Collecting workspace informationDựa trên nghiên cứu dự án của bạn, tôi thấy bạn đã có database schema cho `task_recurrences` và entity `TaskRecurrence`. Đây là các đề xuất phương án triển khai usecase **Recurring Task**:

🔍 Phân tích hiện trạng dự án:

✅ Đã có sẵn:

- Database table `task_recurrences` với các trường cần thiết
- Entity `TaskRecurrence` với enum `RecurrenceType` (DAILY, WEEKLY, MONTHLY, YEARLY)
- TaskService infrastructure và transaction management
- Task history tracking system

❌ Cần bổ sung:

- Service layer cho recurring logic
- Scheduled job để tự động tạo recurring tasks
- Business logic tính toán next due date

🚀 Đề xuất các phương án triển khai:

Phương án 1: Spring Scheduler + Cron Jobs (Khuyến nghị)

Công nghệ: Spring Boot `@Scheduled`, `@EnableScheduling`

Ý tưởng:

- Tạo `RecurringTaskService` để handle recurring logic
- Sử dụng `@Scheduled(fixedRate = 300000)` (5 phút) để check và tạo tasks mới
- Algorithm: Duyệt qua các `task_recurrences` có `is_active = true` và `next_due_date <= now()`

Thuật toán:

1. **Check**: Query các recurring tasks đến hạn
2. **Create**: Clone task gốc với due_date mới
3. **Update**: Tính toán và update `next_due_date` cho lần tiếp theo
4. **History**: Log vào `task_history` với action `RECURRING_CREATED`

Ưu điểm: Simple, reliable, sử dụng infrastructure có sẵn **Nhược điểm:** Cần server chạy liên tục

Phương án 2: Database Triggers + Event Scheduler

Công nghệ: MySQL Event Scheduler, Database Triggers

Ý tưởng:

- Tạo MySQL stored procedure để handle recurring logic
- Sử dụng `CREATE EVENT` để chạy procedure định kỳ
- Application chỉ cần CRUD recurring configurations

Thuật toán:

- **Stored Procedure**: Logic tạo recurring tasks
- **Event**: `EVERY 5 MINUTE` trigger procedure
- **Calculation**: Sử dụng MySQL date functions (`DATE_ADD`, `INTERVAL`)

Ưu điểm: Independent từ application, performance cao **Nhược điểm:** Tight coupling với MySQL, khó debug

Phương án 3: Message Queue + Delayed Jobs

Công nghệ: Redis/RabbitMQ, Spring Boot Queue

Ý tưởng:

- Khi tạo recurring task, schedule một delayed message
- Message consumer sẽ tạo task mới và schedule message tiếp theo
- Self-perpetuating cycle

Thuật toán:

1. **Enqueue**: Schedule message với delay = time until next occurrence
2. **Consume**: Tạo task mới, tính next occurrence, enqueue message mới
3. **Resilience**: Dead letter queue cho failed jobs

Ưu điểm: Scalable, fault-tolerant, distributed **Nhược điểm:** Complex setup, infrastructure overhead

🎯 Khuyến nghị Implementation Strategy:

Phase 1: MVP với Spring Scheduler

Core Components cần tạo:

1. **RecurringTaskService**: Business logic cho recurring
2. **RecurringTaskScheduler**: `@Component` với `@Scheduled` methods
3. **DateCalculationUtil**: Utility tính toán next due date
4. **RecurringTaskDTO/Response**: API DTOs

Algorithm cho Date Calculation:

```
// Pseudo-code
LocalDateTime calculateNextDueDate(RecurrenceType type, int interval, LocalDateTime current) {
    switch(type) {
        case DAILY: return current.plusDays(interval);
        case WEEKLY: return current.plusWeeks(interval);
        case MONTHLY: return current.plusMonths(interval);
        case YEARLY: return current.plusYears(interval);
    }
}
```

Kiến thức cần sử dụng:

- **Spring Scheduling**: `@EnableScheduling`, `@Scheduled`
- **Java Time API**: `LocalDateTime`, `Duration`, `Period`
- **Transaction Management**: `@Transactional` cho consistency
- **Builder Pattern**: Tận dụng existing patterns trong dự án

