1. If you had to pick only 5 libraries as dependencies for your Angular app, which ones would you pick?

- RxJS
- NgRx/Redux
- Angular Material
- Lodash
- ngx-translate

2. What's your go-to strategy for debugging Angular components?

- Console Logs
- Augury
- Breakpoints
- Change Detection
- Unit Tests

3. Have you ever created a custom RxJS operator? What problem did it solve?

- Custom operator to debounce user input and aggregate them into a single request.

4. What unconventional challenges did you have while developing/architecting Angular apps?

- Circular Dependency Issues
- Overcomplicated State Management
- Lazy-Loading Complexities

5. How do you manage API type consistency between frontend, mobile app, and backend?

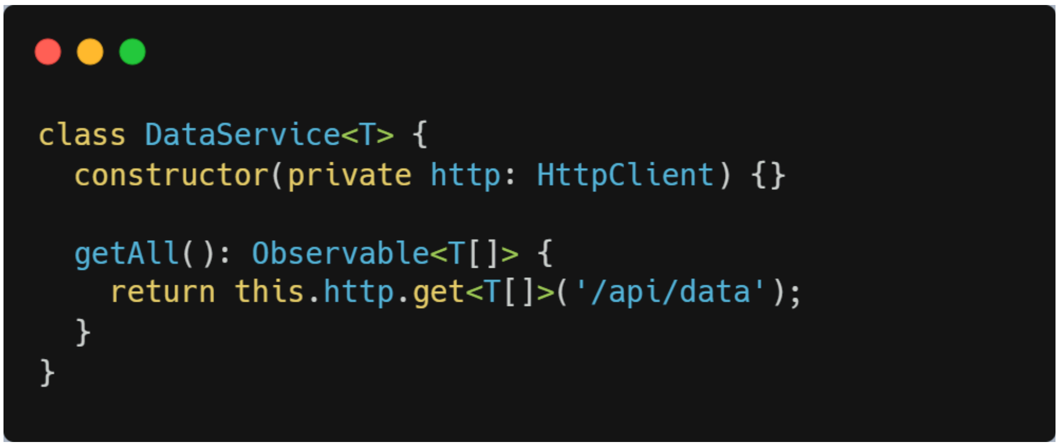
- TypeScript Interfaces
- OpenAPI/Swagger
- Backend Contracts

6. What's your approach to optimizing the performance of large Angular components?

- OnPush Change Detection
- Lazy-Load Child Components
- Memoization
- Avoid Complex Logic in Templates
- TrackBy for ngFor

7. How have you used TypeScript generics to simplify handling complex interfaces in Angular?

Example: `DataService<T>` to handle HTTP requests generically.



```
class DataService<T> {
  constructor(private http: HttpClient) {}

  getAll(): Observable<T[]> {
    return this.http.get<T[]>('/api/data');
  }
}
```

8. Can you use ::ng-deep?

Yes, but it is deprecated. Best to use global styles.

9. How do you ensure reusability in your SCSS for large-scale Angular projects?

- SCSS Variables and Mixins
- Component-Specific Styles
- BEM Methodology

10. What is your strategy for refactoring a bloated Angular component?

- Break Down Logic
- Use Child Components
- State Management
- Remove Unused Code
- Performance Profiling

11. What was the most complex form you worked with?

Multi-step dynamic form with conditional fields, validation, and dependent dropdowns.

