Beginner Angular Interview Questions

1. **What is Angular, and how is it different from AngularJS?**
   - **AngularJS** is the original version (Angular 1.x), while **Angular 2+** is a complete rewrite using TypeScript. Angular is faster, modular, has better tooling, and supports mobile development.
2. **Explain the structure of an Angular application.**
   - An Angular application is composed of **Modules** (NgModules), **Components**, **Services**, **Directives**, **Pipes**, and **Routing**.
3. **What are components in Angular? How do you create them?**
   - Components control parts of the UI, created using `ng generate component`. Components include templates (HTML), styles (CSS), and logic (TypeScript).
4. **What is data binding in Angular? Explain its different types.**
   - Data binding allows data synchronization between the view and component logic:
     - **Interpolation**: `{{ variable }}`
     - **Property Binding**: `[property]="expression"`
     - **Event Binding**: `(event)="expression"`
     - **Two-way Binding**: `[(ngModel)]="property"`
5. **What are Angular directives? Explain the types of directives available in Angular.**
   - **Directives** add behavior to elements. Types:
     - **Structural Directives**: Modify the DOM structure (e.g., `*ngIf`, `*ngFor`).
     - **Attribute Directives**: Change appearance/behavior of elements (e.g., `ngClass`, `ngStyle`).
6. **What is Angular Router? How do you configure routing in Angular?**
   - **Angular Router** enables navigation between different views. Configure with `RouterModule` and a `routes` array, and render using `<router-outlet>`.
7. **What are Angular services, and how do you create and inject a service?**
   - Services are used to encapsulate reusable logic. Create them with `ng generate service` and inject them using Angular's **Dependency Injection** system (`@Injectable()`).
8. **What are pipes in Angular? How are custom pipes created?**
   - **Pipes** transform displayed data. Built-in pipes: `DatePipe`, `CurrencyPipe`, etc. Custom pipes are created using the `@Pipe` decorator.
9. **What are observables in Angular? How are they different from promises?**
   - Observables are asynchronous streams of data, allowing multiple values over time (using RxJS). **Promises** handle single values, while **observables** are more flexible and cancellable.
10. **What is lazy loading in Angular? How do you implement lazy loading of modules?**
    - **Lazy loading** improves performance by loading modules only when required. Implement it using `loadChildren` in the routing configuration.
11. **What is Angular's ngOnInit lifecycle hook?**

  - **ngOnInit** is called after component initialization, used to perform logic such as fetching data once the component has been initialized.

12. **What is the difference between ViewChild and ContentChild?**
    - **ViewChild** accesses a child component or element in the component's template, while **ContentChild** accesses projected content inside a component.
13. **Explain the purpose of Angular forms. What are the differences between template-driven and reactive forms?**
    - Angular forms manage user inputs. **Template-driven forms** are simpler and rely on directives, while **Reactive forms** provide more control and are built programmatically.
14. **How does Angular handle error handling?**
    - Angular uses the **ErrorHandler** class for global error handling. Developers can extend it for custom error handling, e.g., logging errors to a server.
15. **What is Angular's change detection mechanism? How does it work?**
    - Angular's change detection is powered by **Zones**. It tracks asynchronous operations and updates the view when data changes.

Advanced Angular Interview Questions

1. **What is Angular Ivy? How does it improve performance and bundle size in Angular?**
    - **Angular Ivy** is the default rendering engine in Angular that reduces bundle sizes through better tree-shaking, improves AOT compilation, and speeds up rendering.
2. **What are Angular modules (NgModule)? How do you organize an Angular app using feature modules and shared modules?**
    - **NgModules** organize Angular applications. Feature modules group functionality, while shared modules export reusable components/services. Core modules handle global services.
3. **How do you implement Angular NgRx for state management? Explain its architecture.**
    - **NgRx** follows a Redux-like architecture with **actions**, **reducers**, **effects**, and **selectors** to manage the application's state.
4. **What are dynamic components in Angular? How do you create and load dynamic components?**
    - Dynamic components are instantiated at runtime using `ComponentFactoryResolver` and loaded with `ViewContainerRef`.
5. **Explain Angular's Dependency Injection (DI) mechanism in depth.**
    - Angular's DI system allows services and components to be injected where needed. Providers, multi-providers, and tokens can be used, with injectors working hierarchically.
6. **How do you optimize the performance of an Angular application?**

- Optimizations include **lazy loading**, using `OnPush` change detection, **AOT compilation**, minimizing change detection cycles, and RxJS operators like `takeUntil`.

