Batix Plugins

- IDE Setup
- Plugin
- Extensions
  - Service
  - Action
  - Tag
  - Timer Job
  - Request Interceptor
  - Static Content
  - Admin Page
  - WebSocket
  - Event Listener
- Guides
  - Vue App

IDE Setup

Um ein Plugin zu erstellen, ist eine IDE, also eine Entwicklungsumgebung, nötig. Wir empfehlen IntelliJ IDEA von JetBrains (die kostenlose Community Edition reicht aus). Im Zusammenspiel mit dem Build Tool Gradle und der hier bereitgestellten Konfiguration, kann in nur wenigen Schritten direkt losentwickelt werden!

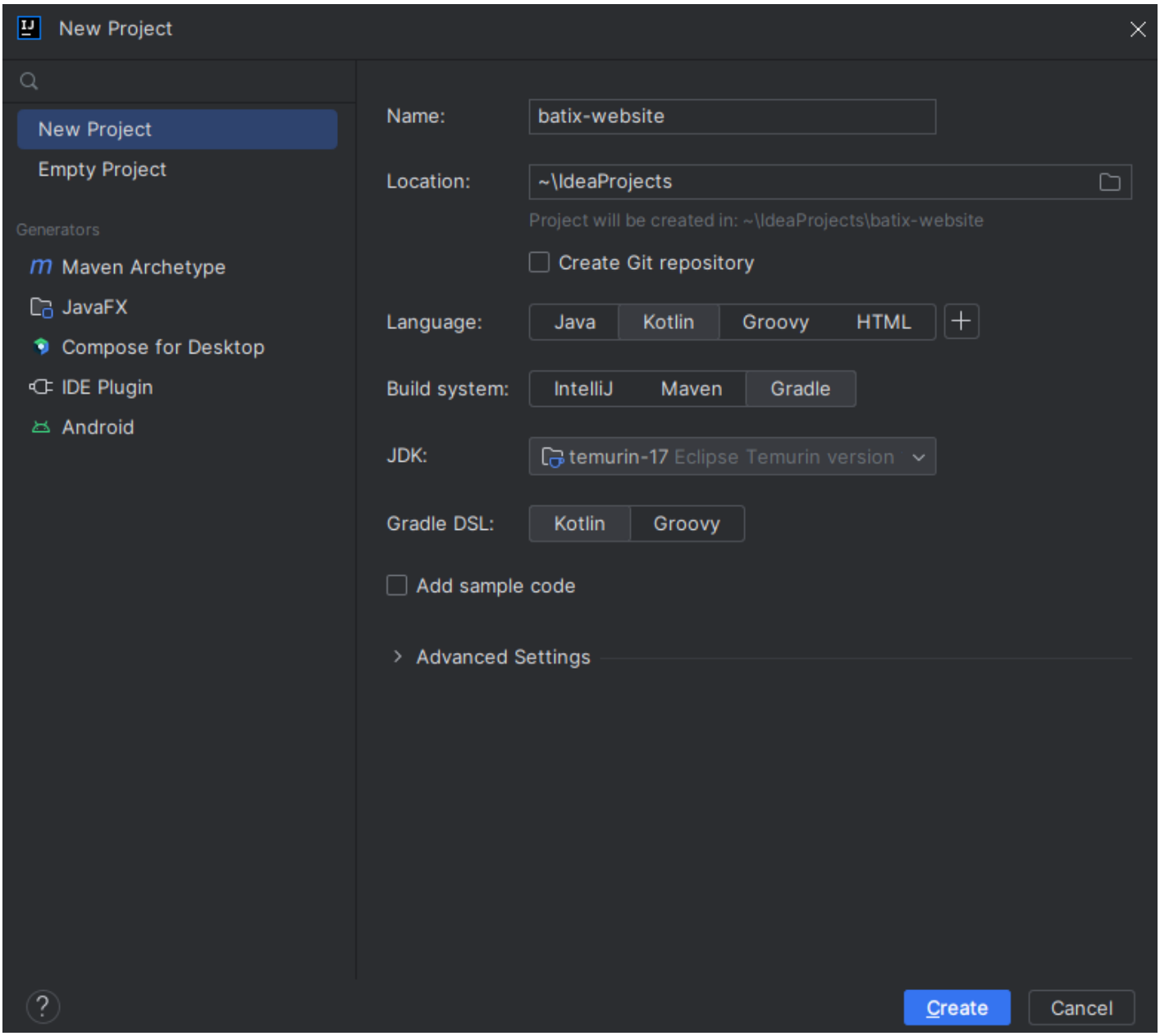
Erfahrene Entwickler können natürlich auch andere IDEs oder sogar die Kommandozeile verwenden. Hier wird nur der Weg für IntelliJ beschrieben.

> **Note**
> Die Beispiele sind für Gradle 8.12 geschrieben. Andere Gradle Versionen benötigen ggf. Anpassungen.

Projekt anlegen

Nach dem Starten von IntelliJ erscheint entweder ein Dialogfenster (*Welcome to IntelliJ IDEA*) oder es wird das zuletzt geöffnete Projekt wiederhergestellt. Um ein neues Projekt zu beginnen, entweder im Dialogfenster auf *New Project* klicken oder über das Menü gehen: *File -> New -> Project...*.

Bei *New Project* gibt man dem Projekt einen Namen (keine Sonderzeichen verwenden). Das kann z. B. *batix-website* sein. *Create Git Repository* und *Add sample code* sollten nicht angehakt sein, das erledigen wir später manuell.



Unter *Language* wird *Kotlin* gewählt, das *Build system* wird auf *Gradle* gestellt (*Gradle DSL* auf *Kotlin* stellen).

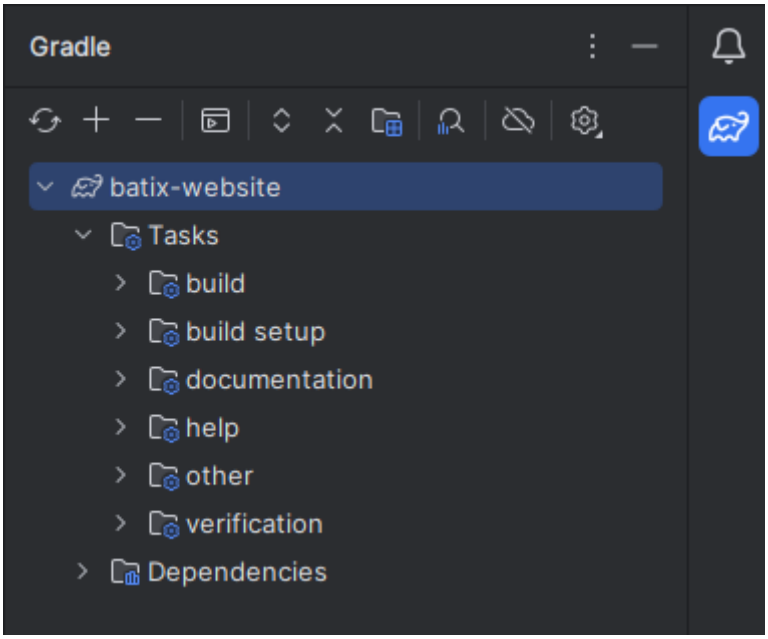
> **Projektstruktur**
> Es wird zunächst ein leeres Projekt erstellt, welches dann um Unterprojekte ergänzt wird. Auch wenn es nur ein Unterprojekt gibt, hat diese Struktur Vorteile, wie z. B. Deduplizierung von Build-Logik.

