<back_end/>

User API

The User API provides endpoints to manage user authentication and registration.

Authentication

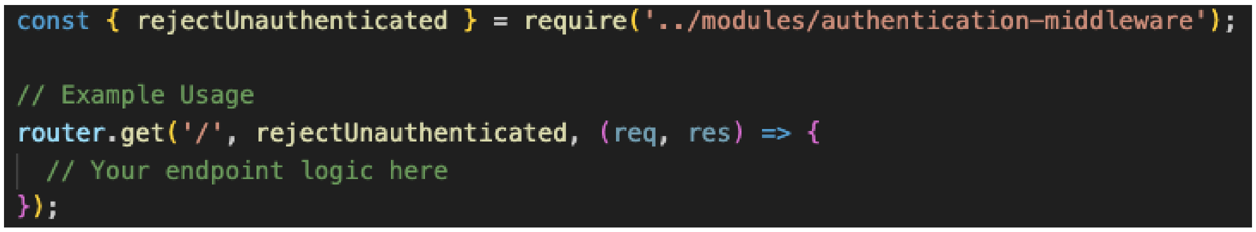
All endpoints in this API require authentication except for the registration endpoint.

Authentication Middleware

The API uses a custom authentication middleware called `rejectUnauthenticated` to ensure that users are authenticated before accessing certain routes. This middleware checks if the user is logged in (authenticated) using cookies and session management.

If a user is not authenticated, the middleware will respond with a `403 Forbidden` status code, indicating that the user does not have access to the requested resource.

Middleware Usage



```
const { rejectUnauthenticated } = require('../modules/authentication-middleware');

// Example Usage
router.get('/', rejectUnauthenticated, (req, res) => {
  // Your endpoint logic here
});
```

<u>Endpoints</u>

`GET `/api/user/``

- Description: Retrieves user information for the authenticated user.
- Authentication: Required (authenticated users only).
- Method: `GET`
- Response Status Code: `200 OK` on success, `403 Forbidden` if the user is not authenticated.
- Response Body:
  - JSON object representing the user's information.

`POST `/api/user/register``

- Description: Registers a new user.
- Authentication: Not required.
- Method: `POST`
- Request Body:
  - `username`: String (required) - The username for the new user.
  - `password`: String (required) - The password for the new user.
- Response Status Code: `201 Created` on successful registration, `500 Internal Server Error` on failure.

`POST `/api/user/login``

- Description: Authenticates the user and creates a session.
- Authentication: Not required (authentication is handled by Passport local strategy).
- Method: `POST`
- Request Body:
  - `username`: String (required) - The username of the user.
  - `password`: String (required) - The password of the user.
- Response Status Code: `200 OK` on successful authentication, `404 Not Found` if authentication fails.

POST `/api/user/logout`

- Description: Logs out the user and clears the session.
- Authentication: Required (authenticated users only).
- Method: `POST`
- Response Status Code: `200 OK` on successful logout.

Encryption Module

The API uses the `bcryptjs` library to securely encrypt and compare user passwords before storing them in the database.

Encryption Functions

1. `encryptPassword(password: string): string`

- Description: Encrypts the provided password using a randomly generated salt.
- Parameters:
  - `password`: String (required) - The plain text password to be encrypted.
- Returns:
  - `encryptedPassword`: String - The encrypted password.

2. `comparePassword(candidatePassword: string, storedPassword: string): boolean`

- Description: Compares a candidate password (user input) with the stored encrypted password.
- Parameters:
  - `candidatePassword`: String (required) - The plain text candidate password.
  - `storedPassword`: String (required) - The stored encrypted password.

- Returns:
  - `isMatch`: Boolean - `true` if the candidate password matches the stored password, `false` otherwise.

Session Management

The API uses the `cookie-session` middleware to manage user sessions. This allows the user to stay logged in after providing their credentials once.