7. **What are Angular interceptors? How do you use them to modify HTTP requests or responses?**
   - **HttpInterceptors** modify HTTP requests/responses globally, used for logging, authentication, or error handling.
8. **What is a resolver in Angular routing? How do you use it to prefetch data?**
   - Resolvers fetch data before a route is activated. They implement the `Resolve` interface and ensure data is loaded before rendering the route.
9. **Explain how Angular's ViewEncapsulation works. What are the different types?**
   - **ViewEncapsulation** controls style scoping in Angular components:
     - **Emulated** (default): Scoped styles using attributes.
     - **Shadow DOM**: Uses the browser's native Shadow DOM.
     - **None**: No encapsulation; global styles.
10. **How do you handle large datasets in Angular?**
    - Use techniques like **pagination**, **infinite scrolling**, and **virtual scrolling** (Angular CDK) to efficiently handle large datasets and improve performance.
11. **What is the RouterModule.forRoot() vs RouterModule.forChild() in Angular?**
    - `RouterModule.forRoot()` is used to configure routes at the root level, while `RouterModule.forChild()` is used for feature modules.
12. **What is differential loading in Angular, and how does it benefit modern browsers?**
    - **Differential loading** creates modern and legacy bundles. Modern browsers get smaller, optimized ES2015 bundles, improving load times and performance.
13. **What is Angular Universal? How do you use it for server-side rendering (SSR)?**
    - **Angular Universal** enables server-side rendering (SSR), improving SEO and performance by rendering the application on the server before sending it to the browser.
14. **How does Angular handle cross-component communication?**
    - Cross-component communication is achieved through:
      - **@Input() and @Output() decorators**
      - **Shared services with RxJS Subjects**
      - **EventEmitters**
      - **State management libraries like NgRx**
15. **Explain the usage of ng-template, ng-container, and ng-content.**
    - These elements are used for advanced templating:
      - **ng-template**: Defines a template that is not rendered immediately.
      - **ng-container**: A logical container that doesn't render any DOM elements.
      - **ng-content**: Projects content into a component from the parent.
16. **How do you manage memory leaks in Angular applications?**
    - Memory leaks are managed by:
      - Unsubscribing from observables (`takeUntil`, `ngOnDestroy`).
      - Using **AsyncPipe**.

        - Cleaning up event listeners.