Phase 2: Advanced Features

Optimizations:

1. **Batch Processing**: Tạo nhiều recurring tasks trong 1 transaction
2. **Timezone Handling**: Support multiple timezones
3. **Exception Days**: Skip holidays/weekends
4. **Bulk Operations**: Mass enable/disable recurring

Monitoring:

- **Health Check**: Endpoint kiểm tra scheduler status
- **Metrics**: Count recurring tasks created/failed
- **Logging**: Detailed logs cho debugging

⚠️ Technical Considerations:

Database Optimizations:

- **Index**: `(is_active, next_due_date)` composite index
- **Partitioning**: Partition by `next_due_date` nếu data lớn
- **Cleanup**: Soft delete expired recurrences

Error Handling:

- **Retry Logic**: Retry failed recurring task creation
- **Circuit Breaker**: Stop scheduling nếu too many failures
- **Alerting**: Notify admin về scheduling failures

Performance:

- **Batch Size**: Limit số lượng recurring tasks process mỗi lần
- **Async Processing**: Non-blocking task creation
- **Caching**: Cache recurring configurations

🎯 Recommended Approach cho dự án của bạn:

Start với Phương án 1 vì:

- ✅ Tương thích với architecture hiện tại
- ✅ Sử dụng được infrastructure có sẵn (TaskService, repositories)
- ✅ Easy to test và debug
- ✅ Có thể extend sau này

Implementation Order:

1. **RecurringTaskService** - Core business logic
2. **DateCalculationUtil** - Date calculation algorithms
3. **RecurringTaskScheduler** - Scheduled job component
4. **API endpoints** - CRUD cho recurring configurations
5. **Testing** - Unit tests và integration tests

Key Success Factors:

- **Idempotency**: Đảm bảo không tạo duplicate recurring tasks
- **Atomicity**: Recurring task creation phải atomic
- **Observability**: Logging và monitoring đầy đủ
- **Graceful Degradation**: Handle failures gracefully

Approach này leverage được strengths của Spring Boot ecosystem trong dự án hiện tại và có thể scale sau này nếu cần. Tôi sẽ đưa ra kiến thức lý thuyết đầy đủ về **Spring Scheduling** và **Advanced Features** cho recurring task system:

🎯 PHẦN I: Spring Scheduling - Kiến thức nền tảng

1. Core Concepts

@EnableScheduling Annotation

- **Vai trò**: Kích hoạt Spring's scheduling capabilities
- **Mechanism**: Tạo `TaskScheduler` bean và `ScheduledAnnotationBeanPostProcessor`
- **Thread Model**: Mặc định sử dụng single thread pool (có thể gây bottleneck)

@Scheduled Annotation Options

```
// Fixed Rate - chạy cố định mỗi X milliseconds (bất kể method execution time)
@Scheduled(fixedRate = 5000)

// Fixed Delay - chờ X milliseconds AFTER method hoàn thành
@Scheduled(fixedDelay = 5000)

// Cron Expression - flexible scheduling
@Scheduled(cron = "0 */5 * * * *") // Mỗi 5 phút

// Initial Delay - delay trước lần chạy đầu tiên
@Scheduled(fixedRate = 5000, initialDelay = 10000)
```

Cron Expression Deep Dive

- **Format**: `second minute hour day-of-month month day-of-week [year]`
- **Special Characters**: `*` (any), `?` (no specific), `-` (range), `,` (list), `/` (step)
- **Examples**:
  - `0 0 12 * * ?` - Hàng ngày 12:00 PM
  - `0 0/15 * * * ?` - Mỗi 15 phút
  - `0 0 9-17 * * MON-FRI` - Giờ hành chính

2. Thread Pool Configuration

Default Behavior Issues

- Spring mặc định dùng **single thread** cho tất cả scheduled tasks
- **Problem**: Tasks chạy tuần tự, blocking nhau
- **Solution**: Custom ThreadPoolTaskScheduler

Custom Task Scheduler Configuration

```
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(10); // 10 concurrent threads
        scheduler.setThreadNamePrefix("recurring-task-");
        scheduler.setWaitForTasksToCompleteOnShutdown(true);
        scheduler.setAwaitTerminationSeconds(30);
        scheduler.initialize();

        taskRegistrar.setTaskScheduler(scheduler);
    }
}
```

