

I'll explain this Java Spring Boot controller class line by line in detail.

```
package com.employeeinsurancemanagement.controller;
```

Declares the package where this class belongs - it's in the controller layer of an employee insurance management system.

Import Statements (Lines 3-28)

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

Importing DTO (Data Transfer Object) classes - these are lightweight objects used to transfer data between different layers. ClaimReportDto holds claim report data, EmployeeReportDto holds employee report data, PremiumReportDto holds premium report data.

```
import com.employeeinsurancemanagement.model.*;
```

Imports all model classes from the model package using wildcard (*) - these represent database entities like Employee, Claim, Policy, etc.

```
import com.employeeinsurancemanagement.repository.ClaimRepository;  
import com.employeeinsurancemanagement.dto.EmployeeCreateRequest;  
import com.employeeinsurancemanagement.repository.EmployeeRepository;
```

Importing Repository interfaces - these provide methods to interact with the database. ClaimRepository handles claim data, EmployeeRepository handles employee data. EmployeeCreateRequest is a DTO for creating new employees.

```
import com.employeeinsurancemanagement.service.*;
```

Imports all service classes - these contain business logic.

```
import com.employeeinsurancemanagement.repository.EnrollmentRepository;
import com.employeeinsurancemanagement.dto.EnrollmentStateFilter;
import com.employeeinsurancemanagement.repository.PolicyRepository;
import com.employeeinsurancemanagement.repository.UserRepository;
```

More repository and DTO imports - EnrollmentRepository for enrollment data, PolicyRepository for policy data, UserRepository for user data, EnrollmentStateFilter for filtering enrollments.

```
import com.employeeinsurancemanagement.security.SecurityUtil;
```

Imports security utility class - used to get current logged-in user information.

```
import jakarta.servlet.http.HttpServletResponse;
```

Imports HttpServletResponse - used to send HTTP responses (like file downloads) back to the client.

```
import jakarta.validation.Valid;
```

Imports validation annotation - triggers validation of request objects.

```
import lombok.RequiredArgsConstructor;
```

Lombok annotation - automatically generates a constructor with required fields (final fields).

```
import org.springframework.format.annotation.DateTimeFormat;
```

Spring annotation for formatting date/time parameters from HTTP requests.

```
import org.springframework.stereotype.Controller;
```

Spring annotation - marks this class as a web controller that handles HTTP requests.

```
import org.springframework.ui.Model;
```

Spring Model interface - used to pass data from controller to view templates.

```
import org.springframework.validation.BindingResult;
```

Holds validation errors after validating form data.

```
import org.springframework.web.bind.annotation.*;
```

Imports all web annotations like @GetMapping, @PostMapping, @RequestParam, etc.

```
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
```

Used to pass flash messages (temporary messages) when redirecting between pages.

```
import java.io.IOException;
import java.time.LocalDate;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
```

Java utility imports - IOException for handling errors, LocalDate for dates, Comparator for sorting, HashMap/Map for key-value storage, List for collections, Collectors for stream operations.

Class Declaration (Lines 30-32)

```
@Controller
```

Marks this class as a Spring MVC Controller - it will handle web requests and return view names.

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

Base URL mapping - all methods in this controller will be under the `/hr` path (e.g., `/hr/dashboard`, `/hr/employees`).

```
@RequiredArgsConstructor
```

Lombok annotation - automatically creates a constructor that initializes all `final` fields.

```
public class HrController {
```

Class definition - this is the HR (Human Resources) controller that handles HR-related operations.

Dependency Declarations (Lines 34-47)

```
private final EmployeeService employeeService;
```

Declares employeeService dependency - handles business logic for employee operations. `final` means it must be initialized in constructor and cannot be changed.

```
private final EmployeeRepository employeeRepository;
```

Repository for database operations on employees - provides methods like `findAll()`, `save()`, etc.

```
private final EnrollmentRepository enrollmentRepository;
```

Repository for enrollment data - handles policy enrollment database operations.

```
private final ClaimRepository claimRepository;
```

Repository for claim data - handles insurance claim database operations.

```
private final PolicyRepository policyRepository;
```

Repository for policy data - handles insurance policy database operations.

```
private final ReportService reportService;
```

Service for generating reports - business logic for creating various reports.

```
private final UserRepository userRepository;
```

Repository for user data - handles user account database operations.

```
private final HrReportService hrReportService;
```

Service specifically for HR reports - specialized reporting logic for HR users.

```
// Exporters (Spring-managed)  
private final ClaimReportExcelExporter claimReportExcelExporter;
```

Excel exporter for claim reports - converts claim data to Excel format. Comment indicates Spring manages this bean.

```
private final ClaimReportPdfExporter claimReportPdfExporter;
```

PDF exporter for claim reports - converts claim data to PDF format.

```
private final EmployeeCoverageExcelExporter employeeCoverageExcelExporter;
```

Excel exporter for employee coverage reports - exports employee insurance coverage to Excel.

```
private final EmployeeCoveragePdfExporter employeeCoveragePdfExporter;
```

PDF exporter for employee coverage reports - exports employee insurance coverage to PDF.