17. **Explain the role of AOT (Ahead-of-Time) compilation in Angular. How does it affect performance?**
    - **AOT** compiles Angular templates during the build process, reducing runtime errors and producing smaller, faster-loading bundles.
18. **What is a custom structural directive in Angular? How do you create one?**
    - A **custom structural directive** is created using `@Directive` and manipulates the DOM with `TemplateRef` and `ViewContainerRef`.
19. **How do you handle routing guards in Angular?**
    - Routing guards like **CanActivate**, **CanDeactivate**, **Resolve**, and **CanLoad** protect routes and control access based on conditions like authentication.
20. **What are zone.js and its significance in Angular?**
    - **Zone.js** helps Angular detect asynchronous operations and trigger change detection automatically. Running code outside of the Angular zone can improve performance (`ngZone.runOutsideAngular()`).
21. **How do you handle complex forms in Angular?**
    - Complex forms are managed with **FormGroup**, **FormArray**, custom validators, and dynamic form controls using **Reactive Forms**.
22. **What are custom validators in Angular, and how do you implement them?**
    - **Custom validators** are created by implementing `ValidatorFn` for synchronous and `AsyncValidatorFn` for asynchronous validators, used to add custom validation logic.
23. **How does Angular handle routing animations?**
    - **Route-based animations** are implemented using Angular's `@angular/animations` module, defining animations in routing transitions and states.
24. **Explain the ControlValueAccessor interface in Angular. How do you use it to create custom form controls?**
    - **ControlValueAccessor** bridges custom form controls and Angular's forms API. Implement this interface to synchronize custom control values with form inputs.
25. **What is NgUpgrade? How do you migrate an AngularJS application to Angular?**
    - **NgUpgrade** helps migrate AngularJS apps to Angular by supporting both frameworks in a hybrid mode, enabling a gradual upgrade without a complete rewrite.
26. **What are Angular preloading strategies, and how do you implement them?**
    - Preloading strategies load lazy-loaded modules in the background. Angular provides `NoPreloading`, `PreloadAllModules`, or custom strategies.
27. **What is Renderer2 in Angular, and how does it differ from direct DOM manipulation?**
    - **Renderer2** provides a platform-agnostic way to manipulate the DOM, making it safer to use than direct DOM manipulation, especially in environments like Web Workers or SSR.
28. **Explain the difference between OnPush and Default change detection strategies.**

    - **OnPush** change detection triggers checks only when input properties change, improving performance. **Default** checks all component bindings on each change detection cycle.
29. **What is ElementRef in Angular, and why should you avoid using it directly?**
    - **ElementRef** provides direct access to DOM elements but can lead to security risks like XSS. Use **Renderer2** for safe, platform-agnostic DOM manipulation.
30. **Explain how the AsyncPipe works. What are its advantages?**
    - **AsyncPipe** automatically subscribes to observables or promises and unsubscribes when the component is destroyed, simplifying the code and avoiding memory leaks.
31. **What is a service worker in Angular? How do you implement it for PWA support?**
    - **Service workers** enable Progressive Web Apps (PWA) with offline capabilities. Implement using `@angular/service-worker`, which handles caching and background synchronization.
32. **What is Angular CLI's ng build --prod flag? How does it optimize a production build?**
    - The `ng build --prod` flag enables optimizations such as AOT, minification, dead code elimination, and differential loading for smaller and faster production builds.
33. **Explain dependency injection (DI) with multiple providers in Angular.**
    - Multiple providers can be registered in Angular using **multi: true**, allowing multiple implementations for the same token, used in scenarios like logging.
34. **How does HttpClient handle interceptors? Can you chain multiple interceptors?**
    - **HttpClient** allows multiple interceptors to be chained, modifying requests and responses for logging, authentication, or error handling.
35. **What is a BehaviorSubject, and how does it differ from a regular Subject in RxJS?**
    - **BehaviorSubject** stores the latest emitted value and emits it to new subscribers, while a regular **Subject** does not store previous values.
36. **How do you optimize Angular applications for better performance in a large-scale enterprise application?**
    - Optimization techniques:
        - **Lazy loading**
        - **OnPush** change detection
        - Using `trackBy` with `ngFor`
        - Running code outside Angular's zone with `ngZone.runOutsideAngular()`
        - Efficient RxJS operators like `takeUntil`
        - Tree-shaking and AOT compilation
37. **What are the major differences between AOT (Ahead-of-Time) and JIT (Just-in-Time) compilation in Angular?**
    - **AOT** compiles templates during the build process, providing faster rendering, smaller bundles, and early error detection. **JIT** compiles templates in the browser, which is slower.

