



[Nueva guía] Descargue la Guía 2018 de IoT: aprovechamient...

[Descargar la guía▶](#)

# Fecha y hora de Java 8

por Steven Gentens · 29 y 18 de abril · Zona de Java · Tutorial

Descargue *Microservices for Java Developers* : una introducción práctica a frameworks y contenedores. Presentado en asociación con Red Hat .

Hoy en día, varias aplicaciones todavía usan las API `java.util.Date` y `java.util.Calendar` , incluidas las bibliotecas, para facilitarnos la vida al trabajar con estos tipos, por ejemplo, *JodaTime*. Java 8, sin embargo, introdujo nuevas API para manejar la fecha y la hora, lo que nos permite tener un control más preciso sobre nuestra representación de fecha y hora, dándonos objetos inmutables de fecha y hora, una API más fluida y en la mayoría de los casos un aumento del rendimiento sin usando bibliotecas adicionales. Echemos un vistazo a lo básico.

## LocalDate / LocalTime / LocalDateTime

Comencemos con las API que están más relacionadas con `java.util.Date` : *LocalDate* una API de fecha que representa una fecha sin tiempo; *LocalTime* , una representación de tiempo sin fecha; y *LocalDateTime* , que es una combinación de los dos anteriores. Todos estos tipos representan la fecha y / o hora local para una región, pero, al igual que `java.util.Date` , contienen información **cero** sobre la zona en la que está representada, solo una representación de la fecha y hora en su zona horaria actual.

En primer lugar, estas API admiten una instanciación sencilla:

```

1   LocalDate date = LocalDate . de ( 2018 , 2 ,
   // Utiliza DateTimeFormatter.ISO_LOCAL_DATE cuy
2   LocalDate date = LocalDate . análisis ( "201
   Hora LocalTime = Hora Local . de ( 6 , 30 );
   // Utiliza DateTimeFormatter.ISO_LOCAL_TIME cuy
6   // esto significa que tanto segundos como nanos
   Hora LocalTime = Hora Local . análisis ( "06
9   LocalDateTime dateTime = LocalDateTime . de
10  // Utiliza DateTimeFormatter.ISO_LOCAL_DATE_TIM
11  // combinación del formato de fecha y hora ISO,
12  LocalDateTime dateTime = LocalDateTime . ané
13

```

Es fácil convertir entre ellos:

```

1   // LocalDate a LocalDateTime
   LocalDateTime dateTime = LocalDate . análisis
2
3
4   // LocalTime a LocalDateTime
   LocalDateTime dateTime = LocalTime . análisis
5
6
7   // LocalDateTime a LocalDate / LocalTime
   LocalDate date = LocalDateTime . análisis (
8   LocalTime time = LocalDateTime . análisis (
9

```

Aparte de eso, es increíblemente fácil realizar operaciones en nuestras representaciones de fecha y hora, usando los métodos `plus` y `menos`, así como algunas funciones de utilidad:

```

1   LocalDate date = LocalDate . análisis ( "201
   LocalDate date = LocalDate . análisis ( "201

```

```

2
3
4      Hora LocalTime = Hora Local . análisis ( "00:00:00" );
5
6      LocalDateTime dateTime = LocalDateTime . of ( fecha , hora , LocalTime );
7
8      // usando TemporalAdjusters, que implementa algo para ajustar la fecha
9      LocalDate date = LocalDate . análisis ( "2018-02-28" );
10

```

Ahora, ¿cómo nos moveríamos de `java.util.Date` a `LocalDateTime` y sus variantes? Bueno, eso es simple: podemos convertir de un tipo Fecha al tipo Instantáneo, que es una representación del tiempo transcurrido desde la época del 1 de enero de 1970, y luego podemos crear una instancia `LocalDateTime` usando el Instantáneo y la zona actual.

```

1      LocalDateTime dateTime = LocalDateTime . ofInstant ( Instant.now() , ZoneOffset.UTC );

