



Facultad de Ciencias
de la **Administración**

TECNICATURA
UNIVERSITARIA EN
**DESARROLLO
WEB**



Server side y Node.js

Semana N.º 1 – Lado del Servidor

Tecnicatura Universitaria en Desarrollo Web

Facultad de Ciencias de la Administración - Universidad Nacional de Entre Ríos

Objetivos de la clase

● Objetivos

- Conocer las responsabilidades y características de la programación del lado del servidor.
- Utilizar los métodos del protocolo HTTP para comunicar clientes y servidores web.
- Crear vistas generadas a partir plantillas HTML renderizadas en el servidor.
- Comprendan la noción de solicitud/respuesta y cómo se gestionan las rutas.

● Temas a desarrollar:

- Revisión de conceptos de Programación Orientada a Objetos, programación sincrónica / asincrónica y gestión de dependencias.
- Características de la programación del lado del servidor. Diferencias con programación del lado del cliente. Tecnologías comunes de programación del lado del servidor.
- Programación del lado del servidor usando NodeJS. Event Loop. Programación basada en eventos.
- Creación de un servidor web. Procesamiento de solicitudes HTTP.
- Introducción al framework Express.js. Disposición de recursos estáticos. Motores de plantillas: HandleBars y Pug.

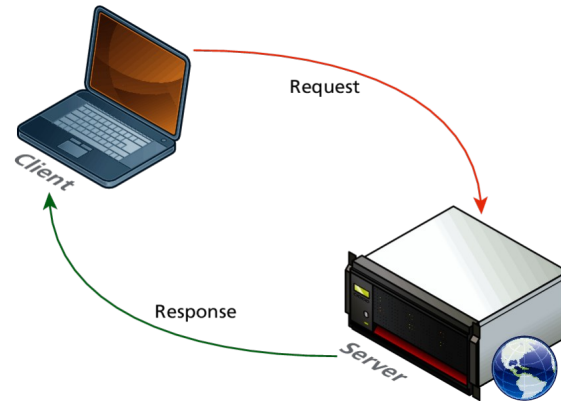
En Introducción al Desarrollo Web aprendimos...

- A estructurar documentos utilizando **HTML**.
- Dar estilo a nuestros documentos utilizando **CSS**.
- Usando **JavaScript** intercambiar información a través de solicitudes asíncronas y alterar por completo la forma en la que se muestra la página web inicialmente.
- Crear componentes reutilizables con **React**.
- Todas estas tecnologías son del lado del cliente y están orientadas a la presentación y control de la interfaz de usuario de una aplicación web.



Modelo Cliente - Servidor

- El **modelo cliente-servidor** describe la relación entre:
 - Un **cliente**: proceso corriendo en un sistema que **requiere** un servicio y
 - Un **servidor**: proceso corriendo en un sistema, que **provee** un servicio.
- La interacción (comunicación) ocurre usualmente por medio de una red.
 - Sin embargo, puede encontrarse esta relación en un escenario más acotado.



¡Join The Server Side!



Ahora... Desarrollo del lado del servidor

- Un servidor web debe alojar archivos y exponerlos para que los clientes los accedan.
- El desarrollo del lado del servidor implica el uso de lenguajes de programación tales como PHP, Java, ASP.NET ó JavaScript, para crear scripts o programas que generen contenido dinámicamente.
- Cuando programamos del lado del servidor creamos software tal y como lo haríamos en una aplicación de consola o de escritorio pero nuestros programas:
 - Corren en un servidor web.
 - Usan solicitudes y respuestas HTTP para la mayoría de las interacciones con los clientes.
- La distinción es importante ya que invalida principios tales como almacenamiento de información y manejo de memoria.

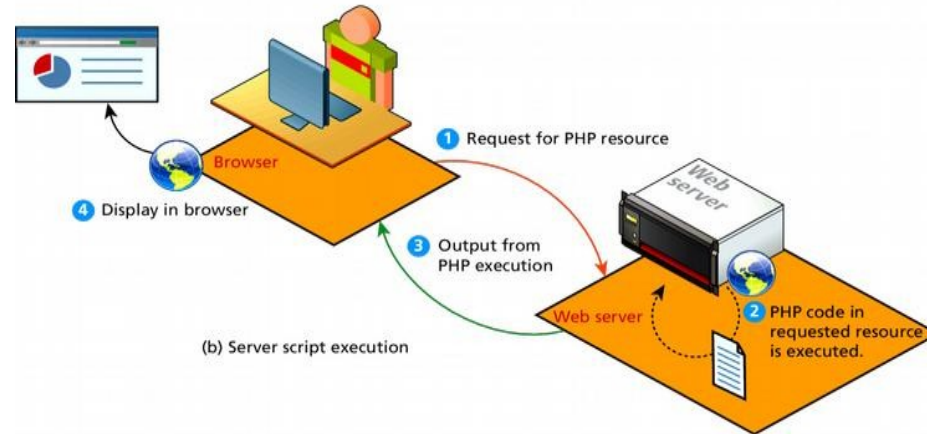
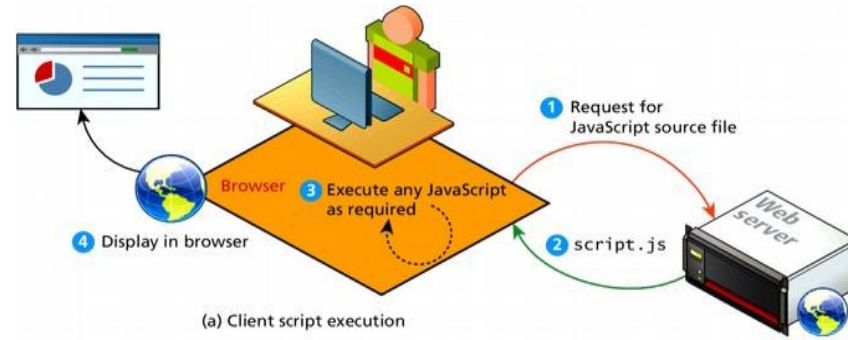
¿La programación del lado del servidor y del lado del cliente es lo mismo?

