Student id: 2022ht66033

Name – Raj Patil

Subject – Service Oriented Computing

Let's consider an example of a Blogging Platform as the application for designing a service-oriented architecture. The basic capabilities of the application could include:

Basic Capabilities:

1. User Management:
   * Registration: Allowing users to create accounts.
   * Authentication: Verifying the identity of users.
   * Profile Management: Updating user information.
2. Blog Post Management:
   * Create a new blog post.
   * Retrieve details of a specific blog post.
   * Update an existing blog post.
   * Delete a blog post.
3. Comment System:
   * Add a comment to a blog post.
   * View comments for a blog post.
   * Edit or delete own comments.
4. Category/Tag Management:
   * Create a new category/tag for organizing blog posts.
   * Retrieve details of a specific category/tag.
   * Update an existing category/tag.
   * Delete a category/tag.

Resources and REST API Endpoints:

1. User Resource:

* Resource Name: users
* Operations:
  + Create a new user
    - URI: /users
    - Method: POST
    - Request Body: User details (e.g., username, email, password)
    - Response Body: User details with an assigned unique ID
  + Get details of a specific user
    - URI: /users/{userID}
    - Method: GET
    - Response Body: User details
  + Update user information
    - URI: /users/{userID}
    - Method: PUT
    - Request Body: Updated user details
    - Response Body: Updated user details

2. Blog Post Resource:

* Resource Name: blog-posts
* Operations:
  + Create a new blog post
    - URI: /blog-posts
    - Method: POST
    - Request Body: Blog post details (e.g., title, content, author)
    - Response Body: Blog post details with an assigned unique ID
  + Get details of a specific blog post
    - URI: /blog-posts/{postID}
    - Method: GET
    - Response Body: Blog post details
  + Update an existing blog post
    - URI: /blog-posts/{postID}
    - Method: PUT
    - Request Body: Updated blog post details
    - Response Body: Updated blog post details
  + Delete a blog post
    - URI: /blog-posts/{postID}
    - Method: DELETE

3. Comment Resource:

* Resource Name: comments
* Operations:
  + Add a comment to a blog post
    - URI: /blog-posts/{postID}/comments
    - Method: POST
    - Request Body: Comment details (e.g., content, author)
    - Response Body: Comment details with an assigned unique ID
  + View comments for a blog post
    - URI: /blog-posts/{postID}/comments
    - Method: GET
    - Response Body: List of comments
  + Edit or delete own comments
    - URI: /comments/{commentID}
    - Method: PUT (for editing)
    - Method: DELETE (for deletion)

4. Category/Tag Resource:

* Resource Name: categories
* Operations:
  + Create a new category/tag
    - URI: /categories
    - Method: POST
    - Request Body: Category/tag details (e.g., name)
    - Response Body: Category/tag details with an assigned unique ID
  + Get details of a specific category/tag
    - URI: /categories/{categoryID}
    - Method: GET
    - Response Body: Category/tag details
  + Update an existing category/tag
    - URI: /categories/{categoryID}
    - Method: PUT
    - Request Body: Updated category/tag details
    - Response Body: Updated category/tag details
  + Delete a category/tag
    - URI: /categories/{categoryID}
    - Method: DELETE

Additional Considerations:

* Pagination: For endpoints that return lists (e.g., list of blog posts, comments), consider implementing pagination for better performance.
* Search: Implement a search endpoint for users to search for blog posts based on keywords, tags, etc.
* Security: Use HTTPS for secure communication, and consider implementing measures such as rate limiting and input validation.
* Error Handling: Define clear error responses for different scenarios (e.g., unauthorized access, resource not found).

This is a high-level design, and in a real-world scenario, you might need to consider more advanced features, scalability, and optimizations.

2 ---

Let's split the identified capabilities into individual resources and provide a brief description for each:

1. User Resource:

Resource Name: users

Operations:

1. Create a new user
   * URI: /users
   * Method: POST
   * Request Body: User details (e.g., username, email, password)
   * Response Body: User details with an assigned unique ID
2. Get details of a specific user
   * URI: /users/{userID}
   * Method: GET
   * Response Body: User details
3. Update user information
   * URI: /users/{userID}
   * Method: PUT
   * Request Body: Updated user details
   * Response Body: Updated user details

2. Blog Post Resource:

Resource Name: blog-posts

Operations:

1. Create a new blog post
   * URI: /blog-posts
   * Method: POST
   * Request Body: Blog post details (e.g., title, content, author)
   * Response Body: Blog post details with an assigned unique ID
2. Get details of a specific blog post
   * URI: /blog-posts/{postID}
   * Method: GET
   * Response Body: Blog post details
3. Update an existing blog post
   * URI: /blog-posts/{postID}
   * Method: PUT
   * Request Body: Updated blog post details
   * Response Body: Updated blog post details
4. Delete a blog post
   * URI: /blog-posts/{postID}
   * Method: DELETE

3. Comment Resource:

Resource Name: comments

Operations:

