12. 날짜시간 API

Java 8의 새로운 날짜와 시간 API

Java 1.0에서는 `java.util.Date` 클래스 하나로 날짜/시간 관련 기능을 제공하였다.

레거시 API의 문제점

Date 클래스의 문제점

불변성 부재

- Date 객체는 불변 객체가 아니라서 생성 후에도 내부 값 변경이 가능하다.
- Thread Safe하지 않아 멀티 스레드 환경에서 문제가 발생한다.

```
Date date = new Date();
date.setYear(2020); // 내부 값 변경 가능하여 버그 가능성이 증가
```

메소드 이름 및 동작의 혼란

```
date.getYear();  // 1900년부터 시작
date.getMonth(); // 9월이면 8을 반환 (0부터 시작)
```

시간대 정보 부재

- Date 자체는 시간대 정보를 포함하지 않는다.
- 단순히 밀리초 기반 타임스탬프만 관리한다.
- 지역/캘린더 고려가 불가능하다.

Calendar 클래스의 문제점

Calendar 클래스는 Date 클래스를 보완하기 위해 등장했지만, 여전히 많은 문제점을 가지고 있다.

복잡성 및 가독성 저하

상수(`Calendar.YEAR` , `Calendar.MONTH` 등)를 조합해서 사용하는 구조로 코드가 장황하며 직관적이지 않다.

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.YEAR, 2025);
cal.set(Calendar.MONTH, Calendar.SEPTEMBER); // 월은 여전히 0부터 시작
```

가변성 문제

- Calendar도 내부 상태를 언제든 변경할 수 있다.
- `add()`, `roll()` 같은 메소드가 원본 객체를 수정하여 의도치 않은 버그가 발생할 수 있다.

시간대 처리의 어려움

- 시간대 변경 시 내부 계산이 예측하기 어렵거나 불편하다.
- 계산 시 직접 로직을 작성해야 한다.

호환성 문제

- Date와 서로 호환되지 않는다.

새로운 날짜와 시간 API

LocalDate와 LocalTime

`java.time` 패키지에는 `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Duration`, `Period` 등의 클래스가 제공된다.

LocalDate

`LocalDate`는 시간을 제외한 날짜를 표현하는 불변 객체이다.

주요 특징

- 어떠한 시간대 정보도 포함하지 않는다.
- 불변 객체이며 Thread Safe하다.

```
// LocalDate의 of 메소드로 인스턴스 생성
LocalDate date = LocalDate.of(2017, 9, 21); // 2017-09-21

int year = date.getYear();              // 2017
Month month = date.getMonth();          // SEPTEMBER
int day = date.getDayOfMonth();         // 21
int len = date.lengthOfMonth();         // 31
boolean leap = date.isLeapYear();       // 윤년 여부
```

팩토리 메소드 `now()`는 시스템 시계의 정보를 이용해서 현재 날짜 정보를 얻는다.

```
LocalDate today = LocalDate.now();
```

TemporalField와 ChronoField

`TemporalField`를 전달해서 정보를 얻는 방법도 존재한다.

> **TemporalField**: 시간 관련 객체에서 어떤 필드의 값에 접근할지 정의하는 인터페이스이다.

TemporalField 인터페이스

시간의 어떤 필드(단위)를 나타낸다. 연도, 월, 일, 시, 분, 초를 표현하며, `LocalDate`나 `LocalTime` 같은 `Temporal` 객체에서 어떤 값을 꺼내거나 설정하는 규약을 정의한다.

```
public interface TemporalField {
    String getDisplayName(Locale locale);  // 필드의 표시 이름
    TemporalUnit getBaseUnit();            // 기본 단위 (예: DAY_OF_MONTH -> DAYS)
    TemporalUnit getRangeUnit();           // 범위 단위 (예: DAY_OF_MONTH ->
MONTHS)
    ValueRange range();                     // 가능한 값 범위
    boolean isDateBased();                  // 날짜 관련 필드인가
    boolean isTimeBased();                  // 시간 관련 필드인가
}
```

ChronoField 열거형

`ChronoField`는 `TemporalField`의 구현체(열거형 Enum)이다.

