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.

DRCP Server - Progress Summary

☑ Completed/Stable

- **User Module:** Model, registration, login, profile update, JWT, validation, error handling, tests.
- **Incident Module:** Model, CRUD, reporting, assignment, status update, validation, error handling, tests.
- **Error Handling:** Centralized, consistent pattern.
- **Validation:** Joi/Mongoose for user, incident, volunteer.
- **Testing:** Jest/Supertest for user, incident, and volunteer flows.
- **Project Structure:** Modular, maintainable, scalable.

🔘 Volunteer Module (Current Focus)

- **Model:** Implemented and stable.
- **Service:** Implemented (assignment, status/location update, get assigned incidents).
- **Controller:** Implemented (assignment, profile update, get assigned incidents).
- **Routes:** Implemented (clean, RESTful).
- **Tests:** Implemented and passing for all major flows (assignment, status/location update, edge cases).

🔘 Remaining/Planned

- **Resource/Donation Module:** Model, service, controller, routes, tests.
- **Chat Module:** Real-time group chat per incident (Socket.io), backend logic, tests.
- **Admin Module:** User/incident/resource/task management, analytics, role-based access, tests.
- **Task Module:** (If needed) For admin/volunteer task assignment.
- **Ownership/Access Middleware:** Fine-grained access control for all resources.
- **API Documentation:** Expand and maintain endpoint docs.
- **Socket.io Integration:** For chat and real-time updates.
- **More Tests:** Edge cases, integration, and security tests for all modules.

Summary

- **User, Incident, and Volunteer modules are implemented, tested, and stable.**
- **Volunteer module is nearly complete and robust.**
- **Next up:** Resource/Donation, Chat, Admin, and Task modules, plus more validation, access control, and documentation.

You are about halfway through the core backend. The foundation is strong and the next modules will build on this structure.

DRCP Backend API Endpoints

User/Auth

Method	Endpoint	Description
POST	/api/users/register	Register user (victim/volunteer/admin)
POST	/api/users/login	Login user
GET	/api/users/profile	Get current user profile
PATCH	/api/users/profile	Update profile (location, status, etc.)
DELETE	/api/users/:userId	Delete user (if not assigned)
POST	/api/users/accept-invitation	Accept gathering invitation
GET	/api/users/incident-history	Get user's incident history (paginated)
GET	/api/users/:userId/incident-history	Get incident history for user (admin/self)
GET	/api/users/:userId/assigned-incident	Get assigned incident for user

Incident

Method	Endpoint	Description
GET	/api/incidents	List all incidents (filtered for volunteers, paginated)
POST	/api/incidents	Create new incident (victim cannot create if assigned)
GET	/api/incidents/:incidentId	Get incident by ID
PATCH	/api/incidents/:incidentId	Update incident
DELETE	/api/incidents/:incidentId	Delete incident
PATCH	/api/incidents/:incidentId/status	Update incident status (victim/volunteer removed on resolve)
POST	/api/incidents/:incidentId/assign	Assign volunteer to incident
GET	/api/incidents/nearby	Get incidents near authenticated user
GET	/api/incidents/:incidentId/victims	List victims for incident (paginated)
GET	/api/incidents/:incidentId/volunteers	List volunteers for incident (paginated)
GET	/api/incidents/:incidentId/reports	Get all reports for incident (paginated)
POST	/api/incidents/:incidentId/reports	Add victim report to incident

Method	Endpoint	Description
POST	/api/incidents/:incidentId/reports/:reportIndex/accept	Accept a report for an incident
POST	/api/incidents/:incidentId/gathering-invitation	Send gathering invitation
POST	/api/incidents/report	Report a new incident (victim/volunteer)
POST	/api/incidents/:incidentId/add-victim	Add victim to incident

Volunteer

Method	Endpoint	Description
POST	/api/volunteers/assign	Assign self to incident
GET	/api/volunteers/assigned-incidents	List incidents assigned to this volunteer (paginated)