Helper Method: getCurrentUser() (Lines 49-52)

```
private User getCurrentUser() {
```

Private helper method that returns the currently logged-in user.

```
String email = SecurityUtil.getCurrentUserEmail();
```

Gets the email of logged-in user from the security context (from session/token).

```
return userRepository.findByEmail(email).orElseThrow();
```

Finds user by email in database. `findByEmail()` returns `Optional`. `orElseThrow()` throws exception if user not found, otherwise returns the `User` object.

```
}
```

Helper Method: getAdminContact() (Lines 54-66)

```
private Map<String, String> getAdminContact() {
```

Private helper method that returns admin contact information as a `Map`.

```
User admin = userRepository.findAll().stream()
```

Gets all users from database and converts to a stream for processing.

```
.filter(u -> u.getRole() == User.Role.ADMIN)
```

Filters to find only ADMIN users - checks if user's role equals ADMIN.

```
.findFirst()
```

Gets the first admin user found - returns Optional.

```
.orElse(null);
```

Returns null if no admin found - provides default value for Optional.

```
Map<String, String> contact = new HashMap<>();
```

Creates empty HashMap to store contact information.

```
if (admin != null) {
```

Checks if admin user exists (wasn't null).

```
contact.put("name", "Admin");
```

Adds "name" key with value "Admin" to the map.

```
contact.put("email", admin.getEmail());
```

Adds "email" key with the admin's email address to the map.

```
}  
return contact;  
}
```

Returns the contact map (empty if no admin found, or with name and email if admin exists).

Dashboard Method (Lines 68-99)

```
@GetMapping("/dashboard")
```

Maps HTTP GET requests to `/hr/dashboard` URL to this method.

```
public String dashboard(Model model) {
```

Method that handles dashboard page. Returns String (view name). `Model` parameter allows passing data to the view.

```
User currentUser = getCurrentUser();
```

Gets the currently logged-in HR user using the helper method.

```
Long orgId = currentUser.getOrganization().getOrganizationId();
```

Gets the organization ID from the current user's organization.

```
List<Employee> allEmployees = employeeRepository.findAll().stream()
```

Gets all employees from database and creates a stream for filtering.

```
.filter(e -> e.getOrganization().getOrganizationId().equals(orgId))
```

Filters to keep only employees that belong to the current user's organization.

```
.collect(Collectors.toList());
```

Collects filtered results back into a List.

```
long activeEmployees = allEmployees.stream()
    .filter(e -> e.getStatus() == EmployeeStatus.ACTIVE).count();
```

Counts active employees - filters employees with ACTIVE status and counts them.

```
long noticeEmployees = allEmployees.stream()
    .filter(e -> e.getStatus() == EmployeeStatus.NOTICE).count();
```

Counts employees in notice period - those who have resigned but haven't left yet.

```
long exitedEmployees = allEmployees.stream()
    .filter(e -> e.getStatus() == EmployeeStatus.EXITED).count();
```

Counts exited employees - those who have already left the organization.

```
List<Claim> pendingClaims = claimRepository.findAll().stream()
```

Gets all claims from database and streams them.

```
.filter(c -> c.getClaimStatus() == ClaimStatus.SUBMITTED)
```

Filters to pending claims - those with SUBMITTED status.

```
.filter(c -> c.getEmployee().getOrganization().getOrganizationId().equals(orgId))
```

Filters to claims from this organization only - checks if claim's employee belongs to current organization.

```
.collect(Collectors.toList()); // for the top card
```

Collects into list - comment indicates this is for displaying in a dashboard card.

```
// Count enrollments for THIS organization only
```

```
long totalEnrollments = enrollmentRepository.findAll().stream()
```

Gets all enrollments and starts streaming.

```
.filter(e -> e.getEmployee().getOrganization().getOrganizationId().equals(or
```

Filters enrollments to only those from this organization.

```
.count();
```

Counts the filtered enrollments.

```
model.addAttribute("currentHR", currentUser);
```

Adds current user to model - makes it available in the view template with key "currentHR".

```
model.addAttribute("organizationName", currentUser.getOrganization().getOr
```

