KAFFEEKLATSCH

Das Magazin rund um Software-Entwicklung

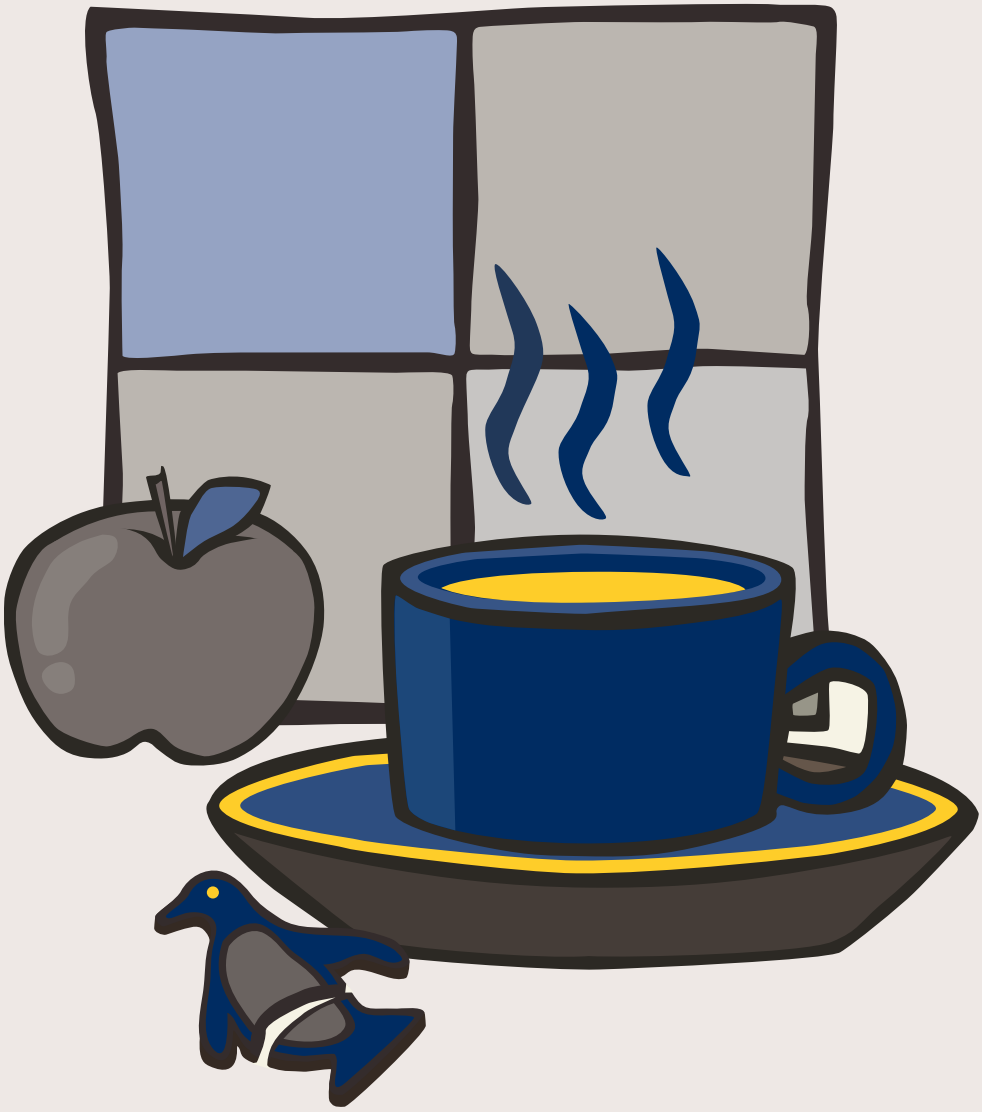
ISSN 1865-682X

10/2017

Unter Beobachtung

Events in CDI

von Werner Eberling



Unter Beobachtung

Events in CDI

von Werner Eberling

Mit der im September 2017 veröffentlichten Java-EE-Version 8 wurde auch ein neues Major Release des CDI-Standards bereitgestellt. Dieses beinhaltet unter anderem die lange erwarteten asynchronen Events. Grund genug, sich einmal genauer mit dem Thema Eventing in CDI zu beschäftigen. Wie werden Events erzeugt, wie empfangen und welche Möglichkeiten zum „Feintuning" gibt es dabei?

Dependency Injection und Contextual Scopes, das sind die Funktionalitäten, die den meisten Entwicklern in den Sinn kommen, wenn sie an den gleichnamigen Java-EE-Standard „Contexts and Dependency Injection (CDI)" denken. Doch dieser hat deutlich mehr zu bieten. Unter anderem auch eine sehr leichtgewichtige Implementierung des Observer Patterns [1] als integraler Bestandteil des Standards.