Resource/Donation

Method	Endpoint	Description
POST	/api/resources/donate	Donate a resource
GET	/api/resources	List all resources (paginated)
GET	/api/resources/:resourceId	Get resource by ID
POST	/api/resources/allocate	Allocate resource to incident
POST	/api/resources/bulk-allocate	Bulk allocate resources
GET	/api/resources/history	Get donation history (paginated)

Chat (REST & Socket.io)

Method	Endpoint	Description
GET	/api/chat/incident/:incidentId	Get all chat messages for incident (paginated)
POST	/api/chat/incident/:incidentId/message	Send chat message

Socket.io Events:

- `joinIncident` (join a chat room)
- `sendMessage` (send message, broadcast to room)
- `newMessage` (receive new message in real-time)

Admin

Method	Endpoint	Description

Method	Endpoint	Description
GET	/api/admin/users	List all users (paginated, if implemented)
GET	/api/admin/incidents	List all incidents (paginated, if implemented)
GET	/api/admin/resources	List all resources (paginated, if implemented)
GET	/api/admin/donations	List all donations (paginated, if implemented)

DRCP Backend API – Complete Reference

Overview

This backend powers the DRCP disaster response platform. It supports user management, incident reporting and resolution, volunteer assignment, resource/donation management, chat, and admin features.
All endpoints are RESTful, use JWT authentication (except registration/login), and return consistent JSON responses.

Data & Response Format

- **All list endpoints** return `{ items/array, pagination }` (e.g., `{ incidents, pagination }`).
- **Pagination object**: `{ page, limit, totalPages, totalItems }`.
- **All incident objects** have `victims` and `volunteers` populated as full user objects.
- **All error responses** use `{ error: "..." }` and correct status codes.
- **All IDs** are MongoDB ObjectIds as strings.

Key Behaviors & Restrictions

- **Role Change Restriction:** Victims/volunteers cannot change their role if assigned to any unresolved incident.
- **Incident/Report Creation Restriction:** Victims cannot create/report a new incident if assigned to an active incident.
- **Incident Resolve Logic:** When a victim or volunteer resolves, they are removed from the incident. The incident is marked as resolved only when all victims and volunteers are removed.
- **Assignment Tracking:** Each user has an `assignedIncident` field, updated on assignment/removal.
- **Deletion Restriction:** Users cannot be deleted if assigned to any active incident.
- **Role-based Access:** All endpoints except registration/login require JWT authentication and enforce role-based access.

Endpoints

User/Auth

Method	Endpoint	Description
POST	/api/users/register	Register user (victim/volunteer/admin)
POST	/api/users/login	Login user
GET	/api/users/profile	Get current user profile
PATCH	/api/users/profile	Update profile (location, status, etc.)
DELETE	/api/users/:userId	Delete user (if not assigned)
POST	/api/users/accept-invitation	Accept gathering invitation
GET	/api/users/incident-history	Get user's incident history (paginated)
GET	/api/users/:userId/incident-history	Get incident history for user (admin/self)
GET	/api/users/:userId/assigned-incident	Get assigned incident for user

Incident

Method	Endpoint	Description
GET	/api/incidents	List all incidents (filtered for volunteers, paginated)
POST	/api/incidents	Create new incident (victim cannot create if assigned)
GET	/api/incidents/:incidentId	Get incident by ID
PATCH	/api/incidents/:incidentId	Update incident
DELETE	/api/incidents/:incidentId	Delete incident
PATCH	/api/incidents/:incidentId/status	Update incident status (victim/volunteer removed on resolve)
POST	/api/incidents/:incidentId/assign	Assign volunteer to incident
GET	/api/incidents/nearby	Get incidents near authenticated user
GET	/api/incidents/:incidentId/victims	List victims for incident (paginated)
GET	/api/incidents/:incidentId/volunteers	List volunteers for incident (paginated)
GET	/api/incidents/:incidentId/reports	Get all reports for incident (paginated)
POST	/api/incidents/:incidentId/reports	Add victim report to incident

Method	Endpoint	Description
POST	/api/incidents/:incidentId/reports/:reportIndex/accept	Accept a report for an incident
POST	/api/incidents/:incidentId/gathering-invitation	Send gathering invitation
POST	/api/incidents/report	Report a new incident (victim/volunteer)
POST	/api/incidents/:incidentId/add-victim	Add victim to incident

