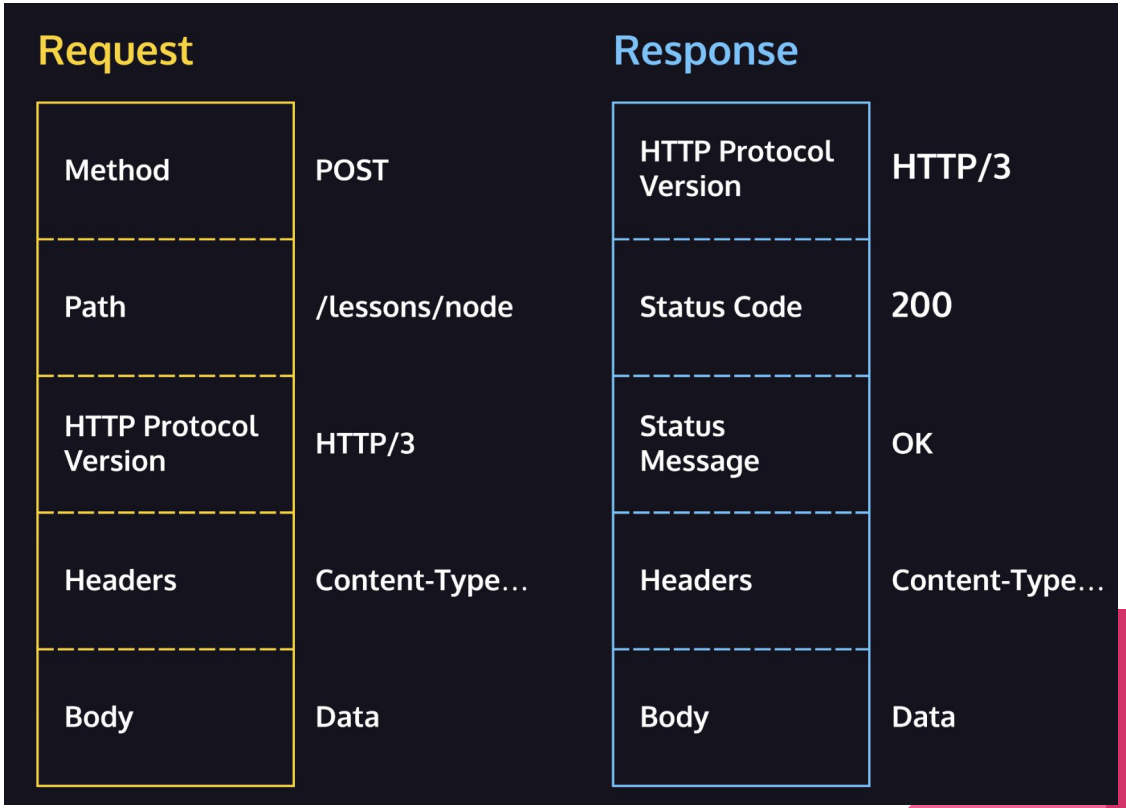
Structure of HTTP



The HTTP module

To process HTTP requests in JavaScript and Node.js, we can use the built-in `http` module.

One of the most commonly used methods within the `http` module is the `.createServer()` method. This method is responsible for doing exactly what its namesake implies; it creates an HTTP server. To implement this method to create a server, the following code can be used:

```
const server = http.createServer((req, res) => {
  res.end('Server is running!');
});

server.listen(8080, () => {
  const { address, port } = server.address();
  console.log(`Server is listening on:
http://${address}:${port}`);
})
```

Anatomy of the URL

HTTP servers have to break down requests into their constituent parts to effectively process them and provide adequate responses. In that same vein, designing an API (Application Programming Interface) with endpoints intended to process specific requests in certain ways requires an understanding of the semantics of these requests, which are ultimately embodied within a URL (Uniform Resource Locator).