38. **What is DeferLoading in Angular? How does it improve performance?**
    - **DeferLoading** delays the loading of non-critical content until necessary, improving the perceived performance of the application.
39. **How do you integrate Web Workers into an Angular application?**
    - **Web Workers** offload heavy computations to a separate thread. Use Angular CLI to generate Web Workers for long-running tasks without blocking the UI.
40. **Explain how Angular ensures security with the DOM via sanitization.**
    - Angular protects against XSS by sanitizing content in templates, and **DomSanitizer** can be used for manually sanitizing dynamic content.
41. **What are Zones in Angular, and how do they affect performance? How can you run code outside Angular's zone?**
    - **Zones** track async operations and run change detection. Running code outside Angular's zone (via `ngZone.runOutsideAngular()`) can improve performance in heavy tasks.
42. **How would you approach testing in an Angular application?**
    - Use **Karma** and **Jasmine** for unit testing, **TestBed** for component testing, and **Protractor** or **Cypress** for end-to-end testing.
43. **What is differential loading, and how does it work in Angular?**
    - **Differential loading** builds modern (ES2015+) and legacy (ES5) bundles, optimizing performance for modern browsers while supporting older ones.
44. **Explain how to handle multiple environments in Angular.**
    - Angular manages environments using the `src/environments` directory. Configuration is set in `angular.json`, allowing different variables (e.g., API endpoints) for dev, staging, and prod.
45. **How does Angular handle internationalization (i18n) and localization (l10n)?**
    - Angular provides built-in **i18n** tools for handling different languages. You can use **@angular/localize** to configure translation files and manage dynamic content translations.
46. **How do you use RouterModule.forRoot() to configure a guard for protecting a child route in Angular?**
    - Configure a route guard (e.g., `CanActivate`) in `RouterModule.forRoot()` to protect child routes by evaluating user permissions or conditions before allowing navigation.
47. **What are Angular zones, and how do they work with change detection?**
    - **Zone.js** tracks asynchronous operations and ensures Angular's change detection is triggered automatically when necessary.
48. **Explain ng-content and how to use content projection in Angular.**
    - **ng-content** allows you to insert external content into a component's template, providing flexibility for building reusable components (single-slot and multi-slot projection).
49. **How do you manage multiple API calls and combine their results using RxJS operators?**

    - **RxJS** operators like `forkJoin`, `combineLatest`, `zip`, and `mergeMap` help combine multiple API calls and handle asynchronous data efficiently.

50. **How do you handle large file uploads in Angular?**
    - Techniques include chunked uploads, background uploads, progress bars, and error handling using Angular's `HttpClient` and `FormData`.

Additional Questions from the Document

1. **What are the common patterns for handling state management in Angular?**
    - Common patterns include using **services with BehaviorSubject**, **NgRx** (Redux-based), and **Akita**. These manage state across components, where services handle simple state management, and NgRx/Akita manage more complex, reactive state management.
2. **How does Angular handle immutability? How would you ensure that your app follows immutability best practices?**
    - Angular handles immutability by leveraging TypeScript. You can ensure immutability using libraries like **Immutable.js** or manually by using **Object.assign()**, the **spread operator**, or using immutable data structures with reactive programming (RxJS).
3. **What is the role of Injector in Angular? How does it differ from NgModule?**
    - **Injector** is a runtime service that creates and manages dependencies. **NgModule** is a compile-time construct that organizes code into cohesive blocks and can provide services, but **Injector** delivers those services.
4. **How do you handle concurrency in Angular using RxJS?**
    - RxJS operators like **mergeMap**, **switchMap**, and **concatMap** handle concurrency. **mergeMap** runs all tasks in parallel, **switchMap** cancels ongoing tasks and starts a new one, while **concatMap** runs tasks sequentially.
5. **What are asynchronous validators in Angular? How do you implement them?**
    - **Asynchronous validators** perform validation using external resources, such as checking the availability of a username from an API. Implement them using **AsyncValidatorFn**.
6. **What is module federation, and how can it be implemented in Angular?**
    - **Module federation** enables the sharing of code between different Angular apps, primarily using **Webpack 5** to allow micro frontends architecture where different Angular applications can be loaded independently.
7. **What is Injector Hierarchy in Angular, and how does it affect service instance creation?**
    - **Injector Hierarchy** defines the scope of services in Angular. There are **root injectors** (for app-wide services) and **module/component injectors** (scoped services). Services can be singleton at the root or have multiple instances if provided at the module/component level.
8. **How do you deal with memory leaks in Angular?**
    - Memory leaks can be avoided by:
        - Unsubscribing from **observables** using `takeUntil` or **AsyncPipe**.

