1). How does async/await help with performance and scalability?

ANS:

When it comes to pure processing power or throughput, async/await has no direct impact on performance or scalability. Nonetheless, it can significantly improve the asynchronous code's readability, maintainability, and concurrency management, which in some circumstances can tangentially lead to better performance and scalability. How to do it is as follows:

Better Readability and Maintainability: Async/await-written asynchronous code is typically easier for developers to comprehend and maintain since it is more readable and resembles synchronous code. This clarity can facilitate quicker debugging and development, which will ultimately produce code that is more effective.

Simplified Error management: In asynchronous programmes, error management is made simpler by async/await. Error management in conventional callback- or promise-based programmes can get layered and confusing, which can result in issues. Try-catch blocks can be used with async/await to handle errors, which simplifies error management and maintains clean code.

Sequential Code Execution: You can write asynchronous code that appears synchronous and runs sequentially with the help of async/await. This can be useful in situations when the outcome of one asynchronous action influences another. Writing code sequentially can help you write code that is more organised and may even perform better because you can avoid writing complicated callback chains or promise chains.

2). Is it possible to use async/await with promise chains? If yes, how can this be achieved?

ANS:

Async/await may be used with Promise chains, yes. Actually, to handle asynchronous tasks in a more understandable and synchronous-like way, async/await is frequently used in conjunction with Promises.

Await can be used to halt the execution of an async function until a Promise is accepted or denied when utilising async/await with Promise chains. This enables you to develop asynchronous code that exhibits synchronous behaviour and appearance.

Here's how to make this happen:

Use await to halt execution of an async function until a promise is fulfilled. Define the async function. After that, you can handle the resolved or rejected Promise by chaining then and catching as normal.

3). Give 3 real world examples where async/await has been used?

ANS

In many different sectors of contemporary JavaScript development, async/await is a common tool. Here are three instances of real-world applications that frequently use async/await:

Web Design:
HTTP Requests:

In web applications, async/await is often used to send asynchronous HTTP requests to backend servers or APIs. This involves utilising libraries like Axios or Fetch API to retrieve data from services.
Database Operations:

Using libraries like Mongoose (for MongoDB) or Sequelize (for SQL databases), async/await makes it easier to handle asynchronous database operations like querying, inserting, updating, or deleting data while working with databases in web development.

Development on the Server Side:

Middleware Functions:

Async/await is frequently used for building middleware functions that carry out asynchronous activities like logging, authentication, input validation, or error handling in server-side development using Node.js or other backend frameworks.
File System activities: Asynchronous file system activities, like reading and writing files, making directories, and deleting files, are carried out using async/await. This is especially helpful for server-side programmes that handle file uploads, file manipulation, or caching based on files.

Web and Mobile Application Development:
Background Tasks:

Async/await is used in desktop and mobile applications to execute asynchronous operations or background tasks without obstructing the primary UI thread. Tasks like file downloads, data processing, and data syncing with distant servers may fall under this category.
Asynchronous APIs: To communicate with asynchronous SDKs or APIs offered by desktop or mobile platforms, use async/await. This could entail utilising async methods to asynchronously access platform-specific services or device features like cameras, sensors, and geolocation.

4).

```
async function inc(x) {
x = x + await 1
return x;
}
async function increment(x){
x = x + 1
```

```
return x
}
inc(1).then(function(x){
increment(x).then(function(x){
console.log(x)
})
```

Find output.

ANS:

Let's examine the code in detail:

An asynchronous function is what the inc function is defined as. It accepts a parameter x, uses the await keyword to add 1 to it, and then returns the outcome.
Even though it doesn't use await, the increment method is asynchronous as well. It just returns the result after adding 1 to the input x.

The argument passed to the inc method is 1. One is added to x inside of inc using await. This code will, however, fail since await is only permitted to be used inside of async functions; it cannot be used outside of them.
As a result, before any output is produced, the code will produce an error.

5).
```
let p = new Promise(function (resolve, reject) {
reject(new Error("some error"));
setTimeout(function(){
reject(new Error("some error"));
},1000)
reject(new Error("some error"));
});
p.then(null, function (err) {
console.log(1);
console.log(err);
}).catch(function (err) {
console.log(2);
console.log(err);
});
```

Find output.

ANS:

The output of this code will be

```
1
Error: some error
```

6).
```
async function f1() {
console.log(1);
}
async function f1() {
console.log(2);
}
console.log(3);
f1();
console.log(1);
f2();
async function f2() {
console.log("Go!");
}
```
Find output.

ANS:
3
1
2
Go!

