Okay, you have the data in your Java service layer (e.g., `List<Product>`). Now you want to convert this into an Excel file (specifically, a byte array or `Blob` for the frontend) and then facilitate its download in Angular.

This involves two main parts:

1. **Backend (Java/Spring Boot):** Generate the Excel file as a byte array and send it in an HTTP response.
2. **Frontend (Angular):** Make an HTTP request to the backend, receive the byte array as a `Blob`, and trigger a file download in the browser.

Part 1: Backend (Java/Spring Boot) Implementation

We'll use Apache POI, a popular Java API for Microsoft document formats.

1. Add Apache POI Dependencies

In your `pom.xml`, add the following dependencies:

```xml
<dependencies>
   <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi</artifactId>
      <version>5.2.5</version> </dependency>
   <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi-ooxml</artifactId>
      <version>5.2.5</version> </dependency>
</dependencies>
```

2. Create an Excel Export Service

This service will handle the logic of taking your `List<Product>` and generating an Excel `Workbook`.

```java
// src/main/java/com/example/service/ProductExcelService.java
package com.example.service;

import com.example.model.Product;
```

```
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import org.springframework.stereotype.Service;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.List;

@Service
public class ProductExcelService {

    public ByteArrayOutputStream exportProductsToExcel(List<Product> products) throws
IOException {
        // Create a new workbook (XSSFWorkbook for .xlsx format)
        Workbook workbook = new XSSFWorkbook();
        Sheet sheet = workbook.createSheet("Products Data"); // Name your sheet

        // Create Header Row
        Row headerRow = sheet.createRow(0);
        String[] headers = {"ID", "Product Name", "Category", "Price", "Stock Quantity"}; // Adjust
as per your Product fields

        // Styling for header cells (optional)
        CellStyle headerCellStyle = workbook.createCellStyle();
        Font headerFont = workbook.createFont();
        headerFont.setBold(true);
        headerCellStyle.setFont(headerFont);
        headerCellStyle.setAlignment(HorizontalAlignment.CENTER);

        for (int i = 0; i < headers.length; i++) {
            Cell cell = headerRow.createCell(i);
            cell.setCellValue(headers[i]);
            cell.setCellStyle(headerCellStyle);
        }

        // Populate Data Rows
        int rowNum = 1;
        for (Product product : products) {
            Row row = sheet.createRow(rowNum++);

            // Adjust cell creation and value setting based on your Product model fields
            row.createCell(0).setCellValue(product.getId());
            row.createCell(1).setCellValue(product.getProductName());
            row.createCell(2).setCellValue(product.getCategory());
```

```
            // Handle BigDecimal price - convert to double for Excel cell
            if (product.getPrice() != null) {
                row.createCell(3).setCellValue(product.getPrice().doubleValue());
            } else {
                row.createCell(3).setCellValue(""); // Or 0.0 if you prefer
            }

            row.createCell(4).setCellValue(product.getStockQuantity());
        }

        // Auto-size columns for better readability (optional)
        for (int i = 0; i < headers.length; i++) {
            sheet.autoSizeColumn(i);
        }

        // Write the workbook to a ByteArrayOutputStream
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        workbook.write(outputStream);
        workbook.close(); // Close the workbook to free resources
        return outputStream;
    }
}
```

