# Table of Content

# Table of Contents

# Big(o)

1. ¿Qué es la notación Big O y por qué es importante en el análisis de algoritmos?
   Big O describe el comportamiento del tiempo o el espacio de un algoritmo en función de la entrada, ayudando a medir su eficiencia.
2. ¿Qué significa O(1) en Big O Notation?
   O(1) indica que el tiempo o espacio requerido por el algoritmo es constante, independientemente del tamaño de la entrada.
3. ¿Qué significa O(n) en Big O Notation?
   O(n) significa que el tiempo o espacio requerido por el algoritmo crece linealmente en proporción al tamaño de la entrada.
4. ¿Qué significa O(n²) y cuándo suele encontrarse este tipo de complejidad?
   O(n²) indica que el tiempo o espacio crece cuadráticamente en relación con el tamaño de la entrada, común en algoritmos de fuerza bruta o de ordenación ineficientes como el bubble sort.
5. ¿Qué es O(log n) y en qué tipo de algoritmos suele aparecer?
   O(log n) representa una complejidad logarítmica, común en algoritmos que dividen la entrada en cada iteración, como la búsqueda binaria.
6. ¿Cuál es la diferencia entre O(n) y O(log n)?
   O(n) crece linealmente con el tamaño de la entrada, mientras que O(log n) crece mucho más lentamente, dividiendo la entrada en cada paso.
7. ¿Qué es O(n log n) y en qué contextos suele aparecer?
   O(n log n) es una combinación de crecimiento lineal y logarítmico, común en algoritmos de ordenación eficientes como merge sort y quicksort.
8. ¿Cómo se compara O(n²) con O(n log n) en términos de eficiencia?
   O(n²) crece más rápido que O(n log n), lo que lo hace menos eficiente para grandes entradas.
9. ¿Qué es el peor caso en el análisis de complejidad y cómo se representa en Big O?
   El peor caso describe el escenario en el que un algoritmo tiene el peor rendimiento posible, y se representa usando la notación Big O para mostrar la mayor cantidad de recursos necesarios.

10. ¿Qué significa O(2^n) y qué tipo de algoritmos suelen tener esta complejidad?
    O(2^n) describe un crecimiento exponencial, típico en algoritmos que exploran todas las combinaciones posibles, como los algoritmos de backtracking.
11. ¿Qué diferencia hay entre el mejor caso y el peor caso en Big O Notation?
    El mejor caso describe el rendimiento óptimo de un algoritmo, mientras que el peor caso indica el escenario donde el algoritmo usa más recursos.
12. ¿Qué es la complejidad de espacio en Big O y cómo difiere de la complejidad temporal?
    La complejidad de espacio mide cuánta memoria utiliza un algoritmo, mientras que la complejidad temporal mide cuánto tiempo toma ejecutarse.
13. ¿Qué significa O(n!) y en qué situaciones se puede encontrar esta complejidad?
    O(n!) describe un crecimiento factorial, característico de algoritmos que generan todas las permutaciones posibles, como el algoritmo de resolución de problemas combinatorios.
14. ¿Qué es la notación Ω (Omega) y cómo se relaciona con Big O?
    Ω (Omega) describe el límite inferior de la complejidad de un algoritmo, representando el mejor caso posible, mientras que Big O describe el límite superior.
15. ¿Qué es la notación Θ (Theta) y cómo se diferencia de Big O y Ω?
    Θ (Theta) describe una cota ajustada, indicando que el tiempo de ejecución de un algoritmo está acotado tanto por arriba como por abajo por una función específica.
16. ¿Cómo afecta el tamaño de la entrada a la complejidad de un algoritmo descrito por O(1)?
    El tamaño de la entrada no afecta a un algoritmo O(1), ya que su tiempo o espacio permanece constante.
17. ¿Cómo afecta la recursividad a la complejidad de un algoritmo en Big O?
    La recursividad puede incrementar la complejidad de un algoritmo, especialmente si cada llamada recursiva genera nuevas operaciones o llamadas adicionales.
18. ¿Qué significa amortized time complexity en Big O Notation?
    La complejidad amortizada mide el costo promedio de una operación en una secuencia de operaciones, distribuyendo el costo de operaciones costosas a lo largo de varias más baratas.
19. ¿Cómo se representa la complejidad de un algoritmo que tiene múltiples entradas con diferentes tamaños?
    Cuando un algoritmo tiene varias entradas, su complejidad se representa con más de una variable, por ejemplo, O(m * n) si hay dos entradas de tamaños diferentes.
20. ¿Qué es el teorema maestro y cómo se utiliza para analizar la complejidad de algoritmos recursivos?
    El teorema maestro proporciona una fórmula para calcular la complejidad de algoritmos recursivos que siguen una forma particular de división y conquista.
21. ¿Qué significa O(√n) y cuándo se suele encontrar esta complejidad?
    O(√n) indica que el algoritmo tiene un crecimiento proporcional a la raíz cuadrada del tamaño de la entrada, común en ciertos algoritmos de búsqueda y factorización.

22. ¿Cómo afecta el uso de estructuras de datos a la complejidad en Big O?
    Las diferentes estructuras de datos pueden cambiar la complejidad de las operaciones básicas (inserción, eliminación, búsqueda), afectando el rendimiento general del algoritmo.
23. ¿Qué es la complejidad polinómica y cómo se compara con la complejidad exponencial en Big O?
    La complejidad polinómica (O(n^k)) crece más lentamente que la exponencial (O(2^n)), lo que hace que los algoritmos polinómicos sean más eficientes para entradas grandes.
24. ¿Qué impacto tiene la constante oculta en Big O Notation?
    Aunque Big O ignora las constantes, en la práctica pueden tener un impacto significativo en el rendimiento de un algoritmo, especialmente para entradas pequeñas.
25. ¿Qué es la complejidad sublineal y en qué contextos se puede encontrar?
    La complejidad sublineal (por ejemplo, O(log n)) indica que el tiempo de ejecución crece más lentamente que el tamaño de la entrada, lo que es típico en algoritmos de búsqueda eficientes.
26. ¿Qué significa decir que un algoritmo es escalable en términos de Big O?
    Un algoritmo escalable mantiene un crecimiento razonable en su tiempo o espacio conforme aumenta el tamaño de la entrada, siendo capaz de manejar grandes conjuntos de datos.
27. ¿Cómo se puede mejorar la complejidad temporal de un algoritmo de O(n²) a O(n log n)?
    Optimizando el algoritmo, por ejemplo, utilizando una técnica de divide y vencerás, como en el caso de mejorar bubble sort a merge sort.
28. ¿Cómo afecta el uso de hash tables a la complejidad en Big O?
    Las tablas hash pueden ofrecer una complejidad promedio de O(1) para operaciones de búsqueda, inserción y eliminación, mejorando significativamente el rendimiento.
29. ¿Cómo se analiza la complejidad de un algoritmo que incluye varias operaciones con diferentes complejidades?
    Se considera la complejidad de la operación más costosa, ya que dominará el comportamiento global del algoritmo.
30. ¿Cuál es la diferencia entre la complejidad asintótica y la complejidad práctica de un algoritmo?
    La complejidad asintótica describe el comportamiento del algoritmo en entradas extremadamente grandes, mientras que la complejidad práctica toma en cuenta factores como las constantes y la arquitectura del sistema en entradas reales.

# 1. Jr

## SQL (MySQL)

1. ¿Qué es una base de datos relacional?
   Es un conjunto de datos organizados en tablas que se relacionan entre sí a través de

claves primarias y claves foráneas.

2. ¿Qué es una tabla en una base de datos relacional?
   Una tabla es una colección de datos organizados en filas y columnas, donde cada fila representa un registro y cada columna, un atributo.
3. ¿Qué es una clave primaria (Primary Key) en MySQL?
   Es un campo único que identifica de manera exclusiva cada registro en una tabla.
4. ¿Qué es una clave foránea (Foreign Key) en MySQL?
   Es un campo en una tabla que se refiere a la clave primaria de otra tabla, estableciendo una relación entre ellas.
5. ¿Qué es una consulta (query) en SQL?
   Es una instrucción escrita en lenguaje SQL que se utiliza para recuperar, insertar, actualizar o eliminar datos en una base de datos.
6. ¿Qué hace la instrucción `SELECT` en MySQL?
   Se utiliza para recuperar datos de una o más tablas en la base de datos.
7. ¿Qué hace la instrucción `INSERT INTO` en MySQL?
   Se utiliza para agregar nuevos registros a una tabla.
8. ¿Cuál es la diferencia entre `WHERE` y `HAVING` en MySQL?
   `WHERE` filtra filas antes de la agregación, mientras que `HAVING` filtra filas después de la agregación en consultas que utilizan funciones como `GROUP BY`.
9. ¿Qué es una `JOIN` en MySQL?
   Es una operación que combina filas de dos o más tablas basándose en una condición relacionada.
10. ¿Cuál es la diferencia entre `INNER JOIN` y `LEFT JOIN`?
    `INNER JOIN` devuelve solo las filas que tienen coincidencias en ambas tablas, mientras que `LEFT JOIN` devuelve todas las filas de la tabla izquierda, incluso si no hay coincidencias en la tabla derecha.
11. ¿Qué hace la instrucción `UPDATE` en MySQL?
    Se utiliza para modificar los datos de uno o más registros existentes en una tabla.
12. ¿Qué hace la instrucción `DELETE` en MySQL?
    Se utiliza para eliminar registros de una tabla.
13. ¿Qué es la normalización en bases de datos?
    Es el proceso de estructurar las tablas para minimizar la redundancia y evitar problemas de inconsistencia de datos.
14. ¿Qué es una transacción en MySQL?
    Es una secuencia de operaciones que se ejecutan como una sola unidad lógica. Si alguna operación falla, se revierte todo el conjunto.
15. ¿Qué es un índice en MySQL y por qué es importante?
    Un índice es una estructura que mejora la velocidad de recuperación de datos en una tabla, pero puede aumentar el tiempo de inserción, actualización y eliminación.

---

# NoSQL (MongoDB)

16. ¿Qué es MongoDB?
    Es una base de datos NoSQL orientada a documentos que almacena los datos en formato BSON (similar a JSON).
17. ¿Qué es una colección en MongoDB?
    Es un conjunto de documentos en MongoDB, equivalente a una tabla en bases de datos relacionales.
18. ¿Qué es un documento en MongoDB?
    Es una estructura de datos en formato BSON (similar a JSON) que contiene datos y sus relaciones.
19. ¿Qué significa que MongoDB sea una base de datos "sin esquema"?
    Significa que los documentos dentro de una colección pueden tener diferentes estructuras, sin necesidad de definir un esquema fijo.
20. ¿Qué hace la instrucción `find()` en MongoDB?
    Se utiliza para recuperar documentos de una colección.
21. ¿Cuál es la diferencia entre `find()` y `findOne()` en MongoDB?
    `find()` recupera todos los documentos que coinciden con una consulta, mientras que `findOne()` devuelve solo el primer documento que coincide.
22. ¿Qué es un índice en MongoDB y por qué se utiliza?
    Un índice en MongoDB acelera las consultas sobre un campo específico, mejorando el rendimiento de búsqueda.
23. ¿Cómo se agrega un nuevo documento a una colección en MongoDB?
    Utilizando la instrucción `insertOne()` para un solo documento o `insertMany()` para varios documentos.
24. ¿Qué hace la instrucción `updateOne()` en MongoDB?
    Se utiliza para actualizar un único documento que coincida con una condición específica.
25. ¿Qué hace la instrucción `deleteOne()` en MongoDB?
    Elimina un solo documento que coincida con una condición específica.
26. ¿Qué es un operador de consulta en MongoDB y cómo funciona `$gte`?
    Los operadores de consulta permiten realizar búsquedas más complejas. `$gte` busca documentos donde un campo es mayor o igual que un valor especificado.
27. ¿Qué es la replicación en MongoDB?
    Es el proceso de sincronizar datos entre múltiples servidores, proporcionando redundancia y mayor disponibilidad de los datos.
28. ¿Qué es la fragmentación (sharding) en MongoDB?
    Es un método de dividir grandes volúmenes de datos en múltiples servidores para mejorar el rendimiento y la escalabilidad.
29. ¿Cuál es la diferencia entre una base de datos relacional y una base de datos NoSQL como MongoDB?
    Las bases de datos relacionales organizan datos en tablas con relaciones predefinidas,

mientras que MongoDB almacena datos en documentos flexibles sin necesidad de un esquema rígido.

30. ¿Qué son las colecciones anidadas en MongoDB?
    Son documentos que contienen otros documentos dentro de ellos, permitiendo una estructura de datos más compleja y jerárquica.

# 2. Ssr

## SQL (MySQL)

1. ¿Qué es una vista en MySQL y para qué se utiliza?
   Una vista es una consulta almacenada que puede ser tratada como una tabla virtual. Se utiliza para simplificar consultas complejas y mejorar la seguridad, restringiendo el acceso a ciertos datos.
2. ¿Cuál es la diferencia entre una subconsulta y una consulta correlacionada en SQL?
   Una subconsulta es una consulta anidada que se ejecuta una vez, mientras que una consulta correlacionada se ejecuta repetidamente para cada fila del resultado de la consulta externa.
3. ¿Qué es una transacción ACID y por qué es importante en bases de datos?
   Una transacción ACID asegura propiedades de Atomicidad, Consistencia, Aislamiento y Durabilidad, garantizando que las operaciones en una base de datos sean seguras y confiables.
4. ¿Cómo funciona el bloqueo de filas (row locking) en MySQL?
   El bloqueo de filas impide que varias transacciones modifiquen las mismas filas simultáneamente, protegiendo la integridad de los datos y evitando condiciones de carrera.
5. ¿Qué es la normalización de tercer nivel (3NF) en una base de datos relacional?
   Es una forma de normalización que elimina las dependencias funcionales transitivas, garantizando que cada columna no clave dependa únicamente de la clave primaria.
6. ¿Qué es una transacción distribuida en MySQL?
   Es una transacción que involucra múltiples bases de datos o sistemas y requiere coordinar la ejecución de operaciones en todas las bases implicadas.
7. ¿Qué es el `JOIN` cruzado (CROSS JOIN) en MySQL?
   Un `CROSS JOIN` devuelve el producto cartesiano de dos tablas, es decir, combina cada fila de la primera tabla con cada fila de la segunda tabla.
8. ¿Cómo se puede optimizar una consulta en MySQL que involucra múltiples `JOIN`?
   Se pueden usar índices adecuados en las columnas que se utilizan en los `JOIN`, reducir la cantidad de datos procesados con filtros en `WHERE` y evitar el uso innecesario de subconsultas.
9. ¿Qué es un índice compuesto en MySQL y cuándo se utiliza?
   Un índice compuesto es un índice que abarca más de una columna. Se utiliza cuando las consultas incluyen varios campos para filtrar o unir los datos.

10. ¿Qué es el control de concurrencia en MySQL y cómo se gestiona?
    El control de concurrencia es el manejo del acceso simultáneo a los datos para evitar inconsistencias. En MySQL, se gestiona mediante bloqueos y niveles de aislamiento de transacciones.
11. ¿Cómo funciona el `AUTO_INCREMENT` en MySQL?
    `AUTO_INCREMENT` es una propiedad que se puede aplicar a una columna para que incremente automáticamente su valor en cada nueva inserción de datos.
12. ¿Qué es un procedimiento almacenado (Stored Procedure) en MySQL y cuáles son sus ventajas?
    Un procedimiento almacenado es un conjunto de consultas SQL que se pueden ejecutar como una sola unidad. Mejora el rendimiento y la reutilización de código, y reduce el tráfico entre la aplicación y la base de datos.
13. ¿Cómo se puede manejar el rendimiento de consultas que realizan agregaciones en grandes tablas en MySQL?
    Se pueden crear índices en las columnas utilizadas en las cláusulas `GROUP BY`, particionar tablas o crear vistas materializadas.
14. ¿Qué es una función de ventana en MySQL y para qué se utiliza?
    Una función de ventana realiza cálculos sobre un conjunto de filas relacionadas, sin necesidad de agruparlas en una única fila. Se utiliza en análisis avanzados de datos, como calcular totales acumulados.
15. ¿Qué es un trigger en MySQL y cuándo se utiliza?
    Un trigger es un bloque de código que se ejecuta automáticamente en respuesta a eventos `INSERT`, `UPDATE` o `DELETE` en una tabla, útil para automatizar acciones.

---

## NoSQL (MongoDB)

16. ¿Cómo maneja MongoDB las relaciones entre documentos?
    MongoDB utiliza referencias o documentos anidados para manejar relaciones, lo que permite flexibilidad en cómo se estructuran los datos relacionados.
17. ¿Qué es la agregación en MongoDB y cuándo se utiliza?
    La agregación es un marco que permite procesar y transformar datos en MongoDB, utilizado para realizar operaciones complejas como filtrado, agrupación y ordenación.
18. ¿Cómo funcionan las operaciones atómicas en MongoDB?
    Las operaciones atómicas en MongoDB se aplican a nivel de documento, garantizando que las modificaciones a un documento sean completamente aplicadas o no aplicadas.
19. ¿Qué es el `sharding` en MongoDB y cómo mejora la escalabilidad?
    El `sharding` es una técnica de partición que distribuye los datos en varios servidores, permitiendo que MongoDB escale horizontalmente para manejar grandes volúmenes de datos.
20. ¿Qué es una colección capada en MongoDB y cuándo se utiliza?
    Una colección capada es una colección de tamaño fijo que sobrescribe los

documentos antiguos cuando se alcanza el límite de almacenamiento, útil en escenarios de registro de eventos.

21. ¿Cómo se crean índices compuestos en MongoDB y para qué sirven?
    Los índices compuestos en MongoDB se crean sobre múltiples campos en un documento y se utilizan para mejorar el rendimiento de las consultas que filtran por más de un campo.
22. ¿Qué es una operación de inserción masiva (bulk insert) en MongoDB?
    Es una operación que permite insertar múltiples documentos en una colección al mismo tiempo, optimizando el rendimiento en escenarios donde se necesita agregar grandes cantidades de datos.
23. ¿Qué es el modelo de consistencia eventual en MongoDB y cuándo es importante?
    El modelo de consistencia eventual garantiza que, tras un tiempo, todas las réplicas de datos alcanzarán el mismo estado, pero puede haber un retraso en la sincronización de los datos entre nodos.
24. ¿Cómo maneja MongoDB las actualizaciones parciales en documentos?
    MongoDB permite actualizar parcialmente los documentos usando operadores como `$set` o `$unset`, modificando solo los campos necesarios en lugar de reemplazar todo el documento.
25. ¿Cómo se pueden manejar esquemas dinámicos en MongoDB?
    En MongoDB, no es necesario definir un esquema fijo, lo que permite almacenar documentos con diferentes estructuras en la misma colección.
26. ¿Qué es la replicación en MongoDB y cómo se configura un conjunto de réplicas?
    La replicación es el proceso de mantener copias de los datos en varios servidores. Un conjunto de réplicas se configura para mejorar la disponibilidad de los datos y se utiliza en aplicaciones distribuidas.
27. ¿Qué es un `write concern` en MongoDB y por qué es importante?
    `Write concern` especifica el nivel de reconocimiento que una operación de escritura debe tener antes de ser considerada completada, asegurando la durabilidad de los datos.
28. ¿Cómo se puede gestionar el rendimiento de las consultas en MongoDB?
    Se puede utilizar la indexación adecuada, realizar consultas eficientes que utilicen filtros específicos, y aprovechar el marco de agregación para procesar los datos a nivel de servidor.
29. ¿Qué es un esquema polimórfico en MongoDB y cuándo se utiliza?
    Un esquema polimórfico permite almacenar diferentes tipos de datos dentro de una misma colección. Es útil cuando se requiere flexibilidad para manejar diferentes estructuras de documentos.
30. ¿Cómo se realiza la migración de datos en MongoDB sin interrumpir el servicio?
    Se puede usar réplicas para migrar datos a un nuevo servidor sin interrupción, o hacer uso de estrategias de migración en línea con `sharding` para mover los datos entre nodos.

# 3. Sr

## SQL (MySQL)

1. ¿Qué es el particionamiento de tablas en MySQL y cuándo se debe utilizar?
   El particionamiento divide una tabla en partes más pequeñas llamadas particiones, que pueden ser gestionadas y consultadas más eficientemente, ideal para tablas con gran cantidad de datos.
2. ¿Cómo optimizarías una base de datos con miles de millones de registros?
   Implementar particionamiento, optimizar índices, utilizar vistas materializadas y configurar adecuadamente el almacenamiento y la memoria caché.
3. ¿Cómo se maneja la consistencia en una base de datos MySQL replicada?
   MySQL soporta replicación asincrónica, semisíncrona y síncrona. La consistencia se garantiza configurando el `write concern` y utilizando réplicas maestras y esclavas para gestionar la latencia.
4. ¿Qué es un bloqueo de nivel de fila frente a un bloqueo de nivel de tabla en MySQL?
   El bloqueo de nivel de fila impide que otras transacciones modifiquen o lean una fila específica, mientras que el bloqueo de nivel de tabla bloquea todas las filas de una tabla durante la transacción.
5. ¿Qué problemas resuelve la serialización de transacciones y cómo afecta el rendimiento?
   La serialización garantiza que las transacciones se ejecuten de manera que los resultados sean consistentes, pero puede afectar el rendimiento debido a la mayor latencia y menor concurrencia.
6. ¿Qué es una consulta de ejecución retardada (delayed query) en MySQL y cuándo se utiliza?
   Una consulta retardada inserta datos en segundo plano para mejorar el rendimiento en inserciones masivas, evitando que las operaciones de escritura bloqueen el procesamiento.
7. ¿Cómo gestionas las deadlocks en una base de datos MySQL?
   Se deben optimizar las transacciones para que soliciten los mismos recursos en el mismo orden y reducir el tiempo de retención de bloqueos para evitar conflictos.
8. ¿Cómo se diseñaría un sistema altamente disponible con MySQL?
   Se implementaría replicación maestro-esclavo o maestro-maestro, con balanceo de carga y monitoreo de salud de los nodos para asegurar la disponibilidad y tolerancia a fallos.
9. ¿Qué es una vista materializada y cómo se diferencia de una vista normal en MySQL?
   Una vista materializada almacena físicamente los resultados de la consulta, mejorando el rendimiento de las consultas repetidas, mientras que una vista normal ejecuta la consulta cada vez que se accede a ella.
10. ¿Cómo se pueden evitar los puntos calientes en el particionamiento de tablas en MySQL?

Distribuyendo los datos de manera uniforme entre las particiones y asegurando que las consultas no se concentren en una sola partición mediante el uso de particionamiento por rango o hash.

11. ¿Qué es el análisis de planos de consulta ( `EXPLAIN` ) y cómo se utiliza para optimizar consultas?
    El análisis de planos de consulta ( `EXPLAIN` ) muestra cómo el motor de MySQL ejecutará una consulta, revelando el uso de índices, uniones y filtrado, lo que ayuda a identificar cuellos de botella en el rendimiento.
12. ¿Cómo afecta el uso de índices a las operaciones de escritura en MySQL?
    Los índices mejoran el rendimiento de las consultas de lectura, pero pueden ralentizar las operaciones de escritura como `INSERT` , `UPDATE` y `DELETE` , ya que deben actualizarse cada vez que los datos cambian.
13. ¿Qué es el control de concurrencia multiversión (MVCC) en MySQL?
    MVCC permite que varias transacciones accedan simultáneamente a los datos sin bloquearse entre sí, proporcionando lecturas consistentes sin bloquear las operaciones de escritura.
14. ¿Cómo manejas el crecimiento exponencial de una base de datos en términos de almacenamiento y rendimiento?
    Se puede utilizar particionamiento, optimización de índices, vistas materializadas, reducción de datos históricos mediante archivado, y escalado horizontal con replicación o sharding.
15. ¿Qué problemas pueden surgir al utilizar `GROUP BY` en grandes tablas y cómo se pueden mitigar?
    Las consultas `GROUP BY` pueden ser lentas en grandes tablas, ya que requieren ordenar y agrupar los datos. Se puede mitigar usando índices compuestos y optimizando la memoria asignada al servidor.

---

# NoSQL (MongoDB)

16. ¿Cómo optimizarías el rendimiento de una consulta de agregación compleja en MongoDB?
    Utilizarías índices adecuados, el framework de agregación con `pipelines` , y reducirías los datos innecesarios lo antes posible en el `pipeline` para minimizar el procesamiento.
17. ¿Qué estrategias de particionamiento (sharding) implementarías para una base de datos MongoDB con un volumen masivo de datos?
    Elegirías una clave de fragmentación adecuada que distribuya uniformemente los datos entre los shards y evitarías puntos calientes. Utilizarías particionamiento por rango o por hash según el patrón de acceso a los datos.
18. ¿Qué es la estrategia de `read preference` en MongoDB y cómo se utiliza en una configuración con réplicas?

`Read preference` define desde qué nodo de un conjunto de réplicas se leen los datos. Se puede usar para distribuir las lecturas entre nodos primarios y secundarios, equilibrando la carga y mejorando el rendimiento.

19. ¿Cómo manejarías la inconsistencia eventual en un sistema distribuido de MongoDB?
    Diseñarías tu aplicación para tolerar lecturas temporales inconsistentes, utilizando mecanismos como `read concern` y configurando adecuadamente la replicación según las necesidades de consistencia.
20. ¿Cómo optimizar el rendimiento de las escrituras en MongoDB en un entorno de alta carga?
    Se puede ajustar el `write concern` para evitar esperas innecesarias, utilizar particionamiento para distribuir la carga de escritura y ajustar la configuración de los índices para reducir la sobrecarga de escrituras.
21. ¿Qué consideraciones tomarías en cuenta al elegir una clave de partición ( `shard key` ) en MongoDB?
    Debe distribuir uniformemente los datos y la carga entre los shards, evitando puntos calientes. Además, la clave debe ser frecuente en las consultas para evitar redireccionamientos innecesarios entre shards.
22. ¿Cómo se gestionan las transacciones en MongoDB y qué limitaciones tienen en comparación con las bases de datos relacionales?
    Las transacciones en MongoDB permiten operaciones atómicas en múltiples documentos o colecciones, pero pueden tener un rendimiento inferior en comparación con transacciones relacionales debido a la falta de bloqueo a nivel de tabla.
23. ¿Cómo manejarías la escalabilidad horizontal en MongoDB para una aplicación global con múltiples zonas horarias?
    Utilizaría `sharding` para distribuir los datos geográficamente, con shards en diferentes centros de datos, y configuraría las réplicas para optimizar las lecturas locales y minimizar la latencia.
24. ¿Qué es una `replica set` en MongoDB y cómo se utiliza para asegurar alta disponibilidad?
    Un `replica set` es un grupo de instancias de MongoDB que replican datos entre sí para asegurar la redundancia y alta disponibilidad. Si el nodo primario falla, uno de los nodos secundarios asume automáticamente el rol de primario.
25. ¿Cómo manejarías el almacenamiento de grandes archivos en MongoDB?
    Utilizaría GridFS, un sistema de almacenamiento distribuido que permite dividir y almacenar archivos grandes en múltiples fragmentos dentro de MongoDB.
26. ¿Qué es la agregación en tiempo real en MongoDB y cómo se puede lograr?
    Se puede lograr mediante el uso de `change streams`, que permiten reaccionar a los cambios en tiempo real en los datos de MongoDB sin la necesidad de consultas periódicas.
27. ¿Cómo se puede mejorar el rendimiento de la replicación en un conjunto de réplicas de MongoDB?
    Ajustando la configuración del `write concern` y `read concern`, optimizando las redes

y hardware, y ajustando la configuración del tamaño del búfer de replicación para garantizar la transferencia eficiente de datos.

