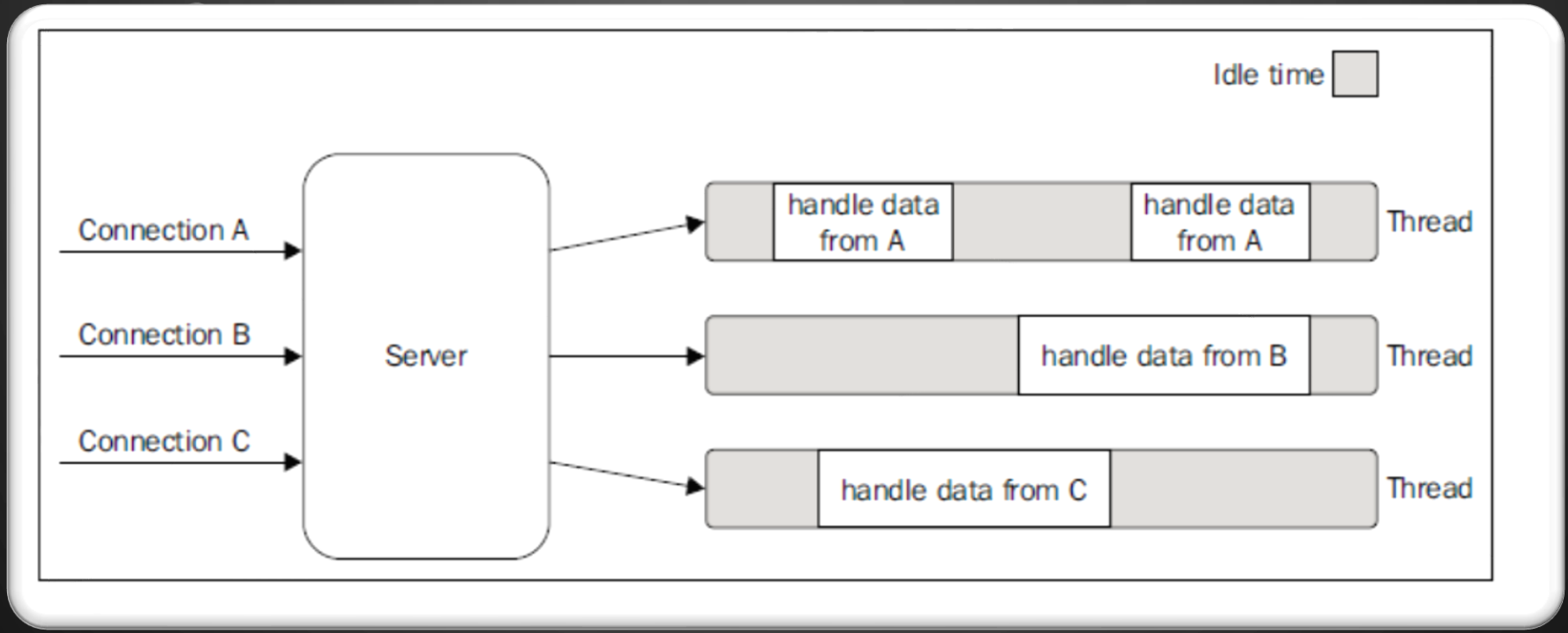
NODE ESSENTIALS

Venugopal Shastri


Idle time
Connection A
Connection B
Connection C
Server
handle data from A
handle data from A
Thread
handle data from B
Thread
handle data from C
Thread

BLOCKING I/O

BUSY-WAITING

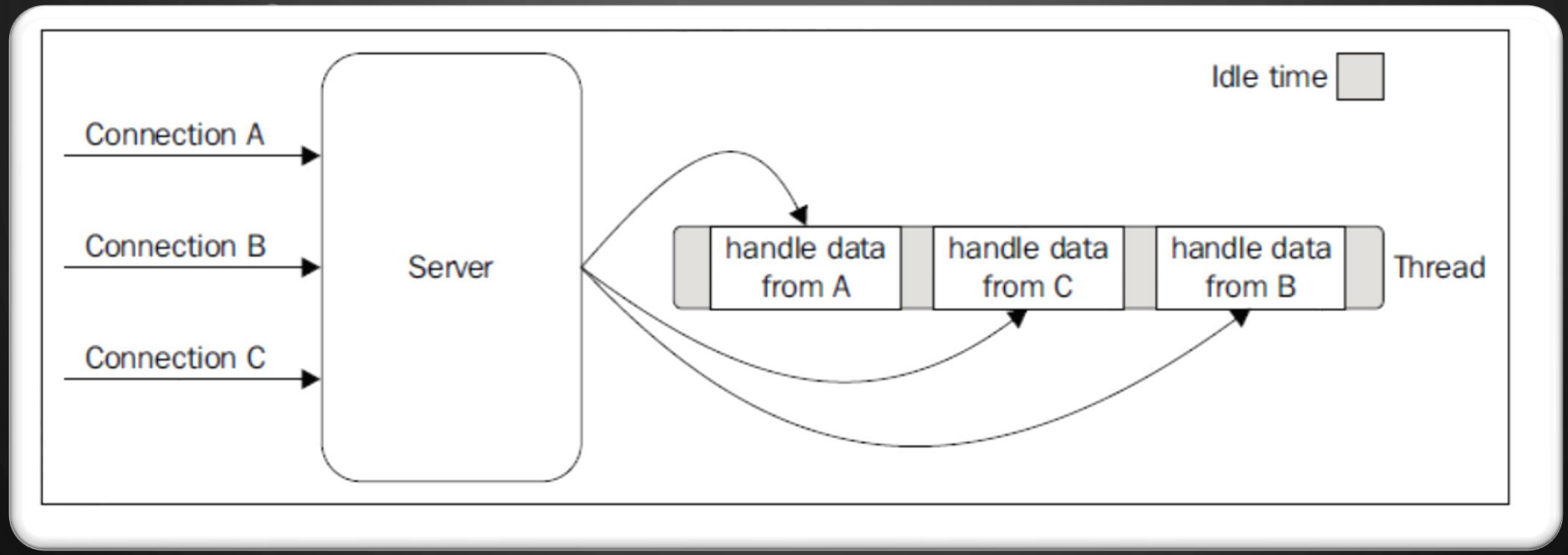
- the loop will consume precious CPU only for iterating over resources that are unavailable most of the time
- Polling algorithms usually result in a huge amount of wasted CPU time

```
resources = [socketA, socketB, pipeA];
while(!resources.isEmpty()) {
  for(i = 0; i < resources.length; i++) {
    resource = resources[i];
    //try to read
    var data = resource.read();
    if(data === NO_DATA_AVAILABLE)
      //there is no data to read at the moment
      continue;
    if(data === RESOURCE_CLOSED)
      //the resource was closed, remove it from the list
      resources.remove(i);
    else
      //some data was received, process it
      consumeData(data);
  }
}
```

EVENT DEMULTIPLEXING

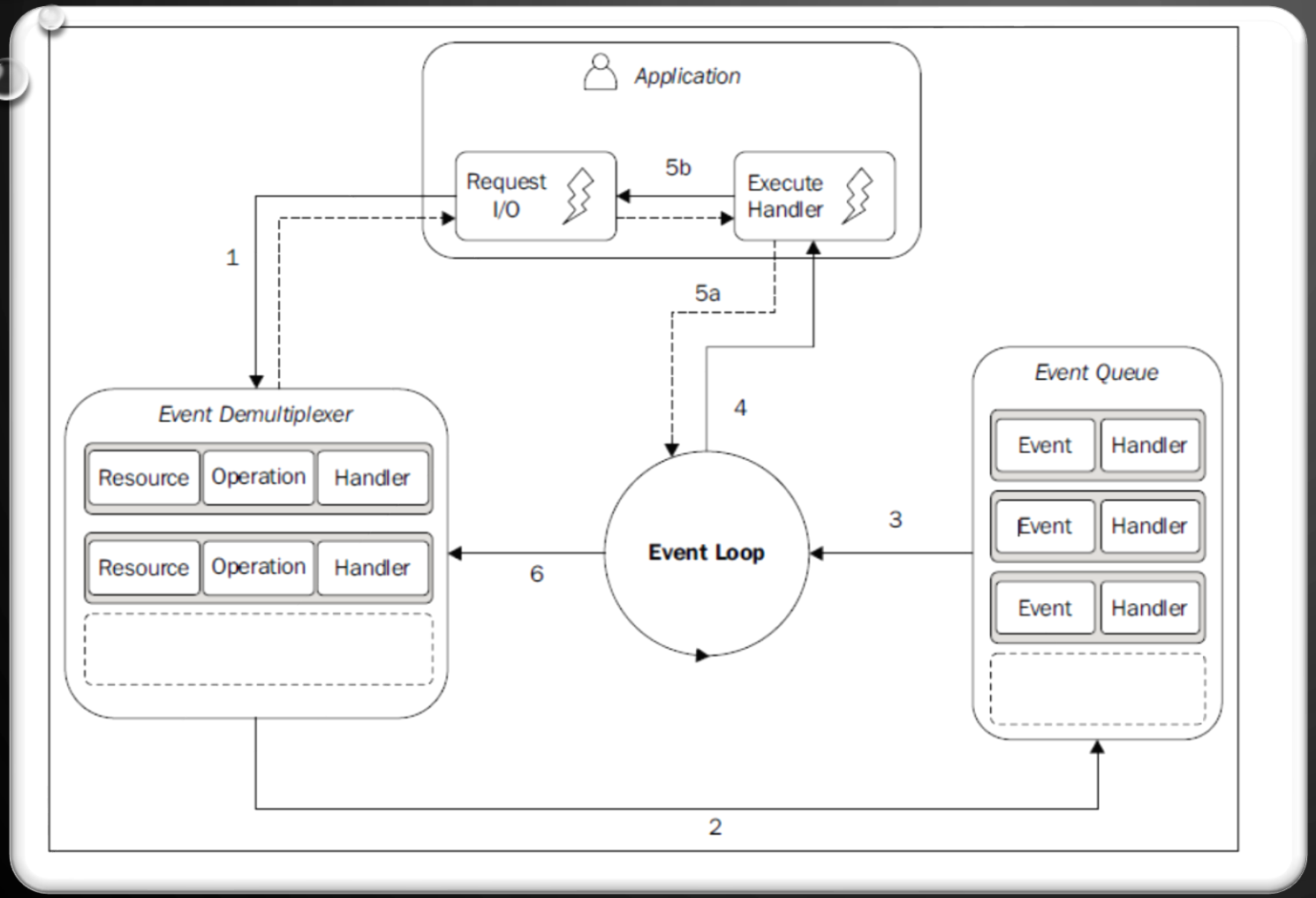
```
socketA, pipeB;
watchedList.add(socketA, FOR_READ);
watchedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(watchedList)) {
  //event loop
  foreach(event in events) {
    //This read will never block and will always return data
    data = event.resource.read();
    if(data === RESOURCE_CLOSED)
      //the resource was closed, remove it from the watched list
      demultiplexer.unwatch(event.resource);
    else
      //some actual data was received, process it
      consumeData(data);
  }
}
```

- The resources are added to a data structure
- The event notifier is set up with the group of resources to be watched (this call is Sync)
- Each event returned by the event demultiplexer is processed
- When all the events are processed, the
- flow will block again on the event demultiplexer until new events are again available to be processed.
  - Event loop



SYNCHRONOUS EVENT DEMULTIPLEXER AND A SINGLE THREAD

THE REACTOR PATTERN



1. The application generates a new I/O operation by submitting a request to the **Event Demultiplexer**

1. Submitting a new request to the Event Demultiplexer is a non-blocking call and it immediately returns the control back to the application
2. When a set of I/O operations completes, the Event Demultiplexer pushes the new events into the **Event Queue**
3. Event Loop iterates over the items of the **Event Queue**
4. For each event, the associated handler is invoked
5. Handler
    a) The handler, will give back the control to the Event Loop when its execution completes
    b) new asynchronous operations might be requested during the execution of the handler
6. When all the items in the Event Queue are processed, the loop will block again on the Event Demultiplexer which will then trigger another cycle.

