Waybill.findOne({
  where: { id: waybillId },
  include: [
    { model: Scan, as: 'scans' },
    { model: Partner, as: 'partner' }
  ]
```

```
});
```

Logging Best Practices

Console Logging:

```
// Use cli-color for important logs
const clc = require('cli-color');

console.log(clc.green('Success: Waybill created'));
console.log(clc.yellow('Warning: Weight discrepancy detected'));
console.log(clc.red('Error: Database connection failed'));
```

Morgan HTTP Logging:

Already configured in `ondeliv-backend.js` with custom colored format.

Testing

Manual Testing Scripts:

- `check_fee_test.js` - Test fee calculation
- `check_price.js` - Test pricing lookup
- `debug_provinces.js` - Debug location data

Run Tests:

```
node check_fee_test.js
node check_price.js
```

Git Workflow

Branch Strategy (from README.md):

- `master/main` - Production branch
- `demo/trial` - Demo environment
- `development` - Development branch
- `hotfix` - Quick fixes from production
- `features` - New features
- `release` - Production-ready staging
- `experimental` - Experimental features

Commit Messages:

```
feat: Add new courier partner integration
fix: Resolve fee calculation bug
docs: Update API documentation
refactor: Improve waybill creation logic
test: Add unit tests for scanning operations
```

Code Review Checklist

Before submitting code:

- [ ] Code follows naming conventions
- [ ] Input validation implemented (AJV schemas)
- [ ] Error handling implemented
- [ ] Database transactions used where needed
- [ ] Authentication/authorization checked
- [ ] Logging added for important operations
- [ ] No sensitive data in logs
- [ ] No hardcoded credentials
- [ ] Comments added for complex logic
- [ ] Tested manually with sample data

Revision #1
Created 24 February 2026 07:01:26 by ondeliveloper
Updated 24 February 2026 07:02:57 by ondeliveloper