# Extensions

- Service
- Action
- Tag
- Timer Job
- Request Interceptor
- Static Content
- Admin Page
- WebSocket
- Event Listener

# Service

Ein Service ist die generischste Schnittstelle, die ein Plugin bereitstellen kann, denn es wird eine beliebige Anzahl Parameter beliebigen Typs entgegengenommen und ein Objekt beliebigen Typs zurückgegeben.

```
fun registerService(serviceName: String, service: Service)
```

`serviceName` ist eine frei wählbare ID, die über alle Services eines Plugins hinweg eindeutig sein muss. Es ist kein Problem, wenn mehrere Plugins einen Service mit derselben ID bereitstellen, da zum Ansprechen eines Services auch die `id` des Plugins herangezogen wird.

Ein Service muss `com.batix.plugins.Service` implementieren. Hier gibt es nur eine Methode, `call`. Diese Methode nimmt eine beliebige Anzahl an Argumenten vom Typ `Any?` (entspricht `Object` in Java) entgegen und gibt ein Objekt vom Typ `Any?` zurück (kann auch `null` sein).

Es liegt an der Implementierung selbst, herauszufinden, wie viele Argumente übergeben wurden, welchen Typ diese haben, dementsprechend Code auszuführen und ein Ergebnis zurückzugeben. Nutzern dieses Services sollte in einer Anleitung die `id` des Plugins und des Services sowie die Semantik mitgeteilt werden, also wie sich die Methode bei welchen Parametern verhält.

> **Typen**
> Die Typen der Parameter und des zurückgegebenen Objektes (und jeweils eventueller weiterer, darin eingebetteter Typen – wie bei Listen oder Maps) sollte sich auf **Standard-JVM- und Framework-Typen** wie beispielsweise `String` oder `ContainerRecord` beschränken.
>
> Das hat den Hintergrund, dass die Klassen des Plugins und seiner Dependencies nicht im Framework und anderen Plugins bekannt sind. Außerdem hilft es Leaks zu vermeiden, wenn das Plugin entladen wird.

Mithilfe der Klasse `com.batix.plugins.Plugin` kann auf einen Service zugegriffen werden. Dafür holt man sich zunächst mittels der statischen Methode `byId` eine Referenz auf das Plugin und dann davon weiter eine Referenz auf den Service via `getService`. Beide Referenzen können befragt werden, ob das Plugin geladen bzw. der Service verfügbar ist.

Die Service-Referenz stellt eine `call` Methode (blockiert den aktuellen Thread) sowie eine asynchrone `callAsync` Methode (gibt ein `Future` Objekt zurück) bereit. Beide Methoden liefern eine Exception, falls das Plugin oder der Service nicht verfügbar ist. Eine andere Möglichkeit stellen die `tryCall` und `tryCallAsync` Methoden bereit. Diese werfen keine Exception, wenn etwas nicht verfügbar ist, sondern haben dann als Ergebnis das Objekt

`com.batix.plugins.PluginRef.ServiceRef.UNAVAILABLE`.

# Beispiel

Die `Service`-Klasse im Beispiel besteht nur aus einer Methode, die eine Anzahl von `String`-Parametern erwartet und einen `String` zurückgibt (je nach Anzahl der Parameter einen anderen).

```
import com.batix.plugins.Service

class HelloService : Service {
  override fun call(vararg args: Any?): Any {
    return when {
      args.isEmpty()   -> "Hello."
      args.size == 1   -> "Hi ${args[0]}!"
      else             -> "Welcome ${args.joinToString(separator = ", ")}."
    }
  }
}
```

In der Plugin-Hauptklasse wird der Service registriert.

```
override fun load() {
  registerService(
 "hello-service"
, HelloService())
}
```

Aufrufen kann man diesen Service dann z. B. mittels Groovy-Code im Framework.

```
import com.batix.plugins.Plugin
import com.batix.plugins.PluginRef

def service = Plugin.byId("com.batix.website:mitarbeiter-import")
  .getService("hello-service")

println(service.call())                        // "Hello."
println(service.call("John"))                  // "Hi John!"
println(service.call("Joe", "Jack", "Jill"))   // "Welcome Joe, Jack, Jill."
```

# Action

Plugins haben die Möglichkeit, Actionbausteine bereitzustellen, die ganz normal in Menüpunkt-Aktionen im Framework verwendet werden können. Plugin-Actionbausteine stehen dann dort (gruppiert nach Plugin) genau wie die Standard-Bausteine zur Auswahl.

```
fun registerAction(actionId: String, actionInfo: ActionInfo)
```

`actionId` ist eine frei wählbare ID, die über alle Actionbausteine eines Plugins hinweg eindeutig sein muss. Es ist kein Problem, wenn mehrere Plugins einen Baustein mit derselben ID bereitstellen, da zum Ansprechen eines Actionbausteins auch die `id` des Plugins herangezogen wird.

`actionInfo` enthält die Metadaten des Actionbausteins, wie Titel und Beschreibung. Es kann auch ein Link zu einer externen Hilfeseite definiert werden. Außerdem wird die Klasse referenziert, welche den Baustein implementiert.

Diese Klasse muss von `com.batix.plugins.PluginAction` abgeleitet sein und die `doAction()` Methode implementieren. Jeder Aufruf einer Aktion erzeugt für jeden Actionbaustein eine eigene Instanz der entsprechenden Klasse. Innerhalb von `doAction()` stehen dann z. B. `request` und `response` zur Verfügung.

# Parameter / Properties

Damit der Baustein in der Verwaltung konfiguriert werden kann, müssen dessen Parameter definiert werden. Diese werden in `actionInfo` hinterlegt und können dann in `doAction()` ausgelesen werden.

Es können verschiedenartige Parameter hinterlegt werden, so gibt es z. B. die Methoden `addTextParameter` für Texteingaben oder `addBooleanParameter` für eine Ja/Nein-Auswahl.

```
fun addTextParameter(
  name: String,
  title: String,
  description: String,
  descriptionIsHtml: Boolean,
  required: Boolean
)
```

`name` ist ein interner Bezeichner, der dann auch wieder beim Auslesen in `doAction` angegeben werden muss.

> **Tip**
> Es empfiehlt sich, diesen internen Wert als Konstante im Code zu definieren, da der Wert an mehreren Stellen im Code (Parameterdefinition und Auslesen) benötigt wird.

