API and REST Questions

Q1. Write RESTful API endpoints for managing customer service profiles in a company adopting the 'API-First' approach. Assume fields like id, name, age, and address, with the domain name 'NextGenServices'.

Answer:

Start with the Base URL

In a RESTful design, resources are usually plural nouns. Here, the resource is `customers`, and the company's domain is given as `NextGenServices`. Following the API-first approach, we design the base endpoint as:

```
https://api.nextgenservices.com/customers
```

This means every action on customer profiles will start from this root path.

Create (POST)

To create a new customer profile, we send data to the collection `/customers` using the **POST** method. POST is used because we're adding something new to the collection.

```
POST /customers
{
  "name": "Alice Smith",
  "age": 29,
  "address": "123 Main St"
}
```

Here, we don't send the `id` because it will typically be generated by the system.

Read (GET)

Reading comes in two flavours in REST:

- **All profiles** → Use GET on the collection itself.

```
GET /customers
```

This would return a list of all customer profiles.

- **Single profile** → Use GET on a specific resource by appending its `id`.

```
GET /customers/{id}
```

Update (PUT or PATCH)

To change details of a customer:

- **PUT** replaces the whole resource.
- **PATCH** updates only part of it.
  Here, if we're only updating age and address, `PATCH` is often more precise, but `PUT` is acceptable too if we're okay with replacing the full record.

```
PUT /customers/{id}
{
  "age": 30,
  "address": "456 Oak Lane"
}
```

Delete (DELETE)

Finally, removing a customer profile is done with the **DELETE** method on that resource's path:

```
DELETE /customers/{id}
```

For example: `/customers/101` would delete the customer with `ID 101`.

Putting It All Together

- **Create:** `POST /customers`
- **Read all:** `GET /customers`
- **Read one:** `GET /customers/{id}`
- **Update:** `PUT /customers/{id} (or PATCH /customers/{id})`
- **Delete:** `DELETE /customers/{id}`

Q2. Provide RESTful API endpoints for a library system that tracks books and their availability. Assume the system has fields for title, author, category, and price.

Answer:

Identify the resource

In REST, the first question is: *what is the core resource we're managing*? Here, the resource is clearly `books`, since we're tracking their details (`title`, `author`, `category`, `price`, and `availability`). That means our base URL should be:

```
https://api.library.com/books
```

Plural form (`books`) is used because the API manages a collection of book records.

Create a new book (POST)

If we want to add a new book, we are creating a fresh resource in the collection. In REST, creation is done with **POST** on the collection endpoint:

```
POST /books
{
  "title": "Data Science 101",
  "author": "Prof. Lee",
  "category": "Education",
  "price": 45.0
}
```

Notice that the system will usually generate an `id` for the new book, so the client only sends the descriptive fields.

Read (GET)

Now we think about retrieving data. REST supports two common patterns:

- **All resources:**

```
GET /books
```

Returns a list of all books, possibly with filters (like `?category=Education`).

- **A single resource:**

```
GET /books/{id}
```

Update (PUT)

If a book's details change (say, price or category), we need to update the resource.

- In REST, **PUT** replaces or updates the resource at the given ID.

```
PUT /books/{id}
{
  "price": 40.0,
  "category": "Data"
}
```

If only partial updates are needed, some APIs prefer **PATCH**, but PUT is the safe default.

Delete (DELETE)

Finally, to remove a book record from the system, REST uses the **DELETE** method:

```
DELETE /books/{id}
```

For instance, `DELETE /books/101` would remove the book with ID 101 from the catalog.

Availability

Since the system needs to track whether a book is available or not, the simplest RESTful way is to treat availability as just another field of the book resource. For example, each book record will have something like:

```
{
  "id": 101,
  "title": "Data Science 101",
  "author": "Prof. Lee",
  "category": "Education",
  "price": 45.0,
  "available": true
}
```

- **When you GET a book:** the API response shows whether the book is available.

  Example:

```
{ "id": 101, "title": "Data Science 101", "available": true }
```

Here, `available: true` means the book can be borrowed.

- **When you UPDATE (PUT/PATCH):** you change that field to reflect the new status.

Example: if the book is borrowed:

```
{ "available": false }
```

Later, when it's returned, you update it back to `true`.

So instead of creating a separate "availability" endpoint, we embed it as part of the book resource. That way, availability status always travels with the rest of the book's details.

Putting it all together

- **Create a book:** `POST /books`
- **Get all books:** `GET /books`
- **Get one book:** `GET /books/{id}`
- **Update book:** `PUT /books/{id}`
- **Delete book:** `DELETE /books/{id}`

Q3. Design APIs for a travel booking system that includes CRUD operations for flights and hotels. Use the domain name 'TravelMaster'.

Answer:
Identify resources

The system manages two main resources: `flights` and `hotels`. In `REST`, each resource gets its own collection endpoint, so we'll have:

```
https://api.travelmaster.com/flights
https://api.travelmaster.com/hotels
```

The domain name is given (`TravelMaster`), so all endpoints live under `api.travelmaster.com`.

