SERVER-SIDE

NODE JS

CROSS-CUTTING THEMES

JAVASCRIPT - THE GOOD PARTS

NODE JS

VERSION

v1.2 7 November 2021

v1.1 4 November 2021

v1.0 26 October 2021

ACKNOWLEDGMENTS

THANKS TO:

- Hamzeh Roumani, who has shaped EECS-4413 into a leading hands-on CS course at EECS and who generously shared all of his course materials and, more importantly, his teaching philosophy with me;
- Parke Godfrey, my long-suffering Master's supervisor and mentor; and
- Suprakash Datta for giving me this opportunity to teach this course.

PRINTABLE VERSION OF THE TALK

Download PDF

CROSS-CUTTING THEMES

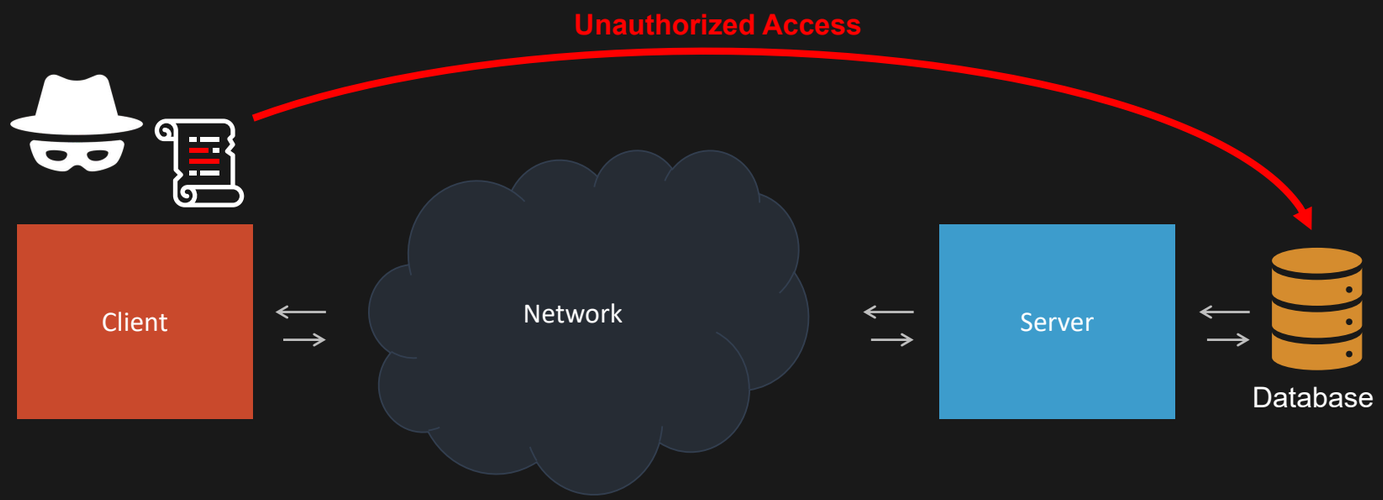
- Security
- Modularity
- Versatility
- Interoperability
- Scalability
- Telemetry

SECURITY

NEVER TRUST THE CLIENT

Always assume all input or data coming from the client might be sent maliciously, because you can't control it. This means authentication and authorization must be checked and enforced server-side. All input must be validated. Never trust the user; there is nothing on the client side that can prevent them from entering dangerous, unexpected input. Always validate input and perform server-side checks on everything. Assume a malicious actor can tampered with the client and its implementation (not just webpages but mobile applications as well). Never provide direct access to the databases.

An API lets us control what features, capabilities and resources the clients have access to and helps us minimize the attack surface a malicious actor can exploit against our systems.

