P.1. Explora el Motor V8 de Google y Node.js:

¿Qué es V8?

- Es el motor de JavaScript de código abierto de Google (usado en Chrome y Node.js). Compila JS a código máquina mediante JIT (Just- In- Time).
- Arquitectura clave: *Parser* → *Ignition* (intérprete bytecode) → *Turbofan* (optimizador JIT). Usa orquestación de garbage collection (Orinoco/Concurrent/Incremental) y estructuras internas como Hidden Classes e Inline Caches para acelerar accesos a propiedades.

Ventajas y eficiencia:

- Ejecución muy rápida (JIT + optimizaciones especulativas).
- Gestión de memoria moderna (GC incremental y concurrente) → evita "pausas" largas.
- Buen rendimiento en objetos y funciones de orden superior.

Comparación rápida con otros motores JS:

- SpiderMonkey (Mozilla): pionero, fuerte en compatibilidad (Firefox).
- JavaScriptCore (Apple): enfoque liviano y optimizaciones agresivas en Safari.
- Chakra (MS, legado): buen soporte de Windows/Edge clásico, ya no es el default.
- V8 destaca por: ecosistema enorme, rendimiento top- tier y soporte activo.

P.2. Peticiones Asíncronas y Un Único Hilo de Ejecución:

Discutir el modelo de ejecución de Node.js basado en un único hilo de ejecución. ¿Cómo maneja Node.js las peticiones asíncronas y cómo asegura que el servidor no se bloquee durante la ejecución de estas tareas?

Explicar el concepto de callback y cómo Node.js gestiona múltiples peticiones sin necesidad de utilizar varios hilos.

Modelo de ejecución (Single Thread + Event Loop):

- El hilo principal ejecuta JS; las operaciones IO (disco, red, DNS, etc.) se delegan a APIs del sistema/libuv.
- Event Loop (fases: timers → pending → idle/prepare → poll → check → close) agenda callbacks y evita bloqueos.

¿Cómo no se bloquea el servidor?

- Las operaciones largas/IO se hacen asíncronas (callbacks/promesas/async/await). El hilo principal no espera, registra un callback y continúa sirviendo otras peticiones.

Callback y gestión de múltiples peticiones:

- Callback: función que se ejecuta cuando una tarea asíncrona finaliza.
- Con Promise/async/await se vuelve más legible, pero la base es la misma: no bloquear el event loop.

P.3. Delegación de Tareas a Procesos Externos:

Discutir cómo Node.js delega ciertas tareas, especialmente aquellas que requieren procesamiento intensivo, a procesos externos. ¿Qué mecanismos utiliza Node.js para evitar que el hilo principal se vea bloqueado por estas tareas?

¿Qué tecnologías o herramientas se pueden utilizar junto con Node.js para gestionar estas tareas?

¿Qué delega Node.js?

- IO (red, archivos), DNS, timers: gestionados por libuv y el sistema operativo.
- Tareas CPU- intensivas: usando Worker Threads o child_process (procesos/hilos separados) para no bloquear el loop.

Mecanismos y herramientas:

- Worker Threads (mismo proceso, distintos hilos; comparten memoria mediante SharedArrayBuffer).
- Child_ process / cluster (multiproceso; aisla fallos, escala en múltiples núcleos).
- Queues (BullMQ/RabbitMQ/SQS) + Jobs (workers) para trabajos diferidos.

En resumen, Node.js mantiene un event loop single-threaded, pero gracias a libuv delega operaciones de I/O al *thread pool* y al sistema operativo, evitando bloqueos en el hilo principal. Para tareas de alto costo en CPU se pueden usar Worker Threads o Child Process/Cluster, y en escenarios más grandes se apoya en colas de trabajos externas (como BullMQ o RabbitMQ), lo que permite aprovechar múltiples núcleos y mantener fluida la aplicación.