```
ChronoField.YEAR              // 연도
ChronoField.MONTH_OF_YEAR     // 월 (1~12)
ChronoField.DAY_OF_MONTH      // 일 (1~31)
ChronoField.HOUR_OF_DAY       // 시 (0~23)
ChronoField.MINUTE_OF_HOUR    // 분 (0~59)
```

사용 예시:

```
int year = date.get(ChronoField.YEAR);
```

`getYear()`, `getMonthValue()`, `getDayOfMonth()` 등을 이용하면 가독성을 더욱 높일 수 있다.

LocalTime

`13:45:20` 같은 시간은 `LocalTime`으로 표현할 수 있다. 정적 메소드 `of`로 `LocalTime` 인스턴스를 만들 수 있으며, 시간/분/초 혹은 시간/분을 인수로 받는다.

```
// 현재 시간
LocalTime now = LocalTime.now();

// 특정 시간
LocalTime time1 = LocalTime.of(14, 30);      // 14:30:00
LocalTime time2 = LocalTime.of(14, 30, 45);  // 14:30:45
```

시간 정보 추출

```
LocalTime time = LocalTime.of(13, 45, 20); // 13:45:20
int hour = time.getHour();                 // 13
int minute = time.getMinute();             // 45
int second = time.getSecond();             // 20
```

문자열 파싱

날짜와 시간 문자열로 `LocalDate` / `LocalTime` 의 인스턴스를 만들 수 있다.

```
LocalDate date = LocalDate.parse("2017-09-21");
LocalTime time = LocalTime.parse("13:45:20");
```

`parse` 메소드에 `DateTimeFormatter` 를 전달할 수 있다. `DateTimeFormatter` 의 인스턴스는 날짜, 시간 객체의 형식을 지정한다.

> `DateTimeFormatter` 는 `DateFormat` 클래스를 대체한다.

LocalDateTime: 날짜와 시간 조합

`LocalDateTime` 은 `LocalDate` 와 `LocalTime` 을 쌍으로 갖는 복합 클래스이다.

주요 특징

- 날짜와 시간을 모두 표현할 수 있다: 년, 월, 일, 시, 분, 초, 나노초
- 불변 객체이며 스레드 안전하다.

```
// 1) 현재 날짜와 시간
LocalDateTime now = LocalDateTime.now();
LocalDateTime dt1 = LocalDateTime.of(2025, 9, 29, 15, 45); // 2025-09-29T15:45

// 2) LocalDate와 LocalTime 합치기
LocalDate date = LocalDate.of(2025, 9, 29);
LocalTime time = LocalTime.of(15, 45);
LocalDateTime dt2 = LocalDateTime.of(date, time);

// 3) 문자열 파싱
LocalDateTime dt3 = LocalDateTime.parse("2025-09-29T15:45:30");

// 4) 날짜/시간 추출
LocalDate extractedDate = dt1.toLocalDate(); // 2025-09-29
LocalTime extractedTime = dt1.toLocalTime(); // 15:45
```

Instant: 기계의 날짜와 시간

`java.time.Instant` 는 UTC 기준의 **순간(시점, timestamp)**을 나타낸다.

주요 특징

- 1970년 1월 1일 00:00:00 UTC(Unix epoch) 기준으로 초와 나노초를 표현한다.
- 불변 객체이며 스레드 안전하다.

사람은 주, 날짜, 시간, 분으로 날짜와 시간을 계산하지만, 기계는 연속된 시간에서 특정 지점을 하나의 큰 수로 표현하는 것이 가장 자연스럽다.

팩토리 메소드 `ofEpochSecond`에 초를 넘겨줘서 `Instant` 클래스 인스턴스를 만들 수 있다. 나노초(10억 분의 1)의 정밀도를 제공하며, 오버로드된 `ofEpochSecond` 메소드는 두 번째 인수를 사용해 나노초 단위로 시간을 보정할 수 있다.

```
Instant now = Instant.now();

Instant instant1 = Instant.ofEpochSecond(3);
Instant instant2 = Instant.ofEpochSecond(3, 0);
Instant instant3 = Instant.ofEpochSecond(2, 1_000_000_000); // 2초 + 10억 나노초 =
3초
Instant instant4 = Instant.ofEpochSecond(4, -1_000_000_000); // 4초 - 10억 나노초
= 3초
```

`Instant`는 초와 나노초 정보를 포함하지만, 사람이 읽을 수 있는 시간 정보는 제공하지 않는다.

Duration과 Period

두 시간 객체 사이의 지속 시간을 구하려고 할 때 `between` 메소드를 사용한다.

Duration

`java.time.Duration`은 **시간 기반의 간격**을 나타낸다.

주요 특징

- 나노초까지 정밀하게 표현할 수 있다.
- 두 시각 사이의 간격을 계산할 때 사용한다.

