WILEY

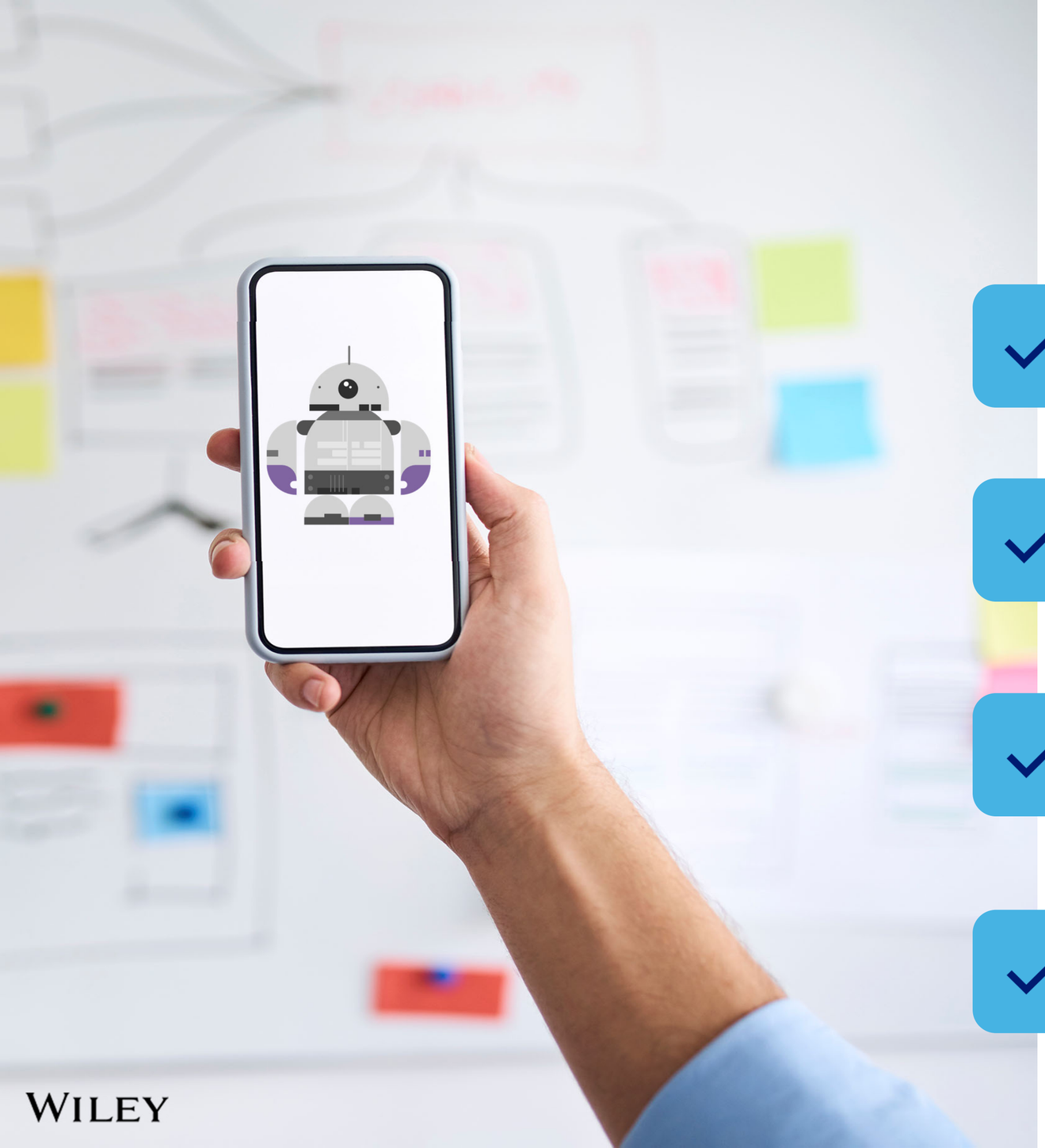
ENABLING DISCOVERY | POWERING EDUCATION | SHAPING WORKFORCES

NodeJS Fundamentals

Introduction to NodeJS

Objectives

- Working with File System Module
- File System Methods
- Node.js Web Server
- Building and Handling an HTTP server



Objectives

- Callbacks vs. Events
- Event Module and EventEmitter Class
- Streams in Node.js
- Readable Stream & Writable Stream

Working with File System

The ‘process’ Object

A collection of Streams

- process.stdin
- process.stdout
- process.stderr

Attributes of the current process

- process.env, process.argv
- process.pid, process.title
- process.uptime()
- process.memoryUsage() , process.cwd()

Process-related Actions

- Process.abort()
- process.chdir()
- process.kill()
- process.setgid() or process.setuid()

An instance of EventEmitter

- event: ‘exit’
- event: ‘uncaughtException’
- POSIX signal events (‘SIGINT’, etc.)

