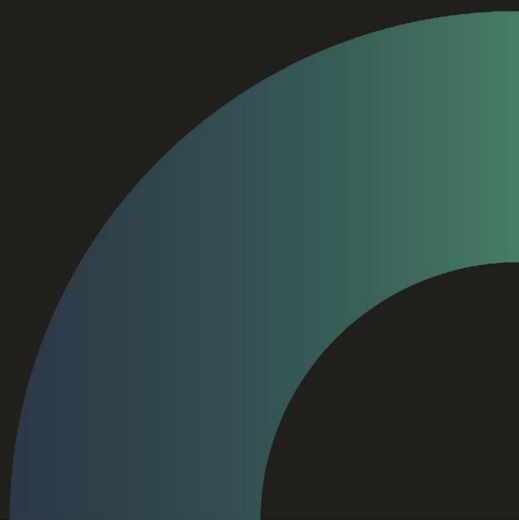


Full Stack Web Development

REST API

-
- Deep Dive into REST API
 - Request, Response and Authentication
 - Routing
 - Middleware
 - REST API Naming Convention
- 

REST Architectural Principle

-
- Uniform interface
 - Statelessness
 - Cacheability
 - Layered system
 - Code on demand

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

Uniform interface imposes four architectural constraints:

- Requests should identify resources. They do so by using a uniform resource identifier.
- Clients have enough information in the resource representation to modify or delete the resource if they want to. The server meets this condition by sending metadata that describes the resource further.
- Clients receive information about how to process the representation further. The server achieves this by sending self-descriptive messages that contain metadata about how the client can best use them.
- Clients receive information about all other related resources they need to complete a task. The server achieves this by sending hyperlinks in the representation so that clients can dynamically discover more resources.

Statelessness means that every HTTP request happens in complete isolation. When the client makes an HTTP request, it includes all information necessary for the server to fulfill the request.

The server never relies on information from previous requests from the client. If any such information is important then the client will send that as part of the current request.

REST API support caching, which is the process of storing some responses on the client or on an intermediary to improve server response time. For example, suppose that you visit a website that has common header and footer images on every page. Every time you visit a new website page, the server must resend the same images.

To avoid this, the client caches or stores these images after the first response and then uses the images directly from the cache. REST API control caching by using API responses that define themselves as cacheable or noncacheable.

Layered system

In a layered system architecture, the client can connect to other authorized intermediaries between the client and server, and it will still receive responses from the server. Servers can also pass on requests to other servers.

You can design your REST API to run on several servers with multiple layers such as security, application, and business logic, working together to fulfill client requests. These layers remain invisible to the client.

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

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.

RESTful APIs require requests to contain the following main components:

Unique resource identifier

The server identifies each resource with unique resource identifiers. For REST services, the server typically performs resource identification by using a Uniform Resource Locator (URL). The URL specifies the path to the resource. A URL is similar to the website address that you enter into your browser to visit any webpage. The URL is also called the request endpoint and clearly specifies to the server what the client requires.

Method

Developers often implement RESTful APIs by using the Hypertext Transfer Protocol (HTTP). An HTTP method tells the server what it needs to do to the resource. (GET, POST, PUT, DELETE, etc)

HTTP headers

Request headers are the metadata exchanged between the client and server. For instance, the request header indicates the format of the request and response, provides information about request status, and so on.

Data

REST API requests might include data for the POST, PUT, and other HTTP methods to work successfully.

Parameters

REST API requests can include parameters that give the server more details about what needs to be done. The following are some different types of parameters:

- Path parameters that specify URL details.
- Query parameters that request more information about the resource.
- Cookie parameters that authenticate clients quickly.

REST principles require the server response to contain the following main components:

Status line

The status line contains a three-digit status code that communicates request success or failure. For instance, 2XX codes indicate success, but 4XX and 5XX codes indicate errors. 3XX codes indicate URL redirection.

The following are some common status codes:

- 200: Generic success response
- 201: POST method success response
- 400: Incorrect request that the server cannot process
- 404: Resource not found
- Etc ...

Message body

The response body contains the resource representation. The server selects an appropriate representation format based on what the request headers contain. Clients can request information in XML or JSON formats, which define how the data is written in plain text. For example, if the client requests the name and age of a person named John, the server returns a JSON representation as follows:

```
'{"name":"John", "age":30}'
```

Headers

The response also contains headers or metadata about the response. They give more context about the response and include information such as the server, encoding, date, and content type.