P.4. ¿Qué podemos hacer con Node.js?:

Presentar un listado de al menos 5 cosas que se pueden hacer con Node.js, desde la creación de aplicaciones web hasta la interacción con bases de datos y la disponibilidad de servicios API REST.

Discutir las ventajas de usar Node.js en comparación con otros entornos de backend.

Cosas que se pueden hacer con Node.js:

1. APIs REST/GraphQL (Express, Fastify, Nest).
2. Aplicaciones en tiempo real (sockets/chat/notificaciones con Socket.IO).
3. SSR y webs (Next.js/Nuxt via adapters).
4. CLI y automatización (scripts, generadores, linters).
5. Integración/ETL (consumo de APIs, pipelines de datos).
6. Bots y microservicios (Discord/Telegram, colas de trabajo).

Ventajas frente a otros backends:

1. Un solo lenguaje (JS/TS) en front-end y back-end.
2. Alto rendimiento IO y concurrencia eficiente.
3. Ecosistema NPM enorme y comunidad activa.

P.5. Uso de Express en Node.js:

Explicar qué es Express y por qué es comúnmente usado para el desarrollo de aplicaciones web y API REST en Node.js.

Comparar Express con Node.js puro en términos de características, velocidad de desarrollo y flexibilidad.

Discutir las ventajas de usar Express en lugar de escribir todo el código desde cero en Node.js, especialmente en términos de simplificación del proceso de desarrollo.

Express es un framework desarrollado para Node.js, que se destaca por ser minimalista y flexible. Es ampliamente usado para el desarrollo de aplicaciones web y APIs REST porque tiene una sintaxis sencilla y facilita mucho la configuración de servidores, creación de middlewares y definición de rutas. También cuenta con una gran integración con motores de plantillas y es rápido y fácil de usar.

Mientras Node.js hace posible ejecutar Javascript del lado del servidor, Express nos brinda una caja de herramientas para facilitar la creación de aplicaciones web, permitiéndonos gestionar solicitudes HTTP, establecer middlewares y definir rutas con facilidad.

En conclusión, es preferible usar Express por la simplicidad que nos brinda al configurar el entorno de un servidor y sus protocolos de comunicación; por su minimalismo, que hace que nuestra aplicación sea escalable según nuestras necesidades; y por su flexibilidad, que nos permite configurar nuevos procesos en pocos pasos. Todos estos factores agilizan enormemente la creación de una aplicación web, en comparación a escribir código desde cero con Node.js.

P.6 Middleware en Express:

Investigar y explicar qué es un middleware en Express y cómo se utiliza en el desarrollo de aplicaciones.

Presentar ejemplos de patrones básicos de middleware y cómo se utilizan en Express para realizar tareas como autenticación, manejo de errores y procesamiento de solicitudes.

Discutir el ciclo de vida de una petición en una aplicación Express, incluyendo el paso por los middlewares.

El concepto de **middlewares** es popular en muchos frameworks de Desarrollo Web. Están, por ejemplo, los que dependen fuertemente del concepto como Express, o los que lo usan detrás de cámaras o como una configuración avanzada, como lo hace Ruby on Rails, etc.

Un **Middleware** tiene como propósito tomar dos piezas de la aplicación y conectarlas, como un puente en el que fluye la información. Normalmente decimos que una rutina de código tiene como propósito recibir información y retornarla transformada, la única característica especial de un Middleware es que la información la obtiene de otra función de código para luego enviársela a una función distinta más.

Los middlewares se montan por múltiples razones, una de ellas por ejemplo es validar la información antes de que llegue a la rutina que enviará respuesta hacia el cliente, también pueden usarse para hacer una consulta y guardar información antes de que pase a las funciones que responderán.

Un middleware en Express es una función cuyo distintivo es que recibe 3 argumentos:
function(req,res,next){}

- **Los primeros dos argumentos**, como cualquier función que responde peticiones del cliente, contiene la información de la solicitud en el primer argumento Request, y el objeto Response como segundo argumento, que nos sirve para modificar la respuesta que se enviará al usuario.