NodeJS File System

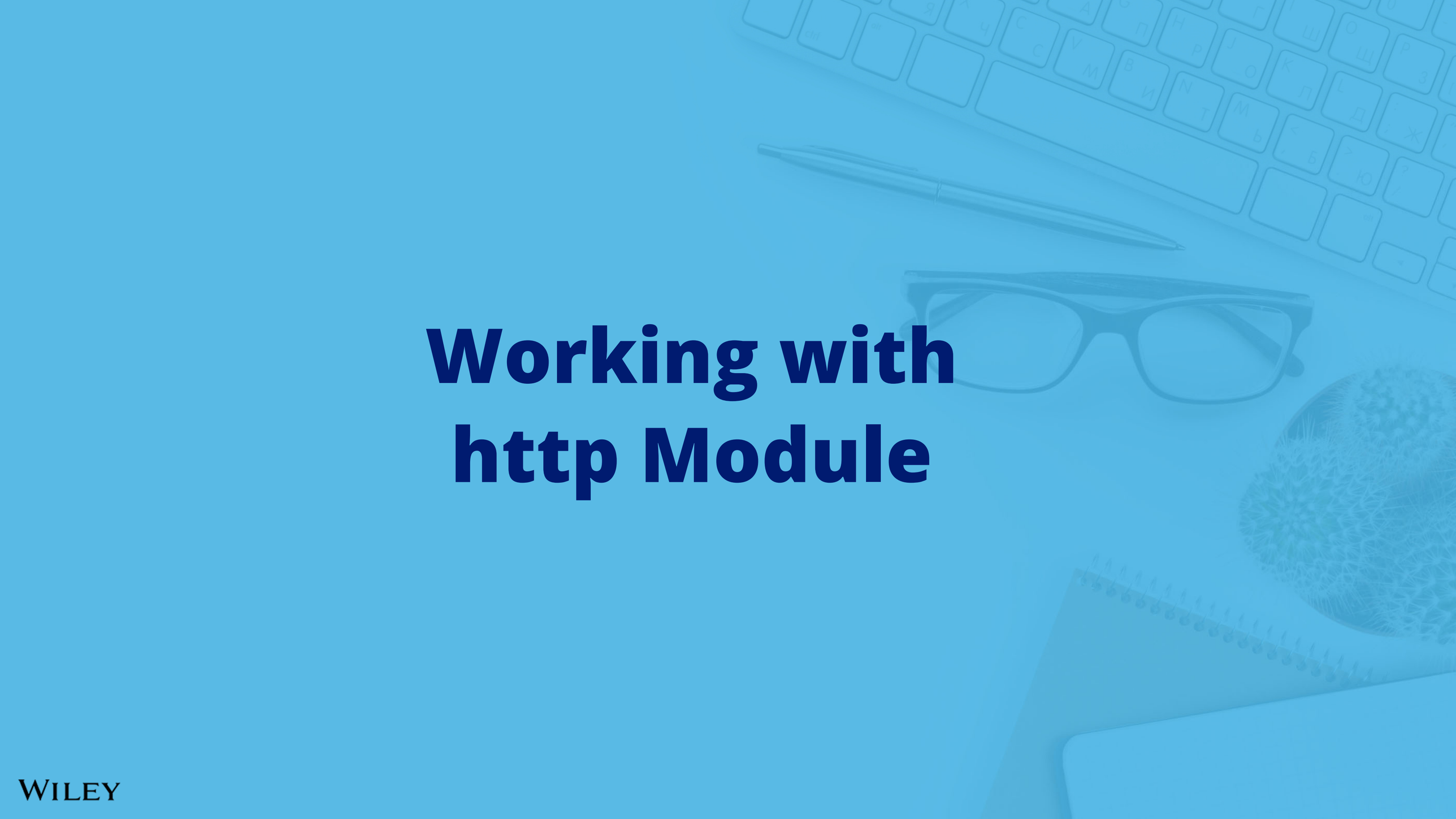
- Node.js includes fs module to access the physical file system.
- The file system offers file I/O and directory I/O functions.
- All file system functions have synchronous (blocking) and asynchronous (non-blocking) operations.
- The synchronous functions have “Sync” with their name and return the value directly and prevent Node from executing any code while the I/O is being performed.
- Asynchronous functions return the value as a parameter to a callback function.