Bei *JDK* ist die korrekte Java-Version zu wählen. Diese ist je nach Framework-Version unterschiedlich:

- Framework älter als v2.9: JDK 1.8
- Framework ab v2.9: JDK 17
- Framework ab v3.0: JDK 21

Falls es bei *JDK* noch kein Java SDK in der gewünschten Version gibt, kann dort im Dropdown mittels *Add SDK* und dann *Download JDK...* automatisch ein JDK heruntergeladen werden. Bei *Version* wird die entsprechende Java-Version und als *Vendor* wird *Eclipse Temurin (AdoptOpenJDK Hotspot)* gewählt. Nach Klick auf *Download* startet der Download im Hintergrund. Ist dieser fertig, kann der Assistent fortgesetzt werden.

Nach Klick auf *Create* legt IntelliJ das Projekt an und initialisiert das Gradle Buildsystem. Der Fortschritt wird ganz unten in IntelliJ angezeigt. Dies kann vor allem beim ersten Mal etwas dauern. Man kann Gradle und IntelliJ jederzeit neu synchronisieren, indem man auf den Sync Button mit den zwei Pfeilen drückt, rechts im Gradle Panel. Dies ist manchmal nötig, wenn man Änderungen an den Buildscripts macht oder das Projekt neu öffnet.



Projekt anpassen

Sobald IntelliJ damit fertig ist, kann das Projekt weiter angepasst werden.

Dotfiles

"Dotfiles" sind Dateien, die mit einem Punkt anfangen. Diese Dateien enthalten meist Einstellungen und werden manchmal auch standardmäßig ausgeblendet. Bitte die folgenden Dateien einfach in das Projektverzeichnis speichern:

- .editorconfig (legt Sachen wie Einrückung und Charset der Quellcode-Dateien fest)
- .gitignore (listet Dateien, die nicht im Git-Repo landen sollen, z. B. kompilierte Klassen)
- .gitattributes (legt fest, dass Zeilenumbrüche automatisch konvertiert werden)

Diese Vorlagen decken schon viel ab, es kann aber natürlich immer Anpassungsbedarf geben. Meistens ist das bei `.gitignore` der Fall - dort sind beispielsweise zusätzliche Sachen zu ergänzen, die nicht von Git getrackt werden sollen (meistens sind das Build-Outputs, Binaries oder generierte Quellcodes). In manchen Fällen müssen auch Einträge entfernt werden, wenn im Git-Repo z. B. bestimmte IDE-Config-Dateien mit liegen sollen.

settings.gradle.kts

Die Datei `settings.gradle.kts` im Hauptverzeichnis des Projekts muss so aussehen:

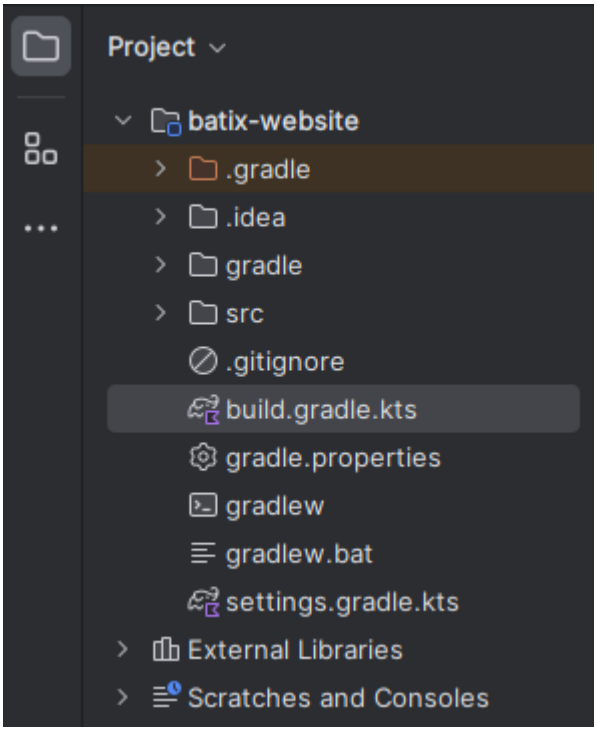
```
plugins {
  // https://plugins.gradle.org/plugin/org.gradle.toolchains.foojay-resolver-convention
  id("org.gradle.toolchains.foojay-resolver-convention") version "0.9.0"
}

rootProject.name = "batix-website"
```

Das angegebene Plugin dient dem automatischen Herunterladen eines JDKs in der gewünschten Java-Version. Hier kann auch der Name des Projekts, wie er z. B. in der IDE angezeigt wird, geändert werden.

build.gradle.kts

Nun öffnet man die Datei `build.gradle.kts`, welche direkt im Projektverzeichnis liegt.



Der komplette Inhalt der Datei wird mit folgenden Zeilen ersetzt:

```
plugins {
  base

  // https://kotlinlang.org/docs/releases.html#release-details
  kotlin("jvm") version "2.1.0" apply false
}

//
// -- determine version from git --
//

fun determineVersionFromGit(): String {
  val stdOut = try {
    providers.exec {
      commandLine(
        "git", "describe", // find the most recent tag and derive a version string from it
        "--dirty", // append -dirty if there are local modifications
        "--tags", // also use lightweight tags (in addition to annotated tags)
        "--match", "v*.*.*", // only consider tags in the form vx.y.z
        "--always" // just use the abbreviated commit if no tags are found
      )
    }.standardOutput.asText.get()
  } catch (e: Exception) {
    project.logger.warn("Cannot determine version via git describe, using 'unknown'.", e)
    return "unknown"
  }

  return stdOut.trim().replace(Regex("^v"), "")
}

var determinedVersion = determineVersionFromGit()

val printVersion by tasks.registering {
  group = "help"
  description = "Prints the current version as calculated by determineVersionFromGit()."

  doLast {
    println(determinedVersion)
```

```
  }
}

allprojects {
  group = "my.company.project"
  version = determinedVersion
}
```

Hier kann dann noch die gewünschte Kotlin Version angepasst werden. Die neuesten Versionen von Kotlin findet man unter dem Link, der auch oben als Kommentar steht.