3. Error Handling trong Scheduled Tasks

Default Behavior

- Exception trong scheduled method sẽ **ngăn không cho** subsequent executions
- **Silent failure** - không có automatic retry

Best Practices

```
@Scheduled(fixedRate = 300000)
public void processRecurringTasks() {
    try {
        // Main logic
        doProcessRecurringTasks();
    } catch (Exception e) {
        // Log error nhưng không re-throw để không stop scheduler
        logger.error("Failed to process recurring tasks", e);
        // Optional: Send alert notification
    }
}
```

4. Transaction Management với Scheduled Tasks

Problem: Scheduled methods không automatic có transaction context

Solution: Explicit `@Transactional` annotation

```
@Service
public class RecurringTaskService {

    @Scheduled(fixedRate = 300000)
    @Transactional // □ Explicit transaction
    public void createRecurringTasks() {
        // Database operations sẽ được wrap trong transaction
    }
}
```

□ PHẦN II: Phase 2 Advanced Features - Kiến thức chuyên sâu

1. Batch Processing Concepts

Lý thuyết

- **Definition**: Xử lý nhiều items cùng lúc thay vì từng item riêng lẻ
- **Benefits**: Giảm database round trips, tận dụng connection pooling
- **Trade-offs**: Memory usage tăng, complexity tăng

Implementation Strategies

```
// Strategy 1: Batch Database Operations
List<Task> tasksToCreate = new ArrayList<>();
for (TaskRecurrence recurrence : dueRecurrences) {
    Task newTask = createTaskFromRecurrence(recurrence);
    tasksToCreate.add(newTask);
}
taskRepository.saveAll(tasksToCreate); // Single database call

// Strategy 2: Pagination
Page<TaskRecurrence> recurrences;
int page = 0;
do {
    PageRequest pageRequest = PageRequest.of(page, 100);
    recurrences = recurringTaskRepository.findDueRecurrences(now, pageRequest);
    processBatch(recurrences.getContent());
    page++;
} while (recurrences.hasNext());
```

2. Timezone Handling

Challenges

- **User Timezone vs Server Timezone**: Users ở multiple timezones
- **Daylight Saving Time**: Tự động adjustment
- **Historical Changes**: Timezone rules thay đổi theo thời gian

Solutions

```
// Strategy 1: Store user timezone
@Entity
public class TaskRecurrence {
    @Column(name = "timezone")
    private String timeZone = "UTC"; // Default UTC

    public LocalDateTime getNextDueDateInUserTimezone() {
        ZoneId userZone = ZoneId.of(timeZone);
        ZonedDateTime userTime = nextDueDate.atZone(ZoneId.systemDefault())
                                            .withZoneSameInstant(userZone);
        return userTime.toLocalDateTime();
    }
}

// Strategy 2: Timezone-aware calculation
public LocalDateTime calculateNextOccurrence(TaskRecurrence recurrence) {
    ZoneId userZone = ZoneId.of(recurrence.getTimeZone());
    ZonedDateTime current = ZonedDateTime.now(userZone);

    switch(recurrence.getRecurrenceType()) {
        case DAILY:
            return current.plusDays(recurrence.getIntervalValue())
                          .toLocalDateTime();
        // ... other cases
    }
}
```

3. Exception Days (Skip Holidays/Weekends)

Concepts

- **Business Days Only**: Skip weekends và holidays
- **Holiday Calendar**: Configurable holiday definitions
- **Skip vs Postpone**: Skip hoàn toàn vs dời sang ngày kế tiếp

Implementation Approach

```
public interface HolidayCalendar {
    boolean isHoliday(LocalDate date);
    boolean isWeekend(LocalDate date);
    LocalDate getNextBusinessDay(LocalDate date);
}

@Service
public class BusinessDayCalculator {

    public LocalDateTime adjustForBusinessDays(LocalDateTime proposedDate,
                                               TaskRecurrence recurrence) {
        if (!recurrence.isSkipWeekendsEnabled()) {
            return proposedDate;
        }

        LocalDate date = proposedDate.toLocalDate();
        while (holidayCalendar.isWeekend(date) || holidayCalendar.isHoliday(date)) {
            date = date.plusDays(1);
        }

        return date.atTime(proposedDate.toLocalTime());
    }
}
```