1. Add a comment to a blog post
   * URI: /blog-posts/{postID}/comments
   * Method: POST
   * Request Body: Comment details (e.g., content, author)
   * Response Body: Comment details with an assigned unique ID
2. View comments for a blog post
   * URI: /blog-posts/{postID}/comments
   * Method: GET
   * Response Body: List of comments
3. Edit or delete own comments
   * URI: /comments/{commentID}
   * Method: PUT (for editing)
   * Method: DELETE (for deletion)

4. Category/Tag Resource:

Resource Name: categories

Operations:

1. Create a new category/tag
   * URI: /categories
   * Method: POST
   * Request Body: Category/tag details (e.g., name)
   * Response Body: Category/tag details with an assigned unique ID
2. Get details of a specific category/tag
   * URI: /categories/{categoryID}
   * Method: GET
   * Response Body: Category/tag details
3. Update an existing category/tag
   * URI: /categories/{categoryID}
   * Method: PUT
   * Request Body: Updated category/tag details
   * Response Body: Updated category/tag details
4. Delete a category/tag
   * URI: /categories/{categoryID}
   * Method: DELETE

Additional Considerations:

* Each resource has its set of CRUD (Create, Read, Update, Delete) operations.
* Ensure proper authentication and authorization mechanisms for secure access.
* Consider implementing error handling for scenarios like unauthorized access and resource not found.
* Use appropriate HTTP methods (POST, GET, PUT, DELETE) for each operation.
* Implement pagination for endpoints returning lists of items (e.g., list of blog posts, comments).
* Use HTTPS for secure communication.
* Define clear and consistent response formats, preferably in JSON.

This breakdown creates a modular and scalable structure for the service-oriented architecture, allowing for efficient development, maintenance, and future expansion of the blogging platform.

4-----

Let's design the URI, choose the appropriate HTTP method, and specify the headers required for each operation:

1. User Resource:

Operation 1: Create a new user

* URI: /users
* HTTP Method: POST
* Headers: Content-Type (e.g., application/json)
* Request Body (Representation accepted by the server): User details (e.g., username, email, password)
* Response Body (Representation served to the client): User details with an assigned unique ID

Operation 2: Get details of a specific user

* URI: /users/{userID}
* HTTP Method: GET
* Headers: None
* Response Body: User details

Operation 3: Update user information

* URI: /users/{userID}
* HTTP Method: PUT
* Headers: Content-Type
* Request Body: Updated user details
* Response Body: Updated user details

2. Blog Post Resource:

Operation 1: Create a new blog post

* URI: /blog-posts
* HTTP Method: POST
* Headers: Authorization (for authentication), Content-Type
* Request Body: Blog post details (e.g., title, content, author)
* Response Body: Blog post details with an assigned unique ID

Operation 2: Get details of a specific blog post

* URI: /blog-posts/{postID}
* HTTP Method: GET
* Headers: None
* Response Body: Blog post details

Operation 3: Update an existing blog post

* URI: /blog-posts/{postID}
* HTTP Method: PUT
* Headers: Authorization, Content-Type
* Request Body: Updated blog post details
* Response Body: Updated blog post details

Operation 4: Delete a blog post

* URI: /blog-posts/{postID}
* HTTP Method: DELETE
* Headers: Authorization

3. Comment Resource:

Operation 1: Add a comment to a blog post

* URI: /blog-posts/{postID}/comments
* HTTP Method: POST
* Headers: Authorization, Content-Type
* Request Body: Comment details (e.g., content, author)
* Response Body: Comment details with an assigned unique ID

Operation 2: View comments for a blog post

* URI: /blog-posts/{postID}/comments
* HTTP Method: GET
* Headers: Authorization
* Response Body: List of comments

Operation 3: Edit or delete own comments

* URI: /comments/{commentID}
* HTTP Method: PUT (for editing)
* HTTP Method: DELETE (for deletion)
* Headers: Authorization

4. Category/Tag Resource:

Operation 1: Create a new category/tag

* URI: /categories
* HTTP Method: POST
* Headers: Authorization, Content-Type
* Request Body: Category/tag details (e.g., name)
* Response Body: Category/tag details with an assigned unique ID

Operation 2: Get details of a specific category/tag

* URI: /categories/{categoryID}
* HTTP Method: GET
* Headers: Authorization
* Response Body: Category/tag details

Operation 3: Update an existing category/tag

* URI: /categories/{categoryID}
* HTTP Method: PUT
* Headers: Authorization, Content-Type
* Request Body: Updated category/tag details
* Response Body: Updated category/tag details

Operation 4: Delete a category/tag

* URI: /categories/{categoryID}
* HTTP Method: DELETE
* Headers: Authorization

Additional Considerations:

* The "Authorization" header is used for secure operations to ensure that only authenticated users can perform certain actions.
* The "Content-Type" header specifies the format of the data in the request body.
* The absence of specific headers (e.g., "Content-Type" for GET requests) indicates that those headers are not required for the operation.

These URI designs, HTTP methods, and headers provide a clear structure for the RESTful API, adhering to best practices for usability, security, and consistency.

5-----

Design the representation(s) accepted by the client and the representation(s) served to the client. Categorize the resources based on their type.