> **Kotlin Version**
> Die zu benutzende Kotlin Version wird hier zentral für alle Unterprojekte festgelegt, damit diese alle dieselbe Version des Kotlin-Compilers und der Kotlin-Runtime benutzen. Diese Vorgehensweise sollte bei allen Gradle-Plugins verwendet werden, die in mehreren Unterprojekten benutzt werden, oder die zwingend auch im Hauptprojekt deklariert werden müssen (z. B. Spring).

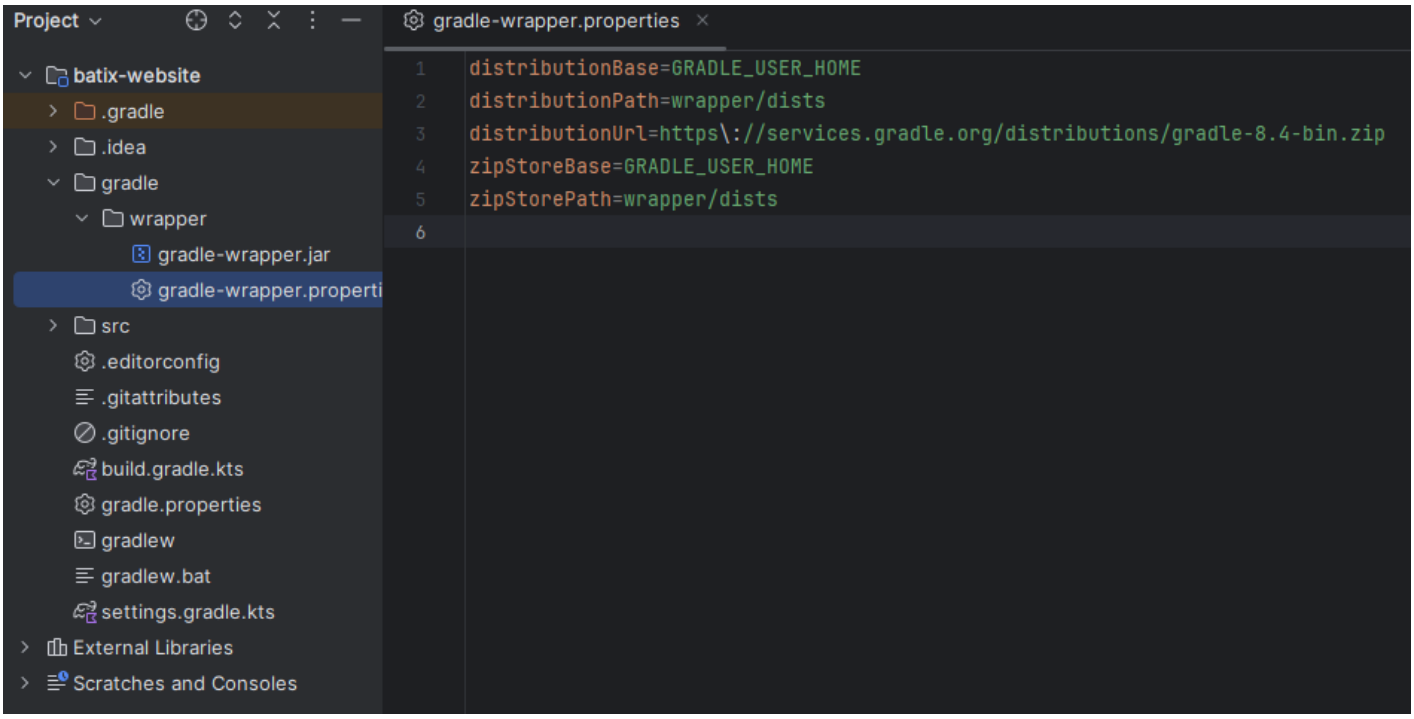
> **Group**
> Es muss auf jeden Fall die `group` Zeile angepasst werden. Diese sollte zur Eindeutigkeit einer Domain des Unternehmens entsprechen, im Falle von Batix könnte die Zeile also lauten:

```
group = "com.batix.website"
```

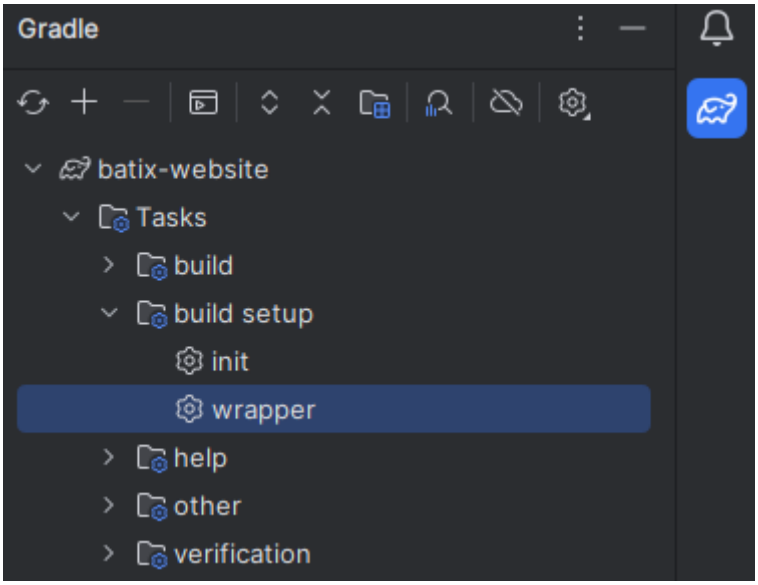
Der große *determine version from git* Block sorgt dafür, dass die Version (welche die Artefakte beim Build-Prozess bekommen) automatisch aus dem Git-Repo erzeugt wird. Dabei werden Git-Tags benutzt, die im Format `v1.2.3` sein müssen. Besonders bei Libraries bietet es sich an, Semantic Versioning zu benutzen. Um die aktuelle Version auszugeben, kann der Task `printVersion` benutzt werden.

gradle-wrapper.properties

Es sollte auch nach der benutzten Gradle Version geschaut werden. Diese steht in der Datei `gradle/wrapper/gradle-wrapper.properties`. Die neuesten Versionen von Gradle findet man unter gradle.org/releases/.



Die Änderungen übernimmt man durch Klick auf den bereits erwähnten Sync Button  im Gradle Panel rechts oder durch Klick auf Gradle-Button im Editor rechts  (nur sichtbar in bestimmten Dateien). Das Tastenkürzel unter Windows dafür ist `Ctrl+Shift+O`. Wurde die Gradle-Wrapper Version angepasst, sollte auch direkt der `wrapper` Gradle Task (unter *build setup*) ausgeführt werden, damit die Gradle Wrapper Dateien im Verzeichnis aktualisiert werden.



