Plugin

Plugins werden als Unterprojekte angelegt. In einem Git-Repo kann es also problemlos mehrere Plugins geben. Inwiefern das organisatorisch sinnvoll ist, muss individuell geklärt werden. Sachen wie Projektzugehörigkeit, Issue-Management und Abhängigkeiten (Dependencies) spielen dabei eine Rolle.

Die Programmiersprache, mit der Plugins entwickelt werden, ist frei wählbar. Sie muss allerdings JVM-kompatibel sein. Es wären also z. B. Java, Groovy, Scala oder Kotlin möglich. Diese Doku beschränkt sich auf Kotlin, das ist auch unsere Empfehlung.

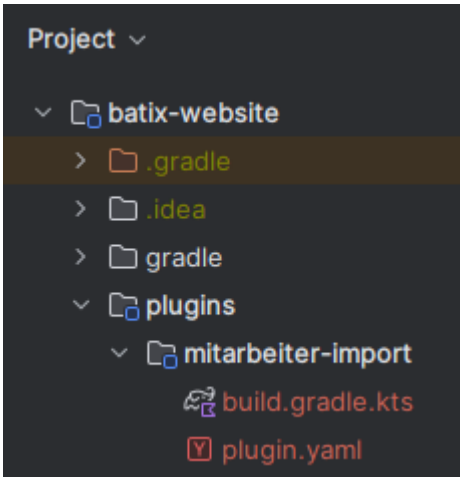
Beim Builden wird das Plugin in ein ZIP-Archiv verpackt, das nebst dem Code und den Dependencies auch Metadaten zum Plugin wie Titel und Version enthält.

Struktur

Eine Konvention, die bei uns oft verwendet wird, fasst Plugin-Unterprojekte im Verzeichnis `plugins` zusammen. Wenn das Git-Repo wächst, hat das den Vorteil, dass man nicht alle Verzeichnisse abklappern muss, um alle Plugins zu finden. Außerdem könnten so auch allgemeine Plugin-Tasks im Unterprojekt `:plugins` angelegt werden.

Ein Plugin besteht im Minimum aus einer kleinen YAML-Datei mit Metainformationen sowie der Plugin-Hauptklasse. Die Metainformationen, der eigene Code, sowie alle Dependencies (externe .jar Dateien) werden durch den `build` oder `packageBatixPlugin` Task in eine .zip Datei geschrieben, die dann im Framework hochgeladen werden kann.

Um beim Beispiel des Import-Plugins zu bleiben, wird also die Unterordner-Struktur `plugins/mitarbeiter-import` angelegt. Dort werden dann zwei neue Dateien angelegt: `build.gradle.kts` und `plugin.yaml`.



In der Datei `settings.gradle.kts` im Hauptverzeichnis wird dieses neue Unterprojekt nun noch Gradle mittels `include` bekannt gegeben:

```
// ...

include(":plugins:mitarbeiter-import")
```

plugin.yaml

In dieser Datei stehen Informationen wie der Titel des Plugins. Außerdem ist ein Verweis auf die Plugin-Hauptklasse enthalten, die den Einstiegspunkt des Plugins darstellt. Ein Plugin ist eindeutig durch seine `id` identifiziert, welche automatisch anhand der `group` und des Projektnamens in Gradle in die finale Datei geschrieben wird. Es können nicht mehrere Plugins mit derselben `id` geladen werden. Die Version des Plugins wird ebenso automatisch in diese Datei geschrieben.

```
id: "$id"
name: "Mitarbeiter Import"
pluginClass: "com.batix.website.mitarbeiter.ImportPlugin"
version: "$version"
```

Die Werte für `id` und `version` sind Platzhalter und dürfen nicht geändert werden. `name` sollte einen kurzen Titel enthalten, dieser wird im Framework an diversen Stellen angezeigt. Der Wert für `pluginClass` muss dem vollqualifizierten Namen der Plugin-Hauptklasse entsprechen (d. h. inklusive Package).