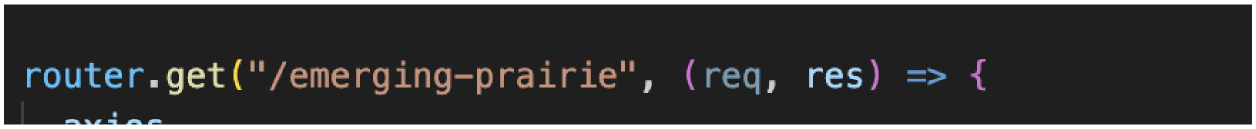
Session Configuration

The session is configured with the following options:

- `secret`: The session secret used to sign the session ID cookie. This should be set in the `.env` file.
- `key`: The name of the req.variable for session data (optional, default is 'user').
- `resave`: Set to `false` to avoid session resaving on every request.
- `saveUninitialized`: Set to `false` to prevent creating sessions for unauthenticated users.
- `maxAge`: The maximum age of the session, set to 7 days (in milliseconds).
- `secure`: Set to `false` as the API does not require secure (HTTPS) connections in this context.

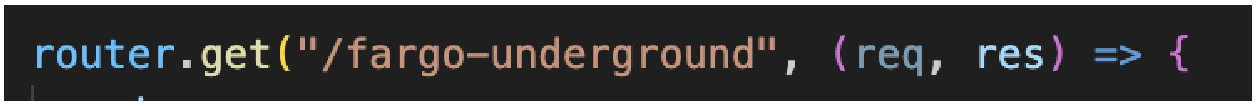
Note: Ensure that you set a secure and unique `SERVER_SESSION_SECRET` in the `.env` file for production use.

Calendar.router.js



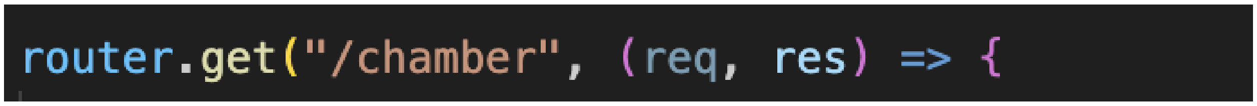
```
router.get("/emerging-prairie", (req, res) => {
```

FULL URL : /api/calendar/emerging-prairie
Pressing the Emerging Prairie button in the calendar component launches the GET request to the designated url. The cheerio technology scrapes the public calendars title, start time, end time, combined date span, location, and description. The dates pulled are run through a parsing function for formatting and then conversion into ISO8601 TimeStamps After parsing, all elements are then consolidated into objects which are put into an array that is sent to the EP_Reducer.



```
router.get("/fargo-underground", (req, res) => {
```

FULL URL : /api/calendar/fargo-underground
Pressing the Fargo Underground button in the calendar component launches the GET request to the designated url. The cheerio technology scrapes the public calendars title, start time, end time, combined date span, location, and description. The dates pulled are run through a parsing function for formatting and then conversion into ISO8601 TimeStamps After parsing, all elements are then consolidated into objects which are put into an array that is sent to the FU_Reducer.

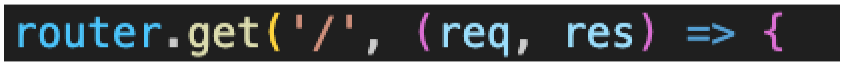


```
router.get("/chamber", (req, res) => {
```

FULL URL : /api/calendar/chamber
Pressing the Chamber of Commerce button in the calendar component launches the GET request to the designated url. The cheerio technology scrapes the public calendars title, start time, end time, and description. This is the only calendar that does not run a parse function over its elements. All elements are then consolidated into objects which are put into an array that is sent to the chamber_Reducer.

WARNING ON ALL CALENDAR ROUTES
There are some significant vulnerabilities in this calendar to be aware of. Since the button press is what directly pulls the information, it will fail or break if that institution's calendar or webpage is down or reworked in the future (StartlingLine.FM was launched August 2023).

Search API



```
router.get('/', (req, res) => {
```

/server/routes/search.router.js
FULL URL : /api/search/{?category=int&stage=int&text=string}

The Search API allows you to perform searches based on specific criteria and retrieve search results, utilizing a text similarity algorithm for more accurate matches.

