Report Module - Complete Technical Documentation

This document explains **every class, method, and line** in the HR report generation module.

Table of Contents

---

1. Architecture Overview

```
flowchart TD
    subgraph Controller Layer
        HC[HrController]
    end

    subgraph Service Layer
        RS[ReportService]
        HRS[HrReportService]
    end

    subgraph Exporter Layer
        AE[AbstractExcelExporter]
        AP[AbstractPdfExporter]
        CE[ClaimReportExcelExporter]
        CP[ClaimReportPdfExporter]
        PE[PremiumReportExcelExporter]
        PP[PremiumReportPdfExporter]
        EE[EmployeeReportExcelExporter]
        EP[EmployeeReportPdfExporter]
    end

    subgraph Utility Layer
        RF[ReportFormatters]
    end

    HC --> RS
    HC --> HRS
    HC --> CE
    HC --> CP
    CE --> AE
    CP --> AP
    PE --> AE
    PP --> AP
    EE --> AE
    EP --> AP
    CE --> RF
    CP --> RF
    PE --> RF
    PP --> RF
```

Why this architecture? - **Separation of Concerns**: Each layer has a single responsibility - **DRY Principle**: Common code is in base classes and utilities - **Testability**: Each component can be tested independently - **Extensibility**: Easy to add new report types

---

2. MIME Types Explained

When the browser downloads a file, it needs to know what type of file it is. This is done via **MIME types** (Multipurpose Internet Mail Extensions).

Excel MIME Type

```
response.setContentType("application/vnd.openxmlformats-officedocument.spreadsheetml.sheet")
```

Part	Meaning
`application`	This is an application file (not text, image, etc.)
`vnd`	"Vendor" - this is a vendor-specific format
`openxmlformats`	Uses Microsoft's Open XML format
`officedocument`	It's an Office document
`spreadsheetml`	It's a spreadsheet (Excel)

Part	Meaning
`sheet`	Specifically a worksheet file (.xlsx)

Why this specific MIME?
For `.xlsx` files (Excel 2007+), this is the official registered MIME type. Using the wrong MIME type would cause the browser to not recognize it as Excel.

PDF MIME Type

```
response.setContentType("application/pdf");
```

Part	Meaning
`application`	This is an application file
`pdf`	Portable Document Format

Why this is simpler?
PDF is a standardized format by Adobe, so it has a simple registered MIME type.

Content-Disposition Header

```
response.setHeader("Content-Disposition", "attachment; filename=claims-report.xlsx");
```

Part	Meaning
`Content-Disposition`	HTTP header that tells browser how to handle the response
`attachment`	Download the file instead of displaying it inline
`filename=...`	Suggested filename for the download

Why `attachment`?
Forces the browser to download the file. Without it, PDFs might open in the browser tab instead.

---

3. ReportService Interface

ReportService.java

```
package com.employeeinsurancemanagement.service;

import com.employeeinsurancemanagement.dto.ClaimReportDto;
import com.employeeinsurancemanagement.dto.EmployeeReportDto;
import com.employeeinsurancemanagement.dto.PremiumReportDto;

import java.util.List;
```

Line-by-line: - `package`: Declares this file belongs to the `service` package - `import`: Imports the DTO classes that will be returned by methods - DTOs (Data Transfer Objects) are simple data containers used to transfer data between layers

```
/**
 * Service interface for report data retrieval.
 * Note: All data queries are implemented in ReportServiceImpl using
 * EntityManager.
 */
public interface ReportService {
```

Why an interface? - **Abstraction**: Hides implementation details from callers - **Dependency Injection**: Spring can inject any implementation - **Testability**: Easy to create mock implementations for testing - **Multiple Implementations**: Could have different implementations (e.g., cached version)

```
  List<EmployeeReportDto> getEmployeeCountByOrganization(Long organizationId);
```