```

Para convertir nuevamente a una fecha, simplemente podemos usar el `Instant` que representa el tipo de tiempo Java 8. Una cosa que tomar nota de, sin embargo, es que si bien `LocalDate`, `LocalTime` y `LocalDateTime` no contienen ninguna zona o información de desplazamiento, que representan la fecha y / o hora local en una región específica, y como tales estén en posesión de la actual compensado en esa región. Por lo tanto, debemos proporcionar un desplazamiento para convertir correctamente el tipo específico a `Instant`.

```

1      // representa mié 28 feb 23:24:43 CET 2018
2      Fecha ahora = nueva Fecha ();
3
4      // representa 2018-02-28T23: 24: 43.106
5      LocalDateTime dateTime = LocalDateTime . ofInstant ( Instant.now() , ZoneOffset.UTC );
6
7      // representa mié 28 feb 23:24:43 CET 2018
8      Fecha fecha = Fecha . desde ( dateTime . toInstant() , ZoneOffset.UTC );

```

```

Fecha Fecha = Fecha . de ( dateTime . toInst
9

```

## Diferencia en el tiempo: duración y período

Como habrás notado, en uno de los ejemplos anteriores hemos usado un `Duration` objeto.

`Duration` y `Period` son dos representaciones de tiempo entre dos fechas, la primera representa la diferencia de tiempo en segundos y nanosegundos, la última en días, meses y años.

¿Cuándo deberías usar esto? `Period` cuando necesita saber la diferencia de tiempo entre dos `LocalDate` representaciones:

```

1 Período del período = Período . entre ( Local

```

`Duration` cuando busca una diferencia entre una representación que contiene información de tiempo:

```

1 Duración duración = Duración . entre ( Local

```

Al enviar `Period` o `Duration` usar `toString()`, se usará un formato especial basado en el estándar ISO-8601. El patrón utilizado para un `Período` es `PnYnMnD`, donde `n` define el número de años, meses o días presentes dentro del período. Esto significa que `P1Y2M3D` define un período de 1 año, 2 meses y 3 días. La 'P' en el patrón es el designador del período, que nos dice que el siguiente formato representa un período. Usando el patrón, también podemos crear un período basado en una cadena usando el `parse()` método.

```

1 // representa un período de 27 días
2 Período del período = Período . análisis ( "

```

Al usarlo `Durations`, nos alejamos ligeramente del estándar ISO-8601, ya que Java 8 no usa los mismos patrones. El patrón definido por ISO-8601 es

`PnYnMnDTnHnMn.nS`. Este es básicamente el `Period` patrón, extendido con una representación de tiempo. En el patrón, `T` es el designador de tiempo, por lo que la parte que sigue define una duración especificada en horas, minutos y segundos.

Java 8 usa dos patrones específicos para `Duration`, a saber, `PnDTnHnMn.nS` al analizar un `String` a un `Duration`, y `PTnHnMn.nS` cuando llama al `toString()` método en una `Duration` instancia.

Por último, pero no menos importante, también podemos recuperar las diversas partes de un período o duración, utilizando el método correspondiente en un tipo. Sin embargo, es importante saber que los distintos tipos de fecha y hora también lo respaldan mediante el uso del `ChronoUnit` tipo de enumeración. Veamos algunos ejemplos:

```

1 // representa PT664H28M
   Duración duración = Duración . entre ( Local
2
3
4 // devuelve 664
   largas horas = duración . toHours ();
5
6
7 // devuelve 664
   largas horas = LocalDateTime . análisis ( "2
8
```

## Trabajo con zonas y compensaciones: `ZonedDateTime` y `OffsetDateTime`

Hasta ahora, hemos demostrado cómo las nuevas API de fecha han facilitado un poco las cosas. Lo que realmente hace la diferencia, sin embargo, es la capacidad de usar fácilmente la fecha y la hora en un contexto de zona horaria. Java 8 nos proporciona `ZonedDateTime` y `OffsetDateTime`, el primero es un `LocalDateTime` con información para una zona específica (por ejemplo, Europa / París), el segundo es un `LocalDateTime` con un desplazamiento. ¿Cuál es la diferencia? `OffsetDateTime` utiliza una diferencia de

tiempo fija entre UTC / Greenwich y la fecha que se especifica, mientras que `ZonedDateTime` especifica la zona en la que se representa el tiempo, y tendrá en cuenta el horario de verano.

