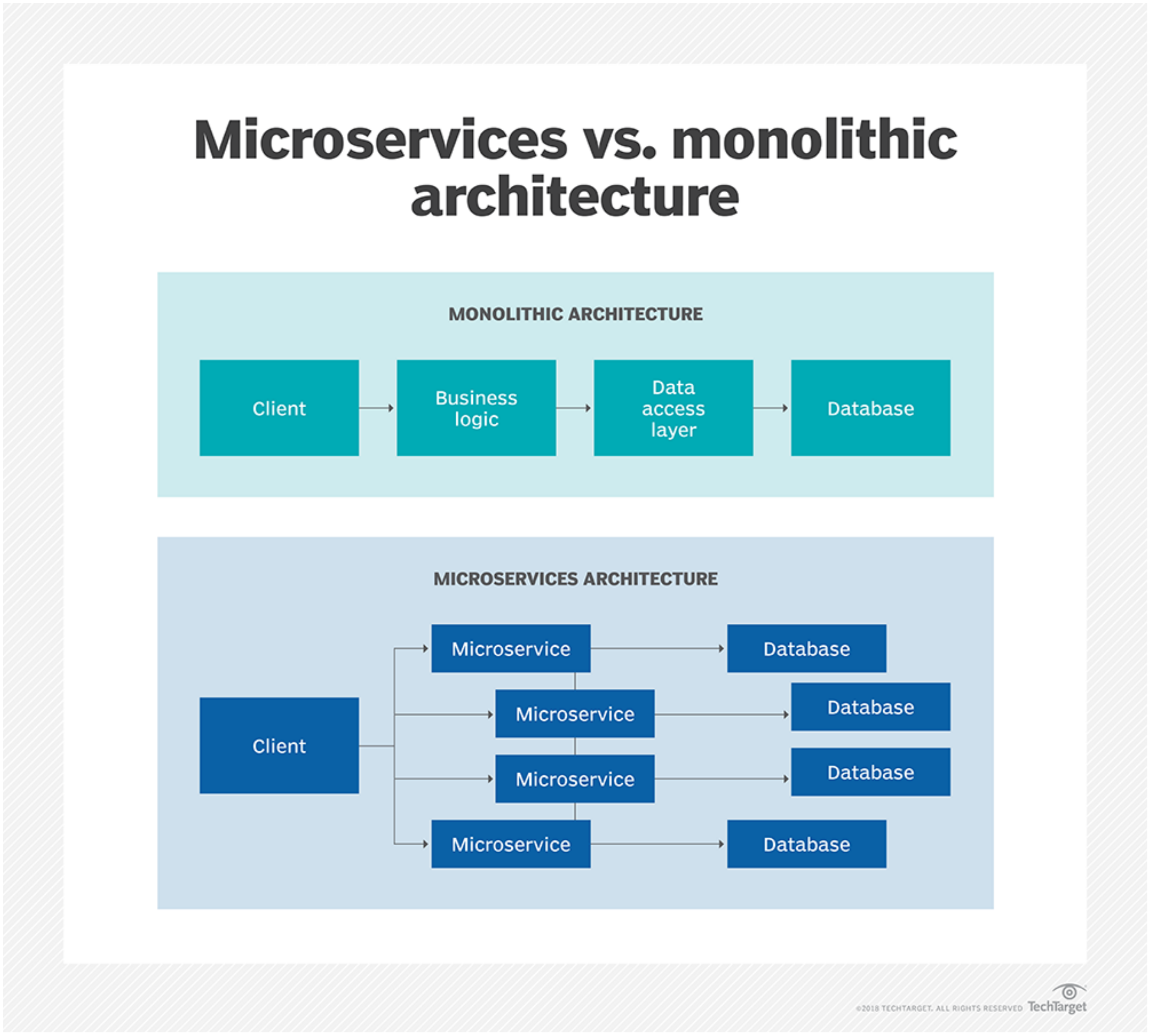
1. What are Microservices?

- Microservices architecture is an approach to application development in which a large application is built from modular components or services. Each module supports a specific task or business goal and uses a simple, well-defined interface, such as an application programming interface (API), to communicate with other sets of services.
- In a microservices architecture, an application is divided into services. Each service runs a unique process and usually manages its own **database**. A service can generate alerts, **log data**, support user interfaces **(UIs)**, handle user identification or authentication and perform various other tasks.
- Each service can be isolated, rebuilt, redeployed and managed independently.



