ISEngineering

Technische Universität Berlin

CSEM: Server-Side Development

Dr. Markus Klems, Prof. Stefan Tai

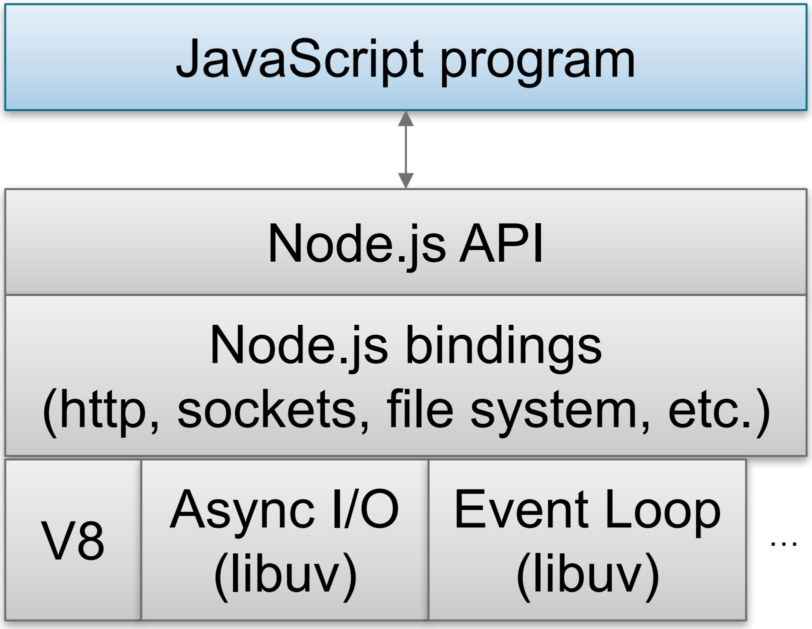
Learning objectives

- Understand the Node.js programming model
  - Event-driven programming with `EventEmitters`
  - Stream-based data processing using the `Stream` API
- Get to know I/O API methods that Node.js offers for
  - working with files
  - processing HTTP requests
- Learn basic API development with the Express framework

SERVER-SIDE JAVASCRIPT

Node.js

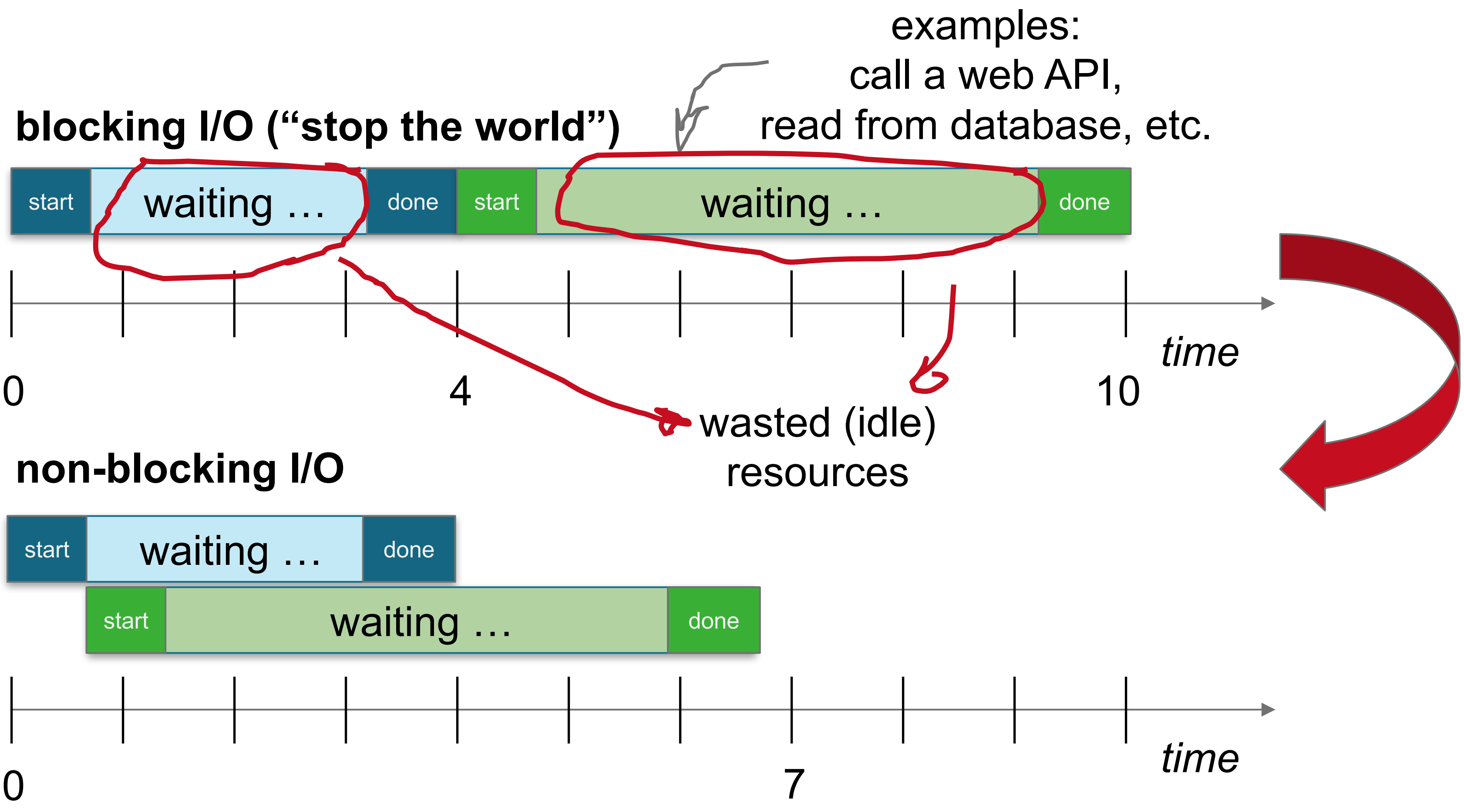
- Node.js (short: Node) is a JavaScript runtime environment which is optimized for server-side programming
  - Asynchronous I/O via *libuv* library.
  - Bindings to low-level Operating System APIs (file system, networking, etc.).
  - Based on Google's V8 JavaScript engine with just-in-time bytecode compiler, optimizer, and garbage collector.



From client-side to server-side JavaScript

- JavaScript’s origins are in client-side web development.
- The Node.js runtime offers several features that are important for server-side JavaScript development:
  - Access to operating system resources, such as sockets and files.
  - Node.js offers core modules for TCP, UDP, HTTP, DNS, etc. which makes it easy to build network-based applications.
  - Node.js programs can use a global process object which for accessing the runtime environment.
  - Binary data can be handled efficiently by using a special `Buffer` class.
- Node.js cannot access the DOM (Document Object Model) via the `Window` global object, as you are used to in browser-based JavaScript applications. The global object in Node.js, for a lack of creativity, is name `global`.

Synchronous (blocking) vs. asynchronous (non-blocking) I/O



examples:
call a web API,
read from database, etc.

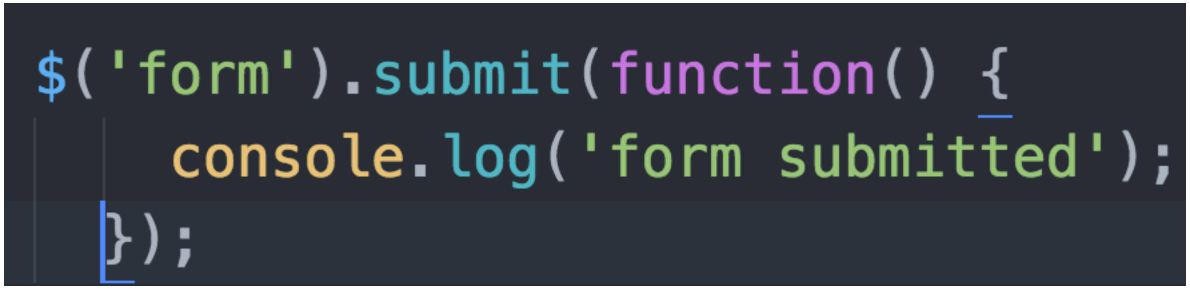
blocking I/O (“stop the world”)

wasted (idle) resources

non-blocking I/O

Even-Driven Programming

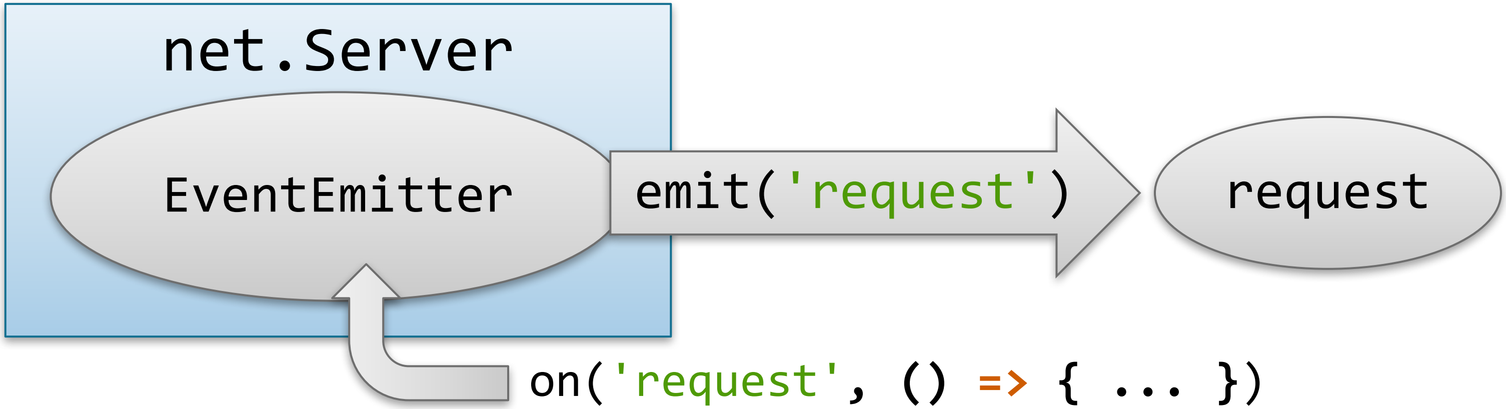
- In event-driven programming, the program flow is determined by events.
- Event-driven programming patterns, such as the observer patterns, are particularly suitable for User Interface (UI) implementation, which is centered around user interactions.
- Client-side JavaScript programs, by using event listeners, usually make extensive use of DOM (Document Object Model) events, such as:
  - Mouse events
  - Keyboard events
  - HTML form events



