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!

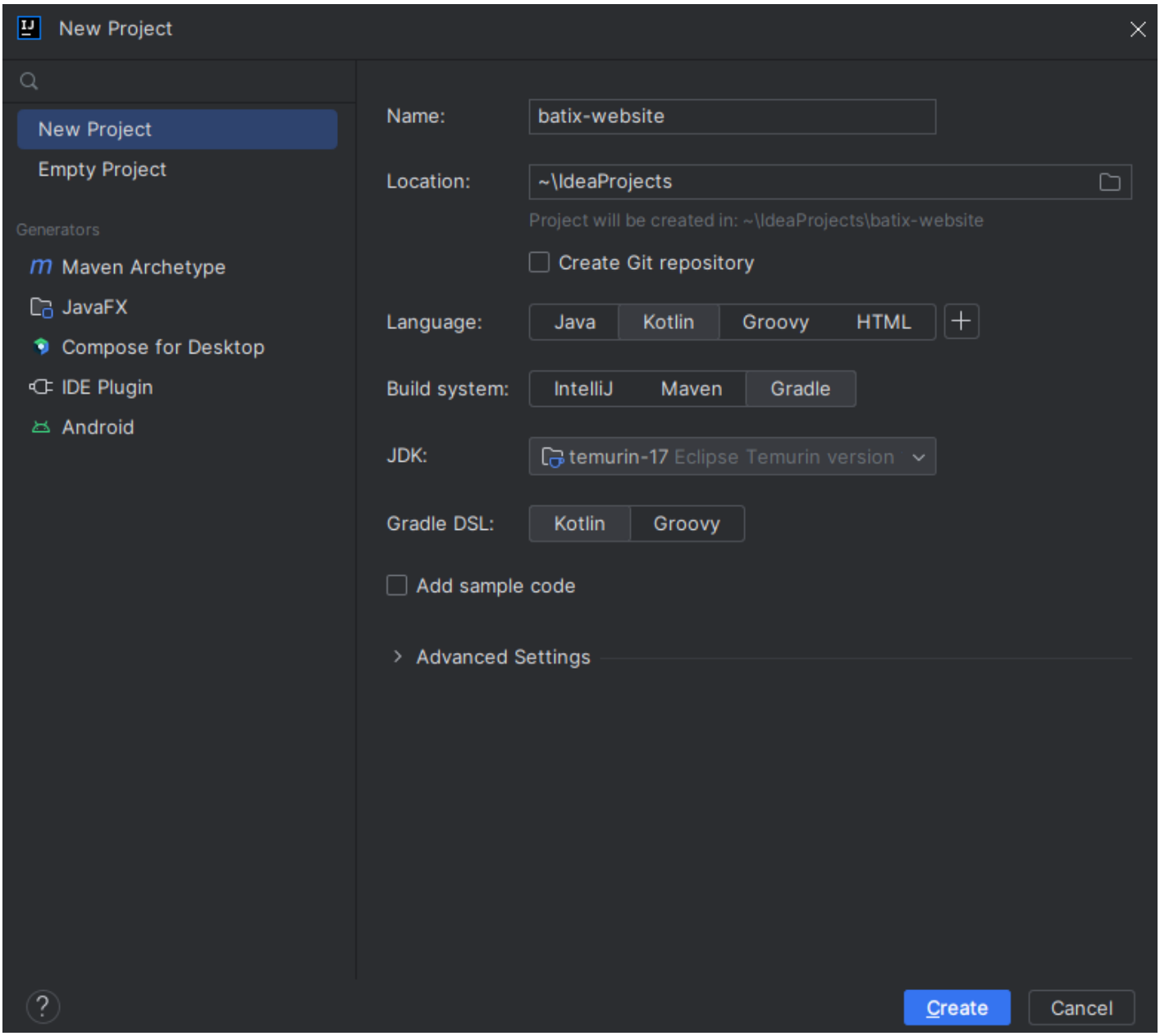
Erfahrene Entwickler können natürlich auch andere IDEs oder sogar die Kommandozeile verwenden. Hier wird nur der Weg für IntelliJ beschrieben.