CRUD for Flights

Start with the resource

The main resource here is **flights**. In REST, resources are represented as **nouns in plural form**, so we'll use `/flights` as the base path.

Create a flight → `POST /flights`

- When we want to add a **new flight**, we're not targeting a specific flight yet; we're asking the server to create one in the collection. That's why we use **POST** on `/flights`.

Example request

```
POST /flights
{
  "origin": "NYC",
  "destination": "LHR",
  "departAt": "2025-12-14T09:30:00Z",
  "price": 540.00
}
```

Here we don't send an `id`, because the server will generate it.

Read flights → `GET`

Retrieving is always done with **GET**. We have two cases:

- To see *all flights*, use:

```
GET /flights
```

This could even support filters like `?origin=NYC&destination=LHR`.

To see *one specific flight*, we need to tell the server which flight by including its `id`:

GET /flights/{id}

For example: `GET /flights/2025`.

Example response for a single flight might look like:

```
{
  "id": 2025,
  "origin": "NYC",
  "destination": "LHR",
  "departAt": "2025-12-14T09:30:00Z",
  "price": 540.00
}
```

Update a flight → PUT /flights/{id}

If the flight details change (say, departure time or price), we're updating an existing record. REST uses **PUT** when replacing or updating a specific resource.

Example request

```
PUT /flights/2025
{
  "origin": "NYC",
  "destination": "LHR",
  "departAt": "2025-12-14T10:00:00Z",
  "price": 520.00
}
```

Why include `{id}`? Because without it, the server wouldn't know which flight to update.

Delete a flight → `DELETE /flights/{id}`

To remove a flight from the system, we use **DELETE** on the specific resource:

```
DELETE /flights/2025
```

This tells the server to remove the record with ID 2025.

Putting it together

So the logic is:

- **POST /flights** → because we're creating a new item in the collection.
- **GET /flights** → because we want the whole collection.
- **GET /flights/{id}** → because we want one item by its identifier.
- **PUT /flights/{id}** → because we want to modify one item.
- **DELETE /flights/{id}** → because we want to remove one item.

CRUD for Hotels

Identify the resource

The resource is hotels. In REST we use a plural noun for the collection, so everything hangs off:

```
/hotels
```

Create a hotel → POST /hotels

We're adding a new hotel to the collection, so we use POST on the collection path.

Example request

```
POST /hotels
Content-Type: application/json

{
  "name": "The Riverside Inn",
  "city": "London",
  "stars": 4,
  "basePrice": 129.00,
  "currency": "GBP"
}
```

Typical response

```
HTTP/1.1 201 Created
Location: /hotels/3107

{
  "id": 3107,
  "name": "The Riverside Inn",
  "city": "London",
  "stars": 4,
  "basePrice": 129.00,
  "currency": "GBP"
}
```

Read hotels → GET

Reading doesn't change state, so we use **GET**. Two common cases:

- **All hotels** (optionally filterable):

```
GET /hotels
```

Single hotel by identifier:

```
GET /hotels/{id}
```

Example: `GET /hotels/3107`

Example single-hotel response

```
{
  "id": 3107,
  "name": "The Riverside Inn",
  "city": "London",
  "stars": 4,
  "basePrice": 129.00,
  "currency": "GBP"
}
```

Update a hotel → PUT /hotels/{id} (or PATCH for partial)

We're changing an existing record, so we target the specific resource with its `{id}`.

Full update with PUT

```
PUT /hotels/3107
Content-Type: application/json

{
  "name": "The Riverside Inn",
  "city": "London",
  "stars": 5,
  "basePrice": 149.00,
  "currency": "GBP"
}
```

Partial update with PATCH

```
PATCH /hotels/3107
Content-Type: application/json

{
  "stars": 5,
  "basePrice": 149.00
}
```

Delete a hotel → DELETE /hotels/{id}

Removal of a specific record uses DELETE on its resource path:

```
DELETE /hotels/3107
```

Typical response

```
HTTP/1.1 204 No Content
```

Putting it together (Hotels)

- **Create:** `POST /hotels`
- **Read all:** `GET /hotels`
- **Read one:** `GET /hotels/{id}`
- **Update:** `PUT /hotels/{id}` **(or** `PATCH /hotels/{id}` **for partial)**
- **Delete:** `DELETE /hotels/{id}`

Q4. Write the API design for an e-commerce site with CRUD operations for product catalog, user profiles, and orders.

Answer:

Name the core resources

An e-commerce MVP revolves around three nouns: products, users, and orders. These become first-class resources with their own collections and item endpoints:

- **Products represent what can be bought.**
- **Users represent customers and their profiles.**
- **Orders represent purchases made by users.**

Pick a clean base URL and versioning

Stable clients need stable URLs. Use a versioned base so future changes don't break existing apps:

```
Base URL: https://api.shoponline.com/v1
Content-Type: application/json
```

Map CRUD semantics to HTTP verbs

CRUD fits naturally onto REST:

- **Create** → POST to a collection.
- **Read** → GET collection or single item.
- **Update** → PUT (replace) or PATCH (partial) on a single item.
- **Delete** → DELETE a single item.