Why `Long organizationId` parameter? - Allows filtering by organization - `null` means "all organizations" - HR users only see their own organization's data

```
  List<ClaimReportDto> getClaimSummaryByEnrollment(String status);
```

Why `String status` instead of `ClaimStatus` enum? - Comes from URL parameter (always a String) - Conversion to enum happens in the implementation - Allows passing "ALL" as a special value

```
  List<PremiumReportDto> getPremiumCollectedByOrganization(Long organizationId);
}
```

Why return `List<T>` instead of raw data? - DTOs are designed for the specific view needs - Decouples the view from the database entities - Can include computed fields not in the database

---

4. ReportServiceImpl

ReportServiceImpl.java

```
@Service
public class ReportServiceImpl implements ReportService {
```

`@Service`: Spring annotation that: 1. Marks this class as a service layer component 2. Tells Spring to create a singleton instance 3. Enables component scanning to find it

```
@PersistenceContext
private EntityManager em;
```

`@PersistenceContext`: JPA annotation that: 1. Injects the EntityManager (database session manager) 2. Lifecycle managed by Spring (thread-safe) 3. Different from **`@Autowired`** - specifically for JPA

Why EntityManager instead of Repository? - Complex aggregate queries with GROUP BY, COUNT, SUM - Direct DTO construction in JPQL - More control over query optimization

getEmployeeCountByOrganization Method

```
@Override
public List<EmployeeReportDto> getEmployeeCountByOrganization(Long organizationId) {
    String query = """
                SELECT new EmployeeReportDto(
                    o.organizationId,
                    o.organizationName,
                    COUNT(e)
                )
                FROM Organization o
                LEFT JOIN o.employee e
                WHERE (:orgId IS NULL OR o.organizationId = :orgId)
                GROUP BY o.organizationId, o.organizationName
            """;
```

Triple quotes `"""`: Java 15+ text blocks for multi-line strings.

`SELECT new EmployeeReportDto(...)`: - JPQL constructor expression - Creates DTOs directly in the query (more efficient than mapping) - Requires DTO to have matching constructor

`LEFT JOIN o.employee e`: - LEFT JOIN includes organizations with zero employees - INNER JOIN would exclude them

`WHERE (:orgId IS NULL OR o.organizationId = :orgId)`: - If `orgId` is null → no filter (all orgs) - If `orgId` is set → filter to that org - Single query handles both cases

`GROUP BY`: - Required because we use `COUNT()` - Groups employees by their organization

```
    return em.createQuery(query, EmployeeReportDto.class)
            .setParameter("orgId", organizationId)
```

```
                .getResultList();
    }
```

`createQuery(query, Class)`: - Parses JPQL string - Second parameter is the result type (for type safety)

`setParameter("orgId", ...)`: - Named parameter binding (prevents SQL injection) - The `:orgId` in the query is replaced with this value

`getResultList()`: - Executes query and returns all results - Returns empty list if no results (never null)

getClaimSummaryByEnrollment Method

```
    @Override
    public List<ClaimReportDto> getClaimSummaryByEnrollment(String status) {
        String query = """
                    SELECT new com.employeeinsurancemanagement.dto.ClaimReportDto(
                        e.enrollmentId,
                        c.claimId,
                        COALESCE(c.approvedAmount, 0.0),
                        c.claimDate,
                        c.claimStatus
                    )
                    FROM Enrollment e
                    LEFT JOIN e.claims c
                    WHERE (:status IS NULL OR c.claimStatus = :status)
                    ORDER BY c.claimDate DESC
                """;
```

Fully qualified class name: `com.employeeinsurancemanagement.dto.ClaimReportDto` - Sometimes needed when JPQL can't resolve short names - Explicit reference avoids ambiguity

`COALESCE(c.approvedAmount, 0.0)`: - SQL function that returns first non-null value - If `approvedAmount` is null → returns 0.0 - Prevents null values in the DTO

`ORDER BY c.claimDate DESC`: - Default sorting by most recent claims first - Can be overridden in controller

```
        var q = em.createQuery(query, ClaimReportDto.class);
        if (status != null && !status.equals("ALL")) {
            try {
                com.employeeinsurancemanagement.model.ClaimStatus statusEnum =
                    com.employeeinsurancemanagement.model.ClaimStatus.valueOf(status.toUpper
                q.setParameter("status", statusEnum);
            } catch (IllegalArgumentException e) {
                q.setParameter("status", null);
            }
        } else {
            q.setParameter("status", null);
        }
```

`var`: Java 10+ local type inference (compiler determines the type).