> **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).

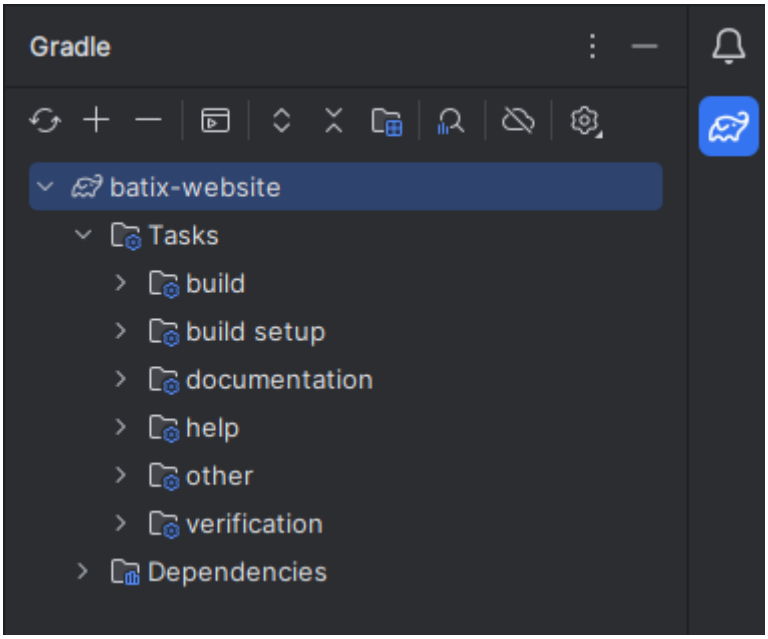
> **Projektstruktur**
> Es wird zunächst ein leeres Projekt erstellt, welches dann um Unterprojekte ergänzt wird. Auch wenn es nur ein Unterprojekt gibt, hat diese Struktur Vorteile, wie z. B. Deduplizierung von Build-Logik.

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:

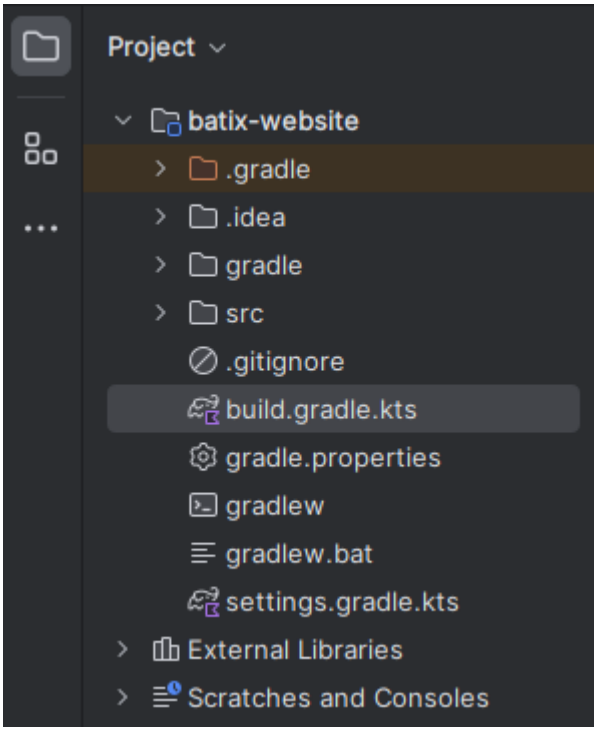
```
plugins {
  // https://plugins.gradle.org/plugin/org.gradle.toolchains.foojay-resolver-convention
  id("org.gradle.toolchains.foojay-resolver-convention") version "0.9.0"
}

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).

