Event Listener

Plugins können sich beim Framework für Events registrieren und selbst Events auslösen. Ein Event besteht immer aus einem Namen sowie zugehörigen Daten.

Name?

Der Name muss gegenüber anderen Events unique sein. Anhand dessen erfolgt das Routing der Events an die interessierten Event-Listener. Es sollte daher ein kontext-spezifischer Präfix gewählt werden, ähnlich den Java-Packages. Das Framework selbst benutzt Namen wie `user.update.backend` und `login.backend.fail`. Für eigene Events wird das Schema `my-company.my-project.my-event` empfohlen.

Daten?

Jedes ausgelöste Event enthält spezielle, relevante Daten. Diese werden in einer Instanz einer Klasse transportiert, die von `com.batix.event.EventData` abgeleitet ist. Diese Basisklasse bringt bereits Felder für `request`, `response` und `application` mit. Die abgeleitete Klasse kann beliebige Felder und Methoden hinzufügen. Die Methode `getEventName` muss in jedem Fall implementiert werden:

```
class MyEventData : EventData() {
  override fun getEventName(): String {
    return "example.tool.test-event"
  }
}
```

Der Rückgabewert von `getEventName` ist der Name des ausgelösten Events. Da es möglich ist, dass sich derselbe Event-Listener für unterschiedliche Events (mit kompatiblen Daten) registriert, kann er hiermit unterschieden, welches Event genau ausgelöst wurde.

> **TIP**
> Werden keine zusätzlichen Daten benötigt, kann die Klasse `com.batix.event.StandardEventData` benutzt werden. Dieser ist dann nur der Event-Name zu übergeben.

`EventManager`?

Der `EventManager` ist das Herzstück der Event-Maschinerie. Mittels seiner Methode `fireEvent` können Events ausgelöst werden.

> **Achtung**
> Die anderen Methoden wie `on` und `off` sollten von Plugins nicht direkt benutzt werden, da es sonst ggf. zu Memory-Leaks kommen kann.

Um ein Event auszulösen, reicht die Übergabe der Daten:

```
val data = MyEventData()
EventManager.fireEvent(data)
```

Es gibt auch Überladungen von `fireEvent`, die noch weitere Daten wie z. B. `request` und `response` entgegennehmen. Sind diese verfügbar, kann man sie mit übergeben, um im Event-Listener darauf zuzugreifen. Sie werden dann automatisch in der `EventData` Grundklasse der Instanz vermerkt.

`EventListener`?

Um Events zu empfangen, wird eine Instanz von `com.batix.event.EventListener` benötigt. Dazu definiert man zunächst eine Klasse, welche die entsprechenden Daten verarbeiten kann:

```
class MyEventListener : EventListener<MyEventData>() {
  override fun syncCallback(
    eventName: String,
    info: MyEventData,
    application: ServletContext?,
    request: HttpServletRequest?,
    response: HttpServletResponse?,
  ): MyEventData? {
    // ...

    return null
  }
}
```

Wie der Name `syncCallback` vermuten lässt, wird diese Methode bei der Abarbeitung des Events synchron aufgerufen. Es wird also gewartet, bis `syncCallback` fertig ist, bevor das Event dem ggf. nächsten Listener übergeben wird. Deshalb sollten in dieser Methode auch keine Sachen ablaufen, die längere Zeit benötigen.

Um dennoch Callbacks abzuarbeiten, die länger dauern, unterstützt der `EventManager` auch asynchrone Callbacks. Über den Rückgabewert von `syncCallback` wird gesteuert, ob ein späterer, asynchroner Aufruf gewünscht ist, oder nicht. Wird `null` zurück gegeben, erfolgt kein späterer Aufruf. Ansonsten gibt man die erhaltenen Daten zurück und ein Worker ruft so bald wie möglich das asynchrone Callback auf. Man kann auch beides kombinieren: z. B. eine einfache Verarbeitung wie Logging synchron und das Wegspeichern von Daten asynchron.

Um Events asynchron verarbeiten zu können, muss nebst `syncCallback` noch die Methode `asyncCallback` implementiert werden:

```
class MyEventListener : EventListener<MyEventData>() {
  override fun syncCallback(
    eventName: String,
    info: MyEventData,
    application: ServletContext?,
    request: HttpServletRequest?,
    response: HttpServletResponse?,
  ): MyEventData? {
    if (canBeDoneQuick) {
      doStuff()
      return null
    }

    return info
  }

  override fun asyncCallback(state: MyEventData) {
    // ...
  }
}
```

Je nachdem, welche Daten bei `fireEvent` übergeben wurden, sind `application`, `request` und `response` sowohl als Parameter als auch in den Event-Daten verfügbar, oder nicht.

> Zu beachten ist, dass in `asyncCallback` der Request höchstwahrscheinlich schon vorbei ist. Daher müssen hier Zugriffe auf `request` und `response` vermieden werden.

`registerForEvent`?

Ein Plugin kann sich mittels `registerForEvent(eventName, listener)` für ein Event anmelden. Dies kann z. B. in seiner `load()` Methode geschehen.

Wie bei den anderen Extensions auch, kann das Plugin zu seiner Laufzeit beliebig Event-Listener an- und abmelden. Ein bestimmter Listener kann mittels `unregisterForEvent(eventName, listener)` deregistriert werden. Alle Listener des Plugins für ein bestimmtes Event können mit `unregisterForEvent(eventName)` abgemeldet werden. Um alle Listener des Plugins abzumelden, genügt der Aufruf `unregisterForAllEvents()` (dies wird automatisch auch beim Entladen des Plugins getan, man muss seine Listener also nicht manuell entfernen).

```
class MyPlugin : Plugin() {
  override fun load() {
    registerForEvent("example.tool.test-event", MyEventListener())
  }
}
```

Revision #1
Created 9 May 2025 06:19:11 by Batix
Updated 9 May 2025 06:19:11 by Batix