# **API Documentation**

## **History-Service:** **Base URL:** The API is hosted on <http://localhost:5006/api/history>.

## **Schema**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Type** | **Description** | **Constraints/Details** |
| userId | mongoose.Schema.Types.ObjectId | Refers to the user associated with the history entry. This is a foreign key to a Usermodel. | **Required**; Must be a valid ObjectId referring to a User. (ref: 'User') |
| type | String | Specifies the type of history entry, e.g., water, electricity, or waste. | **Required**; Enum: 'water', 'electricity', 'waste'. |
| description | String | A textual description of the history entry (e.g., details about the action taken or event). | **Required**; No max length set but should be a concise textual description. |
| coordinates | [Number] | Stores geographic coordinates (latitude and longitude) of the location related to the history entry. | **Required**; Array of two numbers, typically representing [longitude, latitude]. |
| status | String | Represents the status of the history entry (e.g., whether it is completed). | Enum: 'completed' (indicating a finished task). |
| assignedEmployee | mongoose.Schema.Types.ObjectId | Refers to the employee assigned to the task. This links to a User who may be responsible for the entry. | **Optional**; Can be null if no employee is assigned. If assigned, it must be a valid ObjectId referring to a User. |
| createdAt | Date | Timestamp indicating when the history entry was created. | **Default**: Date.now (Auto-generated when a new entry is created). |
| updatedAt | Date | Timestamp indicating when the history entry was last updated. | **Default**: Date.now (This should be manually updated if the document is modified later, as the schema doesn’t auto-handle it). |

## **Endpoints**

### **1. Create a New History**

**Endpoint:** POST /api/history/  
**Description:** Creates a new history entry.

#### **Request**

* **Headers:**
  + Authorization: Bearer token (required for authentication).
* **Body (JSON):**

{  
 "userId": "string",  
 "type": "string",  
 "description": "string",  
 "coordinates": [latitude, longitude],  
 "assignedEmployee": "string (optional)",  
 "status": "string (default: resolved)",  
 "createdAt": "string (ISO 8601 format, optional)",  
 "updatedAt": "string (ISO 8601 format, optional)"}

#### **Response**

* **201 Created**:

{  
 "\_id": "string",  
 "userId": "string",  
 "type": "string",  
 "description": "string",  
 "coordinates": [latitude, longitude],  
 "assignedEmployee": "string",  
 "status": "string",  
 "createdAt": "string",  
 "updatedAt": "string"  
}

* **400 Bad Request**: Invalid coordinates format.
* **500 Internal Server Error**: Server-side issue.

#### **Rate Limiting**

* 100 requests per 15 minutes per IP.

### **2. Get All Histories**

**Endpoint:** GET /api/history/all  
**Description:** Fetches all history entries (Admin only).

#### **Request**

* **Headers:**
  + Authorization: Bearer token (required for authentication).
  + Admin privileges required.

#### **Response**

* **200 OK**:

[  
 {  
 "\_id": "string",  
 "userId": "string",  
 "type": "string",  
 "description": "string",  
 "coordinates": [latitude, longitude],  
 "assignedEmployee": "string",  
 "status": "string",  
 "createdAt": "string",  
 "updatedAt": "string"  
 }  
]

* **500 Internal Server Error**: Server-side issue.

### **3. Get History by User ID**

**Endpoint:** GET /api/history/user/:userId  
**Description:** Fetches all histories for a specific user.

#### **Request**

* **Headers:**
  + Authorization: Bearer token (required for authentication).
* **Params:**
  + userId: ID of the user.

#### **Response**

* **200 OK**:

[  
 {  
 "\_id": "string",  
 "userId": "string",  
 "type": "string",  
 "description": "string",  
 "coordinates": [latitude, longitude],  
 "assignedEmployee": "string",  
 "status": "string",  
 "createdAt": "string",  
 "updatedAt": "string"  
 }  
]

* **404 Not Found**: No history found for the user.
* **500 Internal Server Error**: Server-side issue.

### **4. Health Check**

**Endpoint:** GET /api/history/health  
**Description:** Provides the status of the circuit breaker.

#### **Response**

* **200 OK**:

{  
 "circuitBreaker": {  
 "open": false,  
 "closed": true,  
 "stats": {  
 "successes": number,  
 "failures": number,  
 "rejects": number,  
 "timeouts": number,  
 "latencyTimes": [number]  
 }  
 }  
}

## **Middleware**

1. **Authentication Middleware:** Ensures that the user is authenticated.
2. **Admin Middleware:** Restricts access to certain endpoints to admin users.
3. **Rate Limiter:** Limits requests to specific endpoints to prevent abuse.

## **Error Handling**

* **ValidationError:** Handled during operations like creating a history.
* **Circuit Breaker:** Ensures stability under high load or repeated failures by opening when error thresholds are exceeded.
* **Database Retries:** Up to 3 retries for transient errors with exponential backoff.

## **Architecture Highlights**

## **Circuit Breaker**

### **Purpose:**

The circuit breaker pattern is implemented to handle failures gracefully in database operations. It prevents overloading the system by limiting retries during persistent failures.

