Complete Angular Design Patterns Guide

A comprehensive guide combining Angular-specific patterns, architectural decisions, and practical TypeScript code examples. This guide covers everything from basic Angular patterns to advanced enterprise-level architectures.

Table of Contents

Angular-Specific Patterns

Architectural Patterns

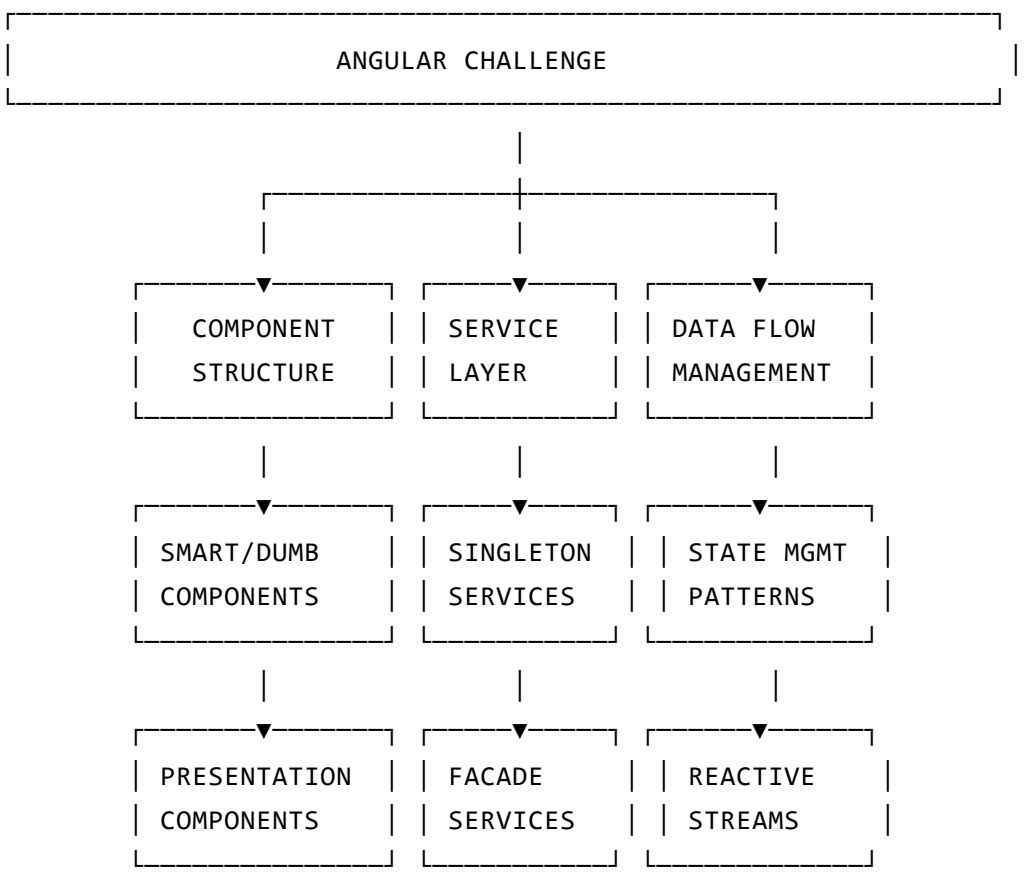
Advanced Patterns

Decision Tools

- Angular Pattern Decision Tree
- Pattern Selection Matrix
- Best Practices Checklist

Angular Pattern Decision Tree

When building Angular applications...



1. Component Patterns

Smart/Dumb Components (Container/Presentational)

Description: Separates components into smart containers (with logic) and dumb presentational components (pure UI). Essential for maintainable Angular applications.

```
// Smart Component (Container)
@Component({
  selector: 'app-user-list',
  template: `
    <div class="user-list-container">
      <h2>Users</h2>
      <app-user-search
        (search)="onSearch($event)"
        [loading]="isLoading">
      </app-user-search>
      <app-user-table
        [users]="users"
        [loading]="isLoading"
        (edit)="onEditUser($event)"
        (delete)="onDeleteUser($event)">
      </app-user-table>
    </div>
  `
})
export class UserListComponent implements OnInit {
  users: User[] = [];
  isLoading = false;

  constructor(
    private userService: UserService,
    private router: Router
  ) {}

  ngOnInit() {
    this.loadUsers();
  }

  onSearch(searchTerm: string) {
    this.isLoading = true;
    this.userService.searchUsers(searchTerm).subscribe(users => {
      this.users = users;
      this.isLoading = false;
    });
  }

  onEditUser(user: User) {
    this.router.navigate(['/users', user.id, 'edit']);
  }
```

```
  onDeleteUser(user: User) {
    this.userService.deleteUser(user.id).subscribe(() => {
      this.loadUsers();
    });
  }

  private loadUsers() {
    this.isLoading = true;
    this.userService.getUsers().subscribe(users => {
      this.users = users;
      this.isLoading = false;
    });
  }
}

// Dumb Component (Presentational)
@Component({
  selector: 'app-user-table',
  template: `
    <div class="user-table">
      <div *ngIf="loading" class="loading">Loading...</div>
      <table *ngIf="!loading">
        <thead>
          <tr>
            <th>Name</th>
            <th>Email</th>
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let user of users">
            <td>{{ user.name }}</td>
            <td>{{ user.email }}</td>
            <td>
              <button (click)="edit.emit(user)">Edit</button>
              <button (click)="delete.emit(user)">Delete</button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  `
```

```
})
export class UserTableComponent {
  @Input() users: User[] = [];
  @Input() loading = false;
  @Output() edit = new EventEmitter<User>();
  @Output() delete = new EventEmitter<User>();
}

// Dumb Component (Search)
@Component({
  selector: 'app-user-search',
  template: `
    <div class="search-container">
      <input
        type="text"
        placeholder="Search users..."
        [value]="searchTerm"
        (input)="onSearchChange($event)"
        [disabled]="loading">
      <button (click)="onSearch()" [disabled]="loading">
        {{ loading ? 'Searching...' : 'Search' }}
      </button>
    </div>
  `
})
export class UserSearchComponent {
  @Input() loading = false;
  @Output() search = new EventEmitter<string>();

  searchTerm = '';

  onSearchChange(event: Event) {
    this.searchTerm = (event.target as HTMLInputElement).value;
  }

  onSearch() {
    this.search.emit(this.searchTerm);
  }
}
```

Component Composition Pattern

Description: Builds complex UIs by composing smaller, reusable components. Promotes reusability and maintainability.

```
// Base Card Component
@Component({
  selector: 'app-card',
  template: `
    <div class="card" [class.card--elevated]="elevated">
      <div class="card__header" *ngIf="headerTemplate">
        <ng-container *ngTemplateOutlet="headerTemplate"></ng-container>
      </div>
      <div class="card__content">
        <ng-content></ng-content>
      </div>
      <div class="card__footer" *ngIf="footerTemplate">
        <ng-container *ngTemplateOutlet="footerTemplate"></ng-container>
      </div>
    </div>
  `
})
export class CardComponent {
  @Input() elevated = false;
  @Input() headerTemplate: TemplateRef<any>;
  @Input() footerTemplate: TemplateRef<any>;
}

// User Card Component (Composed)
@Component({
  selector: 'app-user-card',
  template: `
    <app-card [elevated]="true">
      <ng-template #header>
        <div class="user-card-header">
          <img [src]="user.avatar" [alt]="user.name" class="user-avatar">
          <h3>{{ user.name }}</h3>
        </div>
      </ng-template>

      <div class="user-details">
        <p><strong>Email:</strong> {{ user.email }}</p>
        <p><strong>Role:</strong> {{ user.role }}</p>
        <p><strong>Last Login:</strong> {{ user.lastLogin | date }}</p>
      </div>

      <ng-template #footer>
        <div class="user-actions">
```

```
          <button (click)="onEdit()">Edit</button>
          <button (click)="onDelete()">Delete</button>
        </div>
      </ng-template>
    </app-card>
  `
})
export class UserCardComponent {
  @Input() user: User;
  @Output() edit = new EventEmitter<User>();
  @Output() delete = new EventEmitter<User>();

  onEdit() {
    this.edit.emit(this.user);
  }

  onDelete() {
    this.delete.emit(this.user);
  }
}
```

