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**
> Der Typ des zurückgegebenen Objektes (und eventueller weiterer, darin eingebetteter Typen – wie bei Listen oder Maps) sollte sich auf **Standard-JVM- und Framework-Typen** wie beispielsweise `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."
```

Revision #1
Created 9 May 2025 06:19:10 by Batix
Updated 9 May 2025 06:19:10 by Batix