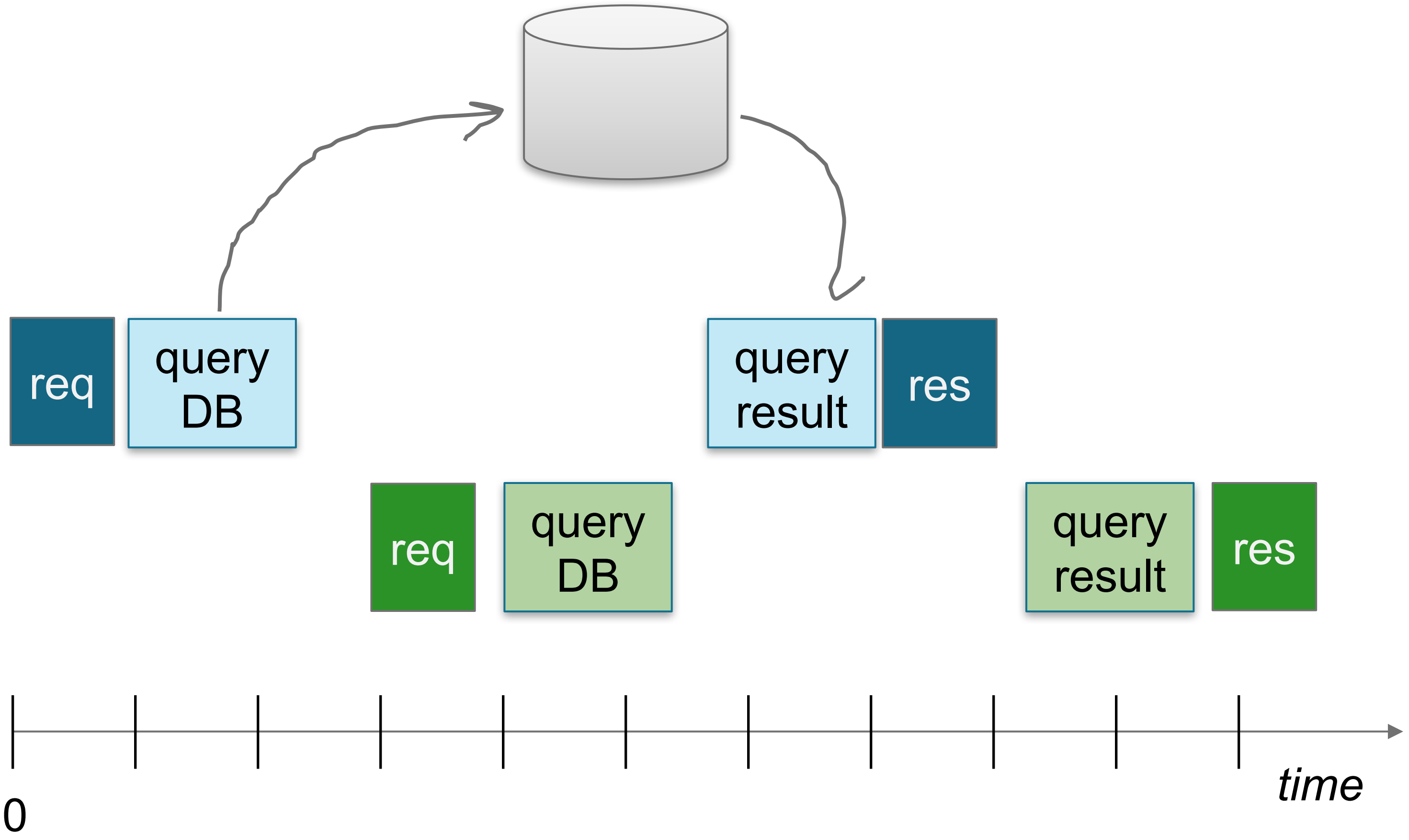
```
$('form').submit(function() {
    console.log('form submitted');
});
```

Node.js Event Emitters

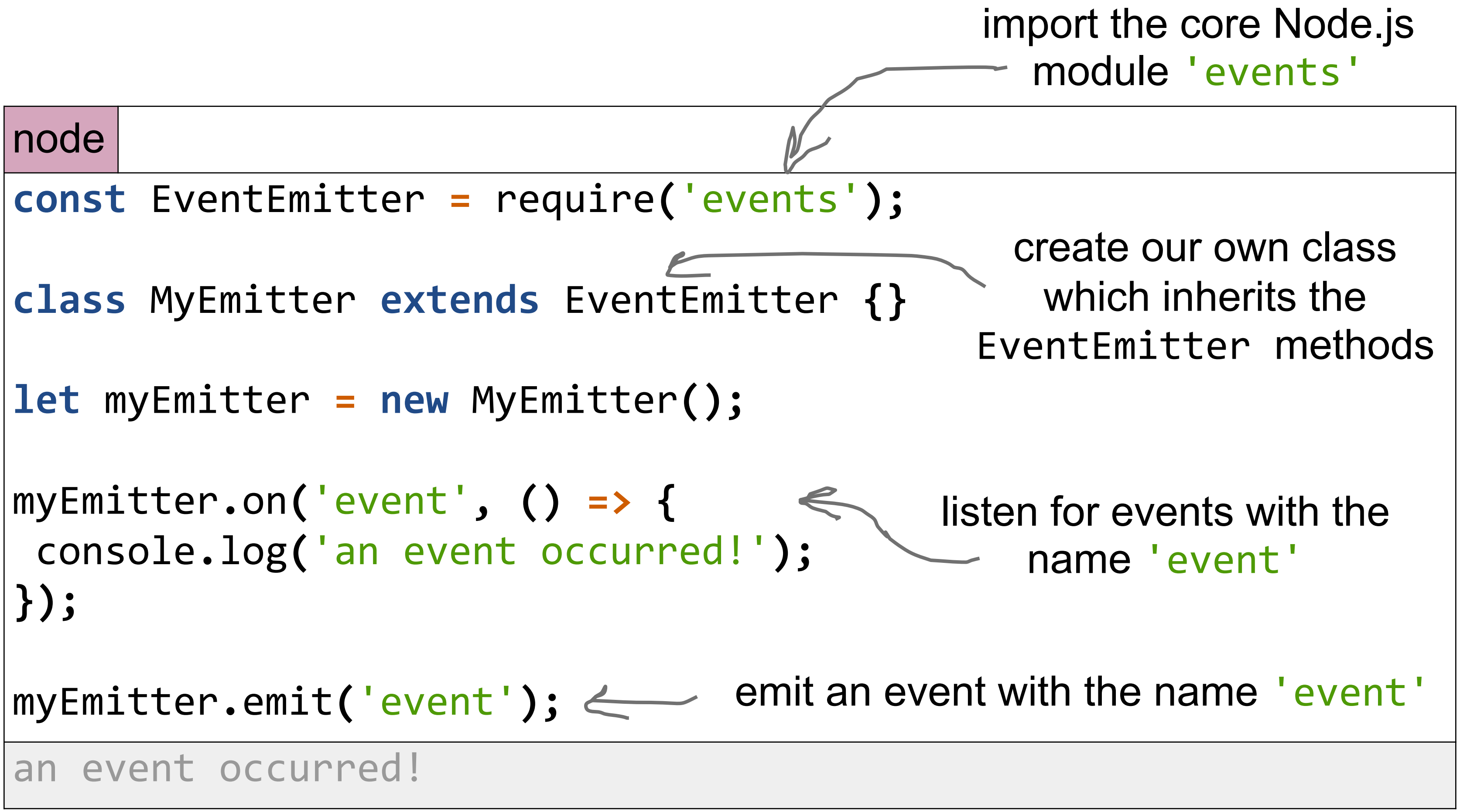
- Different from client-side JavaScript applications, events in Node.js programs are not emitted by the DOM (Document Object Model).
- Instead, Node.js provides the concept of event emitters that generate events. An event emitter provides an API that includes several methods, such as
  - `emitter.on(name, function)`
  - `emitter.emit(name [, args])`