PATTERN AT THE HEART OF NODE.JS.

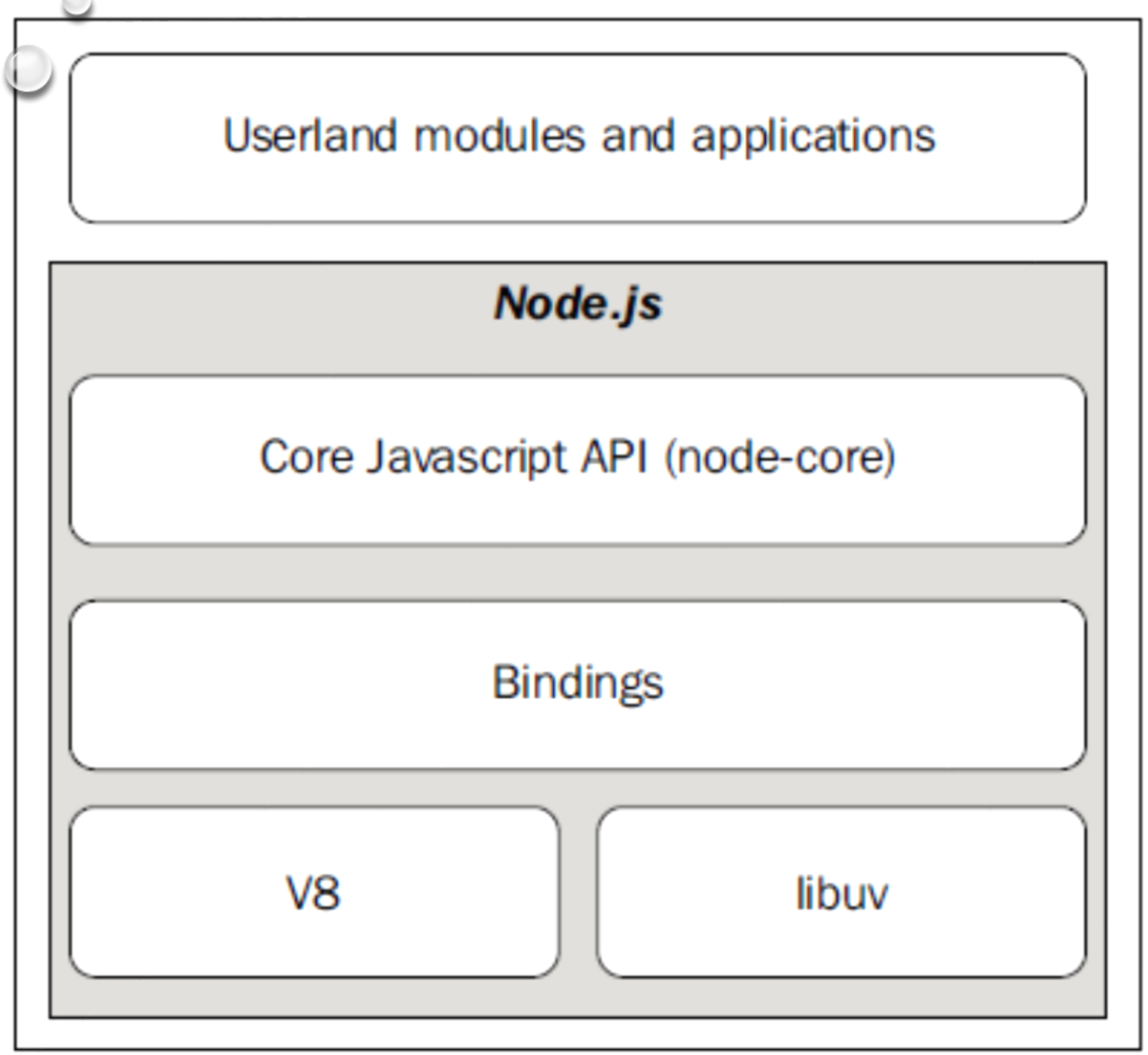
- Reactor

Handles I/O by blocking until new events are available from a set of observed resources, and then reacting by dispatching each event to an associated handler

THE NON-BLOCKING I/O ENGINE

- Linux
  - Epoll
- Mac OS X
  - Kqueue
- Windows
  - IOCP API
- Node
  - Libuv
    - Abstract System Calls
    - Implements Reactor Pattern

NODE JS PLATFORM



- **Bindings** - responsible for wrapping and exposing libuv and other low-level functionality to JavaScript
- **V8** - The JavaScript engine
- **node-core –** A core JavaScript library that implements the high-level Node.js API.

RUN TO COMPLETION AND THE EVENT LOOP

- The JavaScript you write runs on a single thread
  - where are the async tasks and callbacks run?
- The call to http.get() triggers a network request that a separate thread handles
  - Wait—you were just told that JavaScript is single-threaded
- The JavaScript code you write all runs on a single thread, but the code that implements the async tasks (the http.get()) is not part of that Java-Script and is free to run in a separate thread.
- Once the task completes the result needs to be provided to the JavaScript thread

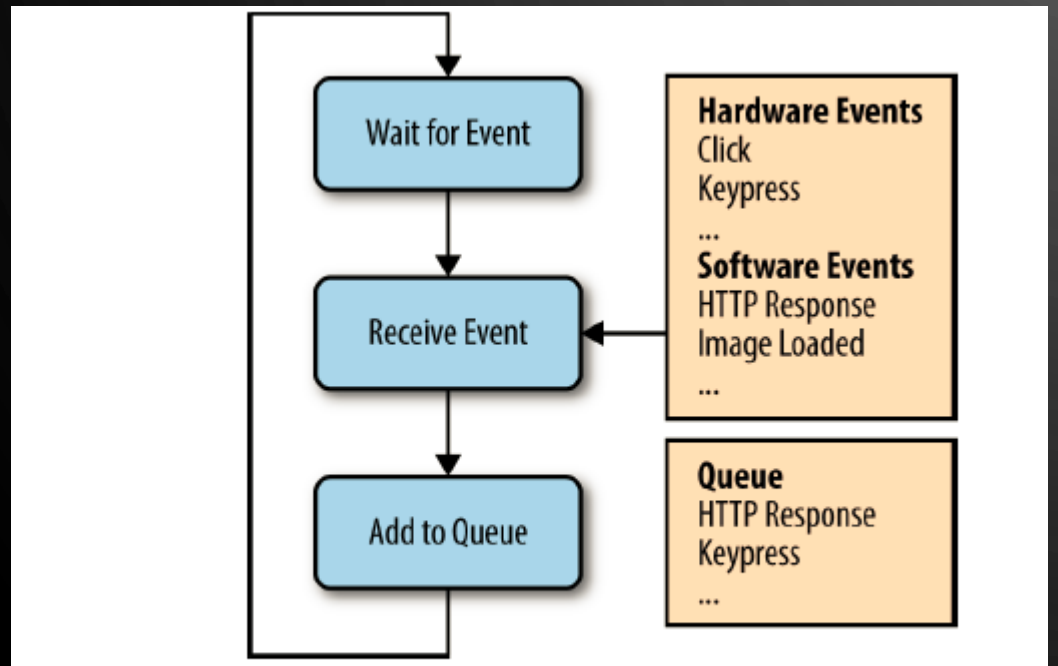
```
var http = require('http');
http.get('http://www.google.com', function (res) {
    console.log('got a response');
});
```

RUN TO COMPLETION

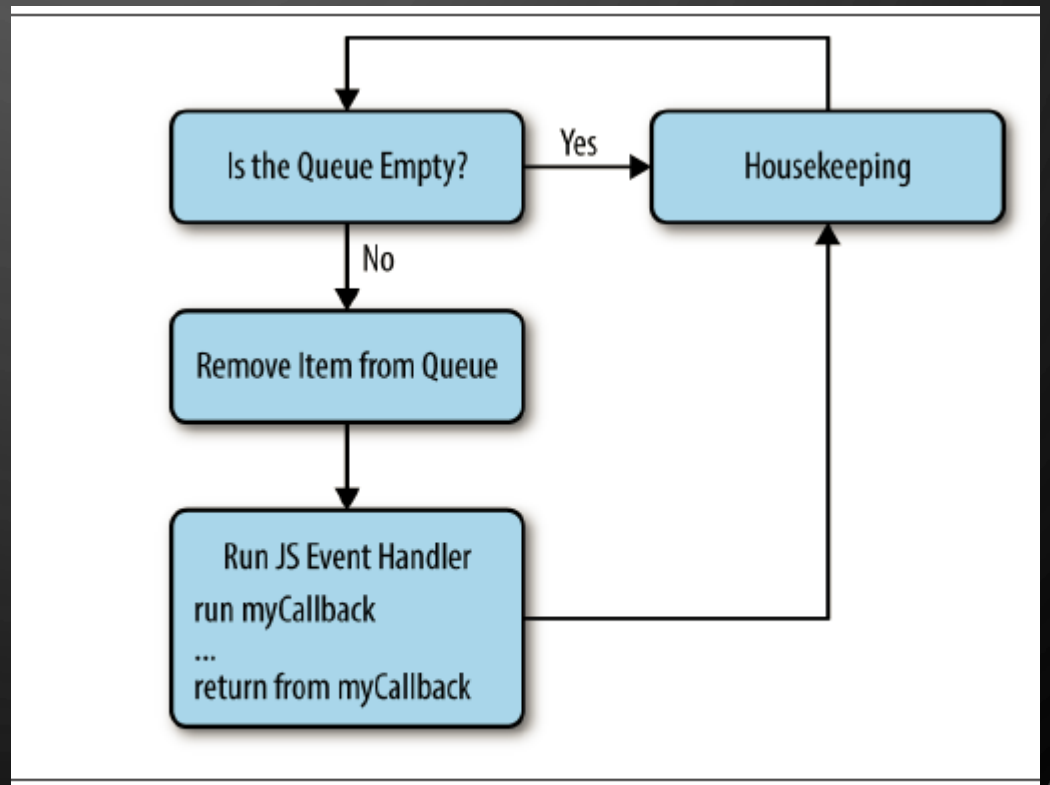
- A multithreaded language might interrupt whatever code was currently executing to provide the results
  - In JavaScript interruptions are forbidden
- run-to-completion rule
  - JavaScript code runs without interruption until it passes control back to the host environment by returning from the function that the host initially called. At that point the callback can be removed from the queue and invoked
  - All other threads communicate with your code by placing items on the queue.
  - Other threads are not permitted to manipulate any other memory accessible to JavaScript
  - Once the Callback is added to the queue, there is no guarantee how long it will have to wait

EVENT LOOP

- The JavaScript runtime simply continues in an endless cycle of pulling an item off the queue if one is available, running the code that the item triggers, and then checking the queue again. This cycle is known as the event loop.