Mit `title` kann für die Anzeige im Backend ein freundlicher Name gesetzt werden. `description` ist ein Erklärungstext, der unter dem Titel angezeigt wird – `descriptionIsHtml` legt fest, ob dieser bereits als HTML formatiert ist oder nicht. Falls `required` gesetzt ist, wird dieser Parameter in der Verwaltung als Pflichteingabe markiert.

In der Action kann dann mit `getProperty`, `getPropertyReplaced` oder `getBooleanProperty` der vom Benutzer eingestellte Wert abgefragt werden.

```
fun getProperty(key: String): String?
fun getProperty(key: String, defaultValue: String?): String?

fun getPropertyReplaced(key: String): String?
fun getPropertyReplaced(key: String, defaultValue: String?): String?

fun getBooleanProperty(key: String): Boolean
```

`getPropertyReplaced` ersetzt Platzhalter in der Form [[paramname]] durch den Wert des Request-Parameters oder Actionscript-Attributs `paramname`.

Mittels `addCustomParameter` kann sogar komplett eigener HTML-Code zur Benutzereingabe geliefert werden. Dazu leitet man am besten eine Klasse von `com.batix.action.ExtendedActionParameter` ab und implementiert im Minimum die folgenden Methoden:

- `getName()` – liefert den internen Bezeichner zurück
- `getTitle()` – gibt den Titel zurück
- `writeAdminView(StringBuffer, String, Connection)`
  - in den `StringBuffer` schreibt man den HTML-Quelltext, der in der Verwaltung angezeigt werden soll
  - der `String` ist der aktuell gespeicherte Wert des Parameters
  - `Connection` ist eine Datenbankverbindung

Wichtig hierbei ist, dass `writeAdminView` ein HTML-Formular-Element mit passendem `name` erzeugt (Prefix `param:` und dann der interne Bezeichner), damit die Eingabe auch gespeichert wird.

# Beispiel

Das Beispiel-Action definiert 3 Parameter: einen vom Typ Text, eine Ja/Nein-Auswahl und einen speziellen Parameter. Die Action-Klasse definiert die internen Bezeichner der Parameter als Konstanten. In der `doAction`-Methode werden die Parameter ausgelesen.

```
import com.batix.plugins.PluginAction

class ImportMitarbeiterAction : PluginAction() {
  companion object {
    const val PROP_API_TOKEN = "api-token"
    const val PROP_SEND_NOTIFICATION = "send-notification"
    const val PROP_IMPORTANT_THING = "important-thing"
  }

  override fun doAction() {
    val apiToken = getPropertyReplaced(PROP_API_TOKEN, "")
    require(!apiToken.isNullOrEmpty()) { "API-Token darf nicht leer sein" }

    val doSendNotification = getBooleanProperty(PROP_SEND_NOTIFICATION)

    val importantThing = getProperty(PROP_IMPORTANT_THING)

    // ...
  }
}
```

Damit der Baustein in der Verwaltung auftaucht, muss dieser dem Application Framework bekannt gegeben werden. Das erfolgt in der `load`-Methode der Plugin-Hauptklasse. Hier werden nebst den Metadaten des Bausteins auch dessen Parameter definiert.

```
override fun load() {
  registerAction(
    "import-mitarbeiter", // actionId
    ActionInfo(
      "Mitarbeiter importieren", // title
      "Importiert Mitarbeiter aus der externen Datenquelle.", // description
      false, // deprecated
      true, // beta
      "https://www.batix.de/unternehmen/unternehmenskultur/team/", // helpLink
      ImportMitarbeiterAction::class.java // actionClass
    ).apply {
      addTextParameter(
```

```
        ImportMitarbeiterAction.PROP_API_TOKEN, // name
        "API Token", // title
        "Token zur Authentifizierung an der abzufragenden API", // description
        false, // descriptionIsHtml
        true // required
      )

      addBooleanParameter(
        ImportMitarbeiterAction.PROP_SEND_NOTIFICATION, // name
        "Benachrichtigung versenden", // title
        "(Standard: nein)", // description
        false, // descriptionIsHtml
        false // required
      )

      addCustomParameter(object : ExtendedActionParameter() {
        override fun getName(): String {
          return ImportMitarbeiterAction.PROP_IMPORTANT_THING
        }

        override fun getTitle(): String {
          return "Wichtige Einstellung"
        }

        override fun writeAdminView(sb: StringBuffer, value: String?, conn: Connection) {
          sb.append("""<input type="text" """)
          sb.append("""name="param:${ImportMitarbeiterAction.PROP_IMPORTANT_THING}" """)
          sb.append("""value="${Tools.htmlEncode(value ?: "")}" """)
          sb.append("""style="border: 1px solid red;">""")
        }
      })
    }
  )
}
```

Der Plugin-Actionbaustein taucht somit in der Auswahl der Actionbausteine auf.

**API Token:** (api-token)

Token zur Authentifizierung an der abzufragenden API

**Benachrichtigung versenden:** (send-notification)

(Standard: nein)

ja nein

**Wichtige Einstellung:**

Freieingabe weiterer Eigenschaften

# Tag

Das Framework kann durch Plugins um Batix-Tags (`<bx:tagname>`) erweitert werden, welche dann in normalen Quelltexten wie Komplettseiten oder Textbausteinen verwendet werden können.

```
fun registerTag(tagInfo: TagInfo)
```

Es gibt zwei Arten von Tags: Frontend- und Backend-Tags. Der Unterschied besteht darin, dass sich Backend-Tags in der Verwaltung darstellen oder dort konfiguriert werden können (wie z. B. `<bx:text>` oder `<bx:containerfilter>`), Frontend-Tags hingegen können dies nicht.

# Frontend-Tags

Frontend-Tags werden mithilfe von `TagInfo.frontend` registriert und müssen von `com.batix.plugins.PluginFrontendTag` abgeleitet sein. Es reicht im einfachsten Fall, die Methode `addFrontendSourceText(StringBuffer)` zu überschreiben. An den `StringBuffer` hängt man die Ausgabe an und gibt diesen am Ende wieder zurück.

```
registerTag(TagInfo.frontend("tagname", MyFrontendTag::class.java, null))
```

> **Injection Vulnerability**
> Es müssen (HTML-)Steuerzeichen passend encoded werden, um Injectionlücken zu vermeiden.