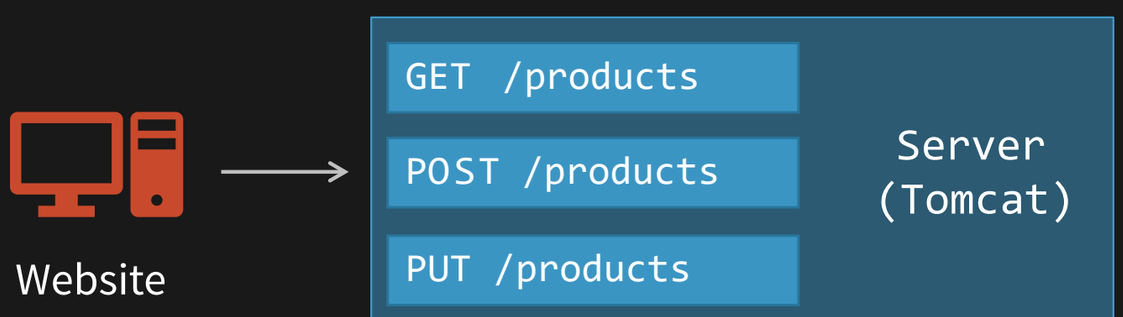


Confidentiality	Protection against disclosure
	Encrypt: link level or end-to-end
	Authentication: Passwords, Accounts, OAuth
Integrity	Protection against alteration
	SQL Injection: Always sanitize incoming parameters and use prepared queries
Availability	Protection against interference
	Network based Measures: IP filtering, black and white lists, DoS, DDoS, …

MODULARITY + VERSATILITY

COMPONENTS SHOULD BE LOOSELY-COUPLED WITH MULTIPLE CLIENT APPS SHARING THE SAME BACKEND

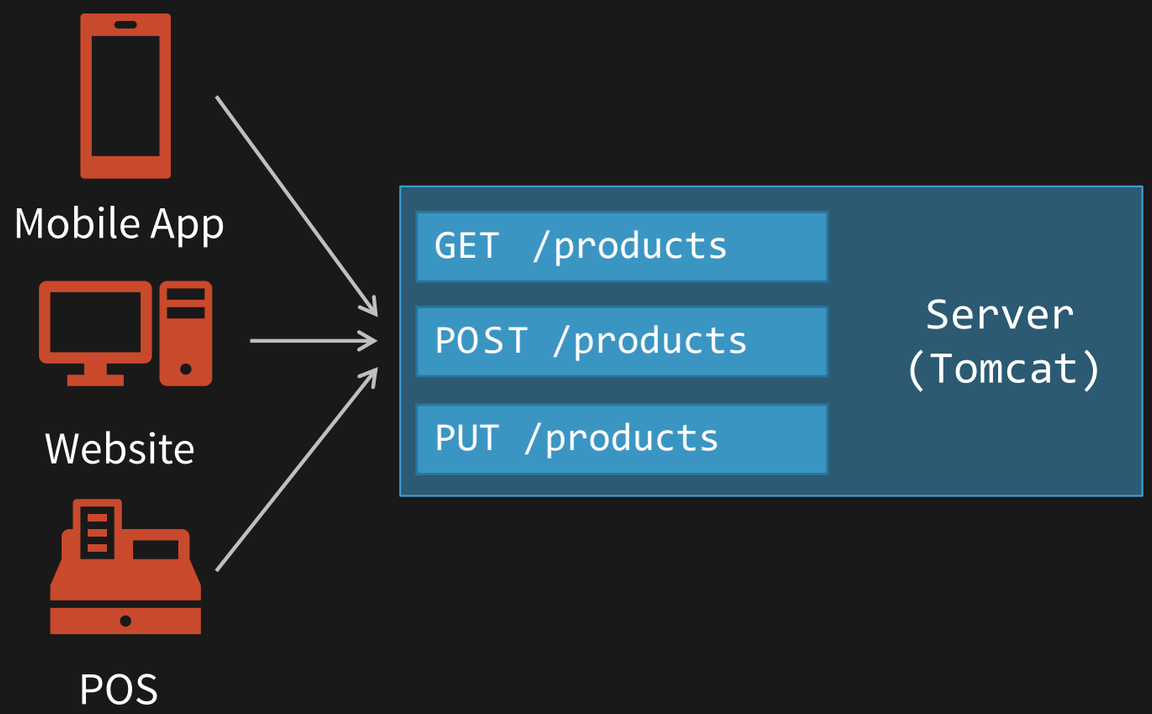
- With an API, the server and clients can evolve independently.
- Allows our system to be platform- and technology-agnostic.