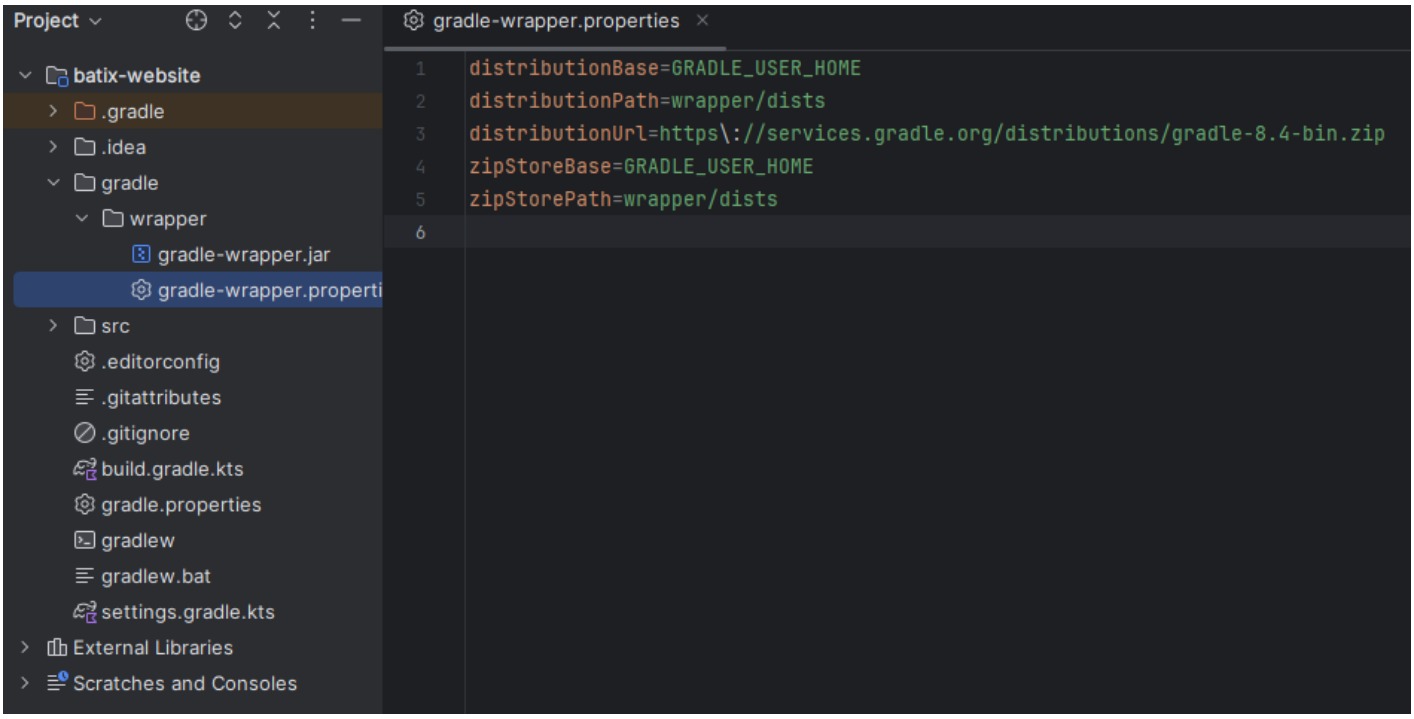
> **Group**
> Es muss auf jeden Fall die `group` Zeile angepasst werden. Diese sollte zur Eindeutigkeit einer Domain des Unternehmens entsprechen, im Falle von Batix könnte die Zeile also lauten:

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

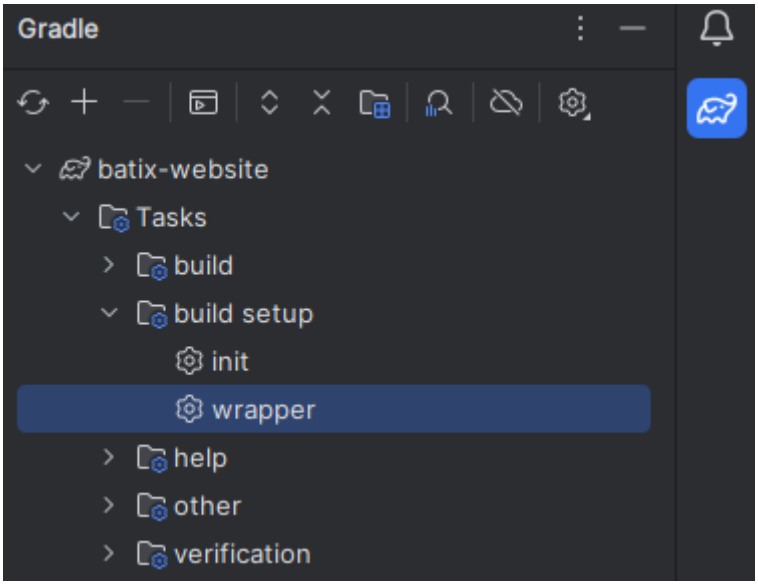
Der große *determine version from git* Block sorgt dafür, dass die Version (welche die Artefakte beim Build-Prozess bekommen) automatisch aus dem Git-Repo erzeugt wird. Dabei werden Git-Tags benutzt, die im Format `v1.2.3` sein müssen. Besonders bei Libraries bietet es sich an, Semantic Versioning zu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