```
Duration d1 = Duration.between(time1, time2);
Duration d2 = Duration.between(dateTime1, dateTime2);
Duration d3 = Duration.between(instant1, instant2);
```

`LocalDateTime`은 사람이 사용하며, `Instant`는 기계가 사용하도록 되어 있어서 서로 혼합해서 사용하면 안 된다.

`Duration`은 나노초/초 단위로 시간 단위를 표현하므로 `LocalDate`를 전달할 수 없다. 날짜 간격을 계산하려면 `Period`를 사용한다.

Period

`java.time.Period`는 **날짜 기반의 간격**을 나타낸다.

주요 특징

- 월/윤년을 고려한다.
- 날짜 간격을 계산할 때 사용한다.

```
Period tenDays = Period.between(
    LocalDate.of(2017, 9, 11),
    LocalDate.of(2017, 9, 21)
);
```

Duration과 Period 직접 생성

두 시간 객체를 사용하지 않고도 `Duration`과 `Period`를 만들 수 있다.

```
Duration threeMinutes = Duration.ofMinutes(3);
Duration threeMinutes2 = Duration.of(3, ChronoUnit.MINUTES);

Period tenDays = Period.ofDays(10);
Period threeWeeks = Period.ofWeeks(3);
Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);
```

중간 정리: 시간 API 구조

Temporal 계층 구조

```
Temporal (시간 개념의 최상위 인터페이스)
  ├── TemporalAccessor (읽기 전용)
  └── Temporal (읽기/쓰기 + 조정 가능)
       └── LocalDate, LocalTime, LocalDateTime, Instant 등
           └── with(), plus(), minus() 제공
```

Temporal/TemporalAccessor

- `TemporalAccessor` : 읽기 전용, 날짜/시간 접근
- `Temporal` : 읽기+쓰기/조정 가능한 날짜/시간 객체

Temporal 구현체

- `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant` 등
- `with()`, `plus()`, `minus()`, `get()` 메소드 제공

TemporalField (필드 단위 접근)

```
TemporalField (필드 단위 접근: 연, 월, 일, 시, 분)
  └── ChronoField (enum)
        └── YEAR, MONTH_OF_YEAR, DAY_OF_MONTH, HOUR_OF_DAY 등
```

날짜/시간의 필드 단위에 접근한다. `ChronoField`는 `TemporalField`의 구현체(enum)이다.

날짜/시간에서 **값을 읽거나 쓸 수 있는 필드(속성)**를 의미한다.

> 날짜의 칸 중 하나 → 2025년 10월 25일 중 "10월"
> 시계의 눈금 → 현재 "15시"

```
date.get(ChronoField.YEAR);           // 2025
date.get(ChronoField.MONTH_OF_YEAR);  // 10
date.get(ChronoField.DAY_OF_MONTH);   // 25
```

TemporalUnit (기간 단위)

```
TemporalUnit (기간 단위: 일, 주, 월, 년, 초, 나노초)
  └── ChronoUnit (enum)
        └── DAYS, WEEKS, MONTHS, YEARS, SECONDS, NANOS 등
```

`ChronoUnit`은 `TemporalUnit`의 구현체(enum)이다. 날짜/시간의 **기간 단위**를 나타낸다.

> 칸과 칸 사이의 단위 → 1개월 간격
> 시간의 흐름 단위 → 1시간 뒤

```
LocalDate date = LocalDate.of(2025, 10, 2);
LocalDate newDate = date.plus(10, ChronoUnit.DAYS); // 2025-10-12
```

TemporalAdjuster (날짜 조정)

```
TemporalAdjuster (날짜 조정) - Interface
  └── TemporalAdjusters (유틸 클래스)
        └── firstDayOfMonth(), lastDayOfMonth(), next(DayOfWeek) 등
```

특정 규칙에 따라 날짜를 조정하는 전략 객체이다. `TemporalAdjusters`는 자주 쓰는 전략을 제공하는 유틸 클래스이다.