Why this complex logic? 1. URL param is always a String 2. Need to convert to `ClaimStatus` enum 3. “ALL” treated as no filter (null) 4. Invalid status values gracefully ignored

`valueOf(status.toUpperCase())`: - Converts String “APPROVED” → ClaimStatus.APPROVED - `toUpperCase()` allows case-insensitive matching

`try-catch`: - `valueOf` throws `IllegalArgumentException` if invalid - Catch block treats invalid status as “no filter”

---

5. ReportFormatters Utility

ReportFormatters.java

```java
package com.employeeinsurancemanagement.util;

import java.text.DecimalFormat;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
```

Why these imports? - `DecimalFormat`: For currency formatting (1,234.56) - `LocalDate`: Java 8+ date type (no time component) - `DateTimeFormatter`: Thread-safe date formatter

```java
/**
 * Utility class for consistent report formatting.
 * All report exporters should use these methods.
 */
public final class ReportFormatters {
```

`final class`: - Cannot be subclassed - Common pattern for utility classes - Signals “use the static methods, don’t extend”

```java
    private static final DateTimeFormatter DATE_FORMAT =
        DateTimeFormatter.ofPattern("dd-MM-yyyy");
```

`private static final`: - `private`: Not accessible from outside - `static`: One instance shared across all usages - `final`: Cannot be reassigned

`DateTimeFormatter.ofPattern("dd-MM-yyyy")`: - Pattern: day-month-year (Indian format) - Thread-safe (unlike SimpleDateFormat)

```
    private static final DecimalFormat CURRENCY_FORMAT =
        new DecimalFormat(" #,##0.00");
```

Pattern breakdown: - : Literal rupee symbol - `#,##0`: Grouping with commas (1,234) - `.00`: Always 2 decimal places

```
    private ReportFormatters() {
        // Private constructor prevents instantiation
    }
```

Private constructor: - Utility classes shouldn't be instantiated - All methods are static - Best practice for utility classes

```
    public static String formatDate(LocalDate date) {
        if (date == null) {
            return "-";
        }
        return date.format(DATE_FORMAT);
    }
```

Null handling: Returns "-" for null dates (graceful degradation).

`date.format(formatter)`: Converts LocalDate to String.

```
    public static String formatCurrency(Double amount) {
        if (amount == null) {
            return " 0.00";
        }
        return CURRENCY_FORMAT.format(amount);
    }
```

Why return " 0.00" for null? - Consistent display in reports - Avoids "null" appearing in exports

```
    public static double formatCurrencyRaw(Double amount) {
        return amount != null ? amount : 0.0;
    }
}
```

Why a "raw" version? - Excel needs numeric values for calculations - String " 1,234.56" can't be summed in Excel - Returns number, Excel applies cell formatting

---

6. AbstractExcelExporter Base Class

AbstractExcelExporter.java

```
package com.employeeinsurancemanagement.service.exporter;

import org.apache.poi.ss.usermodel.*;
import java.io.ByteArrayOutputStream;
```

Apache POI: The library for creating Excel files in Java.