- Tienen diferentes propósitos y preocupaciones.
- Por lo general, no usan los mismos lenguajes de programación (excepto **JavaScript**).
- Se ejecutan dentro de diferentes entornos de sistemas operativos.
- El código que se ejecuta en el navegador se ocupa principalmente de mejorar la **apariencia** y el **comportamiento** de una página web representada.
 - Esto incluye seleccionar y diseñar componentes de la **interfaz de usuario**, crear diseños, navegación, validación de formularios, etc.
- Por el contrario, la **programación de sitios web del lado del servidor** implica principalmente elegir qué **contenido se devuelve al navegador en respuesta a las solicitudes**.
 - El código del lado del servidor maneja tareas como validar los datos y solicitudes enviados, usar bases de datos para almacenar y recuperar datos y enviar los datos correctos al cliente según sea necesario.
- El código del lado del cliente se escribe utilizando **HTML**, **CSS** y **JavaScript**: se ejecuta dentro de un navegador web y tiene poco o ningún acceso al sistema operativo subyacente.
- El código del lado del servidor tiene acceso completo al sistema operativo del servidor y el desarrollador puede elegir qué lenguaje de programación (y versión específica) desea utilizar.

Diferencias scripts Cliente – Servidor

- Entender donde residen los scripts y a qué pueden acceder es esencial para escribir aplicaciones web de calidad.
- Del lado del cliente:
 - Los scripts **JavaScript** son descargados por el navegador donde son ejecutados.
 - El usuario puede ver el código fuente.
 - No hay garantías que la ejecución se lleve a cabo.
 - No pueden acceder a datos del servidor si no es vía **fetch** o similar.
- Del lado del Servidor:
 - El código fuente permanece oculto al cliente mientras el servidor los procesa.
 - El cliente no puede observar el código, solo puede ver los resultados de la ejecución del script. Por ejemplo: código **HTML** o datos en formato **JSON**.
 - No es posible acceder al código **HTML** o modificar el árbol **DOM**.

Diferencias scripts Cliente – Servidor (2)



Recursos de los scripts del lado del Servidor

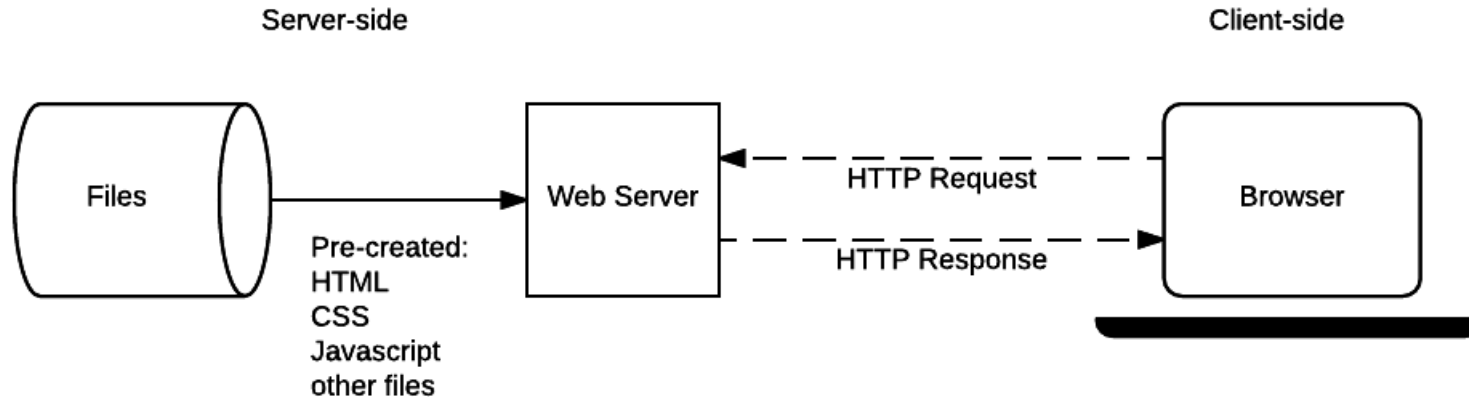
- Un script del lado del servidor puede acceder a cualquier recurso del servidor o al que este tenga acceso. Por ejemplo:
- **Almacenamiento de información:** generalmente disponible a través de la conexión a un Sistema de Gestión de Base de Datos.
 - Un DBMS es un sistema de software destinado a almacenar, recuperar y organizar grandes cantidades de datos.
 - Otra forma de almacenar información es utilizar el sistema de archivos del servidor.
- **Web Services:** comúnmente ofrecidos por proveedores externos, usan el protocolo **HTTP** para devolver datos en **XML**, **JSON** u otros formatos.
 - Extienden las funcionalidades de un sitio Web. Un ejemplo son los servicios de geolocalización que devuelven los nombres de ciudad o país a partir de coordenadas geográficas.
- **Aplicaciones de software:** pueden instalarse en el servidor y accederse a través de una conexión de red.
 - Utilizando software externo, las aplicaciones web pueden por ejemplo, enviar y recibir correos, acceder a servicios de autenticación, se puede conectar una aplicación web a una red telefónica para que envíe mensajes de texto o realice llamadas.

