**Introduction:**

This document provides an overview of a NodeJS application developed using Express and TypeScript, which is connected to a MongoDB database. The application allows users to log in with their credentials and manage developers, assets, and licenses. There are three main entities in the application: developers, assets, and licenses, each with their respective attributes.

**Interesting Links:**

GitHub repository: <https://github.com/ecaracasdev/challange-channelworks>

Additionally, the GitHub repository includes a README file that provides instructions for installing the project. It is recommended to refer to the README file to ensure a smooth installation of the application.

**Entity Attributes:**

1. Asset Attributes:

id: string | number

brand: string

model: string

type: enum<string> ['laptop', 'keyboard', 'mouse', 'headset', 'monitor']

1. License Attributes:

id: string | number

software: string

1. Developer Attributes:

id: string | number

fullname: string

active: Boolean

The application uses Mongoose to model each entity, and simulates a one-to-many relationship between developers and assets and between developers and licenses.

**API Endpoints:**

The application exposes the following API endpoints to manage the entities. In order to access any of the available endpoints in the app, **the user must authenticate first,** which will generate an access token that will allow them to use the app's endpoints.

**Documentation**

GET {base\_url}/api//api-docs

The application also includes a Swagger documentation that can be accessed by sending a GET request to the endpoint. Swagger is a tool that helps to create, document, and test APIs, and it provides a user-friendly interface to explore and understand the endpoints exposed by the application. The Swagger documentation is automatically generated based on the API endpoints and can be customized to provide additional information about the API.

**Authentication**

POST {base\_url}/api/auth/login: Allows users to log in with their credentials.

This endpoint is a login API endpoint that allows users to log in by sending a POST request to the specified route '/login'. The expected input to the endpoint is a JSON body that contains 'username' and 'password' as key value pairs that represent the username and password the user is trying to log in with.

When this endpoint receives the request, it extracts the 'username' and 'password' values from the request's JSON body. It checks whether the username and password provided match the hardcoded 'adminUser' credentials.

If the credentials match, the endpoint creates a JSON Web Token (JWT) with a secret key defined in the configuration file, via the 'jwt.sign' method from the jsonwebtoken library. This JWT token is then set as a response header and sent back to the user. The response body is a JSON object that contains the JWT token and a success message, indicating that the login was successful.

If the credentials do not match, then the endpoint returns an error message stating that the credentials are invalid, with a 401 error code.

**Assets:**

GET {base\_url}/api/assets/: Returns a list of all assets.

DELETE {base\_url}/api/assets/{assetId}: Deletes an asset with the given asset ID.

POST {base\_url}/api/assets/: Creates a new asset.

**Licenses:**

GET {base\_url}/api/licenses/: Returns a list of all licenses.

DELETE {base\_url}/api/licenses/{licenseId}: Deletes a license with the given license ID.

POST {base\_url}/api/licenses/: Creates a new license.

**Developers:**

GET {base\_url}/api/developers/: Returns a list of all developers.

GET {base\_url}/api/developers/{developerID}:

This endpoint handles a GET request at the specified route with a required ":id" URL parameter. The request retrieves information about a developer from the database.

When the endpoint receives a request, it passes control to the 'getDevById()' controller function. This function first performs validation on the provided 'id' URL parameter and the optional query property. If the parameters are invalid, an error message is returned with a 400 status code.

If the input parameters are valid, the function queries the database to retrieve the developer associated with the 'id' URL parameter. If a matching developer is found, the controller associates 'assets' and 'licenses' arrays with the retrieved developer object using the 'populate()' method.

If the optional query property is empty or not provided, the controller returns the full developer object along with a "Developer" success message. If the query property is 'assets', the controller returns an object with only the 'assets' array and a "Developer Assets" success message; if the query parameter is 'licenses', this function returns an object with only the 'licenses' array, along with a "Developer Licenses" success message.

In summary, this endpoint retrieves information about a developer from the database based on the provided 'id' URL parameter. The optional query parameter allows the client to specify whether to retrieve just the developer, developer's assets or developer's licenses.

POST {base\_url}/api/developers/: Creates a new developer.  
  
 This endpoint handles a POST request to add a new developer to the database. When a client sends a request, control is passed to the 'postDeveloper()' controller function.

The function first validates the request body to ensure that the 'assets' and 'licenses' properties are arrays of valid ObjectId strings. If any validation errors are present, the function returns an error message with a 400 status code.

If the input parameters are valid, the controller creates a new developer object from the provided request body and saves it to the database using the 'create()' method on the 'Developer' model.

If the developer is successfully saved, the controller returns a success message with the newly created developer object and a 200 status code. If saving the developer to the database fails, the controller returns an error message with a 500 status code.

PATCH {base\_url}/api/developers/{developerID}: Updates the details of the developer with the given developer ID.

This endpoint handles a PATCH request to set a developer as inactive in the database. When a client sends a request to this endpoint with the ID of the developer to be set as inactive, control is passed to the 'setDeveloperAsInactive()' controller function.

The function first validates the 'id' parameter to ensure that it's a valid ObjectId. If it's not, the function returns an error message with a 400 status code.

If the input parameter is valid, the controller queries the database for the developer with the provided ID using the 'findOneAndUpdate()' method to update the developer document. The 'active' field is set to 'false' and the 'assets' and 'licenses' fields are removed from the developer document.

If the developer is successfully updated, the controller returns a success message with the updated developer document and a 200 status code. If updating the developer fails, the controller returns an error message with a 400 status code.

In summary, this endpoint sets a developer as inactive in the database and removes their assets and licenses from their document. The endpoint validates the input parameter to ensure it's a valid ObjectId before updating the developer document.

PATCH {base\_url}/api/developers/{developerID}/assets-licenses: Associates assets and licenses with the developer.

This endpoint handles PATCH requests to add or remove assets and licenses for a developer. When the client sends a request to this endpoint with the ID of the developer whose asset or license(s) should be updated, control is passed to the 'updateAssetLicense()' controller function.

If the type of update is 'asset', the function loops through each asset in the value array, finds the asset in the database, and returns an error message with a 400 status code if the asset is not found in the database. A switch statement then determines whether the operation to be performed is 'add' or 'remove'. If the operation is 'add', the function filters out assets that the developer already has and adds the remaining assets to the array containing the developer's assets. If the operation is 'remove', the function finds the first asset in the value array and removes it from the developer's asset array.

If the type of update is 'license', the function checks each license in the value array to ensure that it exists in the database, and returns an error message with a 400 status code if not found. If the operation to be performed is 'add', the function adds the license to the developer's licenses array if the license is not already added. If the operation is 'remove', the function finds the first license in the value array and removes it from the developer's license array.

If the type provided is not 'asset' or 'license', the function returns an error message with a 400 status code.

Finally, the function saves the updated developer document to the database and returns a success message with the updated developer document and a 200 status code.

In conclusion, this endpoint enables the update of a developer's assets and licenses array. If the update is successful, the updated developer document is returned with a success message, else an error message is returned.

**Conclusion:**

In summary, this NodeJS application using Express and TypeScript, and MongoDB, allows users to manage developers, assets, and licenses. The application is designed with a RESTful API approach, and provides endpoints to perform CRUD operations on each entity. The application is easy to use and maintain and can be extended to support additional functionalities as required.