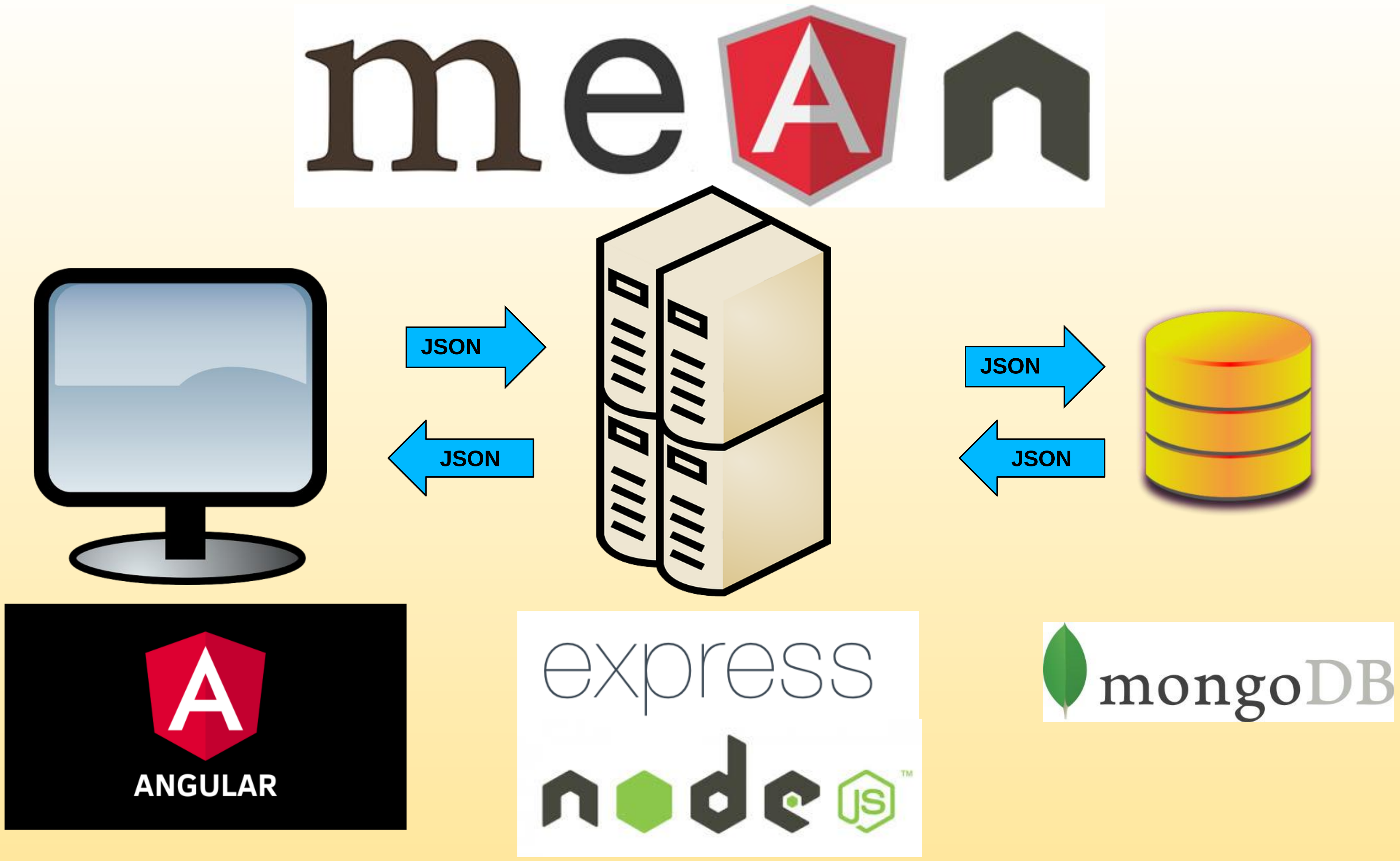
Advance Web Technologies & Programming (CSC350)