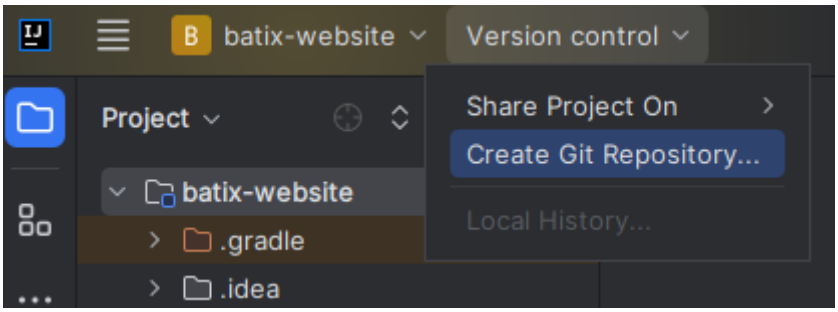
> **Gradle Wrapper**
>
> In der Gradle Dokumentation gibt es eine ausführliche Beschreibung, was der Gradle Wrapper ist und warum man ihn benutzen sollte. Kurz gesagt: so ist sichergestellt, dass jeder Entwickler und auch Sachen wie Continuous Integration (CI) dieselbe Gradle Version verwenden, um Fehler,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

reproduzierbare Builds zu ermöglichen.

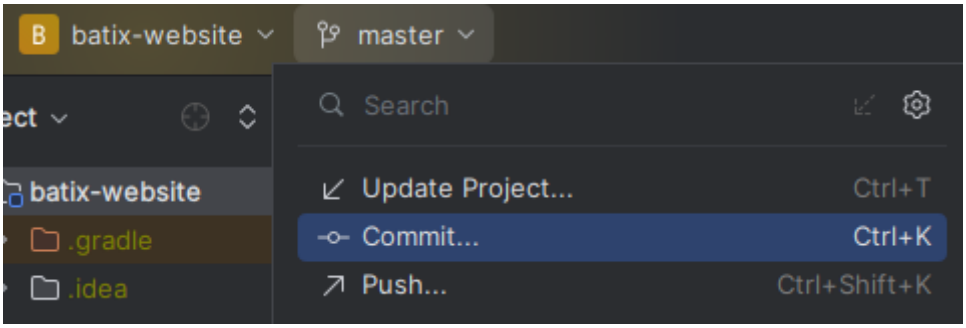
Git Repository

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.