28. ¿Qué diferencias hay entre `replica sets` y `sharding` en MongoDB?
    Los `replica sets` proporcionan alta disponibilidad y redundancia mediante la replicación de datos entre nodos, mientras que el `sharding` distribuye los datos entre múltiples servidores para mejorar la escalabilidad y el rendimiento.
29. ¿Cómo se maneja el versionado de esquemas en MongoDB sin causar interrupciones en la aplicación?
    Se puede implementar un enfoque de migración progresiva en el que los nuevos documentos usen el esquema actualizado y los documentos antiguos se migren gradualmente a medida que se acceden a ellos.
30. ¿Qué consideraciones tomarías al migrar una base de datos relacional a MongoDB?
    Debes identificar qué datos se beneficiarán de un modelo de documentos, rediseñar el esquema para aprovechar la flexibilidad de MongoDB, y tener en cuenta las diferencias en consistencia, atomicidad y transacciones.

# 1. Jr

1. ¿Qué es GitHub?
   GitHub es una plataforma de alojamiento de código fuente que utiliza Git para el control de versiones, permitiendo a los desarrolladores colaborar en proyectos de software.
2. ¿Cuál es la diferencia entre un repositorio público y uno privado en GitHub?
   Un repositorio público está accesible para cualquier persona en Internet, mientras que un repositorio privado solo está accesible para los usuarios que el propietario del repositorio haya autorizado.
3. ¿Cómo puedes clonar un repositorio de GitHub a tu máquina local?
   Utiliza el comando `git clone` seguido de la URL del repositorio en la terminal.
4. ¿Qué comando se usa para enviar tus cambios locales a GitHub?
   Utiliza el comando `git push` para enviar tus cambios locales al repositorio remoto en GitHub.
5. ¿Cómo puedes ver el historial de commits en un repositorio de GitHub?
   Utiliza el comando `git log` para ver el historial de commits en la terminal.
6. ¿Qué es un "fork" en GitHub?
   Un "fork" es una copia personal de un repositorio que permite a los usuarios experimentar con cambios sin afectar el repositorio original.
7. ¿Cómo puedes crear una nueva rama en GitHub?
   Puedes crear una nueva rama utilizando el comando `git branch nombre-de-la-rama` y luego cambiar a ella con `git checkout nombre-de-la-rama`.
8. ¿Qué es un "pull request" en GitHub?
   Un "pull request" es una solicitud para revisar y fusionar cambios de una rama a otra

en un repositorio, generalmente utilizada para integrar cambios en la rama principal del proyecto.

9. ¿Cómo puedes resolver conflictos de fusión (merge conflicts) en GitHub?
   Edita los archivos conflictivos para resolver los conflictos manualmente, luego realiza un commit y un push de los cambios resueltos.
10. ¿Qué es un "commit" en Git?
    Un "commit" es una instantánea de los cambios realizados en el repositorio, que incluye un mensaje descriptivo sobre los cambios realizados.
11. ¿Cómo puedes crear un "tag" en GitHub?
    Utiliza el comando `git tag nombre-del-tag` para crear un nuevo tag en el repositorio local y luego haz un `git push --tags` para subirlo a GitHub.
12. ¿Qué comando se usa para actualizar tu repositorio local con los cambios del repositorio remoto?
    Utiliza el comando `git pull` para actualizar tu repositorio local con los últimos cambios del repositorio remoto.
13. ¿Qué es un "branch" en GitHub?
    Un "branch" (rama) es una línea de desarrollo independiente en un repositorio que permite trabajar en diferentes versiones del proyecto simultáneamente.
14. ¿Cómo puedes eliminar una rama en GitHub?
    Utiliza el comando `git branch -d nombre-de-la-rama` para eliminar una rama localmente y `git push origin --delete nombre-de-la-rama` para eliminarla en el repositorio remoto.
15. ¿Qué es un "merge" en Git?
    Un "merge" es el proceso de combinar los cambios de dos ramas diferentes en una sola rama, integrando los cambios realizados en ambas ramas.
16. ¿Cómo puedes ver los cambios que has hecho en tus archivos antes de hacer un commit?
    Utiliza el comando `git status` para ver los archivos modificados y `git diff` para ver los cambios exactos en los archivos.
17. ¿Qué es un "remote" en Git?
    Un "remote" es una versión del repositorio que está alojada en un servidor remoto, como GitHub, que permite la colaboración y sincronización entre múltiples desarrolladores.
18. ¿Cómo puedes añadir un nuevo remoto a tu repositorio local?
    Utiliza el comando `git remote add nombre-del-remote URL-del-repositorio` para añadir un nuevo remoto a tu repositorio local.
19. ¿Qué es un "repository" en GitHub?
    Un "repository" (repositorio) es un espacio de almacenamiento en GitHub donde se guardan los archivos y el historial de cambios de un proyecto de software.
20. ¿Cómo puedes ver la lista de todos los remotos configurados en tu repositorio local?
    Utiliza el comando `git remote -v` para ver la lista de todos los remotos configurados en tu repositorio local.

21. ¿Qué es el "README" en un repositorio de GitHub?
    El "README" es un archivo que proporciona información sobre el proyecto, como instrucciones de instalación, uso y otros detalles relevantes para los colaboradores y usuarios.
22. ¿Cómo puedes actualizar un repositorio remoto con tus cambios locales?
    Utiliza el comando `git push` para actualizar el repositorio remoto con los cambios que has realizado en tu repositorio local.
23. ¿Qué comando se usa para ver los cambios que se han hecho en un commit específico?
    Utiliza el comando `git show id-del-commit` para ver los cambios realizados en un commit específico.
24. ¿Cómo puedes revertir un commit en Git?
    Utiliza el comando `git revert id-del-commit` para crear un nuevo commit que deshace los cambios introducidos por el commit especificado.
25. ¿Qué es un "git stash" y cómo se usa?
    El comando `git stash` guarda temporalmente los cambios no confirmados en un área de almacenamiento temporal, permitiendo trabajar en otras tareas y restaurar los cambios más tarde con `git stash pop`.
26. ¿Cómo puedes ver los cambios en el historial de un archivo específico?
    Utiliza el comando `git log nombre-del-archivo` para ver el historial de cambios de un archivo específico.
27. ¿Qué es un "pull" en Git y cómo se diferencia de un "fetch"?
    Un "pull" descarga los cambios del repositorio remoto y los fusiona con la rama actual, mientras que un "fetch" solo descarga los cambios sin fusionarlos.
28. ¿Cómo puedes agregar un archivo a tu repositorio para que sea rastreado por Git?
    Utiliza el comando `git add nombre-del-archivo` para agregar un archivo al área de preparación (staging area) para el próximo commit.
29. ¿Qué es un "gitignore" y para qué se usa?
    Un "gitignore" es un archivo que especifica qué archivos y directorios deben ser ignorados por Git, evitando que sean rastreados y versionados en el repositorio.
30. ¿Cómo puedes crear una rama a partir de otra rama existente?
    Utiliza el comando `git checkout -b nombre-de-la-nueva-rama nombre-de-la-rama-existente` para crear y cambiar a una nueva rama basada en otra rama existente.
31. ¿Cómo puedes comparar dos ramas diferentes en Git?
    Utiliza el comando `git diff rama1..rama2` para comparar los cambios entre dos ramas diferentes.
32. ¿Qué es un "commit message" y por qué es importante?
    Un "commit message" es un mensaje descriptivo asociado con un commit que explica los cambios realizados. Es importante para mantener un historial claro y comprensible del proyecto.
33. ¿Cómo puedes buscar un commit específico en el historial de Git?
    Utiliza el comando `git log --grep "texto-de-búsqueda"` para buscar commits que

contengan el texto especificado en el mensaje del commit.

34. ¿Qué es una "pull request" y cómo se usa en GitHub?
    Una "pull request" es una solicitud para revisar y fusionar cambios de una rama a otra en un repositorio, permitiendo colaboración y revisión de código antes de integrar los cambios.
35. ¿Cómo puedes cancelar un commit que aún no has hecho push al repositorio remoto?
    Utiliza el comando `git reset --soft HEAD~1` para deshacer el último commit, manteniendo los cambios en el área de preparación.
36. ¿Qué es el "commit hash" en Git?
    El "commit hash" es un identificador único generado por Git para cada commit, utilizado para referenciar y acceder a commits específicos en el historial.
37. ¿Cómo puedes ver el estado actual de los archivos en tu repositorio?
    Utiliza el comando `git status` para ver el estado actual de los archivos, incluyendo los archivos modificados, agregados y eliminados.
38. ¿Qué es una "branch" y cómo se diferencia de un "fork"?
    Una "branch" es una línea de desarrollo dentro de un repositorio, mientras que un "fork" es una copia completa de un repositorio en tu cuenta de GitHub.
39. ¿Cómo puedes revertir los cambios realizados en un archivo antes de hacer un commit?
    Utiliza el comando `git checkout -- nombre-del-archivo` para descartar los cambios no confirmados en un archivo específico.
40. ¿Qué es un "merge conflict" y cómo se resuelve?
    Un "merge conflict" ocurre cuando Git no puede fusionar automáticamente los cambios de dos ramas. Se resuelve editando los archivos conflictivos, eligiendo qué cambios conservar y luego realizando un commit.
41. ¿Cómo puedes visualizar los cambios en un archivo antes de hacer un commit?
    Utiliza el comando `git diff nombre-del-archivo` para ver los cambios no confirmados en un archivo específico.
42. ¿Qué es un "clone" en GitHub?
    Un "clone" es una copia local de un repositorio remoto que te permite trabajar en el proyecto en tu máquina local.
43. ¿Cómo puedes ver qué archivos están listos para ser comprometidos en tu repositorio?
    Utiliza el comando `git status` para ver los archivos que están en el área de preparación (staging area) listos para el próximo commit.
44. ¿Qué es un "repository URL" y cómo se usa para clonar un repositorio?
    La "repository URL" es la dirección web del repositorio en GitHub, utilizada con el comando `git clone` para crear una copia local del repositorio.
45. ¿Cómo puedes agregar un archivo al área de preparación (staging area) de Git?
    Utiliza el comando `git add nombre-del-archivo` para agregar un archivo específico al área de preparación.

46. ¿Qué comando se utiliza para ver las diferencias entre el estado actual del repositorio y el último commit?
    Utiliza el comando `git diff` para ver las diferencias entre el estado actual del repositorio y el último commit.
47. ¿Cómo puedes cambiar el mensaje de un commit que aún no has enviado al repositorio remoto?
    Utiliza el comando `git commit --amend` para cambiar el mensaje del último commit.
48. ¿Qué es un "origin" en Git?
    "origin" es el nombre predeterminado dado al repositorio remoto desde el que se clonó el repositorio local.
49. ¿Cómo puedes ver los cambios en el historial de un archivo específico?
    Utiliza el comando `git log nombre-del-archivo` para ver el historial de cambios de un archivo específico.
50. ¿Qué es un "git fetch" y cómo se usa?
    El comando `git fetch` descarga los cambios del repositorio remoto sin fusionarlos con la rama actual, permitiendo revisar los cambios antes de integrarlos.

# 2. Ssr

1. ¿Cómo puedes gestionar múltiples remotos en un repositorio de GitHub?
   Utiliza el comando `git remote add nombre-del-remote URL-del-repositorio` para añadir nuevos remotos y `git remote remove nombre-del-remote` para eliminarlos.
2. ¿Cómo puedes comparar las diferencias entre dos ramas en GitHub?
   Utiliza el comando `git diff rama1..rama2` para comparar los cambios entre dos ramas.
3. ¿Qué es el "rebase" y cómo se diferencia del "merge"?
   El "rebase" aplica los commits de una rama sobre otra, reescribiendo el historial, mientras que el "merge" fusiona dos ramas sin modificar el historial.
4. ¿Cómo puedes ver los detalles de un "pull request" en GitHub?
   Navega a la pestaña "Pull Requests" en el repositorio en GitHub y selecciona el pull request para ver sus detalles, comentarios y revisiones.
5. ¿Cómo puedes realizar un "squash" de commits en Git?
   Utiliza el comando `git rebase -i HEAD~n` para combinar los últimos n commits en uno solo, editando el historial de commits.
6. ¿Qué es una "git tag" y cómo se usa?
   Un "git tag" marca puntos específicos en el historial de commits, como versiones de lanzamiento, y se utiliza con `git tag nombre-del-tag` para crearlo.
7. ¿Cómo puedes configurar un flujo de trabajo de integración continua (CI) en GitHub?
   Utiliza GitHub Actions para configurar flujos de trabajo que automatizan pruebas y despliegues cada vez que se realizan cambios en el repositorio.
8. ¿Qué es un "git stash" y cuándo se debe usar?
   El comando `git stash` guarda cambios no confirmados en un área temporal para

permitir cambios en otras ramas sin perder el trabajo en curso.

9. ¿Cómo puedes cambiar el nombre de una rama en GitHub?
   Utiliza el comando `git branch -m nombre-antiguo nombre-nuevo` para cambiar el nombre de una rama local y `git push origin :nombre-antiguo nombre-nuevo` para actualizar el remoto.
10. ¿Cómo puedes ver las diferencias entre el estado actual de tu repositorio y un commit específico?
    Utiliza el comando `git diff id-del-commit` para ver las diferencias entre el estado actual y un commit específico.
11. ¿Cómo puedes configurar un repositorio para trabajar con múltiples ramas de desarrollo y producción?
    Crea y gestiona ramas separadas para desarrollo y producción utilizando `git branch` y realiza merges según sea necesario para integrar cambios.
12. ¿Qué es un "cherry-pick" y cómo se usa?
    El comando `git cherry-pick id-del-commit` aplica cambios de un commit específico a la rama actual sin fusionar toda la rama.
13. ¿Cómo puedes realizar una búsqueda en el historial de commits en GitHub?
    Utiliza el comando `git log --grep "texto-de-búsqueda"` para buscar commits que contengan el texto especificado en el mensaje del commit.
14. ¿Cómo puedes evitar que ciertos archivos se suban a GitHub utilizando `.gitignore`?
    Añade los nombres de los archivos o patrones de archivos al archivo `.gitignore` para que Git los ignore y no los incluya en los commits.
15. ¿Cómo puedes revertir cambios en un archivo después de un `git pull`?
    Utiliza el comando `git checkout -- nombre-del-archivo` para descartar cambios no confirmados después de un `git pull`.
16. ¿Cómo puedes administrar los permisos de acceso a un repositorio en GitHub?
    Configura los permisos en la sección "Settings" del repositorio, bajo "Manage access", para controlar quién puede ver, editar o administrar el repositorio.
17. ¿Qué es un "release" en GitHub y cómo se crea?
    Un "release" es una versión específica del proyecto que se puede marcar y etiquetar. Se crea en la pestaña "Releases" del repositorio y se asocia con un tag.
18. ¿Cómo puedes comparar dos ramas utilizando la interfaz web de GitHub?
    Ve a la pestaña "Pull Requests" y selecciona "New pull request", luego elige las ramas que deseas comparar.
19. ¿Cómo puedes configurar notificaciones para cambios en un repositorio?
    Ve a "Watch" en la parte superior del repositorio en GitHub y selecciona el nivel de notificación que deseas recibir.
20. ¿Cómo puedes hacer un "revert" de un merge en GitHub?
    Utiliza el comando `git revert -m 1 id-del-merge` para revertir un merge commit y aplicar los cambios que deshacen la fusión.
21. ¿Qué es un "fork" y cómo se utiliza en GitHub?
    Un "fork" es una copia independiente de un repositorio que permite hacer cambios sin

afectar el repositorio original. Se utiliza para experimentar y contribuir a proyectos de otros.

22. ¿Cómo puedes agregar colaboradores a un repositorio en GitHub?
    En la sección "Settings" del repositorio, selecciona "Manage access" y agrega colaboradores ingresando sus nombres de usuario o correos electrónicos.
23. ¿Cómo puedes ver el tamaño de los archivos en tu repositorio en GitHub?
    En la interfaz web de GitHub, utiliza la pestaña "Insights" y selecciona "Repository size" para ver el tamaño total del repositorio.
24. ¿Cómo puedes trabajar con submódulos en GitHub?
    Utiliza el comando `git submodule add URL-del-repositorio` para agregar un submódulo y `git submodule update` para sincronizarlo con el repositorio principal.
25. ¿Cómo puedes listar todos los tags en un repositorio de GitHub?
    Utiliza el comando `git tag` para listar todos los tags en el repositorio local.
26. ¿Qué comando se usa para eliminar un tag en Git?
    Utiliza el comando `git tag -d nombre-del-tag` para eliminar un tag local y `git push origin --delete nombre-del-tag` para eliminarlo del remoto.
27. ¿Cómo puedes ver el historial de cambios de un archivo en GitHub?
    Utiliza el comando `git log nombre-del-archivo` para ver el historial de cambios de un archivo específico.
28. ¿Qué es un "git hook" y cómo se utiliza?
    Un "git hook" es un script que se ejecuta automáticamente en ciertos puntos del flujo de trabajo de Git, como antes de un commit o después de un push, y se configura en el directorio `.git/hooks`.
29. ¿Cómo puedes actualizar la URL del remoto en GitHub?
    Utiliza el comando `git remote set-url nombre-del-remote nueva-URL` para actualizar la URL del remoto en tu repositorio local.
30. ¿Cómo puedes eliminar un archivo del historial de Git sin eliminarlo del sistema de archivos?
    Utiliza el comando `git rm --cached nombre-del-archivo` para eliminar un archivo del historial de Git pero mantenerlo en el sistema de archivos.
31. ¿Cómo puedes verificar el estado de los submódulos en un repositorio de Git?
    Utiliza el comando `git submodule status` para verificar el estado de los submódulos en tu repositorio.
32. ¿Cómo puedes fusionar ramas utilizando la interfaz web de GitHub?
    Crea un pull request para fusionar ramas y utiliza la opción "Merge pull request" en la interfaz web para completar la fusión.
33. ¿Qué es un "commit amend" y cuándo se usa?
    El comando `git commit --amend` permite modificar el último commit, añadiendo cambios adicionales o editando el mensaje del commit.
34. ¿Cómo puedes proteger una rama de cambios en GitHub?
    Configura reglas de protección de ramas en la sección "Settings" del repositorio, estableciendo requisitos como revisiones obligatorias antes de fusionar cambios.

35. ¿Cómo puedes ver las estadísticas de contribuciones en GitHub?
    Navega a la pestaña "Insights" del repositorio y selecciona "Contributors" para ver las estadísticas de contribuciones.
36. ¿Qué es un "git reflog" y cómo se usa?
    El comando `git reflog` muestra un registro de todas las referencias de Git, permitiendo recuperar commits que no están en el historial de commits actual.
37. ¿Cómo puedes clonar un repositorio de GitHub con submódulos?
    Utiliza el comando `git clone --recurse-submodules URL-del-repositorio` para clonar un repositorio con sus submódulos.
38. ¿Cómo puedes crear una plantilla de repositorio en GitHub?
    En la sección "Settings" del repositorio, selecciona "Template repository" para permitir que el repositorio sea usado como plantilla para nuevos repositorios.
39. ¿Cómo puedes agregar una clave SSH a tu cuenta de GitHub?
    En la sección "Settings" de tu cuenta, selecciona "SSH and GPG keys" y añade una nueva clave SSH copiando y pegando la clave pública desde tu máquina local.
40. ¿Cómo puedes integrar GitHub con herramientas de CI/CD como Travis CI o CircleCI?
    Configura la integración en la sección "Settings" del repositorio y añade un archivo de configuración específico para la herramienta de CI/CD en tu repositorio.
41. ¿Cómo puedes revertir varios commits en Git utilizando la interfaz de línea de comandos?
    Utiliza el comando `git revert HEAD~n..HEAD` para revertir los últimos n commits en el historial de commits.
42. ¿Cómo puedes revisar las solicitudes de extracción (pull requests) en GitHub?
    Ve a la pestaña "Pull Requests" del repositorio y revisa las solicitudes de extracción pendientes, revisando cambios y comentarios.
43. ¿Cómo puedes resolver conflictos de merge utilizando la interfaz web de GitHub?
    Utiliza el editor de conflictos integrado en la interfaz web de GitHub para resolver conflictos y completar el merge.
44. ¿Qué es un "git bisect" y cómo se utiliza?
    El comando `git bisect` permite buscar un commit específico que introdujo un error, utilizando una búsqueda binaria en el historial de commits.
45. ¿Cómo puedes configurar notificaciones para un repositorio específico en GitHub?
    Utiliza la opción "Watch" en la parte superior del repositorio para configurar notificaciones de cambios en el repositorio.
46. ¿Cómo puedes proteger tu repositorio de GitHub contra ataques de fuerza bruta?
    Utiliza autenticación de dos factores (2FA) y establece políticas de seguridad en la configuración del repositorio.
47. ¿Cómo puedes ver las métricas de código en GitHub?
    Utiliza herramientas de análisis de código y métricas disponibles en la pestaña "Insights" del repositorio.
48. ¿Cómo puedes colaborar en un proyecto utilizando ramas y pull requests?
    Crea ramas para desarrollar nuevas características, utiliza pull requests para solicitar

revisiones y fusionar cambios en la rama principal.

49. ¿Cómo puedes utilizar los "milestones" en GitHub para gestionar proyectos?
    Crea "milestones" para agrupar issues y pull requests relacionados con objetivos específicos del proyecto y realizar un seguimiento del progreso.
50. ¿Cómo puedes obtener estadísticas sobre los cambios en un repositorio de GitHub?
    Utiliza la pestaña "Insights" y selecciona "Network" o "Pulse" para ver estadísticas y gráficos sobre los cambios y actividades en el repositorio.

# 3. Sr

1. ¿Cómo puedes realizar un "rebase interactivo" en Git para reordenar, modificar o combinar commits?
   Utiliza el comando `git rebase -i HEAD~n` para iniciar un rebase interactivo sobre los últimos n commits, permitiendo reordenar, modificar o combinar commits.
2. ¿Cómo puedes configurar y utilizar GitHub Actions para automatizar el despliegue en múltiples entornos?
   Crea flujos de trabajo en `.github/workflows` utilizando YAML para definir trabajos y pasos que automatizan el despliegue en diferentes entornos de producción y pruebas.
3. ¿Cómo puedes manejar y resolver conflictos en un rebase utilizando la interfaz de línea de comandos?
   Durante un rebase, Git pausará el proceso en conflictos. Usa `git status` para ver los archivos conflictivos, resuelve los conflictos manualmente, y continúa con `git rebase --continue`.
4. ¿Cómo puedes proteger ramas en GitHub para requerir revisiones de equipo antes de permitir fusiones?
   Configura reglas de protección de ramas en la sección "Settings" del repositorio, estableciendo requisitos como revisiones obligatorias y aprobaciones antes de permitir fusiones.
5. ¿Cómo puedes utilizar "git submodule" para gestionar dependencias en un repositorio?
   Añade submódulos utilizando `git submodule add URL-del-repositorio`, y actualiza los submódulos con `git submodule update` para sincronizarlos con el repositorio principal.
6. ¿Cómo puedes utilizar `git bisect` para encontrar el commit que introdujo un bug específico?
   Inicia un bisect con `git bisect start`, marca el commit actual como malo y un commit anterior como bueno, y luego sigue las instrucciones para identificar el commit problemático.
7. ¿Cómo puedes crear y gestionar "GitHub Apps" para integrar aplicaciones personalizadas con tu repositorio?
   Crea una GitHub App en la sección "Developer settings" de GitHub, configura permisos y eventos, y utiliza el token de autenticación para interactuar con la API de GitHub desde tu aplicación.

8. ¿Cómo puedes configurar y utilizar "GitHub Codespaces" para desarrollar directamente en la nube?
   Activa GitHub Codespaces desde la pestaña "Code" en tu repositorio y utiliza el entorno de desarrollo basado en la nube proporcionado para escribir, ejecutar y depurar código.
9. ¿Cómo puedes realizar un análisis de seguridad en tu código utilizando GitHub Advanced Security?
   Habilita GitHub Advanced Security en la configuración del repositorio para utilizar herramientas de análisis estático y revisión de seguridad en el código fuente.
10. ¿Cómo puedes gestionar permisos y accesos a nivel de repositorio utilizando equipos y organizaciones en GitHub?
    Configura permisos para equipos y miembros dentro de una organización en GitHub, asignando roles y accesos a nivel de repositorio para gestionar colaboraciones y protecciones.
11. ¿Cómo puedes realizar un "squash merge" desde la interfaz web de GitHub y qué ventajas ofrece?
    Utiliza la opción "Squash and merge" en la interfaz web de GitHub para combinar todos los commits de un pull request en uno solo, simplificando el historial del repositorio.
12. ¿Cómo puedes utilizar "GitHub Actions" para implementar un flujo de trabajo de pruebas y despliegue continuo (CI/CD)?
    Define flujos de trabajo en archivos YAML dentro de `.github/workflows` para automatizar la ejecución de pruebas y despliegues continuos en tu repositorio.
13. ¿Cómo puedes gestionar dependencias de submódulos en diferentes ramas utilizando Git?
    Cambia de rama en el repositorio principal y actualiza los submódulos con `git submodule update`, asegurándote de sincronizar las versiones de submódulos con las ramas correspondientes.
14. ¿Qué es una "policy" de GitHub y cómo se configura para restringir el acceso a ramas específicas?
    Configura políticas en la sección "Branch protection rules" para restringir el acceso a ramas específicas, requiriendo revisiones, pruebas y restricciones de fuerza de push.
15. ¿Cómo puedes integrar GitHub con herramientas de análisis de código estático como SonarQube o CodeClimate?
    Configura integraciones en GitHub Actions o utiliza la sección "Integrations" en el repositorio para conectar con herramientas de análisis de código estático y mostrar resultados en el repositorio.
16. ¿Cómo puedes gestionar la historia de commits y revisiones utilizando `git reflog`?
    Utiliza el comando `git reflog` para acceder al registro de todas las referencias de Git, permitiendo recuperar commits y referencias que han sido movidos o eliminados.
17. ¿Cómo puedes configurar una "GitHub Pages" para publicar un sitio web estático desde tu repositorio?
    Crea una rama específica como `gh-pages` o utiliza la carpeta `docs/` y configura

GitHub Pages en la sección "Settings" para publicar un sitio web estático desde tu repositorio.