. Filling the queue



The JavaScript event loop

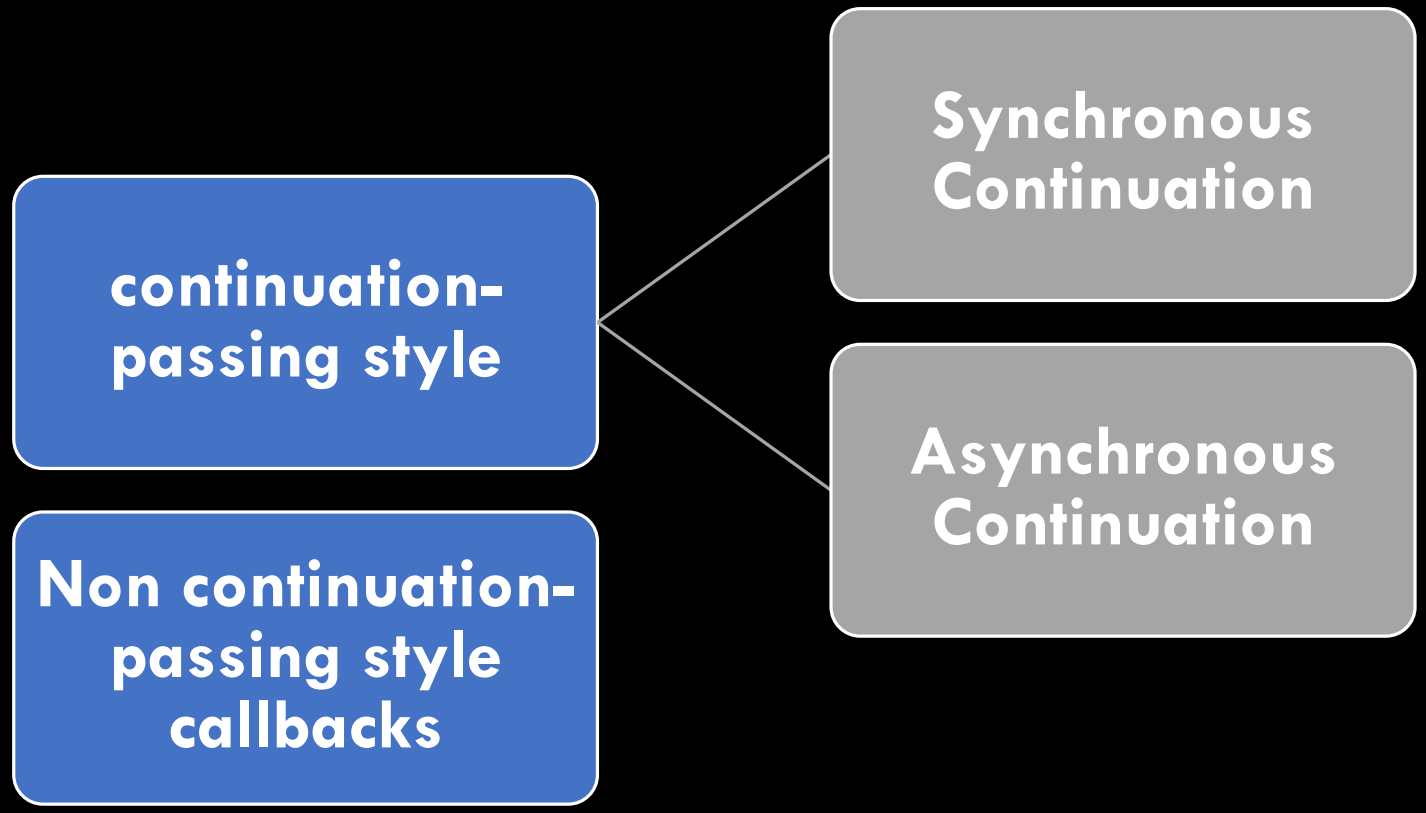
```
var async = true;
var xhr = new XMLHttpRequest();
xhr.open('get', 'data.json', async);
xhr.send();

setTimeout(function delayed() { // Creates race condition!
    function listener() {
        console.log('greetings from listener');
    }
    xhr.addEventListener('load', listener);
    xhr.addEventListener('error', listener);
}, 3000);
```

RACE CONDITION

CALLBACK

- Callbacks are functions that are invoked to propagate the result of an operation
- Practically replace the use of the return instruction in function
- **closures** are an ideal construct for implementing callbacks
  - Allows to reference the environment in which a function was created, practically
  - Allows to maintain the context in which the asynchronous operation was requested, no matter when or where its callback is invoked

CALLBACK PATTERN

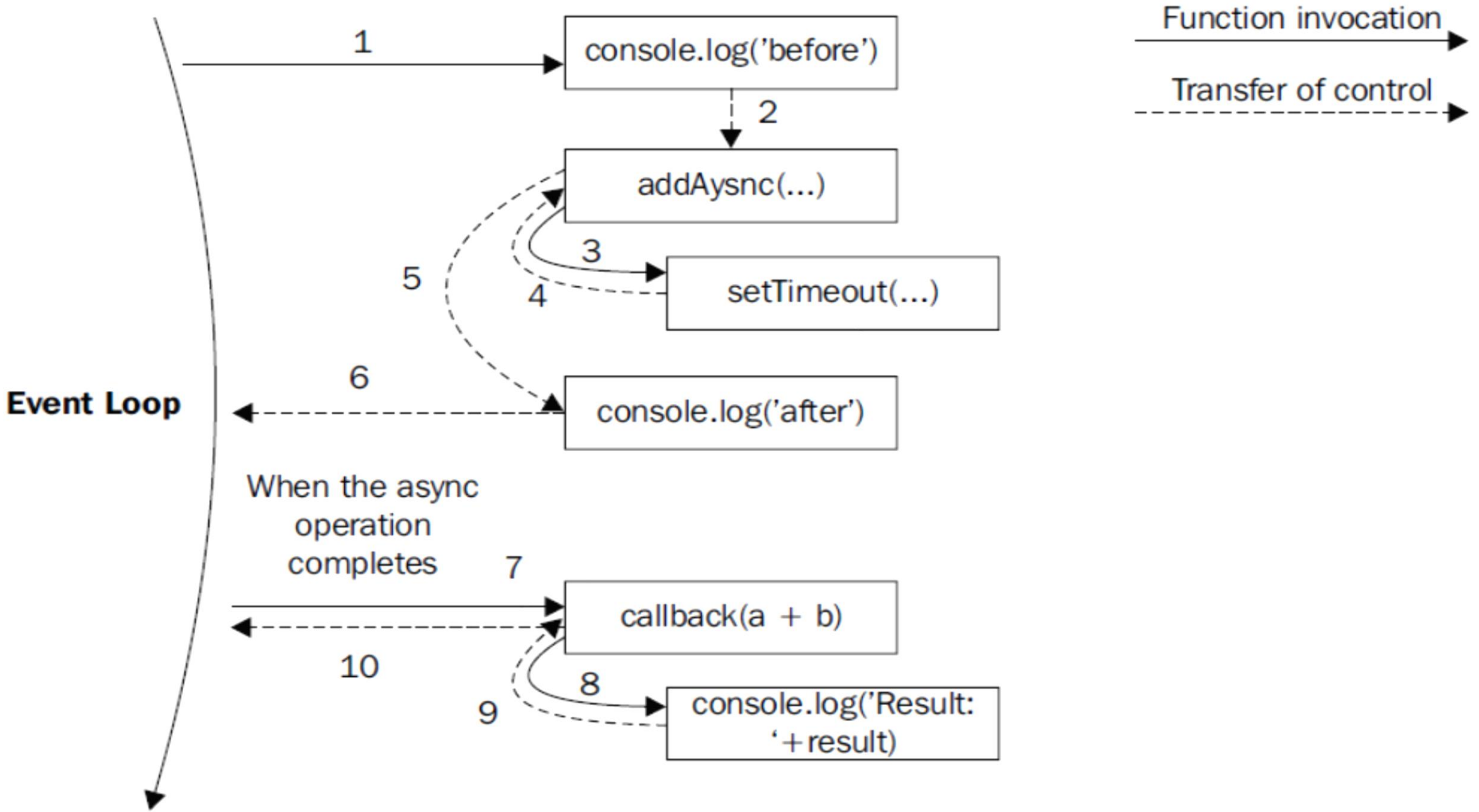


```
function add(a, b, callback) {
  callback(a + b);
}
```

SYNCHRONOUS CONTINUATION-PASSING STYLE

```
function addAsync(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 100);
}
```

ASYNCHRONOUS CONTINUATION-PASSING STYLE


Function invocation
Transfer of control
1
console.log('before')
2
addAysnc(...)
3
4
5
setTimeout(...)
6
Event Loop
console.log('after')
When the async operation completes
7
callback(a + b)
10
8
9
console.log('Result: '+result)

```
var result = [1, 5, 7].map(function(element) {
  return element - 1;
});
```

NON CONTINUATION-PASSING STYLE CALLBACKS

If you have an API which takes a callback,and *sometimes* that callback is called immediately, and o*ther times* that callback is called at some point in the future, then you will render any code using this API impossible to reason about, and cause the release of Zalgo.

DO NOT RELEASE ZALGO

Unleashing Zalgo

```
var fs = require('fs');
var cache = {};
function inconsistentRead(filename, callback) {
  if(cache[filename]) {
    //invoked synchronously
    callback(cache[filename]);
  } else {
    //asynchronous function
   fs.readFile(filename, 'utf8', function(err, data) {
     cache[filename] = data;
     callback(data);
   });
 }
}
```

```
function createFileReader(filename) {
  var listeners = [];
  inconsistentRead(filename, function(value) {
    listeners.forEach(function(listener) {
      listener(value);
    });
  });

  return {
    onDataReady: function(listener) {
      listeners.push(listener);
    }
  };
}
```

```
var reader1 = createFileReader('data.txt');
reader1.onDataReady(function(data) {
  console.log('First call data: ' + data);
```

```
var reader2 = createFileReader('data.txt');
reader2.onDataReady(function(data) {
  console.log('Second call data: ' + data);
});
});
```

SOLUTION TO ZALGO

Purely synchronous functions

- prefer the direct style for purely synchronous functions.
- Use blocking API only when they don't affect the ability of the application to serve concurrent requests.

Purely Asynchronous Functions

- **Deferred execution**

```
var fs = require('fs');
var cache = {};
function consistentReadSync(filename) {
  if(cache[filename]) {
    return cache[filename];
  } else {
    cache[filename] = fs.readFileSync(filename, 'utf8');
    return cache[filename];
  }
}
```

SYNCHRONOUS API

PURE ASYNCHRONOUS API

- we guarantee that a callback is invoked asynchronously by deferring its execution using process.nextTick().
- Process.nextTick()
  - defers the execution of a function until the next pass of the event loop

```
var fs = require('fs');
var cache = {};
function consistentReadAsync(filename, callback) {
  if(cache[filename]) {
    process.nextTick(function() {
      callback(cache[filename]);
    });
  } else {
    //asynchronous function
    fs.readFile(filename, 'utf8', function(err, data) {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

NODE.JS CALLBACK CONVENTIONS

- **Callbacks come last**
- **Error comes first**

```
fs.readFile(filename, [options], callback)
```

```
fs.readFile('foo.txt', 'utf8', function(err, data) {
                                  if(err)
                                      handleError(err);
                                   else
                                    processData(data);
                                   });
```

THE MODULE SYSTEM AND ITS PATTERNS

- **Revealing module pattern**
- *CommonJS modules*
- *ES 6 Modules*