REST API Authentication Method

A REST API must authenticate requests before it can send a response. Authentication is the process of verifying an identity. For example, you can prove your identity by showing an ID card or driver's license. Similarly, RESTful service clients must prove their identity to the server to establish trust.

HTTP authentication

HTTP defines some authentication schemes that you can use directly when you are implementing REST API. The following are two of these schemes:

Basic authentication

In basic authentication, the client sends the username and password in the request header. It encodes them with base64, which is an encoding technique that converts the pair into a set of 64 characters for safe transmission.

Bearer authentication

The term bearer authentication refers to the process of giving access control to the token bearer. The bearer token is typically an encrypted string of characters that the server generates in response to a login request. The client sends the token in the request headers to access resources.

API keys

API keys are another option for REST API authentication. In this approach, the server assigns a unique generated value to a first-time client. Whenever the client tries to access resources, it uses the unique API key to verify itself. API keys are less secure because the client has to transmit the key, which makes it vulnerable to network theft.

OAuth

OAuth combines passwords and tokens for highly secure login access to any system. The server first requests a password and then asks for an additional token to complete the authorization process. It can check the token at any time and also over time with a specific scope and longevity.

Routing with ExpressJS

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.



```
app.METHOD(PATH, HANDLER);
```

- app is an instance of express.
- METHOD is an HTTP request method, in lowercase.
- PATH is a path on the server.
- HANDLER is the function executed when the route is matched.

Routing with ExpressJS




```
app.get("/", (req, res) => {  
  res.send("Hello World!");  
});  
  
app.post("/", (req, res) => {  
  res.send("Got a POST request");  
});  
  
app.put("/user", (req, res) => {  
  res.send("Got a PUT request at /user");  
});  
  
app.delete("/user", (req, res) => {  
  res.send("Got a DELETE request at /user");  
});
```

Route Methods

A route method is derived from one of the HTTP methods, and is attached to an instance of the express class.

The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.



```
// GET method route
app.get("/", (req, res) => {
  res.send("GET request to the homepage");
});

// POST method route
app.post("/", (req, res) => {
  res.send("POST request to the homepage");
});
```

Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

The characters `?`, `+`, `*`, and `()` are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

If you need to use the dollar character (`$`) in a path string, enclose it escaped within (`[` and `]`). For example, the path string for requests at `"/data/$book"`, would be `"/data/([\$])book"`.

Query strings are not part of the route path.

Route Paths



```
// This route path will match requests to the root route, /.
app.get("/", (req, res) => {
  res.send("root");
});

// This route path will match requests to /about.
app.get("/about", (req, res) => {
  res.send("about");
});

// This route path will match requests to /random.text.
app.get("/random.text", (req, res) => {
  res.send("random.text");
});
```

Route Paths



```
// This route path will match acd and abcd.  
app.get("/ab?cd", (req, res) => {  
  res.send("ab?cd");  
});  
  
// This route path will match abcd, abbcd, abbbcd, and so on.  
app.get("/ab+cd", (req, res) => {  
  res.send("ab+cd");  
});  
  
// This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.  
app.get("/ab*cd", (req, res) => {  
  res.send("ab*cd");  
});  
  
// This route path will match /abe and /abcde.  
app.get("/ab(cd)?e", (req, res) => {  
  res.send("ab(cd)?e");  
});
```

Route Paths



```
// This route path will match anything with an "a" in it (using Regex)
app.get(/a/, (req, res) => {
  res.send("/a/");
});

// This route path will match butterfly and dragonfly, but not butterflyman, dragonflyman, and so on.
app.get(/.*fly$/, (req, res) => {
  res.send("/.*fly$/");
});
```

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the **req.params** object, with the name of the route parameter specified in the path as their respective keys.

```
// Route path : /users/:userId/books/:bookId
// Request URL : http://localhost:3000/users/34/books/8989
// Req.params : { "userId": "34", "bookId": "8989" }

app.get("/users/:userId/books/:bookId", (req, res) => {
  res.send(req.params);
});
```

Route handlers, is a function that executed when the route is matched. You can provide multiple callback functions that behave like middleware to handle a request.

The only exception is that these callbacks might invoke **next('route')** to bypass the remaining route callbacks. You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.

