REST

Rest stands for Representational State Transfer. It is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. REST-compliant systems, often called RESTful systems, are characterized by how they are stateless and separate the concerns of client and server.

In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

As long as each side knows what format of messages to send to the other, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns, we improve the flexibility of the interface across platforms and improve scalability by simplifying the server components. Additionally, the separation allows each component the ability to evolve independently.

By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

**Statelessness**

Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of resources, rather than commands. Resources are the nouns of the Web - they describe any object, document, or thing that you may need to store or send to other services.

Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of resources, rather than commands. Resources are the nouns of the Web - they describe any object, document, or thing that you may need to store or send to other services.

**Communication between Client and Server**

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests. Let’s take a look at the standard ways to make requests and send responses.

**Making Requests**

REST requires that a client make a request to the server in order to retrieve or modify data on the server. A request generally consists of:

* an HTTP verb, which defines what kind of operation to perform
* a *header*, which allows the client to pass along information about the request
* a path to a resource
* an optional message body containing data

**HTTP Verbs**

There are 4 basic HTTP verbs we use in requests to interact with resources in a REST system:

* GET — retrieve a specific resource (by id) or a collection of resources
* POST — create a new resource
* PUT — update a specific resource (by id)
* DELETE — remove a specific resource by id

**Headers and Accept parameters**

In the header of the request, the client sends the type of content that it is able to receive from the server. This is called the Accept field, and it ensures that the server does not send data that cannot be understood or processed by the client. The options for types of content are MIME Types (or Multipurpose Internet Mail Extensions.

For example, a text file containing HTML would be specified with the type text/html. If this text file contained CSS instead, it would be specified as text/css. A generic text file would be denoted as text/plain. This default value, text/plain, is not a catch-all, however. If a client is expecting text/css and receives text/plain, it will not be able to recognize the content.

Other types and commonly used subtypes:

image — image/png, image/jpeg, image/gif

audio — audio/wav, audio/mpeg

video — video/mp4, video/ogg

application — application/json, application/pdf, application/xml, application/octet-stream

For example, a client accessing a resource with id 23 in an articles resource on a server might send a GET request like this:

GET /articles/23

Accept: text/html, application/xhtml

The Accept header field in this case is saying that the client will accept the content in text/html or application/xhtml.

**Paths**

Requests must contain a path to a resource that the operation should be performed on. In RESTful APIs, paths should be designed to help the client know what is going on.

Conventionally, the first part of the path should be the plural form of the resource. This keeps nested paths simple to read and easy to understand.

A path like fashionboutique.com/customers/223/orders/12 is clear in what it points to, even if you’ve never seen this specific path before, because it is hierarchical and descriptive. We can see that we are accessing the order with id 12 for the customer with id 223.

Paths should contain the information necessary to locate a resource with the degree of specificity needed. When referring to a list or collection of resources, it is not always necessary to add an id. For example, a POST request to the fashionboutique.com/customers path would not need an extra identifier, as the server will generate an id for the new object.

If we are trying to access a single resource, we would need to append an id to the path. For example: GET fashionboutique.com/customers/:id — retrieves the item in the customers resource with the id specified. DELETE fashionboutique.com/customers/:id — deletes the item in the customers resource with the id specified.

**Sending Responses**

Content Types

In cases where the server is sending a data payload to the client, the server must include a content-type in the header of the response. This content-type header field alerts the client to the type of data it is sending in the response body. These content types are MIME Types, just as they are in the accept field of the request header. The content-type that the server sends back in the response should be one of the options that the client specified in the accept field of the request.

For example, when a client is accessing a resource with id 23 in an articles resource with this GET Request:

GET /articles/23 HTTP/1.1

Accept: text/html, application/xhtml

The server might send back the content with the response header:

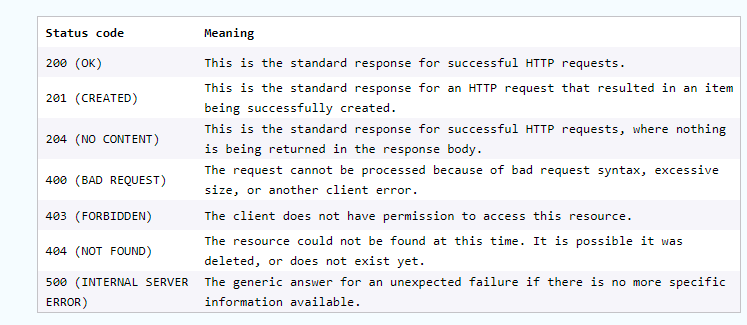
HTTP/1.1 200 (OK)

Content-Type: text/html

This would signify that the content requested is being returned in the response body with a content-type of text/html, which the client said it would be able to accept.

Response Codes

Responses from the server contain status codes to alert the client to information about the success of the operation. As a developer, you do not need to know every status code (there are many of them), but you should know the most common ones and how they are used:



For each HTTP verb, there are expected status codes a server should return upon success:

GET — return 200 (OK)

POST — return 201 (CREATED)

PUT — return 200 (OK)

DELETE — return 204 (NO CONTENT) If the operation fails, return the most specific status code possible corresponding to the problem that was encountered.

References

<https://www.codecademy.com/article/what-is-rest>