Mit der neuen Major-Version [2] des CDI-Standards, die seit September 2017 im Rahmen der Java EE 8 verfügbar ist, wurde auch im Bereich der Events noch einmal nachgelegt. Eine gute Gelegenheit, dieses, oft wenig beachtete, CDI-Feature hier einmal genauer vorzustellen.

Events erzeugen

Anders als in den „klassischen" Implementierungen des Observer Patterns, ist es bei der Nutzung von CDI-Events nicht notwendig für die Event-Zustellung besondere Listener-Interfaces zu definieren. Da das Pattern elementarer Bestandteil des Standards ist, sind alle benötigten Typen direkt über die API verfügbar.

Zur Erzeugung von Events ist dies im besonderen das Interface *javax.enterprise.event.Event<T>*. Über dieses Interface kann eine CDI-Komponente Events eines von ihr definierten Typs erzeugen. Der Typ-Parameter definiert dabei den konkreten Typ des zu versendenden Events, der durch eine beliebige Java-Klasse realisiert sein kann.

Um konkrete Events zu erzeugen, reicht es aus, sich das Event-Interface mit dem entsprechenden Typ-Parameter injizieren zu lassen.

```
@Inject
private Event<User> userEvent;
```

Anschliessend kann die Referenz dazu genutzt werden, um das Objekt der Wahl als Event zu feuern.

```
User userLoggedIn = new User(username);
userEvent.fire(userLoggedIn);
```

Der so erzeugte Event wird synchron zugestellt und verarbeitet. Die Abarbeitung ist also beendet, bevor die nächste Codezeile des Event-Erzeugers ausgeführt wird. Dies ist speziell bei der Propagierung von Exceptions entscheidend, die während der Abarbeitung des Events auftreten. Diese schlagen – zur Laufzeit – in Folge des *fire()*-Aufrufs direkt beim Erzeuger auf.

Eine asynchrone Zustellung des Events war im CDI-Standard bisher nicht möglich. Dies hat sich mit der aktuellen Version 2.0 geändert. Über die Methode *fireAsync()* lassen sich Events nun auch asynchron propagieren.

```
userEvent.fireAsync(userLoggedIn);
```

Wie und wann die Abarbeitung des Events im asynchronen Fall erfolgt, ist der Container-Implementierung überlassen. Es wird lediglich garantiert, dass dies nicht im Rahmen des Threads geschieht, der den Event erzeugt hat.

Sollte bei der Abarbeitung des Events eine Exception auftreten, so kann auf diese nur über die *CompletionStage* reagiert werden, die vom Methodenaufruf zurückgegeben wird. Konzeptionell sollte dies aber meist nicht notwendig sein, da in einem solchen Fall eine synchrone Verarbeitung in der Regel das bessere Mittel der Wahl gewesen wäre.[1]

1 Die genannte CompletionStage kann auch dazu genutzt werden, das Ergebnis der asynchronen Abarbeitung im Nachgang noch weiterzuverarbeiten. Eine Vorgehensweise die einen eigenen Artikel wert ist und daher hier nicht weiter beleuchtet wird.

Events empfangen

Um auf geworfene Events zu reagieren, reicht es aus, eine Methode bereitzustellen, die den Typ des Events als Parameter erwartet und diesen mit *@Observes* zu annotieren.

```
public void auditLogin(@Observes User user) {
  …
}
```

Auch hier wird explizit zwischen synchronem und asynchronem Empfang unterschieden. Während *@Observes* für den synchronen Empfang verwendet wird, so muss für den asynchronen die Annotation *@ObservesAsync* verwendet werden.

```
public void asyncAuditLogin(
  @ObservesAsync User user) {
  …
}
```

Diese explizite Unterscheidung von synchroner und asynchroner Event-Propagierung hat mehrere Gründe.

Einerseits ist die asynchrone Variante eine Neuerung in Version CDI 2.0. Würden die „alten" Annotationen nun nicht mehr synchron, sondern auf einmal asynchron funktionieren, so wäre dies eine semantisch inkompatible Änderung, die das Verhalten vieler bestehender CDI-Anwendungen ungewollt verändern würde. Eine solche Änderung wäre für eine *public API* also absolut nicht vertretbar.

Auf der anderen Seite ist der Kreis der Empfänger im synchronen und asynchronen Fall unterschiedlich! Werden Events synchron geworfen, so werden alle Observer benachrichtigt, die in einem der aktiven Scopes des Threads existieren, der den Event erzeugt. So werden beispielsweise alle passenden Observer innerhalb der selben Session benachrichtigt, aber kein Observer in parallel existierenden Sessions, auch wenn sie auf Events des selben Typs registriert wurden.

