TABLE OF CONTENTS

- INTRODUCTION
- MODELS
- ROUTES
- SERVER
- DEPENDENCIES
- CONCLUSION

INTRODUCTION

WELCOME TO THE TERSANO CODING CHALLENGE FOR JUNIOR DEVELOPERS!

We are excited to offer an opportunity for aspiring developers to demonstrate their skills in full-stack development. This challenge is designed to test your abilities in creating a dynamic web application using modern technologies such as React, TypeScript, and Express. Your task will involve building a Product Management Interface complete with user authentication—a key component of many commercial applications today.

This project not only showcases your technical prowess but also your ability to implement practical features in a real-world scenario. By participating in this challenge, you will gain hands-on experience with front-end and back-end development, highlighting your versatility and readiness to tackle software development tasks in a professional environment.

We look forward to seeing your innovative solutions and wish you the best of luck in this endeavor!

USER MODEL

This model is designed to handle user data, specifically for authentication and user management purposes. Here are the key components of the `userSchema`:

- **firstName and lastName**: These fields store the user's first and last names, respectively. Both are required fields, meaning the database will not accept entries where these fields are empty. The `trim` option is set to true to remove any whitespace from the beginning and end of the names.
- **email**: This is a unique identifier for each user. It is also required and set to convert all entries to lowercase to avoid case-sensitive issues with user login. The `unique` attribute ensures that no two users can register with the same email address, preventing duplicate entries.
- **password**: A required field that stores the user's password. Before a user record is saved to the database, the password is encrypted using bcrypt. This encryption is crucial for security, ensuring that plain text passwords are never stored in the database.
- **Pre-save hook**: This is a function that runs before each save operation on the `User` model. If the password field has been modified (i.e., on user creation or password update), it hashes the new password using bcrypt before saving it to the database.
- **isCorrectPassword method**: This instance method provides a way to compare a given password with the hashed password stored in the database. This is used during user login to validate credentials.

PRODUCT MODEL

This model manages product information, which includes:

- **name**: The name of the product, which is required. The `trim` option ensures no extraneous whitespace.
- **price**: This is also a required field and must be a number that is not negative, as enforced by the `min` option set to 0. This prevents the entry of invalid price values.
- **description**: A brief description of the product, required for each entry. Like other string fields, it is trimmed of leading and trailing whitespace.

USAGE

These models are used to interact with the corresponding collections in the MongoDB database. They make use of Mongoose's capabilities to enforce data integrity and security through schema validations, unique constraints, and pre-save operations. This setup ensures that the data

conforms to the expected format and that sensitive information, like passwords, is appropriately handled.

In a full-stack application, these models would be interacted with via API endpoints that allow for operations like creating new users, logging in, adding products, and querying product lists—all of which require checks and data manipulation as defined in these schemas

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: true,
    trim: true
  },
  lastName: {
    type: String,
    required: true,
    trim: true
  },
  email: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    lowercase: true
  },
  password: {
    type: String,
    required: true
  }
});
```

```
const mongoose = require('mongoose');

const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Product name is required"],
    trim: true
  },
  price: {
    type: Number,
    required: [true, "Product price is required"],
    min: [0, "Price cannot be negative"]
  },
  description: {
    type: String,
    required: [true, "Product description is required"],
    trim: true
  }
});

const Product = mongoose.model('Product', productSchema);