> **Konstante id**
> Der Wert von `id` sollte sich nach der ersten veröffentlichten Version nicht mehr ändern, da sonst Zuordnungen, die im Framework getroffen wurden (wie z. B. Plugin-Preferences), verloren gehen!
>
> `group` in build.gradle.kts sowie die Verzeichnisstruktur zum Plugin-Unterprojekt sollten also final sein.

Es kann auch eine minimale System-Version festgelegt werden, unter der das Plugin laufen muss.

```
minSystemVersion: "2.7.1"
```

Diese Datei wird im Build-Prozess mit den passenden Werten gefüllt und am Ende in das Plugin-ZIP gepackt, wo sie vom Framework als einer der ersten Vorgänge beim Plugin-Laden ausgelesen wird.

build.gradle.kts

Auch ein Gradle-Unterprojekt wird über seine `build.gradle.kts` Datei konfiguriert. Diese Datei im Unterprojekt wird, je nachdem welche **Framework Version** im Einsatz ist, mit den entsprechenden Zeilen gefüllt:

ab v3.0

```
plugins {
  kotlin("jvm")
}

//version = "1.0.0"

repositories {
  mavenCentral()

  maven {
    // https://git.batix.gmbh/maven-packages/cms/-/packages
    url = uri("https://git.batix.gmbh/api/v4/projects/477/packages/maven")

    authentication {
      create<HttpHeaderAuthentication>("header")
    }

    if (!System.getenv("CI_JOB_TOKEN").isNullOrEmpty()) {
      credentials(HttpHeaderCredentials::class) {
        name = "Job-Token"
        value = System.getenv("CI_JOB_TOKEN")
      }
    } else {
      credentials(HttpHeaderCredentials::class) {
        name = "Private-Token"
        value = property("batix.gitlab.pat").toString()
      }
```

```
    }
  }
}

dependencies {
  implementation("org.jetbrains.kotlin:kotlin-stdlib")

  // https://git.batix.gmbh/maven-packages/cms
  // https://git.batix.gmbh/maven-packages/cms/-/packages
  compileOnly("com.batix:batix-cm:3.0.0")

  // via Tomcat 10.1
  compileOnly("jakarta.servlet:jakarta.servlet-api:6.0.0")

  // via Tomcat 10.1
  compileOnly("jakarta.servlet.jsp:jakarta.servlet.jsp-api:3.1.1")

  // via Tomcat 10.1
  compileOnly("jakarta.websocket:jakarta.websocket-api:2.1.1")

  // via Tomcat 10.1
  compileOnly("jakarta.websocket:jakarta.websocket-client-api:2.1.1")

  // via CMS
  compileOnly("org.apache.groovy:groovy:4.0.9")
  compileOnly("org.apache.groovy:groovy-dateutil:4.0.9")
  compileOnly("org.apache.groovy:groovy-json:4.0.9")
  compileOnly("org.apache.groovy:groovy-sql:4.0.9")
  compileOnly("org.apache.groovy:groovy-xml:4.0.9")

  // via CMS
  //compileOnly("com.google.code.gson:gson:2.7")
}

java {
  toolchain {
    languageVersion.set(JavaLanguageVersion.of(21))
  }
}
```

```
//
// -- packaging --
//

val packageBatixPlugin by tasks.registering(Zip::class) {
  group = "build"
  dependsOn("cleanPackageBatixPlugin")

  val distDir = layout.buildDirectory.dir("dist")
  destinationDirectory.set(distDir)
  outputs.dir(distDir)

  from("plugin.yaml") {
    expand(
      mapOf(
        "id" to "${project.group}:${project.name}",
        "version" to project.version
      )
    )
  }

  from(tasks.named("jar")) {
    into("lib")
  }

  from(configurations.runtimeClasspath) {
    into("lib")
    exclude("slf4j-api-*.jar")
  }

  from("static") {
    into("static")
  }
}

tasks.named("assemble") {
  dependsOn(packageBatixPlugin)
}
```

