CLAIM MODULE - COMPREHENSIVE TECHNICAL DOCUMENTATION

Health Insurance Claim Management System (HICMS)

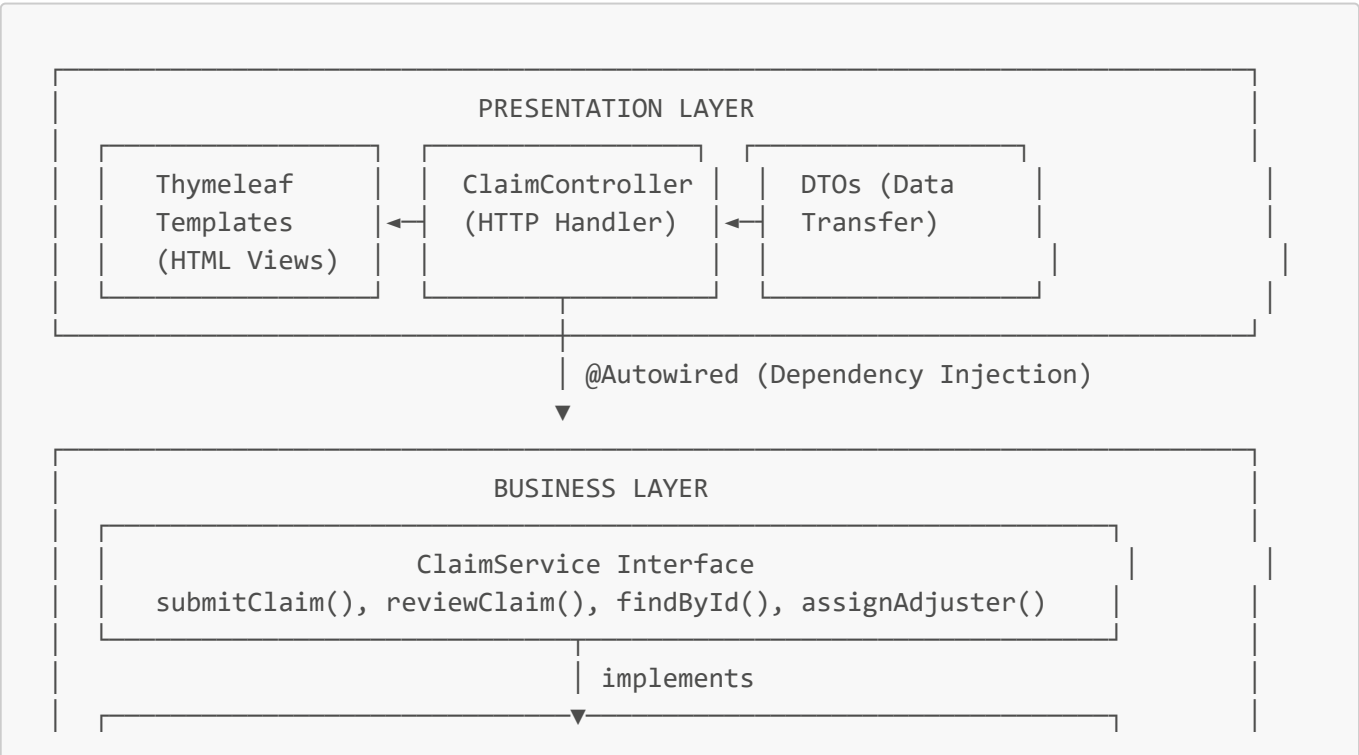
TABLE OF CONTENTS

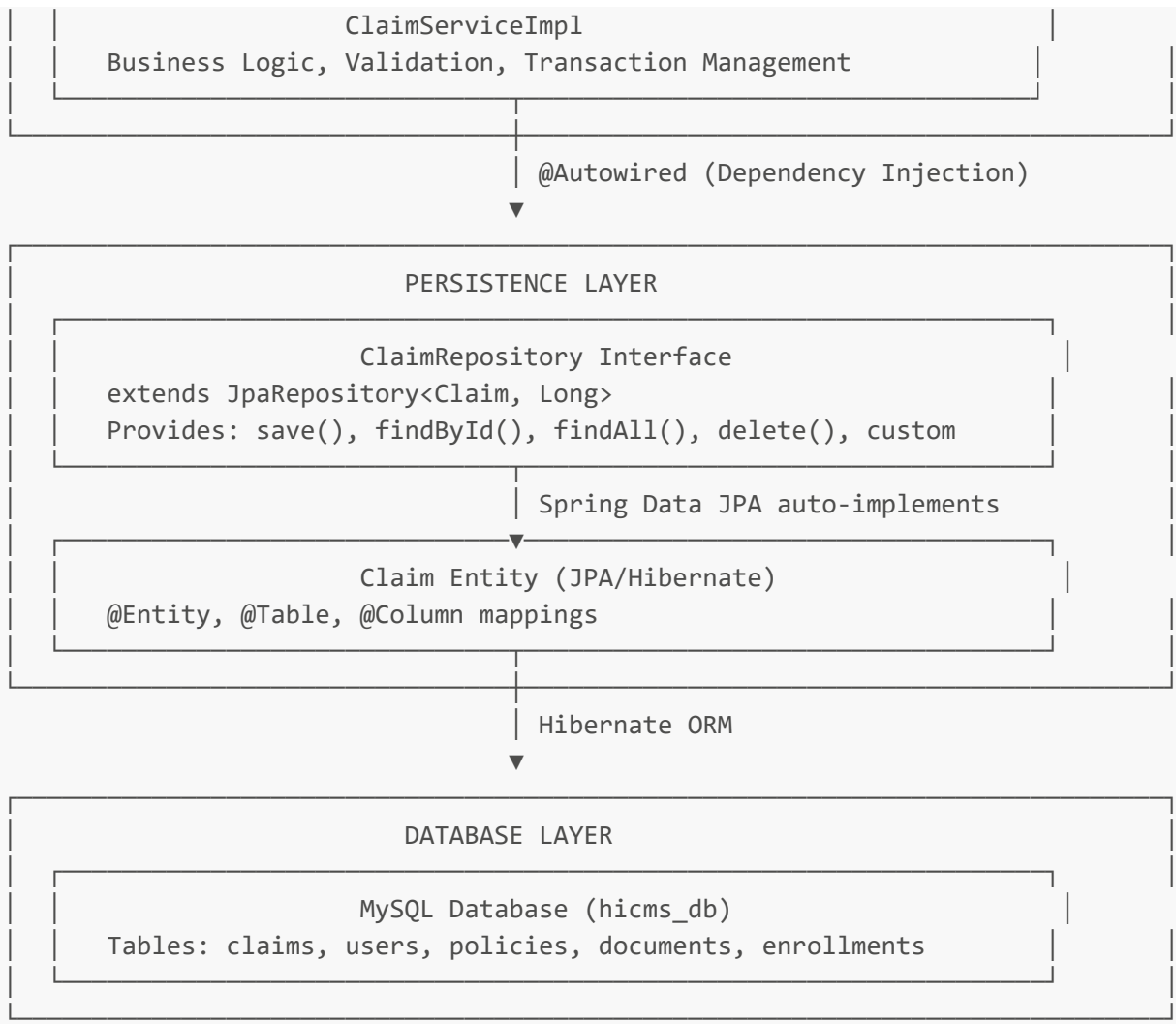
1. MODULE OVERVIEW & ARCHITECTURE

What is the Claim Module?

The Claim Module manages the complete lifecycle of insurance claims—from submission by policyholders to review and approval/rejection by claim adjusters.