3. Add a Controller Endpoint to Trigger Download

This endpoint will receive the request from the frontend, fetch the data (potentially with the same filters applied for display), call the `ProductExcelService`, and return the Excel file as a byte array.

```java
// src/main/java/com/example/controller/ProductController.java
package com.example.controller;

import com.example.dao.ProductDAO;
import com.example.dto.PaginatedResponse;
import com.example.model.Product;
import com.example.service.ProductExcelService; // Import the new service
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductDAO productDAO; // Assuming ProductDAO is already wired
    private final ProductExcelService productExcelService; // Inject the new service

    public ProductController(ProductDAO productDAO, ProductExcelService
productExcelService) {
        this.productDAO = productDAO;
        this.productExcelService = productExcelService;
    }

    // Existing endpoint for fetching paginated and filtered data (keep this)
    @GetMapping
    public ResponseEntity<PaginatedResponse<Product>> getProducts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(required = false) String globalSearchKeyword, // Optional global
search
            @RequestParam Map<String, String> allRequestParams, // Capture all request
parameters
            @RequestParam(required = false) String sortBy,
            @RequestParam(required = false) String sortDirection
    ) {
        Map<String, String> columnFilters = new HashMap<>();
        for (Map.Entry<String, String> entry : allRequestParams.entrySet()) {
            String paramName = entry.getKey();
            if (!"page".equals(paramName) && !"size".equals(paramName) &&
                !"globalSearchKeyword".equals(paramName) && !"sortBy".equals(paramName) &&
!"sortDirection".equals(paramName)) {
                columnFilters.put(paramName, entry.getValue());
            }
```

```
        }

        PaginatedResponse<Product> result = productDAO.getProductsPaginatedAndFiltered(
            page, size, globalSearchKeyword, columnFilters, sortBy, sortDirection
        );
        return ResponseEntity.ok(result);
    }

    // NEW ENDPOINT FOR EXCEL DOWNLOAD
    @GetMapping("/download/excel")
    public ResponseEntity<byte[]> downloadProductsExcel(
            // Use the same parameters as your getProducts endpoint to allow filtering the export
            @RequestParam(required = false) String globalSearchKeyword,
            @RequestParam Map<String, String> allRequestParams,
            @RequestParam(required = false) String sortBy,
            @RequestParam(required = false) String sortDirection
    ) {
        Map<String, String> columnFilters = new HashMap<>();
        for (Map.Entry<String, String> entry : allRequestParams.entrySet()) {
            String paramName = entry.getKey();
            // Exclude parameters that are not filters for the DAO call, if any
            if (!"globalSearchKeyword".equals(paramName) && !"sortBy".equals(paramName) &&
!"sortDirection".equals(paramName)) {
                columnFilters.put(paramName, entry.getValue());
            }
        }

        // Fetch ALL products that match the current filters (not paginated for export)
        // You might want to adjust the DAO call to fetch all records matching filters,
        // rather than using page 0 and a very large size.
        // For simplicity, I'll use a very large page size to get all filtered results.
        // A better approach would be to add a ProductDAO method like
'getAllProductsFiltered(keyword, filters, sortBy, sortDirection)'
        PaginatedResponse<Product> filteredData =
productDAO.getProductsPaginatedAndFiltered(
            0, Integer.MAX_VALUE, globalSearchKeyword, columnFilters, sortBy, sortDirection
        );
        List<Product> productsToExport = filteredData.getContent();

        try {
            ByteArrayOutputStream excelStream =
productExcelService.exportProductsToExcel(productsToExport);
            byte[] excelBytes = excelStream.toByteArray();
```

```
            // Prepare headers for the response
            HttpHeaders headers = new HttpHeaders();
            // Set Content-Disposition to "attachment" to prompt download
            String fileName = "products_" +
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyyMMdd_HHmmss")) + ".xlsx";
            headers.add(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
fileName + "\"");
            // Set Content-Type to
application/vnd.openxmlformats-officedocument.spreadsheetml.sheet for .xlsx

headers.setContentType(MediaType.parseMediaType("application/vnd.openxmlformats-officedo
cument.spreadsheetml.sheet"));
            headers.setContentLength(excelBytes.length); // Set content length

            return ResponseEntity.ok()
                    .headers(headers)
                    .body(excelBytes);

        } catch (IOException e) {
            e.printStackTrace();
            return ResponseEntity.status(500).build(); // Internal Server Error
        }
    }
}
```

Important Note for `ProductDAO`:
In the `downloadProductsExcel` endpoint, I used `productDAO.getProductsPaginatedAndFiltered(0, Integer.MAX_VALUE, ...)` to fetch all filtered products. This is a quick workaround. For a robust solution, consider adding a dedicated method to your `ProductDAO` like `List<Product> getAllProductsFiltered(String globalSearchKeyword, Map<String, String> columnFilters, String sortBy, String sortDirection)` that performs the filtering and sorting but *without* pagination limits.

Part 2: Frontend (Angular) Implementation

1. Update `ProductService` (Angular)

Add a new method to handle the Excel download. Crucially, set `responseType: 'blob'`.

```typescript
// src/app/product.service.ts
```

```
// ... (Keep existing imports and interfaces) ...
import { HttpClient, HttpParams } from '@angular/common/http';

// ... (Rest of your ProductService class) ...

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

  private apiUrl = 'http://localhost:8080/api/products'; // Adjust if your port/path is different

  constructor(private http: HttpClient) { }

  // Existing getProducts method for paginated/filtered data
  getProducts(params: ProductFetchParams): Observable<ProductFetchResult> {
    let httpParams = new HttpParams();
    httpParams = httpParams.set('page', params.pageNumber.toString());
    httpParams = httpParams.set('size', params.pageSize.toString());

    if (params.globalSearchKeyword) {
      httpParams = httpParams.set('globalSearchKeyword', params.globalSearchKeyword);
    }
    if (params.columnFilters) {
      for (const column in params.columnFilters) {
        if (params.columnFilters.hasOwnProperty(column) && params.columnFilters[column]) {
          httpParams = httpParams.set(column, params.columnFilters[column]);
        }
      }
    }
    if (params.sortBy) {
      httpParams = httpParams.set('sortBy', params.sortBy);
      if (params.sortDirection) {
        httpParams = httpParams.set('sortDirection', params.sortDirection);
      }
    }

    return this.http.get<ProductFetchResult>(this.apiUrl, { params: httpParams });
  }

  // NEW: Method to download Excel file
  downloadProductsExcel(params: Omit<ProductFetchParams, 'pageNumber' | 'pageSize'>):
Observable<Blob> {
    let httpParams = new HttpParams();
```

```
    // Add global search keyword if provided
    if (params.globalSearchKeyword) {
      httpParams = httpParams.set('globalSearchKeyword', params.globalSearchKeyword);
    }

    // Add column-specific filters if provided
    if (params.columnFilters) {
      for (const column in params.columnFilters) {
        if (params.columnFilters.hasOwnProperty(column) && params.columnFilters[column]) {
          httpParams = httpParams.set(column, params.columnFilters[column]);
        }
      }
    }

    // Add sorting parameters if provided (important if you want sorted export)
    if (params.sortBy) {
      httpParams = httpParams.set('sortBy', params.sortBy);
      if (params.sortDirection) {
        httpParams = httpParams.set('sortDirection', params.sortDirection);
      }
    }

    console.log('Requesting Excel download with params:', httpParams.toString());

    // Crucially set responseType to 'blob'
    return this.http.get(`${this.apiUrl}/download/excel`, {
      params: httpParams,
      responseType: 'blob' // <-- Tell Angular to expect a binary blob response
    });
  }
}
```

