PER SCHOLAS

ALAB 308A.3.1:

Promises and async/await

Version 1.0, 10/17/23

Click here to open in a separate window.

Introduction

This assignment showcases some practical uses of asynchronous JavaScript techniques, including Promises and the async/await syntax. Asynchronous code is *extremely* common in web application development, and its uses are varied. This assignment highlights one of the most common uses.

Objectives

- Use async/await syntax to gather data from asynchronous sources.
- Use Promises to handle asynchronous code.

Submission

Submit your completed lab using the **Start Assignment** button on the assignment page in Canvas.

Your submission should include:

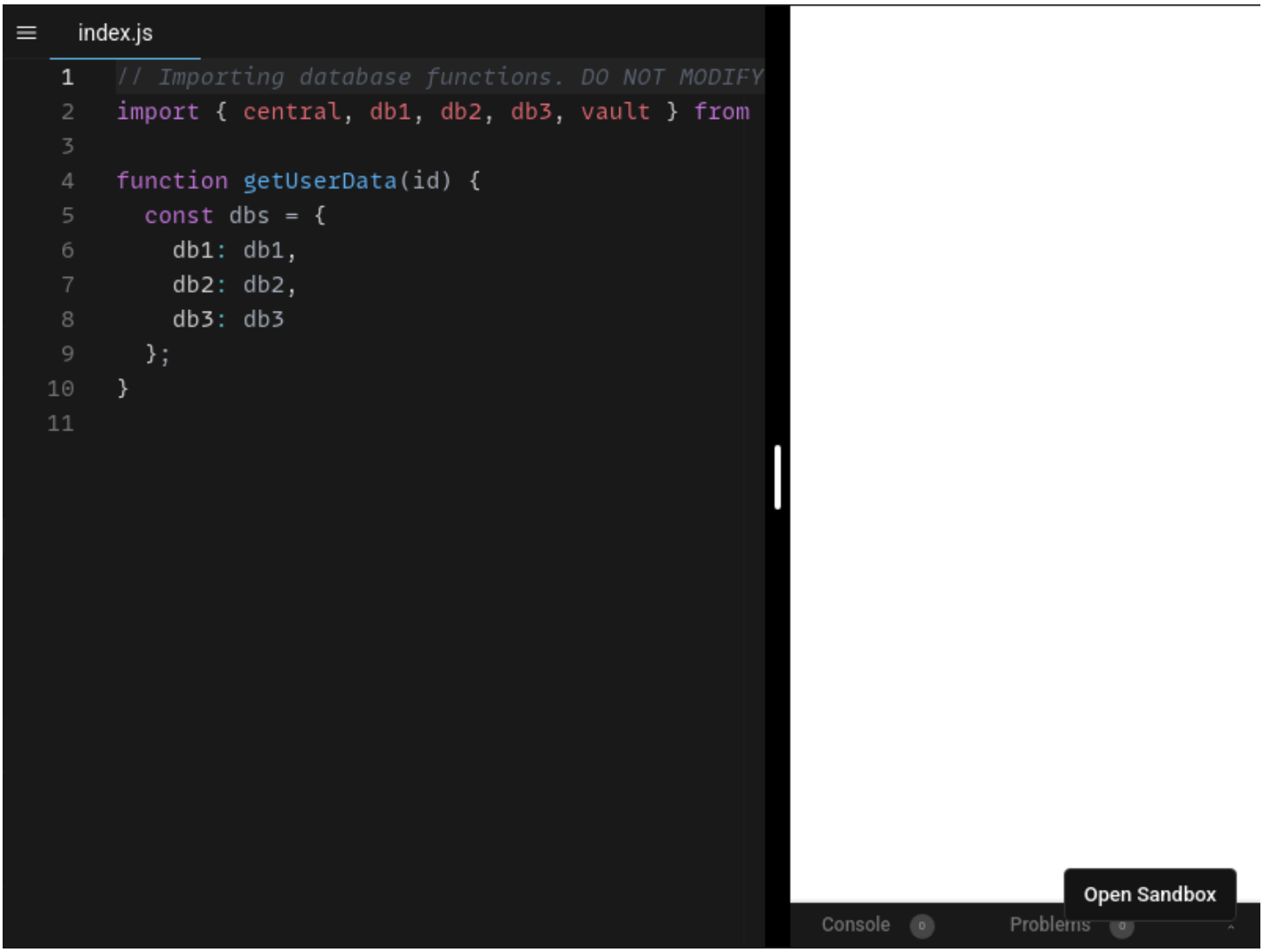
- A GitHub link to your completed project repository, OR
- A CodeSandbox link to the completed project.

