Angular Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a custom structural directive that conditionally renders content based on a numeric value range

Solution:

Create a structural directive that shows/hides content when a number falls within a specified range:

```
@Directive({
  selector: '[inRange]'
})
export class InRangeDirective {
  @Input('inRange') set range([min, max]: [number, number]) {
    const value = this.view.context.$implicit;
    this.viewContainer.clear();
    if (value >= min && value <= max) {
      this.viewContainer.createEmbeddedView(this.template);
    }
  }
}
```

2. Create a custom RxJS operator that debounces user input and filters out empty strings

Solution:

```
const filterValidInput = () => pipe(
  debounceTime(300),
  distinctUntilChanged(),
  filter(text => text.trim().length > 0),
  map(text => text.toLowerCase())
);
```

Usage:

```
searchInput$.pipe(
  filterValidInput()
).subscribe(value => this.search(value));
```

3. Implement a custom form control that validates and formats credit card numbers in real-time

Solution:

```
@Component({
  selector: 'credit-card-input',
  providers: [{
    provide: NG_VALUE_ACCESSOR,
    useExisting: forwardRef(() => CreditCardComponent),
    multi: true
  }]
})
export class CreditCardComponent implements ControlValueAccessor {
  formatInput(value: string): string {
    return value.replace(/\D/g, '').replace(/(.{4})/g, '$1 ').trim();
  }
}
```

4. Create a performance-optimized virtual scroll implementation for a large dataset

Solution:

```
@Component({
  selector: 'virtual-scroll',
  template: `









`
})
```

5. Implement a custom decorator that caches method results based on arguments

Solution:

```
function Memoize() {
  return function (target: any, key: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    const cache = new Map();
    descriptor.value = function (...args: any[]) {
      const key = JSON.stringify(args);
      if (!cache.has(key)) cache.set(key, original.apply(this, args));
      return cache.get(key);
    };
  };
}
```

6. Create a custom change detection strategy that optimizes rendering of a complex tree structure

Solution:

```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `



`
})
export class TreeComponent implements OnInit {
  state$ = this.store.select(getTreeState).pipe(distinctUntilChanged());
}
```

7. Implement a custom Angular pipe that formats file sizes with appropriate units

Solution:

```
@Pipe({
  name: 'fileSize'
})
export class FileSizePipe implements PipeTransform {
  transform(bytes: number): string {
    const units = ['B', 'KB', 'MB', 'GB'];
    let i = 0;
```

```
    while (bytes >= 1024 && i < units.length - 1) {
      bytes /= 1024;
      i++;
    }
    return `${bytes.toFixed(1)} ${units[i]}`;
  }
}
```

8. Create a reusable error boundary component that gracefully handles component errors

Solution:

```
@Component({
  selector: 'error-boundary',
  template: `




{{error?.message}}

  `
})
export class ErrorBoundaryComponent implements ErrorHandler {
  handleError(error: Error) { this.error = error; }
}
```

9. Implement a custom preloading strategy that prioritizes specific feature modules

Solution:

```
@Injectable()
export class PriorityPreloadingStrategy implements PreloadAllModules {
  preload(route: Route, load: () => Observable): Observable {
    if (route.data?.['preload'] === true) {
      return load();
    }
    return of(null);
  }
}
```

Usage in RouterModule:

```
RouterModule.forRoot(routes, {
  preloadingStrategy: PriorityPreloadingStrategy
})
```

10. Create a custom HTTP interceptor that implements retry logic with exponential backoff

Solution:

```
@Injectable()
export class RetryInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest, next: HttpHandler): Observable {
    return next.handle(req).pipe(
      retryWhen(errors => errors.pipe(
        concatMap((error, index) =>
          timer(Math.min(1000 * Math.pow(2, index), 10000))
        ),
        take(3)
      ))
    );
  }
}
```

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU Cache in Angular using TypeScript

Solution

Here's an efficient LRU Cache implementation using Map:

```
class LRUCache {
  private cache = new Map();
  constructor(private capacity: number) {}
  get(key: number): number {
    if (!this.cache.has(key)) return -1;
    const value = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, value);
    return value;
  }}
```