```
var module = (function() {
    var privateFoo = function() { };
    var privateVar = [];
    var projection = {
        publicFoo: function() {},
        publicBar: function() {}
    }
    return peojection;
    })();
```

REVEALING MODULE PATTERN

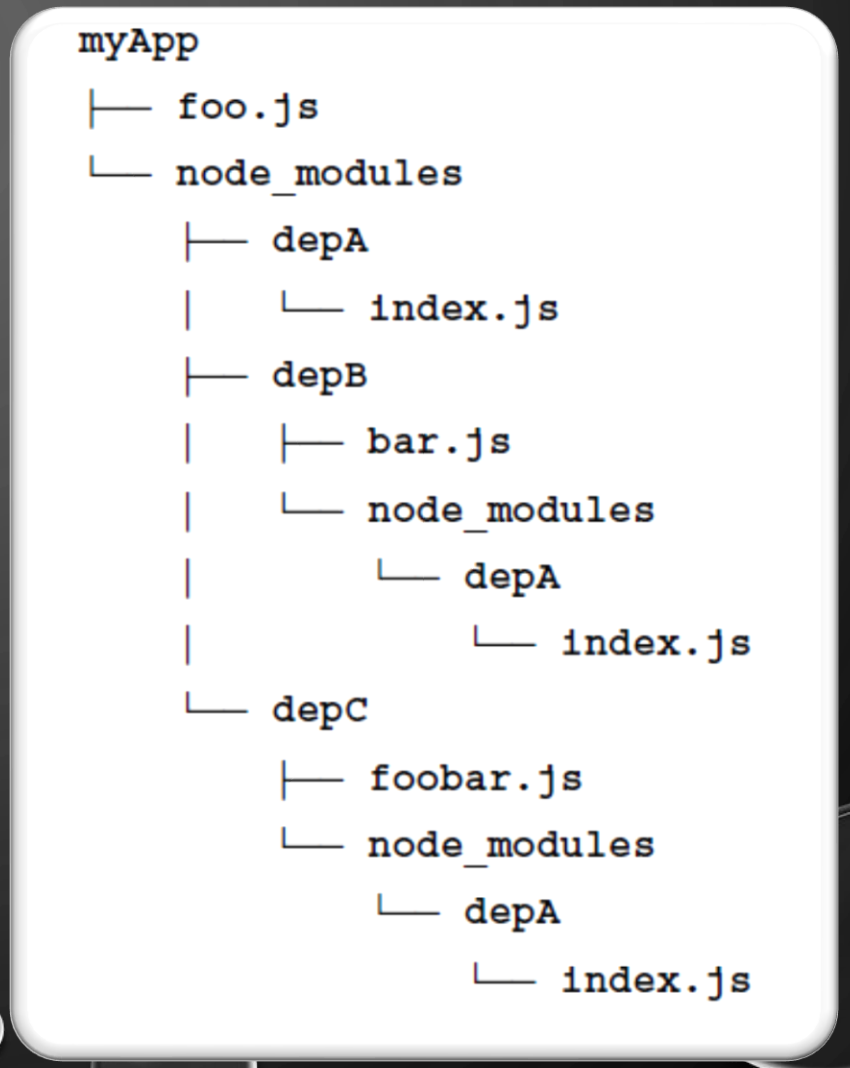
THE RESOLVING ALGORITHM

- How require() resolves module
  - File module
    - If moduleName is prefixed with "/" or "./"
  - Core module
    - If moduleName is not prefixed with "/" or "./", the algorithm will first try to search within the core Node.js modules
  - Package module
    - If no core module is found matching moduleName, then the search continues by looking for a matching module into the first node_modules directory that is found navigating up in the directory structure starting from the requiring module. The algorithm continues to search for a match by looking into the next node_modules directory up in the directory tree, until it reaches the root of the filesystem

Calling require('depA') from /myApp/foo.js will load /myApp/node_modules/depA/index.js

Calling require('depA') from /myApp/node_modules/depB/bar.js will load /myApp/node_modules/depB/node_modules/depA/index.js

Calling require('depA') from /myApp/node_modules/depC/foobar.js will load /myApp/node_modules/depC/node_modules/depA/index.js



THE MODULE CACHE

- Modules are cached after the first time they are loaded
- Caching is crucial for performances
- It guarantees, to some extent, that always the same instance is returned when requiring the same module from within a given package
- require.cache – Allows direct access to the cache
  - A common use case is to invalidate any cached module by deleting the relative key in the require.cache

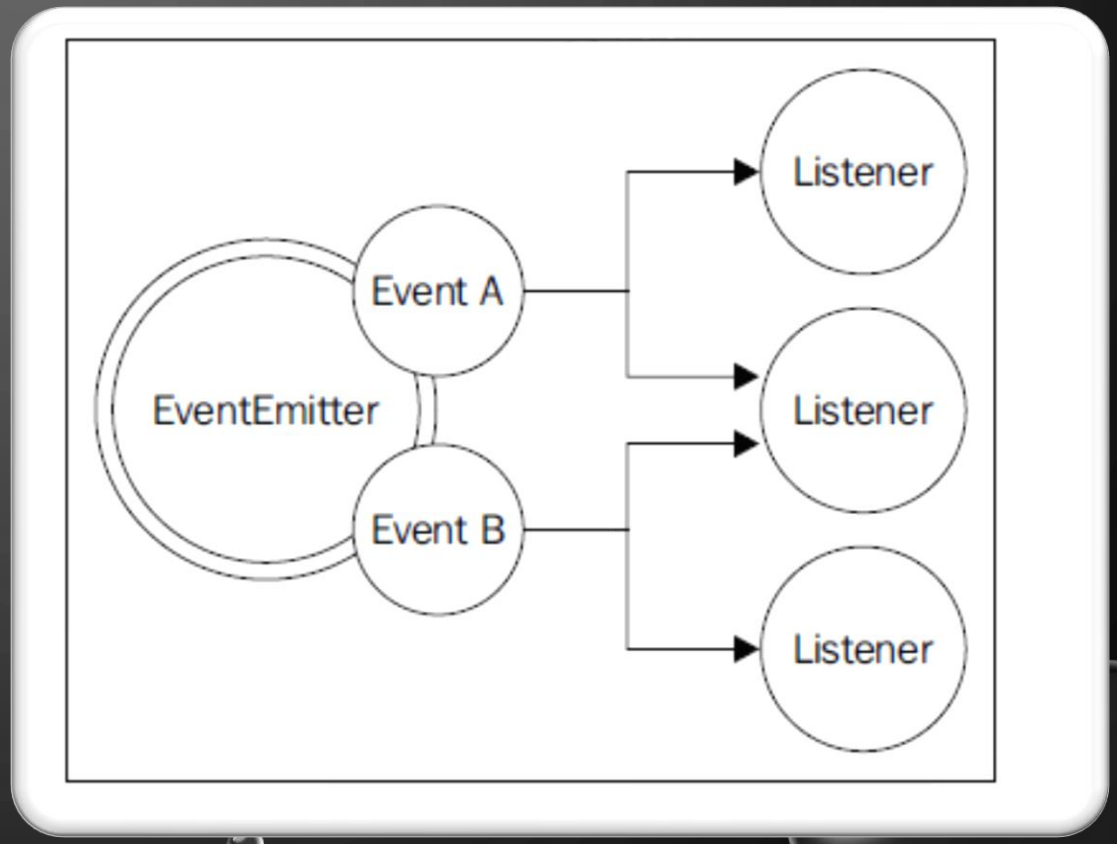
```
//file logger.js

module.exports = function(message) {
  console.log('info: ' + message);
};

module.exports.verbose = function(message) {
  console.log('verbose: ' + message);
};


//file main.js
var logger = require('./logger');
logger('This is an informational message');
logger.verbose('This is a verbose message');
```

SUBSTACK PATTERN

- EXPOSE THE MAIN FUNCTIONALITY OF A MODULE BY EXPORTING ONLY ONE FUNCTION. USE THE EXPORTED FUNCTION AS NAMESPACE TO EXPOSE ANY AUXILIARY FUNCTIONALITY.

THE EVENTEMITTER

```
var EventEmitter = require('events').EventEmitter;
var eeInstance = new EventEmitter();
```



MAKING OBJECT OBSERVABLE

```
var EventEmitter = require('events').EventEmitter;
var util = require('util');
var fs = require('fs');

function FindPattern(regex) {
  EventEmitter.call(this);
  this.regex = regex;
  this.files = [];
}
util.inherits(FindPattern, EventEmitter);

FindPattern.prototype.addFile = function(file) {
  this.files.push(file);
  return this;
};
```

EVENTEMITTER VS CALLBACKS

```
function helloEvents() {
  var eventEmitter = new EventEmitter();
  setTimeout(function() {
    eventEmitter.emit('hello', 'world');
  }, 100);
  return eventEmitter;
}
function helloCallback(callback) {
  setTimeout(function() {
    callback('hello', 'world');
  }, 100);
}
```

- A callbacks can notify only that particular callback
- Events can notify registered multiple listeners
- A callback, in fact, is expected to be invoked exactly once, whether the operation is successful or not
- Events can occur multiple times

COMBINE CALLBACKS AND EVENTEMITTER

- create a function that accepts a callback and returns an EventEmitter, thus providing a simple and clear entry point for the main functionality, while emitting more fine-grained events using the EventEmitter

```
var glob = require('glob');
glob('data/*.txt', function(error, files) {
  console.log('All files found: ' + JSON.stringify(files));
}).on('match', function(match) {
  console.log('Match found: ' + match);
});
```

WIRING MODULES

- what's the best way to pass an instance of component X into module Y?
- Hardcoded dependency
- Dependency injection
- Service locator
- Dependency injection containers

HARDCODED DEPENDENCY

DEPENDENCY INJECTION

- Use Factory in the module to return an instance of service
- Parameterize the dependency
- The different types of dependency injection
  - Constructor Injection
    - var service = new Service(dependencyA, dependencyB);
  - Setter or Property Injection
    - var service = new Service();
    - service.dependencyA = anInstanceOfDependencyA;

```
module.exports = function(authService) {
  var authController = {};

  authController.login = function (req, res, next) {
    //...same as in the previous version
  };
```

THE DEPENDENCY DEADLOCK

```
function Afactory(b) {
  return {
    foo: function() {
      b.say();
    },
    what: function() {
      return 'Hello!';
    }
  }
}

function Bfactory(a) {
  return {
    a: a,
    say: function() {
      console.log('I say: ' + a.what);
    }
  }
}
```

SERVICE LOCATOR

- Central **registry** in order to manage the components of the system and to act as a mediator whenever a module needs to load a dependency
- Types of Service Locators
  - Hardcoded dependency on service locator
    - introduces a tight coupling with the component
  - Injected service locator
    - *service locator* is referenced by a component through dependency injection
  - Global service locator
    - Referencing a service locator is directly from the global scope
- The Node.js module system already implements a variation of the service locator pattern, with require() representing the global instance of the service locator itself.