Component Communication Patterns

Description: Various ways components can communicate in Angular applications.

```typescript
// 1. Parent-Child Communication (@Input/@Output)
@Component({
  selector: 'app-parent',
  template: `
    <app-child
      [data]="parentData"
      (dataChange)="onChildDataChange($event)">
    </app-child>
  `
})
export class ParentComponent {
  parentData = 'Hello from parent';

  onChildDataChange(newData: string) {
    this.parentData = newData;
  }
}

@Component({
  selector: 'app-child',
  template: `
    <div>
      <p>Received: {{ data }}</p>
      <button (click)="sendDataBack()">Send Data Back</button>
    </div>
  `
})
export class ChildComponent {
  @Input() data: string;
  @Output() dataChange = new EventEmitter<string>();

  sendDataBack() {
    this.dataChange.emit('Hello from child');
  }
}

// 2. Service Communication
@Injectable({
  providedIn: 'root'
})
export class CommunicationService {
  private dataSubject = new BehaviorSubject<string>('');
  data$ = this.dataSubject.asObservable();
```

```
  sendData(data: string) {
    this.dataSubject.next(data);
  }
}

@Component({
  selector: 'app-sender',
  template: `<button (click)="sendMessage()">Send Message</button>`
})
export class SenderComponent {
  constructor(private commService: CommunicationService) {}

  sendMessage() {
    this.commService.sendData('Message from sender');
  }
}

@Component({
  selector: 'app-receiver',
  template: `<p>Received: {{ receivedData }}</p>`
})
export class ReceiverComponent implements OnInit, OnDestroy {
  receivedData = '';
  private subscription: Subscription;

  constructor(private commService: CommunicationService) {}

  ngOnInit() {
    this.subscription = this.commService.data$.subscribe(data => {
      this.receivedData = data;
    });
  }

  ngOnDestroy() {
    this.subscription?.unsubscribe();
  }
}

// 3. ViewChild Communication
@Component({
  selector: 'app-parent',
  template: `
```

```
    <app-child #childRef></app-child>
    <button (click)="callChildMethod()">Call Child Method</button>
  `
})
export class ParentComponent {
  @ViewChild('childRef') childComponent: ChildComponent;

  callChildMethod() {
    this.childComponent.childMethod();
  }
}

@Component({
  selector: 'app-child',
  template: `<p>Child Component</p>`
})
export class ChildComponent {
  childMethod() {
    console.log('Method called from parent');
  }
}
```

2. Service Patterns

Singleton Services

Description: Services provided at root level for application-wide state and functionality.

```
// User Service (Singleton)
@Injectable({
  providedIn: 'root'
})
export class UserService {
  private usersSubject = new BehaviorSubject<User[]>([]);
  users$ = this.usersSubject.asObservable();

  private currentUserSubject = new BehaviorSubject<User | null>(null);
  currentUser$ = this.currentUserSubject.asObservable();

  constructor(private http: HttpClient) {
    this.loadUsers();
  }

  getUsers(): Observable<User[]> {
    return this.users$;
  }

  getUserById(id: number): Observable<User | undefined> {
    return this.users$.pipe(
      map(users => users.find(user => user.id === id))
    );
  }

  addUser(user: User): Observable<User> {
    return this.http.post<User>('/api/users', user).pipe(
      tap(newUser => {
        const currentUsers = this.usersSubject.value;
        this.usersSubject.next([...currentUsers, newUser]);
      })
    );
  }

  updateUser(user: User): Observable<User> {
    return this.http.put<User>(`/api/users/${user.id}`, user).pipe(
      tap(updatedUser => {
        const currentUsers = this.usersSubject.value;
        const index = currentUsers.findIndex(u => u.id === user.id);
        if (index !== -1) {
          currentUsers[index] = updatedUser;
          this.usersSubject.next([...currentUsers]);
        }
```

```
      })
    );
  }

  deleteUser(id: number): Observable<void> {
    return this.http.delete<void>(`/api/users/${id}`).pipe(
      tap(() => {
        const currentUsers = this.usersSubject.value;
        this.usersSubject.next(currentUsers.filter(user => user.id !== id));
      })
    );
  }

  setCurrentUser(user: User) {
    this.currentUserSubject.next(user);
  }

  private loadUsers() {
    this.http.get<User[]>('/api/users').subscribe(users => {
      this.usersSubject.next(users);
    });
  }
}

// Configuration Service
@Injectable({
  providedIn: 'root'
})
export class ConfigService {
  private config: any = {};

  constructor(private http: HttpClient) {}

  loadConfig(): Observable<any> {
    return this.http.get('/api/config').pipe(
      tap(config => {
        this.config = config;
      })
    );
  }

  get(key: string): any {
    return this.config[key];
```

```
  }

  getApiUrl(): string {
    return this.config.apiUrl || 'http://localhost:3000/api';
  }

  getEnvironment(): string {
    return this.config.environment || 'development';
  }
}
```

Facade Services

Description: Provides a simplified interface to complex subsystems or multiple services.

```
// User Management Facade
@Injectable({
  providedIn: 'root'
})
export class UserManagementFacade {
  constructor(
    private userService: UserService,
    private authService: AuthService,
    private notificationService: NotificationService,
    private auditService: AuditService
  ) {}

  // Simplified interface for user operations
  createUser(userData: CreateUserRequest): Observable<User> {
    return this.userService.addUser(userData).pipe(
      tap(user => {
        this.notificationService.showSuccess(`User ${user.name} created successfully`);
        this.auditService.logAction('USER_CREATED', user.id);
      }),
      catchError(error => {
        this.notificationService.showError('Failed to create user');
        return throwError(error);
      })
    );
  }

  updateUserProfile(userId: number, profileData: UpdateProfileRequest): Observable<User>
    return this.userService.updateUser({ ...profileData, id: userId }).pipe(
      tap(user => {
        this.notificationService.showSuccess('Profile updated successfully');
        this.auditService.logAction('PROFILE_UPDATED', userId);
      }),
      catchError(error => {
        this.notificationService.showError('Failed to update profile');
        return throwError(error);
      })
    );
  }

  deleteUserAccount(userId: number): Observable<void> {
    return this.userService.deleteUser(userId).pipe(
      tap(() => {
        this.notificationService.showSuccess('User account deleted');
```

```
        this.auditService.logAction('USER_DELETED', userId);
      }),
      catchError(error => {
        this.notificationService.showError('Failed to delete user account');
        return throwError(error);
      })
    );
  }

  // Complex operation combining multiple services
  transferUserData(fromUserId: number, toUserId: number): Observable<void> {
    return forkJoin([
      this.userService.getUserById(fromUserId),
      this.userService.getUserById(toUserId)
    ]).pipe(
      switchMap(([fromUser, toUser]) => {
        if (!fromUser || !toUser) {
          throw new Error('One or both users not found');
        }

        return this.userService.transferData(fromUserId, toUserId);
      }),
      tap(() => {
        this.notificationService.showSuccess('User data transferred successfully');
        this.auditService.logAction('DATA_TRANSFERRED', { fromUserId, toUserId });
      }),
      catchError(error => {
        this.notificationService.showError('Failed to transfer user data');
        return throwError(error);
      })
    );
  }
}
```

Factory Services

Description: Creates instances of services or objects based on configuration or runtime conditions.

```
// Service Factory
@Injectable({
  providedIn: 'root'
})
export class ServiceFactory {
  constructor(
    private http: HttpClient,
    private configService: ConfigService
  ) {}

  createApiService(apiType: 'users' | 'products' | 'orders'): ApiService {
    const baseUrl = this.configService.getApiUrl();

    switch (apiType) {
      case 'users':
        return new UserApiService(this.http, `${baseUrl}/users`);
      case 'products':
        return new ProductApiService(this.http, `${baseUrl}/products`);
      case 'orders':
        return new OrderApiService(this.http, `${baseUrl}/orders`);
      default:
        throw new Error(`Unknown API type: ${apiType}`);
    }
  }

  createNotificationService(type: 'toast' | 'modal' | 'banner'): NotificationService {
    switch (type) {
      case 'toast':
        return new ToastNotificationService();
      case 'modal':
        return new ModalNotificationService();
      case 'banner':
        return new BannerNotificationService();
      default:
        return new ToastNotificationService();
    }
  }
}

// Abstract API Service
export abstract class ApiService<T> {
  constructor(
    protected http: HttpClient,
```

```
    protected baseUrl: string
  ) {}

  abstract getAll(): Observable<T[]>;
  abstract getById(id: number): Observable<T>;
  abstract create(item: Partial<T>): Observable<T>;
  abstract update(id: number, item: Partial<T>): Observable<T>;
  abstract delete(id: number): Observable<void>;
}

// Concrete API Services
export class UserApiService extends ApiService<User> {
  getAll(): Observable<User[]> {
    return this.http.get<User[]>(this.baseUrl);
  }

  getById(id: number): Observable<User> {
    return this.http.get<User>(`${this.baseUrl}/${id}`);
  }

  create(user: Partial<User>): Observable<User> {
    return this.http.post<User>(this.baseUrl, user);
  }

  update(id: number, user: Partial<User>): Observable<User> {
    return this.http.put<User>(`${this.baseUrl}/${id}`, user);
  }

  delete(id: number): Observable<void> {
    return this.http.delete<void>(`${this.baseUrl}/${id}`);
  }
}
```