Fetch Search [GET]

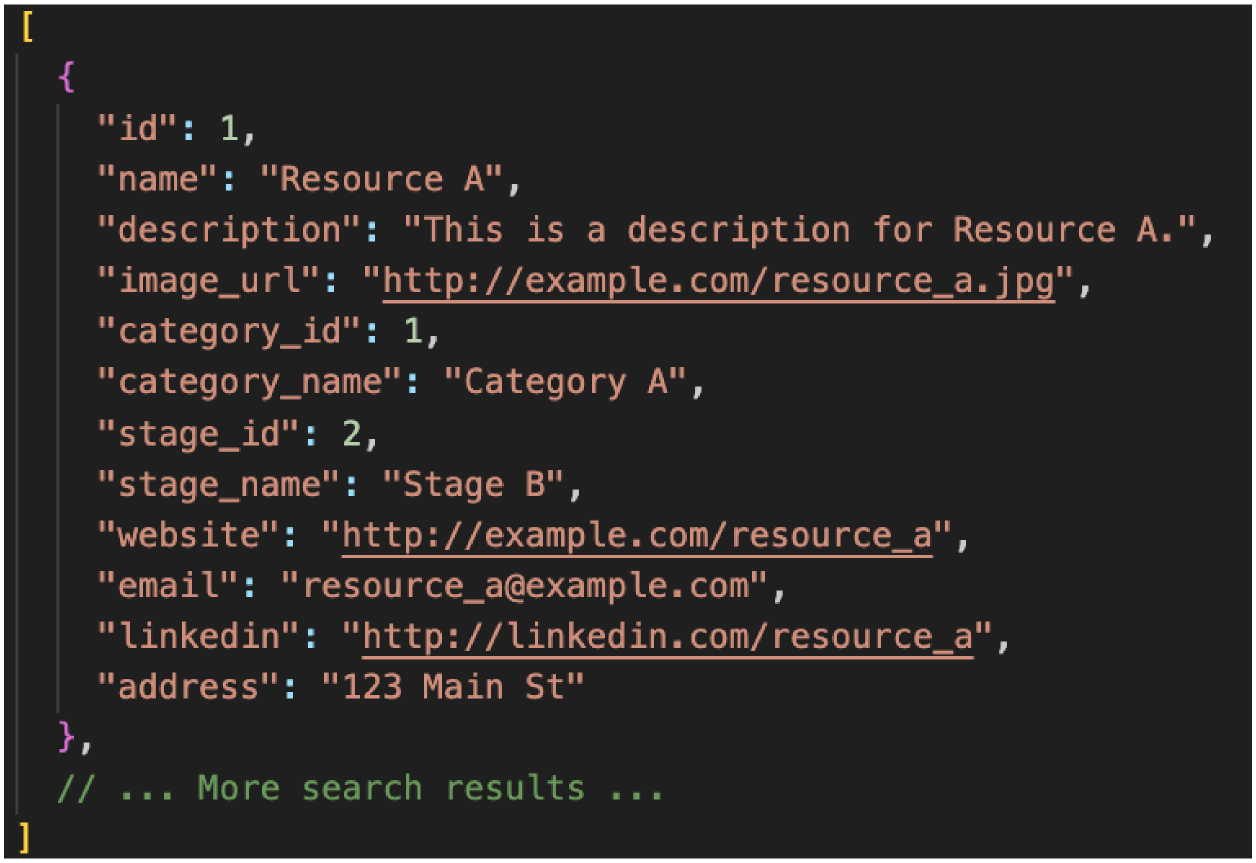
This endpoint is used to fetch search results based on the provided query parameters and utilizes a text similarity algorithm to enhance search accuracy.

Request

- Method: GET
- Query Parameters:
  - `category` (optional): Specifies the category to filter the search results.
  - `stage` (optional): Specifies the stage to filter the search results.
  - `text` (optional): Specifies the text to search for in the resource names and descriptions.