18. ¿Cómo puedes utilizar "git filter-branch" para reescribir el historial de un repositorio y eliminar datos sensibles?
    Utiliza `git filter-branch` para realizar cambios en el historial de commits, como eliminar archivos sensibles, reescribiendo el historial y forzando el push de los cambios.
19. ¿Cómo puedes configurar notificaciones personalizadas para eventos específicos en GitHub utilizando Webhooks?
    Configura Webhooks en la sección "Settings" del repositorio para recibir notificaciones personalizadas sobre eventos específicos, enviando datos a un endpoint configurado.
20. ¿Cómo puedes gestionar versiones de lanzamiento (releases) y changelogs utilizando GitHub?
    Crea versiones de lanzamiento en la pestaña "Releases", asigna tags y utiliza los changelogs para documentar los cambios entre versiones.
21. ¿Cómo puedes automatizar la gestión de dependencias y actualizaciones en tu proyecto utilizando GitHub Actions?
    Configura flujos de trabajo en GitHub Actions para automatizar la actualización de dependencias y la ejecución de pruebas para asegurar la integridad del proyecto.
22. ¿Cómo puedes utilizar `git reflog` para recuperar commits que han sido eliminados por un `git reset`?
    Usa `git reflog` para encontrar el hash del commit perdido y luego utiliza `git checkout -b nueva-rama hash-del-commit` para recuperar el commit eliminado.
23. ¿Cómo puedes personalizar el flujo de trabajo de CI/CD en GitHub Actions utilizando diferentes estrategias de caché?
    Configura cachés en tus flujos de trabajo de GitHub Actions utilizando la clave de caché y los pasos `actions/cache` para optimizar el tiempo de ejecución y reutilizar dependencias.
24. ¿Cómo puedes automatizar el despliegue de aplicaciones web utilizando "GitHub Actions" y "Heroku"?
    Configura un flujo de trabajo en GitHub Actions que utilice la acción `heroku/deploy` para desplegar automáticamente tu aplicación a Heroku cada vez que se realice un push a la rama principal.
25. ¿Cómo puedes utilizar "GitHub Insights" para analizar el rendimiento y la actividad del equipo en un repositorio?
    Navega a la pestaña "Insights" para acceder a métricas y gráficos sobre la actividad del equipo, la frecuencia de commits y la colaboración en el repositorio.
26. ¿Cómo puedes utilizar el comando `git merge --no-ff` para asegurar un historial de commits más claro?
    Utiliza `git merge --no-ff` para realizar una fusión que preserva el historial de commits de la rama, evitando la fusión rápida (fast-forward) y manteniendo un registro claro.

27. ¿Cómo puedes configurar y utilizar "GitHub Codespaces" para proporcionar entornos de desarrollo estandarizados a todo tu equipo?
    Configura plantillas de Codespaces con dependencias y configuraciones necesarias para estandarizar los entornos de desarrollo y facilitar el onboarding de nuevos miembros del equipo.
28. ¿Cómo puedes utilizar `git filter-repo` para limpiar y reescribir el historial de un repositorio?
    Utiliza el comando `git filter-repo` para reescribir el historial de commits y realizar tareas de limpieza, como eliminar archivos o directorios del historial.
29. ¿Cómo puedes colaborar en un repositorio utilizando pull requests, revisiones y comentarios?
    Crea pull requests para proponer cambios, realiza revisiones y comentarios en el código, y coordina con el equipo para fusionar los cambios aprobados en la rama principal.
30. ¿Cómo puedes utilizar "GitHub Marketplace" para encontrar y utilizar herramientas y acciones predefinidas en tu flujo de trabajo de CI/CD?
    Navega por GitHub Marketplace para encontrar acciones, aplicaciones y herramientas que se integran con tu flujo de trabajo de CI/CD, y agrégalas a tus flujos de trabajo en `.github/workflows`.
31. ¿Cómo puedes configurar "GitHub Actions" para realizar despliegues automáticos a múltiples entornos (staging y producción)?
    Define flujos de trabajo separados en `.github/workflows` para manejar despliegues a diferentes entornos, utilizando variables y secretos para gestionar configuraciones específicas.
32. ¿Cómo puedes gestionar el historial de cambios en un repositorio utilizando `git log` y sus opciones avanzadas?
    Utiliza `git log --oneline --graph --decorate` para visualizar el historial de commits de forma compacta, gráfica y con decoraciones de ramas y tags.
33. ¿Cómo puedes realizar un "cherry-pick" de un commit desde una rama a otra utilizando GitHub Desktop?
    Utiliza la interfaz de GitHub Desktop para seleccionar un commit en una rama y aplicarlo a otra rama mediante la opción de "cherry-pick" disponible en la interfaz de usuario.
34. ¿Cómo puedes integrar GitHub con sistemas de gestión de proyectos como Jira o Trello para mejorar la coordinación del equipo?
    Utiliza aplicaciones y conexiones disponibles en GitHub Marketplace para sincronizar issues y pull requests con sistemas de gestión de proyectos como Jira o Trello.
35. ¿Cómo puedes asegurar el código fuente utilizando "GitHub Advanced Security" y configuraciones de escaneo de vulnerabilidades?
    Configura GitHub Advanced Security para habilitar el escaneo de código en busca de vulnerabilidades y dependencias inseguras, y revisa los informes generados para asegurar el código.

36. ¿Cómo puedes gestionar y visualizar diferentes versiones de tu código utilizando "GitHub Releases" y tags?
    Crea versiones de lanzamiento y tags en GitHub para etiquetar versiones específicas del código, y utiliza la sección "Releases" para gestionar y documentar cambios importantes.
37. ¿Cómo puedes colaborar con equipos distribuidos utilizando "GitHub Discussions" para resolver problemas y compartir conocimientos?
    Habilita y utiliza GitHub Discussions para permitir que los miembros del equipo realicen preguntas, compartan ideas y resuelvan problemas en un espacio de discusión colaborativo.
38. ¿Cómo puedes utilizar "GitHub Projects" para gestionar tareas y sprints dentro de un repositorio?
    Configura tableros en GitHub Projects para gestionar tareas, planificar sprints y realizar un seguimiento del progreso de las actividades dentro del repositorio.
39. ¿Cómo puedes configurar una política de revisión de código en GitHub para requerir revisiones de múltiples miembros del equipo antes de fusionar cambios?
    Establece políticas en la sección "Branch protection rules" para requerir revisiones de un número específico de revisores antes de permitir la fusión de pull requests.
40. ¿Cómo puedes utilizar `git stash` para guardar cambios temporales y recuperar un estado limpio en tu repositorio?
    Usa `git stash` para guardar cambios no confirmados en un stash temporal, y recupera el estado limpio del repositorio para trabajar en otras tareas antes de aplicar el stash nuevamente.
41. ¿Cómo puedes integrar "GitHub Actions" con servicios de notificaciones como Slack para recibir alertas sobre eventos importantes en tu repositorio?
    Configura flujos de trabajo en GitHub Actions para enviar notificaciones a Slack utilizando un webhook y la acción correspondiente, para mantener al equipo informado sobre eventos importantes.
42. ¿Cómo puedes utilizar "GitHub API" para automatizar tareas administrativas y gestionar repositorios de forma programática?
    Utiliza la GitHub API para realizar operaciones como crear y gestionar repositorios, issues, pull requests y otros aspectos del repositorio mediante solicitudes HTTP programáticas.
43. ¿Cómo puedes gestionar la privacidad de un repositorio utilizando configuraciones avanzadas de visibilidad y acceso en GitHub?
    Ajusta la visibilidad del repositorio (público o privado) y configura permisos y accesos específicos para colaboradores y equipos en la sección "Settings" del repositorio.
44. ¿Cómo puedes utilizar "GitHub Marketplace" para encontrar y configurar herramientas de automatización para tu flujo de trabajo de desarrollo?
    Navega por GitHub Marketplace para encontrar herramientas y acciones que se integren con tus flujos de trabajo, y configúralas en tus archivos de flujo de trabajo en `.github/workflows`.

45. ¿Cómo puedes realizar auditorías de seguridad en el código fuente utilizando herramientas integradas de GitHub?
    Utiliza herramientas de auditoría de seguridad como "CodeQL" y "Dependabot" para revisar el código fuente y las dependencias en busca de vulnerabilidades y problemas de seguridad.
46. ¿Cómo puedes colaborar en un repositorio de código abierto utilizando "GitHub Sponsors" para apoyar a los mantenedores y contribuyentes?
    Utiliza "GitHub Sponsors" para apoyar a mantenedores y contribuyentes de proyectos de código abierto mediante donaciones y patrocinios, promoviendo la sostenibilidad del proyecto.
47. ¿Cómo puedes utilizar `git cherry-pick` para aplicar un commit específico desde una rama a otra sin fusionar ramas?
    Usa `git cherry-pick commit-hash` para aplicar un commit específico de una rama a otra, evitando la fusión completa y aplicando solo el commit seleccionado.
48. ¿Cómo puedes utilizar `git rebase` para actualizar una rama con los últimos cambios de la rama principal?
    Utiliza `git rebase main` mientras estás en la rama secundaria para aplicar los últimos cambios de la rama principal a tu rama, manteniendo un historial de commits limpio.
49. ¿Cómo puedes configurar `git hooks` para automatizar tareas como linters y pruebas antes de realizar un commit?
    Crea y configura `git hooks` en el directorio `.git/hooks` para ejecutar scripts personalizados antes de commits, merges u otras operaciones, automatizando tareas como linters y pruebas.
50. ¿Cómo puedes gestionar y actualizar submódulos en un repositorio utilizando GitHub?
    Actualiza submódulos en el repositorio principal utilizando `git submodule update --remote` y realiza cambios en el repositorio principal para reflejar las actualizaciones de los submódulos.

# Comandos de Git

1. `git init`
   Inicializa un nuevo repositorio Git en el directorio actual.
2. `git clone <URL-del-repositorio>`
   Clona un repositorio remoto a tu máquina local.
3. `git add <nombre-del-archivo>`
   Añade un archivo específico al área de staging.
4. `git commit -m "mensaje del commit"`
   Confirma los cambios en el área de staging con un mensaje de commit.
5. `git status`
   Muestra el estado actual del repositorio, incluyendo archivos modificados y cambios no confirmados.

6. `git log`
   Muestra el historial de commits en el repositorio.
7. `git diff`
   Muestra las diferencias entre el estado actual y el último commit.
8. `git checkout <nombre-de-la-rama>`
   Cambia a una rama específica en el repositorio.
9. `git branch`
   Muestra una lista de ramas en el repositorio y la rama actual.
10. `git merge <nombre-de-la-rama>`
    Fusiona una rama específica en la rama actual.
11. `git rebase <nombre-de-la-rama>`
    Reaplica los commits de la rama actual sobre otra rama para actualizarla con los últimos cambios.
12. `git cherry-pick <hash-del-commit>`
    Aplica un commit específico de otra rama a la rama actual.
13. `git stash`
    Guarda los cambios no confirmados en un stash temporal para limpiar el estado del repositorio.
14. `git stash pop`
    Recupera y aplica los cambios guardados en el stash más reciente.
15. `git reset`
    Deshace cambios en el área de staging o en el repositorio, dependiendo de las opciones utilizadas.
16. `git rm <nombre-del-archivo>`
    Elimina un archivo del repositorio y del área de staging.
17. `git fetch`
    Descarga los cambios del repositorio remoto sin aplicarlos al repositorio local.
18. `git pull`
    Descarga y aplica los cambios del repositorio remoto a la rama actual.
19. `git push`
    Envía los commits locales a un repositorio remoto.
20. `git tag`
    Crea, lista o elimina tags en el repositorio para marcar puntos específicos en la historia del proyecto.
21. `git remote`
    Gestiona las conexiones a repositorios remotos, como añadir, eliminar o listar remotos.
22. `git log --oneline`
    Muestra el historial de commits en una sola línea por commit.
23. `git commit --amend`
    Modifica el último commit para añadir cambios o actualizar el mensaje de commit.

24. `git revert <hash-del-commit>`
    Crea un nuevo commit que deshace los cambios introducidos por un commit específico.
25. `git rebase -i <hash-del-commit>`
    Realiza un rebase interactivo para reordenar, modificar o combinar commits.
26. `git merge --squash <nombre-de-la-rama>`
    Combina todos los commits de una rama en un solo commit y los añade al área de staging.
27. `git reflog`
    Muestra el historial de referencias de HEAD, útil para recuperar commits perdidos.
28. `git clean`
    Elimina archivos no rastreados del directorio de trabajo.
29. `git submodule`
    Gestiona submódulos en un repositorio Git, que son repositorios dentro de un repositorio principal.
30. `git grep <patrón>`
    Busca un patrón en el contenido de archivos en el repositorio.

# 1. Jr

1. ¿Qué es JavaScript?
   Es un lenguaje de programación utilizado principalmente para agregar interactividad a las páginas web.
2. ¿Cuál es la diferencia entre `var`, `let` y `const`?
   `var` tiene un alcance de función, mientras que `let` y `const` tienen un alcance de bloque. `const` no permite reasignar el valor.
3. ¿Qué es una función en JavaScript?
   Es un bloque de código que realiza una tarea específica y puede ser reutilizado.
4. ¿Qué es una variable en JavaScript?
   Es un contenedor que almacena datos que pueden cambiar durante la ejecución del programa.
5. ¿Qué es un array en JavaScript?
   Es una estructura que permite almacenar múltiples valores en una sola variable.
6. ¿Qué es un objeto en JavaScript?
   Es una colección de propiedades, donde cada propiedad tiene un nombre y un valor.
7. ¿Qué es el operador `==` en JavaScript?
   Es un operador que compara dos valores sin tener en cuenta su tipo de dato.
8. ¿Qué es el operador `===` en JavaScript?
   Es un operador que compara dos valores y sus tipos de datos.
9. ¿Qué significa `NaN` en JavaScript?
   Significa "Not-a-Number", y se produce cuando una operación matemática no tiene un resultado válido.

10. ¿Qué es el `typeof` en JavaScript?
    Es un operador que devuelve el tipo de dato de una variable o expresión.
11. ¿Qué es `null` en JavaScript?
    Es un valor que indica la ausencia intencional de un valor.
12. ¿Qué es `undefined` en JavaScript?
    Indica que una variable ha sido declarada pero no se le ha asignado un valor.
13. ¿Qué es el hoisting en JavaScript?
    Es el comportamiento por el cual las declaraciones de variables y funciones se elevan al comienzo de su contexto de ejecución.
14. ¿Qué es el ámbito de una variable en JavaScript?
    Es la región del código en la que una variable es accesible.
15. ¿Qué es una expresión de función en JavaScript?
    Es una función que puede ser asignada a una variable.
16. ¿Qué es una función flecha en JavaScript?
    Es una forma concisa de escribir funciones usando la sintaxis `=>`.
17. ¿Qué es el objeto `this` en JavaScript?
    `this` hace referencia al objeto al que pertenece la función que lo llama.
18. ¿Qué es una promesa en JavaScript?
    Es un objeto que representa la eventual finalización o falla de una operación asíncrona.
19. ¿Qué es el modelo de eventos en JavaScript?
    Es un mecanismo que permite manejar interacciones de usuario mediante eventos como clics o entradas de teclado.
20. ¿Qué es el `event.preventDefault()` en JavaScript?
    Es un método que detiene el comportamiento predeterminado de un evento.
21. ¿Qué es el `addEventListener()` en JavaScript?
    Es un método que permite escuchar un evento específico y ejecutar una función cuando ese evento ocurre.
22. ¿Qué es el DOM en JavaScript?
    El DOM (Document Object Model) es una representación estructurada del contenido HTML de una página web.
23. ¿Qué es un bucle en JavaScript?
    Es una estructura de control que permite repetir un bloque de código múltiples veces.
24. ¿Qué es un callback en JavaScript?
    Es una función que se pasa como argumento a otra función y que se ejecuta después de que se completa una operación.
25. ¿Qué es un operador ternario en JavaScript?
    Es una forma concisa de escribir una expresión condicional con `?` y `:`.
26. ¿Qué es el método `Array.prototype.map()` en JavaScript?
    Es un método que crea un nuevo array con los resultados de aplicar una función a cada elemento de un array.

27. ¿Qué es el método `Array.prototype.filter()` en JavaScript?
    Es un método que crea un nuevo array con los elementos que cumplen con una condición especificada.
28. ¿Qué es el método `Array.prototype.reduce()` en JavaScript?
    Es un método que aplica una función a un acumulador y a cada elemento de un array, reduciéndolo a un solo valor.
29. ¿Qué es el JSON en JavaScript?
    JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos.
30. ¿Qué es el `JSON.stringify()` en JavaScript?
    Es un método que convierte un objeto de JavaScript en una cadena JSON.

# 2. Ssr

1. ¿Qué es el Event Loop en JavaScript?
   Es el mecanismo que maneja la ejecución de operaciones asíncronas y coordina las tareas en la cola de eventos.
2. ¿Qué es el call stack en JavaScript?
   Es una estructura que lleva el control de las funciones que se están ejecutando en el programa.
3. ¿Qué son las closures en JavaScript?
   Es una función que puede recordar y acceder a su ámbito léxico, incluso cuando se ejecuta fuera de dicho ámbito.
4. ¿Qué diferencia hay entre `call()`, `apply()` y `bind()` en JavaScript?
   Son métodos que permiten cambiar el contexto ( `this` ) de una función.
5. ¿Qué son los módulos en JavaScript?
   Los módulos permiten dividir el código en partes reutilizables y organizadas, exportando e importando funciones, objetos o valores.
6. ¿Qué es el patrón de diseño módulo en JavaScript?
   Es un patrón que encapsula variables y métodos dentro de una función, exponiendo solo lo necesario.
7. ¿Qué es la coerción en JavaScript?
   Es el proceso mediante el cual JavaScript convierte automáticamente un tipo de dato en otro.
8. ¿Cuál es la diferencia entre coerción implícita y explícita?
   La coerción implícita ocurre automáticamente por el motor de JavaScript, mientras que la explícita es forzada por el programador.
9. ¿Qué es el `event delegation` en JavaScript?
   Es una técnica para manejar eventos de manera más eficiente delegando el manejo de eventos a un solo ancestro común.
10. ¿Qué es el bubbling y el capturing en JavaScript?
    Son fases en las que los eventos se propagan por el DOM, desde el elemento más

profundo hasta el más externo (bubbling) o desde el más externo al más profundo (capturing).

11. ¿Qué es un `polyfill` en JavaScript?
    Es un pedazo de código que implementa funcionalidades que no están disponibles en navegadores antiguos.
12. ¿Qué es el `debouncing` en JavaScript?
    Es una técnica que asegura que una función no se ejecute hasta que haya pasado un tiempo determinado desde la última vez que fue invocada.
13. ¿Qué es el `throttling` en JavaScript?
    Es una técnica que limita la frecuencia con la que se ejecuta una función en un periodo determinado.
14. ¿Cuál es la diferencia entre `setTimeout` y `setInterval`?
    `setTimeout` ejecuta una función una vez después de un retraso, mientras que `setInterval` ejecuta una función repetidamente a intervalos regulares.
15. ¿Qué es la recursión en JavaScript?
    Es una técnica en la que una función se llama a sí misma para resolver un problema.
16. ¿Qué es la delegación de prototipos en JavaScript?
    Es el mecanismo mediante el cual los objetos en JavaScript heredan propiedades y métodos de su prototipo.
17. ¿Cuál es la diferencia entre herencia prototípica y herencia clásica?
    La herencia prototípica se basa en la copia de objetos, mientras que la herencia clásica se basa en clases y objetos instanciados.
18. ¿Qué es el ámbito léxico en JavaScript?
    Es la estructura de alcance de una función en función de dónde fue declarada en el código.
19. ¿Cómo funciona el `strict mode` en JavaScript?
    El `strict mode` es una característica que introduce reglas más estrictas en el lenguaje, ayudando a evitar errores comunes.
20. ¿Qué es la memoización en JavaScript?
    Es una técnica de optimización en la que se almacena el resultado de una función para evitar cálculos repetidos.
21. ¿Qué son los iteradores y generadores en JavaScript?
    Los iteradores son objetos que permiten recorrer colecciones de datos, mientras que los generadores son funciones que pueden pausar su ejecución y luego reanudarla.
22. ¿Qué es el concepto de inmutabilidad en JavaScript?
    Es la idea de que ciertos valores, como los tipos primitivos, no pueden cambiarse una vez creados.
23. ¿Qué son los `weakMap` y `weakSet` en JavaScript?
    Son estructuras de datos que permiten almacenar referencias a objetos sin evitar que estos sean eliminados por el recolector de basura.
24. ¿Qué es el currying en JavaScript?
    Es una técnica que permite transformar una función con múltiples argumentos en una

secuencia de funciones que reciben un solo argumento.

25. ¿Cuál es la diferencia entre el modelo síncrono y asíncrono en JavaScript?
    El modelo síncrono ejecuta las tareas de manera secuencial, mientras que el asíncrono permite realizar tareas sin bloquear el hilo principal.
26. ¿Qué son las funciones de orden superior en JavaScript?
    Son funciones que pueden tomar otras funciones como argumentos o devolver funciones como resultado.
27. ¿Cuál es la diferencia entre `localStorage` y `sessionStorage`?
    `localStorage` almacena datos sin fecha de expiración, mientras que `sessionStorage` los almacena solo durante la sesión del navegador.
28. ¿Qué es el `Service Worker` en JavaScript?
    Es un script que el navegador ejecuta en segundo plano, lo que permite el almacenamiento en caché y la funcionalidad fuera de línea en aplicaciones web.
29. ¿Qué es la propagación de eventos en JavaScript?
    Es el proceso por el cual los eventos se transmiten a través del DOM, desde el elemento objetivo hasta sus ancestros.
30. ¿Qué es el operador de propagación ( `spread operator` ) en JavaScript?
    Es una sintaxis que permite expandir elementos de arrays o propiedades de objetos en otros arrays u objetos.

# 3. Sr

1. ¿Qué es el concepto de `event loop` y cómo afecta el rendimiento de aplicaciones JavaScript?
   Es el proceso que maneja las operaciones asíncronas y la ejecución de código, coordinando la ejecución de las tareas y los eventos.
2. ¿Qué son los microtasks y macrotasks en JavaScript, y cómo difieren en términos de ejecución?
   Microtasks tienen mayor prioridad y se ejecutan antes que las macrotasks después de cada ciclo del `event loop`.
3. ¿Qué son los módulos ES6 y cómo mejoran la estructura del código en JavaScript?
   Los módulos permiten dividir el código en archivos reutilizables, ofreciendo una mejor separación de responsabilidades y evitando conflictos en el espacio de nombres.
4. ¿Cómo funcionan las `weak references` y el `garbage collector` en JavaScript?
   Las `weak references` permiten hacer referencias a objetos sin impedir que el `garbage collector` los elimine cuando ya no son necesarios.
5. ¿Qué es la optimización de tail-call en JavaScript y cuándo se aplica?
   Es una técnica que optimiza el uso de la pila en funciones recursivas, permitiendo que la llamada final de una función no aumente el tamaño del call stack.
6. ¿Cómo afectan los motores JavaScript como V8 a la ejecución y optimización del código?

Los motores como V8 convierten el código JavaScript en código máquina optimizado, mejorando el rendimiento a través de técnicas como JIT (Just-In-Time) compilation.

7. ¿Cómo funciona el ciclo de vida de un Promise en JavaScript y cuáles son sus estados?
   Una Promise tiene tres estados: pendiente, resuelta y rechazada. Cambia de estado cuando la operación asíncrona finaliza.
8. ¿Qué es la optimización de "hidden classes" en los motores JavaScript?
   Es una técnica de los motores como V8 para optimizar el acceso a las propiedades de los objetos mediante la creación de clases ocultas que mejoran el rendimiento.
9. ¿Qué es el "defer" y "async" en la carga de scripts, y cuándo deberías usarlos?
   Son atributos que controlan el momento en que se ejecutan los scripts, `defer` espera a que el DOM se cargue, mientras que `async` permite la ejecución inmediata sin bloquear la carga del DOM.
10. ¿Cómo puedes mejorar el rendimiento de una aplicación JavaScript utilizando técnicas de lazy loading?
    El lazy loading carga los recursos de manera diferida, mejorando el tiempo de carga inicial al descargar solo lo necesario cuando es requerido.
11. ¿Cómo se implementa el `requestAnimationFrame` para optimizar la animación en JavaScript?
    `requestAnimationFrame` sincroniza la actualización de la animación con la frecuencia de actualización de la pantalla, mejorando la fluidez y el rendimiento.
12. ¿Cuál es la diferencia entre el modelo de concurrencia de otros lenguajes y el de JavaScript?
    JavaScript utiliza un modelo de concurrencia basado en un único hilo de ejecución con un `event loop`, a diferencia de otros lenguajes que pueden usar múltiples hilos.
13. ¿Qué es el concepto de `functional programming` en JavaScript y cuáles son sus beneficios?
    Es un paradigma que trata las funciones como ciudadanos de primera clase, promoviendo la inmutabilidad, funciones puras y la composición.
14. ¿Cómo funciona la gestión de memoria en JavaScript y qué es el heap y el stack?
    El heap es donde se almacenan los objetos dinámicos, y el stack es donde se gestionan las llamadas a funciones y las variables primarias de corto plazo.
15. ¿Qué problemas puede resolver el uso de `proxy` en JavaScript?
    Los proxies permiten interceptar y redefinir operaciones fundamentales de objetos como lectura, escritura y borrado de propiedades.
16. ¿Cómo implementas el patrón de diseño Singleton en JavaScript y cuándo es útil?
    El patrón Singleton asegura que una clase tenga solo una instancia, siendo útil para gestionar configuraciones globales.
17. ¿Qué es el concepto de `memoization` en JavaScript y cómo mejora el rendimiento de las aplicaciones?
    Es una técnica que almacena los resultados de una función para evitar cálculos repetidos, mejorando el rendimiento en operaciones costosas.