Es empfiehlt sich daher die Methode `writeEncodedOutput(String, StringBuffer)` zu benutzen. Dieser übergibt man als ersten Parameter den gewünschten (uncodierten) Ausgabetext und reicht als zweiten Parameter den `StringBuffer` durch, den man übergeben bekommen hat. Die Methode sorgt automatisch dafür, dass entsprechend der aktuellen Frontendseite das passende Encoding (z. B. htmlencode bei HTML-Seiten) gewählt wird. Außerdem unterstützt das Tag damit automatisch den Parameter `encode` (also z. B. `<bx:tagname encode="plain" />`).

# Backend-Tags

Backend-Tags werden mithilfe von `TagInfo.backend` registriert und müssen von `com.batix.plugins.PluginBackendTag` abgeleitet sein. Hier müssen neben `addFrontendSourceText(StringBuffer)` noch die Methoden `getDataTable()` und `addAdminSourceText(StringBuffer)` überschrieben werden.

```
registerTag(TagInfo.backend("tagname", MyBackendTag::class.java, null))
```

Der erste Parameter (`"tagname"`) ist dabei der Name des Tags, wie er dann auch hinter `<bx:` verwendet wird. Als zweiter Parameter wird die implementierende Klasse übergeben. Der letzte Parameter ist ein optionales String-Array, falls das Tag nur für bestimmte Projekte (anhand webdir) zur Verfügung stehen soll.

Ist das Tag offen verwendbar (also z. B. `<bx:tagname>etwas Inhalt...</bx:tagname>`), kann der Inhalt (*Body*) mittels `computeBody()` ausgeführt und das Ergebnis ausgelesen werden. Eventuelle Batix-Tags im Body werden damit auch evaluiert.

# Parameter

Einem Tag können im Quelltext Parameter übergeben werden.

```
<bx:tagname mode="short" maxlen="5" pretty />
```

Diese Parameter können mit den `get*Parameter(key: String)`-Methoden ausgelesen werden – `key` ist der Name des Parameters. Mit `containsParameter(key: String)` kann festgestellt werden, ob ein Parameter angegeben wurde oder nicht.

```
fun getStringParameter(key: String): String?
fun getStringParameter(key: String, defaultValue: String?): String?

fun getIntParameter(key: String): Integer
fun getIntParameter(key: String, defaultValue: Integer): Integer

fun getBooleanParameter(key: String): Boolean
```

# Backend-Tag Speicherung

Backend-Tags stellen ein Eingabeelement für Redakteure oder Admins bereit. Dafür muss im Quellcode ein sogenannter Titel am Tag vergeben werden. Bei `<bx:tagname.Anrede />` ist der Titel beispielsweise "Anrede". Dieser wird dem Benutzer im Backend angezeigt.

Das bedeutet, dass die vom Benutzer getätigten Eingaben in der Datenbank gespeichert werden müssen. Die Methode `getDataTable()` teilt dem Framework mit, in welcher Tabelle die Daten zu speichern sind. Für kurze, einzeilige Texte ist das "EZT", für längere Texte "MZT". Der entsprechende Wert ist von `getDataTable()` einfach als String zurückzugeben.

Die Methode `addAdminSourceText(StringBuffer)` ist für das Rendern des entsprechenden Eingabefeldes zuständig. Für kurze Texte kann das ein `<input type="text">` sein. Der Name des Inputs muss als Prefix die Tabelle (gleicher Wert wie bei `getDataTable()`), einen Punkt als Separator und als Suffix den Titel des Tags enthalten (dieser ist als Feld `titel` verfügbar). Er muss also beispielsweise "EZT.Anrede" lauten. Ist bereits ein Wert gespeichert, ist das Feld `dataId` gefüllt. Dessen Wert muss dann noch, inklusive einem weiteren Punkt, an den Name angehangen werden.

> **Vorgefertigte Methoden**
> Als Best-Practice empfiehlt sich die Verwendung der Methoden `appendAdminHeadline(StringBuffer)` und `appendPublishStatus(StringBuffer)`, um eine konsistente Ausgabe des Titels und des Veröffentlichungsstatus zu erreichen.

Der Aufruf `getData("INHALT")` gibt den aktuell in der Datenbank gespeicherten Wert zurück. Dieser kann dann zur Ausgabe im Frontend sowie zur Vorbefüllung des Eingabefelds im Backend verwendet werden.

# Beispiel

## Frontend-Tag

Das Tag wird in der Plugin-Hauptklasse mithilfe von `TagInfo.frontend` registriert.

```
override fun load() {
  registerTag(TagInfo.frontend("unixtime", UnixTimeTag::class.java, null))
}
```

Dieses Beispiel-Tag gibt den aktuellen Epoch-Timestamp aus.

```
import com.batix.plugins.PluginFrontendTag
import com.batix.tags.BatixTagData
import java.time.Instant

class UnixTimeTag(data: BatixTagData?) : PluginFrontendTag(data) {
  override fun addFrontendSourceText(sb: StringBuffer): StringBuffer {
    val time = Instant.now().epochSecond
    writeEncodedOutput(time.toString(), sb)
    return sb
  }
}
```

Der Aufruf im Quellcode ist minimal.

```
<bx:unixtime />
```

Die Ausgabe im Frontend ist dann z. B. *1584620787*.

# Backend-Tag

In der Plugin-Hauptklasse wird das Tag mit `TagInfo.backend` registriert.

```
override fun load() {
  registerTag(TagInfo.backend("formattedtime", FormattedTimeTag::class.java, null))
}
```

Die Tag-Klasse implementiert die Frontend-Logik sowie die Anzeige des Eingabeelements im Backend.

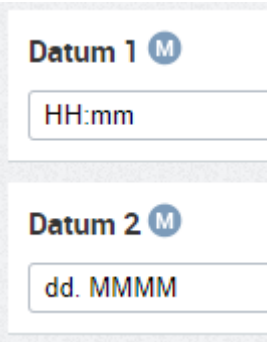
```
import com.batix.Tools
import com.batix.plugins.PluginBackendTag
import com.batix.tags.BatixTagData
import java.time.LocalDateTime
import java.time.format.DateTimeFormatter
import java.util.*

class FormattedTimeTag(data: BatixTagData?) : PluginBackendTag(data) {
  override fun addFrontendSourceText(sb: StringBuffer): StringBuffer {
    // Backendeingabe lesen
    val pattern = getData("INHALT") as String? ?: "dd.MM.yyyy HH:mm:ss"

    // Parameter aus Quelltext lesen
    val locale = Locale.forLanguageTag(getStringParameter("locale", "de-DE"))

    val now = LocalDateTime.now()
    val formatter = DateTimeFormatter.ofPattern(pattern, locale)
    writeEncodedOutput(formatter.format(now), sb)
    return sb
  }

  override fun getDataTable(): String {
    return "EZT"
```

```
  }

  override fun addAdminSourceText(sb: StringBuffer): StringBuffer {
    val inhalt = getData("INHALT") as String? ?: ""

    appendAdminHeadline(sb)
    appendPublishStatus(sb)

    var inputName = "$dataTable.$titel"
    if (dataId != null && dataId != "del") {
      inputName += ".$dataId"
    }

    sb.append("""<input type="text" name="${Tools.htmlEncode(inputName)}" """)
    sb.append("""value="${Tools.htmlEncode(inhalt)}">""")

    return sb
  }
}
```