Response

- The API responds with a JSON array containing the search results.
- Example response



```
[
  {
    "id": 1,
    "name": "Resource A",
    "description": "This is a description for Resource A.",
    "image_url": "http://example.com/resource_a.jpg",
    "category_id": 1,
    "category_name": "Category A",
    "stage_id": 2,
    "stage_name": "Stage B",
    "website": "http://example.com/resource_a",
    "email": "resource_a@example.com",
    "linkedin": "http://linkedin.com/resource_a",
    "address": "123 Main St"
  },
  // ... More search results ...
]
```

Text Similarity Algorithm

The search API utilizes a text similarity algorithm called `` `WORD_SIMILARITY` ``, which takes the following column titles as the first argument and the search query as the second argument:

- `` `resource.description` ``
- `` `resource.name` ``
- `` `stage.name` ``
- `` `category.name` ``
- `` `website` ``
- `` `email` ``
- `` `address` ``
- `` `linkedin` ``

The algorithm returns search results according to a similarity threshold between 0 and 1. A similarity threshold of 0.4 is used in this API to ensure relevant matches.

Error Handling

If an error occurs during the search process, the API will respond with an error message and an appropriate status code.

Usage

To fetch search results, make a GET request to the `/api/search/` endpoint with the desired search criteria as query parameters. You can use any combination of `category`, `stage`, and `text` parameters to perform more specific searches. The text similarity algorithm will automatically compare the provided search query with various fields to find relevant results.

For example, if you want to search for resources related to "example", you can use the `text` parameter as follows:

```
GET /api/search/?text=example
```

This will return resources with names, descriptions, websites, emails, addresses, and LinkedIn profiles that are similar to the word "example".

You can combine multiple parameters as needed to refine your search results.

SQL Injection Prevention

The API is protected against SQL injection. The `specificSearch` module and queryText variable ensures that user input is properly sanitized before executing the queries, mitigating the risk of SQL injection attacks.

todo.router.js
.get list itself
.post add new item to list
.put update list items
.delete single item off of list
.delete whole list

ADMIN

This Node.js Admin Router package is designed to streamline the management of administrative tasks related to three key data entities: Resources, Categories, and Stages. Encapsulating a series of RESTful API endpoints, the package provides the ability to execute CRUD (Create, Read, Update, Delete) operations for these entities, with each operation gated behind authentication and authorization middleware for secure and reliable administrative access control.

Key Features:
Resource Management: Leverage the API to create, read, update, and delete resource records. Each resource holds specific

details such as stage_id, category_id, name, image_url, description, website, email, address, and LinkedIn URL.

Category Management: Manage categories through the dedicated endpoints that allow you to create, read, update, and delete category records. Each category consists of a unique name.

Stage Management: Utilize the API endpoints to create, read, update, and delete stage records. Each stage has a unique name and description.

Secure Access Control: With a two-tier middleware layer, access to administrative functions is tightly controlled. The rejectUnauthenticated middleware rejects non-authenticated users, while the isAdmin middleware checks the authenticated user's admin privileges, offering another layer of access control.

The Admin Router package ensures that all these operations are securely accessible to authenticated users who have been granted admin privileges, ensuring your sensitive operations remain secure and your data integrity uncompromised.

Middleware

- rejectUnauthenticated

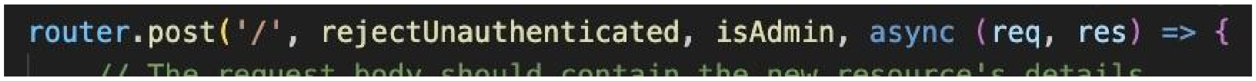
This middleware ensures that only authenticated users can perform certain operations.

- isAdmin

This middleware ensures that only users with admin privileges can perform certain operations.