12. How do you implement validation for complex forms?

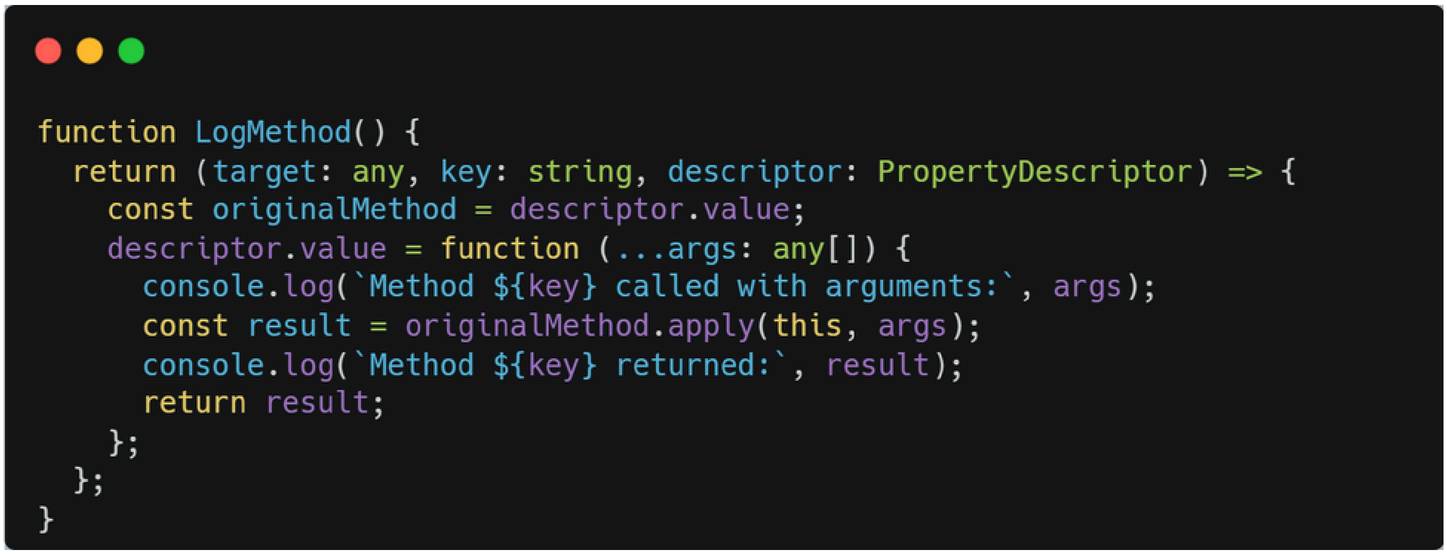
- Custom Validators
- Cross-Field Validation
- Async Validators
- Nested FormGroups

13. Signals, Observables, Promises - tell me all about differences and when to use which.

- Promises for one-time events.
- Observables for streams.
- Signals for reactive state management.

14. Have you implemented custom decorators? If so, what was the use case?

Example: Custom logging decorator to log method arguments and return values.



```
function LogMethod() {
  return (target: any, key: string, descriptor: PropertyDescriptor) => {
    const originalMethod = descriptor.value;
    descriptor.value = function (...args: any[]) {
      console.log(`Method ${key} called with arguments:`, args);
      const result = originalMethod.apply(this, args);
      console.log(`Method ${key} returned:`, result);
      return result;
    };
  };
}
```