### Representation Design:

#### 1. User Resource:

##### Representation Accepted by the Server (Request Body for Create and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "username": "Raj", "email": "2022ht66033@wilpbits-pilani.ac.in", "password": "securepassword" }

##### Representation Served to the Client (Response Body for Create, Read, and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "userID": "123", "username": "Raj", "email": "2022ht66033@wilpbits-pilani.ac.in " }

#### 2. Blog Post Resource:

##### Representation Accepted by the Server (Request Body for Create and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "title": "Introduction to RESTful APIs", "content": "This is a detailed guide on designing RESTful APIs...", "author": "john\_doe" }

##### Representation Served to the Client (Response Body for Create, Read, and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "postID": "456", "title": "Introduction to RESTful APIs", "content": "This is a detailed guide on designing RESTful APIs...", "author": "john\_doe" }

#### 3. Comment Resource:

##### Representation Accepted by the Server (Request Body for Create and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "content": "Great post! Very informative.", "author": "alice" }

##### Representation Served to the Client (Response Body for Create, Read, and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "commentID": "789", "content": "Great post! Very informative.", "author": "alice" }

#### 4. Category/Tag Resource:

##### Representation Accepted by the Server (Request Body for Create and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "name": "Technology" }

##### Representation Served to the Client (Response Body for Create, Read, and Update operations):

* Format: JSON
* Example:

jsonCopy code

{ "categoryID": "101", "name": "Technology" }

### Resource Categorization:

#### Type: User-related Resources

* Resource Name: /users

#### Type: Blog Post-related Resources

* Resource Name: /blog-posts

#### Type: Comment-related Resources

* Resource Name: /blog-posts/{postID}/comments
* Resource Name: /comments/{commentID}

#### Type: Category/Tag-related Resources

* Resource Name: /categories

These representation designs and resource categorizations provide a clear understanding of the data format expected by the server and the format served to the client for each type of resource in the service-oriented architecture. The use of JSON for representation offers simplicity and ease of parsing.

Creating a functional prototype of a web application involves both front-end (client-side) and back-end (server-side) development. Below, I'll provide a simplified example using JavaScript for both the front-end (using HTML and JavaScript) and the back-end (using Node.js with Express). Please note that this is a basic demonstration for educational purposes, and a real-world application would involve more robust architecture, security considerations, and data persistence.

### 1. Front-end (HTML and JavaScript):

Create an index.html file:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Blogging Platform</title>

</head>

<body>

<h1>Blogging Platform</h1>

<!-- User Interface for Creating a New Blog Post -->

<h2>Create a New Blog Post</h2>

<form id="createPostForm">

<label for="title">Title:</label>

<input type="text" id="title" required><br>

<label for="content">Content:</label>

<textarea id="content" required></textarea><br>

<label for="author">Author:</label>

<input type="text" id="author" required><br>

<button type="button" onclick="createBlogPost()">Create Post</button>

</form>

<!-- Display Area for Blog Posts -->

<h2>Blog Posts</h2>

<div id="blogPosts"></div>

<script src="frontend.js"></script>

</body>

</html>

Create a frontend.js file:

// Front-end JavaScript code

document.addEventListener('DOMContentLoaded', function () {

// Fetch and display existing blog posts on page load

fetchBlogPosts();

// Function to create a new blog post

window.createBlogPost = function () {

const title = document.getElementById('title').value;

const content = document.getElementById('content').value;

const author = document.getElementById('author').value;

fetch('/blog-posts', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({ title, content, author }),

})

.then(response => response.json())

.then(data => {

console.log('Blog post created:', data);

fetchBlogPosts(); // Refresh the displayed blog posts

})

.catch(error => console.error('Error creating blog post:', error));

};

// Function to fetch and display blog posts

function fetchBlogPosts() {

fetch('/blog-posts')

.then(response => response.json())

.then(data => {

const blogPostsDiv = document.getElementById('blogPosts');

blogPostsDiv.innerHTML = '<h3>Blog Posts</h3>';

data.forEach(post => {

const postDiv = document.createElement('div');

postDiv.innerHTML = `<strong>${post.title}</strong><br>${post.content}<br>Author: ${post.author}<hr>`;

blogPostsDiv.appendChild(postDiv);

});

})

.catch(error => console.error('Error fetching blog posts:', error));

}

});

### 2. Back-end (Node.js with Express):

Install necessary packages:

npm init -y

npm install express body-parser

Create a backend.js file:

// Back-end Node.js code

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

const port = 3000;

// Mock data (in-memory storage, not suitable for production)

const blogPosts = [];

app.use(bodyParser.json());

// Endpoint to create a new blog post

app.post('/blog-posts', (req, res) => {

const { title, content, author } = req.body;

const newPost = { title, content, author };

blogPosts.push(newPost);

res.json(newPost);

});

// Endpoint to fetch all blog posts

app.get('/blog-posts', (req, res) => {

res.json(blogPosts);

});

app.listen(port, () => {

console.log(`Server is running at http://localhost:${port}`);

});

Run the Application:

1. Start the back-end server: node backend.js.
2. Open index.html in a web browser.

This is a simplified example to demonstrate the basic functionality. In a real-world scenario, you would need to implement proper error handling, data validation, user authentication, and potentially use a database for persistent data storage.