Volunteer

Method	Endpoint	Description
POST	/api/volunteers/assign	Assign self to incident
GET	/api/volunteers/assigned-incidents	List incidents assigned to this volunteer (paginated)

Resource/Donation

Method	Endpoint	Description
POST	/api/resources/donate	Donate a resource
GET	/api/resources	List all resources (paginated)
GET	/api/resources/:resourceId	Get resource by ID
POST	/api/resources/allocate	Allocate resource to incident
POST	/api/resources/bulk-allocate	Bulk allocate resources
GET	/api/resources/history	Get donation history (paginated)

Chat (REST & Socket.io)

Method	Endpoint	Description
GET	/api/chat/incident/:incidentId	Get all chat messages for incident (paginated)
POST	/api/chat/incident/:incidentId/message	Send chat message

Socket.io Events:

- `joinIncident` (join a chat room)
- `sendMessage` (send message, broadcast to room)
- `newMessage` (receive new message in real-time)

Admin

Method	Endpoint	Description

Method	Endpoint	Description
GET	/api/admin/users	List all users (paginated, if implemented)
GET	/api/admin/incidents	List all incidents (paginated, if implemented)
GET	/api/admin/resources	List all resources (paginated, if implemented)
GET	/api/admin/donations	List all donations (paginated, if implemented)

Error Handling

- All errors use `{ error: "..." }` and correct status codes.
- Consistent error format across all controllers/services.
- Validation errors use `{ error: "..." }` and status 400.

Pagination

- All list endpoints support `page` and `limit` query parameters.
- Response format:

```
{
  "status": "success",
  "incidents": [ /* array of incident objects */ ],
  "pagination": {
    "page": 2,
    "limit": 5,
    "totalPages": 4,
    "totalItems": 20
  }
}
```

- Applies to: incidents, users, incident history, assigned incidents, resources, donations, chat messages, reports, victims, volunteers.

Authentication & Access Control

- All endpoints except registration/login require JWT authentication.
- Role-based access enforced in controllers/middlewares.
- Victims/volunteers cannot change role if assigned to any incident.
- Victims cannot create/report a new incident if assigned to an active incident.
- Users cannot be deleted if assigned to any active incident.

Incident Resolution Logic

- When a victim or volunteer sends a resolve request, they are removed from the incident.

- Incident is marked as resolved only when all victims and volunteers are removed.
- All assignments are cleared when incident is resolved.

Response Format

- All incident responses have populated victims and volunteers as full user objects.
- All IDs are MongoDB ObjectIds as strings.
- All responses are JSON.

Validators & Middlewares

- Joi validators for all request bodies.
- Centralized error handler middleware.
- Authentication and role/ownership middlewares.
- Input validation for all endpoints.

Usage Summary

- Victims report incidents and add reports.
- Volunteers assign themselves to incidents and accept reports.
- Admins can view, update, and delete incidents.
- All users can view their incident history and assigned incident.
- Admins (or authorized users) can add additional victims to an incident.
- When a victim/volunteer resolves, they are removed from the incident. When all are resolved, the incident is marked as resolved.

Example Error Response

```
{
  "error": "Cannot change role while assigned to any incident."
}
```

Example Success Response (Paginated List)

```
{
  "status": "success",
  "incidents": [ /* array of incident objects */ ],
  "pagination": {
    "page": 1,
    "limit": 10,
    "totalPages": 2,
    "totalItems": 15
```

```
  }
}
```

Example Incident Object

```
{
  "_id": "incidentId123",
  "title": "Flood in Area",
  "description": "Severe flooding reported",
  "status": "open",
  "location": { "type": "Point", "coordinates": [77, 12] },
  "victims": [
    { "_id": "userId123", "name": "Alice", "email": "alice@example.com", ... }
  ],
  "volunteers": [
    { "_id": "volunteerId456", "name": "Bob", "email": "bob@example.com", ... }
  ],
  "reports": [ /* ... */ ],
  "resources": [],
  "createdAt": "2025-07-14T17:05:40.386Z"
}
```

For full details, see individual controller/service files and endpoint documentation.