MODULARITY + VERSATILITY

COMPONENTS SHOULD BE LOOSELY-COUPLED WITH MULTIPLE CLIENT APPS SHARING THE SAME BACKEND

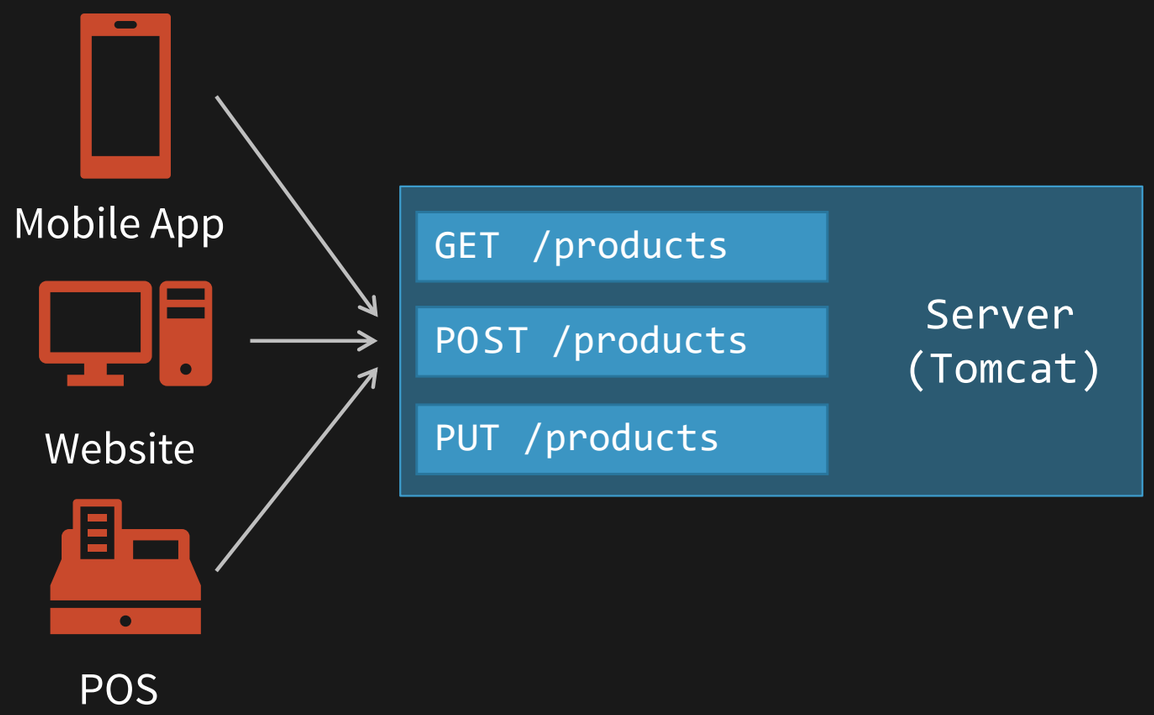
- With an API, the server and clients can evolve independently.
- Allows our system to be platform- and technology-agnostic.
- Multiple different client implementations can use the same API:
  - Web applications
  - Mobile apps
  - Desktop apps
  - Point-of-sales terminals