```
module.exports = function() {
  var dependencies = {};
  var factories = {};
  var serviceLocator = {};

  serviceLocator.factory = function(name, factory) {
    factories[name] = factory;
  };

  serviceLocator.register = function(name, instance) {
    dependencies[name] = instance;
  };

  serviceLocator.get = function(name) {
    if(!dependencies[name]) {
      var factory = factories[name];
      dependencies[name] = factory && factory(serviceLocator);
      if(!dependencies[name]) {
        throw new Error('Cannot find module: ' + name);
      }
    }
    return dependencies[name];
  };

  return serviceLocator;
};
```

```
var svcLoc = require('./lib/serviceLocator')();

svcLoc.register('dbName', 'example-db');
svcLoc.register('tokenSecret', 'SHHH!');
svcLoc.factory('db', require('./lib/db'));
svcLoc.factory('authService', require('./lib/authService'));
svcLoc.factory('authController', require('./lib/authController'))

var authController = svcLoc.get('authController');
```

DEPENDENCY INJECTION CONTAINER

```
module.exports = function(db, tokenSecret) {
  //...
}
```

```
module.exports = function(a, b) {};
module.exports._inject = ['db', 'another/dependency'];
```

```
module.exports = ['db', 'another/depencency',
                  function(a, b) {}];
```

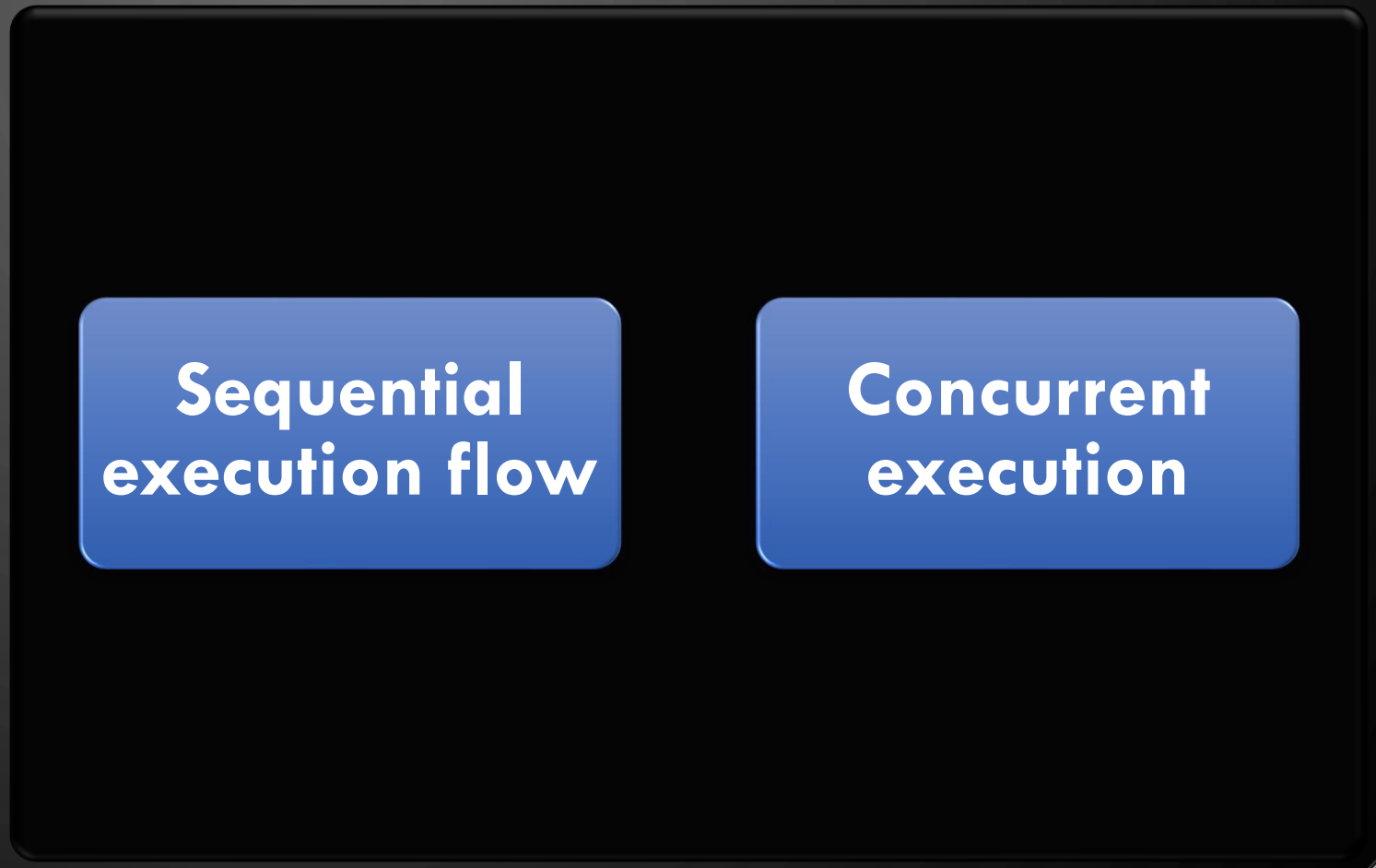
- A DI container is essentially a service locator
- it identifies the dependency requirements of a module *before* instantiating it
- How does DI Container determine dependencies of module
  - Argument names used in a factory or constructor
  - Use a special property attached to the factory function
  - specify a module as an array of dependency names followed by the factory function:

ASYNCHRONOUS CONTROL FLOW PATTERNS WITH CALLBACKS

CALLBACK DISCIPLINE

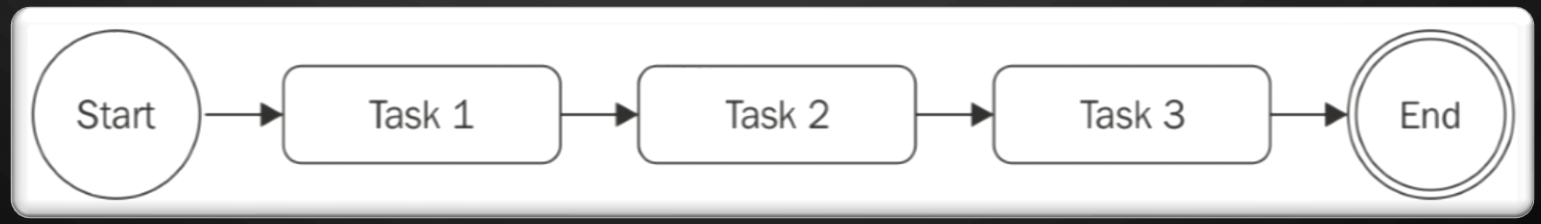
- You must exit as soon as possible. Use return, continue, or break
  - Depending on the context exit immediately from the current statement instead of writing (and nesting) complete if...else statements. This will help keep our code shallow
- You need to create named functions for callbacks
  - Naming our functions will also make them look better in stack traces.
- You need to modularize the code
  - Split the code into smaller, reusable functions

ASYNC-CONTROL-FLOW-PATTERNS

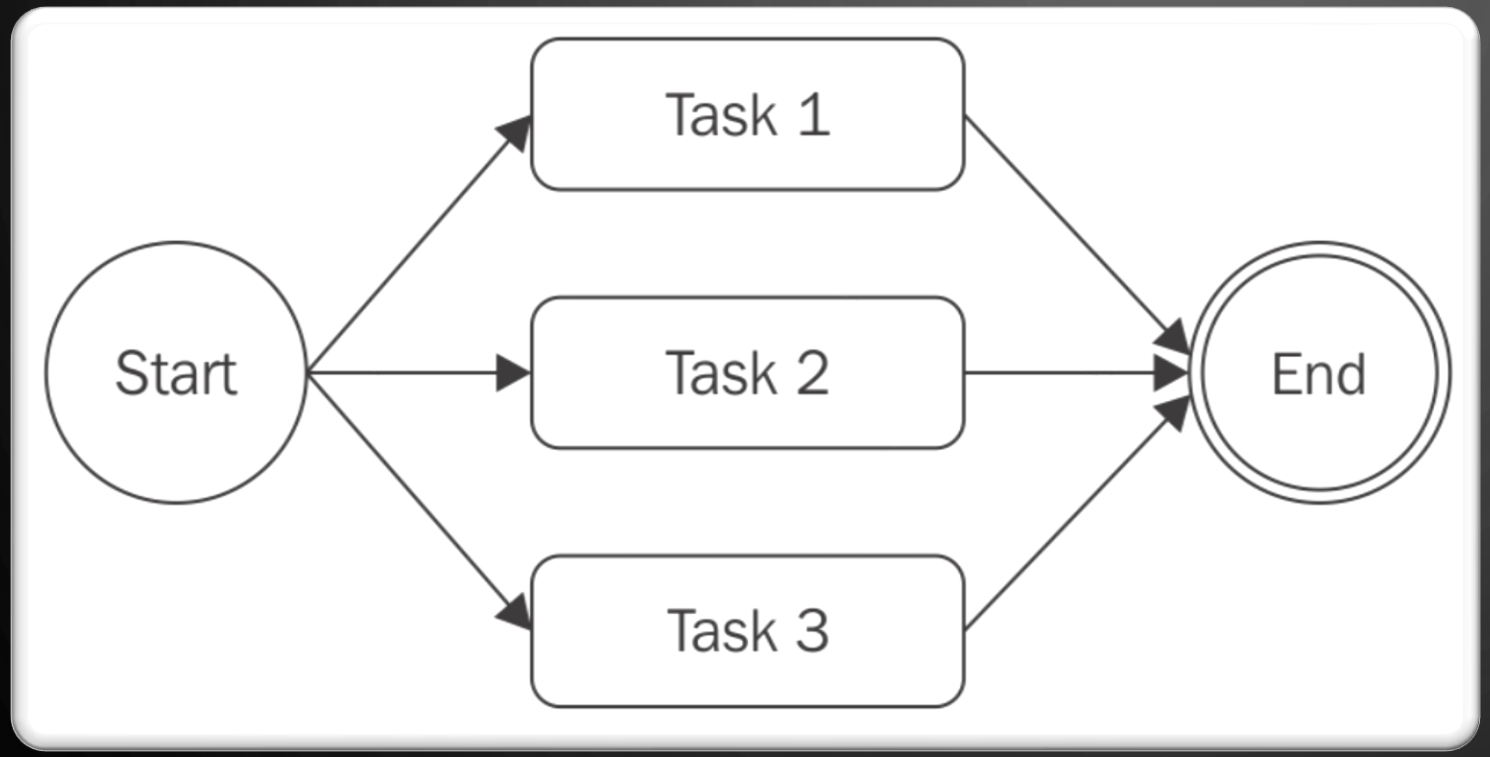


SEQUENTIAL CONTROL FLOW

- Executing a set of known tasks in sequence, without chaining or propagating results
- Using the output of a task as the input for the next (also known as *chain*, *pipeline*, or *waterfall*)
- Iterating over a collection while running an asynchronous task on each element, one after the other
- works perfectly if we know in advance what and how many tasks are to be executed



CONCURRENT EXECUTION



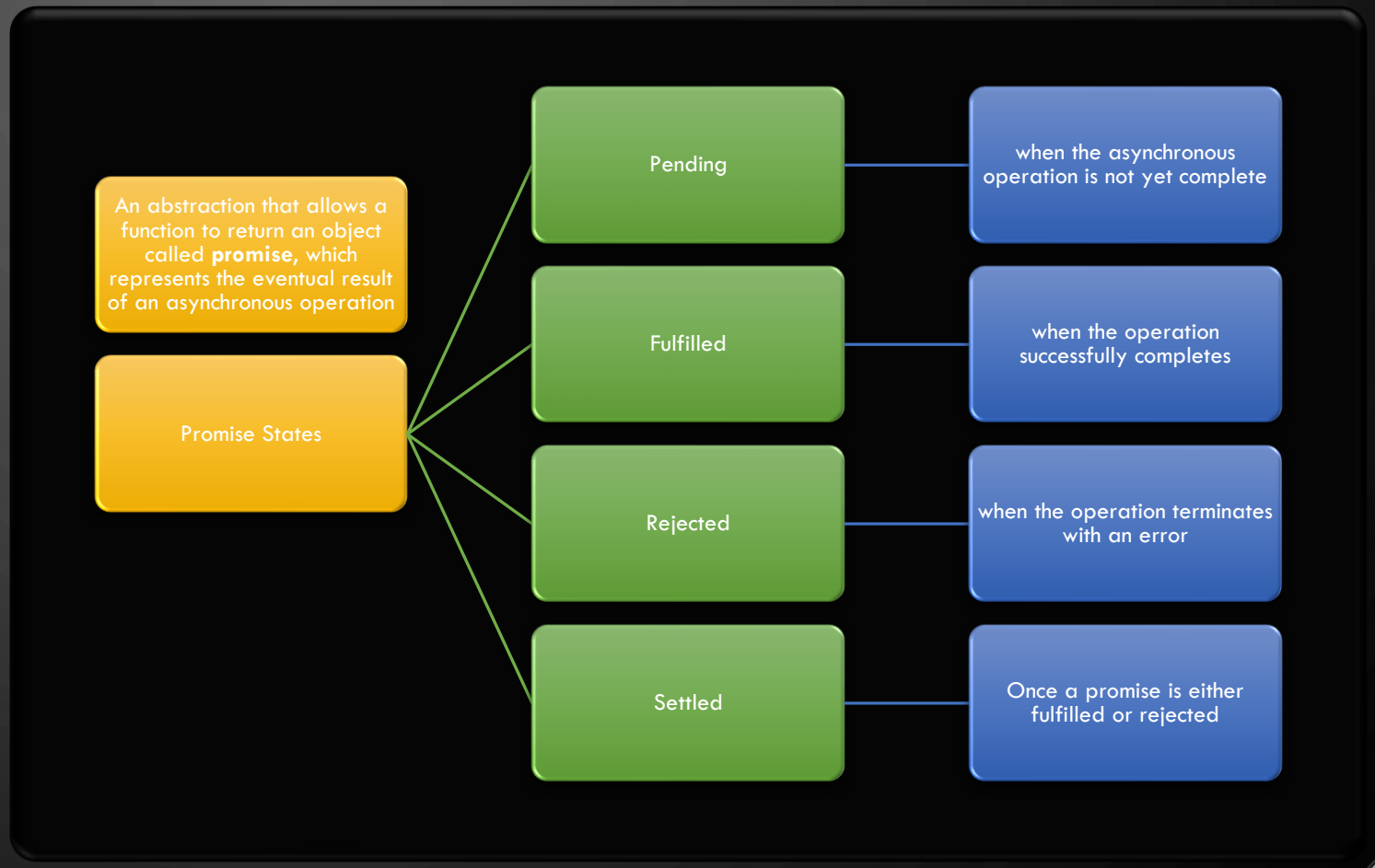
- The order of execution of a set of asynchronous tasks is not important
- Application just to be notified when all those running tasks are completed

