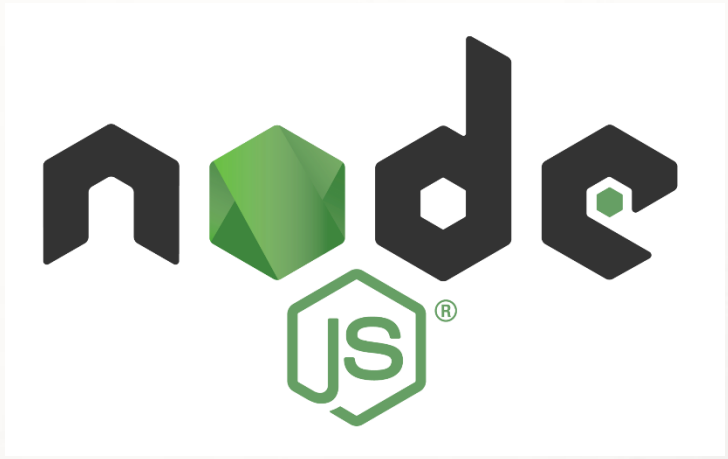
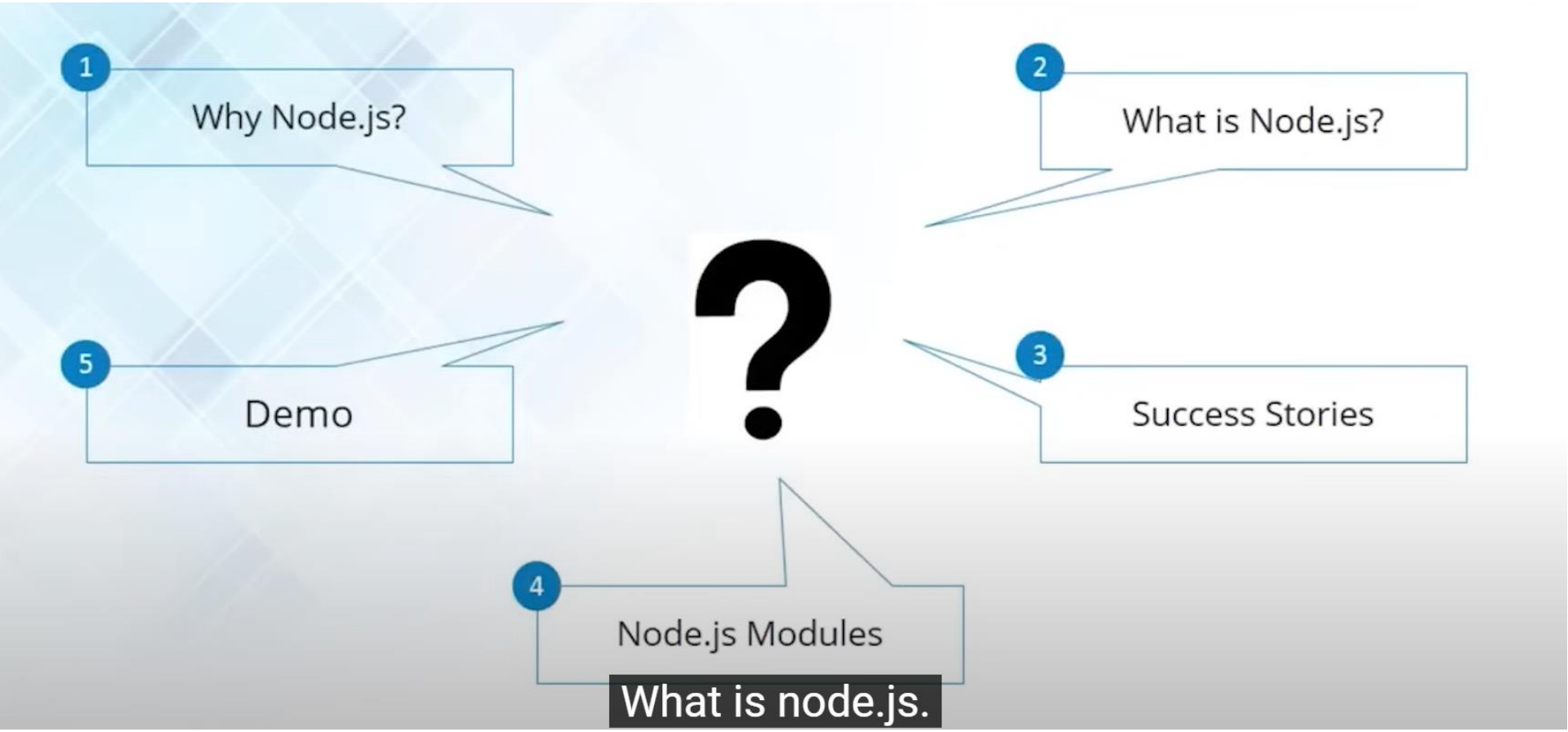
Application Delivery Fundamentals



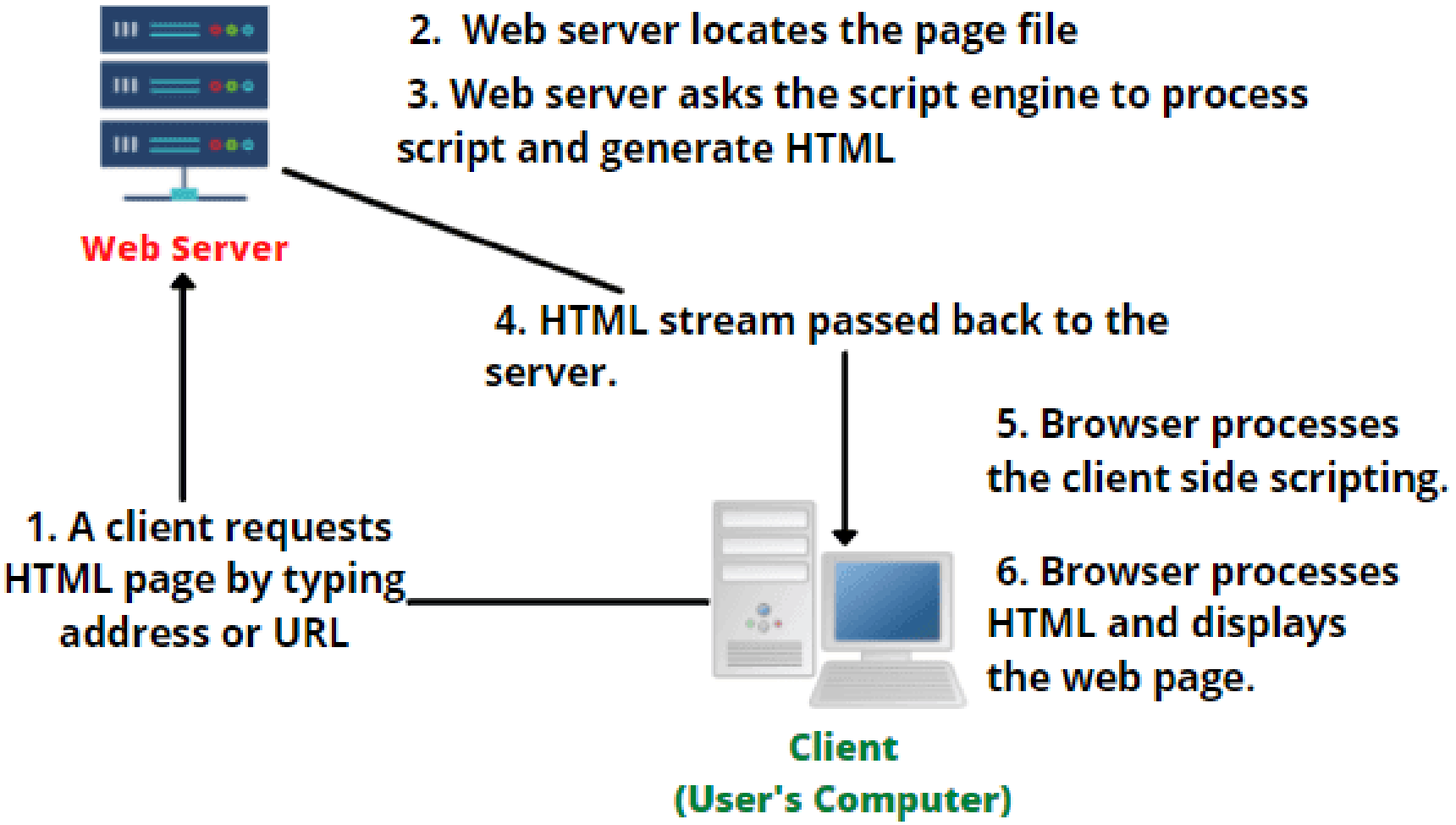
Parameswari Ettiappan

consulting | technology | outsourcing

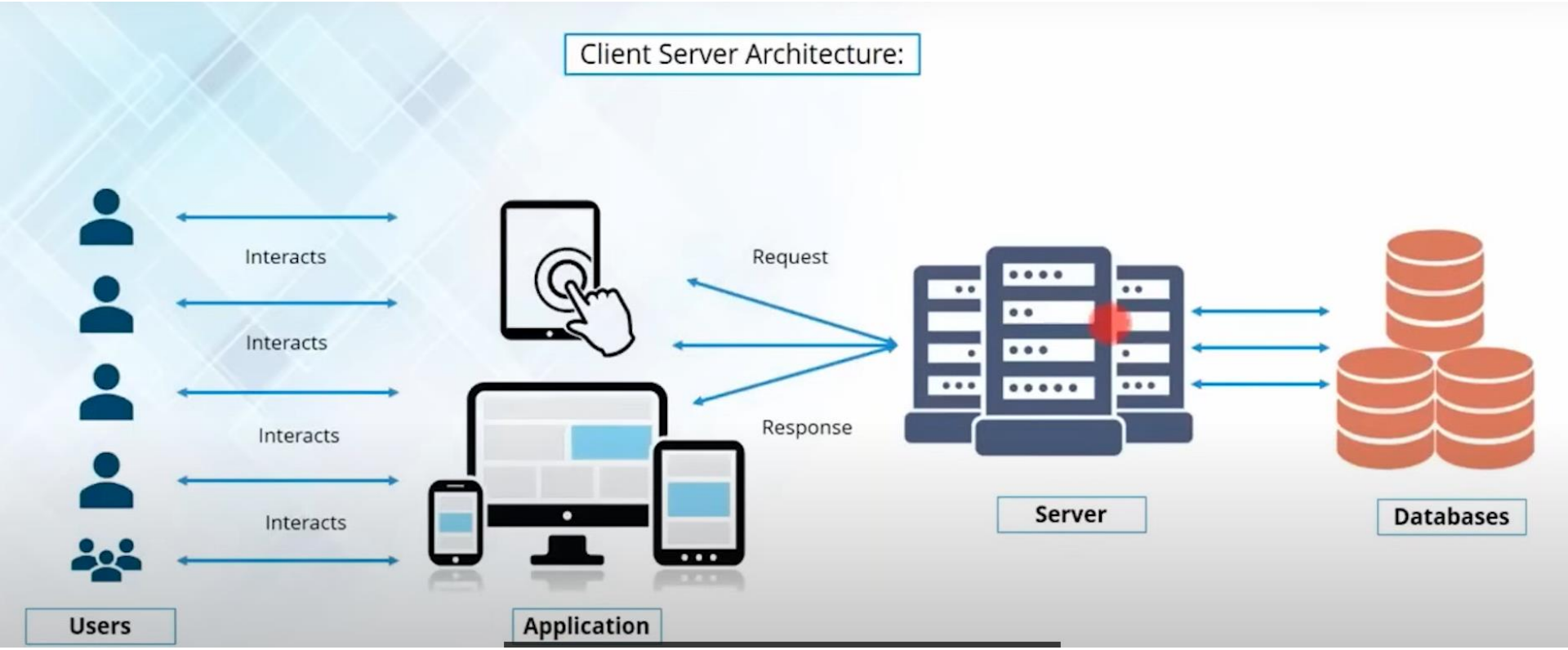
Goals


1
Why Node.js?
2
What is Node.js?
?
5
Demo
3
Success Stories
4
Node.js Modules
What is node.js.

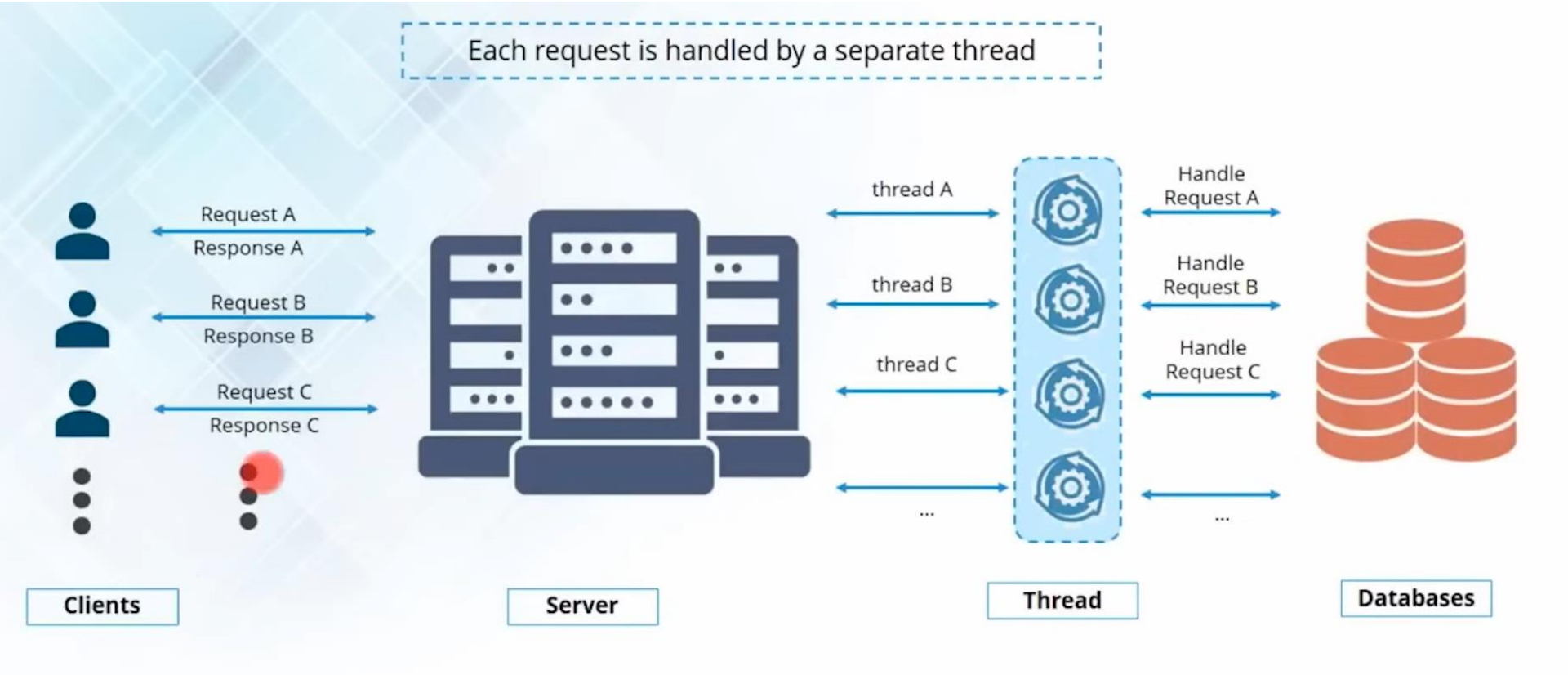
Client-side Java script



Client Server Architecture

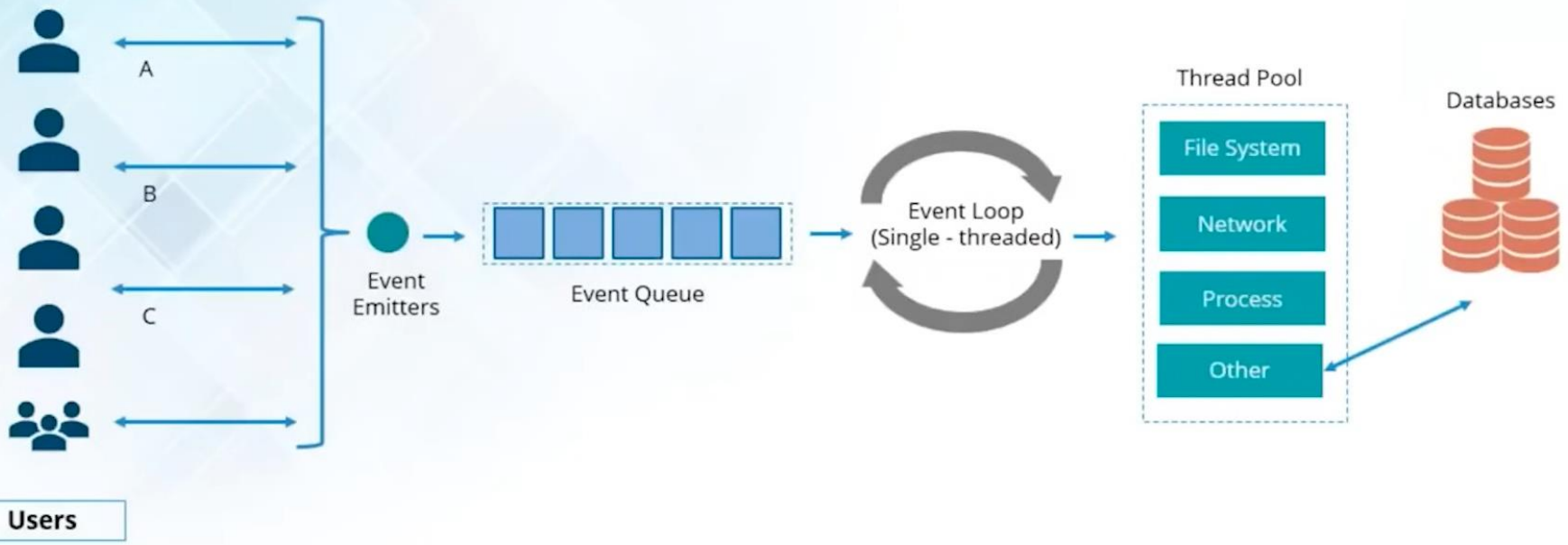


Multi Thread Model



Single Thread Model

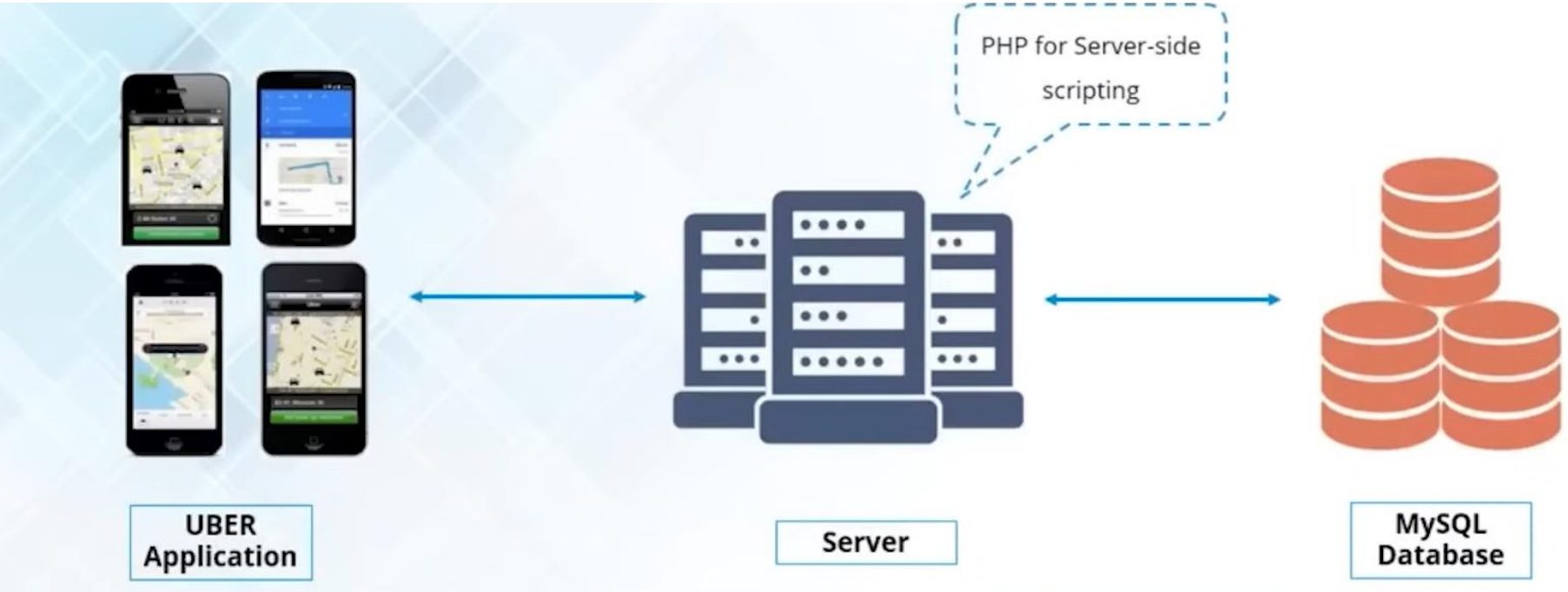
- Node.js is event driven, handling all requests asynchronously from single thread
- Almost no function in Node directly performs I/O, so the process never blocks



Multi Thread vs Single/Event Thread Model

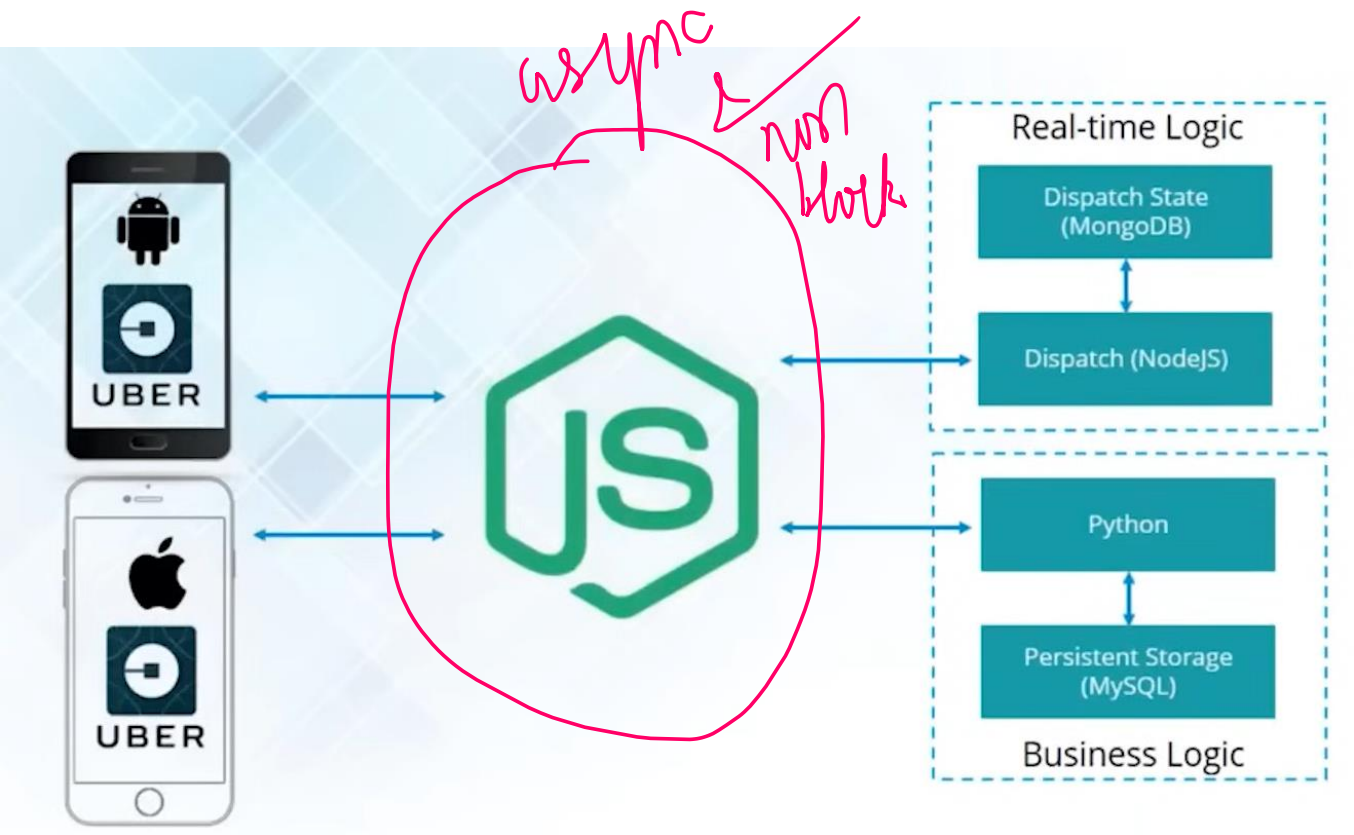
Multi-Threaded	Asynchronous Event-driven
Lock application / request with listener-workers threads	Only one thread, which repeatedly fetches an event
Using incoming-request model	Using queue and then processes it
Multithreaded server might block the request which might involve multiple events	Manually 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 or O_NONBLOCK) environments

Uber Old Architecture



- Since PHP is a multithreaded language , each user's request is handled in a separate thread
- Reason was car dispatch operation was executed from multiple threads
- Once one car is dispatched for a user, in between the same car get dispatched to another user

Uber New Architecture



- Well-suited for distributed systems that make a lot of network requests
- Errors can be addressed on the fly without requiring a restart
- Active open source community

Success Stories

Nexflix used JavaScript and NodeJS to transform their website into a single page application.

NETFLIX

PayPal team developed the application simultaneously using Java and Javascript. The JavaScript team build the product both faster and more efficiently.

PayPal

Uber has built its massive driver / rider matching system on Node.js Distributed Web Architecture.

UBER

Node enables to build quality applications, deploy new features, write unit and integration tests easily.

GoDaddy.com

When LinkedIn went to rebuild their Mobile application they used Node.js for their Mobile application server which acts as a REST endpoint for Mobile devices.

LinkedIn

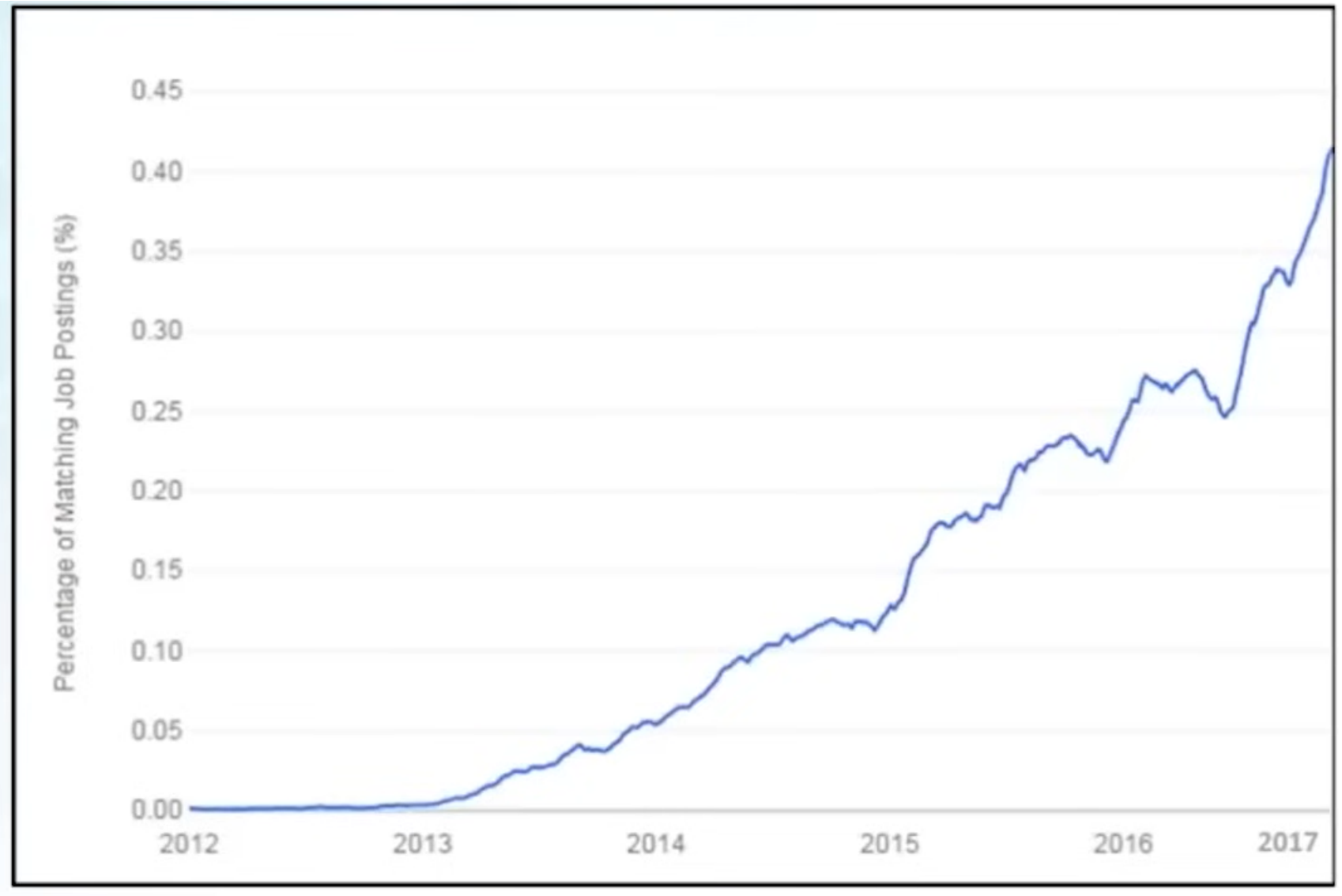
They had two primary requirements: first to make the application as real time as possible. Second was to orchestrate a huge number of eBay-specific services.

ebay

What is Nodejs

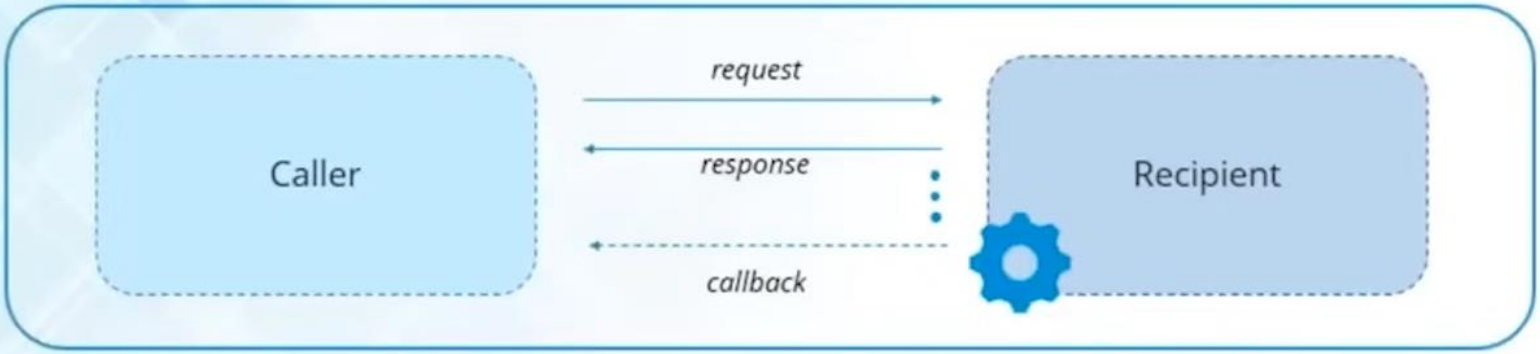
- Node.js is an open source runtime environment for server-side and networking applications and is single threaded.
- Uses Google JavaScript V8 Engine to execute code.
- It is cross platform environment and can run on OS X, Microsoft Windows, Linux and FreeBSD.
- Provides an event driven architecture and non blocking I/O that is optimized and scalable.