HTTP Server + Events



Custom Event Emitters: Example



import the core Node.js module `'events'`

node

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

let myEmitter = new MyEmitter();

myEmitter.on('event', () => {
 console.log('an event occurred!');
});

myEmitter.emit('event');
```

```
an event occurred!
```

create our own class which inherits the `EventEmitter` methods

listen for events with the name `'event'`

emit an event with the name `'event'`

Node.js Streams

- Data streaming is a core Node functionality that allows efficient data processing.
- If you work with files or network resources (*e.g.,* HTTP requests/responses), you are using streams.
- The Node.js Streams API offers an abstract interface for working with data streams. This is what you should know about Node.js Streams:
    - Streams are instances of `EventEmitter` that support event types, such as `'data'`, `'error'`, and `'end'`.
    - Streams can be readable, writable, duplex (both readable and writable), or transform streams (modify data while it is written or read).
    - You can pipe data from a readable to a writable stream.
    - Buffering:
        - Readable streams store data in an internal read buffer.
        - Writable streams store data in an internal write buffer.
        - Backpressure mechanism: throttle writes for slow readers.

Readable Streams

- Readable streams are an abstraction for a source from which data is consumed.
- Examples of Readable streams include:
  - HTTP responses, on the client
  - HTTP requests, on the server
  - fs read streams
  - `process.stdin`
- All Readable streams implement the interface defined by the `stream.Readable` class.

https://nodejs.org/api/stream.html#stream_readable_streams

Readable Streams: Two Modes

- Readable streams effectively operate in one of two modes: flowing and paused.
- When in flowing mode, data is read from the underlying system automatically and provided to an application as quickly as possible using events via the `EventEmitter` interface.
- In paused mode, the `stream.read()` method must be called explicitly to read chunks of data from the stream.
- All Readable streams begin in paused mode but can be switched to flowing mode in one of the following ways:
  - Adding a `'data'` event handler.
  - Calling the `stream.resume()` method.
  - Calling the `stream.pipe()` method to send the data to a `Writable Stream`.

https://nodejs.org/api/stream.html#stream_readable_streams

Readable Streams: Two Modes

- The Readable can switch back to paused mode:
  - by calling the `stream.pause()` method
  - by removing any `'data'` event handlers + removing all pipe destinations by calling the `stream.unpipe()` method
- The important concept to remember is that a Readable will not generate data until a mechanism for either consuming or ignoring that data is provided.
- If the consuming mechanism is disabled or taken away, the Readable will attempt to stop generating the data.

https://nodejs.org/api/stream.html#stream_readable_streams

Readable Streams: Example

node

```
const Readable = require('stream').Readable;

let readable = new Readable();
readable.push('foo\n');
readable.push('hello world\n');
readable.push(null); // we are done

readable.on('data', chunk => {
 console.log(`Received ${chunk.length} bytes of data.`);
 console.log(`Chunk: ${chunk}`);
});


readable.on('error', err => {
 console.error(err);
});
```

When the Readable is operating in flowing mode, the data added with `readable.push()` will be delivered by emitting `'data'` events.

```
Received 4 bytes of
data. Chunk: foo
Received 12 bytes of
data. Chunk: hello world
```

Writable Streams

- All Writable streams implement the interface defined by the `stream.Writable` class.
- While specific instances of Writable streams may differ in various ways, all Writable streams follow the same fundamental usage pattern as illustrated in the example below:

```
let writable = // get writable stream
writable.write('foo');
writable.write('bar');
writable.end();
```

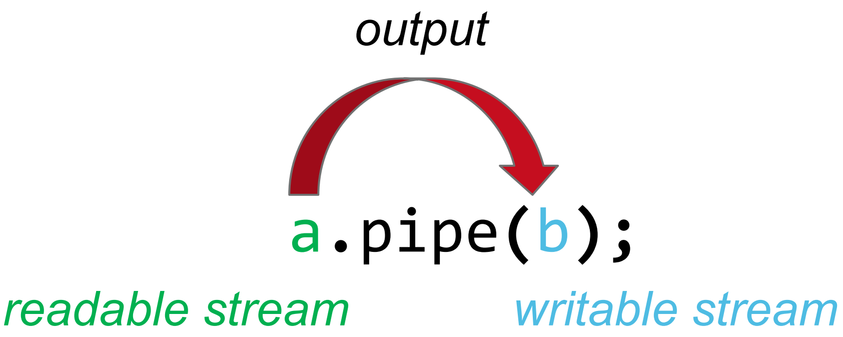
https://nodejs.org/api/stream.html#stream_writable_streams

Writable Streams: Example

node

```
const fs = require('fs');
let writable = fs.createWriteStream('hello.txt');

writable.write('foo\n', 'utf8', () => {
 console.log('foo chunk has been flushed.');
});
writable.write('hello world\n', 'utf8', () => {
 console.log('hello world chunk has been flushed.');
});
writable.end();

writable.on('error', err => {
 console.error(err);
});
writable.on('finish', () => {
 console.log('All writes have finished.');
});
```

Node.js Pipes



output

```
a.pipe(b);
```

readable stream *writable stream*

```
x.pipe(y).pipe(z);
```

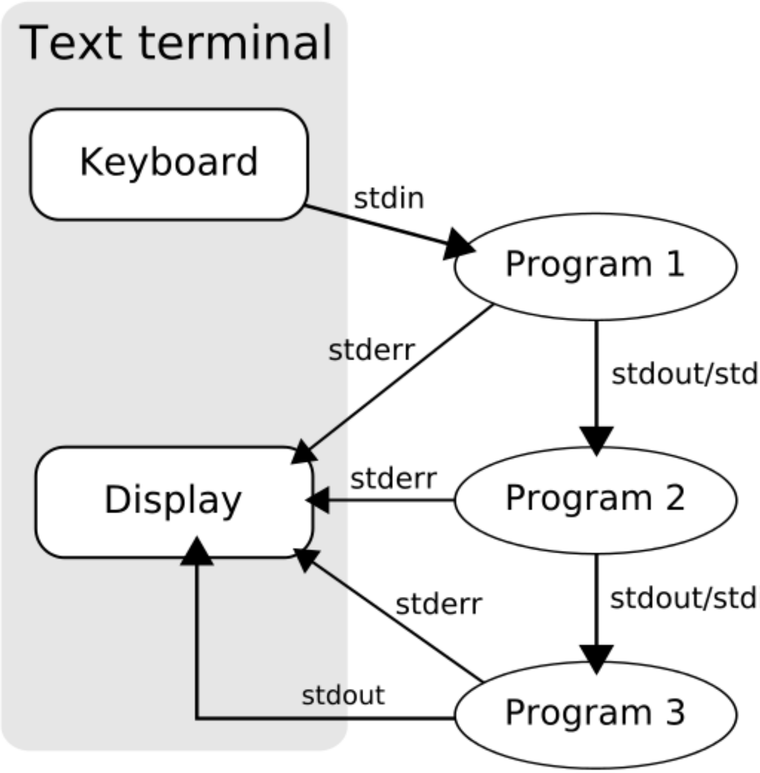
readable stream *duplex stream* *writable stream*

=

```
x.pipe(y);
y.pipe(z);
```

Background: Unix Pipes

- Node.js stream pipes are similar to Unix pipes.
- In Unix-like computer operating systems, a pipeline is a sequence of processes chained together by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) to the next one.
- The concept of pipelines was championed by Douglas McIlroy at Unix's ancestral home of Bell Labs, during the development of Unix, shaping its toolbox philosophy. It is named by analogy to a physical pipeline.



Each process takes input from the previous process and produces output for the next process via standard streams.

https://en.wikipedia.org/wiki/Pipeline_(Unix)

Node.js Readable and Writable Streams

- Let’s write something on a built-in writable stream: `process.stdout`

node

```
// Let's write something on the stdout stream
process.stdout.write('hello stream\n');
```

```
hello stream
```

- Here is a simple example that reads data from the built-in readable stream `process.stdin` and pipes it directly into the `process.stdout` stream:

node

```
process.stdin.pipe(process.stdout);
```

Node.js Modules

- You can turn your directory into a node package/project by simply adding a *package.json* file.
  - With a *package.json* file, you can document the dependencies of your project and make the build and other tasks reproducible by others.
  - Generate an initial *package.json* file: `$ npm init -y`

package.json

```
{
 "name": "test-app",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 ...
}
```

Export/import Node.js Modules

Define a Node.js module

index.js

```
let doStuff = function() {
   // does stuff
}

module.exports = {
   doStuff: doStuff
}
```

Load a Node.js module

main.js

```
let m = require('./index.js');
m.doStuff();
```

Node.js Modules

- You can add node modules as dependencies to your package/project via the npm command line with `npm install` (or short `npm i`). For example, install the “commander” module like this:

```
$ npm
npm i commander
test-app@1.0.0 /path/to/test-app
└─┬ commander@2.9.0
  └── graceful-readlink@1.0.1
```

- What did just happen? Two modules have been downloaded from the public npm registry and unpacked into your local *node_modules* subdirectory.
- The “commander” module depends on another module: “graceful-readlink”.
- The code of the “commander” module is in a single *index.js* file in the *node_modules/commander* subdirectory, which requires the “graceful-readlink” module, which consists of a single *index.js* file in the *node_modules/graceful-readlink*.

Node.js Modules

- You can add the `--save` parameter to your installation command, thereby automatically adding the dependency property to your *package.json* file.

$ npm