### **How It Works:**

* **Operation Execution:** Database operations (e.g., creating, fetching records) are wrapped inside the circuit breaker.
* **Failure Threshold:** If more than 50% of requests fail, the circuit breaker "opens," temporarily stopping further database operations to prevent resource exhaustion.
* **Timeout:** The breaker allows a maximum execution time of 15 seconds per request.
* **Reset Mechanism:** After 30 seconds, the circuit breaker transitions to "half-open," where it allows a few operations to check if the system is recovering.
* **Event Logging:** Logs circuit breaker state changes (open, half-open, closed).

### **Advantages:**

* **System Stability:** Prevents cascading failures by limiting retries under high load or repeated failures.
* **Resilience:** Protects the system during outages or degraded performance.

## **Rate Limiting**

### **Purpose:**

Rate limiting ensures that API endpoints are not abused by limiting the number of requests from a single IP address.

### **Implementation Details:**

* **Middleware:** Used express-rate-limit to apply limits.
* **Limits:**
  + **Create History Endpoint:** Maximum of 100 requests per 15 minutes per IP address.
* **User Feedback:** If the limit is exceeded, the client receives a 429 Too Many Requests response with a custom error message.

### **Advantages:**

* **Protection Against Abuse:** Reduces the risk of spamming and denial-of-service (DoS) attacks.
* **Fair Usage:** Ensures that resources are fairly distributed among users.

## **MongoDB Transactions (ACID properties)**

### **Purpose:**

Transactions are used to ensure atomicity when performing operations on the database, especially during the creation of a new history entry.

### **Steps in the Transaction:**

1. **Session Start:** A new session is started before the operation.
2. **Transaction Initiation:** The session begins a transaction using session.startTransaction().
3. **Operation Execution:** The history document is created and saved within the transaction.
4. **Commit or Rollback:**
   1. If successful, the transaction is committed using session.commitTransaction().
   2. If an error occurs, the transaction is rolled back using session.abortTransaction().

### **Advantages:**

* **Consistency:** Ensures that all related database operations succeed or fail as a whole.
* **Error Recovery:** Allows rolling back changes if an error occurs, preventing partial updates.

## **Database Retry Mechanism**

### **Purpose:**

Retries add resilience to database operations by handling transient errors that might resolve upon retrying.

### **How It Works:**

* **Library:** Uses async-retry for implementing retries.
* **Retry Configuration:**
  + **Attempts:** Up to 3 retries.
  + **Backoff Strategy:** Exponential backoff with a doubling delay between retries.
  + **Bail-Out Condition:** Certain errors, like validation errors, are not retried.

### **Advantages:**

* **Resilience:** Handles temporary issues (e.g., network interruptions, temporary database unavailability).
* **Improved Reliability:** Reduces the impact of transient issues.

## **Middleware Layers**

### **Authentication Middleware:**

Ensures that users accessing endpoints are authenticated. The Authorization header is validated to verify the user's identity.

### **Admin Middleware:**

Restricts access to admin-specific endpoints, like fetching all history records.

### **Advantages:**

* **Security:** Prevents unauthorized access to sensitive endpoints.
* **Granular Control:** Enables differentiated access based on user roles.

## **Health Check Endpoint**

### **Purpose:**

The /health endpoint monitors the circuit breaker’s state to provide insights into the API's operational health.

### **Response Details:**

* **Circuit Breaker Status:**
  + open: Whether the circuit is open or not.
  + closed: Whether the circuit is closed or not.
  + **Statistics:** Includes data such as the number of successes, failures, and latency times.

### **Advantages:**

* **Proactive Monitoring:** Helps developers detect system issues early.
* **Observability:** Provides detailed insights into the circuit breaker’s performance and database health.

## **Error Handling Strategy**

### **Purpose:**

Comprehensive error handling ensures that the API can gracefully handle different types of errors.

### **Details:**

* **Validation Errors:** Handled separately to prevent unnecessary retries.
* **Transient Errors:** Automatically retried up to 3 times.
* **Fallback Response:** For unrecoverable errors, a clear and user-friendly error message is returned to the client.

### **Advantages:**

* **User Experience:** Provides meaningful error messages to clients.
* **System Robustness:** Ensures stability by gracefully managing failures.

## **Scalability Considerations**

### **Components Supporting Scalability:**

* **Rate Limiting:** Prevents overloading of API resources under high traffic.
* **Circuit Breaker:** Protects against downstream system failures.
* **MongoDB Transactions:** Ensures consistent data operations in multi-service setups.

### **Benefits:**

* **Load Management:** Controls system behavior under high traffic.
* **Error Isolation:** Minimizes the impact of localized failures on the overall system.