NodeJS File System Functions

Method	Description
fs.readFile(fileName [,options], callback)	Reads existing file.
fs.writeFile(filename, data[, options], callback)	Writes to the file.
fs.open(path, flags[, mode], callback)	Opens file for reading or writing.
fs.rename(oldPath, newPath, callback)	Renames an existing file.
fs.chown(path, uid, gid, callback)	Asynchronous chown.
fs.stat(path, callback)	Returns fs.stat object which includes statistics.
fs.link(srcpath, dstpath, callback)	Links file asynchronously.
fs.symlink(destination, path[, type], callback)	Symlink asynchronously.

NodeJS File System Functions

Method	Description
fs.rmdir(path, callback)	Renames an existing directory.
fs.mkdir(path[, mode], callback)	Creates a new directory.
fs.readdir(path, callback)	Reads the content of the specified directory.
fs.utimes(path, atime, mtime, callback)	Changes the timestamp of the file.
fs.exists(path, callback)	Determines whether the specified file exists or not.
fs.access(path[, mode], callback)	Tests a user's permissions for the specified file.
fs.appendFile(file, data[, options], callback)	Appends new content to the existing file.
fs.rmdir(path, callback)	Renames an existing directory.

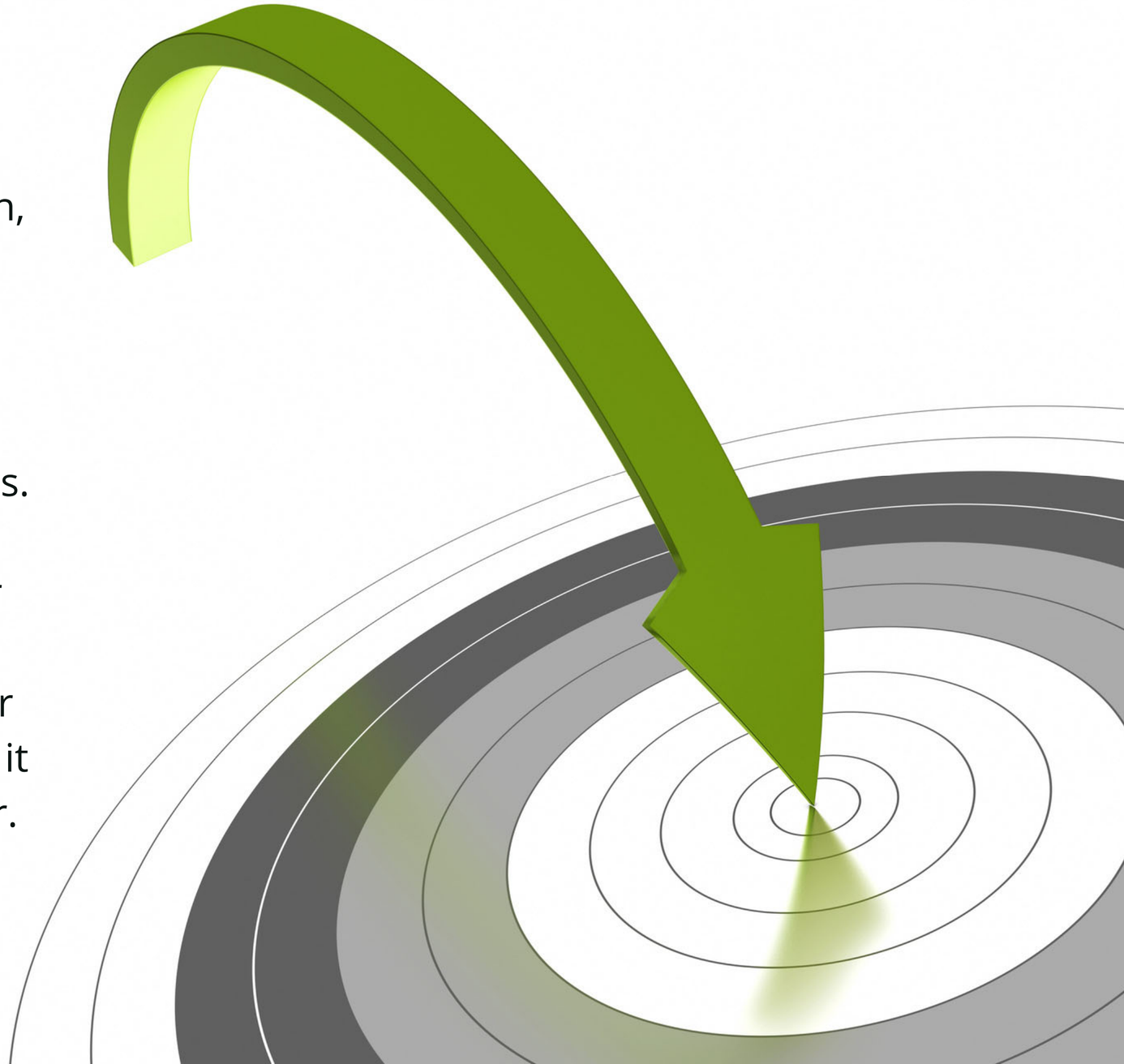


Working with http Module

NodeJS Web Server

- To access web pages of any web application, you need a web server. The web server will handle all the http requests for the web application e.g., IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

- Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.



BUILDING AN HTTP SERVER

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

Instance of http.ServerRequest (a ReadableStream)

var server = http.createServer(function(req, res) {
  // process request
});
server.listen(port, [host]);

Instance of http.ServerResponse (a WritableStream)
```

Building an HTTP server

- Each request is provided via either callback or as a “request” event on the server object.
- The ServerRequest can be read from (or piped) for POST uploads.
- The ServerResponse can be piped to when returning stream-oriented data in a response.