Route Handlers

```

● ● ●

// A single callback function can handle a route. For example:
app.get("/example/a", (req, res) => {
  res.send("Hello from A!");
});

// More than one callback function can handle a route
// (make sure you specify the next object). For example:
app.get(
  "/example/b",
  (req, res, next) => {
    console.log("the response will be sent by the next function ...");
    next();
  },
  (req, res) => {
    res.send("Hello from B!");
  }
);
```

Response Methods

The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

Method	Description
<u>res.download()</u>	Prompt a file to be downloaded.
<u>res.end()</u>	End the response process.
<u>res.json()</u>	Send a JSON response.
<u>res.jsonp()</u>	Send a JSON response with JSONP support.
<u>res.redirect()</u>	Redirect a request.
<u>res.render()</u>	Render a view template.
<u>res.send()</u>	Send a response of various types.
<u>res.sendFile()</u>	Send a file as an octet stream.
<u>res.sendStatus()</u>	Set the response status code and send its string representation as the response body.
<u>res.status()</u>	Sets the HTTP status for the response.

A router object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.



```
const router = express.Router();

// invoked for any requests passed to this router
router.use((req, res, next) => {
  // .. some logic here .. like any other middleware
  next();
});

// will handle any request that ends in /events
// depends on where the router is "use()'d"
router.get("/events", (req, res, next) => {
  // ..
});

.
.
.

// only requests to /calendar/* will be sent to our "router"
app.use("/calendar", router);
```

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next middleware function in the application's request-response cycle. The **next middleware function** is commonly denoted by a variable named **next**.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call **next()** to pass control to the next middleware function. Otherwise, the request will be left hanging.

Application-level Middleware

Bind application-level middleware to an instance of the app object by using the **app.use()**.



```
// This example shows a middleware function with no mount path.  
// The function is executed every time the app receives a request.  
app.use((req, res, next) => {  
  console.log("Time:", Date.now());  
  next();  
});  
  
// This example shows a middleware function mounted on the /user/:id path.  
// The function is executed for any type of HTTP request on the /user/:id path.  
app.use("/user/:id", (req, res, next) => {  
  console.log("Request Type:", req.method);  
  next();  
});
```

Router-level Middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of **express.Router()**.

```
// A middleware function with no mount path.  
// This code is executed for every request to the router.  
router.use((req, res, next) => {  
  console.log("Time:", Date.now());  
  next();  
});  
  
// A middleware sub-stack shows request info  
// for any type of HTTP request to the /user/:id path  
router.use(  
  "/user/:id",  
  (req, res, next) => {  
    console.log("Request URL:", req.originalUrl);  
    next();  
  },  
  (req, res, next) => {  
    console.log("Request Type:", req.method);  
    next();  
  }  
);
```


Error-handling Middleware

Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature (err, req, res, next):



```
app.use((err, req, res, next) => {  
  console.error(err.stack);  
  res.status(500).send("Something broke!");  
});
```

Third-party middleware

Use third-party middleware to add functionality to Express apps. Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.



```
// npm install cookie-parser

import express from "express";
import cookieParser from "cookie-parser";

const app = express();

// load the cookie-parsing middleware
app.use(cookieParser());
```

REST API Naming Convention

- Use nouns to represent resources
- Use forward slash (/) to indicate hierarchical relationships
- Do not use trailing forward slash (/) in URIs
- Use hyphens (-) to improve the readability of URIs
- Do not use underscores (_)
- Use lowercase letters in URIs
- Do not use file extensions
- Never use CRUD function names in URIs
- Use query component to filter URI collection

In REST, the primary data representation is called resource. Having a consistent and robust REST resource naming strategy – will prove one of the best design decisions in the long term.

- **Singleton and Collection Resources.** A resource can be a singleton or a collection.
 - For example, “customers” is a collection resource and “customer” is a singleton resource (in a banking domain).
 - We can identify “customers” collection resource using the URI “/customers”. We can identify a single “customer” resource using the URI “/customers/{customerId}”.
- **Collection and Sub-collection Resources.** A resource may contain sub-collection resources also.
 - For example, sub-collection resource “accounts” of a particular “customer” can be identified using the URI “/customers/{customerId}/accounts” (in a banking domain).
 - Similarly, a singleton resource “account” inside the sub-collection resource “accounts” can be identified as follows: “/customers/{customerId}/accounts/{accountId}”.
- REST APIs use Uniform Resource Identifiers (URIs) to address resources. REST API designers should create URIs that convey a REST API’s resource model to the potential clients of the API. When resources are named well, an API is intuitive and easy to use. If done poorly, that same API can be challenging to use and understand.