# **User Management API**

## **Base URL:** [**http://localhost:5000/api/users**](http://localhost:5000/api/users)

## **Schema:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field Name** | **Data Type** | **Constraints** | **Description** |
| \_id | ObjectId | Automatically generated by MongoDB | Unique identifier for each user. |
| name | String | required: true | Full name of the user. |
| email | String | required: true, unique: true | Email address of the user. Must be unique. |
| password | String | required: true | User's hashed password (not stored in plain text). |
| role | String | enum: ['admin', 'employee', 'citizen'], default: 'citizen' | User's role in the system. Default is citizen. |
| createdAt | Date | Automatically generated by Mongoose | Timestamp when the user was created. |
| updatedAt | Date | Automatically updated by Mongoose | Timestamp when the user was last modified. |

## **API Endpoints**

### **1. Get All Users**

**Description**: Retrieve all users in the database.

* **Endpoint**: GET /api/users
* **Request Body**: None
* **Response**:
  + **200 OK**:

[  
 {  
 "\_id": "63c1a2f5e6b2f72a5d98c1b4",  
 "name": "John Doe",  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)",  
 "role": "citizen",  
 "createdAt": "2024-01-01T12:00:00.000Z",  
 "updatedAt": "2024-01-01T12:00:00.000Z"  
 }  
]

### **2. Get User by ID**

**Description**: Retrieve a specific user by their unique ID.

* **Endpoint**: GET /api/users/:id
* **Request Parameters**:
  + id (required): MongoDB Object ID of the user.
* **Response**:
  + **200 OK**:

{  
 "\_id": "63c1a2f5e6b2f72a5d98c1b4",  
 "name": "John Doe",  
 "email": "[johndoe@example.com](mailto:johndoe@example.com)",  
 "role": "citizen",  
 "createdAt": "2024-01-01T12:00:00.000Z",  
 "updatedAt": "2024-01-01T12:00:00.000Z"  
}

* + **404 Not Found**:

{ "message": "User not found" }

### **3. Create a New User**

**Description**: Add a new user to the database.

* **Endpoint**: POST /api/users
* **Request Body**:

{  
 "name": "Jane Doe",  
 "email": "[janedoe@example.com](mailto:janedoe@example.com)",  
 "password": "securepassword",  
 "role": "admin"  
}

* **Response**:
  + **201 Created**:

{  
 "\_id": "63c1a2f5e6b2f72a5d98c1b5",  
 "name": "Jane Doe",  
 "email": "[janedoe@example.com](mailto:janedoe@example.com)",  
 "role": "admin",  
 "createdAt": "2024-01-01T12:05:00.000Z",  
 "updatedAt": "2024-01-01T12:05:00.000Z"  
}

* + **400 Bad Request**:

{ "message": "Validation error: email already exists" }

### **4. Update User by ID**

**Description**: Modify details of an existing user.

* **Endpoint**: PUT /api/users/:id
* **Request Parameters**:
  + id (required): MongoDB Object ID of the user.
* **Request Body**:

{  
 "name": "Jane Doe Updated",  
 "role": "employee"  
}

* **Response**:
  + **200 OK**:

{  
 "\_id": "63c1a2f5e6b2f72a5d98c1b5",  
 "name": "Jane Doe Updated",  
 "email": "[janedoe@example.com](mailto:janedoe@example.com)",  
 "role": "employee",  
 "createdAt": "2024-01-01T12:05:00.000Z",  
 "updatedAt": "2024-01-01T12:10:00.000Z"  
}

* + **404 Not Found**:

{ "message": "User not found" }

### **5. Delete User by ID**

**Description**: Remove a user from the database.

* **Endpoint**: DELETE /api/users/:id
* **Request Parameters**:
  + id (required): MongoDB Object ID of the user.
* **Response**:
  + **200 OK**:

{ "message": "User deleted successfully" }

* + **404 Not Found**:

{ "message": "User not found" }

## **Error Handling**

All API responses include descriptive error messages for invalid requests:

* **400 Bad Request**: Validation errors (e.g., missing or invalid fields).
* **404 Not Found**: Resource not found (e.g., invalid user ID).
* **500 Internal Server Error**: General server errors.

## **Middleware**

### **CORS**

Enables cross-origin requests to ensure compatibility with front-end clients.

### **JSON Parsing**

Parses incoming JSON requests.

### **Architecture Highlights**

#### **Key Components**

1. **Express Framework**:
   1. Used to create the server and handle routing.
   2. Middleware like cors and express.json enhance functionality.
2. **MongoDB with Mongoose**:
   1. MongoDB serves as the database.
   2. Mongoose provides a schema-based solution to model data, ensuring data consistency and validation.
3. **Routes**:
   1. Organized into modular route files like userRoutes for better separation and code organization.
4. **Configuration Management**:
   1. Environment variables (via process.env) allow dynamic configuration (e.g., PORT, MONGO\_URI).

#### **Features and Functionalities**

1. **CRUD Operations**:
   1. Full CRUD support for managing user entities (Create, Read, Update, Delete).
2. **Role Management**:
   1. Roles like admin, employee, and citizen are enforced using enumerations in the schema.
3. **Timestamp Tracking**:
   1. Automatic tracking of createdAt and updatedAt timestamps via Mongoose.