4. Bulk Operations

Mass Enable/Disable Pattern

```
// Strategy 1: Database-level bulk update
@Query("UPDATE TaskRecurrence tr SET tr.isActive = :status WHERE tr.taskId IN :taskIds")
int bulkUpdateRecurrenceStatus(@Param("taskIds") List<Long> taskIds,
                               @Param("status") Boolean status);

// Strategy 2: Event-driven bulk operations
@EventListener
public void handleBulkTaskStatusChange(BulkTaskStatusChangeEvent event) {
    List<Long> taskIds = event.getTaskIds();
    Boolean newStatus = event.getNewStatus();

    // Update recurrences in batches
    Lists.partition(taskIds, 100).forEach(batch -> {
        recurringTaskRepository.bulkUpdateRecurrenceStatus(batch, newStatus);
    });
}
```

□ PHẦN III: Technical Considerations - Giải thích chi tiết

1. Database Optimizations

Composite Index Strategy

```
-- Primary index cho scheduled job query
CREATE INDEX idx_recurring_due ON task_recurrences (is_active, next_due_date);

-- Secondary index cho user queries
CREATE INDEX idx_user_recurring ON task_recurrences (created_by, is_active);

-- Covering index để avoid table lookup
CREATE INDEX idx_recurring_cover ON task_recurrences
(is_active, next_due_date)
INCLUDE (task_id, recurrence_type, interval_value);
```

Lý do Composite Index hiệu quả:

- **Selectivity**: `is_active` filter trước (ít rows)
- **Range Scan**: `next_due_date` cho phép range queries
- **Query Plan**: Database có thể sử dụng index-only scan

Partitioning Strategy

```
-- Partition by next_due_date range
CREATE TABLE task_recurrences (
    -- columns
) PARTITION BY RANGE (YEAR(next_due_date)) (
    PARTITION p2024 VALUES LESS THAN (2025),
    PARTITION p2025 VALUES LESS THAN (2026),
    PARTITION p_future VALUES LESS THAN MAXVALUE
);
```

Benefits của Partitioning:

- **Partition Pruning**: Query chỉ scan relevant partitions
- **Maintenance**: Drop old partitions thay vì DELETE
- **Parallel Processing**: Different partitions trên different storage

2. Error Handling Deep Dive

Retry Logic với Exponential Backoff

```
@Retryable(
    value = {DataAccessException.class},
    maxAttempts = 3,
    backoff = @Backoff(delay = 1000, multiplier = 2)
)
public void createRecurringTask(TaskRecurrence recurrence) {
    // Implementation với auto-retry
}

@Recover
public void recoverFromRecurringTaskFailure(DataAccessException ex,
                                            TaskRecurrence recurrence) {
    // Fallback logic - log to error table, send alert
    errorLogService.logRecurringTaskFailure(recurrence, ex);
}
```

Circuit Breaker Pattern

```
@Component
public class RecurringTaskCircuitBreaker {
    private final CircuitBreaker circuitBreaker;

    public RecurringTaskCircuitBreaker() {
        this.circuitBreaker = CircuitBreaker.ofDefaults("recurringTasks");
        circuitBreaker.getEventPublisher()
                      .onStateTransition(event ->
                          logger.warn("Circuit breaker state: " + event.getStateTransition()));
    }

    public void executeWithCircuitBreaker(Runnable task) {
        circuitBreaker.executeRunnable(task);
    }
}
```

3. Performance Considerations

Batch Size Tuning

```
@Value("${recurring.batch.size:100}")
private int batchSize;

@Value("${recurring.batch.timeout:30}")
private int batchTimeoutSeconds;

public void processBatch(List<TaskRecurrence> recurrences) {
    // Split into smaller batches if too large
    Lists.partition(recurrences, batchSize).forEach(batch -> {
        CompletableFuture.supplyAsync(() -> processBatchAsync(batch))
                         .orTimeout(batchTimeoutSeconds, TimeUnit.SECONDS)
                         .join();
    });
}
```