Responsabilidades de un Servidor Web

- Un **servidor web** tiene muchas responsabilidades:
 - Manejar conexiones **HTTP**.
 - Responder a solicitudes de recursos estáticos y dinámicos.
 - Administración de permisos y acceso a ciertos recursos.
 - Encriptación y compresión de datos.
 - Manejo de múltiples dominios y **URLs**.
 - Administración de conexiones a bases de datos.
 - Manejo de estados y cookies.
 - Carga (upload) y administración de archivos.

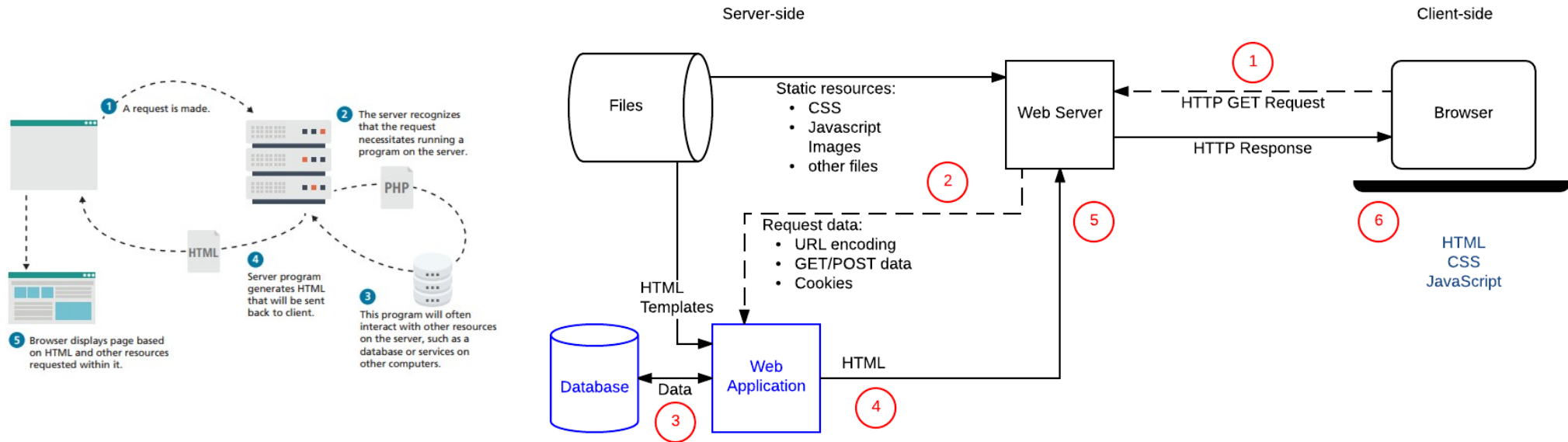
Sitios estáticos

- Un **sitio estático** es un sitio que retorna desde el servidor contenido codificado previamente.
- Cuando el usuario quiere navegar hacia una página el navegador envía al servidor una solicitud **HTTP** del tipo **GET** a una **URL**.
- El servidor recupera el documento solicitado desde su sistema de archivos y retorna una respuesta **HTTP** conteniendo el documento y una indicación de estado exitoso (**200 OK**).
- Si por alguna razón no se pudo completar se retornará un código de estado error del lado del cliente o servidor.-