High-Level Architecture





Design Patterns Used

Pattern	Where Used	Purpose
MVC	Controller-Service-Repository	Separation of concerns
DTO	ClaimDTO, ClaimReviewDTO	Data transfer between layers
Repository	ClaimRepository	Abstracts data access
Dependency Injection	@Autowired/@RequiredArgsConstructor	Loose coupling
Builder	Claim.builder()	Object construction
Template Method	@PrePersist, @PreUpdate	Lifecycle callbacks

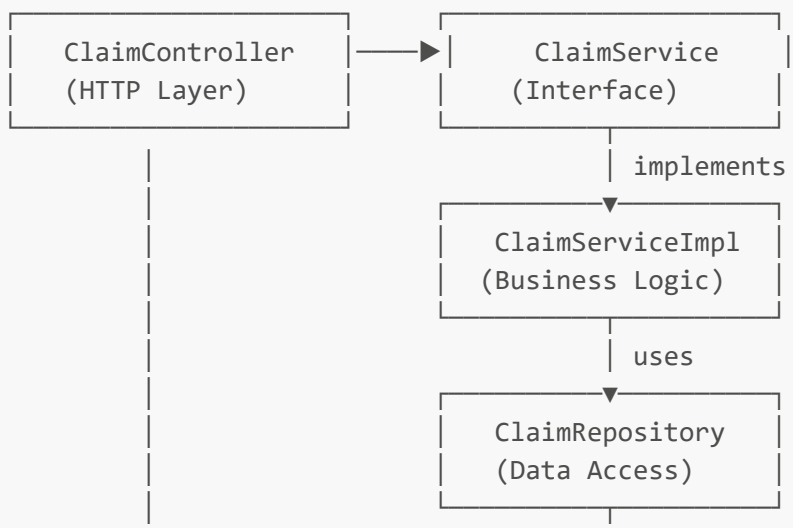
2. FILE STRUCTURE & ORGANIZATION

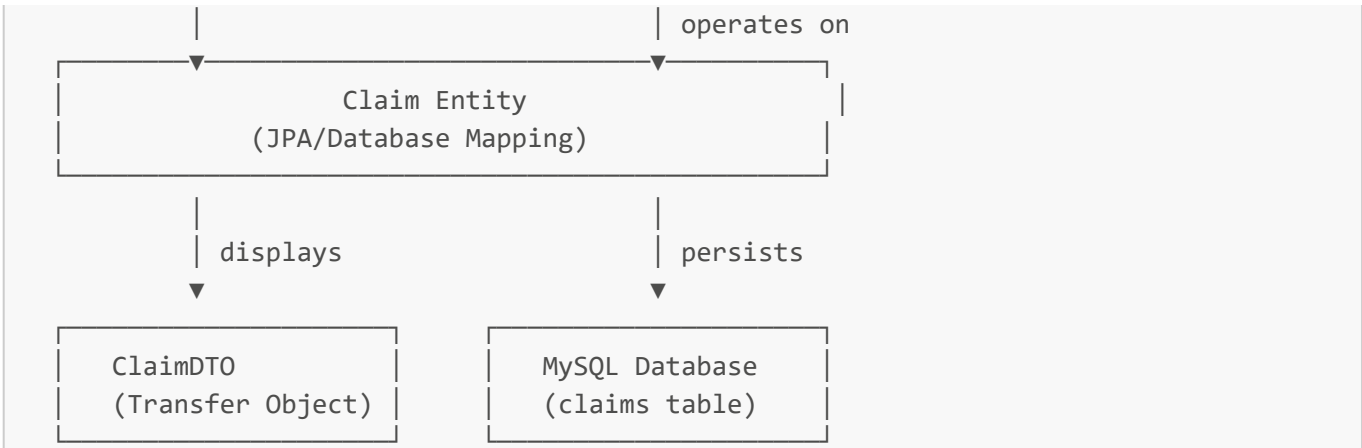
Directory Layout

```
src/main/java/com/hicms/
├── entity/                          # JPA Entities (Database Tables)
│   ├── Claim.java                   # Main claim entity
│   ├── ClaimStatus.java             # Enum: PENDING, APPROVED, REJECTED, etc.
│   ├── User.java                    # User entity
│   ├── Policy.java                  # Policy entity
│   └── Document.java                # Document entity
│
├── dto/                             # Data Transfer Objects
│   ├── ClaimDTO.java                # For claim submission/display
│   └── ClaimReviewDTO.java          # For claim review operations
│
├── repository/                      # Data Access Layer
│   └── ClaimRepository.java         # CRUD + custom queries
│
├── service/                         # Business Logic Layer
│   ├── ClaimService.java            # Interface (contract)
│   └── impl/
│       └── ClaimServiceImpl.java    # Implementation
│
├── controller/                      # HTTP Request Handlers
│   └── ClaimController.java         # REST/Web endpoints
│
└── config/                          # Configuration
    └── SecurityConfig.java          # Spring Security setup

src/main/resources/
├── application.properties           # Database, JPA, server config
└── templates/claim/                 # Thymeleaf HTML templates
    ├── list.html                    # Claim listing page
    ├── view.html                    # Claim details page
    ├── submit.html                  # Claim submission form
    └── review.html                  # Claim review form
```

File Interconnections Diagram





3. LAYER-BY-LAYER DEEP DIVE

3.1 ENTITY LAYER (Claim.java)

What is an Entity?

An Entity is a Java class that maps to a database table. Each instance = one row. Each field = one column.

Complete Code with Explanations

```
package com.hicms.entity;

import jakarta.persistence.*;      // JPA annotations
import lombok.*;                    // Reduces boilerplate
import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity                             // ← Marks as JPA entity (database table)
@Table(name = "claims")             // ← Explicit table name
@Getter @Setter                     // ← Lombok: generates all getters/setters
@NoArgsConstructor                  // ← Lombok: public Claim() {}
@AllArgsConstructor                 // ← Lombok: constructor with all fields
@Builder                            // ← Lombok: enables Claim.builder().build()
public class Claim {

    // ═══════════════════════════════════════════════════════════
    // PRIMARY KEY
    // ═══════════════════════════════════════════════════════════
    @Id                                             // ← This is the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY)  // ← Auto-increment
    private Long claimId;
    // SQL: claim_id BIGINT PRIMARY KEY AUTO_INCREMENT
```

```
    // ═══════════════════════════════════════════════════════════════
    // BUSINESS IDENTIFIER
    // ═══════════════════════════════════════════════════════════════
    @Column(unique = true, nullable = false, length = 50)
    private String claimNumber;
    // SQL: claim_number VARCHAR(50) UNIQUE NOT NULL
    // Example: CLM-20260131-001

    // ═══════════════════════════════════════════════════════════════
    // RELATIONSHIPS (Foreign Keys)
    // ═══════════════════════════════════════════════════════════════

    // Many claims belong to ONE policy
    @ManyToOne(fetch = FetchType.LAZY)       // ← Lazy loading (load when needed)
    @JoinColumn(name = "policy_id", nullable = false)  // ← FK column name
    private Policy policy;
    // SQL: policy_id BIGINT NOT NULL FOREIGN KEY REFERENCES policies(policy_id)

    // Many claims filed by ONE claimant (user)
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "claimant_id", nullable = false)
    private User claimant;

    // Many claims may be filed by ONE agent (optional)
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "agent_id")            // ← No nullable=false, so it can be
NULL
    private User agent;

    // Many claims reviewed by ONE adjuster (optional initially)
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "adjuster_id")
    private User adjuster;

    // ═══════════════════════════════════════════════════════════════
    // MONETARY FIELDS
    // ═══════════════════════════════════════════════════════════════
    @Column(name = "claim_amount", precision = 12, scale = 2, nullable = false)
    private BigDecimal claimAmount;
    // SQL: claim_amount DECIMAL(12,2) NOT NULL
    // precision=12: total digits, scale=2: decimal places
    // Example: 9999999999.99

    @Column(name = "approved_amount", precision = 12, scale = 2)
    private BigDecimal approvedAmount;        // ← Set when approved

    // ═══════════════════════════════════════════════════════════════
    // DATE FIELDS
    // ═══════════════════════════════════════════════════════════════
    @Column(name = "claim_date", nullable = false)
    private LocalDate claimDate;
    // SQL: claim_date DATE NOT NULL

    // ═══════════════════════════════════════════════════════════════
```

```
    // TEXT FIELDS
    // ═══════════════════════════════════════════════════════════════
    @Column(length = 1000)
    private String description;                // ← Detailed claim description

    @Column(length = 500)
    private String reason;                     // ← Short reason for claim

    @Column(length = 1000)
    private String remarks;                    // ← Adjuster's comments

    // ═══════════════════════════════════════════════════════════════
    // ENUM FIELD (Status)
    // ═══════════════════════════════════════════════════════════════
    @Enumerated(EnumType.STRING)               // ← Store as "PENDING" not 0
    @Column(name = "claim_status", nullable = false)
    private ClaimStatus claimStatus;
    // SQL: claim_status VARCHAR(255) NOT NULL
    // Values: PENDING, UNDER_REVIEW, APPROVED, REJECTED, CANCELLED

    // ═══════════════════════════════════════════════════════════════
    // AUDIT TIMESTAMPS
    // ═══════════════════════════════════════════════════════════════
    @Column(name = "created_date")
    private LocalDateTime createdDate;

    @Column(name = "updated_date")
    private LocalDateTime updatedDate;

    // ═══════════════════════════════════════════════════════════════
    // ONE-TO-MANY RELATIONSHIP (Documents)
    // ═══════════════════════════════════════════════════════════════
    @Builder.Default                           // ← Initialize in builder
    @OneToMany(mappedBy = "claim",             // ← The "claim" field in Document
               cascade = CascadeType.ALL,      // ← Operations cascade to
documents
               fetch = FetchType.LAZY)
    private List<Document> documents = new ArrayList<>();

    // ═══════════════════════════════════════════════════════════════
    // LIFECYCLE CALLBACKS (Automatic timestamp management)
    // ═══════════════════════════════════════════════════════════════
    @PrePersist                                // ← Called BEFORE INSERT
    protected void onCreate() {
        createdDate = LocalDateTime.now();
        updatedDate = LocalDateTime.now();
        if (claimDate == null) {
            claimDate = LocalDate.now();
        }
    }

    @PreUpdate                                 // ← Called BEFORE UPDATE
    protected void onUpdate() {
        updatedDate = LocalDateTime.now();
```

```
    }
}
```