MODULARITY + VERSATILITY

COMPONENTS SHOULD BE LOOSELY-COUPLED WITH MULTIPLE CLIENT APPS SHARING THE SAME BACKEND

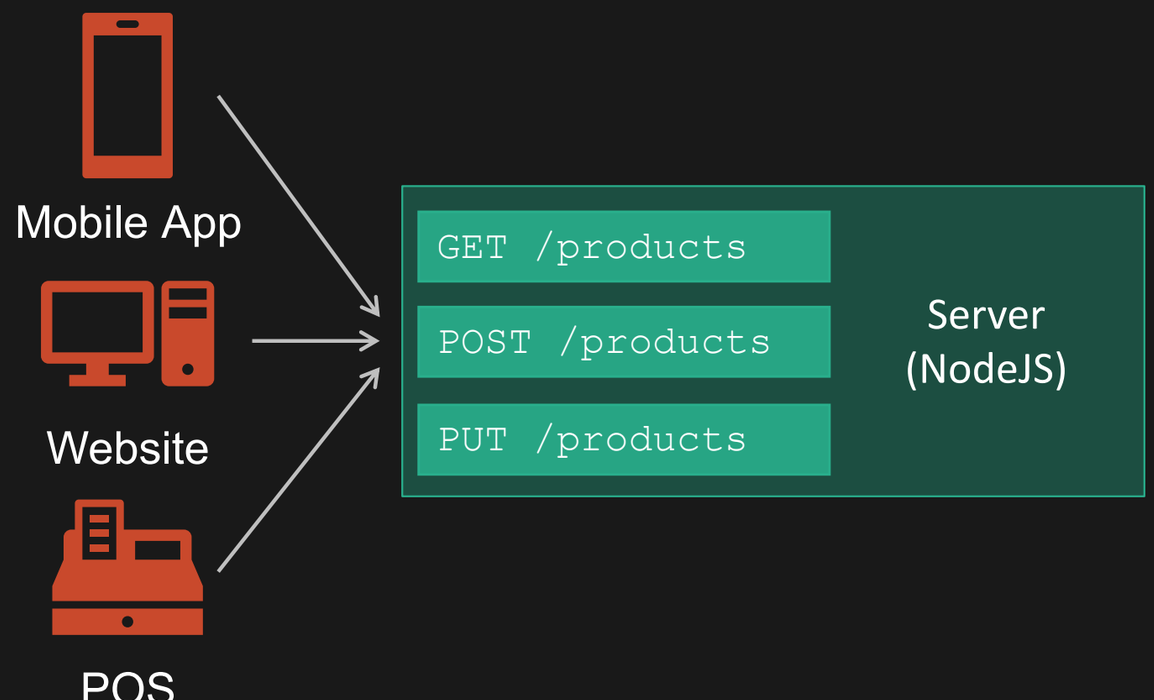
- With an API, the server and clients can evolve independently.
- Allows our system to be platform- and technology-agnostic.
- Multiple different client implementations can use the same API:
  - Web applications
  - Mobile apps
  - Desktop apps
  - Point-of-sales terminals
- All apps share the same API, so data is synchronized and consistent.
- If multiple apps are implemented in the same language, they can share the same code for accessing the API.
- We can publish our APIs so that 3rd-parties can use them, build 3rd-party apps or extend our APIs as 3rd-party web services.



INTEROPERABILITY

COMPONENTS SHOULD BE LOOSELY-COUPLED AND REPLACEABLE WITHOUT BREAKING OTHER COMPONENTS

- With an API, the server and clients can evolve independently.
- Allows our system to be platform- and technology-agnostic.
- That means we can reimplement the server in any programming language we want.
- As long as the new implementation replicates all of the existing features and API remains the same as before, the clients should not notice any difference.
- That gives us, implementers the flexibility to choose the technology that best suits our application's requirements as oppose to being locked into a certain technology or programming language; the democraticization of technology.



SCALABILITY

Scalability: the property of the systems of remaining efficient where there is a significant increase in the number of users and resources.