module.exports = Product;
```

PRODUCT ROUTES

These routes handle CRUD (Create, Read, Update, Delete) operations for products. They are secured with JWT-based authentication, meaning only authenticated users can access these endpoints.

1. **GET `/getall`**:
   - **Purpose**: Fetches all products from the database.
   - **Authentication**: Protected by the `authenticateToken` middleware which checks if the user's request includes a valid JWT.
   - **Functionality**: Uses Mongoose's `find()` method to retrieve all product entries. If successful, it sends back the list of products; otherwise, it returns an error.
2. **POST `/add`**:
   - **Purpose**: Adds a new product to the database.
   - **Authentication**: Requires a valid JWT.
   - **Functionality**: Extracts product details (name, price, description) from the request body and creates a new `Product` instance which is then saved to the database. On success, it returns the newly created product; on failure, an error message.
3. **PUT `/:productId`**:
   - **Purpose**: Updates an existing product identified by `productId`.
   - **Authentication**: JWT verification is required.
   - **Functionality**: Updates the product details using Mongoose's `findByIdAndUpdate()` method. If the product is found and updated successfully, it returns the updated product; if not, a not found error.
4. **DELETE `/:productId`**:
   - **Purpose**: Deletes a product based on the provided `productId`.
   - **Authentication**: Secured by a JWT.
   - **Functionality**: Uses `findByIdAndDelete()` to remove the product. If successful, it sends a confirmation message; if the product doesn't exist, it returns a not found error.

```
// POST product - Add a new product
router.post('/add', authenticateToken, async (req, res) => {
  try {
    const { name, price, description } = req.body;
    const product = new Product({ name, price, description });
    await product.save();
    console.log("Adding Product ....")
    console.log("New product added:", product);  // Logs the newly created product
    res.status(201).send(product);
  } catch (error) {
    res.status(400).send({ message: "Failed to create product", error });
  }
});
```

```
// GET products - List all products
router.get('/getall', authenticateToken, async (req, res) => {
  try {
    const products = await Product.find();
    console.log("Fetching Data .....")
    console.log(products);  // This line logs the fetched products to the console
    res.send(products);
  } catch (error) {
    res.status(500).send({ message: "Failed to fetch products", error });
  }
```

```
// PUT product - Update a product
router.put('/:productId', authenticateToken, async (req, res) => {
  try {
    const { productId } = req.params;
    const { name, price, description } = req.body;
    const product = await Product.findByIdAndUpdate(productId, { name, price, description }, { new: true, runVal
    if (!product) {
      return res.status(404).send({ message: "Product not found" });
    }
    console.log("Updating .....")
    console.log("Product updated:", product);  // Logs the updated product
    res.send(product);
  } catch (error) {
    res.status(400).send({ message: "Failed to update product", error });
  }
});
```

```
// DELETE product - Delete a product
router.delete('/:productId', authenticateToken, async (req, res) => {
  try {
    const { productId } = req.params;
    const product = await Product.findByIdAndDelete(productId);
    if (!product) {
      return res.status(404).send({ message: "Product not found" });
    }
    console.log("Deleting .....")
    console.log("Product deleted:", productId);  // Logs the ID of the deleted product
    res.send({ message: "Product deleted successfully" });
  } catch (error) {
    res.status(500).send({ message: "Failed to delete product", error });
  }
});
```

USER ROUTES

These routes handle user registration, login, and logout, utilizing JWTs for session management and authentication.

1. **POST `/signup`**:
    - **Purpose**: Registers a new user.
    - **Functionality**: After checking that the provided passwords match, it creates a new `User` with the given details. If the user is saved successfully, a JWT is generated and returned along with the user data.
2. **POST `/login`**:
    - **Purpose**: Authenticates a user.
    - **Functionality**: Verifies the user's credentials by finding the user by email and comparing the provided password with the stored hash. If authentication is successful, a JWT is issued and returned with the user's details; if not, it returns an error.
3. **POST `/logout`**:
    - **Purpose**: Facilitates user logout.
    - **Functionality**: Since JWTs are stateless, the logout process primarily involves the client discarding the stored token. The server logs the logout attempt for record-keeping but does not need to alter the token itself.

These routes form the backbone of the server-side logic in a full-stack application, ensuring secure and functional interaction with the database for managing user sessions and product data. They are crucial for the proper operation and security of web applications that handle sensitive user data and critical business functions.