15. What kind of states do we have in the app and how do you manage them?

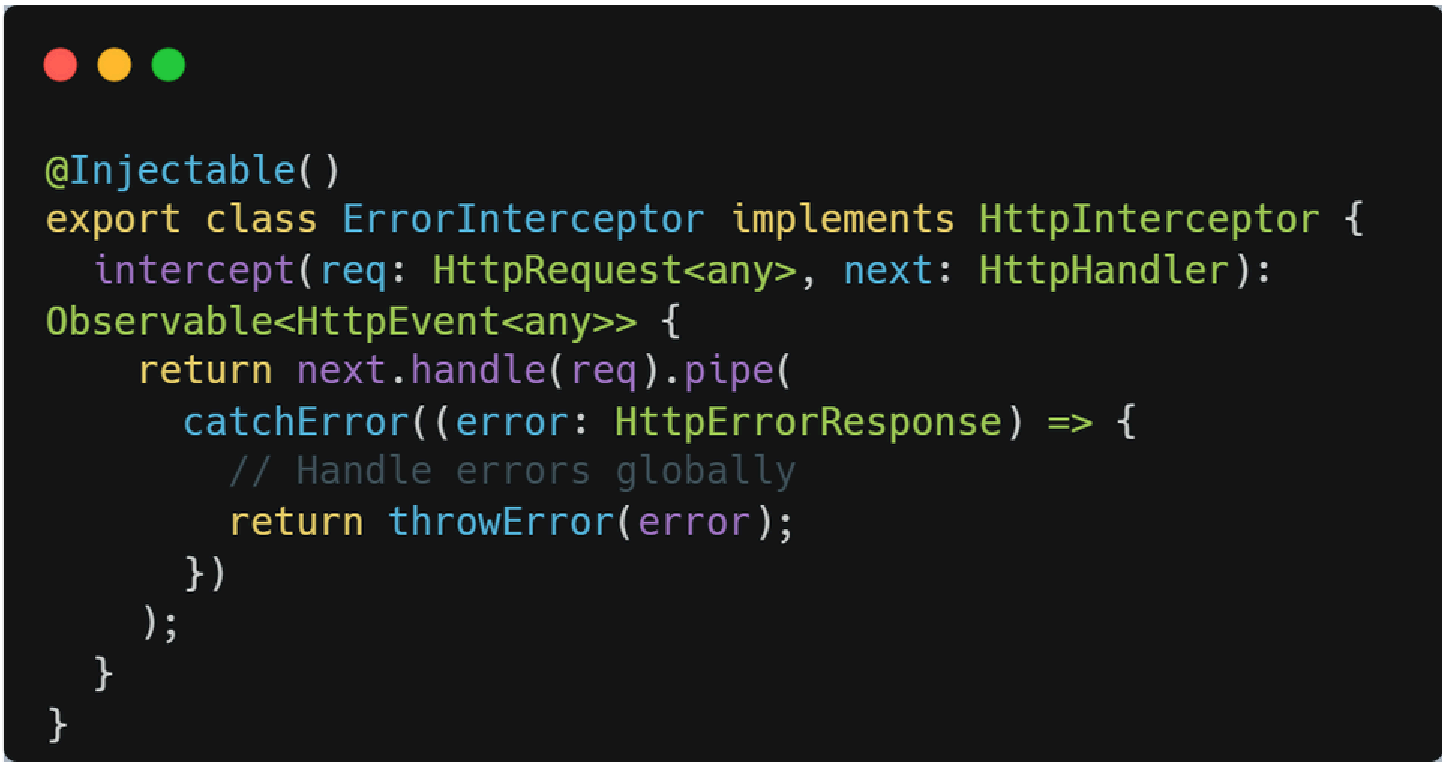
- UI State
- Global State
- Router State

16. What's your approach to lazy-loading modules in Angular?

Use loadChildren to lazy-load feature modules.

17. How do you handle HTTP errors globally?

Implement an HttpInterceptor to catch and handle errors globally.



```
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        // Handle errors globally
        return throwError(error);
      })
    );
  }
}
```

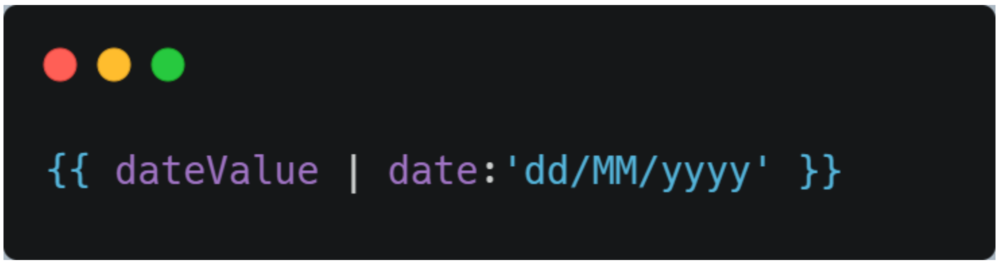
Here are the answers to the provided Angular-related questions

Why do we use `ngOnDestroy`?
The `ngOnDestroy` lifecycle hook is used in Angular to perform cleanup tasks when a component or directive is destroyed. This includes unsubscribing from observables, detaching event handlers, or releasing other resources to prevent memory leaks.