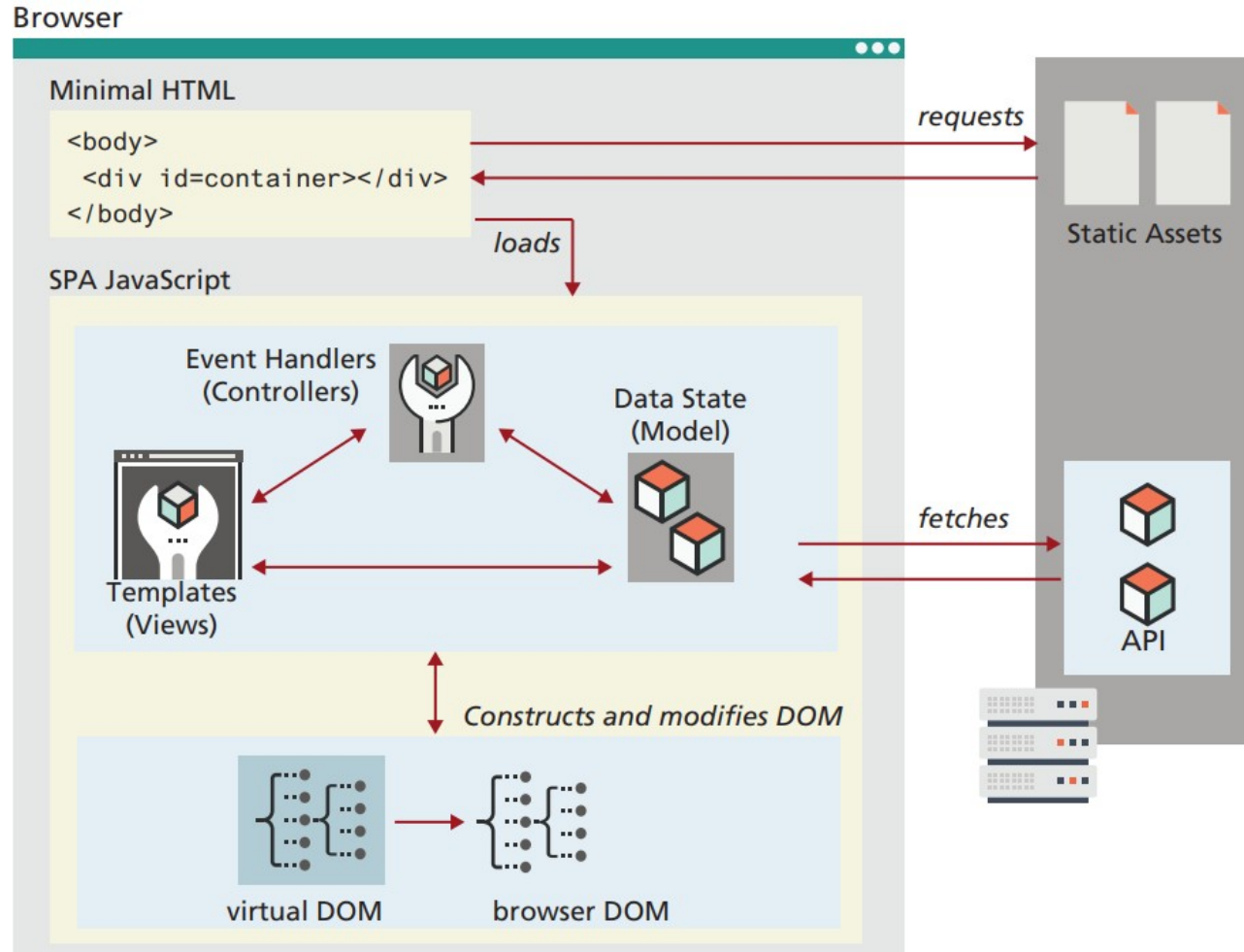


Sitio dinámico

- Un **sitio dinámico** es aquel que, cuando es requerido, genera contenido dinámicamente para la respuesta.
- En un sitio **HTML** dinámico las páginas son creadas insertando información desde la base de datos en posiciones definidas de antemano en plantillas **HTML**.
- Un sitio dinámico puede retornar diferentes datos para una misma URL basándose en información provista por el usuario o preferencias previamente almacenadas.



SPA - Usando Frameworks



- **Node.js** (o simplemente **Node**) es un entorno de ejecución asíncrono y controlado por eventos que utiliza **JavaScript**.
- Fue desarrollado por Ryan Dahl en 2009 como una mejor manera de manejar los problemas de concurrencia entre clientes y servidores.
- Es equivalente a otras tecnologías del lado del servidor como **PHP**. Una aplicación de **Node** puede generar **HTML** en respuesta a solicitudes **HTTP**, excepto que usa **JavaScript** como lenguaje de programación.
- **Node** es un **entorno de ejecución** extremadamente eficiente y eficaz. **No es un lenguaje de programación**.
 - Un **entorno de ejecución** es todo lo que se ejecuta para que podamos correr nuestros programas.
 - Otros lenguajes de programación tienen su propio entorno de ejecución:
 - Java → Java Runtime Environment (JRE).
 - .NET → Common Language Runtime (CLR).





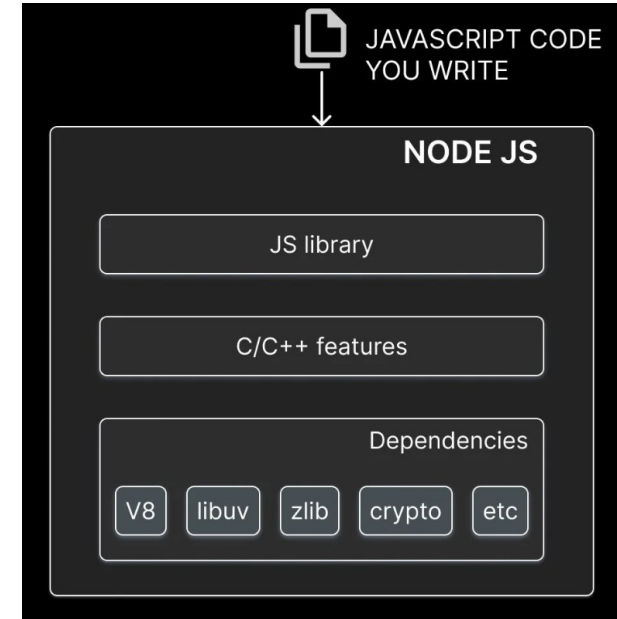
- **Node** hace uso de **V8**, el motor **JavaScript** de código abierto de Google (escrito en C++) que compila código fuente de **JavaScript** a código máquina nativo en tiempo de ejecución. Fue creado para optimizar la ejecución de **JavaScript** dentro de los navegadores.
- **JavaScript** es un lenguaje "interpretado", pero **V8** compila el código y optimiza la ejecución, permitiendo que esta se realice sobre el código compilado.
- **V8** aplica el paradigma de compilación **JIT** (Justo a Tiempo).
 - Combina compilación e interpretación haciendo las traducciones y ejecuciones más rápidas.
 - El código es ejecutado a través del intérprete y durante la ejecución se marcan los segmentos que pueden ser reutilizados (para evitar re-traducciones) así como también otras optimizaciones.
- Además **V8** proporciona **recolección de basura de objetos** en tiempo de ejecución.

Antes de comenzar...