```
npm i --save commander
```

- The *package.json* file now looks like this:

package.json

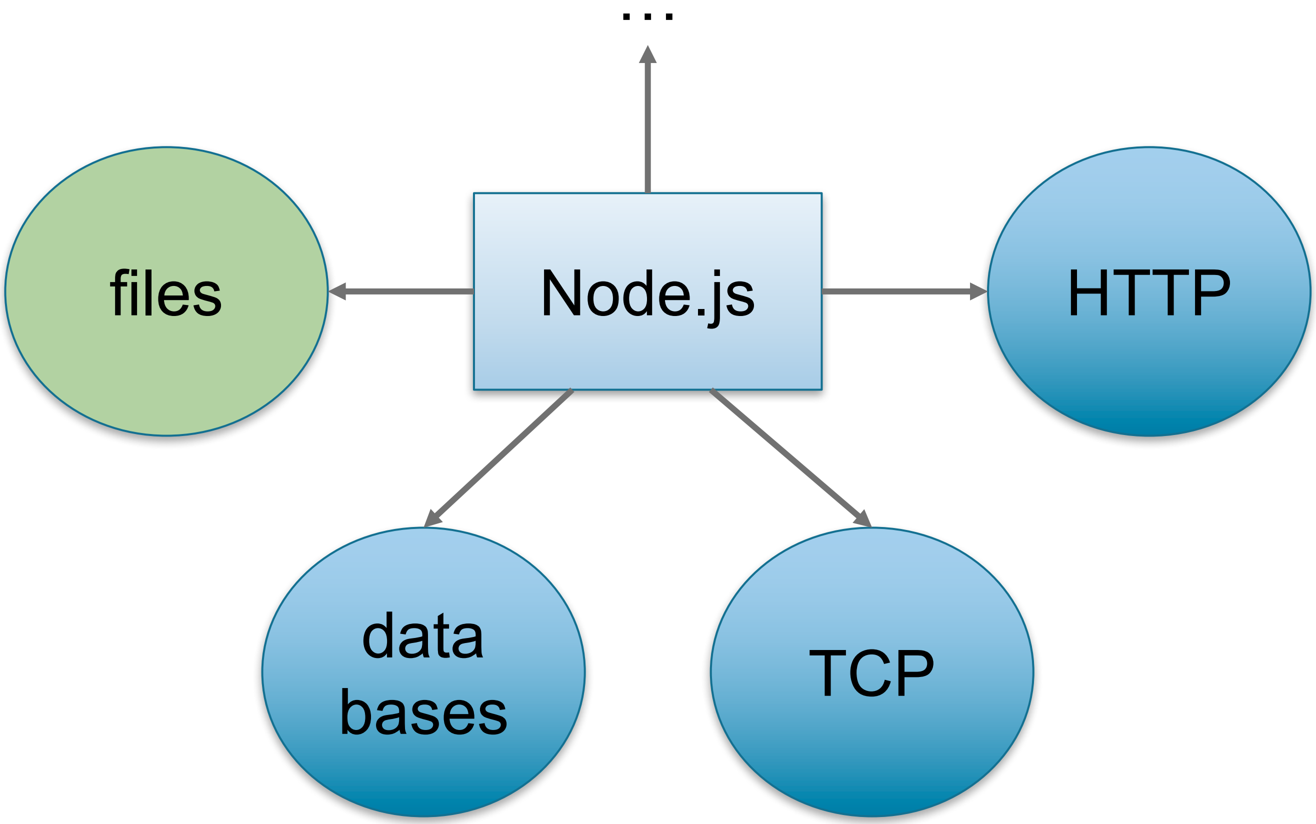
```
{
  ... as before ...
  "dependencies": {
    "commander": "^2.9.0"
  }
}
```

Node.js modules in a nutshell

- How to use a Node.js module?
  - You import a module to your program via `require`
  - If you don’t specify a path (such as *‘./my_mod’* or *‘/home/markus/modules/my_mod’*), the module is searched in the *node_modules* directory
- How to build a Node.js module?
  - Simply create a .js file and `module.exports` the object that you want to expose publicly
  - Other code which is not exported remains private
- npm – the node package manager
  - You can install packages locally via `$ npm i <pkg>`
  - Better: create a *package.json* file

  (`npm init` then `npm i --save <pkg>`)

Node.js APIs: non-blocking, stream-based I/O



Working with files

- Node.js provides a file system abstraction layer that works across different operating systems (based on the POSIX standard).
- Let’s look at a program that reads the content from a file (using non-stream methods) and logs the content on the console:

node

```
const fs = require('fs');
let pathToFile = process.argv[2];
// see https://nodejs.org/api/fs.html
fs.readFile(pathToFile, 'utf8', (err, data) => {
 if (err) {
   console.error(err);
 }
 console.log(data);
});
console.log('do something else');
```

```
do something else
... file content ...
```

all data must fit into memory
👎 large files
👎 many concurrent `readFile()` invocations

👎 user must wait until `readFile()` method has read & buffered the entire file

Working with files

Here is an equivalent program that uses the Stream API:

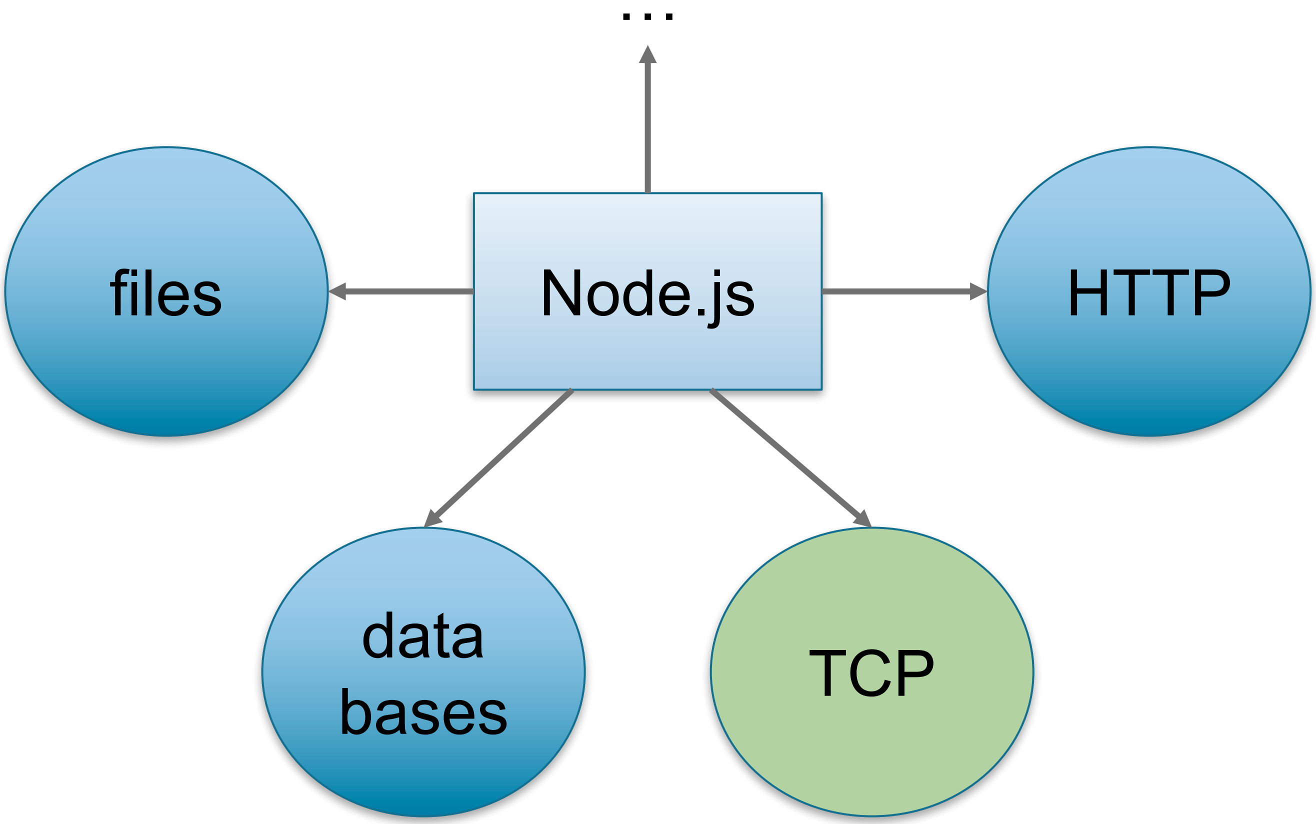
node

```
const fs = require('fs');
let pathToFile = process.argv[2];
let input = fs.createReadStream(pathToFile, { encoding: 'utf8' });

input.on('data', chunk => {
 console.log(chunk);
});
input.on('end', () => {
 console.log('k thx by');
});
console.log('do something else');
```

```
do something else
... file content ...
k thx by
```

listen for data events on the input stream
👍 process small data chunks

Node.js APIs: non-blocking, stream-based I/O



TCP Server

server.js node

```
// TCP server
const net = require('net');

net.createServer(conn => {
 console.log(`Connected
   ${conn.remoteAddress}:${conn.remotePort}`);
 conn.on('data', data => {
   console.log(`Received ${data}.`);
 });
}).listen(9876); // listening on port 9876
```