3. Data Flow Patterns

Reactive Data Flow with RxJS

Description: Uses RxJS streams for managing data flow and state changes throughout the application.

```
// Reactive Data Service
@Injectable({
  providedIn: 'root'
})
export class ReactiveDataService {
  private dataSubject = new BehaviorSubject<any[]>([]);
  private loadingSubject = new BehaviorSubject<boolean>(false);
  private errorSubject = new BehaviorSubject<string | null>(null);

  // Public observables
  data$ = this.dataSubject.asObservable();
  loading$ = this.loadingSubject.asObservable();
  error$ = this.errorSubject.asObservable();

  // Combined state observable
  state$ = combineLatest([
    this.data$,
    this.loading$,
    this.error$
  ]).pipe(
    map(([data, loading, error]) => ({
      data,
      loading,
      error,
      hasData: data.length > 0,
      isEmpty: data.length === 0 && !loading
    }))
  );

  constructor(private http: HttpClient) {}

  loadData(): Observable<any[]> {
    this.setLoading(true);
    this.clearError();

    return this.http.get<any[]>('/api/data').pipe(
      tap(data => {
        this.dataSubject.next(data);
        this.setLoading(false);
      }),
      catchError(error => {
        this.setError('Failed to load data');
        this.setLoading(false);
```

```
        return throwError(error);
      })
    );
  }

  addItem(item: any): Observable<any> {
    return this.http.post<any>('/api/data', item).pipe(
      tap(newItem => {
        const currentData = this.dataSubject.value;
        this.dataSubject.next([...currentData, newItem]);
      }),
      catchError(error => {
        this.setError('Failed to add item');
        return throwError(error);
      })
    );
  }

  private setLoading(loading: boolean) {
    this.loadingSubject.next(loading);
  }

  private setError(error: string) {
    this.errorSubject.next(error);
  }

  private clearError() {
    this.errorSubject.next(null);
  }
}

// Component using reactive data flow
@Component({
  selector: 'app-data-list',
  template: `
    <div class="data-list">
      <div *ngIf="state$ | async as state">
        <div *ngIf="state.loading" class="loading">Loading...</div>
        <div *ngIf="state.error" class="error">{{ state.error }}</div>
        <div *ngIf="state.isEmpty" class="empty">No data available</div>
        <div *ngIf="state.hasData" class="data-container">
          <div *ngFor="let item of state.data" class="data-item">
            {{ item.name }}
```

```
          </div>
        </div>
      </div>
    </div>
  `
})
export class DataListComponent implements OnInit {
  state$ = this.dataService.state$;

  constructor(private dataService: ReactiveDataService) {}

  ngOnInit() {
    this.dataService.loadData().subscribe();
  }
}
```

State Management with NgRx

Description: Implements centralized state management using NgRx for complex applications.

```
// State
export interface AppState {
  users: UserState;
  loading: LoadingState;
}

export interface UserState {
  users: User[];
  selectedUser: User | null;
  error: string | null;
}

export interface LoadingState {
  isLoading: boolean;
  loadingMessage: string | null;
}

// Actions
export const UserActions = createActionGroup({
  source: 'Users',
  events: {
    'Load Users': emptyProps(),
    'Load Users Success': props<{ users: User[] }>(),
    'Load Users Failure': props<{ error: string }>(),
    'Select User': props<{ user: User }>(),
    'Add User': props<{ user: User }>(),
    'Update User': props<{ user: User }>(),
    'Delete User': props<{ userId: number }>()
  }
});

// Reducer
export const userReducer = createReducer(
  initialState,
  on(UserActions.loadUsers, (state) => ({
    ...state,
    error: null
  })),
  on(UserActions.loadUsersSuccess, (state, { users }) => ({
    ...state,
    users,
    error: null
  })),
```

```
  on(UserActions.loadUsersFailure, (state, { error }) => ({
    ...state,
    error
  })),
  on(UserActions.selectUser, (state, { user }) => ({
    ...state,
    selectedUser: user
  })),
  on(UserActions.addUser, (state, { user }) => ({
    ...state,
    users: [...state.users, user]
  })),
  on(UserActions.updateUser, (state, { user }) => ({
    ...state,
    users: state.users.map(u => u.id === user.id ? user : u)
  })),
  on(UserActions.deleteUser, (state, { userId }) => ({
    ...state,
    users: state.users.filter(u => u.id !== userId)
  }))
);

// Effects
@Injectable()
export class UserEffects {
  loadUsers$ = createEffect(() =>
    this.actions$.pipe(
      ofType(UserActions.loadUsers),
      switchMap(() =>
        this.userService.getUsers().pipe(
          map(users => UserActions.loadUsersSuccess({ users })),
          catchError(error => of(UserActions.loadUsersFailure({
            error: error.message
          })))
        )
      )
    )
  );

  constructor(
    private actions$: Actions,
    private userService: UserService
  ) {}
```

```
}

// Selectors
export const selectUserState = (state: AppState) => state.users;

export const selectAllUsers = createSelector(
  selectUserState,
  (state: UserState) => state.users
);

export const selectSelectedUser = createSelector(
  selectUserState,
  (state: UserState) => state.selectedUser
);

export const selectUserError = createSelector(
  selectUserState,
  (state: UserState) => state.error
);

// Component using NgRx
@Component({
  selector: 'app-user-list',
  template: `
    <div class="user-list">
      <div *ngIf="loading$ | async" class="loading">Loading users...</div>
      <div *ngIf="error$ | async as error" class="error">{{ error }}</div>
      <div *ngIf="users$ | async as users">
        <div *ngFor="let user of users" class="user-item">
          {{ user.name }} - {{ user.email }}
          <button (click)="selectUser(user)">Select</button>
        </div>
      </div>
    </div>
  `
})
export class UserListComponent implements OnInit {
  users$ = this.store.select(selectAllUsers);
  loading$ = this.store.select(selectLoading);
  error$ = this.store.select(selectUserError);

  constructor(private store: Store<AppState>) {}
```

```
  ngOnInit() {
    this.store.dispatch(UserActions.loadUsers());
  }

  selectUser(user: User) {
    this.store.dispatch(UserActions.selectUser({ user }));
  }
}
```

4. State Management Patterns

Local Component State

Description: Manages state within individual components using Angular's built-in state management.

```
// Component with local state
@Component({
  selector: 'app-user-form',
  template: `
    <form [formGroup]="userForm" (ngSubmit)="onSubmit()">
      <div class="form-group">
        <label>Name:</label>
        <input formControlName="name" type="text">
        <div *ngIf="userForm.get('name')?.hasError('required')" class="error">
          Name is required
        </div>
      </div>

      <div class="form-group">
        <label>Email:</label>
        <input formControlName="email" type="email">
        <div *ngIf="userForm.get('email')?.hasError('email')" class="error">
          Invalid email format
        </div>
      </div>

      <button type="submit" [disabled]="userForm.invalid || isSubmitting">
        {{ isSubmitting ? 'Saving...' : 'Save User' }}
      </button>
    </form>
  `
})
export class UserFormComponent implements OnInit {
  userForm: FormGroup;
  isSubmitting = false;
  private user: User | null = null;

  constructor(
    private fb: FormBuilder,
    private userService: UserService,
    private route: ActivatedRoute
  ) {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]]
    });
  }
```

```
  ngOnInit() {
    // Load user data if editing
    this.route.params.pipe(
      switchMap(params => {
        const userId = params['id'];
        return userId ? this.userService.getUserById(+userId) : of(null);
      })
    ).subscribe(user => {
      this.user = user;
      if (user) {
        this.userForm.patchValue(user);
      }
    });
  }

  onSubmit() {
    if (this.userForm.valid) {
      this.isSubmitting = true;
      const userData = this.userForm.value;

      const operation = this.user
        ? this.userService.updateUser({ ...userData, id: this.user.id })
        : this.userService.addUser(userData);

      operation.subscribe({
        next: (user) => {
          this.isSubmitting = false;
          console.log('User saved:', user);
        },
        error: (error) => {
          this.isSubmitting = false;
          console.error('Error saving user:', error);
        }
      });
    }
  }
}
```