What if we do not unsubscribe from an observable?
Failing to unsubscribe from an observable can lead to memory leaks, as the subscription remains active even when the component is destroyed. This can cause unintended behavior and degrade application performance.

What are the built-in pipes in Angular, and which modules do they come from?
Angular provides several built-in pipes, such as `DatePipe`, `CurrencyPipe`, `DecimalPipe`, and `PercentPipe`. These pipes are part of the `CommonModule`.

How to modify a date to the expected format in Angular?

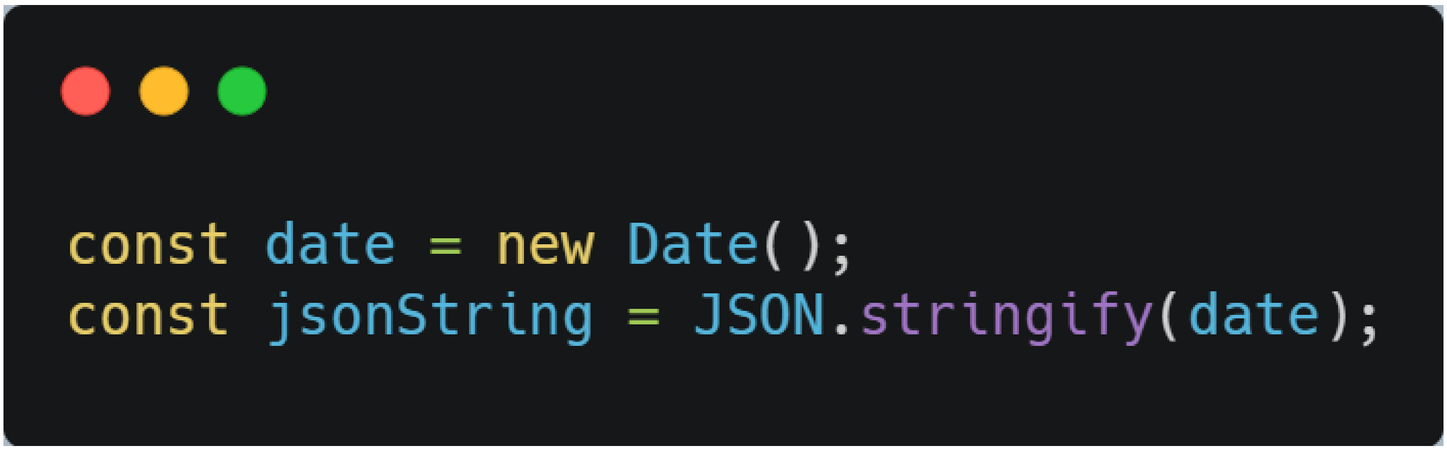


```
{{ dateValue | date:'dd/MM/yyyy' }}
```

To format a date in Angular, you can use the `DatePipe` in your templates:
This formats `dateValue` to display as `dd/MM/yyyy`.

What are other methods to format dates in Angular?
Apart from `DatePipe`, you can use external libraries like [date-fns](https://date-fns.org/) or [moment.js](https://momentjs.com/) for more advanced date formatting and manipulation.