ab v2.9

```
plugins {
  kotlin("jvm")
}

//version = "1.0.0"

repositories {
  mavenCentral()

  maven {
    // https://git.batix.gmbh/maven-packages/cms/-/packages
    url = uri("https://git.batix.gmbh/api/v4/projects/477/packages/maven")

    authentication {
      create<HttpHeaderAuthentication>("header")
    }

    if (!System.getenv("CI_JOB_TOKEN").isNullOrEmpty()) {
      credentials(HttpHeaderCredentials::class) {
        name = "Job-Token"
        value = System.getenv("CI_JOB_TOKEN")
      }
    } else {
      credentials(HttpHeaderCredentials::class) {
        name = "Private-Token"
        value = property("batix.gitlab.pat").toString()
      }
    }
  }
}

dependencies {
  implementation("org.jetbrains.kotlin:kotlin-stdlib")

  // https://git.batix.gmbh/maven-packages/cms
  // https://git.batix.gmbh/maven-packages/cms/-/packages
  compileOnly("com.batix:batix-cm:2.9.0.3")
```

```
  // via Tomcat 9
  compileOnly("javax.servlet:javax.servlet-api:4.0.0")

  // via Tomcat 9
  compileOnly("javax.websocket:javax.websocket-api:1.1")

  // via CMS
  compileOnly("org.apache.groovy:groovy:4.0.9")
  compileOnly("org.apache.groovy:groovy-dateutil:4.0.9")
  compileOnly("org.apache.groovy:groovy-json:4.0.9")
  compileOnly("org.apache.groovy:groovy-sql:4.0.9")
  compileOnly("org.apache.groovy:groovy-xml:4.0.9")

  // via CMS
  //compileOnly("com.google.code.gson:gson:2.7")
}

java {
  toolchain {
    languageVersion.set(JavaLanguageVersion.of(17))
  }
}

//
// -- packaging --
//

val packageBatixPlugin by tasks.registering(Zip::class) {
  group = "build"
  dependsOn("cleanPackageBatixPlugin")

  val distDir = layout.buildDirectory.dir("dist")
  destinationDirectory.set(distDir)
  outputs.dir(distDir)

  from("plugin.yaml") {
    expand(
      mapOf(
```

```
        "id" to "${project.group}:${project.name}",
        "version" to project.version
      )
    )
  }

  from(tasks.named("jar")) {
    into("lib")
  }

  from(configurations.runtimeClasspath) {
    into("lib")
    exclude("slf4j-api-*.jar")
  }

  from("static") {
    into("static")
  }
}

tasks.named("assemble") {
  dependsOn(packageBatixPlugin)
}
```

vor v2.9

```
plugins {
  kotlin("jvm")
}

//version = "1.0.0"

repositories {
  mavenCentral()

  // https://git.batix.gmbh/pub/maven/-/packages
```

```
  maven("https://git.batix.gmbh/api/v4/projects/324/packages/maven")
}

dependencies {
  implementation("org.jetbrains.kotlin:kotlin-stdlib")

  // https://git.batix.gmbh/pub/maven/-
/packages/?orderBy=created_at&sort=desc&search%5B%5D=com%2Fbatix%2Fcms-api
  compileOnly("com.batix:cms-api:2.8.1.2")

  // via Tomcat 8.5
  compileOnly("javax.servlet:javax.servlet-api:3.1.0")

  // via Tomcat 8.5
  compileOnly("javax.websocket:javax.websocket-api:1.1")

  // via CMS
  compileOnly("org.codehaus.groovy:groovy-all:2.4.10")

  // via CMS
  //compileOnly("com.google.code.gson:gson:2.7")
}

java {
  toolchain {
    languageVersion.set(JavaLanguageVersion.of(8))
  }
}

//
// -- packaging --
//

val packageBatixPlugin by tasks.registering(Zip::class) {
  group = "build"
  dependsOn("cleanPackageBatixPlugin")

  val distDir = layout.buildDirectory.dir("dist")
  destinationDirectory.set(distDir)
```

```
  outputs.dir(distDir)

  from("plugin.yaml") {
    expand(
      mapOf(
        "id" to "${project.group}:${project.name}",
        "version" to project.version
      )
    )
  }

  from(tasks.named("jar")) {
    into("lib")
  }

  from(configurations.runtimeClasspath) {
    into("lib")
    exclude("slf4j-api-*.jar")
  }

  from("static") {
    into("static")
  }
}

tasks.named("assemble") {
  dependsOn(packageBatixPlugin)
}
```