Generated Database Table

```
CREATE TABLE claims (
    claim_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    claim_number VARCHAR(50) UNIQUE NOT NULL,
    policy_id BIGINT NOT NULL,
    claimant_id BIGINT NOT NULL,
    agent_id BIGINT,
    adjuster_id BIGINT,
    claim_amount DECIMAL(12,2) NOT NULL,
    approved_amount DECIMAL(12,2),
    claim_date DATE NOT NULL,
    description VARCHAR(1000),
    reason VARCHAR(500),
    claim_status VARCHAR(255) NOT NULL,
    remarks VARCHAR(1000),
    created_date DATETIME,
    updated_date DATETIME,

    FOREIGN KEY (policy_id) REFERENCES policies(policy_id),
    FOREIGN KEY (claimant_id) REFERENCES users(user_id),
    FOREIGN KEY (agent_id) REFERENCES users(user_id),
    FOREIGN KEY (adjuster_id) REFERENCES users(user_id)
);
```

3.2 REPOSITORY LAYER (ClaimRepository.java)

What is a Repository?

A Repository provides database operations. Spring Data JPA **auto-implements** methods based on naming conventions!

Complete Code with Explanations

```
package com.hicms.repository;

import com.hicms.entity.Claim;
import com.hicms.entity.ClaimStatus;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import java.util.List;
import java.util.Optional;
```

```java
@Repository   // ← Marks as Spring data access component (optional but
recommended)
public interface ClaimRepository extends JpaRepository<Claim, Long> {
    //                                                  ↑       ↑
    //                                               Entity    PK Type

    // ═══════════════════════════════════════════════════════════════════
    // INHERITED FROM JpaRepository (No code needed!)
    // ═══════════════════════════════════════════════════════════════════
    // save(Claim claim)              → INSERT or UPDATE
    // findById(Long id)              → SELECT * WHERE claim_id = ?
    // findAll()                      → SELECT * FROM claims
    // deleteById(Long id)            → DELETE FROM claims WHERE claim_id = ?
    // count()                        → SELECT COUNT(*) FROM claims
    // existsById(Long id)            → SELECT COUNT(*) > 0 WHERE claim_id = ?

    // ═══════════════════════════════════════════════════════════════════
    // QUERY METHODS BY NAMING CONVENTION
    // Spring parses method name → generates SQL automatically!
    // ═══════════════════════════════════════════════════════════════════

    Optional<Claim> findByClaimNumber(String claimNumber);
    // Generated: SELECT * FROM claims WHERE claim_number = ?

    boolean existsByClaimNumber(String claimNumber);
    // Generated: SELECT COUNT(*) > 0 FROM claims WHERE claim_number = ?

    List<Claim> findByClaimStatus(ClaimStatus status);
    // Generated: SELECT * FROM claims WHERE claim_status = ?

    List<Claim> findByClaimStatusIn(List<ClaimStatus> statuses);
    // Generated: SELECT * FROM claims WHERE claim_status IN (?, ?, ...)

    // Navigating relationships: claimant.userId
    List<Claim> findByClaimantUserId(Long claimantId);
    // Generated: SELECT c.* FROM claims c
    //            JOIN users u ON c.claimant_id = u.user_id
    //            WHERE u.user_id = ?

    List<Claim> findByAgentUserId(Long agentId);
    // Generated: SELECT * FROM claims WHERE agent_id = ?

    List<Claim> findByAdjusterUserId(Long adjusterId);
    // Generated: SELECT * FROM claims WHERE adjuster_id = ?

    List<Claim> findByPolicyPolicyId(Long policyId);
    // Generated: SELECT * FROM claims WHERE policy_id = ?

    // ═══════════════════════════════════════════════════════════════════
    // CUSTOM QUERIES (JPQL - Java Persistence Query Language)
    // When naming convention isn't enough
    // ═══════════════════════════════════════════════════════════════════

    @Query("SELECT c FROM Claim c WHERE c.claimStatus = 'PENDING' OR c.claimStatus
```

```
= 'UNDER_REVIEW'")
    List<Claim> findPendingClaims();
    // Uses entity/field names, not table/column names

    @Query("SELECT c FROM Claim c WHERE c.claimant.userId = :userId ORDER BY
c.createdDate DESC")
    List<Claim> findClaimsByClaimantOrderByDate(@Param("userId") Long userId);
    // :userId is a named parameter, bound via @Param

    @Query("SELECT c FROM Claim c WHERE c.adjuster IS NULL AND c.claimStatus =
'PENDING'")
    List<Claim> findUnassignedClaims();

    @Query("SELECT COUNT(c) FROM Claim c WHERE c.claimStatus = :status")
    long countByClaimStatus(@Param("status") ClaimStatus status);
}
```

Query Method Naming Rules

Keyword	Sample	Generated SQL
findBy	findByClaimStatus(status)	WHERE claim_status = ?
And	findByStatusAndAmount(s, a)	WHERE status=? AND amount=?
Or	findByStatusOrAmount(s, a)	WHERE status=? OR amount=?
Between	findByAmountBetween(a, b)	WHERE amount BETWEEN ? AND ?
LessThan	findByAmountLessThan(a)	WHERE amount < ?
GreaterThan	findByAmountGreaterThan(a)	WHERE amount > ?
Like	findByDescriptionLike(d)	WHERE description LIKE ?
Containing	findByDescriptionContaining(d)	WHERE description LIKE '%?%'
In	findByStatusIn(list)	WHERE status IN (?, ?, ...)
OrderBy	findByStatusOrderByDateDesc()	ORDER BY date DESC
IsNull	findByAdjusterIsNull()	WHERE adjuster_id IS NULL
IsNotNull	findByAdjusterIsNotNull()	WHERE adjuster_id IS NOT NULL

3.3 SERVICE LAYER (ClaimService & ClaimServiceImpl)

What is a Service?

The Service layer contains **business logic**. It validates data, applies rules, and coordinates database operations.

Interface (Contract)

```
package com.hicms.service;

public interface ClaimService {
    // CREATE
    Claim submitClaim(ClaimDTO claimDTO, User claimant);
    Claim submitClaimWithAgent(ClaimDTO claimDTO, User claimant, User agent);

    // READ
    Optional<Claim> findById(Long claimId);
    List<Claim> findAllClaims();
    List<Claim> findClaimsByClaimant(Long claimantId);

    // UPDATE
    Claim assignAdjuster(Long claimId, User adjuster);
    Claim reviewClaim(Long claimId, ClaimReviewDTO reviewDTO, User adjuster);

    // CONVERSION
    ClaimDTO convertToDTO(Claim claim);
}
```