        - Cleaning up **event listeners** and DOM references in `ngOnDestroy()`.
        - Using tools like **Chrome DevTools** or **Angular DevTools** to profile memory usage.
9. **Explain HttpClientModule's features such as interceptors, handling request headers, and retry mechanisms.**
    - **HttpClientModule** provides:
        - **Interceptors** for modifying requests and responses (logging, authentication, etc.).
        - **Request headers** can be added using `HttpHeaders`.
        - **Retry mechanisms** using RxJS operators like `retry()` or `retryWhen()` for failed HTTP requests.
10. **What is Angular Schematics, and how can you create a custom schematic?**
    - **Angular Schematics** automate tasks like generating components, services, and other elements. You can create custom schematics to scaffold code structures and automate repetitive tasks using the **@angular-devkit/schematics** package.
11. **How would you handle authorization and authentication in an Angular application?**
    - Implement **JWT (JSON Web Token)** for token-based authentication, use **guards** (`CanActivate`, `CanLoad`) to protect routes, and interceptors to inject tokens into HTTP requests. You can also integrate third-party solutions like **Auth0** or **Firebase Authentication**.
12. **How does NgZone work, and what is the purpose of ngZone.runOutsideAngular()?**
    - **NgZone** tracks asynchronous tasks and triggers change detection. **ngZone.runOutsideAngular()** improves performance by running non-critical code outside Angular's zone, avoiding unnecessary change detection cycles.
13. **What are the differences between Subject, BehaviorSubject, ReplaySubject, and AsyncSubject in RxJS?**
    - **Subject**: Emits values to subscribers but does not store them.
    - **BehaviorSubject**: Stores the latest emitted value and sends it to new subscribers.
    - **ReplaySubject**: Stores a specified number of past values and replays them to new subscribers.
    - **AsyncSubject**: Emits the last value (only) when the observable completes.
14. **How do you implement route resolvers to prefetch data before route activation?**
    - Implement the `Resolve` interface to fetch data from APIs or services before the route loads, ensuring the component has the necessary data before being displayed.
15. **Explain the purpose and benefits of tree-shaking in Angular.**
    - **Tree-shaking** removes unused code from the final bundle, reducing its size and improving performance. Angular's **AOT** compiler and **Webpack** help with tree-shaking by eliminating dead code during the build process.
16. **What are pure and impure pipes in Angular? What are the performance implications of each?**

- **Pure pipes** are recalculated only when their inputs change, making them efficient. **Impure pipes** are recalculated on every change detection cycle, which can impact performance negatively if overused.

17. **What is Zone.js and how does it relate to Angular's change detection?**
    - **Zone.js** patches asynchronous tasks (e.g., setTimeout, promises) and ensures Angular's change detection runs automatically when these tasks complete, updating the UI accordingly.
18. **How do you handle API pagination in Angular using HttpClient and RxJS?**
    - Handle API pagination using **RxJS operators** like `mergeMap` or `concatMap` to manage paginated data streams, and implement **infinite scrolling** or manual pagination controls in the UI.
19. **How do you configure multi-language support (i18n) in Angular?**
    - Use Angular's **i18n** tools (`@angular/localize`), configure translation files (e.g., XLIFF), and set up the app to switch between languages at runtime or load language-specific modules dynamically.
20. **What is differential loading, and why is it important in Angular applications?**
    - **Differential loading** allows Angular to build modern (ES2015+) and legacy (ES5) bundles, serving modern code to modern browsers and reducing the bundle size for improved performance.
21. **How would you design an Angular app for offline support using Service Workers?**
    - Use **@angular/service-worker** to configure service workers for caching static assets and API responses, enabling offline support. Implement strategies for background sync and push notifications.
22. **How do you prevent duplicate HTTP requests in Angular?**
    - Use **RxJS operators** like `shareReplay()` to cache results and avoid duplicate requests, or use **interceptors** to handle request deduplication and debouncing where necessary.
23. **How would you handle large media (images or videos) uploads in Angular?**
    - Implement **chunked uploads** for large files, show progress bars for the user, handle retries for failed chunks, and use `FormData` to handle file uploads through Angular's `HttpClient`.
24. **What are decorators in Angular, and how are they implemented in TypeScript?**
    - Decorators like `@Component`, `@Injectable`, and `@Input` add metadata to classes or methods. They are implemented in TypeScript as functions that return a new class or augment class behavior.
25. **How do you secure an Angular application against common web vulnerabilities like XSS, CSRF, and Clickjacking?**
    - Use Angular's built-in **DOM sanitization** to prevent XSS, **HttpClient** XSRF tokens to prevent CSRF, and configure HTTP headers like **X-Frame-Options** to mitigate Clickjacking.
26. **How does Angular Universal handle server-side rendering (SSR), and what are the benefits of SSR?**