Lecture 5

Node Modules and MEAN

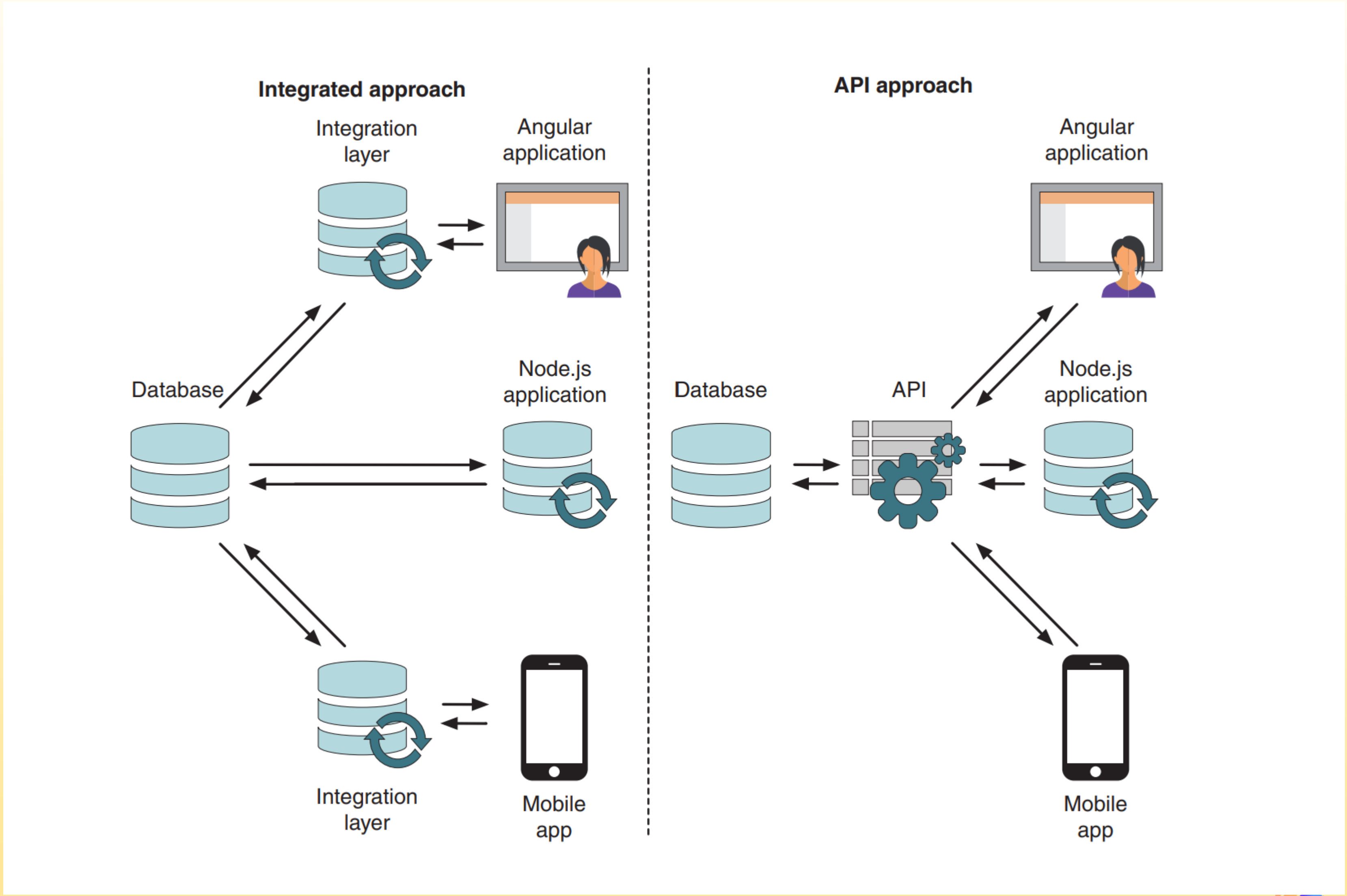
Road Map

- How Mean Stack works?
- Introduction to Node.JS
  - Concurrency
  - Event Loop/Event Emitter
  - Non Blocking I/O
  - Performance
- Node.JS Modules
- NPM
- Installation
- Getting Started



Collection of JavaScript based technologies used to develop web applications.

Data Integration Differences



What is NodeJS?

- A JavaScript runtime environment running Google Chrome's V8 engine
  - a.k.a. a server-side solution of JS
  - Compiles JS, making it really fast
- Runs over the command line
- Designed for high concurrency
  - Without threads or new processes
- Never blocks, not even for I/O
- Uses the CommonJS framework
  - Making it a little closer to a real OO language

Concurrency: The Event Loop

- Node.JS is single threaded.
  - But it can support concurrency via the concept of **event** and **callbacks**.
- Instead of threads Node uses an event loop with a stack
- Alleviates overhead of context switching.
- Node uses observer pattern.
- Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Event Loop

- Every API of Node.js is asynchronous and being single-threaded
- JavaScript engine maintains several queues of unhandled tasks.
  - These queues include things such as events, timers, intervals, and immediates.
  - Each execution of the event loop, known as a cycle, causes one or more tasks to be dequeued and executed.

Event Loop Features

- An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more tasks.
- The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
- The event loop allows us to use callbacks and promises.
- The event loop executes the tasks starting from the oldest first.

Event Loop Features

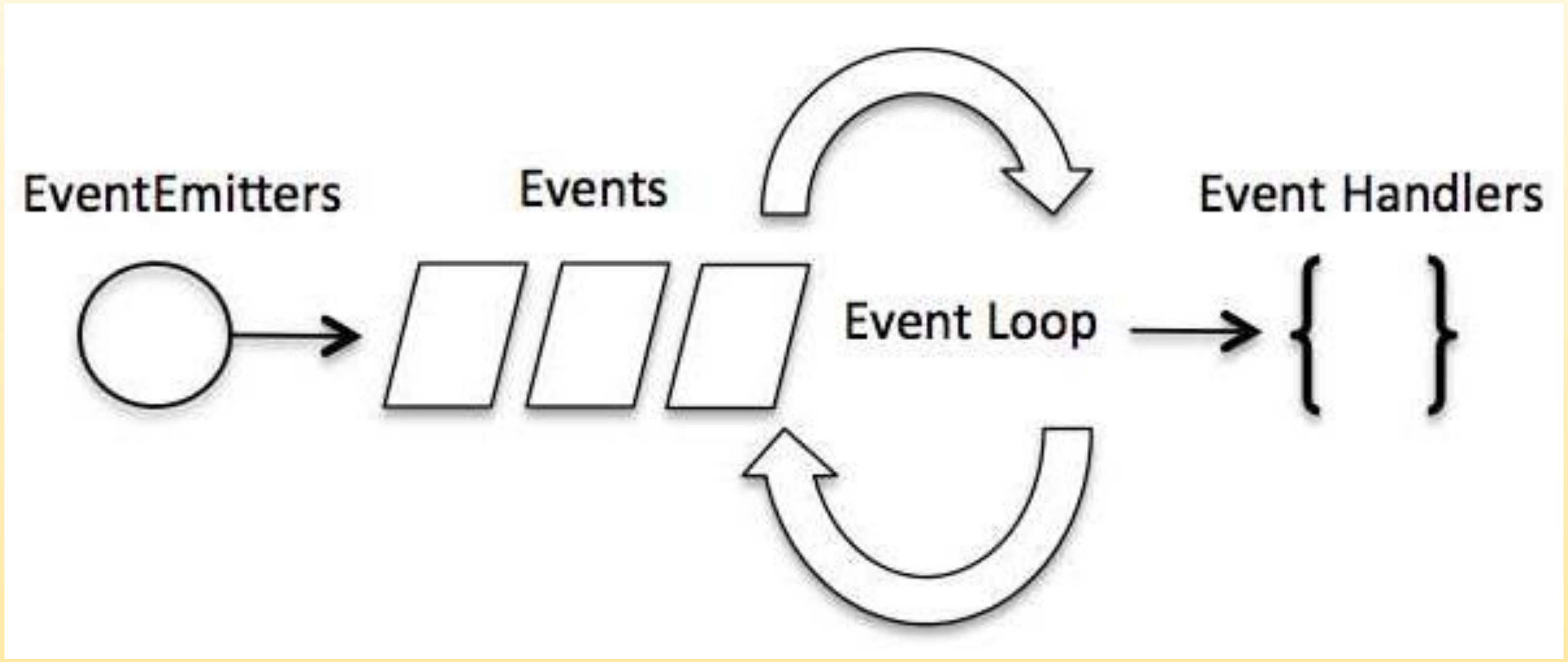
- **Explanation:**
- First statement is pushed to the call stack, executed and task is popped from the stack.
- setTitmeout is pushed to the queue and task is sent to the OS and the timer is set for the task. The task is then popped from the stack.
- The third statement is pushed to the stack, executed and popped from the stack.

```
console.log("This is the first statement");

setTimeout(function(){
                console.log("This is the second statement");
}, 1000);

console.log("This is the third statement");
```

