Okay, this is a very common and effective way to implement filtering in Angular Material tables, especially when dealing with backend-driven pagination.

We'll achieve this by:
1. **Toggling UI:** Using a boolean map to show/hide filter input fields.
2. **Backend Integration:** Sending filter parameters to your backend API.
3. **Pagination Reset:** Resetting the paginator to the first page when a filter changes.

Let's assume you have an Angular project set up with Angular Material installed.

---

1. `Product` Interface (models your data)

```typescript
// src/app/product.model.ts
export interface Product {
  id: number;
  productName: string;
  category: string;
  price: number;
  stockQuantity: number;
  // Add other properties as per your entity
}
```

2. `ProductService` (Simulates Backend API calls)

This service will simulate making API calls to your backend (which, in a real scenario, would be connected to your Hibernate/Java application).

```typescript
// src/app/product.service.ts
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Product } from './product.model';
import { delay, map } from 'rxjs/operators';

interface ProductFetchParams {
  pageNumber: number;
  pageSize: number;
  filters: { [key: string]: string }; // Map of column name to filter keyword
  sortBy?: string;
  sortDirection?: 'asc' | 'desc';
```

```
}

interface ProductFetchResult {
  content: Product[];
  totalElements: number;
}

// Dummy data (replace with actual backend calls)
const ALL_PRODUCTS: Product[] = [
  { id: 1, productName: 'Laptop', category: 'Electronics', price: 1200, stockQuantity: 50 },
  { id: 2, productName: 'Mouse', category: 'Electronics', price: 25, stockQuantity: 200 },
  { id: 3, productName: 'Keyboard', category: 'Electronics', price: 75, stockQuantity: 100 },
  { id: 4, productName: 'Monitor', category: 'Electronics', price: 300, stockQuantity: 75 },
  { id: 5, productName: 'Webcam', category: 'Accessories', price: 50, stockQuantity: 150 },
  { id: 6, productName: 'Speaker', category: 'Audio', price: 150, stockQuantity: 80 },
  { id: 7, productName: 'Headphones', category: 'Audio', price: 100, stockQuantity: 120 },
  { id: 8, productName: 'Microphone', category: 'Audio', price: 90, stockQuantity: 60 },
  { id: 9, productName: 'Router', category: 'Networking', price: 80, stockQuantity: 90 },
  { id: 10, productName: 'Printer', category: 'Office', price: 200, stockQuantity: 40 },
  { id: 11, productName: 'External SSD', category: 'Storage', price: 180, stockQuantity: 70 },
  { id: 12, productName: 'USB Hub', category: 'Accessories', price: 30, stockQuantity: 250 },
  { id: 13, productName: 'Gaming Mouse', category: 'Electronics', price: 55, stockQuantity: 120 },
  { id: 14, productName: 'Mechanical Keyboard', category: 'Electronics', price: 120,
stockQuantity: 80 },
  { id: 15, productName: 'Smartwatch', category: 'Wearables', price: 250, stockQuantity: 90 },
  { id: 16, productName: 'Tablet', category: 'Electronics', price: 400, stockQuantity: 70 },
  { id: 17, productName: 'Projector', category: 'Office', price: 500, stockQuantity: 30 },
  { id: 18, productName: 'Graphic Tablet', category: 'Creative', price: 150, stockQuantity: 45 },
];


@Injectable({
  providedIn: 'root'
})
export class ProductService {

  constructor() { }

  // Simulates an API call to your backend
  getProducts(params: ProductFetchParams): Observable<ProductFetchResult> {
    console.log('Fetching products with params:', params);

    let filteredProducts = ALL_PRODUCTS;
```

```
    // Apply filters from params (simulating backend filtering)
    if (params.filters) {
      for (const column in params.filters) {
        if (params.filters.hasOwnProperty(column) && params.filters[column]) {
          const filterKeyword = params.filters[column].toLowerCase();
          filteredProducts = filteredProducts.filter(product => {
            const columnValue = (product as any)[column]; // Access property dynamically
            return columnValue && String(columnValue).toLowerCase().includes(filterKeyword);
          });
        }
      }
    }

    const totalElements = filteredProducts.length;

    // Apply pagination (simulating backend pagination)
    const startIndex = params.pageNumber * params.pageSize;
    const endIndex = startIndex + params.pageSize;
    const paginatedProducts = filteredProducts.slice(startIndex, endIndex);

    return of({
      content: paginatedProducts,
      totalElements: totalElements
    }).pipe(delay(300)); // Simulate network latency
  }
}
```