Endpoints

POST /



```
router.post('/', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Creates a new resource.
- Headers:

```
Content-Type: application/json
Authorization: Bearer <token>
```

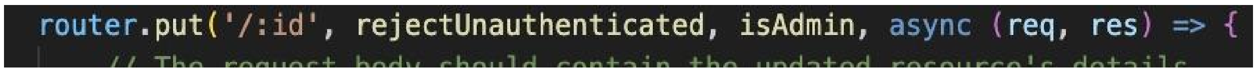
- Body:

```
{
  "stage_id": "<stage_id>",
  "category_id": "<category_id>",
  "name": "<name>",
  "image_url": "<image_url>",
  "description": "<description>",
  "website": "<website>",
  "email": "<email>",
  "address": "<address>",
  "linkedin": "<linkedin>"
}
```

- Response

```
201 on success
500 on server error
```

PUT /:id



```
router.put('/:id', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Updates a specific resource.

- Headers:

```
Content-Type: application/json
Authorization: Bearer <token>
```

- Parameters: id - ID of the resource to update

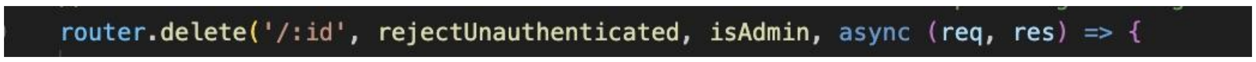
- Body

```
{
  "stage_id": "<stage_id>",
  "category_id": "<category_id>",
  "name": "<name>",
  "image_url": "<image_url>",
  "description": "<description>",
  "website": "<website>",
  "email": "<email>",
  "address": "<address>",
  "linkedin": "<linkedin>"
}
```

- Response

200 on success
500 on server error

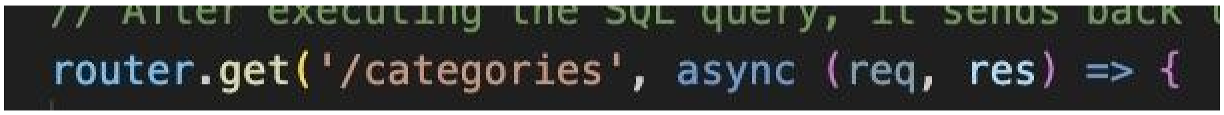
DELETE /:id



```
router.delete('/:id', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Deletes a specific resource.
- Headers:
  - Authorization: Bearer <token>
- Parameters: id - ID of the resource to delete
- Response: 200 on success, 500 on server error.

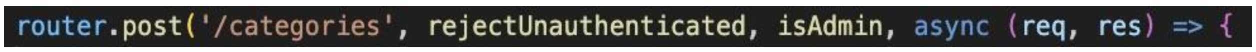
GET /categories



```
// After executing the SQL query, it sends back
router.get('/categories', async (req, res) => {
```

- Description: Fetches all categories.
- Response: 200 on success with an array of categories, 500 on server error.

POST /categories



```
router.post('/categories', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Creates a new category.
- Headers:
  - Content-Type: application/json
  - Authorization: Bearer <token>
- Body

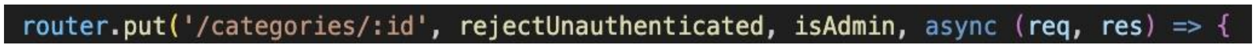
```
{
  "name": "<name>"
}
```

- Response

201 on success

500 on server error

PUT /categories/:id



```
router.put('/categories/:id', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Updates a specific category.
- Headers:
  - Content-Type: application/json
  - Authorization: Bearer <token>
  - Parameters: id - ID of the category to update
  -
- Body:

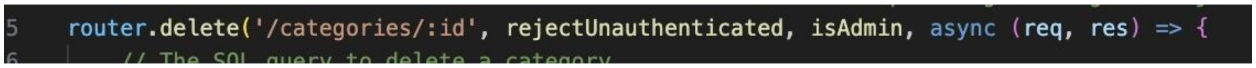
```
{
  "name": "<name>"
}
```

- Response

200 on success

500 on server error

DELETE /categories/:id



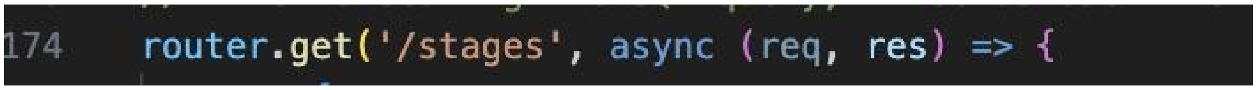
```
  router.delete('/categories/:id', rejectUnauthenticated, isAdmin, async (req, res) => {
    // The SQL query to delete a category
```

- Description: Deletes a specific category.

- Headers
    - Authorization: Bearer <token>
- Parameters id - ID of the category to delete
- Response

200 on success
500 on server error

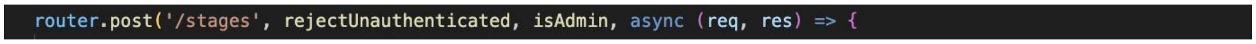
GET /stages