Shared State with Services

Description: Uses services to share state between multiple components.

```typescript
// Shared State Service
@Injectable({
  providedIn: 'root'
})
export class SharedStateService {
  // Private subjects for state
  private userSubject = new BehaviorSubject<User | null>(null);
  private themeSubject = new BehaviorSubject<'light' | 'dark'>('light');
  private sidebarOpenSubject = new BehaviorSubject<boolean>(false);

  // Public observables
  currentUser$ = this.userSubject.asObservable();
  theme$ = this.themeSubject.asObservable();
  sidebarOpen$ = this.sidebarOpenSubject.asObservable();

  // State getters
  get currentUser(): User | null {
    return this.userSubject.value;
  }

  get currentTheme(): 'light' | 'dark' {
    return this.themeSubject.value;
  }

  get isSidebarOpen(): boolean {
    return this.sidebarOpenSubject.value;
  }

  // State setters
  setCurrentUser(user: User | null) {
    this.userSubject.next(user);
  }

  toggleTheme() {
    const newTheme = this.currentTheme === 'light' ? 'dark' : 'light';
    this.themeSubject.next(newTheme);
  }

  toggleSidebar() {
    this.sidebarOpenSubject.next(!this.isSidebarOpen);
  }

  // Computed state
```

```
  isLoggedIn$ = this.currentUser$.pipe(
    map(user => user !== null)
  );

  userDisplayName$ = this.currentUser$.pipe(
    map(user => user ? `${user.firstName} ${user.lastName}` : 'Guest')
  );
}

// Header Component using shared state
@Component({
  selector: 'app-header',
  template: `
    <header class="header">
      <div class="header-content">
        <button (click)="toggleSidebar()" class="menu-button">
          <i class="menu-icon"></i>
        </button>

        <h1 class="app-title">My App</h1>

        <div class="header-actions">
          <button (click)="toggleTheme()" class="theme-toggle">
            {{ (theme$ | async) === 'light' ? '🌙' : '🌞' }}
          </button>

          <div class="user-info" *ngIf="isLoggedIn$ | async">
            <span>{{ userDisplayName$ | async }}</span>
            <button (click)="logout()">Logout</button>
          </div>
        </div>
      </div>
    </header>
  `
})
export class HeaderComponent {
  theme$ = this.sharedState.theme$;
  isLoggedIn$ = this.sharedState.isLoggedIn$;
  userDisplayName$ = this.sharedState.userDisplayName$;

  constructor(private sharedState: SharedStateService) {}

  toggleSidebar() {
```

```
    this.sharedState.toggleSidebar();
  }

  toggleTheme() {
    this.sharedState.toggleTheme();
  }

  logout() {
    this.sharedState.setCurrentUser(null);
  }
}
```

5. Routing Patterns

Feature-Based Routing

Description: Organizes routes by feature modules for better maintainability and lazy loading.

```
// App Routing Module
const routes: Routes = [
  {
    path: '',
    redirectTo: '/dashboard',
    pathMatch: 'full'
  },
  {
    path: 'dashboard',
    loadChildren: () => import('./features/dashboard/dashboard.module').then(m => m.Dashb
  },
  {
    path: 'users',
    loadChildren: () => import('./features/users/users.module').then(m => m.UsersModule)
  },
  {
    path: 'products',
    loadChildren: () => import('./features/products/products.module').then(m => m.Product
  },
  {
    path: 'orders',
    loadChildren: () => import('./features/orders/orders.module').then(m => m.OrdersModul
  },
  {
    path: '**',
    component: NotFoundComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {
    enableTracing: false,
    preloadingStrategy: PreloadAllModules
  })],
  exports: [RouterModule]
})
export class AppRoutingModule {}

// Feature Module Routing (Users)
const routes: Routes = [
  {
    path: '',
    component: UserListComponent,
```

```
    data: { title: 'Users' }
  },
  {
    path: 'create',
    component: UserCreateComponent,
    data: { title: 'Create User' }
  },
  {
    path: ':id',
    component: UserDetailComponent,
    data: { title: 'User Details' }
  },
  {
    path: ':id/edit',
    component: UserEditComponent,
    data: { title: 'Edit User' }
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class UsersRoutingModule {}
```

Route Guards and Resolvers

Description: Implements route protection and data preloading using guards and resolvers.

```
// Auth Guard
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(
    private authService: AuthService,
    private router: Router
  ) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | UrlTree {
    if (this.authService.isAuthenticated()) {
      return true;
    }

    // Redirect to login with return URL
    return this.router.createUrlTree(['/login'], {
      queryParams: { returnUrl: state.url }
    });
  }

  canActivateChild(
    childRoute: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | UrlTree {
    return this.canActivate(childRoute, state);
  }
}

// Role Guard
@Injectable({
  providedIn: 'root'
})
export class RoleGuard implements CanActivate {
  constructor(
    private authService: AuthService,
    private router: Router
  ) {}

  canActivate(route: ActivatedRouteSnapshot): boolean {
```

```
    const expectedRoles = route.data['roles'] as string[];
    const userRole = this.authService.getCurrentUserRole();

    if (expectedRoles.includes(userRole)) {
      return true;
    }

    this.router.navigate(['/unauthorized']);
    return false;
  }
}

// Data Resolver
@Injectable({
  providedIn: 'root'
})
export class UserResolver implements Resolve<User> {
  constructor(private userService: UserService) {}

  resolve(route: ActivatedRouteSnapshot): Observable<User> {
    const userId = route.paramMap.get('id');
    if (!userId) {
      throw new Error('User ID is required');
    }

    return this.userService.getUserById(+userId).pipe(
      catchError(error => {
        console.error('Error resolving user:', error);
        return throwError(error);
      })
    );
  }
}

// Routes with guards and resolvers
const routes: Routes = [
  {
    path: 'users',
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        component: UserListComponent
```

```
      },
      {
        path: 'create',
        component: UserCreateComponent,
        canActivate: [RoleGuard],
        data: { roles: ['admin', 'manager'] }
      },
      {
        path: ':id',
        component: UserDetailComponent,
        resolve: { user: UserResolver }
      },
      {
        path: ':id/edit',
        component: UserEditComponent,
        canActivate: [RoleGuard],
        data: { roles: ['admin', 'manager'] },
        resolve: { user: UserResolver }
      }
    ]
  }
];
```

Dynamic Routing

Description: Creates routes dynamically based on configuration or user permissions.

```
// Dynamic Route Configuration
@Injectable({
  providedIn: 'root'
})
export class DynamicRouteService {
  private routes: Route[] = [];

  constructor(
    private router: Router,
    private configService: ConfigService
  ) {}

  initializeRoutes(): Observable<Route[]> {
    return this.configService.getRouteConfig().pipe(
      tap(config => {
        this.routes = this.buildRoutesFromConfig(config);
        this.updateRouterConfig();
      }),
      map(() => this.routes)
    );
  }

  private buildRoutesFromConfig(config: RouteConfig[]): Route[] {
    return config.map(routeConfig => ({
      path: routeConfig.path,
      component: this.getComponent(routeConfig.component),
      canActivate: this.getGuards(routeConfig.guards),
      data: routeConfig.data,
      children: routeConfig.children ? this.buildRoutesFromConfig(routeConfig.children) :
    }));
  }

  private getComponent(componentName: string): Type<any> {
    const componentMap: { [key: string]: Type<any> } = {
      'UserListComponent': UserListComponent,
      'ProductListComponent': ProductListComponent,
      'DashboardComponent': DashboardComponent
    };

    return componentMap[componentName] || NotFoundComponent;
  }

  private getGuards(guardNames: string[]): any[] {
```

```
    const guardMap: { [key: string]: any } = {
      'AuthGuard': AuthGuard,
      'RoleGuard': RoleGuard
    };

    return guardNames.map(name => guardMap[name]).filter(Boolean);
  }

  private updateRouterConfig() {
    const currentConfig = this.router.config;
    const newConfig = [...currentConfig, ...this.routes];
    this.router.resetConfig(newConfig);
  }
}

// App Component initializing dynamic routes
@Component({
  selector: 'app-root',
  template: `
    <div class="app-container">
      <app-header></app-header>
      <main class="main-content">
        <router-outlet></router-outlet>
      </main>
    </div>
  `
})
export class AppComponent implements OnInit {
  constructor(private dynamicRouteService: DynamicRouteService) {}

  ngOnInit() {
    this.dynamicRouteService.initializeRoutes().subscribe();
  }
}
```