- Controlling the cost of physical resources
- Controlling the performance loss
- Preventing software resources from running out
- Avoiding performance bottlenecks

TECHNIQUES

- Multithreading
- Throttling
- Thread Pooling
- Scale through containers (e.g. Docker, Kubernetes)
- Scale through boxes (pods and nodes) or VMs
- Load Balancing
- Auto Scaling

TELEMETRY

- System Logs
- Performance Timing
- Metrics
- Analytics
- Data Mining
- Machine Learning

FAILURE HANDLING

- If a part of the system fails, the system should still functions
- Detecting failure:
  - Some failures can be detected: for example a checksum can detect corrupted data in a message, a server not responding
- Masking failures: when a failure is detected it can be hidden or made less severe
  - Messages can be retransmitted or corrected, a server is restarted
- Tolerating failure: if a browser cannot contact a server, it does not try forever; it gives up and inform the user
- Recovery from failures: the software should be able to “roll-back” to a stable and know state
- Redundancy: services can tolerate failures by using redundant components; most web applications run on clusters
- Unsolved problems: denial of service

JAVASCRIPT

THE GOOD PARTS

Source: The JavaScript Trilogy and Crockford on JavaScript presented by Douglas Crockford in 2007 and 2011.

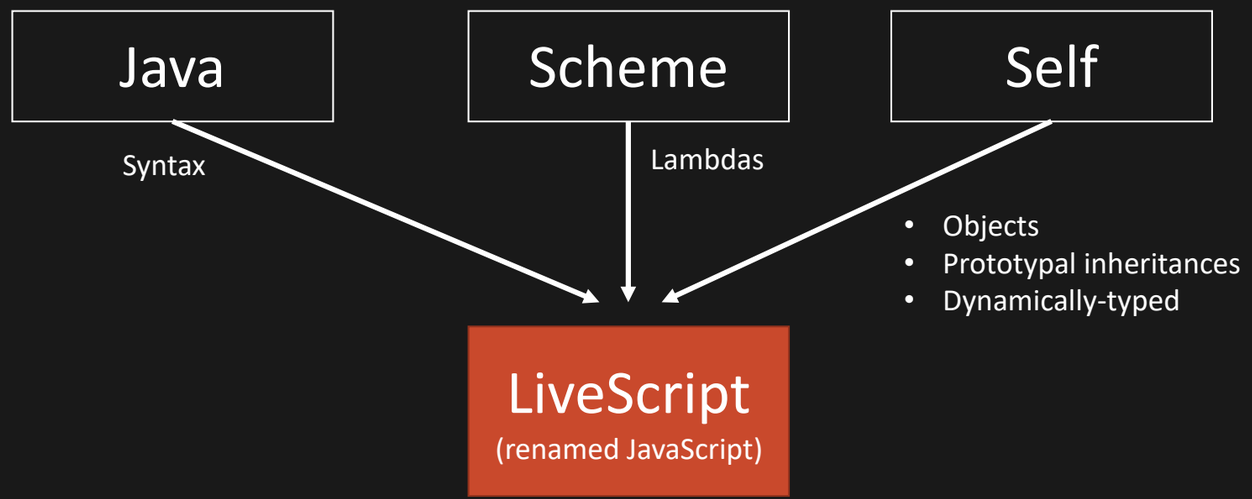
JAVASCRIPT

During these formative years of the Web, web pages could only be static, lacking the capability for dynamic behavior after the page was loaded in the browser. In 1995, Netscape decided to add a scripting language to Netscape Navigator. They pursued two routes to achieve this: collaborating with Sun Microsystems to embed the Java programming language, while also hiring Brendan Eich to embed a scripting language in the browser.

Brendan Eich to devise a scripting language, with syntax similar to Java, but borrowed a lot of ideas from Scheme for lambda functions and Self for dynamic typing and prototypal object-oriented inheritance. Although the new language and its interpreter implementation were called LiveScript when first shipped as part of a Netscape Navigator beta in September 1995, the name was changed to JavaScript for the official release in December 1995.

Microsoft reverse-engineered Netscape Navigator's JavaScript interpreter to create its own, called JScript. In November 1996, Netscape submitted JavaScript to Ecma International, as the starting point for a standard specification that all browser vendors could conform to. This led to the official release of the first ECMAScript language specification in June 1997.