`org.apache.poi.ss.usermodel.*`: - ss = SpreadSheet - Contains interfaces: Workbook, Sheet, Row, Cell, CellStyle

```
public abstract class AbstractExcelExporter {
```

`abstract class`: - Cannot be instantiated directly - Meant to be extended - Can contain both abstract and concrete methods

createHeaderStyle Method

```
    protected CellStyle createHeaderStyle(Workbook workbook) {
        CellStyle style = workbook.createCellStyle();
        Font font = workbook.createFont();
        font.setBold(true);
        style.setFont(font);
        style.setFillForegroundColor(IndexedColors.GREY_25_PERCENT.getIndex());
        style.setFillPattern(FillPatternType.SOLID_FOREGROUND);
        style.setBorderBottom(BorderStyle.THIN);
        style.setBorderTop(BorderStyle.THIN);
        style.setBorderLeft(BorderStyle.THIN);
        style.setBorderRight(BorderStyle.THIN);
        return style;
    }
```

Why `protected`? - Accessible by subclasses - Not public (internal use only)

`workbook.createCellStyle()`: - Styles belong to the workbook, not the cell - Creates a reusable style object

`Font font = workbook.createFont()`: - Fonts are also workbook-level objects - Set properties then assign to style

`setFillForegroundColor()`: - Sets background color (confusing naming!) - `IndexedColors.GREY_25_PERCENT` is light gray

`setFillPattern(FillPatternType.SOLID_FOREGROUND)`: - Required for color to appear - SOLID_FOREGROUND = solid background fill

Borders: - `BorderStyle.THIN` = 1px border - Set all 4 sides for complete border

createCurrencyCellStyle Method

```
    protected CellStyle createCurrencyCellStyle(Workbook workbook) {
        CellStyle style = workbook.createCellStyle();
        CreationHelper createHelper = workbook.getCreationHelper();
        style.setDataFormat(createHelper.createDataFormat().getFormat(" #,##0.00"));
        return style;
    }
```

`CreationHelper`: Factory for creating format objects.

`setDataFormat()`: - Excel data format (not just display format) - Cell stores number, displays as 1,234.56 - Allows Excel to do calculations on the value

autoSizeColumns Method

```
    protected void autoSizeColumns(Sheet sheet, int columnCount) {
        for (int i = 0; i < columnCount; i++) {
            sheet.autoSizeColumn(i);
        }
    }
```

`autoSizeColumn(i)`: - Adjusts column width to fit content - Scans all rows to find widest content

writeToBytes Method

```
    protected byte[] writeToBytes(Workbook workbook) {
        try (ByteArrayOutputStream out = new ByteArrayOutputStream()) {
            workbook.write(out);
            return out.toByteArray();
        } catch (Exception e) {
            throw new RuntimeException("Excel generation failed", e);
        }
    }
}
```

`try-with-resources`: - `try (resource)` syntax - Automatically closes the stream when done

`ByteArrayOutputStream`: - Writes to memory (byte array) - Not to a file

`workbook.write(out)`: Serializes the workbook to the stream.

`out.toByteArray()`: Converts stream to byte array for HTTP response.

---

7. AbstractPdfExporter Base Class

AbstractPdfExporter.java

```
import com.itextpdf.text.*;
import com.itextpdf.text.pdf.*;
```

iTextPDF: Library for creating PDF files in Java. - `Document`: Represents the PDF document - `Font`: Font settings - `PdfPTable`, `PdfPCell`: Table elements

```
public abstract class AbstractPdfExporter {
```

Why abstract? Same reasons as Excel exporter.

createTitleFont Method

```
    protected Font createTitleFont() {
        return new Font(Font.FontFamily.HELVETICA, 16, Font.BOLD);
    }
```

Font constructor: `(FontFamily, size, style)` - `HELVETICA`: Safe PDF font (always available) - `16`: 16pt size - `BOLD`: Bold style

addTitle Method

```
    protected void addTitle(Document document, String title) throws DocumentException {
        Paragraph titlePara = new Paragraph(title, createTitleFont());
        titlePara.setAlignment(Element.ALIGN_CENTER);
        titlePara.setSpacingAfter(20f);
        document.add(titlePara);
    }
```

`Paragraph`: Block-level text element.

