Web API Assignment Report

Course Code: 1DV027
Course Name: Webben som applikationsplattform
Name: Dongzhu Tan

Introduction

This report details the development of a RESTful API for a second-hand store application. The primary goal was to create a robust, scalable, and secure API that adheres to REST principles and incorporates modern web development practices. The application allows users to manage items, categories, and user accounts in a second-hand store context.

1. HATEOAS Implementation

Here's how I have incorporated HATEOAS:

1.1 Root Endpoint:

The main entry point (`/`) provides links to key resources:

```
router.get('/', (req, res) => {
  res.json({
    message: 'Welcome to Second-Hand Store API. Here you can find links to where you want to go!',
    auth: {
      register: { href: `${req.protocol}://${req.get('host')}/api/v1/auth/register` },
      login: { href: `${req.protocol}://${req.get('host')}/api/v1/auth/login` }
    },
    items: { href: `${req.protocol}://${req.get('host')}/api/v1/items` },
    categories: { href: `${req.protocol}://${req.get('host')}/api/v1/categories` }
  })
})
```

1.2 Resource Listings:

When fetching lists of resources (e.g., items), links for navigation and related actions are inclued:

```
  res.status(200).json({
    items: itemsWithDetails,
    currentPage: data.currentPage,
    totalPages: data.totalPages,
    totalItems: data.totalItems,
    message: data.message || 'Items fetching successful!',
    _links: {
      self: { href: `${req.protocol}://${req.get('host')}${req.originalUrl}` },
      createItem: { href: `${req.protocol}://${req.get('host')}/api/v1/items/create`,
method: 'POST' }
    }
  })
```

1.3 Individual Resources:

Each item includes a link to itself:

```
  _links: {
    self: { href: `${req.protocol}://${req.get('host')}/api/v1/items/${item._id}` }
  }
```

This implementation allows clients to navigate the API dynamically, discovering available actions and related resources without prior knowledge of the API structure. It enhances the API's self-descriptiveness and makes it more flexible to changes.

2. Multiple Representations of Resources

Currently, the API doesn't support multiple representations, but this is how I could implement it:

2.1 Content Negotiation:

Support for multiple content types can be added via the 'Accept' header. For instance:

```
  app.use((req, res, next) => {
    res.format({
      'application/json': () => { // Render JSON representation. },
      'application/xml': () => { // Render XML representation. },
      'text/html': () => { // Render HTML representation.}
      default: () => {
        res.status(406).send('Not Acceptable')
      }
```

```
  })
```

2.2 Serialization Layer:

Create a serialization layer that can convert data models into different formats (JSON, XML, etc.).

3. Authentication Solution

The application authentication solution uses JSON Web Tokens (JWT) for stateless authentication.

3.1 JWT Generation:

Upon successful login, a JWT is created and sent to the client:

```
const accessToken = jwt.sign(payload, privateKey, {
  algorithm: 'RS256',
  expiresIn: process.env.ACCESS_TOKEN_LIFE
})
```

3.2 Token Verification:

Incoming requests are verified using a middleware:

```
jwt.verify(token, publicKey, { algorithms: ['RS256'] }, (err, user) => {
  if (err) {
    // Handle error
  }
  req.user = user
  next()
})
```

3.3 Role-based Access Control:

Access is controlled using middleware based on user roles:

```
export const authenticateAdmin = (req, res, next) => {
  if (req.user && req.user.role === 'admin') {
    next()
  } else {
    res.status(403).json({ message: 'Access denied. Admin access required.' })
  }
}
```

```

### 3a. Alternative Authentication Solutions

I could implement session-based authentication, storing session information on the server and providing the client with a session ID, as I learned in previous courses.

### 3b. Pros and Cons of JWT

Pros:
- Stateless: No need to store session information on the server.
- Rich in Information: Can include user roles and permissions.

Cons:
- Can't be Invalidated: Once issued, a JWT is valid until it expires.
- Complexity: Managing secrets and token lifecycles requires careful handling. In my application, sometimes tokens become invalid, requiring further debugging, which takes time.

## 4. Webhook Implementation

My webhook system allows registered users to subscribe to specific items in my application. Here's how it works:

### 4.1 Registration:

Registered users can register a webhook URL:

```
router.post('/register', (req, res) => {
  const { url } = req.body
    if (!url) {
       return res.status(400).json({ error: 'Webhook URL is required' })
        }
  const id = webhookManager.registerWebhook(url)
  res.status(201).json({ id, message: 'Webhook registered successfully' })
})
```

### 4.2 Event Triggering:

When a event (e.g., price change) occurs, the webhook is triggered:

```
async notifyPriceChange (itemObjectId, newPrice) {
  // ...
  for (const [id, url] of this.webhooks) {
```
```

```
      try {
        const response = await fetch(url, {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ itemObjectId, newPrice, notifications })
        })
        console.log(`Webhook ${id} notified:`, response.status)
      } catch (error) {
        console.error(`Failed to notify webhook ${id}:`, error)
      }
    }
  }
```

This allows real-time notifications for registered users when a product's price changes.

5. Reflections and Potential Improvements

Looking back at this assignment, there are several areas where improvements could be made:

1. Testing: Implement comprehensive unit and integration tests to ensure reliability.

2. Rate Limiting: Add rate limiting to protect the API from abuse and ensure fair usage.

3. Read requirements clearly before coding: When time is limited, I should avoid implementing unnecessary or overly complex features. This would also simplify API testing and debugging.

6. Linguistic Design Rules

To improve API usability, I followed several linguistic design principles:

1. Consistency: I maintain consistent naming conventions across endpoints (e.g., /item', /categories), this makes the API more intuitive and easier to learn.

2. Noun-based Resource Naming: Using nouns for resources (e.g., /items, /categories) aligns with REST principles.

3. Plurality for Collections: Collections use plural nouns (e.g., /items instead of /item).

4. Hierarchical Relationships: Resources are organized hierarchically where appropriate (e.g., /categories/{categoryName}).

5. Versioning: The API is versioned (e.g., /api/v1/), allowing for future changes.

6. HTTP Methods for Actions: The correct HTTP methods (GET, POST, PUT, PATCH, DELETE) are used.

7. Descriptive Error Messages: Error responses provide clear, informative messages.

8. Hyphens improve URI readability (e.g., /register-admin, /users-with-items).

9. Lowercase URIs: All URIs are lowercase.

10. There is no trailing forward slash (/) , underscores (_) or file extensions included in URIs.

11. The query component of a URI are used to paginate collection and get the specific items. (e.g., {{baseUrl}}/api/v1/items?category=clothes&minPrice=50&maxPrice=300&page=1&limit=10)

7. Extra Features

I implemented the following additional features beyond the assignment's core requirements:

1. Role-based Access Control: Implemented different access levels for regular users and admins.

2. Pagination: Supports paginated resource listings.

3. Filtering and Sorting: Implemented filtering and sorting capabilities for item listings.

4. Security Headers: Implemented security headers using Helmet.js.

5. Environment-based Configuration: Used environment variables for configuration, enhancing security and flexibility.