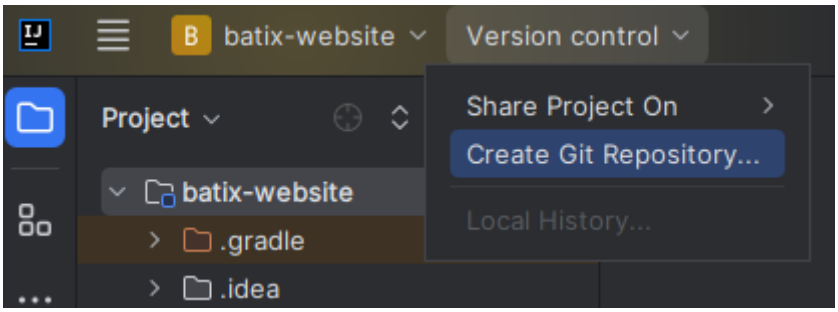
> **Gradle Wrapper**
>
> In der Gradle Dokumentation gibt es eine ausführliche Beschreibung, was der Gradle Wrapper ist und warum man ihn benutzen sollte. Kurz gesagt: so ist sichergestellt, dass jeder Entwickler und auch Sachen wie Continuous Integration (CI) dieselbe Gradle Version verwenden, um Fehler, die z. B. aus Inkompatibilitäten zwischen der Gradle-Version und den eingesetzten Gradle-Plugin-Versionen entstehen können, auszuschließen und

reproduzierbare Builds zu ermöglichen.

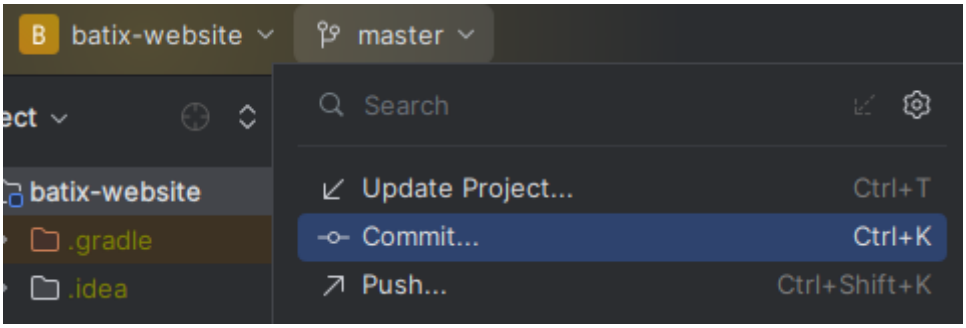
Git Repository

Versionsverwaltung
Es gehört mittlerweile zum Standard, Quellcodes in einem Versionsverwaltungssystem abzulegen, um gemeinsames, verteiltes Entwickeln besser zu ermöglichen und Änderungen nachzuverfolgen. Das ist auch bei Hobby- oder Test-Projekten sinnvoll - so kann man experimentieren und einfach den vorherigen Code-Stand aller oder bestimmter Dateien vergleichen und wiederherstellen.
Um die Quellcode-Dateien zu tracken, muss zunächst ein Git Repository (kurz *Repo*) angelegt werden. Dieses speichert u. a. alle abgelegten Änderungen (*Commits*) und Entwicklungslinien (*Branches*). Diese Daten können dann an einen zentralen Server geschickt (*Push*) und von dort auch Updates von anderen Entwicklern abgeholt (*Fetch*, *Pull*) werden. Dies macht Git zu einem *verteilten* Versionsverwaltungssystem, da sich jeder Entwickler eine komplette Historie des Repos lokal speichern, unabhängig von anderen Entwicklern Commits tätigen und Änderungen jederzeit synchronisieren kann.

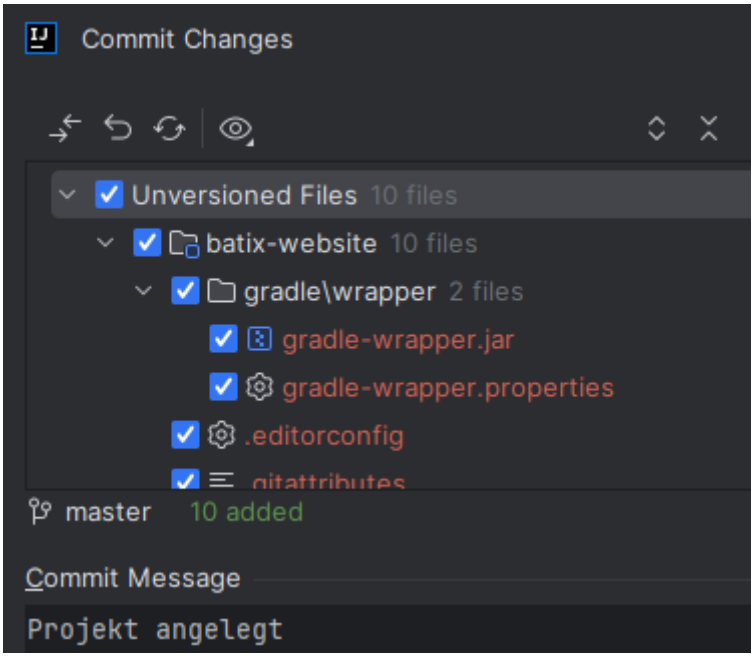
In der Dateiliste links wird das Projektverzeichnis angeklickt, damit dieses ausgewählt ist (hier soll das Git Repo erstellt werden). Dann kann über das Menü *Version control > Create Git Repository...* ein Git Repo angelegt werden. Im sich öffnenden Dialog sollte nochmals geprüft werden, ob auch wirklich das Projekthauptverzeichnis ausgewählt ist.



Es empfiehlt sich, den initialen Projektstand als Git Commit festzuhalten. Dazu klickt man auf *Commit...* im VCS-Menü oder benutzt unter Windows die Tastenkombination `Ctrl+K`.

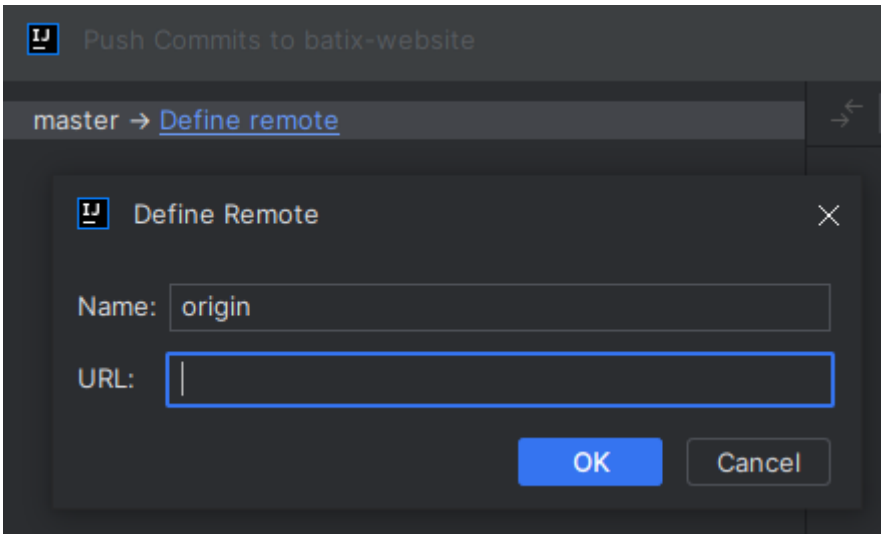


Es erscheint der Dialog *Commit Changes*. Hier müssen alle Dateien angehakt, eine kurze Beschreibung der Änderungen eingetragen und dann auf *Commit* geklickt werden.