Im asynchronen Fall ist der Kreis der möglichen Empfänger deutlich kleiner. Hier werden lediglich Observer im ApplicationScope und im RequestScope des Event erzeugenden Threads benachrichtigt.

Mehr als ein Empfänger

Ist mehr als ein Observer für den Empfang eines Event definiert, so ist die Reihenfolge des Aufrufs der einzelnen Instanzen nicht definiert. Ist eine bestimmte Reihenfolge im Empfang eines Events notwendig, so können die Observer über die Annotation *@Priority* in eine relative Reihenfolge gebracht werden. Ein niedrigerer Wert steht dabei in der Aufrufreihenfolge vor einem höheren Wert.

```
public void auditLogin(
  @Observes @Priority(100) User user) {
    …
}
```

Die Sortierung über Prioritäten ist bereits aus dem Bereich der CDI-Interceptoren bekannt. In diesem Zusammenhang erscheint eine vordefinierte Reihenfolge unter gewissen Umständen auch tatsächlich als sinnvoll. Im Zusammenhang mit dem Empfang und der Abarbeitung von Events ist der Einsatz einer solchen Priorisierung allerdings stark zu hinterfragen. Hier widersprechen sich die Idee der Entkopplung per Event und die scheinbare Notwendigkeit einer festgelegten Reihenfolge in deren Abarbeitung aus Designsicht doch erheblich.

Qualifizierung

So einfach kann es sein, in CDI Events zu feuern und zu verarbeiten. Im Detail sind die Anforderungen meist natürlich etwas höher. Hier reicht es nicht aus, einfach Benutzer, Kundenobjekte oder ähnliches durch die Gegend zu werfen und zu hoffen, dass jeder weiss, was dies semantisch bedeutet soll. Hier ist eine genauere Qualifizierung der Events notwendig.

Dies könnte durch die Implementierung spezieller Event-Klassen gelöst werden, die die jeweiligen Objekte kapseln und damit an die „klassischen" Event-Klassen aus z. B. Java Swing erinnern. Dieser Ansatz riecht einerseits deutlich nach Boilerplate-Code, andererseits müsste für jeden konkreten Event-Typ ein entsprechendes Event-Objekt injiziert werden.

Das erste Problem kann durch ein weiteres CDI-Konzept gelöst werden, dem Einsatz von Qualifiern. Diese Qualifier beschreiben einen Typ per Annotation näher und ermöglichen so eine genauere Unterscheidung. So könnte der User-Event über zwei, selbst definierte Qualifier *@LoggedIn* und *@LoggedOut* genauer qualifiziert werden.

```
@Inject @LoggedIn
private Event<User> userLoginEvent;

@Inject @LoggedIn
private Event<User> userLogoutEvent;
```

Durch diese Unterscheidung kann ein Observer nun entscheiden, ob er sich für *LogIn-Events* oder *LogOut-Events* interessiert. Bei geschickter Namenswahl unterstützen die Qualifier dabei auch die Lesbarkeit der Methodensignatur im Observer.

```
public void auditLogin(
  @Observes @LoggedIn User user) {
  …
}
```

Einen Beigeschmack hat die gewählte Lösung allerdings. Es muss für jedes qualifizierte Event ein eigener Injectionpoint definiert werden. Dies ist für „statische" Qualifier, die für Events gelten, die an verschiedenen Stellen im Code geworfen werden, sicher sinnvoll. Für spezielle Konkretisierungen, die nur in besonderen Situationen vorkommen, bietet sich aber ein anderer Weg an, die dynamische Erzeugung von spezialisierten Events mit Hilfe von Qualifier-Literalen.

Dynamik durch Annotations-Literale

Soll, ausgehend von einem allgemeinen Event, ein genauer qualifizierter Event erzeugt werden, so kann dieser über die Methode *Event.select(...)* erzeugt werden. Hierbei können sowohl die Eventklasse spezialisert als auch ein oder mehrere zusätzliche Qualifierer dynamisch hinzugefügt werden.

```
@Inject
private Event<User> userLogoutEvent;

private void fireLogInEvent(User user){
  // Entspricht
  // @LoggedIn Event<User>.fire(user)
  userLogoutEvent.select(
    new LoggedInQualifier()
  ).fire(user);
}
```

Der hier verwendete *LoggedInQualifier* nutzt dabei die Möglichkeit zur Inline-Erzeugung von Annotationen über AnnotationsLiterale.

```
public class LoggedInQualifier extends
  AnnotationLiteral<LoggedIn> { … }
```