#### **Middleware**

1. **CORS**:
   1. Enables secure API access across different domains.
2. **JSON Parsing**:
   1. Automatically parses incoming JSON request bodies.

#### **Scalability Considerations**

1. **Horizontal Scaling**:
   1. The use of MongoDB ensures horizontal scalability by adding more database nodes.
2. **API Extensions**:
   1. New routes can be added easily due to the modular design.
3. **Middleware Extensibility**:
   1. Additional middleware like authentication or logging can be integrated seamlessly.

#### **Security Best Practices**

1. **Password Handling**:
   1. Although not explicitly shown, password fields should ideally be hashed using libraries like bcrypt.
2. **Environment Variables**:
   1. Sensitive configurations (e.g., database URIs) are stored in environment variables.

#### **High-Level Data Flow**

1. **Request Handling**:
   1. Clients make HTTP requests to endpoints (e.g., /api/users).
2. **Routing**:
   1. Requests are routed to userRoutes where specific logic is executed.
3. **Data Operations**:
   1. Mongoose models handle CRUD operations against MongoDB.
4. **Response**:
   1. Results are sent back to the client in JSON format.

## **Email Service API Documentation**

## **Base URL:** The API is hosted on http://localhost:5005/send-email.

### **POST /send-email**

#### **Description:** Sends an email using the Mailjet API.

#### **Request Headers**

* Content-Type: application/json

#### **Request Body**

The API expects the request body in JSON format containing the following fields:

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Type** | **Required** | **Description** |
| to | String | Yes | Recipient's email address. |
| subject | String | Yes | Subject of the email. |
| data | String/Object | Yes | Email body content (plain text or JSON object). |

#### **Example Request**

POST /send-email HTTP/1.1  
Content-Type: application/json  
  
{  
 "to": "[recipient@example.com](mailto:recipient@example.com)",  
 "subject": "Welcome to Our Service",  
 "data": "Hello, thank you for joining our service!"  
}

#### **Environment Variables:** The service uses the following environment variables for configuration:

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| MAILJET\_API\_KEY | Your Mailjet API key. |
| MAILJET\_API\_SECRET | Your Mailjet API secret. |
| MAILJET\_SENDER\_EMAIL | Sender's email address (used as the "From" email). |
| PORT | Port for the server to listen on (default: 5005). |

#### **Example Environment File (.env)**

MAILJET\_API\_KEY=your\_mailjet\_api\_key  
MAILJET\_API\_SECRET=your\_mailjet\_api\_secret  
[MAILJET\_SENDER\_EMAIL=sender@example.com](mailto:MAILJET_SENDER_EMAIL=sender@example.com)PORT=5005

#### **Success Response**

|  |  |
| --- | --- |
| **Status Code** | **Description** |
| 200 | Email sent successfully. |

Example:

{  
 "message": "Email sent successfully"  
}

#### **Error Responses**

|  |  |
| --- | --- |
| **Status Code** | **Description** |
| 400 | Missing required fields in request. |
| 500 | Failed to send the email. |

Examples:

* Missing Fields:

{  
 "error": "Missing required fields"  
}

* Sending Error:

{  
 "error": "Failed to send email"  
}

### **Architectural Highlights** **Middleware Integration**

* **CORS Middleware**:
  + Enables cross-origin resource sharing, allowing the API to be consumed by clients from different domains.
* **Body-Parser Middleware**:
  + Parses incoming JSON payloads and makes them accessible via req.body.

**Why it matters**: Middleware ensures smooth communication between clients and the server, handling common tasks like parsing and cross-origin requests.

### **Integration with External Service (Mailjet API)**

* The service integrates with Mailjet, a third-party email API provider, using the **node-mailjet** SDK.
* Mailjet handles:
  + Email delivery.
  + Management of sender and recipient details.
  + Support for structured email content (e.g., text or JSON).

**Why it matters**: Leveraging an external service offloads the complexity of email delivery, ensuring **scalability** and high delivery rates.

### **Asynchronous Programming**

* The **async/await** pattern is used for handling asynchronous calls to Mailjet.
* Ensures non-blocking operations, allowing the server to handle multiple requests efficiently.

**Why it matters**: Asynchronous programming improves performance and responsiveness, especially in I/O-intensive applications.

### **Error Logging and Handling**

* Logs errors to the console for debugging purposes.
* Provides meaningful error messages to clients (Failed to send email, Missing required fields).

**Why it matters**: Proper error handling ensures the application is robust and user-friendly, even in failure scenarios.

### **Security Considerations**

* **Environment Variables**:
  + Hides sensitive API keys and credentials from the codebase.
* **Input Validation**:
  + Prevents invalid or incomplete requests from being processed.
* **CORS**:
  + Restricts which clients can access the service, mitigating security risks like cross-origin attacks.

**Why it matters**: Security best practices reduce vulnerabilities and protect sensitive information.

### **Scalability**

* **Stateless Architecture**:
  + The API doesn’t maintain state between requests, making it easy to scale horizontally by adding more instances.
* **Third-Party Email Service**:
  + Mailjet can handle increased email volume, ensuring scalability without modifying the codebase.