```
// Login route
router.post('/login', async (req, res) => {
  try {
    const { email, password } = req.body;
    console.log('Login attempt:', email);  // Log login attempt
    const user = await User.findOne({ email });
    if (!user || !(await user.isCorrectPassword(password))) {
        console.log('Login failed for:', email);  // Log failed login
      return res.status(400).send({ error: 'Invalid login credentials' });
    }
    const token = generateToken(user);  // Generate JWT when user logs in
    console.log('User logged in:', user._id);  // Log user ID
    res.send({ user, token });
  } catch (error) {
    console.error('Login error:', error);  // Log errors
    res.status(500).send(error);
  }
});
```

```
// Signup route
router.post('/signup', async (req, res) => {
  const { firstName, lastName, email, password, rePassword } = req.body;

  console.log('Signup request:', firstName, lastName, email);  // Log input data

  // Check if passwords match
  if (password !== rePassword) {
    console.log('Password mismatch for:', email);  // Log password mismatch
    return res.status(400).send({ error: 'Passwords do not match' });
  }

  try {
    const user = new User({ firstName, lastName, email, password });
    await user.save();
    const token = generateToken(user);  // Generate JWT when user signs up
    console.log('User created:', user._id);  // Log user ID
    res.status(201).send({ user, token });
  } catch (error) {
    console.error('Signup error:', error);  // Log errors
    res.status(400).send(error);
  }
});
```

```
// Logout
router.post('/logout', (req, res) => {
    console.log('Logout request from user ID:', req.userId);  // Log logout attempt
    // Since JWTs are stateless, logout is mostly a client-side action where the token is discarded.
    res.send({ message: "Logged out successfully" });
  });
```

ENVIRONMENT VARIABLES AND MODULES

1. **Environment Variables**:
    - `require('dotenv').config();`: This line loads environment variables from a `.env` file into `process.env`, allowing you to configure aspects like database URIs and ports externally and securely.
2. **Module Imports**:

- `const express = require("express");`: Imports the Express library for server-side logic.
- `const mongoose = require("mongoose");`: Imports Mongoose, an ODM (Object Data Modeling) library for MongoDB and Node.js.
- `const cors = require("cors");`: Imports the CORS (Cross-Origin Resource Sharing) middleware to enable cross-origin requests, which are essential for a client-server architecture where the client and server might be hosted on different domains.

SERVER CONFIGURATION

3. **Express App Setup**:
   - `const app = express();`: Creates an instance of an Express app.
   - `app.use(cors());`: Applies the CORS middleware globally, allowing access to your server's resources from different domains.
   - `app.use(express.json());`: Middleware to parse JSON payloads from incoming requests, a common format for client-server communication.

DATABASE CONNECTION

4. **MongoDB Connection**:
   - `mongoose.connect()`: Connects to the MongoDB database using a connection URI provided by the `DB_URI` environment variable. It includes configuration options like `useNewUrlParser` and `useUnifiedTopology` to ensure the connection uses the latest MongoDB driver features without deprecated elements.

ROUTES SETUP

5. **Route Imports and Application**:
   - Routes for authentication and product management are imported and then applied to specific base paths:
   - `app.use('/api/auth', authRoutes);`: Binds authentication-related routes (`authRoutes`) under the `/api/auth` base path.
   - `app.use('/api/products', productRoutes);`: Binds product-related routes (`productRoutes`) under the `/api/products` base path. This organization helps maintain a clean and manageable API structure.

SERVER INITIALIZATION

6. **Starting the Server**:
   - `const PORT = process.env.PORT || 3001;`: Determines the server's port from an environment variable with a fallback to `3001` if not specified.
   - `app.listen(PORT, ...);`: Starts the server on the specified `PORT`, listening for incoming requests. A callback function logs a success message to the console indicating that the server is running.

SUMMARY

This `server.js` configuration sets up a secure, well-organized, and efficient server for a full-stack web application. It handles essential functions like environmental configuration, CORS for cross-domain accessibility, JSON parsing for request data, persistent database connection handling with MongoDB, and structured routing for authentication and product management. This setup is ideal for modern web applications that require robust backend capabilities and secure data management.