- **El tercer argumento** es muy importante, este es el que distingue un middleware de una función de respuesta. Este tercer argumento es una función que contiene el siguiente middleware o función a ejecutar luego de que el actual termine su ejecución.

Algunos ejemplos de uso que podemos presentar:

- **Autenticación de usuarios.**

```
function authMiddleware(req,  res, next) {
        const token = req.headers['authorization'];
        if (token === '12345') { // Ejemplo: validación simple
            next(); // pasa al siguiente middleware o ruta
         } else {
           res.status(401).json({ mensaje: 'No autorizado' });
         }
}

module.exports = authMiddleware;
```

- **Manejo de errores.**

```
function manejarErrores(err, req, res, next) {
   console.error(err.stack);
   res.status(500).json({
      error: 'Error interno del servidor',
      mensaje: err.message
   });
}

module.exports = manejarErrores;
```

- **Procesamiento de datos de entrada.**

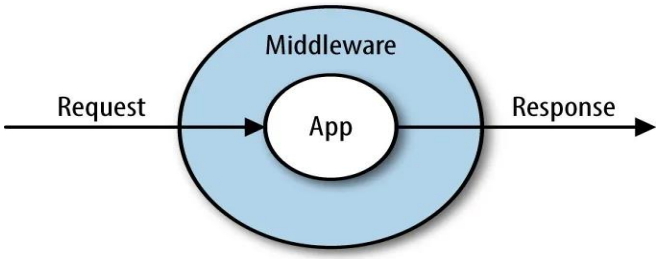
```
function loggerMiddleware(req,  res, next) {
          console.log(`${req.method} ${req.url}`);
           next(); // continúa hacia el siguiente middleware o ruta
}

module.exports = loggerMiddleware;
```

- **Middleware de Logging y Monitoreo**

```
javascriptconst logger = (req, res, next) => {
   const inicio = Date.now();
   // Interceptar cuando termine la respuesta
   res.on('finish', () => {
      const duracion = Date.now() - inicio;
      console.log(
         `${req.method} ${req.url} - ${res.statusCode} - ${duracion}ms - ${req.ip}`
      );
   });
   next();
};
```

app.use(logger);

El ciclo de vida de la petición:
1. La solicitud llega al servidor.
2. Pasa por los middlewares configurados.
3. Se ejecuta la función de la ruta correspondiente.
4. Se envía la respuesta al cliente.



P. 7. Renderización Dinámica en Express:

Explicar cómo Express puede utilizar un motor de plantillas para renderizar contenido dinámico.

¿Cuáles son las ventajas de usar renderización dinámica frente a una estática? ¿Cuáles son los casos de uso comunes para esta técnica?

Express permite usar motores de plantillas (EJS, Pug, Handlebars, etc.) para generar HTML dinámico en el servidor antes de enviarlo al navegador. Esto significa que no entregamos solo archivos .html estáticos, sino que combinamos plantillas + datos para generar vistas personalizadas.

Ventajas de la renderización dinámica vs estática:

Renderización Estática	**Renderización Dinámica**
Archivos html fijos en el servidor.	HTML se genera “al vuelo” combinando plantilla + datos.
No se adapta al usuario ni contexto.	Se personaliza según usuario, sesión, parámetros, etc.
Rápido para contenido que nunca cambia.	Ideal cuando los datos provienen de BD o API.
Ej: página de políticas, landing estática.	Ej: panel de usuario, lista de productos, dashboards.

Casos de uso comunes:
1. **Perfiles de usuario** → cada usuario ve información personalizada.
2. **E-commerce** → renderizar productos con precios, imágenes y stock.
3. **Dashboards** → mostrar datos en tiempo real de una base de datos.
4. **Sistemas de noticias o blogs** → generar páginas a partir de artículos almacenados.
5. **Aplicaciones multi-idioma** → cargar dinámicamente las traducciones.