Nodejs Job Trends



Nodejs Features

Asynchronous and Event Driven :

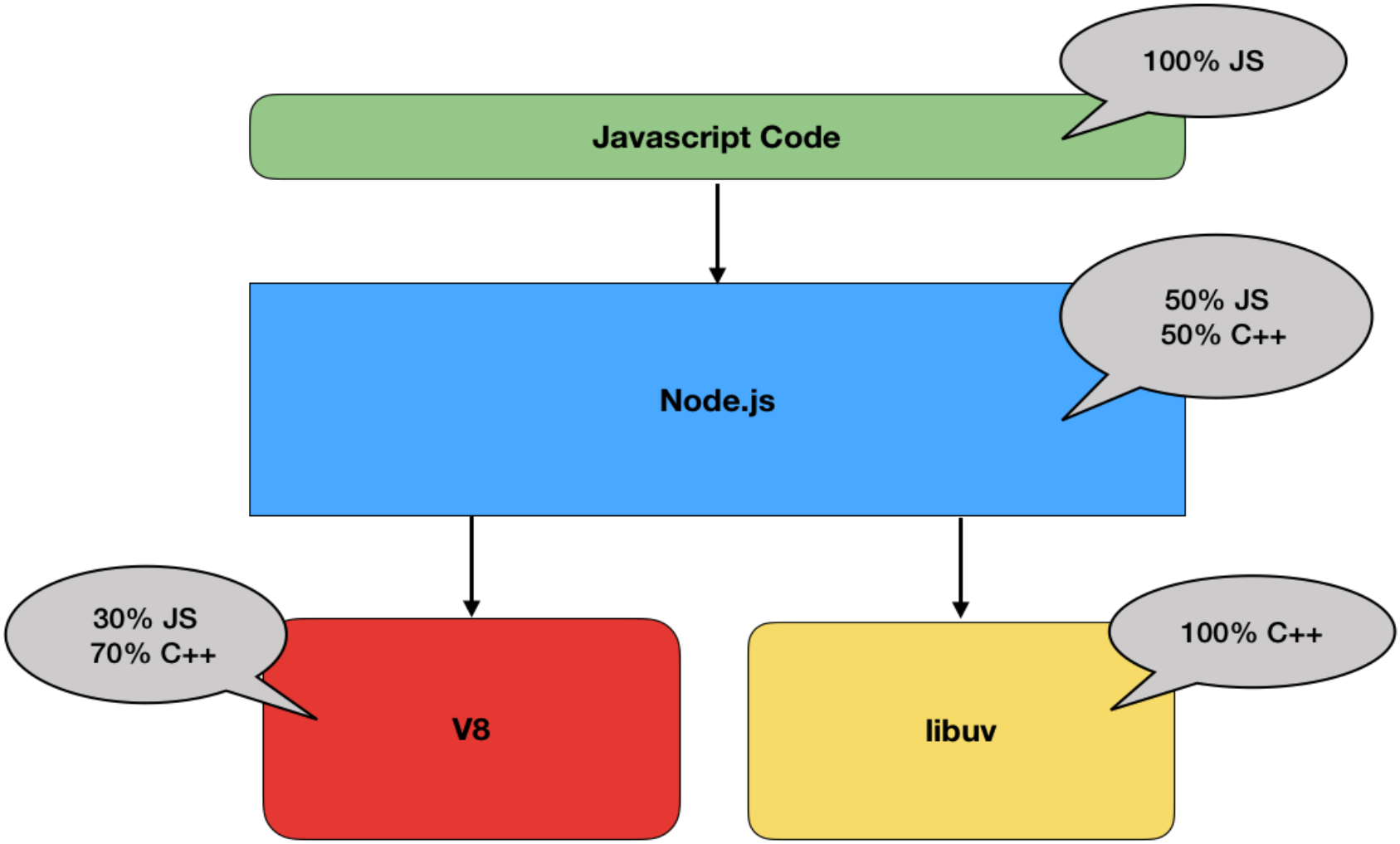


- When request is made to server, instead of waiting for the request to complete, server continues to process other requests
- When request processing completes, the response is sent to caller using callback mechanism

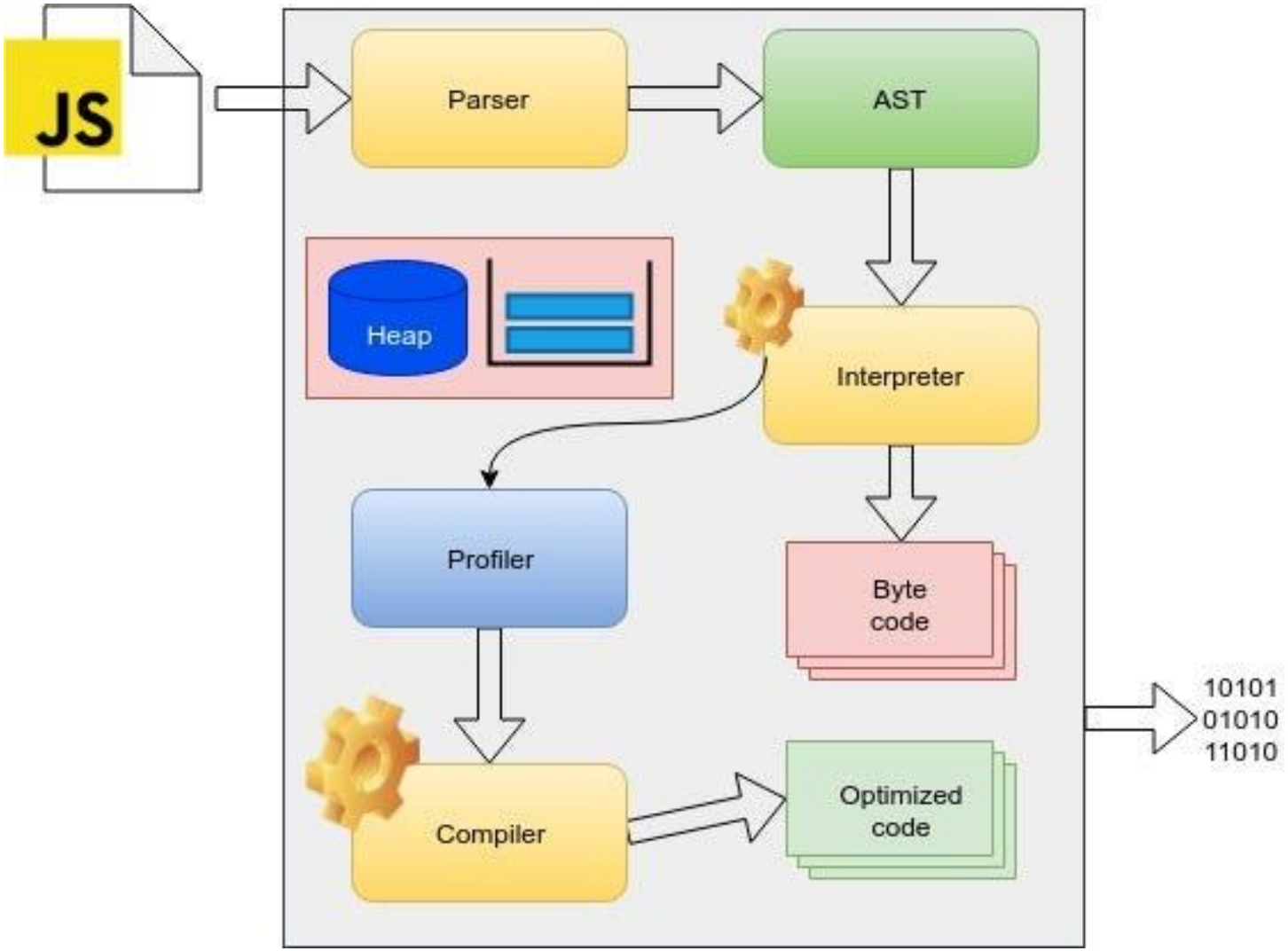
Node js Introduction

- js file (source code) → compiled into machine code (based on the CPU it runs)
- The system that compiles the java script files into machine code and manage the program's memory is called java script runtime.
- Chrome V8 engine originally ran in Chrome web browsers.
- But later it was used to run the java script without a browser and that exactly what Node.js is.
- Node.js is written in C++ and build on top of the Chrome V8 engine which is also written in C++. V8 provides the runtime for the Java script.

Node js Introduction



Node js Introduction

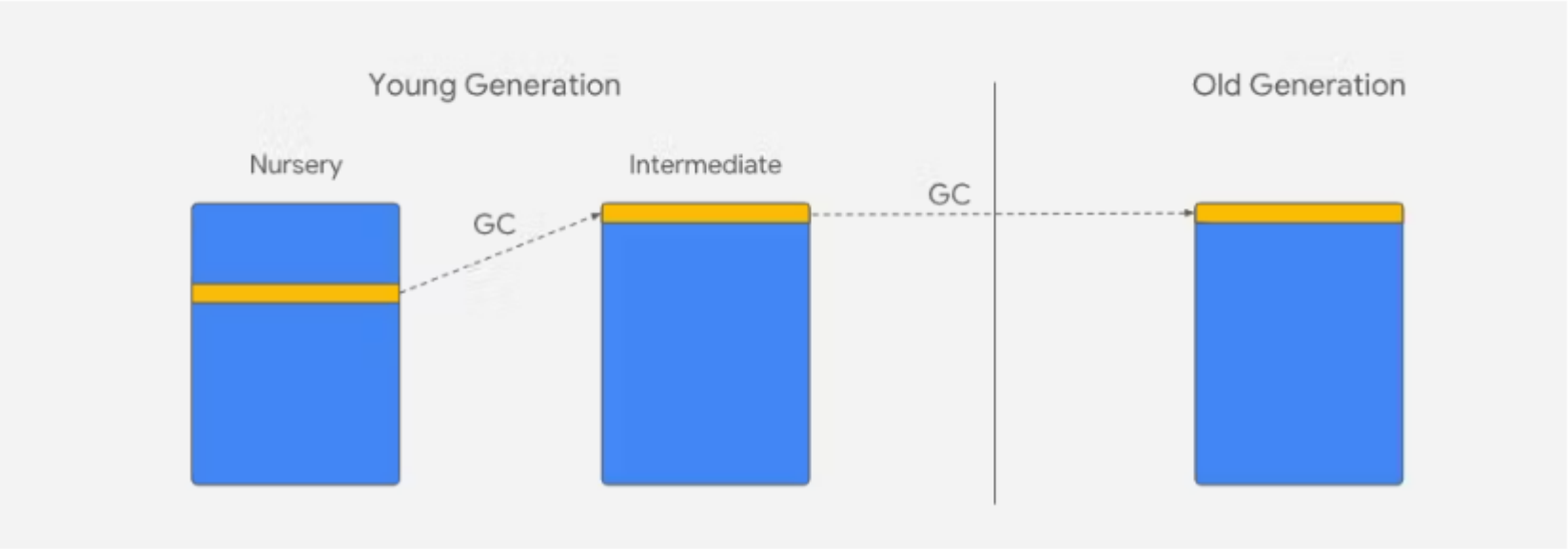


Node js Byte Code

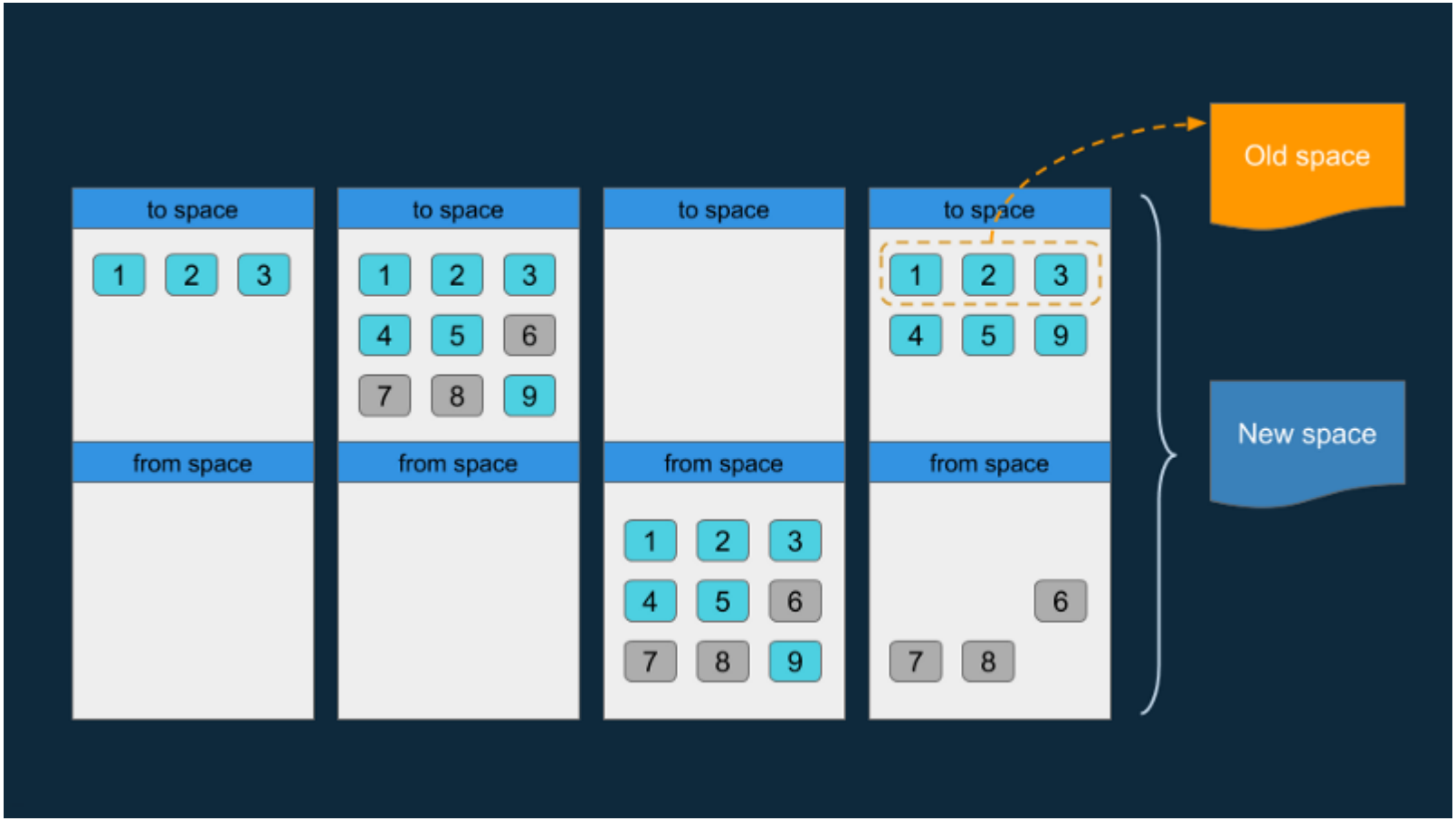
Administrator: Node.js command prompt

```
I:\Nodejs_ANZ\FileHandling>node --print-bytecode ReadFile.js
[generated bytecode for function:  (0x0085c7739aa9 <SharedFunctionInfo>)]
Bytecode length: 5
Parameter count 1
Register count 0
Frame size 0
Bytecode age: 0
    0 E> 00000085C7739BEE @    0 : 80 00 00 00       CreateClosure [0], [0], #0
 1211 S> 00000085C7739BF2 @    4 : a9                Return
Constant pool (size = 1)
00000085C7739BA1: [FixedArray] in OldSpace
 - map: 0x023d36e80211 <Map(FIXED_ARRAY_TYPE)>
 - length: 1
           0: 0x0085c7739b69 <SharedFunctionInfo>
Handler Table (size = 0)
Source Position Table (size = 7)
0x0085c7739bf9 <ByteArray[7]>
[generated bytecode for function:  (0x0085c7739b69 <SharedFunctionInfo>)]
Bytecode length: 120
Parameter count 5
```

Node js GC

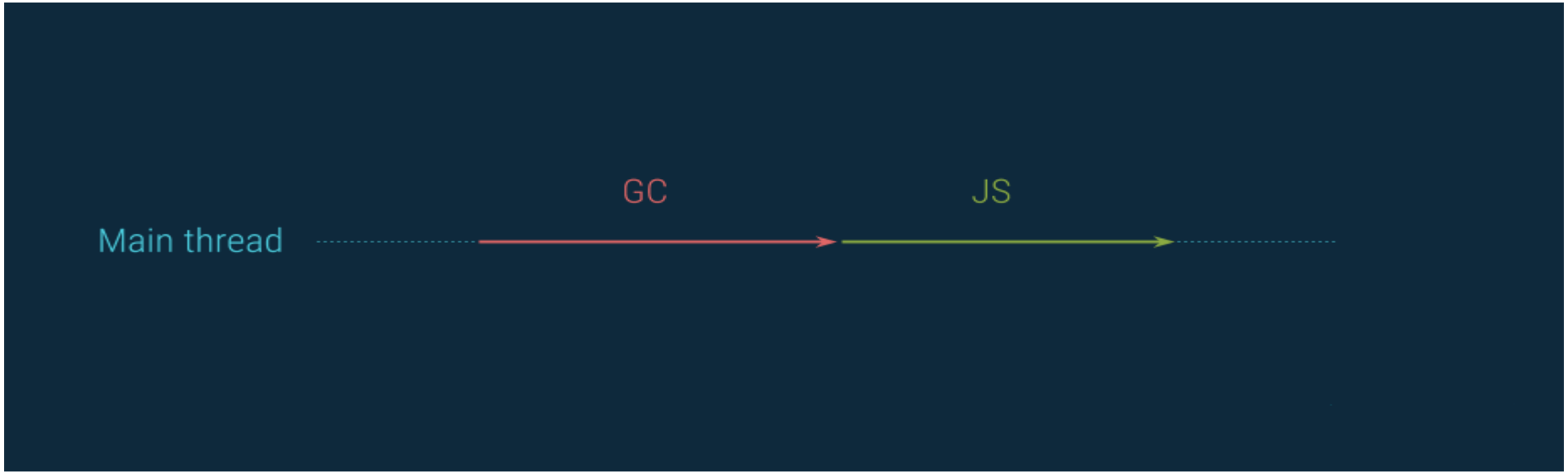

Young Generation
Old Generation
Nursery
Intermediate
GC
GC

Node js GC

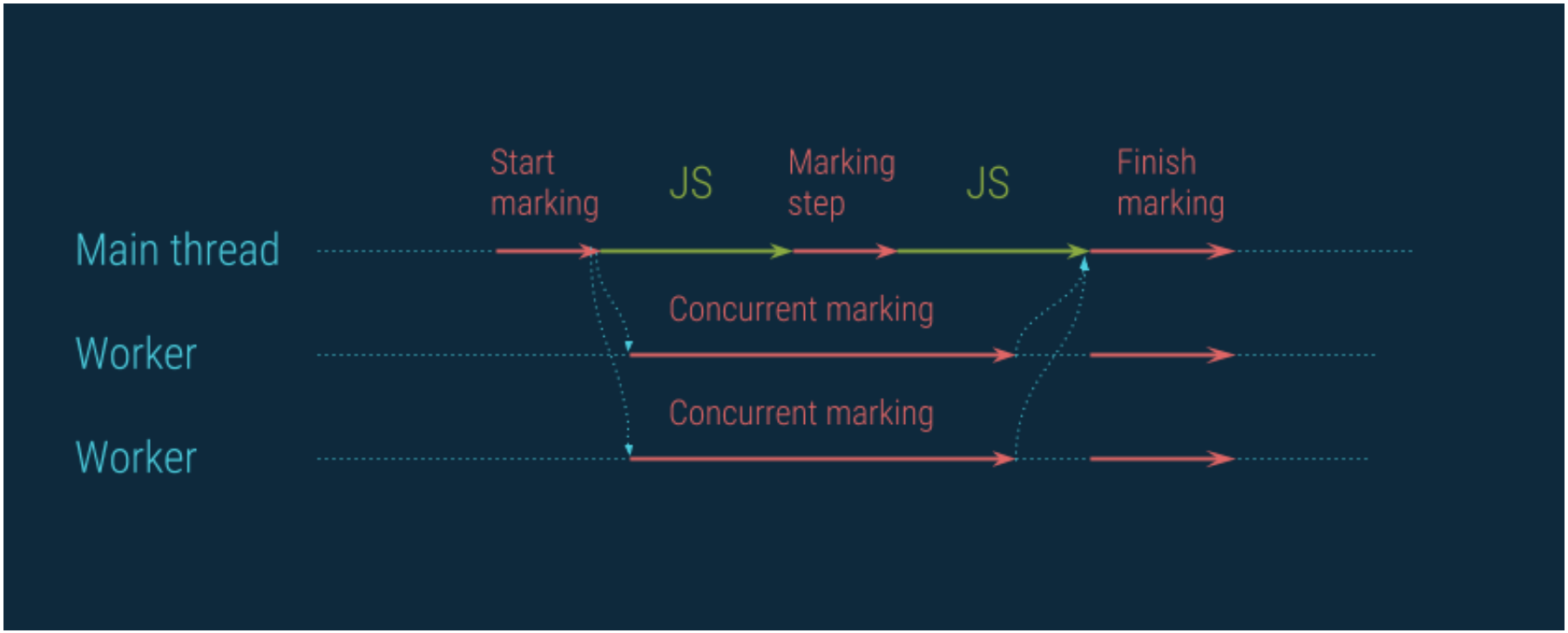

to space
1
2
3
from space
to space
1
2
3
4
5
6
7
8
9
from space
to space
from space
1
2
3
4
5
6
7
8
9
to space
1
2
3
4
5
9
from space
6
7
8
Old space
New space

What's new in Onirocco?

- Cleaning and compacting memory are not the only improvements and optimizations that V8 can do for you.
- The initial mechanism behind GC cycles was the following:
  - The program is stopped so GC can do its job "stop the world" syndrome.



What's new in Onirocco?



What's new in Onirocco?

Code

```
var gc = (require('gc-stats'))();

gc.on('stats', function (stats) {
    console.log('GC happened', stats);
});
```

Output

```
GC happened {
  startTime: 9426055813976,
  endTime: 9426057735390,
  pause: 1921414,
  pauseMS: 1,
  gctype: 1,
  before: {
    ...
  },
  after: {
    ...
  },
  diff: {
    ...
  }
}
```

What's new in Onirocco?

Administrator: Node.js command prompt

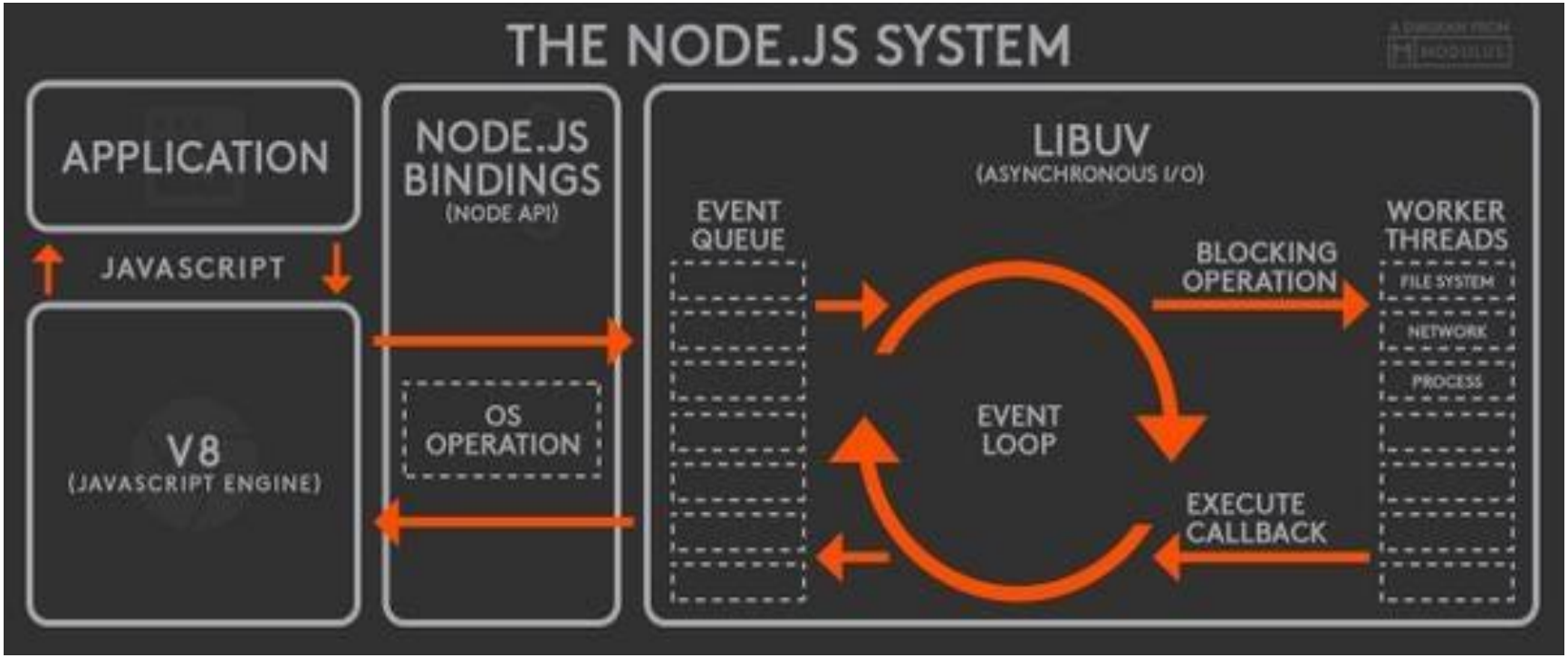
```
I:\Nodejs_ANZ\FileHandling>node --trace_gc ReadFile.js
Going to get file info!
Stats {
  dev: 1447159954,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: 4096,
  ino: 281474978402566,
  size: 246,
  blocks: 0,
  atimeMs: 1714291419989.1245,
  mtimeMs: 1503544098633.244,
  ctimeMs: 1503544153236.7322,
  birthtimeMs: 1514637065222.98,
  atime: 2024-04-28T08:03:39.989Z,
  mtime: 2017-08-24T03:08:18.633Z,
  ctime: 2017-08-24T03:09:13.237Z,
  birthtime: 2017-12-30T12:31:05.223Z
```

Node js GC

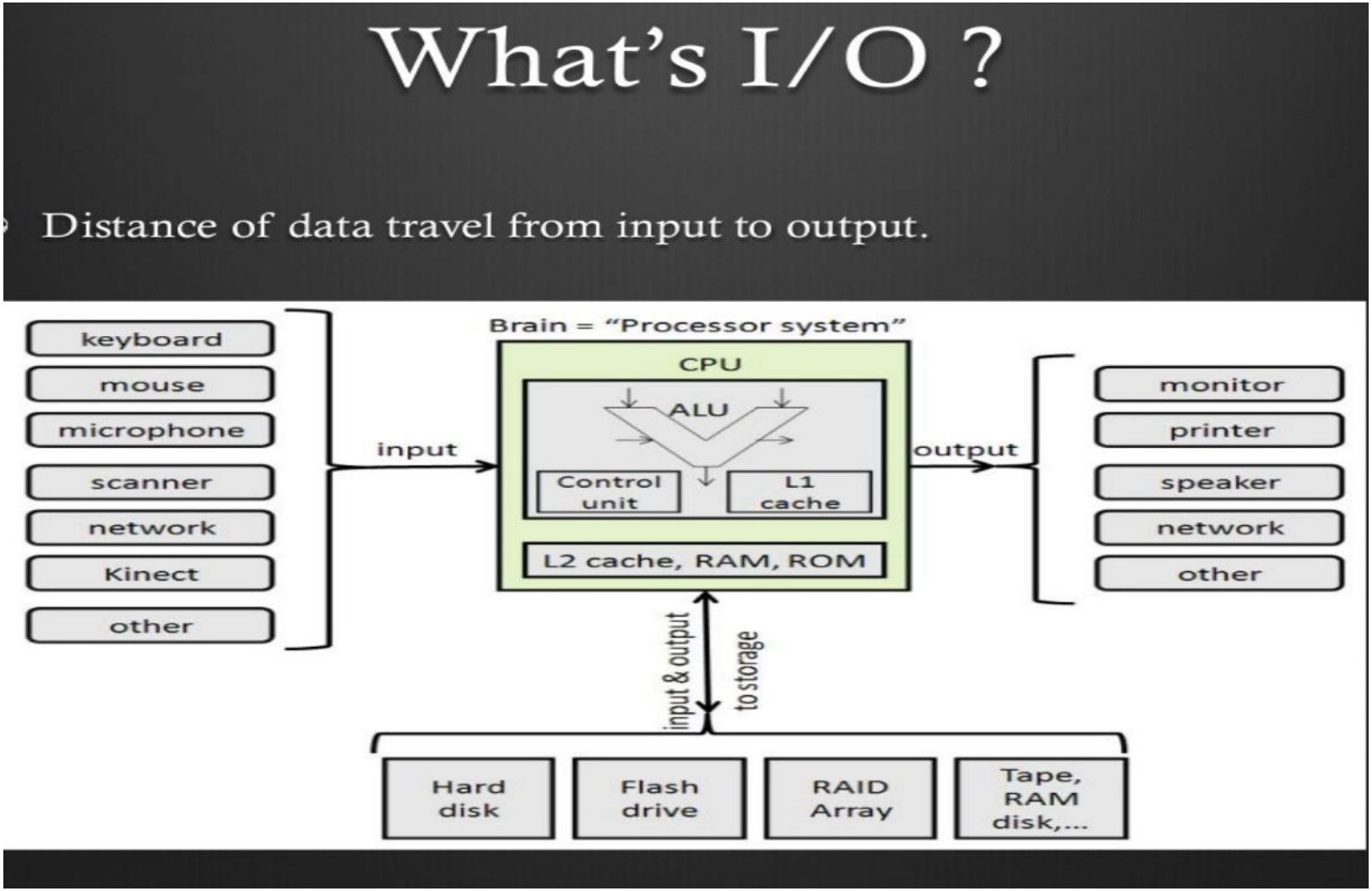
- node index.js --max-old-space-size=8000
- Or

```
{
  //other package.json stuff

  "scripts":{
     "start": "node --max-old-space-size=4076 server.js"
  }

}
```

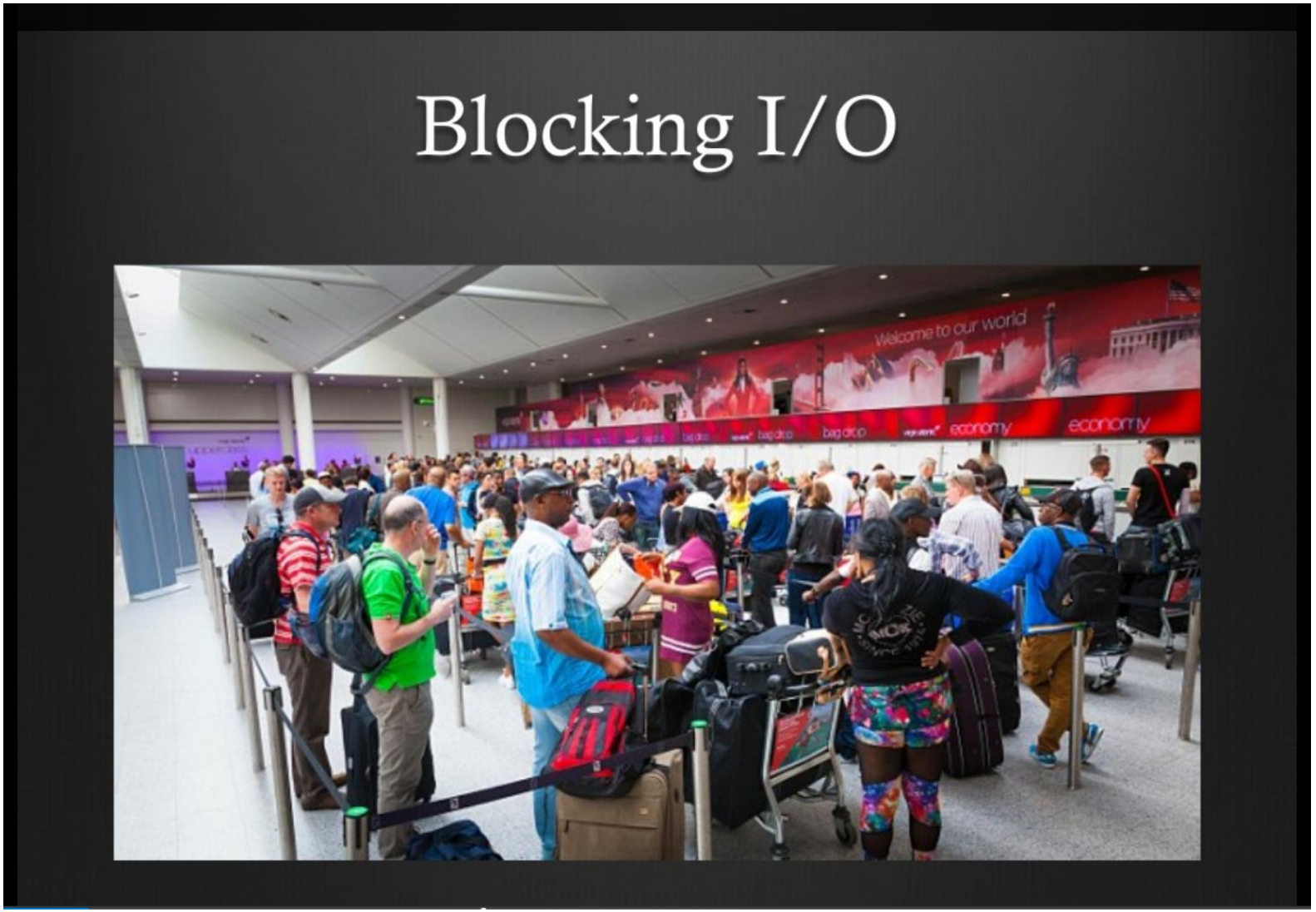
Nodejs System



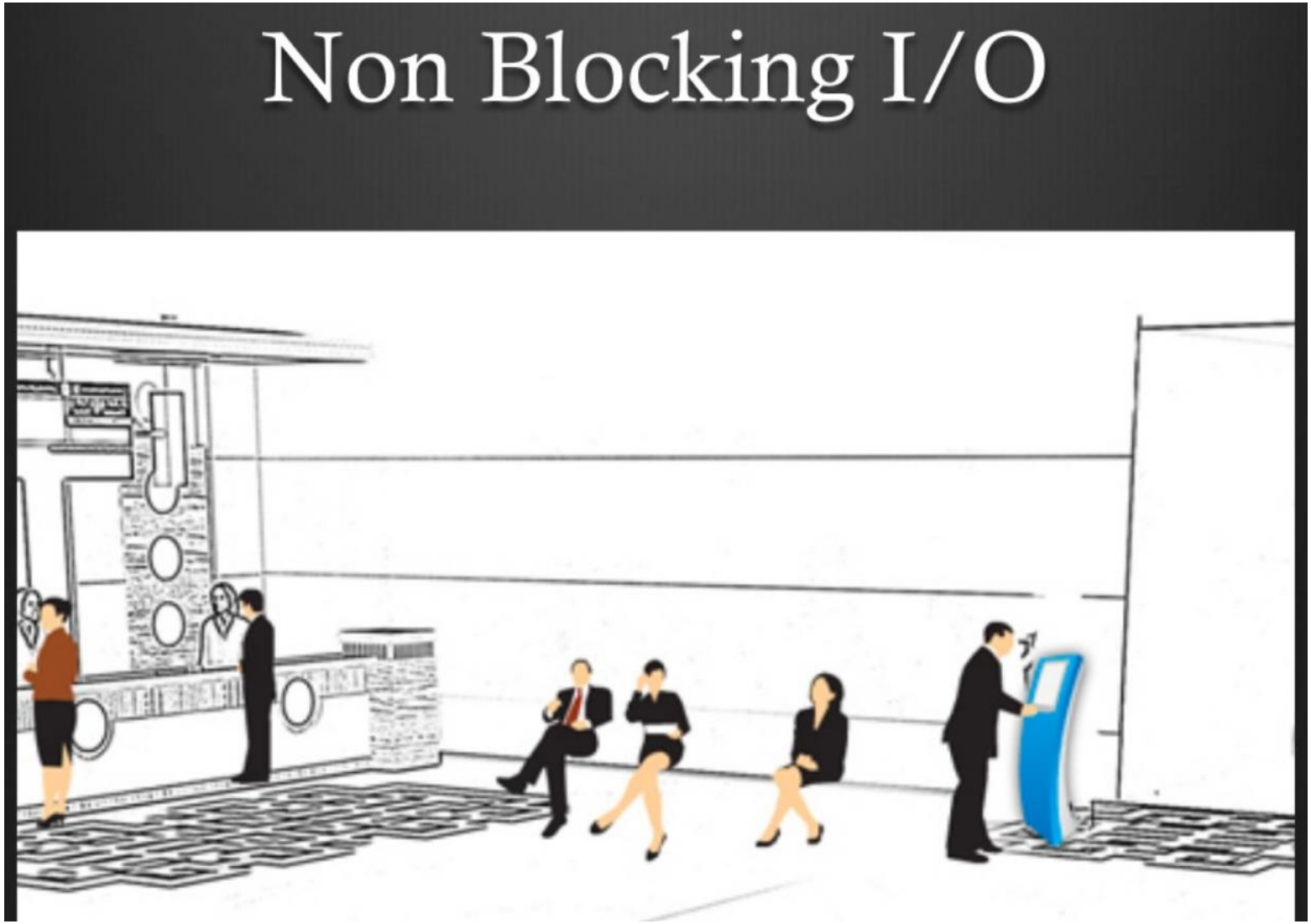
Nodejs Introduction



Nodejs Introduction



Nodejs Introduction



Blocking and Non Blocking

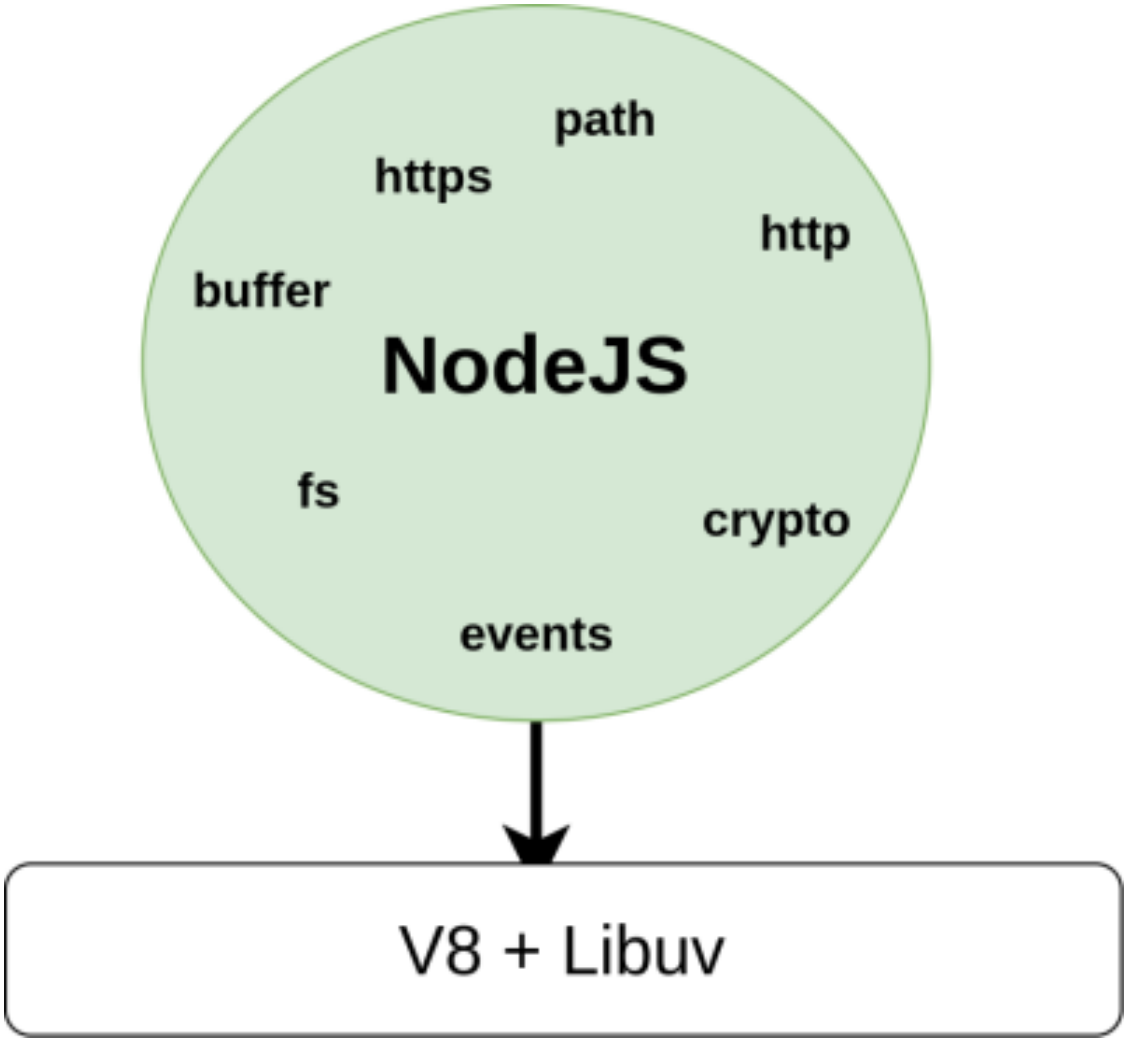
How is Java script compiled and Run?