Adds organization name to model - for display in the view.

```
model.addAttribute("totalEmployees", allEmployees.size());
```

Adds total employee count to model.

```
model.addAttribute("activeEmployees", activeEmployees);
```

Adds active employee count to model.

```
model.addAttribute("noticeEmployees", noticeEmployees);
```

Adds notice period employee count to model.

```
model.addAttribute("exitedEmployees", exitedEmployees);
```

Adds exited employee count to model.

```
model.addAttribute("totalEnrollments", totalEnrollments);
```

Adds total enrollment count to model.

```
model.addAttribute("pendingClaimsCount", pendingClaims.size());
```

Adds pending claims count to model.

```
model.addAttribute("recentEmployees", allEmployees.stream().limit(5).collect(C
```

Adds first 5 employees to model for "recent employees" section.

```
model.addAttribute("adminContact", getAdminContact());
```

Adds admin contact info to model using helper method.

```
model.addAttribute("recentPendingClaims", pendingClaims.stream().limit(5).cc
```

Adds first 5 pending claims for "recent pending claims" section.

```
return "hr/dashboard";
}
```

Returns view name "hr/dashboard" - Spring will look for a template file at templates/hr/dashboard.html .

Employees List Method (Lines 101-115)

```
@GetMapping("/employees")
public String employees(Model model) {
```

Handles GET requests to /hr/employees - displays list of all employees.

```
User currentUser = getCurrentUser();
Long orgId = currentUser.getOrganization().getOrganizationId();
```

Gets current user and their organization ID.

```
List<Employee> employees = employeeRepository.findAll().stream()
    .filter(e -> e.getOrganization().getOrganizationId().equals(orgId))
    .collect(Collectors.toList());
```

Gets all employees from this organization using the same filtering pattern.

```
model.addAttribute("employees", employees);
```

Adds employee list to model for display in view.

```
model.addAttribute("organizationId", orgId);
```

Adds organization ID to model.

```
model.addAttribute("organizationName", currentUser.getOrganization().getOrg
```

Adds organization name to model.

```
model.addAttribute("adminContact", getAdminContact());
```

Adds admin contact info to model.

```
return "hr/employees";
```

```
}
```

Returns "hr/employees" view name - will render the employees list page.

Create Employee Form (Lines 117-125)

```
@GetMapping("/create-employee")
public String createEmployeeForm(Model model) {
```

Handles GET request to show employee creation form.

```
User currentUser = getCurrentUser();
```

Gets current logged-in user.

```
model.addAttribute("employeeCreateRequest", new EmployeeCreateRequest());
```

Adds empty EmployeeCreateRequest object to model - form will bind to this object.

```
model.addAttribute("organizationId", currentUser.getOrganization().getOrganizationId());
model.addAttribute("organizationName", currentUser.getOrganization().getOrganizationName());
model.addAttribute("adminContact", getAdminContact());
```

Adds organization details and admin contact to model.

```
return "hr/create-employee";
}
```

Returns create employee form view.

Create Employee POST Handler (Lines 127-151)

```
@PostMapping("/create-employee")
```

Handles POST request when employee creation form is submitted.

```
public String createEmployee(  
    @Valid @ModelAttribute EmployeeCreateRequest employeeCreateRequest,
```

Method parameter with annotations: `@Valid` triggers validation, `@ModelAttribute` binds form data to the object.

```
BindingResult bindingResult,
```

Holds validation errors if any fields fail validation.

```
Model model,  
RedirectAttributes redirectAttributes) {
```

Model for view data, RedirectAttributes for flash messages after redirect.

```
User currentUser = getCurrentUser();
```

Gets current user.

```
if (bindingResult.hasErrors()) {
```

Checks if validation failed - if form data is invalid.

```
model.addAttribute("error", "Validation failed. Please check the fields below.
```

Adds error message to model for display.

```
model.addAttribute("organizationId", currentUser.getOrganization().getOrga  
model.addAttribute("adminContact", getAdminContact());
```

Re-adds necessary data to model (needed because we're returning to form, not redirecting).

```
return "hr/create-employee";
```

Returns to the form view to show errors.

```
}
```

```
try {
```

Start try block to catch any errors during employee creation.

```
employeeService.registerEmployee(employeeCreateRequest, currentUser.ge
```

Calls service method to register the new employee in database.

```
redirectAttributes.addFlashAttribute("success", "Employee registered succes
```

Adds success message to flash attributes - will be shown once after redirect.

```
} catch (Exception e) {
```

Catches any exception during registration.

```
model.addAttribute("error", e.getMessage());
```

Adds error message from exception to model.

```
model.addAttribute("organizationId", currentUser.getOrganization().getOrga
model.addAttribute("adminContact", getAdminContact());
return "hr/create-employee";
```

Re-adds data and returns to form to show error.

```

    }
    return "redirect:/hr/employees";
}

```

Redirects to employee list page on success. The `redirect:` prefix tells Spring to send HTTP redirect.