**Why it matters**: The architecture is designed to handle growth in both traffic and functionality.

## **Geolocation Service API Documentation**

## **Base URL:** The API is hosted on http://localhost:5002/routes/locationRoutes

### **Endpoints**

#### **1. GET /api/location**

Fetches detailed location information for the provided latitude and longitude using the OpenCage Geocoding API.

##### **Request Parameters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **Type** | **Required** | **Description** |
| latitude | String | Yes | Latitude of the location. |
| longitude | String | Yes | Longitude of the location. |

##### **Request Example**

GET /api/location?latitude=48.8566&longitude=2.3522

|  |  |
| --- | --- |
| **Status Code** | **Description** |
| 200 | Location data fetched successfully. |

**Response Body Example**:

{  
 "address": "5 Rue des Écoles",  
 "city": "Paris, Île-de-France",  
 "postalCode": "75005",  
 "country": "France",  
 "coordinates": ["2.3522", "48.8566"]  
}

|  |  |
| --- | --- |
| **Status Code** | **Description** |
| 400 | Missing or invalid latitude or longitude parameters. |
| 404 | No location data found for the provided coordinates. |
| 429 | Too many requests (rate limit exceeded). |
| 500 | Internal error (e.g., OpenCage API failure). |

**Error Example**:

{  
 "message": "Location not found for the provided coordinates"  
}

#### **2. GET /api/health**

Provides health status information about the service, including the state of the circuit breaker.

|  |  |
| --- | --- |
| **Status Code** | **Description** |
| 200 | Health data returned. |

**Response Body Example**:

{  
 "circuitBreaker": {  
 "open": false,  
 "closed": true,  
 "stats": {  
 "totalSuccesses": 5,  
 "totalFailures": 2,  
 "totalRequests": 7  
 }  
 }  
}

### **Features**

#### **Rate Limiting**

* **Library**: express-rate-limit.
* **Policy**: Limits to **100 requests per 15 minutes per IP**.
* **Response on Limit Exceeding**:

{  
 "message": "Too many requests, please try again later."  
}

#### **Circuit Breaker for Fault Tolerance**

* **Library**: opossum.
* Protects the service from cascading failures due to issues with the OpenCage API.
* Circuit breaker configuration:
  + Retries failed requests up to **3 times** with exponential backoff.
  + Opens the circuit when **50% of requests fail**.
  + Automatically attempts to reset after **20 seconds**.
  + Timeout for individual operations: **10 seconds**.

**Event Logs**:

* **Open**: Logs when the circuit is open (calls are stopped).
* **Half-Open**: Logs when the circuit is testing recovery.
* **Close**: Logs when the circuit is closed (normal operation resumes).

#### **Input Validation**

* Validates that latitude and longitude are provided.
* Returns 400 Bad Request for invalid or missing inputs.

#### **External API Integration**

* Uses the **OpenCage Geocoding API** to fetch location data.
* Requires the following **environment variable**:
  + OPENCAGE\_API\_KEY: API key for OpenCage services.

#### **Robust Error Handling**

* Logs errors to the console for debugging.
* Categorizes errors into:
  + **Validation Errors**: Handled without retries.
  + **Transient Errors**: Retried based on the circuit breaker logic.

#### **Monitoring**

The /api/health endpoint provides:

* Circuit breaker state (open or closed).
* Statistics about service usage (e.g., total successes, failures, and requests).

### **Public Information and Announcements Service**

## **Base URL:** <http://localhost:5004/api/announcements>**Schema:**

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Constraints** |
| \_id | ObjectId | Primary Key, auto-generated by MongoDB |
| title | String | Required |
| content | String | Required |
| type | String | Enum: ['maintenance', 'policy', 'safety'], Required |
| startDate | Date | Optional |
| endDate | Date | Optional |
| createdAt | Date | Default: Date.now |
| updatedAt | Date | Default: Date.now, updates on modification |

## **Endpoints**

### **1. Create an Announcement**

**POST** /

* **Description:** Create a new announcement.
* **Authorization:** Required.
* **Request Headers:**
  + Authorization: Bearer <token>
* **Request Body:** (JSON)

{  
 "title": "Announcement Title",  
 "content": "Detailed content of the announcement",  
 "type": "Event/Alert/Info",  
 "startDate": "YYYY-MM-DDTHH:mm:ssZ",  
 "endDate": "YYYY-MM-DDTHH:mm:ssZ"  
}

* **Response:**
  + **201 Created** (Success)

{  
 "\_id": "announcementId",  
 "title": "Announcement Title",  
 "content": "Detailed content of the announcement",  
 "type": "Event",  
 "startDate": "2024-12-05T00:00:00.000Z",  
 "endDate": "2024-12-10T00:00:00.000Z",  
 "createdAt": "2024-12-05T12:34:56.000Z",  
 "updatedAt": "2024-12-05T12:34:56.000Z"  
}