`setAlignment(Element.ALIGN_CENTER)`: Centers the title.

`setSpacingAfter(20f)`: 20pt space below the title.

`document.add()`: Adds element to the PDF.

createHeaderCell Method

```
    protected PdfPCell createHeaderCell(String text, Font font, BaseColor bgColor) {
        PdfPCell cell = new PdfPCell(new Phrase(text, font));
        cell.setBackgroundColor(bgColor);
        cell.setHorizontalAlignment(Element.ALIGN_CENTER);
        cell.setPadding(8f);
        return cell;
    }
```

`PdfPCell`: Table cell element.

`Phrase`: Inline text element (goes inside cell).

`setBackgroundColor()`: Cell background color.

`setPadding(8f)`: 8pt internal padding.

---

8. ClaimReportExcelExporter

ClaimReportExcelExporter.java

```
@Component
public class ClaimReportExcelExporter extends AbstractExcelExporter {
```

`@Component`: - Spring annotation for component scanning - Creates singleton bean - Enables dependency injection

`extends AbstractExcelExporter`: - Inherits all protected methods - Reuses header style, auto-sizing, etc.

export Method

```
    public byte[] export(List<ClaimReportDto> data) {
        try (Workbook workbook = new XSSFWorkbook()) {
            Sheet sheet = workbook.createSheet("Claims Summary");
```

`try (Workbook workbook = new XSSFWorkbook())`: - Creates new Excel 2007+ workbook (.xlsx) - XSSF = XML SpreadSheet Format - Auto-closes when done (try-with-resources)

`createSheet("Claims Summary")`: - Creates a worksheet tab - Name appears on the tab at bottom of Excel

```
            CellStyle headerStyle = createHeaderStyle(workbook);
            CellStyle currencyStyle = createCurrencyCellStyle(workbook);
```

Creates reusable styles from base class methods.

```
            // Header row
            createHeaderRow(sheet, headerStyle,
                    "Enrollment ID", "Claim ID", "Approved Amount", "Claim Date", "Status")
```

`createHeaderRow()`: Base class method creates row with styled headers.

```
            // Data rows
            int rowIdx = 1;
            for (ClaimReportDto dto : data) {
                Row row = sheet.createRow(rowIdx++);
```

`createRow(rowIdx++)`: - Creates row at index - `rowIdx++` increments after use (post-increment) - Starts at 1 (0 is header)

```
                row.createCell(0).setCellValue(dto.getEnrollmentId());

                Cell claimIdCell = row.createCell(1);
                if (dto.getClaimId() != null) {
                    claimIdCell.setCellValue(dto.getClaimId());
                } else {
                    claimIdCell.setCellValue("-");
                }
```

Null handling for claim ID: - Some enrollments may have no claims - Display "-" instead of blank

```
                Cell amountCell = row.createCell(2);
                amountCell.setCellValue(ReportFormatters.formatCurrencyRaw(dto.getApprovedAr
                amountCell.setCellStyle(currencyStyle);
```

Currency cell: - `formatCurrencyRaw()` returns number (not string) - Cell stores the number - `currencyStyle` displays it as 1,234.56 - Allows Excel SUM, AVG formulas to work

```
                row.createCell(3).setCellValue(ReportFormatters.formatDate(dto.getClaimDate(

                row.createCell(4).setCellValue(dto.getClaimStatus());
            }
```

Date formatting: Uses utility method for consistent format.

```
            autoSizeColumns(sheet, 5);
            return writeToBytes(workbook);
        } catch (Exception e) {
            throw new RuntimeException("Excel generation failed", e);
        }
    }
}
```

`autoSizeColumns(sheet, 5)`: Auto-sizes all 5 columns.

`writeToBytes(workbook)`: Converts to byte array for HTTP response.

---

9. ClaimReportPdfExporter

ClaimReportPdfExporter.java

```
@Component
public class ClaimReportPdfExporter extends AbstractPdfExporter {

    public byte[] export(List<ClaimReportDto> data) {
        Document document = new Document();
        ByteArrayOutputStream out = new ByteArrayOutputStream();
```

`Document`: Represents the PDF being built.