Resign Employee (Lines 153-165)

```
@PostMapping("/employees/{employeeId}/resign")
```

Handles POST to resign an employee. `{employeeId}` is a path variable.

```

public String resignEmployee(
    @PathVariable Long employeeId,

```

@PathVariable extracts `employeeId` from URL (e.g., `/employees/123/resign` gives `employeeId=123`).

```
@RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate
```

@RequestParam gets parameter from request. **@DateTimeFormat** converts string to `LocalDate` in ISO format (yyyy-MM-dd).

```
RedirectAttributes redirectAttributes) {
```

For flash messages after redirect.

```

try {
    employeeService.resignEmployee(employeeId, resignationDate);

```

Calls service to process resignation - likely changes status to NOTICE.

```
redirectAttributes.addFlashAttribute("success", "Employee moved to notice period");
```

Adds success flash message.

```
    } catch (Exception e) {
        redirectAttributes.addFlashAttribute("error", e.getMessage());
    }
```

Catches errors and adds error flash message.

```
    }
    return "redirect:/hr/employees";
}
```

Redirects back to employee list.

Exit Employee (Lines 167-179)

```
@PostMapping("/employees/{employeeId}/exit")
public String exitEmployee(
    @PathVariable Long employeeId,
    RedirectAttributes redirectAttributes) {
```

Handles POST to exit an employee - similar structure to resign.

```
    try {
        employeeService.exitEmployee(employeeId);
```

Calls service to exit employee - changes status to EXITED, calculates exit benefits.

```
        redirectAttributes.addFlashAttribute("success", "Employee exited successfully");
    } catch (Exception e) {
        redirectAttributes.addFlashAttribute("error", e.getMessage());
    }
```

```
return "redirect:/hr/employees";
}
```

Standard try-catch-redirect pattern.

Enrollments List (Lines 181-196)

```
@GetMapping("/enrollments")
public String enrollments(Model model) {
```

Shows list of policy enrollments.

```
User currentUser = getCurrentUser();
Organization organization = currentUser.getOrganization();
Long orgId = organization.getOrganizationId();
```

Gets current user, their organization, and org ID.

```
List<Enrollment> enrollments = enrollmentRepository.findAll()
    .stream()
    .filter(e -> e.getEmployee().getOrganization().getOrganizationId().equals(orgId))
    .collect(Collectors.toList());
```

Gets all enrollments for this organization - filters by comparing org IDs.

```
model.addAttribute("enrollments", enrollments);
model.addAttribute("organizationName", currentUser.getOrganization().getName());
model.addAttribute("adminContact", getAdminContact());
```

Adds data to model.

```
return "hr/enrollments";
}
```

Returns enrollments view.

Claims List (Lines 198-219)

```
@GetMapping("/claims")
public String claims(Model model) {
```

Shows list of insurance claims.

```
User currentUser = getCurrentUser();
Organization organization = currentUser.getOrganization();
Long orgId = organization.getOrganizationId();
```

Gets organization info.

```
List<Claim> allClaims = claimRepository.findAll().stream()
    .filter(c -> c.getEmployee().getOrganization().getOrganizationId().equals(orgId));
```

Gets all claims from this organization.

```
.sorted(Comparator.comparing((Claim c) -> c.getClaimStatus() == ClaimStatus.SUBMITTED,
    Comparator.reverseOrder()));
```

Sorts claims by status first - SUBMITTED claims get value 0, others get 1, so SUBMITTED appears first.

```
.thenComparing(Claim::getClaimDate, Comparator.reverseOrder()))
```

Then sorts by claim date in descending order (newest first) - `thenComparing` is a secondary sort.

```
.collect(Collectors.toList());
```

Collects sorted results.

```
List<Claim> pendingClaims = allClaims.stream()
    .filter(c -> c.getClaimStatus() == ClaimStatus.SUBMITTED)
```

```
.collect(Collectors.toList());
```

Creates separate list of just pending claims.

```
model.addAttribute("allClaims", allClaims);
model.addAttribute("pendingClaims", pendingClaims);
model.addAttribute("organizationName", organization.getOrganizationName());
model.addAttribute("adminContact", getAdminContact());
return "hr/claims";
}
```

Adds both lists to model and returns claims view.