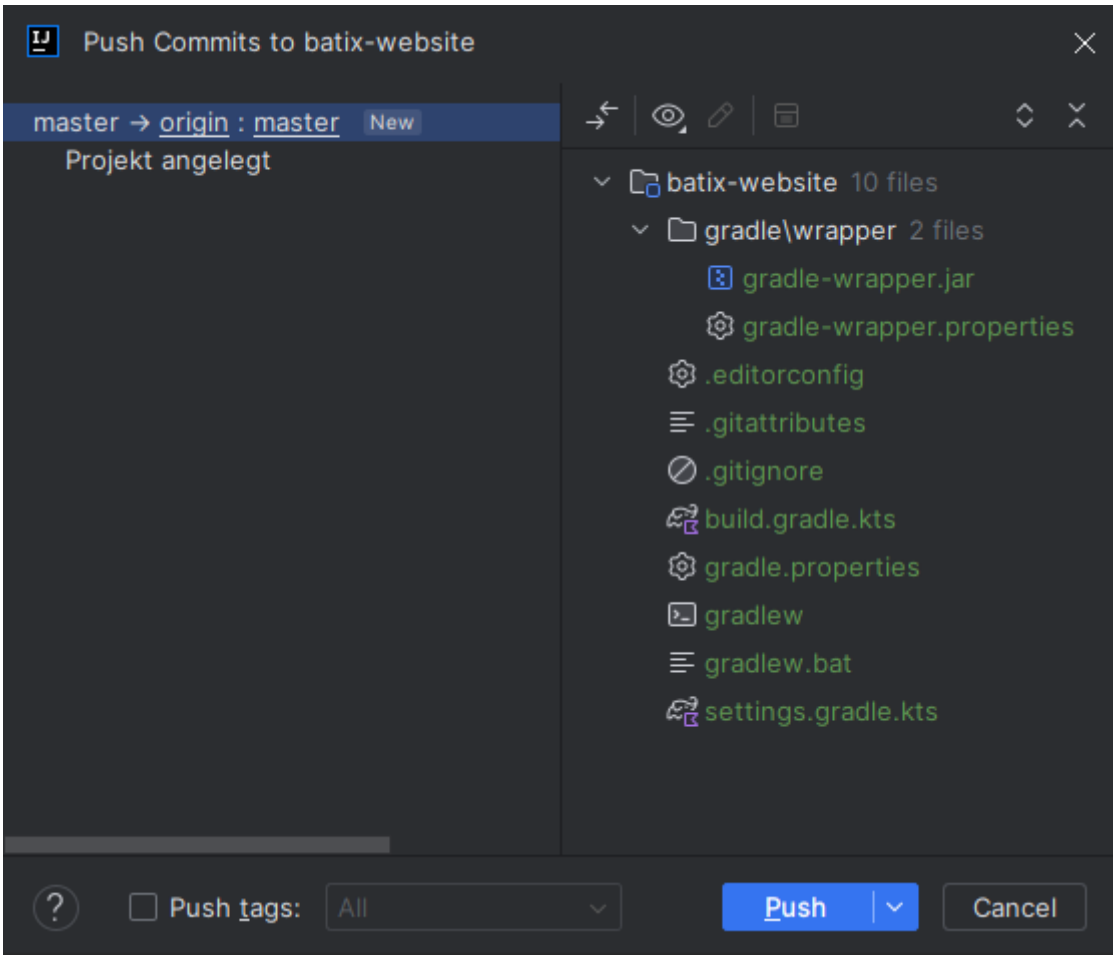


Man sollte nun regelmäßig weitere Commits machen, um später seine Änderungen noch nachverfolgen und zu früheren Code-Ständen zurückspringen zu können. Wann genau ein neuer Commit gemacht wird, ist Geschmackssache, es sollten aber nicht zu viele Änderungen in einen Commit einfließen - lieber kleine, in sich größtenteils abgeschlossene, Häppchen bevorzugen.

Um die eigenen Commits an den zentralen Server zu senden, wird der Eintrag *Push* direkt unterhalb von *Commit* (oder die Tastenkombination `Ctrl+Shift+K`) benutzt. Wurde das Git Repo lokal angelegt, muss zunächst noch die URL zum Remote-Server angegeben werden.



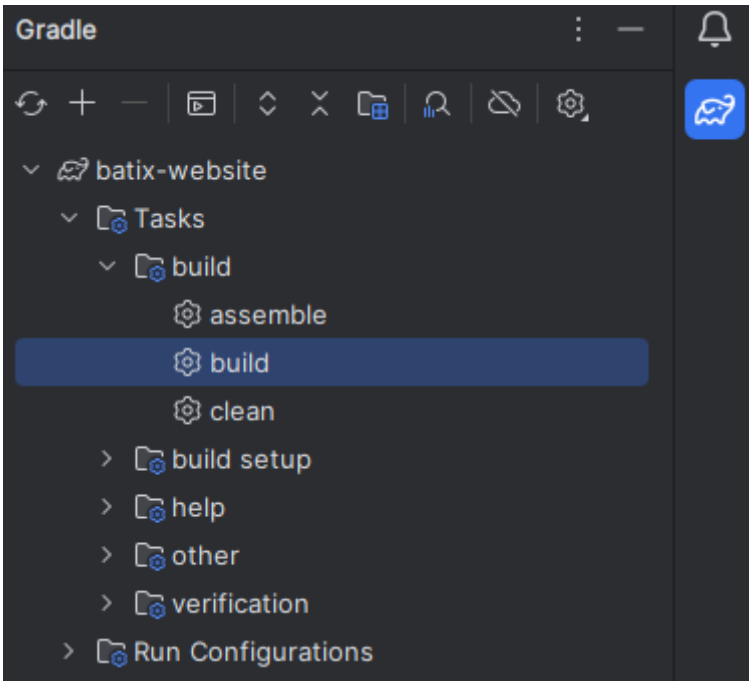
Ist dies erledigt oder wurde das Git Repo nicht lokal erzeugt, sondern initial vom Remote-Server geladen (*Clone*), werden die zu übertragenen Commits angezeigt.