The URL module

Typically, an HTTP server will require information from the request URL to accurately process a request. This request URL is located on the `url` property contained within the `req` object itself. To parse the different parts of this URL easily, Node.js provides the built-in `url` module. The core of the `url` module revolves around the `URL` class. A new `URL` object can be instantiated using the `URL` class as follows:

```
const url = new URL('https://www.example.com/p/a/t/h?
query=string');

const host = url.hostname; // example.com
const pathname = url.pathname; // /p/a/t/h
const searchParams = url.searchParams; // {query:
'string'}
```

The URL module

You can also use the URL class to construct a URL.

```
const createdUrl = new URL('https://www.example.com');
createdUrl.pathname = '/p/a/t/h';
createdUrl.search = '?query=string';

createUrl.toString(); // Creates
https://www.example.com/p/a/t/h?query=string
```

The Querystring module

The `querystring` module is dedicated to providing utilities solely focused on parsing and formatting URL query strings.

- `.parse()`: This method is used for parsing a URL query string into a collection of key-value pairs. The `.decode()` method does the same.
- `.stringify()`: This method is used for producing a URL query string from a given object via iteration of the object's "own properties." The `.encode()` method does the same.
- `.escape()`: This method is used for performing percent-encoding on a given query string.
- `.unescape()`: This method is used to decode percent-encoded characters within a given query string.

```
const str = 'prop1=value1&prop2=value2';
querystring.parse(str); // Returns { prop1: value1,
prop2: value2}
```

```
const props = { "prop1": value1, "prop2": value2};
querystring.stringify(props); // Returns
'prop1=value1&prop2=value2'
```

Routing

To process and respond to requests appropriately, servers need to do more than look at a request and dispatch a response. Internally, a server needs to maintain a way to handle each request based on specific criteria such as `method`, `pathname`, etc. The process of handling requests in specific ways based on the information provided within the request is known as *routing*.

```
const server = http.createServer((req, res) => {
  const { method } = req;

  switch(method) {
    case 'GET':
      return handleGetRequest(req, res);
    case 'POST':
      return handlePostRequest(req, res);
    case 'DELETE':
      return handleDeleteRequest(req, res);
    case 'PUT':
      return handlePutRequest(req, res);
    default:
      throw new Error(`Unsupported request method:
${method}`);
  }
})
```

Routing

We can distinguish one request from another of the same method through the use of the `pathname`. The `pathname` allows the server to understand what resource is being targeted.

```
function handleGetRequest(req, res) {
  const { pathname } = new URL(req.url);
  let data = {};

  if (pathname === '/projects') {
    data = await getProjects();
    res.setHeader('Content-Type', 'application/json');
    return res.end(JSON.stringify(data));
  }

  res.statusCode = 404;
  return res.end('Requested resource does not exist');

}
```

HTTP Status Codes

Once a request is processed, a response must be returned to the client to inform it of what happened. To build a response for the client, several pieces of information are required. One of these pieces of information is the HTTP response status code, which is responsible for indicating whether a specific HTTP request has been successfully completed.

Response status codes are grouped into five classes:

- Informational: Range from `100` to `199`.
- Successful: Range from `200` to `299`.
- Redirects: Range from `300` to `399`.
- Client Errors: Range from `400` to `499`.
- Server Errors: Range from `500` to `599`.

```
const handleGetRequest = (req, res) => {
  res.statusCode = 200;
  return res.end(JSON.stringify({ data: [] }));
}
```

Interacting with other backend API

There are a few methods provided by the `http` module that facilitate making HTTP requests to external services. One of these methods is the `request()` method. The `request()` method takes two arguments; it takes a configuration object containing details about the request as well as a callback to handle the response.

```
const handleGetRequest = (req, res) => {
  const options = {
    hostname: 'example.com',
    port: 8080,
    path: '/test',
    method: 'GET',
    headers: {
      'Content-Type': 'application/json'
    }
  };

  const request = https.request(options, response => {
    let data = "";
    response.on('data', val => {
      data += val;
    });
    response.on('end', () => {
      console.log(data);
      res.end(data);
    });
  });
  request.end();
}
```

Activity