Async Processing với CompletableFuture

```
@Async("recurringTaskExecutor")
public CompletableFuture<Void> processRecurrenceAsync(TaskRecurrence recurrence) {
    try {
        createTaskFromRecurrence(recurrence);
        updateNextDueDate(recurrence);
        return CompletableFuture.completedFuture(null);
    } catch (Exception e) {
        return CompletableFuture.failedFuture(e);
    }
}
```

Caching Strategy

```
@Cacheable(value = "holidayCalendar", key = "#year")
public Set<LocalDate> getHolidaysForYear(int year) {
    // Expensive operation - cache results
    return holidayService.loadHolidays(year);
}

@CacheEvict(value = "recurringTasks", key = "#taskId")
public void updateRecurringTask(Long taskId, TaskRecurrenceDTO dto) {
    // Invalidate cache when updating
}
```

4. Monitoring và Observability

Custom Metrics với Micrometer

```
@Component
public class RecurringTaskMetrics {
    private final Counter tasksCreated;
    private final Counter tasksFailed;
    private final Timer processingTime;

    public RecurringTaskMetrics(MeterRegistry meterRegistry) {
        this.tasksCreated = Counter.builder("recurring_tasks_created")
                                .description("Number of recurring tasks created")
                                .register(meterRegistry);
        // ... other metrics
    }

    public void recordTaskCreated() {
        tasksCreated.increment();
    }
}
```

Health Check Implementation

```
@Component
public class RecurringTaskHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        long overdueCount = recurringTaskRepository.countOverdueTasks();
        long failedCount = errorLogRepository.countRecentFailures();

        if (overdueCount > 100 || failedCount > 50) {
            return Health.down()
                        .withDetail("overdueCount", overdueCount)
                        .withDetail("failedCount", failedCount)
                        .build();
        }

        return Health.up()
                    .withDetail("status", "All recurring tasks processing normally")
                    .build();
    }
}
```

🎯 Tổng kết Implementation Strategy

Phase 1: Foundation (Week 1-2)

1. **Basic Scheduling**: `@Scheduled` với `fixedRate`
2. **Core Service**: `RecurringTaskService` với basic CRUD
3. **Date Calculation**: Simple next due date calculation
4. **Basic Error Handling**: Try-catch với logging

Phase 2: Production Ready (Week 3-4)

1. **Thread Pool**: Custom `TaskScheduler` configuration
2. **Batch Processing**: Process multiple recurrences cùng lúc
3. **Comprehensive Error Handling**: Retry, circuit breaker
4. **Database Optimization**: Proper indexing

Phase 3: Advanced Features (Week 5-6)

1. **Timezone Support**: Multi-timezone handling
2. **Holiday Calendar**: Business day calculations

3. **Monitoring**: Metrics, health checks, alerting
4. **Performance Tuning**: Async processing, caching

Key Success Metrics:

- **Reliability**: >99.9% recurring tasks created successfully
- **Performance**: <100ms average processing time per recurrence
- **Scalability**: Handle 10,000+ active recurring tasks
- **Observability**: Full monitoring và alerting coverage

Tôi sẽ đưa ra kiến thức lý thuyết đầy đủ về **Spring Scheduling** và **Advanced Features** cho recurring task system:

□ PHẦN I: Spring Scheduling - Kiến thức nền tảng

1. Core Concepts

@EnableScheduling Annotation

- **Vai trò**: Kích hoạt Spring's scheduling capabilities
- **Mechanism**: Tạo `TaskScheduler` bean và `ScheduledAnnotationBeanPostProcessor`
- **Thread Model**: Mặc định sử dụng single thread pool (có thể gây bottleneck)

@Scheduled Annotation Options

```
// Fixed Rate - chạy cố định mỗi X milliseconds (bất kể method execution time)
@Scheduled(fixedRate = 5000)

// Fixed Delay - chờ X milliseconds AFTER method hoàn thành
@Scheduled(fixedDelay = 5000)

// Cron Expression - flexible scheduling
@Scheduled(cron = "0 */5 * * * *") // Mỗi 5 phút

// Initial Delay - delay trước lần chạy đầu tiên
@Scheduled(fixedRate = 5000, initialDelay = 10000)
```