Mit Klick auf *Push* werden diese übertragen und stehen ab dann anderen Entwicklern zur Verfügung.

Build

Jetzt kann der *build* Task gestartet werden, um zu überprüfen, ob alles korrekt eingerichtet ist.



Es sollte folgender Build Output angezeigt werden:

```
Executing 'build'...

> Task :assemble UP-TO-DATE
> Task :check UP-TO-DATE
> Task :build UP-TO-DATE

BUILD SUCCESSFUL in 214ms
Execution finished 'build'.
```

Gradle Tasks
Im Wesentlichen führt Gradle einfach nur Tasks aus. Diese Tasks können von verschiedenster Art sein (Code überprüfen, Code compilen, Tests ausführen, Archive packen, ...) und sich untereinander bedingen. Je nach (Unter-)Projekt-Typ und eingesetzten Gradle-Plugins werden unterschiedliche Tasks bereitgestellt. Es können auch selbst Tasks definiert werden, wie wir später noch sehen werden.

Der `build` Task ist sozusagen der "Über-Task", der diverse andere Tasks enthält. Man kann es sich wie eine Baum-Struktur vorstellen. Der `build` Task teilt sich z. B. in vielen Projekten in diese Sub-Tasks auf (Auszug):
- `assemble` (Archive und Distributionen zusammenbauen)
  - `jar` (die .jar Datei erzeugen)
    - `classes` (Quellcodes kompilieren)
      - `compileJava` (Java-Quellcodes kompilieren)
      - `compileKotlin` (Kotlin-Quellcodes kompilieren)
      - `processResources` (Ressourcen-Dateien zusammenstellen)

- `check` (Projekt überprüfen)
  - `test` (Unit-Tests ausführen)
    - `testClasses` (Test-Quellcodes kompilieren)
      - `compileTestJava` (Java-Test-Quellcodes kompilieren)
      - `compileTestKotlin` (Kotlin-Test-Quellcodes kompilieren)
      - `processTestResources` (Test-Ressourcen-Dateien zusammenstellen)
  - An dieser Stelle könnten z. B. auch noch Linter-Tasks angesiedelt werden

Die einzelnen Tasks können natürlich auch separat aufgerufen werden. So kann z. B. direkt `test` gestartet werden (der aber dann wiederum mindestens die Compile-Tasks bedingt, auch die oberen wie `compileKotlin`).

Tasks aus Unterprojekten werden auch bei allen Über-Projekten angezeigt. Startet man z. B. im Hauptprojekt den Task `test`, so wird dieser auch in jedem Unterprojekt ausgeführt, in dem es diesen Task gibt.

Projekt-Referenzen und Tasks werden in Gradle durch Doppelpunkte getrennt. Gibt es z. B. ein Unterprojekt im Ordner `plugins/mitarbeiter-import`, so kann dessen `test` Task als `:plugins:mitarbeiter-import:test` angesprochen werden.

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.

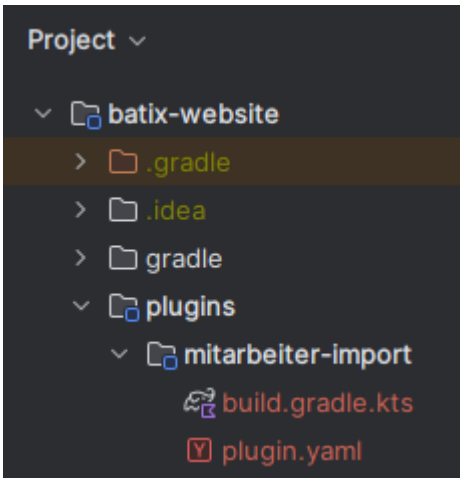
Beim Builden wird das Plugin in ein ZIP-Archiv verpackt, das nebst dem Code und den Dependencies auch Metadaten zum Plugin wie Titel und Version enthält.

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 v3.3

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

//version = "1.0.0"

repositories {
  mavenCentral()

  val batixUrls = listOf(
    "https://git.batix.gmbh/api/v4/projects/477/packages/maven", // CMS-API
    "https://git.batix.gmbh/api/v4/projects/322/packages/maven", // Plugin-API
    "https://git.batix.gmbh/api/v4/projects/324/packages/maven",
  )

  batixUrls.forEach { url ->
    maven {
```

```
      this.url = uri(url)
      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:3.3.0")

  // via Tomcat 11.0
  compileOnly("jakarta.servlet:jakarta.servlet-api:6.1.0")

  // via Tomcat 11.0
  compileOnly("jakarta.servlet.jsp:jakarta.servlet.jsp-api:4.0.0")

  // via Tomcat 11.0
  compileOnly("jakarta.websocket:jakarta.websocket-api:2.2.0")

  // via Tomcat 11.0
  compileOnly("jakarta.websocket:jakarta.websocket-client-api:2.2.0")
```

```
  // via CMS
  compileOnly("org.apache.groovy:groovy:4.0.32")
  compileOnly("org.apache.groovy:groovy-dateutil:4.0.32")
  compileOnly("org.apache.groovy:groovy-json:4.0.32")
  compileOnly("org.apache.groovy:groovy-sql:4.0.32")
  compileOnly("org.apache.groovy:groovy-xml:4.0.32")

  // via CMS
  //compileOnly("com.google.code.gson:gson:2.14")
}

java {
  toolchain {
    languageVersion.set(JavaLanguageVersion.of(25))
  }
}

//
// -- 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 v3.0

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

//version = "1.0.0"

repositories {
  mavenCentral()

  val batixUrls = listOf(
    "https://git.batix.gmbh/api/v4/projects/477/packages/maven", // CMS-API
    "https://git.batix.gmbh/api/v4/projects/322/packages/maven", // Plugin-API
    "https://git.batix.gmbh/api/v4/projects/324/packages/maven",
  )

  batixUrls.forEach { url ->
    maven {
      this.url = uri(url)
      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:3.2.20")



  // 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()

  val batixUrls = listOf(
    "https://git.batix.gmbh/api/v4/projects/477/packages/maven", // CMS-API
    "https://git.batix.gmbh/api/v4/projects/322/packages/maven", // Plugin-API
    "https://git.batix.gmbh/api/v4/projects/324/packages/maven",
  )

  batixUrls.forEach { url ->
    maven {
      this.url = uri(url)
      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:2.9.3.10")