```
Connected ::ffff:127.0.0.1:59059
Received Hello, TCP server..
```

TCP Client

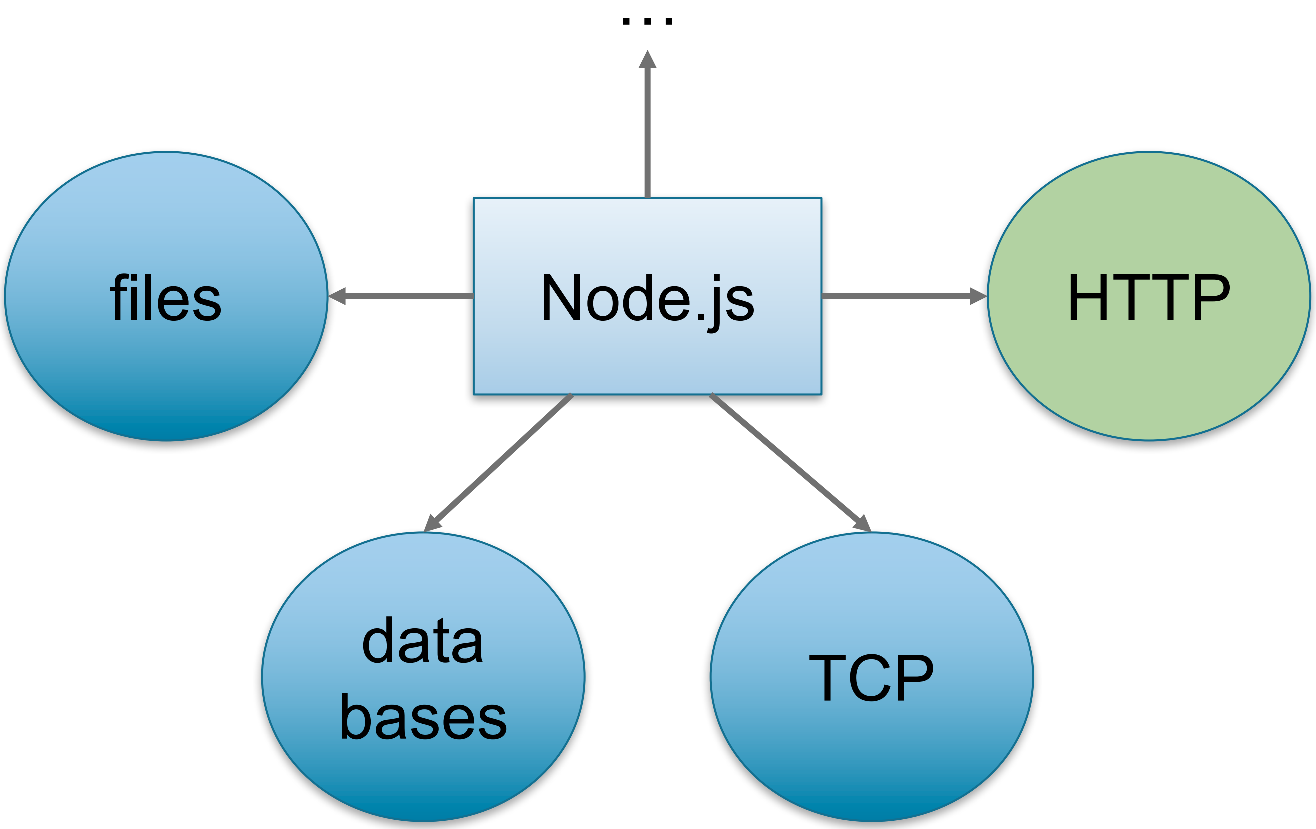
client.js node

```
// TCP client
const net = require('net');
let client = new net.Socket();

// connecting to TCP server at localhost:9876
client.connect(9876, 'localhost', () => {
 console.log('Client connected to server.');
 client.write('Hello, TCP server.');
});
```

```
Client connected to server.
```

Node.js APIs: non-blocking, stream-based I/O

Technische Universität Berlin



Networking with HTTP

- Let’s build an HTTP server.
- As a first step, we import the `http` module and create an http `server` object using the `createServer()` method.
- The function that we pass as an argument into the `createServer()` method, is executed once for each HTTP request.

node

```
const http = require('http');

// Create an HTTP server
let server = http.createServer((req, res) => {
 // Executed once per HTTP request
});
```

Networking with HTTP

The `http.Server` object inherits from `net.Server` which inherits from `EventEmitter`.

node

```
const http = require('http');
let server = http.createServer((req, res) => {
 // Executed once per HTTP request
});
```

same thing

node

```
const http = require('http');
let server = http.createServer();
server.on('request', (req, res) => {
 // Executed once per HTTP request
});
```

The event handler is invoked whenever a request event has been received

Networking with HTTP

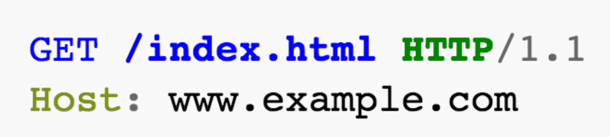
- For listening to HTTP request events, you must invoke the `server.listen()` method with the port number that the server is listening on (and optionally additional arguments):

node

```
const http = require('http');

// Create an HTTP server
let server = http.createServer((req, res) => {
 // Executed once per HTTP request
});

server.listen(3000, 'localhost', () => {
 console.log('Server is listening...');
});
```

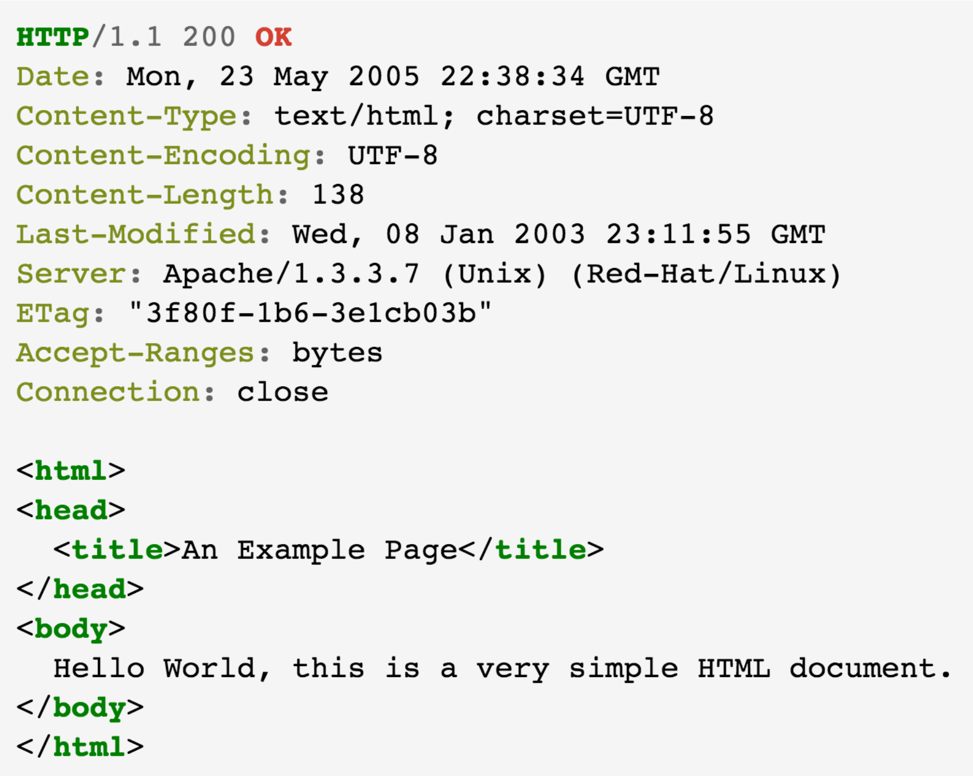
The server listens on port 3000

HTTP Methods