Time Complexity: O(1) for both get and put operations

2. Write a function to find pairs in an array that sum to a target value

Solution

```
function findPairs(arr: number[], target: number): number[][] {
  const seen = new Set();
  const result = [];
  for (const num of arr) {
    if (seen.has(target - num)) {
      result.push([num, target - num]);
    }
    seen.add(num);
  }
```

Time Complexity: O(n) where n is array length

3. Implement a Stack data structure with push, pop, and getMin operations in O(1) time

Solution

```
class MinStack {
  private stack: number[] = [];
  private minStack: number[] = [];
  push(val: number): void {
    this.stack.push(val);
    if (!this.minStack.length || val <= this.minStack[this.minStack.length-1]) {
      this.minStack.push(val);
    }
  }
```

Key Points:

- Uses auxiliary stack to track minimums
- All operations are O(1)

4. Implement a sliding window maximum algorithm

Solution

```
function maxSlidingWindow(nums: number[], k: number): number[] {
  const result: number[] = [];
  const deque: number[] = [];
  for (let i = 0; i < nums.length; i++) {
    while (deque.length && nums[deque[deque.length-1]] <= nums[i]) {
      deque.pop();
    }
  }
```

Time Complexity: O(n) where n is array length

5. Implement a Queue using two Stacks

Solution

```
class Queue {
  private stack1: number[] = [];
  private stack2: number[] = [];
  enqueue(val: number): void {
    this.stack1.push(val);
  }
  dequeue(): number {
    if (!this.stack2.length) this.transfer();
    return this.stack2.pop() ?? -1;
  }
```

Time Complexity: O(1) amortized for both operations

6. Design a data structure that supports insert, delete, getRandom in O(1)

Solution

```
class RandomizedSet {
  private map = new Map();
  private list: number[] = [];
  insert(val: number): boolean {
    if (this.map.has(val)) return false;
    this.map.set(val, this.list.length);
    this.list.push(val);
    return true;
  }
```

Key Features:

- Uses Map for O(1) lookups
- Array for O(1) random access

7. Implement a Trie (Prefix Tree) data structure

Solution

```
class TrieNode {
  children = new Map();
  isEndOfWord = false;
}
class Trie {
  root = new TrieNode();
  insert(word: string): void {
    let current = this.root;
    for (const char of word) {
      current.children.set(char, current.children.get(char) || new TrieNode());
    }
  }
```

Time Complexity: O(m) for insertions where m is word length

8. Implement a circular buffer/ring buffer

Solution

```
class CircularBuffer {
  private buffer: (T | undefined)[];
  private head = 0;
  private tail = 0;
  constructor(private capacity: number) {
    this.buffer = new Array(capacity);
  }
  write(item: T): boolean {
    if (this.isFull()) return false;
  }
```

Applications:

- Stream processing
- Event handling

9. Implement a thread-safe singleton pattern in TypeScript

Solution

```
class Singleton {
  private static instance: Singleton;
  private constructor() {}
  public static getInstance(): Singleton {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
```

Key Points:

- Private constructor
- Static instance management

10. Implement a debounce function with TypeScript

Solution

```
function debounce any>(
  fn: T,
  delay: number
): (...args: Parameters) => void {
  let timeoutId: ReturnType;
  return (...args: Parameters) => {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn(...args), delay);
  };
}
```

Usage: Rate limiting UI events

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly

Key Components & Considerations:

- **API Gateway**: Handle incoming requests, rate limiting, authentication
- **URL Generation Service**: Create unique short URLs using base62 encoding or counter-based approach
- **Storage**: NoSQL database (e.g. DynamoDB) for URL mappings
- **Cache Layer**: Redis for frequently accessed URLs
- **Analytics Service**: Track clicks and user metrics

Sample URL Generation Code:

```
function generateShortUrl(counter) {
  const base62chars = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
  let shortUrl = '';
  while (counter > 0) {
    shortUrl = base62chars[counter % 62] + shortUrl;
    counter = Math.floor(counter / 62);
  }
  return shortUrl;
}
```

2. Design a real-time chat application with support for group messaging

Architecture Components:

- **WebSocket Server**: Handle real-time bidirectional communication
- **Message Queue**: RabbitMQ/Kafka for message processing
- **Database**: MongoDB for message persistence
- **Presence Service**: Track online/offline status
- **Notification Service**: Push notifications

