Filtering:

Filtering involves displaying a subset of data that meets specific criteria. This is commonly used to narrow down large datasets based on user preferences. In Angular, you can implement filtering in various ways:

Built-in Pipes: Angular provides built-in pipes like *ngFor and *ngIf that can be used to filter and display data.

Custom Pipes: You can create your own custom filtering pipes that allow more complex filtering logic.

Filtering in Component: Implement filtering logic directly in the component using methods and properties.

Filtering with Reactive Forms: For more advanced scenarios, you can use Angular's reactive forms to build interactive filtering components

Here's a basic example of how you can implement filtering in Angular using built-in features:

Suppose you have a list of movies and you want to implement filtering by genre. Here's how you can do it:

1. Component:

Assuming you have a component where you fetch and display the list of movies:

```
import { Component } from '@angular/core';
import { MovieService } from './movie.service';

@Component({
  selector: 'app-movie-list',
  template: `
    <input type="text" [(ngModel)]="genreFilter" placeholder="Filter by Genre" />
    <ul>
      <li *ngFor="let movie of filteredMovies">{{ movie.title }} - {{ movie.genre }}</li>
    </ul>
  `,
})
export class MovieListComponent {
  movies = [];
  filteredMovies = [];
  genreFilter = '';

  constructor(private movieService: MovieService) {
    this.movies = this.movieService.getMovies();
    this.applyFilter();
  }

  applyFilter() {
    this.filteredMovies = this.movies.filter((movie) =>
      movie.genre.toLowerCase().includes(this.genreFilter.toLowerCase())
    );
  } }
```

2. Service :

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class MovieService {
  getMovies() {
    return [
      { title: 'Movie A', genre: 'Action' },
      { title: 'Movie B', genre: 'Comedy' },
      { title: 'Movie C', genre: 'Drama' },
      // ... more movie objects
    ];
  }
}
```

Pagination :

Pagination is the practice of breaking up a large dataset into smaller chunks (pages) to improve performance and user experience. Angular provides several techniques to implement pagination:

Custom Pagination Logic: You can implement your own pagination logic using arrays and indexes to show different chunks of data.

Pagination Libraries: There are third-party libraries like ngx-pagination that offer pre-built pagination components and logic.

Server-Side Pagination: For large datasets, consider implementing server-side pagination to load only the necessary data.

Here's how you can implement pagination using built-in features:

Suppose you have a list of movies and you want to implement pagination for displaying them.

1. Component :

Assuming you have a component where you fetch and display the list of movies:

```
import { Component } from '@angular/core';
import { MovieService } from './movie.service';

@Component({
  selector: 'app-movie-list',
  template: `
    <ul>
      <li *ngFor="let movie of displayedMovies">{{ movie.title }} - {{ movie.genre }}</li>
    </ul>
    <div>
      <button (click)="prevPage()">Previous</button>
      <span>{{ currentPage }}</span>
      <button (click)="nextPage()">Next</button>
    </div>
  `,
})
export class MovieListComponent {
  movies = [];
  displayedMovies = [];
  itemsPerPage = 5; // Number of items to display per page
  currentPage = 1;

  constructor(private movieService: MovieService) {
    this.movies = this.movieService.getMovies();
    this.updateDisplayedMovies();
  }

  updateDisplayedMovies() {
    const startIndex = (this.currentPage - 1) * this.itemsPerPage;
    const endIndex = startIndex + this.itemsPerPage;
    this.displayedMovies = this.movies.slice(startIndex, endIndex);
  }

  nextPage() {
    this.currentPage++;
    this.updateDisplayedMovies();
  }

  prevPage() {
    if (this.currentPage > 1) {
      this.currentPage--;
      this.updateDisplayedMovies();
    }
  }
}
```

2. Service :

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class MovieService {
  getMovies() {
    return [
      { title: 'Movie A', genre: 'Action' },
      { title: 'Movie B', genre: 'Comedy' },
      { title: 'Movie C', genre: 'Drama' },
      // ... more movie objects
    ];
  }
}

```

In this example, the MovieListComponent has a predefined number of items to display per page (itemsPerPage). The updateDisplayedMovies method calculates the range of movies to display based on the current page and itemsPerPage.

The pagination buttons (Next and Previous) allow users to navigate between pages. The nextPage and prevPage methods update the currentPage and call updateDisplayedMovies to refresh the displayed movies.