La conversión a cualquiera de estos tipos es muy fácil:

```

1  OffsetDateTime offsetDateTime = LocalDateTime
   // Utiliza DateTimeFormatter.ISO_OFFSET_DATE_TIME
2  // ISO_LOCAL_DATE_TIME seguido del desplazamiento
   OffsetDateTime offsetDateTime = OffsetDateTime
3  ZonedDateTime zonedDateTime = LocalDateTime
   // Utiliza DateTimeFormatter.ISO_ZONED_DATE_TIME
4  // ISO_OFFSET_DATE_TIME seguido de ZoneId entre
   ZonedDateTime zonedDateTime = ZonedDateTime
5  // tenga en cuenta que el desplazamiento no importa
6  // El siguiente ejemplo también devolverá un de
   ZonedDateTime zonedDateTime = ZonedDateTime.parse
7

```

When switching between them, you have to keep in mind that converting from a `ZonedDateTime` to `OffsetDateTime` will take daylight saving time into account, while converting in the other direction, from `OffsetDateTime` to `ZonedDateTime`, means you will not have information about the region of the zone, nor will there be any rules applied for daylight saving time. That is because an offset does not define any time zone rules, nor is it bound to a specific region.

```

1  ZonedDateTime winter = LocalDateTime.parse
   ZonedDateTime summer = LocalDateTime.parse
2  // offset será +01: 00
   OffsetDateTime offsetDateTime = invierno.to
3  // offset será +02: 00
   OffsetDateTime offsetDateTime = verano.to
4

```

```

7
8
OffsetDateTime offsetDateTime = zonedDateTime
9
10
OffsetDateTime offsetDateTime = LocalDateTime
11
ZonedDateTime zonedDateTime = offsetDateTime
12

```

Ahora, ¿y si nos gustaría saber cuál es el tiempo para una zona o compensación específica en nuestra propia zona horaria? ¡También hay algunas funciones útiles definidas para eso!

```

// timeInMacau representa 2018-02-14T13: 30 + 0
1
ZonedDateTime timeInMacau = LocalDateTime .
2
// timeInParis representa 2018-02-14T06: 30 + 0
3
ZonedDateTime timeInParis = timeInMacau . wi
4
5
OffsetDateTime offsetInMacau = LocalDateTime
6
OffsetDateTime offsetInParis = offsetInMacau
7

```

Sería una molestia si tuviéramos que convertir manualmente entre estos tipos todo el tiempo para obtener el que necesitamos. Aquí es donde Spring Framework viene en nuestra ayuda. Spring nos proporciona bastantes conversores de fecha y hora listos `ConversionRegistry` para usar , que están registrados en y se pueden encontrar en la `org.springframework.format.datetime.standard.DateTimeEConverters` clase.

Al utilizar estos convertidores, es importante saber que no convertirá el tiempo entre regiones o desplazamientos. El

`ZonedDateTimeToLocalDateTimeConverter` , por ejemplo, devolverá el `LocalDateTime` para la zona en la que se especificó, no el `LocalDateTime` que representaría en la región de su aplicación.

```

ZonedDateTime zonedDateTime = LocalDateTime
1

```

```
// representará 2018-01-14T06: 30, independent
2
LocalDateTime localDateTime = conversionServ
3
```

Por último, puede consultar

`ZoneId.getAvailableZoneIds()` para encontrar todas las zonas horarias disponibles o utilizar el mapa

`ZoneId.SHORT_IDS` , que contiene una versión abreviada de algunas zonas horarias, como EST, CST y más.