    - **Angular Universal** pre-renders the app on the server, providing faster page loads, improved SEO, and a better perceived user experience. It handles API calls, lazy loading, and state rehydration.
27. **What are the different methods of communication between components in Angular?**
    - Methods include **@Input() and @Output()**, **shared services** with RxJS, **EventEmitter**, and parent-child communication using **ViewChild** and **ContentChild**.
28. **What is lazy loading, and how can you implement it in an Angular app?**
    - **Lazy loading** delays the loading of feature modules until they are needed, reducing the initial bundle size. Implement it with `loadChildren` in the routing configuration.
29. **How do you handle error handling globally in an Angular application?**
    - Implement global error handling by extending the **ErrorHandler** class, creating a custom error service for logging, and using **interceptors** to handle HTTP errors across the app.
30. **What are Dynamic Components in Angular, and how do you load them at runtime?**
    - Dynamic components are created at runtime using `ComponentFactoryResolver` or `ViewContainerRef.createComponent()`, allowing you to inject and load components dynamically based on user interaction or logic.
31. **How do you set up testing in an Angular project, and how would you write a unit test for a service that makes HTTP calls?**
    - Set up **Karma** and **Jasmine** for unit tests. Write tests for services with HTTP calls using `HttpTestingController` to mock HTTP responses and verify request behavior.
32. **How do you handle long-running tasks or background processing in Angular?**
    - Use **Web Workers** to offload CPU-intensive tasks to separate threads or manage long-running tasks with RxJS observables. Alternatively, use **background tasks** on the server and notify the client when completed.
33. **How would you optimize an Angular app for a mobile-first experience?**
    - Use responsive design with **CSS frameworks** like Bootstrap or Angular Material, **lazy load** resources, use **image compression**, and implement **PWA features** for a better mobile experience.
34. **What is WebSocket communication in Angular, and how do you implement it?**
    - Implement WebSocket communication using `WebSocketSubject` from RxJS for real-time data exchange, handling events like message, open, and close, for apps like live feeds or chat systems.
35. **How does Angular handle dependency injection in child modules vs. root modules?**
    - Services provided at the **root module** level are singletons and available throughout the app. In child modules, services can have separate instances if provided at the component or module level.

36. **What are forwardRef() and Optional() in Angular, and when would you use them?**
    - **forwardRef()** resolves circular dependencies in Angular's DI system, while **Optional()** allows injection of services that may not exist, preventing errors when a service is not available.