- Recordemos algunos conceptos de **JavaScript**.
- **JavaScript es síncrono:** Si tenemos dos funciones que registran mensajes en la consola, el código se ejecuta de arriba hacia abajo, con solo una línea ejecutándose en un momento dado.
- **JavaScript es bloqueante:** Debido a su naturaleza sincrónica. No importa cuánto tiempo tarde un procesamiento, los posteriores no comenzarán hasta que termine.
- **JavaScript es single-threaded:**
 - Un **proceso** implica varias tareas que se ejecutan en un programa desde el inicio hasta el final.
 - Consta de todos los pasos que un programa realiza para ejecutarse hasta su finalización pudiendo tener uno o más hilos en su interior.
 - Un **hilo (thread)** es una unidad única de ejecución que forma parte de un proceso, como una tarea en un programa. Un **hilo** tiene un **ID**, un **conjunto de registros** y **una pila**. Un **hilo** también **comparte** su sección de código, sección de datos, recursos del sistema operativo y espacio de memoria con otros hilos en un proceso.
 - A diferencia de otros lenguajes que soportan la ejecución en múltiples hilos y, por lo tanto, pueden ejecutar múltiples tareas en paralelo, **JavaScript tiene un solo hilo llamado hilo principal para ejecutar cualquier código**.

Espera en JavaScript

- El modelo de **JavaScript** crea un problema porque tenemos que esperar a que se obtengan los datos antes de poder continuar con la ejecución del código.
- Esta espera puede tardar varios segundos, durante los cuales no podemos ejecutar ningún código adicional. Si **JavaScript** procede sin esperar, encontraremos un error.
- Necesitamos una alternativa que permita un comportamiento asíncrono en **JavaScript**. Aquí entra en juego **Node.js**.
 - En su núcleo, el entorno de **Node** consiste en tres componentes principales:
 - Dependencias externas, como **V8**, **libuv**, crypto, necesarias para el funcionamiento del entorno.
 - Funcionalidades en C++ que proporcionan acceso al sistema de archivos y redes.
 - Una biblioteca de **JavaScript** que proporciona funciones y utilidades para aprovechar las funcionalidades en C++ desde nuestro código **JavaScript**.
 - Aunque todas las partes son importantes, el componente clave para la programación asíncrona en **Node.js** es la dependencia externa, **libuv**.



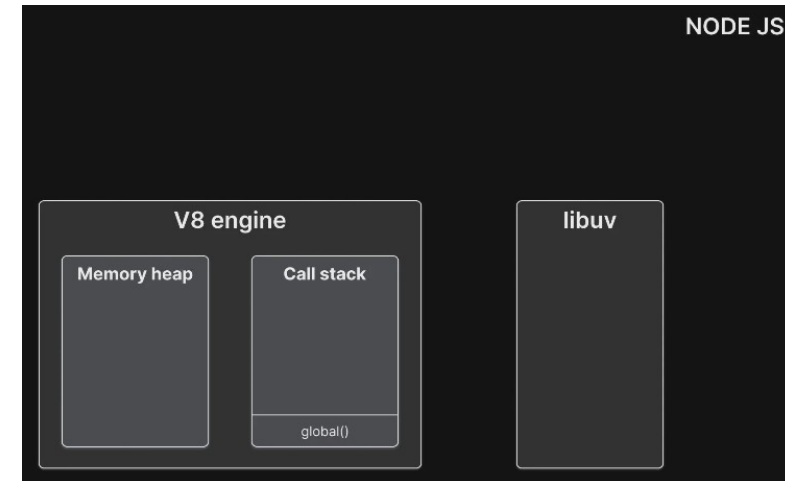


- **Libuv** es una biblioteca de código abierto y multiplataforma escrita en C. Fue inicialmente desarrollada para **Node** pero también es utilizada por **Julia**, Luvit, entre otros.
- En **Node.js**, su papel es proporcionar soporte para manejar operaciones asincrónicas.
- El motor **V8** maneja la ejecución del código **JavaScript**. **V8** consta de una región de memoria de tipo **heap (Memory Heap)** y **una pila de llamadas (Call Stack)**.
- Cada vez que declaramos variables o funciones, se asigna memoria en el **Heap**, y cada vez que ejecutamos código, las funciones se colocan en la **Call Stack**. Cuando una función retorna, se elimina de la **Call Stack**.
- En el lado derecho de la imagen, tenemos a **libuv**, que es responsable de manejar los métodos asincrónicos.
- Cuando es necesario ejecutar un método asincrónico, **libuv** se encarga de ello utilizando mecanismos asincrónicos nativos del sistema operativo. En caso de que los mecanismos nativos no estén disponibles o sean inadecuados, utiliza su propio grupo de hilos (thread pool) para ejecutar la tarea, asegurando que el hilo principal no se bloquee.

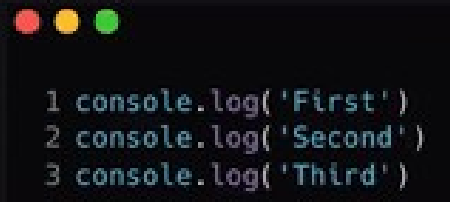
Memory Heap



Call Stack

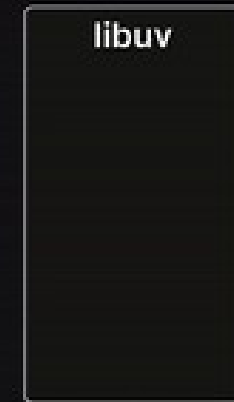
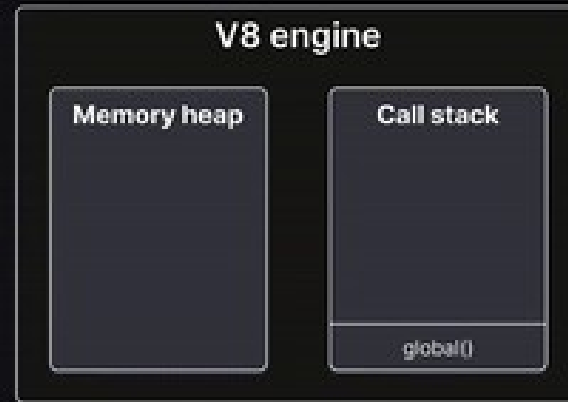
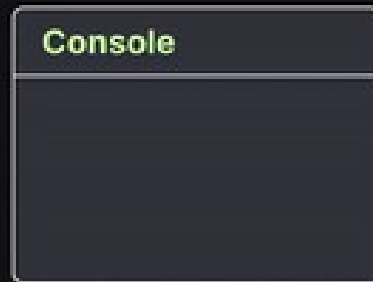


