Assignment 06 - Exploring the world

Owner	 Pankaj Kumar

1: What is `Microservice`?

`Microservice` - also known as the microservice architecture - is an architectural method that relies on a series of `Independently deployable services.` These services have their own business logic and database with a specific goal. `Updating, testing, deployment, and scaling` occur within each service. Microservices decouple major business, domain-specific concerns into separate, independent code bases.

- Benefits of Microservices:
  - Easier to Test
  - Flexible Scaling
  - Easy Deployment
  - Technological Freedom
  - Reusable Code
  - Resilience

2: What is `Monolith architecture`?

A `Monolith architecture` is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications. A monolithic architecture is `a singular, large computing network with` one code base `that couples all of the business concerns together`. To make a change or a small change in it (We need to deploy the hole project init.For instance we changed a button we use deploy our Whole project.) to this sort of application requires `updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface.` This makes updates restrictive and time-consuming.

Means we are not dividing software into small, well-defined modules, we use every services like, database, server or a UI of the application, in one Application file.

- **Benefits of “Monolith architecture”:**

1. **Easy deployment:** One executable file or directory makes deployment easier.
2. **Development:** When an application is built with one code base, it is easier to develop.
3. **Performance** In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
4. **Simplified testing:** Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application. **Easy debugging** – With all code located in one place, it’s easier to follow a request and find an issue.

3: What is the `difference` between `Monolith and Microservice?

Monolith	Microservice
Every service is inside the application	Services are scattered
Single code base	Code base is divided into separated applications
Hard to maintain	Easy to maintain
Deployment takes more time	Deployment is easy

Parameters	Monolith Architecture	Microservices Architecture
Development	When an application is built with one code base, it is easier to develop. This is true for small applications, but when the application takes larger, development becomes slower and complex.	Micro services add more complexity compared to monolith arch. If development sprawl isn’t properly managed, it results in slower development speed and poor operational performance.

Parameters	Monolith Architecture	Microservices Architecture
Testing	Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.	Teams can experiment with new features and roll back if something doesn’t work. This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services.
Performance	In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.	Though performance could be an issue in microservices, it could be over come by various performance optimisation techniques.
Debugging	With all code located in one place, it’s easier to follow a request and find an issue.	Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.
Scalability	You can’t scale individual components.	If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure.
Relaibility	If there’s an error in any module, it could affect the entire application’s availability.	You can deploy changes for a specific service, without the threat of bringing down the entire application.
Tech Adoption	Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.	Any new tech changes can eaily be adopted as an independent service.
Deployment	One executable file or directory makes deployment easier. But, a small change to a monolithic application requires the redeployment of the entire monolith.	Microservices make it easier for teams to update code and accelerate release cycles with continuous integration and continuous delivery (CI/CD).
Agility	There is no agility in monolith.	Promote agile ways of working with small teams that deploy frequently.

4: Why do we need `useEffect Hook`?

`useEffect Hook` is javascript function provided by `react`. The useEffect Hook allows you to `eliminate side effects` in your components. Some examples of side effects are: `fetching API data`, `directly updating the DOM`, and `setting up subscriptions or timers`, etc can be lead to unwarranted side-effects.
useEffect accepts `two arguments`, first is a `callback function` and a `dependency array`. The second argument is optional.

```
Syntax
`useEffect(() => {}, [])`
```

The `() => {}` is callback function and `[]` is called a empty dependency array.
If anything that we pass (suppose currentState) inside the `[]` it trigger the callback function and changes the state of the application.

```
useEffect(() => {
    setCurrentState("true");
}, [currentState])
```

If we do not pass empty dependency array then the useEffect runs everytime when the UI is rendered.

```
useEffect(() => {})
```

5: What is `Optional Chaining`?

`Optional Chaining` (`?.`) operator accesses an object's property or calls a function. If the object accessed or function called is `undefined or null` , it returns `undefined` instead of throwing an error.

`Optional Chaining` (`?.`) is good way of accessing the object keys, it prevents the application from being crashed if the key that we are trying to access is not present. If the key is not present then instead of a throwing key error, it returns `undefined`.

6: What is `Shimmer UI`?

A shimmer screen is a version of the UI that doesn't contain actual content. Instead, it mimics the page's layout by showing its elements in a shape similar to the actual

content as it is loading and becoming available (i.e. when network latency allows).