Example WebSocket Handler:

```
@WebSocketGateway()
class ChatGateway {
  @SubscribeMessage('message')
  handleMessage(client: Socket, payload: any): void {
    this.server.to(payload.roomId).emit('newMessage', payload);
  }
}
```

3. Design a social media feed system with infinite scroll

Core Components:

- **Feed Service**: Aggregate and rank posts
- **Content Delivery Network (CDN)**: Cache media content
- **Cache Layer**: Redis for feed composition
- **Database**: Cassandra for post storage

Feed Pagination Implementation:

```
interface FeedResponse {
```

```
  posts: Post[];
  cursor: string;
}

async getFeed(userId: string, cursor?: string): Promise {
  const limit = 20;
  return this.feedService.fetchPosts(userId, cursor, limit);
}
```

4. Design a distributed task scheduling system

System Components:

- **Job Queue**: Redis/RabbitMQ for task distribution
- **Worker Nodes**: Process tasks in parallel
- **State Store**: Track job status and results
- **Monitoring Service**: Health checks and metrics

Task Scheduler Example:

```
class TaskScheduler {
  schedule(task: Task, options: ScheduleOptions): string {
    const jobId = generateUUID();
    this.queue.push({ id: jobId, task, options });
    return jobId;
  }
}
```

5. Design a rate limiting system for a REST API

Implementation Approaches:

- **Token Bucket Algorithm**: Flexible rate limiting
- **Redis**: Track request counts
- **Distributed Configuration**: Consistent limits across nodes

Rate Limiter Implementation:

```
class RateLimiter {
  async isAllowed(userId: string): Promise {
    const current = await this.redis.incr(`${userId}:requests`);
    if (current === 1) this.redis.expire(`${userId}:requests`, 3600);
    return current <= this.maxRequests;
  }
}
```

6. Design a notification service supporting multiple channels (email, SMS, push)

Architecture Components:

- **Notification Gateway**: Route notifications
- **Channel Adapters**: Interface with providers
- **Template Engine**: Notification content
- **Retry Mechanism**: Handle failures

Notification Handler:

```
class NotificationService {
  async send(notification: Notification): Promise {
    const adapter = this.getAdapter(notification.channel);
    await adapter.send(notification);
    this.track(notification);
  }
}
```

7. Design a distributed caching system

Key Components:

- **Cache Nodes**: Distributed storage
- **Consistent Hashing**: Data distribution
- **Replication**: Data redundancy
- **Cache Invalidation**: Consistency management

Cache Implementation:

```
class DistributedCache {
  async get(key: string): Promise {
    const node = this.getNode(this.hash(key));
    return node.get(key) || this.loadFromDatabase(key);
  }
}
```

8. Design a real-time analytics pipeline

System Components:

- **Event Collector**: Gather analytics data
- **Stream Processing**: Apache Kafka/Kinesis
- **Processing Engine**: Apache Spark
- **Time-series DB**: InfluxDB/Prometheus

Event Processing:

```
interface AnalyticsEvent {
  timestamp: number;
  eventType: string;
  payload: any;
}

class EventProcessor {
  process(event: AnalyticsEvent): void {
    this.streamProcessor.emit(event);
  }
}
```

9. Design a content recommendation system

Core Components:

- **Feature Extraction**: Content analysis
- **Recommendation Engine**: ML models
- **User Profile Service**: Preference tracking
- **A/B Testing Framework**: Algorithm optimization

Recommendation Logic:

```
class RecommendationEngine {
  async getRecommendations(userId: string): Promise {
    const userProfile = await this.profileService.get(userId);
    return this.rankContent(userProfile);
  }
}
```

10. Design a distributed search engine

System Components:

- **Indexer Service**: Document processing
- **Search Nodes**: Distributed index storage
- **Query Parser**: Search request handling
- **Ranking Service**: Result relevance

Search Implementation:

```
class SearchEngine {
  async search(query: string): Promise {
    const parsedQuery = this.queryParser.parse(query);
    return this.searchNodes.executeQuery(parsedQuery);
  }
}
```

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Create a custom debounce decorator in Angular that delays function execution until after wait milliseconds have elapsed since the last time it was invoked.