Synchronous Code Execution in Node.js



```
1 console.log('First')  
2 console.log('Second')  
3 console.log('Third')
```

Console



Asynchronous Code Execution in Node.js

```
1 console.log('First')
2 fs.readFile(__filename, () => {
3   console.log('Second')
4 })
5 console.log('Third')
```

Console

V8 engine

Memory heap

Call stack

libuv

Ejecución de código asíncrono

- El fragmento de código tiene tres sentencias de tipo `console.log()`, pero la segunda sentencia está dentro de una función de callback pasada a `fs.readFile()`.
- El hilo principal de ejecución siempre comienza en el ámbito global. La función `global` se coloca en la pila. La ejecución llega entonces a la línea 1. A los 1 ms, se registra **"First"** en la consola, y la función se elimina de la pila. La ejecución pasa luego a la línea 2. A los 2 ms, el método `readFile` se coloca en la pila. Dado que `readFile` es una operación asíncrona, se delega a `libuv`.
- **JavaScript** elimina el método `readFile` de la pila de llamadas porque su trabajo ha terminado en lo que respecta a la ejecución de la línea 3. En segundo plano, `libuv` comienza a leer el contenido del archivo en un hilo separado. A los 3 ms, **JavaScript** continúa con la línea 7, coloca el `console.log` en la pila, se registra **"Third"** en la consola, y la función se elimina de la pila.
- A aproximadamente 4 ms, supongamos que la tarea de lectura del archivo se completa en el grupo de hilos. La función de callback asociada ahora se ejecuta en la call stack. Dentro de la función de callback, se encuentra la sentencia `console.log`.
- Esa sentencia se coloca en la call stack, se muestra **"Second"** en pantalla, y la sentencia `console.log` se elimina. Como no hay más declaraciones para ejecutar en la función de callback, también se elimina. Ya no hay más código para ejecutar, por lo que la función `global` también se elimina de la pila.