`ByteArrayOutputStream`: Collects PDF bytes in memory.

```
        try {
            PdfWriter.getInstance(document, out);
            document.open();
```

PdfWriter.getInstance(document, out): - Connects document to output stream - PDF content written as document is built

document.open(): Opens document for writing.

```
            addTitle(document, "Claims Summary Report");

            PdfPTable table = new PdfPTable(5);
            table.setWidthPercentage(100);
            table.setSpacingBefore(10f);
```

PdfPTable(5): Table with 5 columns.

setWidthPercentage(100): Table spans full page width.

setSpacingBefore(10f): 10pt space above table.

```
            // Headers
            Font headerFont = createHeaderFont();
            BaseColor headerBg = getHeaderBackgroundColor();
            String[] headers = {"Enrollment ID", "Claim ID", "Amount", "Date", "Status"};
            for (String h : headers) {
                table.addCell(createHeaderCell(h, headerFont, headerBg));
            }
```

Loop creates styled header cells using base class method.

```
            // Data
            Font dataFont = createDataFont();
            for (ClaimReportDto dto : data) {
                table.addCell(new Phrase(String.valueOf(dto.getEnrollmentId()), dataFont));
                table.addCell(new Phrase(dto.getClaimId() != null ?
                    String.valueOf(dto.getClaimId()) : "-", dataFont));
                table.addCell(new Phrase(ReportFormatters.formatCurrency(dto.getApprovedAmou
                table.addCell(new Phrase(ReportFormatters.formatDate(dto.getClaimDate()), da
                table.addCell(new Phrase(dto.getClaimStatus() != null ?
                    dto.getClaimStatus().name() : "-", dataFont));
            }
```

For PDF: `formatCurrency()` returns String “ 1,234.56” (not raw number).

name(): Gets enum constant name as String.

```
            document.add(table);
            document.close();
        } catch (Exception e) {
            throw new RuntimeException("PDF generation failed", e);
        }

        return out.toByteArray();
```

```
    }
}
```

`document.add(table)`: Adds table to PDF.

`document.close()`: Finalizes PDF (required!).

10-13. Other Exporters

The other exporters (Premium, Employee) follow the same patterns: - `@Component` annotation - Extend base class - Use `ReportFormatters` - Create workbook/document → add content → return bytes

---

14. HrReportService

HrReportService.java

This service handles the Employee Coverage Report with filtering, sorting, and pagination.

```
@Service
@RequiredArgsConstructor
public class HrReportService {
```

`@RequiredArgsConstructor`: - Lombok annotation - Generates constructor for all `final` fields - Spring uses this for dependency injection

```
    private final EmployeeRepository employeeRepository;
    private final EnrollmentRepository enrollmentRepository;
```

Dependencies injected via constructor (Lombok generates it).

```
    private static final Map<String, String> SORT_FIELD_MAP = Map.of(
            "name", "employeeName",
            "joiningDate", "joiningDate",
            "status", "status",
            "category", "category");
```

Whitelist of sortable fields: - Security: Prevents sorting by arbitrary fields (SQL injection protection) - Maps URL param name → DTO field name

getEmployeeCoverageReport Method

```
    public EmployeeCoverageReportResult getEmployeeCoverageReport(
            Long organizationId,
            String statusFilter,
            String categoryFilter,
            EnrollmentStateFilter enrollmentStateFilter,
            String sortBy,
            String sortDir,
            int page,
            int pageSize) {
```

All filtering/sorting/pagination params come from controller.

```
        // 1. Fetch all employees for organization
        List<Employee> employees = employeeRepository.findByOrganizationOrganizationId(organ
```

