# DRCP Server - Backend Overview

This backend implements the Disaster Relief Coordination Platform (DRCP) API, following a modular, scalable, and maintainable structure inspired by best practices and the SQLEARN/BACK project.

---

## Key Backend Goals

- Secure, role-based authentication (victim, volunteer, admin, etc.)
- Incident reporting, updating, and tracking
- Volunteer selection, assignment, and navigation for incidents
- Real-time group chat per incident (victims & volunteers)
- Resource and donation management per incident
- Admin management of users, incidents, tasks, resources, analytics
- Ownership and access control for all sensitive operations
- Input validation and consistent error responses
- Scalable, maintainable, and testable codebase

---

## Project Structure

- **app.js**: Main Express app, middleware, routes, error handler
- **server.js**: Server entry point
- **controllers/**: Thin route handlers, delegate to services
- **services/**: Business logic, DB operations, transactions
- **routes/**: Express routers, organized by resource (incidents, users, chat, donations, etc.)
- **validators/**: Joi or Mongoose validation for request bodies
- **middlewares/**: Auth, role/ownership checks, request-level logic
- **errorHandler/**: Centralized error class and Express error middleware
- **db/**: Database connection logic (Mongoose/MongoDB)
- **models/**: Mongoose schemas/models for all resources
- **utils/**: Utility functions, helpers, and test suites

---

## Modules to Implement

- [ ] **User/Auth**: Registration, login, JWT, role management
- [ ] **Incident**: CRUD, status, assignment, reporting
- [ ] **Volunteer**: Profile, incident selection, assignment, navigation
- [ ] **Chat**: Real-time group chat per incident (Socket.io)
- [ ] **Resource**: Resource and donation management, allocation
- [ ] **Admin**: User, incident, resource, analytics management
- [ ] **Validation**: Joi/Mongoose schemas for all input
- [ ] **Ownership/Access**: Middleware for resource access control
- [ ] **Error Handling**: Use `CustomError` and centralized error middleware
- [ ] **Testing**: Jest/Supertest for API and logic

## Error Handling Pattern

All errors should be passed to the next middleware using:

```
next(error instanceof CustomError ? error : new CustomError(
    error.message || "Failed to [action]. Please try again.",
    error.statusCode || STATUS_CODE.INTERNAL_SERVER_ERROR,
    error
));
```

---

## Current Implementation Status

### ☑ Implemented

- **Project Structure & Scaffolding**
  - Modular folders: `controllers/`, `services/`, `routes/`, `models/`, `middlewares/`, `errorHandler/`, etc.
- **User Module**
  - User model with password hashing, JWT methods, geospatial location, and role support
  - User registration and login (with hashed passwords and JWT)
  - User CRUD: get, update, delete, get all, delete all
  - User service layer (all business logic in `services/userService.js`)
  - User controller (thin, delegates to service, robust error handling)
  - User routes (with authentication middleware placeholder)
- **Incident Module**
  - Incident model (basic schema)
  - Incident controller (basic CRUD)
  - Incident routes (basic)
- **Error Handling**
  - Centralized error handler middleware
  - Custom error classes (`CustomError`, etc.)
  - Consistent error handling pattern in controllers/services
- **Environment & Config**
  - `.env` usage for secrets and DB connection
  - `config.js` for config management
- **Dependencies**
  - All core dependencies in `package.json` (Express, Mongoose, JWT, bcryptjs, Joi, etc.)

### ◍ In Progress / Planned

- **Role-based Access Control**
  - Middleware for admin/role checks (to be implemented)
- **Input Validation**
  - Joi validation for request bodies (to be implemented)

- **Other Modules**
  - Volunteer, Resource, Donation, Chat, Admin, Task models/services/controllers/routes
- **Socket.io Integration**
  - Real-time chat and updates (to be implemented)
- **Testing**
  - Jest/Supertest setup (to be implemented)
- **API Documentation**
  - Endpoint documentation (ongoing)

---

**Summary:**
User and incident modules are scaffolded and functional with robust error handling and modular structure. Other modules, validation, role checks, and real-time features are planned next.

---

## Next Steps & Implementation Guidelines

1. **Scaffold All Folders**
   - Create empty folders: `controllers/`, `services/`, `routes/`, `validators/`, `middlewares/`, `errorHandler/`, `db/`, `models/`, `utils/`, and `tests/`.
2. **Set Up Core Files**
   - Implement `app.js`, `server.js`, and `config.js` as described.
   - Add a sample `.env` file for environment variables.
3. **Implement Modules Incrementally**
   - Start with User/Auth (registration, login, JWT, role).
   - Proceed to Incidents, Volunteers, Chat, Resources, Admin, etc.
   - For each module, create: model, validator, service, controller, route.
4. **Follow Error Handling Pattern**
   - Always use the provided error handling pattern in all async handlers.
5. **Validation**
   - Use Joi or Mongoose validation for all incoming data.
6. **Testing**
   - Write tests for each route and service using Jest/Supertest.
7. **Documentation**
   - Document each endpoint and module in this README or a dedicated API docs file.

---

**Follow this structure for a clean, maintainable, and scalable DRCP backend.**