Implementation (ClaimServiceImpl.java)

```
package com.hicms.service.impl;

@Service                          // ← Marks as Spring service bean
@RequiredArgsConstructor          // ← Lombok: generates constructor for final
fields
@Transactional                    // ← All methods run in database transaction
public class ClaimServiceImpl implements ClaimService {

    // ═══════════════════════════════════════════════════════════════
    // DEPENDENCY INJECTION (via constructor - best practice)
    // ═══════════════════════════════════════════════════════════════
    private final ClaimRepository claimRepository;
    private final PolicyRepository policyRepository;
    private final PolicyEnrollmentRepository enrollmentRepository;
    // Lombok's @RequiredArgsConstructor generates:
    // public ClaimServiceImpl(ClaimRepository cr, PolicyRepository pr, ...) {
    //     this.claimRepository = cr;
    //     this.policyRepository = pr;
    //     ...
    // }

    // ═══════════════════════════════════════════════════════════════
    // SUBMIT CLAIM - Main business logic
    // ═══════════════════════════════════════════════════════════════
    @Override
    public Claim submitClaimWithAgent(ClaimDTO claimDTO, User claimant, User
agent) {
```

```
        // STEP 1: Retrieve the policy
        Policy policy = policyRepository.findById(claimDTO.getPolicyId())
                .orElseThrow(() -> new RuntimeException("Policy not found"));

        // STEP 2: BUSINESS RULE - Verify enrollment
        boolean isEnrolled = enrollmentRepository
                .existsByPolicyholderUserIdAndPolicyPolicyIdAndEnrollmentStatusIn(
                        claimant.getUserId(),
                        claimDTO.getPolicyId(),
                        Arrays.asList(PolicyEnrollment.EnrollmentStatus.ACTIVE)
                );

        if (!isEnrolled) {
            throw new RuntimeException("User is not enrolled in this policy");
        }

        // STEP 3: BUSINESS RULE - Validate claim amount
        if (claimDTO.getClaimAmount().compareTo(policy.getCoverageAmount()) > 0) {
            throw new RuntimeException("Claim amount exceeds coverage amount");
        }

        // STEP 4: Build entity using Builder pattern
        Claim claim = Claim.builder()
                .claimNumber(generateClaimNumber())    // Generate unique number
                .policy(policy)
                .claimant(claimant)
                .agent(agent)                          // May be null
                .claimAmount(claimDTO.getClaimAmount())
                .claimDate(LocalDate.now())
                .description(claimDTO.getDescription())
                .reason(claimDTO.getReason())
                .claimStatus(ClaimStatus.PENDING)      // Initial status
                .build();

        // STEP 5: Persist to database
        return claimRepository.save(claim);
        // Hibernate: INSERT INTO claims (...) VALUES (...)
    }

    // ═══════════════════════════════════════════════════════════════════
    // READ OPERATIONS - @Transactional(readOnly = true) for optimization
    // ═══════════════════════════════════════════════════════════════════
    @Override
    @Transactional(readOnly = true)    // ← Hibernate optimizations for reads
    public Optional<Claim> findById(Long claimId) {
        return claimRepository.findById(claimId);
        // Hibernate: SELECT * FROM claims WHERE claim_id = ?
    }

    @Override
    @Transactional(readOnly = true)
    public List<Claim> findAllClaims() {
        return claimRepository.findAll();
        // Hibernate: SELECT * FROM claims
```

```
    }

    // ========================================================================
    // REVIEW CLAIM - Update with decision
    // ========================================================================
    @Override
    public Claim reviewClaim(Long claimId, ClaimReviewDTO reviewDTO, User
adjuster) {
        // Find existing claim
        Claim claim = claimRepository.findById(claimId)
                .orElseThrow(() -> new RuntimeException("Claim not found"));

        // Update fields
        claim.setAdjuster(adjuster);
        claim.setClaimStatus(reviewDTO.getClaimStatus());
        claim.setApprovedAmount(reviewDTO.getApprovedAmount());
        claim.setRemarks(reviewDTO.getRemarks());

        // Save updates
        return claimRepository.save(claim);
        // Hibernate: UPDATE claims SET adjuster_id=?, claim_status=?, ... WHERE
claim_id=?
    }

    // ========================================================================
    // ENTITY TO DTO CONVERSION
    // ========================================================================
    @Override
    public ClaimDTO convertToDTO(Claim claim) {
        return ClaimDTO.builder()
                .claimId(claim.getClaimId())
                .claimNumber(claim.getClaimNumber())
                .policyId(claim.getPolicy().getPolicyId())
                .policyName(claim.getPolicy().getPolicyName())
                .claimantId(claim.getClaimant().getUserId())
                .claimantName(claim.getClaimant().getFullName())
                // Handle nullable relationships
                .adjusterId(claim.getAdjuster() != null ?
                        claim.getAdjuster().getUserId() : null)
                .adjusterName(claim.getAdjuster() != null ?
                        claim.getAdjuster().getFullName() : null)
                .claimAmount(claim.getClaimAmount())
                .approvedAmount(claim.getApprovedAmount())
                .claimStatus(claim.getClaimStatus())
                .build();
    }

    // ========================================================================
    // PRIVATE HELPER - Generate unique claim number
    // ========================================================================
    private String generateClaimNumber() {
        // Format: CLM-YYYYMMDD-XXX (e.g., CLM-20260131-001)
        String datePrefix = LocalDate.now()
                .format(DateTimeFormatter.ofPattern("yyyyMMdd"));
```

```
        String baseNumber = "CLM-" + datePrefix + "-";

        int counter = 1;
        while (claimRepository.existsByClaimNumber(
                baseNumber + String.format("%03d", counter))) {
            counter++;
        }

        return baseNumber + String.format("%03d", counter);
    }
}
```

3.4 CONTROLLER LAYER (ClaimController.java)

What is a Controller?

The Controller handles **HTTP requests**. It receives input, calls services, and returns responses (views or data).

Complete Code with Explanations

```
package com.hicms.controller;

@Controller                          // ← Marks as Spring MVC controller (returns
views)
@RequestMapping("/claims")           // ← Base URL: all endpoints start with /claims
@RequiredArgsConstructor             // ← Constructor injection for dependencies
public class ClaimController {

    // ═══════════════════════════════════════════════════════════════════
    // DEPENDENCY INJECTION
    // ═══════════════════════════════════════════════════════════════════
    private final ClaimService claimService;
    private final PolicyEnrollmentService enrollmentService;
    private final UserService userService;
    private final DocumentService documentService;

    // ═══════════════════════════════════════════════════════════════════
    // LIST CLAIMS - GET /claims
    // ═══════════════════════════════════════════════════════════════════
    @GetMapping                      // ← Handles GET /claims
    public String listClaims(
            @AuthenticationPrincipal UserDetails userDetails,  // ← Current
logged-in user
            Model model) {                                     // ← Data carrier
to view

        // Get full user entity from username
        User user = userService.findByUsername(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException("User not found"));
```

```
        List<Claim> claims;

        // ROLE-BASED FILTERING
        switch (user.getRole()) {
            case ADMIN:
                claims = claimService.findAllClaims();          // See all
                break;
            case AGENT:
                claims = claimService.findClaimsByAgent(user.getUserId());
                break;
            case CLAIM_ADJUSTER:
                claims = claimService.findClaimsByAdjuster(user.getUserId());
                break;
            default:  // USER
                claims = claimService.findClaimsByClaimant(user.getUserId());
        }

        // Add data to model (available in Thymeleaf template)
        model.addAttribute("claims", claimService.convertToDTOList(claims));
        model.addAttribute("userRole", user.getRole());

        return "claim/list";      // ← Returns templates/claim/list.html
    }

    // ═══════════════════════════════════════════════════════════════
    // SHOW SUBMIT FORM - GET /claims/submit
    // ═══════════════════════════════════════════════════════════════
    @GetMapping("/submit")
    @PreAuthorize("hasAnyRole('USER', 'AGENT')")  // ← Security: only USER or
AGENT
    public String submitClaimForm(
            @AuthenticationPrincipal UserDetails userDetails,
            Model model) {

        User user = userService.findByUsername(userDetails.getUsername())
                .orElseThrow();

        model.addAttribute("claim", new ClaimDTO());  // Empty form object
        model.addAttribute("userRole", user.getRole());

        if (user.getRole() == Role.AGENT) {
            // Agent can file for any customer
            model.addAttribute("customers",
userService.findUsersByRole(Role.USER));
        } else {
            // User sees only their active policy enrollments
            List<PolicyEnrollment> activeEnrollments =

enrollmentService.findActiveEnrollmentsByUser(user.getUserId());
            model.addAttribute("enrollments",
                    enrollmentService.convertToDTOList(activeEnrollments));
        }

        return "claim/submit";    // ← Returns templates/claim/submit.html
```

```
    }

    // ═══════════════════════════════════════════════════════════════════
    // PROCESS SUBMIT - POST /claims/submit
    // ═══════════════════════════════════════════════════════════════════
    @PostMapping("/submit")
    @PreAuthorize("hasAnyRole('USER', 'AGENT')")
    public String submitClaim(
            @Valid @ModelAttribute("claim") ClaimDTO claimDTO,  // ← Form data + validation
            BindingResult result,                               // ← Validation errors
            @RequestParam(required = false) Long customerId,    // ← Query param (for agents)
            @RequestParam(value = "documents", required = false)
                    List<MultipartFile> documents,              // ← File uploads
            @AuthenticationPrincipal UserDetails userDetails,
            RedirectAttributes redirectAttributes,              // ← Flash messages
            Model model) {

        User currentUser = userService.findByUsername(userDetails.getUsername())
                .orElseThrow();

        // CHECK VALIDATION ERRORS
        if (result.hasErrors()) {
            // Return to form with error messages
            model.addAttribute("userRole", currentUser.getRole());
            // ... repopulate dropdowns ...
            return "claim/submit";
        }

        try {
            Claim savedClaim;

            if (currentUser.getRole() == Role.AGENT && customerId != null) {
                // Agent submitting for customer
                User customer = userService.findById(customerId).orElseThrow();
                savedClaim = claimService.submitClaimWithAgent(claimDTO, customer, currentUser);
            } else {
                // User submitting for themselves
                savedClaim = claimService.submitClaim(claimDTO, currentUser);
            }

            // HANDLE FILE UPLOADS
            if (documents != null && !documents.isEmpty()) {
                for (MultipartFile file : documents) {
                    if (!file.isEmpty()) {
                        documentService.uploadDocument(
                                savedClaim.getClaimId(), file, currentUser);
                    }
                }
            }
```

```
            // SUCCESS - redirect with message
            redirectAttributes.addFlashAttribute("successMessage",
                    "Claim submitted successfully!");
            return "redirect:/claims";  // ← PRG pattern (Post-Redirect-Get)

        } catch (Exception e) {
            // ERROR - stay on form
            model.addAttribute("errorMessage", e.getMessage());
            return "claim/submit";
        }
    }

    // ═══════════════════════════════════════════════════════════════
    // REVIEW CLAIM - POST /claims/review/{id}
    // ═══════════════════════════════════════════════════════════════
    @PostMapping("/review/{id}")
    @PreAuthorize("hasAnyRole('CLAIM_ADJUSTER', 'ADMIN')")  // ← Only
adjusters/admins
    public String reviewClaim(
            @PathVariable Long id,                          // ← From URL:
/review/5
            @ModelAttribute("review") ClaimReviewDTO reviewDTO,
            @AuthenticationPrincipal UserDetails userDetails,
            RedirectAttributes redirectAttributes) {
        try {
            User adjuster = userService.findByUsername(userDetails.getUsername())
                    .orElseThrow();

            claimService.reviewClaim(id, reviewDTO, adjuster);

            redirectAttributes.addFlashAttribute("successMessage",
                    "Claim reviewed successfully!");
            return "redirect:/claims";

        } catch (Exception e) {
            redirectAttributes.addFlashAttribute("errorMessage", e.getMessage());
            return "redirect:/claims/review/" + id;
        }
    }
}
```