Output:

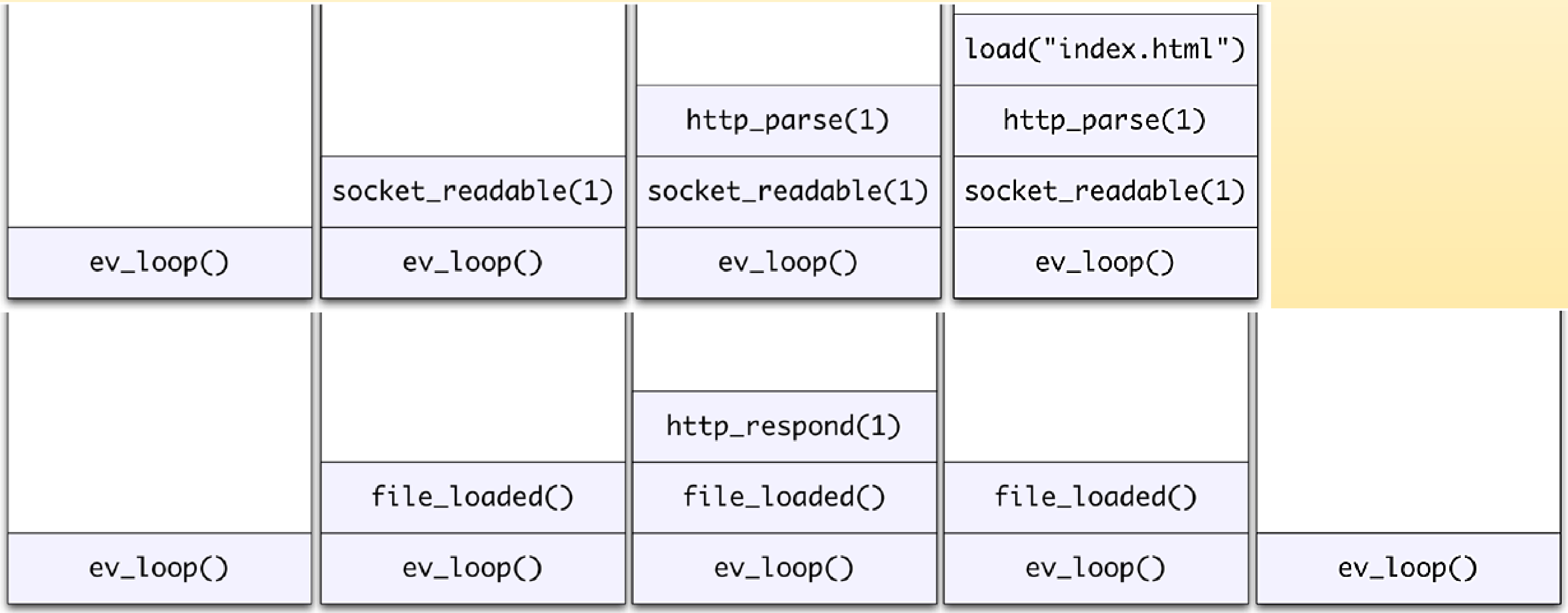
```
This is the first statement
This is the third statement
This is the second statement
```