```
router.get('/stages', async (req, res) => {
```

- Description: Fetches all stages.
- Response
    - 200 on success with an array of stages
    - 500 on server error

POST /stages



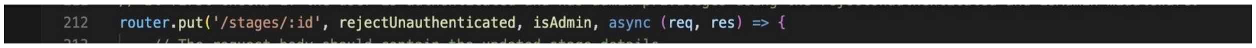
```
router.post('/stages', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Creates a new stage.
- Headers:
    - Content-Type: application/json
    - Authorization: Bearer <token>
- Body

```
{
  "name": "<name>",
  "description": "<description>"
}
```

- Response: 201 on success, 500 on server error.

PUT /stages/:id



```
router.put('/stages/:id', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Updates a specific stage.
- Headers:
    - Content-Type: application/json

    - Authorization: Bearer <token>
- Parameters: id - ID of the stage to update
- Body:

```
{
  "name": "<name>",
  "description": "<description>"
}
```

Response: 200 on success, 500 on server error.

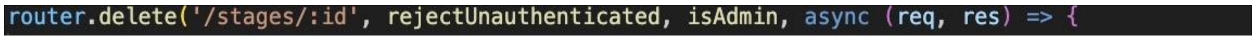
```
{
  "name": "<name>",
  "description": "<description>"
}
```

- Response

200 on success

500 on server error

DELETE /stages/:id



```
router.delete('/stages/:id', rejectUnauthenticated, isAdmin, async (req, res) => {
```

- Description: Deletes a specific stage.
- Headers
    - Authorization: Bearer <token>
- Parameters: id - ID of the stage to delete
- Response
    - 200 on success
    - 500 on server error

Note: <token> refers to the user's authentication token. Replace <stage_id>, <category_id>, <name>, <image_url>, <description>, <website>, <email>, <address>, and <linkedin> with the actual values when making requests.

Error Handling

The Admin Router package is designed to handle potential errors gracefully and provide meaningful feedback to the developer. Here are some common error scenarios and how they are handled:

Unauthenticated Access: If a user attempts to access an endpoint without being authenticated, the request will be rejected, and the API will return a 401 (Unauthorized) status code along with a descriptive error message.

Insufficient Privileges: If an authenticated user tries to perform an action that they do not have the necessary admin privileges for, the API will return a 403 (Forbidden) status code along with a meaningful error message.

Invalid Request Parameters: If the API receives a request with invalid parameters, such as trying to update a resource that doesn't exist, the API will return a 400 (Bad Request) status code along with an error message detailing what was invalid about the request.

Server Errors: If an error occurs on the server during the processing of a request, such as a failure to connect to the database, the API will return a 500 (Internal Server Error) status code along with a general error message.

Whenever an error is encountered, it is recommended to check the status code and error message returned by the API for clues on how to resolve the issue. If the error persists, please consult the detailed documentation or reach out to our support team.

Sure, here's an API documentation for the provided Express router:

To-Do List API Documentation

This API handles CRUD operations for managing to-do list items and titles.

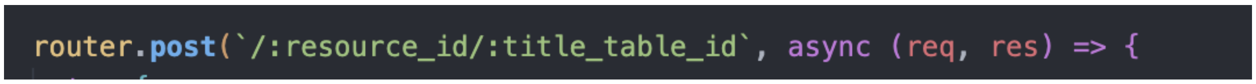
Base URL

/api/todo

Endpoints

Create a New To-Do Item

Creates a new to-do item for a specific resource.



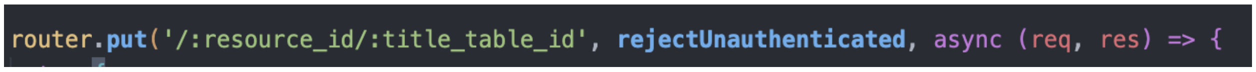
```
router.post(`/:resource_id/:title_table_id`, async (req, res) => {
```

- **FULL URL:** `/api/todo/:resource_id/:title_table_id`
- **Method:** `POST`
- **Authentication:** Required
- **Request Parameters:**
  - `resource_id` (URL Parameter): ID of the resource for which to create the to-do item.
  - `title_table_id` (URL Parameter): ID of the title table associated with the to-do item.
- **Request Body:**
  ```json
  {
    "notes": "Task description",
    "completed": false
  }
  ```
- **Response:**

  - Status Code: `201 Created`
  - Data: Newly created to-do item object

Update a To-Do Item

Updates an existing to-do item.

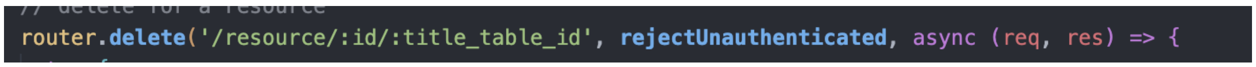