4. ANNOTATIONS REFERENCE GUIDE

4.1 JPA/Hibernate Annotations (Entity Layer)

Annotation	Location	Purpose	Example
`@Entity`	Class	Marks class as database table	`@Entity public class Claim`

Annotation	Location	Purpose	Example
`@Table`	Class	Specifies table name	`@Table(name = "claims")`
`@Id`	Field	Marks primary key	`@Id private Long claimId`
`@GeneratedValue`	Field	Auto-generate PK	`@GeneratedValue(strategy = GenerationType.IDENTITY)`
`@Column`	Field	Column customization	`@Column(nullable = false, length = 50)`
`@ManyToOne`	Field	Many-to-one relationship	`@ManyToOne private Policy policy`
`@OneToMany`	Field	One-to-many relationship	`@OneToMany(mappedBy = "claim")`
`@JoinColumn`	Field	Foreign key column	`@JoinColumn(name = "policy_id")`
`@Enumerated`	Field	Store enum as string/ordinal	`@Enumerated(EnumType.STRING)`
`@PrePersist`	Method	Before INSERT callback	Auto-set createdDate
`@PreUpdate`	Method	Before UPDATE callback	Auto-update updatedDate

4.2 Spring Annotations

Annotation	Layer	Purpose
`@Repository`	Repository	Data access component, enables exception translation
`@Service`	Service	Business logic component
`@Controller`	Controller	Web MVC controller (returns views)
`@RestController`	Controller	REST API controller (returns JSON)
`@Component`	Any	Generic Spring-managed bean
`@Autowired`	Field/Constructor	Dependency injection
`@RequiredArgsConstructor`	Class	Lombok: generates constructor for final fields
`@Transactional`	Class/Method	Database transaction management

4.3 Web/MVC Annotations

Annotation	Purpose	Example
`@RequestMapping`	Base URL mapping	`@RequestMapping("/claims")`

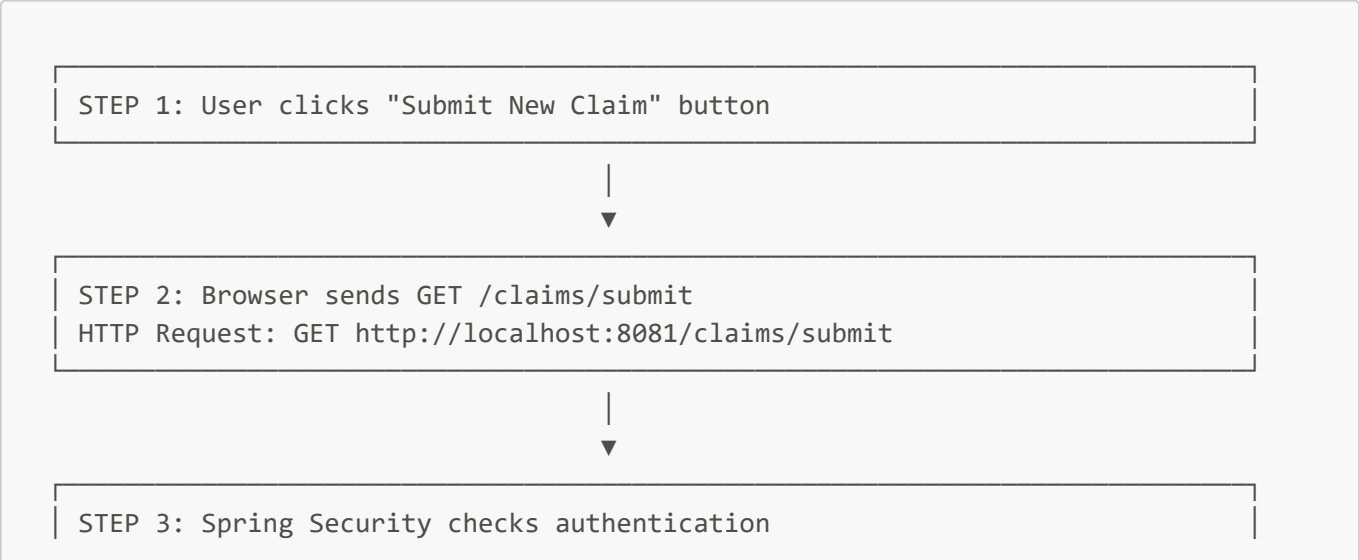
Annotation	Purpose	Example
`@GetMapping`	Handle GET requests	`@GetMapping("/view/{id}")`
`@PostMapping`	Handle POST requests	`@PostMapping("/submit")`
`@PathVariable`	URL path parameter	`/claims/view/{id}` → `id`
`@RequestParam`	Query/form parameter	`?customerId=5` → `customerId`
`@ModelAttribute`	Bind form to object	HTML form → ClaimDTO
`@Valid`	Trigger validation	Check @NotNull, @Size, etc.
`@AuthenticationPrincipal`	Get logged-in user	Current user details
`@PreAuthorize`	Method-level security	`hasRole('ADMIN')`

4.4 Validation Annotations (DTO Layer)

Annotation	Purpose	Example
`@NotNull`	Cannot be null	`@NotNull private Long policyId`
`@NotBlank`	Not null/empty/whitespace	`@NotBlank private String description`
`@Size`	String length limits	`@Size(max = 1000)`
`@DecimalMin`	Minimum numeric value	`@DecimalMin("0.01")`
`@DecimalMax`	Maximum numeric value	`@DecimalMax("999999.99")`
`@Email`	Valid email format	`@Email private String email`
`@Pattern`	Regex pattern	`@Pattern(regexp = "...")`