- Java script Program (source code) → Chome V8 Engine (compiles) → Machine Code (X86_64, MIPS, ARM, etc) → Runs on CPU.
- Different CPU architecture requires different machine code based on its architecture.
- So, the program needs to be converted into the machine code that it can understand.
- Java script program is compiled, and the bytecode (machine code) is optimized and finally, the machine code runs on the CPU.
- All these operations are performed by the V8.

What is Node.js?

- Node.js is an open-source server-side runtime environment built on Chrome's V8 JavaScript engine.
- It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side application using JavaScript.
- Node.js can be used to build different types of applications such as command line application, web application, real-time chat application, REST API server etc.

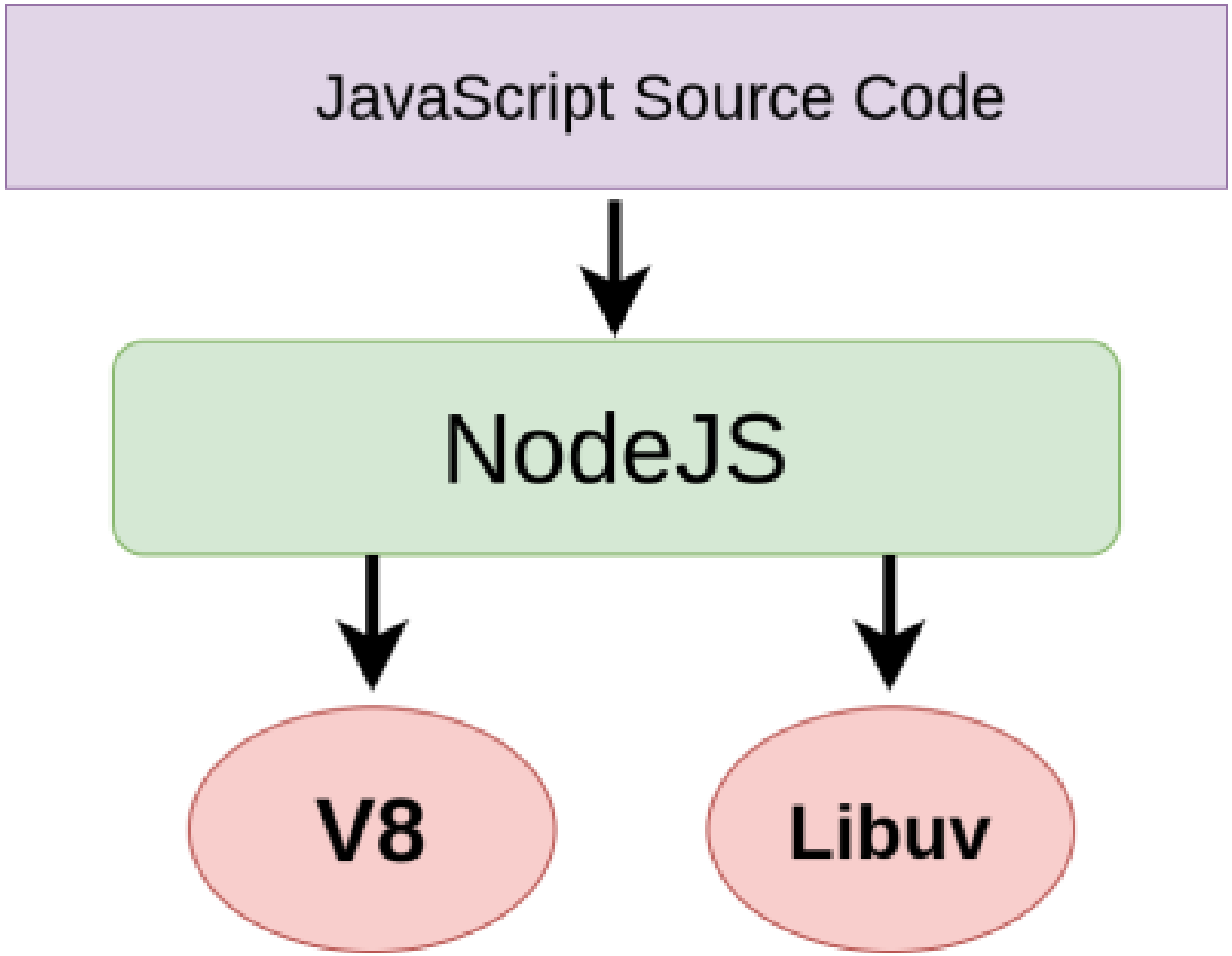
Node.js API support



What is Node.js?

- It is mainly used to build network programs like web servers, like PHP, Java, or ASP.NET.
- Node.js was written and introduced by Ryan Dahl in 2009.

Main Building Blocks



Main Building Blocks

- V8 — a Google's opensource high-performance JavaScript engine, written in C++.
- It is also used in Google Chrome browser and others. Node.js controls V8 via V8 C++ API.
- libuv — a multi-platform support library with a focus on asynchronous I/O, written in C.
- It was primarily developed for use by Node.js, but it's also used by Luvit, t, Julia, pyuv, and others.

Main Building Blocks

- . Node.js uses libuv to abstract non-blocking I/O operations to a unified interface across all supported platforms.
- This library provides mechanisms to handle file system, DNS, network, child processes, pipes, signal handling, polling and streaming.
- It also includes a thread pool, also known as Worker Pool, for offloading work for some things that cannot be done asynchronously at the OS level.

Main Building Blocks

- Other open-source, low-level components, mostly written in C/C++:
- - c-ares — a C library for asynchronous DNS requests, which is used for some DNS requests in Node.js.
- - http-parser — a lightweight HTTP request/response parser library.
- - OpenSSL — a well-known general-purpose cryptography library. Used in tls and crypto modules.
- - zlib — a lossless data-compression library. Used in zlib module.

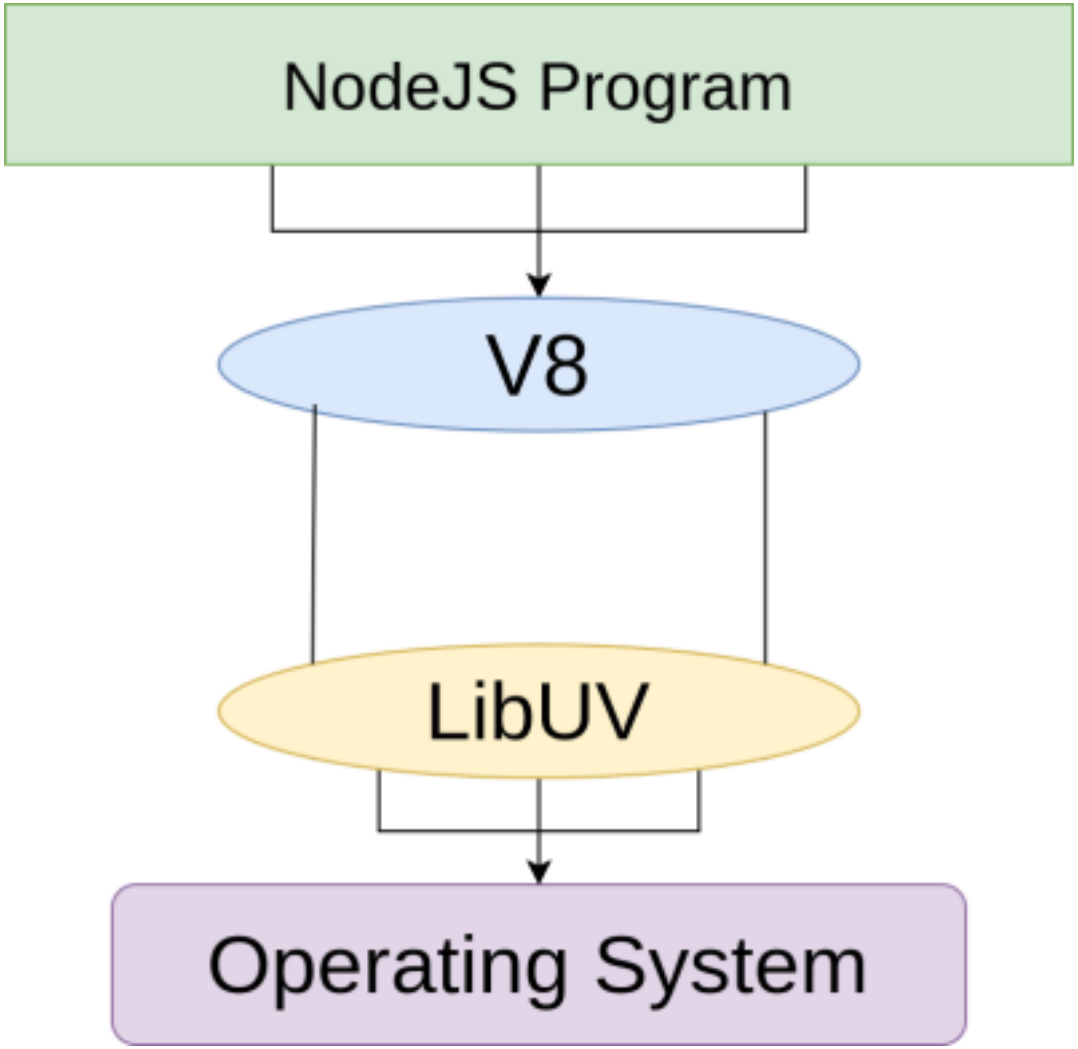
Main Building Blocks

- The application — it's your application's code and standard Node.js modules, written in JavaScript.
- C/C++ bindings — wrappers around C/C++ libraries, built with N-API, a C API for building native Node.js addons, or other APIs for bindings..

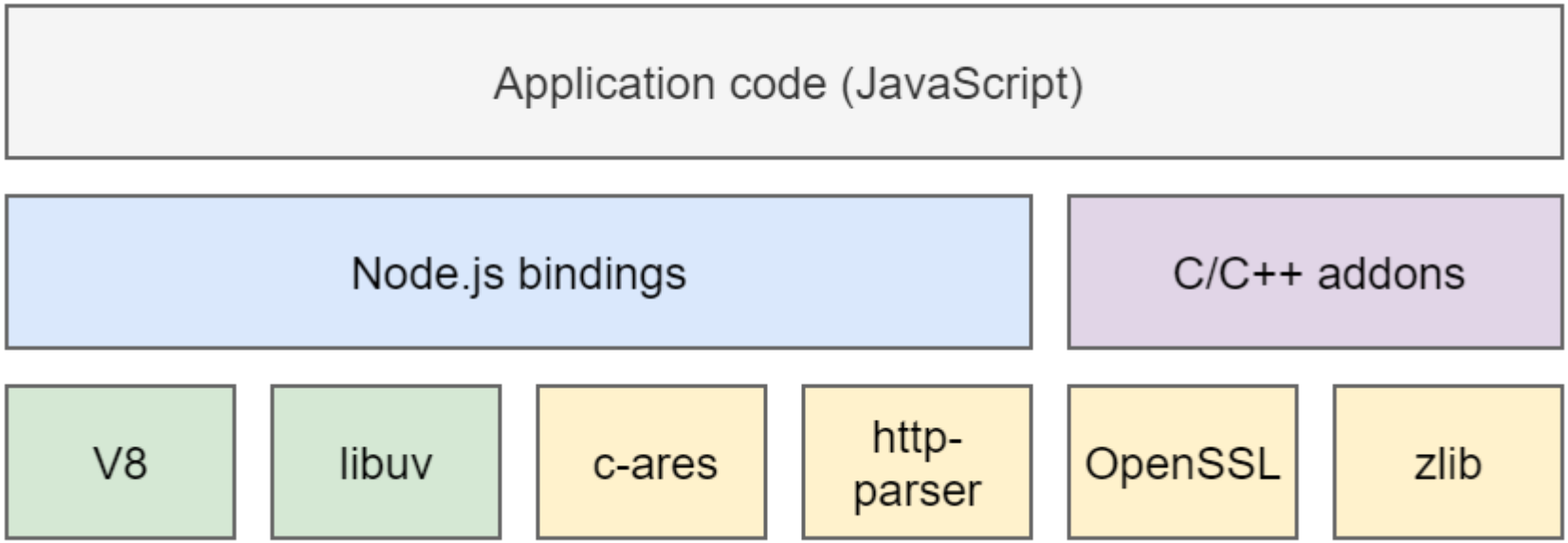
Main Building Blocks

- Some bundled tools that are used in Node.js infrastructure:
- - npm — a well-known package manager (and ecosystem).
- - gyp — a python-based project generator copied from V8. Used by node-gyp, a cross-platform command-line tool written in Node.js for compiling native addon modules.
- - gtest — Google’s C++ test framework. Used for testing native code.

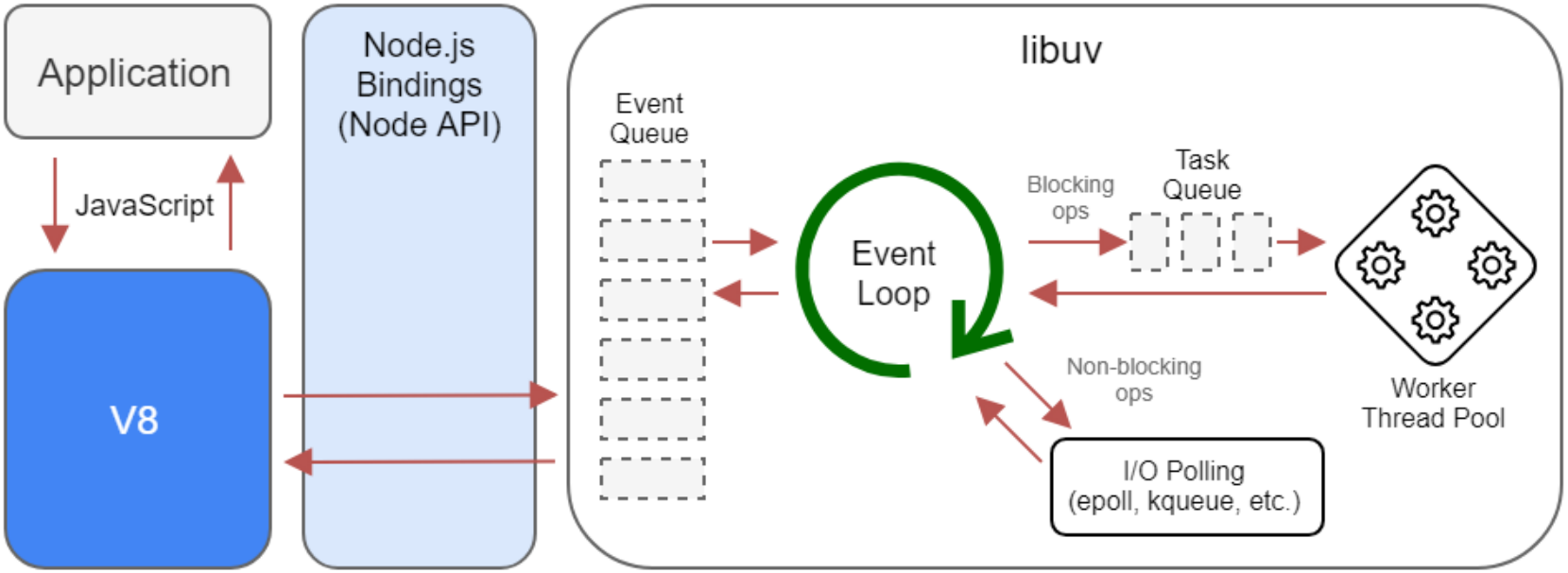
Nodejs Low Level



Main Node.js components



Node.js Runtime



Introduction to Google V8

- V8 is Google's open source high-performance JavaScript and WebAssembly engine, written in C++
- V8 Does:
  - Compiles and executes JS code
  - Handling call stack—running your JS functions in some order
  - Managing memory allocation for objects—the memory heap
  - Garbage collection—of objects which are no longer in use
  - Provide all the data types, operators, objects and functions

Introduction to Google V8

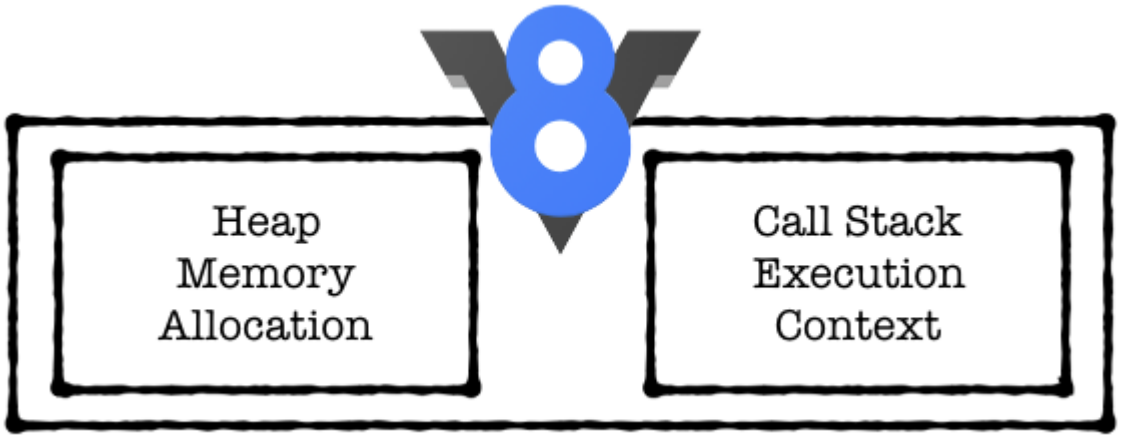
- V8 Can:
  - Provide the event loop, but this is sometimes implemented by the browser as well
- V8 Doesn't:
  - Know anything about the Document Object Model (DOM)—which is provided by the browser, and obviously irrelevant to Node.js for example

Introduction to Google V8

- V8 is a single threaded execution engine.
- It’s built to run exactly one thread per JavaScript execution context.
- You can actually run two V8 engines in the same process—e.g. web-workers, but they won’t share any variables or context like real threads.
- On the runtime, V8 is mainly managing the heap memory allocation and the single threaded call stack.
- The call stack is mainly a list of function to execute, by calling order.

Introduction to Google V8

- Every function which calls another function will be inserted one after the other directly, and callbacks will be sent to the end.
- This is actually why calling a function with setTimeout of zero milliseconds sends it to the end of the current line and doesn't call it straight away (0 milliseconds).



V8 Key Components

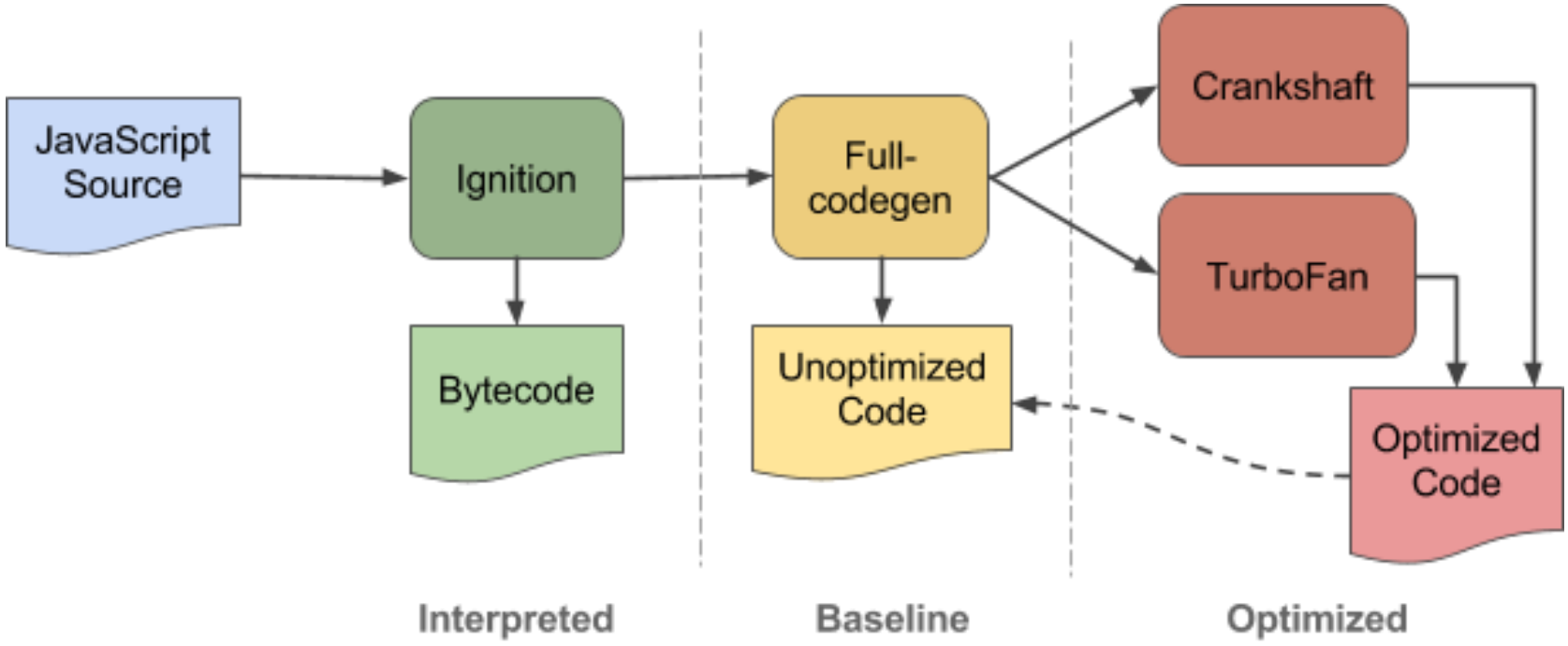
- JS Interpreter—Ignition & Optimization Compiler—TurboFan & Crankshaft.

- V8 gets its speed from just-in-time (JIT) compilation of JavaScript to native machine code, just before executing it.
- First of all, the code is compiled by a baseline compiler, which quickly generates non-optimized machine code.
- On runtime, the compiled code is analyzed and can be re-compiled for optimal performance.
- Ignition provides the first while TruboFan & Crankshaft the second.

V8 Key Components

- JIT compilation result machine code can take a large amount of memory, while it might be executed once.
- This is solved by the Ignition, which is executing code with less memory overhead.
- The TurboFan project started in 2013 to improve the weakness of Crankshaft which isn't optimized for some part of the JavaScript functionality e.g. error handling.
- It was designed for optimizing both existing and future planned features at the time.

V8 Key Components



WebAssembly—Liftoff

- Achieving great performance is also key in the browser, and this is the task Liftoff is used for—generating machine code.
- Not using the complex multi-tier compilation, Liftoff is a simpler code generator, which generates code for each opcode (a single portion of machine code, specifying an operation to be performed) at a time.
- Liftoff generates code much faster than TurboFan (~10x) which is obviously less performant (~50%).

WebAssembly—Liftoff

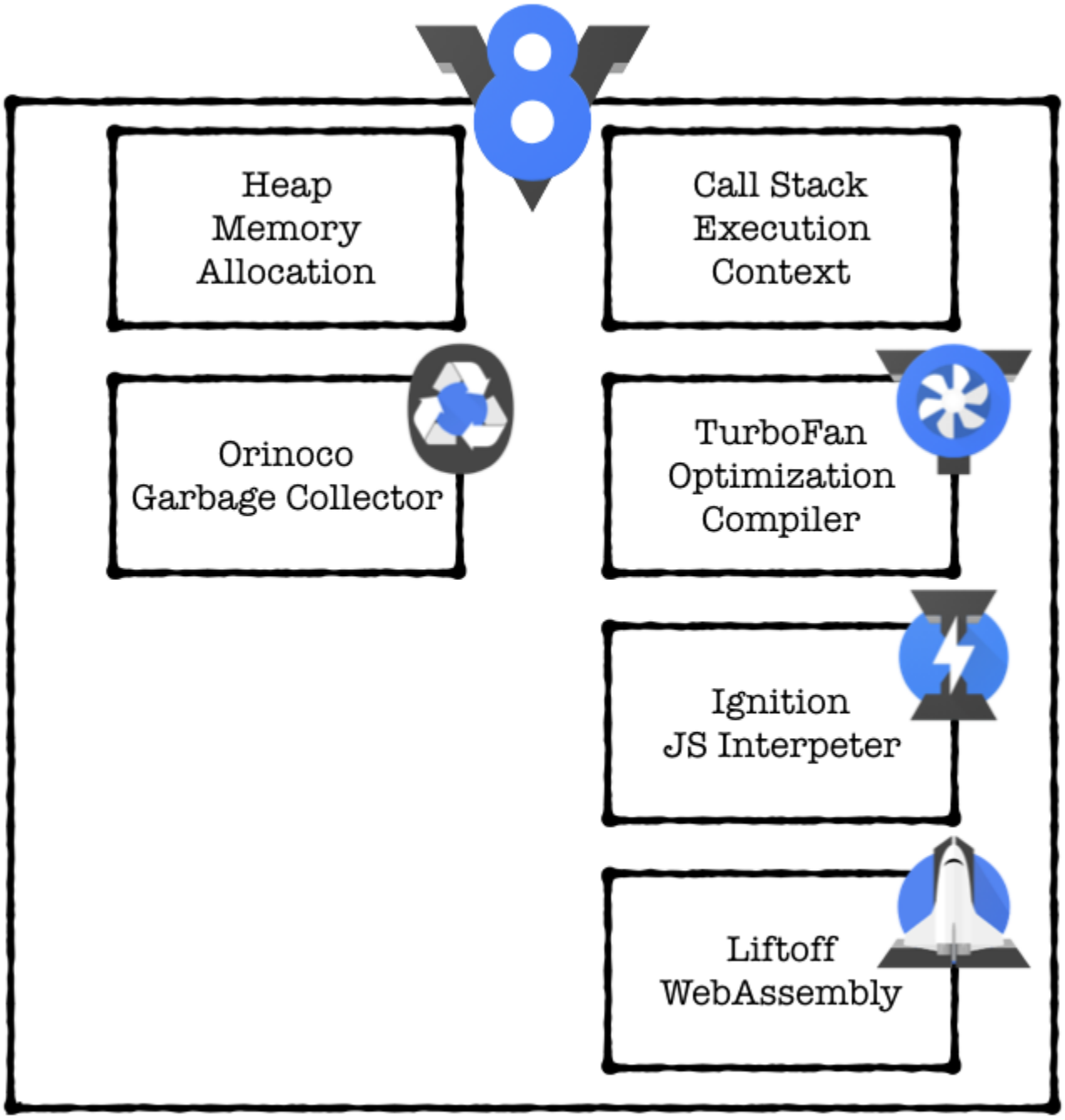
- Achieving great performance is also key in the browser, and this is the task Liftoff is used for—generating machine code.
- Not using the complex multi-tier compilation, Liftoff is a simpler code generator, which generates code for each opcode (a single portion of machine code, specifying an operation to be performed) at a time.
- Liftoff generates code much faster than TurboFan (~10x) which is obviously less performant (~50%).

Garbage Collection—Orinoco

- Running over the memory heap, looking for disconnected memory allocations is the Orinoco.
- Implementing a generational garbage collector, moving objects within the young generation, from the young to the old generation, and within the old generation.
- These moves leave holes, and Orinoco performs both evacuation and compaction to free space for more objects.

Garbage Collection—Orinoco

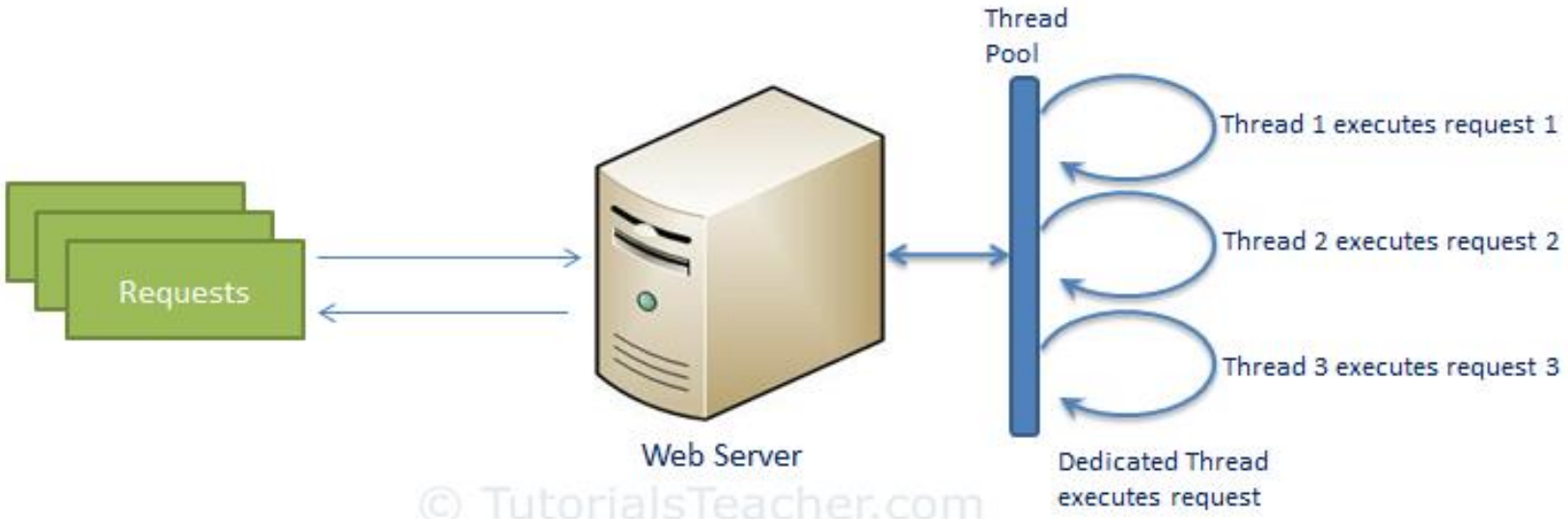
- Another optimization performed by Orinoco is in the way it searches through the heap to find all pointers that contain the old location of the objects moved and update them with the new location.
- This is made using a data structure called remembered set.
- On top of these, black allocation is added, which basically means the garbage collection process automatically marks living objects in black in order to speed up the iterative marking process..