18. ¿Cuál es la diferencia entre `deep copy` y `shallow copy` en JavaScript?
    Una copia superficial ( `shallow copy` ) copia solo las referencias de los objetos, mientras que una copia profunda ( `deep copy` ) copia el objeto completo, incluyendo las referencias anidadas.
19. ¿Qué son los `generators` y cómo facilitan el manejo de iteraciones en JavaScript?
    Los `generators` son funciones que pueden pausar su ejecución, devolviendo valores múltiples a lo largo del tiempo, lo que facilita la creación de iteradores personalizados.
20. ¿Qué es la "head of line blocking" en JavaScript y cómo se puede mitigar en aplicaciones web?
    Es un problema de bloqueo en las colas de solicitudes que puede afectar el rendimiento. Se puede mitigar usando HTTP/2 para múltiples streams simultáneos.
21. ¿Cómo manejas y previenes `memory leaks` en aplicaciones JavaScript?
    Se deben evitar referencias no utilizadas, gestionar eventos adecuadamente y liberar memoria asignada para evitar que los objetos queden en el heap de forma indefinida.
22. ¿Qué es un `Web Worker` en JavaScript y cuándo se debe usar?
    Un `Web Worker` permite ejecutar scripts en hilos en segundo plano, lo que mejora la capacidad de realizar tareas intensivas sin bloquear el hilo principal.
23. ¿Qué son los `Decorators` en JavaScript y cómo se pueden utilizar?
    Los `Decorators` son una propuesta en ECMAScript que permite modificar el comportamiento de clases y métodos de forma declarativa.
24. ¿Cómo implementas y gestionas correctamente la asincronía con `async/await` en JavaScript?
    `async/await` facilita el manejo de promesas, permitiendo escribir código asíncrono que se comporta como código sincrónico.
25. ¿Qué es el concepto de "hoisting" en JavaScript y cómo afecta el flujo de ejecución?
    El hoisting es el comportamiento de JavaScript de mover las declaraciones de variables y funciones al principio de su contexto de ejecución.
26. ¿Cuál es la diferencia entre `null` y `undefined` en JavaScript?
    `null` es un valor asignado intencionalmente para representar la ausencia de valor, mientras que `undefined` indica que una variable ha sido declarada pero no asignada.
27. ¿Qué es la recolección de basura en JavaScript y cómo funciona el "mark-and-sweep"?
    La recolección de basura es el proceso mediante el cual el motor de JavaScript libera memoria de objetos que ya no son accesibles, y el "mark-and-sweep" es un algoritmo que marca y elimina estos objetos.
28. ¿Cómo puedes usar `Intl` para la internacionalización en JavaScript?
    `Intl` es una API que facilita la localización de números, fechas y cadenas en diferentes formatos según la configuración regional del usuario.
29. ¿Qué es el "tail call optimization" y cómo impacta la recursividad en JavaScript?
    La "tail call optimization" es una característica que permite reutilizar el stack frame para llamadas recursivas, mejorando la eficiencia en funciones recursivas.

30. ¿Cuál es la diferencia entre la ejecución síncrona y asíncrona en JavaScript y cómo se puede gestionar la asíncronía?
    La ejecución síncrona bloquea el flujo de ejecución hasta que una operación se complete, mientras que la asíncrona permite que otras tareas se ejecuten sin esperar. La asíncronía se gestiona con `callbacks`, `promises` y `async/await`.

# 1. Jr

1. ¿Qué es Node.js y cuáles son sus principales características?
   Node.js es un entorno de ejecución para JavaScript basado en el motor V8 de Google Chrome. Sus principales características incluyen la ejecución de código JavaScript del lado del servidor, un modelo de I/O no bloqueante y un sistema de módulos para gestionar dependencias.
2. ¿Qué es el Event Loop en Node.js y cómo funciona?
   El Event Loop es un mecanismo que permite a Node.js manejar operaciones asíncronas. Funciona en un solo hilo y se encarga de gestionar las operaciones de I/O, delegar tareas a otros hilos cuando sea necesario y ejecutar callbacks cuando las operaciones se completan.
3. ¿Cómo se manejan los errores en una aplicación Node.js?
   Los errores se manejan utilizando bloques `try-catch` para código síncrono y callbacks de error para código asíncrono. También se pueden usar `process.on('uncaughtException')` y `process.on('unhandledRejection')` para manejar errores no capturados y promesas rechazadas.
4. ¿Qué es el sistema de módulos de Node.js y cómo se utiliza?
   El sistema de módulos de Node.js permite dividir el código en archivos separados para una mejor organización. Se utiliza `require()` para importar módulos y `module.exports` para exportar funcionalidades de un archivo a otro.
5. ¿Qué es Express y por qué se utiliza en aplicaciones Node.js?
   Express es un framework para Node.js que simplifica el desarrollo de aplicaciones web y APIs. Se utiliza para manejar rutas, middleware y solicitudes HTTP, facilitando la creación y gestión de servidores.
6. ¿Cómo se define una ruta en Express?
   Una ruta en Express se define utilizando métodos del objeto `app`, como `app.get()`, `app.post()`, `app.put()`, y `app.delete()`, donde se especifica la URL de la ruta y el controlador que maneja la solicitud.
7. ¿Qué es middleware en Express y cómo se utiliza?
   Middleware en Express son funciones que se ejecutan durante el ciclo de vida de una solicitud. Se utilizan para realizar tareas como la autenticación, la validación de datos y el manejo de errores. Se definen utilizando `app.use()` y se aplican a rutas específicas o globalmente.
8. ¿Cómo se manejan las solicitudes POST en Express?
   Las solicitudes POST en Express se manejan definiendo una ruta con `app.post()` y

procesando los datos del cuerpo de la solicitud utilizando el middleware `body-parser` o el parser de cuerpo integrado en Express.

9. ¿Qué es el middleware `body-parser` y cómo se configura en una aplicación Express?
   `body-parser` es un middleware que analiza el cuerpo de las solicitudes HTTP y lo convierte en un objeto JavaScript accesible en `req.body`. Se configura en Express utilizando `app.use(bodyParser.json())` para analizar JSON y `app.use(bodyParser.urlencoded({ extended: true }))` para analizar datos de formularios.
10. ¿Cómo se puede manejar el enrutamiento en una aplicación Express?
    El enrutamiento en Express se maneja definiendo rutas utilizando métodos como `app.get()`, `app.post()`, etc. También se pueden utilizar routers modulares con `express.Router()` para organizar las rutas en diferentes archivos.
11. ¿Qué es el objeto `req` en Express y qué información contiene?
    El objeto `req` (request) en Express representa la solicitud HTTP realizada por el cliente. Contiene información como los parámetros de la URL, el cuerpo de la solicitud (`req.body`), los encabezados de la solicitud (`req.headers`) y las cookies (`req.cookies`).
12. ¿Qué es el objeto `res` en Express y cómo se utiliza para enviar respuestas?
    El objeto `res` (response) en Express representa la respuesta HTTP que se enviará al cliente. Se utiliza para enviar respuestas con métodos como `res.send()`, `res.json()`, `res.redirect()`, y `res.status()`.
13. ¿Cómo se configura una aplicación Express para servir archivos estáticos?
    Se configura utilizando el middleware `express.static()` para especificar el directorio donde se almacenan los archivos estáticos, como imágenes, hojas de estilo y scripts. Se utiliza de la siguiente manera: `app.use(express.static('public'))`.
14. ¿Qué es una promesa en Node.js y cómo se utiliza?
    Una promesa es un objeto que representa la eventual finalización o fallo de una operación asíncrona. Se utiliza para manejar operaciones asíncronas con métodos como `.then()` y `.catch()` para encadenar acciones y manejar errores.
15. ¿Cómo se puede manejar la asincronía en Node.js con `async/await`?
    La asincronía en Node.js se puede manejar utilizando `async/await` para escribir código asíncrono de manera más legible. `async` convierte una función en una función que devuelve una promesa, y `await` se utiliza para esperar la resolución de una promesa.
16. ¿Cómo se implementan las rutas dinámicas en Express?
    Las rutas dinámicas en Express se implementan utilizando parámetros de ruta, que se definen en la URL de la ruta con `:`. Por ejemplo, en `app.get('/user/:id', (req, res) => { ... })`, `:id` es un parámetro dinámico que puede ser accedido con `req.params.id`.
17. ¿Qué es `app.use()` en Express y cómo se utiliza?
    `app.use()` es un método que se utiliza para definir middleware que se aplicará a

todas las rutas de la aplicación o a rutas específicas. Se utiliza para realizar tareas como el manejo de errores, la autenticación o el análisis de cuerpos de solicitud.

18. ¿Cómo se configuran variables de entorno en una aplicación Node.js?
    Las variables de entorno se configuran utilizando archivos `.env` y la biblioteca `dotenv`. Se carga el archivo `.env` con `require('dotenv').config()` y se accede a las variables con `process.env.NOMBRE_VARIABLE`.
19. ¿Qué es el método `next()` en Express y para qué se utiliza?
    El método `next()` se utiliza en middleware para pasar el control al siguiente middleware en la pila. Es esencial para encadenar múltiples middleware y para pasar el control al manejador de rutas o al siguiente middleware.
20. ¿Cómo se maneja la autenticación y autorización en Express?
    La autenticación y autorización en Express se manejan utilizando middleware que verifica las credenciales del usuario y controla el acceso a rutas protegidas. Se pueden usar bibliotecas como `passport` para la autenticación y `jsonwebtoken` para la autorización.
21. ¿Qué es `app.listen()` en Express y cómo se utiliza?
    `app.listen()` es un método que se utiliza para iniciar el servidor y hacer que escuche en un puerto específico. Se utiliza de la siguiente manera: `app.listen(port, () => { console.log(` Server running on port ${port} `); })`.
22. ¿Cómo se pueden manejar las solicitudes de tipo JSON en Express?
    Las solicitudes de tipo JSON se manejan utilizando el middleware `express.json()` para analizar el cuerpo de la solicitud y convertirlo en un objeto JavaScript accesible a través de `req.body`.
23. ¿Qué es `app.route()` y cómo se utiliza para manejar rutas en Express?
    `app.route()` es un método que permite encadenar múltiples manejadores de rutas para una URL específica. Se utiliza para definir rutas y métodos HTTP asociados de manera más organizada: `app.route('/path').get(handler).post(handler)`.
24. ¿Cómo se pueden realizar pruebas de una aplicación Express?
    Las pruebas de una aplicación Express se pueden realizar utilizando bibliotecas como `mocha`, `chai`, y `supertest`. Estas herramientas permiten escribir pruebas unitarias y de integración para las rutas y la lógica de la aplicación.
25. ¿Qué es un middleware de manejo de errores en Express y cómo se configura?
    Un middleware de manejo de errores se utiliza para capturar y manejar errores en la aplicación. Se configura utilizando una función con cuatro parámetros `(err, req, res, next)` que maneja el error y envía una respuesta adecuada al cliente.
26. ¿Cómo se pueden validar los datos en las solicitudes en Express?
    Los datos en las solicitudes se pueden validar utilizando bibliotecas como `express-validator` o `joi`. Estas herramientas permiten definir reglas de validación y procesar los errores de validación en el middleware.
27. ¿Qué es `res.sendFile()` y cómo se utiliza en Express?
    `res.sendFile()` es un método que se utiliza para enviar un archivo estático al cliente.

Se utiliza proporcionando la ruta del archivo y, opcionalmente, configurando el encabezado de la respuesta.

28. ¿Cómo se pueden manejar las solicitudes concurrentes en una aplicación Node.js?
    Node.js maneja solicitudes concurrentes utilizando su modelo de I/O no bloqueante y el Event Loop, lo que permite procesar múltiples solicitudes al mismo tiempo sin bloquear el hilo principal.
29. ¿Qué es `app.get()` en Express y cómo se utiliza?
    `app.get()` es un método que define una ruta para manejar solicitudes HTTP GET. Se utiliza especificando la URL de la ruta y una función de controlador que procesa la solicitud y envía una respuesta.
30. ¿Cómo se puede gestionar la configuración de una aplicación Express para diferentes entornos?
    La configuración para diferentes entornos se puede gestionar utilizando variables de entorno y la biblioteca `dotenv`. Se puede definir diferentes valores en el archivo `.env` para cada entorno y acceder a ellos mediante `process.env

# 2. Ssr

1. ¿Cómo se implementa la gestión de sesiones en una aplicación Express?
   La gestión de sesiones se implementa utilizando el middleware `express-session` para almacenar información de sesión en el servidor o en una base de datos. Configura el middleware con opciones como el nombre de la cookie, el secreto y el almacenamiento de sesiones.
2. ¿Qué es el middleware `helmet` en Express y para qué se utiliza?
   `helmet` es un middleware de seguridad para Express que ayuda a proteger la aplicación contra vulnerabilidades web al configurar varios encabezados de seguridad HTTP. Se utiliza para mejorar la seguridad del servidor aplicando buenas prácticas de seguridad.
3. ¿Cómo se configura un servidor Express para manejar solicitudes HTTPS?
   Se configura utilizando el módulo `https` de Node.js junto con `fs` para proporcionar certificados SSL/TLS. Se crea un servidor HTTPS y se pasa la instancia de Express como el manejador de solicitudes.
4. ¿Qué son las promesas en Node.js y cómo se manejan en una aplicación Express?
   Las promesas son objetos que representan el resultado eventual de una operación asíncrona. En una aplicación Express, se manejan utilizando `async/await` o los métodos `.then()` y `.catch()` para gestionar operaciones asíncronas y errores.
5. ¿Cómo se utiliza el middleware `morgan` para el registro de solicitudes en Express?
   `morgan` es un middleware de registro de solicitudes que se utiliza para registrar detalles de cada solicitud HTTP en la consola o en un archivo de registro. Se configura en Express con `app.use(morgan('combined'))` o el formato deseado.
6. ¿Cómo se implementan las pruebas unitarias para rutas en una aplicación Express?
   Las pruebas unitarias para rutas se implementan utilizando bibliotecas como `mocha`,

`chai` y `supertest`. Se escriben pruebas para verificar el comportamiento de las rutas y sus manejadores utilizando estas herramientas.

7. ¿Qué son los `sub-routers` en Express y cómo se utilizan?
   Los `sub-routers` son routers modulares que se utilizan para organizar rutas relacionadas en módulos separados. Se crean con `express.Router()` y se montan en la aplicación principal con `app.use('/prefix', router)`.
8. ¿Cómo se maneja la paginación en una API REST con Express?
   La paginación en una API REST se maneja agregando parámetros de consulta como `page` y `limit` en la solicitud. En el manejador de rutas, se utilizan estos parámetros para limitar y omitir resultados en la consulta a la base de datos.
9. ¿Qué es `express-validator` y cómo se utiliza para validar datos de entrada?
   `express-validator` es una biblioteca que proporciona un conjunto de middleware para validar y sanitizar datos en solicitudes. Se utiliza definiendo cadenas de validación en el middleware y manejando errores en el controlador.
10. ¿Cómo se implementa la lógica de reintento en solicitudes HTTP con `axios` en una aplicación Express?
    La lógica de reintento se implementa utilizando una biblioteca como `axios-retry` para configurar los intentos de reintento en caso de fallos en las solicitudes HTTP. Se configura con el número de reintentos y las condiciones para intentar nuevamente.
11. ¿Cómo se maneja la carga de archivos en Express utilizando `multer`?
    `multer` es un middleware para manejar la carga de archivos en Express. Se configura creando una instancia de `multer` con opciones como el almacenamiento y el límite de tamaño, y se utiliza en las rutas para procesar archivos cargados.
12. ¿Qué es `express.Router()` y cómo se diferencia de `app.use()`?
    `express.Router()` es una herramienta para definir rutas y middleware en módulos separados, proporcionando un enrutador modular. `app.use()` se utiliza para aplicar middleware y montar routers en la aplicación principal.
13. ¿Cómo se configura la caché en una aplicación Express?
    La caché se configura utilizando middleware como `apicache` o `cache-control`. Se define el tiempo de expiración y el comportamiento de caché en las respuestas HTTP para mejorar el rendimiento.
14. ¿Qué es el patrón de diseño Middleware en Express y cómo se aplica en una aplicación?
    El patrón de diseño Middleware en Express permite encadenar funciones que procesan solicitudes y respuestas en la aplicación. Se aplica utilizando `app.use()` para definir funciones de middleware que se ejecutan en el ciclo de vida de una solicitud.
15. ¿Cómo se gestionan las conexiones a una base de datos en una aplicación Express?
    Las conexiones a una base de datos se gestionan utilizando bibliotecas específicas para cada base de datos, como `mongoose` para MongoDB o `pg` para PostgreSQL. Se establecen conexiones en el archivo de configuración o en el archivo principal de la aplicación.

16. ¿Qué es `app.locals` en Express y cómo se utiliza?
    `app.locals` es un objeto que se utiliza para almacenar variables globales accesibles en todas las vistas y middlewares de la aplicación. Se utiliza para almacenar configuraciones, valores o datos que deben estar disponibles en toda la aplicación.
17. ¿Cómo se implementa la internacionalización en una aplicación Express?
    La internacionalización se implementa utilizando bibliotecas como `i18n` para gestionar traducciones y adaptaciones de contenido a diferentes idiomas. Se configura con middleware para cargar y aplicar traducciones según la configuración del usuario.
18. ¿Qué es `res.locals` en Express y cómo se utiliza?
    `res.locals` es un objeto que se utiliza para almacenar variables específicas de la respuesta que están disponibles para las vistas que se renderizan. Se utiliza para pasar datos a las vistas sin modificar el objeto `req`.
19. ¿Cómo se implementan las tareas en segundo plano en una aplicación Express?
    Las tareas en segundo plano se implementan utilizando bibliotecas como `bull` para gestionar colas de trabajos y tareas programadas. Se configuran para procesar trabajos en segundo plano sin bloquear el hilo principal del servidor.
20. ¿Qué es el middleware `compression` y cómo se utiliza en Express?
    `compression` es un middleware que se utiliza para comprimir las respuestas HTTP utilizando algoritmos como gzip. Se utiliza para reducir el tamaño de las respuestas y mejorar el rendimiento de la aplicación.
21. ¿Cómo se gestionan las variables de entorno en una aplicación Express?
    Las variables de entorno se gestionan utilizando el archivo `.env` y la biblioteca `dotenv`. Se cargan las variables de entorno con `require('dotenv').config()` y se accede a ellas mediante `process.env`.
22. ¿Qué es el middleware `cors` y cómo se configura en Express?
    `cors` es un middleware que se utiliza para habilitar el intercambio de recursos de origen cruzado (CORS) en la aplicación. Se configura con opciones como los orígenes permitidos y los métodos HTTP permitidos para controlar el acceso desde otros dominios.
23. ¿Cómo se manejan las solicitudes de API RESTful en Express utilizando controladores?
    Las solicitudes de API RESTful se manejan definiendo rutas en el archivo de rutas y delegando la lógica de manejo a controladores separados. Los controladores se encargan de procesar la solicitud, interactuar con la base de datos y enviar la respuesta.
24. ¿Cómo se implementa la lógica de autenticación y autorización en Express utilizando JWT?
    La lógica de autenticación y autorización se implementa utilizando JSON Web Tokens (JWT) para verificar la identidad del usuario y controlar el acceso a rutas protegidas. Se generan tokens durante el inicio de sesión y se verifican en rutas protegidas utilizando middleware.

25. ¿Qué es el patrón de diseño de Inversión de Control (IoC) y cómo se aplica en Express?
    El patrón de diseño de Inversión de Control (IoC) se aplica en Express utilizando inyección de dependencias para desacoplar la lógica de la aplicación de sus dependencias. Se puede implementar utilizando bibliotecas como `awilix` o `inversify`.
26. ¿Cómo se maneja la sincronización de datos entre diferentes instancias de un servidor Express?
    La sincronización de datos entre diferentes instancias de un servidor Express se maneja utilizando mecanismos de sincronización como bases de datos compartidas, servicios de mensajería como RabbitMQ o mecanismos de consenso distribuido.
27. ¿Qué es `express-async-errors` y cómo se utiliza en una aplicación Express?
    `express-async-errors` es una biblioteca que se utiliza para manejar errores en funciones asíncronas de Express sin necesidad de `try-catch`. Se instala y se requiere en la aplicación para simplificar el manejo de errores en rutas y middleware asíncronos.
28. ¿Cómo se implementa la gestión de transacciones en una aplicación Express con bases de datos relacionales?
    La gestión de transacciones se implementa utilizando la funcionalidad de transacciones proporcionada por el ORM o el cliente de la base de datos. Se utilizan métodos para iniciar, confirmar y deshacer transacciones según sea necesario.
29. ¿Qué es `express-rate-limit` y cómo se configura para proteger una aplicación Express de ataques de denegación de servicio?
    `express-rate-limit` es un middleware que se utiliza para limitar el número de solicitudes permitidas desde una misma dirección IP en un período de tiempo. Se configura especificando la cantidad máxima de solicitudes permitidas y el tiempo de expiración.
30. ¿Cómo se implementan las tareas programadas en una aplicación Express?
    Las tareas programadas se implementan utilizando bibliotecas como `node-cron` o

# 3. Sr

1. ¿Cómo se optimiza el rendimiento de una aplicación Express para manejar un alto volumen de solicitudes?
   Se optimiza utilizando técnicas como el balanceo de carga, la implementación de cachés en memoria y en disco, el uso de `cluster` para aprovechar múltiples núcleos de CPU, y la optimización de consultas a bases de datos.
2. ¿Cómo se implementa un sistema de microservicios utilizando Express?
   Se implementa dividiendo la aplicación en varios servicios independientes que se comunican entre sí mediante APIs REST o mensajería. Cada microservicio se desarrolla y despliega de manera autónoma, utilizando Express para manejar las solicitudes.

3. ¿Qué estrategias se pueden emplear para manejar la escalabilidad en una aplicación Express distribuida?
   Estrategias incluyen el uso de balanceadores de carga para distribuir las solicitudes entre instancias, la implementación de cachés distribuidos, y el uso de bases de datos escalables horizontalmente.
4. ¿Cómo se realiza la monitorización y el logging en una aplicación Express en producción?
   Se realiza utilizando herramientas de monitorización y logging como `winston` para registrar eventos, `prometheus` y `grafana` para métricas, y servicios como `New Relic` o `Datadog` para el monitoreo del rendimiento.
5. ¿Qué consideraciones de seguridad deben tomarse en cuenta al desarrollar una API RESTful con Express?
   Consideraciones incluyen la implementación de CORS adecuado, la validación y sanitización de datos de entrada, el uso de HTTPS, la protección contra ataques de inyección y la gestión segura de tokens de autenticación.
6. ¿Cómo se maneja la autenticación y autorización en una aplicación Express utilizando OAuth2?
   Se maneja implementando flujos de autorización y token utilizando bibliotecas como `passport` con estrategias de OAuth2. Se configuran endpoints para el flujo de autorización, la obtención de tokens y la validación de permisos.
7. ¿Cómo se asegura la integridad y la consistencia de los datos en una aplicación Express con múltiples instancias?
   Se asegura utilizando mecanismos de consenso distribuidos y bases de datos transaccionales que soportan consistencia eventual. También se pueden implementar técnicas de sincronización de datos y bloqueo optimista.
8. ¿Qué es el patrón de diseño CQRS (Command Query Responsibility Segregation) y cómo se aplica en Express?
   CQRS es un patrón que separa las operaciones de lectura y escritura en diferentes modelos. En Express, se aplica creando servicios separados para manejar comandos (escritura) y consultas (lectura), y utilizando diferentes bases de datos o modelos para cada uno.
9. ¿Cómo se realiza la configuración y gestión de entornos en una aplicación Express para diferentes entornos de despliegue?
   Se realiza utilizando archivos de configuración y variables de entorno específicas para cada entorno (desarrollo, prueba, producción). Se cargan las configuraciones adecuadas utilizando `dotenv` y se utilizan variables de entorno para manejar diferencias.
10. ¿Qué técnicas se pueden utilizar para asegurar que una aplicación Express sea resistente a fallos?
    Técnicas incluyen la implementación de retries en operaciones críticas, el uso de mecanismos de failover, la implementación de pruebas de resiliencia y la supervisión constante del estado del sistema.

11. ¿Cómo se integra una aplicación Express con sistemas de mensajería como RabbitMQ o Kafka?
    Se integra utilizando bibliotecas específicas como `amqplib` para RabbitMQ o `kafkajs` para Kafka. Se configuran productores y consumidores para enviar y recibir mensajes entre la aplicación y los sistemas de mensajería.
12. ¿Qué es el patrón de diseño API Gateway y cómo se aplica en un entorno con múltiples servicios Express?
    El patrón API Gateway centraliza la gestión de las solicitudes a múltiples servicios. Se aplica utilizando un servicio que actúa como intermediario, enruta las solicitudes a los servicios correspondientes y maneja la autenticación, la autorización y la agregación de respuestas.
13. ¿Cómo se implementa el versionado de una API RESTful en Express?
    Se implementa utilizando un esquema de versionado en las rutas, como `/api/v1/resource` y `/api/v2/resource`. Se gestionan diferentes versiones de la API en el código y se proporciona soporte para las versiones anteriores según sea necesario.
14. ¿Cómo se realiza la integración continua y la entrega continua (CI/CD) para una aplicación Express?
    Se realiza utilizando herramientas de CI/CD como Jenkins, GitHub Actions o GitLab CI. Se configuran pipelines para automatizar pruebas, construcciones y despliegues de la aplicación Express en diferentes entornos.
15. ¿Qué es el patrón de diseño de circuito breaker y cómo se aplica en Express?
    El patrón de diseño de circuito breaker protege a la aplicación contra fallos en servicios dependientes. Se aplica utilizando bibliotecas como `opossum` para monitorizar y gestionar fallos en las llamadas a servicios externos, evitando que fallos se propaguen.
16. ¿Cómo se implementa un sistema de autenticación basado en JWT en una aplicación Express?
    Se implementa generando y validando tokens JWT para autenticar usuarios. Se utiliza middleware para verificar el token en cada solicitud protegida, y se configura el proceso de emisión y revocación de tokens según sea necesario.
17. ¿Cómo se gestionan las conexiones a bases de datos distribuidas en una aplicación Express?
    Se gestionan utilizando bibliotecas que soportan bases de datos distribuidas y técnicas de replicación. Se configuran conexiones a múltiples nodos y se implementan estrategias de manejo de fallos y consistencia de datos.
18. ¿Qué es el patrón de diseño Repository y cómo se aplica en Express?
    El patrón Repository abstrae la lógica de acceso a datos en una capa de repositorios. En Express, se aplica creando clases o módulos que encapsulan la lógica de acceso a la base de datos, facilitando el mantenimiento y la prueba del código.
19. ¿Cómo se maneja la carga de trabajo de alto rendimiento en una aplicación Express?
    Se maneja utilizando técnicas como la implementación de balanceadores de carga, el

uso de cachés en memoria, la optimización de consultas a bases de datos, y la distribución de la carga entre múltiples instancias de la aplicación.