Source: JavaScript - Wikipedia



JAVASCRIPT RESOURCES:

- JavaScript - Mozilla Developer Network
- ECMAScript 2021 (ECMA-402), June 2021
- The JavaScript Trilogy (2007)
- Crockford on JavaScript (2011)

JAVASCRIPT'S KEY IDEAS

- Load and go delivery
- Loose Typing
- Objects as general containers
- Prototypal inheritance
- Lambda
- Linkage through global variables

JAVASCRIPT AND JAVA

JavaScript and Java are similar in some ways but fundamentally different in some others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript follows most Java expression syntax, naming conventions and basic control-flow constructs which was the reason why it was renamed from LiveScript to JavaScript.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Source: JavaScript Guide - Mozilla Developer Network

JAVASCRIPT

Object-oriented. No distinction between types of objects.Inheritance is through the prototype mechanism, and properties and methods canbe added to any object dynamically.

Variable data types are not declared (dynamic typing, loosely typed).

Cannot automatically write to hard disk.

JAVA

Class-based. Objects are divided into classes and instanceswith all inheritance through the class hierarchy. Classes and instances cannothave properties or methods added dynamically.

Variable data types must be declared (static typing, strongly typed).

Can automatically write to hard disk.

JAVASCRIPT TYPES

PRIMITIVE TYPES

- `string`
- `number`
- `boolean`

COMPLEX TYPES

- `object`
- `function`

EMPTY TYPES

- `null`
- `undefined`

BUILT-IN OBJECTS

- `Object`
- `Function`
- `Array`
- `Date`
- `String`
- `Number`
- `Boolean`
- `Symbol`
- `RegExp`
- `Error`
- `Promise`
- `Map`
- `WeakMap`

STATIC OBJECTS

- `Math`
- `JSON`

For the full list of standard built-in objects, refer to here.

JAVASCRIPT GLOBALS

VALUE PROPERTIES

These global properties return a simple value. They have no properties or methods.

- `Infinity`
- `NaN`
- `undefined`
- `globalThis`

FUNCTION PROPERTIES

These global functions—functions which are called globally, rather than on an object—directly return their results to the caller.

- `eval()`
- `isFinite()`
- `isNaN()`
- `parseFloat()`
- `parseInt()`
- `encodeURI()`
- `encodeURIComponent()`
- `decodeURI()`
- `decodeURIComponent()`

JAVASCRIPT - typeof

You can use the `typeof` operator to find the data type of a JavaScript variable.

```
typeof "John"                    // Returns "string
typeof 3.14                      // Returns "number
typeof NaN                       // Returns "number
typeof false                     // Returns "boolea
typeof [1,2,3,4]                 // Returns "object
typeof {name:'John', age:34}     // Returns "object
typeof new Date()                // Returns "object
typeof function () {}            // Returns "functi
typeof myCar                     // Returns "undefi
typeof null                      // Returns "object
```

PLEASE OBSERVE:

- `NaN` is `number`.
- An `Array` is `object`.
- A `Date` is `object`.
- `null` is `object`.
- An `undefined` variable is `undefined`.
- A variable that has not been assigned a value is also `undefined`.