Event Handling

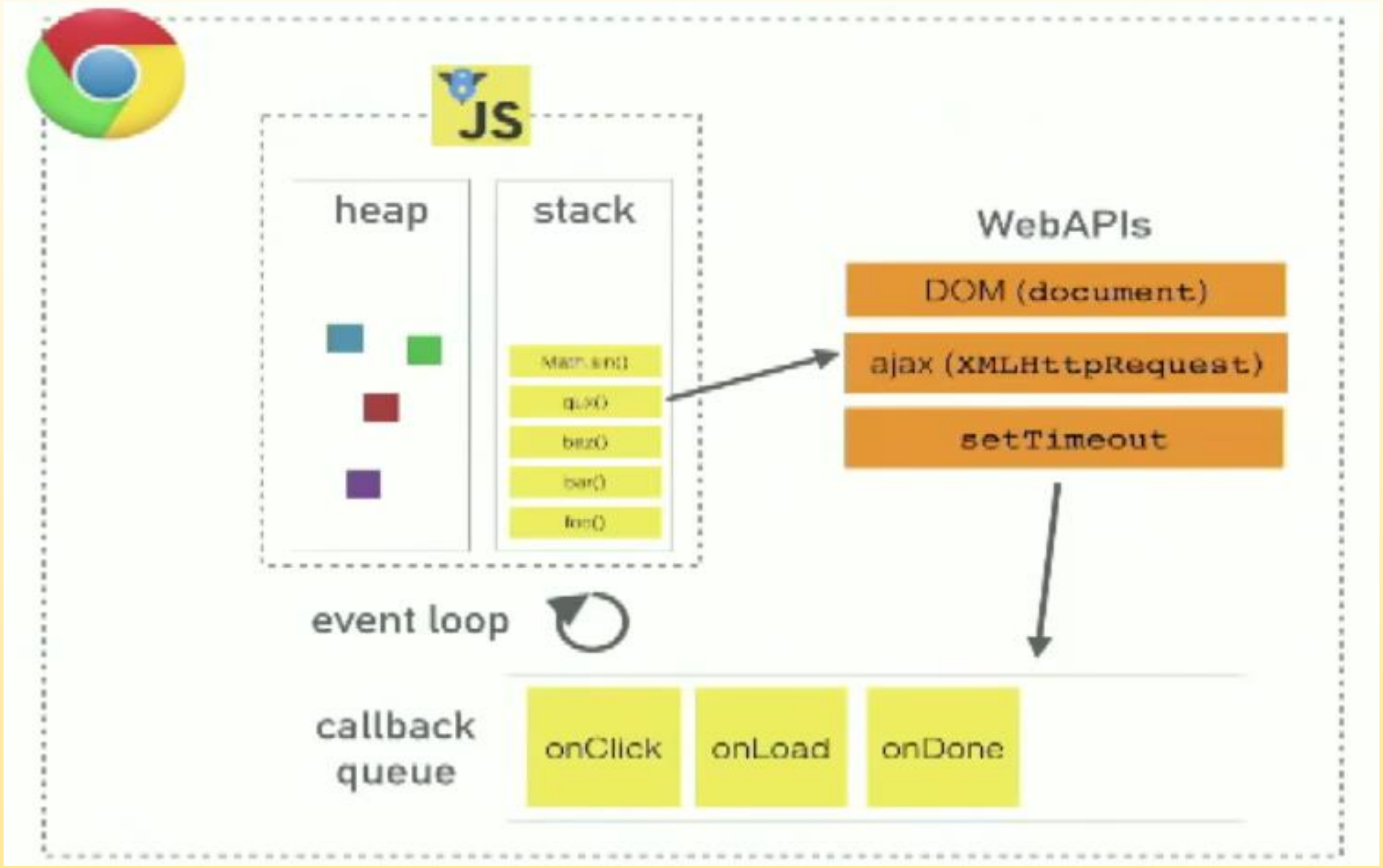


Event Loop Example

- Request for “index.html” comes in
- Stack unwinds and ev_loop goes to sleep
- File loads from disk and is sent to the client