```
const tasks = [ /* ... */ ];
let completed = 0;
tasks.forEach(task => {
  task(() => {
    if(++completed === tasks.length) {
      finish();
    }
  });
});

function finish() {
  //all the tasks completed
}
```

GENERIC CONCURRENT PATTERN

ACFP USING PROMISES, GENERATORS AND ASYNC/AWAIT

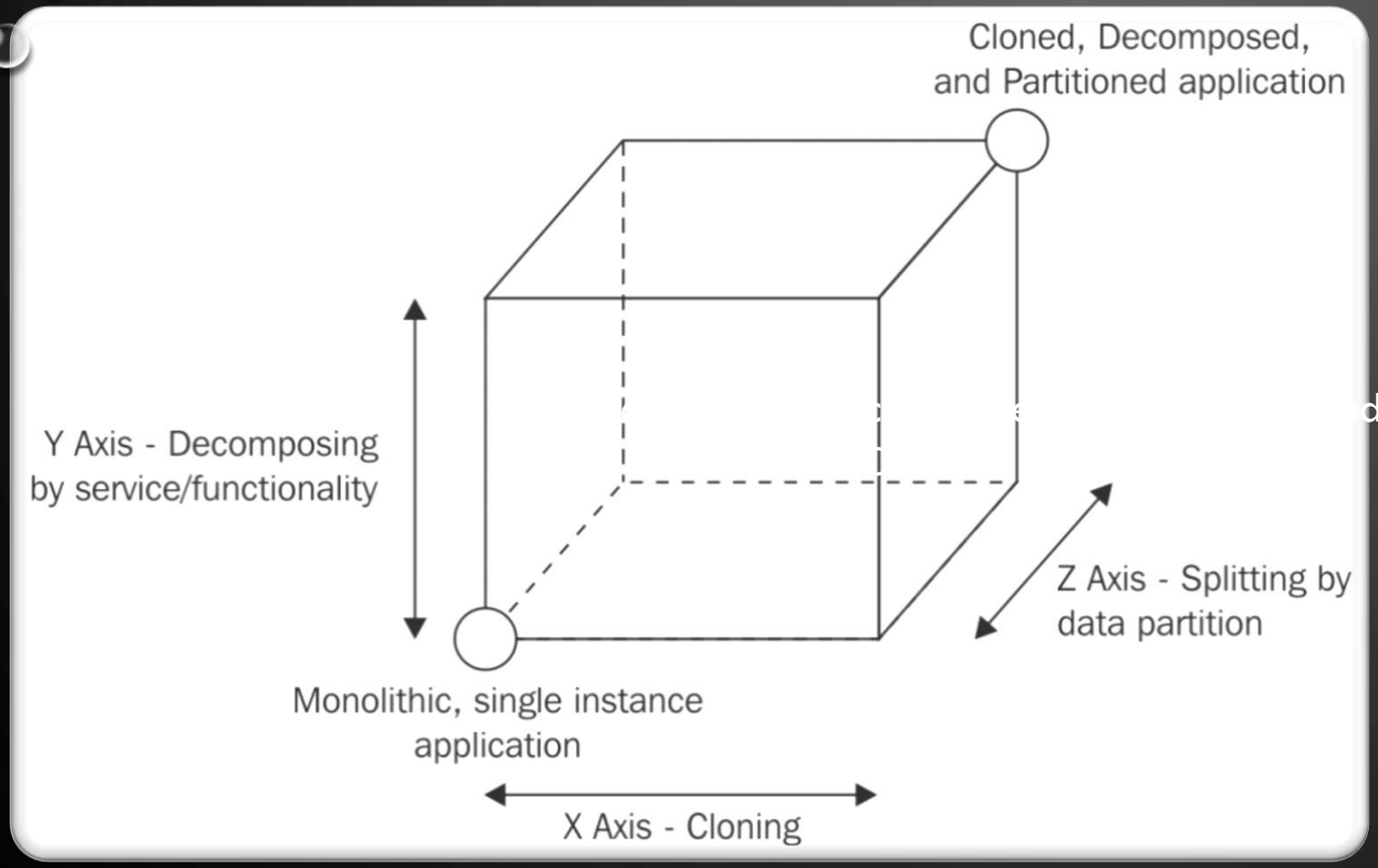
WHAT IS A PROMISE?



SCALABILITY AND ARCHITECTURAL PATTERNS

- What the scale cube is
- How to scale by running multiple instances of the same application
- How to leverage a load balancer when scaling an application
- What a service registry is and how it can be used
- How to design a microservice architecture out of a monolithic application
- How to integrate a large number of services through the use of some simple architectural patterns

SCALE CUBE



- Reference :-

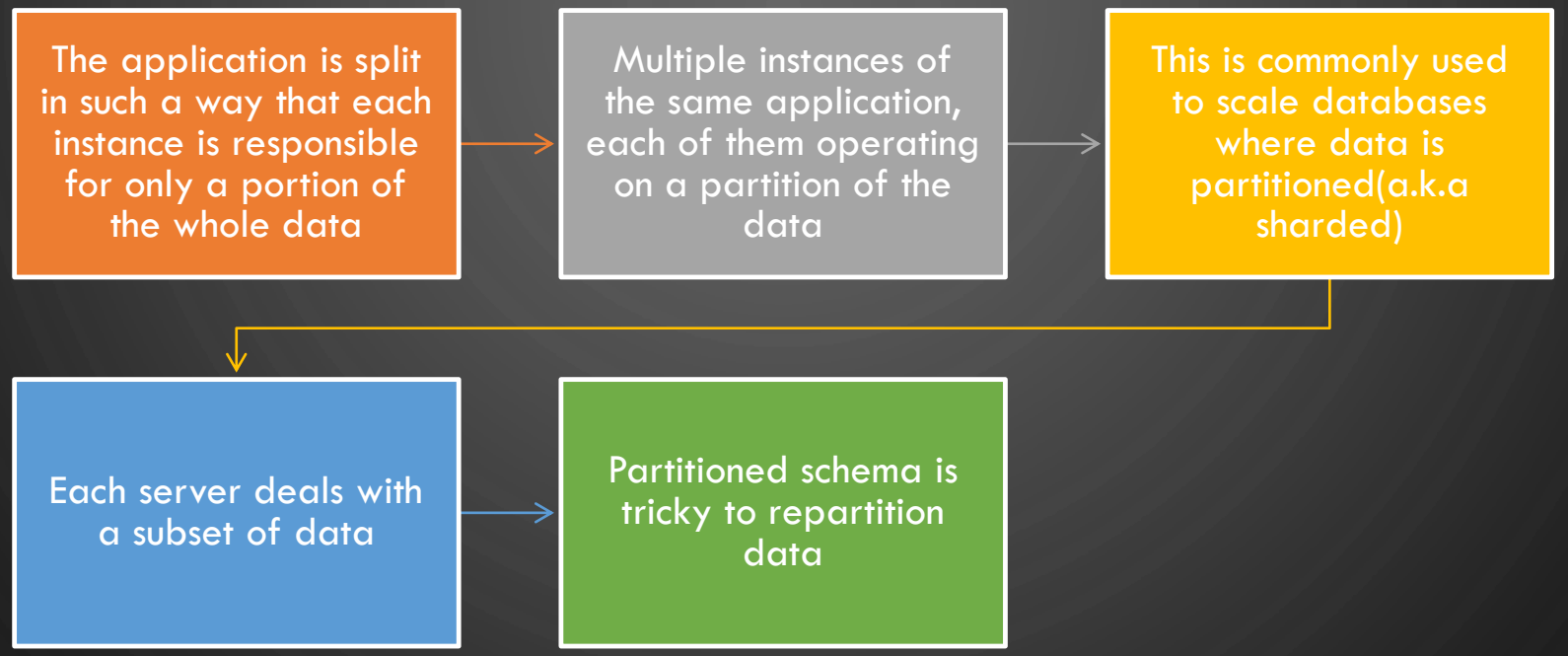
The Art of Scalability by *Martin L. Abbott* and *Michael T. Fisher*

X-AXIS SCALING

- Running multiple copies of an application behind a load balancer
- Each copy can handle 1/N of the load
- Each copy accesses all the data, cache size will be higher.
- Doesn't solve increasing development and application complexity

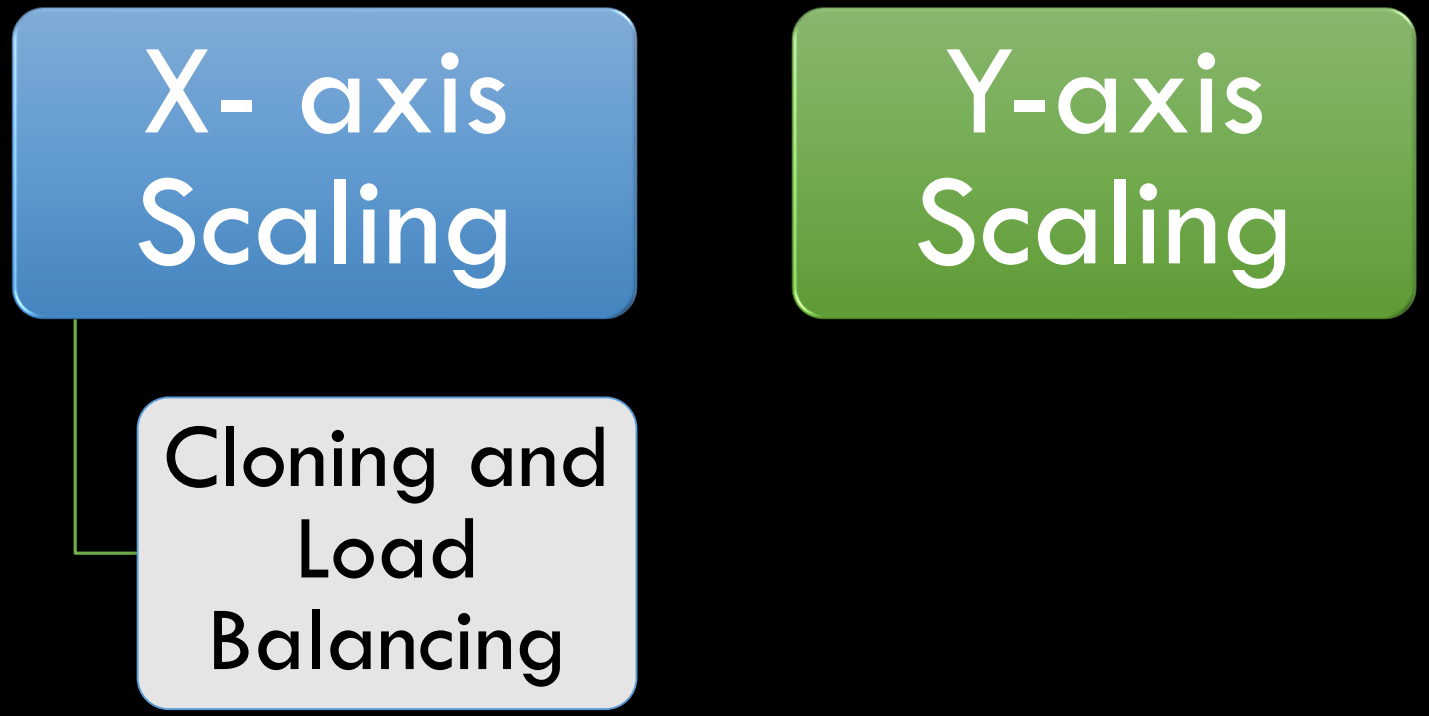
Y-AXIS SCALING

- Splitting the application into multiple, different services (microservices)
- Then more infra resources can be added to only the micro-service which is bottleneck in the architecture.
- Decomposing the application based on its functionalities, services, or use cases



Z-AXIS SCALING

SCALING NODE JS APPLICATION



CLONING AND LOAD BALANCING

Vertical Scaling

Adding more resources to a single machine

Horizontal Scaling

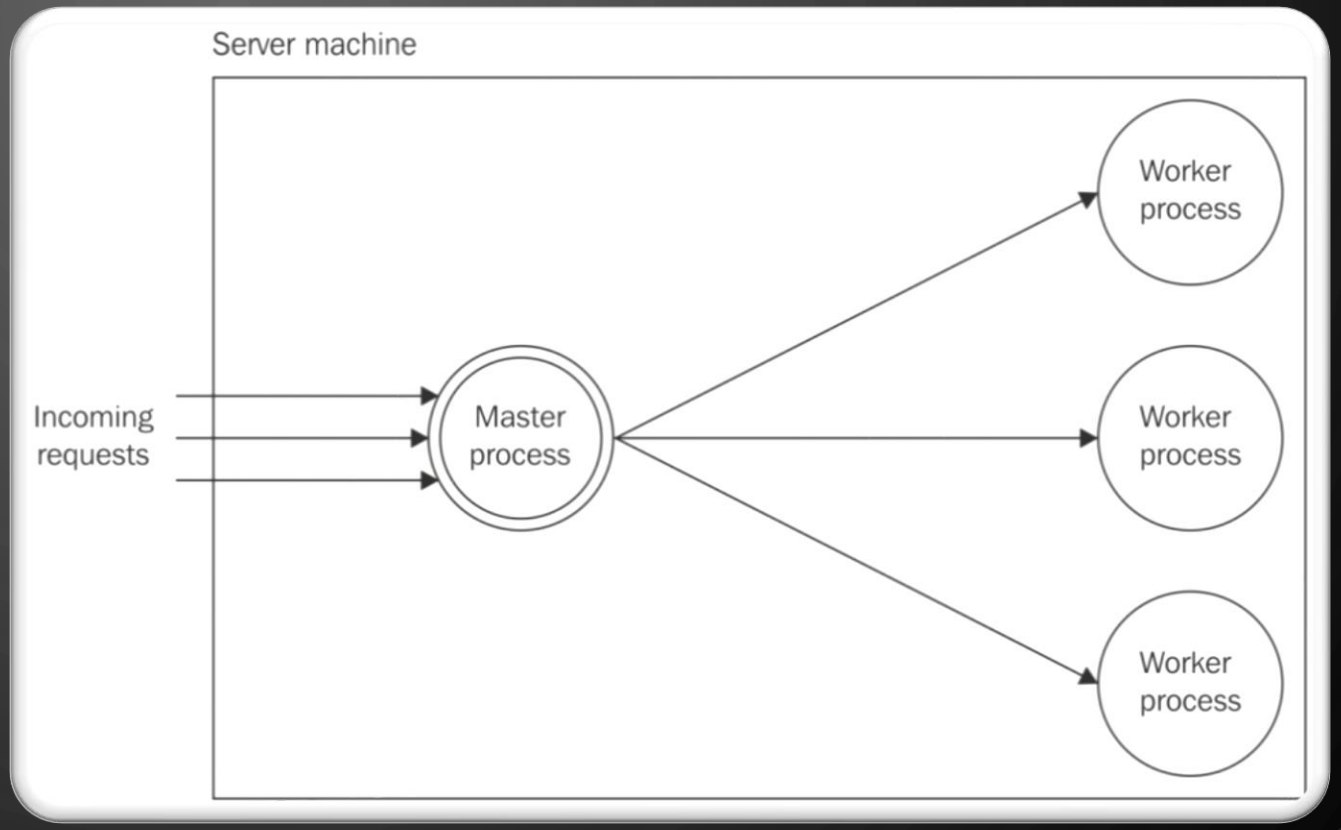
Adding more machines to the infrastructure

Node

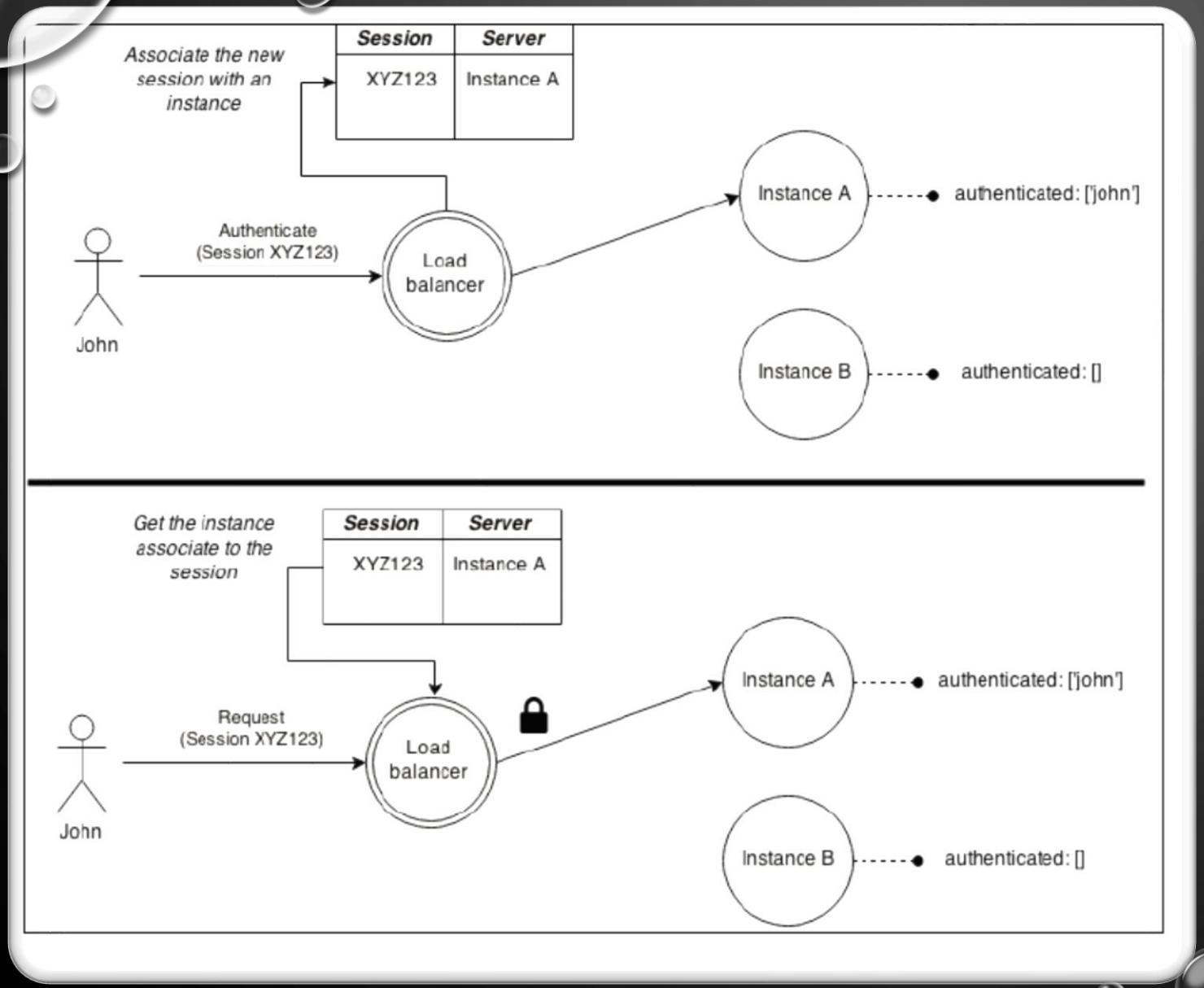
Vertical Scaling= Horizontal Scaling

CLUSTER MODULE

- Simplifies the *forking* of new instances of the same application and automatically distributes incoming connections across them
- The **Master process** is responsible for spawning a number of processes (**workers**)
