Understanding REST

What is REST?

REST stands for Representational State Transfer. It is a software architectural style followed when designing APIs. It proposes a set of rules web developers should follow when building APIs. REST was discovered by Roy Fielding and was presented first in 2000 in his famous dissertation. JSON or XML is used to pass the data. REST is lightweight, scalable and maintainable.

The 2 main important rules we should follow when designing RESTful services are:

1. Use HTTP Request Verbs to make requests.

2. Use specific patterns of routes/endpoint URLs to make requests.

There are 5 HTTP request verbs that should be used while making requests.
They are:

HTTP Request verbs

1. **GET**: Used to fetch a new resource.

2. **POST**: Used to insert a new resource.

3. **PUT**: Used to replace an existing resource.

4. **PATCH**: Used to update an existing resource.

5. **DELETE**: Used to delete a resource.

Why REST?

At present, there is a huge number of technologies used when developing programs. So it is not possible for a developer to know all the programming languages. So there should be a simple lightweight method to communicate between client and server irrespective of the platform. Therefore REST enables web applications that are built using various programming languages to communicate with each other.

The second main reason is applications are slowly moving to the cloud and these cloud providers provide a lot of API's based on the RESTful architecture. Because all Cloud-based architectures are built on the REST concept, it makes more sense to program web services using the REST services architecture to get the most out of Cloud-based services.

Constraints of REST

1. **Client Server -** The client sends the request to the web service on the server. The server would either reject the request or accept the request and respond to the client. This clearly separates user interfaces and services, making the client portable. HTTP stack is used as the communication platform to send requests and responses**.**
2. **Stateless —** The server doesn't store the state of the client. The session state will be kept on the client-side. The client asks a question by providing all the necessary information and the server responds with the answer. The server will respond to each scenario independently and doesn't remember the previous question-answer scenario.
3. **Cacheable —** Cacheable concept is implemented on the client to store requests which have already been sent to the server. Therefore if the client makes the same request, instead of going to the server, it will go to the cache and retrieve the information needed. This reduces the amount of network traffic between the client and the server.
4. **Layered System - A**ny extra layer, such as a middleware layer, may be placed between the client and the real server hosting the RESTFul web service in a layered system.
5. **Uniform Interface** — A uniform interface between components so that information is transferred in a standard format. HTTP web layer is the underlying layer of RESTful web services
6. **Code on demand** — This is an optional constraint. The client functionality can be extended by sending executable code from the server to the client when requested in the form of applets or scripts.

How RESTful APIs Work –

A RESTful API relies on stateless client-server communication, connecting the web-based client and server through HTTP protocol. When clients want to perform an action or retrieve data, they make a request to the server.

This process typically involves the following HTTP methods, which correspond to create, read, update, and delete (CRUD) operations:

- **GET:** Requests a representation of the specified resource. Requests using GET should only retrieve data and have no other effect.
- **POST:** Submits an entity to the specified resource, often causing a change in state or side effects on the server.
- **PUT:** Replaces all current representations of the target resource with the request payload.
- **DELETE:** Removes the specified resource.
- **PATCH:** Partially updates a resource.

To better understand, let's take the example of a RESTful API designed for a bookstore:

- **GET `/books`:** Lists all the books available in the bookstore.
- **POST `/books`:** Adds a new book to the bookstore.
- **GET `/books/{id}`:** Retrieves the details of the book with the given ID.
- **PUT `/books/{id}`:** Updates the details of the book with the given ID.
- **DELETE `/books/{id}`:** Deletes the book with the given ID.

Responses

When a client sends a request to a RESTful API, it expects a response. This response not only includes the requested data (in case of a GET request) or the outcome of an operation (in case of POST, PUT, PATCH, or DELETE requests) but also a relevant HTTP status code.