P.8. Desarrollo de una API REST con Express:

Discutir cómo se desarrolla una API RESTful usando Express.

¿Cuáles son los principales métodos HTTP utilizados en una API REST (GET, POST, PUT, DELETE)? ¿Cómo se gestionan en Express?

Una API es un conjunto de reglas que manejan la interacción entre aplicaciones y que nos permiten intercambiar datos entre cliente y servidor.

Para el desarrollo de APIs RESTful en Express, se requieren 3 pasos principales:

1. Configuración del servidor en Express.
2. Definición de rutas.
3. Uso de métodos HTTP (GET, POST, PUT Y DELETE) para la configuración de las acciones deseadas.

Funciones de los métodos HTTP:

1. GET: obtener una información o dato.
2. POST: crear un nuevo recurso.
3. PUT: actualizar un recurso existente.
4. DELETE: eliminar un recurso.

P.9. Conexión a Bases de Datos y Consumo de API REST:

Discutir cómo Node.js puede conectarse a bases de datos (por ejemplo, MongoDB, MySQL, PostgreSQL).

Explicar cómo Node.js consume APIs REST externas. ¿Qué bibliotecas y técnicas son útiles para consumir datos de APIs REST en una aplicación Node.js?

Node.js puede comunicarse con bases de datos utilizando **drivers específicos** que cada sistema de gestión de base de datos (DBMS) ofrece. Por ejemplo, mysql2 para MySQL, pg para PostgreSQL o mongoose para MongoDB. Estos paquetes permiten que Node.js ejecute consultas, inserte registros y obtenga información de manera directa desde el servidor, todo a través de código JavaScript. La instalación se realiza con **NPM (Node Package Manager)**, lo que facilita agregar y gestionar dependencias en el proyecto.
En cuanto al consumo de **APIs REST externas**, Node.js puede usar cualquier librería que haga solicitudes HTTP. Las más comunes son **node-fetch** y **axios**, que permiten enviar peticiones (GET, POST, PUT, DELETE) y trabajar fácilmente con las respuestas en formato JSON. Esto es muy útil cuando la aplicación necesita conectarse con servicios externos, como sistemas de pagos, redes sociales o datos abiertos de terceros.

P.10. Depuración de Código en Node.js:

Explicar las mejores prácticas para depurar código en Node.js y Express.
Presentar herramientas de depuración comunes como el debugger de Node.js y el uso de console.log().

Para depurar código en Node.js y Express, se pueden usar comandos como console.log, console.error o console.warn, que nos permiten imprimir información sobre el flujo de ejecución de código y valor de las variables, levantar errores y advertencias.

Además, Node.js incluye un debugger propio, que permite ejecutar líneas de código por pasos e inspeccionar variables.
También, Node.js puede integrarse con editores de código que cuentan con herramientas de depuración visuales, que permiten establecer puntos de quiebre (breakpoints) y seguir el flujo de la ejecución de código desde la aplicación.

Por último, podemos valernos de herramientas de monitoreo en tiempo real compatibles con Node.js, como “PM2” o “nodemon” para reiniciar el código si se detectan cambios o errores en éste.

¿Cuáles son las principales dificultades que enfrentan los desarrolladores al depurar aplicaciones Node.js y cómo pueden superarlas?

Depurar en Node.js suele ser difícil por el manejo de la asincronía: los callbacks y promesas hacen que el flujo de ejecución no sea tan claro y los errores sean más difíciles de rastrear. También las trazas de error a veces resultan poco informativas y los problemas de rendimiento o memoria no siempre se reproducen fuera de producción.

Para superar esto se suele recurrir a async/await (para simplificar la lectura del código), a buenos sistemas de logging con contexto, y a las herramientas que trae el propio Node (--inspect, Chrome DevTools, clinic.js). Además, mantener entornos de desarrollo y producción lo más parecidos posible ayuda a reproducir fallos y resolverlos más rápido.