5. COMPLETE CODE FLOW WALKTHROUGH

5.1 Flow: User Submits a New Claim



```
| - Is user logged in? → YES (has session cookie)                                  |
| - @PreAuthorize("hasAnyRole('USER', 'AGENT')") → User has USER role → OK   |
└──────────────────────────────────────────────────────────────────────────┘
                                    |
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
| STEP 4: ClaimController.submitClaimForm() executes                       |
|                                                                          |
| @GetMapping("/submit")                                                   |
| public String submitClaimForm(@AuthenticationPrincipal UserDetails user,   |
|                               Model model) {                             |
|     User user = userService.findByUsername(userDetails.getUsername());     |
|     model.addAttribute("claim", new ClaimDTO());  // Empty form object     |
|     model.addAttribute("enrollments", enrollmentService.find...(userId));  |
|     return "claim/submit";  // ← Returns view name                       |
| }                                                                        |
└──────────────────────────────────────────────────────────────────────────┘
                                    |
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
| STEP 5: Thymeleaf renders templates/claim/submit.html                    |
| - Populates dropdown with user's active enrollments                      |
| - Creates empty form bound to ClaimDTO                                   |
| - Browser displays the form to user                                      |
└──────────────────────────────────────────────────────────────────────────┘
                                    |
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
| STEP 6: User fills form and clicks Submit                                |
| - Selects policy: Health Plus (policyId: 1)                              |
| - Amount: $2,500.00                                                      |
| - Description: "Hospital visit for surgery"                              |
| - Uploads: medical_report.pdf                                            |
└──────────────────────────────────────────────────────────────────────────┘
                                    |
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
| STEP 7: Browser sends POST /claims/submit                                |
| HTTP Request: POST http://localhost:8081/claims/submit                   |
| Content-Type: multipart/form-data                                        |
| Body: policyId=1, claimAmount=2500.00, description=..., documents=file     |
└──────────────────────────────────────────────────────────────────────────┘
                                    |
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
| STEP 8: Spring Security validates (same as step 3)                       |
└──────────────────────────────────────────────────────────────────────────┘
                                    |
                                    ▼
┌──────────────────────────────────────────────────────────────────────────┐
| STEP 9: Spring MVC binds form data to ClaimDTO                           |
| @ModelAttribute("claim") ClaimDTO claimDTO                               |
|                                                                          |
| ClaimDTO {                                                                |
```

```
    policyId: 1,
    claimAmount: 2500.00,
    description: "Hospital visit for surgery"
}
```

▼

```
STEP 10: @Valid triggers Bean Validation
- @NotNull policyId → OK (has value)
- @DecimalMin("0.01") claimAmount → OK (2500 > 0.01)
- @NotBlank description → OK (has text)
- BindingResult.hasErrors() → false
```

▼

```
STEP 11: ClaimController calls ClaimService

savedClaim = claimService.submitClaim(claimDTO, currentUser);
```

▼

```
STEP 12: ClaimServiceImpl.submitClaim() executes

12a. Fetch Policy:
     policyRepository.findById(1) → Policy{id:1, name:"Health Plus",...}
     Hibernate SQL: SELECT * FROM policies WHERE policy_id = 1

12b. Validate Enrollment:
     enrollmentRepository.existsByPolicyholderUserIdAndPolicy...
     Hibernate SQL: SELECT COUNT(*) FROM policy_enrollments
                    WHERE policyholder_id=5 AND policy_id=1
                    AND enrollment_status='ACTIVE'
     Result: true (user is enrolled)

12c. Validate Amount:
     2500 <= 100000 (coverage amount) → OK

12d. Generate Claim Number:
     "CLM-20260131-001" (checks for uniqueness)

12e. Build Claim Entity:
     Claim claim = Claim.builder()
         .claimNumber("CLM-20260131-001")
         .policy(policy)
         .claimant(currentUser)
         .claimAmount(2500.00)
         .claimStatus(PENDING)
         .build();
```

▼

```
┌─────────────────────────────────────────────────────────────────────┐
│ STEP 13: ClaimRepository.save(claim) executes                       │
│                                                                     │
│ Before save:                                                        │
│ - @PrePersist callback triggers                                     │
│ - Sets createdDate = 2026-01-31T10:30:00                            │
│ - Sets updatedDate = 2026-01-31T10:30:00                            │
│                                                                     │
│ Hibernate generates SQL:                                            │
│ INSERT INTO claims (claim_number, policy_id, claimant_id, claim_amount,    │
│                     claim_date, description, claim_status, created_date,   │
│                     updated_date)                                   │
│ VALUES ('CLM-20260131-001', 1, 5, 2500.00, '2026-01-31',            │
│         'Hospital visit...', 'PENDING', '2026-01-31 10:30:00', ...)        │
│                                                                     │
│ MySQL auto-generates claim_id: 10                                   │
│ Hibernate sets claim.claimId = 10                                   │
└─────────────────────────────────────────────────────────────────────┘
                                  │
                                  ▼
┌─────────────────────────────────────────────────────────────────────┐
│ STEP 14: Controller handles document upload                         │
│                                                                     │
│ for (MultipartFile file : documents) {                              │
│     documentService.uploadDocument(10, file, currentUser);          │
│ }                                                                   │
│ - Saves file to: ./uploads/documents/10/medical_report.pdf          │
│ - Creates Document entity with claimId=10                           │
└─────────────────────────────────────────────────────────────────────┘
                                  │
                                  ▼
┌─────────────────────────────────────────────────────────────────────┐
│ STEP 15: Controller returns redirect                                │
│                                                                     │
│ redirectAttributes.addFlashAttribute("successMessage", "Claim submitted!");  │
│ return "redirect:/claims";                                          │
│                                                                     │
│ HTTP Response: 302 Redirect to /claims                              │
│ Browser automatically sends GET /claims                             │
└─────────────────────────────────────────────────────────────────────┘
                                  │
                                  ▼
┌─────────────────────────────────────────────────────────────────────┐
│ STEP 16: User sees claim list with success message                  │
│                                                                     │
│ "Claim submitted successfully!"                                     │
│ New claim CLM-20260131-001 appears in the list                      │
└─────────────────────────────────────────────────────────────────────┘
```

6. DATABASE & HIBERNATE INTEGRATION

6.1 Connection Configuration (application.properties)

```
# Database Connection
spring.datasource.url=jdbc:mysql://localhost:3306/hicms_db?
createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Hibernate/JPA Configuration
spring.jpa.hibernate.ddl-auto=update        # Auto-create/update tables
spring.jpa.show-sql=true                     # Log SQL to console
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

DDL-Auto Options

Value	Behavior
`none`	Do nothing
`validate`	Validate schema, don't change
`update`	Update schema to match entities
`create`	Drop and recreate on startup
`create-drop`	Create on start, drop on shutdown

6.2 How Hibernate Translates Operations

save() Operation

```
// Java code
Claim claim = Claim.builder()
        .claimNumber("CLM-20260131-001")
        .policy(policy)
        .claimant(user)
        .claimAmount(new BigDecimal("2500.00"))
        .claimStatus(ClaimStatus.PENDING)
        .build();

claimRepository.save(claim);
```

```
-- Hibernate generates (for new entity with null ID):
INSERT INTO claims
    (claim_number, policy_id, claimant_id, claim_amount, claim_date,
     description, claim_status, created_date, updated_date)
```

```
VALUES
    ('CLM-20260131-001', 1, 5, 2500.00, '2026-01-31',
     'Hospital visit...', 'PENDING', '2026-01-31 10:30:00', '2026-01-31
10:30:00');
```

findById() Operation

```
// Java code
Optional<Claim> claim = claimRepository.findById(10L);
```

```
-- Hibernate generates:
SELECT c.claim_id, c.claim_number, c.policy_id, c.claimant_id,
       c.claim_amount, c.approved_amount, c.claim_date, c.description,
       c.claim_status, c.remarks, c.created_date, c.updated_date
FROM claims c
WHERE c.claim_id = 10;
```

Lazy Loading (Relationship)

```
// Java code
Claim claim = claimRepository.findById(10L).orElseThrow();
String policyName = claim.getPolicy().getPolicyName();  // ← Triggers lazy load
```

```
-- First query (findById):
SELECT * FROM claims WHERE claim_id = 10;

-- Second query (when accessing policy - LAZY loading):
SELECT * FROM policies WHERE policy_id = 1;
```

Custom Query Method