NODE JS

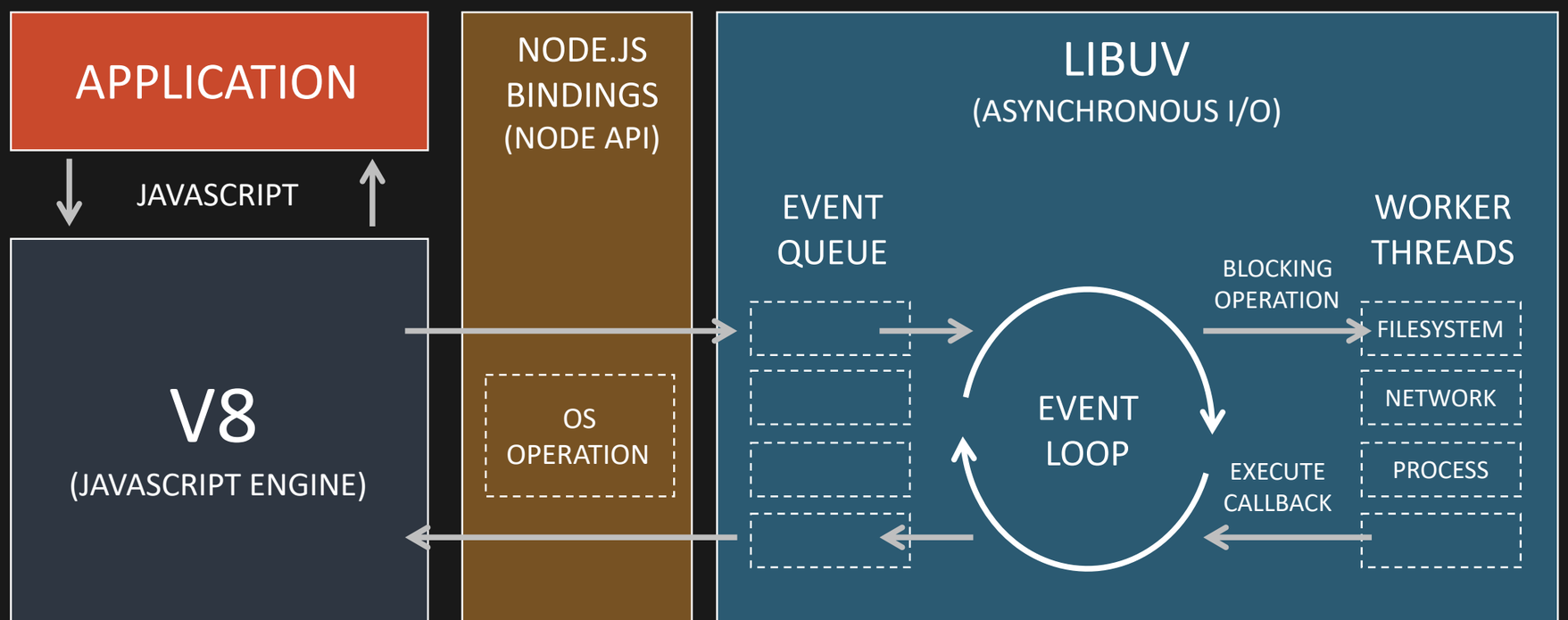
SERVER-SIDE JAVASCRIPT

ABOUT NODE.JS

Node.js is an asynchronous event-driven JavaScript runtime, designed to build scalable network applications. It is an open-source, cross-platform, back-end runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. It enables developers to use JavaScript to write command line tools and server-side scripts. Developed in 2009 by Ryan Dahl.

Node.js is similar in design to, and influenced by, systems like Ruby's Event Machine and Python's Twisted. Node.js takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems, there is always a blocking call to start the event-loop.

NODE JS: THE SYSTEM



NPM PACKAGE MANAGER

npm is a package manager for the JavaScript programming language maintained by npm, Inc.

npm is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called `npm`, and an online database of public and paid-for private packages, called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website. The package manager and the registry are managed by npm, Inc.

npm stands for Node Package Manager. It was released back in 2010, beginning a new era in web development. Until then, the project dependencies were downloaded and managed manually. **npm** was the magic wand that pushed the Web to the next level.

npm actually involves three things:

- a website for managing various aspects of your npm experience
- a registry for accessing an extensive public database of JavaScript packages
- a command-line interface (CLI) for interacting with **npm** via the terminal