6. Form Patterns

Reactive Forms with Validation

Description: Implements complex forms using Angular's reactive forms with comprehensive validation.

```
// Complex Form Component
@Component({
  selector: 'app-user-form',
  template: `
    <form [formGroup]="userForm" (ngSubmit)="onSubmit()">
      <div class="form-section">
        <h3>Personal Information</h3>

        <div class="form-group">
          <label>First Name *</label>
          <input formControlName="firstName" type="text" class="form-control">
          <div *ngIf="firstName?.invalid && firstName?.touched" class="error-message">
            <div *ngIf="firstName?.errors?.['required']">First name is required</div>
            <div *ngIf="firstName?.errors?.['minlength']">
              First name must be at least 2 characters
            </div>
          </div>
        </div>

        <div class="form-group">
          <label>Last Name *</label>
          <input formControlName="lastName" type="text" class="form-control">
          <div *ngIf="lastName?.invalid && lastName?.touched" class="error-message">
            <div *ngIf="lastName?.errors?.['required']">Last name is required</div>
          </div>
        </div>

        <div class="form-group">
          <label>Email *</label>
          <input formControlName="email" type="email" class="form-control">
          <div *ngIf="email?.invalid && email?.touched" class="error-message">
            <div *ngIf="email?.errors?.['required']">Email is required</div>
            <div *ngIf="email?.errors?.['email']">Invalid email format</div>
            <div *ngIf="email?.errors?.['emailExists']">Email already exists</div>
          </div>
        </div>
      </div>

      <div class="form-section">
        <h3>Address Information</h3>

        <div formGroupName="address">
          <div class="form-group">
```

```
      <label>Street</label>
      <input formControlName="street" type="text" class="form-control">
    </div>

    <div class="form-group">
      <label>City</label>
      <input formControlName="city" type="text" class="form-control">
    </div>

    <div class="form-group">
      <label>Country</label>
      <select formControlName="country" class="form-control">
        <option value="">Select Country</option>
        <option *ngFor="let country of countries" [value]="country.code">
          {{ country.name }}
        </option>
      </select>
    </div>
  </div>
</div>

<div class="form-section">
  <h3>Preferences</h3>

  <div formArrayName="preferences">
    <div *ngFor="let preference of preferences.controls; let i = index"
         [formGroupName]="i" class="preference-item">
      <input formControlName="name" type="text" placeholder="Preference name">
      <input formControlName="value" type="text" placeholder="Preference value">
      <button type="button" (click)="removePreference(i)">Remove</button>
    </div>
    <button type="button" (click)="addPreference()">Add Preference</button>
  </div>
</div>

<div class="form-actions">
  <button type="button" (click)="resetForm()" [disabled]="isSubmitting">
    Reset
  </button>
  <button type="submit" [disabled]="userForm.invalid || isSubmitting">
    {{ isSubmitting ? 'Saving...' : 'Save User' }}
  </button>
</div>
```

```
    </form>
  `
})
export class UserFormComponent implements OnInit {
  userForm: FormGroup;
  isSubmitting = false;
  countries = [
    { code: 'US', name: 'United States' },
    { code: 'CA', name: 'Canada' },
    { code: 'UK', name: 'United Kingdom' }
  ];

  constructor(
    private fb: FormBuilder,
    private userService: UserService,
    private route: ActivatedRoute
  ) {
    this.userForm = this.createForm();
  }

  ngOnInit() {
    // Load user data if editing
    this.route.params.pipe(
      switchMap(params => {
        const userId = params['id'];
        return userId ? this.userService.getUserById(+userId) : of(null);
      })
    ).subscribe(user => {
      if (user) {
        this.userForm.patchValue(user);
        this.loadUserPreferences(user.id);
      }
    });
  }

  private createForm(): FormGroup {
    return this.fb.group({
      firstName: ['', [Validators.required, Validators.minLength(2)]],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email], [this.emailExistsValidator.bin
      address: this.fb.group({
        street: [''],
        city: [''],
```

```
      country: ['']
    }),
    preferences: this.fb.array([])
  });
}

// Custom async validator
emailExistsValidator(control: AbstractControl): Observable<ValidationErrors | null> {
  if (!control.value) {
    return of(null);
  }

  return this.userService.checkEmailExists(control.value).pipe(
    map(exists => exists ? { emailExists: true } : null),
    catchError(() => of(null))
  );
}

get firstName() { return this.userForm.get('firstName'); }
get lastName() { return this.userForm.get('lastName'); }
get email() { return this.userForm.get('email'); }
get preferences() { return this.userForm.get('preferences') as FormArray; }

addPreference() {
  const preferenceGroup = this.fb.group({
    name: ['', Validators.required],
    value: ['', Validators.required]
  });
  this.preferences.push(preferenceGroup);
}

removePreference(index: number) {
  this.preferences.removeAt(index);
}

private loadUserPreferences(userId: number) {
  this.userService.getUserPreferences(userId).subscribe(preferences => {
    preferences.forEach(preference => {
      this.addPreference();
      const lastIndex = this.preferences.length - 1;
      this.preferences.at(lastIndex).patchValue(preference);
    });
  });
```

```
  }

  onSubmit() {
    if (this.userForm.valid) {
      this.isSubmitting = true;
      const formData = this.userForm.value;

      const operation = formData.id
        ? this.userService.updateUser(formData)
        : this.userService.addUser(formData);

      operation.subscribe({
        next: (user) => {
          this.isSubmitting = false;
          console.log('User saved:', user);
        },
        error: (error) => {
          this.isSubmitting = false;
          console.error('Error saving user:', error);
        }
      });
    }
  }

  resetForm() {
    this.userForm.reset();
    this.preferences.clear();
  }
}
```

Dynamic Form Generation

Description: Creates forms dynamically based on configuration or API responses.

```typescript
// Dynamic Form Configuration
export interface FormFieldConfig {
  key: string;
  type: 'text' | 'email' | 'password' | 'number' | 'select' | 'checkbox' | 'textarea';
  label: string;
  placeholder?: string;
  required?: boolean;
  validators?: ValidatorFn[];
  options?: { value: any; label: string }[];
  conditional?: {
    field: string;
    value: any;
    operator: 'equals' | 'notEquals' | 'contains';
  };
}

// Dynamic Form Service
@Injectable({
  providedIn: 'root'
})
export class DynamicFormService {
  constructor(private fb: FormBuilder) {}

  createFormFromConfig(config: FormFieldConfig[]): FormGroup {
    const formControls: { [key: string]: AbstractControl } = {};

    config.forEach(field => {
      const validators = field.validators || [];
      if (field.required) {
        validators.push(Validators.required);
      }

      formControls[field.key] = this.fb.control('', validators);
    });

    return this.fb.group(formControls);
  }

  getFormFieldConfig(): Observable<FormFieldConfig[]> {
    // This would typically come from an API
    return of([
      {
        key: 'firstName',
```

```
        type: 'text',
        label: 'First Name',
        placeholder: 'Enter your first name',
        required: true,
        validators: [Validators.minLength(2)]
      },
      {
        key: 'email',
        type: 'email',
        label: 'Email Address',
        placeholder: 'Enter your email',
        required: true,
        validators: [Validators.email]
      },
      {
        key: 'country',
        type: 'select',
        label: 'Country',
        required: true,
        options: [
          { value: 'US', label: 'United States' },
          { value: 'CA', label: 'Canada' },
          { value: 'UK', label: 'United Kingdom' }
        ]
      },
      {
        key: 'newsletter',
        type: 'checkbox',
        label: 'Subscribe to Newsletter',
        required: false
      },
      {
        key: 'comments',
        type: 'textarea',
        label: 'Comments',
        placeholder: 'Enter your comments here'
      }
    ]);
  }
}

// Dynamic Form Component
@Component({
```

```
selector: 'app-dynamic-form',
template: `
  <form [formGroup]="dynamicForm" (ngSubmit)="onSubmit()">
    <div *ngFor="let field of formFields" class="form-group">
      <label [for]="field.key">{{ field.label }}</label>

      <!-- Text Input -->
      <input
        *ngIf="field.type === 'text' || field.type === 'email' || field.type === 'passw
        [id]="field.key"
        [type]="field.type"
        [formControlName]="field.key"
        [placeholder]="field.placeholder"
        class="form-control">

      <!-- Number Input -->
      <input
        *ngIf="field.type === 'number'"
        [id]="field.key"
        type="number"
        [formControlName]="field.key"
        [placeholder]="field.placeholder"
        class="form-control">

      <!-- Select -->
      <select
        *ngIf="field.type === 'select'"
        [id]="field.key"
        [formControlName]="field.key"
        class="form-control">
        <option value="">Select {{ field.label }}</option>
        <option *ngFor="let option of field.options" [value]="option.value">
          {{ option.label }}
        </option>
      </select>

      <!-- Checkbox -->
      <input
        *ngIf="field.type === 'checkbox'"
        [id]="field.key"
        type="checkbox"
        [formControlName]="field.key"
        class="form-check-input">
```

```
        <!-- Textarea -->
        <textarea
          *ngIf="field.type === 'textarea'"
          [id]="field.key"
          [formControlName]="field.key"
          [placeholder]="field.placeholder"
          class="form-control"
          rows="4">
        </textarea>

        <!-- Error Messages -->
        <div *ngIf="getFieldControl(field.key)?.invalid && getFieldControl(field.key)?.to
             class="error-message">
          <div *ngIf="getFieldControl(field.key)?.errors?.['required']">
            {{ field.label }} is required
          </div>
          <div *ngIf="getFieldControl(field.key)?.errors?.['email']">
            Invalid email format
          </div>
          <div *ngIf="getFieldControl(field.key)?.errors?.['minlength']">
            {{ field.label }} must be at least {{ getFieldControl(field.key)?.errors?.['m
          </div>
        </div>
      </div>

      <div class="form-actions">
        <button type="submit" [disabled]="dynamicForm.invalid">
          Submit
        </button>
        <button type="button" (click)="resetForm()">
          Reset
        </button>
      </div>
    </form>
  `
})
export class DynamicFormComponent implements OnInit {
  dynamicForm: FormGroup;
  formFields: FormFieldConfig[] = [];

  constructor(
    private dynamicFormService: DynamicFormService
```

```
  ) {}

  ngOnInit() {
    this.dynamicFormService.getFormFieldConfig().subscribe(config => {
      this.formFields = config;
      this.dynamicForm = this.dynamicFormService.createFormFromConfig(config);
    });
  }

  getFieldControl(fieldKey: string): AbstractControl | null {
    return this.dynamicForm.get(fieldKey);
  }

  onSubmit() {
    if (this.dynamicForm.valid) {
      console.log('Form submitted:', this.dynamicForm.value);
    }
  }

  resetForm() {
    this.dynamicForm.reset();
  }
}
```