```
// Java code
List<Claim> claims = claimRepository.findByClaimantUserId(5L);
```

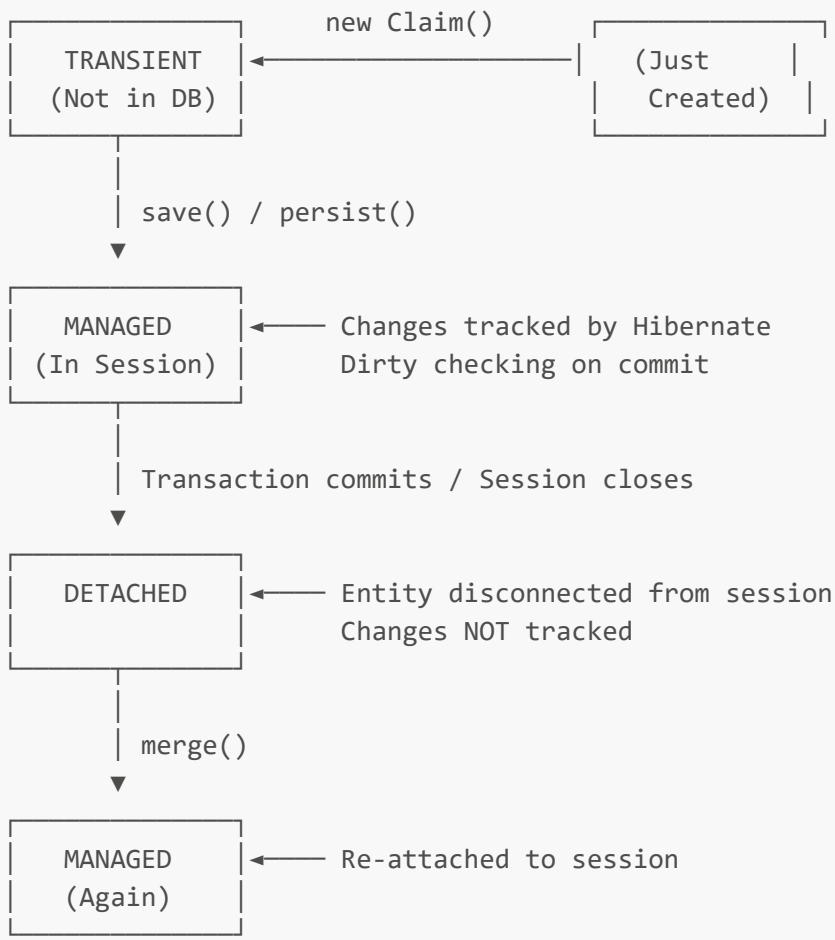
```
-- Hibernate generates from method name:
SELECT c.* FROM claims c
WHERE c.claimant_id = 5;
```

JPQL Query

```
// Java code
@Query("SELECT c FROM Claim c WHERE c.adjuster IS NULL AND c.claimStatus =
'PENDING'")
List<Claim> findUnassignedClaims();
```

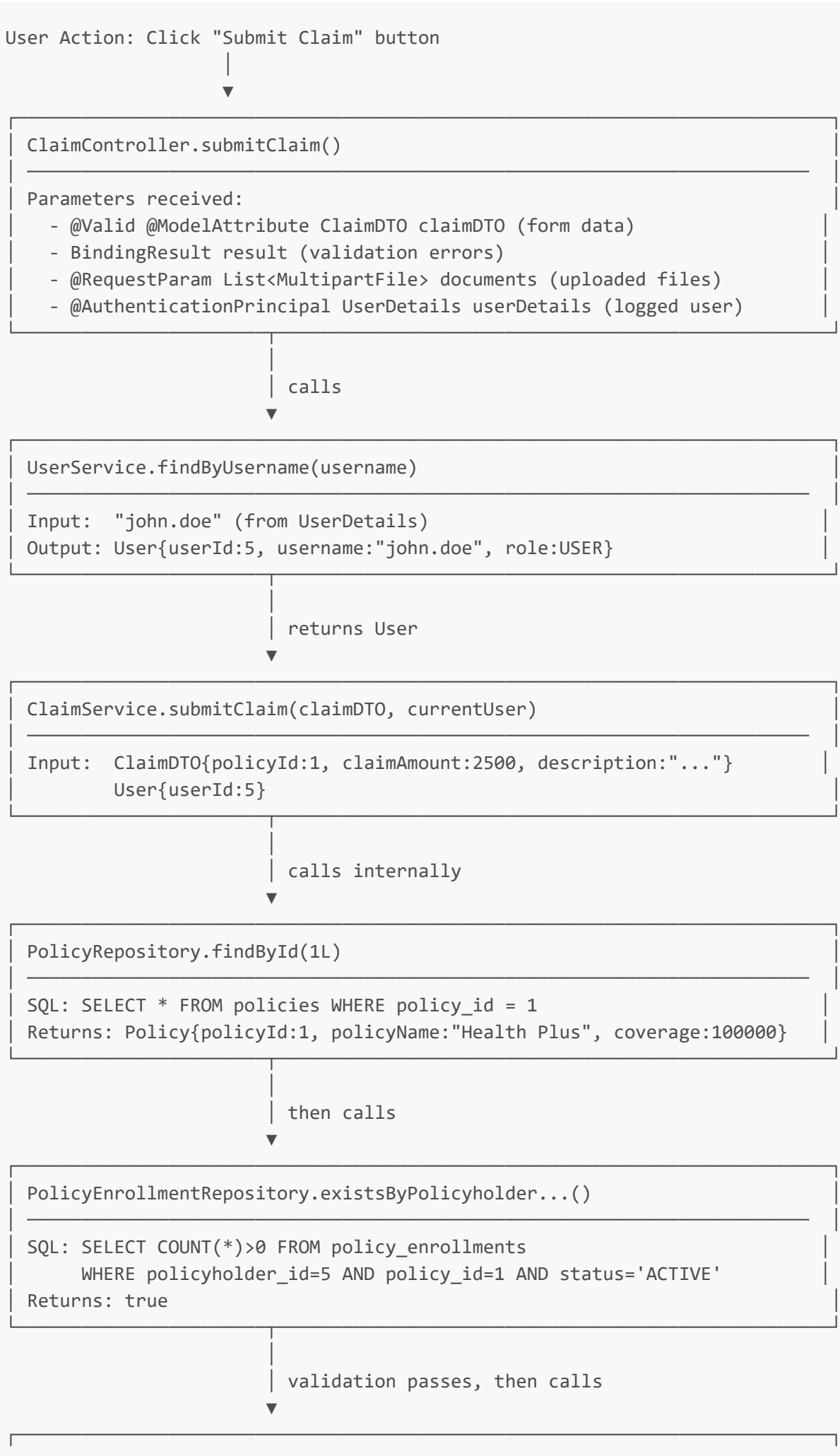
```
-- Hibernate translates JPQL to SQL:
SELECT c.* FROM claims c
WHERE c.adjuster_id IS NULL
  AND c.claim_status = 'PENDING';
```

6.3 Entity Lifecycle States



7. METHOD CONNECTIVITY & DATA FLOW

7.1 Complete Method Call Chain


User Action: Click "Submit Claim" button
ClaimController.submitClaim()
Parameters received:
- @Valid @ModelAttribute ClaimDTO claimDTO (form data)
- BindingResult result (validation errors)
- @RequestParam List<MultipartFile> documents (uploaded files)
- @AuthenticationPrincipal UserDetails userDetails (logged user)
calls
UserService.findByUsername(username)
Input: "john.doe" (from UserDetails)
Output: User{userId:5, username:"john.doe", role:USER}
returns User
ClaimService.submitClaim(claimDTO, currentUser)
Input: ClaimDTO{policyId:1, claimAmount:2500, description:"..."}
User{userId:5}
calls internally
PolicyRepository.findById(1L)
SQL: SELECT * FROM policies WHERE policy_id = 1
Returns: Policy{policyId:1, policyName:"Health Plus", coverage:100000}
then calls
PolicyEnrollmentRepository.existsByPolicyholder...()
SQL: SELECT COUNT(*)>0 FROM policy_enrollments
WHERE policyholder_id=5 AND policy_id=1 AND status='ACTIVE'
Returns: true
validation passes, then calls