As with filtering, this is a basic example. For a more comprehensive pagination solution, you might want to consider integrating a pagination library, handling asynchronous data loading, and providing more user-friendly pagination controls.

State Management :

State management involves managing the application's data and its changes in a consistent and organized manner. Angular provides various options for managing application state:

Component State: Use component properties and services to manage state at a component level.

RxJS Observables: Use RxJS observables to manage and propagate state changes across components.

NgRx: NgRx is a popular state management library that implements Redux architecture in Angular applications. It provides actions, reducers, selectors, and effects to manage complex application states.

Akita: Akita is another state management library that simplifies state management by offering stores, entities, and querying capabilities.

When dealing with filtering, pagination, and state management in Angular, consider the complexity of your application and the specific requirements.

For simple cases, built-in Angular features might suffice, while for more complex applications, using third-party libraries or state management solutions like NgRx or Akita can provide better structure and maintainability.

Implementation :

One popular approach to state management in Angular is using the NgRx library, which is inspired by Redux.

Here's a simple example of implementing state management using NgRx for managing a list of movies:

- Install NgRx: Start by installing the NgRx packages:

  npm install @ngrx/store @ngrx/effects @ngrx/entity

1. Create Actions:

movie.actions.ts:

```
import { createAction, props } from '@ngrx/store';
import { Movie } from '../models/movie.model';

export const loadMovies = createAction('[Movie List] Load Movies');
export const moviesLoaded = createAction('[Movie List] Movies Loaded',
props<{ movies: Movie[] }>());

```

2. Create Reducer :

movie.reducer.ts:

```
import { createReducer, on } from '@ngrx/store';
import { loadMovies, moviesLoaded } from './movie.actions';
import { EntityState, EntityAdapter, createEntityAdapter } from '@ngrx/entity';
import { Movie } from '../models/movie.model';

export interface MovieState extends EntityState<Movie> {
  // Additional state properties can be added here
}

export const adapter: EntityAdapter<Movie> = createEntityAdapter<Movie>();

export const initialState: MovieState = adapter.getInitialState({
  // Additional initial state properties can be added here
});

export const movieReducer = createReducer(
  initialState,
  on(moviesLoaded, (state, { movies }) => adapter.setAll(movies, state)));
```

3. Create Effects :

movie.effects.ts:

```
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { EMPTY } from 'rxjs';
import { map, mergeMap, catchError } from 'rxjs/operators';
import { MovieService } from '../services/movie.service';
import * as MovieActions from './movie.actions';

@Injectable()
export class MovieEffects {
  loadMovies$ = createEffect(() =>
    this.actions$.pipe(
      ofType(MovieActions.loadMovies),
      mergeMap(() =>
        this.movieService.getMovies().pipe(
          map(movies => MovieActions.moviesLoaded({ movies })),
          catchError(() => EMPTY)
        )
      )
    )
  );

  constructor(private actions$: Actions, private movieService: MovieService) {}
}
```

4. Create Selectors:

movie.selectors.ts:

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { adapter, MovieState } from './movie.reducer';

export const selectMovieState = createFeatureSelector<MovieState>('movies');

export const {
  selectIds,
  selectEntities,
  selectAll,
  selectTotal,
} = adapter.getSelectors();

export const selectAllMovies = createSelector(selectMovieState, selectAll);
```

5. Module Configuration:

app.module.ts: In your module, configure NgRx Store, Effects, and provide the related services.

```
import { StoreModule } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';
import { movieReducer } from './state/movie.reducer';
import { MovieEffects } from './state/movie.effects';

@NgModule({
  imports: [
    StoreModule.forRoot({ movies: movieReducer }),
    EffectsModule.forRoot([MovieEffects]),
    // Other module imports
  ],
  // ...
})
export class AppModule {}
```

6. Service:

movie.service.ts:

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Movie } from '../models/movie.model';

@Injectable({
  providedIn: 'root',
})
export class MovieService {
  getMovies(): Observable<Movie[]> {
    // Replace with actual HTTP request or mock data
    return of([
      { id: 1, title: 'Movie A' },
      { id: 2, title: 'Movie B' },
      // ... more movies
    ]);
  }
}
```

Now, we have implemented a basic state management flow using NgRx in your Angular application