A shimmer screen is essentially a wireframe of the page, with placeholder boxes for text and images.

7: What is the `difference` between `JS expression and JS statement`?

`JS Expression` - A js expression returns a value. A number, string,ternary conditions (return true or false), math opertions and array map method (returns new array) are all examples of js expression.

```
1. "Pankaj" ----> String is a valid expression

2. 1234 ----> Number is a valid expression

3. (isLoggedIn) ? "Logout" : "Login"  ----> Ternary operator returning value is
   a valid expression

4. [1,2,3,4].map(num => num*2)----> Array map function is a valid expression
    which returns a new array after transformation

5. (1+2+3) ---> Math operation is a valid expression
```

`JS Statement` - A js statement just executes/performs an action but does not return/produce a value. A variable assignment, if condition (with no return) and for loops are examples of js statement.

```
1. console.log("This is a js statement") ----> This does not return any value,
   just prints the content on screen.

2. let name = "Pankaj"; ----> Variable assigment is a statement

3. if(true){
    console.log("true");
  } else {
    console.log("true");
  }   ----> This does not return any value

4. for(let i=0; i< 5; i++) {
    arr.push(i);
  }
```

Having said that, we can't put any js code inside {} in jsx of React. Only `js expressions` can be enclosed within {} of jsx.

If we want to use `JS expression` in JSX, we have to wrap in `{/* expression slot */}` and if we want to use `JS statement` in JSX, we have to wrap in `{(/* statement slot */)}`.

8: What is `Conditional Rendering`? explain with a code example.

A conditional rendering is a way of rendering components based on the a state. If the condition is true for a component, then it gets rendered; otherwise, the other component is rendered.

For example : We load a shimmer UI before our component is loaded completely. We can create a state variable that will keep the value of our current application state. i.e.

```
const [isLoaded, setIsLoaded] = useState(false)
```

In the above example, we are creating a state variable that is initially set to false, since our data has not been loaded in our application yet.

Untill our data is loaded completely we can show a shimmer UI to the user and when our data gets loaded we can render the data on the page. The conditional rendering is done via a ternery operator `?:` for example :

```
isLoaded ? <Body /> : <Shimmer />
```

In the above expression, if `isLoaded` is set to `false` then, `Shimmer` component will be loaded , when the data loading is completed, the `Body` component will be rendered.

- `&&` operator : if the condition is true, display the right-side code else display nothing.

  { errorMsg && {errorMsg} }

9: What is `CORS`?

CORS stands for *Cross Origin Resource Sharing* , It is a HTTP-header based machanism that allows a server to indicate any origin other that it's own. We can create requests to other domains or ports to get the data from our browser.

(OR)

`Cross-Origin Resource Sharing (CORS)` is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "pre-flight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that pre-flight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

10: What is `async and await`?

Async/await are keywords to make a normal function behave like a asynchornous funtion.

`async` function always returns a promise, any values are automatically wrapped inside a resolved promise.

`await: Await function is used to wait for the promise to be settled. It could be used within the` ‘async` block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

For example : Let's try to write a function getRestaurants() to fetch restaurant data from a public API.

First, let's try to write it with `Promise chaining` : fetch(url) returns a promise (resolve or reject), which can be consumed by the `then` (success) handler or `catch` (error) handler

```
function getRestaurants() {
  fetch(url).then((data)=>{data.json()})
    .then((json)=>{
    console.log(json);
  }).catch((err)=>{
    console.log(err);
  })
}
```

Using `async` and `await` : await waits until fetch(url) returns a promise with the data and headers which again needs to be resolved using .json() method to get the data. If any of promise inside try block is rejected, the control jumps to catch block.

```
async function getRestaurants() {
  try {
    const data = await fetch(url);
    const json = await data.json();
    console.log(json);
  } catch(err) {
    console.log(err);
  }
}
```

11: What is the use of `const json = await data.json()`; in `getRestaurants()`?

The `data` object, returned by the `await fetch()`, is a generic placeholder for multiple data formats.
so we can extract the `JSON object` from a `fetch` response by using `await data.json()`.
`data.json()` is a method on the data object that lets you extract a `JSON object` from the data or response. The method returns a promise because we have used `await` keyword.
so `data.json()` returns a promise resolved to a `JSON object`.