Cron Expression Deep Dive

- **Format**: `second minute hour day-of-month month day-of-week [year]`
- **Special Characters**: `*` (any), `?` (no specific), `-` (range), `,` (list), `/` (step)
- **Examples**:
  - `0 0 12 * * ?` - Hàng ngày 12:00 PM
  - `0 0/15 * * * ?` - Mỗi 15 phút
  - `0 0 9-17 * * MON-FRI` - Giờ hành chính

2. Thread Pool Configuration

Default Behavior Issues

- Spring mặc định dùng **single thread** cho tất cả scheduled tasks
- **Problem**: Tasks chạy tuần tự, blocking nhau
- **Solution**: Custom ThreadPoolTaskScheduler

Custom Task Scheduler Configuration

```
@Configuration
@EnableScheduling
public class SchedulingConfig implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        ThreadPoolTaskScheduler scheduler = new ThreadPoolTaskScheduler();
        scheduler.setPoolSize(10); // 10 concurrent threads
        scheduler.setThreadNamePrefix("recurring-task-");
        scheduler.setWaitForTasksToCompleteOnShutdown(true);
        scheduler.setAwaitTerminationSeconds(30);
        scheduler.initialize();

        taskRegistrar.setTaskScheduler(scheduler);
    }
}
```

3. Error Handling trong Scheduled Tasks

Default Behavior

- Exception trong scheduled method sẽ **ngăn không cho** subsequent executions
- **Silent failure** - không có automatic retry

Best Practices

```
@Scheduled(fixedRate = 300000)
public void processRecurringTasks() {
    try {
        // Main logic
        doProcessRecurringTasks();
    } catch (Exception e) {
        // Log error nhưng không re-throw để không stop scheduler
        logger.error("Failed to process recurring tasks", e);
        // Optional: Send alert notification
    }
}
```

4. Transaction Management với Scheduled Tasks

Problem: Scheduled methods không automatic có transaction context

Solution: Explicit `@Transactional` annotation

```
@Service
public class RecurringTaskService {

    @Scheduled(fixedRate = 300000)
    @Transactional // □ Explicit transaction
    public void createRecurringTasks() {
        // Database operations sẽ được wrap trong transaction
    }
}
```

□ PHẦN II: Phase 2 Advanced Features - Kiến thức chuyên sâu

1. Batch Processing Concepts

Lý thuyết

- **Definition**: Xử lý nhiều items cùng lúc thay vì từng item riêng lẻ
- **Benefits**: Giảm database round trips, tận dụng connection pooling
- **Trade-offs**: Memory usage tăng, complexity tăng

Implementation Strategies

```
// Strategy 1: Batch Database Operations
List<Task> tasksToCreate = new ArrayList<>();
for (TaskRecurrence recurrence : dueRecurrences) {
    Task newTask = createTaskFromRecurrence(recurrence);
    tasksToCreate.add(newTask);
}
taskRepository.saveAll(tasksToCreate); // Single database call

// Strategy 2: Pagination
Page<TaskRecurrence> recurrences;
int page = 0;
do {
    PageRequest pageRequest = PageRequest.of(page, 100);
    recurrences = recurringTaskRepository.findDueRecurrences(now, pageRequest);
    processBatch(recurrences.getContent());
    page++;
} while (recurrences.hasNext());
```

2. Timezone Handling

Challenges

- **User Timezone vs Server Timezone**: Users ở multiple timezones
- **Daylight Saving Time**: Tự động adjustment
- **Historical Changes**: Timezone rules thay đổi theo thời gian

Solutions

```
// Strategy 1: Store user timezone
@Entity
public class TaskRecurrence {
    @Column(name = "timezone")
    private String timeZone = "UTC"; // Default UTC

    public LocalDateTime getNextDueDateInUserTimezone() {
        ZoneId userZone = ZoneId.of(timeZone);
        ZonedDateTime userTime = nextDueDate.atZone(ZoneId.systemDefault())
                                            .withZoneSameInstant(userZone);
        return userTime.toLocalDateTime();
    }
}

// Strategy 2: Timezone-aware calculation
public LocalDateTime calculateNextOccurrence(TaskRecurrence recurrence) {
    ZoneId userZone = ZoneId.of(recurrence.getTimeZone());
    ZonedDateTime current = ZonedDateTime.now(userZone);

    switch(recurrence.getRecurrenceType()) {
        case DAILY:
            return current.plusDays(recurrence.getIntervalValue())
                          .toLocalDateTime();
        // ... other cases
    }
}
```