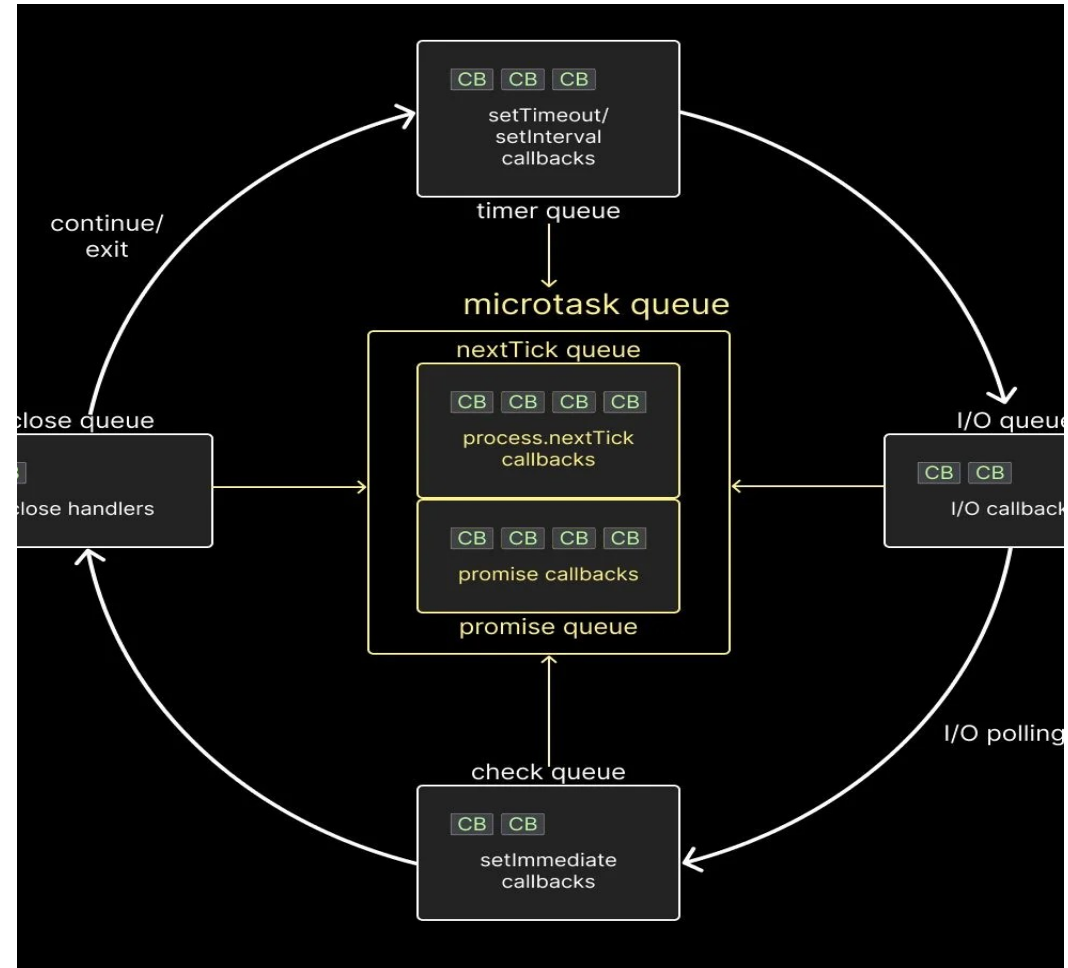
Libuv y operaciones asíncronas

- **Libuv** ayuda a manejar las operaciones asíncronas en **Node.js**.
- Para operaciones asíncronas como manejar una solicitud de red, **libuv** se basa en los mecanismos nativos del sistema operativo. Para operaciones asíncronas como leer un archivo que no tiene soporte nativo del sistema operativo, **libuv** se basa en su grupo de hilos para asegurar que el hilo principal no se bloquee.
- Sin embargo, esto plantea algunas preguntas.
 - ¿Cuándo una tarea asíncrona se completa en **libuv**, en qué momento decide **Node** ejecutar la función de callback asociada en la pila de llamadas?
 - ¿Espera **Node** a que la pila de llamadas esté vacía antes de ejecutar la función de callback, o interrumpe el flujo normal de ejecución para ejecutar la función de callback?
 - ¿Qué pasa con otros métodos asíncronos como **setTimeout** y **setInterval**, que también retrasan la ejecución de una función de callback?
 - Si dos tareas asíncronas como **setTimeout** y **readFile** se completan al mismo tiempo, ¿Cómo decide **Node** qué función de callback ejecutar primero en la call stack? ¿Una tiene prioridad sobre la otra?
- Todas estas preguntas pueden responderse comprendiendo la parte central de **libuv**, que es el **event loop**.



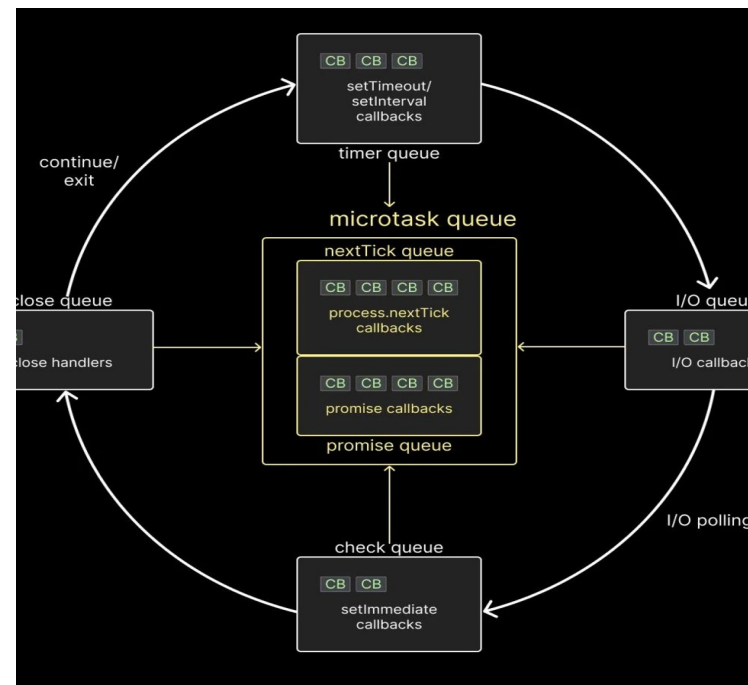
Event Loop

- El **event loop** orquesta y coordina la ejecución de código síncrono y asíncrono en **Node.js**.
- El **event loop** es un bucle que se ejecuta mientras una aplicación **Node.js** esté en funcionamiento.
- Hay seis colas diferentes en cada ciclo, cada una conteniendo una o más funciones de callback que eventualmente deben ejecutarse en la pila de llamadas.



Event Loop (2)

- Hay seis colas diferentes en cada ciclo, cada una conteniendo una o más funciones de callback que eventualmente deben ejecutarse en la call stack.
 - **Timer queue:** ó cola de temporizadores (técnicamente un min-heap), que contiene las funciones de callback asociadas con **setTimeout** y **setInterval**.
 - **I/O queue:** ó cola de E/S, que contiene las funciones de callback asociadas con todos los métodos asíncronos, como los métodos asociados con los módulos **fs** y **http**.
 - **Check queue:** o cola de verificación, que contiene las funciones de callback asociadas con la función **setImmediate**, que es específica de **Node**.
 - **Close queue:** ó cola de cierre, que contiene las funciones de callback asociadas con el evento de cierre de una tarea asíncrona.
 - **Microtask queues:** Finalmente, está la cola de microtareas, que contiene dos colas separadas:
 - La cola **nextTick**, que contiene las funciones de callback asociadas con la función **process.nextTick**.
 - La cola de **Promises**, que contiene las funciones de callback asociadas con las **Promises** nativas en JavaScript.