7. HTTP & API Patterns

HTTP Interceptors

Description: Implements cross-cutting concerns like authentication, error handling, and logging using HTTP interceptors.

```
// Authentication Interceptor
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const authToken = this.authService.getToken();

    if (authToken) {
      const authReq = req.clone({
        headers: req.headers.set('Authorization', `Bearer ${authToken}`)
      });
      return next.handle(authReq);
    }

    return next.handle(req);
  }
}

// Error Handling Interceptor
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  constructor(
    private notificationService: NotificationService,
    private router: Router
  ) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        let errorMessage = 'An error occurred';

        if (error.error instanceof ErrorEvent) {
          // Client-side error
          errorMessage = error.error.message;
        } else {
          // Server-side error
          switch (error.status) {
            case 401:
              errorMessage = 'Unauthorized access';
              this.router.navigate(['/login']);
              break;
            case 403:
```

```
              errorMessage = 'Access forbidden';
              break;
            case 404:
              errorMessage = 'Resource not found';
              break;
            case 500:
              errorMessage = 'Internal server error';
              break;
            default:
              errorMessage = error.error?.message || `Error ${error.status}`;
          }
        }

        this.notificationService.showError(errorMessage);
        return throwError(error);
      })
    );
  }
}

// Loading Interceptor
@Injectable()
export class LoadingInterceptor implements HttpInterceptor {
  private activeRequests = 0;

  constructor(private loadingService: LoadingService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Skip loading for certain requests
    if (req.url.includes('/api/heartbeat')) {
      return next.handle(req);
    }

    this.activeRequests++;
    this.loadingService.setLoading(true);

    return next.handle(req).pipe(
      finalize(() => {
        this.activeRequests--;
        if (this.activeRequests === 0) {
          this.loadingService.setLoading(false);
        }
      })
```

```
    );
  }
}

// Logging Interceptor
@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
  constructor(private logger: LoggerService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const startTime = Date.now();

    this.logger.log(`HTTP Request: ${req.method} ${req.url}`);

    return next.handle(req).pipe(
      tap(event => {
        if (event instanceof HttpResponse) {
          const duration = Date.now() - startTime;
          this.logger.log(`HTTP Response: ${req.method} ${req.url} - ${event.status} (${d
        }
      }),
      catchError(error => {
        const duration = Date.now() - startTime;
        this.logger.error(`HTTP Error: ${req.method} ${req.url} - ${error.status} (${dura
        return throwError(error);
      })
    );
  }
}

// Module Configuration
@NgModule({
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: ErrorInterceptor,
      multi: true
    },
```

```
    {
      provide: HTTP_INTERCEPTORS,
      useClass: LoadingInterceptor,
      multi: true
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: LoggingInterceptor,
      multi: true
    }
  ]
})
export class InterceptorModule {}
```

API Service Patterns

Description: Implements consistent API communication patterns with proper error handling and caching.

```
// Base API Service
export abstract class BaseApiService<T> {
  protected baseUrl: string;

  constructor(
    protected http: HttpClient,
    protected endpoint: string
  ) {
    this.baseUrl = `${environment.apiUrl}/${endpoint}`;
  }

  getAll(): Observable<T[]> {
    return this.http.get<T[]>(this.baseUrl).pipe(
      retry(2),
      catchError(this.handleError)
    );
  }

  getById(id: number): Observable<T> {
    return this.http.get<T>(`${this.baseUrl}/${id}`).pipe(
      retry(2),
      catchError(this.handleError)
    );
  }

  create(item: Partial<T>): Observable<T> {
    return this.http.post<T>(this.baseUrl, item).pipe(
      catchError(this.handleError)
    );
  }

  update(id: number, item: Partial<T>): Observable<T> {
    return this.http.put<T>(`${this.baseUrl}/${id}`, item).pipe(
      catchError(this.handleError)
    );
  }

  delete(id: number): Observable<void> {
    return this.http.delete<void>(`${this.baseUrl}/${id}`).pipe(
      catchError(this.handleError)
    );
  }
```

```
  protected handleError(error: HttpErrorResponse): Observable<never> {
    console.error('API Error:', error);
    return throwError(() => new Error(error.message || 'Server error'));
  }
}

// User API Service
@Injectable({
  providedIn: 'root'
})
export class UserApiService extends BaseApiService<User> {
  constructor(http: HttpClient) {
    super(http, 'users');
  }

  searchUsers(query: string): Observable<User[]> {
    return this.http.get<User[]>(`${this.baseUrl}/search`, {
      params: { q: query }
    }).pipe(
      retry(2),
      catchError(this.handleError)
    );
  }

  getUserProfile(userId: number): Observable<UserProfile> {
    return this.http.get<UserProfile>(`${this.baseUrl}/${userId}/profile`).pipe(
      retry(2),
      catchError(this.handleError)
    );
  }

  updateUserProfile(userId: number, profile: Partial<UserProfile>): Observable<UserProfil
    return this.http.put<UserProfile>(`${this.baseUrl}/${userId}/profile`, profile).pipe(
      catchError(this.handleError)
    );
  }

  uploadAvatar(userId: number, file: File): Observable<{ avatarUrl: string }> {
    const formData = new FormData();
    formData.append('avatar', file);

    return this.http.post<{ avatarUrl: string }>(`${this.baseUrl}/${userId}/avatar`, formD
      catchError(this.handleError)
```

```
    );
  }
}

// Cached API Service
@Injectable({
  providedIn: 'root'
})
export class CachedApiService<T> extends BaseApiService<T> {
  private cache = new Map<string, Observable<T[]>>();
  private cacheTimeout = 5 * 60 * 1000; // 5 minutes

  constructor(
    http: HttpClient,
    endpoint: string,
    private cacheService: CacheService
  ) {
    super(http, endpoint);
  }

  getAll(): Observable<T[]> {
    const cacheKey = `${this.endpoint}_all`;
    const cached = this.cacheService.get<T[]>(cacheKey);

    if (cached) {
      return of(cached);
    }

    const request$ = super.getAll().pipe(
      tap(data => {
        this.cacheService.set(cacheKey, data, this.cacheTimeout);
      }),
      shareReplay(1)
    );

    this.cache.set(cacheKey, request$);
    return request$;
  }

  getById(id: number): Observable<T> {
    const cacheKey = `${this.endpoint}_${id}`;
    const cached = this.cacheService.get<T>(cacheKey);
```

```
    if (cached) {
      return of(cached);
    }

    return super.getById(id).pipe(
      tap(data => {
        this.cacheService.set(cacheKey, data, this.cacheTimeout);
      })
    );
  }

  invalidateCache(): void {
    this.cache.clear();
    this.cacheService.clear();
  }
}
```