Fetches from database once, then filters in memory.

```
        // 2. Map to DTOs with resolved category and enrollment count
        List<EmployeeCoverageReportDTO> dtos = employees.stream()
                .map(emp -> {
                    Employee.EmployeeCategory resolvedCategory = resolveCategory(emp);
                    int activeEnrollments = enrollmentRepository.countByEmployeeAndEnrollmen
                            emp, EnrollmentStatus.ACTIVE);
                    return EmployeeCoverageReportDTO.fromEmployee(emp, resolvedCategory, act
                })
                .collect(Collectors.toList());
```

Stream operations: - `stream()`: Creates stream from list - `map()`: Transforms each Employee → DTO - `collect(Collectors.toList())`: Collects results into new list

`resolveCategory(emp)`: Calculates SENIOR/JUNIOR based on tenure.

`countByEmployeeAndEnrollmentStatus()`: Counts active enrollments per employee.

```
        // 3. Apply status filter
        if (statusFilter != null && !statusFilter.isEmpty() && !statusFilter.equalsIgnoreCas
            try {
                EmployeeStatus status = EmployeeStatus.valueOf(statusFilter.toUpperCase());
                dtos = dtos.stream()
                        .filter(dto -> dto.getStatus() == status)
                        .collect(Collectors.toList());
            } catch (IllegalArgumentException ignored) {
                // Invalid status - ignore filter
            }
        }
```

In-memory filtering: - Converts String → enum - `filter()` keeps only matching items - Gracefully ignores invalid values

```
        // 6. Apply sorting (whitelisted fields only)
        Comparator<EmployeeCoverageReportDTO> comparator = getComparator(sortBy, sortDir);
        dtos.sort(comparator);
```

Sorting: Uses Java Comparator with whitelist validation.

```
        // 7. Calculate pagination
        int totalElements = dtos.size();
        int totalPages = (int) Math.ceil((double) totalElements / pageSize);
        int fromIndex = Math.min(page * pageSize, totalElements);
        int toIndex = Math.min(fromIndex + pageSize, totalElements);

        List<EmployeeCoverageReportDTO> pageContent = dtos.subList(fromIndex, toIndex);
```

Manual pagination: - `totalElements`: Total matching records - `totalPages`: Ceiling division for page count - `subList()`: Gets page slice

```
        return new EmployeeCoverageReportResult(
                pageContent,
                page,
                pageSize,
                totalElements,
                totalPages);
    }
```

Returns result wrapper with pagination metadata.

EmployeeCoverageReportResult Record

```
    public record EmployeeCoverageReportResult(
            List<EmployeeCoverageReportDTO> content,
            int currentPage,
            int pageSize,
            int totalElements,
            int totalPages) {
        public boolean hasNext() {
            return currentPage < totalPages - 1;
        }

        public boolean hasPrevious() {
            return currentPage > 0;
        }
    }
```

record (Java 14+): - Immutable data class - Auto-generates constructor, getters, equals, hashCode, toString - Perfect for DTOs

---

15. HrController Export Endpoints

HrController.java

```
@Controller
@RequestMapping("/hr")
```

```
@RequiredArgsConstructor
public class HrController {
```

`@Controller`: Spring MVC controller (returns views or handles responses).

`@RequestMapping("/hr")`: Base URL for all endpoints in this controller.

```
    // Exporters (Spring-managed)
    private final ClaimReportExcelExporter claimReportExcelExporter;
    private final ClaimReportPdfExporter claimReportPdfExporter;
    private final EmployeeCoverageExcelExporter employeeCoverageExcelExporter;
    private final EmployeeCoveragePdfExporter employeeCoveragePdfExporter;
```

Dependency Injection: Spring injects the `@Component` exporters.

exportClaimsReportExcel Endpoint

```
    @GetMapping("/reports/claims/export/excel")
    public void exportClaimsReportExcel(@RequestParam(required = false) String sortBy,
            @RequestParam(required = false) String status,
            HttpServletResponse response) throws IOException {
```

`@GetMapping`: HTTP GET request handler.

