Entrypoint (App.js)

This is a basic Express.js application setup. Here's a step-by-step explanation:

1. `require('dotenv').config();` - This line loads environment variables from a `.env` file into `process.env`. This is useful for managing secrets like API keys and database URIs.

2. `const express = require("express");` - This line imports the Express.js library.

3. `const app = express();` - This line creates a new Express application.

4. The next few lines import various route handlers:

    - `const authRoutes = require("./routes/auth");` - This imports the authentication routes.
    - `const tasksRoutes = require("./routes/tasksRouters");` - This imports the task-related routes.
    - `const categorieRoutes = require("./routes/categoryRoutes");` - This imports the category-related routes.
    - `const queryRoutes = require("./routes/qureyRoutes");` - This imports the query-related routes.

5. `const mongoose = require('mongoose');` - This line imports Mongoose, a MongoDB object modeling tool.

6. `app.use(express.json());` - This line adds middleware to parse JSON bodies from incoming requests.

7. The next few lines mount the imported route handlers at their respective paths:

    - `app.use("/users", authRoutes);` - This mounts the authentication routes at `/users`.
    - `app.use("/tasks", tasksRoutes);` - This mounts the task-related routes at `/tasks`.
    - `app.use("/categories", categorieRoutes);` - This mounts the category-related routes at `/categories`.
    - `app.use("/query", queryRoutes);` - This mounts the query-related routes at `/query`.

8. `const PORT = process.env.PORT || 3000;` - This line sets the port to the value of `process.env.PORT` if it's defined, otherwise it defaults to `3000`.

9. `mongoose.connect(process.env.DB_URI)...` - This line connects to the MongoDB database using the URI stored in `process.env.DB_URI`.

10. `app.listen(PORT, () => {...` - This line starts the server on the specified port and logs a message to the console.

11. `module.exports = app;` - This line exports the Express application, which allows it to be imported and used in other files.

Environment Variables (.env)

This `.env` file contains environment variables for your application. Here's what each of them means:

- `PORT=3000`: This sets the port number that your server will listen on. In this case, it's set to `3000`.
- `JWT_SECRET=71f48d1487181685f4d494ba43a6da114cea8979edb91be0ecb20b360cfeb0e4`: This is the secret key used to sign and verify JSON Web Tokens (JWTs) in your application. It's important to keep this secret and not expose it publicly.
- `DB_URI=mongodb+srv://<mongo-URI-Prefix>.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0`: This is the connection string for your MongoDB database. It includes the username (`db-user`), password (`password`), and the address of your MongoDB server . The `retryWrites=true&w=majority` part ensures that write operations are retried if they fail, and `appName=Cluster0` sets the name of the application connecting to the MongoDB server.

Remember, `.env` files should not be committed to your version control system as they often contain sensitive information. Instead, you should provide a `.env.example` file with the keys and example values, and instruct users to create their own `.env` file based on it.

Middleware (authMiddleware.js)

This is an Express middleware function named `verifyToken`. Its purpose is to authenticate requests by verifying a JSON Web Token (JWT) that should be included in the `Authorization` header of the request.

Here's a step-by-step explanation of what this middleware does:

1. It tries to get the `Authorization` header from the request. This header should contain the JWT prefixed with the word `Bearer`.
2. If the `Authorization` header is not present, it sends a response with the HTTP status code `401` (Unauthorized) and a JSON object `{ error: "Access denied" }`, then ends the middleware function.
3. If the `Authorization` header is present, it splits the header value by space to extract the actual JWT (which should be the second part of the header value after `Bearer`).

4. It then tries to verify the JWT using the `jwt.verify()` function from the `jsonwebtoken` library. The `jwt.verify()` function decodes the token and checks its signature to ensure it's valid and hasn't been tampered with. The secret key for verifying the token is obtained from `process.env.JWT_SECRET`.

5. If the token is valid, it assigns the decoded `userId` to `req.userId` and then calls `next()` to pass control to the next middleware function in the stack.

6. If the token is not valid (for example, if it's expired or has been tampered with), it catches the error, logs it to the console, and sends a response with the HTTP status code `401` (Unauthorized) and a JSON object `{ error: "Invalid token" }`.

This middleware is typically used to protect routes that require authentication. By placing it before a route handler, you can ensure that the route handler is only executed if the request includes a valid JWT.

Models and Relation (DB Schema)

User Model (`User.js`)

The User model defines the structure of the user documents in MongoDB. It uses Mongoose for object data modeling (ODM).

- **Fields**:
    - `_id`: The unique identifier for the user. It's of type `ObjectId`.
    - `username`: The username of the user. It's a string.
    - `password`: The password of the user. It's a string.
    - `email`: The email of the user. It's a string.
    - `categories`: An array of category IDs that the user is associated with. Each ID is of type `ObjectId` and references the `Category` model.
    - `tasks`: An array of task IDs that the user is associated with. Each ID is of type `ObjectId` and references the `Task` model.
- **Methods**:
    - `comparePassword(password)`: This method compares a given password with the user's password. It uses bcrypt for password comparison and returns a promise that resolves to `true` if the passwords match and `false` otherwise.
- **Export**: The User model is exported as a Mongoose model named 'User'.