8. Testing Patterns

Component Testing Patterns

Description: Implements comprehensive testing strategies for Angular components.

```
// Component Test Example
describe('UserListComponent', () => {
  let component: UserListComponent;
  let fixture: ComponentFixture<UserListComponent>;
  let userService: jasmine.SpyObj<UserService>;
  let router: jasmine.SpyObj<Router>;

  beforeEach(async () => {
    const userServiceSpy = jasmine.createSpyObj('UserService', [
      'getUsers', 'deleteUser', 'searchUsers'
    ]);
    const routerSpy = jasmine.createSpyObj('Router', ['navigate']);

    await TestBed.configureTestingModule({
      declarations: [UserListComponent, UserTableComponent, UserSearchComponent],
      providers: [
        { provide: UserService, useValue: userServiceSpy },
        { provide: Router, useValue: routerSpy }
      ],
      imports: [CommonModule, FormsModule]
    }).compileComponents();

    fixture = TestBed.createComponent(UserListComponent);
    component = fixture.componentInstance;
    userService = TestBed.inject(UserService) as jasmine.SpyObj<UserService>;
    router = TestBed.inject(Router) as jasmine.SpyObj<Router>;
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should load users on init', () => {
    const mockUsers = [
      { id: 1, name: 'John Doe', email: 'john@example.com' },
      { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
    ];
    userService.getUsers.and.returnValue(of(mockUsers));

    component.ngOnInit();

    expect(userService.getUsers).toHaveBeenCalled();
    expect(component.users).toEqual(mockUsers);
```

```
  expect(component.isLoading).toBeFalse();
});

it('should handle search', () => {
  const searchTerm = 'john';
  const mockUsers = [{ id: 1, name: 'John Doe', email: 'john@example.com' }];
  userService.searchUsers.and.returnValue(of(mockUsers));

  component.onSearch(searchTerm);

  expect(userService.searchUsers).toHaveBeenCalledWith(searchTerm);
  expect(component.users).toEqual(mockUsers);
  expect(component.isLoading).toBeFalse();
});

it('should navigate to edit user', () => {
  const user = { id: 1, name: 'John Doe', email: 'john@example.com' };

  component.onEditUser(user);

  expect(router.navigate).toHaveBeenCalledWith(['/users', user.id, 'edit']);
});

it('should delete user and reload list', () => {
  const user = { id: 1, name: 'John Doe', email: 'john@example.com' };
  const remainingUsers = [{ id: 2, name: 'Jane Smith', email: 'jane@example.com' }];

  userService.deleteUser.and.returnValue(of(void 0));
  userService.getUsers.and.returnValue(of(remainingUsers));

  component.onDeleteUser(user);

  expect(userService.deleteUser).toHaveBeenCalledWith(user.id);
  expect(userService.getUsers).toHaveBeenCalledTimes(2); // Once on init, once after de
  expect(component.users).toEqual(remainingUsers);
});

it('should display loading state', () => {
  userService.getUsers.and.returnValue(of([]).pipe(delay(100)));

  component.ngOnInit();
  expect(component.isLoading).toBeTrue();
```

```
    fixture.detectChanges();
    const loadingElement = fixture.debugElement.query(By.css('.loading'));
    expect(loadingElement).toBeTruthy();
  });
});

// Service Test Example
describe('UserService', () => {
  let service: UserService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });
    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify();
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('should get users', () => {
    const mockUsers = [
      { id: 1, name: 'John Doe', email: 'john@example.com' },
      { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
    ];

    service.getUsers().subscribe(users => {
      expect(users).toEqual(mockUsers);
    });

    const req = httpMock.expectOne('/api/users');
    expect(req.request.method).toBe('GET');
    req.flush(mockUsers);
  });
```

```
  it('should handle error when getting users', () => {
    service.getUsers().subscribe({
      next: () => fail('should have failed'),
      error: (error) => {
        expect(error).toBeTruthy();
      }
    });

    const req = httpMock.expectOne('/api/users');
    req.flush('Error', { status: 500, statusText: 'Server Error' });
  });

  it('should add user', () => {
    const newUser = { name: 'John Doe', email: 'john@example.com' };
    const createdUser = { id: 1, ...newUser };

    service.addUser(newUser).subscribe(user => {
      expect(user).toEqual(createdUser);
    });

    const req = httpMock.expectOne('/api/users');
    expect(req.request.method).toBe('POST');
    expect(req.request.body).toEqual(newUser);
    req.flush(createdUser);
  });
});

// Integration Test Example
describe('User Management Integration', () => {
  let fixture: ComponentFixture<UserListComponent>;
  let component: UserListComponent;
  let httpMock: HttpTestingController;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [UserListComponent, UserTableComponent, UserSearchComponent],
      imports: [HttpClientTestingModule, RouterTestingModule, CommonModule, FormsModule],
      providers: [UserService]
    }).compileComponents();

    fixture = TestBed.createComponent(UserListComponent);
    component = fixture.componentInstance;
    httpMock = TestBed.inject(HttpTestingController);
```

```
});

it('should complete user workflow', fakeAsync(() => {
  const mockUsers = [
    { id: 1, name: 'John Doe', email: 'john@example.com' },
    { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
  ];

  // Load users
  component.ngOnInit();

  const loadReq = httpMock.expectOne('/api/users');
  loadReq.flush(mockUsers);

  tick();
  fixture.detectChanges();

  expect(component.users).toEqual(mockUsers);

  // Search users
  component.onSearch('john');

  const searchReq = httpMock.expectOne('/api/users/search?q=john');
  searchReq.flush([mockUsers[0]]);

  tick();
  fixture.detectChanges();

  expect(component.users).toEqual([mockUsers[0]]);

  // Delete user
  component.onDeleteUser(mockUsers[0]);

  const deleteReq = httpMock.expectOne('/api/users/1');
  deleteReq.flush(null);

  const reloadReq = httpMock.expectOne('/api/users');
  reloadReq.flush([mockUsers[1]]);

  tick();
  fixture.detectChanges();

  expect(component.users).toEqual([mockUsers[1]]);
```

```
  }));
});
```

9. Module Architecture

Feature Module Pattern

Description: Organizes application into feature modules for better maintainability and lazy loading.

```
// Feature Module Structure
@NgModule({
  declarations: [
    UserListComponent,
    UserDetailComponent,
    UserCreateComponent,
    UserEditComponent,
    UserTableComponent,
    UserSearchComponent
  ],
  imports: [
    CommonModule,
    UsersRoutingModule,
    SharedModule,
    ReactiveFormsModule,
    MaterialModule
  ],
  providers: [
    UserService,
    UserResolver
  ]
})
export class UsersModule {}

// Feature Module Routing
const routes: Routes = [
  {
    path: '',
    component: UserListComponent,
    data: { title: 'Users' }
  },
  {
    path: 'create',
    component: UserCreateComponent,
    data: { title: 'Create User' },
    canActivate: [RoleGuard],
    data: { roles: ['admin', 'manager'] }
  },
  {
    path: ':id',
    component: UserDetailComponent,
    data: { title: 'User Details' },
    resolve: { user: UserResolver }
```

```
  },
  {
    path: ':id/edit',
    component: UserEditComponent,
    data: { title: 'Edit User' },
    canActivate: [RoleGuard],
    data: { roles: ['admin', 'manager'] },
    resolve: { user: UserResolver }
  }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class UsersRoutingModule {}
```

Shared Module Pattern

Description: Creates reusable components, directives, and pipes for use across feature modules.

```
// Shared Module
@NgModule({
  declarations: [
    LoadingSpinnerComponent,
    ErrorMessageComponent,
    ConfirmDialogComponent,
    HighlightDirective,
    TruncatePipe,
    FormatCurrencyPipe
  ],
  imports: [
    CommonModule,
    MaterialModule,
    ReactiveFormsModule
  ],
  exports: [
    CommonModule,
    MaterialModule,
    ReactiveFormsModule,
    LoadingSpinnerComponent,
    ErrorMessageComponent,
    ConfirmDialogComponent,
    HighlightDirective,
    TruncatePipe,
    FormatCurrencyPipe
  ]
})
export class SharedModule {}

// Reusable Components
@Component({
  selector: 'app-loading-spinner',
  template: `
    <div class="loading-spinner" *ngIf="loading">
      <div class="spinner"></div>
      <p *ngIf="message">{{ message }}</p>
    </div>
  `,
  styles: [`
    .loading-spinner {
      display: flex;
      flex-direction: column;
      align-items: center;
```

```
      justify-content: center;
      padding: 2rem;
    }
    .spinner {
      width: 40px;
      height: 40px;
      border: 4px solid #f3f3f3;
      border-top: 4px solid #3498db;
      border-radius: 50%;
      animation: spin 1s linear infinite;
    }
    @keyframes spin {
      0% { transform: rotate(0deg); }
      100% { transform: rotate(360deg); }
    }
  `]
})
export class LoadingSpinnerComponent {
  @Input() loading = false;
  @Input() message: string;
}

@Component({
  selector: 'app-error-message',
  template: `
    <div class="error-message" *ngIf="error">
      <div class="error-icon">⚠</div>
      <div class="error-content">
        <h4>{{ title || 'Error' }}</h4>
        <p>{{ error }}</p>
        <button *ngIf="retry" (click)="onRetry.emit()" class="retry-button">
          Try Again
        </button>
      </div>
    </div>
  `
})
export class ErrorMessageComponent {
  @Input() error: string;
  @Input() title: string;
  @Input() retry = false;
  @Output() onRetry = new EventEmitter<void>();
}
```

```
// Custom Directives
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() appHighlight: string = 'yellow';
  @Input() highlightClass: string = '';

  constructor(private el: ElementRef) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.appHighlight);
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}

// Custom Pipes
@Pipe({
  name: 'truncate'
})
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit: number = 50, trail: string = '...'): string {
    if (!value) return '';
    return value.length > limit ? value.substring(0, limit) + trail : value;
  }
}

@Pipe({
  name: 'formatCurrency'
})
export class FormatCurrencyPipe implements PipeTransform {
  transform(value: number, currency: string = 'USD', locale: string = 'en-US'): string {
    if (value == null) return '';
    return new Intl.NumberFormat(locale, {
      style: 'currency',
```

```
      currency: currency
    }).format(value);
  }
}
```