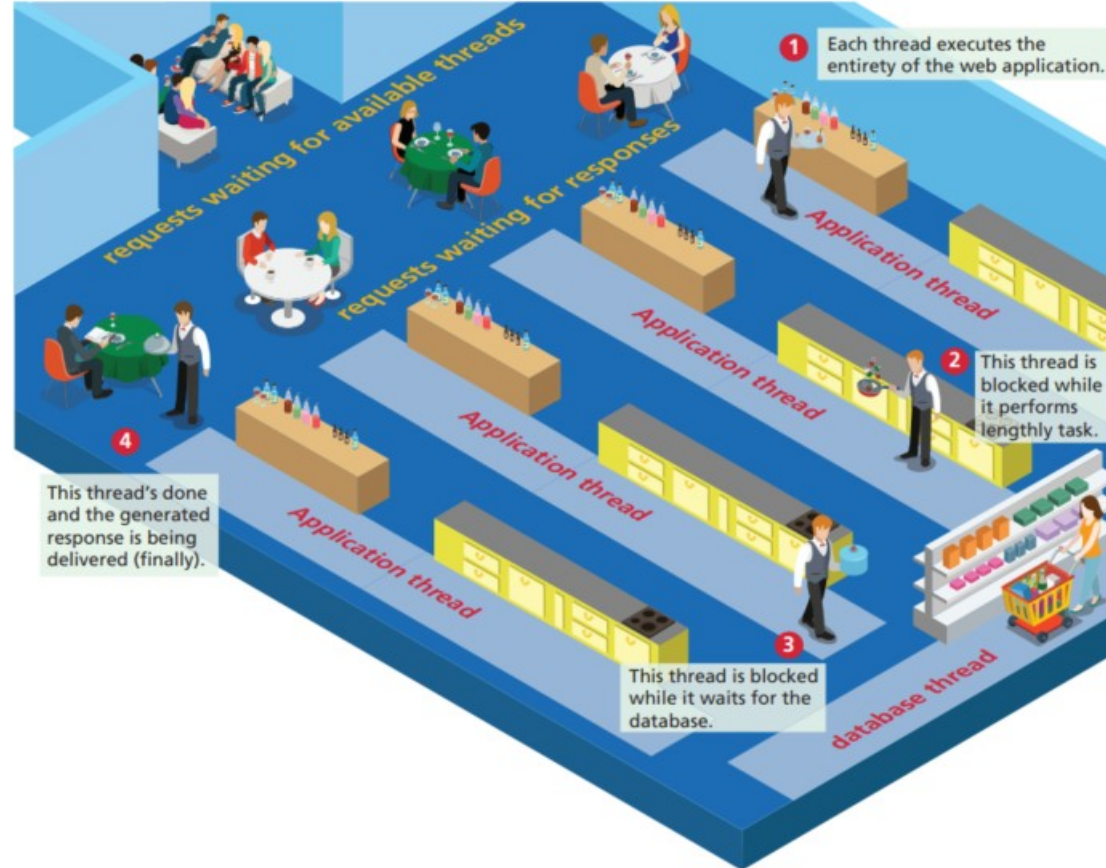
Event Loop (3) – Orden de ejecución

- A continuación explicaremos el orden de prioridad de las colas. Es importante saber que todo el **código JavaScript sincrónico** escrito por el usuario **tiene prioridad sobre el código asíncrono** que el entorno de ejecución desea ejecutar. Esto significa que solo después de que la pila de llamadas esté vacía, el event loop entra en juego.
- Dentro del **event loop**, cada fase ejecuta los callbacks programados para esa cola en particular, pero hay bastantes reglas que debes tener en cuenta:
 - 1) Se ejecutan los callbacks en la cola de **microtareas** (microtask). Primero, las tareas en la cola **nextTick** y solo entonces las tareas en la cola de **Promises**. Antes de que se comience a ejecutar una fase del event loop, se limpian primero las microtareas acumuladas hasta ese momento. Esto ocurre después de que se ejecuta una fase del loop y antes de pasar a la siguiente.
 - 2) Se ejecutan todos los callbacks dentro de la **cola de temporizadores**. Estos representan los callbacks de temporizadores que han expirado y que ahora están listos para ser procesados. Por ejemplo: **setTimeout()**, **setInterval()**.
 - 3) Los callbacks en la cola de microtareas (si están presentes) se ejecutan después de cada callback en la cola de temporizadores. Primero, las tareas en la cola **nextTick**, y luego las tareas en la cola de **Promises**.

Event Loop (4) – Orden de ejecución

- 4) Se ejecutan todos los callbacks dentro de la **cola de I/O** (fase de I/O). Por ejemplo, lectura de archivos con **fs.readFile()**. Si hay operaciones de red esta divide en dos considerando las poll
- 5) Se ejecutan los callbacks en las colas de **microtareas** (si están presentes), comenzando con la cola **nextTick** y luego la cola de Promises.
- 6) Se ejecutan todos los callbacks en la **cola de verificación** (fase de verificación).
- 7) Los callbacks en las colas de **microtareas** (si están presentes) se ejecutan después de cada callback en la cola de verificación. Primero, las tareas en la cola **nextTick**, y luego las tareas en la cola de **Promises**.
- 8) Se ejecutan todos los callbacks en la **cola de cierre** (fase de close callbacks).
- 9) Por última vez en el mismo ciclo, se ejecutan las colas de **microtareas**. Primero, las tareas en la cola **nextTick**, y luego las tareas en la cola de **Promises**.

Otras arquitecturas – Multithreading



Arquitectura de Node.js – Single threaded



Bibliografía

- **Libro:** Randy Connolly, Ricardo Hoar. ***“Fundamentals of Web Development, Global Edition”***. 3era Edition. Ed. Pearson. 2022.
- **Libro:** Mario Casciaro, Luciano Mammino. ***“Node.js Design Patterns”***. 3era Edition. Ed. Packt Pub. 2020.
- **Web:** Mozilla Developers Network. ***“Introduction to the server side”***.
URL: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction
- **Web:** Builder.io. ***“A Complete Visual Guide to Understanding the Node.js Event Loop”***. URL:
<https://www.builder.io/blog/visual-guide-to-nodejs-event-loop>