7).
```
function resolveAfterNSeconds(n,x) {
return new Promise(resolve => {
setTimeout( ( ) = {
resolve(x);
}, n);
});
}
(function(){
let a = resolveAfterNSeconds(1000,1)
a.then(async function(x){
let y = await resolveAfterNSeconds(2000,2)
let z = await resolveAfterNSeconds(1000,3)
let p = resolveAfterNSeconds(2000,4)
let q = resolveAfterNSeconds(1000,5)
console.log(x+y+z+await p +await q);
})
})()
```
Find output.

ANS:
the output will be the sum of these values: `1 + 2 + 3 + 4 + 5 = 15`.

8). Is it possible to nest async functions in JavaScript? Explain with examples.

ANS: regular functions. This allows you to create asynchronous code that is easier to read and maintain, especially when dealing with complex asynchronous operations. Here's an example to illustrate nesting async functions:

```
async function outerFunction() {
  console.log('Outer function start');

  // Inside the outer function, we define an inner async function
  async function innerFunction() {
    console.log('Inner function start');

    // Some asynchronous operation
    await new Promise(resolve => setTimeout(resolve, 1000));

    console.log('Inner function end');

    return 42;
  }

  // Call the inner async function
  const result = await innerFunction();

  console.log('Outer function end');

  return result;
}

// Call the outer async function
outerFunction().then(result => {
  console.log('Result:', result);
});
```

In this instance:
Our async outside function is called outerFunction.
We define an inner async function called innerFunction within outerFunction.
An asynchronous operation (a setTimeout in this case) is carried out by innerFunction.
The outcome of innerFunction is anticipated by outerFunction.

When outerFunction is invoked, its execution pauses until innerFunction is finished.
Ultimately, we invoke outerFunction, and upon its completion, we record the outcome.
This illustrates how to efficiently handle asynchronous code flow in JavaScript by nesting async methods.

9). What is the best way to avoid deadlocks when using async/await?

ANS: When utilizing async/await, it's important to carefully design your asynchronous code to avoid scenarios where several asynchronous activities are waiting on one another to finish in order to avoid deadlocks. Best methods to prevent deadlocks include the following:

Employ Non-blocking Operations: Make sure your asynchronous processes don't pause while waiting on one another to finish. When handling I/O tasks like file reading, network requests, or database queries, this is very crucial. Make use of the asynchronous APIs that libraries or Node.js built-in functions provide.

Steer Clear of Synchronous Calls: Steer clear of combining asynchronous and synchronous code in the same execution context. If synchronous operations must be carried out within an async function, think about assigning the synchronous tasks to a different thread or process or utilizing asynchronous alternatives.

Be Cautious When Using Locks and Mutexes: If we code makes use of locks or mutexes to synchronize access to shared resources, be cautious when using them within async functions. Don't hold them for too long. Deadlocks can occur when a lock is held while waiting for another asynchronous operation.

Avoid Nested Async Functions: While excessive nesting can make code more difficult to understand and increase the chance of deadlocks, it is occasionally important to nest async functions. Divide intricately nested async functions into more understandable, simpler functions.

When we have several asynchronous activities to wait for, use Promise.all() to run them concurrently instead than sequentially. This lets each operation proceed on its own, preventing deadlocks.

10). In which scenarios would you use synchronous code instead of asynchronous code?

ANS:

Usually, synchronous code is utilized when
Simple and Sequential tasks: Synchronous code can be simpler and easier to understand for short, straightforward tasks that don't require extensive processing or input/output.

Performance: When it comes to CPU-bound jobs when the overhead of asynchronous operations outweighs the advantages, synchronous code may occasionally perform better than asynchronous code.

Error Handling: Synchronous programming makes it easier to manage exceptions and errors inside the same execution context by enabling more straightforward error handling with try/catch blocks.

Control Flow: Synchronous code offers a simpler control flow that facilitates managing state changes and reasoning about the sequence of actions.

Small scripts and rapid prototyping: Without having to handle asynchronous behavior, synchronous code may be more than sufficient for short scripts or rapid prototyping.

Evaluating the trade-offs between synchronous and asynchronous programming is crucial, though. Long-running processes, actions that benefit from parallelism and concurrency, and scenarios involving I/O operations (such network requests, file I/O, or database searches) are often better suited for asynchronous code.

Asynchronous programming using techniques like Promises, async/await, and event-driven architecture is frequently preferred in contemporary JavaScript applications when responsiveness and scalability are critical in order to manage concurrent tasks quickly without stopping the event loops.