10. Performance Patterns

OnPush Change Detection Strategy

Description: Optimizes performance by using OnPush change detection strategy and immutable data patterns.

```
// Optimized Component with OnPush
@Component({
  selector: 'app-user-list',
  template: `
    <div class="user-list">
      <div *ngFor="let user of users; trackBy: trackByUserId" class="user-item">
        <app-user-card [user]="user" (edit)="onEditUser($event)">
        </app-user-card>
      </div>
    </div>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserListComponent implements OnInit {
  users: User[] = [];

  constructor(
    private userService: UserService,
    private cdr: ChangeDetectorRef
  ) {}

  ngOnInit() {
    this.userService.getUsers().subscribe(users => {
      this.users = [...users]; // Create new array reference
      this.cdr.markForCheck(); // Trigger change detection
    });
  }

  onEditUser(user: User) {
    // Handle edit logic
  }

  trackByUserId(index: number, user: User): number {
    return user.id;
  }
}

// Immutable Data Service
@Injectable({
  providedIn: 'root'
})
export class ImmutableUserService {
  private usersSubject = new BehaviorSubject<User[]>([]);
```

```
  users$ = this.usersSubject.asObservable();

  constructor(private http: HttpClient) {}

  addUser(user: User): Observable<User> {
    return this.http.post<User>('/api/users', user).pipe(
      tap(newUser => {
        const currentUsers = this.usersSubject.value;
        // Create new array with spread operator
        this.usersSubject.next([...currentUsers, newUser]);
      })
    );
  }

  updateUser(user: User): Observable<User> {
    return this.http.put<User>(`/api/users/${user.id}`, user).pipe(
      tap(updatedUser => {
        const currentUsers = this.usersSubject.value;
        // Create new array with updated user
        const updatedUsers = currentUsers.map(u =>
          u.id === user.id ? { ...u, ...updatedUser } : u
        );
        this.usersSubject.next(updatedUsers);
      })
    );
  }

  deleteUser(userId: number): Observable<void> {
    return this.http.delete<void>(`/api/users/${userId}`).pipe(
      tap(() => {
        const currentUsers = this.usersSubject.value;
        // Create new array without deleted user
        this.usersSubject.next(currentUsers.filter(u => u.id !== userId));
      })
    );
  }
}
```

Virtual Scrolling

Description: Implements virtual scrolling for large lists to improve performance.

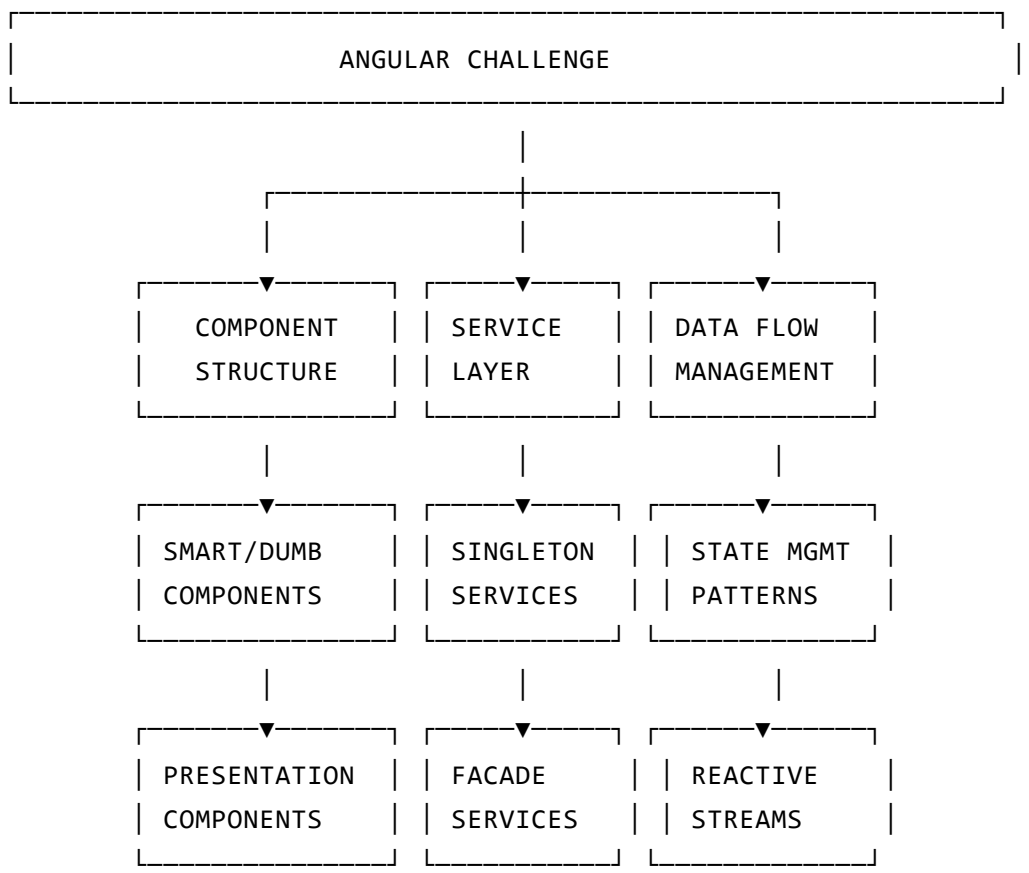
```
// Virtual Scrolling Component
@Component({
  selector: 'app-virtual-user-list',
  template: `
    <cdk-virtual-scroll-viewport itemSize="100" class="viewport">
      <div *cdkVirtualFor="let user of users; trackBy: trackByUserId"
           class="user-item">
        <div class="user-content">
          <img [src]="user.avatar" [alt]="user.name" class="avatar">
          <div class="user-info">
            <h3>{{ user.name }}</h3>
            <p>{{ user.email }}</p>
            <p>{{ user.department }}</p>
          </div>
          <div class="user-actions">
            <button (click)="editUser(user)">Edit</button>
            <button (click)="deleteUser(user)">Delete</button>
          </div>
        </div>
      </div>
    </cdk-virtual-scroll-viewport>
  `,
  styles: [`
    .viewport {
      height: 400px;
      width: 100%;
    }
    .user-item {
      height: 100px;
      display: flex;
      align-items: center;
      padding: 10px;
      border-bottom: 1px solid #eee;
    }
    .user-content {
      display: flex;
      align-items: center;
      width: 100%;
    }
    .avatar {
      width: 50px;
      height: 50px;
      border-radius: 50%;
```

```
      margin-right: 15px;
    }
    .user-info {
      flex: 1;
    }
    .user-actions {
      display: flex;
      gap: 10px;
    }
  `]
})
export class VirtualUserListComponent implements OnInit {
  users: User[] = [];

  constructor(private userService: UserService) {}

  ngOnInit() {
    this.userService.getUsers().subscribe(users => {
      this.users = users;
    });
  }

  trackByUserId(index: number, user: User): number {
    return user.id;
  }

  editUser(user: User) {
    // Handle edit
  }

  deleteUser(user: User) {
    // Handle delete
  }
}

// Module with Virtual Scrolling
@NgModule({
  declarations: [VirtualUserListComponent],
  imports: [
    CommonModule,
    ScrollingModule
  ]
```

```
})
export class VirtualScrollingModule {}
```

Angular Pattern Decision Tree

When building Angular applications...



Pattern Selection Matrix

Scenario	Primary Pattern	Secondary Pattern	When to Use
User Management	Smart/Dumb Components	Reactive Forms	User CRUD operations

Scenario	Primary Pattern	Secondary Pattern	When to Use
Data Display	Virtual Scrolling	OnPush Strategy	Large data lists
State Management	NgRx	Service State	Complex application state
API Communication	HTTP Interceptors	Base API Service	RESTful API integration
Form Handling	Reactive Forms	Dynamic Forms	Complex form validation
Routing	Feature Modules	Route Guards	Multi-page applications
Performance	OnPush Strategy	Virtual Scrolling	Performance-critical apps
Testing	Component Testing	Service Testing	Comprehensive test coverage

Best Practices Checklist

Before Building Angular Applications:

1. **Architecture Planning**
   - Use feature modules for organization
   - Implement lazy loading for performance
   - Plan component hierarchy (Smart/Dumb)
2. **Component Design**
   - Use OnPush change detection strategy
   - Implement trackBy functions for *ngFor
   - Separate concerns (presentation vs logic)
3. **Service Layer**
   - Use singleton services for shared state
   - Implement HTTP interceptors for cross-cutting concerns
   - Use reactive programming with RxJS
4. **State Management**
   - Choose appropriate state management (local vs global)

- Use immutable data patterns
- Implement proper error handling

5. **Performance**
   - Use virtual scrolling for large lists
   - Implement proper change detection strategies
   - Optimize bundle size with lazy loading
6. **Testing**
   - Write unit tests for components and services
   - Implement integration tests for workflows
   - Use proper mocking strategies

Red Flags:

- **Over-engineering** simple components
- **Not using** OnPush change detection
- **Mixing** presentation and business logic
- **Ignoring** performance optimizations
- **Skipping** proper error handling
- **Not implementing** proper testing

Conclusion

This comprehensive Angular Design Patterns Guide provides a systematic approach to building scalable, maintainable Angular applications. Remember:

- **Start with architecture** - Plan your module structure first
- **Use Smart/Dumb components** - Separate presentation from logic
- **Leverage reactive programming** - Use RxJS for data flow
- **Implement proper testing** - Write tests for all components and services
- **Optimize for performance** - Use OnPush strategy and virtual scrolling
- **Follow Angular best practices** - Use the framework's conventions

The biggest mistake I see developers make is not leveraging Angular's built-in patterns and trying to reinvent the wheel. Angular provides excellent patterns out of the box - use them!

Good luck with your Angular development!