20. ¿Cómo se realiza la autenticación y autorización en una API GraphQL en lugar de REST con Express?
    Se realiza utilizando middleware específico para GraphQL que maneja la autenticación y autorización. Se pueden utilizar librerías como `graphql-middleware` para aplicar políticas de acceso y validar tokens en las consultas y mutaciones.
21. ¿Qué son los middlewares de autorización y cómo se implementan en una aplicación Express?
    Los middlewares de autorización verifican si un usuario tiene permisos para acceder a ciertos recursos. Se implementan como funciones que comprueban roles o permisos en el objeto de solicitud y determinan si la solicitud debe ser permitida.
22. ¿Cómo se implementa la compresión de respuestas en una aplicación Express para mejorar el rendimiento?
    Se implementa utilizando el middleware `compression` que comprime las respuestas HTTP utilizando algoritmos como gzip. Se configura con opciones para controlar el nivel de compresión y los tipos de respuestas que deben ser comprimidos.
23. ¿Cómo se maneja la coherencia eventual en una aplicación Express que utiliza una base de datos NoSQL?
    Se maneja utilizando técnicas como la replicación de datos y la sincronización entre nodos. Se implementan estrategias de resolución de conflictos y consistencia eventual según el modelo de consistencia del sistema de base de datos NoSQL.
24. ¿Cómo se implementa un sistema de autorización basado en roles en una aplicación Express?
    Se implementa utilizando middleware que verifica los roles del usuario y determina si tienen permisos para acceder a ciertos recursos. Se definen roles y permisos en la base de datos y se aplican políticas de autorización en el middleware.
25. ¿Qué es el patrón de diseño de Proxy y cómo se aplica en una aplicación Express?
    El patrón de diseño de Proxy actúa como intermediario entre el cliente y el servidor, controlando el acceso y la comunicación. En Express, se puede aplicar creando middleware que actúa como un proxy para manejar solicitudes y respuestas.
26. ¿Cómo se implementa un sistema de control de versiones en una API con Express?
    Se implementa utilizando rutas versionadas y técnicas de despliegue que soportan múltiples versiones de la API. Se gestiona la compatibilidad con versiones anteriores y se proporcionan rutas y documentación para cada versión.
27. ¿Cómo se maneja la integración de servicios externos en una aplicación Express, como servicios de terceros o APIs?
    Se maneja utilizando módulos o bibliotecas específicas para interactuar con servicios externos. Se configuran clientes para hacer solicitudes a servicios externos, se gestionan credenciales de acceso y se manejan respuestas y errores de manera adecuada.

28. ¿Qué es el patrón de diseño de Strategy y cómo se aplica en Express?
    El patrón de diseño de Strategy permite definir una familia de algoritmos y hacer que sean intercambiables. En Express, se puede aplicar para manejar diferentes estrategias de procesamiento o autenticación de solicitudes de manera modular.
29. ¿Cómo se implementa la gestión de errores global en una aplicación Express?
    Se implementa utilizando middleware de manejo de errores que captura y gestiona errores en toda la aplicación. Se define un middleware con cuatro parámetros (err, req, res, next) para manejar errores y enviar respuestas adecuadas.
30. ¿Cómo se optimiza la latencia de una aplicación Express que realiza muchas operaciones de E/S?
    Se optimiza utilizando técnicas como el uso de conexiones asíncronas, la implementación de cachés en memoria para

# Patrones

1. ¿Qué es un patrón de arquitectura de software?
   Un patrón de arquitectura de software es una solución reutilizable y generalizada para un problema común en el diseño de software, que proporciona una estructura y principios para construir sistemas de manera eficiente y efectiva.
2. ¿Cuál es el propósito del patrón de arquitectura de microservicios?
   El propósito del patrón de microservicios es dividir una aplicación en servicios independientes que se comunican entre sí a través de APIs, lo que facilita la escalabilidad, el mantenimiento y la implementación continua.
3. ¿Qué es el patrón de arquitectura de capas (Layered Architecture)?
   El patrón de arquitectura de capas organiza el software en capas distintas, cada una con una responsabilidad específica, como la capa de presentación, la capa de negocio y la capa de datos, para mejorar la separación de preocupaciones y la modularidad.
4. ¿Cómo funciona el patrón de arquitectura en espiral (Spiral Architecture)?
   El patrón de arquitectura en espiral se basa en un enfoque iterativo de desarrollo que combina el diseño y la construcción en ciclos repetitivos, permitiendo la evolución del sistema a través de prototipos y retroalimentación continua.
5. ¿Qué es el patrón de arquitectura de cliente-servidor?
   El patrón de arquitectura de cliente-servidor divide el sistema en dos partes: el cliente, que solicita servicios, y el servidor, que proporciona esos servicios. Esto permite una separación clara entre el cliente y el servidor, facilitando la comunicación y el mantenimiento.
6. ¿Qué es el patrón de arquitectura basada en eventos (Event-Driven Architecture)?
   El patrón de arquitectura basada en eventos se basa en la producción, detección y respuesta a eventos, permitiendo que los componentes del sistema reaccionen a cambios y eventos en tiempo real, facilitando la escalabilidad y la flexibilidad.
7. ¿Cómo funciona el patrón de arquitectura orientada a servicios (SOA)?
   El patrón de arquitectura orientada a servicios organiza el sistema en servicios

independientes que se comunican entre sí a través de interfaces bien definidas, promoviendo la reutilización de servicios y la integración entre diferentes aplicaciones.

8. ¿Qué es el patrón de arquitectura de repositorio (Repository Pattern)?
   El patrón de arquitectura de repositorio proporciona una capa de abstracción entre la capa de acceso a datos y la lógica de negocio, permitiendo un acceso más organizado y desacoplado a los datos de la aplicación.
9. ¿Cómo se aplica el patrón de arquitectura de capas en una aplicación web?
   En una aplicación web, el patrón de arquitectura de capas puede dividirse en capas como la capa de presentación (UI), la capa de lógica de negocio (servicios), y la capa de acceso a datos (repositorios), cada una encargada de una responsabilidad específica.
10. ¿Qué es el patrón de arquitectura de fachada (Facade Pattern)?
    El patrón de fachada proporciona una interfaz simplificada para un conjunto complejo de interfaces o subsistemas, facilitando la interacción del cliente con el sistema y ocultando la complejidad interna.
11. ¿Cuál es el objetivo del patrón de arquitectura de puente (Bridge Pattern)?
    El patrón de arquitectura de puente desacopla una abstracción de su implementación, permitiendo que ambas evolucionen independientemente. Esto se logra mediante la separación de la interfaz de la implementación subyacente.
12. ¿Qué es el patrón de arquitectura de productor-consumidor (Producer-Consumer Pattern)?
    El patrón de productor-consumidor organiza la comunicación entre dos componentes en el que el productor genera datos y el consumidor procesa esos datos, utilizando una cola intermedia para gestionar la sincronización y el flujo de datos.
13. ¿Cómo se utiliza el patrón de arquitectura de inyección de dependencias (Dependency Injection)?
    El patrón de inyección de dependencias proporciona objetos necesarios a una clase desde el exterior, en lugar de crear los objetos dentro de la clase. Esto facilita la modularidad y la prueba del sistema al permitir la inyección de dependencias.
14. ¿Qué es el patrón de arquitectura de método de plantilla (Template Method Pattern)?
    El patrón de método de plantilla define el esqueleto de un algoritmo en una clase base, permitiendo que las subclases implementen pasos específicos del algoritmo sin cambiar su estructura general.
15. ¿Cómo funciona el patrón de arquitectura de proxy (Proxy Pattern)?
    El patrón de proxy actúa como un intermediario entre un cliente y un objeto real, proporcionando un control adicional sobre el acceso al objeto real, como la gestión de permisos, la optimización de rendimiento o la carga diferida.
16. ¿Qué es el patrón de arquitectura de comando (Command Pattern)?
    El patrón de comando encapsula una solicitud como un objeto, permitiendo la parametrización de clientes con diferentes solicitudes, la cola o el registro de solicitudes y la deshacer de operaciones.

17. ¿Cómo se aplica el patrón de arquitectura de decorador (Decorator Pattern)?
    El patrón de decorador añade funcionalidades a un objeto de manera dinámica y flexible, envolviendo el objeto original con un decorador que proporciona la funcionalidad adicional sin modificar la estructura del objeto original.
18. ¿Qué es el patrón de arquitectura de estado (State Pattern)?
    El patrón de estado permite a un objeto cambiar su comportamiento cuando su estado interno cambia, lo que hace que el objeto parezca cambiar su clase. Esto se logra mediante la delegación del comportamiento a objetos de estado específicos.
19. ¿Cómo se utiliza el patrón de arquitectura de estrategia (Strategy Pattern)?
    El patrón de estrategia define una familia de algoritmos, encapsula cada uno y los hace intercambiables, permitiendo que el algoritmo varíe independientemente del cliente que lo utiliza.
20. ¿Qué es el patrón de arquitectura de observador (Observer Pattern)?
    El patrón de observador define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
21. ¿Cómo funciona el patrón de arquitectura de iterador (Iterator Pattern)?
    El patrón de iterador proporciona una manera de acceder a los elementos de una colección de objetos secuencialmente sin exponer la representación interna de la colección.
22. ¿Qué es el patrón de arquitectura de adaptador (Adapter Pattern)?
    El patrón de adaptador permite que dos interfaces incompatibles colaboren entre sí, convirtiendo la interfaz de una clase en otra que el cliente espera, actuando como un puente entre las dos interfaces.
23. ¿Cómo se aplica el patrón de arquitectura de fábrica abstracta (Abstract Factory Pattern)?
    El patrón de fábrica abstracta proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas, permitiendo la creación de objetos que cumplen con un conjunto específico de requisitos.
24. ¿Qué es el patrón de arquitectura de singleton (Singleton Pattern)?
    El patrón de singleton garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia, controlando la creación y el acceso a la instancia única.
25. ¿Cómo funciona el patrón de arquitectura de cadena de responsabilidad (Chain of Responsibility Pattern)?
    El patrón de cadena de responsabilidad permite pasar una solicitud a lo largo de una cadena de objetos receptores hasta que uno de ellos maneja la solicitud, desacoplando el emisor de la solicitud y el receptor.
26. ¿Qué es el patrón de arquitectura de fachada (Facade Pattern)?
    El patrón de fachada proporciona una interfaz simplificada para un conjunto complejo de interfaces o subsistemas, facilitando la interacción del cliente con el sistema y ocultando la complejidad interna.

27. ¿Cómo se utiliza el patrón de arquitectura de observador (Observer Pattern)?
    El patrón de observador define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.
28. ¿Qué es el patrón de arquitectura de mediador (Mediator Pattern)?
    El patrón de mediador define un objeto que encapsula cómo interactúan un conjunto de objetos, promoviendo la comunicación entre objetos sin que estén directamente acoplados entre sí.
29. ¿Cómo se aplica el patrón de arquitectura de prototipo (Prototype Pattern)?
    El patrón de prototipo permite copiar objetos existentes en lugar de crear nuevos, proporcionando una manera de crear nuevos objetos basados en una instancia prototipo en lugar de instanciar clases directamente.
30. ¿Qué es el patrón de arquitectura de objeto compuesto (Composite Pattern)?
    El patrón de objeto compuesto permite tratar objetos individuales y composiciones de objetos de manera uniforme, facilitando el trabajo con estructuras jerárquicas y la manipulación de grupos de objetos.

# 2. Ssr

1. ¿Cómo puedes manejar efectos secundarios dependientes del estado con `useEffect`?
   Al incluir el estado relevante en el array de dependencias de `useEffect`, puedes manejar efectos secundarios que dependen de cambios en ese estado.
2. ¿Qué es el `React.memo` y cómo se relaciona con `useCallback`?
   `React.memo` es una función de orden superior que memoriza un componente funcional para evitar renderizaciones innecesarias, y se puede usar con `useCallback` para memorizar funciones pasadas como props.
3. ¿Cómo se usa `useReducer` para manejar el estado complejo en lugar de `useState`?
   `useReducer` permite manejar estados complejos mediante la definición de una función reductora que maneja las transiciones de estado y las acciones.
4. ¿Qué es el hook `useDeferredValue` y cómo puede mejorar el rendimiento?
   `useDeferredValue` permite diferir la actualización de un valor hasta que el navegador esté libre, lo que ayuda a evitar bloqueos en la interfaz de usuario.
5. ¿Cómo puedes utilizar `useEffect` para sincronizar datos con una API?
   Puedes usar `useEffect` para realizar solicitudes a una API y actualizar el estado del componente con los datos recibidos.
6. ¿Qué es el hook `useTransition` y cómo se diferencia de `useDeferredValue`?
   `useTransition` se usa para gestionar transiciones en las actualizaciones de estado, mientras que `useDeferredValue` se usa para diferir la actualización de un valor hasta que el navegador esté menos ocupado.
7. ¿Cómo puedes gestionar la caché de datos en un hook personalizado?
   Puedes gestionar la caché usando `useRef` para almacenar datos entre

renderizaciones y evitar llamadas repetidas a APIs.

8. ¿Qué ventajas tiene el uso de `useCallback` y `useMemo` en un componente de alto rendimiento?
   `useCallback` y `useMemo` ayudan a prevenir renderizaciones innecesarias y cálculos costosos al memorizar funciones y valores.
9. ¿Cómo puedes crear un hook personalizado que maneje el formulario de entrada?
   Puedes crear un hook personalizado que use `useState` para manejar el estado del formulario y `useEffect` para manejar efectos secundarios relacionados con el formulario.
10. ¿Qué es el hook `useContext` y cómo se utiliza para evitar el "prop drilling"?
    `useContext` permite acceder al contexto creado por `React.createContext` sin necesidad de pasar props a través de múltiples niveles de componentes.
11. ¿Cómo puedes usar el hook `useImperativeHandle` para personalizar la instancia del componente?
    `useImperativeHandle` permite exponer sólo ciertas propiedades y métodos a los componentes padres cuando se utiliza con `forwardRef`.
12. ¿Qué consideraciones debes tener al usar `useEffect` con dependencias cambiantes?
    Debes asegurarte de que las dependencias incluidas en el array de dependencias sean estables y no cambien en cada renderización para evitar ejecuciones innecesarias.
13. ¿Cómo puedes combinar múltiples hooks en un hook personalizado?
    Puedes combinar varios hooks dentro de un hook personalizado para encapsular y reutilizar lógica de estado y efectos en diferentes componentes.
14. ¿Cómo se manejan las actualizaciones asíncronas con `useState` y `useEffect`?
    Se deben manejar actualizaciones asíncronas con `useEffect` para asegurarse de que el estado se actualice correctamente después de las operaciones asíncronas.
15. ¿Cómo puedes utilizar `useMemo` para optimizar componentes que renderizan listas grandes?
    `useMemo` puede memorizar el resultado del cálculo de la lista para evitar recálculos innecesarios y mejorar el rendimiento.
16. ¿Cómo se puede gestionar el estado global en un hook personalizado?
    Puedes utilizar `useReducer` o `useContext` dentro de un hook personalizado para manejar el estado global y compartirlo entre múltiples componentes.
17. ¿Qué es el hook `useLayoutEffect` y en qué situaciones es preferible a `useEffect`?
    `useLayoutEffect` se ejecuta inmediatamente después de las actualizaciones del DOM, lo que es útil para lecturas y escrituras que afectan el diseño del DOM.
18. ¿Cómo puedes controlar el comportamiento de los hooks cuando los componentes se actualizan?
    Puedes controlar el comportamiento usando hooks como `useCallback` y `useMemo` para gestionar cómo y cuándo se actualizan las funciones y valores.
19. ¿Qué es el hook `useCallback` y cómo se utiliza para evitar renderizaciones innecesarias?

`useCallback` se usa para memorizar funciones y evitar que se creen nuevas instancias en cada renderización, lo que puede ayudar a prevenir renderizaciones innecesarias de componentes hijos.

20. ¿Cómo puedes depurar hooks personalizados en React?
    Puedes utilizar `useDebugValue` para mostrar valores en las herramientas de depuración de React DevTools y verificar el estado y comportamiento de los hooks personalizados.

# 3. Sr

1. ¿Cómo se implementa una lógica de middleware personalizada en Redux Toolkit?
   Se implementa creando una función que sigue el patrón de middleware y añadiéndola a la configuración de `middleware` en `configureStore`.
2. ¿Qué técnicas avanzadas se pueden aplicar para manejar el estado en una aplicación con múltiples slices interdependientes?
   Utiliza técnicas como la normalización de datos, selectors memoizados, y thunks para manejar la interacción entre múltiples slices y mantener el estado consistente.
3. ¿Cómo se maneja el estado de un componente que necesita sincronización bidireccional con el store?
   Usa `useSelector` para leer del store y `useDispatch` para enviar acciones que actualicen el estado en respuesta a cambios, asegurando la sincronización bidireccional.
4. ¿Cómo puedes optimizar el rendimiento de los selectores en una aplicación con Redux Toolkit?
   Optimiza el rendimiento utilizando `reselect` para crear selectores memoizados que evitan cálculos innecesarios y minimizan los re-renderizados.
5. ¿Qué consideraciones debes tener al manejar datos grandes y complejos en Redux Toolkit?
   Considera la normalización de datos, el uso de `createAsyncThunk` para manejar la carga asíncrona, y el diseño eficiente de slices para gestionar el estado de manera efectiva.
6. ¿Cómo puedes implementar y manejar la persistencia de estado en Redux Toolkit?
   Puedes implementar la persistencia de estado utilizando librerías como `redux-persist` para almacenar el estado en el almacenamiento local o en el backend y restaurarlo al cargar la aplicación.
7. ¿Qué es el patrón de `ducks` y cómo puedes adaptarlo a Redux Toolkit para una mejor organización?
   El patrón de `ducks` organiza actions, reducers y selectors en un solo módulo. En Redux Toolkit, puedes adaptar este patrón creando slices modulares y combinándolos en el store.
8. ¿Cómo puedes integrar Redux Toolkit con TypeScript para una mejor tipificación?
   Integra Redux Toolkit con TypeScript definiendo tipos para el estado, actions y thunks,

y utilizando `createSlice` con tipos genéricos para asegurar la tipificación correcta en la aplicación.

9. ¿Qué estrategias puedes usar para manejar la sincronización de estado en aplicaciones con múltiples ventanas o pestañas?
   Utiliza `localStorage` o `sessionStorage` para compartir el estado entre ventanas/pestañas, y escucha los eventos de almacenamiento para sincronizar los cambios en tiempo real.
10. ¿Cómo puedes implementar una lógica de optimización avanzada para manejar actualizaciones masivas en el estado global?
    Utiliza técnicas como la normalización de datos, actualizaciones por lotes y thunks para manejar actualizaciones masivas de manera eficiente y evitar cuellos de botella en el rendimiento.
11. ¿Cómo puedes usar `createSlice` para manejar estados dependientes y complejos en una aplicación grande?
    Define múltiples reducers en un slice para manejar distintos aspectos del estado dependiente y usa `createAsyncThunk` para gestionar la lógica compleja.
12. ¿Qué es el patrón de diseño `feature slices` y cómo se aplica en Redux Toolkit?
    El patrón `feature slices` organiza el estado y la lógica en slices modulares basados en funcionalidades específicas, aplicándolo en Redux Toolkit mediante `createSlice` para cada funcionalidad.
13. ¿Cómo puedes manejar la lógica de transacciones y la integridad del estado en una aplicación con Redux Toolkit?
    Maneja la lógica de transacciones utilizando thunks para coordinar múltiples acciones y reducers para garantizar la integridad del estado y manejar los errores correctamente.
14. ¿Qué técnicas avanzadas puedes utilizar para la depuración y monitoreo del estado en Redux Toolkit?
    Utiliza herramientas como Redux DevTools, middleware de logging personalizado, y técnicas de registro en el `reducer` para depurar y monitorear el estado.
15. ¿Cómo puedes implementar y gestionar una lógica de actualización en tiempo real en una aplicación con Redux Toolkit?
    Implementa la lógica de actualización en tiempo real utilizando websockets o servicios en tiempo real, y actualiza el estado en el store a través de thunks o reducers.
16. ¿Cómo puedes manejar múltiples instancias del store en una aplicación que requiere diferentes contextos de estado?
    Maneja múltiples instancias creando diferentes configuraciones de store y utilizando contextos separados para cada instancia, asegurando la independencia de los estados.
17. ¿Qué es la `lazy-loading` de slices y cómo puedes aplicarla en Redux Toolkit?
    `Lazy-loading` de slices se refiere a cargar slices solo cuando son necesarios para reducir el tamaño inicial del bundle. Aplica este patrón cargando slices de forma dinámica y añadiéndolos al store.

18. ¿Cómo puedes utilizar `createSlice` y `createAsyncThunk` para manejar casos de uso avanzados como paginación y filtrado?
    Utiliza `createSlice` para manejar el estado de paginación y filtrado y `createAsyncThunk` para manejar las llamadas a la API y las actualizaciones del estado basadas en los resultados.
19. ¿Qué consideraciones debes tener al implementar `redux-toolkit` en aplicaciones con alto tráfico y grandes volúmenes de datos?
    Considera la eficiencia en la actualización del estado, el uso de selectores memoizados, la normalización de datos y la implementación de técnicas de optimización para manejar grandes volúmenes de datos.
20. ¿Cómo puedes estructurar el código para asegurar la escalabilidad y mantenibilidad en aplicaciones grandes utilizando Redux Toolkit?
    Estructura el código dividiendo el estado en slices modulares basados en funcionalidades, utiliza `createSlice` para definir reducers y actions, y organiza los archivos en una estructura clara y mantenible.

# 1. Jr

1. ¿Qué es Redux Toolkit y para qué se usa en React?
   Redux Toolkit es una biblioteca oficial que simplifica el proceso de configuración y uso de Redux para la gestión de estados globales en aplicaciones React.
2. ¿Cómo se crea un store utilizando Redux Toolkit?
   Para crear un store, se utiliza la función `configureStore` de Redux Toolkit, que simplifica la configuración del store y la integración de middlewares.
3. ¿Qué es un slice en Redux Toolkit?
   Un slice es una parte del estado y la lógica relacionada en Redux Toolkit, definida utilizando la función `createSlice`, que incluye el estado inicial y los reducers.
4. ¿Cómo se definen los reducers en Redux Toolkit?
   Los reducers se definen dentro de un slice utilizando la propiedad `reducers` en la configuración del slice.
5. ¿Qué función cumple el hook `useDispatch` en Redux Toolkit?
   El hook `useDispatch` se utiliza para enviar acciones al store desde los componentes funcionales.
6. ¿Cómo se utiliza el hook `useSelector` en Redux Toolkit?
   El hook `useSelector` se usa para seleccionar y acceder al estado del store desde un componente funcional.
7. ¿Qué es un thunk en Redux Toolkit y cómo se usa?
   Un thunk es una función que permite realizar operaciones asíncronas o lógicas complejas antes de despachar una acción, definido utilizando `createAsyncThunk`.
8. ¿Qué es `createAsyncThunk` y para qué sirve?
   `createAsyncThunk` es una función de Redux Toolkit que simplifica la creación de acciones asíncronas y maneja automáticamente el ciclo de vida de las promesas.

9. ¿Cómo puedes manejar errores en `createAsyncThunk`?
   Puedes manejar errores dentro de `createAsyncThunk` utilizando los bloques `try/catch` en la función asíncrona y actualizando el estado en los reducers correspondientes.
10. ¿Qué es el `configureStore` y qué opciones de configuración ofrece?
    `configureStore` es una función que simplifica la creación del store de Redux, ofreciendo opciones para agregar reducers, middleware y configurar devtools.
11. ¿Cómo se integran los middleware en Redux Toolkit?
    Los middleware se integran a través de la opción `middleware` en la función `configureStore`, permitiendo añadir funcionalidades adicionales al flujo de datos.
12. ¿Qué es el estado inmaduro en Redux Toolkit?
    El estado inmaduro es el estado inicial de un slice que se define cuando se crea el slice, antes de que se apliquen las acciones.
13. ¿Cómo se utiliza `createSlice` para definir un slice en Redux Toolkit?
    `createSlice` se utiliza para definir un slice, incluyendo el estado inicial, los reducers y los nombres de las acciones, simplificando la definición del estado y las actualizaciones.
14. ¿Qué es un selector en Redux Toolkit y cómo se crea?
    Un selector es una función que extrae y devuelve una parte específica del estado. Se puede crear como una función normal que recibe el estado y devuelve un valor derivado.
15. ¿Cómo se combina el estado de múltiples slices en Redux Toolkit?
    El estado de múltiples slices se combina automáticamente en el store a través de la configuración de `reducers` en `configureStore`, donde se agregan los reducers de cada slice.
16. ¿Cómo puedes depurar el estado en Redux Toolkit?
    Puedes utilizar las herramientas de desarrollo como Redux DevTools para inspeccionar el estado y las acciones en el store.
17. ¿Qué es el `reducer` en un slice de Redux Toolkit y cómo se define?
    El `reducer` es una función que maneja las actualizaciones del estado en respuesta a las acciones. Se define como una propiedad de `reducers` en `createSlice`.
18. ¿Qué son los `reducers` y cómo se actualizan en Redux Toolkit?
    Los `reducers` son funciones que reciben el estado actual y una acción, y devuelven un nuevo estado actualizado. Se definen en `createSlice` y actualizan el estado de forma inmutable.
19. ¿Cómo se configuran los `devTools` en Redux Toolkit?
    `devTools` se configuran automáticamente en `configureStore` si está disponible en el entorno, pero también se puede personalizar mediante la opción `devTools`.
20. ¿Qué es el `store` en Redux Toolkit y qué papel juega en la aplicación?
    El `store` es el objeto que mantiene el estado de la aplicación, permite el despacho de acciones y proporciona métodos para acceder al estado y suscribirse a cambios.

# 2. Ssr

1. ¿Cómo puedes optimizar el rendimiento utilizando `createAsyncThunk` en Redux Toolkit?
   Optimiza el rendimiento utilizando `createAsyncThunk` para manejar la lógica asíncrona de manera eficiente y evitando re-renderizados innecesarios mediante la selección específica del estado.
2. ¿Qué es el `middleware` en Redux Toolkit y cómo puedes crear uno personalizado?
   El `middleware` es una extensión que permite interceptar y modificar las acciones y el flujo de datos. Puedes crear uno personalizado definiendo una función que se ejecute en cada acción despachada.
3. ¿Cómo se utiliza `createSlice` para manejar el estado complejo con múltiples reducers?
   `createSlice` permite definir múltiples reducers dentro de un mismo slice, organizando y manejando el estado complejo de manera estructurada y modular.
4. ¿Qué son los `reducers` anidados y cómo se manejan en Redux Toolkit?
   Los `reducers` anidados son reducers que manejan partes del estado que están anidadas dentro de otros reducers. Se manejan utilizando el `combineReducers` o directamente en el `reducers` del slice.
5. ¿Cómo se maneja el estado de un slice que depende de múltiples fuentes de datos?
   Se maneja combinando múltiples `reducers` y usando `createAsyncThunk` para manejar las actualizaciones basadas en datos de múltiples fuentes.
6. ¿Cómo puedes usar `reselect` con Redux Toolkit para mejorar la eficiencia de los selectores?
   `reselect` permite crear selectores memoizados que calculan datos derivados de manera eficiente y evitan recalculos innecesarios, mejorando el rendimiento.
7. ¿Qué son los `actions` y cómo se definen en un slice de Redux Toolkit?
   Los `actions` son eventos que describen un cambio en el estado. Se definen automáticamente en `createSlice` y se pueden usar directamente desde el slice.
8. ¿Cómo puedes manejar la sincronización entre diferentes slices utilizando Redux Toolkit?
   La sincronización se maneja mediante el diseño de slices que se actualizan de manera coherente y la combinación de reducers en `configureStore`.
9. ¿Qué consideraciones debes tener al manejar estados asíncronos en un slice?
   Debes considerar cómo manejar los estados de carga, éxito y error, y cómo actualizar el estado de manera consistente utilizando `createAsyncThunk`.
10. ¿Cómo puedes aplicar la normalización de datos en Redux Toolkit?
    Puedes aplicar la normalización de datos transformando datos complejos en una estructura más manejable y almacenando las entidades en un formato plano en el estado.
11. ¿Qué es la `cache` en el contexto de Redux Toolkit y cómo se puede implementar?
    La `cache` es un mecanismo para almacenar datos temporalmente y evitar llamadas

repetidas a APIs. Se puede implementar en los thunks o en los reducers utilizando `useRef` o `createSlice`.