Reports Page (Lines 221-256)

```
@GetMapping("/reports")
public String reports(@RequestParam(required = false) String sortBy,
                     @RequestParam(required = false) String status,
                     Model model) {
```

Shows reports page with optional sorting and filtering. `required = false` means parameters are optional.

```
try {
```

Try block to catch report generation errors.

```
User currentUser = getCurrentUser();
model.addAttribute("currentHR", currentUser);
model.addAttribute("organizationName", currentUser.getOrganization().getO
```

Gets user info and adds to model.

```
model.addAttribute("employeeReports",
```

```
reportService.getEmployeeCountByOrganization(currentUser.getOrgani
```

Gets employee count report from service and adds to model.

```
model.addAttribute("premiumReports",
    reportService.getPremiumCollectedByOrganization(currentUser.getOrg;
```

Gets premium collection report and adds to model.

```
// Fetch claim reports with status filter
List<ClaimReportDto> claimReports = reportService.getClaimSummaryByEnr
```

Gets claim summary report - filtered by status if provided.

```
// Sorting Logic
if ("dateDesc".equalsIgnoreCase(sortBy) || sortBy == null) { // Default
```

Checks if sort is "dateDesc" or null (default). `equalsIgnoreCase` compares strings ignoring case.

```
claimReports.sort(Comparator.comparing(ClaimReportDto::getClaimDate,
    Comparator.nullsLast(Comparator.reverseOrder())));
```

Sorts by claim date descending - newest first. `nullsLast` puts null dates at the end.

```
} else if ("dateAsc".equalsIgnoreCase(sortBy)) {
    claimReports.sort(Comparator.comparing(ClaimReportDto::getClaimDate,
        Comparator.nullsLast(Comparator.naturalOrder())));
```

Sorts by claim date ascending (oldest first) if sortBy is "dateAsc".

```
} else if ("status".equalsIgnoreCase(sortBy)) {
    claimReports.sort(Comparator.comparing(ClaimReportDto::getClaimStatu
```

Sorts by claim status if sortBy is "status".

```
}
```

```
model.addAttribute("claimReports", claimReports);
model.addAttribute("selectedSortBy", sortBy);
model.addAttribute("selectedStatus", status);
```

Adds sorted reports and current filter values to model (so form can show selected values).

```
return "hr/reports";
```

Returns reports view.

```
} catch (Exception e) {
    model.addAttribute("error", "Failed to load reports: " + e.getMessage());
    return "redirect:/hr/dashboard";
```

On error, adds error message and redirects to dashboard.

```
}
```

```
}
```

Export Claims to Excel (Lines 258-278)

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

Exports claims report to Excel file. void return type - writes directly to response.
throws IOException declares possible exception.

```
List<ClaimReportDto> data = reportService.getClaimSummaryByEnrollment(sta
```

Gets claim data with status filter.

```
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));
}
```

Same sorting logic as the reports page.

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

Generates Excel file as byte array using the exporter.

```
response.setContentType("application/vnd.openxmlformats-officedocument.s
```

Sets response content type to Excel MIME type - tells browser it's an Excel file.

```
response.setHeader("Content-Disposition", "attachment; filename=claims-repo
```

Sets header to trigger download with filename "claims-report.xlsx". `attachment` makes browser download instead of display.

```
response.getOutputStream().write(excelBytes);
```

Writes Excel bytes to response output stream.

```
response.getOutputStream().flush();
```

Flushes stream - ensures all data is sent.

```
}
```

Export Claims to PDF (Lines 280-300)

```
@GetMapping("/reports/claims/export/pdf")
public void exportClaimsReportPdf(@RequestParam(required = false) String sortBy,
    @RequestParam(required = false) String status,
    HttpServletResponse response) throws IOException {
    List<ClaimReportDto> data = reportService.getClaimSummaryByEnrollment(status);
    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));
    }

    byte[] pdfBytes = claimReportPdfExporter.export(data);
```

Same as Excel export but generates PDF instead of Excel.

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

Sets content type to PDF.

```
response.setHeader("Content-Disposition", "attachment; filename=claims-report.pdf");
response.getOutputStream().write(pdfBytes);
response.getOutputStream().flush();
}
```

Sets PDF filename and writes to response.