3. Angular Component (`app.component.ts`)

```typescript
// src/app/app.component.ts
import { AfterViewInit, Component, OnInit, ViewChild } from '@angular/core';
import { MatTableDataSource } from '@angular/material/table';
import { MatPaginator, PageEvent } from '@angular/material/paginator';
import { MatSort } from '@angular/material/sort'; // Optional, but good for tables
import { Product } from './product.model';
import { ProductService } from './product.service';
import { Subject, merge } from 'rxjs';
import { debounceTime, distinctUntilChanged, tap } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, AfterViewInit {
  displayedColumns: string[] = ['id', 'productName', 'category', 'price', 'stockQuantity'];
  dataSource = new MatTableDataSource<Product>();
  totalElements = 0; // Total number of elements for paginator

  @ViewChild(MatPaginator) paginator!: MatPaginator;
  @ViewChild(MatSort) sort!: MatSort; // Optional

  // State for toggling filter inputs
  showFilter: { [key: string]: boolean } = {}; // e.g., { 'productName': true, 'category': false }

  // Store filter values for each column
  filterValues: { [key: string]: string } = {};

  // A Subject to emit filter changes and debounce them
  private filterChange = new Subject<void>();

  constructor(private productService: ProductService) {}

  ngOnInit(): void {
    // Initialize filter toggles to false for all columns
    this.displayedColumns.forEach(col => {
      this.showFilter[col] = false;
      this.filterValues[col] = ''; // Initialize filter value
    });

    // Subscribe to filter changes with a debounce time
    // This prevents hitting the backend on every keystroke
    this.filterChange.pipe(
      debounceTime(300), // Wait for 300ms of no new input
      distinctUntilChanged(), // Only emit if filter values actually changed
      tap(() => {
        this.paginator.pageIndex = 0; // Reset to first page on filter change
        this.loadProducts();
      })
    ).subscribe();
  }

  ngAfterViewInit(): void {
    // Combine paginator and sort events to trigger data load
    // If you don't use sorting, you can remove the sort part.
    merge(this.paginator.page, this.sort.sortChange)
```

```
    .pipe(
      tap(() => this.loadProducts())
    )
    .subscribe();

  this.loadProducts(); // Initial data load
}

// Toggles the visibility of the filter input for a given column
toggleFilter(column: string): void {
  this.showFilter[column] = !this.showFilter[column];
  if (!this.showFilter[column]) {
    // If filter is hidden, clear its value and trigger reload
    if (this.filterValues[column]) {
      this.filterValues[column] = '';
      this.filterChange.next(); // Trigger reload
    }
  }
}

// Called when input in a filter field changes
onFilterChange(): void {
  this.filterChange.next(); // Emit event to subject
}

private loadProducts(): void {
  const params = {
    pageNumber: this.paginator.pageIndex,
    pageSize: this.paginator.pageSize,
    filters: this.filterValues,
    sortBy: this.sort?.active, // Optional: send sort info to backend
    sortDirection: this.sort?.direction // Optional: send sort info to backend
  };

  this.productService.getProducts(params).subscribe(data => {
    this.dataSource.data = data.content;
    this.totalElements = data.totalElements;
  });
}

// Handle page changes from MatPaginator (already handled by merge above)
onPageChange(event: PageEvent): void {
  // This method is called, but the logic is primarily in the merge observable in ngAfterViewInit
```

```
    // You could put specific logic here if needed, but for triggering loadProducts, merge is
cleaner.
  }
}
```

4. Angular HTML Template (`app.component.html`)

```html
<div class="table-container">
  <h1>Product List</h1>

  <mat-card>
    <mat-table [dataSource]="dataSource" matSort>
      <ng-container *ngFor="let col of displayedColumns" [matColumnDef]="col">
        <mat-header-cell *matHeaderCellDef mat-sort-header [ngClass]="{'filter-active':
showFilter[col]}">
          <div class="header-content">
            <span class="column-name">{{ col | titlecase }}</span>
            <button mat-icon-button (click)="toggleFilter(col); $event.stopPropagation()">
              <mat-icon>filter_list</mat-icon>
            </button>
          </div>
          <mat-form-field *ngIf="showFilter[col]" class="filter-input-field" appearance="fill">
            <input matInput
                   placeholder="Filter {{ col }}"
                   [(ngModel)]="filterValues[col]"
                   (input)="onFilterChange()"
                   (click)="$event.stopPropagation()"> </mat-form-field>
        </mat-header-cell>
        <mat-cell *matCellDef="let element"> {{ element[col] }} </mat-cell>
      </ng-container>

      <mat-header-row *matHeaderRowDef="displayedColumns"></mat-header-row>
      <mat-row *matRowDef="let row; columns: displayedColumns;"></mat-row>

      <tr class="mat-row" *matNoDataRow>
        <td class="mat-cell" colspan="9999">No data matching the filter "{{ filterValues | json
}}"</td>
      </tr>
    </mat-table>

    <mat-paginator [length]="totalElements"
                   [pageSize]="5"
```

```
                [pageSizeOptions]="[5, 10, 25, 100]"
                aria-label="Select page of products">
    </mat-paginator>
  </mat-card>
</div>
```