2. Update `AppComponent` (Angular)

Add a button and a method to trigger the download.

```typescript
// src/app/app.component.ts
// ... (Keep existing imports) ...

@Component({
  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, AfterViewInit {
  // ... (Keep existing properties like displayedColumns, dataSource, etc.) ...

  constructor(private productService: ProductService) {} // Ensure HttpClientModule is imported
in app.module.ts

  // ... (Keep existing ngOnInit, ngAfterViewInit, toggleFilter, onFilterChange, loadProducts) ...

  // NEW: Method to handle Excel download
  downloadExcel(): void {
    // Collect the current search and filter parameters
    const params = {
      globalSearchKeyword: this.globalSearchKeyword.trim() || undefined,
      columnFilters: this.filterValues,
      sortBy: this.sort?.active,
      sortDirection: this.sort?.direction
    };

    this.productService.downloadProductsExcel(params).subscribe({
      next: (responseBlob: Blob) => {
        // Create a URL for the blob
        const url = window.URL.createObjectURL(responseBlob);

        // Create a temporary anchor tag to trigger the download
        const a = document.createElement('a');
        a.href = url;

        // Extract filename from Content-Disposition header if available
        // Note: You might need to expose Content-Disposition header on your backend
        // For simplicity, we'll use a hardcoded name or generate one.
        // If your backend correctly sets Content-Disposition, you can get it:
        // const contentDisposition = response.headers.get('Content-Disposition');
        // const filenameMatch = contentDisposition &&
contentDisposition.match(/filename="([^"]+)"/);
        // const filename = filenameMatch ? filenameMatch[1] : 'products.xlsx';

        const filename = `products_export_${new Date().toISOString().slice(0, 10)}.xlsx`;
        a.download = filename; // Set the download filename

        document.body.appendChild(a); // Append to body (required for Firefox)
        a.click(); // Programmatically click the link to trigger download
```

```
        document.body.removeChild(a); // Clean up the temporary link
        window.URL.revokeObjectURL(url); // Free up the object URL
        console.log('Excel file downloaded successfully!');
      },
      error: (error) => {
        console.error('Error downloading Excel file:', error);
        // Handle error (e.g., show a snackbar message to the user)
      }
    });
  }
}
```

3. Update `app.component.html` (Angular)

Add a button to trigger the Excel download.

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

  <mat-card>
    <mat-form-field appearance="outline" class="global-search-field">
      <mat-label>Global Search</mat-label>
      <input matInput
             placeholder="Search all columns..."
             [(ngModel)]="globalSearchKeyword"
             (input)="onFilterChange()">
      <mat-icon matSuffix>search</mat-icon>
    </mat-form-field>

    <button mat-raised-button color="primary" (click)="downloadExcel()"
class="download-button">
      <mat-icon>cloud_download</mat-icon> Download Excel
    </button>

    </mat-card>
</div>
```

Important Considerations:

1. **Backend `ProductDAO` for Export:**

    * The current `downloadProductsExcel` endpoint uses `productDAO.getProductsPaginatedAndFiltered(0, Integer.MAX_VALUE, ...)` to fetch all data. While functional, it's generally better to have a dedicated DAO method like `List<Product> getAllProductsFiltered(String globalSearchKeyword, Map<String, String> columnFilters, String sortBy, String sortDirection)` that *only* applies filters/sorting and *doesn't* deal with pagination limits. This makes it clearer that you want *all* matching data for export.

2. **`Content-Disposition` Header (CORS):**
    * For the frontend to correctly get the filename from the `Content-Disposition` header, your backend might need to explicitly expose this header in its CORS configuration (if your frontend and backend are on different domains/ports).
    * In Spring Boot, you might add something like `.exposedHeaders("Content-Disposition")` to your CORS configuration.

3. **Error Handling:**
    * Both backend and frontend code include basic error handling. You should expand this to provide meaningful feedback to the user (e.g., "Export failed, please try again").

With these steps, you'll have a robust system for downloading Excel files from your Angular frontend, generated from data fetched and processed by your Hibernate backend.
Citations: [[1]](https://www.tutorialspoint.com/selenium/selenium_hybrid_driven_framework.htm), [[2]](https://github.com/gokulzt1005/IMS), [[3]](https://github.com/helmiReg/angularBoraq)