Heap
Memory
Allocation
Call Stack
Execution
Context
Orinoco
Garbage Collector
TurboFan
Optimization
Compiler
Ignition
JS Interpeter
Liftoff
WebAssembly

Traditional Web Server Model

- In the traditional web server model, each request is handled by a dedicated thread from the thread pool.
- If no thread is available in the thread pool at any point of time then the request waits till the next available thread.
- Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

Traditional Web Server Model



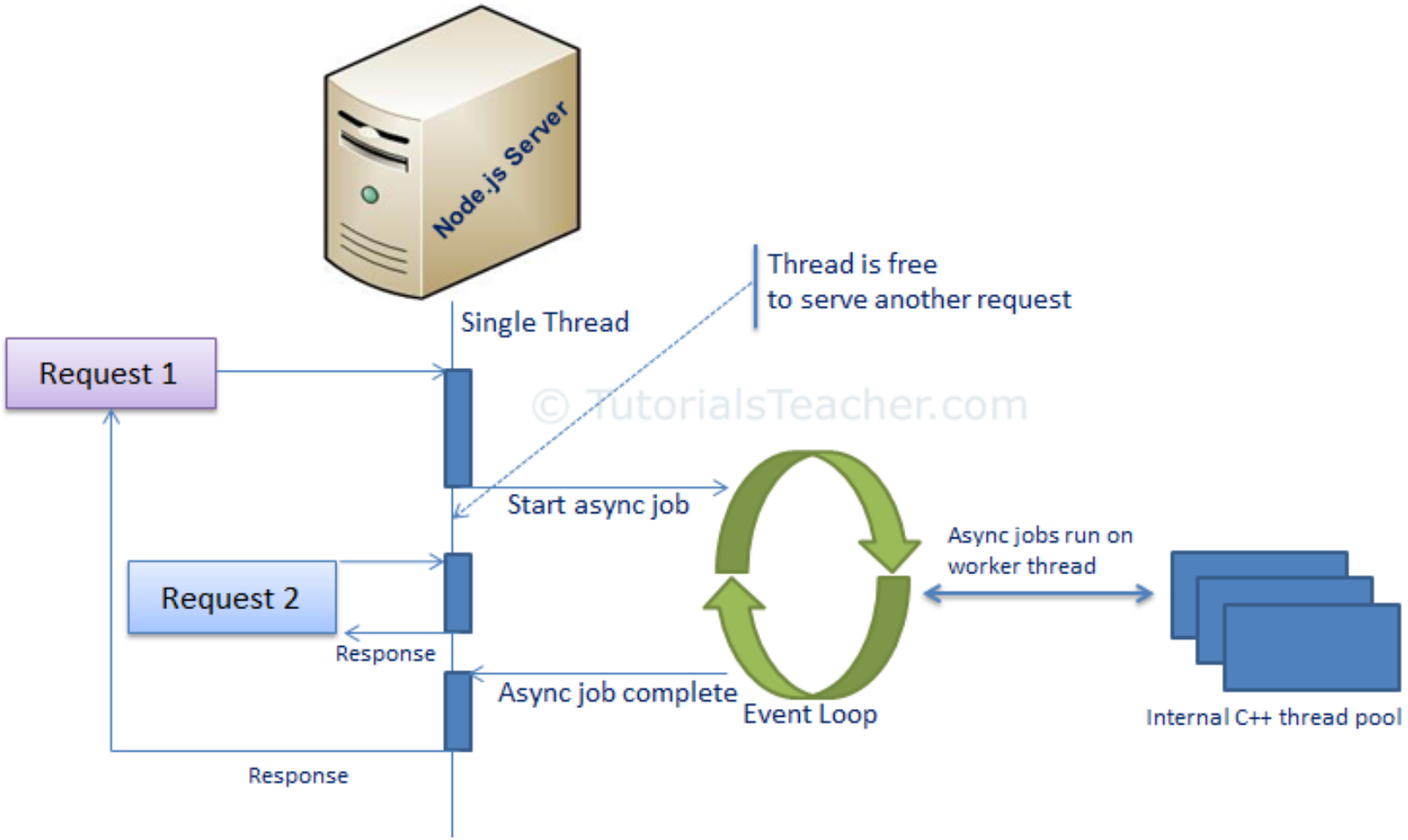
Node.js Process Model

- Node.js processes user requests differently when compared to a traditional web server model.
- Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.
- All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request.
- So, this single thread doesn't have to wait for the request to complete and is free to handle the next request.
- When asynchronous I/O work completes then it processes the request further and sends the response.

Node.js Process Model

- An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes.
- Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O..

Nodejs Thread Model



Nodejs Process Model

- Node.js process model increases the performance and scalability with a few caveats.
- Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

Why are threads bad?

- Hard to program
- Shared state and locks
- Deadlocks
- Giant locks decrease concurrency
- Fine-grained locks increase complexity
- Race conditions
- Context switching costs

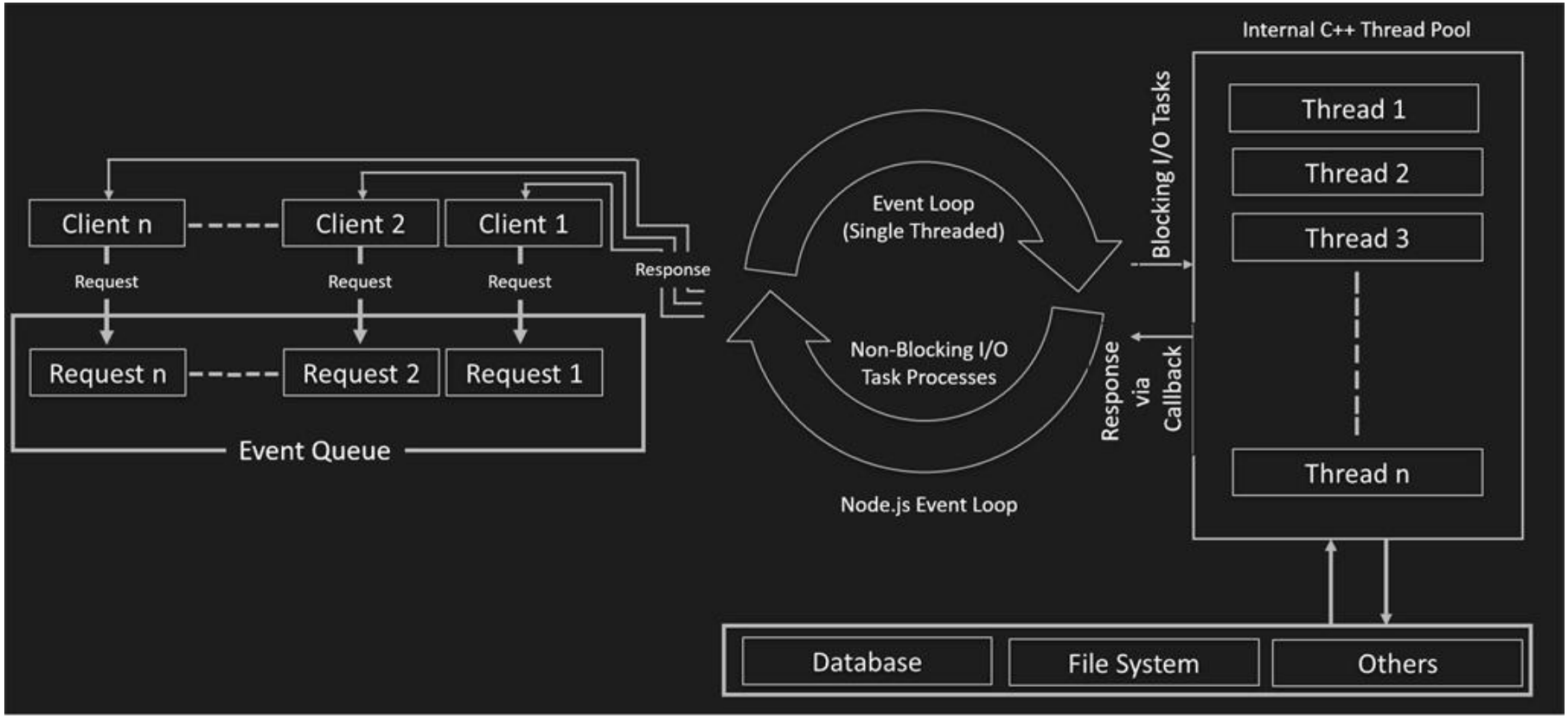
When threads are good?

- Support Multi-core CPUs
- CPU-heavy work
- Little or no shared state
- Threads count == CPU cores count

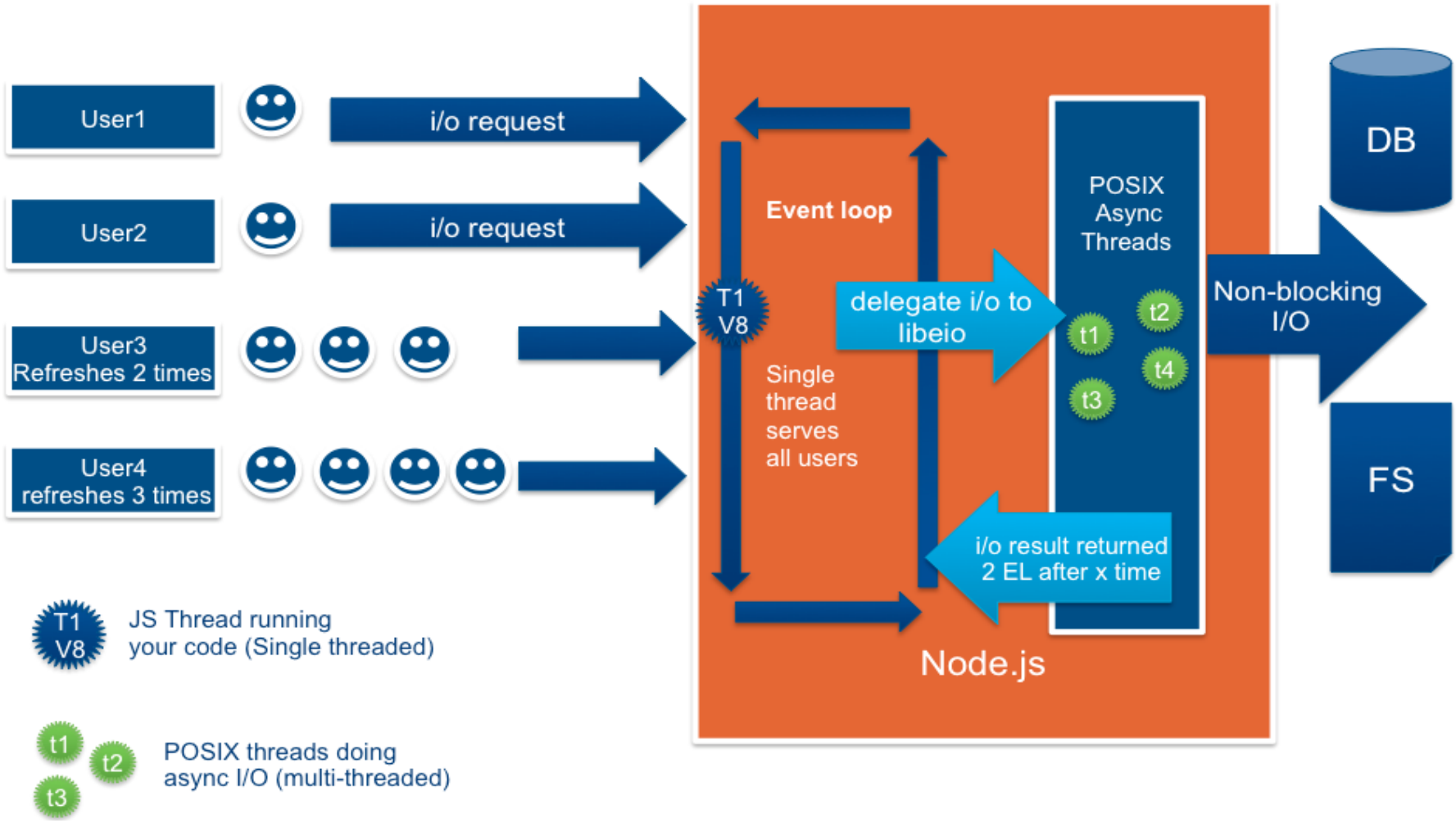
Node JS Architecture – Single Threaded Event Loop

- Node JS Platform does not follow Request/Response Multi-Threaded Stateless Model.
- It follows Single Threaded with Event Loop Model.
- Node JS Processing model mainly based on Javascript Event based model with Javascript callback mechanism.

Node.js Single Threading



Node.js Single Threading



Single Threaded Event Loop Model Processing Steps:

- Clients Send request to Web Server.
- Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.
- Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.
- Node JS Web Server internally has a Component, known as “Event Loop”. Why it got this name is that it uses indefinite loop to receive requests and process them

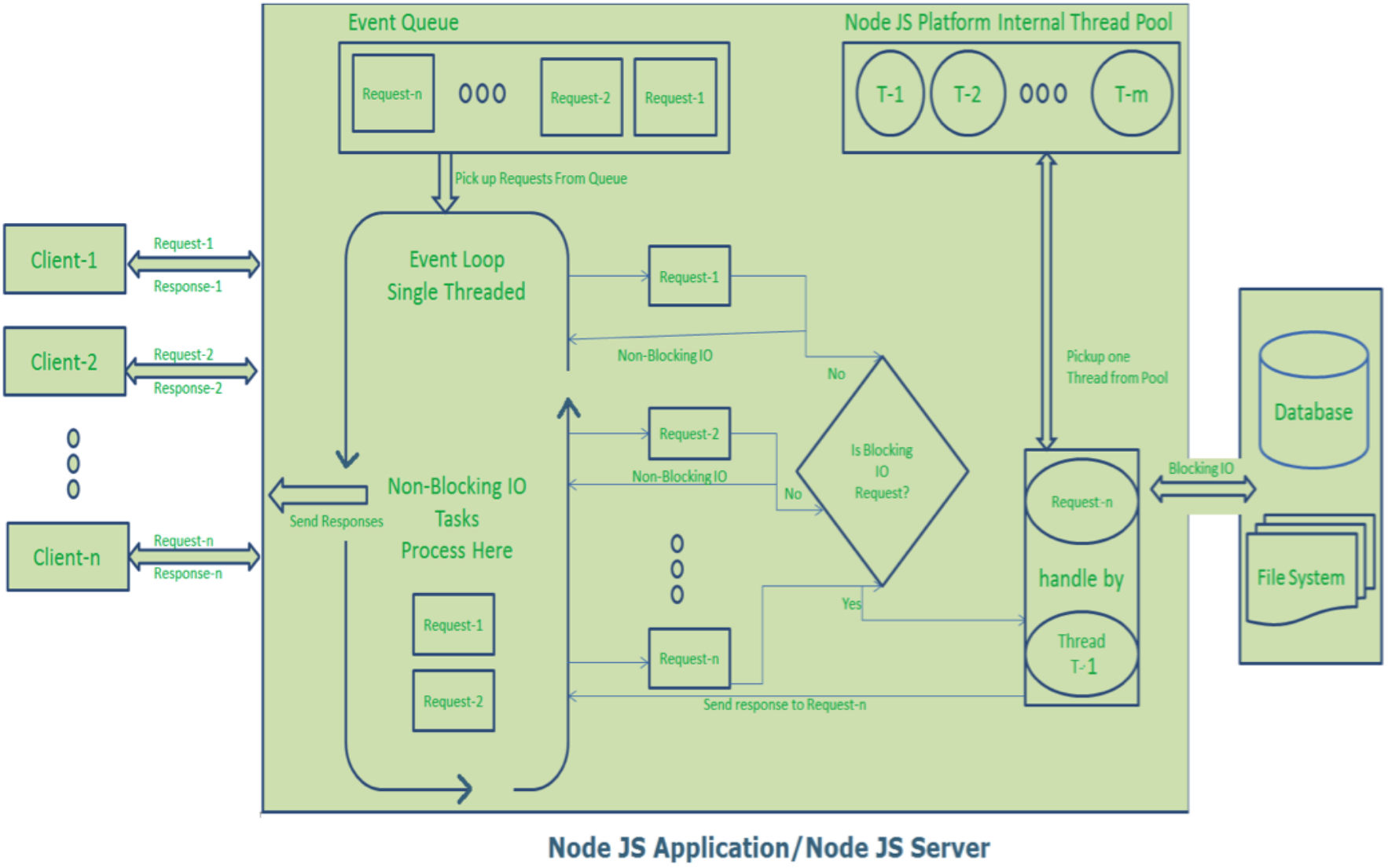
Single Threaded Event Loop Model Processing Steps:

- Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.
- Even Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.

Single Threaded Event Loop Model Processing Steps:

- If yes, then pick up one client request from event queue.
- It starts process for that client request.
- If that client request does not require any Blocking IO Operations, then process everything, prepare response and send it back to client.
- If that client request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
- Checks Threads availability from Internal Thread Pool
- Picks up one thread and assign this Client Request to that thread.
- That thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop.
- Event Loop in turn, sends that Response to the respective Client

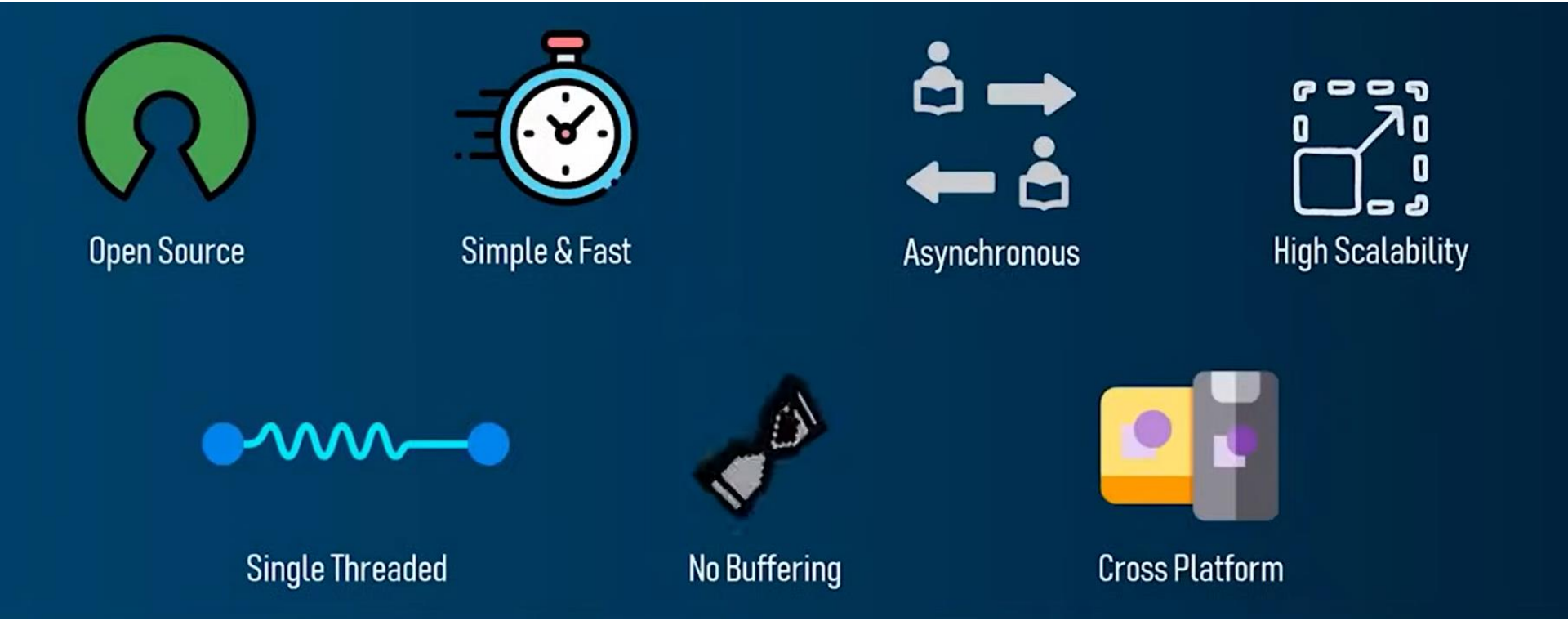
Single Thread Model



Node JS Architecture – Single Threaded Event Loop Advantages

- Handling more and more concurrent client's request is very easy.
- Even though our Node JS Application receives more and more Concurrent client requests, there is no need of creating more and more threads, because of Event loop.
- Node JS application uses less Threads so that it can utilize only less resources or memory

Features of Node JS



Features of Node JS

- Modularity
  - Major advantage of Node JS Platform is that it's modularity.
  - Each and every functionality is divided and implemented as a separate module or package.
  - When we install Node JS Platform, by default it installs only few modules.
  - If our application requires other modules, then we can easily install and configure then at any point of our application development phases.

Features of Node JS

- Node JS has thousands of thousands modules. Some modules were developed by Node JS Community and some were by Third-party Clients.

Features of Node JS

- Express JS
  - Node JS is used to develop Server-side Java Script. It also contains a separate module for Web Application Framework i.e. Express JS.
  - Non-blocking or Asynchronous IO
  - Node JS supports Non-blocking IO i.e. it uses Asynchronous IO Model to interact with File system or to do Socket communication or network communication.
  - Asynchronous IO Model means if IO processing is taking more time, then it permits other processing to continue before the transmission has finished.

Features of Node JS

- Java Developers are very much familiar about Blocking IO and Non-Blocking IO.
- Java IO package follows Blocking IO or Synchronous IO Architecture whereas Java NIO 2 follows Non-Blocking IO or Asynchronous IO Architecture.
- Event-Driven Asynchronous Platform
- Node JS Platform follows Even-Driven Loop architecture to interact or handle requests.

Features of Node JS

- MongoDB Wrappers API
  - Node JS platform contains a separate module to integrate MongoDB No SQL database with applications. It provides a MongoDB wrapper API.
  - We can use this API to write JavaScript easily to interact with MongoDB database. Node JS-mongodb module uses Mongoose to interact with MongoDB database.

Features of Node JS

- **Redis Client Library API**Node JS platform contains a separate module to integrate Redis No SQL database with applications.
- It provides a Redis wrapper API. We can use this API to write JavaScript easily to interact with Redis database.

Features of Node JS

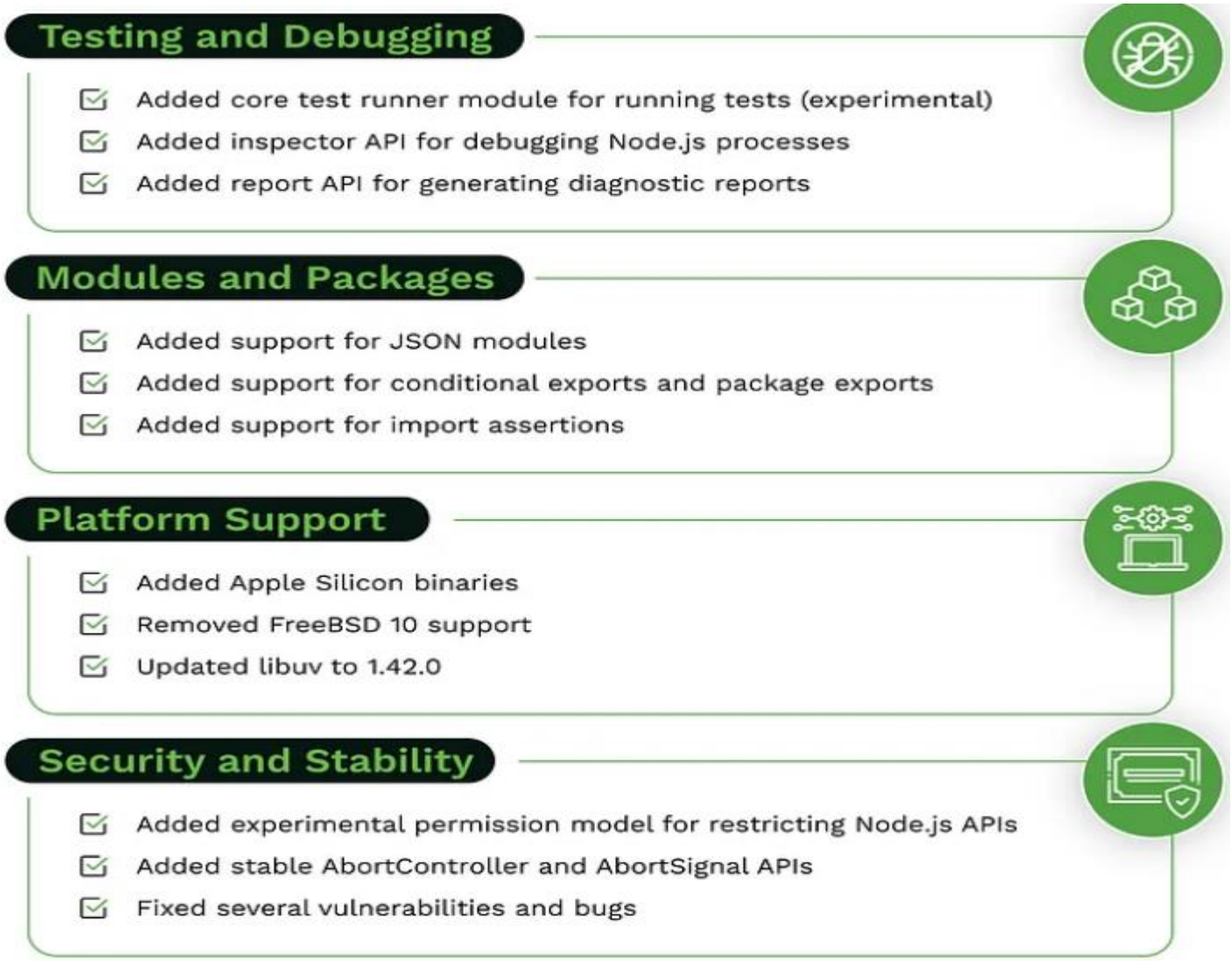
- Jade Template Engine
  - Node JS platform supports many template engines to write HTML. Default template engine supported by Node JS is “Jade”.
  - Jade is a whitespace-sensitive template engine for developing HTML applications very easily.
  - We can write template pages once and reuse them very easily to reduce development time. We will discuss how to install and write Jade templates in a separate post.

Features of Node JS

- Web Server
  - We can develop and use HTTP Web Server within no time.
  - We need to use http package to implement Web Server.
- Better Socket API
  - Node JS Platform provides very good Socket Module API to develop Real-time, Multi-User Chat and Multi-Player Gaming Applications very easily.
  - It supports Unix Socket programming like pipe().

Features of Node JS 18

Features of Node JS 18



Advantages of Node JS

- One Language and One Data Format
- Open Source
- Highly Scalable
- Better Performance and Low Latency
- Caching Modules
- Less Problems with Concurrency
- Easy to Extend and Lightweight
- Faster Development and Easy to Maintain

Advantages of Node JS

- REST API
- Active Development Community
- Unit Testing
- Streaming Data
- Creating Servers
- It can handle thousands of concurrent connections with minimal overhead (CPU/Memory) on a single process
- Easy Module Loading process

Who uses Node JS Platform

- DOW JONES
- When-To-Manage
- Linkedin
- MicroSoft
- Twelephone (It is a service for making calls right from Twitter account)
- Yahoo
- PayPal
- eBay
- STRONGLoop
- Wallmart
- Groupon
- SAP
- PINT Web Platform (PWP)
- Cloud9 IDE
- Dropbox
- ZingChart
- Chess (It is an online game available on facebook.com)

Node JS Applications

- Network applications
- Asynchronous Events
- Intelligent Networking Proxies
- Proxy Server
- Multiplayer Game Applications
- Data Intensive Real-time applications
- Web Applications
- High Definition(HD) Voice and Video Communications
- File Uploading Tools
- Process Monitoring Tools
- Many-To-Many Instant Chatting Applications
- Many-To-Many Instant Messing Applications
- HTTP Web Server
- Real-time Logistics Systems
- Streaming Server
- High Concurrency Applications
- Communication Hubs
- Coordinators
- DNS Server
- Static File Server
- TCP Server
- Stock-Trading Dashboard

Nodejs Versions

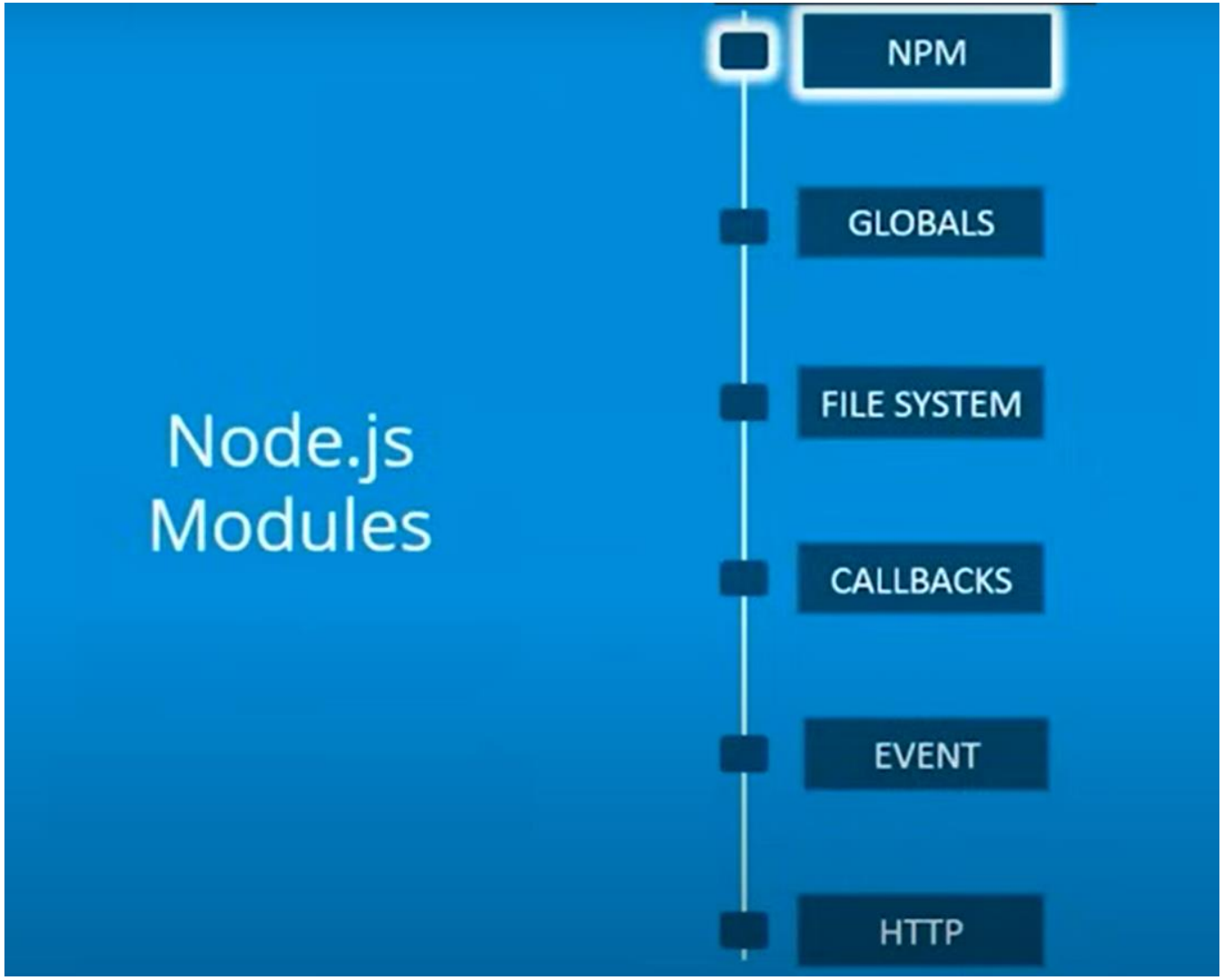
Release[53]	Status	Code name	Release date	Maintenance end
0.10.x	End-of-Life		2013-03-11	2016-10-31
0.12.x	End-of-Life		2015-02-06	2016-12-31
4.x	End-of-Life	Argon[54]	2015-09-08	2018-04-30
5.x	End-of-Life		2015-10-29	2016-06-30
6.x	End-of-Life	Boron[54]	2016-04-26	2019-04-30
7.x	End-of-Life		2016-10-25	2017-06-30
8.x	End-of-Life	Carbon[54]	2017-05-30	2019-12-31
9.x	End-of-Life		2017-10-01	2018-06-30
10.x	End-of-Life	Dubnium[54]	2018-04-24	2021-04-30
11.x	End-of-Life		2018-10-23	2019-06-01
12.x	End-of-Life	Erbium[54]	2019-04-23	2022-04-30
13.x	End-of-Life		2019-10-22	2020-06-01
14.x	End-of-Life	Fermium[54]	2020-04-21	2023-04-30

Nodejs Versions

15.x	End-of-Life		2020-10-20	2021-06-01
16.x	End-of-Life	Gallium[54]	2021-04-20	2023-09-11[55]
17.x	End-of-Life		2021-10-19	2022-06-01
18.x	Maintenance LTS	Hydrogen[54]	2022-04-19	2025-04-30
19.x	End-of-Life		2022-10-18	2023-06-01
20.x	Active LTS	Iron[56]	2023-04-18	2026-04-30
21.x	Maintenance	[54]	2023-10-17	2024-05-30
22.x	**Current**	Jod [56][54]	2024-04-24	2027-04-30
23.x	Planned	[54]	2024-10-14	2025-05-27

Legend: Old version | Older version, still maintained | **Latest version** | Future release

NPM

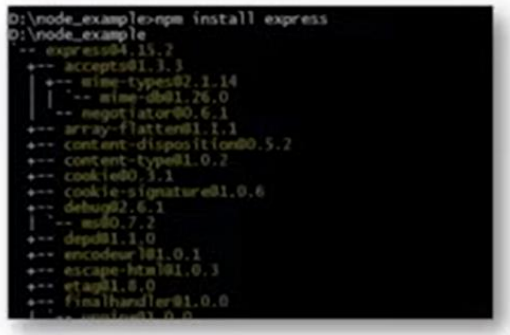

Node.js
Modules
NPM
GLOBALS
FILE SYSTEM
CALLBACKS
EVENT
HTTP

NPM

- NPM => Node Package Manager
- Provides online repositories for node.js packages/modules
- Provides command line utility to install Node.js packages

```
D:\node_example>npm version
{ npm: '3.10.10',
  ares: '1.10.1-DEV',
  http_parser: '2.7.0',
  icu: '57.1',
  modules: '48',
  node: '6.9.5',
  openssl: '1.0.2k',
  uv: '1.9.1',
  v8: '5.1.281.89',
  zlib: '1.2.8' }

D:\node_example>
```

npm install	→ Install all the modules as specified in package.json
npm install **<Module** Name>	→ Install Module using npm
npm install **<Module** Name> **-g**	→ Install dependency globally



Global Objects

These objects are available in all modules

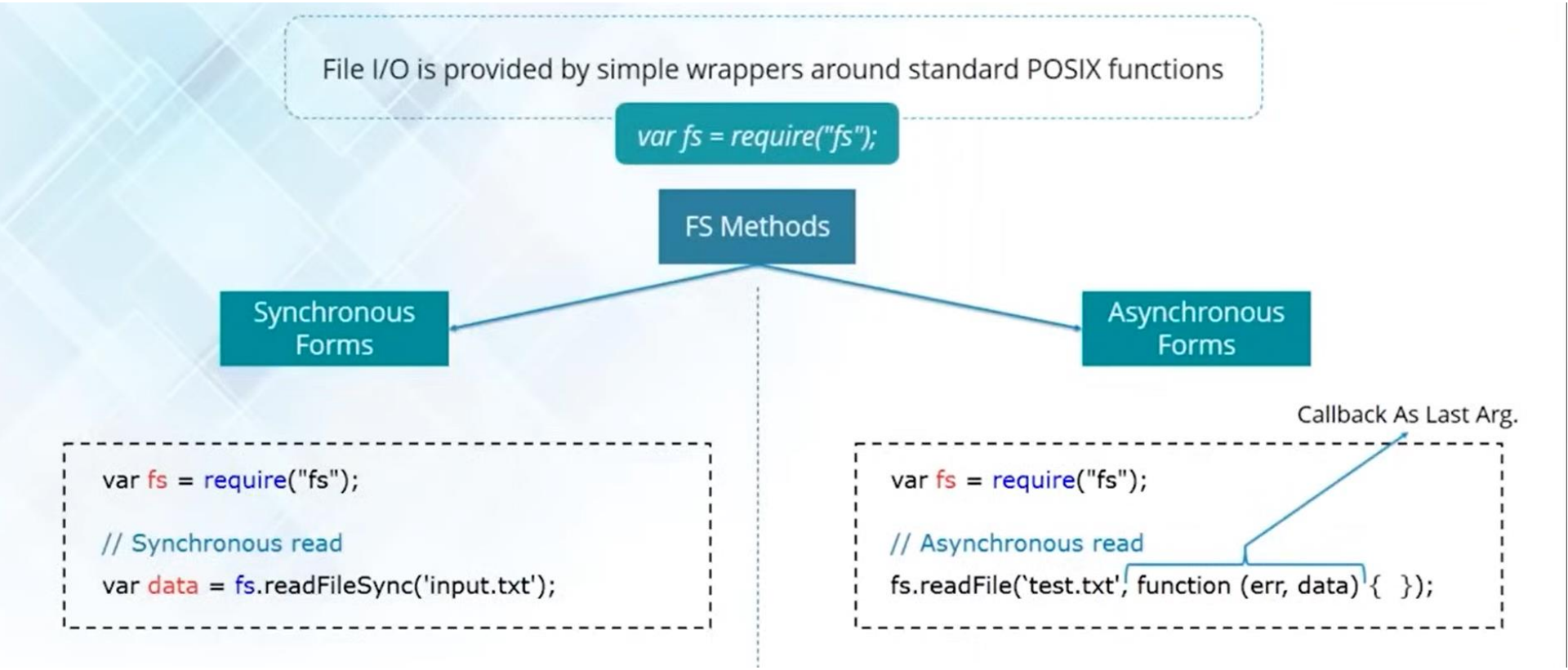
__dirname → specifies the name of the directory that currently contains the code.