Im Quelltext wird das Tag dann inklusive Titel verwendet.

```
<bx:formattedtime.Datum_1 />

<bx:formattedtime.Datum_2 locale="en-US" />
```

In der Verwaltung können Eingaben getätigt werden.




Im Frontend wird dann z. B. folgender Text ausgegeben:

```
13:28

19. March
```

# Timer Job

Plugins können neue Timer-Tasks für die im Framework eingebaute Zeitsteuerung mitbringen. Diese können dann beim Anlegen neuer Zeitsteuerungen ausgewählt und parametrisiert werden.

```
fun registerTimerJob(jobId: String, jobInfo: JobInfo)
```

`jobId` ist eine frei wählbare ID, die über alle Timer Jobs eines Plugins hinweg eindeutig sein muss. Es ist kein Problem, wenn mehrere Plugins einen Job mit derselben ID bereitstellen, da zum Ansprechen eines Jobs auch die `id` des Plugins herangezogen wird.

Der Titel des Jobs, sowie eine Instanz der implementierenden Klasse und die Standard-Parameter werden in `jobInfo` festgehalten. Die Standard-Parameter sind ein String-Array - jedes Element davon wird in der Oberfläche im Textfeld in eine eigene Zeile geschrieben (Kommentarzeilen beginnen mit `#`).

> **Thread-Safety**
> Die Instanz wird für alle Ausführungen nachgenutzt, sollte also thread-safe programmiert werden.

Die Klasse muss die `run(JobContext)` Methode aus `com.batix.plugins.Job` implementieren. `JobContext` enthält Informationen zum aktuellen Aufruf, wie z. B. die vom Benutzer eingestellten Parameter (`properties`) oder das Projekt, in dem die Zeitsteuerung definiert wurde (`web`).

# Beispiel

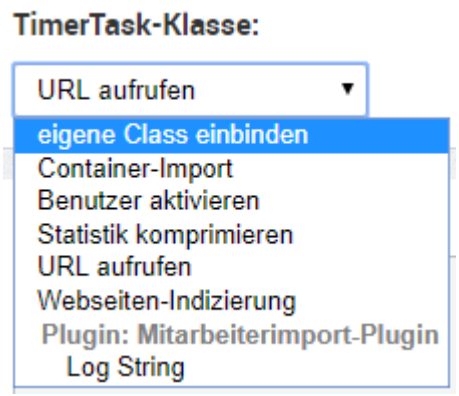
Der Timer Job wird dem Framework in der Plugin-Hauptklasse bekannt gegeben.

```
override fun load() {
  registerTimerJob(
    "logString", // jobId
    JobInfo("Log String", LogStringJob(), arrayOf("level", "#text"))
  )
}
```

Diese Beispiel-Job-Klasse liest die eingestellten Parameter und loggt die festgelegte Nachricht mit dem entsprechenden Level.

```
import com.batix.Log
import com.batix.plugins.Job
import com.batix.plugins.JobContext

class LogStringJob : Job {
  override fun run(context: JobContext) {
    val level = context.properties.getProperty("level", "INFO")
    val text = context.properties.getProperty("text", "DEFAULT TEXT")

    when (level.lowercase()) {
      "error" -> Log.error(text)
      "warn" -> Log.warn(text)
      "notice" -> Log.notice(text)
      "debug" -> Log.debug(text)
      else -> Log.info(text)
    }
  }
}
```

Der Job ist dann in der TimerTask-Auswahl einer Zeitsteuerung sichtbar.




Wählt man diesen aus, werden die Steuerparameter vorausgefüllt.

```
level=
#text=
```

# Request Interceptor

(ehemals "Request Handler")

Bestimmte, von Clients (z. B. Browser) angefragte Pfade, können durch Plugins überwacht und optional direkt beantwortet werden, ohne die Standardabläufe des Frameworks (wie Projekt und Menüpunkt finden) zu involvieren.

```
fun registerRequestInterceptor(
  pathPattern: String,
  interceptor: RequestInterceptor,
  priority: Int = 50
)
fun registerRequestInterceptor(
  condition: RequestCondition,
  interceptor: RequestInterceptor
)
```

Im einfachsten Fall wird ein regulärer Ausdruck (`pathPattern`) definiert und alle Requests mit passendem Pfad (also dem Text hinter der Domain ab `/`) werden an den `interceptor` gegeben. Mit einer `RequestCondition` kann zusätzlich noch die Domain (auch mittels regulärem Ausdruck) abgefragt werden. Außerdem kann damit nur auf Forwards reagiert (`onlyForwards()`) oder es können Forwards ausgeschlossen werden (`noForwards()`).

Der `interceptor` muss das Interface `com.batix.plugins.RequestInterceptor` implementieren. Hier gibt es drei Methoden, von denen eine oder mehrere überschrieben werden können.

Die `priority` bestimmt die Ausführungsreihenfolge der Interceptors und kann gesetzt werden, wenn man mit einem Interceptor andere beeinflussen will.

## `RequestInterceptor`-Interface

> **Thread-Safety**
> Dieselbe Instanz wird für alle Requests verwendet, muss also thread-safe implementiert werden.

Es können drei Methoden implementiert werden:

```
fun handlePre(event: RequestEvent)
fun handlePost(event: RequestEvent)

fun handleFrontendException(event: RequestEvent, exception: FrontendException)
```

Das `RequestEvent` beinhaltet dabei den `event.request: HttpServletRequest` und den `event.response: HttpServletResponse` sowie Methoden, um die folgende Requestbehandlung zu beeinflussen.