```
LocalDate date = LocalDate.now();
LocalDate lastDay = date.with(TemporalAdjusters.lastDayOfMonth());
```

열거형 타입

DayOfWeek (요일 열거형)

- `MONDAY(1)` ~ `SUNDAY(7)`
- 요일을 표현하는 enum

```
DayOfWeek dow = DayOfWeek.MONDAY; // getValue() 하면 1
```

Month (월 열거형)

- `JANUARY(1)` ~ `DECEMBER(12)`
- 월을 표현하는 enum

```
Month m = Month.OCTOBER; // getValue() 하면 10
```

Duration vs Period

Period는 날짜 기반(년/월/일)이고, Duration은 시간 기반(초/나노초)이다.

날짜 조정, 파싱, 포매팅

날짜 조정

`withAttribute` 메소드로 기존 `LocalDate`를 바꾼 버전을 간단하게 만들 수 있다.

```
LocalDate date1 = LocalDate.of(2017, 9, 21);              // 2017-09-21
LocalDate date2 = date1.withYear(2011);                   // 2011-09-21
LocalDate date3 = date2.withDayOfMonth(25);               // 2011-09-25
LocalDate date4 = date3.with(ChronoField.MONTH_OF_YEAR, 2); // 2011-02-25
```

이러한 `with` 메소드는 기존의 `Temporal` 객체를 바꾸지 않는다. **필드를 갱신한 복사본을 만든다.**

`Temporal` 객체가 지정된 필드를 지원하지 않으면 예외가 발생한다. `Instant`에 `ChronoField`를 사용하는 경우가 대표적이다.

상대적인 방식으로 속성 변경

```
LocalDate date1 = LocalDate.of(2017, 9, 1);
LocalDate date2 = date1.plusWeeks(1);               // 2017-09-08
LocalDate date3 = date2.minusYears(6);              // 2011-09-08
LocalDate date4 = date3.plus(6, ChronoUnit.MONTHS); // 2012-03-08
```

`plus`, `minus` 메소드도 `Temporal` 인터페이스에 정의되어 있다. 이를 이용해 `Temporal`을 특정 시간만큼 앞뒤로 이동할 수 있으며, `TemporalUnit`을 활용할 수 있다.

TemporalAdjusters 사용

`TemporalAdjusters`는 날짜 조정을 위한 정적 유틸리티 클래스이다.

주요 특징

- `with(TemporalAdjuster)`와 함께 사용해서 `LocalDate`, `LocalDateTime` 같은 날짜 객체를 특정 규칙에 맞게 변경할 수 있다.
- 속성을 직접 바꾸는 것이 아닌 규칙 기반을 제공한다.

```
import static java.time.temporal.TemporalAdjusters.*;

LocalDate date1 = LocalDate.of(2014, 3, 18);
LocalDate date2 = date1.with(nextOrSame(DayOfWeek.SUNDAY)); // 2014-03-23
LocalDate date3 = date2.with(lastDayOfMonth());              // 2014-03-31
```

주요 메소드

- `dayOfWeekInMonth` : 서수 요일에 해당하는 날짜를 반환
- `firstDayOfMonth` : 현재 달의 첫 번째 날짜를 반환
- `firstDayOfNextMonth` : 다음 달의 첫 번째 날짜를 반환
- `lastDayOfMonth` : 현재 달의 마지막 날짜를 반환
- `next(DayOfWeek)` : 다음 요일 날짜를 반환
- `nextOrSame(DayOfWeek)` : 다음 요일이거나 같은 날짜를 반환
- `previous(DayOfWeek)` : 이전 요일 날짜를 반환

이처럼 `TemporalAdjuster`를 이용하면 좀 더 복잡한 날짜 조정을 쉽게 해결할 수 있다.

날짜와 시간 객체 출력과 파싱

날짜/시간을 문자열로 바꾸는 것을 **포맷(format)**이라 하고, 문자열을 날짜/시간으로 바꾸는 것을 **파싱(parsing)**이라고 한다.

DateTimeFormatter

`java.time.format.DateTimeFormatter`는 날짜와 시간 포매팅을 담당한다.

주요 특징

- 불변 객체이며 스레드 안전하다.
- 문자열로 포맷하거나 문자열을 객체로 파싱할 때 사용한다.

정적 팩토리 메소드와 상수를 이용해 만들 수 있다.

날짜 포매팅

```
LocalDate date = LocalDate.of(2014, 3, 18);
String s1 = date.format(DateTimeFormatter.BASIC_ISO_DATE);  // 20140318
String s2 = date.format(DateTimeFormatter.ISO_LOCAL_DATE);  // 2014-03-18
```

날짜 파싱

문자열을 파싱해서 날짜 객체를 다시 만들 수 있다.