Remaining Scenario-Based and Additional Questions

1. **How would you migrate a large-scale Angular application from Angular 7 to Angular 14?**
    - To migrate from Angular 7 to Angular 14:
        - **Step 1**: Start by reading the official **Angular upgrade guide** for specific steps based on your version.
        - **Step 2**: Update Angular dependencies using `ng update @angular/cli @angular/core`.
        - **Step 3**: Ensure that all third-party libraries are compatible with Angular 14.
        - **Step 4**: Resolve any deprecations and breaking changes by reviewing migration warnings.
        - **Step 5**: Refactor code based on new best practices introduced in later Angular versions (e.g., Ivy compiler, new forms API, etc.).
        - **Step 6**: Test thoroughly using unit tests and e2e tests.
        - **Step 7**: Perform performance testing to ensure there are no regressions.
2. **You have a component with a large number of event listeners (e.g., scroll, mouse movements), and performance is slowing down. How would you optimize it?**
    - To optimize a component with a large number of event listeners:
        - **Debounce or throttle** event listeners using RxJS operators like `debounceTime()` or `throttleTime()` to reduce the frequency of event calls.
        - **Run outside of Angular's zone** using `ngZone.runOutsideAngular()` for non-critical events, like scroll or mousemove, to avoid unnecessary change detection.
        - **Detach event listeners** when they are no longer needed to avoid memory leaks.
        - **Use passive event listeners** where appropriate for events like scrolling to improve performance on mobile devices.
3. **An Angular application's initial load time is too slow. What steps would you take to reduce this?**
    - To improve the initial load time:
        - **Lazy load** feature modules using Angular's `loadChildren` in the routing configuration.

        - Enable **AOT (Ahead-of-Time) compilation** for smaller bundle sizes and faster rendering.
        - Implement **tree-shaking** to remove unused code during the build process.
        - Apply **code-splitting** and **differential loading** to serve modern browsers optimized bundles.
        - Optimize asset loading (images, fonts) by using **compression** techniques like **gzip** or **Brotli**.
        - Implement **preloading strategies** to load critical modules after the main bundle is loaded.

4. **How do you handle API pagination in Angular using HttpClient and RxJS?**
    - To handle API pagination:
        - Implement RxJS operators like `mergeMap()`, `concatMap()`, or `switchMap()` to handle paginated API requests.
        - Create a paginated API service that retrieves chunks of data by passing pagination parameters (e.g., page number and size).
        - Use `HttpClient` to send HTTP requests with pagination query parameters.
        - For better UX, you can implement **infinite scrolling** using Angular's **CDK Virtual Scroller** or **manual pagination** using buttons.
5. **How do you handle large datasets in Angular for optimized rendering and user experience?**
    - For large datasets, implement the following strategies:
        - **Virtual Scrolling**: Use Angular's **CDK Virtual Scroller** (`cdk-virtual-scroll-viewport`) to render only the visible items, improving performance for large lists.
        - **Pagination**: Divide data into smaller pages and fetch only what's needed.
        - **Infinite scrolling**: Load more data dynamically as the user scrolls down the page.
        - Use `trackBy` in `*ngFor` to improve DOM rendering performance by tracking unique identifiers.
6. **What are the best practices for managing global state in large Angular applications?**
    - Best practices include:
        - Use a **state management library** like **NgRx**, **Akita**, or **NGXS** to manage complex global states with actions, reducers, effects, and selectors.
        - Use **BehaviorSubjects** or **Services** for simple state management without the need for third-party libraries.
        - Ensure state is **immutable** to make it easier to track changes and debug.
        - Avoid deep component hierarchies; instead, use **smart and dumb components** to isolate state-related logic from the UI.
        - Organize the state into **feature-based slices** for scalability.