12. ¿Cómo se integran los `reducers` de un slice con `configureStore`?
    Los `reducers` se integran agregándolos al objeto `reducers` en `configureStore`, donde se combinan con otros reducers para formar el estado global.
13. ¿Qué es el `reducer` de actualización parcial y cómo se usa en Redux Toolkit?
    Un `reducer` de actualización parcial maneja actualizaciones específicas de una parte del estado, y se usa en los slices para actualizar sólo las partes relevantes del estado.
14. ¿Cómo puedes depurar acciones y estados en Redux Toolkit?
    Puedes utilizar herramientas como Redux DevTools para inspeccionar el estado y las acciones, y añadir logs o `console.log` en los reducers y thunks para depuración.
15. ¿Qué es el patrón `ducks` en Redux y cómo se aplica en Redux Toolkit?
    El patrón `ducks` organiza los actions, reducers y selectors en un solo archivo para cada módulo, y se aplica en Redux Toolkit mediante la creación de slices modulares.
16. ¿Cómo se maneja el estado global en una aplicación grande utilizando Redux Toolkit?
    Se maneja dividiendo el estado en slices modulares y combinándolos en el store, utilizando `configureStore` para integrar y gestionar el estado global.
17. ¿Qué son los `selectors` y cómo puedes optimizarlos con `createSlice`?
    Los `selectors` son funciones que extraen datos del estado. Puedes optimizarlos utilizando `reselect` para memorizar resultados y evitar cálculos costosos.
18. ¿Cómo puedes combinar `createAsyncThunk` con `createSlice` para manejar efectos secundarios?
    Combina `createAsyncThunk` para manejar acciones asíncronas y `createSlice` para gestionar los estados de carga, éxito y error asociados con esos efectos secundarios.
19. ¿Qué es el `action creator` y cómo se genera con Redux Toolkit?
    El `action creator` es una función que crea una acción con un tipo y payload. En Redux Toolkit, los action creators se generan automáticamente al definir reducers en `createSlice`.
20. ¿Cómo puedes estructurar una aplicación grande para que sea mantenible con Redux Toolkit?
    Estructura la aplicación dividiendo el estado en slices modulares, utilizando `createSlice` para definir la lógica, y gestionando el store con `configureStore` para mantener la mantenibilidad y organización.

# 3. Sr

1. ¿Cómo se implementa una lógica de middleware personalizada en Redux Toolkit?
   Se implementa creando una función que sigue el patrón de middleware y añadiéndola a la configuración de `middleware` en `configureStore`.
2. ¿Qué técnicas avanzadas se pueden aplicar para manejar el estado en una aplicación con múltiples slices interdependientes?

Utiliza técnicas como la normalización de datos, selectors memoizados, y thunks para manejar la interacción entre múltiples slices y mantener el estado consistente.

3. ¿Cómo se maneja el estado de un componente que necesita sincronización bidireccional con el store?
   Usa `useSelector` para leer del store y `useDispatch` para enviar acciones que actualicen el estado en respuesta a cambios, asegurando la sincronización bidireccional.
4. ¿Cómo puedes optimizar el rendimiento de los selectores en una aplicación con Redux Toolkit?
   Optimiza el rendimiento utilizando `reselect` para crear selectores memoizados que evitan cálculos innecesarios y minimizan los re-renderizados.
5. ¿Qué consideraciones debes tener al manejar datos grandes y complejos en Redux Toolkit?
   Considera la normalización de datos, el uso de `createAsyncThunk` para manejar la carga asíncrona, y el diseño eficiente de slices para gestionar el estado de manera efectiva.
6. ¿Cómo puedes implementar y manejar la persistencia de estado en Redux Toolkit?
   Puedes implementar la persistencia de estado utilizando librerías como `redux-persist` para almacenar el estado en el almacenamiento local o en el backend y restaurarlo al cargar la aplicación.
7. ¿Qué es el patrón de `ducks` y cómo puedes adaptarlo a Redux Toolkit para una mejor organización?
   El patrón de `ducks` organiza actions, reducers y selectors en un solo módulo. En Redux Toolkit, puedes adaptar este patrón creando slices modulares y combinándolos en el store.
8. ¿Cómo puedes integrar Redux Toolkit con TypeScript para una mejor tipificación?
   Integra Redux Toolkit con TypeScript definiendo tipos para el estado, actions y thunks, y utilizando `createSlice` con tipos genéricos para asegurar la tipificación correcta en la aplicación.
9. ¿Qué estrategias puedes usar para manejar la sincronización de estado en aplicaciones con múltiples ventanas o pestañas?
   Utiliza `localStorage` o `sessionStorage` para compartir el estado entre ventanas/pestañas, y escucha los eventos de almacenamiento para sincronizar los cambios en tiempo real.
10. ¿Cómo puedes implementar una lógica de optimización avanzada para manejar actualizaciones masivas en el estado global?
    Utiliza técnicas como la normalización de datos, actualizaciones por lotes y thunks para manejar actualizaciones masivas de manera eficiente y evitar cuellos de botella en el rendimiento.
11. ¿Cómo puedes usar `createSlice` para manejar estados dependientes y complejos en una aplicación grande?

Define múltiples reducers en un slice para manejar distintos aspectos del estado dependiente y usa `createAsyncThunk` para gestionar la lógica compleja.

12. ¿Qué es el patrón de diseño `feature slices` y cómo se aplica en Redux Toolkit?
    El patrón `feature slices` organiza el estado y la lógica en slices modulares basados en funcionalidades específicas, aplicándolo en Redux Toolkit mediante `createSlice` para cada funcionalidad.
13. ¿Cómo puedes manejar la lógica de transacciones y la integridad del estado en una aplicación con Redux Toolkit?
    Maneja la lógica de transacciones utilizando thunks para coordinar múltiples acciones y reducers para garantizar la integridad del estado y manejar los errores correctamente.
14. ¿Qué técnicas avanzadas puedes utilizar para la depuración y monitoreo del estado en Redux Toolkit?
    Utiliza herramientas como Redux DevTools, middleware de logging personalizado, y técnicas de registro en el `reducer` para depurar y monitorear el estado.
15. ¿Cómo puedes implementar y gestionar una lógica de actualización en tiempo real en una aplicación con Redux Toolkit?
    Implementa la lógica de actualización en tiempo real utilizando websockets o servicios en tiempo real, y actualiza el estado en el store a través de thunks o reducers.
16. ¿Cómo puedes manejar múltiples instancias del store en una aplicación que requiere diferentes contextos de estado?
    Maneja múltiples instancias creando diferentes configuraciones de store y utilizando contextos separados para cada instancia, asegurando la independencia de los estados.
17. ¿Qué es la `lazy-loading` de slices y cómo puedes aplicarla en Redux Toolkit?
    `Lazy-loading` de slices se refiere a cargar slices solo cuando son necesarios para reducir el tamaño inicial del bundle. Aplica este patrón cargando slices de forma dinámica y añadiéndolos al store.
18. ¿Cómo puedes utilizar `createSlice` y `createAsyncThunk` para manejar casos de uso avanzados como paginación y filtrado?
    Utiliza `createSlice` para manejar el estado de paginación y filtrado y `createAsyncThunk` para manejar las llamadas a la API y las actualizaciones del estado basadas en los resultados.
19. ¿Qué consideraciones debes tener al implementar `redux-toolkit` en aplicaciones con alto tráfico y grandes volúmenes de datos?
    Considera la eficiencia en la actualización del estado, el uso de selectores memoizados, la normalización de datos y la implementación de técnicas de optimización para manejar grandes volúmenes de datos.
20. ¿Cómo puedes estructurar el código para asegurar la escalabilidad y mantenibilidad en aplicaciones grandes utilizando Redux Toolkit?
    Estructura el código dividiendo el estado en slices modulares basados en funcionalidades, utiliza `createSlice` para definir reducers y actions, y organiza los archivos en una estructura clara y mantenible.

# 1. Jr

1. ¿Qué es Zustand y para qué se utiliza en React?
   Zustand es una biblioteca para la gestión de estado en React que proporciona una solución simple y flexible para manejar el estado global.
2. ¿Cómo se crea un store básico utilizando Zustand?
   Se crea utilizando la función `create` de Zustand, pasando una función que define el estado inicial y los métodos para actualizarlo.
3. ¿Cómo se accede al estado en un componente utilizando Zustand?
   Se accede al estado mediante el hook creado por Zustand, que devuelve el estado y los métodos definidos en el store.
4. ¿Qué son los `actions` en Zustand y cómo se definen?
   Los `actions` son métodos que permiten modificar el estado. Se definen como funciones dentro del objeto de configuración del store.
5. ¿Cómo puedes actualizar el estado en un store de Zustand?
   Puedes actualizar el estado llamando a los métodos definidos en el store o directamente modificando el estado en las funciones del store.
6. ¿Qué es el `middleware` en Zustand y cómo se utiliza?
   El `middleware` en Zustand permite extender la funcionalidad del store, como la persistencia del estado. Se utiliza envolviendo el store con el middleware correspondiente.
7. ¿Cómo se maneja la persistencia del estado en Zustand?
   La persistencia del estado se maneja utilizando el middleware `persist`, que guarda el estado en el almacenamiento local y lo recupera al cargar la aplicación.
8. ¿Cómo puedes crear un store con Zustand que gestione múltiples estados?
   Puedes definir múltiples estados dentro del objeto de configuración del store y utilizar métodos separados para actualizar cada parte del estado.
9. ¿Qué es `subscribe` en Zustand y cómo se usa?
   `subscribe` es una función que permite suscribirse a los cambios en el estado del store y ejecutar una función de callback cuando el estado cambia.
10. ¿Cómo se utiliza el hook `useStore` para acceder al estado en Zustand?
    `useStore` es el hook creado por Zustand que se utiliza en los componentes para acceder al estado y métodos definidos en el store.
11. ¿Cómo se puede combinar Zustand con otros hooks en React?
    Puedes combinar Zustand con otros hooks de React, como `useEffect`, para manejar efectos secundarios basados en cambios en el estado del store.
12. ¿Qué es el `shallow` en Zustand y cuándo se utiliza?
    `shallow` es una función que permite una comparación superficial del estado para evitar renderizados innecesarios en componentes que dependen del estado del store.
13. ¿Cómo puedes utilizar Zustand con TypeScript para tipar el estado y las acciones?
    Puedes utilizar TypeScript definiendo tipos para el estado y las acciones y aplicando esos tipos al crear el store con Zustand.

14. ¿Qué diferencias existen entre Zustand y Redux en términos de gestión de estado?
    Zustand es más sencillo y tiene una API más pequeña en comparación con Redux, que requiere configuraciones adicionales y más boilerplate para manejar el estado.
15. ¿Cómo puedes manejar estados derivados en Zustand?
    Puedes manejar estados derivados calculando valores basados en el estado actual dentro del store y exponiéndolos a través de métodos o propiedades del store.
16. ¿Cómo se gestionan los errores en el estado de un store de Zustand?
    Se gestionan añadiendo propiedades o métodos en el store para manejar y actualizar el estado de error, y utilizando lógica condicional para mostrar mensajes de error en los componentes.
17. ¿Qué es `immer` en Zustand y cómo ayuda a manejar el estado?
    `immer` es una biblioteca que permite trabajar con el estado de manera inmutable, simplificando la actualización del estado en Zustand al utilizar un enfoque basado en mutaciones.
18. ¿Cómo se integra Zustand con React DevTools para depuración?
    Zustand se integra con React DevTools proporcionando una extensión que permite inspeccionar el estado y las acciones en el store durante el desarrollo.
19. ¿Qué es el `selector` en Zustand y cómo se usa?
    El `selector` es una función que permite seleccionar y derivar partes específicas del estado del store, optimizando el acceso a los datos necesarios en los componentes.
20. ¿Cómo puedes manejar el estado en una aplicación grande utilizando Zustand?
    Maneja el estado en una aplicación grande dividiendo el estado en múltiples stores y combinándolos según sea necesario, y manteniendo una estructura modular y clara.

# 2. Ssr

1. ¿Cómo se optimiza el rendimiento en Zustand con estado grande y complejo?
   Optimiza el rendimiento utilizando `shallow` para comparaciones superficiales y selectors para seleccionar solo las partes necesarias del estado.
2. ¿Qué es el `persist` middleware en Zustand y cómo lo configuras?
   El middleware `persist` permite guardar el estado en almacenamiento local y recuperarlo al cargar la aplicación. Se configura envolviendo el store con el middleware `persist`.
3. ¿Cómo puedes manejar la sincronización del estado entre múltiples instancias del store?
   Maneja la sincronización utilizando técnicas de almacenamiento compartido, como `localStorage`, y escuchando eventos de cambio para actualizar las instancias del store.
4. ¿Cómo puedes implementar una lógica de transacciones utilizando Zustand?
   Implementa la lógica de transacciones mediante el uso de acciones que agrupan varias actualizaciones del estado en una sola operación, asegurando que el estado se actualice de manera coherente.

5. ¿Qué es `subscribeWithSelector` en Zustand y cómo mejora la eficiencia?
   `subscribeWithSelector` permite suscribirse a partes específicas del estado utilizando selectors, mejorando la eficiencia al evitar renderizados innecesarios en componentes.
6. ¿Cómo se maneja el estado asíncrono en Zustand utilizando efectos secundarios?
   Maneja el estado asíncrono utilizando efectos secundarios con `useEffect` para realizar operaciones asíncronas y actualizar el estado del store en función de los resultados.
7. ¿Cómo puedes implementar la lógica de actualización masiva en Zustand?
   Implementa la lógica de actualización masiva utilizando acciones que actualicen múltiples partes del estado de manera eficiente y agrupada.
8. ¿Cómo puedes combinar Zustand con otras bibliotecas de gestión de estado en una aplicación?
   Combina Zustand con otras bibliotecas utilizando múltiples stores para manejar diferentes partes del estado y sincronizar los cambios según sea necesario.
9. ¿Qué son los `middlewares` personalizados en Zustand y cómo se crean?
   Los `middlewares` personalizados permiten extender la funcionalidad del store. Se crean definiendo una función que modifique o amplíe el comportamiento del store y aplicándola en la configuración.
10. ¿Cómo puedes usar Zustand para manejar la lógica de formularios complejos?
    Maneja la lógica de formularios complejos utilizando Zustand para almacenar el estado del formulario y definir acciones para actualizar los campos y manejar la validación.
11. ¿Cómo se maneja el estado global en una aplicación con múltiples módulos utilizando Zustand?
    Maneja el estado global dividiendo el estado en múltiples stores modulares y combinándolos según sea necesario, manteniendo una estructura organizada.
12. ¿Qué técnicas avanzadas puedes aplicar para depurar el estado en Zustand?
    Utiliza técnicas como logging avanzado, `React DevTools`, y técnicas de prueba para depurar el estado y las acciones en Zustand.
13. ¿Cómo puedes implementar `lazy-loading` de partes del estado en Zustand?
    Implementa `lazy-loading` cargando el estado de manera dinámica cuando sea necesario y utilizando técnicas de carga diferida para reducir el tamaño inicial del bundle.
14. ¿Cómo se maneja la lógica de autorización y autenticación en Zustand?
    Maneja la lógica de autorización y autenticación almacenando el estado relacionado en el store y utilizando acciones para actualizar el estado de autenticación y permisos.
15. ¿Qué es el patrón de `actions creators` en Zustand y cómo se aplica?
    El patrón `actions creators` organiza y encapsula la lógica de actualización del estado en funciones, facilitando la reutilización y mantenibilidad del código.
16. ¿Cómo puedes gestionar el estado compartido entre múltiples componentes utilizando Zustand?
    Gestiona el estado compartido utilizando un store centralizado y accediendo al estado y métodos desde los componentes según sea necesario.

17. ¿Qué consideraciones debes tener al integrar Zustand con aplicaciones de gran escala?
    Considera la modularidad del estado, el rendimiento, la sincronización de datos y la estructura del store para asegurar la escalabilidad y mantenibilidad en aplicaciones grandes.
18. ¿Cómo puedes usar Zustand con librerías de enrutamiento como React Router?
    Integra Zustand con librerías de enrutamiento utilizando hooks para acceder y actualizar el estado en función de los cambios en la URL y la navegación
19. ¿Cómo puedes aplicar técnicas de memoización en Zustand para optimizar el rendimiento?
    Aplica técnicas de memoización utilizando selectors y comparaciones superficiales para evitar cálculos y renderizados innecesarios en componentes.
20. ¿Qué son los `drafts` en Zustand y cómo se utilizan para manejar actualizaciones inmutables?
    `Drafts` son una técnica de `immer` que permite trabajar co n el estado de manera mutable dentro de un `draft`, facilitando actualizaciones inmutables y evitando errores.

# 3. Sr

1. ¿Cómo puedes implementar una lógica de manejo de errores global en Zustand?
   Implementa una lógica de manejo de errores global almacenando el estado de error en el store y utilizando acciones para actualizar y manejar los errores en toda la aplicación.
2. ¿Qué técnicas avanzadas puedes usar para sincronizar el estado entre el cliente y el servidor con Zustand?
   Utiliza técnicas como sincronización en tiempo real con websockets, almacenamiento persistente y sincronización de estados mediante llamadas API para mantener la coherencia entre cliente y servidor.
3. ¿Cómo puedes gestionar múltiples stores en una aplicación grande utilizando Zustand?
   Puedes gestionar múltiples stores dividiendo el estado en stores modulares para cada funcionalidad y combinándolos según sea necesario en los componentes.
4. ¿Qué consideraciones de rendimiento debes tener en cuenta al manejar grandes volúmenes de datos con Zustand?
   Considera el uso de memoización con selectors, el middleware `shallow` para evitar renderizados innecesarios y la normalización de datos para mejorar la eficiencia del estado.
5. ¿Cómo se pueden manejar estados dependientes entre múltiples stores en Zustand?
   Maneja estados dependientes creando acciones que interactúan entre múltiples stores o utilizando un patrón centralizado para coordinar los cambios entre ellos.
6. ¿Qué es `subscribeWithSelector` y cómo mejora la eficiencia en aplicaciones de gran escala con Zustand?

`SubscribeWithSelector` permite suscribirse a cambios específicos en el estado, mejorando la eficiencia al evitar renderizados y cálculos innecesarios en componentes.

7. ¿Cómo puedes manejar la rehidratación de estado en aplicaciones que utilizan almacenamiento persistente con Zustand?
   Maneja la rehidratación utilizando el middleware `persist` para restaurar el estado desde el almacenamiento local o de servidor al iniciar la aplicación.
8. ¿Qué son las `sagas` y cómo puedes replicar su comportamiento en Zustand?
   Las `sagas` son funciones que manejan efectos secundarios complejos en Redux. En Zustand, puedes replicar su comportamiento utilizando `subscribe` o middleware personalizados para manejar flujos de trabajo asíncronos.
9. ¿Cómo puedes optimizar el rendimiento de los selectores en una aplicación con Zustand?
   Optimiza el rendimiento utilizando memoización en los selectores y asegurándote de que los componentes se rendericen solo cuando las partes relevantes del estado cambien.
10. ¿Qué técnicas puedes aplicar para manejar la concurrencia y las actualizaciones simultáneas en el estado de Zustand?
    Utiliza técnicas como `optimistic updates` y control de versiones en el estado para manejar actualizaciones concurrentes, asegurando la consistencia en la aplicación.
11. ¿Cómo puedes implementar un patrón de `event sourcing` en Zustand?
    Implementa `event sourcing` almacenando una lista de eventos que modifican el estado y reproduciendo esos eventos para regenerar el estado actual cuando sea necesario.
12. ¿Cómo puedes estructurar el estado en Zustand para asegurar la escalabilidad en aplicaciones empresariales?
    Estructura el estado dividiendo la lógica en stores modulares, utilizando el patrón de `feature slices`, y organizando el código para facilitar el crecimiento y la mantenibilidad.
13. ¿Qué es la `code splitting` de stores en Zustand y cómo se puede aplicar?
    La `code splitting` de stores implica cargar solo las partes del estado necesarias para mejorar el rendimiento. Se puede aplicar cargando stores de forma dinámica cuando se necesiten.
14. ¿Cómo puedes utilizar Zustand para gestionar estados complejos y derivados en aplicaciones de alto rendimiento?
    Utiliza selectors memoizados, middlewares personalizados y optimizaciones como `shallow` para gestionar estados complejos y derivados sin comprometer el rendimiento.
15. ¿Cómo puedes manejar la lógica de paginación y filtrado en una aplicación con Zustand?
    Maneja la paginación y el filtrado almacenando los criterios en el store y utilizando acciones para actualizar los resultados según los filtros y la página actual.

16. ¿Cómo se puede integrar Zustand con librerías de autenticación y autorización avanzadas?
    Integra librerías de autenticación y autorización utilizando el store para almacenar el estado de autenticación, tokens y permisos, y actualizando el estado según los flujos de autenticación.
17. ¿Qué técnicas puedes usar para implementar la sincronización de estado en aplicaciones multiusuario en tiempo real con Zustand?
    Implementa la sincronización de estado en tiempo real utilizando websockets o servicios como Firebase y actualizando el store en función de los eventos recibidos.
18. ¿Cómo puedes asegurar la consistencia del estado en una aplicación distribuida con Zustand?
    Asegura la consistencia del estado utilizando mecanismos de sincronización como bases de datos en tiempo real, APIs REST con polling y manejo de eventos para actualizar el store.
19. ¿Cómo puedes manejar estados inmutables complejos utilizando Zustand con `immer`?
    Maneja estados inmutables complejos utilizando `immer` dentro del store para realizar actualizaciones inmutables de manera más simple y eficiente, evitando la mutación directa del estado.
20. ¿Qué técnicas avanzadas puedes implementar para pruebas unitarias en aplicaciones que utilizan Zustand?
    Utiliza técnicas de simulación (mocking) de acciones y stores, junto con pruebas unitarias para los reducers y el comportamiento del estado, asegurando que la lógica del store funcione correctamente.

# Técnicas

1. ¿Qué es React y cuáles son sus principales características?
   React es una biblioteca de JavaScript para construir interfaces de usuario. Sus principales características incluyen la creación de componentes reutilizables, el uso de un Virtual DOM para mejorar el rendimiento y la unidireccionalidad del flujo de datos.
2. ¿Qué es el Virtual DOM en React y cómo mejora el rendimiento?
   El Virtual DOM es una representación en memoria del DOM real que React utiliza para hacer actualizaciones eficientes. Al comparar el Virtual DOM con el DOM real mediante un proceso llamado reconciliación, React minimiza las actualizaciones directas del DOM y mejora el rendimiento.
3. ¿Cuál es la diferencia entre un componente de clase y un componente funcional en React?
   Los componentes de clase en React se definen utilizando la clase `React.Component` y pueden tener un estado interno y métodos de ciclo de vida. Los componentes funcionales son funciones que reciben props y devuelven JSX, y con la introducción de Hooks, también pueden manejar el estado y efectos secundarios.
4. ¿Qué son los Hooks en React y por qué son importantes?
   Los Hooks son funciones que permiten usar el estado y otros aspectos del ciclo de

vida en componentes funcionales. Son importantes porque permiten manejar el estado y los efectos secundarios en componentes funcionales, lo que facilita la reutilización de lógica y la creación de componentes más simples.

5. ¿Qué es `useState` en React y cómo se utiliza?
   `useState` es un Hook que permite agregar estado a componentes funcionales. Se utiliza llamando a `useState` dentro de un componente para definir una variable de estado y una función para actualizarla.
6. ¿Cómo funciona el Hook `useEffect` y cuándo se debe usar?
   `useEffect` es un Hook que permite realizar efectos secundarios en componentes funcionales. Se debe usar para manejar operaciones como la suscripción a eventos, la recuperación de datos o la manipulación del DOM. Se ejecuta después de que el componente se monta y se actualiza.
7. ¿Qué son los efectos secundarios en React y cómo se gestionan con `useEffect`?
   Los efectos secundarios son operaciones que afectan el sistema fuera del componente, como llamadas a APIs o manipulación del DOM. Se gestionan con `useEffect`, que permite ejecutar código después de que el componente se renderiza o se actualiza.
8. ¿Cómo se pueden manejar múltiples efectos secundarios en un solo componente?
   Se pueden manejar múltiples efectos secundarios utilizando varios Hooks `useEffect` dentro de un mismo componente, cada uno con su propia lógica y dependencias.
9. ¿Qué es el Hook `useContext` y cómo se utiliza?
   `useContext` es un Hook que permite acceder al contexto de React en componentes funcionales. Se utiliza para consumir el valor del contexto creado por `React.createContext` y evitar el paso de props a través de múltiples niveles de componentes.
10. ¿Qué es el contexto en React y para qué se utiliza?
    El contexto en React es una forma de pasar datos a través de la estructura de componentes sin tener que pasar props manualmente en cada nivel. Se utiliza para compartir valores como temas, datos del usuario o configuraciones globales.
11. ¿Cómo se implementa un proveedor de contexto en React?
    Un proveedor de contexto se implementa utilizando `Context.Provider` y envolviendo los componentes que necesitan acceder al contexto. El valor del contexto se pasa como una prop al `Provider`.
12. ¿Qué es `useReducer` y cuándo se debería usar en lugar de `useState`?
    `useReducer` es un Hook que maneja el estado mediante un reductor, similar a la forma en que Redux maneja el estado global. Se debería usar en lugar de `useState` cuando el estado es complejo y requiere lógica de actualización más elaborada.
13. ¿Cómo se pueden optimizar los componentes en React para mejorar el rendimiento?
    Se pueden optimizar los componentes utilizando técnicas como `React.memo` para evitar renderizados innecesarios, `useCallback` para memorizar funciones y `useMemo` para memorizar valores calculados.