`handlePre` wird noch vor allen Framework-Funktionen aufgerufen und bestimmt durch das übergebene `RequestEvent`, ob der Request komplett vom Plugin behandelt wurde und deshalb von folgenden Plugins oder vom Framework nicht weiter beachtet werden soll. Wird auf dem Event `preventDefault` aufgerufen, wird das Framework seine Standardabläufe (wie z. B. Projekt und Menüpunkt finden) nicht durchführen. Ohne den Aufruf läuft der Request normal weiter - in diesem Fall sind Modifikationen an `response` mit Vorsicht zu genießen, da die weiteren Abläufe ggf. zu einem Redirect führen oder gesetzte Werte wieder überschrieben werden können. Wird auf dem Event `stopPropagation` aufgerufen, werden (nach priority) nachfolgende Plugins den Request nicht erhalten und damit auch nicht auf diesen reagieren können.

Der Aufruf von `handlePost` erfolgt, nachdem alle anderen Abläufe im Framework erledigt sind, auch falls es Fehler gab. In dieser Phase könnten schon Teile der Response-Header oder sogar des Response-Bodys geschrieben sein, Modifikationen dort sollten also mit Bedacht durchgeführt werden, wenn überhaupt. Methodenaufrufe auf dem `RequestEvent` zur Beeinflussung des nachfolgenden Ablaufs sind hier wirkungslos - das Framework hat den Request bereits behandelt, und nachfolgende Plugins werden auch nur dann übersprungen, falls in `handlePre` der `stopPropagation`-Aufruf erfolgte.

Mit `handleFrontendException` kann auf Fehler aufgrund verschiedener Dinge, wie z. B. Projekt / Menüpunkt / Datei nicht gefunden, fehlende Authentifizierung oder auch Serverfehler, reagiert werden - wobei der genaue Fehler(-Grund) in `exception` festgehalten ist. Hier kann lediglich mit `preventDefault` die nachfolgende Fehlerbehandlung des Frameworks unterbunden werden - nicht aber die nachfolgender Plugins.

Möchte man die Methoden je Request korrelieren, so bieten sich Request-Attribute an, in denen man seinen State festhalten kann. In der `RequestInterceptor`-implementierenden Klasse selbst können keine Felder o. ä. benutzt werden, da die Instanz der Klasse ja threadübergreifend verwendet wird.

Es werden übrigens nicht nur Frontendaufrufe gematcht. Auch Aufrufe des Backends können überwacht werden. So könnte man dieses z. B. durch eine Prüfung der Client-IP, oder anderen Merkmalen, noch weiter abschotten.

Eine Sonderform von Request Interceptors ist für statischen Content verfügbar.

# Priority

Die `priority` bestimmt die Ausführungsreihenfolge der Interceptors, auch über mehrere Plugins hinweg: Je niedriger die Priorität, desto früher wird ein Interceptor aufgerufen. Im Regelfall spielt sie kaum eine Rolle, ist aber entscheidend um zu bestimmen, welche Interceptors denn von Aufrufen von `stopPropagation` betroffen werden sollen.

`stopPropagation` kann aber in jedem Fall nur Interceptors mit einem größeren `priority`-Value betreffen - mehrere Interceptors derselben Priorität können sich nicht gegenseitig abbrechen. Zudem wird sich der Prioritätswert gemerkt, falls in `handlePre` `stopPropagation` aufgerufen wird, und gewährleistet, dass auch das `handlePost` der verhinderten Interceptors nicht aufgerufen wird.

`registerRequestInterceptor( condition: RequestCondition, interceptor: RequestInterceptor)` nimmt keine gesonderte `priority`, da diese intern Teil der `RequestCondition` ist und direkt auf ihr definiert werden kann.

# Beispiel

Hier werden drei Request Interceptors registriert. Einer misst die Dauer von Requests, ein Weiterer überprüft Authentifizierung bei API-Zugriffen, ein Letzter führt eine Fehlerstatistik.

```
override fun load() {
  registerRequestInterceptor(
    "^/api/.*",
    AuthInterceptor(),
    RequestCondition.AUTH_PRIORITY
  )

  registerRequestInterceptor("^/api/.*", TimingInterceptor())
  registerRequestInterceptor("^(?!/verwaltung/).*", ErrorMetricsInterceptor())
}
```

Der `AuthInterceptor` prüft bei jedem Request, ob es sich um einen authentifizierten `BxUser` handelt, der einer bestimmten Gruppe angehört und blockt unauthentifizierte Anfragen ab, indem er die Ausführung nachfolgender Interceptors (in diesem Fall `TimingInterceptor` - aber auch andere Interceptors anderer Plugins, die auf dem selben Pfad lauschen) und Framework-Funktionen verhindert. Die `AUTH_PRIORITY` entspricht einem Wert von *15* und liegt damit vor allen anderen registrierten Interceptors. Hier wird zwar nur `handlePre` implementiert, trotzdem wird implizit auch `handlePost` vom `TimingInterceptor` verhindert.

```
import com.batix.Log
import com.batix.ConnectionPool
import com.batix.modul.BxUser
import com.batix.plugins.RequestEvent
import com.batix.plugins.RequestInterceptor
import jakarta.servlet.http.HttpServletResponse

class AuthInterceptor : RequestInterceptor {
  override fun handlePre(event: RequestEvent) {
    val user = BxUser.findInstance(event.request) // User-Instanz
    if (user != null) { // angemeldet
      ConnectionPool.withSystemConnectionDo { conn ->
        if ("17AD26C9C4C" in user.getGroups(conn).map { it.id }) {
          Log.debug("user is authenticated")
          return
        }
      }
    }

    event.preventDefault()
    event.stopPropagation()
    event.response?.sendError(HttpServletResponse.SC_UNAUTHORIZED)
  }
}
```

Der `TimingInterceptor` merkt sich zu Beginn aller Requests, die mit `/api/` anfangen, deren Start in einem Request-Attribut. Wenn der Request fertig ist, wird der Startzeitpunkt wieder ausgelesen, die Dauer berechnet und geloggt. Ohne explizite Angabe einer `priority` erhält er die `RequestCondition.DEFAULT_PRIORITY` von *50*.

```
import com.batix.Log
import com.batix.plugins.RequestEvent
import com.batix.plugins.RequestInterceptor
import java.time.Duration
import java.time.Instant

class TimingInterceptor : RequestInterceptor {
  override fun handlePre(event: RequestEvent) {
    event.request.setAttribute(REQUEST_STARTED_ATTRIBUTE, Instant.now())
    // event.preventDefault() // CMS-Weiterbehandlung stoppen
```

```
    // event.stopPropagation() // Plugin-Weiterbehandlung stoppen
  }

  override fun handlePost(event: RequestEvent) {
    val instant = event.request.getAttribute(REQUEST_STARTED_ATTRIBUTE) as Instant
    val duration = Duration.between(instant, Instant.now())
    Log.debug("request of ${event.request.requestURL} took $duration")
  }

  companion object {
    private const val REQUEST_STARTED_ATTRIBUTE = "timing-interceptor-started"
  }
}
```