## Formateo: uso de `DateTimeFormatter`

Por supuesto, varias regiones del mundo usan diferentes formatos para especificar la hora. Una aplicación puede usar MM-dd-aaaa, mientras que otra usa dd / MM / aaaa. Algunas aplicaciones quieren eliminar toda confusión y representan sus fechas por aaaa-MM-dd. Cuando lo usemos

`java.util.Date` , nos moveremos rápidamente a usar formateadores múltiples. La `DateTimeFormatter` clase, sin embargo, nos proporciona patrones opcionales, de modo que podemos usar un solo formateador para varios formatos. Echemos un vistazo usando algunos ejemplos.

```
// Digamos que queremos convertir todos los pat
1
// 09-23-2018, 23/09/2018 y 2018-09-23 deben tc
2
DateTimeFormatter formatter = DateTimeFormat
3
LocalDate . parse ( "09-23-2018" , formateador
4
LocalDate . parse ( "23/09/2018" , formateador
5
LocalDate . parse ( "2018-09-23" , formateador
6
```

Los corchetes en un patrón definen una parte opcional en el patrón. Al hacer que nuestros diversos formatos sean opcionales, el primer patrón que coincida con la cadena se usará para convertir nuestra representación de fecha. Esto puede ser bastante difícil de leer cuando se utilizan múltiples patrones, así que echemos un vistazo a la creación de nuestro `DateTimeFormatter` patrón de construcción.



```

1      DateTimeFormatter formatter = new DateTimeF
2      . appendOptional ( DateTimeFormatter . ofPatter
3      . optionalStart (). appendPattern ( "dd / MM /
4      . optionalStart (). appendPattern ( "MM-dd-aaaa
5      . toFormatter ();

```

Estos son los conceptos básicos para incluir varios patrones, pero ¿qué pasa si nuestros patrones solo difieren ligeramente? Echemos un vistazo a aaaa-MM-dd y aaaa-MMM-dd.

```

1      // 2018-09-23 y 2018-Sep-23 deberían convertirs
2      // El uso del ejemplo de ofPattern que hemos us
3      DateTimeFormatter formatter = DateTimeFormat
4      LocalDate . parse ( "2018-09-23" , formateador
5      LocalDate . parse ( "2018-Sep-23" , formateador
6
7      // Usando el ejemplo ofPattern donde reutilizan
8      DateTimeFormatter formatter = DateTimeFormat
9      LocalDate . parse ( "2018-09-23" , formateador
10     LocalDate . parse ( "2018-Sep-23" , formateador

```

Sin embargo, no debe usar un formateador que admita formatos múltiples al convertir a una cadena, porque cuando usemos nuestro formateador para formatear nuestra fecha en una representación de cadena, también usará los patrones opcionales.

```

1      LocalDate date = LocalDate . análisis ( "201
2      // resultará en 2018-09-232018-Sep-23
3      fecha . formato ( DateTimeFormatter . ofPatterr
4      // dará como resultado 2018-09-23Sep-23
5      fecha . formato ( DateTimeFormatter . ofPatterr

```

Dado que estamos en el siglo XXI, obviamente tenemos que tomar en cuenta la globalización, y queremos ofrecer fechas localizadas para nuestros usuarios. Para asegurarse de que

`DateTimeFormatter` regrese a una configuración regional específica, simplemente puede hacer lo siguiente:

```

1  DateTimeFormatter formatter = DateTimeFormat
2
3  DateTimeFormatter formatter = new DateTimeF

```

Para encontrar qué configuraciones regionales están disponibles, puede usar

`Locale.getAvailableLocales()` .

Ahora bien, es posible que el patrón de fecha que reciba contenga más información que el tipo que está utilizando. A `DateTimeFormatter` arrojará una excepción tan pronto como una representación de fecha proporcionada no esté de acuerdo con el patrón. Echemos un vistazo más de cerca al problema y cómo solucionarlo.

```

// El problema: esto generará una excepción.
1  LocalDate date = LocalDate . análisis ( "201
2
// Proporcionamos un DateTimeFormatter que puec
3
// El resultado será una retención LocalDate 20
4
LocalDate date = LocalDate . parse ( "2018-0
5

```

Vamos a crear un formateador que pueda manejar la fecha ISO, la hora y los patrones de fecha y hora.

```

1  DateTimeFormatter formatter = new DateTimeF
2
3  . appendOptional ( DateTimeFormatter . ISO_LOCA
4
5  . optionalStart (). appendLiteral ( "T" ). opci
6
7  . appendOptional ( DateTimeFormatter . ISO_LOCA
8
9