- SSL support is provided by a similar https.createServer().

HANDLE HTTP REQUEST

- **“options” can be A URL string or An object specifying values for host, port, method, path, headers, etc.**
- **The returned ClientRequest can be written/piped to for POST requests**
- **The ClientResponse object is provided via either callback (shown above) or as a “response” event on the request object.**

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

var req = http.request(options, function(res) {
  // process callback
});
```

Instance of http.ClientRequest (a WritableStream)

Instance of http.ClientResponse (a ReadableStream)

Working with http Module

Callback vs. Events

Callbacks:

```
getThem(param, function(err, items) {
  // check for error
  // operate on array of items
});
```

- Request / Reply
- No results until all results
- Either error or results

Events:

```
var results = getThem(param);

results.on('item', function(i) {
  // do something with this one item
});

results.on('done', function() {
  // No more items
});

results.on('error', function(err) {
  // React to error
});
```

- Publish / Subscribe
- Act on results as they arrive
- Partial results before error

EVENTS MODULE

Events Module:

- Node.js core API is based on asynchronous event-driven architecture.
- Here, the events module is used for event handling.
- The events module exposes EventEmitter class to deal with events.

EventEmitter Class:

- The event module includes EventEmitter class. Event Emitters are objects that generate events.
- All objects that emit events are instances of the EventEmitter class.
- Event Emitters are based on publish/subscribe pattern.
- EventEmitter class provides Listeners for listing events and Emitters for emitting events.

EventEmitter Class

The publisher:

```
emitter.emit(event, [args]);
```

The subscriber:

```
emitter.on(event, listener);
```

- The “event” can be any string
- An event can be emitted with zero or more arguments
- The set of events and their arguments constitute a “interface” exposed to the subscriber by the publisher (emitter).

Two common patterns for EventEmitter:

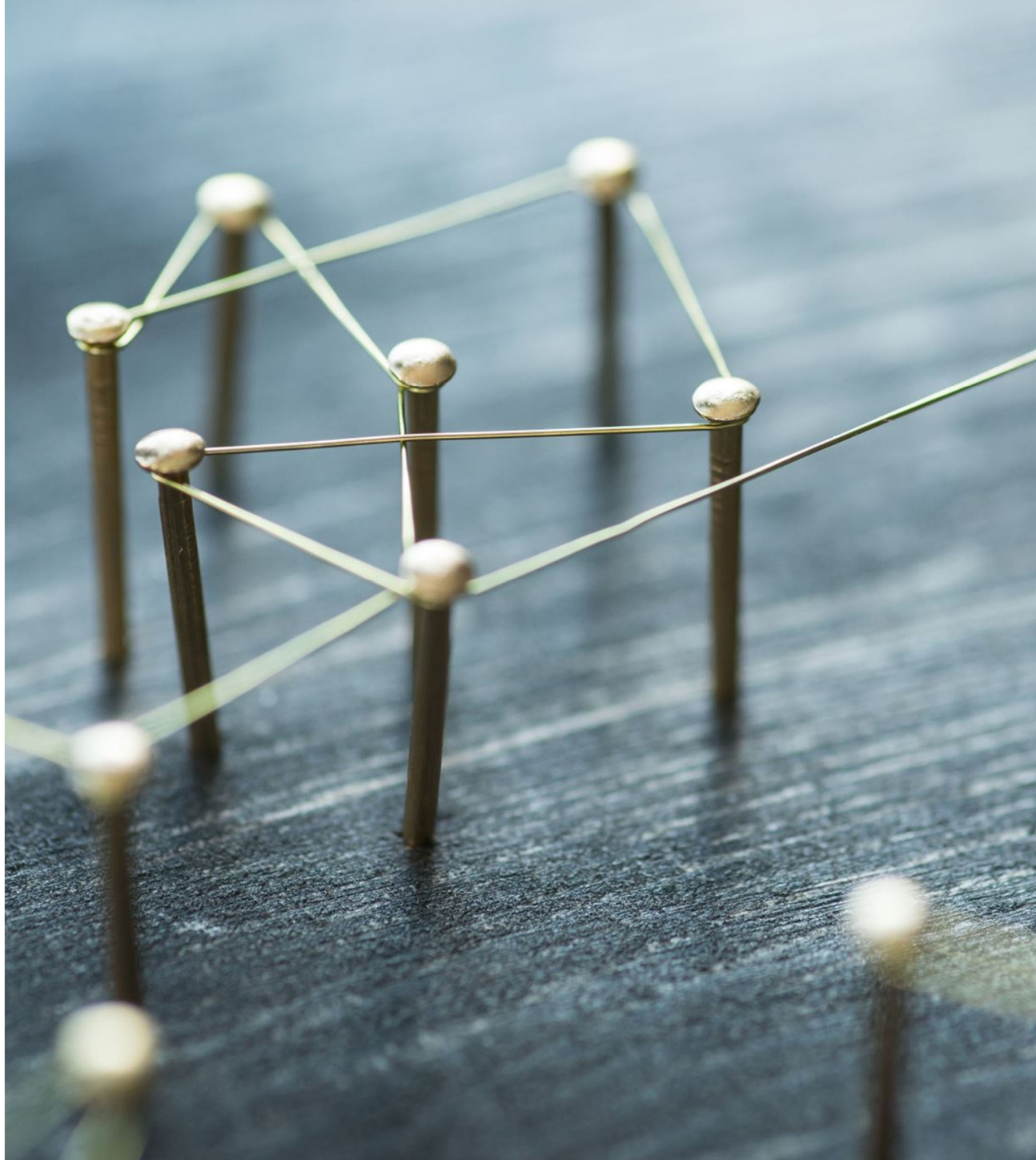
- Return EventEmitter from a function and Extend the EventEmitter class

Streams in NodeJS

- All streams are instances of EventEmitter.
- A stream is a unified abstraction for managing data flow, including:
- Network traffic (http requests & responses, tcp sockets)
- File I/O
- stdin / stdout / stderr and more!

A stream is an instance of either

- ReadableStream
- WritableStream
- … or both! (Duplex)