Sketch the data models (so endpoints return something useful)

Thinking in **minimal but practical** fields keeps the design coherent.

Product

```
{
  "id": 202,
  "sku": "TEE-BLK-XL",
  "name": "Classic Tee",
  "description": "Soft cotton T-shirt",
  "price": 19.99,
  "currency": "GBP",
  "stock": 150,
  "status": "active",               // active | archived
  "category_id": 12,
  "images": ["https://.../p202-front.jpg"],
  "created_at": "2025-09-01T10:20:30Z",
  "updated_at": "2025-09-10T15:42:05Z"
}
```

User

```
{
  "id": 10,
  "email": "alex@example.com",
  "name": "Alex Green",
  "phone": "+44 20 1234 5678",
  "addresses": [
    {
      "id": 501,
      "label": "Home",
      "line1": "42 Green Rd",
      "city": "London",
      "postcode": "E1 6AN",
      "country": "GB"
    }
  ],
  "default_address_id": 501,
  "created_at": "2025-08-30T08:00:00Z"
}
```

Order

Orders are multi-item in real shops, so we model an array of items rather than a single `product_id`. This is more flexible than the one-product example and still easy to use.

```
{
  "id": 9001,
  "user_id": 10,
  "status": "placed",               // draft | placed | paid |
shipped | delivered | cancelled
  "items": [
    { "product_id": 202, "name": "Classic Tee", "unit_price": 19.99,
"quantity": 2, "subtotal": 39.98 },
    { "product_id": 305, "name": "Logo Cap",    "unit_price": 12.50,
"quantity": 1, "subtotal": 12.50 }
  ],
  "currency": "GBP",
  "amounts": {
    "items_total": 52.48,
    "shipping_fee": 3.99,
    "tax": 10.50,
    "grand_total": 66.97
  },
  "shipping_address": { "line1": "42 Green Rd", "city": "London",
"postcode": "E1 6AN", "country": "GB" },
  "created_at": "2025-09-13T09:05:00Z",
  "updated_at": "2025-09-13T09:05:00Z"
}
```

Write the endpoints (CRUD for each resource)

Products

```
GET    /v1/products
POST   /v1/products
GET    /v1/products/{product_id}
PUT    /v1/products/{product_id}
PATCH  /v1/products/{product_id}
DELETE /v1/products/{product_id}       // typically "archive" in
```

```
commerce; you may soft-delete
```

Create example:

```
POST /v1/products
{
  "sku": "TEE-BLK-XL",
  "name": "Classic Tee",
  "description": "Soft cotton T-shirt",
  "price": 19.99,
  "currency": "GBP",
  "stock": 150,
  "status": "active",
  "category_id": 12,
  "images": ["https://.../p202-front.jpg"]
}
```

Users

```
GET    /v1/users                       // admin only
POST   /v1/users                       // sign-up
GET    /v1/users/{user_id}             // self or admin
PATCH  /v1/users/{user_id}             // self or admin
DELETE /v1/users/{user_id}             // admin (or GDPR erase
workflow)
GET    /v1/me                          // convenience: current user
PATCH  /v1/me
GET    /v1/me/orders                   // current user's orders
```

Update example:

```
PATCH /v1/me
{ "name": "Alex G", "phone": "+44 20 1234 5678" }
```

Orders

```
GET    /v1/orders                      // admin; users see only
their own with ?user_id=me
```

```
POST   /v1/orders                        // create an order
GET    /v1/orders/{order_id}
PATCH  /v1/orders/{order_id}             // update permissible fields
(e.g., status by admin; address before shipping)
DELETE /v1/orders/{order_id}             // optional; often replaced
by explicit cancellation
POST   /v1/orders/{order_id}/cancel      // domain-friendly way to
"delete"
```

Create example (idempotent):

```
POST /v1/orders
Idempotency-Key: 5b8e1f1a-0b2a-4c21-9c8f-1f4eaa21b8aa
{
  "user_id": 10,
  "items": [
    { "product_id": 202, "quantity": 2 },
    { "product_id": 305, "quantity": 1 }
  ],
  "shipping_address": {
    "line1": "42 Green Rd",
    "city": "London",
    "postcode": "E1 6AN",
    "country": "GB"
  }
```

}

Response:

```
201 Created
Location: /v1/orders/9001
{
  "id": 9001,
  "status": "placed",
  "user_id": 10,
  "items": [
    { "product_id": 202, "name": "Classic Tee", "unit_price": 19.99,
"quantity": 2, "subtotal": 39.98 },
    { "product_id": 305, "name": "Logo Cap",    "unit_price": 12.50,
"quantity": 1, "subtotal": 12.50 }
  ],
  "currency": "GBP",
  "amounts": { "items_total": 52.48, "shipping_fee": 3.99, "tax":
10.50, "grand_total": 66.97 },
  "shipping_address": { "line1": "42 Green Rd", "city": "London",
"postcode": "E1 6AN", "country": "GB" },
  "created_at": "2025-09-13T09:05:00Z"
}
```