```
router.put('/:resource_id/:title_table_id', rejectUnauthenticated, async (req, res) => {
```

- **FULL URL:** `/api/todo/:resource_id/:title_table_id`
- **Method:** `PUT`
- **Authentication:** Required
- **Request Parameters:**
  - `resource_id` (URL Parameter): ID of the resource to which the to-do item belongs.
  - `title_table_id` (URL Parameter): ID of the title table associated with the to-do item.
- **Request Body:**
  ```json
  {
    "todo_id": 1,
    "notes": "Updated task description",
    "completed": true
  }
  ```
- **Response:**
  - Status Code: `204 No Content`

Delete a To-Do Item

Deletes a specific to-do item.



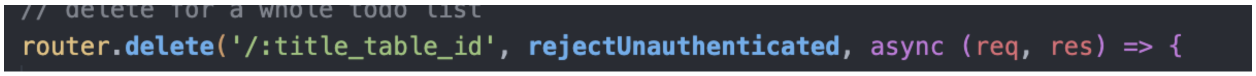
```
// delete for a resource
router.delete('/resource/:id/:title_table_id', rejectUnauthenticated, async (req, res) => {
```

- **FULL URL:** `/api/todo/resource/:id/:title_table_id`
- **Method:** `DELETE`
- **Authentication:** Required
- **Request Parameters:**
  - `id` (URL Parameter): ID of the to-do item to be deleted.

  - `title_table_id` (URL Parameter): ID of the title table associated with the to-do item.
- **Response:**
  - Status Code: `204 No Content`

Delete a Whole To-Do List

Deletes an entire to-do list (including all items) associated with a title table.

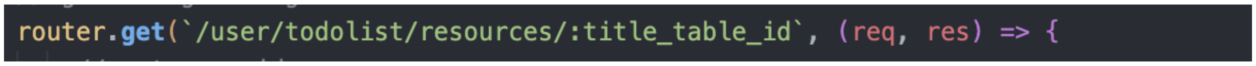


```
// delete for a whole todo list
router.delete('/:title_table_id', rejectUnauthenticated, async (req, res) => {
```

- **FULL URL:** `/api/todo/:title_table_id`
- **Method:** `DELETE`
- **Authentication:** Required
- **Request Parameters:**
  - `title_table_id` (URL Parameter): ID of the title table for which the to-do list will be deleted.
- **Response:**
  - Status Code: `204 No Content`

Get Resources for To-Do List

Retrieves resources for a specific to-do list.

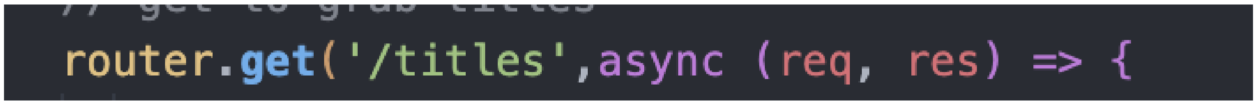