3. Exception Days (Skip Holidays/Weekends)

Concepts

- **Business Days Only**: Skip weekends và holidays
- **Holiday Calendar**: Configurable holiday definitions
- **Skip vs Postpone**: Skip hoàn toàn vs dời sang ngày kế tiếp

Implementation Approach

```
public interface HolidayCalendar {
    boolean isHoliday(LocalDate date);
    boolean isWeekend(LocalDate date);
    LocalDate getNextBusinessDay(LocalDate date);
}

@Service
public class BusinessDayCalculator {

    public LocalDateTime adjustForBusinessDays(LocalDateTime proposedDate,
                                               TaskRecurrence recurrence) {
        if (!recurrence.isSkipWeekendsEnabled()) {
            return proposedDate;
        }

        LocalDate date = proposedDate.toLocalDate();
        while (holidayCalendar.isWeekend(date) || holidayCalendar.isHoliday(date))
        {
            date = date.plusDays(1);
        }

        return date.atTime(proposedDate.toLocalTime());
    }
}
```

4. Bulk Operations

Mass Enable/Disable Pattern

```
// Strategy 1: Database-level bulk update
@Query("UPDATE TaskRecurrence tr SET tr.isActive = :status WHERE tr.taskId IN :taskIds")
int bulkUpdateRecurrenceStatus(@Param("taskIds") List<Long> taskIds,
                               @Param("status") Boolean status);

// Strategy 2: Event-driven bulk operations
@EventListener
public void handleBulkTaskStatusChange(BulkTaskStatusChangeEvent event) {
    List<Long> taskIds = event.getTaskIds();
    Boolean newStatus = event.getNewStatus();

    // Update recurrences in batches
    Lists.partition(taskIds, 100).forEach(batch -> {
        recurringTaskRepository.bulkUpdateRecurrenceStatus(batch, newStatus);
    });
}
```

□ PHẦN III: Technical Considerations - Giải thích chi tiết

1. Database Optimizations

Composite Index Strategy

```
-- Primary index cho scheduled job query
CREATE INDEX idx_recurring_due ON task_recurrences (is_active, next_due_date);

-- Secondary index cho user queries
CREATE INDEX idx_user_recurring ON task_recurrences (created_by, is_active);

-- Covering index để avoid table lookup
CREATE INDEX idx_recurring_cover ON task_recurrences
(is_active, next_due_date)
INCLUDE (task_id, recurrence_type, interval_value);
```

Lý do Composite Index hiệu quả:

- **Selectivity**: `is_active` filter trước (ít rows)
- **Range Scan**: `next_due_date` cho phép range queries
- **Query Plan**: Database có thể sử dụng index-only scan

Partitioning Strategy

```
-- Partition by next_due_date range
CREATE TABLE task_recurrences (
    -- columns
) PARTITION BY RANGE (YEAR(next_due_date)) (
    PARTITION p2024 VALUES LESS THAN (2025),
    PARTITION p2025 VALUES LESS THAN (2026),
    PARTITION p_future VALUES LESS THAN MAXVALUE
);
```

Benefits của Partitioning:

- **Partition Pruning**: Query chỉ scan relevant partitions
- **Maintenance**: Drop old partitions thay vì DELETE
- **Parallel Processing**: Different partitions trên different storage

2. Error Handling Deep Dive

Retry Logic với Exponential Backoff

```
@Retryable(
    value = {DataAccessException.class},
    maxAttempts = 3,
    backoff = @Backoff(delay = 1000, multiplier = 2)
)
public void createRecurringTask(TaskRecurrence recurrence) {
    // Implementation với auto-retry
}

@Recover
public void recoverFromRecurringTaskFailure(DataAccessException ex,
                                            TaskRecurrence recurrence) {
    // Fallback logic - log to error table, send alert
    errorLogService.logRecurringTaskFailure(recurrence, ex);
}
```

