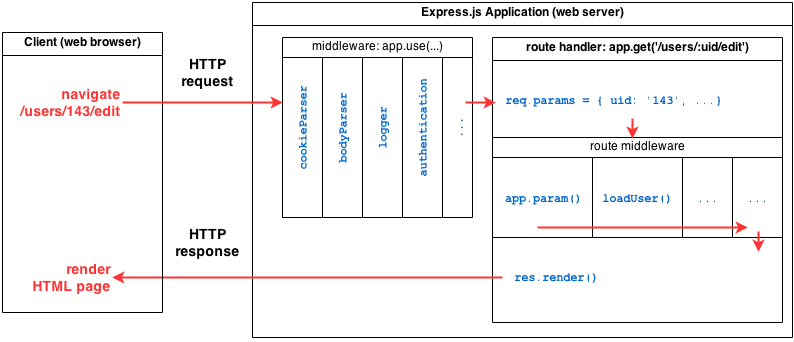
**ExpressJS and Middlewares**

ExpressJS is a complete web framework solution. It has HTML template solutions (jade, ejs, handlebars, hogan.js) and CSS precompilers (less, stylus, compass). Through middlewares layers, it handles: cookies, sessions, caching, CSRF, compression and many more.

**Middlewares** are a stack of processors that runs on each request made to the server. You can have any number of middlewares that will process the request one by one in a serial fashion. Some might alter the request input, log outputs, add data and pass it to the next() middleware in the chain.

Middlewares are added to ExpressJS stack using app.use for any method or the app.VERB (e.g., app.get, app.delete, app.post, app.update, …)



Below is a table summarizing recommended return values of the primary HTTP methods in combination with the resource URIs:

| **HTTP Verb** | **Entire Collection (e.g. /customers)** | **Specific Item (e.g. /customers/{id})** |
| --- | --- | --- |
| GET | 200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single customer. 404 (Not Found), if ID not found or invalid. |
| PUT | 404 (Not Found), unless you want to update/replace every resource in the entire collection. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| POST | 201 (Created), 'Location' header with link to /customers/{id} containing new ID. | 404 (Not Found). |
| DELETE | 404 (Not Found), unless you want to delete the whole collection—not often desirable. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

The HTTP GET method is used to retrieve (or read) a representation of a resource. In the “happy” (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.

**Examples:**

* *GET http://www.example.com/customers/12345*
* *GET http://www.example.com/customers/12345/orders*
* *GET http://www.example.com/buckets/sample*

PUT is most-often utilized for update capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.

PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.

**Examples:**

* *PUT http://www.example.com/customers/12345*
* *PUT http://www.example.com/customers/12345/orders/98765*
* *PUT http://www.example.com/buckets/secret\_stuff*

The POST verb is most-often utilized for creation of new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

**Examples:**

* *POST http://www.example.com/customers*
* *POST http://www.example.com/customers/12345/orders*

The POST verb is most-often utilized for creation of new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

**Examples:**

* *POST http://www.example.com/customers*
* *POST http://www.example.com/customers/12345/orders*

DELETE is pretty easy to understand. It is used to delete a resource identified by a URI.

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This makes DELETE operations no longer idempotent, but is an appropriate compromise if resources are removed from the database instead of being simply marked as deleted.

**Examples:**

* *DELETE http://www.example.com/customers/12345*
* *DELETE http://www.example.com/customers/12345/orders*
* *DELETE http://www.example.com/bucket/sample*