Solution:

Here's how to implement a debounce decorator:

```
function Debounce(delay: number = 300) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    let timeoutRef: any;
    const original = descriptor.value;
    descriptor.value = function (...args: any[]) {
      clearTimeout(timeoutRef);
      timeoutRef = setTimeout(() => original.apply(this, args), delay);
    };
    return descriptor;
  };
}
```

Usage:

```
@Debounce(500)
search(term: string) {
  // search implementation
}
```

2. Implement a custom pipe that flattens a nested array in Angular.

Solution:

```
@Pipe({
  name: 'flatten'
})
export class FlattenPipe implements PipeTransform {
  transform(arr: any[]): any[] {
    return arr.reduce((flat, toFlat) =>
      flat.concat(Array.isArray(toFlat) ? this.transform(toFlat) : toFlat), []);
  }
}
```

Usage:

```
{{ [1, [2, 3], [4, [5, 6]]] | flatten }}
```

3. Create a custom directive that implements infinite scroll in Angular.

Solution:

```
@Directive({
  selector: '[infiniteScroll]'
})
export class InfiniteScrollDirective {
  @Output() scrolled = new EventEmitter();
  @HostListener('window:scroll', [])
  onScroll(): void {
    if ((window.innerHeight + window.scrollY) >= document.body.offsetHeight) {
      this.scrolled.emit();
```

```
    }
  }
}
```

4. Implement a custom error handler in Angular that logs errors to a service and shows a user-friendly message.

Solution:

```
@Injectable()
export class CustomErrorHandler implements ErrorHandler {
  constructor(private errorService: ErrorService) {}
  handleError(error: Error) {
    this.errorService.logError(error);
    console.error(error);
    alert('An error occurred. Please try again later.');
  }
}
```

Registration:

```
providers: [{ provide: ErrorHandler, useClass: CustomErrorHandler }]
```

5. Create a memory leak detector service that monitors component lifecycle and detects potential memory leaks.

Solution:

```
@Injectable({
  providedIn: 'root'
})
export class MemoryLeakDetectorService {
  private componentRefs = new WeakMap();
  track(component: any, name: string): void {
    this.componentRefs.set(component, `${name}_${Date.now()}`);
    console.log(`Component ${name} instantiated`);
  }
}
```

6. Implement a custom RouteReuseStrategy that caches and reuses route components.

Solution:

```
export class CustomRouteReuseStrategy implements RouteReuseStrategy {
  private storedRoutes = new Map();
  shouldDetach(route: ActivatedRouteSnapshot): boolean {
    return route.data?.['reuse'] === true;
  }
  store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle): void {
    this.storedRoutes.set(route.routeConfig!.path!, handle);
  }
}
```

7. Create a custom structural directive that implements a for loop with step functionality.

Solution:

```
@Directive({
  selector: '[stepFor]'
})
export class StepForDirective {
  constructor(private template: TemplateRef,
              private vcr: ViewContainerRef) {}
  @Input() set stepForOf(config: {start: number, end: number, step: number}) {
    this.vcr.clear();
    for(let i = config.start; i <= config.end; i += config.step) {
      this.vcr.createEmbeddedView(this.template, { $implicit: i });
    }
```

```
  }
}
```

8. Implement a custom validator that ensures a form control value is a valid JSON string.

Solution:

```
export function jsonValidator(): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    try {
      JSON.parse(control.value);
      return null;
    } catch (e) {
      return { invalidJson: true };
    }
  };
}
```

Usage:

```
this.form = new FormGroup({
  jsonField: new FormControl('', [jsonValidator()])
});
```

9. Create a custom preloading strategy that preloads modules based on user roles.

Solution:

```
@Injectable()
export class RoleBasedPreloadingStrategy implements PreloadAllModules {
  constructor(private auth: AuthService) {}
  preload(route: Route, load: () => Observable): Observable {
    return this.auth.hasRole(route.data?.['requiredRole'])
      ? load()
      : EMPTY;
  }
}
```

10. Implement a custom change detection strategy that only triggers on specific property changes.

Solution:

```
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class CustomComponent implements OnChanges {
  @Input() data: any;
  ngOnChanges(changes: SimpleChanges) {
    if (changes['data'] && !this.isEqual(changes['data'].previousValue,
        changes['data'].currentValue)) {
      this.update();
    }
  }
}
```