Status codes are grouped into the following categories:

- `2xx` Success codes, e.g., `200 OK` or `201 Created`
- `3xx` Redirection codes, indicating that further action needs to be taken by the client
- `4xx` Client error codes, e.g., `404 Not Found` or `401 Unauthorized`
- `5xx` Server error codes, e.g., `500 Internal Server Error`

The response body typically contains the data in a standard format such as JSON or XML, although JSON is more common due to its lighter weight and ease of use with JavaScript.

Endpoints

Endpoints are the specific addresses where the resources can be accessed by the client. An endpoint is defined by its URI and the HTTP method used, which together define the action performed on the resource.

Following our library example, some endpoints might be:

- `POST /api/books` to create a new book
- `GET /api/books` to retrieve all books

- `GET /api/books/{id}` to retrieve a book by ID
- `PUT /api/books/{id}` to update a book by ID
- `DELETE /api/books/{id}` to delete a book by ID

Real-world Example: Building a Simple RESTful API

Setup

First, ensure Node.js and npm (Node Package Manager) are installed on your system. Then, set up a new Node.js project and install Express:

```
mkdir todo-api
cd todo-api
npm init -y
npm install express --save
```

Now, create an `index.js` file in your project directory with the following basic Express setup:

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

app.listen(port, () => {
  console.log(`To-Do API server listening at http://localhost:${port}`);
});
```

Define a Model

In a real-world application, you would typically interact with a database, but for simplicity, this example will use an in-memory array to store to-do items:

```
let todos = [
  { id: 1, title: 'Do homework', completed: false },
  { id: 2, title: 'Read a book', completed: false },
];
```

Implementing CRUD Operations

CREATE a To-Do Item

To create a new to-do item, define a POST endpoint:

```
app.post('/todos', (req, res) => {
  const { title } = req.body;
  const newTodo = {
    id: todos.length + 1,
    title: title,
    completed: false
  };
  todos.push(newTodo);
  res.status(201).json(newTodo);
});
```

READ To-Do Items

To retrieve all to-do items and a specific to-do item by ID, define two GET endpoints:

```
// Get all to-do items
app.get('/todos', (req, res) => {
  res.json(todos);
```

```
});

// Get a single to-do item
app.get('/todos/:id', (req, res) => {
  const todo = todos.find(t => t.id === parseInt(req.params.id));
  if (!todo) return res.status(404).send('The to-do item was not found.');
  res.json(todo);
});
```

UPDATE a To-Do Item

To update a to-do item, define a PUT endpoint:

```
app.put('/todos/:id', (req, res) => {
  let todo = todos.find(t => t.id === parseInt(req.params.id));
  if (!todo) return res.status(404).send('The to-do item was not found.');

  const { title, completed } = req.body;
  todo.title = title !== undefined ? title : todo.title;
  todo.completed = completed !== undefined ? completed : todo.completed;
  res.json(todo);
});
```

DELETE a To-Do Item

Finally, to delete a to-do item, define a DELETE endpoint:

```
app.delete('/todos/:id', (req, res) => {
  const todoIndex = todos.findIndex(t => t.id === parseInt(req.params.id));
  if (todoIndex === -1) return res.status(404).send('The to-do item was not
found.');

  const deletedTodo = todos.splice(todoIndex, 1);
  res.json(deletedTodo);
});
```

Testing the API

You can test this API using various tools such as Postman, cURL, or even through the browser for GET requests.

For instance, using cURL you would test the GET endpoint as follows:

```
curl http://localhost:3000/todos
```

And to create a new to-do using cURL:

```
curl -X POST -H "Content-Type: application/json" -d '{"title":"Learn RESTful APIs"}' http://localhost:3000/todos
```

This example outlines a simple RESTful API for a to-do application. Although the example is basic and doesn't interact with a real database or implement authentication, it demonstrates the fundamental operations and patterns that underpin most RESTful services.

Remember, a real-world API would require additional considerations such as input validation, error handling, persistent data storage, security measures, and comprehensive testing to ensure reliability and robustness.