

**C.I.T.L. EXPERIMENT 3**

**Submitted By:**

| Akash Panicker | 2021300089 |
| --- | --- |
| Mahesh Patil | 2021300095 |
| Rohit Phalke | 2021300100 |
| Adwait Purao | 2021300101 |

**Submitted To:**

Prof. Sunil Ghane

**Inventory Management System**

**Aim:**

Create a Restful webservice to demonstrate different HTTP methods.

**Problem Statement:**

Develop an inventory management system for a retail store that efficiently tracks and manages the inventory of products. The system should provide real-time updates on stock levels, generate alerts for low stock items, enable easy addition and removal of products, and offer insights into sales trends to optimize restocking decisions.

**Theory:**

Web services are a foundational technology for enabling communication

and data exchange between different software applications over the

internet. They facilitate interoperability by providing a standardized

method for systems to interact.

SOAP (Simple Object Access Protocol) web services are a protocol-based

approach that uses XML for structuring messages. SOAP offers a strict set

of rules for message format and communication, making it highly secure

and reliable. It provides built-in error handling and comprehensive

protocol support, suitable for enterprise-level applications. However, the

verbosity of XML and the complexity of the protocol can make SOAP

services relatively heavyweight and less efficient for simple, resourcecentric

tasks.

In contrast, RESTful (Representational State Transfer) web services are

an architectural style based on simplicity and standard HTTP methods.

REST emphasizes resource-centric design, statelessness, and a uniform

interface. It uses standard HTTP verbs like GET, POST, PUT, and DELETE

for data manipulation. REST is known for its lightweight nature,

scalability, and ease of use, making it ideal for web and mobile

applications, as well as APIs for the internet of things (IoT).

**What are the benefits of RESTful APIs?**

RESTful APIs include the following benefits:

**Scalability**

* Systems that implement REST APIs can scale efficiently because REST optimizes client-server interactions. Statelessness removes server load because the server does not have to retain past client request information. Well-managed caching partially or completely eliminates some client-server interactions. All these features support scalability without causing communication bottlenecks that reduce performance.

**Flexibility**

* RESTful web services support total client-server separation. They simplify and decouple various server components so that each part can evolve independently. Platform or technology changes at the server application do not affect the client application. The ability to layer application functions increases flexibility even further. For example, developers can make changes to the database layer without rewriting the application logic.

**Independence**

* REST APIs are independent of the technology used. You can write both client and server applications in various programming languages without affecting the API design. You can also change the underlying technology on either side without affecting the communication.

**HTTP METHODS:**

HTTP methods, also known as HTTP verbs, are fundamental to the RESTful

architecture, and they play a crucial role in specifying the desired action to be

performed on a resource. Following HTTP methods are commonly used in REST

APIs and their key features:

* **GET:**

Purpose: Retrieve data from the server.

Idempotent: Repeated GET requests do not change the server's state.

Safe: Should not have any side effects on the server.

Caching: Responses can be cached, reducing the need for redundant

requests.

Parameters: Data can be sent in the URL as query parameters.

* **POST:**

Purpose: Create a new resource on the server.

Not Idempotent: Repeated POST requests may result in multiple resource

creations.

Data Format: Allows for sending complex data in the request body, often in

JSON or XML.

* **PUT:**

Purpose: Update an existing resource or create it if it doesn't exist.

Idempotent: Repeated PUT requests should not have different outcomes.

Full Update: Typically replaces the entire resource with the new data

provided in the request body.

* **PATCH:**

Purpose: Partially update an existing resource.

Idempotent: Repeated PATCH requests with the same data should not have

different outcomes.

Partial Update: Only modifies the fields specified in the request body, leaving

others unchanged.

* **DELETE:**

Purpose: Remove a resource from the server.

Idempotent: Repeated DELETE requests should not change the state after the

initial deletion.

Resource Removal: Permanently removes the specified resource.

* **HEAD:**

Purpose: Retrieve only the headers of a resource, useful for checking

metadata or resource availability.

Idempotent: Like GET, it doesn't change server state, making it safe for

caching and conditional requests.

* **OPTIONS:**

Purpose: Retrieve information about the communication options available

for a resource.