Policies List (Lines 302-326)

```
@GetMapping("/policies")
public String policies(Model model) {
```

Shows list of insurance policies.

```
User currentUser = getCurrentUser();
Organization organization = currentUser.getOrganization();
Long orgId = organization.getOrganizationId();
```

Gets organization info.

```
// Get ALL policies for this organization from PolicyRepository
List<Policy> allOrgPolicies = policyRepository.findByOrganization(organization)
```

Gets all policies for this organization using a custom repository method.

```
// Count active enrollments per policy - filter by organization ID
Map<Long, Long> enrollmentCountByPolicy = enrollmentRepository.findAll().st
```

Creates a map of policy ID to enrollment count.

```
.filter(e -> e.getPolicy().getOrganization().getOrganizationId().equals(orgId))
```

Filters enrollments to this organization.

```
.filter(e -> e.getEnrollmentStatus() == EnrollmentStatus.ACTIVE)
```

Filters to only ACTIVE enrollments.

```
.collect(Collectors.groupingBy(
    e -> e.getPolicy().getPolicyId(),
    Collectors.counting()));
```

Groups by policy ID and counts each group. Result: Map<PolicyId, Count>.

```
// Calculate total enrollments (sum of all enrollment counts)
long totalEnrollments = enrollmentCountByPolicy.values().stream().mapToLong{
```

Sums all counts from the map - total active enrollments across all policies.

```
model.addAttribute("policies", allOrgPolicies);
model.addAttribute("enrollmentCounts", enrollmentCountByPolicy);
model.addAttribute("totalEnrollments", totalEnrollments);
model.addAttribute("organizationName", organization.getOrganizationName());
model.addAttribute("adminContact", getAdminContact());

return "hr/policies";
}
```

Adds data to model and returns policies view.

Employee Coverage Report (Lines 328-383)

```
/**
 * HR Employee Coverage Report with filtering, sorting, and pagination.
 * Controller stays thin - all logic is in HrReportService.
 */
```

JavaDoc comment explaining the method's purpose.

```
@GetMapping("/employees-report")
public String employeesReport(
```

Shows detailed employee coverage report with advanced filtering.

```
@RequestParam(defaultValue = "ALL") String status,
@RequestParam(defaultValue = "ALL") String category,
@RequestParam(defaultValue = "ALL") String enrollmentState,
```

Request parameters for filtering - all default to "ALL" (no filter).

```
@RequestParam(defaultValue = "name") String sortBy,
@RequestParam(defaultValue = "asc") String sortDir,
```

Sorting parameters - sort by "name" in "ascending" order by default.

```
@RequestParam(defaultValue = "10") int pageSize,
@RequestParam(defaultValue = "0") int page,
```

Pagination parameters - show 10 records per page, start at page 0.

```
Model model) {

    User currentUser = getCurrentUser();
    Organization organization = currentUser.getOrganization();
```

Gets current user and organization.

```
// Parse enrollment state filter
EnrollmentStateFilter enrollmentFilter;
try {
    enrollmentFilter = EnrollmentStateFilter.valueOf(enrollmentState.toUpperCa
```

Converts string to enum - `valueOf` converts "ALL" to `EnrollmentStateFilter.ALL`.

```
} catch (IllegalArgumentException e) {
    enrollmentFilter = EnrollmentStateFilter.ALL;
```

If conversion fails (invalid value), default to ALL.

```
}
```

```
// Validate page size (must be between 1 and 300)
if (pageSize < 1 || pageSize > 300) {
```

```
    pageSize = 10; // default fallback
}
```

Validates page size - must be 1-300, otherwise use default 10.

```
List<Employee> employees = employeeRepository
    .findByOrganizationId(organization.getOrganizationId());
int totalEmployees = employees.size();
```

Gets all employees for this organization and counts them.

```
// Validate page size (must be between 1 and totalEmployees, capped at 300)
if (pageSize < 1) {
    pageSize = 10; // fallback
} else if (pageSize > totalEmployees) {
    pageSize = totalEmployees; // cap at employee count
}
```

Further validation - caps page size at total employee count.

```
// Optional: still enforce an absolute max of 300
if (pageSize > 300) {
    pageSize = 300;
}
```

Enforces absolute maximum of 300 records per page.

```
// Get report from service (all logic is there)
HrReportService.EmployeeCoverageReportResult result = hrReportService.getReport(
    organization.getOrganizationId(),
    status,
    category,
    enrollmentFilter,
    sortBy,
    sortDir,
```

```

        page,
        pageSize);

```

Calls service to generate report - passes all filter/sort/pagination parameters.

```

// Pass data to view
model.addAttribute("employees", result.content());

```

Adds employee list for current page to model.

```

model.addAttribute("currentPage", result.currentPage());
model.addAttribute("pageSize", result.pageSize());
model.addAttribute("totalElements", result.totalElements());
model.addAttribute("totalPages", result.totalPages());

```

Adds pagination metadata to model.

```

model.addAttribute("hasNext", result.hasNext());
model.addAttribute("hasPrevious", result.hasPrevious());

```

Adds navigation flags - can user go to next/previous page?

```

// Pass current filter values back to view for form state
model.addAttribute("selectedStatus", status);
model.addAttribute("selectedCategory", category);
model.addAttribute("selectedEnrollmentState", enrollmentState);
model.addAttribute("selectedSortBy", sortBy);
model.addAttribute("selectedSortDir", sortDir);

```

Adds current filter values - so form shows what's selected.

```

model.addAttribute("organizationName", organization.getOrganizationName())

return "hr/employees-report";
}

```

Adds org name and returns view.