```
GET /index.html HTTP/1.1
Host: www.example.com
```

client request



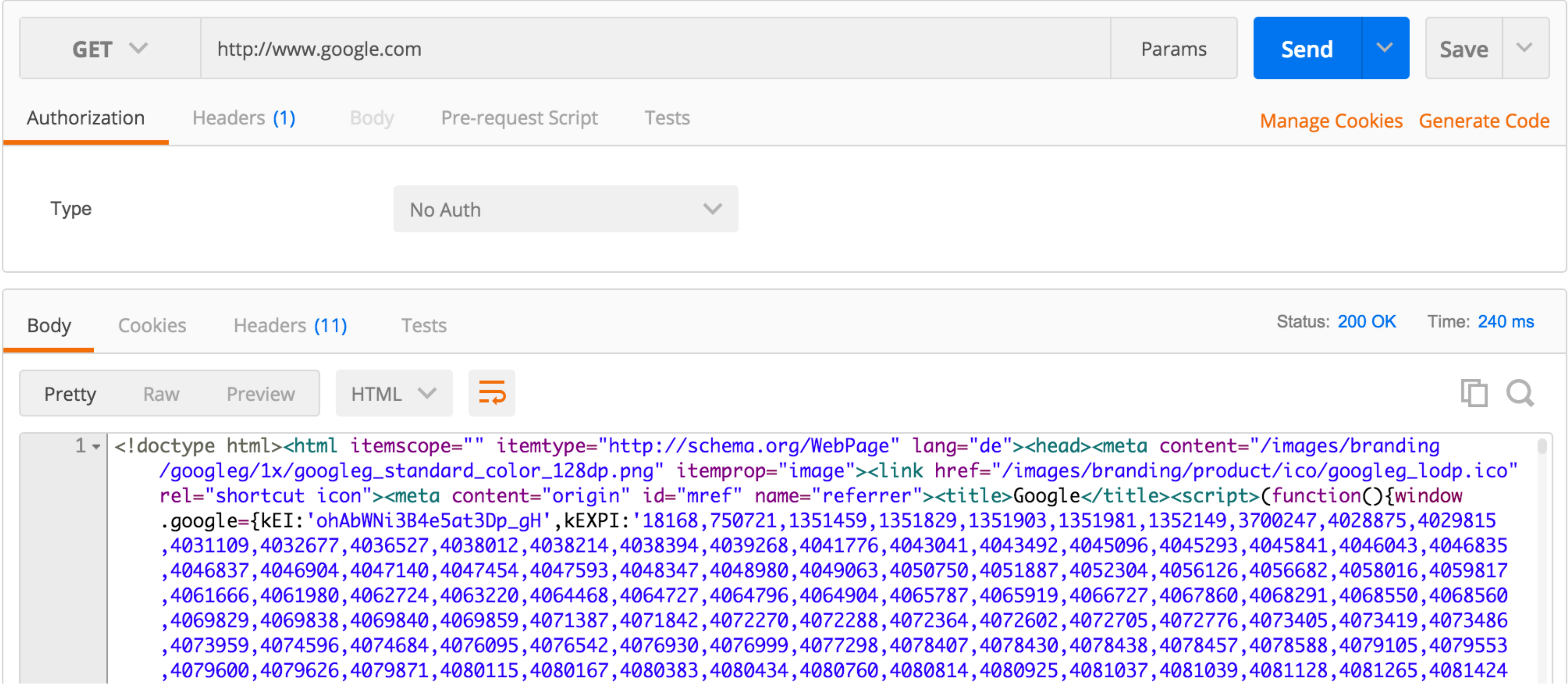
```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

server response

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

HTTP Methods

You can use developer tools for HTTP API testing, such as Postman.



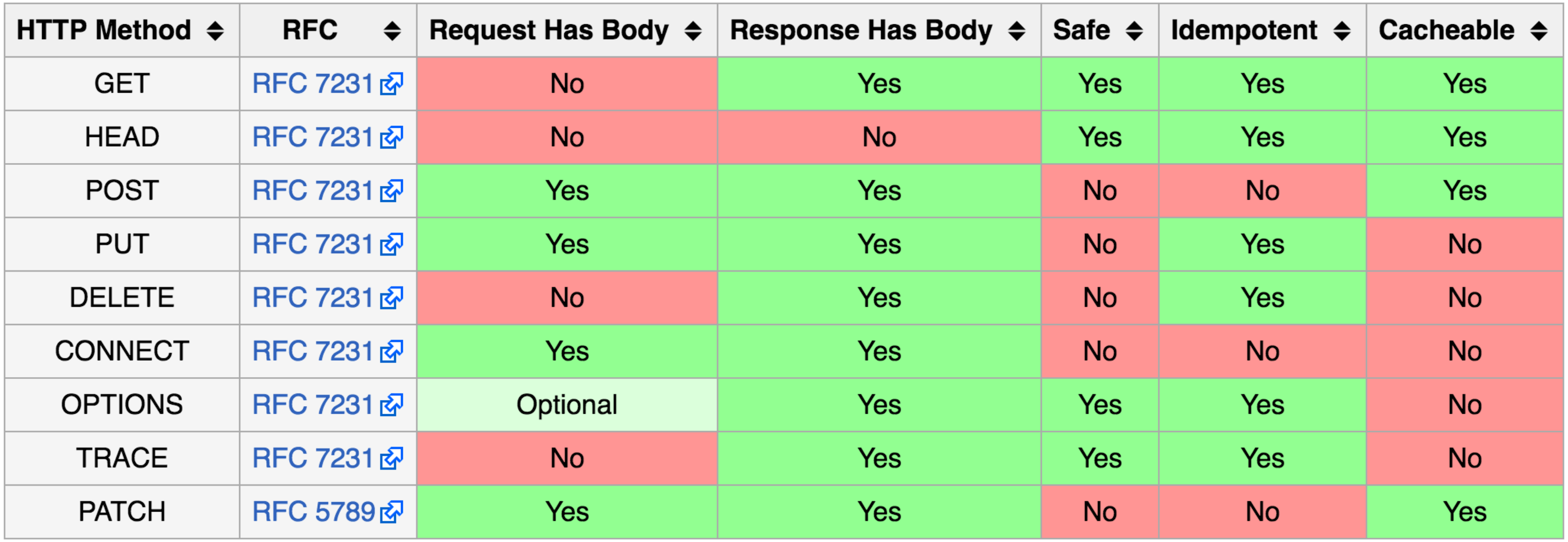
Type: No Auth

Status: 200 OK Time: 240 ms

```
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="de"><head><meta content="/images/branding
    /googleg/1x/googleg_standard_color_128dp.png" itemprop="image"><link href="/images/branding/product/ico/googleg_lodp.ico"
    rel="shortcut icon"><meta content="origin" id="mref" name="referrer"><title>Google</title><script>(function(){window
    .google={kEI:'ohAbWNi3B4e5at3Dp_gH',kEXPI:'18168,750721,1351459,1351829,1351903,1351981,1352149,3700247,4028875,4029815
    ,4031109,4032677,4036527,4038012,4038214,4038394,4039268,4041776,4043041,4043492,4045096,4045293,4045841,4046043,4046835
    ,4046837,4046904,4047140,4047454,4047593,4048347,4048980,4049063,4050750,4051887,4052304,4056126,4056682,4058016,4059817
    ,4061666,4061980,4062724,4063220,4064468,4064727,4064796,4064904,4065787,4065919,4066727,4067860,4068291,4068550,4068560
    ,4069829,4069838,4069840,4069859,4071387,4071842,4072270,4072288,4072364,4072602,4072705,4072776,4073405,4073419,4073486
    ,4073959,4074596,4074684,4076095,4076542,4076930,4076999,4077298,4078407,4078430,4078438,4078457,4078588,4079105,4079553
    ,4079600,4079626,4079871,4080115,4080167,4080383,4080434,4080760,4080814,4080925,4081037,4081039,4081128,4081265,4081424
```

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

HTTP Methods



HTTP Method	RFC	Request Has Body	Response Has Body	Safe	Idempotent	Cacheable
GET	RFC 7231	No	Yes	Yes	Yes	Yes
HEAD	RFC 7231	No	No	Yes	Yes	Yes
POST	RFC 7231	Yes	Yes	No	No	Yes
PUT	RFC 7231	Yes	Yes	No	Yes	No
DELETE	RFC 7231	No	Yes	No	Yes	No
CONNECT	RFC 7231	Yes	Yes	No	No	No
OPTIONS	RFC 7231	Optional	Yes	Yes	Yes	No
TRACE	RFC 7231	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	Yes

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Safe and idempotent HTTP Methods

- Safe methods are intended only for information retrieval and should not change the state of the server.
  - In other words, they should not have side effects, beyond relatively harmless effects such as logging, caching, the serving of banner advertisements or incrementing a web counter.
  - Making arbitrary GET requests without regard to the context of the application's state should therefore be considered safe.
  - However, this is not mandated by the standard, and it is explicitly acknowledged that it cannot be guaranteed.
- Idempotent methods mean that multiple identical requests should have the same effect as a single request.
  - The methods PUT and DELETE are defined to be idempotent.
  - The methods GET, HEAD, OPTIONS and TRACE, being prescribed as safe, should also be idempotent, as HTTP is a stateless protocol.

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

HTTP Request Object

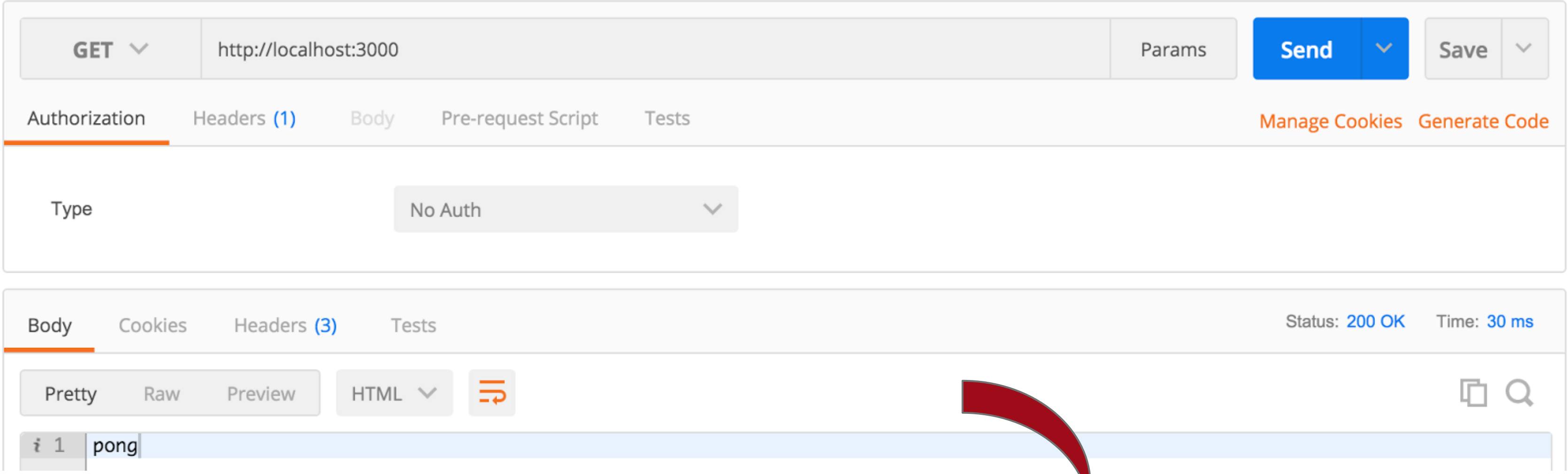
- The server can retrieve the URL, HTTP method, and HTTP header information that the client sent, as shown in the following example.

node

```
const http = require('http');

let server = http.createServer((req, res) => {
 // Retrieve URL, HTTP method, and HTTP headers
 let url = req.url;
 let method = req.method;
 let headers = req.headers;
 console.log('URL:', url);
 console.log('HTTP Method:', method);
 console.log('Headers:', headers);
 res.end('pong');
});

server.listen(3000, 'localhost', () => {
 console.log('Server is listening...');
});
```

HTTP Request Object

- When you start the program with `$ node`, you can send HTTP requests, for example using Postman, and see the request info printed on the console.