`@RequestParam(required = false)`: Optional URL parameter.

`HttpServletResponse`: Direct access to HTTP response.

`void` return: We write directly to response (no view).

```
        List<ClaimReportDto> data = reportService.getClaimSummaryByEnrollment(status);
```

Fetches data from service layer.

```
        if ("dateDesc".equalsIgnoreCase(sortBy) || sortBy == null) {
            data.sort(Comparator.comparing(ClaimReportDto::getClaimDate,
                    Comparator.nullsLast(Comparator.reverseOrder())));
        } else if ("dateAsc".equalsIgnoreCase(sortBy)) {
            data.sort(Comparator.comparing(ClaimReportDto::getClaimDate,
                    Comparator.nullsLast(Comparator.naturalOrder())));
        } else if ("status".equalsIgnoreCase(sortBy)) {
            data.sort(Comparator.comparing(ClaimReportDto::getClaimStatus));
        }
```

Sorting logic: - `Comparator.comparing()`: Creates comparator from method reference - `nullsLast()`: Null values go to end - `reverseOrder()`: Descending order

```
        byte[] excelBytes = claimReportExcelExporter.export(data);
```

Calls injected exporter to generate Excel.

```
        response.setContentType("application/vnd.openxmlformats-officedocument.spreadsheetml
        response.setHeader("Content-Disposition", "attachment; filename=claims-report.xlsx")
        response.getOutputStream().write(excelBytes);
        response.getOutputStream().flush();
    }
```

Response writing: 1. Set MIME type (Excel) 2. Set download header 3. Write bytes to output stream 4. Flush to ensure all bytes are sent

---

16. Complete Flow Diagram

```
                         USER CLICKS EXPORT




 Browser sends GET request:
 /hr/reports/claims/export/excel?sortBy=dateDesc&status=APPROVED




 HrController.exportClaimsReportExcel():
 1. Extract @RequestParam values (sortBy, status)
 2. Call reportService.getClaimSummaryByEnrollment(status)




 ReportServiceImpl.getClaimSummaryByEnrollment():
 1. Build JPQL query with DTO constructor
 2. Convert status String → ClaimStatus enum
 3. Execute query via EntityManager
 4. Return List<ClaimReportDto>




 HrController (continued):
 1. Sort data based on sortBy parameter
 2. Call claimReportExcelExporter.export(data)
```

```
ClaimReportExcelExporter.export():
1. Create XSSFWorkbook
2. Call createHeaderStyle() (from AbstractExcelExporter)
3. Call createHeaderRow() (from AbstractExcelExporter)
4. Loop through data:
   - Create rows and cells
   - Use ReportFormatters.formatDate() for dates
   - Use ReportFormatters.formatCurrencyRaw() for amounts
5. Call autoSizeColumns() (from AbstractExcelExporter)
6. Call writeToBytes() (from AbstractExcelExporter)
7. Return byte[]
```

```
HrController (final steps):
1. response.setContentType("application/vnd...spreadsheetml.sheet")
2. response.setHeader("Content-Disposition", "attachment; filename=...")
3. response.getOutputStream().write(excelBytes)
4. response.getOutputStream().flush()
```

```
Browser:
1. Receives HTTP response with MIME type
2. Sees "Content-Disposition: attachment" header
3. Downloads file as "claims-report.xlsx"
```

Why This Design is Best

Design Choice	Why It's Good
Interface + Impl pattern	Abstraction, testability, flexibility
`@Component` exporters	Dependency injection, singleton, testable
Base classes	DRY, consistent styling, maintainable
Utility class	Centralized formatting, single source of truth
MIME types	Correct file type recognition by browsers
EntityManager for reports	Complex queries, DTO projections

Design Choice	Why It's Good
Enum comparisons	Type-safe, refactoring-friendly
Null checks everywhere	Graceful handling, no NPEs
try-with-resources	Automatic resource cleanup

Document generated for the Employee Insurance Management system report module.