Metadata: Returns information about the supported HTTP methods,

request/response formats, and available headers for a resource

**Screenshots:**

**GET REQUEST**

**//orders by vendors**

**router.get("/orders\_c", async (req, res) => {**

**let email;**

**if (req.cookies) {**

**if (req.cookies.inv\_man) {**

**if (req.cookies.inv\_man.role) {**

**email = req.cookies.inv\_man.email;**

**}**

**} else {**

**return res.status(500).json({ error: "Please login to continue" });**

**}**

**} else {**

**return res.status(500).json({ error: "Please login to continue" });**

**}**

**try {**

**const orders = await Order.find({ c\_email: email });**

**if (!orders) {**

**return res.status(400).json({ error: "No orders found" });**

**}**

**orders.reverse()**

**return res.status(200).json(orders);**

**} catch (error) {**

**console.error(error);**

**return res.status(500).json({ error: "Internal server error" });**

**}**

**});**

**POST REQUEST**

**router.post("/addstock\_c", async (req, res) => {**

**const { email, quantity, pid } = req.body;**

**// console.log("Request Body: ", req.body);**

**if (isNaN(quantity)) {**

**return res.status(422).json({ error: "Invalid request made" });**

**}**

**try {**

**const company = await Company.findOne({ email: email });**

**if (!company) {**

**return res.status(400).json({ error: "Company not found" });**

**}**

**const product = company.products.find((product) => product.pid === pid);**

**if (!product) {**

**return res.status(400).json({ error: "Product not found" });**

**}**

**// Ensure the quantity is valid and subtract it from the product**

**product.quantity += quantity;**

**// vendor.find(product).quantity += quantity;**

**// await vendor.save(); // Save the updated vendor document**

**await Company.replaceOne({ email: email }, company);**

**return res.status(200).json({ message: "Stock added successfully" });**

**} catch (error) {**

**console.error(error);**

**return res.status(500).json({ error: "Internal server error" }); // Handle errors properly**

**}**

**});**

**PUT REQUEST**

**router.put("/updateprofile", async (req, res) => {**

**const { name, email, phone, address, companyGenre, logo, GSTNO, dob } = req.body;**

**let role;**

**if (req.cookies) {**

**if (req.cookies.inv\_man) {**

**if (req.cookies.inv\_man.role) {**

**role = req.cookies.inv\_man.role;**

**}**

**} else {**

**return res.status(500).json({ error: "Please login to continue" });**

**}**

**} else {**

**return res.status(500).json({ error: "Please login to continue" });**

**}**

**if (!name || !phone) {**

**return res.status(422).json({ error: "All fields need to be filled" });**

**}**

**try {**

**const user = await Profile.findOne({ email: email });**

**if (!user) {**

**return res.status(400).send({ error: "User not found" });**

**}**

**user.name = name;**

**user.phone = phone;**

**user.address = address;**

**user.companyGenre = companyGenre;**

**user.logo = logo;**

**user.GSTNO = GSTNO;**

**user.dob = dob;**

**await Profile.replaceOne({ email: email }, user);**

**let user1 = await Vendor.findOne({ email: email });**

**if (!user1) {**

**user1 = await Company.findOne({ email: email });**

**user1.name = name;**

**user1.phone = phone;**

**await Company.replaceOne({ email: email }, user1);**

**}**

**else{**

**user1.name = name;**

**user1.phone = phone;**

**await Vendor.replaceOne({ email: email }, user1);**

**}**

**return res.status(200).json({ msg: "Profile updated successfully" });**

**} catch (error) {**

**console.error(error);**

**return res.status(500).json({ error: "Internal server error" });**

**}**

**});**

**DELETE REQUEST**

**router.delete("/revokeRequest", async (req, res) => {**

**const id = req.body.id;**

**try {**

**const order = await Order.findOne({ \_id: id });**

**if (!order) {**

**return res.status(400).json({ error: "No order found" });**

**}**

**order.status = "Revoked";**

**await order.deleteOne();**

**return res.status(200).json({ msg: "Order revoked successfully" });**

**} catch (error) {**

**console.error(error);**

**return res.status(500).json({ error: "Internal server error" });**

**}**

**});**

**References:**

<https://www.geeksforgeeks.org/restful-web-services/>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

<https://nextjs.org/docs/app>

**Conclusion:**

By conducting this experiment, we gained insights into the concept of RESTful web services, recognizing their necessity and understanding their integration into backend applications. We acquired the knowledge of utilizing RESTful services to develop an API, employing various HTTP methods, and successfully applied this understanding to construct our intended project.