14. ¿Qué es `React.memo` y cómo se utiliza?
    `React.memo` es una función que memoiza un componente funcional, evitando su renderizado si las props no han cambiado. Se utiliza para mejorar el rendimiento de componentes que reciben props inmutables.
15. ¿Cómo se usa `useCallback` y qué problema resuelve?
    `useCallback` es un Hook que memoiza una función para que no se recree en cada renderizado. Resuelve el problema de que las funciones se redefinan y provoquen renderizados innecesarios en los componentes hijos.
16. ¿Qué es `useMemo` y cuándo deberías usarlo?
    `useMemo` es un Hook que memoiza el resultado de una función para evitar cálculos innecesarios en cada renderizado. Deberías usarlo cuando tienes cálculos costosos que no deben ejecutarse a menos que sus dependencias cambien.
17. ¿Qué es `React.lazy` y cómo se utiliza para la carga diferida de componentes?
    `React.lazy` es una función que permite la carga diferida de componentes en React. Se utiliza para dividir el código en partes más pequeñas y cargar componentes solo cuando son necesarios, mejorando el rendimiento de la aplicación.
18. ¿Cómo funciona el mecanismo de `suspense` en React y cómo se integra con `React.lazy`?
    `Suspense` es un componente que permite manejar la carga de componentes de manera asincrónica, mostrando una interfaz de carga mientras el componente se está cargando. Se integra con `React.lazy` para mostrar una interfaz de usuario mientras se carga el componente diferido.
19. ¿Qué es un `Error Boundary` en React y cómo se implementa?
    Un `Error Boundary` es un componente que captura errores en el árbol de componentes y muestra una interfaz de usuario alternativa. Se implementa creando un componente de clase con el método `componentDidCatch` y `static getDerivedStateFromError`.
20. ¿Cómo se pueden manejar las actualizaciones de estado basadas en valores previos en React?
    Se pueden manejar utilizando la función de actualización del estado proporcionada por `useState` o en los reducers de `useReducer`, que permite acceder al estado anterior para calcular el nuevo estado.
21. ¿Qué son los `Refs` en React y cómo se utilizan?
    Los `Refs` permiten acceder directamente a un nodo del DOM o a una instancia de un componente. Se utilizan con `React.createRef` o `useRef` para manipular elementos del DOM directamente o para almacenar valores mutables.
22. ¿Cómo se puede usar `forwardRef` para pasar `Refs` a componentes funcionales?
    `forwardRef` es una función que permite pasar `Refs` a componentes funcionales. Se utiliza envolviendo el componente funcional y pasando el `Ref` como un argumento adicional.
23. ¿Qué es `React.StrictMode` y cómo ayuda en el desarrollo?
    `React.StrictMode` es un componente que activa comprobaciones adicionales y

advertencias en el desarrollo para detectar problemas potenciales en la aplicación, como el uso de APIs obsoletas y efectos secundarios no deseados.

24. ¿Cómo se puede implementar un patrón de diseño `Higher-Order Component` (HOC) en React?
    Un HOC es una función que toma un componente y devuelve un nuevo componente con funcionalidades adicionales. Se implementa creando una función que recibe el componente original y devuelve un nuevo componente con las mejoras deseadas.
25. ¿Qué es un `render prop` y cómo se utiliza en React?
    Un `render prop` es una técnica que permite a un componente pasar una función como prop para controlar el contenido que se renderiza. Se utiliza para compartir lógica entre componentes y proporcionar una interfaz flexible para el renderizado.
26. ¿Cómo se pueden gestionar las rutas en una aplicación React?
    Las rutas en una aplicación React se gestionan utilizando librerías como `react-router-dom`, que proporciona componentes y hooks para definir y manejar rutas, y para navegar entre diferentes vistas o páginas.
27. ¿Qué es el `context API` y cómo se diferencia de la gestión de estado con Redux?
    El `context API` es una herramienta de React para pasar datos a través del árbol de componentes sin necesidad de props. Se diferencia de Redux en que está diseñado para el manejo de estado a nivel de contexto y no para el manejo global de estado.
28. ¿Cómo se pueden realizar pruebas unitarias en componentes React?
    Las pruebas unitarias en componentes React se pueden realizar utilizando librerías como `Jest` para pruebas y `React Testing Library` para renderizar componentes y verificar su comportamiento y apariencia.
29. ¿Qué es `React Router` y cómo se configura para manejar rutas en una aplicación React?
    `React Router` es una librería para manejar el enrutamiento en aplicaciones React. Se configura importando los componentes `BrowserRouter`, `Route` y `Switch` para definir y manejar rutas y la navegación entre páginas.
30. ¿Cómo se puede implementar la paginación en una aplicación React?
    La paginación se puede implementar utilizando componentes para manejar la lógica de paginación y el estado de la página actual. También se pueden usar bibliotecas de terceros que proporcionan componentes y funcionalidades para la paginación.
31. ¿Qué son los `portals` en React y cómo se utilizan?
    Los `portals` permiten renderizar un componente en un nodo del DOM que está fuera del árbol DOM del componente padre. Se utilizan para crear modales, tooltips o elementos que necesitan estar en un nivel diferente del DOM.
32. ¿Cómo se pueden manejar los eventos en React?
    Los eventos en React se manejan utilizando props que se pasan a los componentes, como `onClick`, `onChange`, y `onSubmit`. Los manejadores de eventos se definen como funciones que se ejecutan en respuesta a las acciones del usuario.
33. ¿Qué es `React.Fragment` y para qué se utiliza?
    `React.Fragment` es un componente que permite agrupar una lista de hijos sin añadir

nodos extra al DOM. Se utiliza para evitar la creación de elementos adicionales en el árbol del DOM mientras se agrupan varios elementos.

34. ¿Cómo se pueden evitar renderizados innecesarios en React?
    Se pueden evitar renderizados innecesarios utilizando `React.memo` para memoizar componentes, `useCallback` para evitar la recreación de funciones y `useMemo` para memoizar valores calculados.
35. ¿Qué son los `Custom Hooks` en React y cómo se crean?
    Los `Custom Hooks` son funciones que encapsulan lógica de estado y efectos secundarios para ser reutilizados en diferentes componentes. Se crean como funciones que utilizan otros Hooks y devuelven el estado o la lógica que se quiere compartir.
36. ¿Cómo se puede manejar el estado global en una aplicación React?
    El estado global se puede manejar utilizando el `context API`, `Redux`, o librerías similares como `Zustand` o `Recoil`. Estas herramientas permiten compartir el estado entre diferentes componentes sin tener que pasar props manualmente.
37. ¿Qué es `React.StrictMode` y cómo puede afectar al rendimiento?
    `React.StrictMode` activa comprobaciones y advertencias adicionales en el desarrollo para detectar problemas potenciales. Aunque no afecta al rendimiento en producción, puede ayudar a identificar problemas que podrían afectar el rendimiento.
38. ¿Cómo se utilizan los `Hooks` para manejar el ciclo de vida en componentes funcionales?
    Los `Hooks` como `useEffect` y `useLayoutEffect` permiten manejar el ciclo de vida en componentes funcionales, reemplazando métodos de ciclo de vida de componentes de clase como `componentDidMount` y `componentDidUpdate`.
39. ¿Qué es `useImperativeHandle` y cómo se usa?
    `useImperativeHandle` es un Hook que permite personalizar la instancia del `ref` que se expone a los componentes padres. Se utiliza para controlar los métodos y propiedades expuestos por un componente con `ref`.
40. ¿Cómo se pueden manejar errores en componentes funcionales?
    Los errores en componentes funcionales se pueden manejar utilizando `Error Boundaries` en componentes de clase que envuelven los componentes funcionales. También se pueden capturar errores en el código de efectos con `try-catch` dentro de `useEffect`.
41. ¿Qué es `useTransition` y cómo mejora la experiencia del usuario en React?
    `useTransition` es un Hook que permite gestionar transiciones de estado de manera asíncrona, proporcionando una forma de marcar ciertas actualizaciones de estado como no urgentes y mejorar la experiencia del usuario al mantener la interfaz reactiva.
42. ¿Cómo se implementa la autenticación en una aplicación React?
    La autenticación en una aplicación React se puede implementar utilizando `context API` para manejar el estado de autenticación y `react-router-dom` para proteger rutas privadas, redirigiendo a los usuarios no autenticados a la página de inicio de sesión.

43. ¿Qué es `useDeferredValue` y cómo se utiliza?
    `useDeferredValue` es un Hook que permite diferir la actualización de valores no críticos, mejorando la respuesta de la interfaz de usuario en situaciones donde los datos pueden cambiar rápidamente.
44. ¿Cómo se puede optimizar el rendimiento de una lista larga en React?
    Se puede optimizar el rendimiento utilizando técnicas como `windowing` con bibliotecas como `react-window` o `react-virtualized`, que renderizan solo los elementos visibles y mantienen un bajo consumo de memoria.
45. ¿Qué es `useLayoutEffect` y en qué se diferencia de `useEffect`?
    `useLayoutEffect` es similar a `useEffect`, pero se ejecuta sincrónicamente después de que todas las mutaciones del DOM se hayan aplicado, lo que permite medir el DOM y realizar cambios antes de que el navegador pinte la pantalla.
46. ¿Cómo se puede gestionar la internacionalización en una aplicación React?
    La internacionalización se puede gestionar utilizando bibliotecas como `react-i18next` que proporcionan funcionalidades para traducir textos y adaptar la interfaz a diferentes idiomas y regiones.
47. ¿Qué es un `Controlled Component` en React?
    Un `Controlled Component` es un componente de formulario cuyo valor es controlado por el estado de React. Los valores del formulario se gestionan a través del estado del componente y se actualizan mediante eventos.
48. ¿Cómo se puede implementar la búsqueda en tiempo real en una aplicación React?
    La búsqueda en tiempo real se puede implementar utilizando un estado que se actualiza a medida que el usuario escribe y filtrando los resultados en función del texto de búsqueda. También se pueden usar técnicas de debounce para optimizar las solicitudes.
49. ¿Qué es `useSyncExternalStore` y para qué se utiliza?
    `useSyncExternalStore` es un Hook que permite suscribirse a una tienda externa de manera sincrónica, garantizando que el componente se actualice cuando la tienda cambie, y es útil para integraciones con sistemas de estado externos.
50. ¿Cómo se pueden manejar formularios complejos en React?
    Los formularios complejos se pueden manejar utilizando librerías como `react-hook-form` o `formik`, que proporcionan funcionalidades para la validación, manejo de estado y gestión de eventos de formularios complejos.

# Testing Back

## Jr

1. ¿Qué es Jest y cómo se utiliza para probar aplicaciones backend?
   Jest es un framework de pruebas de JavaScript que se utiliza para escribir y ejecutar pruebas unitarias e integradas en aplicaciones backend. Se configura en el archivo `jest.config.js` y se usa para verificar que el código funcione como se espera.

2. ¿Cómo se escribe una prueba básica con Jest para una función en el backend?
   Se escribe utilizando `test()` o `it()` para definir la prueba, y `expect()` para hacer afirmaciones sobre los resultados de la función que se está probando.
3. ¿Qué es `jest.mock()` y cómo se utiliza para simular módulos en pruebas backend?
   `jest.mock()` se usa para reemplazar módulos con versiones simuladas, permitiendo controlar el comportamiento de dependencias durante las pruebas, como bases de datos o servicios externos.
4. ¿Cómo se configura Jest para trabajar con bases de datos en pruebas backend?
   Se configura utilizando una base de datos en memoria o un entorno de pruebas separado, y se inicializa y limpia la base de datos en `beforeEach` y `afterEach` para asegurar pruebas consistentes.
5. ¿Qué es un "test suite" en Jest y cómo se organiza para pruebas de backend?
   Un "test suite" es un conjunto de pruebas que se agrupan bajo una misma descripción utilizando `describe()`. Se organiza para agrupar pruebas relacionadas, como las pruebas de diferentes funciones de un módulo.
6. ¿Cómo se utiliza `beforeEach()` y `afterEach()` para preparar y limpiar el entorno de pruebas en Jest?
   `beforeEach()` se utiliza para ejecutar código antes de cada prueba, como configurar datos de prueba, y `afterEach()` para limpiar el entorno, como borrar datos o restablecer estados.
7. ¿Cómo se escriben pruebas para endpoints de una API REST utilizando Jest y `supertest`?
   Se escriben utilizando `supertest` para hacer solicitudes HTTP a los endpoints y verificar las respuestas utilizando Jest para hacer afirmaciones sobre el estado de la respuesta y el contenido.
8. ¿Qué es `jest.spyOn()` y cómo se utiliza para espiar funciones en pruebas backend?
   `jest.spyOn()` se usa para espiar funciones, permitiendo interceptar llamadas a funciones y verificar cómo y con qué argumentos se llaman durante las pruebas.
9. ¿Cómo se manejan las pruebas de errores y excepciones en Jest para aplicaciones backend?
   Se manejan utilizando `expect().toThrow()` para verificar que una función lanza una excepción esperada y asegurarse de que el manejo de errores funciona correctamente.
10. ¿Cómo se prueban las funciones asíncronas en Jest?
    Se prueban utilizando `async/await` en combinación con `expect()` para manejar promesas y verificar que se resuelven o rechazan correctamente.
11. ¿Qué son los "fixtures" en el contexto de pruebas y cómo se usan en Jest para el backend?
    Los "fixtures" son datos predefinidos que se utilizan para configurar el entorno de prueba. Se utilizan para proporcionar datos consistentes durante las pruebas y asegurar que los resultados sean reproducibles.

12. ¿Cómo se realizan pruebas de integración para servicios backend utilizando Jest?
    Se realizan escribiendo pruebas que interactúan con múltiples componentes del sistema, como bases de datos y servicios externos, para verificar que funcionen juntos correctamente.
13. ¿Qué es `jest.fn()` y cómo se usa para crear funciones simuladas en pruebas?
    `jest.fn()` se utiliza para crear funciones simuladas que permiten rastrear llamadas y resultados, y para proporcionar implementaciones personalizadas durante las pruebas.
14. ¿Cómo se configuran las pruebas para una aplicación que utiliza servicios externos en Jest?
    Se configuran utilizando mocks para simular las respuestas de los servicios externos, permitiendo verificar el comportamiento de la aplicación sin realizar llamadas reales.
15. ¿Cómo se implementan pruebas de rendimiento básico en el backend utilizando Jest?
    Se implementan midiendo el tiempo de ejecución de funciones o bloques de código utilizando herramientas de medición como `performance.now()` y analizando el impacto en el rendimiento.
16. ¿Cómo se utilizan los "matchers" en Jest para hacer afirmaciones en pruebas?
    Los "matchers" son funciones que permiten hacer afirmaciones sobre los resultados de las pruebas, como `toBe()`, `toEqual()`, y `toContain()`, para verificar que el código funciona como se espera.
17. ¿Cómo se prueba la lógica de negocio en una aplicación backend utilizando Jest?
    Se prueba escribiendo pruebas unitarias que verifican que la lógica de negocio se comporta correctamente en diferentes escenarios y condiciones.
18. ¿Cómo se prueban las rutas y controladores en una aplicación backend con Jest y `supertest`?
    Se prueban enviando solicitudes HTTP a las rutas y verificando las respuestas, el estado de la respuesta, y el contenido utilizando `supertest` y Jest.
19. ¿Qué son los "test doubles" y cómo se utilizan en Jest para pruebas de backend?
    Los "test doubles" son versiones simuladas de funciones o módulos que se utilizan para controlar el comportamiento durante las pruebas, como mocks y stubs.
20. ¿Cómo se utilizan las configuraciones de Jest para manejar pruebas en diferentes entornos de ejecución?
    Se utilizan configuraciones como `testEnvironment` en `jest.config.js` para definir el entorno de ejecución, como `node` para pruebas de backend, y ajustar la configuración según sea necesario.

# Ssr

1. ¿Cómo se integran las pruebas de Jest en un pipeline de CI/CD para aplicaciones backend?
   Se integran configurando el pipeline para ejecutar Jest en cada commit o solicitud de extracción, utilizando herramientas de CI/CD como GitHub Actions, Jenkins o GitLab CI, y configurando pasos para ejecutar pruebas y generar informes de cobertura.

2. ¿Qué técnicas avanzadas se pueden utilizar para mejorar el rendimiento de las pruebas en Jest?
   Técnicas incluyen la ejecución paralela de pruebas utilizando `--runInBand` para reducir la sobrecarga de I/O, optimización de pruebas individuales para que sean menos costosas en términos de tiempo, y el uso de `jest --watch` para ejecutar solo pruebas relacionadas con los archivos modificados.
3. ¿Cómo se configuran pruebas para aplicaciones que utilizan bases de datos NoSQL en Jest?
   Se configuran utilizando una base de datos en memoria o una base de datos de prueba específica para el entorno de pruebas, inicializando y limpiando la base de datos antes y después de cada prueba para asegurar consistencia.
4. ¿Cómo se maneja el testing de servicios que interactúan con APIs externas utilizando Jest?
   Se maneja utilizando mocks para simular las respuestas de las APIs externas, permitiendo verificar que los servicios manejan las respuestas y errores de manera correcta sin realizar llamadas reales.
5. ¿Qué es el "test isolation" y cómo se asegura en Jest para pruebas de integración de backend?
   El "test isolation" asegura que las pruebas sean independientes entre sí. Se asegura utilizando mocks y spies para controlar el entorno de prueba y limpiando el estado global entre pruebas para evitar interferencias.
6. ¿Cómo se implementan pruebas de seguridad en el backend utilizando Jest?
   Se implementan utilizando herramientas de análisis de seguridad estática y dinámica junto con Jest, para identificar vulnerabilidades en el código y asegurar que las mejores prácticas de seguridad se siguen.
7. ¿Cómo se configuran pruebas para componentes que utilizan `async/await` en Jest?
   Se configuran utilizando `async/await` en las pruebas para manejar promesas y verificar que las funciones asíncronas se resuelven o rechazan correctamente.
8. ¿Cómo se realiza el "test data management" en Jest para aplicaciones grandes?
   Se realiza creando y gestionando datos de prueba utilizando fixtures, configurando datos en `beforeEach` y limpiando los datos en `afterEach` para asegurar un entorno de prueba limpio y consistente.
9. ¿Qué técnicas se utilizan para realizar pruebas de rendimiento en el backend con Jest?
   Se utilizan técnicas como medir el tiempo de ejecución de funciones, utilizar herramientas de benchmarking, y analizar el impacto de cambios en el rendimiento del backend.
10. ¿Cómo se escriben pruebas de integración para servicios y bases de datos utilizando Jest?
    Se escriben configurando el entorno de prueba para interactuar con servicios y bases de datos, y verificando que los componentes funcionen correctamente en conjunto.

11. ¿Cómo se utilizan los "test doubles" avanzados como stubs y mocks para pruebas de backend en Jest?
    Se utilizan para reemplazar componentes o módulos con versiones simuladas que permiten controlar el comportamiento durante las pruebas y verificar la interacción entre partes del sistema.
12. ¿Cómo se implementan pruebas para endpoints que utilizan autenticación en el backend con Jest?
    Se implementan configurando pruebas que simulan la autenticación y verifican que los endpoints manejen correctamente los tokens y permisos.
13. ¿Cómo se manejan las pruebas de errores y excepciones en Jest para aplicaciones backend más complejas?
    Se manejan utilizando `expect().toThrow()` para verificar que las funciones lancen excepciones esperadas y manejando errores en componentes de backend para asegurar que el sistema responde correctamente a condiciones excepcionales.
14. ¿Cómo se configuran las pruebas de rendimiento para aplicaciones backend con Jest?
    Se configuran midiendo el tiempo de ejecución de funciones o procesos críticos, utilizando herramientas complementarias para realizar análisis de rendimiento y verificar el impacto de cambios en el rendimiento del backend.
15. ¿Qué estrategias se pueden utilizar para reducir la flakiness en pruebas de backend con Jest?
    Estrategias incluyen asegurar que las pruebas sean independientes, utilizar datos consistentes, limpiar el estado global entre pruebas, y evitar dependencias externas que puedan variar.
16. ¿Cómo se manejan las pruebas para aplicaciones backend que utilizan diferentes entornos de base de datos?
    Se manejan configurando Jest para utilizar diferentes bases de datos para distintos entornos, y utilizando fixtures y scripts de configuración para asegurar que las pruebas se ejecuten en el entorno correcto.
17. ¿Qué es el "test coverage" y cómo se puede mejorar en pruebas de backend con Jest?
    El "test coverage" mide el porcentaje de código cubierto por pruebas. Se puede mejorar escribiendo pruebas adicionales para áreas no cubiertas, utilizando herramientas de análisis de cobertura y revisando los informes de cobertura generados por Jest.
18. ¿Cómo se realizan pruebas de integración para servicios distribuidos utilizando Jest?
    Se realizan configurando el entorno para interactuar con múltiples servicios distribuidos y verificando que funcionen correctamente en conjunto, utilizando mocks y stubs para controlar la interacción entre servicios.
19. ¿Cómo se implementan pruebas para aplicaciones backend que utilizan microservicios?
    Se implementan configurando pruebas que verifican la comunicación entre microservicios, utilizando mocks para simular servicios externos y asegurar que los microservicios funcionen correctamente en conjunto.

20. ¿Cómo se integran pruebas de rendimiento en el ciclo de vida de desarrollo de aplicaciones backend utilizando Jest?
    Se integran configurando el pipeline de CI/CD para incluir pruebas de rendimiento, utilizando herramientas de benchmarking y análisis de rendimiento para identificar problemas y asegurar que las aplicaciones mantengan un rendimiento aceptable.

# Sr

1. ¿Cómo se implementa una estrategia de pruebas de rendimiento para aplicaciones backend a gran escala utilizando Jest?
   Se implementa configurando pruebas de rendimiento para medir el tiempo de ejecución de funciones y procesos críticos, utilizando herramientas de benchmarking y análisis de rendimiento para identificar cuellos de botella y optimizar el rendimiento.
2. ¿Qué técnicas avanzadas se utilizan para asegurar la confiabilidad y consistencia de las pruebas en un entorno de CI/CD?
   Técnicas incluyen la utilización de entornos de prueba dedicados, la ejecución paralela de pruebas, el uso de contenedores para aislar pruebas y la configuración de pipelines de CI/CD para ejecutar pruebas en diferentes entornos.
3. ¿Cómo se gestionan y se integran las pruebas en aplicaciones backend que utilizan múltiples bases de datos en Jest?
   Se gestionan configurando Jest para interactuar con diferentes bases de datos según el entorno de prueba, utilizando scripts de configuración para inicializar y limpiar las bases de datos antes y después de cada prueba.
4. ¿Cómo se aborda la prueba de sistemas distribuidos y microservicios con Jest?
   Se aborda configurando pruebas que verifican la comunicación y la integración entre microservicios, utilizando mocks y stubs para simular servicios externos, y asegurando que todos los componentes del sistema funcionen correctamente en conjunto.
5. ¿Qué estrategias avanzadas se utilizan para reducir el tiempo de ejecución de pruebas en aplicaciones backend grandes?
   Estrategias incluyen la ejecución de pruebas en paralelo, la optimización de pruebas individuales, el uso de bases de datos en memoria para pruebas y la eliminación de pruebas redundantes.
6. ¿Cómo se maneja el testing de servicios que dependen de APIs externas en aplicaciones backend utilizando Jest?
   Se maneja utilizando mocks para simular las respuestas de las APIs externas, permitiendo verificar que los servicios manejen las respuestas y errores de manera correcta sin realizar llamadas reales.
7. ¿Cómo se integran pruebas de seguridad y vulnerabilidad en el ciclo de vida de desarrollo de aplicaciones backend con Jest?
   Se integran utilizando herramientas de análisis de seguridad y vulnerabilidad junto con Jest, realizando escaneos de seguridad automáticos y verificando que el código cumpla con las mejores prácticas de seguridad.

8. ¿Cómo se realiza el "test data management" para grandes conjuntos de datos en aplicaciones backend utilizando Jest?
   Se realiza utilizando técnicas como la creación de datos de prueba en `beforeEach`, el uso de fixtures para datos consistentes y la limpieza de datos en `afterEach` para mantener un entorno de prueba limpio.
9. ¿Qué son las pruebas de integración y cómo se configuran para servicios backend utilizando Jest?
   Las pruebas de integración verifican que diferentes componentes del sistema funcionen correctamente en conjunto. Se configuran utilizando Jest para interactuar con servicios y bases de datos, y verificar que la integración funcione como se espera.
10. ¿Cómo se implementan pruebas de carga y estrés para aplicaciones backend utilizando Jest?
    Se implementan utilizando herramientas de benchmarking y análisis de rendimiento, configurando pruebas para simular cargas de usuario y evaluar cómo el sistema maneja el estrés y la carga.
11. ¿Cómo se manejan las pruebas de compatibilidad y de regresión en aplicaciones backend utilizando Jest?
    Se manejan configurando pruebas para verificar que el sistema sea compatible con diferentes versiones de dependencias y que los cambios no introduzcan regresiones en el comportamiento esperado.
12. ¿Cómo se utilizan los informes de cobertura para mejorar la calidad del código en pruebas de backend con Jest?
    Se utilizan revisando los informes de cobertura generados por Jest para identificar áreas no cubiertas por pruebas, priorizando la escritura de pruebas adicionales y asegurando que el código esté bien cubierto.
13. ¿Qué es la "test flakiness" y cómo se aborda en el contexto de pruebas de backend con Jest?
    La "test flakiness" se refiere a pruebas que fallan de forma intermitente. Se aborda asegurando que las pruebas sean independientes, utilizando datos consistentes y limpiando el estado global entre pruebas para evitar efectos secundarios.
14. ¿Cómo se configuran y utilizan los "test doubles" avanzados como spies, mocks y stubs en Jest para pruebas de backend?
    Se configuran utilizando `jest.spyOn()`, `jest.mock()` y `jest.fn()` para crear versiones simuladas de funciones y módulos, permitiendo controlar el comportamiento durante las pruebas y verificar la interacción entre componentes.
15. ¿Cómo se realizan pruebas de compatibilidad entre diferentes versiones de dependencias en aplicaciones backend utilizando Jest?
    Se realizan configurando pruebas para verificar que el sistema funcione correctamente con diferentes versiones de dependencias y ejecutando pruebas en entornos de versiones variadas para asegurar compatibilidad.
16. ¿Cómo se gestionan las pruebas de aplicaciones backend en entornos distribuidos y en la nube con Jest?

Se gestionan configurando entornos de prueba en la nube, utilizando herramientas de gestión de configuración y asegurando que las pruebas se ejecuten en entornos que simulen las condiciones de producción.