__filename → specifies the name of the file that currently contains the code.

A timer in Node.js is an internal construct that calls a given function after a certain period of time

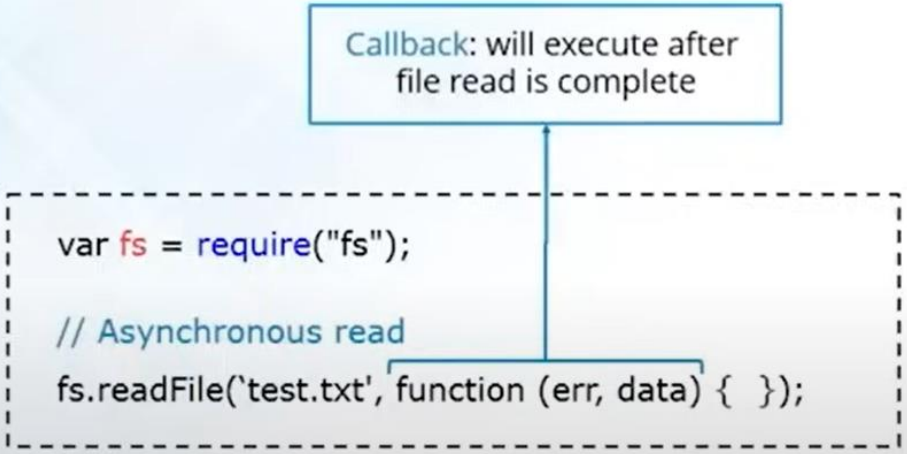
1. setTimeout(callback, delay[, ...args])
2. setInterval(callback, delay[, ...args])
3. setImmediate(callback, [,..args])

File System


File I/O is provided by simple wrappers around standard POSIX functions
var fs = require("fs");
FS Methods
Synchronous Forms
Asynchronous Forms
var fs = require("fs");
// Synchronous read
var data = fs.readFileSync('input.txt');
Callback As Last Arg.
var fs = require("fs");
// Asynchronous read
fs.readFile('test.txt', function (err, data){ });

Callback

Callback is an asynchronous equivalent for a function and is called at the completion of each task



Event

- Node.js follows event-driven architecture
- Certain objects (emitters) periodically emit events which further invokes the listeners
- Node.js provide concurrency by using the concept of events and callbacks
- All objects that emit events are instances of the EventEmitter class

```
var fs =  require('fs');
var event = require('events');

const myEmitter = new event.EventEmitter();

fs.readFile('test1.txt',(err, data) => {
    console.log(data.toString());
    myEmitter.emit('readFile');
});

myEmitter.on('readFile', () => {
  console.log('\nRead Event Occurred!');
});
```

Import Events Module

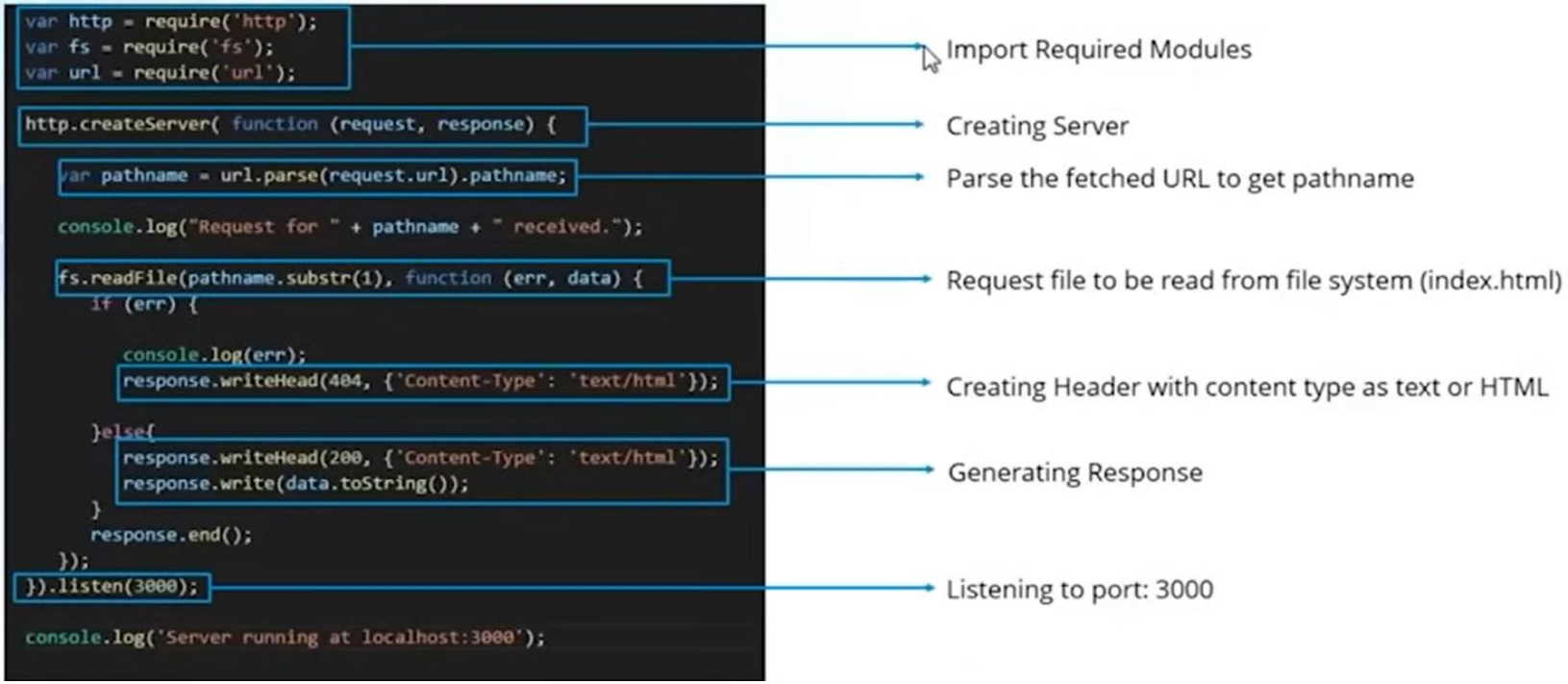
Creating object of EventEmitter

Emitting event

Registering Listener and defining event handler

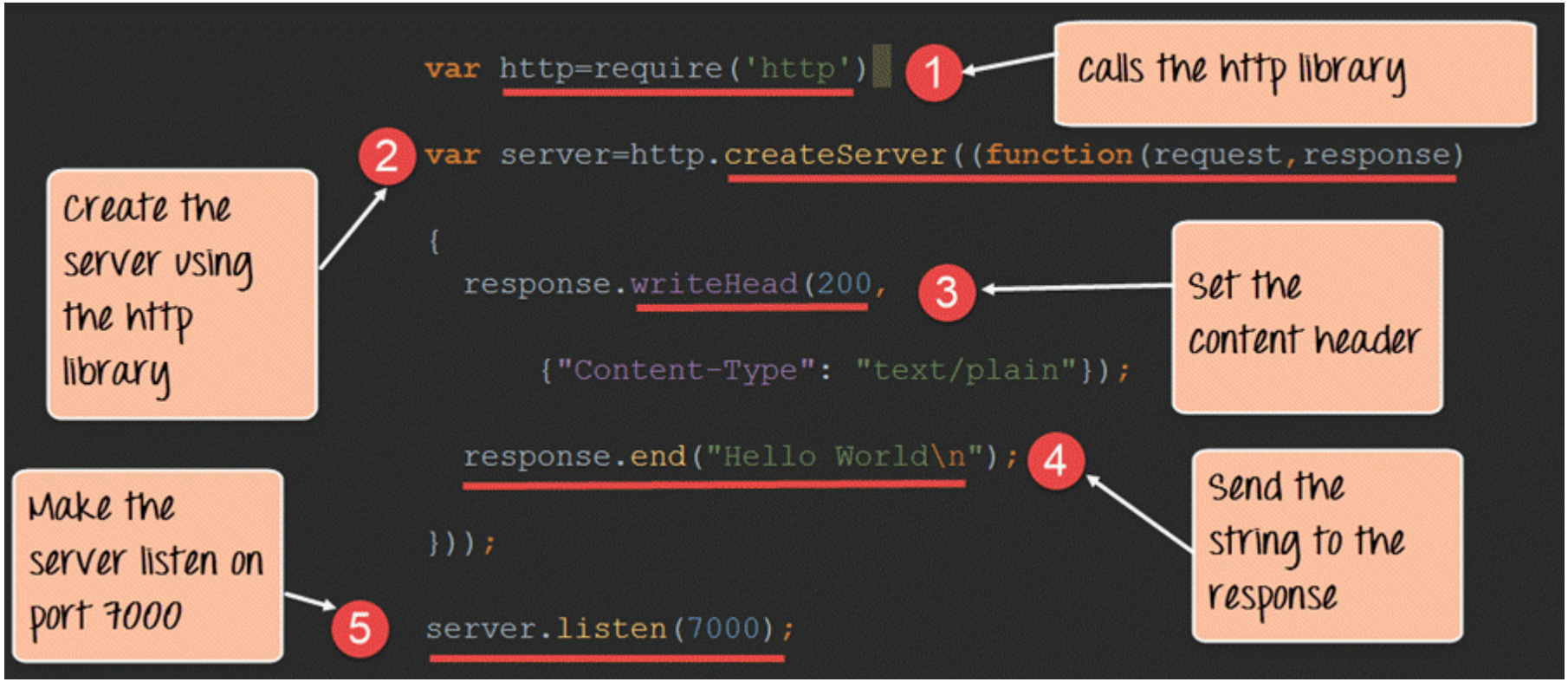
Event

- To use the HTTP server and client one must require('http')
- The HTTP interfaces in Node.js are designed to support many features of the protocol



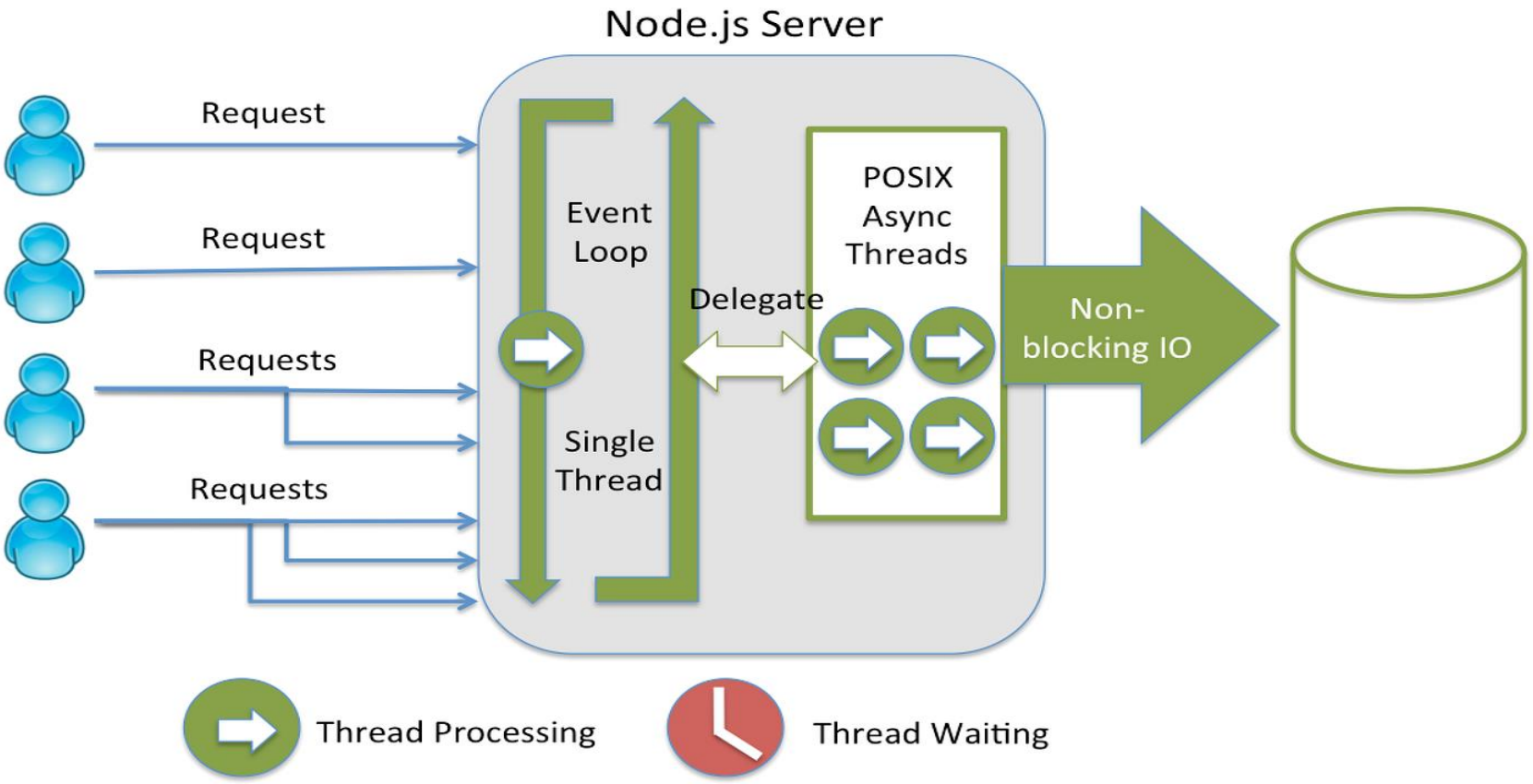
Http

- Http Call and Server Setup

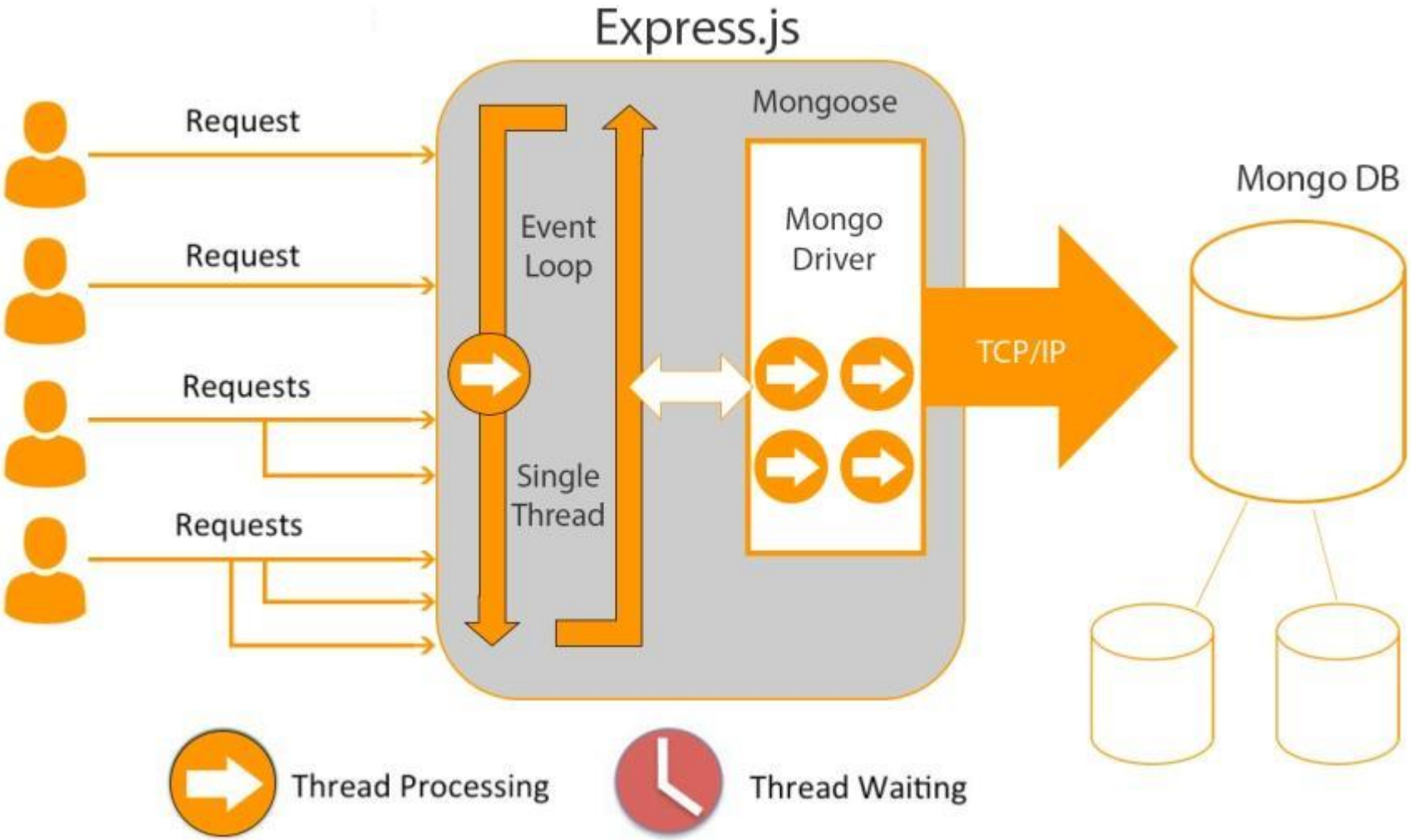


Http

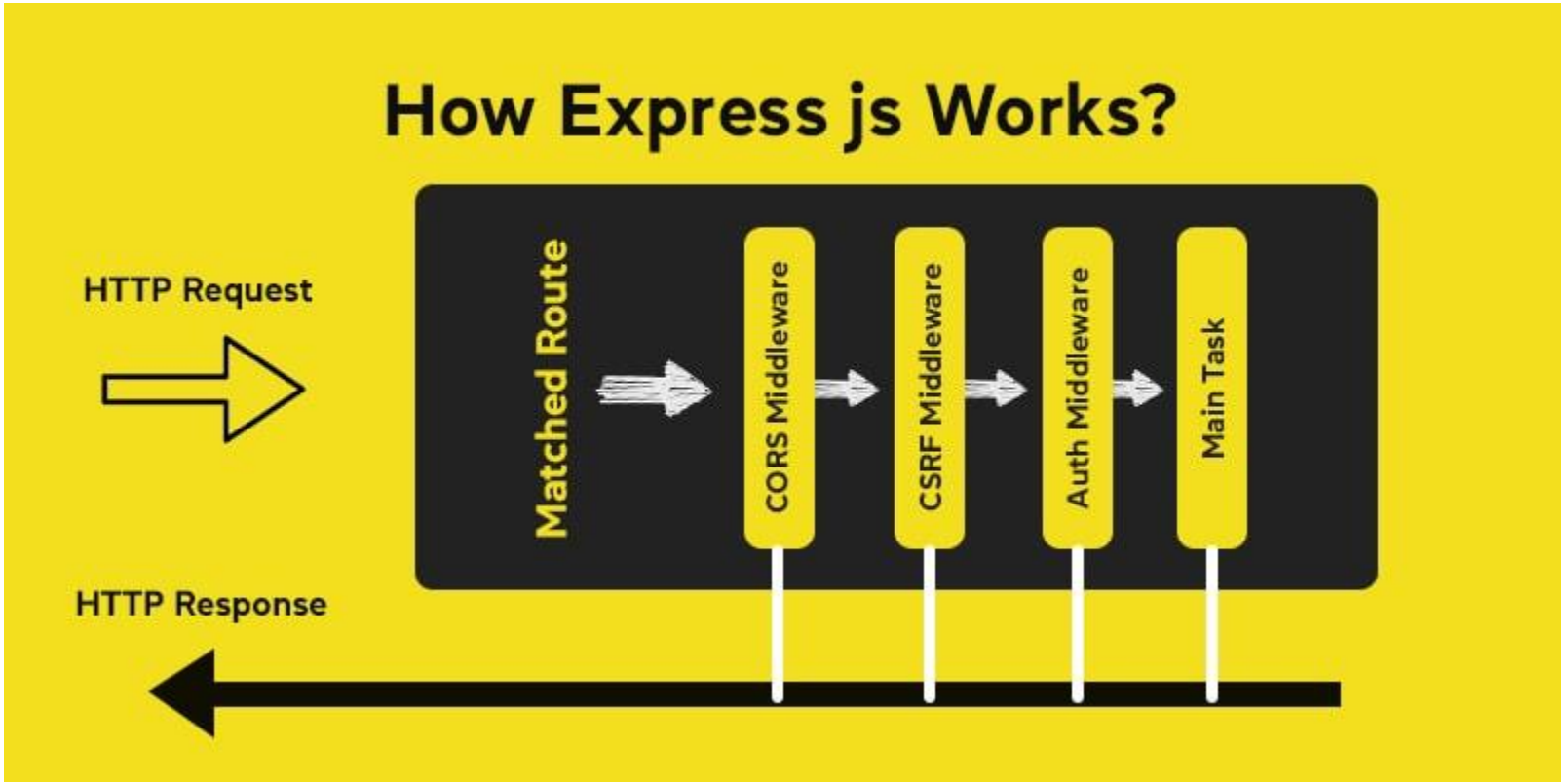
- Http Call and Server Setup



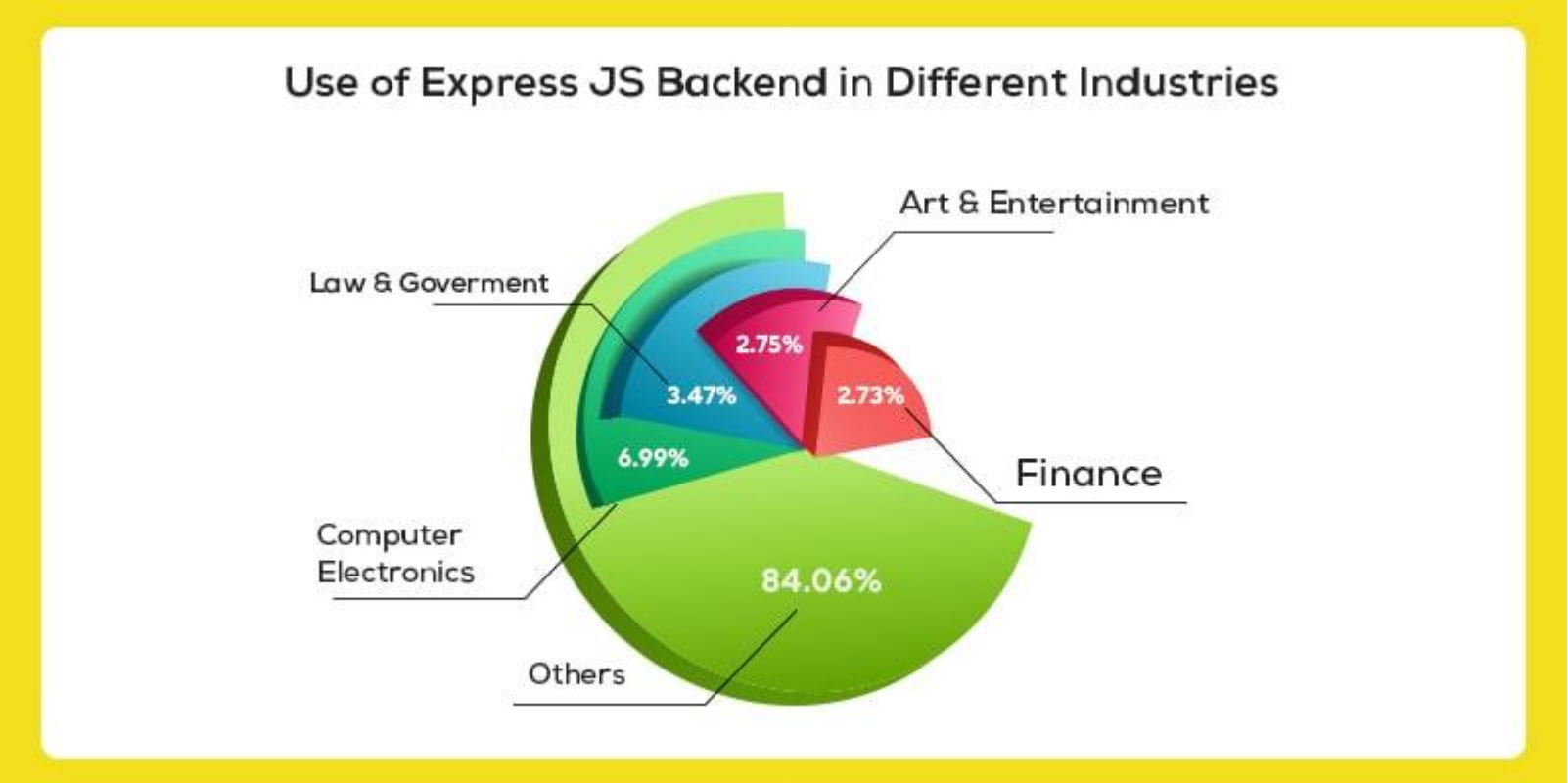
Expressjs


Express.js
Request
Request
Requests
Requests
Event Loop
Single Thread
Mongoose
Mongo Driver
TCP/IP
Mongo DB
Thread Processing
Thread Waiting

Expressjs


How Express js Works?
HTTP Request
Matched Route
CORS Middleware
CSRF Middleware
Auth Middleware
Main Task
HTTP Response

Expressjs


Use of Express JS Backend in Different Industries
Art & Entertainment
Law & Goverment
2.75%
3.47%
2.73%
6.99%
Finance
Computer
Electronics
84.06%
Others

Expressjs

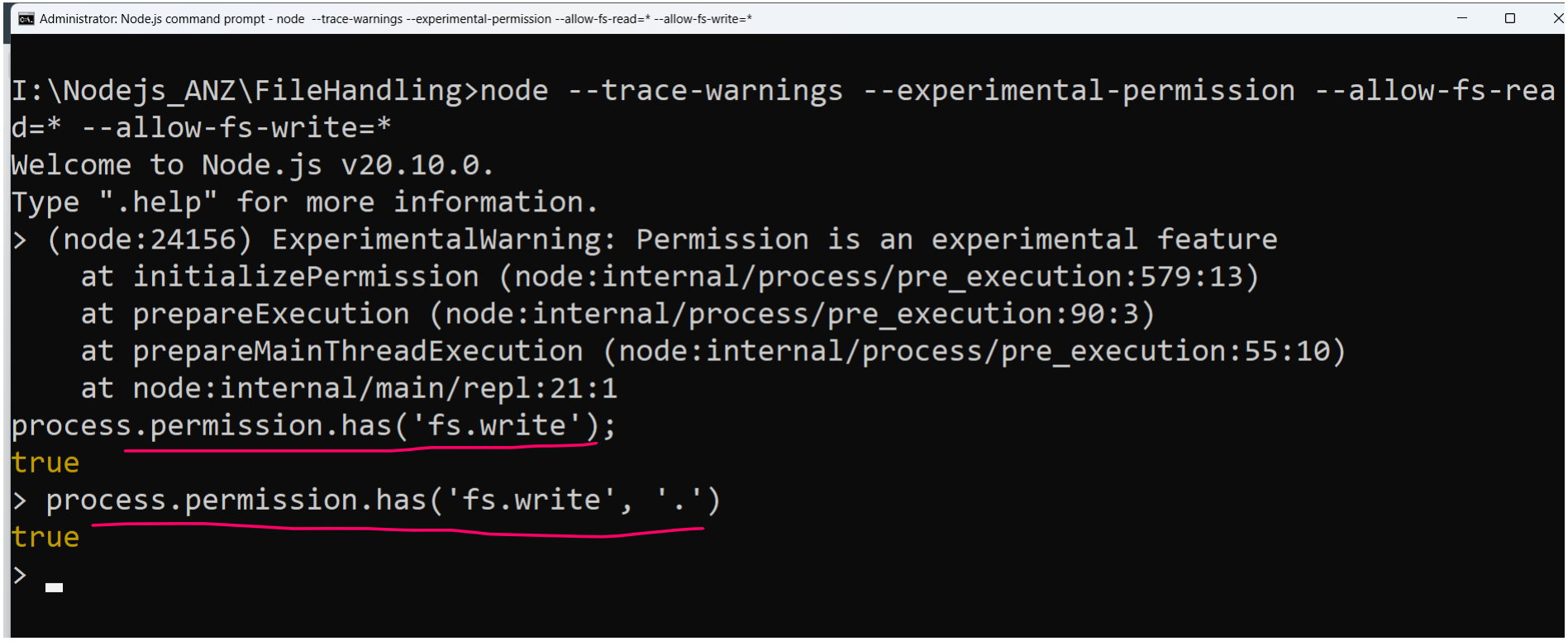

Advantages Of Express JS
Has Detailed Documentation
Huge Supporting Community
Supports Many Third-Party Plugins
Very Simple & Fast
API
Powerful Routed API

Nodejs 20.x Features

- Permission Model
  - The Node.js Permission Model is an experimental mechanism for restricting access to specific resources during execution.
  - In this first release containing the Permission Model, the features come with the following abilities:
    - Restrict access to the file system (read and write)
    - Use --allow-fs-read and --allow-fs-write
    - Restrict access to child_process
    - Use --allow-child-process
    - Restrict access to worker_threads
    - Use --allow-worker
    - Restrict access to native addons (same as --no-addons flag)
  - When starting Node.js with --experimental-permission, the ability to access the file system, spawn processes, and use node:worker_threads will be restricted.