Yarn stands for **Yet Another Resource Negotiator**. The Yarn package manager is an alternative to npm, released by Facebook in October 2016. The original goal of Yarn was to deal with npm drawbacks, such as performance and security issues. Yarn was positioned as a safe, fast, and reliable JavaScript dependency management tool. But the npm team learned their lesson and rapidly filled the npm gaps by implementing the missing features.

```
$ npm init
$ npm install -s express ...
```

package.json

```
{
  "name": "node-project-example",
  "version": "1.0.0",
  "main": "index.js",
  "dependencies": {
    "express": "^4.17.1",
    "express-session": "^1.17.2",
    "lodash": "^4.17.21",
    "sequelize": "^6.7.0",
    "sequelize-cli": "^5.0.0",
    "sqlite3": "^4.2.0"
  }
}
```

TCP CLIENT

```
1  const { createConnection } = require('net');
2
3  const host   = process.argv[2];
4  const port   = process.argv[3];
5  const client = createConnection(port, host);
6
7  let response = [];
8  let connection;
9
10 client.on('error', () => console.log(`Error when connecting ${connection}.`));
11 client.on('close', () => console.log(`Disconnected from ${connection}`));
12 client.on('connect', () => {
13   connection = `${client.remoteAddress}:${client.remotePort}`;
14   console.log(`Connected to ${connection}`);
15   client.write('...'); // send request
16   client.end();
17 });
18 client.on('data', (chunk) => response.push(chunk));
19 client.on('end', () => {
20   console.log(response.join());
21 });
```

TCP SERVER

```
1  const { createServer } = require('net');
2
3  const port   = process.argv[2] || 4413;
4  const server = createServer();
5
6  server.listen(port, () => {
7    const address = `${server.address().address}:${server.address().port}`;
8    console.log(`Server Listening at: ${address}`);
9  });
10
11 server.on('connection', (socket) => {
12   const client = `${socket.remoteAddress}:${socket.remotePort}`;
13   console.log(`Connection from ${client}`);
14   socket.on('error', (err) => console.log(`Error: ${err}`));
15   socket.on('end', () => console.log(`Closing connection with ${client}`));
16   socket.on('data', function (request) {
17     socket.write('...'); // send response
18     socket.end();
19   });
20 });
```

HTTP CLIENT

```
1  const { get } = require('http'); // require('https')
2
3  const request = get(process.argv[2]); // send GET request
4
5  request.on('error', (e) => console.error(`Connection Error: ${e.message}`));
6  request.on('response', (res) => {
7    const { statusCode } = res;
8    const contentType = res.headers['content-type'];
9    // error-checking ...
10   let response = [];
11   res.setEncoding('utf8');
12   res.on('data', (chunk) => response.push(chunk));
13   res.on('end', () => {
14     let data = response.join();
15     // use the response data ...
16   });
17 });
```

HTTP SERVER

```
1  const { createServer } = require('http'); // require('https')
2
3  const port   = process.argv[2] || 4413;
4  const server = createServer();
5
6  server.listen(port, () => {
7    const address = `${server.address().address}:${server.address().port}`;
8    console.log(`Server Listening at: ${address}`);
9  });
10
11 server.on('request', (req, res) => {
12   console.log(`Request Received`);
13
14   res.writeHead(200, { 'Content-Type': 'application/json' });
15   res.write('...'); // send response
16   res.end();
17 });
```

DATABASE ACCESS IN NODE.JS

```
1  const os      = require('os');
2  const path    = require('path');
3  const sqlite3 = require('sqlite3');
4
5  const dbfile  = '4413/pkg/sqlite/Models_R_US.db';
6  const dbpath  = path.join(os.homedir(), ...dbfile.split('/'));
7  const db      = new (sqlite3.verbose()).Database(dbpath);
8
9  module.exports = {
10   getProductsByVendor(venId, success, failure) {
11     db.all('SELECT * FROM Product WHERE venId = ?', [venId], (err, rows) => {
12       if (err == null) {
13         success(rows);
14       } else {
15         failure(err);
16       }
17     });
18   }
19 };
```

Reference: https://github.com/mapbox/node-sqlite3/wiki/API

This slide is intentionally left blank.

Return to Course Page or Part II.