```
│ ClaimRepository.save(claim)                                                  │
│ ──────────────────────────────────────────────────────────────────────────── │
│ Input: Claim{claimNumber:"CLM-20260131-001", policyId:1, claimantId:5,       │
│              claimAmount:2500, status:PENDING}                               │
│                                                                              │
│ Hibernate triggers @PrePersist → sets createdDate, updatedDate              │
│                                                                              │
│ SQL: INSERT INTO claims (...) VALUES (...)                                   │
│ Returns: Claim{claimId:10, ...} (with generated ID)                          │
└──────────────────────────────────────────────────────────────────────────────┘
                          │
                          │
                          │ returns saved Claim
                          ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│ DocumentService.uploadDocument(10, file, currentUser)                        │
│ ──────────────────────────────────────────────────────────────────────────── │
│ - Saves file to: ./uploads/documents/10/medical_report.pdf                   │
│ - Creates Document entity                                                    │
│ - documentRepository.save(document)                                          │
│ SQL: INSERT INTO documents (...) VALUES (...)                                │
└──────────────────────────────────────────────────────────────────────────────┘
                          │
                          │
                          │ returns to Controller
                          ▼
┌──────────────────────────────────────────────────────────────────────────────┐
│ ClaimController returns "redirect:/claims"                                   │
│ ──────────────────────────────────────────────────────────────────────────── │
│ - Adds flash message: "Claim submitted successfully!"                        │
│ - HTTP 302 Redirect                                                          │
│ - Browser follows redirect to GET /claims                                    │
└──────────────────────────────────────────────────────────────────────────────┘
```

7.2 Data Transformation Through Layers

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                          DATA FLOW DIAGRAM                                   │
└──────────────────────────────────────────────────────────────────────────────┘


BROWSER → CONTROLLER
════════════════════

HTML Form Data (key=value pairs)
    │
    │ Spring MVC @ModelAttribute binding
    ▼
ClaimDTO (Java object)
┌──────────────────────────┐
│ policyId: 1              │
│ claimAmount: 2500.00     │
│ description: "Hospital..." │
│ reason: null             │
└──────────────────────────┘
```

```
CONTROLLER → SERVICE
═══════════════════

ClaimDTO + User objects passed as method parameters

SERVICE → REPOSITORY
═══════════════════

Service builds Claim entity from DTO
┌──────────────────────────────┐
│ Claim (Entity)               │
├──────────────────────────────┤
│ claimId: null (not saved)    │
│ claimNumber: "CLM-2026..."   │
│ policy: Policy{id:1}         │  ← Relationship object
│ claimant: User{id:5}         │  ← Relationship object
│ claimAmount: 2500.00         │
│ claimStatus: PENDING         │
│ createdDate: null            │
└──────────────────────────────┘

REPOSITORY → DATABASE
═══════════════════

Hibernate converts Entity to SQL

INSERT INTO claims (
    claim_number,      ← from claimNumber field
    policy_id,         ← from policy.policyId (FK)
    claimant_id,       ← from claimant.userId (FK)
    claim_amount,      ← from claimAmount field
    claim_status,      ← from claimStatus enum (as STRING)
    created_date       ← from @PrePersist callback
) VALUES (
    'CLM-20260131-001', 1, 5, 2500.00, 'PENDING', '2026-01-31 10:30:00'
)

DATABASE → REPOSITORY (Return)
═══════════════════════════

MySQL returns auto-generated ID
Hibernate populates claimId: 10

SERVICE → CONTROLLER (Return)
═══════════════════════════

Returns Claim entity with generated ID

CONTROLLER → VIEW (For display)
═══════════════════════════

Service converts Entity to DTO for view
┌──────────────────────────────┐
│ ClaimDTO                     │
├──────────────────────────────┤
│ claimId: 10                  │
│ claimNumber: "CLM-2026..."   │
│ policyName: "Health Plus"    │  ← Flattened from relationship
│ claimantName: "John Doe"     │  ← Flattened from relationship
```

```
  │ claimAmount: 2500.00        │
  │ claimStatus: PENDING        │
  └─────────────────────────────┘

  VIEW → BROWSER
  ──────────────
  Thymeleaf renders HTML with DTO data
```

---

8. INTERVIEW-READY EXPLANATIONS

Q1: "Explain the overall architecture of the Claim module."

Answer: "The Claim module follows a layered architecture with clear separation of concerns:

1. **Entity Layer** - `Claim.java` maps to the database table using JPA annotations. It defines the data structure and relationships.
2. **Repository Layer** - `ClaimRepository` extends `JpaRepository` for CRUD operations. Spring Data JPA auto-implements methods based on naming conventions.
3. **Service Layer** - `ClaimService` interface with `ClaimServiceImpl` contains business logic like enrollment validation, amount checking, and claim number generation.
4. **Controller Layer** - `ClaimController` handles HTTP requests, binds form data, and returns views.
5. **DTO Layer** - `ClaimDTO` transfers data between controller and view, with validation annotations.

Data flows: Browser → Controller → Service → Repository → Database, and back."

Q2: "How does @Transactional work in the service layer?"

Answer: "The `@Transactional` annotation on `ClaimServiceImpl` means:

1. When any method is called, Spring starts a database transaction.
2. All database operations within that method share the same transaction.
3. If the method completes successfully, the transaction is committed.
4. If any exception occurs, the transaction is rolled back - all changes are undone.

For example, if `submitClaim()` saves a claim but document upload fails, without `@Transactional`, the claim would remain. With it, everything rolls back, keeping data consistent.

I also use `@Transactional(readOnly = true)` on read methods for performance optimization - Hibernate skips dirty checking."

Q3: "How does Hibernate map entity to database table?"

Answer: "Hibernate uses annotations to map Java classes to database tables:

- `@Entity` marks the class as a persistent entity
- `@Table(name = 'claims')` specifies the table name
- `@Id` marks the primary key field
- `@GeneratedValue(strategy = IDENTITY)` tells MySQL to auto-increment
- `@Column(nullable = false, length = 50)` defines column constraints
- `@ManyToOne` defines foreign key relationships

When I call `claimRepository.save(claim)`, Hibernate:

1. Checks if entity is new (claimId is null) → generates INSERT
2. Or if existing → generates UPDATE
3. Executes @PrePersist callback to set timestamps
4. Executes the SQL against MySQL
5. Sets the generated ID back on the entity

With `show-sql=true`, I can see the exact SQL being executed."

Q4: "Explain the difference between Entity and DTO."

Answer: "Entity and DTO serve different purposes:

Entity (Claim.java):

- Represents database table structure
- Contains JPA annotations for ORM
- Has relationships (Policy, User) as object references
- Managed by Hibernate persistence context
- Should not be exposed directly to views (security, lazy loading issues)

DTO (ClaimDTO.java):

- Data transfer between layers
- Contains validation annotations (@NotNull, @Size)
- Flattens relationships (policyName instead of Policy object)
- Safe to pass to views
- No database coupling

I convert Entity → DTO in the service layer using `convertToDTO()` method. This protects internal structure and handles lazy loading within the transaction."

Q5: "How does Spring Data JPA generate queries from method names?"

Answer: "Spring Data JPA has a query derivation mechanism. It parses method names and generates SQL:

```
findByClaimantUserId(Long userId)
```

Spring parses this as:

- `find` → SELECT query
- `By` → WHERE clause starts
- `Claimant` → Navigate to claimant relationship
- `UserId` → Access userId field

Generated SQL:

```
SELECT * FROM claims WHERE claimant_id = ?
```

Other examples:

- `findByClaimStatusIn(List)` → `WHERE status IN (?,...)`
- `existsByClaimNumber(String)` → `SELECT COUNT(*) > 0 WHERE claim_number = ?`
- `countByClaimStatus(Status)` → `SELECT COUNT(*) WHERE status = ?`

For complex queries, I use `@Query` with JPQL, which uses entity names instead of table names."

Q6: "How does the claim submission flow work end-to-end?"

Answer: "When a user submits a claim:

1. **Browser** sends GET `/claims/submit` → Controller returns the form with user's active enrollments
2. User fills form and clicks Submit → Browser sends POST `/claims/submit` with form data
3. **Spring Security** checks authentication and `@PreAuthorize` role
4. **Controller** receives request:
    - `@ModelAttribute` binds form data to ClaimDTO
    - `@Valid` triggers validation (checks @NotNull, @DecimalMin)
    - If validation fails, return form with errors
5. **Service** validates business rules:
    - Check user is enrolled in selected policy
    - Check amount doesn't exceed coverage
    - Generate unique claim number
6. **Repository.save()** triggers:
    - `@PrePersist` sets timestamps
    - Hibernate generates INSERT SQL
    - MySQL executes and returns generated ID
7. **Controller** handles file uploads via DocumentService
8. **Redirect** to `/claims` with success flash message

The entire service method runs in a single transaction, ensuring atomicity."

End of Technical Documentation