Use nouns to represent resources

RESTful URI should refer to a resource that is a thing (noun) instead of referring to an action (verb) because nouns have properties that verbs do not have – similarly, resources have attributes. Some examples of a resource are:

- Users of the system
- User Accounts
- Network Devices etc.

and their resource URIs can be designed as below :

```
http://api.example.com/device-management/managed-devices
```

```
http://api.example.com/device-management/managed-devices/{device-id}
```

```
http://api.example.com/user-management/users
```

```
http://api.example.com/user-management/users/{id}
```

Use forward slash (/) to indicate hierarchical relationships

The forward-slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources.

```
http://api.example.com/device-management
```

```
http://api.example.com/device-management/managed-devices
```

```
http://api.example.com/device-management/managed-devices/{id}
```

```
http://api.example.com/device-management/managed-devices/{id}/scripts
```

```
http://api.example.com/device-management/managed-devices/{id}/scripts/{id}
```

REST API Naming Convention

Do not use trailing forward slash (/) in URIs

As the last character within a URI's path, a forward slash (/) adds no semantic value and may confuse. It's better to drop it from the URI.

```
http://api.example.com/device-management/managed-devices/
```

```
http://api.example.com/device-management/managed-devices /*This is much better version*/
```

REST API Naming Convention

Use hyphens (-) to improve the readability of URIs

To make your URIs easy for people to scan and interpret, use the hyphen (-) character to improve the readability of names in long path segments.

```
http://api.example.com/device-management/managed-devices/
```

```
http://api.example.com/device-management/managed-devices /*This is much better version*/
```


Do not use underscores (_)

It's possible to use an underscore in place of a hyphen to be used as a separator – But depending on the application's font, it is possible that the underscore (_) character can either get partially obscured or completely hidden in some browsers or screens.

To avoid this confusion, use hyphens (-) instead of underscores (_).

```
http://api.example.com/inventory-management/managed-entities/{id}/install-script-location  
//More readable
```

```
http://api.example.com/inventory-management/managedEntities/{id}/installScriptLocation  
//Less readable
```

REST API Naming Convention

Use lowercase letters in URIs

When convenient, lowercase letters should be consistently preferred in URI paths.

`http://api.example.org/my-folder/my-doc` //1

`HTTP://API.EXAMPLE.ORG/my-folder/my-doc` //2

`http://api.example.org/My-Folder/my-doc` //3

Do not use file extensions

File extensions look bad and do not add any advantage. Removing them decreases the length of URIs as well. No reason to keep them.

Apart from the above reason, if you want to highlight the media type of API using file extension, then you should rely on the media type, as communicated through the **Content-Type** header, to determine how to process the body's content.

```
http://api.example.com/device-management/managed-devices.xml /*Do not use it*/  
http://api.example.com/device-management/managed-devices /*This is correct URI*/
```

Never use CRUD function names in URIs

We should not use URIs to indicate a CRUD function. URIs should only be used to uniquely identify the resources and not any action upon them.

We should use HTTP request methods to indicate which CRUD function is performed.

```
// Get all devices
HTTP GET  http://api.example.com/device-management/managed-devices
// Create new Device
HTTP POST http://api.example.com/device-management/managed-devices
// Get device for given Id
HTTP GET  http://api.example.com/device-management/managed-devices/{id}
// Update device for given Id
HTTP PUT  http://api.example.com/device-management/managed-devices/{id}
// Delete device for given Id
HTTP DELETE http://api.example.com/device-management/managed-devices/{id}
```

Use query component to filter URI collection

Often, you will encounter requirements where you will need a collection of resources sorted, filtered, or limited based on some specific resource attribute.

For this requirement, do not create new APIs – instead, enable sorting, filtering, and pagination capabilities in resource collection API and pass the input parameters as query parameters. e.g.

```
http://api.example.com/device-management/managed-devices
```

```
http://api.example.com/device-management/managed-devices?region=USA
```

```
http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ
```

```
http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ&sort=installation-date
```

Thank You!