Gehen wir diese Datei (in der Version für v2.9) mal auszugsweise durch. Da im Grunde alle Gradle-Projekte nach diesem Schema aufgebaut sind, ist dieser kleine Exkurs hoffentlich auch für andere Projekte hilfreich.

plugins

```
plugins {
  kotlin("jvm")
```

```
}
```

Da wir das Plugin in Kotlin schreiben, brauchen wir auch das Kotlin Gradle-Plugin, damit sich der Kotlin-Compiler in den Build-Prozess einklinkt. Die Version des Kotlin-Plugins wird hier nicht angegeben, diese wurde ja schon im Hauptprojekt definiert.

version

```
//version = "1.0.0"
```

Wie schon erwähnt wird die Version eigentlich anhand der Git-Tags vergeben. Hat man aber völlig verschiedenartige Unterprojekte, kann man hier auch die Version je Unterprojekt überschreiben. Da in unserem Fall die automatische Git-Version benutzt werden soll, ist diese Zeile auskommentiert.

repositories

```
repositories {
  mavenCentral()

  // ...
}
```

Hier werden alle Maven-Repositories angegeben, in denen nach Dependencies gesucht werden soll. `mavenCentral()` registriert dabei das weltweite Standard-Maven-Repo. Weitere `maven` Repos können hinzugefügt werden, falls Dependencies geladen werden, die nicht öffentlich sind - wie es z. B. bei der Framework-API (ab v2.9) der Fall ist, gegen die Plugins entwickelt werden.

> **Note**
> Der Zugriff auf das private Maven-Repo ist bei Batix zu erfragen.

dependencies

```
dependencies {
  implementation("org.jetbrains.kotlin:kotlin-stdlib")

  // ...
}
```

Jedes Unterprojekt deklariert hier die Dependencies, die es zur Kompilation oder zur Ausführung benötigt. Die Koordinaten der Dependencies (Gruppe, Artefakt, Version - durch Doppelpunkt

getrennt) sind auf den jeweiligen Projektseiten oder z. B. via mvnrepository.com zu ermitteln. Ein Link als Kommentar über der Dependency-Zeile, unter dem man die Versionen der Dependency sehen kann, ist hilfreich und sollte immer eingefügt werden (mvnrepository macht das automatisch, wenn man aus dem "Gradle (Kotlin)" Tab kopiert).

Die erste Dependency (`org.jetbrains.kotlin:kotlin-stdlib`) enthält die Kotlin Runtime, welche immer für Kotlin-Projekte benötigt wird. Hier ist keine Version angegeben. Diese wird automatisch vom Kotlin-Gradle-Plugin vorgegeben.

Die nächste Dependency (`com.batix:batix-cm:2.9.0.3`) ist die Framework-API. Hier sollte die Version verwendet werden, unter der auch das Framework läuft, in dem das Plugin dann benutzt wird.

Die anderen 3 Dependencies (`servlet-api`, `websocket-api` und `groovy-all`) sind Sachen, die zur Laufzeit zur Verfügung stehen, da die Runtime bzw. das Framework diese mitbringt.

> **Tip**
> Da die GSON-Bibliothek auch beim Framework mitgeliefert wird, kann diese `compileOnly` Dependency noch aktiviert werden (so muss diese Dependency nicht im Plugin mitgeliefert werden).

Danach können dann eigene Dependencies deklariert werden. Um beispielweise Jackson zu nutzen, fügt man folgende Zeilen im `dependencies` Block hinzu:

```
// https://mvnrepository.com/artifact/com.fasterxml.jackson.module/jackson-module-kotlin
implementation("com.fasterxml.jackson.module:jackson-module-kotlin:2.15.3")
```

Ein Gradle-Sync macht die neuen Klassen der IDE bekannt und diese werden dann auch von der Autovervollständigung vorgeschlagen.