Here's an example of how you can use the User model:

```
const User = require('./User');

// Create a new user
const user = new User({
  _id: new mongoose.Types.ObjectId(),
  username: 'testuser',
  password: 'testpass',
  email: 'testuser@test.com',
});

// Save the user to the database
user.save().then(() => console.log('User saved successfully'));
```

Task Model (`Task.js`)

The Task model defines the structure of the task documents in MongoDB. It uses Mongoose for object data modeling (ODM).

- **Fields**:
  - `_id`: The unique identifier for the task. It's of type `ObjectId`.
  - `title`: The title of the task. It's a string.
  - `description`: The description of the task. It's a string.
  - `status`: The status of the task. It's a string and defaults to "pending".
  - `dueDate`: The due date of the task. It's a date.
  - `category`: The category the task belongs to. It's an `ObjectId` that references the `Category` model.
  - `user`: The user who created the task. It's an `ObjectId` that references the `User` model.
- **Export**: The Task model is exported as a Mongoose model named 'Task'.

Here's an example of how you can use the Task model:

```
const Task = require('./Task');

// Create a new task
const task = new Task({
  _id: new mongoose.Types.ObjectId(),
  title: 'Test Task',
  description: 'This is a test task',
  status: 'pending',
  dueDate: new Date(),
  category: new mongoose.Types.ObjectId(), // Replace with actual category ID
  user: new mongoose.Types.ObjectId(), // Replace with actual user ID
});

// Save the task to the database
task.save().then(() => console.log('Task saved successfully'));
```

Category Model (`Category.js`)

The Category model defines the structure of the category documents in MongoDB. It uses Mongoose for object data modeling (ODM).

- **Fields**:
  - `_id`: The unique identifier for the category. It's of type `ObjectId`.
  - `name`: The name of the category. It's a string and is required.
  - `user`: The user who created the category. It's an `ObjectId` that references the `User` model.
  - `tasks`: An array of task IDs that belong to the category. Each ID is of type `ObjectId` and references the `Task` model.
- **Export**: The Category model is exported as a Mongoose model named 'Category'.

Here's an example of how you can use the Category model:

```
const Category = require('./Category');

// Create a new category
const category = new Category({
  _id: new mongoose.Types.ObjectId(),
  name: 'Test Category',
  user: new mongoose.Types.ObjectId(), // Replace with actual user ID
});

// Save the category to the database
category.save().then(() => console.log('Category saved successfully'));
```

Routes

User Routes (`/users`)

This endpoint is used for user registration, login, fetching user profile, and updating user profile.

- **POST `/users/register`**: Register a new user.
  - Request body: `{ "username": "string", "password": "string", "email": "string" }`
  - Response: `{ "message": "User registered successfully" }` or `{ "error": "Registration Failed!" }` in case of an error.
- **POST `/users/login`**: Log in a user.
  - Request body: `{ "username": "string", "password": "string" }`
  - Response: `{ "token": "string" }` or `{ "error": "Authentication failed" }` in case of an error.
- **GET `/users/me`**: Get the profile of the logged-in user. This route is protected by the `verifyToken` middleware, so you need to include a valid JWT in the `Authorization` header of your request.
  - Response: `{ "_id": "string", "username": "string", "email": "string" }` or `{ "error": "string" }` in case of an error.
- **PUT `/users/me`**: Update the profile of the logged-in user. This route is protected by the `verifyToken` middleware, so you need to include a valid JWT in the `Authorization` header of your request.
  - Request body: `{ "username": "string", "password": "string", "email": "string" }`
  - Response: `{ "message": "User updated" }` or `{ "error": "string" }` in case of an error.

Task Routes (`/tasks`)

This endpoint is used for creating, fetching, updating, and deleting tasks. All routes are protected by the `verifyToken` middleware, so you need to include a valid JWT in the `Authorization` header of your request.

- **POST `/tasks`**: Create a new task.
  - Request body: `{ "title": "string", "description": "string", "status": "string", "dueDate": "date", "category": "string" }`
  - Response: `{ "_id": "string", "title": "string", "description": "string", "status": "string", "dueDate": "date", "category": "string", "user": "string" }` or `{ "error": "string" }` in case of an error.
- **GET `/tasks`**: Get all tasks of the logged-in user.
  - Response: `[ { "_id": "string", "title": "string", "description": "string", "status": "string", "dueDate": "date", "category": "string", "user": "string" }, ... ]` or `{ "error": "string" }` in case of an error.
- **GET `/tasks/:id`**: Get a specific task by ID.