* + **503 Service Unavailable**

{ "message": "Service temporarily unavailable. Please try again later." }

### **2. Get All Announcements**

**GET** /

* **Description:** Retrieve all announcements.
* **Authorization:** Not required.
* **Response:**
  + **200 OK** (Success)

[  
 {  
 "\_id": "announcementId",  
 "title": "Announcement Title",  
 "content": "Detailed content",  
 "type": "Event",  
 "startDate": "2024-12-05T00:00:00.000Z",  
 "endDate": "2024-12-10T00:00:00.000Z",  
 "createdAt": "2024-12-05T12:34:56.000Z",  
 "updatedAt": "2024-12-05T12:34:56.000Z"  
 }  
]

* + **503 Service Unavailable**

{ "message": "Service temporarily unavailable. Please try again later." }

### **3. Update an Announcement**

**PUT** /:announcementId

* **Description:** Update an existing announcement.
* **Authorization:** Required.
* **Request Headers:**
  + Authorization: Bearer <token>
* **Request Body:** (JSON)

{  
 "title": "Updated Title",  
 "content": "Updated content",  
 "type": "Alert",  
 "startDate": "YYYY-MM-DDTHH:mm:ssZ",  
 "endDate": "YYYY-MM-DDTHH:mm:ssZ"  
}

* **Response:**
  + **200 OK** (Success)

{  
 "\_id": "announcementId",  
 "title": "Updated Title",  
 "content": "Updated content",  
 "type": "Alert",  
 "startDate": "2024-12-05T00:00:00.000Z",  
 "endDate": "2024-12-15T00:00:00.000Z",  
 "createdAt": "2024-12-05T12:34:56.000Z",  
 "updatedAt": "2024-12-06T12:34:56.000Z"  
}

* + **404 Not Found**

{ "message": "Announcement not found" }

### **4. Delete an Announcement**

**DELETE** /:announcementId

* **Description:** Delete an existing announcement.
* **Authorization:** Required.
* **Request Headers:**
  + Authorization: Bearer <token>
* **Response:**
  + **200 OK** (Success)

{ "message": "Announcement deleted" }

* + **404 Not Found**

{ "message": "Announcement not found" }

### **5. Health Check**

**GET** /health

* **Description:** Check the health of the service.
* **Authorization:** Not required.
* **Response:**
  + **200 OK**

{  
 "circuitBreaker": {  
 "open": false,  
 "closed": true,  
 "stats": {  
 "failures": 0,  
 "successes": 10,  
 "timeouts": 0,  
 "rejects": 0,  
 "fires": 10  
 }  
 }  
}

## **Middleware**

### **Rate Limiting**

* Each IP is limited to 100 requests per 15 minutes.
* Rate limit violation response:

{ "message": "Too many requests from this IP, please try again after 15 minutes." }

### **Authentication**

* Protected routes require a valid JWT token in the Authorization header.

### **Architecture Highlights**

### **Scalability and Performance**

* **Rate Limiting:**
  + Implements express-rate-limit to throttle excessive API calls and protect against denial-of-service (DoS) attacks.
  + Each IP is restricted to 100 requests per 15-minute window.
* **Asynchronous Operations:**
  + Operations are asynchronous and non-blocking, improving request handling efficiency and supporting scalability.
* **Worker Threads:**
  + Utilizes worker\_threads to offload CPU-intensive tasks (if integrated in unprovided parts of the system).

### **Robust Error Handling**

* **Retry Logic:**
  + Database operations use async-retry to handle transient issues, ensuring fault tolerance.
* **Graceful Degradation:**
  + When the circuit breaker is open, the system returns a **503 Service Unavailable** response, signaling temporary downtime without crashing.
* **Error Classification:**
  + Differentiates between transient and permanent errors (e.g., validation errors) for appropriate handling.

### **Security**

* **Authentication Middleware:**
  + Protected routes require a valid JWT token in the Authorization header.
  + Ensures only authenticated users can create, update, or delete announcements.
* **Cross-Origin Resource Sharing (CORS):**
  + Configured via cors middleware to prevent unauthorized cross-origin requests.
* **Input Validation:**
  + Ensures input data integrity and prevents invalid data from being processed.

### **Observability**

* **Health Check Endpoint:**
  + /health endpoint provides real-time insights into the circuit breaker’s state and operational metrics (e.g., success/failure counts).
* **Logging and Alerts:**
  + Logs circuit breaker state transitions (open, half-open, closed) for easier debugging and monitoring.

### **Scalability Readiness**

* **Horizontal Scalability:**
  + The stateless architecture (e.g., JWT for authentication) enables easy horizontal scaling.
* **Worker Thread Offloading:**
  + Support for background processing using worker threads facilitates high-performance scenarios.

## **Utility Reporting**

### **Base URL:** <http://localhost:5001/api/reports>

## **Authentication**

Most routes are protected and require an authentication middleware (authMiddleware) to ensure only authenticated users can access them. Some administrative routes also require additional middleware (isAdmin).