GET http://localhost:3000

Type: No Auth

Status: 200 OK   Time: 30 ms

```
1 pong
```

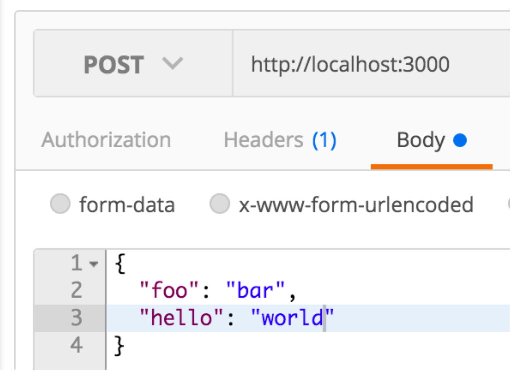
```
Server is listening...
URL: /
HTTP Method: GET
Headers: { host: 'localhost:3000',
 connection: 'keep-alive',
 'user-agent': 'Mozilla/5.0 ...',
 'cache-control': 'no-cache',
 'postman-token': '…',
 'content-type': 'application/json',
 accept: '*/*',
 'accept-encoding': 'gzip, deflate',
 'accept-language': 'en-US' }
```

HTTP Request Object

- The request object is a `http.IncomingMessage` which implements the `Readable Stream` interface.
- Thereby, you can read the request body as a readable stream:

node

```
let server = http.createServer((req, res)
=> {
 let body = [];
 req.on('data', chunk => {
   body.push(chunk);
 }).on('end', () => {
   body = Buffer.concat(body).toString();
   console.log('BODY:', body);
 });
 res.end('pong');
});
```



```
{
  "foo": "bar",
  "hello": "world"
}
```

```
BODY: {
 "foo": "bar",
 "hello": "world"
}
```

Error Handling

- When an error occurs while reading the request, we don’t want our node.js server to crash.
- Therefore, the `'error'` event should be handled

node

```
// as before ...
let server = http.createServer((req, res) => {
  req.on('error', err => {
    console.error(err.stack);
    res.end('error!');
   }).on('data', chunk => {
     // Process data
   }).on('end', () => {
     // End of request
 });
});
```

HTTP Response Object

The `response` object implements a `Writeable Stream` interface.

node

```
// as before ...
let server = http.createServer((req, res) => {
 req.on('end', () => {
   // Send some HTML as response
   res.on('error', err => {
     console.error('Response error:', err);
   });
   res.statusCode = 200;
   res.setHeader('Content-Type', 'text/html');
   res.write('<h1>Hi</h1>');
   res.write(`<p>You sent this: ${body || 'nothing'}</p>`);
   res.end();
 });
});
```

set HTTP response status code and headers

Write the response body

The `response.end()` method MUST be called on each response to signal that the response message is complete.

Pipe request stream to response stream

- You can also pipe data from the readable request stream to the writable response stream.

node

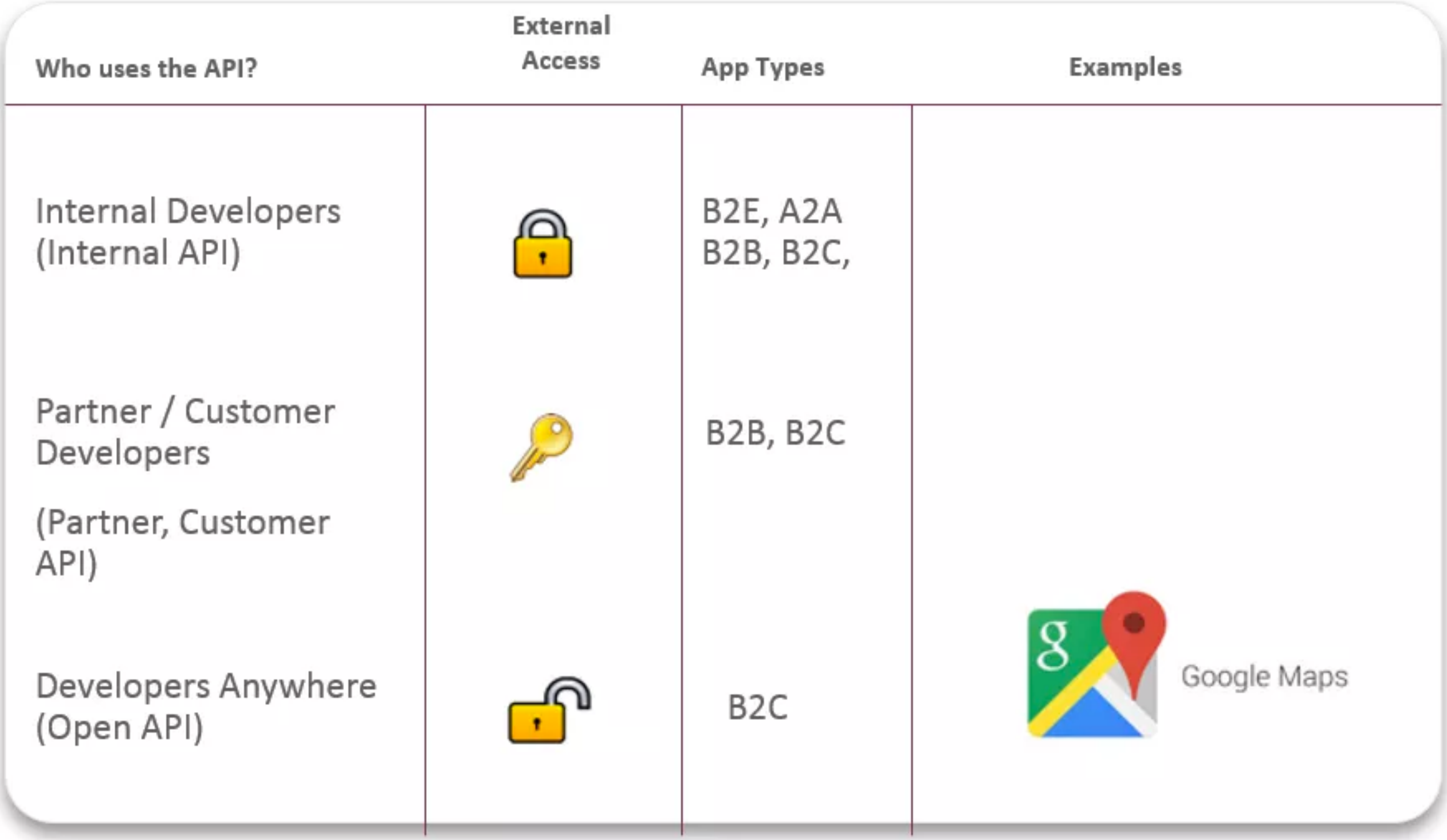
```
const http = require('http');
const zlib = require('zlib');

let server = http.createServer((req, res) => {
   // Set response headers, etc.
   let gzip = zlib.createGzip();
   req.pipe(gzip).pipe(res);
});
```

We pipe the request stream through a gzip transformation stream and then pipe the compressed data into the response stream.

Kahoot Quiz

API DEVELOPMENT

Different types of APIs (by API user)



Who uses the API?	External Access	App Types	Examples
Internal Developers (Internal API)	🔒	B2E, A2A B2B, B2C,	
Partner / Customer Developers (Partner, Customer API)	🔑	B2B, B2C	
Developers Anywhere (Open API)	🔓	B2C	Google Maps

https://apifriends.com/2017/09/20/different-types-of-apis/

Different types of APIs

- Data APIs – provide CRUD access to data stores
- Internal service APIs – expose internal (legacy) web services
- External service APIs – 3rd party services
- Composite APIs – combination of multiple data or service APIs

https://apifriends.com/2017/09/20/different-types-of-apis/

Building an API with the Express Framework

- Express is a lightweight HTTP routing framework that enables developers to quickly build HTTP APIs.

- Thereby, Node.js backend functions can be exposed according to the REST architectural style.

- Install Express as npm package:
  ```
  $ npm install --save express
  ```

Express Framework: Routing

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.