ReadableStream & WritableStream

ReadableStream

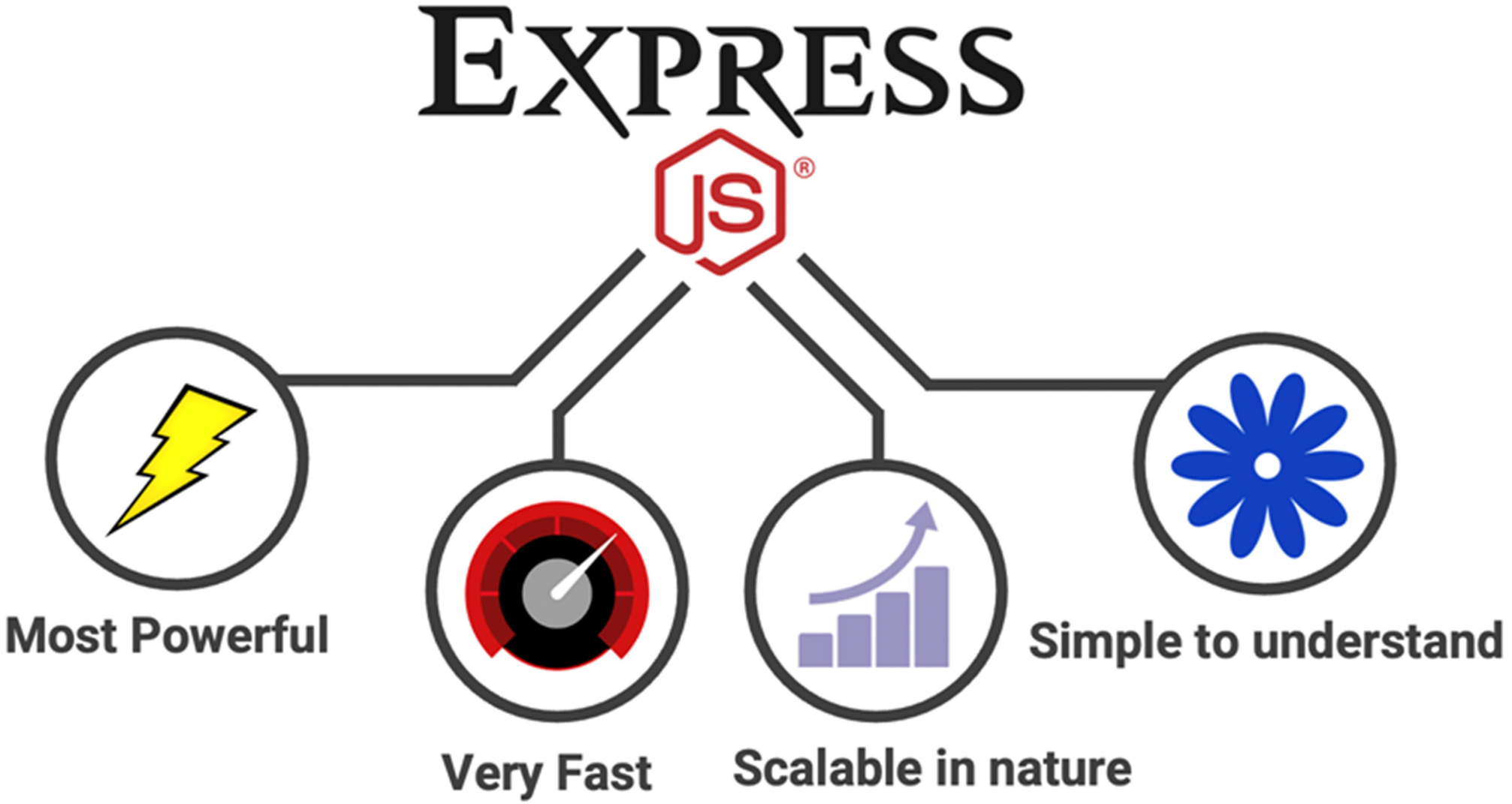
- readable [boolean]
- event: ‘data’
- event: ‘end’
- event: ‘error’
- event: ‘close’
- pause() & resume()
- destroy()
- pipe()

WritableStream

- writable [boolean]
- event: ‘drain’
- event: ‘error’
- event: ‘close’
- event: ‘pipe’
- write()
- end() or destroy()
- destroySoon()

Express – Web Framework

Why Express.js?



What is Express.js?

- Express is a minimal and flexible Node.js web application framework.
- This framework provides a robust set of features for web and mobile applications.
- Creating a robust API is quick and easy with a myriad of HTTP uniform methods.
- Express provides a thin layer of fundamental web application features.
- Many popular frameworks are based on Express.

Express application generator

Use the application generator tool, express-generator, to quickly create an application skeleton.
You can run the application generator with the npx command (available in Node.js 8.2.0).

```
$ npx express-generator
```

- For earlier Node versions, install the application generator as a global package and then launch it

```
$ npm install -g express-generator
$ express
```

Serving static files in Express

- To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.The function signature is:

```
express.static(root, [options])
```

- The root argument specifies the root directory from which to serve static assets. For more information on the options argument, see express.static. **For example,** use the following code to serve images, CSS files, and JavaScript files in a directory named public:

```
app.use(express.static('public'))
```

WRITING MIDDLEWARE FOR EXPRESS

Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware. 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 in the stack.

Thank You