## **Schema**

|  |  |  |  |
| --- | --- | --- | --- |
| **Field** | **Type** | **Required** | **Description/Details** |
| userId | ObjectId (ref: User) | Yes | The ID of the user who reported the issue. It references the User model. |
| type | String | Yes | Type of report (e.g., 'water', 'electricity', 'waste'). |
| description | String | Yes | A description of the issue being reported. |
| location | Object (subdocument) | Yes | Contains details about the location where the issue was observed. |
| location.address | String | Yes | Street address of the location. |
| location.city | String | Yes | City where the issue was observed. |
| location.postalCode | String | Yes | Postal code of the location. |
| location.country | String | Yes | Country where the location is situated. |
| location.coordinates | [Number] | Yes | Geo-coordinates (longitude, latitude) in an array format, used for location-based queries. |
| location.type | String | Yes | The type of the GeoJSON object (must be 'Point' for a single location point). |
| status | String | No | The current status of the report ('pending', 'in-progress', or 'resolved'). Default is 'pending'. |
| assignedEmployee | ObjectId (ref: User) | No | The employee assigned to resolve the issue, referenced from the User model. Defaults to null. |
| createdAt | Date | No | Automatically generated timestamp when the report was created. |
| updatedAt | Date | No | Automatically generated timestamp when the report was last updated. |

### **Notes:**

* The **location** field is an object that contains nested fields for address, city, postal code, country, coordinates, and type. These fields are required to create a valid location object, and the **coordinates** should follow GeoJSON conventions (i.e., [longitude, latitude]).
* **status** is optional but defaults to 'pending'. It tracks the current state of the report.
* **assignedEmployee** is also optional but can be assigned a user ID if an employee is designated to handle the report.
* The **timestamps** feature will automatically create createdAt and updatedAt fields.

## **Endpoints**

### **Create a Report**

* **Method**: POST
* **Endpoint**: /
* **Middleware**:
  + authMiddleware (User must be authenticated)
  + createReportLimiter (Rate limit: 100 requests per 15 minutes)

{  
 "type": "String",  
 "description": "String",  
 "location": {  
 "address": "String",  
 "city": "String",  
 "postalCode": "String",  
 "country": "String",  
 "coordinates": [longitude, latitude]  
 }  
}

* **Response**:
  + 201 Created: The report was created successfully.
  + 500 Internal Server Error: Server error.
* **Rate Limiting**: Users are limited to 100 requests per 15 minutes.

### **Get User Reports**

* **Method**: GET
* **Endpoint**: /
* **Middleware**: authMiddleware
* **Response**:
  + 200 OK: Returns all reports created by the authenticated user.
  + 500 Internal Server Error: Server error.

### **Get All Reports (Admin)**

* **Method**: GET
* **Endpoint**: /all
* **Middleware**:
  + authMiddleware
  + isAdmin (Only admins can access this route)
* **Response**:
  + 200 OK: Returns all reports.
  + 500 Internal Server Error: Server error.

### **Update Report Status**

* **Method**: PUT
* **Endpoint**: /status
* **Middleware**: authMiddleware

{  
 "reportId": "String",  
 "status": "String"  
}

* **Response**:
  + 200 OK: Returns the updated report.
  + 404 Not Found: Report with the given ID was not found.
  + 500 Internal Server Error: Server error.

### **Assign Report to Employee**

* **Method**: PUT
* **Endpoint**: /assign
* **Middleware**: authMiddleware

{  
 "reportId": "String",  
 "employeeId": "String"  
}

* **Response**:
  + 200 OK: Returns the updated report.
  + 404 Not Found: Report with the given ID was not found.
  + 500 Internal Server Error: Server error.

### **Get Reports Assigned to Employee**

* **Method**: GET
* **Endpoint**: /assigned
* **Middleware**: authMiddleware
* **Response**:
  + 200 OK: Returns all reports assigned to the authenticated employee.
  + 500 Internal Server Error: Server error.

### **Delete Report**

* **Method**: DELETE
* **Endpoint**: /:reportId
* **Middleware**: authMiddleware
* **Response**:
  + 204 No Content: Report deleted successfully.
  + 404 Not Found: Report with the given ID was not found.
  + 500 Internal Server Error: Server error.

### **Health Check**

* **Method**: GET
* **Endpoint**: /health
* **Response**:
  + 200 OK: Returns the status of the circuit breaker.

{  
 "circuitBreaker": {  
 "open": "Boolean",  
 "closed": "Boolean",  
 "stats": "Object"  
 }  
}

## **Error Responses**

* **Validation Errors**: Occur when the provided input is invalid or missing.
* **Server Errors**: General server-related errors, including database issues.

## **Rate Limiting:** The createReport endpoint is rate-limited using express-rate-limit:

* **Window**: 15 minutes
* **Max Requests**: 100
* **Response on Limit**: 429 Too Many Requests

## **Middleware**

* **authMiddleware**: Ensures the user is authenticated.
* **isAdmin**: Restricts access to admin-specific routes.
* **createReportLimiter**: Limits requests for creating reports.