5. Angular CSS (`app.component.css`)

```css
/* src/app/app.component.css */
.table-container {
  padding: 20px;
  max-width: 90%;
  margin: 20px auto;
}

mat-card {
  box-shadow: 0 4px 8px rgba(0,0,0,0.1);
}

mat-table {
  width: 100%;
}

mat-header-cell {
  position: relative;
  display: flex; /* Use flexbox for header content alignment */
  flex-direction: column; /* Stack column name and filter input */
  align-items: flex-start; /* Align content to the start */
  padding-bottom: 5px; /* Adjust padding if needed */
  padding-top: 10px; /* Adjust padding if needed */
  vertical-align: top; /* Align header cell content to the top */
}

.header-content {
  display: flex;
  justify-content: space-between;
  align-items: center;
  width: 100%; /* Take full width of the header cell */
  min-height: 24px; /* Ensure space for icon/text */
}

.column-name {
```

```
  flex-grow: 1; /* Allow column name to take available space */
  font-weight: bold;
}

mat-icon-button {
  min-width: 24px; /* Adjust as needed for icon size */
  width: 24px;
  height: 24px;
  line-height: 24px;
  padding: 0;
  margin-left: 5px;
}

.filter-input-field {
  width: 100%; /* Make filter input take full width */
  margin-top: 5px;
  font-size: 14px; /* Adjust font size if necessary */
}

/* Hide the Material Design underline and focus line for a cleaner look */
.filter-input-field .mat-form-field-wrapper {
  padding-bottom: 0 !important;
}
.filter-input-field .mat-form-field-underline {
  display: none !important;
}
.filter-input-field .mat-form-field-flex {
  padding: 0.2em 0.5em 0 0.5em !important;
  background-color: rgba(0, 0, 0, 0.04); /* Light background for the input */
  border-radius: 4px 4px 0 0;
}
.filter-input-field .mat-form-field-appearance-fill .mat-form-field-flex {
    border-radius: 4px 4px 0 0;
}

/* Optional: Highlight header when filter is active */
.mat-header-cell.filter-active {
  background-color: rgba(0, 0, 0, 0.03); /* Slightly grey background */
}
```

Explanation:

1. **`ProductService` (Backend Simulation):**

    * The `getProducts` method takes `pageNumber`, `pageSize`, and a `filters` object.
    * It simulates applying filters and pagination to `ALL_PRODUCTS` (your dummy data).
    * **Crucially, in a real application, you would replace the dummy data and filtering logic with an actual `HttpClient` call to your Java backend, passing these `params` as query parameters.**

2. **`app.component.ts`:**
    * **`showFilter: { [key: string]: boolean }`**: An object that stores a boolean for each column, indicating whether its filter input should be shown.
    * **`filterValues: { [key: string]: string }`**: An object to store the current filter keyword for each column.
    * **`toggleFilter(column: string)`**:
        * Toggles `showFilter[column]`.
        * If the filter input is *hidden* (i.e., `showFilter[column]` becomes `false`), it clears the `filterValues[column]` and immediately triggers `onFilterChange()` to apply the cleared filter.
    * **`filterChange: Subject<void>`**: This is a powerful RxJS pattern.
        * When `onFilterChange()` is called (on every input keystroke), it simply emits a value to this `Subject`.
        * The `filterChange.pipe(debounceTime(300), distinctUntilChanged(), tap(...))` handles this:
            * `debounceTime(300)`: Waits for 300 milliseconds of inactivity before processing the event. This prevents excessive API calls as the user types.
            * `distinctUntilChanged()`: Ensures the filter only re-applies if the *actual filter values* (after debounce) have changed.
            * `tap(() => { this.paginator.pageIndex = 0; this.loadProducts(); })`: When the debounced filter changes, it resets the paginator to the first page (`pageIndex = 0`) and then calls `loadProducts()`.
    * **`loadProducts()`**: This method constructs the `params` object containing current pagination state (`pageIndex`, `pageSize`), current `filterValues`, and optional sorting info. It then calls `productService.getProducts()` to fetch data.
    * **`ngAfterViewInit()`**: Combines `paginator.page` events (when page changes) and `sort.sortChange` events (if you add `matSort`) to trigger `loadProducts()`. This ensures that any change in pagination or sorting parameters reloads the data from the backend.

3. **`app.component.html`:**
    * **`mat-header-cell`:**
        * The column name (`{{ col | titlecase }}`) and the filter icon button (`<button mat-icon-button>`) are placed together.
        * `(click)="toggleFilter(col); $event.stopPropagation()"`: The icon button toggles the filter. `$event.stopPropagation()` is crucial to prevent the `mat-sort-header` from also triggering a sort when you click the filter icon or input.
        * **`*ngIf="showFilter[col]"`**: Conditionally renders the `mat-form-field` and `input` for the filter.
        * **`[(ngModel)]="filterValues[col]"`**: Binds the input field to the corresponding entry in your `filterValues` object.

* **`(input)="onFilterChange()"`**: Triggers the `onFilterChange()` method in the component whenever the input value changes.
* **`mat-paginator`**: Set `[length]="totalElements"` to bind it to the total number of records received from your backend.

This setup provides a highly interactive and efficient way to filter your Mat Table data with backend pagination, offering a great user experience. Remember to replace the `ProductService` dummy data and logic with your actual API calls.
Citations:
[[1]](https://git.swl.informatik.uni-oldenburg.de/projects/API_APPS/repos/webstudio/browse/frontend/src/app/+queries/dashboard/query-dashboard.component.html?at=08e287fc7abdcad08727505227c021b3b941aa32), [[2]](https://github.com/Ravitej-ui/safetyWorks), [[3]](https://github.com/LaurianeDP/TourOfHeroes-Front)