2. What is Monolith architecture?

- Monolithic architecture is a software architecture where all components of an application are combined into a single, self-contained unit. This unit typically includes the user interface, business logic, and data storage components all in one place.

Advantage :

- Easy to develop and deploy, as all components are combined into a single unit
- Simple to understand and maintain, as the entire application is in one codebase
- Can handle high levels of traffic and transactions due to its ability to handle many requests at once

Disadvantages:

- It can become difficult to scale as the application grows
- Can be difficult to update or make changes to individual components without affecting the entire application
- Testing and debugging can be more challenging as all components are tightly coupled
- Hard to deploy new features and fix bugs in production without downtime

3. What is the difference between monolithic and microservice?

- In a monolithic architecture, all of the code is in one principal executable file. This can be tougher to troubleshoot, test and update, because a problem in a code base could be located anywhere within the software. More and longer testing is required due to the amount of monolithic code involved. Also, any small change or update in a monolithic application requires an organization to build and deploy an entirely new version of the application. Altogether, monolithic application development entails significant planning, preparation, time, and expense.
- Microservices architectures offer faster software development and deployment compared to monolithic software architecture, which enhances business agility. Microservices can make it easier to test and deploy changes. Separation of services improves fault isolation -- if a problem occurs in the software, the faulty service can be isolated,

remediated, tested, and redeployed without the need to regression test the entire application as with traditional monolithic application architectures.
- Monolithic applications can be difficult to scale. When a monolithic application reaches a capacity limitation, such as data throughput or some other bottleneck, the only practical option is to deploy another complete iteration of the entire monolithic application and manage traffic between the instances using load balancers.

4. Why do we need a useEffect Hook?

- By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API.

5. What is Optional Chaining in JS?

- The **optional chaining '?.'** is an error-proof way to access nested object properties, even if an intermediate property doesn't exist. It was recently introduced by ECMA International, Technical Committee 39 – ECMAScript which was authored by Claude Pache, Gabriel Isenberg, Daniel Rosenwasser, and Dustin Savery. It works similarly to Chaining '.' except that it does not report the error, instead it returns a value that is undefined. It also works with function calls when we try to make a call to a method that may not exist.

```
const user = {
  dog: {
    name: "Alex"
  }
};

console.log(user.cat?.name);
  //undefined
console.log(user.dog?.name); //Alex
console.log(user.cat.name);
```

6. What is shimmer UI?

- A shimmer UI **resembles the page's actual UI**, so users will understand how quickly the web or mobile app will load even before the content has shown up. It gives people an idea of what's about to come and what's happening (it's currently loading) when a page full of content/data takes more than 3 - 5 seconds to load.

7. What is the difference between JS Expressions and JS statement?

JS Expression

- let a = 10   // this is js expression

JS Statement

- if(a=0){   // this if block is js statement

  }

8. What is cors ?

- The CORS mechanism supports secure cross-origin requests and data transfers between browsers and servers. Modern browsers use CORS in APIs such as XMLHttpRequest or Fetch to mitigate the risks of cross-origin HTTP requests.

9. What is async and await?

- There's a special syntax to work with promises in a more comfortable fashion, called "async/await". It's surprisingly easy to understand and use.
- Let's start with the async keyword. It can be placed before a function, like this:
- async function f() {
-   return 1;
- }
- The word **"async"** before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.
- The keyword **await** makes JavaScript wait until that promise settles and returns its result.

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

10. What is the use of `const json = await data.json();` in getRestraurants()?

- The `response` object, returned by the `await fetch()`, is a generic placeholder for multiple data formats.
- For example, you can extract the JSON object from a fetch response:

```
async function fetchMoviesJSON() {
 const response = await fetch('/movies');
 const movies = await response.json();
 return movies;
}
fetchMoviesJSON().then(movies => {
 movies; // fetched movies
});
```

- response.json() is a method on the Response object that lets you extract a JSON object from the response. The method returns a promise, so you have to wait for the JSON: await response.json().
- The Response object offers a lot of useful methods (all returning promises):
- response.json() returns a promise resolved to a JSON object
- response.text() returns a promise resolved to raw text
- response.formData() returns a promise resolved to FormData
- response.blob() returns a promise resolved to a Blob (a file-like object of raw data)
- response.arrayBuffer()() returns a promise resolved to an ArryBuffer (raw generic binary data)