Nodejs 20.x Features

- $ node --experimental-permission --allow-fs-read=* --allow-fs-write=*



Nodejs Project Creation Best Practices

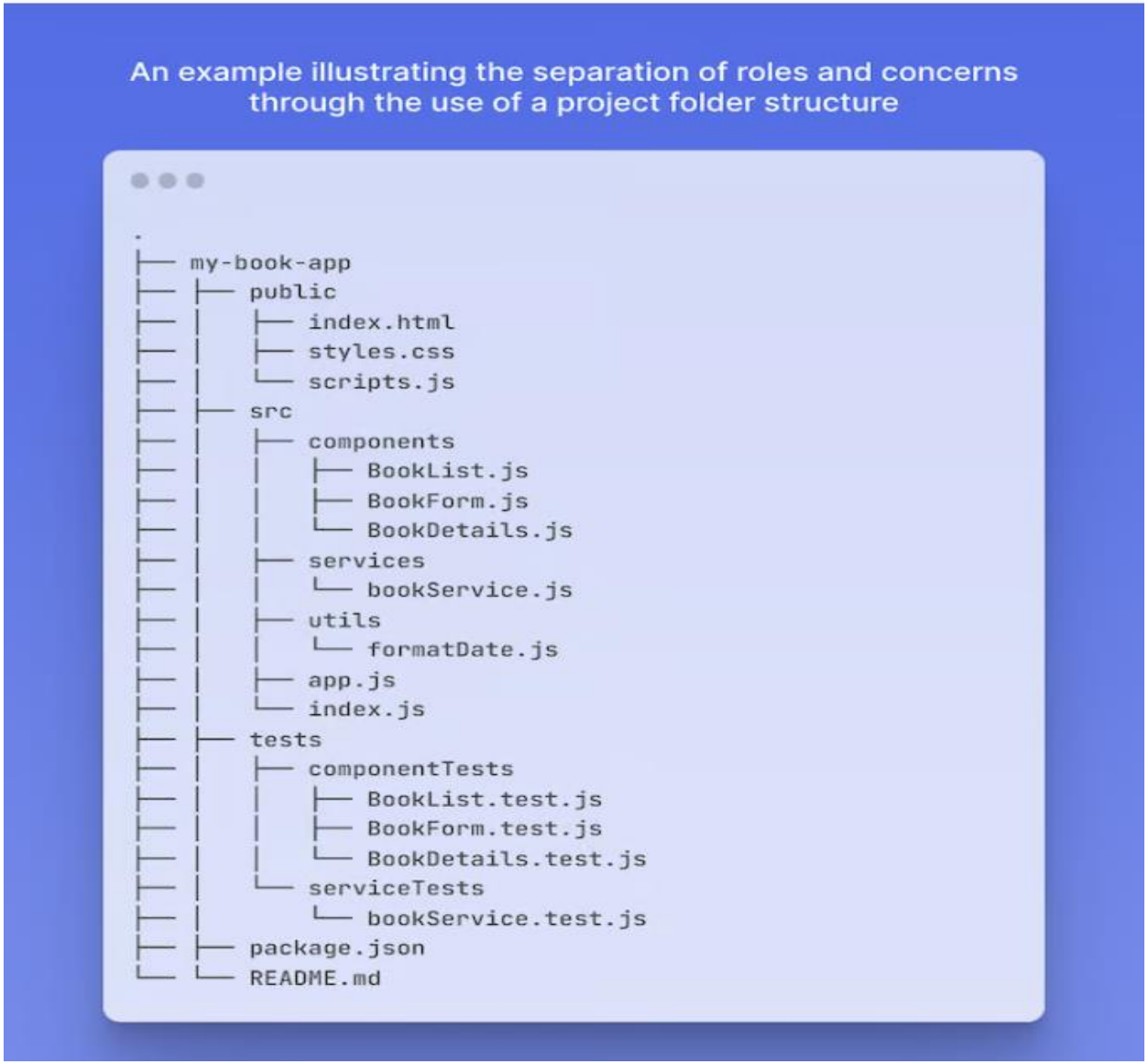
- Separating roles and concerns using folder structures
- Practice modular code
- Focus on code readability
- Separate business logic and API routes
- Utilize the MVC pattern
- Use service and data access layers
- Organize configuration separately
- Separate development scripts from the main code
- Use dependency injection

Nodejs Project Creation Best Practices

- Conduct unit testing
- Use another layer for third-party services calls
- Use a JavaScript code linter
- Use a style guide
- Optimize performance with caching
- Consider serverless architecture
- Use gzip compression
- Use promises and async/await
- Handle errors more often

Nodejs Project Creation Best Practices

An example illustrating the separation of roles and concerns through the use of a project folder structure



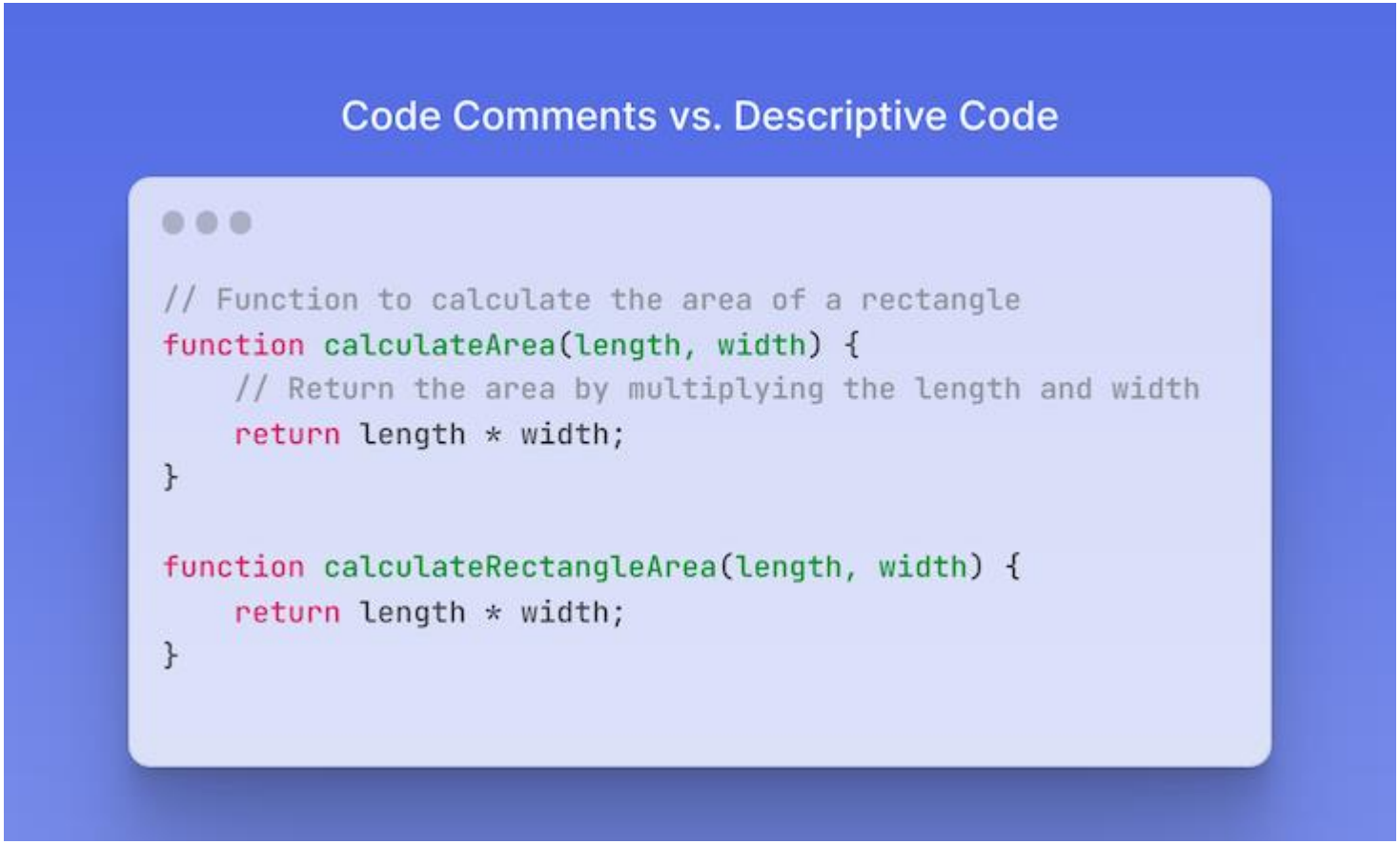
Practice modular code

- Node.js project architecture is to break your application into smaller modules, each handling a specific functionality.
- Following the Single Responsibility Principle (SRP) of SOLID software development to design each module will ensure a single responsibility or purpose, making it easier to understand, test, and maintain.
- “A module should be responsible to one, and only one, actor.”

Practice modular code

- It’s also recommended to minimize the use of global variables as they can lead to tightly-coupled code and make it challenging to identify dependencies.
- Instead, encapsulate variables within modules and expose only the necessary interfaces.
- If the code of a class, function, or file becomes excessively lengthy, consider splitting it into smaller modules wherever possible and bundle them within related folders.
- This approach helps in grouping related files together and is essential for enhancing code modularity and organization.

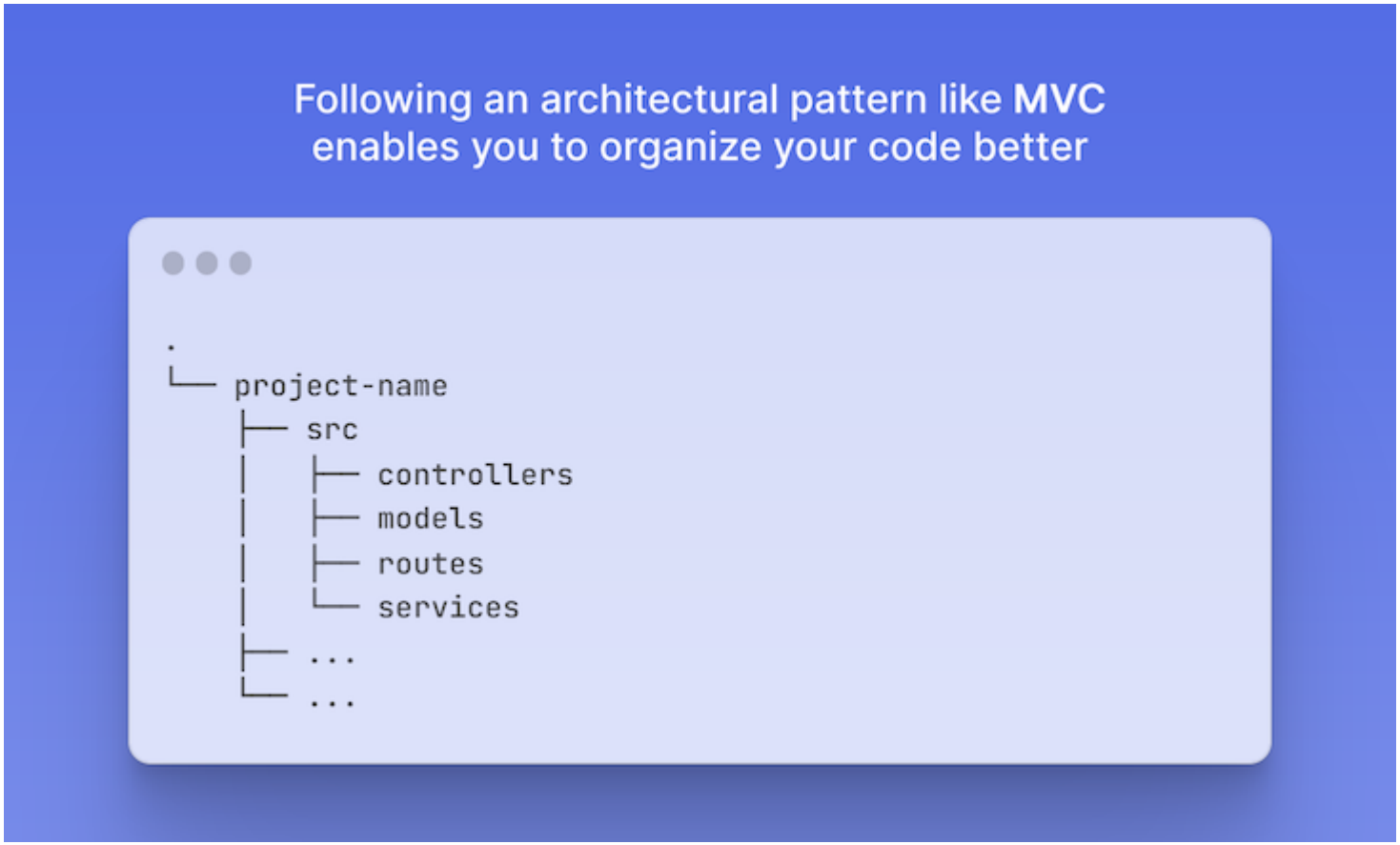
Focus on code readability



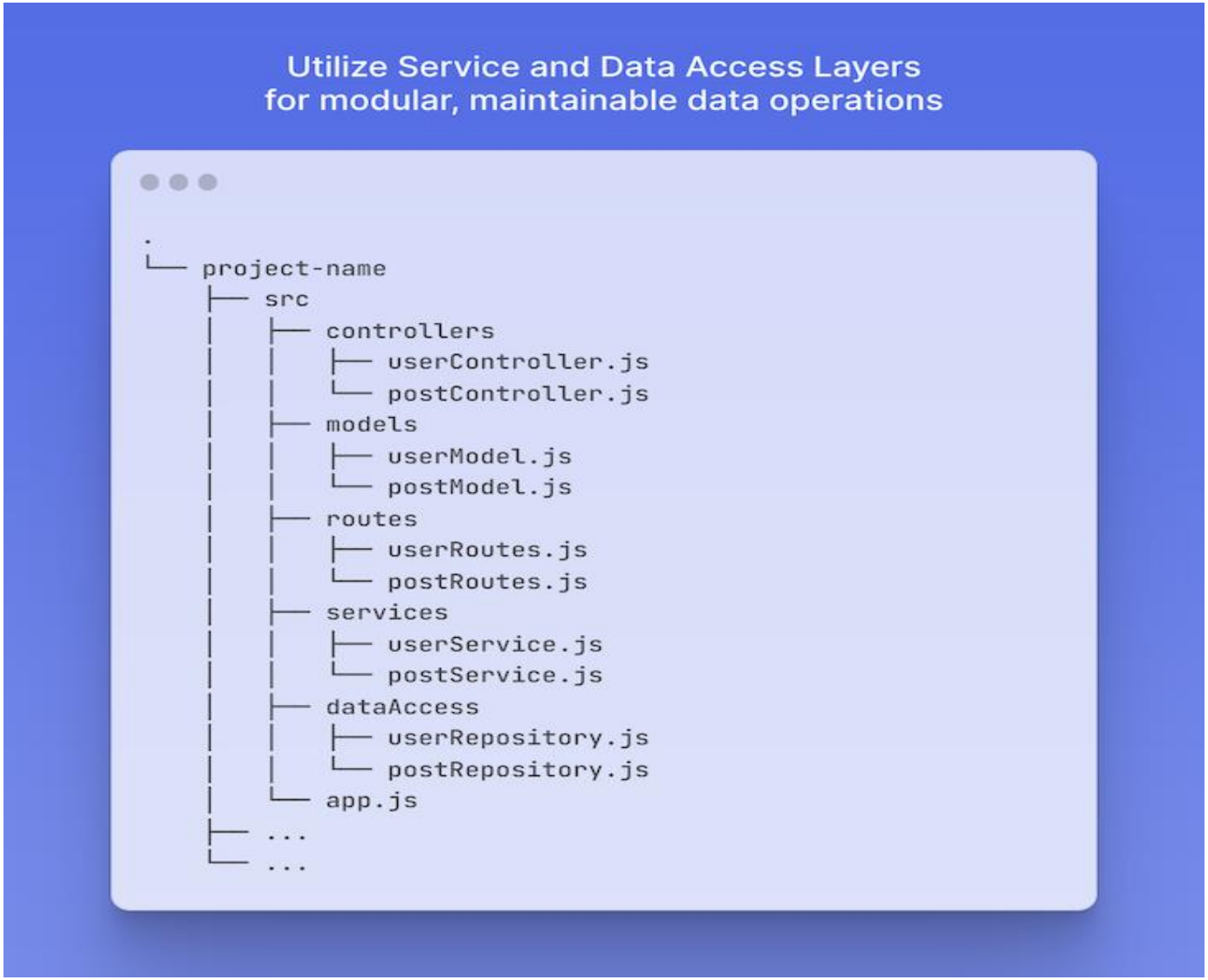
Separate business logic and API routes

- Frameworks like Express.js offer incredible features for managing requests, views, and routes.
- With such support, it can be tempting to place our business logic directly in our API routes.
- Unfortunately, this approach quickly leads to the creation of large, monolithic blocks that become unmanageable, difficult to read, and prone to decomposition in the long run.

Nodejs Project Creation Best Practices



Nodejs Project Creation Best Practices



Nodejs Project Creation Best Practices

Use dependency injection

- Streamlined unit testing: Inject dependencies directly into modules for easier and more comprehensive unit testing
- Reduced module coupling: Avoid tight coupling between modules, leading to better code maintainability and reusability
- Accelerated Git flow: Define stable interfaces for dependencies, reducing conflicts and facilitating collaborative development
- Enhanced scalability: Add or modify components without extensive changes to the existing codebase, enabling easier extension and adaptation
- Great code reusability: Decouple components and inject dependencies to reuse modules across different parts of the application or other projects

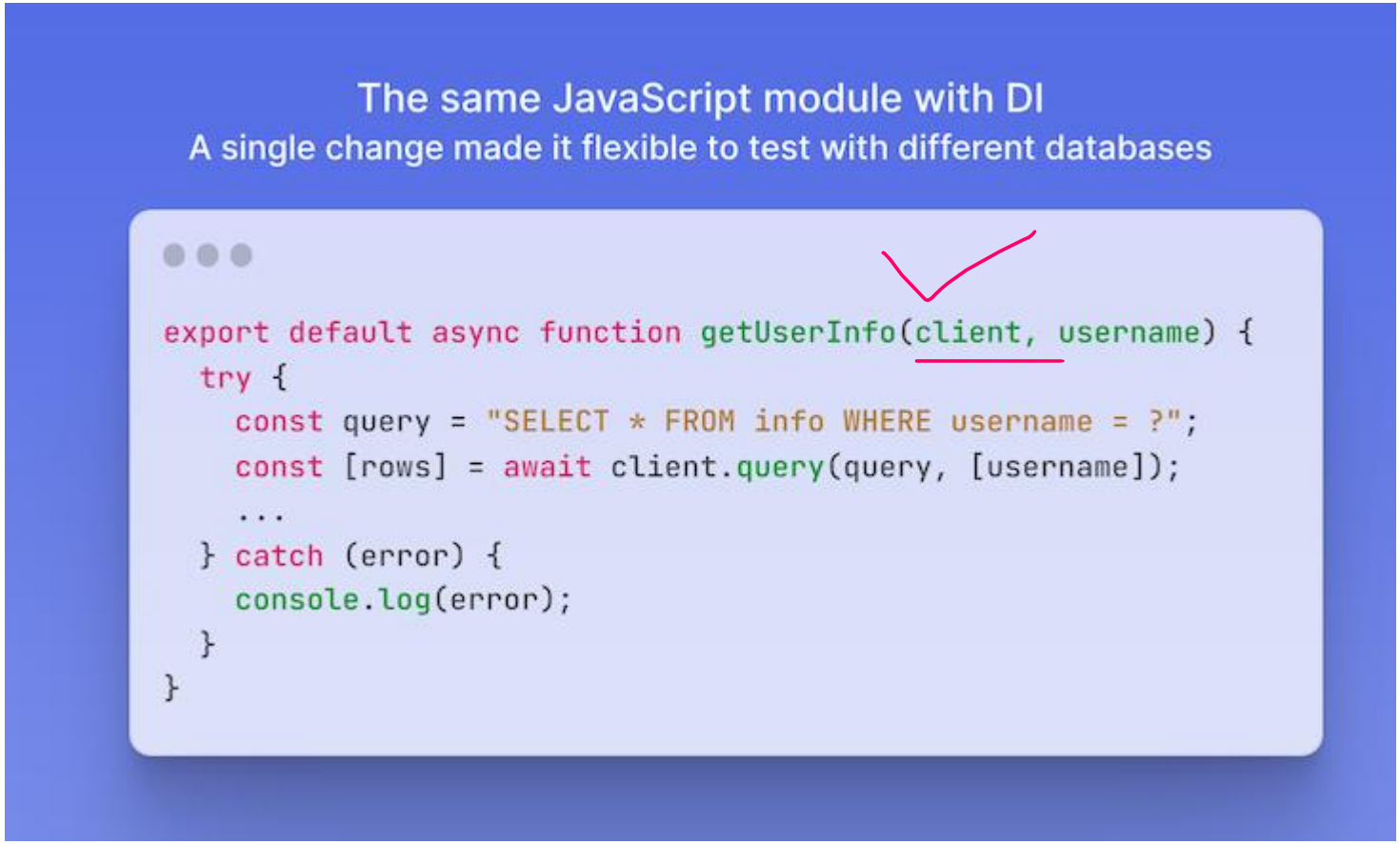
Use dependency injection

A JavaScript module without DI

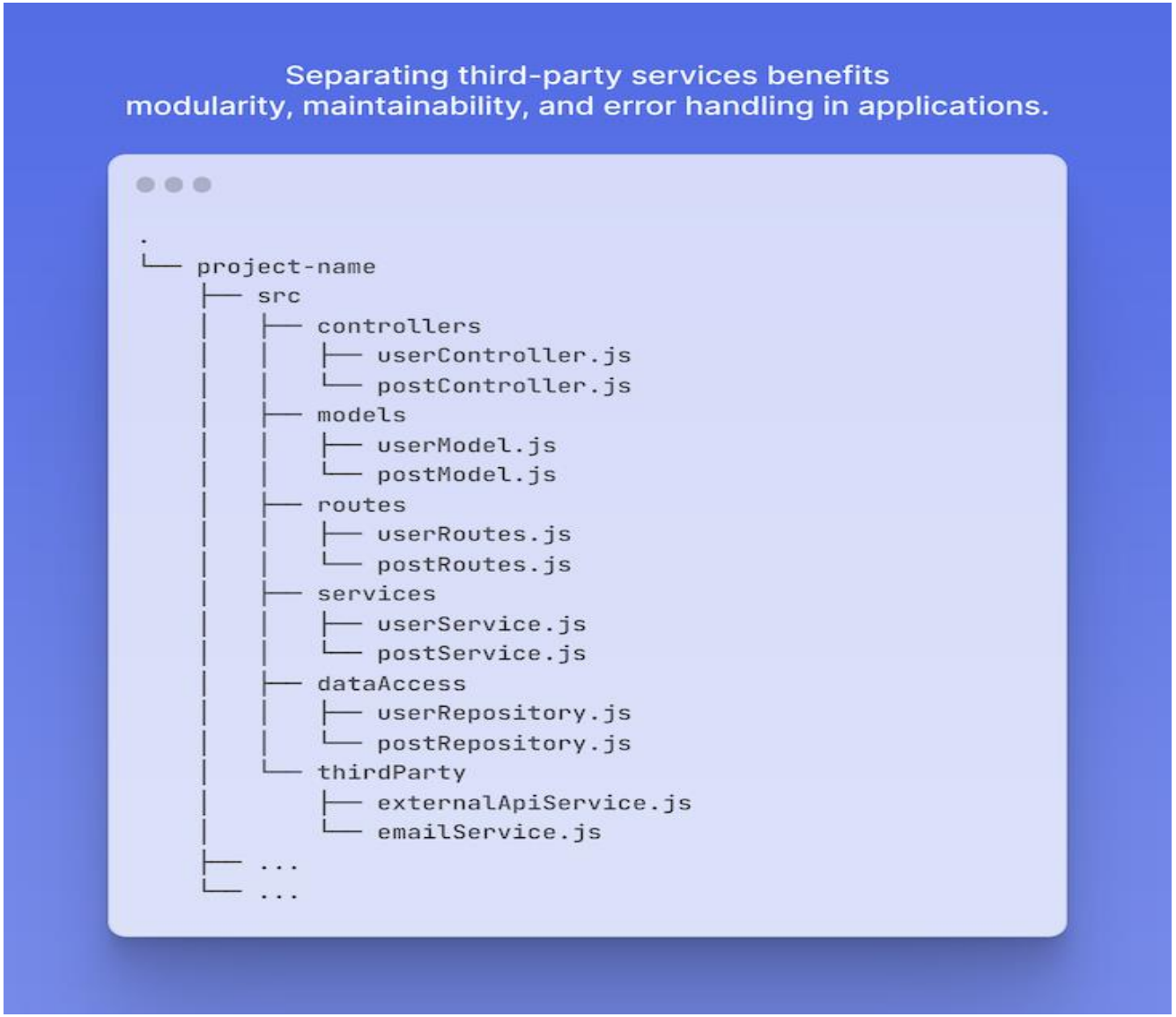
Simple but not flexible enough to support testing

```
import client from "../config/database.js";

export default async function getUserInfo(username) {
  try {
    const query = "SELECT * FROM info WHERE username = ?";
    const [rows] = await client.query(query, [username]);
    ...
  } catch (error) {
    console.log(error);
  }
}
```

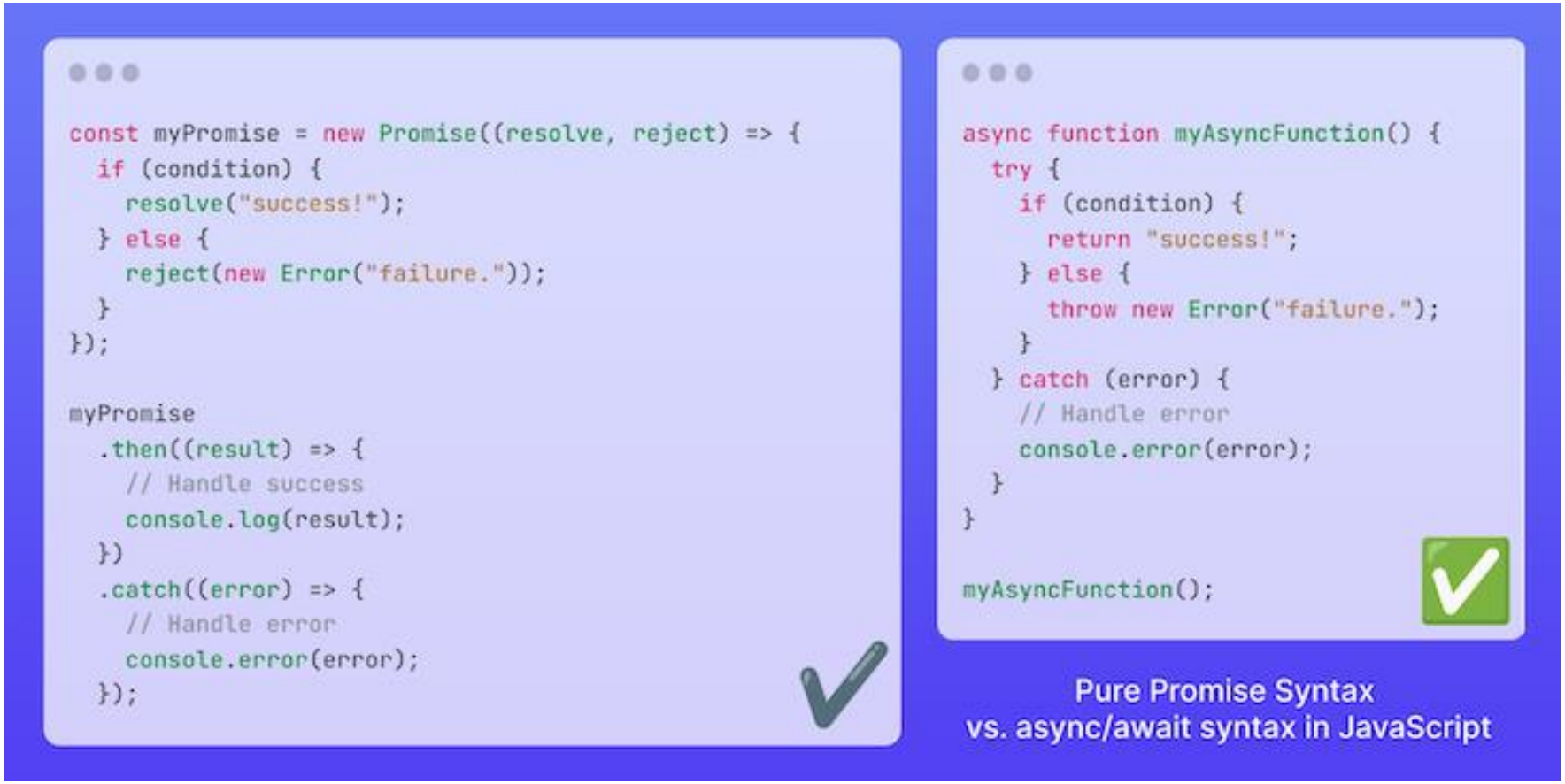
Use dependency injection



Nodejs Project Creation Best Practices



Use promises and async/await



Use promises and async/await

```
async function getData() {
return new Promise((resolve, reject) => {
setTimeout(() => {
const data = 'Some data';
resolve(data);
}, 1000);
});
}

async function main() {
const data = await getData();
console.log(data);
}
main();
```

Use promises and async/await

```
function getData() {
return new Promise((resolve, reject) => {
setTimeout(() => {
const data = 'Some data';
resolve(data);
}, 1000);
});
}

getData()
.then((data) => {
console.log(data);
})
.catch((error) => {
console.log(error);
});
```

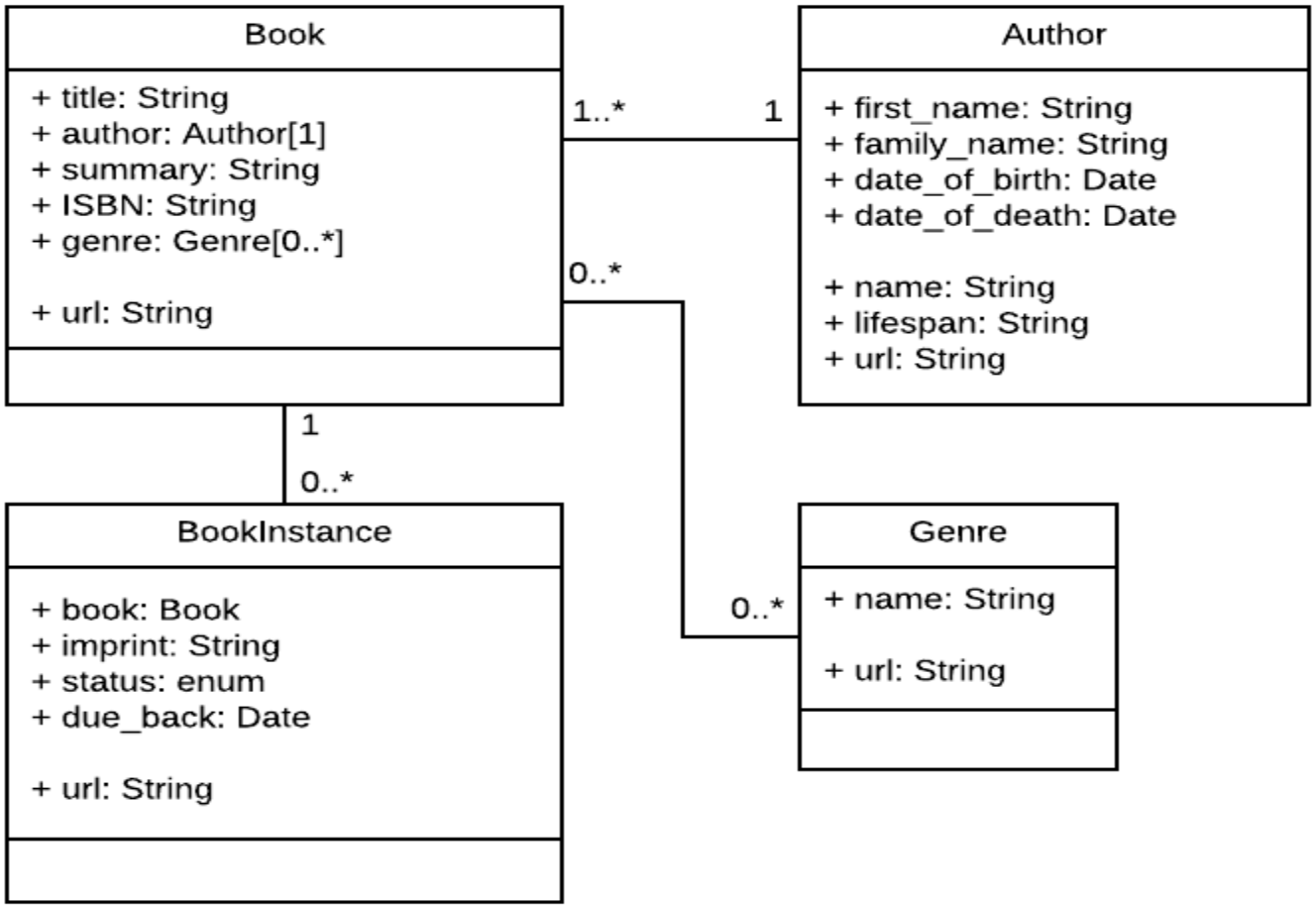
Use promises and async/await

```
function getData() {
return new Promise((resolve, reject) => {
setTimeout(() => {
const data = 'Some data';
resolve(data);
}, 1000);
});
}

getData()
.then((data) => {
console.log(data);
})
.catch((error) => {
console.log(error);
});
```

Use promises and async/await

```
function getData(callback) {
setTimeout(() => {
const data = 'Some data';
callback(data);
}, 1000);
}

getData((data) => {
console.log(data);
});
```

Design Schema



Design Schema

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const BookSchema = new Schema({
  title: { type: String, required: true },
  author: { type: Schema.Types.ObjectId, ref: "Author", required: true },
  summary: { type: String, required: true },
  isbn: { type: String, required: true },
  genre: [{ type: Schema.Types.ObjectId, ref: "Genre" }],
});

// Virtual for book's URL
BookSchema.virtual("url").get(function () {
  // We don't use an arrow function as we'll need the this object
  return `/catalog/book/${this._id}`;
});

// Export model
module.exports = mongoose.model("Book", BookSchema);
```

Design Schema

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const AuthorSchema = new Schema({
  first_name: { type: String, required: true, maxLength: 100 },
  family_name: { type: String, required: true, maxLength: 100 },
  date_of_birth: { type: Date },
  date_of_death: { type: Date },
});

// Virtual for author's full name
AuthorSchema.virtual("name").get(function () {
  // To avoid errors in cases where an author does not have either a family name or first name
  // We want to make sure we handle the exception by returning an empty string for that case
  let fullname = "";
  if (this.first_name && this.family_name) {
    fullname = `${this.family_name}, ${this.first_name}`;
  }

  return fullname;
});

// Virtual for author's URL
AuthorSchema.virtual("url").get(function () {
  // We don't use an arrow function as we'll need the this object
  return `/catalog/author/${this._id}`;
});

// Export model
module.exports = mongoose.model("Author", AuthorSchema);
```

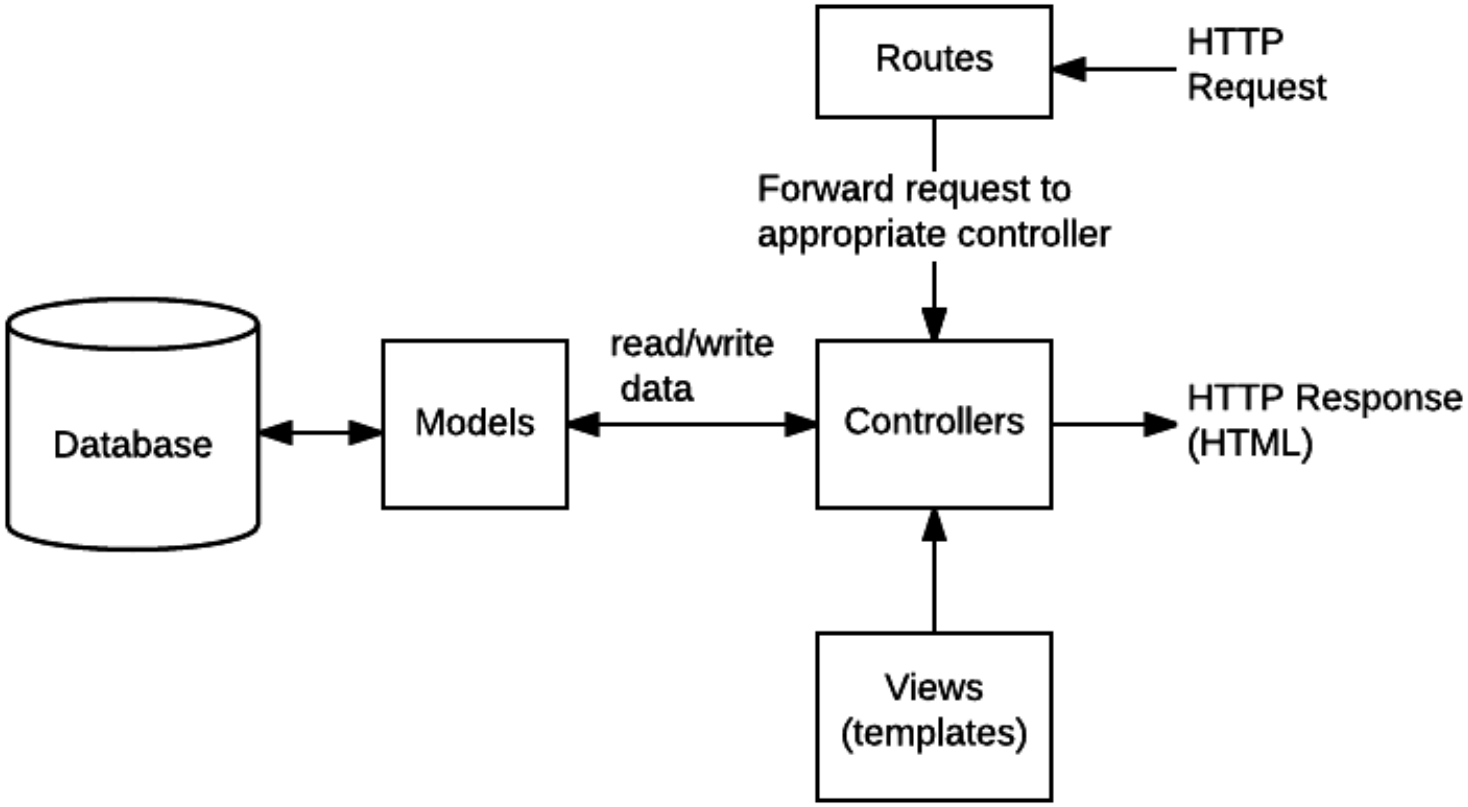
Design Schema

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const BookInstanceSchema = new Schema({
  book: { type: Schema.Types.ObjectId, ref: "Book", required: true }, // reference to the
associated book
  imprint: { type: String, required: true },
  status: {
    type: String,
    required: true,
    enum: ["Available", "Maintenance", "Loaned", "Reserved"],
    default: "Maintenance",
  },
  due_back: { type: Date, default: Date.now },
});