Circuit Breaker Pattern

```
@Component
public class RecurringTaskCircuitBreaker {
    private final CircuitBreaker circuitBreaker;

    public RecurringTaskCircuitBreaker() {
        this.circuitBreaker = CircuitBreaker.ofDefaults("recurringTasks");
        circuitBreaker.getEventPublisher()
                      .onStateTransition(event ->
                          logger.warn("Circuit breaker state: " + event.getStateTransition()));
    }

    public void executeWithCircuitBreaker(Runnable task) {
        circuitBreaker.executeRunnable(task);
    }
}
```

3. Performance Considerations

Batch Size Tuning

```
@Value("${recurring.batch.size:100}")
private int batchSize;

@Value("${recurring.batch.timeout:30}")
private int batchTimeoutSeconds;

public void processBatch(List<TaskRecurrence> recurrences) {
    // Split into smaller batches if too large
    Lists.partition(recurrences, batchSize).forEach(batch -> {
        CompletableFuture.supplyAsync(() -> processBatchAsync(batch))
                         .orTimeout(batchTimeoutSeconds, TimeUnit.SECONDS)
                         .join();
    });
}
```

Async Processing với CompletableFuture

```
@Async("recurringTaskExecutor")
public CompletableFuture<Void> processRecurrenceAsync(TaskRecurrence recurrence) {
    try {
        createTaskFromRecurrence(recurrence);
        updateNextDueDate(recurrence);
        return CompletableFuture.completedFuture(null);
    } catch (Exception e) {
        return CompletableFuture.failedFuture(e);
    }
}
```

Caching Strategy

```
@Cacheable(value = "holidayCalendar", key = "#year")
public Set<LocalDate> getHolidaysForYear(int year) {
    // Expensive operation - cache results
    return holidayService.loadHolidays(year);
}

@CacheEvict(value = "recurringTasks", key = "#taskId")
public void updateRecurringTask(Long taskId, TaskRecurrenceDTO dto) {
    // Invalidate cache when updating
}
```

4. Monitoring và Observability

Custom Metrics với Micrometer

```
@Component
public class RecurringTaskMetrics {
    private final Counter tasksCreated;
    private final Counter tasksFailed;
    private final Timer processingTime;

    public RecurringTaskMetrics(MeterRegistry meterRegistry) {
        this.tasksCreated = Counter.builder("recurring_tasks_created")
                                .description("Number of recurring tasks created")
                                .register(meterRegistry);
        // ... other metrics
    }

    public void recordTaskCreated() {
        tasksCreated.increment();
    }
}
```

Health Check Implementation

```
@Component
public class RecurringTaskHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        long overdueCount = recurringTaskRepository.countOverdueTasks();
        long failedCount = errorLogRepository.countRecentFailures();

        if (overdueCount > 100 || failedCount > 50) {
            return Health.down()
                        .withDetail("overdueCount", overdueCount)
                        .withDetail("failedCount", failedCount)
                        .build();
        }

        return Health.up()
                    .withDetail("status", "All recurring tasks processing normally")
                    .build();
    }
}
```

🎯 Tổng kết Implementation Strategy

Phase 1: Foundation (Week 1-2)

1. **Basic Scheduling**: `@Scheduled` với `fixedRate`
2. **Core Service**: `RecurringTaskService` với basic CRUD
3. **Date Calculation**: Simple next due date calculation
4. **Basic Error Handling**: Try-catch với logging

Phase 2: Production Ready (Week 3-4)

1. **Thread Pool**: Custom `TaskScheduler` configuration
2. **Batch Processing**: Process multiple recurrences cùng lúc
3. **Comprehensive Error Handling**: Retry, circuit breaker
4. **Database Optimization**: Proper indexing

Phase 3: Advanced Features (Week 5-6)

1. **Timezone Support**: Multi-timezone handling
2. **Holiday Calendar**: Business day calculations

3. **Monitoring**: Metrics, health checks, alerting
4. **Performance Tuning**: Async processing, caching

Key Success Metrics:

- **Reliability**: >99.9% recurring tasks created successfully
- **Performance**: <100ms average processing time per recurrence
- **Scalability**: Handle 10,000+ active recurring tasks
- **Observability**: Full monitoring và alerting coverage