**Assignment 5 – Let’s get hooked**

Q1: What is the difference between Named Export, Default export and **\* as** export?

The export declaration is used to export values (variable, function, object anything) from a JavaScript module. Exported values can then be imported into other programs by either using the import declaration or dynamic import.

Every module can have two different types of exports- named export and default export.

Named export: You can have multiple named exports. Then import the exports surrounded by {} curly braces.

export const Component1 = ()=>{…};

const Component2 = function() {… }

export {Component3}

function Component3(){..}

//module.exports = {Component2}//If one value, avoid the {}

import { Component1, Component2,Component3 } from ”yourFile";

  Component1();

  Component2();

Component3();

Note: Export declarations are not subject to temporal dead zone rules. You can declare the module export ‘Component3’ before the name itself is declared.

When importing this module**, named exports must be referred to by the exact same name,** optionally renaming it with **as**

export let Component4 = ()=>{}

import {Component4 as cmp4} from “yourFile”

Default export: You can have only one default export. Various ways to export.

let MyCard1 = () =>{}

export {MyCard1 as default}

export default MyCard1;

export default MyCard = () =>{}

import MyCard1 from “yourFile”;//one module can have only one default export

import MyCard from “yourFile2”

As a special case, functions and classes are exported as declarations, not expressions, and these declarations can be anonymous. This means functions will be hoisted. Refer MDN

export default function(){ }

export default class{…}

import myFunc from “file1”;

import myClass from “file2”

**Default export can be imported with any name.**

Using export … from: A module can act as a re-exporter(also know as aggregator) to export values from other modules. Instead of writing separate import/export statements, use “**export … from**”

Let's take an example where we have the following hierarchy:

File1.js: exporting myFunction and myVariable as default

File2.js: exporting MyClass

parent.js: acting as an aggregator (and doing nothing else)

top level module: consuming the exports of parentModule.js

let myVariable = 3;

export { myVariable as default };

export const myFunction = () => console.log("Hello world");

class myClass {

  constructor(x) {

    this.x = x;

  }

}

export { myClass };

export { myClass } from "./file2.mjs";

export { myFunction, default as myVariable } from "./file1.mjs";

//export { myFunc, default } from "./file1"; same as line2 , as can be omitted

import { myClass, myFunction, myVariable } from "./parent.mjs";

myFunction(); //Hello World

console.log(new myClass(10));// myClass { x: 10 }

console.log(myVariable); //3

Using \* as export: When you have multiple exports, you can reexport using export \* as:

In file1.js

export let myVariable = 3;

export const myFunction = () => console.log("Hello world");

export function MyComponent() {

  return "Learning is constant";

}

In file2.js – reexport

export \* as myExports from "./file3.mjs";

In file3.js

import { myExports } from "./parent.mjs";

myExports.myFunction();

console.log(myExports.MyComponent());

You can omit file2.js in the above example by using

import \* as myExports from "./file3.mjs";

console.log(myExports.myVariable);//3

myExports.myFunction();//Hello world

console.log(myExports.MyComponent());//Learning is constant

Note: import \* as does not import default exports – value “undefined” . No error

Q2: What is the importance of config.js file?

Config files are commonly used in software development to store configuration parameters that control the behavior of a program or application.

Uses :1 To store static info that is used at multiple places in the application like initial data, image links etc.

2. Application settings like port, database connection string etc .

Stored as PropertyName = Value

Values can be exported and then imported in the request module.

Q3: What are React Hooks?

React Hooks are special utility functions provided by React that lets you “Hook into” React features. Some rules to remember:

1. Only Functional component can have hooks. Not allowed in Class Components. Do not use hooks in normal JavaScript functions also.
2. Don’t call hooks inside loops, conditions, or nested functions.
3. Hooks can only be called at the top level of React function before any early returns. By following this rule, you ensure that the hook is called in the same order each time the component renders. You “use” React features at the top of your component like how you “import” modules at the top of your file.

Q4: Why do we need a useState Hook?

The components that we see on a page serve a purpose and change when there is an interaction. For example, clicking on the Search button should search for restaurants based on the search criteria entered in the input field. Components need to remember the current input value, the current image etc. In React, this kind of component specific memory is known as State.

The React hook useState allows functional components to manage state. It provides a way to declare State Variable(variable to store and manage dynamic data) and setter function to update the state variable and trigger React to render the component again.

useState() returns an array with exactly 2 items. Array destructuring is used to read these 2 items into state(state variable) and setState(setter function).

initialValue can be of any type->number, string, Boolean, array,object,empty string

import { useState } from 'react';

const [state, setState] = useState(initialValue);