`ErrorMetricsInterceptor` überwacht für alle Requests, die **nicht** mit `/verwaltung/` anfangen, aufgetretene Fehler und aktualisiert eine fiktive Statistik.

```
import com.batix.plugins.RequestInterceptor
import com.batix.tags.FrontendException
import com.batix.tags.FrontendExceptionInterface
import com.batix.plugins.RequestEvent

class ErrorMetricsInterceptor : RequestInterceptor {
  override fun handleFrontendException(event: RequestEvent, exception: FrontendException) {
    val errorCode = exception.errorCode
    stats.errors.withInternalCode(errorCode).increment()

    if (errorCode == FrontendExceptionInterface.Code.UNKNOWN_WEB.bxCode) {
      stats.unknownProjects.countDomain(event.request.serverName)
    }
  }
}
```

> **javax / jakarta**
> Ab Framework v3.0 müssen die `jakarta` anstatt der `javax` Klassen verwendet werden.

# Statische Ressourcen schützen

Ein anderes Anwendungsgebiet für Request Interceptors ist das Schützen der statischen Ressourcen, die in der Verwaltung unter *Ressourcen > Vorlagen > statische Ressourcen* gepflegt

werden.

Hierfür wird einfach ein Request Interceptor für die zu schützenden Pfade / Dateien definiert und die gewünschten User-Checks durchgeführt.

Simples Beispiel:

```
registerRequestInterceptor("^/static/my-project/assets-for-vips/.*$", object :
RequestInterceptor {
  override fun handlePre(event: RequestEvent) {
    val user = BxUser.findInstance(event.request)
    val vipsGroup = UserGroup.findGroup("196243C94D0")
    val allowed = user != null && ConnectionPool.withSystemConnectionDo { conn ->
      user.isInGroup(conn, vipsGroup)
    }

    if (!allowed) {
      event.response.status = HttpServletResponse.SC_UNAUTHORIZED // 401
      event.preventDefault()
    }
  }
})
```

> **Verfügbarkeit**
> Nur verfügbar für Systeme, die in Docker laufen oder speziell konfiguriert sind (`/static/` via Tomcat). Das ist standardmäßig ab Framework Version 2.9 der Fall.

# Static Content

> **Verfügbarkeit**
> Ab Batix Application Framework Version 2.7.1 verfügbar.

Plugins können in ihre .zip Datei statische Ressourcen-Dateien wie HTML-Seiten, Bilder, Schriften oder JS-/CSS-Dateien integrieren. Diese müssen in einem Unterordner unterhalb des Projektordners namens `static` liegen. Falls im Plugin-Projektverzeichnis ein Ordner `static` existiert, wird dessen Inhalt automatisch in die ZIP übernommen.

Es können auch komplette Vue Apps oder auch mit VitePress erstellte Dokumentationen ausgeliefert werden. Dank Gradle landet sogar automatisch zur Build-Zeit die kompilierte Vue-App oder die erstellte VitePress-Seite in der .zip Datei. Siehe dazu auch den Guide Vue App ausliefern.

> **INFO**
> Hier ist die Rede von `RequestHandler`n, obwohl es im vorherigen Kapitel immer `RequestInterceptor` hieß. Das hat historische Gründe - `RequestHandler` war der alte Name und zugunsten einfacherer Backwards-Compatibility wurde hier der Name nicht angepasst - es handelt sich aber intern trotzdem um `RequestInterceptor`s.

```
fun registerRequestHandlerForStaticContent(
  pathPrefix: String,
  staticPath: String,
  fallback: String,
  guard: StaticContentHandler.Guard = null
)

fun registerRequestHandlerForStaticContent(
  condition: RequestCondition,
  pathMapper: Function<HttpServletRequest, String>,
  staticPath: String,
  fallback: String,
  guard: StaticContentHandler.Guard = null
)
```

Die erste Methode eignet sich für einfache Fälle. `pathPrefix` ist der Anfang des Pfads, auf den reagiert werden soll - dies ist kein regulärer Ausdruck, sondern ein normaler String. Es wird dabei auf jede Domain reagiert. `staticPath` ist der Name des Ordners im `static` Ordner des Plugin-ZIPs.

`fallback` ist ein optionaler Parameter, der eine Alternativ-Datei angibt, welche ausgeliefert wird, falls eine nicht-existierende Datei angefordert wird - bei SPAs ist dies meistens index.html. `guard` ist ein Callback, der das Functional Interface `StaticContentHandler.Guard` und damit die Methode `intercept` implementiert. Hier kann entschieden werden, ob die Ressource doch nicht (z. B. wegen fehlender Authorisierung) ausgeliefert werden soll (in diesem Fall sollte der Guard `true` zurückgeben).

Hat man als `pathPrefix` beispielsweise /import-plugin/frontend/ und als `staticPath` den Ordner "import-frontend" definiert (und `fallback` mit `null` angegeben), dann würde ein Aufruf von domain.tld/import-plugin/frontend/import.html die Datei <ZIP>/static/import-frontend/import.html ausliefern, falls diese existiert. Ein Aufruf von domain.tld/import-plugin/frontend/ (entspricht also dem `pathprefix`) liefert die index.html Datei (<ZIP>/static/import-frontend/index.html) aus.

Mit der zweiten Methode hat man dank einer `RequestCondition` mehr Kontrolle, unter welchen Pfaden die statischen Dateien ausgeliefert werden. Dafür muss man mittels `pathMapper` aber auch eine Funktion übergeben, die aus dem Request den entsprechenden Pfad zur statischen Datei extrahiert (da es ja kein festes Prefix gibt).

Wird eine angefragte Datei nicht gefunden und es ist kein `fallback` angegeben (oder `fallback` wird auch nicht gefunden), wird eine Fehlerseite mit Status 404 erzeugt. Dateien werden mit passendem MIME-Type und Cache-Headern ausgeliefert.