Export Employee Report to Excel (Lines 385-424)

```
/**
 * Export filtered employee report to Excel.
 * Exports ALL matching records (not just current page).
 */
```

Comment explains this exports everything, not just one page.

```
@GetMapping("/employees-report/export/excel")
public void exportEmployeesReportExcel(
    @RequestParam(defaultValue = "ALL") String status,
    @RequestParam(defaultValue = "ALL") String category,
    @RequestParam(defaultValue = "ALL") String enrollmentState,
    @RequestParam(defaultValue = "name") String sortBy,
    @RequestParam(defaultValue = "asc") String sortDir,
    HttpServletResponse response) throws IOException {
```

Export endpoint - takes same filter parameters but no pagination.

```
User currentUser = getCurrentUser();
Organization organization = currentUser.getOrganization();

// Parse enrollment state filter
EnrollmentStateFilter enrollmentFilter;
try {
    enrollmentFilter = EnrollmentStateFilter.valueOf(enrollmentState.toUpperCase());
} catch (IllegalArgumentException e) {
    enrollmentFilter = EnrollmentStateFilter.ALL;
}
```

Same filter parsing logic.

```
// Get ALL filtered data (large page size to get all)
HrReportService.EmployeeCoverageReportResult result = hrReportService.getEmployeeCoverageReport(
    organization.getOrganizationId(),
    status,
    category,
    enrollmentFilter,
    sortBy,
    sortDir,
    0,
    10000 // Get all records
);
```

Gets report with page size 10000 - effectively gets all records.

```
// Build filter description for export header
String filters = buildFilterDescription(status, category, enrollmentState, sortBy,
```

Creates human-readable description of applied filters.

```
// Generate Excel
byte[] excelBytes = employeeCoverageExcelExporter.export(result.content(), fil
```

Generates Excel file with data and filter description.

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

Standard Excel download response.

Export Employee Report to PDF (Lines 426-464)

```

/**
 * Export filtered employee report to PDF.
 * Exports ALL matching records (not just current page).
 */
@GetMapping("/employees-report/export/pdf")
public void exportEmployeesReportPdf(
    @RequestParam(defaultValue = "ALL") String status,
    @RequestParam(defaultValue = "ALL") String category,
    @RequestParam(defaultValue = "ALL") String enrollmentState,
    @RequestParam(defaultValue = "name") String sortBy,
    @RequestParam(defaultValue = "asc") String sortDir,
    HttpServletResponse response) throws IOException {

    User currentUser = getCurrentUser();
    Organization organization = currentUser.getOrganization();

    // Parse enrollment state filter
    EnrollmentStateFilter enrollmentFilter;
    try {
        enrollmentFilter = EnrollmentStateFilter.valueOf(enrollmentState.toUpperCase());
    } catch (IllegalArgumentException e) {
        enrollmentFilter = EnrollmentStateFilter.ALL;
    }

    // Get ALL filtered data
    HrReportService.EmployeeCoverageReportResult result = hrReportService.getEmployeeCoverageReport(
        organization.getOrganizationId(),
        status,
        category,
        enrollmentFilter,
        sortBy,
        sortDir,
        0,
        10000 // Get all records
    );

    // Build filter description for export header
    String filters = buildFilterDescription(status, category, enrollmentState, sortBy,

```

```
// Generate PDF
byte[] pdfBytes = employeeCoveragePdfExporter.export(result.content(), filters);

// Set response headers
response.setContentType("application/pdf");
response.setHeader("Content-Disposition", "attachment; filename=employee-c");
response.getOutputStream().write(pdfBytes);
response.getOutputStream().flush();
}
```

Identical to Excel export but generates PDF instead.