How to send a date in JSON format in an API call?
In TypeScript, you can convert a `Date` object to a JSON string using `JSON.stringify()`:



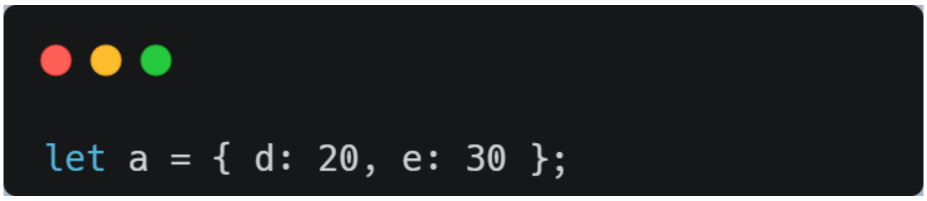
```
const date = new Date();
const jsonString = JSON.stringify(date);
```

This `jsonString` can then be sent in an API call.
Can we create a custom pipe to format dates in Angular?

Yes, you can create a custom pipe by implementing the `PipeTransform` interface. This allows you to define custom formatting logic for dates or other data types.

What is the type of the `Date` object in TypeScript?
In TypeScript, the `Date` object is of type `Date`.

Given `let b = 20; let c = 30; let a = (d: 20, e: 30);`, what is the value of `a`?
The expression `let a = (d: 20, e: 30);` is not standard TypeScript syntax and will result in a syntax error. If you intend to assign the values of `d` and `e` to `a`, you should use object destructuring:



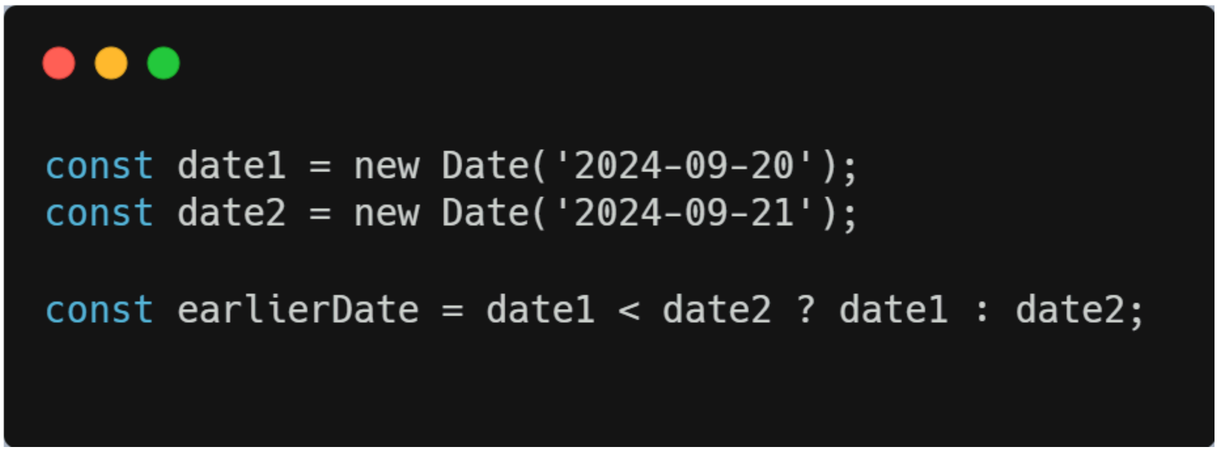
```
let a = { d: 20, e: 30 };
```

Is there any built-in Angular service for date formatting?
Angular provides the `DatePipe` for date formatting in templates. For more advanced formatting in TypeScript, you can use external libraries as mentioned above.

How to create a date picker and validate the selected date in Angular?
You can use Angular Material's `mat-datepicker` component to create a date picker. To validate the selected date, you can use Angular's form validation features, such as reactive forms and custom validators.

How to handle time zones in Angular?
Handling time zones involves converting dates to the desired time zone using JavaScript's `Date` object methods or external libraries. It's important to ensure consistency across the application, especially when dealing with users from different regions.

How to compare two dates and return the earlier one in TypeScript?
You can compare two `Date` objects using the `<` operator:



```
const date1 = new Date('2024-09-20');
const date2 = new Date('2024-09-21');

const earlierDate = date1 < date2 ? date1 : date2;
```

This will assign the earlier of the two dates to `earlierDate`.