// Virtual for bookinstance's URL
BookInstanceSchema.virtual("url").get(function () {
  // We don't use an arrow function as we'll need the this object
  return `/catalog/bookinstance/${this._id}`;
});

// Export model
module.exports = mongoose.model("BookInstance", BookInstanceSchema);
```

Design Schema

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const GenreSchema = new Schema({
  name: { type: String, required: true, minLength: 3, maxLength: 100 },
});

// Virtual for this genre instance URL.
GenreSchema.virtual("url").get(function () {
  return "/catalog/genre/" + this._id;
});

// Export model.
module.exports = mongoose.model("Genre", GenreSchema);
```

API Design

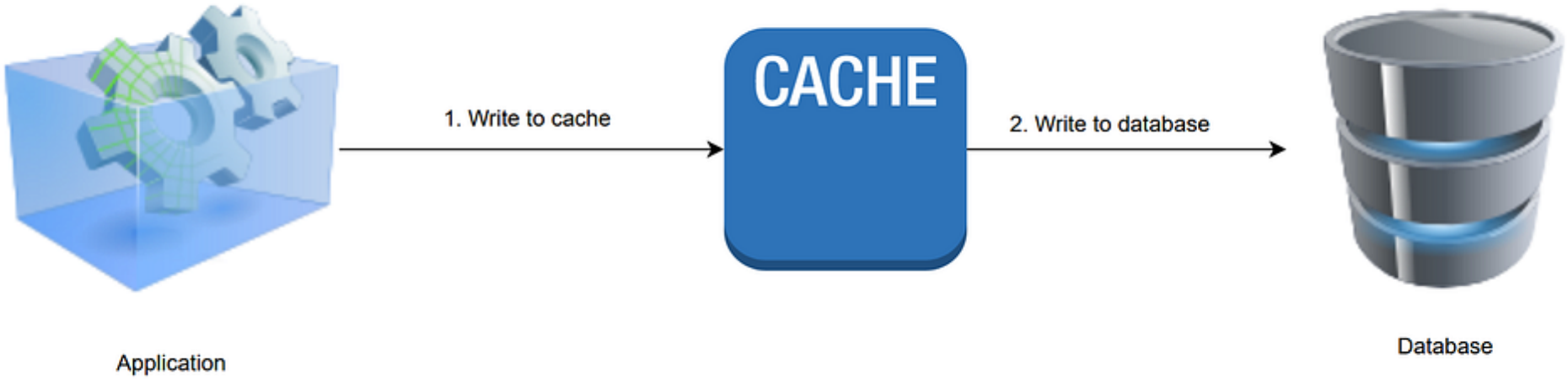

Routes
HTTP
Request
Forward request to
appropriate controller
Database
Models
read/write
data
Controllers
HTTP Response
(HTML)
Views
(templates)

Nodejs Profiling

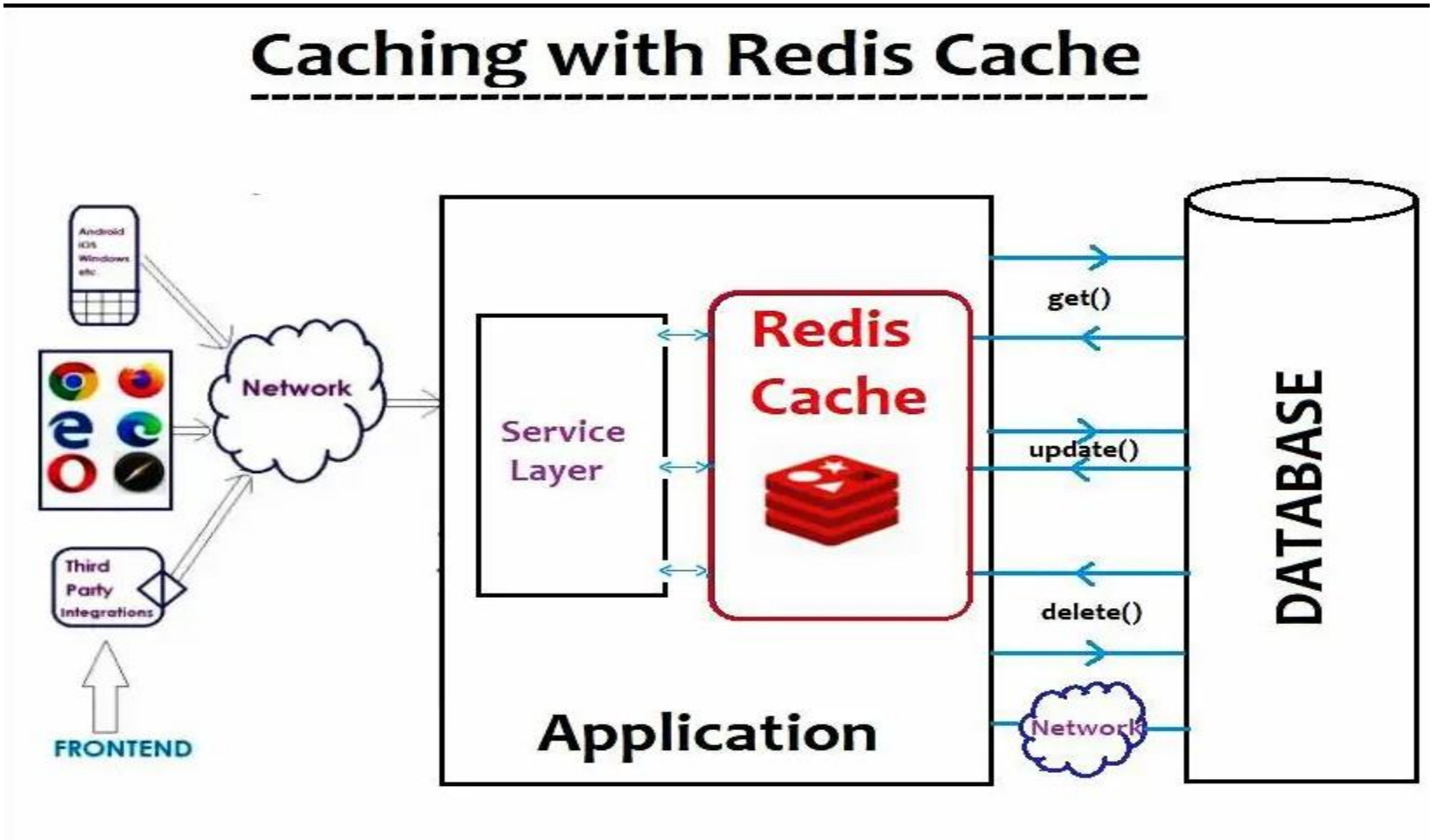
- Create config folder with production.json
- set NODE_ENV=production

```
E:\mugrugappanodejstrainingv1\registrationapi>set NODE_ENV=production

E:\mugrugappanodejstrainingv1\registrationapi>npm start

> registrationapi@1.0.0 start
> nodemon app.js

[nodemon] 3.1.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
```

Nodejs Caching



Nodejs Caching

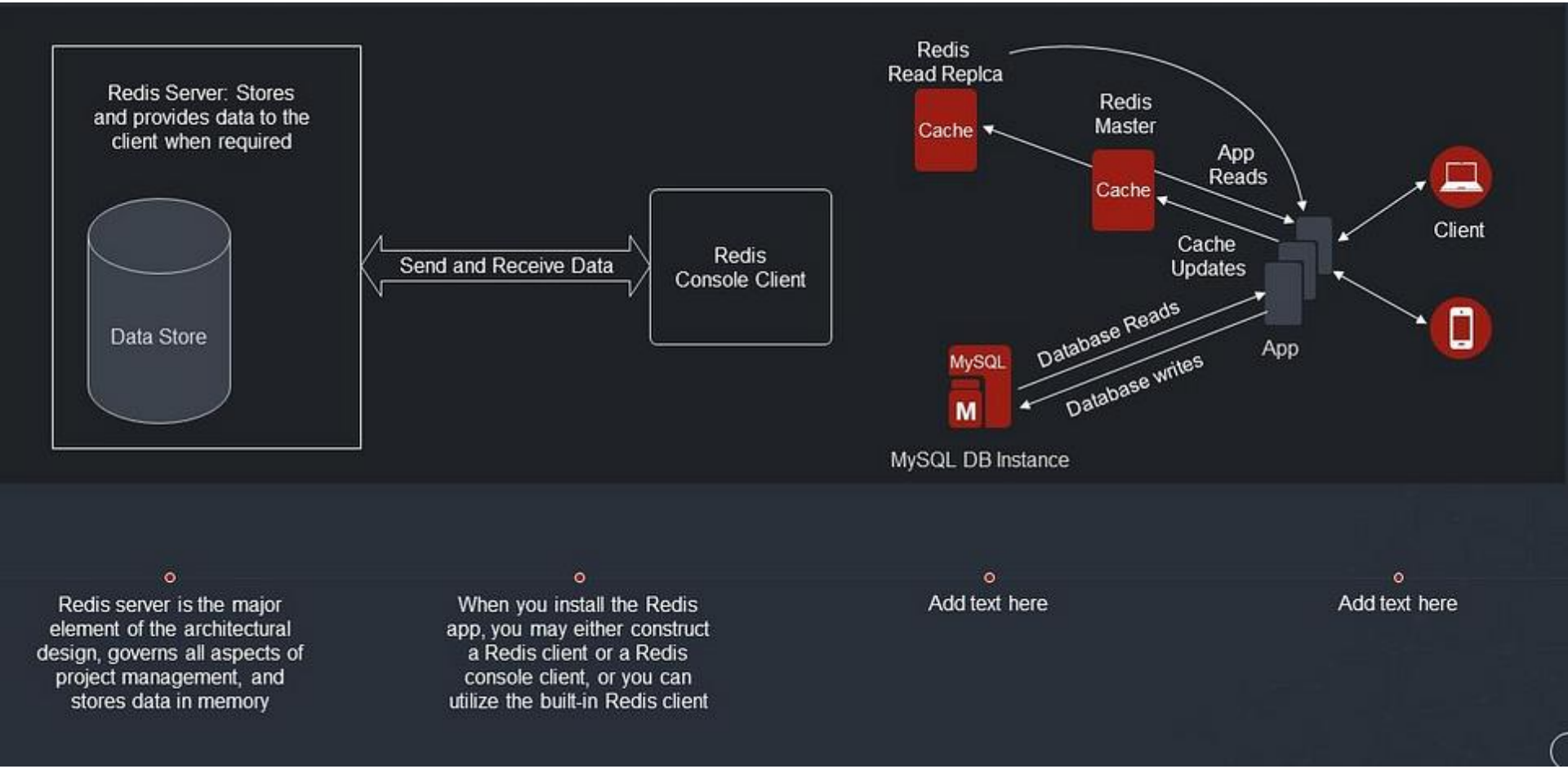


Redis client test

Administrator: C:\Windows\system32\cmd.exe - "node" "C:\Users\Dell\AppData\Roaming\npm\\node_modules\redis-cli\bin\rdcli" -h 18.234.253.37 -p 6379

```
C:\Windows\System32>f:

F:\Local disk\Nodejs>npm install -g redis-cli
npm WARN deprecated mkdirp@0.5.1: Legacy versions of mkdirp are no longer supported. Plea
se update to mkdirp 1.x. (Note that the API surface has changed to use Promises in 1.x.)

added 55 packages in 24s

4 packages are looking for funding
  run `npm fund` for details

F:\Local disk\Nodejs>rdcli
127.0.0.1:6379>
Abort!

F:\Local disk\Nodejs>rdcli -h 18.234.253.37 -p 6379
18.234.253.37:6379>
```

Nodejs Caching



Redis server is the major element of the architectural design, governs all aspects of project management, and stores data in memory

When you install the Redis app, you may either construct a Redis client or a Redis console client, or you can utilize the built-in Redis client

Add text here

Add text here

Nodejs Caching

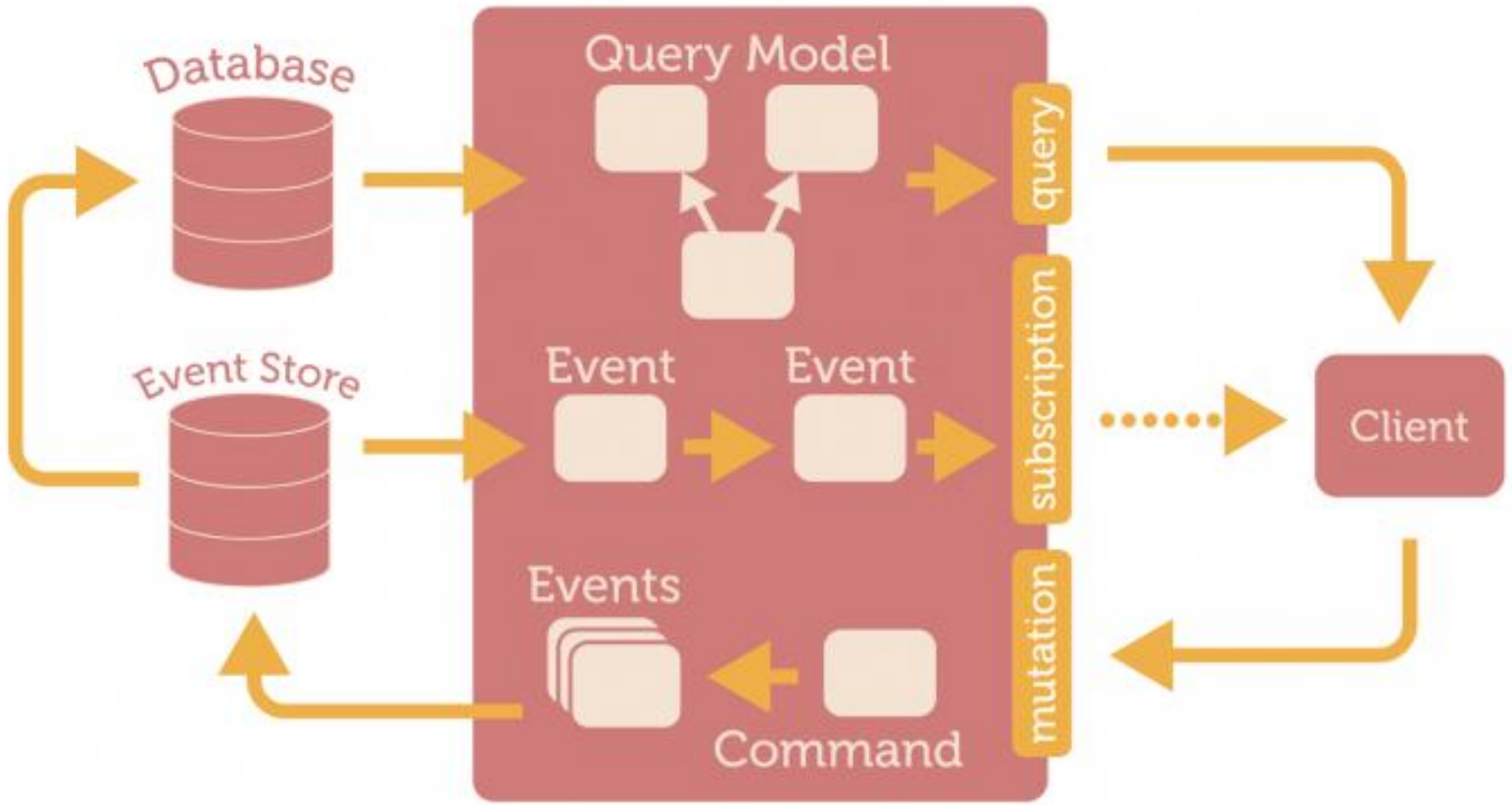
	Node Caching	Redis Caching
Data Storage	In-memory cache within the Node.js application	Separate in-memory or disk-based data store
Scalability	Limited by available memory on the Node.js server	Highly scalable, can handle large data volumes
Persistence	Cache is lost if the Node.js process restarts	Data can be persisted even after restarts
Expiration	Manual handling required for cache expiration	Built-in expiration mechanisms for cache entries
Distributed Caching	Not inherently designed for distributed setups	Designed for distributed cache architectures
Advanced Features	Limited functionality and features	Rich set of features like pub/sub, transactions, etc.
Use Cases	Small-scale applications with limited data	Large-scale applications requiring high performance and scalability
Integration	Directly integrated into Node.js application code	Requires separate Redis server and client library

Graphql

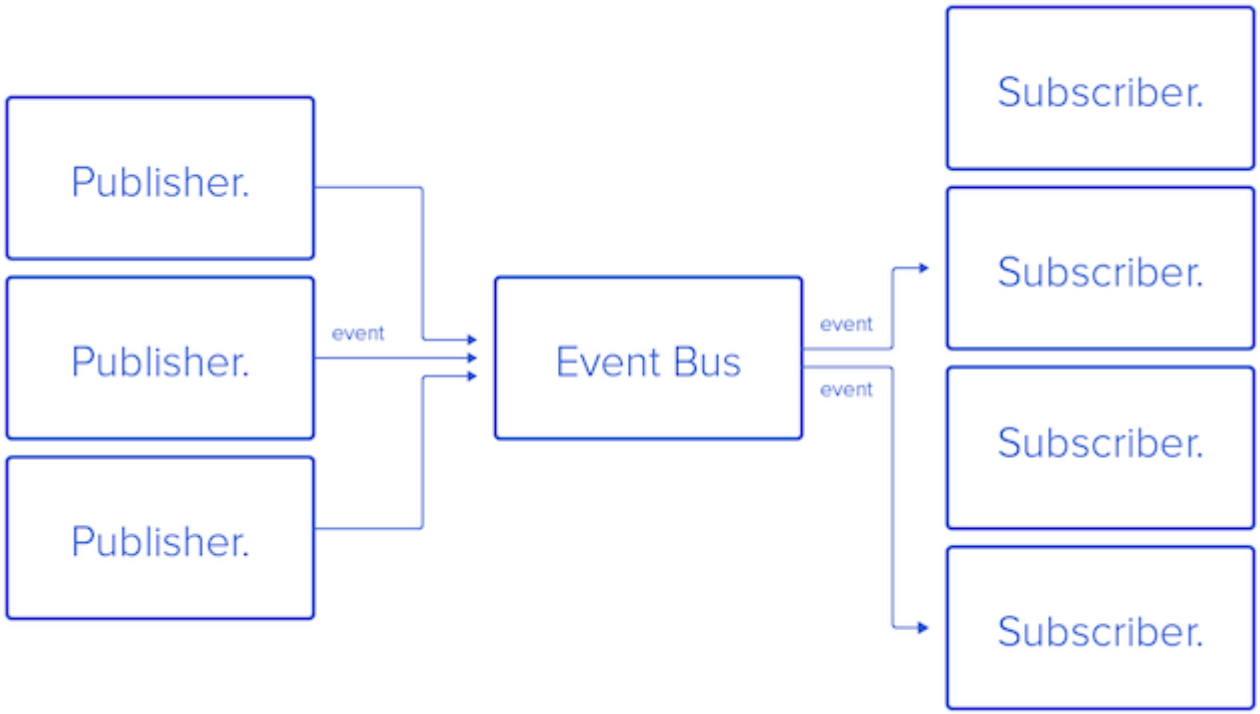
Administrator: Node.js command prompt

```
E:\mugrugappanodejstrainingv1\registrationapi>npm install --force
npm WARN using --force Recommended protections disabled.
npm WARN ERESOLVE overriding peer dependency
npm WARN While resolving: express-graphql@0.12.0
npm WARN Found: graphql@16.8.1
npm WARN node_modules/graphql
npm WARN   graphql@"^16.6.0" from the root project
npm WARN
npm WARN Could not resolve dependency:
npm WARN peer graphql@"^14.7.0 || ^15.3.0" from express-graphql@0.12.0
npm WARN node_modules/express-graphql
npm WARN   express-graphql@"^0.12.0" from the root project
npm WARN
npm WARN Conflicting peer dependency: graphql@15.8.0
npm WARN node_modules/graphql
npm WARN   peer graphql@"^14.7.0 || ^15.3.0" from express-graphql@0.12.0
npm WARN   node_modules/express-graphql
npm WARN     express-graphql@"^0.12.0" from the root project
npm WARN deprecated express-graphql@0.12.0: This package is no longer maintained. We reco
mmend using `graphql-http` instead. Please consult the migration document https://github.
com/graphql/graphql-http#migrating-express-grpahql.
```

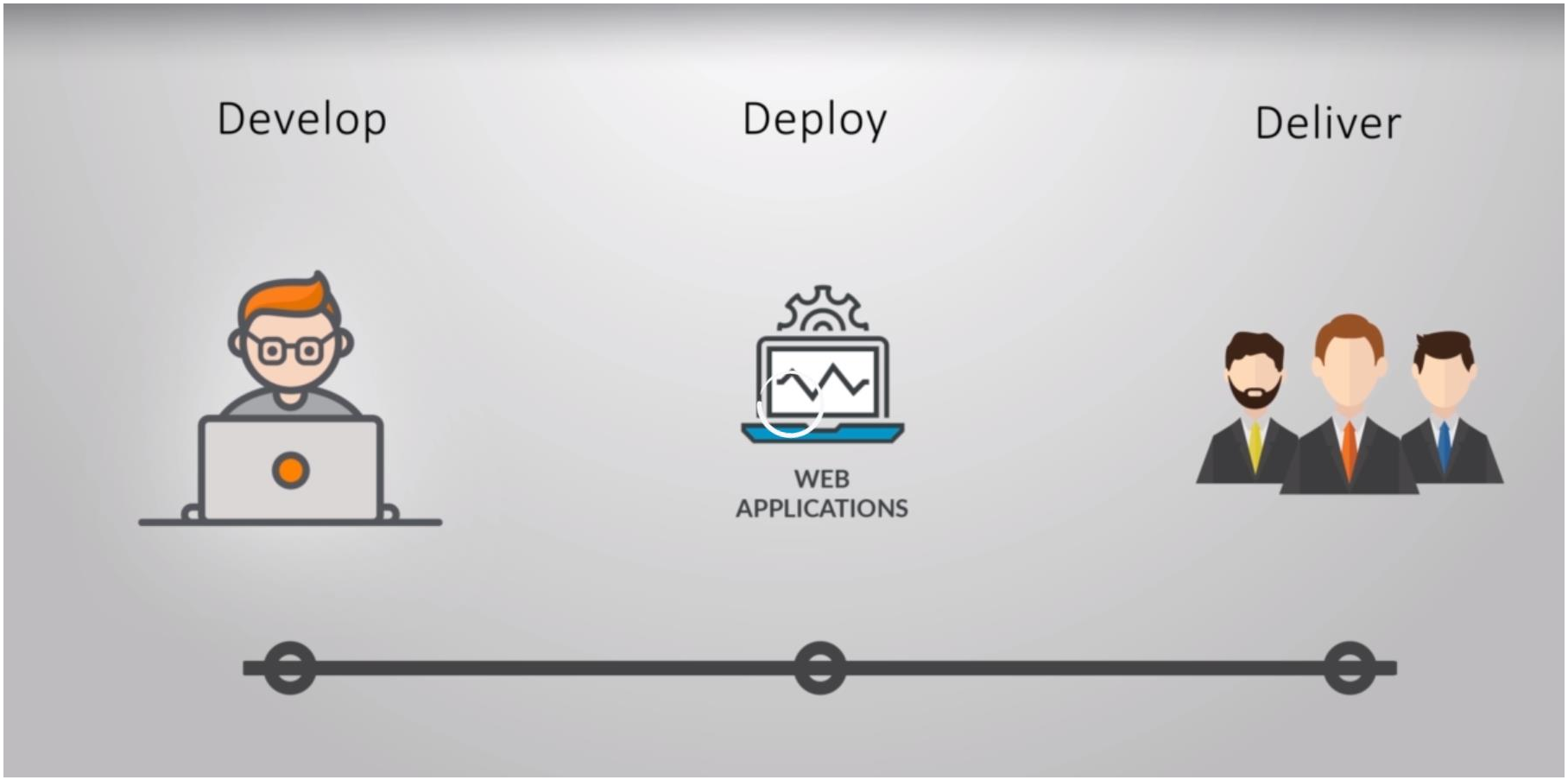
How do GraphQL subscriptions work?



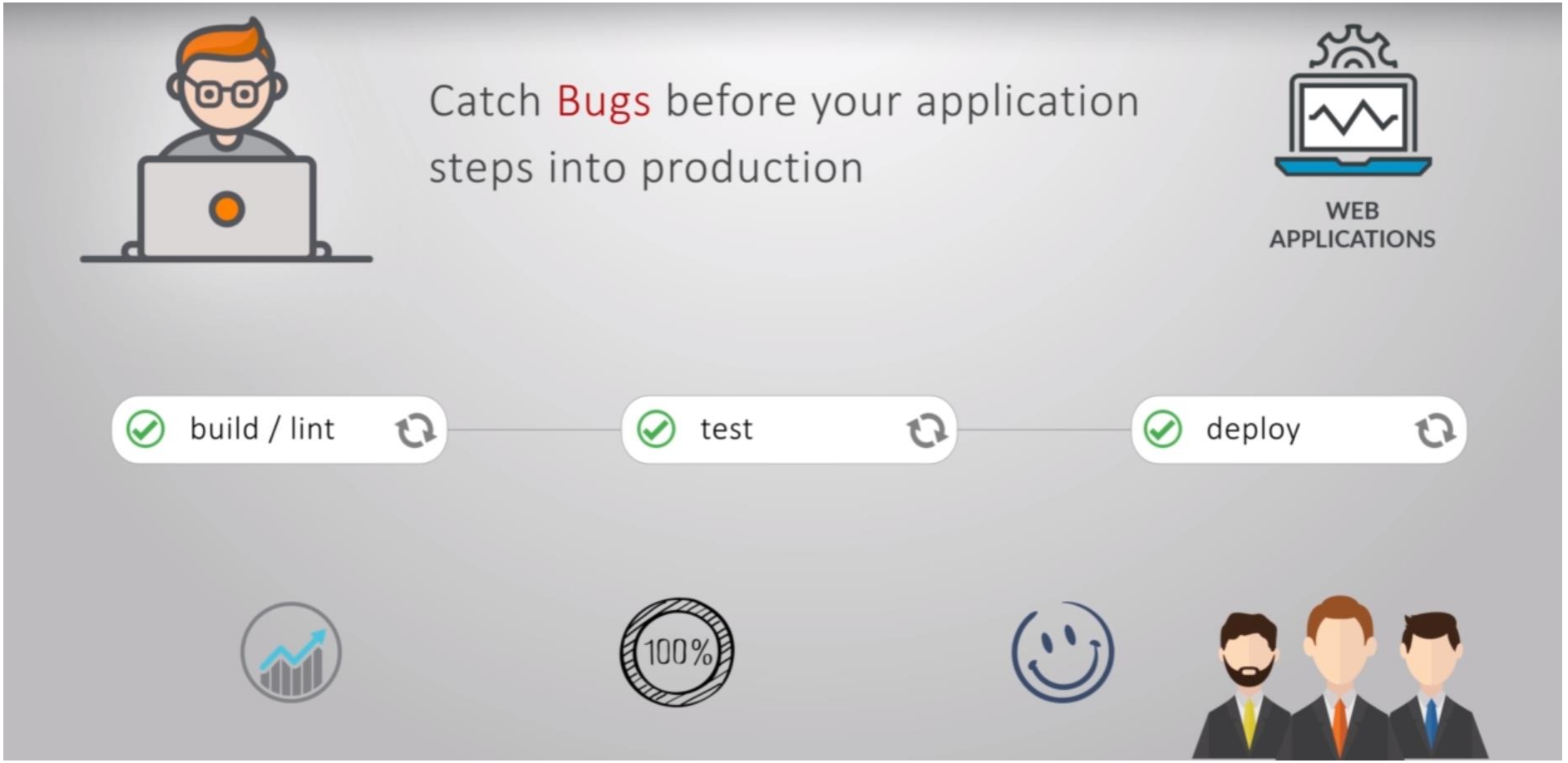
How do GraphQL subscriptions work?