- Each worker is a different Node.js process with its own event loop, memory space, and loaded modules



STICKY LOAD BALANCING



- Allows Load Balancer to route all the requests associated with a session always to the same instance of the application

REVERSE PROXY

A reverse proxy can distribute the load across several machines, not just several processes

The most popular reverse proxies on the market support sticky load balancing

A reverse proxy can route a request to any available server, regardless of its programming language or platform

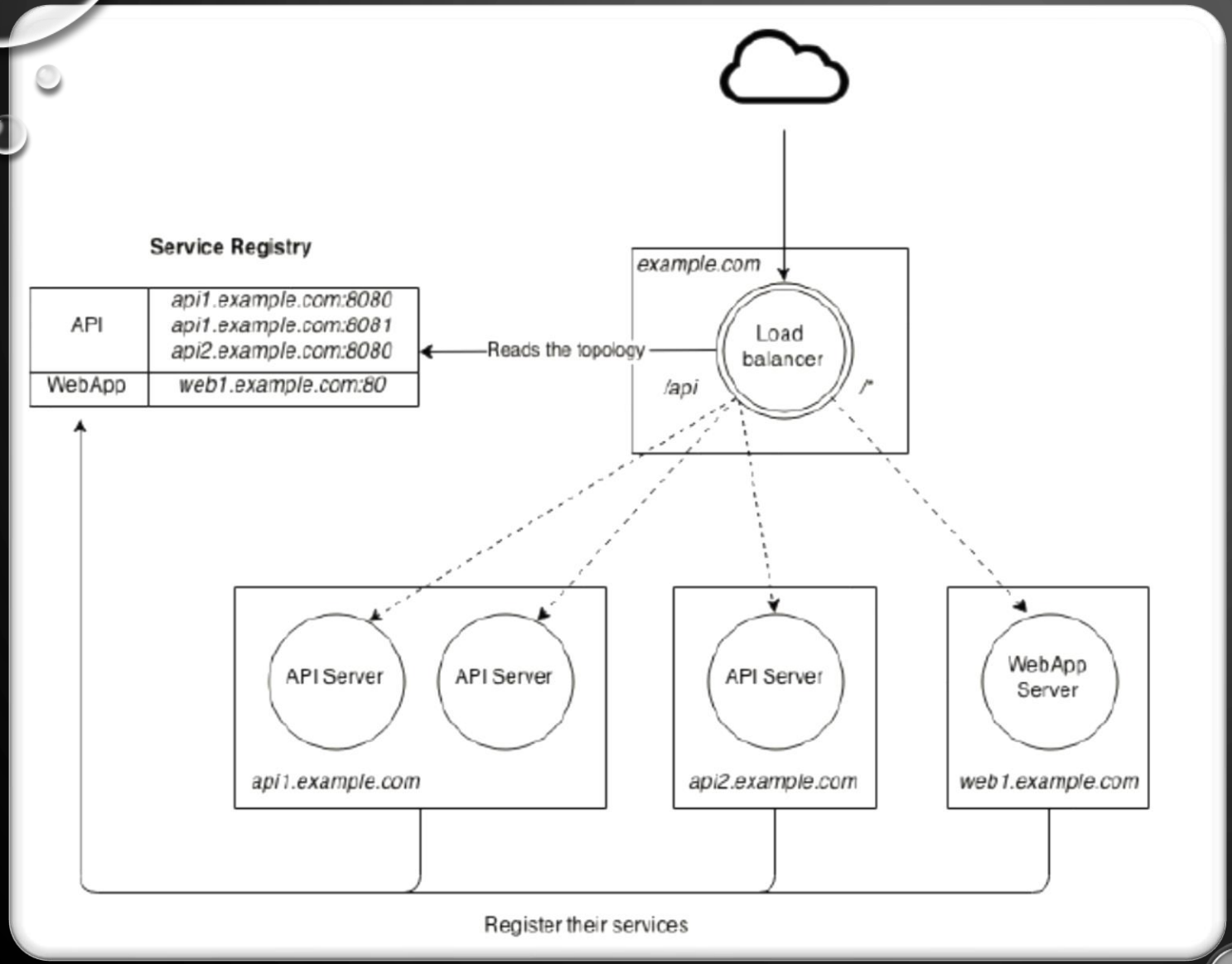
We can choose more powerful load balancing algorithms

Many reverse proxies also offer other services such as URL rewrites, caching, SSL termination point, or even the functionality of fully-fledged web servers that can be used, for example, to serve static files

REVERSE PROXY OPTIONS

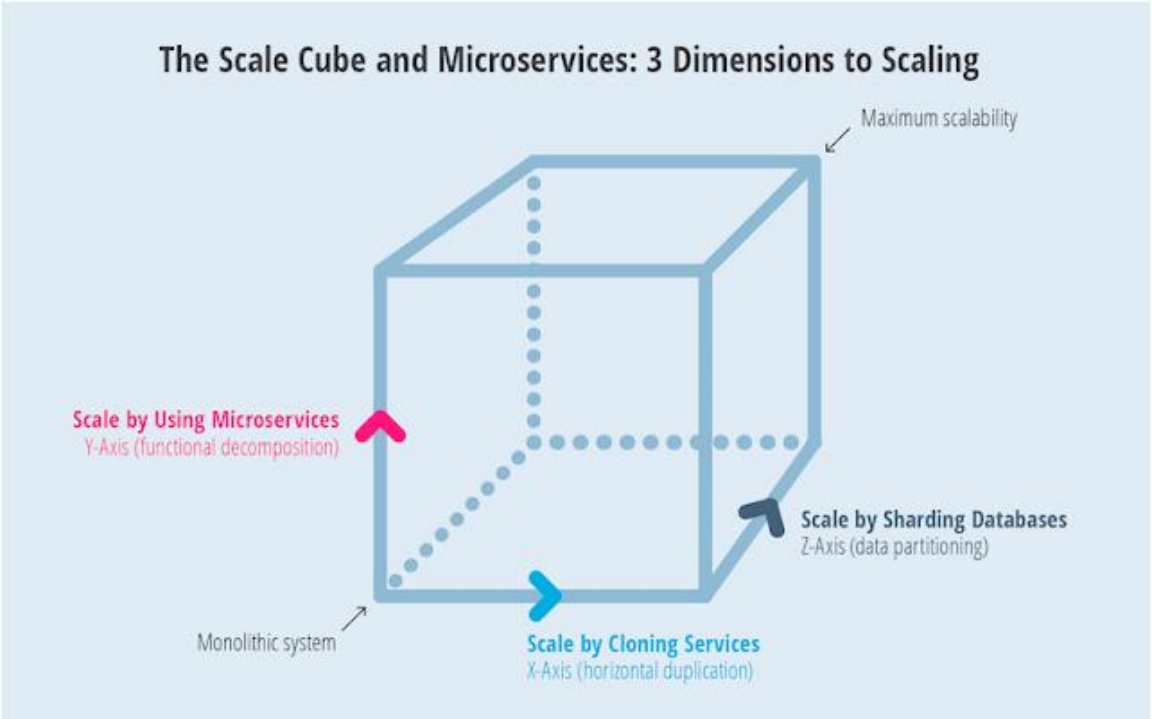
- **Nginx** (http://nginx.org): This is a web server, reverse proxy, and load balancer, built upon the non-blocking I/O model.
- **HAProxy** (http://www.haproxy.org): This is a fast load balancer for TCP/HTTP traffic.
- **Node.js-based proxies**: There are many solutions for the implementation of reverse proxies and load balancers directly in Node.js
- **Cloud-based proxies**: In the era of cloud computing, it's not rare to utilize a load balancer as-a-service. This can be convenient because it requires minimal maintenance, it's usually highly scalable, and sometimes it can support dynamic configurations to enable on-demand scalability.

SERVICE REGISTRY



- Use a central repository to store an always up-to-date view of the servers and the services available in a system.
- This pattern use to decouple a service type from the servers providing it.
- A Service Locator design pattern applied to network services.

SCALE CUBE


The Scale Cube and Microservices: 3 Dimensions to Scaling
Maximum scalability
Scale by Using Microservices
Y-Axis (functional decomposition)