### **Architectural Highlights**

### **Error Resilience**

* **Circuit Breaker Pattern**:
  + Implements the opossum library to safeguard against cascading failures in the database layer.
  + Automatically retries transient errors using async-retry with exponential backoff.
  + Circuit states (open, half-open, closed) manage failure handling effectively, enhancing reliability.
* **Rate Limiting**: Prevents abuse by limiting the number of requests a client can make to sensitive endpoints (createReport).

### **Middleware-Driven Functionality**

* **Authentication and Authorization**:
  + authMiddleware ensures only authenticated users can access protected routes.
  + isAdmin middleware provides role-based access control for administrative endpoints.
* **Rate Limiting**: Uses express-rate-limit to restrict the frequency of API calls to certain routes.
* **Validation and Error Handling**: Handles non-recoverable errors gracefully while retrying transient errors for better user experience.

### **Scalability**

* **Horizontal Scaling**: The application uses stateless design principles, making it easy to deploy multiple instances behind a load balancer.
* **API Gateway Ready**: Clear RESTful API structure allows seamless integration with API gateways or external services.
* **GeoJSON Support**: Location data is stored in GeoJSON format, enabling advanced geographical queries and integration with GIS tools.

### **Security**

* **Authentication**: Requires user authentication for most endpoints, protecting sensitive operations like report creation, status updates, and deletions.
* **Role-Based Access**: Admin-only routes ensure restricted access to operations like fetching all reports.
* **Input Validation**: Validates user-provided data like location and report details to prevent malformed requests.

### **Performance Optimization**

* **Asynchronous Operations**: Uses async/await and promises to handle I/O operations efficiently.
* **Rate Limiting**: Prevents overloading of server resources by throttling excessive requests.
* **Efficient Query Design**: Filters and paginates report data to minimize database query overhead.

### **Health Monitoring**

* **Health Check Endpoint**:
  + /health endpoint provides insights into the circuit breaker status.
  + Allows for real-time monitoring of the application’s resilience and failure handling capabilities.

### **Consuming microservices in Frontend**

#### **Base URLs**

* **Authentication Service**: ${process.env.REACT\_APP\_API\_URL}
* **Reports Service**: <http://localhost:5001/api/reports>
* **Location Service**: <http://localhost:5002/api/location>
* **Announcements Service**: <http://localhost:5004/api/announcements>
* **Email Service**: <http://localhost:5005/send-email>
* **History Service**: <http://localhost:5006/api/history>

### **Authentication**

1. **Register User**
   1. **URL**: POST /register
   2. **Payload**: { username, email, password }
   3. **Response**: User object
2. **Login User**
   1. **URL**: POST /login
   2. **Payload**: { email, password }
   3. **Response**: JWT token
3. **Get User by ID**
   1. **URL**: GET /:userId
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Response**: User object
4. **Update User**
   1. **URL**: PUT /:userId
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Payload**: Updated user data
   4. **Response**: Updated user object
5. **Delete User**
   1. **URL**: DELETE /:userId
   2. **Headers**: { Authorization: Bearer <token> }

### **Reports**

1. **Create Report**
   1. **URL**: POST /api/reports
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Payload**: Report data
   4. **Response**: Created report
2. **Fetch Reports**
   1. **URL**: GET /api/reports
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Response**: List of reports
3. **Delete Report**
   1. **URL**: DELETE /api/reports/:reportId
   2. **Headers**: { Authorization: Bearer <token> }
4. **Fetch All Reports (Admin)**
   1. **URL**: GET /api/reports/all
   2. **Headers**: { Authorization: Bearer <token> }
5. **Assign Report**
   1. **URL**: PUT /api/reports/assign
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Payload**: { reportId, employeeId }

### **Announcements**

1. **Fetch Announcements**
   1. **URL**: GET /api/announcements
   2. **Response**: List of announcements
2. **Create Announcement**
   1. **URL**: POST /api/announcements
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Payload**: Announcement data
   4. **Response**: Created announcement
3. **Update Announcement**
   1. **URL**: PUT /api/announcements/:announcementId
   2. **Headers**: { Authorization: Bearer <token> }
   3. **Payload**: Updated announcement data
4. **Delete Announcement**
   1. **URL**: DELETE /api/announcements/:announcementId
   2. **Headers**: { Authorization: Bearer <token> }

### **Other Services**

1. **Location**
   1. **Fetch Location**
      1. **URL**: GET /api/location?latitude=<lat>&longitude=<lon>
      2. **Response**: Location data
2. **Email**
   1. **Send Email**
      1. **URL**: POST /send-email
      2. **Payload**: { to, subject, data }
      3. **Response**: Email sent status
3. **History**
   1. **Create History**
      1. **URL**: POST /api/history
      2. **Headers**: { Authorization: Bearer <token> }
      3. **Payload**: History data
      4. **Response**: Created history
   2. **Fetch All History (Admin)**
      1. **URL**: GET /api/history/all
      2. **Headers**: { Authorization: Bearer <token> }
   3. **Fetch History by User ID**
      1. **URL**: GET /api/history/user/:userId
      2. **Headers**: { Authorization: Bearer <token> }