Instructions

In order to demonstrate practical application of these concepts, this activity makes use of two concepts that have not yet been discussed: data fetching from external sources and modules and imports.

Each of these concepts will be addressed in the next lessons, and do not need to be understood to complete this assignment. However, it may be beneficial to revisit this assignment after each of those lessons to gain a better understanding of how this code works in the background.

To begin, take a look through the following CodeSandbox:



Within index.js, we have defined some basic data and a function. The first line of code imports some variables, and the function defines a simple dictionary object for later use. The purpose of this object will become clear as you continue reading.

The second file within the example, database.js, contains a fake database system that you will use throughout the assignment. You do not need to understand how this database works to accomplish the tasks below, but it may be worth revisiting this code in the future to enhance your understanding.

Once look over the code, choose one of the options below to get started:

- Download the CodeSandbox starter code above and create a local git repository.
- Fork the CodeSandbox starter code above and continue with the project in CodeSandbox.
  - If you choose to continue in CodeSandbox, remember that you can sync the sandbox with a GitHub repository to continue practicing proper version control!

Commit frequently! Every time something works, you should commit it. Remember, you can always go back to a previous commit if something breaks.

Part 1: The Scenario

You are a developer in a very large corporation that splits its data across multiple databases.

Your job is to assemble this information using a single function that takes an id parameter and returns a Promise that resolves to an object containing specific data.

The object must contain the following information, which will be gathered from the databases:

```
{
    id: number,
    name: string,
    username: string,
    email: string,
    address: {
      street: string,
      suite: string,
      city: string,
      zipcode: string,
      geo: {
        lat: string,
        lng: string
      }
    },
    phone: string,
    website: string,
    company: {
      name: string,
      catchPhrase: string,
      bs: string
    }
}
```

The databases are defined as follows:

central: There are too many users to store in a single database, so the central database identifies which database the users are stored within.

The central database will return a string that identifies which database to access for that particular user's information. You can access the central database like so:

```
const returnedValue = await central(id);
// or
central(id).then((returnedValue) => { ... });
```

While we are pretending that there is a massive database of users, in reality there are only ten unique user values. Accordingly, you should test your function using id values between 1 and 10 (inclusive). Use values outside of this range to test for error cases.

db1, db2, and db3: These databases contain the user's basic information, including username, website, and company. Accessing these databases will return an object with these properties. If one of these databases encounters an error, your function should return a rejected promise indicating which database failed.

You can access these databases like so:

```
const returnedValue = await db1(id);
// or
db1(id).then((returnedValue) => { ... });
```

This is where the dbs object in the starter code can become useful. Using this object, you can access each database directly using the string returned from central by using square bracket notation, e.g.:

- `dbs[valueReturnedFromCentral](id)`

This can help circumvent some conditional logic that would otherwise be required.

vault: The personal data for each user is contained within the vault database since its access and usage is restricted by law. The vault will return an object with the user's name, email, address, and phone, and can be accessed like so:

```
const returnedValue = await vault(id);
// or
vault(id).then((returnedValue) => { ... });
```

Part 2: The Implementation

Your task is to assemble this information using a single function that takes an id parameter and returns a Promise that resolves to an object containing specific data associated with the user with the given id, as described above.

To accomplish this, you may choose to use either Promise chaining via then() statements, or use async/await syntax. Either is a valid approach. As an added challenge, attempt to refactor your code into the opposite solution if you have enough time, and save both versions.

As an additional requirement, note that each database request takes 100ms to respond. However, your function must complete in 200ms or less. Since there are three different databases, you must query; one might assume that the minimum time to do so would be 300ms, but that is not the case.

Remember that asynchronous code is intended to run in parallel – two requests can occur simultaneously. Make use of Promise.all to handle requests concurrently where applicable. Promises only need to be sequential if they depend on the previous Promise's result!

Using this technique in application development can significantly increase the speed at which tasks are accomplished, and improve the overall user experience.

When complete, test your code by passing it many different values for id, including:

- Valid numbers – 1 through 10 (inclusive).
- Invalid numbers – less than 1 or higher than 10.
- Invalid data types – strings, Booleans, etc.