```
LocalDate date1 = LocalDate.parse("20140318", DateTimeFormatter.BASIC_ISO_DATE);
LocalDate date2 = LocalDate.parse("2014-03-18",
DateTimeFormatter.ISO_LOCAL_DATE);
```

주요 포맷터

포맷터	설명	예시
`ISO_LOCAL_DATE`	날짜만 (ISO-8601 기본 형식)	2025-10-24
`ISO_LOCAL_TIME`	시간만	13:45:30
`ISO_LOCAL_DATE_TIME`	날짜 + 시간	2025-10-24T13:45:30
`BASIC_ISO_DATE`	구분자 없는 ISO 형식	20251024

커스텀 패턴

`ofPattern()` 메소드로 직접 패턴을 지정할 수 있다.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
String formatted = LocalDateTime.now().format(formatter); // 2025/10/24 13:45:30
```

패턴 기호:

- `H` : 시(0-23)
- `h` : 시(1-12)
- `a` : 오전/오후

시간대 사용하기

ZoneId

기존 `java.util.TimeZone` (구버전)에서 `java.time.ZoneId` (신버전)로 변경되었다.

TimeZone vs ZoneId 비교

항목	TimeZone (구버전)	ZoneId (신버전)
타입	클래스	추상 클래스
가변성	변경 가능	불변 객체
스레드 안전성	안전하지 않음	스레드 안전
표준	Olson DB 표준	IANA Time Zone DB 표준
오프셋 표현	`int` 밀리초 단위 (`getRawOffset()`)	`ZoneOffset` 객체 (`+09:00`)
서머타임 처리	수동	자동 (`ZonedDateTime` 에서 자동 처리)
API 스타일	절차적	함수형, 불변식 설계

ZoneId 생성

표준 시간이 같은 지역을 묶어서 시간대 규칙 집합을 생성한다. `ZoneRules` 클래스는 약 40개 정도의 시간대를 가지며, `ZoneId` 의 `getRules()` 를 이용해 시간대의 규정을 획득할 수 있다.

```
ZoneId seoulZone = ZoneId.of("Asia/Seoul");
```

특정 지역 ID로 `ZoneId` 객체를 생성한다. 지역 ID는 `"지역/도시"` 형식이다.

기존 TimeZone을 ZoneId로 변환

`ZoneId` 의 새로운 메소드인 `toZoneId()` 로 기존의 `TimeZone` 객체를 `ZoneId` 객체로 변환할 수 있다.

```
ZoneId zoneId = TimeZone.getDefault().toZoneId();
```

ZonedDateTime

`ZoneId` 객체를 얻은 뒤 `LocalDate` , `LocalDateTime` , `Instant` 를 이용해 `ZonedDateTime` 인스턴스로 변환할 수 있다.

`ZonedDateTime` 은 지정한 시간대에 상대적인 시점을 표현한다.

```
ZoneId seoulZone = ZoneId.of("Asia/Seoul");

LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
ZonedDateTime zdt1 = date.atStartOfDay(seoulZone);

LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
ZonedDateTime zdt2 = dateTime.atZone(seoulZone);

Instant instant = Instant.now();
ZonedDateTime zdt3 = instant.atZone(seoulZone);
```

ZoneOffset: UTC/GMT 기준의 고정 오프셋

UTC/GMT 기준으로 시간대를 표현한다. `ZoneId`의 서브클래스인 `ZoneOffset` 클래스로 런던의 그리니치 0도 자오선과 시간 값의 차이를 표현할 수 있다.

ZoneId vs ZoneOffset

- `ZoneId` : 지역 기반 시간대 (서머타임 반영, 지역 기반)
- `ZoneOffset` : UTC 기준 고정 시차 (UTC 기준 고정 오프셋)

```
ZoneOffset seoulOffset = ZoneOffset.of("+09:00");  // 서울
ZoneOffset nyOffset = ZoneOffset.ofHours(-5);       // 뉴욕
```

실무 가이드: DB 저장 시 주의사항

날짜/시간 API 종류

레거시 API (Java 1.0 ~ 1.1)

`java.util.Date`, `java.util.Calendar`

문제점

- 가변 객체이며 스레드 안전하지 않다.
- 월이 0부터 시작한다.
- Timezone 혼합 및 직관성이 부족하다.