Example:

index.js

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

app.get('/', (req, res) => res.send('Hello World!'));
```

expressjs.com

Express Framework: Routing

Express uses path-to-regexp matching for route paths, meaning that you can use “wildcard” characters in your path and execute the handler function on all matching paths. For example, the following route path '/app(les)?/or/*' will match '/apples/or/oranges' and '/apples/or/' and 'app/or/x' etc.

Example:

index.js

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

app.get('/app(les)?/or/*', (req, res) =>
res.send('app(les)!'));
```

expressjs.com

Express Framework: Routing

Express allows you to extract parameters from your URL endpoint, as in the following example:

index.js

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

app.get('/hotels/:chain-:location/rooms/:roomId', (req, res)
=> res.send(req.params));
```

The route path `'/hotels/VulcanInn-Berlin/rooms/7'` results in the following `req.params` object:

`{"chain":"VulcanInn","location":"Berlin","roomId":"7"}`.

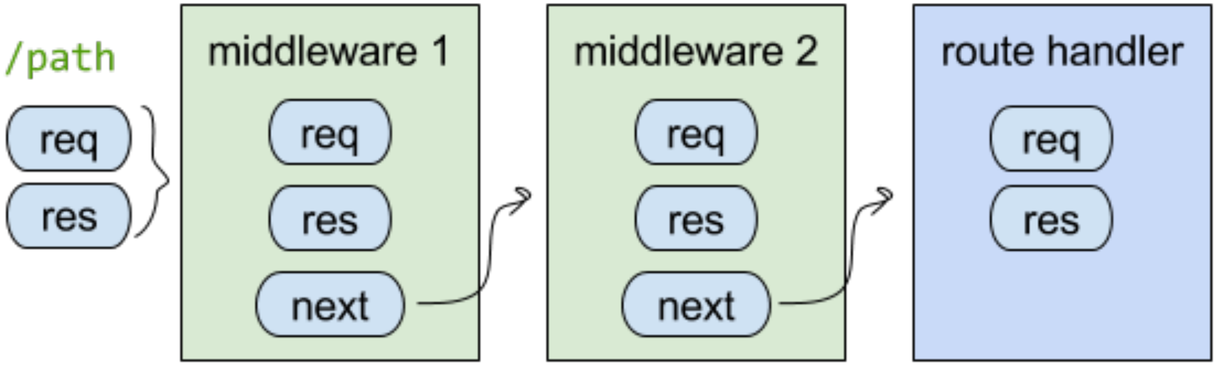
expressjs.com

Express Framework: Middleware

Express implements a chain-of-responsibility pattern that allows to put so-called middleware functions in-between the request-response cycle of your express application.

The middleware function

- has access to the request and response objects and can manipulate them,
- can execute code,
- optionally call the next middleware function via a callback function (which by convention is named next) or,
- end the request-response cycle, *i.e.,* let the route handler execute its code.



expressjs.com

Express Framework: Middleware

Let’s add a few middleware functions to our hotels router module. First, it would be nice to log all incoming requests. We print out the timestamp and the URL of the request.

routes/hotels.js

```
router.use((req, res, next) => {
 console.log('[' + Date.now() + '] ' + req.url);
 next();
});
```

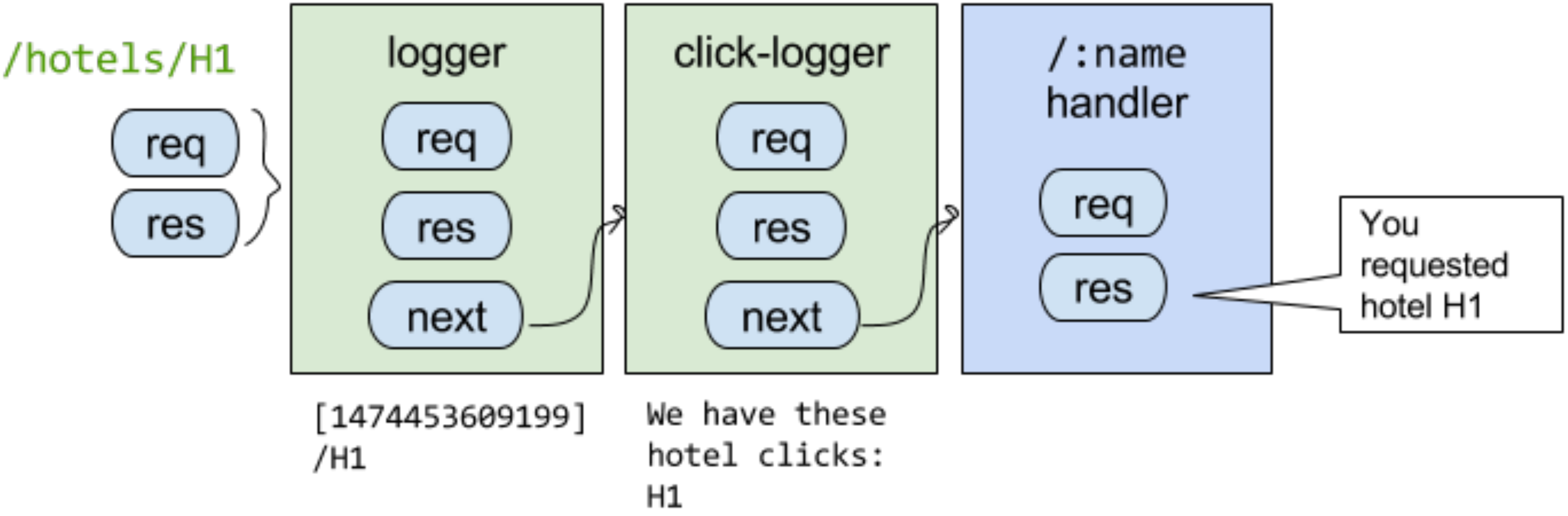
If we open our web browser at *http://localhost:3000/hotels/* and *http://localhost:3000/hotels/VulcanInn* nothing seems to have changed. If we look at our console, however, we now see log statements like these:

```
[1474449688267] /
[1474449688291] /VulcanInn
```

expressjs.com

Express Framework: Middleware

Let’s add another middleware function that intercepts requests for a particular hotel at the '/:name' path and pushes the hotel name request parameter into a clicks array which records our clickstream.



expressjs.com

Express Framework: Middleware

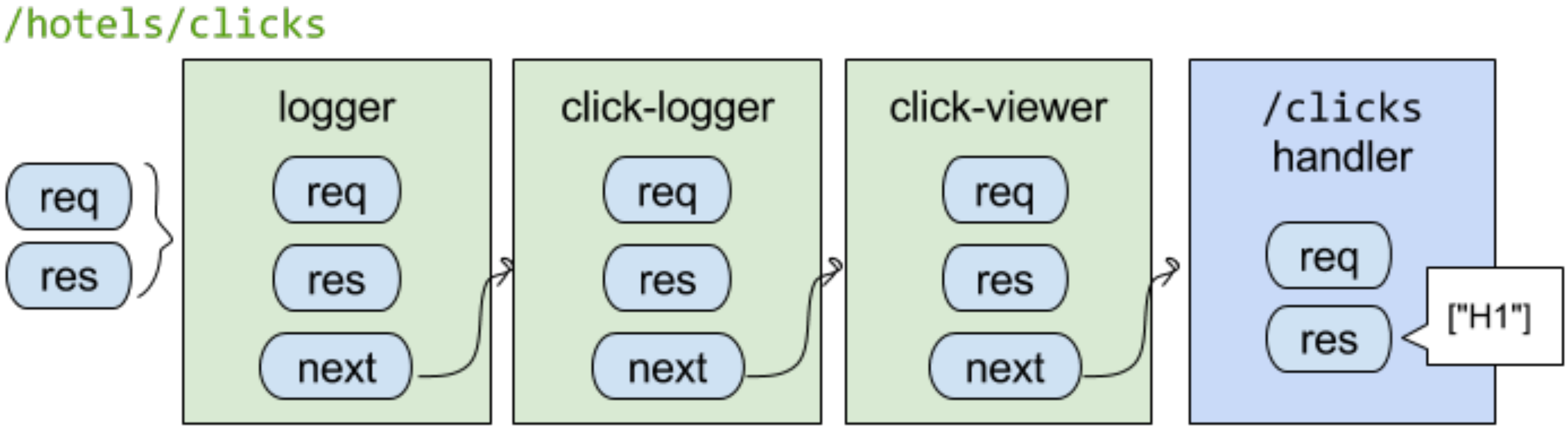
routes/hotels.js

```
const express = require('express');
const router = express.Router();
let clicks = [];

router.use('/:name', (req, res, next) => {
 if (req.params.name !== 'clicks') {
   clicks.push(req.params.name);
 }
 console.log('We have these hotel clicks: ' + clicks);
 next();
});

/* GET 1 hotel. */
router.get('/:name', (req, res) => {
 res.send('You requested hotel ' + req.params.name);
});
```

expressjs.com

Express Framework: Middleware

The next middleware function intercepts '/clicks' requests and sets our clicks array as an additional request parameter. Then we add a route handler for '/clicks' that passes the `req.clicks` array into the response.



```
[1474453609240]
/clicks
```

```
We have these
hotel clicks:
H1
```

```
req.clicks =
clicks;
```

expressjs.com

Express Framework: Middleware

routes/hotels.js

```
/* other middleware and routes */

router.use('/clicks', (req, res, next) => {
 req.clicks = clicks;
 next();
});

/* GET hotel clicks. */
router.get('/clicks', (req, res, next) => {
 res.send(req.clicks);
});
```

expressjs.com