```

```

4
5 . toFormatter ();

```

Ahora podemos ejecutar perfectamente todo lo siguiente:

```

1 // resultados en 2018-03-16
2 LocalDate date = LocalDate . parse ( "2018-03-16" );
3
4 // resultados en 06:30
5 Hora LocalTime = Hora Local . parse ( "2018-03-16T06:30" );
6
7 LocalDateTime localDateTime = LocalDateTime . of ( date, localTime );

```

LocalDateTime? ¿Qué sucede si espera un LocalTime y le dan una representación de fecha o viceversa?

```

1 // lanzará una excepción
2 LocalDateTime localDateTime = LocalDateTime . of ( "2018-03-16", "06:30" );
3
4 LocalDate localDate = LocalDate . parse ( "2018-03-16" );
5
6 LocalTime localTime = LocalTime . parse ( "06:30" );

```

En estos dos últimos casos, no existe una única solución correcta, pero depende de lo que requiera o de lo que esas fechas y horas representen o puedan representar. La magia se encuentra en el uso de `TemporalQuery`, que puede usar para crear valores predeterminados para una parte del patrón.

Si comenzamos con a `LocalDateTime`, y solo desea la `LocalDate` o `LocalTime`, recibirá la parte correspondiente de `LocalDateTime`. Para crear un `LocalDateTime`, necesitaremos valores predeterminados para la fecha y hora que se mantiene. Digamos que si no proporciona información sobre una fecha, le devolveremos la fecha de hoy y, si no proporciona un horario, asumiremos que se refería al inicio del día.

Ya que estamos devolviendo a `LocalDateTime`, no se analizará con una `LocalDate` o `LocalTime`, así que

vamos a usar la `ConversionService` para obtener el tipo correcto.

```

TemporalQuery < TemporalAccessor > myCustomQue
1
// resultados en 2018-03-16
LocalDateTime localDateTime = conversionServ
2
// resultados en 00:00
LocalTime localTime = conversionService . cc
3
4
5
6
clase MyCustomTemporalQuery implementa Tempc
7
{
8
    @Anular
9
    public TemporalAccessor queryFrom ( Tempc
10
        LocalDate date = temporal . isSupport
11
        ? LocalDate . ofEpochDay ( temporal .
12
        LocalTime time = temporal . isSupport
13
        ? LocalTime . ofNanoOfDay ( temporal .
14
        devolver LocalDateTime . de ( fecha ,
15
16
17
    }
}

```

El uso `TemporalQuery` nos permite verificar qué información está presente y proporcionar los valores predeterminados para cualquier información que falta, lo que nos permite convertir fácilmente al tipo requerido, utilizando la lógica que tiene sentido en nuestra aplicación.

Para aprender a redactar patrones de tiempo válidos, consulte la `DateTimeFormatter` documentación .

## Conclusión

La mayoría de las características nuevas requieren algo de tiempo para comprenderlas y acostumbrarse a ellas, y la API de fecha / hora de Java 8 no es diferente. Las API nos proporcionan un mejor acceso al formato correcto necesario, así como a una manera más estandarizada y legible de trabajar con operaciones de fecha y hora. Usando estos consejos y


operaciones de fecha y hora. Usando estos consejos y trucos, podemos cubrir prácticamente todos nuestros casos de uso.

---

Descargar Building Reactive Microservices en Java : diseño de aplicaciones asíncronas y basadas en eventos. Presentado en asociación con Red Hat .

---

Temas: JAVA 8, API DE TIEMPO, FECHA Y HORA, JAVA, TUTORIAL

Publicado en DZone con el permiso de Steven Gentens. [Vea el artículo original aquí.](#)   
Las opiniones expresadas por los contribuidores de DZone son suyas.

## Recursos para socios de Java

---

Migrar a bases de datos de Microservicio

Programa de desarrollo de Red Hat



Comience con Spring Security 5.0 y OpenID Connect (OIDC)

Okta



Diseño de sistemas reactivos: el papel de los actores en la arquitectura distribuida

Lightbend



Profundo conocimiento de su código con IntelliJ IDEA.

JetBrains