    - Response: `{ "_id": "string", "title": "string", "description": "string", "status": "string", "dueDate": "date", "category": "string", "user": "string" }` or `{ "message": "Task not found" }` if the task is not found or `{ "error": "string" }` in case of an error.

- **PUT** `/tasks/:id`: Update a specific task by ID.
    - Request body: `{ "title": "string", "description": "string", "status": "string", "dueDate": "date", "category": "string" }`
    - Response: `{ "message": "Task updated" }` or `{ "error": "string" }` in case of an error.

- **DELETE** `/tasks/:id`: Delete a specific task by ID.
    - Response: `{ "message": "Task deleted" }` or `{ "error": "string" }` in case of an error.

Category Routes (`/category`)

This endpoint is used for creating, fetching, updating, and deleting categories. All routes are protected by the `verifyToken` middleware, so you need to include a valid JWT in the `Authorization` header of your request.

- **POST** `/category`: Create a new category.
    - Request body: `{ "name": "string" }`
    - Response: `{ "_id": "string", "name": "string", "user": "string" }` or `{ "error": "string" }` in case of an error.

- **GET** `/category`: Get all categories of the logged-in user.
    - Response: `[ { "_id": "string", "name": "string", "user": "string" }, ... ]` or `{ "error": "string" }` in case of an error.

- **GET** `/category/:id`: Get a specific category by ID.
    - Response: `{ "_id": "string", "name": "string", "user": "string" }` or `{ "message": "Category not found" }` if the category is not found or `{ "error": "string" }` in case of an error.

- **PUT** `/category/:id`: Update a specific category by ID.
    - Request body: `{ "name": "string" }`
    - Response: `{ "message": "Category updated" }` or `{ "error": "string" }` in case of an error.

- **DELETE** `/category/:id`: Delete a specific category by ID.
    - Response: `{ "message": "Category deleted" }` or `{ "error": "string" }` in case of an error.

Query Routes (`/query`)

This endpoint is used for fetching tasks based on query parameters. The route is protected by the `verifyToken` middleware, so you need to include a valid JWT in the `Authorization` header of your request.

- **GET `/query`**: Get tasks based on query parameters.
  - Query parameters:
    - `categoryId`: The ID of the category to filter tasks by.
    - `status`: The status to filter tasks by.
    - `dueDate`: The due date to filter tasks by. This should be in ISO date format (YYYY-MM-DD).
  - Response: `[ { "_id": "string", "title": "string", "description": "string", "status": "string", "dueDate": "date", "category": "string", "user": "string" }, ... ]` or `{ "error": "string" }` in case of an error.

Here's an example of how you can use the `/query` endpoint:

```
GET /query?categoryId=60d5ecf8b484193a7849f4a6&status=pending&dueDate=2022-12-31
Authorization : Bearer <your-jwt-token>
```

Status Code

HTTP status codes:

- `201 Created` for successful user registration. This indicates that the request has been fulfilled and has resulted in one or more new resources being created.

- `200 OK` for successful login and successful user profile update. This indicates that the request has been successfully processed.

- `401 Unauthorized` for failed login. This indicates that the request requires user authentication.

- `500 Internal Server Error` for server errors during registration, login, fetching user profile, and updating user profile. This indicates that the server encountered an unexpected condition that prevented it from fulfilling the request.

Data Prototypes

Hare are some prototype data for `user`, `category`, and `tasks`.

User

```
{
  "_id": "60d5ecf8b484193a7849f4a6",
  "name": "John Doe",
  "email": "john.doe@example.com",
  "password": "$2a$10$V9DPoHVue2sVxkq1T5bbCep1w2AJ8zWRH3fBiCXndY8aR9M7oT.a6" // hashed pass
}
```

Category

```
{
  "_id": "60d5edadb484193a7849f4a7",
  "name": "Work",
  "user": "60d5ecf8b484193a7849f4a6",
  "Tasks": ["60d5ee3fb484193a7849f4a8"]
}
```

Tasks

```
[
  {
    "_id": "60d5ee3fb484193a7849f4a8",
    "title": "Finish report",
    "description": "Finish the annual report by the end of the week",
    "status": "In Progress",
    "dueDate": "2022-12-31",
    "category": "60d5edadb484193a7849f4a7",
    "user": "60d5ecf8b484193a7849f4a6"
  },
  {
    "_id": "60d5ee59b484193a7849f4a9",
    "title": "Plan project",
    "description": "Plan the new project for next quarter",
    "status": "Not Started",
    "dueDate": "2023-01-31",
    "category": "60d5edadb484193a7849f4a7",
    "user": "60d5ecf8b484193a7849f4a6"
  }
]
```

Please note that the `password` in the `user` data is hashed for security reasons. In a real-world application, you would use a library like bcrypt to hash passwords before storing them. The `user` field in the `category` and `tasks` data is a reference to the `_id` of the `user` who owns the category or task.