Der Aufruf ohne Dateiangabe (also z. B. domain.tld/import-plugin/frontend/) liefert die Datei index.html aus (falls diese existiert).

Ein in beiden Fällen optionaler `guard` kann die `intercept`-Methode überschreiben und auf Request, Response sowie den Ressourcennamen reagieren um die Auslieferung ggf. zu unterbinden (`return true` heißt blockieren).

```
fun interface Guard {
  fun intercept(
    request: HttpServletRequest,
    response: HttpServletResponse,
    resource: String
  ): Boolean
}
```

# Beispiel

Es wird folgender Inhalt der Plugin-ZIP angenommen (Ausschnitt).

```
<ZIP>
└── static
```

```
    └── import-frontend
        ├── css
        │   └── bootstrap.min.css
        ├── import.html
        ├── index.html
        └── js
            └── bootstrap.min.js
```

Der folgende Handler wird registriert.

```
override fun load() {
  registerRequestHandlerForStaticContent(
    "/import-plugin/frontend/", // pathPrefix
    "import-frontend", // staticPath
    "index.html" // fallback
  )
}
```

Daraus ergeben sich folgende Request-zu-Datei Mappings.

| Request | Datei |
| --- | --- |
| domain.tld/import-plugin/frontend/import.html | <ZIP>/static/import-frontend/import.html |
| domain.tld/import-plugin/frontend/ | <ZIP>/static/import-frontend/index.html |
| domain.tld/import-plugin/frontend/existiert.nicht | <ZIP>/static/import-frontend/index.html |

Innerhalb der .html Dateien können die Scripts und Styles mittels relativem Pfad angesprochen werden.

```
<link rel="stylesheet" href="css/bootstrap.min.css" />
<script src="js/bootstrap.min.js"></script>
```

# Admin Page

Für Backend-Benutzer können eigenständige Seiten innerhalb der Verwaltung zur Verfügung gestellt werden. Das entspricht den bisherigen "Plugin-Seiten" auf Projektebene, die allerdings direkt im Backend als normale Menüpunkte angelegt wurden.

Admin Pages stehen in allen Projekten zur Verfügung und beziehen ihren Inhalt aus der ZIP-Datei des Plugins, siehe zu dieser Funktionsweise auch die Static Content Extension.

```
fun registerAdminPage(info: AdminPageInfo)
```

`AdminPageInfo` besteht aus `id`, `title`, `staticPath`, `handler` und `guard`. `id` ist eine frei wählbare ID, die über alle Admin Pages eines Plugins hinweg eindeutig sein muss. Es ist kein Problem, wenn mehrere Plugins eine Admin Page mit derselben ID bereitstellen, da zum Ansprechen einer Admin Page auch die `id` des Plugins herangezogen wird.

`title` ist der Titel, wie er im Verwaltungsmenü auftaucht. `staticPath` ist der Unterordner im `static` Ordner des Plugin-ZIPs, aus dem die Dateien ausgeliefert werden.

Der optionale Wert `handler` ist ein `RequestHandler`, der aufgerufen wird, falls eine Datei angefordert wird, die nicht existiert. Hiermit können die Actions der bisherigen Plugin-Seiten nachgebildet werden.

Der optionale Wert `guard` ist ein `StaticContentHandler.Guard`, wie er auch bei Static Content verwendet wird. Falls definiert, wird die `intercept` Methode von `guard` in folgenden Fällen mit diesen Werten aufgerufen:

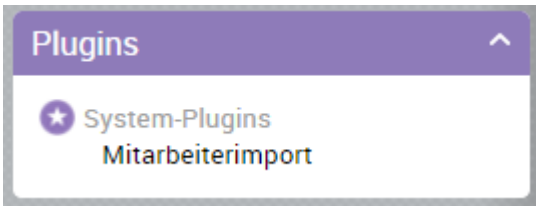
- `resource=""` - um zu prüfen, ob die Admin Page dem Backend-Benutzer in der Navigation angezeigt werden soll oder nicht. Wird `true` zurückgegeben, taucht die Admin Page in der Navigation nicht auf.
- `resource="index.html"`, `resource="assets/style.css"`, `resource="dynamic/do-something-action"` - um zu prüfen, ob der Request auf diese Ressource erlaubt werden soll oder nicht. Wird `true` zurückgegeben, erfolgt die Auslieferung der Ressource nicht, ebenfalls wird weder `handler` aufgerufen noch eine eigene Fehlermeldung generiert.

# Beispiel?

In der Pluginhauptklasse wird die Admin Page registriert. In der Plugin-ZIP müssen die Dateien dementsprechend im Ordner static/import-ma/ liegen.

```
override fun load() {
  registerAdminPage(
    AdminPageInfo(
      "import-mitarbeiter", // id
      "Mitarbeiterimport", // title
      "import-ma", // staticPath
      null, // handler
      null // guard
    )
  )
}
```

Im Backend stellt sich die Admin Page im Menü wie folgt dar.



# WebSocket

Auch WebSockets können von Plugins ganz einfach genutzt werden. Dabei können sogar erweiterte Funktionalitäten wie Subprotokolle verwendet oder der Handshake modifiziert werden.

```
fun registerWebSocket(socketId: String, handler: WebSocketHandler)
```

`socketId` ist eine frei wählbare ID, die über alle WebSockets eines Plugins hinweg eindeutig sein muss. Es ist kein Problem, wenn mehrere Plugins einen WebSocket mit derselben ID bereitstellen, da zum Ansprechen eines WebSockets auch die `id` des Plugins herangezogen wird.

Der WebSocket wird dann unter einem Pfad bereitgestellt, der nach folgendem Schema aufgebaut ist.

```
/.well-known/bx-websockets/<plugin ID>/<socketId>
```

Als `handler` übergibt man eine Instanz, die `com.batix.plugins.WebSocketHandler` implementiert.

> **Thread-Safety**
> Dieselbe Instanz wird für alle Requests verwendet, muss also thread-safe implementiert werden.