17. ¿Cómo se implementan pruebas para aplicaciones backend que utilizan arquitecturas basadas en eventos utilizando Jest?
    Se implementan simulando eventos y verificando que los componentes reaccionen y manejen los eventos de manera correcta, utilizando mocks y stubs para controlar el comportamiento de los eventos.
18. ¿Cómo se integran las pruebas de backend con pruebas de frontend utilizando Jest?
    Se integran configurando el entorno de pruebas para ejecutar pruebas tanto de frontend como de backend, utilizando herramientas como `supertest` para pruebas de integración y asegurando que el frontend y backend funcionen correctamente juntos.
19. ¿Qué técnicas avanzadas se utilizan para asegurar la estabilidad y confiabilidad de las pruebas en aplicaciones backend a gran escala?
    Técnicas incluyen la ejecución en entornos aislados, la integración de herramientas de análisis de rendimiento y seguridad, y la configuración de pipelines de CI/CD para asegurar que las pruebas se ejecuten de manera confiable y consistente.
20. ¿Cómo se manejan las pruebas en aplicaciones backend que utilizan arquitecturas de microservicios con Jest?
    Se manejan configurando pruebas para verificar la comunicación y la integración entre microservicios, utilizando mocks y stubs para simular servicios externos y asegurando que los microservicios funcionen correctamente en conjunto.

# Testing Front

## Jr

1. ¿Qué es Jest y para qué se utiliza en el testing de aplicaciones frontend?
   Jest es un framework de testing para JavaScript que se utiliza para realizar pruebas unitarias, de integración y de extremo a extremo en aplicaciones frontend. Permite escribir pruebas para verificar que el código se comporta como se espera.
2. ¿Cómo se configura Jest en un proyecto de React?
   Se configura instalando Jest y sus dependencias mediante npm o yarn, y creando un archivo de configuración `jest.config.js` si es necesario. También se puede configurar Jest en el archivo `package.json`.
3. ¿Qué son los snapshots en Jest y cómo se utilizan en el testing de componentes de React?
   Los snapshots en Jest son una forma de guardar una representación del componente en un momento dado. Se utilizan para realizar pruebas de regresión, comparando el componente actual con el snapshot guardado anteriormente para detectar cambios inesperados.
4. ¿Cómo se realiza una prueba unitaria básica con Jest en un componente React?
   Se realiza importando el componente y renderizándolo con herramientas como

`@testing-library/react`, luego se utilizan métodos de aserción para verificar que el componente se comporta como se espera.

5. ¿Qué es `@testing-library/react` y cómo se utiliza en conjunto con Jest para probar componentes React?
   `@testing-library/react` es una biblioteca que proporciona utilidades para renderizar componentes React y realizar aserciones sobre su comportamiento. Se utiliza con Jest para escribir pruebas más centradas en el comportamiento del usuario.
6. ¿Cómo se escribe una prueba que verifica si un componente React renderiza correctamente un texto?
   Se escribe renderizando el componente con `@testing-library/react` y utilizando el método `getByText` para verificar que el texto esperado está presente en el documento.
7. ¿Cómo se simulan eventos como clics en pruebas con Jest y `@testing-library/react`?
   Se simulan eventos utilizando el método `fireEvent` de `@testing-library/react`, que permite disparar eventos como clics, cambios de texto y otros eventos del DOM.
8. ¿Cómo se utiliza Jest para probar funciones o métodos en un archivo de utilidad?
   Se importan las funciones o métodos en el archivo de prueba y se escriben pruebas unitarias que verifican su comportamiento esperado utilizando métodos de aserción de Jest.
9. ¿Qué es el archivo de configuración `jest.config.js` y qué opciones comunes se configuran en él?
   `jest.config.js` es el archivo de configuración para Jest donde se pueden especificar opciones como el entorno de prueba, los directorios a ignorar y las transformaciones de archivos. Opciones comunes incluyen `testEnvironment`, `transform` y `moduleNameMapper`.
10. ¿Cómo se utilizan los mocks en Jest para simular dependencias en una prueba?
    Se utilizan funciones de mock como `jest.mock()` para reemplazar módulos o dependencias con implementaciones simuladas durante la prueba, permitiendo controlar y verificar su comportamiento.
11. ¿Qué es `jest.fn()` y cómo se utiliza para crear funciones simuladas?
    `jest.fn()` es una función que se utiliza para crear funciones simuladas o mock functions. Permite espiar llamadas a funciones, proporcionar implementaciones simuladas y verificar llamadas.
12. ¿Cómo se realiza una prueba que verifica si un componente React se renderiza correctamente con propiedades?
    Se realiza pasando las propiedades al componente al renderizarlo con `@testing-library/react` y utilizando métodos de aserción para verificar que el componente muestra el contenido esperado basado en esas propiedades.
13. ¿Qué es un "spy" en Jest y cómo se utiliza para espiar llamadas a funciones?
    Un "spy" en Jest es una función mock que permite interceptar y registrar llamadas a

una función. Se utiliza para verificar que una función ha sido llamada con ciertos argumentos o un número específico de veces.

14. ¿Cómo se realiza una prueba que verifica si un componente maneja correctamente el estado interno?
    Se realiza renderizando el componente, interactuando con él para cambiar el estado (por ejemplo, haciendo clic en un botón), y luego verificando que el estado interno se actualiza como se espera.
15. ¿Cómo se utiliza `beforeEach` y `afterEach` en Jest para preparar y limpiar el entorno de prueba?
    `beforeEach` se utiliza para ejecutar código antes de cada prueba, como configurar el estado inicial, mientras que `afterEach` se utiliza para limpiar o restablecer el estado después de cada prueba.
16. ¿Qué es `describe` en Jest y cómo se utiliza para agrupar pruebas relacionadas?
    `describe` es una función que se utiliza para agrupar pruebas relacionadas en bloques. Permite organizar las pruebas en suites y aplicar configuraciones específicas a un grupo de pruebas.
17. ¿Cómo se escriben pruebas asíncronas en Jest para componentes que realizan operaciones asíncronas?
    Se escriben utilizando `async/await` para manejar operaciones asíncronas en las pruebas, y se asegura que las promesas se resuelvan antes de que las pruebas terminen utilizando `await` en la llamada a funciones asíncronas.
18. ¿Cómo se realiza una prueba que verifica si un componente React maneja correctamente errores?
    Se realiza simulando una condición de error (por ejemplo, pasando un error al componente) y verificando que el componente muestra la información de error adecuada o realiza la acción esperada.
19. ¿Cómo se utilizan los "mock modules" en Jest para simular módulos completos?
    Se utilizan `jest.mock()` para reemplazar un módulo completo con una implementación simulada, permitiendo controlar el comportamiento del módulo y verificar interacciones durante las pruebas.
20. ¿Qué son los "test doubles" y cómo se utilizan en Jest?
    Los "test doubles" son objetos utilizados en pruebas para reemplazar dependencias reales. En Jest, se utilizan mocks, stubs y spies como test doubles para simular comportamientos y verificar interacciones.

# Ssr

1. ¿Cómo se integra Jest con herramientas de pruebas end-to-end como Cypress?
   Se integra utilizando Jest para pruebas unitarias y de integración, mientras que Cypress se utiliza para pruebas end-to-end. Ambos pueden coexistir en el mismo proyecto, con Jest manejando las pruebas de unidades y Cypress las pruebas de flujos completos.

2. ¿Qué es `jest.mock()` y cómo se usa para simular módulos específicos en pruebas?
   `jest.mock()` se utiliza para reemplazar un módulo completo con una versión simulada. Permite definir cómo debería comportarse el módulo simulado, proporcionando un control total sobre las dependencias.
3. ¿Cómo se realizan pruebas de rendimiento en componentes React utilizando Jest?
   Se realizan utilizando herramientas y técnicas adicionales como `react-performance-testing` junto con Jest para medir y analizar el rendimiento de los componentes, identificando posibles cuellos de botella.
4. ¿Qué son los "test coverage reports" en Jest y cómo se generan?
   Los "test coverage reports" proporcionan información sobre qué partes del código están cubiertas por pruebas. Se generan ejecutando Jest con la opción `--coverage`, que crea informes detallados de cobertura de código.
5. ¿Cómo se realiza la simulación de módulos de terceros en Jest cuando se prueba un componente React?
   Se realiza utilizando `jest.mock()` para simular módulos de terceros, como bibliotecas o servicios, permitiendo controlar sus comportamientos y verificar cómo el componente React interactúa con ellos.
6. ¿Qué es `jest.spyOn()` y cómo se utiliza para espiar métodos de objetos en pruebas?
   `jest.spyOn()` se utiliza para espiar métodos de objetos, permitiendo interceptar y registrar llamadas a métodos. Se utiliza para verificar que los métodos se llaman con los argumentos correctos y en el momento adecuado.
7. ¿Cómo se realizan pruebas de integración de componentes React utilizando Jest y `@testing-library/react`?
   Se realizan renderizando varios componentes juntos, interactuando con ellos y verificando que se comportan como se espera en conjunto. Se prueban interacciones y la integración entre componentes.
8. ¿Cómo se implementan pruebas de regresión con Jest en una aplicación React?
   Se implementan utilizando snapshots para guardar la representación de los componentes en un momento dado. Las pruebas de regresión comparan el snapshot actual con el guardado previamente para detectar cambios inesperados.
9. ¿Qué es el "test environment" en Jest y cómo se configura para pruebas frontend?
   El "test environment" define el entorno en el que se ejecutan las pruebas. En Jest, se configura con la opción `testEnvironment` en `jest.config.js`, configurando el entorno como `jsdom` para pruebas frontend en un entorno simulado de navegador.
10. ¿Cómo se realizan pruebas de accesibilidad en componentes React utilizando Jest?
    Se realizan utilizando bibliotecas de pruebas de accesibilidad como `jest-axe` junto con Jest para verificar que los componentes cumplen con las normas de accesibilidad y no presentan problemas como contrastes insuficientes.
11. ¿Cómo se utiliza Jest para realizar pruebas de componentes que dependen de un contexto de React?
    Se utiliza envolviendo el componente en un proveedor de contexto durante la

renderización de las pruebas, permitiendo que el componente acceda y utilice el contexto necesario para su funcionamiento.

12. ¿Qué técnicas se pueden usar para mejorar la velocidad de las pruebas en Jest?
    Técnicas incluyen la ejecución paralela de pruebas utilizando `--runInBand` para reducir la sobrecarga de I/O, la optimización de pruebas individuales para que sean menos costosas en términos de tiempo, y el uso de `jest --watch` para ejecutar solo pruebas relacionadas con los archivos modificados.
13. ¿Cómo se simulan respuestas de APIs en las pruebas con Jest?
    Se simulan respuestas de APIs utilizando `jest.mock()` para reemplazar módulos que hacen llamadas a APIs con versiones simuladas que devuelven respuestas predefinidas.
14. ¿Cómo se escribe una prueba de integración que verifica la comunicación entre varios componentes en React?
    Se escribe renderizando los componentes juntos en un entorno de prueba y verificando que interactúan correctamente entre sí, como pasar datos de un componente a otro o verificar la actualización del estado.
15. ¿Cómo se utilizan los mocks para pruebas en Jest para verificar el comportamiento de llamadas a APIs?
    Se utilizan `jest.mock()` para crear versiones simuladas de funciones que realizan llamadas a APIs, permitiendo verificar que se llaman con los argumentos correctos y manejar respuestas simuladas para las pruebas.
16. ¿Qué es `jest.clearAllMocks()` y cuándo se debe utilizar en pruebas?
    `jest.clearAllMocks()` se utiliza para restablecer todos los mocks en Jest, limpiando el historial de llamadas y restaurando las implementaciones originales. Se debe utilizar en `beforeEach` o `afterEach` para evitar efectos secundarios entre pruebas.
17. ¿Cómo se configuran pruebas para componentes React que usan Hooks personalizados?
    Se configuran renderizando el componente que utiliza el Hook personalizado y verificando su comportamiento a través de la interfaz pública del componente, como los efectos secundarios y las actualizaciones del estado.
18. ¿Cómo se implementa el testing de componentes con React Router utilizando Jest?
    Se implementa envolviendo los componentes en un proveedor de `Router` durante la renderización de las pruebas, permitiendo simular la navegación y verificar que los componentes se comportan correctamente en diferentes rutas.
19. ¿Cómo se realizan pruebas de rendimiento en componentes React con Jest?
    Se realizan midiendo el tiempo que tarda en ejecutarse un componente o función utilizando herramientas como `performance.now()` y analizando los resultados para identificar posibles problemas de rendimiento.
20. ¿Qué son los "mock implementations" en Jest y cómo se utilizan para pruebas más avanzadas?
    Los "mock implementations" son implementaciones simuladas de funciones o módulos que se utilizan para controlar el comportamiento de las dependencias durante las

pruebas. Se definen utilizando `jest.fn()` o `jest.mock()` con una implementación personalizada.

## Sr

1. ¿Cómo se integra Jest con un pipeline de CI/CD para garantizar la ejecución automática de pruebas?
   Se integra configurando el pipeline para ejecutar Jest en cada commit o solicitud de extracción, utilizando herramientas de CI/CD como GitHub Actions, Jenkins o GitLab CI, y configurando pasos para ejecutar pruebas y generar informes de cobertura.
2. ¿Qué estrategias se pueden emplear para manejar el estado global en pruebas con Jest para una aplicación grande?
   Se pueden emplear estrategias como el uso de mocks y spies para simular el estado global, el uso de contextos y proveedores de estado para configurar el estado necesario en cada prueba, y la limpieza del estado global entre pruebas para evitar interferencias.
3. ¿Cómo se implementan pruebas de carga para una aplicación frontend utilizando Jest?
   Se implementan utilizando herramientas complementarias como `artillery` o `k6` para simular carga en la aplicación y verificar el rendimiento bajo condiciones de alta demanda, mientras Jest se encarga de las pruebas unitarias y de integración.
4. ¿Qué es el "test isolation" y cómo se asegura en Jest para pruebas de componentes complejos?
   El "test isolation" asegura que las pruebas sean independientes entre sí, evitando que una prueba afecte a otra. Se asegura en Jest utilizando mocks para controlar dependencias, configurando el entorno de prueba adecuadamente y limpiando el estado global entre pruebas.
5. ¿Cómo se utilizan los "test doubles" avanzados como stubs y mocks en Jest para pruebas de integración complejas?
   Se utilizan para reemplazar componentes o módulos con versiones simuladas que controlan el comportamiento durante las pruebas, permitiendo verificar la interacción entre partes del sistema sin depender de implementaciones reales.
6. ¿Cómo se realizan pruebas de seguridad en una aplicación frontend utilizando Jest?
   Se realizan utilizando herramientas de análisis de seguridad estática y dinámica junto con Jest, para identificar vulnerabilidades y asegurar que el código cumple con las mejores prácticas de seguridad.
7. ¿Qué técnicas se utilizan para escribir pruebas que sean resistentes a cambios en el código base?
   Se utilizan técnicas como pruebas basadas en el comportamiento del usuario en lugar de la implementación, el uso de mocks y spies para controlar el entorno de prueba, y la escritura de pruebas de integración que verifican la funcionalidad general en lugar de detalles específicos.

8. ¿Cómo se integra Jest con herramientas de análisis estático de código para mejorar la calidad de las pruebas?
   Se integra utilizando herramientas como ESLint y Prettier en el pipeline de CI/CD junto con Jest para asegurar que el código está bien formateado y sigue las mejores prácticas, y para detectar errores potenciales antes de ejecutar las pruebas.
9. ¿Cómo se manejan las pruebas de componentes React que interactúan con APIs externas en Jest?
   Se manejan utilizando mocks para simular las respuestas de las APIs externas, permitiendo que las pruebas verifiquen cómo los componentes manejan las respuestas y los errores sin hacer llamadas reales a las APIs.
10. ¿Qué son los "fixtures" en el contexto de pruebas y cómo se utilizan en Jest?
    Los "fixtures" son datos de prueba predefinidos que se utilizan para configurar el entorno de prueba. Se utilizan en Jest para proporcionar datos consistentes y controlados para las pruebas, asegurando que los resultados sean reproducibles.
11. ¿Cómo se implementa el testing de rendimiento para aplicaciones frontend utilizando Jest?
    Se implementa midiendo el tiempo de ejecución de funciones y componentes, y utilizando herramientas como `benchmark.js` para realizar pruebas de rendimiento y analizar el impacto de cambios en el código en el tiempo de ejecución.
12. ¿Qué es el "test data management" y cómo se gestiona en Jest para aplicaciones grandes?
    El "test data management" se refiere a la creación, gestión y limpieza de datos de prueba. Se gestiona en Jest utilizando fixtures, creando datos de prueba en `beforeEach` y limpiando los datos en `afterEach` para asegurar un entorno de prueba limpio.
13. ¿Cómo se maneja el testing de componentes que utilizan Web APIs como `fetch` en Jest?
    Se maneja utilizando `jest.mock()` para simular el comportamiento de `fetch`, proporcionando respuestas predefinidas para las pruebas y verificando cómo los componentes manejan las respuestas y errores de las Web APIs.
14. ¿Cómo se realizan pruebas de compatibilidad entre navegadores utilizando Jest?
    Se realizan configurando Jest para ejecutarse en diferentes entornos de prueba o utilizando herramientas de pruebas específicas para compatibilidad entre navegadores, como `browserstack`, junto con Jest para validar el comportamiento en múltiples navegadores.
15. ¿Cómo se realizan pruebas de usabilidad en una aplicación frontend utilizando Jest?
    Se realizan utilizando herramientas de pruebas de interfaz de usuario que se integran con Jest para verificar que la interfaz es intuitiva y fácil de usar, y para identificar problemas de usabilidad en el diseño de la aplicación.
16. ¿Cómo se configuran pruebas para aplicaciones frontend con SSR (Server-Side Rendering) en Jest?
    Se configuran renderizando componentes en el entorno del servidor y verificando que

se comportan correctamente durante el renderizado del lado del servidor, utilizando herramientas como `react-dom/server` junto con Jest.

17. ¿Cómo se gestionan las pruebas de componentes React que dependen de servicios externos en Jest?
    Se gestionan utilizando mocks para simular los servicios externos y proporcionar respuestas controladas durante las pruebas, permitiendo verificar el comportamiento de los componentes sin depender de servicios reales.
18. ¿Qué es el "test flakiness" y cómo se evita en Jest?
    El "test flakiness" se refiere a pruebas que fallan de forma intermitente. Se evita en Jest asegurando que las pruebas sean independientes, utilizando datos consistentes y limpiando el estado entre pruebas para evitar efectos secundarios.
19. ¿Cómo se utilizan los "test reports" para analizar y mejorar la calidad del testing en Jest?
    Se utilizan para revisar los resultados de las pruebas y los informes de cobertura, identificando áreas problemáticas y oportunidades de mejora. Los informes ayudan a analizar la eficacia de las pruebas y a priorizar áreas de refactorización.
20. ¿Cómo se manejan las pruebas de componentes que interactúan con la base de datos en Jest?
    Se manejan utilizando mocks para simular la interacción con la base de datos, proporcionando respuestas predefinidas y verificando cómo los componentes manejan los datos, sin realizar operaciones reales en la base de datos.

# TypeScript

## Nivel Junior

1. ¿Qué es TypeScript?
   TypeScript es un superset de JavaScript que añade tipado estático opcional, lo que permite a los desarrolladores escribir código más robusto y detectar errores antes de la ejecución.
2. ¿Cuál es la diferencia entre `let`, `const` y `var` en TypeScript?
   `let` y `const` son formas de declarar variables con un ámbito de bloque, mientras que `var` tiene un ámbito de función. `const` crea una referencia inmutable.
3. ¿Qué es la inferencia de tipos en TypeScript?
   La inferencia de tipos es una característica de TypeScript que permite determinar automáticamente el tipo de una variable según su valor inicial sin necesidad de declarar explícitamente su tipo.
4. ¿Cómo defines un tipo personalizado en TypeScript?
   Se puede definir un tipo personalizado utilizando `type` o `interface`. Ejemplo: `type Persona = { nombre: string, edad: number };`
5. ¿Qué es una interfaz en TypeScript y para qué se usa?
   Una interfaz en TypeScript define la forma de un objeto y se utiliza para describir la estructura de datos, permitiendo que diferentes objetos sigan el mismo contrato.

6. ¿Qué es el archivo `tsconfig.json` y para qué sirve?
   El archivo `tsconfig.json` es el archivo de configuración de un proyecto TypeScript. Define las opciones del compilador y los archivos que deben incluirse o excluirse en el proceso de compilación.
7. ¿Qué es el tipo `any` y cuándo debería evitarse su uso?
   El tipo `any` permite asignar cualquier tipo de valor a una variable, anulando las verificaciones de tipos. Debe evitarse porque elimina las ventajas del tipado estático.
8. ¿Qué es el tipo `unknown` en TypeScript y en qué se diferencia de `any`?
   `unknown` es un tipo más seguro que `any`. No permite acceder a las propiedades o métodos del valor sin realizar una verificación previa de tipo, lo que lo hace más seguro que `any`.
9. ¿Qué son los tipos literales en TypeScript?
   Los tipos literales permiten definir un valor exacto que una variable puede tomar. Ejemplo: `let dirección: "norte" | "sur" | "este" | "oeste";`
10. ¿Qué son los `tuples` en TypeScript?
    Una tupla es un tipo de array con un número fijo de elementos y tipos específicos para cada posición. Ejemplo: `let persona: [string, number] = ["Juan", 25];`

## Nivel Semi-Senior (SSR)

1. ¿Qué son los tipos genéricos en TypeScript y cómo se usan?
   Los tipos genéricos permiten crear componentes reutilizables que funcionan con varios tipos. Se definen utilizando una variable de tipo entre los símbolos `<>`. Ejemplo: `function identidad<T>(arg: T): T { return arg; }`.
2. ¿Cómo se manejan los tipos opcionales en TypeScript?
   Los tipos opcionales se indican con un `?` después del nombre del parámetro. Ejemplo: `function saludar(nombre?: string) {}`. Esto indica que el parámetro puede ser omitido.
3. ¿Qué son los `union types` y cómo funcionan en TypeScript?
   Los `union types` permiten que una variable pueda tener más de un tipo. Se usan con el símbolo `|`. Ejemplo: `let id: number | string;` que permite que `id` sea tanto un número como un string.
4. ¿Cómo se define un tipo personalizado en TypeScript?
   Puedes definir un tipo personalizado usando `type`. Ejemplo: `type ID = string | number;` crea un tipo `ID` que puede ser un `string` o un `number`.
5. ¿Cómo funciona la herencia en TypeScript?
   La herencia en TypeScript se implementa con la palabra clave `extends`, lo que permite que una clase herede de otra. Ejemplo: `class Estudiante extends Persona {}`.
6. ¿Qué son los `mapped types` en TypeScript?
   Los `mapped types` permiten crear nuevos tipos a partir de las propiedades de un tipo existente. Ejemplo: `type SoloLectura<T> = { readonly [K in keyof T]: T[K] };` crea un tipo donde todas las propiedades son de solo lectura.

7. ¿Qué es el `strictNullChecks` en TypeScript y cómo ayuda a prevenir errores?
   Cuando `strictNullChecks` está habilitado en TypeScript, las variables no pueden ser `null` o `undefined` a menos que estén explícitamente tipadas para permitir estos valores, lo que ayuda a evitar errores en tiempo de ejecución.
8. ¿Qué son los `decorators` en TypeScript y para qué se utilizan?
   Los decoradores son funciones que se aplican a clases, métodos, propiedades o parámetros para modificar su comportamiento. Se implementan utilizando `@` antes del nombre de la función decoradora.
9. ¿Cómo se manejan las sobrecargas de funciones en TypeScript?
   La sobrecarga de funciones permite definir varias firmas para una función con diferentes tipos de parámetros. La implementación final debe manejar todos los tipos permitidos. Ejemplo:

```
function combinar(a: number, b: number): number; function combinar(a:
string, b: string): string; function combinar(a: any, b: any): any {
return a + b; }```
```

10. ¿Qué es el narrowing de tipos en TypeScript?
    El narrowing de tipos es la técnica que permite refinar el tipo de una variable dentro de un bloque de código mediante condicionales o verificaciones de tipo. Ejemplo: `if (typeof valor === "string") { /* valor es un string aquí */ }`.

## Nivel Senior (SR)

1. ¿Qué son los condicionales de tipos en TypeScript?
   Los condicionales de tipos permiten aplicar lógica condicional a los tipos en función de otros tipos. Esto se hace utilizando la sintaxis `T extends U ? X : Y`.
2. ¿Cómo funcionan los `mapped types` avanzados en TypeScript?
   Los `mapped types` avanzados permiten transformar las propiedades de un tipo de manera dinámica, aplicando operaciones a cada una de las propiedades de un tipo existente para crear un nuevo tipo.
3. ¿Qué es el operador `keyof` en TypeScript y cómo se usa?
   `keyof` es un operador que permite obtener un tipo compuesto por las claves de un objeto. Es útil para garantizar que solo se usen propiedades válidas de un objeto en tiempo de compilación.
4. ¿Qué es un tipo condicional distribuido y cómo se utiliza?
   Un tipo condicional distribuido se aplica cuando un tipo es una unión. Los tipos condicionales se distribuyen a cada miembro de la unión, permitiendo aplicar lógica condicional a cada tipo en la unión.
5. ¿Cómo se usan los tipos inferidos dentro de condicionales en TypeScript?
   Se utiliza el operador `infer` para extraer tipos dentro de los condicionales,

permitiendo manipular tipos de forma dinámica según los tipos inferidos en tiempo de compilación.

6. ¿Qué es un `lookup type` en TypeScript y cómo funciona?
   Un `lookup type` permite acceder al tipo de una propiedad específica de un objeto. Se usa la sintaxis `T[K]`, donde `T` es el objeto y `K` es una clave dentro de ese objeto.
7. ¿Qué es el `utility type` `Partial` en TypeScript y cómo se utiliza?
   `Partial<T>` es un tipo utilitario que convierte todas las propiedades de un tipo `T` en opcionales. Esto permite crear versiones incompletas de un objeto sin necesidad de definir todas sus propiedades.
8. ¿Qué son los tipos `readonly` en TypeScript y cuándo deberían usarse?
   Los tipos `readonly` permiten definir propiedades que no pueden ser modificadas después de su inicialización. Esto es útil para asegurar la inmutabilidad de los objetos.
9. ¿Cómo se puede optimizar el uso de tipos avanzados para mejorar el rendimiento del código en tiempo de compilación?
   El uso de tipos más simples, evitando condicionales complejos innecesarios o inferencias de tipos excesivas, puede mejorar significativamente el tiempo de compilación en proyectos grandes.
10. ¿Qué son los archivos de declaración ( `.d.ts` ) en TypeScript y cuál es su importancia en proyectos grandes?
    Los archivos de declaración ( `.d.ts` ) proporcionan definiciones de tipos para bibliotecas o módulos que no están escritos en TypeScript. Son importantes para garantizar que se respeten los contratos de tipos en proyectos que mezclan JavaScript y TypeScript.