JDBC 호환 API (JDBC 1.0)

`java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`

문제점

- DB 연동용이지만 `java.util.Date`와 호환성이 혼란스럽다.

현대 API (Java 8+)

`LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Instant`

장점

- 불변 객체
- 명확한 의미
- 타임존 분리
- 포맷하기 쉽다

Java 타입과 DB 타입 매핑

Java 타입	의미	DB 매핑 타입
`LocalDate`	날짜 (YYYY-MM-DD)	`DATE`
`LocalTime`	시간 (HH:mm:ss)	`TIME`
`LocalDateTime`	날짜 + 시간	`TIMESTAMP` 또는 `DATETIME`
`ZonedDateTime`	날짜 + 시간 + 타임존	`TIMESTAMP WITH TIME ZONE` (DB가 지원해야 함)
`Instant`	UTC 기준 타임스탬프	`TIMESTAMP`

참고

- Hibernate 5.2+ 에서 `java.time.*` 자동 매핑을 지원한다. 별도의 `@Temporal` 이 필요 없다.
- MySQL에서는 `TIMESTAMP WITH TIME ZONE` 이 없어서 `DATETIME` 을 사용하고, UTC 변환은 애플리케이션 단에서 처리해야 한다.

DB 저장 시 포맷 주의사항

주요 포맷

ISO 8601 (기본)

```
2025-11-01T20:30:15
```

`LocalDateTime` 기본 문자열 형식

UTC 시간

```
2025-11-01T11:30:15Z
```

SQL 포맷

```
YYYY-MM-DD HH:MM:SS
```

SQL INSERT 시 일반 문자열로 들어간다.

일반적으로 JPA/Hibernate는 포맷 지정이 불필요하지만, JSON 직렬화(Jackson) 시에는 명시해야 한다.

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
private LocalDateTime createdAt;
```

시간대 처리 주의사항

문제 상황

서버 JVM이 `Asia/Seoul`이고 MySQL 설정이 `UTC`인 경우:

```
LocalDateTime.now(); // 현재 시간: 2025-11-01 20:00:00 (KST)
// DB에 저장 → 2025-11-01 11:00:00 (9시간 손실)
```

실제 시간이 20시인데 9시간이 손실된다.

원인 분석

`LocalDateTime`은 단순히 "2025년 11월 1일 20시 0분 0초"라는 날짜/시간만 가진다.

- 서울 20시인지, UTC 20시인지, LA 20시인지 알 수 없다.
- 시간대 정보(`ZoneId`)가 포함되지 않은 순수 시각이다.

MySQL 타입 특성

- `DATETIME` : 타임존 정보 없이 숫자 그대로 저장
- `TIMESTAMP` : 내부적으로 UTC로 변환해서 저장

Hibernate가 `LocalDateTime`을 `TIMESTAMP`로 매핑하면 문제가 발생한다.

Hibernate 변환 과정

1. `LocalDateTime` → `Instant`(UTC) 기준으로 변환 필요
2. `LocalDateTime`에는 타임존 정보 없음 → 기본 JVM 타임존 참조
3. `2025-11-01T20:00:00 (KST)` → UTC로 변환 → `2025-11-01T11:00:00Z`
4. MySQL에는 `TIMESTAMP` 타입으로 UTC 저장 → 내부에 다른 값이 들어감

해결 방법

1. Spring Boot 설정

```
spring:
  jackson:
    time-zone: UTC
  jpa:
    properties:
      hibernate:
        jdbc:
          time_zone: UTC
```

2. MySQL 연결 URL 보강

```
jdbc:mysql://localhost:3306/testdb?serverTimezone=UTC
```

3. DB 설계 시 타입 주의

Java 타입	DB 타입	용도
`Instant`	`TIMESTAMP`	UTC 절대 시각 저장
`LocalDateTime`	`DATETIME`	지역 시간 그대로 저장
`LocalDate`	`DATE`	날짜만 저장

4. JSON 직렬화 시 주의

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", timezone = "Asia/Seoul")
private LocalDateTime createdAt;
```

5. 테스트 환경 설정

항상 UTC로 맞춘다.

```
@BeforeAll
static void init() {
    TimeZone.setDefault(TimeZone.getTimeZone("UTC"));
}
```

테스트 머신의 OS 타임존 때문에 테스트 결과가 로컬 환경별로 달라질 수 있다.