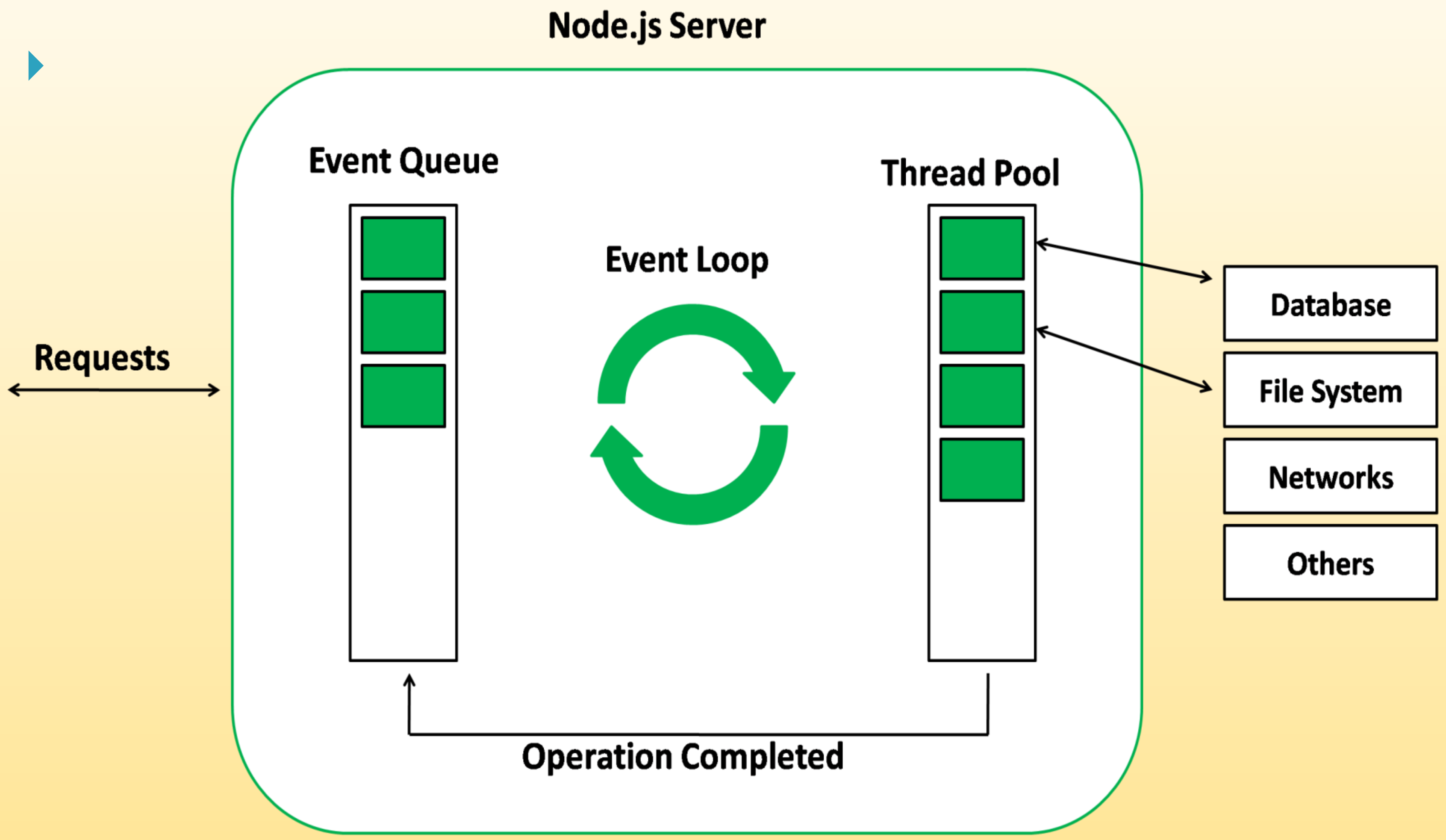


Event Loop



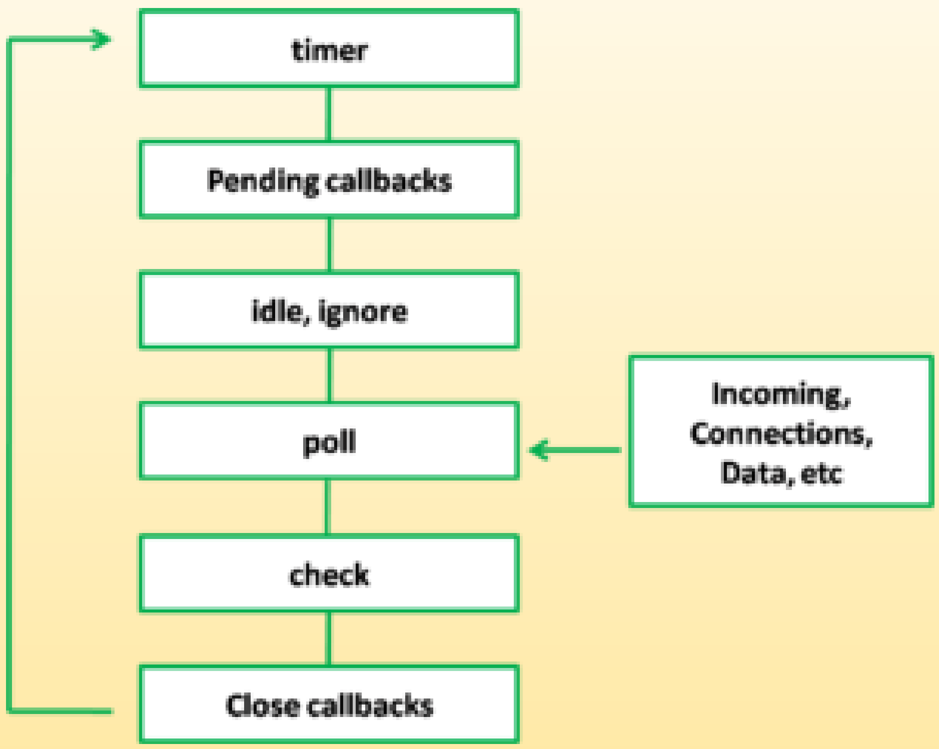
JS
heap
stack
Main.main()
qux()
baz()
bar()
foo()
WebAPIs
DOM (document)
ajax (XMLHttpRequest)
setTimeout
event loop
callback queue
onClick
onLoad
onDone

Working of the Event Loop



Phases of the Event Loop

- Each phase perform a specific task.



Simplified overview of event loop order

Event Loop

- As these tasks execute, they can add more tasks to the internal queues.
- They use **async function calls** to maintain concurrency. Node uses observer pattern.
- Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event(Event Emit) which signals the event-listener function to execute.

Event Emitter

- Many objects in a Node emit events.
  - For example, a Server emits an event each time a peer connects to it, an fs.readStream emits an event when the file is opened.
  - In the browser, an event could be a mouse click or key press.
- All objects which emit events are the instances of events.EventEmitter.
- Elsewhere in the code, subscribers can listen for these events and react to them when they occur.

Event Listening

- To set up an event listener in Node, use the on(), addListener(), and once() methods.
  - The on() and addListener() work in exactly the same manner: they create listeners for a specific type of event.
  - We prefer on() over addListener() as it requires less characters.
  - Once() only invoke call back function one time and stop the listener.
  - Error events can be handled like any other events. Following code shows how an error event is handled using on():