Im Handler können eine oder mehrere der folgenden Methoden überschrieben werden. Die Methodennamen sind selbsterklärend.

```
fun onOpen(client: WebSocketContext)
fun onClose(client: WebSocketContext, closeReason: CloseReason)
fun onTextMessage(client: WebSocketContext, message: String)
fun onBinaryMessage(client: WebSocketContext, data: ByteArray)
fun onError(client: WebSocketContext, throwable: Throwable) // ab Framework v2.7.8
```

Mittels `client` kann der Gegenüber identifiziert werden. In `WebSocketContext` stecken die `httpSession` (falls eine beim Öffnen des WebSockets vorhanden war) sowie die `webSocketSession` mit der z. B. Nachrichten geschickt oder der Socket geschlossen werden kann. Außerdem gibt es noch die Methode `getAllClients()` mit der alle aktuell zu diesem WebSocket Handler verbundenen Clients aufgelistet werden (das schließt auch `client` mit ein).

Der häufigste Fall ist eine Textnachricht zu versenden, dies kann wie folgt erledigt werden.

```
client.webSocketSession.basicRemote.sendText("Text...")
```

Eine Verbindung kann mittels `close()` geschlossen werden, dabei ist auch die Angabe einer `CloseReason` möglich.

```
client.webSocketSession.close(CloseReason(
  CloseReason.CloseCodes.NORMAL_CLOSURE, "Verbindung beendet."
))
```

# Erweiterte Funktionalitäten?

Für diese Funktionalitäten muss der WebSocket schon beim Start des Frameworks initialisiert werden. Dafür ist es nötig die entsprechenden Informationen in der `plugin.yaml` Datei zu hinterlegen und das Plugin beim Systemstart zu laden. Dazu wird in `plugin.yaml` ein neuer Hauptkey `configuredWebSockets` ergänzt.

```
configuredWebSockets:
  advanced-socket:
    subprotocols:
      - proto1
      - proto2
    configurator: com.company.plugin.MySocketConfigurator
```

Dies ist eine Map, wobei der Key (im Beispiel hier `advanced-socket`) der `socketId` entspricht. Diese muss dann durch das Plugin noch registriert werden (wie normale WebSockets). Dem Beispiel folgend, müsste also noch folgender Aufruf erfolgen.

```
registerWebSocket("advanced-socket", SomeHandler())
```

`subprotocols` und `configurator` können beide zusammen oder auch einzeln pro WebSocket verwendet werden.

> **Nur nach Framework-Neustart verfügbar**
> Diese speziellen WebSockets sind nur nach einem Neustart des Frameworks, und nur wenn das Plugin zu diesem Zeitpunkt automatisch geladen wird, verfügbar. Damit Änderungen an `subprotocols` und eine neue `configurator` Instanz wirksam werden, muss das Framework neugestartet werden.

# Subprotokolle?

Falls der Server mehrere Subprotokolle unterstützt, können diese unter `subprotocols` aufgelistet werden (Liste von Strings). Schickt ein Client seinerseits auch Subprotokolle mit, wird das erste verwendet, welches auch der Server unterstützt. Diese Auswahl kann optional durch einen Configurator angepasst werden (s. u.).

Das ausgehandelte Subprotokoll kann folgendermaßen ausgelesen werden. Wurde sich auf kein Subprotokoll geeinigt (oder gab es keine), ist dieser Wert leer.

```
client.webSocketSession.negotiatedSubprotocol
```

# Configurator?

Weitere Details können durch einen sogenannten Configurator angepasst werden. Dies ist eine von `jakarta.websocket.server.ServerEndpointConfig.Configurator` abgeleitete Klasse.

> **javax / jakarta**
> Ab Framework v3.0 müssen die `jakarta` anstatt der `javax` Klassen verwendet werden.

Der vollqualifizierte Name der Klasse (d. h. inklusive Package) ist als `configurator` des entsprechenden WebSockets in der `plugin.yaml` anzugeben. Hier gibt es verschiedene Methoden, die überschrieben werden können. Es folgt eine Auswahl.

```
fun getNegotiatedSubprotocol(supported: List<String>, requested: List<String>): String
```

Hiermit kann die Auswahl des Subprotokolls getroffen werden. Es ist ein String zurückzugeben, der sowohl in `supported` (Liste der Subprotokolle des **Servers**), als auch in `requested` (Liste der Subprotokolle des **Clients**) vorkommt. Ist kein Subprotokoll akzeptabel, muss ein Leerstring zurückgegeben werden.

```
fun checkOrigin(originHeaderValue: String): Boolean
```

Mit dieser Methode kann der Origin-Header des Clients überprüft werden, falls für diesen nur bestimmte Werte zugelassen sein sollen. Der Rückgabewert ist, ob der Check erfolgreich war. Diesen Header schicken ziemlich alle Browser mit, andere Clients aber ggf. nicht (können ihn auch fälschen).

```
fun modifyHandshake(
  sec: ServerEndpointConfig,
```

```
  request: HandshakeRequest,
  response: HandshakeResponse
)
```

Hierdurch kann die HTTP-Response des Verbindungsaufbaus angepasst werden. Subprotokolle und der Origin-Check sind an dieser Stelle schon durchlaufen wurden.

# Beispiel?

Das Beispiel implementiert den simpelst-möglichen Broadcast-WebSocket, d. h. eine eingehende Nachricht wird an alle verbundenen Clients geschickt (inkl. dem Sender).

```
override fun load() {
  registerWebSocket("broadcast", Broadcaster())
}
```

Lautet die Plugin-ID `com.batix.website:import-mitarbeiter`, so ist der WebSocket dann unter folgender URL erreichbar.

```
wss://domain.tld/.well-known/bx-websockets/com.batix.website:import-mitarbeiter/broadcast
```

> **ws/wss und Domain auslesen**
> Um in JavaScript das passende Protokoll (`ws` oder `wss`) und die Domain herauszufinden, kann folgendes Snippet verwendet werden (Standardports vorausgesetzt).

```
const wsPath = "/.well-known/bx-websockets/<plugin ID>/<socketId>";
const protocolPrefix = (window.location.protocol === 'https:') ? 'wss:' : 'ws:';
const socket = new WebSocket(protocolPrefix + "//" + location.host + wsPath);
```

> Grundsätzlich empfiehlt sich eine verschlüsselte HTTPS-Verbindung, damit auch die WebSocket-Verbindung verschlüsselt ist.

Im Handler ist lediglich die `onTextMessage` Methode zu überschreiben.

```
import com.batix.plugins.WebSocketContext
import com.batix.plugins.WebSocketHandler

class Broadcaster : WebSocketHandler {
```

```
  override fun onTextMessage(client: WebSocketContext, message: String) {
    client.allClients.forEach {
      it.webSocketSession.basicRemote.sendText(message)
    }
  }
}
```

# 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())
  }
}
```