> **Tip**
>
> Bei Gradle gibt es das Konzept von *Configurations*. Die Gradle Hilfe dazu geht ins Detail, hier sei nur das Wichtigste erwähnt.
> In einer Configuration werden mehrere Dependencies gesammelt. Es gibt verschiedene Configurations, in denen Dependencies gesammelt werden, welche zum Compilen und zur Laufzeit gebraucht werden, andere Configurations beschreiben Dependencies, die nur zum Compilen gebraucht werden und nicht mit ausgeliefert werden sollen. Je nach verwendeten Gradle-Plugins sind außerdem andere Configurations verfügbar (Beispiel Java Gradle Plugin).
> Die wichtigsten Configurations und deren Eigenschaften sind:
> - `implementation` - zum Compilen verfügbar, landet auch in Distributionen (z. B. Plugin ZIP)
> - `compileOnly` - nur zum Compilen verfügbar, wird nicht mit ausgeliefert
> - `runtimeOnly` - beim Compilen nicht verfügbar, wird aber mit ausgeliefert

> - `testImplementation` - zum Compilen von Tests verfügbar
> - `compile` (deprecated) - findet man noch in einigen alten Tutorials, ist meistens durch `implementation` zu ersetzen
>
> Baut man kein Framework-Plugin, sondern eine Library (also Code, der in anderen Projekten nachgenutzt werden kann) bringt das Java Library Gradle Plugin noch `api` mit, was `implementation` ähnelt. Der Unterschied ist, dass die Dependencies aus `api` auch im Library-Consumer sichtbar sind, die aus `implementation` allerdings nicht. Beide werden in Distributionen ausgeliefert.

JVM Version

```
java {
  toolchain {
    languageVersion.set(JavaLanguageVersion.of(21))
  }
}
```

Hier wird die JDK-Version festgelegt, mit der die Quellcodes kompiliert werden. Es entsteht Bytecode, der mit dieser Java-Version (und neueren, aber nicht älteren) kompatibel ist.

packageBatixPlugin

```
//
// -- packaging --
//

val packageBatixPlugin by tasks.registering(Zip::class) {
  //...
}

tasks.named("assemble") {
  dependsOn(packageBatixPlugin)
}
```

Dieser Block definiert einen eigenen Task namens "packageBatixPlugin". Er ist vom Typ `Zip`, erstellt also ein Archiv. `group = "build"` gibt an, unter welcher Gruppe der Task im IntelliJ Gradle Panel auftauchen soll.

Mittels `dependsOn("cleanPackageBatixPlugin")` wird gesagt, dass immer das Outputverzeichnis (welches mit `destinationDirectory` und `outputs` festgelegt wird) geleert werden soll, bevor die

Plugin-ZIP dort abgelegt werden soll. Das ist hilfreich, wenn sich beim Entwickeln die Version ändert, damit nicht mehrere ZIPs mit unterschiedlichen Versionen im Outputverzeichnis liegen (und man die falsche Datei erwischt).

Die `from` Blöcke beschreiben, welche Dateien in der ZIP landen sollen. Nebst der `plugin.yaml` (in der hier auch die Platzhalter ersetzt werden), wird noch die kompilierte `jar` Datei des Plugin-Projektes sowie die Dependencies, die zur Laufzeit nötig sind (`configurations.runtimeClasspath`) hinzugefügt. Falls es im Unterprojekt ein Verzeichnis `static` gibt, wird auch dieses in die ZIP gepackt.

Hauptklasse

Die Plugin-Hauptklasse muss von `com.batix.plugins.Plugin` abgeleitet werden. In ihr kann man die Methoden `load()` und (falls nötig) `unload()` überschreiben.