Example

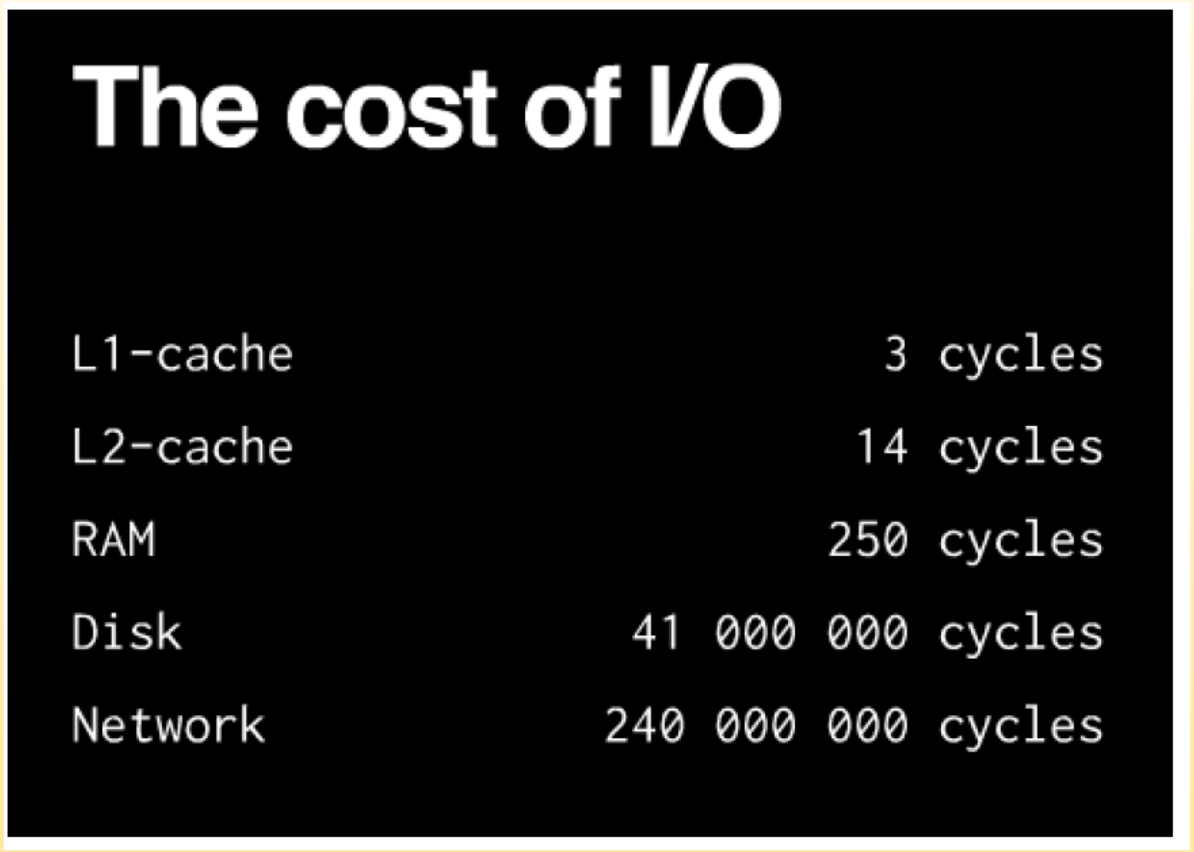
```
var http = require("http");
http.createServer(function (request, response) {
response.writeHead(200, {'Content-Type': 'text/plain'});
response.end('Hello World\n');
}).listen(8081);
console.log('Server running at http://127.0.0.1:8081/’);
```

- Here “listen” is a wrapper function of on() event listener

Threads VS Event-driven

Threads	Asynchronous Event-driven
Lock application / request with listener-workers threads	only one thread, which repeatedly fetches an event
Using incoming-request model	Using queue and then processes it
multithreaded server might block the request which might involve multiple events	manually saves state and then goes on to process the next event
Using context switching	no contention and no context switches
Using multithreading environments where listener and workers threads are used frequently to take an incoming-request lock	Using asynchronous I/O facilities (callbacks, not poll/select) environments

Cost of I/O



The cost of I/O

L1-cache	3 cycles
L2-cache	14 cycles
RAM	250 cycles
Disk	41 000 000 cycles
Network	240 000 000 cycles

Non-blocking I/O

- Servers do nothing but I/O
  - Scripts waiting on I/O requests degrades performance
- To avoid blocking, Node makes use of the event driven nature of JS by attaching callbacks to I/O requests
- Scripts waiting on I/O waste no space because they get popped off the stack when their non-I/O related code finishes executing

I/O Example

```
<?php
$result = mysql_query('SELECT * FROM ...');
while($r = mysql_fetch_array($result)){
    // Do something
}

// Wait for query processing to finish...
?>

<script type="text/javascript">
mysql.query('SELECT * FROM ...', function (err, result, fields){
        // Do something
    });

// Don't wait, just continue executing
</script>
```

Consistency

- Use of JS on both the client and server-side should remove need to “context switch”
  - Client-side JS makes heavy use of the DOM, no access to files/databases
  - Server-side JS deals mostly in files/databases, no DOM