```
require('dotenv').config();
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors"); // Import CORS
const app = express();
app.use(cors()); // Enable CORS for all requests
app.use(express.json());

const authRoutes = require('./routes/authRoutes');
const productRoutes = require('./routes/productRoutes');  // Import product routes

mongoose
  .connect(process.env.DB_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() => console.log("Connected to DB"))
  .catch((error) => console.error(error));


  app.use('/api/auth', authRoutes);
  app.use('/api/products', productRoutes);  // Use product routes with authentication


const PORT = process.env.PORT || 3001;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

DEPENDENCIES EXPLAINED

1. **`bcryptjs`**: A library to help you hash passwords securely. In your project, it's used to hash and check passwords stored in your database, which is crucial for maintaining user privacy and security.
2. **`body-parser`**: Middleware for parsing incoming request bodies in a middleware before your handlers, available under the `req.body` property. It's now mainly built into Express.js (from version 4.16.0 onward), so you may not need to separately install it unless for specific use cases.

3. **`cors`**: Enables Cross-Origin Resource Sharing (CORS) with various options. It allows your backend to accept requests from different domains, essential when your backend and frontend are served from different origins.
4. **`dotenv`**: Loads environment variables from a `.env` file into `process.env`, providing a secure and flexible way to handle configuration without hard-coding sensitive information in your source code.
5. **`express`**: Fast, unopinionated, minimalist web framework for Node.js. It's fundamental for handling routing, middleware, and web server creation.
6. **`jsonwebtoken`**: Implements JSON Web Tokens to securely transmit information between parties as a JSON object, crucial for handling user authentication and sessions in your application.
7. **`mongoose`**: MongoDB object modeling tool designed to work in an asynchronous environment. It provides a straightforward solution for defining schemas, handling database interaction, and modeling your application data in a MongoDB database.

DEVELOPMENT DEPENDENCIES

1. **`nodemon`**: A utility that monitors for any changes in your source and automatically restarts your server. It's particularly useful during development to save time and effort in manually restarting the server after each change.

SCRIPTS

- **`test`**: Currently set to a placeholder that just throws an error since no tests are specified. In a fully developed application, you'd replace this with scripts to run your unit tests or integration tests.
- **`start`**: Uses `nodemon` to run your server, which enhances your development process by reloading your app automatically whenever file changes in the directory are detected.

DEPLOYMENT CONSIDERATIONS WITH VERCEL

Deploying your Node.js backend to Vercel:

- **Vercel**: Primarily known for hosting front-end applications and static sites, Vercel can also host serverless functions, which can serve your Express.js routes. However, for traditional Express applications that maintain an open server connection, you may need to adjust your application architecture to fit into the serverless function model.
- **Environment Variables**: Use Vercel's dashboard to configure the production environment variables like `DB_URI`.
- **Adaptation**: You might need to adapt your Express application into smaller serverless functions, which can be a shift from the traditional setup. Each route could potentially be a separate serverless function.

```
"name": "backend",
"version": "1.0.0",
"description": "",
"main": "index.js",
"engines": {
  "node": "20.x"
},
 Debug
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "nodemon server.js"
},
"keywords": [],
"author": "",
"license": "ISC",
"dependencies": {
  "bcryptjs": "^2.4.3",
  "body-parser": "^1.20.2",
  "cors": "^2.8.5",
  "dotenv": "^16.4.5",
  "express": "^4.19.2",
  "jsonwebtoken": "^9.0.2",
  "mongoose": "^8.3.2"
},
"devDependencies": {
  "nodemon": "^3.1.0"
}
```

In conclusion, this project stands as a compelling showcase of advanced full-stack development capabilities, employing industry-standard technologies such as React, TypeScript, Express, and MongoDB. By integrating comprehensive user authentication and detailed product management functionalities, the application not only meets essential business requirements but also adheres to the highest standards of security and data integrity.

The modular design of the backend, coupled with the use of JWTs for secure transmission of information and bcryptjs for robust password hashing, ensures that the application is not only secure but also scalable. This structure supports seamless integration and future enhancements without disrupting existing functionalities.

Deployment on Vercel highlights our commitment to leveraging cutting-edge cloud technologies, offering scalable and efficient solutions that enhance operational reliability and performance. The use of environment variables and CORS policy management further underpins our focus on creating flexible and secure environments suitable for dynamic market conditions.