  // 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(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)
}
```

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.

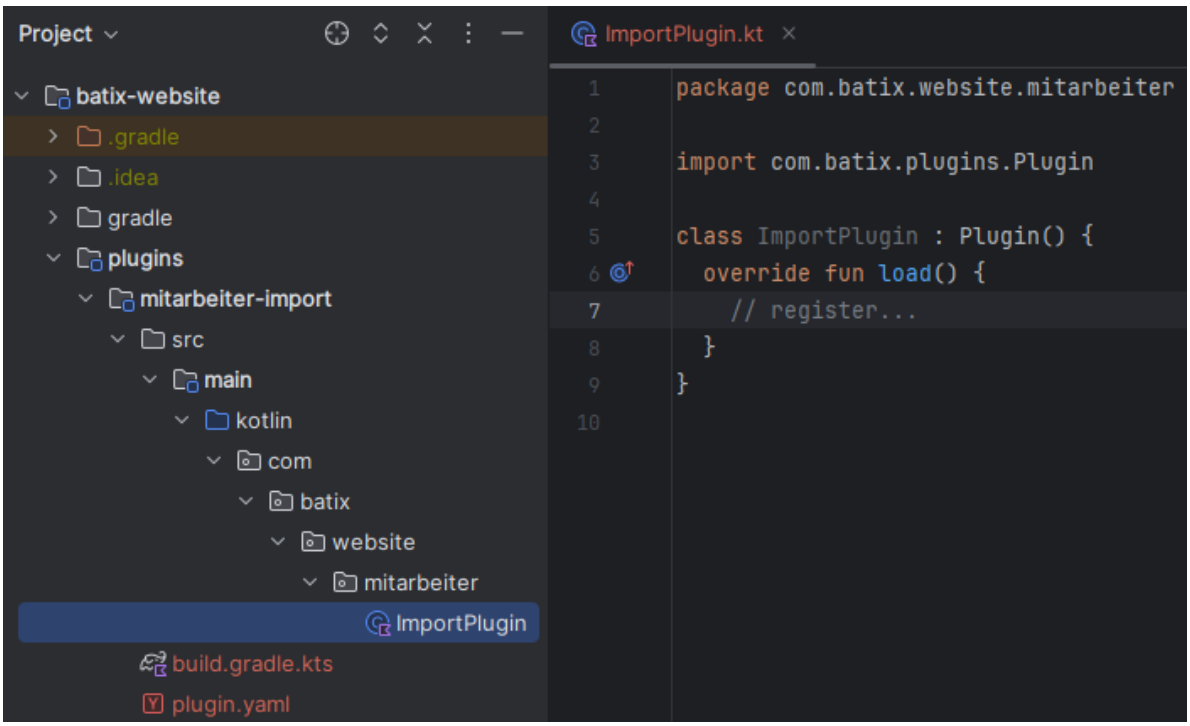
In `load()` können externe Dependencies oder interne Sachen initialisiert werden. Außerdem werden hier dem Application Framework die einzelnen Erweiterungen mitgeteilt, die das Plugin mitbringt.

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.

Extensions

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.

Plugin: Mitarbeiterimport-Plugin

Mitarbeiter importieren *(Funktion noch in Testphase)*
Importiert Mitarbeiter aus der externen Datenquelle.

Doku-Seite

Eigenschaften

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)}">""")

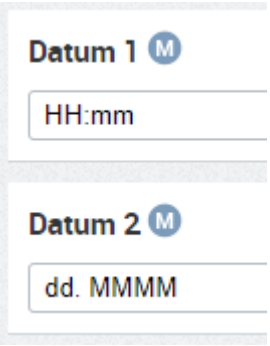
    return sb
  }
}
```

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"))
  )
}
```

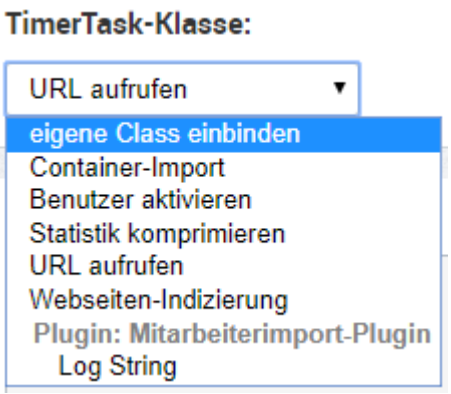
Diese Beispiel-Job-Klasse liest die eingestellten Parameter und loggt die festgelegte Nachricht mit dem entsprechenden Level.

```
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:

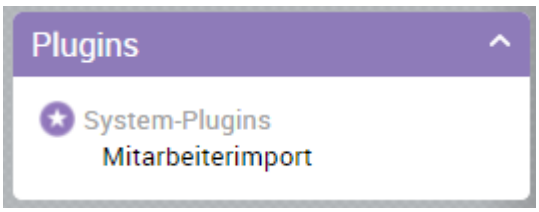
- `resource=""` - um zu prüfen, ob die Admin Page dem Backend-Benutzer in der Navigation angezeigt werden soll oder nicht. Wird `true` zurückgegeben, taucht die Admin Page in der Navigation nicht auf.
- `resource="index.html"`, `resource="assets/style.css"`, `resource="dynamic/do-something-action"` - 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())
  }
}
```

Guides

Vue App

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

Vue.js Anwendungen bestehen aus JavaScript, HTML und CSS. Singe File Components bieten sogar die Möglichkeit diese drei Sachen für eine Komponente in einer einzigen .vue Datei zu definieren. Das hilft Ordnung zu schaffen, benötigt aber einen Build-Step, d. h. es muss aus dieser .vue Datei erst etwas generiert werden, womit der Browser etwas anfangen kann.

Es ist zwar auch möglich Vue-Templates (der HTML-Teil einer Komponente oder App mit Platzhaltern) zur Laufzeit im Browser compilen zu lassen, dies geht aber auf die Performance und bläht das finale JavaScript auf, da der Compiler mitgeliefert werden muss. Besser ist es also, schon zur Build-Zeit diese aufwendigen Schritte zu durchlaufen. Außerdem stehen so noch weitere Features zur Verfügung, wie z. B. npm-Dependencies oder CSS-Transformatoren.

Die Seite Static Content beschreibt die Möglichkeit mittels Plugins, fertig generierte, statische Dateien unter einem bestimmten Pfad auszuliefern. Wie diese Dateien zur Build-Zeit erzeugt werden können, beschreibt dieser Guide.

Vorbereitungen

Auf dem Entwicklungs-Computer wird eine aktuelle Node.js Version benötigt, um eine neue Vue App anzulegen. Die eingesetzte IDE (z. B. VS Code oder IntelliJ IDEA) sollte auch aktuell sein.

Außerdem muss schon das Grundgerüst eines Plugins vorhanden sein (siehe IDE Setup und Plugin). Ein Ordner names "static" muss nicht angelegt werden, da Gradle so konfiguriert wird, dass es beim Anlegen der .zip Datei die Dateien selbst vom richtigen Ort holt, nachdem diese dort erstellt wurden.

Vue App erzeugen

Das Erzeugen der Vue-Anwendung ist nicht Teil dieses Guides. Es sei an dieser Stelle auf die offizielle Vue Doku verwiesen. Im weiteren Text wird davon ausgegangen, dass die Vue-App im Ordner `vue/import-frontend` (ausgehend vom Hauptprojekt) erzeugt wurde. Es wird hier keine `build.gradle.kts` Datei erzeugt, da die Vue-App nicht als separates Gradle-Unterprojekt angelegt wird, sondern die Build-Steps direkt als Tasks im entsprechenden Plugin angelegt werden.

Build Tasks

In der `build.gradle.kts` Datei des Plugins, welches die Vue-Anwendung ausliefern soll, wird folgender Block ergänzt:

```
//
// -- node --
//

val npmInstall by tasks.registering(Exec::class) {
  val nodeProjectDir = "$rootDir/vue/import-frontend"
  workingDir = file(nodeProjectDir)

  inputs.file("$nodeProjectDir/package.json")
  inputs.file("$nodeProjectDir/package-lock.json")

  outputs.dir("$nodeProjectDir/node_modules")

  val isWindows = org.apache.tools.ant.taskdefs.condition.Os.isFamily(
    org.apache.tools.ant.taskdefs.condition.Os.FAMILY_WINDOWS
  )
  commandLine(
    if (isWindows) "npm.cmd" else "npm",
    "install"
  )
}

val npmRunBuild by tasks.registering(Exec::class) {
  val nodeProjectDir = "$rootDir/vue/import-frontend"
  workingDir = file(nodeProjectDir)

  group = "build"
```

```
  dependsOn(npmInstall)

  inputs.dir("$nodeProjectDir/public")
  inputs.dir("$nodeProjectDir/src")
  inputs.file("$nodeProjectDir/package.json")
  inputs.file("$nodeProjectDir/package-lock.json")
  inputs.file("$nodeProjectDir/vite.config.js")

  outputs.dir("$nodeProjectDir/dist")

  val isWindows = org.apache.tools.ant.taskdefs.condition.Os.isFamily(
    org.apache.tools.ant.taskdefs.condition.Os.FAMILY_WINDOWS
  )
  commandLine(
    if (isWindows) "npm.cmd" else "npm",
    "run",
    "build"
  )

  doFirst {
    createVersionFile()
  }
}

fun createVersionFile() {
  val envVersionFile = Paths.get("$nodeProjectDir/env/.env.local.version")
  if (envVersionFile.parent != null) {
    Files.createDirectories(envVersionFile.parent)
  }
  val envVersionFileContent = "VERSION=${project.version}"
    .replace("dirty", "d") // "-dirty" → "-d" damit sich kein Frontend user wundert
  Files.write(envVersionFile, envVersionFileContent.toByteArray())
}
```

Die beiden hier definierten Tasks sind vom Typ `Exec`, führen im Hintergrund also einfach das entsprechende `npm` Command aus. Liegt die Vue-App in einem anderen Verzeichnis, müssen lediglich die zwei `nodeProjectDir` Zeilen (6 & 24) angepasst werden.

Besonderes Augenmerk muss auf die `inputs` Zeilen im zweiten Task (`npmRunBuild`) gelegt werden. Hier sind alle Dateien und Verzeichnisse aufzuführen, die dafür sorgen, dass sich die Vue-Anwendung ändert. Das sind also beispielsweise JS/Vue Quellcodes, CSS-Dateien, statische Assets,

aber auch Dateien, die zur Build-Konfiguration der Vue-App genutzt werden. Die Liste der Inputs darf ruhig länger sein, Hauptsache alle Dateien sind erfasst, damit Gradle die App auch bei Änderungen neu baut. Hier aufgeführte Dateien müssen existieren, sonst meldet Gradle einen Fehler.

Bei Quasar-Projekten muss das `outputs.dir` zu `"$nodeProjectDir/dist/spa"` geändert werden.

Längeres Beispiel für Inputs

```
    inputs.file("$nodeProjectDir/.env.development")
    inputs.file("$nodeProjectDir/.env.development.local")
    inputs.file("$nodeProjectDir/.eslintignore")
    inputs.file("$nodeProjectDir/.eslintrc.js")
    inputs.file("$nodeProjectDir/.postcssrc.js")
    inputs.file("$nodeProjectDir/babel.config.js")
    inputs.file("$nodeProjectDir/jsconfig.json")
    inputs.file("$nodeProjectDir/package.json")
    inputs.file("$nodeProjectDir/package-lock.json")
    inputs.file("$nodeProjectDir/quasar.conf.js")
    inputs.file("$nodeProjectDir/quasar.extensions.json")
```

Um die Version des Git-Tags im Vue-Projekt auslesen zu können, wird beim build eine `env/.env.local.version`-Datei im vue-Ordner erzeugt. Im Falle z.B. eines Quasar-Projektes muss in der `quasar.config.js` noch unter `build.envFiles` der Name der Datei (als String-Array) ergänzt werden, damit der Inhalt der Datei in `process.env` aufgenommen wird.

Der bereits angelegte Task `packageBatixPlugin` wird um den folgenden `from` Block ergänzt:

```
from(npmRunBuild) {
  into("static/import-mitarbeiter-frontend")
}
```

> Gradle Task Caching
> Gradle Tasks können definieren, was deren Input- und Output-Dateien sind. Anhand der Inputs kann Gradle bestimmen, ob ein Task überhaupt laufen muss, oder nicht. Wenn sich die Inputs seit dem letzten Run nicht geändert haben, ist der Task `UP-TO-DATE` und muss nicht noch einmal laufen. Bedingung ist natürlich, dass die Inputs korrekt erfasst wurden.
>
> Im ersten Task (`npmInstall`), der die benötigten NPM Packages installiert, wurde `package.json` als Input definiert. Sobald dort also eine neue Dependency eingetragen wurde,

> sieht Gradle, dass sich die Datei geändert hat, und führt den Task erneut aus.
>
> Dank der Outputs müssen andere Tasks nicht genau wissen, welche Dateien ein Task erzeugt, um diese beispielsweise in ein Archiv zu kopieren. Dieser Umstand wird hier im `packageBatixPlugin` Task genutzt.

Plugin Code

Damit die statischen Dateien auch ausgeliefert werden, ist die `load()` Methode in der Plugin-Hauptklasse um folgenden Code zu erweitern

```
registerRequestHandlerForStaticContent(
  "/mitarbeiter-plugin/import/",
  "import-mitarbeiter-frontend",
  null
)
```

Der erste Parameter (`pathPrefix`) muss dem Public Path der Vue-Anwendung entsprechen (bei Quasar: `quasar.config.js: build: publicPath`). Der zweite Parameter (`staticPath`) muss zur `into` Zeile aus dem `from` Block oben passen. Den dritten Parameter (`fallback`) lassen wir hier leer. Falls die Vue-App aber eine Single Page Application (SPA) ist, dann sollte hier `index.html` stehen.

Aufruf

Wird nun das Plugin über den Gradle Task `build` erstellt, ins System hochgeladen und aktiviert, erscheint die Vue-App unter dem angegebenen Pfad. □□