Performance
Node.js VS Apache

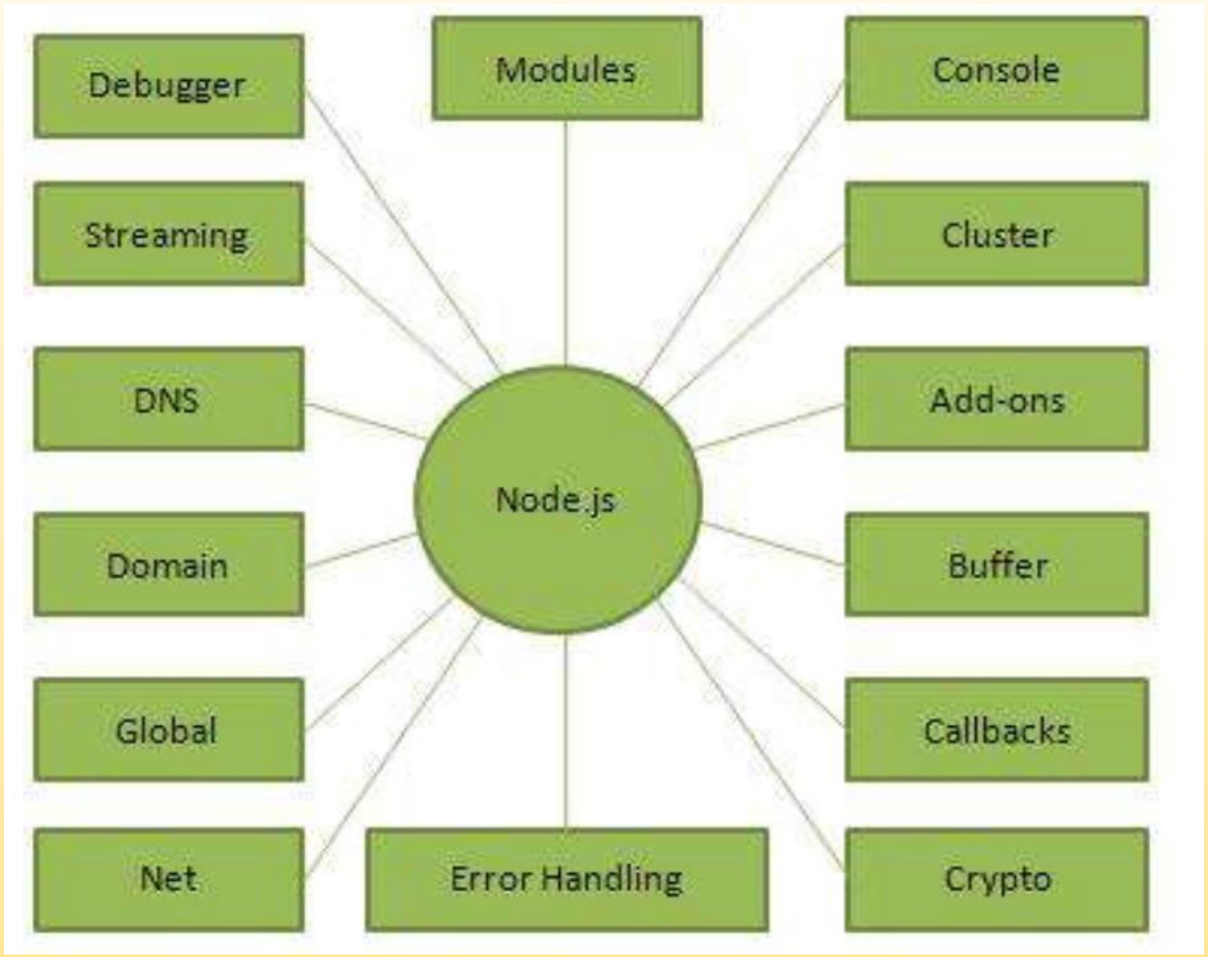
1. It's fast.(As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.)
2. It can handle tons of concurrent requests
3. It's written in JavaScript (which means you can use the same code server side and client side)

Platform	Number of request per second
PHP (via Apache)	3187,27
Static (via Apache)	2966,51
Node.js	5569,30

Node. JS

» Introduction to its modules

Node.JS major Components



Node Modules

- Import using require()
  - System module: require("fs"); // Looks in node_module directories
  - From a file: require("./XXX.js"); // Reads specified file
  - From a directory: require("./myModule"); // Reads myModule/index.js
- Module files have a private scope
  - Can declare variables that would be global in the browser
  - Require returns what is assigned to module.exports

```
var notGlobal;
function func1() {}
function func2() {}
module.exports = {func1: func1, func2: func2};
```

Node Modules

- Many standard Node modules
  - File system, process access, networking, timers, devices, crypto, etc.
- Huge library of modules (npm(Node Package Manager))
  - Do pretty much anything you want
- We use:
  - Express – Fast, unopinionated, minimalist web framework (speak HTTP)
  - Mongoose – Elegant mongodb object modeling (speak to the database)

Major Node Modules

- Buffer
- C/C++ Addons
- Child Processes
- Cluster
- Console
- Crypto
- Debugger
- DNS
- Errors
- Events
- File System
- Globals
- Timers

- TLS/SSL
- TTY
- UDP/Datagram
- URL
- Utilities
- V8
- VM
- ZLIB
- HTTP
- HTTPS
- Modules
- Net
- OS
- Path
- Process
- Punycode

- Query Strings
- Readline
- REPL
- Stream
- String Decoder

Example Node.JS reading a file

```
var fs = require("fs"); // require is a Node module call
// fs object wraps OS sync file system calls
// OS read() is synchronous but Node's fs.readFile is asynchronous
fs.readFile("smallFile", readDoneCallback); // Start read
function readDoneCallback(error, dataBuffer) {
// Node callback convention: First argument is JavaScript Error object
// dataBuffer is a special Node Buffer object
if (!error) {
console.log("smallFile contents", dataBuffer.toString());
}
}
```

Starting with Node.JS

- you need to have the following two software on your computer,
  - (a) Text Editor
  - (b) Node.js binary installables.
- The source files for Node.js programs are typically named with the extension ".js"

Installation On Windows

- Use the MSI file and follow the prompts to install Node.js.
  - By default, the installer uses the Node.js distribution in C:\Program Files\nodejs.
  - The installer should set the C:\Program Files\nodejs\bin directory in Window's PATH environment variable.
  - Restart any open command prompts for the change to take effect.
- Create a **js** file named main.js on your machine (Windows or Linux) having the following code.

```
/* Hello, World! program in node.js */
console.log("Hello, World!")
```

Installation verification

- Now execute main.js using Node.js interpreter to see the result:

```
$ node main.js
```

- If everything is fine with your installation, it should produce the following result:

```
Hello, World!
```

Steps to Create Application

- A Node.js application consists of the following three important components:
- 1. Import required modules: We use the require directive to load Node.js modules.
- 2. Create server: A server which will listen to client's requests similar to Apache HTTP Server.
- 3. Read request and return response: The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Step 1

- **Step 1 – Import Required Module**
- We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows:

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

Step 2

- **Step 2 – Create Server**
  - We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the listen method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

```
http.createServer(function (request, response) {
// Send the HTTP header & HTTP Status: 200 : OK
// Content Type: text/plain
response.writeHead(200, {'Content-Type': 'text/plain'});
// Send the response body as "Hello World"
response.end('Hello World\n');
}).listen(8081);
// Console will print the message
```

- console.log('Server running at http://127.0.0.1:8081/');

Step 3

```
var http = require("http");
http.createServer(function (request, response) {
// Send the HTTP header
// HTTP Status: 200 : OK
// Content Type: text/plain
response.writeHead(200, {'Content-Type': 'text/plain'});
// Send the response body as "Hello World"
response.end('Hello World\n');
}).listen(8081);
// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Step 3

- Now execute the main.js to start the server as follows:

```
$ node main.js
```

- Verify the Output. Server has started.