7. **How would you implement SSR (Server-Side Rendering) with Angular Universal to improve SEO and performance?**
    - To implement **Angular Universal** for SSR:
        - Install Angular Universal by running `ng add @nguniversal/express-engine`.
        - Modify the app to support server-side rendering, handling dynamic content, lazy loading, and API calls.
        - Use `ng run` to generate the server bundle.
        - Implement strategies for **SEO improvements** (e.g., rendering meta tags on the server).
        - Implement caching mechanisms to minimize server load and ensure that the app loads quickly.
8. **How do you handle nested forms in Angular?**
    - Use **FormGroup** and **FormArray** to manage nested forms in Angular:
        - Create parent **FormGroups** and nest **FormGroups** or **FormArrays** inside them to represent child forms.
        - Dynamically add or remove controls from **FormArray** for flexible form structures.
        - Use custom validators and nested form controls to validate related form fields.
9. **What are the best practices for securing Angular applications against XSS and CSRF attacks?**
    - For **XSS** (Cross-Site Scripting) prevention:
        - Rely on Angular's **built-in DOM sanitization** for template rendering.
        - Use **DomSanitizer** to manually sanitize dynamic HTML content.
    - For **CSRF** (Cross-Site Request Forgery) protection:
        - Use Angular's **HttpClient** module, which automatically adds **XSRF tokens** to outgoing HTTP requests.
        - Implement security headers like **Content Security Policy (CSP)** and **X-Frame-Options**.
10. **How would you manage multiple themes in an Angular application?**
    - To support multiple themes:
        - Use **CSS variables** or **SCSS** for theme customization and switch themes dynamically by changing the root variables.
        - Alternatively, load different **CSS files** dynamically based on user preferences.
        - For Angular Material applications, utilize **Angular Material's theming** capabilities to define multiple themes and switch between them.
11. **How do you handle large file uploads in Angular?**
    - For large file uploads:
        - Implement **chunked uploads**, breaking files into smaller parts and uploading them sequentially.
        - Use `FormData` with Angular's `HttpClient` to handle file uploads.

        - Display **progress bars** using RxJS operators like `tap()` to show upload status to users.
        - Implement retry mechanisms to handle upload failures.

12. **How do you handle complex form validations in Angular using custom validators?**
    - Implement **custom validators** by creating `ValidatorFn` for synchronous validators and `AsyncValidatorFn` for asynchronous validators.
    - Combine multiple validators in the form control by passing an array of validators.
    - Use cross-field validation by creating custom validators that operate across different form controls within a **FormGroup**.
13. **What is the purpose of the ControlValueAccessor interface, and how do you implement a custom form control?**
    - **ControlValueAccessor** is an interface that allows Angular forms to interact with custom form controls. To implement it:
        - Implement `writeValue()`, `registerOnChange()`, and `registerOnTouched()` methods.
        - Handle two-way data binding between the custom form control and the form.
14. **How would you design a reusable Angular component library for internal use?**
    - Design a modular component library by:
        - Creating highly reusable components with **Input** and **Output** properties to handle dynamic data and events.
        - Use **ng-packagr** to bundle the component library.
        - Define versioning and proper documentation for maintainability.
        - Ensure that the library is tree-shakable by following Angular's best practices for AOT and lazy loading.
15. **How does Angular handle nullish coalescing and optional chaining in templates?**
    - Angular templates support **nullish coalescing (`??`)** and **optional chaining (`?.`)** operators:
        - Use `?.` to safely access nested properties, avoiding null/undefined errors.
        - Use `??` to provide default values when the left-hand operand is `null` or `undefined`.
16. **How would you implement skeleton loading in Angular?**
    - **Skeleton loading** improves perceived performance by displaying placeholder content while the actual data loads:
        - Create a skeleton loader component with placeholders.
        - Conditionally display the loader component until the data is fetched.
        - Remove the skeleton once the actual data is available.
17. **What is the Renderer2 service in Angular, and when would you use it instead of direct DOM manipulation?**
    - **Renderer2** is a platform-agnostic service used to manipulate DOM elements safely, ensuring compatibility with server-side rendering (SSR) or Web Workers.
    - Use `Renderer2` instead of direct DOM manipulation to avoid XSS risks and ensure cross-platform compatibility.

18. **How do you configure Angular to work with different environments (e.g., development, staging, production)?**
    - Angular uses **environment-specific configuration** files located in the `src/environments` directory.
    - Use different configurations for each environment by creating separate files (e.g., `environment.ts`, `environment.prod.ts`).
    - Define different build settings in `angular.json` to replace these environment files during the build process.