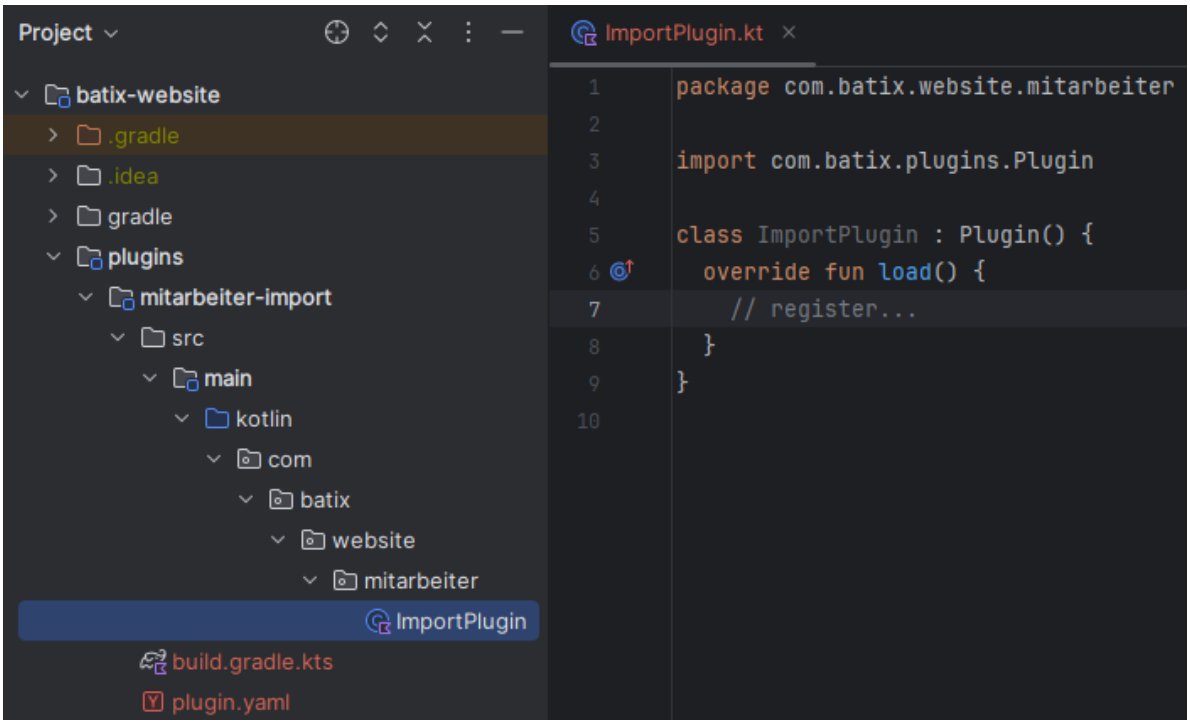
In `load()` können externe Dependencies oder interne Sachen initialisiert werden. Außerdem werden hier dem Application Framework die einzelnen Erweiterungen mitgeteilt, die das Plugin mitbringt.

In `unload()` müssen benutzte Ressourcen wieder aufgeräumt werden, das sind z. B. angelegte Threads, Listener oder Worker. Es ist die Anleitung der externen Dependencies zurate zu ziehen, wie man diese korrekt herunterfährt / aufräumt, falls nötig. Benutzt man beispielsweise Kotlin Coroutines, so kann hier `Dispatchers.shutdown()` aufgerufen werden.

> **Tip**
>
> Framework-Erweiterungen, welche in `load()` bekannt gemacht wurden (Service, Action, Tag, etc.), müssen nicht manuell unregistriert werden, dies erfolgt automatisch beim Entladen des Plugins.

> **Qualifizierter Name**
>
> Der qualifizierte Name der Klasse (also inklusive Package) muss dem entsprechen, was in der plugin.yaml unter `pluginClass` angegeben wurde. Ansonsten kann das Plugin vom Framework nicht geladen werden.

Im Falle des Beispiels wird also unter dem entsprechenden Pfad (hier ist das `src/main/kotlin/com/batix/website/mitarbeiter`) die Klasse `ImportPlugin` angelegt und von `com.batix.plugins.Plugin` abgeleitet.



Extensions

Framework-Erweiterungen wie z. B. Actions und Tags registriert man mit den entsprechenden `register*()` Methoden, also beispielsweise `registerService(serviceName, service)`. Diese werden auf den nächsten Seiten näher beschrieben. Passend dazu gibt es `unregister*()` Methoden, z. B. `unregisterService(serviceName)` und `unregisterAllServices()`, um die Extensions dynamisch entfernen zu können (ansonsten werden sie automatisch beim Plugin-Unload entfernt).