```
router.get(`/user/todolist/resources/:title_table_id`, (req, res) => {
```

- **FULL URL:** `/api/todo/user/todolist/resources/:title_table_id`
- **Method:** `GET`
- **Authentication:** Required
- **Request Parameters:**
  - `title_table_id` (URL Parameter): ID of the title table associated with the to-do list.
- **Response:**
  - Status Code: `200 OK`
  - Data: Array of resource objects

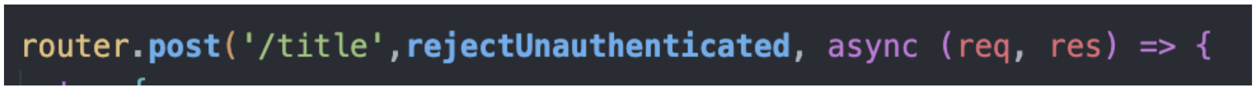
Get Titles

Retrieves all title tables associated with a user.


```
router.get('/titles',async (req, res) => {
```
- **URL:** `/api/todo/titles`
- **Method:** `GET`
- **Authentication:** Required
- **Response:**
  - Status Code: `200 OK`
  - Data: Array of title table objects

Create a New Title

Creates a new title table.


```
router.post('/title',rejectUnauthenticated, async (req, res) => {
```
- **URL:** `/api/todo/title`
- **Method:** `POST`
- **Authentication:** Required
- **Request Body:**
  ```json
  {
    "title": "My New Title"
  }
  ```
- **Response:**
  - Status Code: `201 Created`
  - Data: Newly created title table object

Error Handling

- `500 Internal Server Error`: An error occurred on the server.
- `401 Unauthorized`: Authentication is required and has failed or has not been provided.

Authentication

- Some routes require authentication using an access token.
- What this means is that only a user cna access that particular route.

Note

- Replace `:resource_id`, `:title_table_id`, and other placeholders with actual values in the API requests.
- Such as `1`, `3`. The way the database is constructed is each resource is tied to an identification number on a to-do list.

Github User Guide

How to Access a GitHub Repository in an Organization: A Beginner's Guide

Step 1: Access the Organization's Page

1. After logging in to your GitHub account, go to the GitHub homepage.
2. In the top right corner, click on your profile picture and select "Your organizations" from the dropdown menu.

Step 2: Select the Organization

1. You'll see a list of organizations you're a member of or have access to.
2. Click on the name of the organization you're interested in to access its page.

Step 3: Navigate to the Repository

1. On the organization's page, you'll find various tabs and sections. Look for the "Repositories" tab or section.
2. Click on "Repositories" to see a list of repositories within the organization.

Step 4: Access the Repository

1. Scroll through the list of repositories and find the one you want to access.
2. Click on the repository's name to open its main page.

Step 5: Explore the Repository

1. You are now on the repository's page. Here, you can find valuable information about the project.
2. Tabs at the top of the repository page allow you to access different sections, such as "Code," "Issues," "Pull Requests," and more.

Step 6: Viewing Files

1. To view files within the repository, click on the "Code" tab.
2. You'll see a list of files and folders in the repository. You can navigate through the directory structure by clicking on folders.
3. Click on a file's name to view its contents.

Step 7: Downloading or Cloning

1. To download the entire repository as a ZIP file, click the green "Code" button and select "Download ZIP."
2. If you're familiar with Git, you can clone the repository to your local machine using a Git client. Copy the repository's URL and use it to clone the repository.

Step 8: Interacting with the Repository

1. You can contribute to the repository by creating issues, making pull requests, and more. These options are available in the tabs at the top of the repository page.