Werden diese Literale als Inner Classes der jeweiligen Annotation definiert, kommt hier auch wieder zusammen, was zusammen gehört und am Ende steht eine schön lesbare Event-Selektion[2].

```
@Retention(RetentionPolicy.RUNTIME)
@Target({...} )
@Qualifier
public @interface LoggedIn {
  public static final class Literal extends
    AnnotationLiteral<LoggedIn> {}
}

private void fireLogInEvent(User user){
  // Entspricht
  // @LoggedIn Event<User>.fire(user)
  userLogoutEvent.select(
    new LoggedIn.Literal()
  ).fire(user);
}
```

Events und Transaktionen

Ein letztes Detail, das nicht immer direkt ins Auge springt, ist das Zusammenspiel zwischen Eventing und Transaktionssteuerung.

Was hilft es dem Empfänger eines Events über die Erzeugung eines neuen Kunden in der Datenbank informiert zu werden, wenn die Transaktion, in deren Rahmen die Erzeugung stattfindet, im Anschluss zurückgerollt wird? Muss der Empfänger nun selbst seinen Ursprungszustand wiederherstellen?

Nicht, wenn der Empfang des Events auf das Commit der Transaktion verschoben wird! Dadurch wird der Empfänger tatsächlich erst dann benachrichtigt, wenn der entsprechende Datensatz in der Datenbank angelegt wurde.

Ob sich ein (synchroner) Event-Empfänger direkt benachrichtigen lassen möchte, oder erst dann, wenn die entsprechende Transaktion beendet wurde, kann über das Attribut during der Annotation *@Observes* angegeben werden. Hier kann der Empfänger entscheiden, in welcher Phase der Transaktion er den Event empfangen möchte: zum Beispiel am Ende der Transaktion (AFTER_COMPLETION), nur bei einem erfolgreichen Commit (AFTER_SUCCESS) oder im Falle eines Rollbacks (AFTER_FAILURE). Wird kein konkretes Verhalten vorgegeben, so erfolgt die Zustellung des Events direkt ohne Rücksicht

2) Dieser Ansatz wird auch vom CDI-Standard selbst genutzt, um ab Version 2.0 Literale für die gängigen Standardannotationen bereitzustellen (Stichwort: „Built-in annotation literals").

auf den weiteren Verlauf der Transaktion (IN_PROGRESS). Die Abarbeitung wird dabei Teil der beim Feuern des Events aktiven Transaktion.

```
public void neuerKundeErzeugt(
   @Observes(
      during=TransactionPhase.AFTER_SUCCESS)
   KUNDE kunde){ ... }

public void keinKundeErzeugt(
   @Observes(
     during=TransactionPhase.AFTER_FAILURE)
   KUNDE kunde){ ... }
```

Die Verbindung von Transaktion und Events-Zustellung ist nur im synchronen Fall möglich. Asynchrone Observer laufen, aus konzeptionellen Gründen, immer unabhängig von der Transaktion des Event-Erzeugers.

Fazit

Mit dem integrierten Event-Mechanismus bietet der CDI-Standard eine leichtgewichtige Möglichkeit zur Realisierung des Observer-Patterns. Durch die Integration in die Transaktionssteuerung und nicht zuletzt durch die Unterstützung asynchroner Events ab Version 2.0 hat der geneigte CDI-Entwickler hier ein mächtiges Mittel in der Hand, um mit wenigen Zeilen Code zu schön entkoppelten Anwendungen zu gelangen.

Und Dank des Supports von CDI auch in Java-SE-Umgebungen braucht es hierzu noch nicht einmal einen Application-Server ...

Referenzen

[1] OBSERVER PATTERN *A Gang of Four design pattern*
http://www.blackwasp.co.uk/Observer.aspx

[2] CDI 2.0 *Final Release*
https://www.jcp.org/en/jsr/detail?id=365

Kurzbiografie

Werner Eberling (@Wer_Eb) ist als Principal Consultant und Trainer für die MATHEMA Software GmbH tätig. Seit 1999 arbeitet er im Bereich verteilter Systeme auf der Basis von CORBA und der Java EE. Mit CDI befasst er sich seit den Anfängen des Standards im Jahre 2008 (damals noch unter dem Namen „WebBeans"). Sein aktuelles Steckenpferd sind Anwendungen für mobile Endgeräte und deren Anbindung an Enterprise-Systeme. Neben seiner Projekttätigkeit hält er Vorträge und Technologie-Trainings in Deutschland und der Schweiz. Er ist Autor eines Buches zu Enterprise JavaBeans (EJB), das im Hanser Verlag erschienen ist.