> **Tip**
> Extensions können, genau wie das gesamte Plugin, jederzeit im laufenden Betrieb aktiviert und deaktiviert werden. So kann man bestimmte Funktionen in Abhängigkeit von anderen Sachen (z. B. Konfigurationen) zur Verfügung stellen, oder auch nicht.

Logging

In der Hauptklasse sind ein paar Hilfsmethoden verfügbar, die zum Loggen benutzt werden können. Dabei wird automatisch der Plugin-Titel vorangestellt, sodass man die Nachrichten dem entsprechenden Plugin zuordnen kann.

```
fun logD(msg: String)
fun logI(msg: String)
fun logN(msg: String)
fun logW(msg: String)
```

```
fun logE(msg: String)
fun logE(msg: String, ex: Exception)
```

Diese können innerhalb der Hauptklasse einfach benutzt werden.

```
logI("Kotlin version: ${KotlinVersion.CURRENT}")
```

Möchte man diese Methoden auch an anderen Stellen wie den Extension-Klassen verwenden, so sollte die Plugin-Instanz an diese Klassen weitergereicht werden.

Preferences / Storage

Mittels `storage` (`getStorage()`) bekommt man eine für das aktuelle Plugin gültige Instanz von `PluginStorage`. Die darin abgelegten Sachen überstehen einen Reload des Plugins sowie einen Neustart des Frameworks. Ein Plugin hat nur Zugriff auf seine eigenen Preferences (identifiziert anhand der `id` des Plugins).

> **Benutzung**
> `storage` ist erst initialisiert, sobald die `load()` Methode aufgerufen wird. Vorher (also z. B. bei der Initialisierung von Feldern in der Plugin-Klasse) darf `storage` noch nicht benutzt werden.

Für alle primitiven Typen gibt es jeweils eine `set*` und `get*` Methode, über die Einstellungen anhand eines String-Keys abgelegt und wieder geholt werden können.

```
fun getStringPref(key: String): String
fun setStringPref(key: String, value: String)

fun getBooleanPref(key: String): Boolean
fun setBooleanPref(key: String, value: Boolean)

// usw. für die anderen primitiven Typen
```

Diese Methoden können unter `storage` benutzt werden.

```
var startCount = storage.getIntegerPref("startCount") ?: 0
storage.setIntegerPref("startCount", ++startCount)
logI("This is start #$startCount")
```

Falls komplexere Objekte wie Maps oder eigene Datenklassen abgelegt werden sollen, können die `*ObjectPref` Methoden benutzt werden. Zu beachten ist, dass diese Objekte nicht zu komplex sein

dürfen, da sie im JSON-Format abgelegt werden (wenige MB verfügbar je Preference).

```
data class StartupInfo(val count: Int)


val prevStartupInfo = storage.getObjectPref("startupInfo", StartupInfo::class.java)
logI("prevStartupInfo: $prevStartupInfo")
storage.setObjectPref("startupInfo", StartupInfo(startCount))
```

Um Einstellungen zu löschen, gibt es `clear*` Methoden.

```
fun clearPref(key: String)
fun clearAllPrefs()
```

Falls größere Mengen oder Binärdaten gespeichert werden müssen, oder ein temporäres Verzeichnis gebraucht wird, kann `storage.dataDir` (`getStorage().getDataDir()`) benutzt werden. Hier erhält man einen `Path`, der auf ein (für das aktuelle Plugin gültige) Verzeichnis auf der Festplatte zeigt, in dem das Plugin Dateien / Ordner ablegen kann. Dieses Verzeichnis überdauert auch einen Neustart.

> **Plugin id Abhängigkeit**
> Wenn sich die `id` eines Plugins ändert, sind die zuvor gesetzten Preferences und abgelegten Dateien nicht mehr zugänglich! Eine Änderung der Version des Plugins ist davon nicht betroffen.

Revision #1
Created 9 May 2025 06:19:10 by Batix
Updated 9 May 2025 06:19:11 by Batix