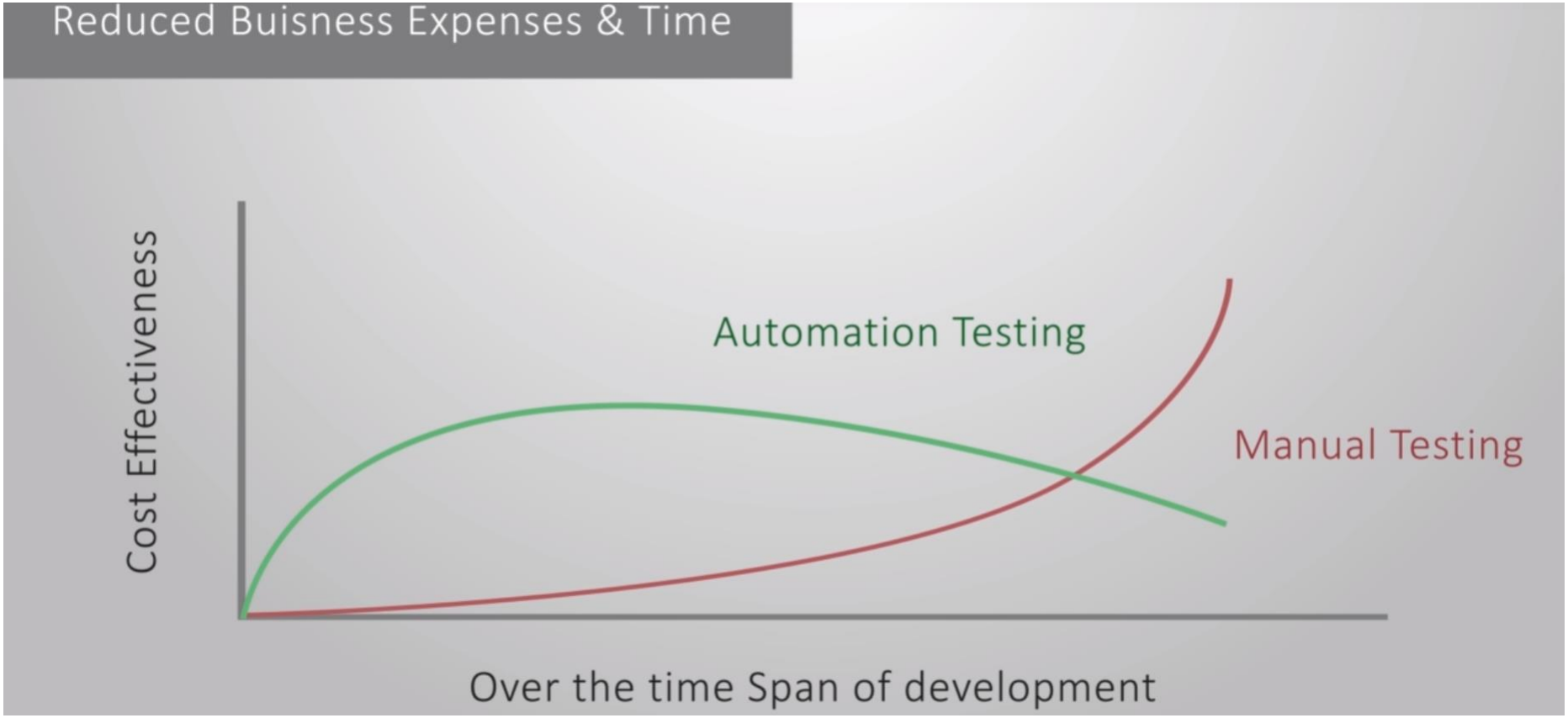
Automation Testing



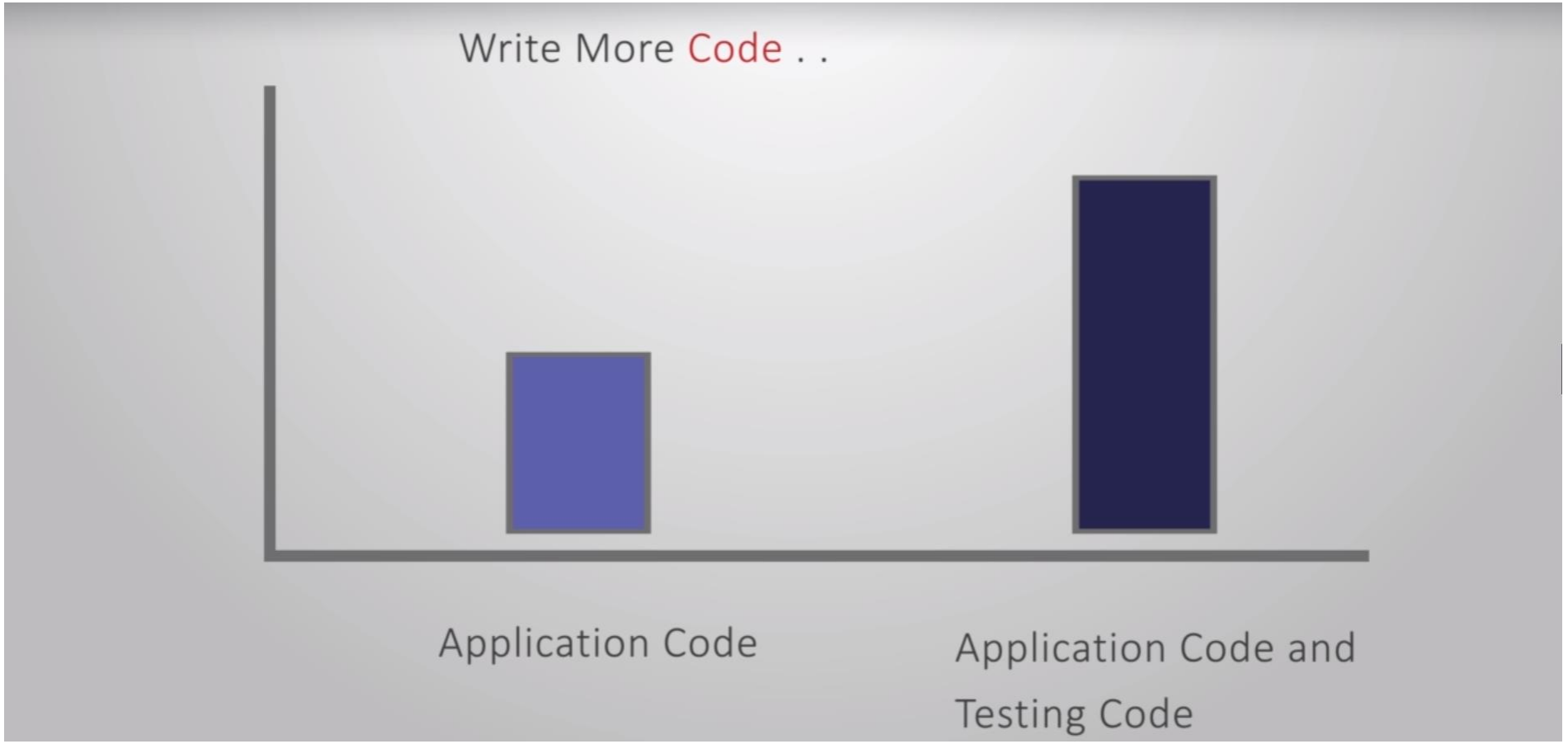
Automation Testing



Automation Testing



Automation Testing



Automation Testing

Automation Testing

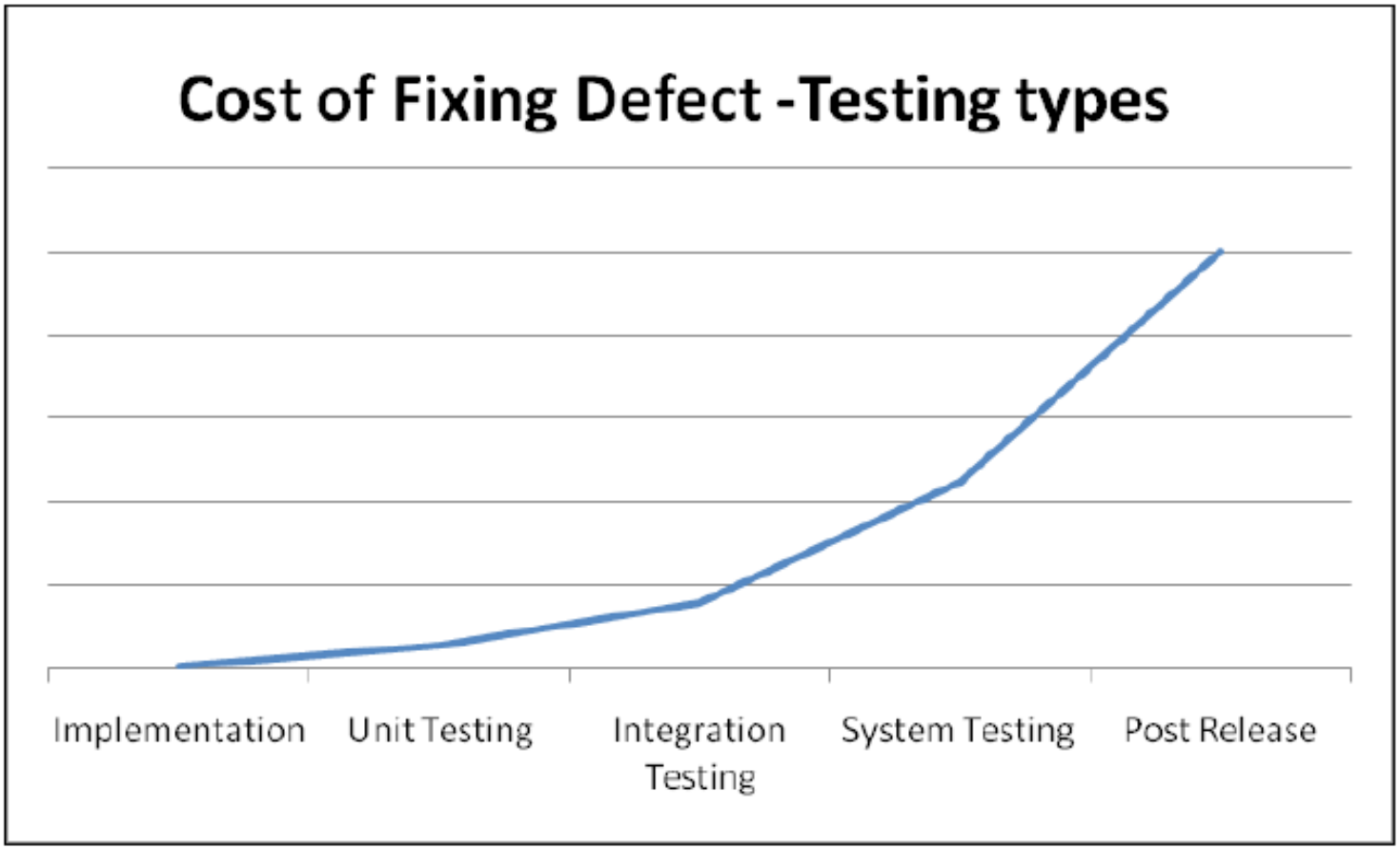
Testing Javascript

- JavaScript Unit Testing is a testing method to test code written for a web page or web application module.
- These unit tests are then organized in the test suite.
- Every suite contains several tests designed to be executed for a separate module.

Challenges in JavaScript Unit Testing

- Many other languages support unit testing in browsers, in the stable as well as in runtime environment but JavaScript can not.
- Some JavaScript are written for a web application may have multiple dependencies
- JavaScript is good to use in combination with HTML and CSS rather than on the web
- Difficulties with page rendering and DOM manipulation
- Sometimes we find the error message on your screen regarding such as ‘Unable to load example.js’ or any other JavaScript error regarding version control, these vulnerabilities comes under Unit Testing JavaScript.

Types of Testing



Best JavaScript Unit Testing Frameworks

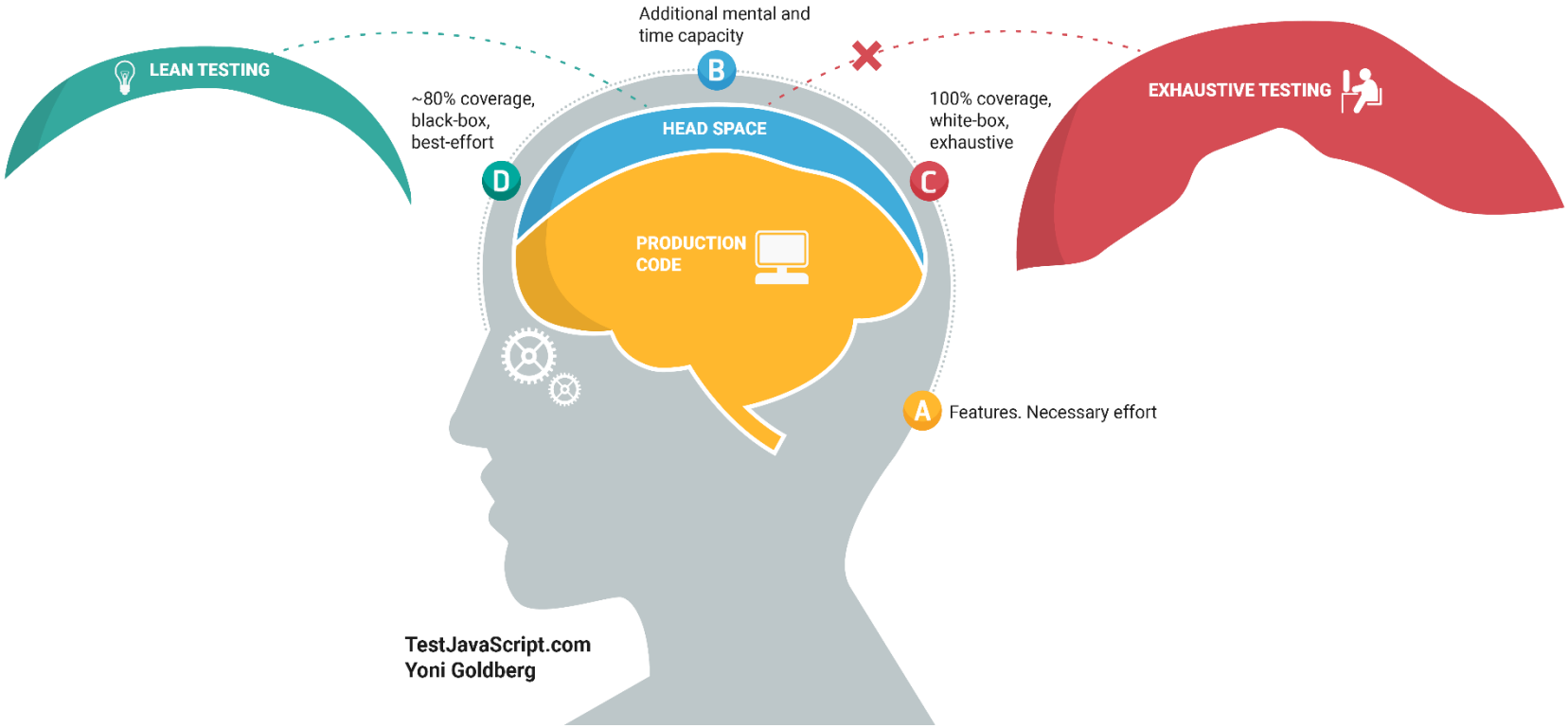
- Unit.js (Assertion Library)
- Qunit (Client and Server Side Unit Testing)
- Jasmine(behavior-driven development framework to unit test JavaScript)
- Karma (Karma is an open source productive testing environment)
- Mocha(Mocha runs on Node.js and in the browser. Mocha performs asynchronous Testing)
- Jest (Jest is used by Facebook so far to test all of the JavaScript code. It provides the ‘zero-configuration’ testing experience)

Best JavaScript Unit Testing Frameworks

- AVA (AVA is simple JavaScript Unit Testing Framework. Tests are being run in parallel and serially. Parallel tests run without interrupting each other)

Testing Strategy

- The Golden Rule: Design for lean testing



Testing Strategy

- The Test Anatomy
  - (1) What is being tested?
  - (2) Under what circumstances and scenario? For example, no price is passed to the method
  - (3) What is the expected result? For example, the new product is not approved
  - ❌ Otherwise: A deployment just failed, a test named “Add product” failed. Does this tell you what exactly is malfunctioning?

Testing Strategy

- Structure tests by the AAA pattern
  - Structure your tests with 3 well-separated sections Arrange, Act & Assert (AAA). Following this structure guarantees that the reader spends no brain-CPU on understanding the test plan:
  - 1st A - Arrange: All the setup code to bring the system to the scenario the test aims to simulate. This might include instantiating the unit under test constructor, adding DB records, mocking/stubbing on objects and any other preparation code
  - 2nd A - Act: Execute the unit under test. Usually 1 line of code
  - 3rd A - Assert: Ensure that the received value satisfies the expectation. Usually 1 line of code
  - ❌ Otherwise: Not only do you spend hours understanding the main code, but what should have been the simplest part of the day (testing) stretches your brain

Testing Strategy

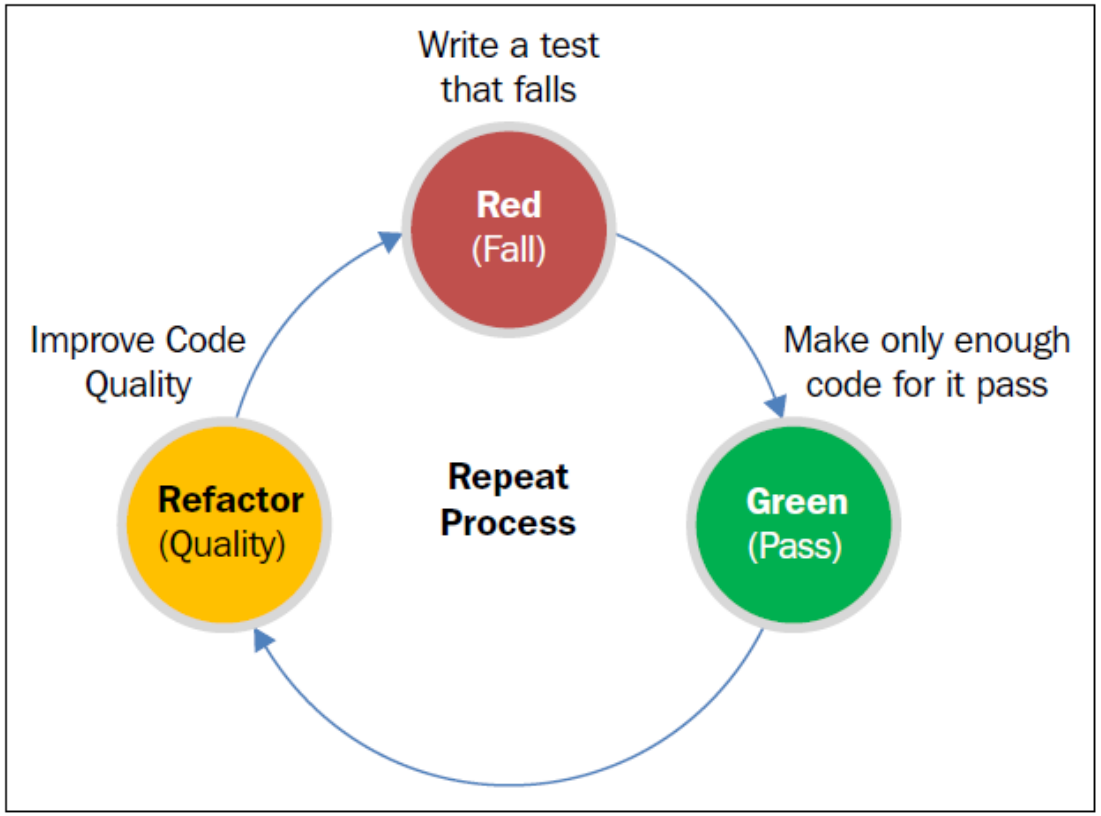
- Describe expectations in a product language: use BDD-style assertions
  - Do: Coding your tests in a declarative-style allows the reader to get the grab instantly without spending even a single brain-CPU cycle.
  - When you write imperative code that is packed with conditional logic, the reader is forced to exert more brain-CPU cycles.
  - In that case, code the expectation in a human-like language, declarative BDD style using expect or should and not using custom code.
  - If Chai & Jest doesn't include the desired assertion and it's highly repeatable, consider extending Jest matcher (Jest) or writing a custom Chai plugin
  - ❌ Otherwise: The team will write less tests and decorate the annoying ones with .skip()

Testing Strategy

- Choose the right test doubles: Avoid mocks in favor of stubs and spies
- Don’t “foo”, use realistic input data
- Test many input combinations using Property-based testing
- If needed, use only short & inline snapshots
- Tag your tests

TDD Mindset

- Refer TDD ppt



Testing with Mocha.js

- Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun.
- Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

Testing with Mocha.js

- Mocha.js is an open-source JavaScript test framework that runs on Node.js and in the browser.
- It’s designed for testing both synchronous and asynchronous code with a very simple interface.
- Mocha.js runs tests serially to deliver flexible and accurate reporting while mapping uncaught exceptions to their corresponding test cases.

What is Mocha.js used for?

- Mocha.js provides functions that execute in a specific order and logs the results in the terminal window.
- It also cleans the state of the software being tested to ensure that test cases run independently of each other.
- While it can be used with most assertion libraries, Mocha.js is commonly used with Chai, a popular assertion library for Node.js and the browser.

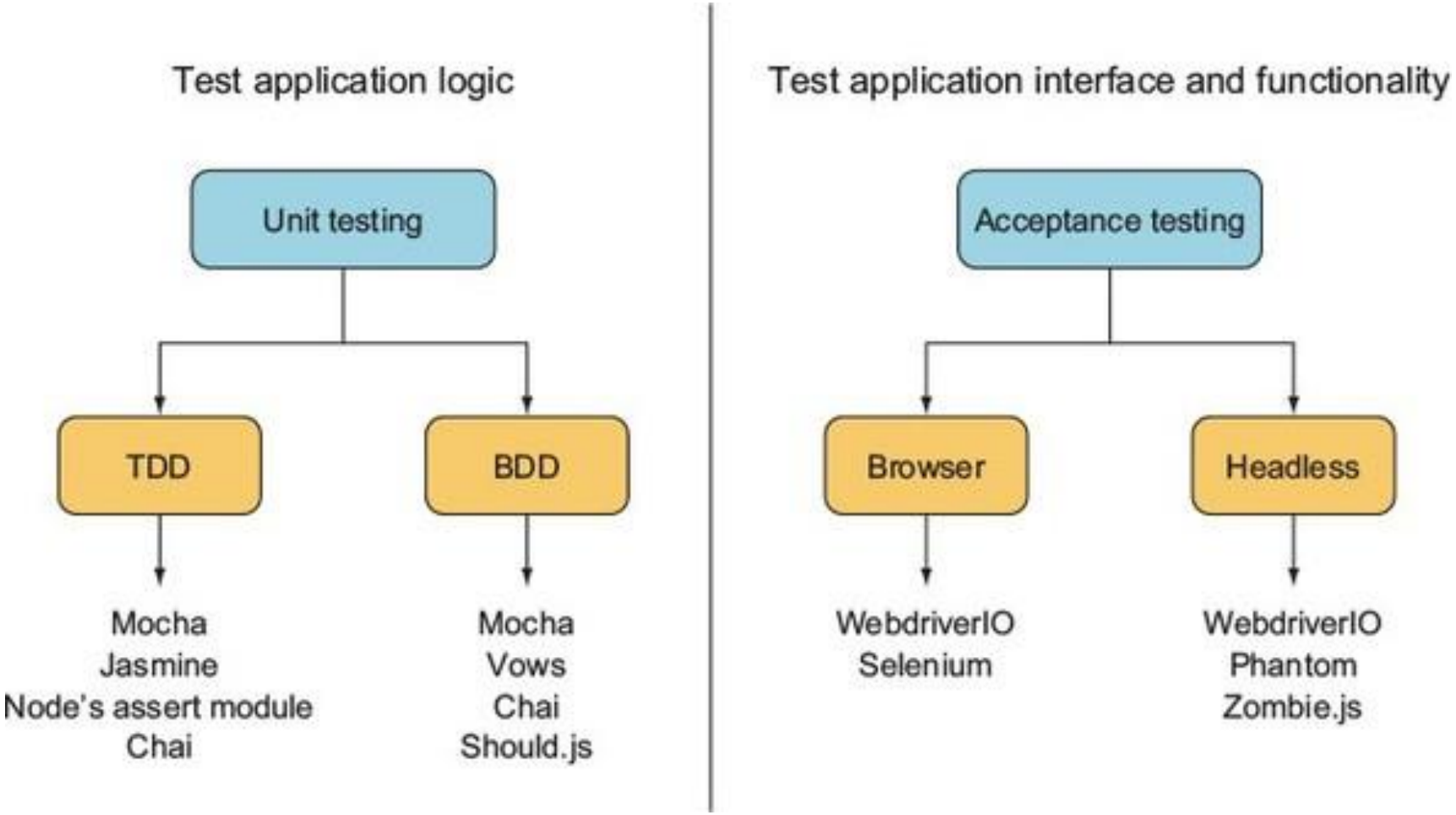
Is Mocha.js a BDD tool?

- Behavior-driven development (BDD) aims to help developers build software that is predictable, resilient to changes, and not error-prone.
- It evolved from test-driven development (TDD), which requires you to:
  - Write tests for the required software functionality
  - Run the tests for the software functionality
  - Implement the software functionality
  - Fix bugs and refactor until all tests pass
  - Repeat the cycle for any new functionality

Is Mocha.js a BDD tool?

- The main difference between TDD and BDD is that BDD calls for writing test cases in a shared language.
- This is to simplify communication between technical and nontechnical stakeholders, such as developers, QA teams, and business leaders.

Test Framework Overview



Using Mocha.js and Chai.js

- Writing tests often requires using an assertion library.
- Mocha.js does not discriminate, regardless of which assertion library you choose to use.
- If you’re using Mocha in a Node.js environment, you can use the built-in assert module as your assertion library.
- However, there are more extensive assertion libraries you can use, such as Chai, Expect.js, Should.js, etc.

Mocha Installation

- Install with npm globally:
  - $ npm install --global mocha
- or as a development dependency for your project:
  - $ npm install --save-dev mocha

Create Test Project

- npm init
- Add mocha dependency
- npm i --save-dev mocha
- To create chai library
- npm i --save-dev chai

Mocha Structure

- Mocha gives us the ability to describe the features that we are implementing by giving us a describe function that encapsulates our expectations.
- The first argument is a simple string that describes the feature, while the second argument is a function that represents the body of the description.

```
describe("Color Code Converter", function() {
  // specification code
});
```

Mocha Structure

- describe(): Used to group one or more test cases and can be nested.
- it(): the test case is laid down here
- before(): It’s a hook to run before the first it() or describe();
- beforeEach(): It’s a hook to run before each it() or describe();
- after(): It’s a hook to run after it() or describe();
- afterEach(): It’s a hook to run after each it() or describe();

What Is an Assert

- Chai provides the following assertion styles:
  - Assert style
  - Expect style
  - Should Style

Using the Mocha.js BDD interface

```
// begin a test suite of one or more tests
describe('#sum()', function() {
  // add a test hook
  beforeEach(function() {
    // ...some logic before each test is run
  })
    // test a functionality
  it('should add numbers', function() {
    // add an assertion
    expect(sum(1, 2, 3, 4, 5)).to.equal(15);
  })
  // ...some more tests
})
```

Testing Async

- The Mocha test framework has excellent support for async tests.
- There are 3 ways to structure async tests with Mocha:
  - async/await
  - promise chaining
  - Callbacks (done)

  [Error: Timeout of 2000ms exceeded. For async tests and hooks, ensure "done()" is called]

Timeout

- The default timeout for mocha tests is 2000 ms. There are multiple ways to change this:

```
describe("testing promises", function () {
    this.timeout(5000);
    it('test1', function(){ ... });
});

describe("testing promises", function () {
    it('test1', function(){
        this.timeout(5000);
     });
});
```

Timeout

```
describe("testing promises", () => {
    it('test1', () => {          ...
     }).timeout(5000);
});
```

Hooks

- Mocha is a feature-rich JavaScript test framework for Node.js.
- Mocha provides several built-in hooks that can be used to set up preconditions and clean up after your tests.
- The four most commonly used hooks are:
  - **before(), after(), beforeEach(), and afterEach().**

Scoping in Hooks

- before(name, fn)
  - name: Optional string for description
  - fn: Function to run once before the first test case
- after(name, fn)
  - name: Optional string for description
  - fn: Function to run once after the last test case
- beforeEach(name, fn)
  - name: Optional string for description
  - fn: Function to run before each test case

Scoping in Hooks

- afterEach(name, fn)
  - name: Optional string for description
  - fn: Function to run after each test case

PENDING TESTS

- “Pending” — as in “someone should write these test cases eventually” — **test-cases are those without a callback:**

```
describe('Array', function() {
  describe('#indexOf()', function() {
    // pending test below
    it('should return -1 when the value is not present');
  });
});
```

```
Pending test 'should return -1 when the value is not present'
```

ARROW FUNCTIONS

- Passing arrow functions (aka “lambdas”) to Mocha is discouraged.
- Lambdas lexically bind this and cannot access the Mocha context.
- For example, the following code will fail:

```
describe('my suite', () => {
  it('my test', () => {
    // should set the timeout of this test to 1000 ms; instead will fail
    this.timeout(1000);
    assert.ok(true);
  });
});
```

Parameterized Test

- Given Mocha's use of function expressions to define suites and test cases, it's straightforward to generate your tests dynamically.
- No special syntax is required — plain JavaScript can be used to achieve functionality like "parameterized" tests.

BDD Style Assertions(Chai)

- The assert style is very similar to node.js' included assert module, with a bit of extra sugar.
- Of the three style options, assert is the only one that is not chainable.
- assert(expression, message)
  - @param { Mixed } expression to test for truthiness
  - @param { String } message to display on error

BDD Style Assertions(Chai)

- .fail([message])
- .fail(actual, expected, [message], [operator])
  - @param { Mixed } actual
  - @param { Mixed } expected
  - @param { String } message
  - @param { String } operator

BDD Style Assertions(Chai)

- .isOk(object, [message])
  - @param { Mixed } object to test
  - @param { String } message

BDD Style Assertions(Chai)

- .isNotOk(object, [message])
  - @param { Mixed } object to test
  - @param { String } message

BDD Style Assertions(Chai)

- .equal(actual, expected, [message])
  - @param { Mixed } actual
  - @param { Mixed } expected
  - @param { String } message

BDD Style Assertions(Chai)

- .notEqual(actual, expected, [message])
  - @param { Mixed } actual
  - @param { Mixed } expected
  - @param { String } message

BDD Style Assertions(Chai)

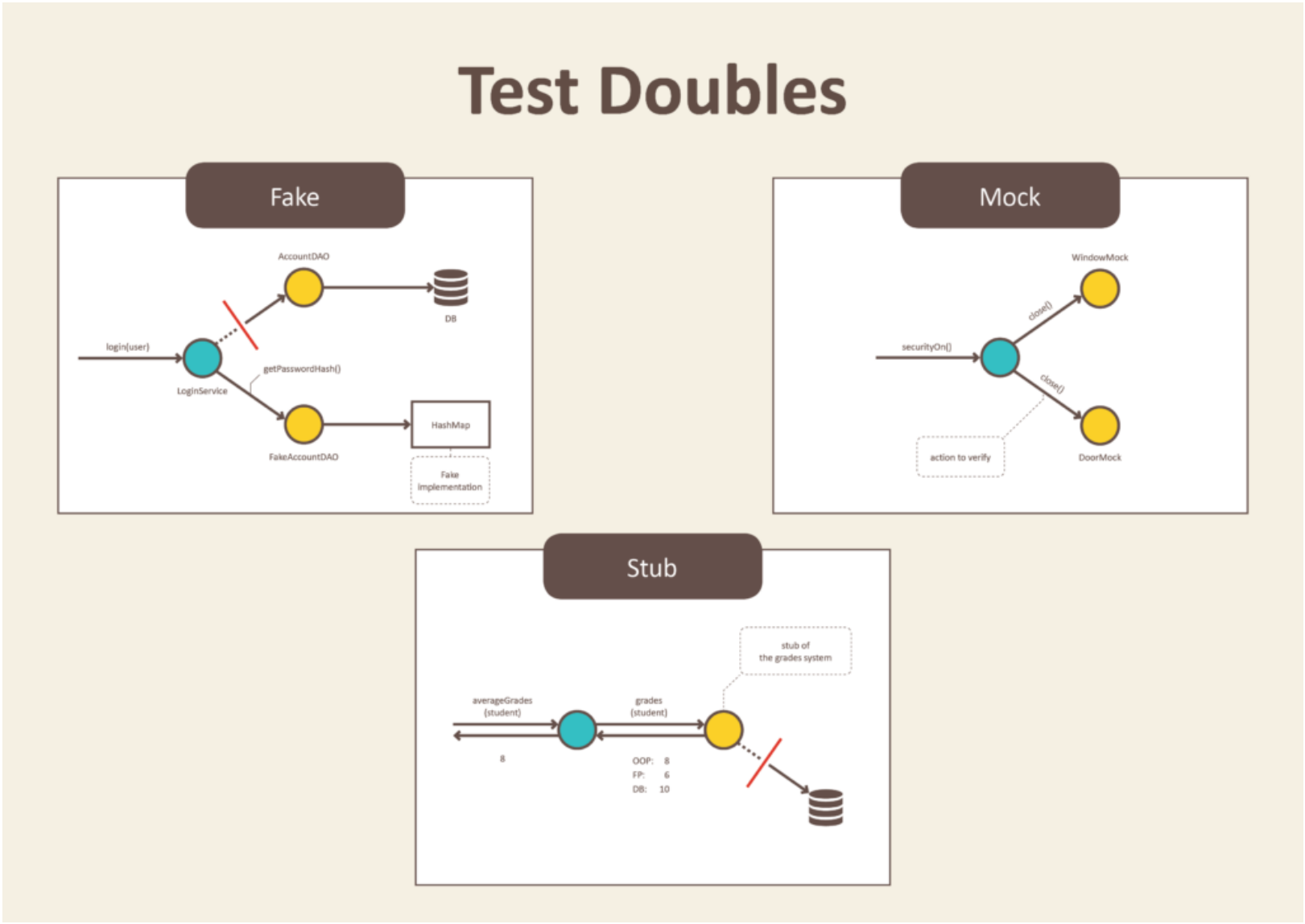
- .deepEqual(actual, expected, [message])
  - @param { Mixed } actual
  - @param { Mixed } expected
  - @param { String } message

  - Refer https://www.chaijs.com/api/assert/

Chai Null Test

- Refer assert,should and expect with null options
- (G:\Local disk\Nodejs\mochaws\day1\tests\test\chai)

Spy, Stubs and Mocks



Working with Spys

- A spy is an object in testing that tracks calls made to a method.
- By tracking its calls, we can verify that it is being used in the way our function is expected to use it.
- True to its name, a spy gives us details about how a function is used.
- How many times was it called? What arguments were passed into the function?

Working with Spys

- Let's consider a function that checks if a user exists,
- Create one in our database if it doesn’t.
- We may stub the database responses and get the correct user data in our test.
- But how do we know that the function is actually creating a user in cases we don't have pre-existing user data?
- With a spy, we will observe how many times the create-user function is called and be certain.

Why Use Spies?

- Spies excel in giving insight into the behavior of the function we are testing.
- While validating the inputs and outputs of a test is crucial, examining how the function behaves can be crucial in many scenarios:
- When our function has side effects that are not reflected in its results, we should spy on the methods it uses.

Spies with Anonymous Functions

- G:\Local disk\Nodejs\mochaws\day1\src\spy

Spies as Function or Method Wrappers

- G:\Local disk\Nodejs\mochaws\day1\tests\test\spy\functionindextest.js

Mock

- Test "mocks" are objects that replace real objects while simulating their functions.
- A mock also has expectations about how the functions being tested will be used.
- In some unit test cases we may want to combine the functionality of spies, to observe a method's behavior under call.
- Stubs to replace a method's functionality, in ensuring that we do not make an actual function call but are still able to monitor the behavior of our target function accordingly.
- In such a case, we can use mocks.

Mock

- Mocks combine the functionality of both spies and stubs, which means that they replace the target function but at the same time provide us with the ability to observe how the function was called.
- For example, let's consider a function that communicates with a database to save a contact's details.
- With a mock, instead of a real database, our function will hit a fake database object.
- We can determine what kind of response it will give.
- We'll also state how many times the database should be called and the arguments it should be called with.
- Finally, as part of the test, we verify that our database mock was called the exact amount of times we expected.
- We also check that it got called with only the arguments our function supposed to provide it.

Why Use Mocks?

- Mocks are useful when validating how an external dependency is used within a function. Use mocks when you are in interested in:
  - Confirming that your external dependency is used at all
  - Verifying that your external dependency is used correctly
  - Ensuring that your function can handle different responses from external dependencies.

Code Coverage Istanbul

- Step 1: Install NYC
- NYC is Istanbul's command line utility. The first thing you'll need to do is install NYC:
  - npm install nyc --save-dev
  - npm install nyc --save-dev
- Step 2: Use NYC to Call Mocha
- With Istanbul installed, prepend your existing Mocha command with the NYC binary.
- For instance, your package.json would look like the following:
- JavaScript
- {
- "scripts": {
- "test": "nyc mocha"
- }
- }

Code Coverage - Istanbul

- Istanbul is a test coverage tool that works with many different frameworks.
- It tracks which parts of your code are executed by your unit tests.
- Thus, you can use Istanbul to view and see coverage gaps, or you can integrate it integrated into your CI pipeline to enforce coverage levels.
- Ultimately, Istanbul enables data-driven testing.

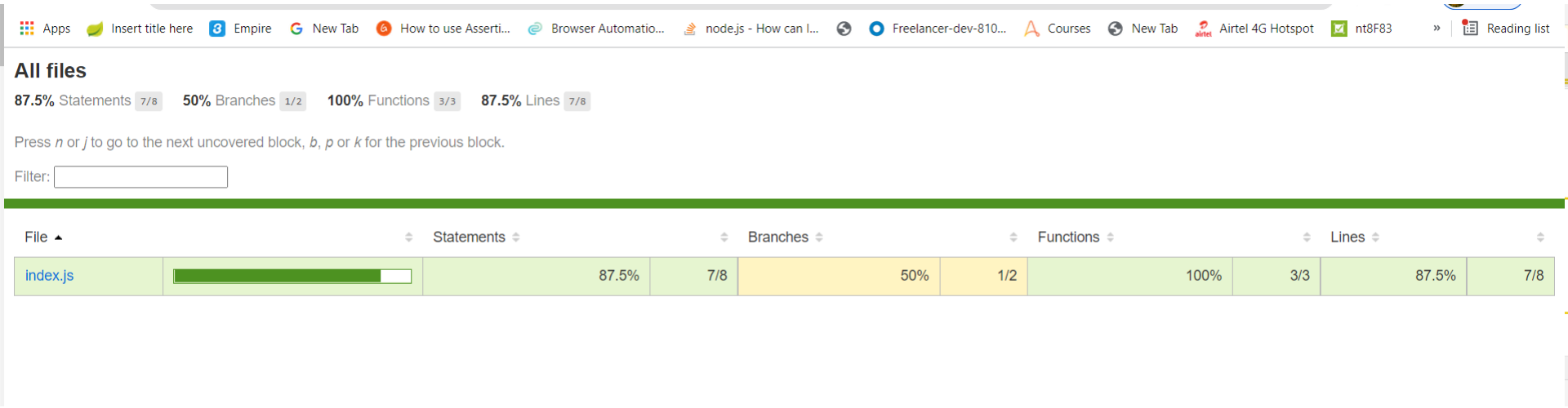
Code Coverage - Istanbul

- Go to test case folder
- nyc mocha ./*.js --timeout=3000

```
PS G:\Local disk\Nodejs\mochaws\day1\tests\test\mock> nyc mocha ./*.js --timeout=3000


  with mock: getPhotosByAlbumId
    √ should getPhotosByAlbumId


  1 passing (115ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |    87.5 |       50 |     100 |    87.5 |
```

Code Coverage - Istanbul

nyc --reporter=html --reporter=text mocha ./*.js --timeout=3000



Questions