Build Filter Description Helper (Lines 466-481)

```
/**
 * Build a human-readable filter description for export headers.
 */
private String buildFilterDescription(String status, String category, String enrollment,
String sortBy, String sortDir) {
```

Helper method to create readable filter text.

```
StringBuilder sb = new StringBuilder();
```

Creates StringBuilder for efficient string building.

```
if (!"ALL".equalsIgnoreCase(status)) {
    sb.append("Status: ").append(status).append(" | ");
}
```

If status filter applied, adds it to description.

```
if (!"ALL".equalsIgnoreCase(category)) {
    sb.append("Category: ").append(category).append(" | ");
}
```

If category filter applied, adds it.

```
if (!"ALL".equalsIgnoreCase(enrollmentState)) {
    sb.append("Enrollment: ").append(enrollmentState.replace("_", " ")).append('
}
}
```

If enrollment filter applied, adds it (replaces underscores with spaces for readability).

```
sb.append("Sorted by: ").append(sortBy).append(" (").append(sortDir.toUpperC
```

Always adds sort info - e.g., "Sorted by: name (ASC)".

```
return sb.toString();
}
```

Returns the built string.

Export Employee Summary to Excel (Lines 483-495)

```
@GetMapping("/reports/employees/excel")
public void exportEmployeesReportExcel(HttpServletResponse response) throws
```

Exports simple employee count report to Excel (different from detailed coverage report).

```
User currentUser = getCurrentUser();
Long orgId = currentUser.getOrganization().getOrganizationId();
```

Gets organization ID.

```
List<EmployeeReportDto> data = reportService.getEmployeeCountByOrganiza
```

Gets employee count summary data.

```
EmployeeReportExcelExporter exporter = new EmployeeReportExcelExporter()
```

Creates new exporter instance - not injected, created manually.

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

Generates Excel bytes.

```
response.setContentType("application/vnd.openxmlformats-officedocument.s
response.setHeader("Content-Disposition", "attachment; filename=employee-r
response.getOutputStream().write(excelBytes);
response.getOutputStream().flush();
}
```

Standard Excel download.

Export Employee Summary to PDF (Lines 497-509)

```
@GetMapping("/reports/employees/pdf")
public void exportEmployeesReportPdf(HttpServletResponse response) throws IO
    User currentUser = getCurrentUser();
    Long orgId = currentUser.getOrganization().getOrganizationId();

    List<EmployeeReportDto> data = reportService.getEmployeeCountByOrganiza

    EmployeeReportPdfExporter exporter = new EmployeeReportPdfExporter();
    byte[] pdfBytes = exporter.export(data);

    response.setContentType("application/pdf");
    response.setHeader("Content-Disposition", "attachment; filename=employee-r
    response.getOutputStream().write(pdfBytes);
    response.getOutputStream().flush();
}
```

Same as Excel but PDF - exports employee count summary.

Export Premium Report to Excel (Lines 511-523)

```
@GetMapping("/reports/premium/export/excel")
public void exportPremiumReportExcel(HttpServletResponse response) throws IOException {
```

Exports premium collection report to Excel.

```
    User currentUser = getCurrentUser();
    Long orgId = currentUser.getOrganization().getOrganizationId();

    List<PremiumReportDto> data = reportService.getPremiumCollectedByOrganization(orgId);
```

Gets premium data for organization.

```
    PremiumReportExcelExporter exporter = new PremiumReportExcelExporter();
    byte[] excelBytes = exporter.export(data);

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

Standard Excel export.

Export Premium Report to PDF (Lines 525-537)

```
@GetMapping("/reports/premium/export/pdf")
public void exportPremiumReportPdf(HttpServletResponse response) throws IOException {
    User currentUser = getCurrentUser();
    Long orgId = currentUser.getOrganization().getOrganizationId();

    List<PremiumReportDto> data = reportService.getPremiumCollectedByOrganization(orgId);

    PremiumReportPdfExporter exporter = new PremiumReportPdfExporter();
    byte[] pdfBytes = exporter.export(data);
```

```
response.setContentType("application/pdf");  
response.setHeader("Content-Disposition", "attachment; filename=premium-r  
response.getOutputStream().write(pdfBytes);  
response.getOutputStream().flush();  
}
```

Same as Excel but PDF - exports premium report.

```
}
```

End of class.

Summary of What This Controller Does:

This is an HR management controller for an employee insurance system. It provides endpoints for:

1. **Dashboard** - Overview with stats and recent activity
2. **Employee Management** - List, create, resign, and exit employees
3. **Enrollments** - View policy enrollments
4. **Claims** - View and manage insurance claims
5. **Policies** - View insurance policies and enrollment counts
6. **Reports** - Multiple types (employee, premium, claims, coverage) with filtering, sorting, and pagination
7. **Exports** - Download reports as Excel or PDF files

All operations are **scoped to the HR user's organization** - they only see data for their own organization.