```
Server running at http://127.0.0.1:8081/
```

- **Make a Request to the Node.js Server**
- Open http://127.0.0.1:8081/ in any browser and observe the following result.
- Congratulations, you have your first HTTP server up and running which is responding to all the HTTP requests at port 8081.

NPM

- $ npm --version

```
2.7.1
```

- $ sudo npm install npm -g//update NPM

```
/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
npm@2.7.1 /usr/lib/node_modules/npm
```

- Installing Modules using NPM
  - There is a simple syntax to install any Node.js module:

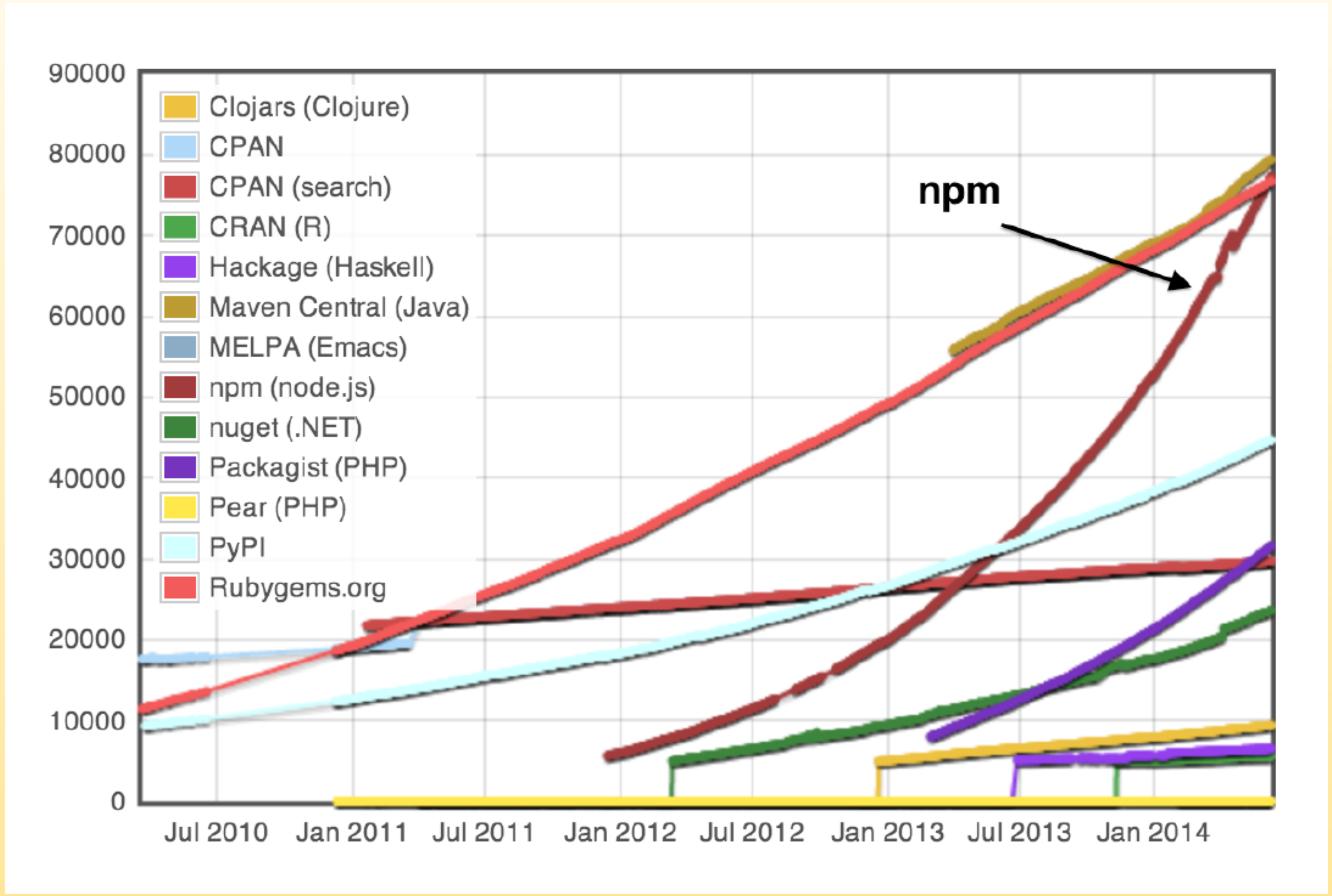
```
$ npm install <Module Name>
```

- For example, following is the command to install a famous Node.js web framework module called express:

```
$ npm install express
```

- Rate of approximately170 modules per day

Growth of NPM


Clojars (Clojure)
CPAN
CPAN (search)
CRAN (R)
Hackage (Haskell)
Maven Central (Java)
MELPA (Emacs)
npm (node.js)
nuget (.NET)
Packagist (PHP)
Pear (PHP)
PyPI
Rubygems.org
npm
Jul 2010
Jan 2011
Jul 2011
Jan 2012
Jul 2012
Jan 2013
Jul 2013
Jan 2014