Scale by Sharding Databases
Z-Axis (data partitioning)
Monolithic system
Scale by Cloning Services
X-Axis (horizontal duplication)

MESSAGING AND INTEGRATION PATTERNS

- The fundamentals of a messaging system
- The publish/subscribe pattern
- Pipelines and task distribution patterns
- Request/reply patterns

FUNDAMENTALS OF A MESSAGING SYSTEM

- The direction of the communication, which can be one-way only or a request/reply exchange
- The purpose of the message, which also determines its content
- The timing of the message, which can be sent and received immediately or at a later time (asynchronously)
- The delivery of the message, which can happen directly or via a broker

MESSAGE TYPES

Command Message

- Trigger the execution of an action or a task on the receiver
- The Command Message can be used to implement **Remote Procedure Call** (**RPC**) systems

Event Message

- **Notification**
- The use of events is a very important integration mechanism in distributed applications, as it enables us to keep all the nodes of the system on the same page.
- Long Polling and WebSockets

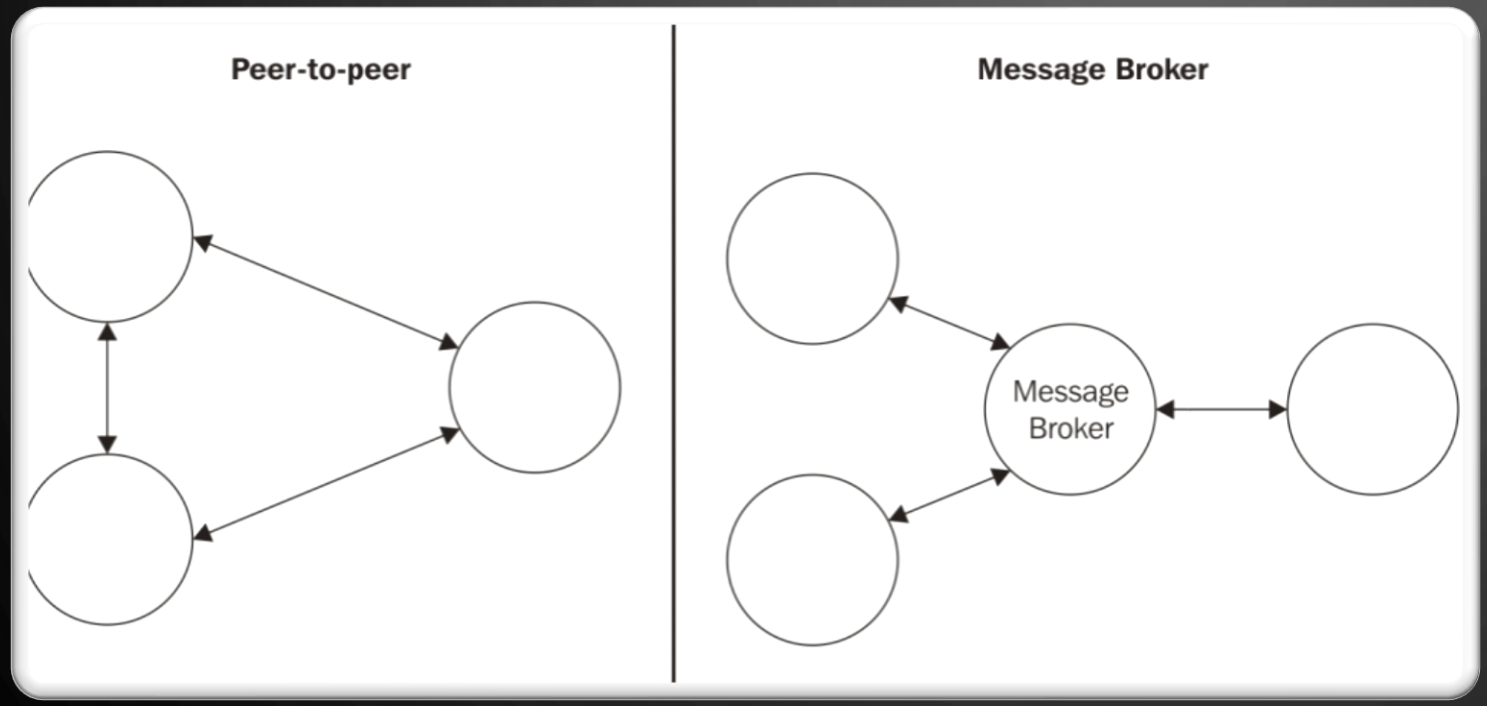
Document Message

- Reply of Command Message
- Contains Only Data – No Action or Occurrence of Event

MESSAGING PROTOCOL

- MQTT (http://mqtt.org)
  - Lightweight messaging protocol, specifically designed for machine-to-machine communications (Internet of Things).
- AMQP (http://www.amqp.org) is a more complex protocol, which is designed to be an open source alternative to proprietary messaging middleware.
- STOMP (http://stomp.github.io) is a lightweight text-based protocol, which comes from *the HTTP school of design*.
- All three are application layer protocols, and based on TCP/IP.

PEER-TO-PEER OR BROKER-BASED MESSAGING



- Removing a single point of failure
- A broker has to be scaled, while in a peer-to-peer architecture we only need to scale the single nodes
- Exchanging messages without intermediaries can greatly reduce the latency of the transmission