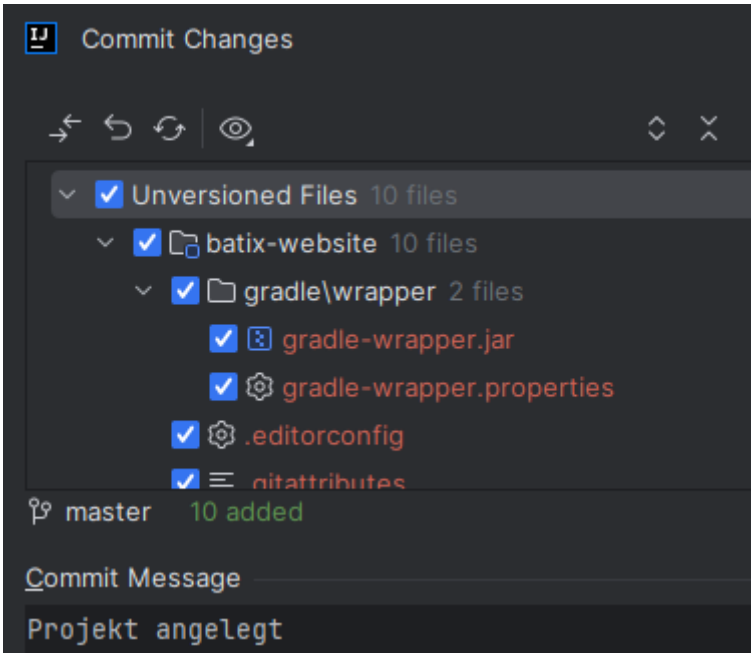
In der Dateiliste links wird das Projektverzeichnis angeklickt, damit dieses ausgewählt ist (hier soll das Git Repo erstellt werden). Dann kann über das Menü *Version control > Create Git Repository...* ein Git Repo angelegt werden. Im sich öffnenden Dialog sollte nochmals geprüft werden, ob auch wirklich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

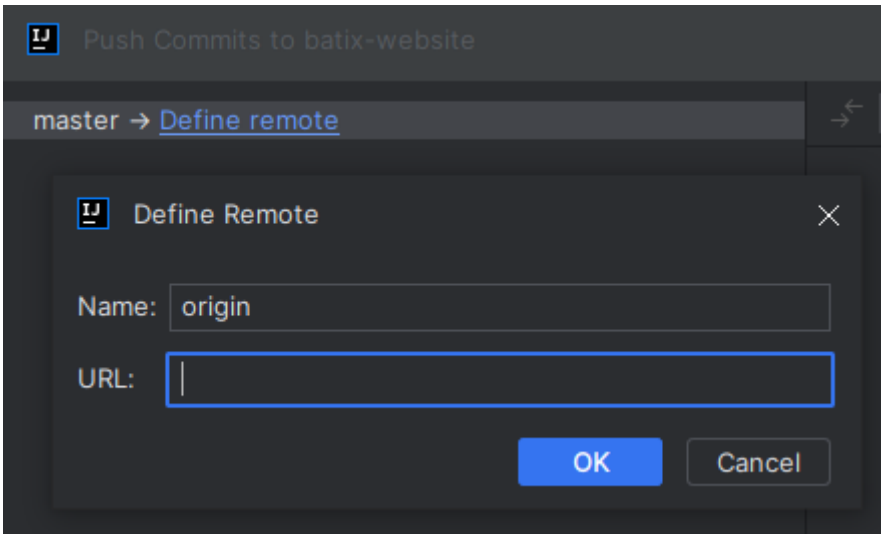


Es erscheint der Dialog *Commit Changes*. Hier müssen alle Dateien angehakt, eine kurze Beschreibung der Änderungen eingetragen und dann auf *Commit* geklickt werden.

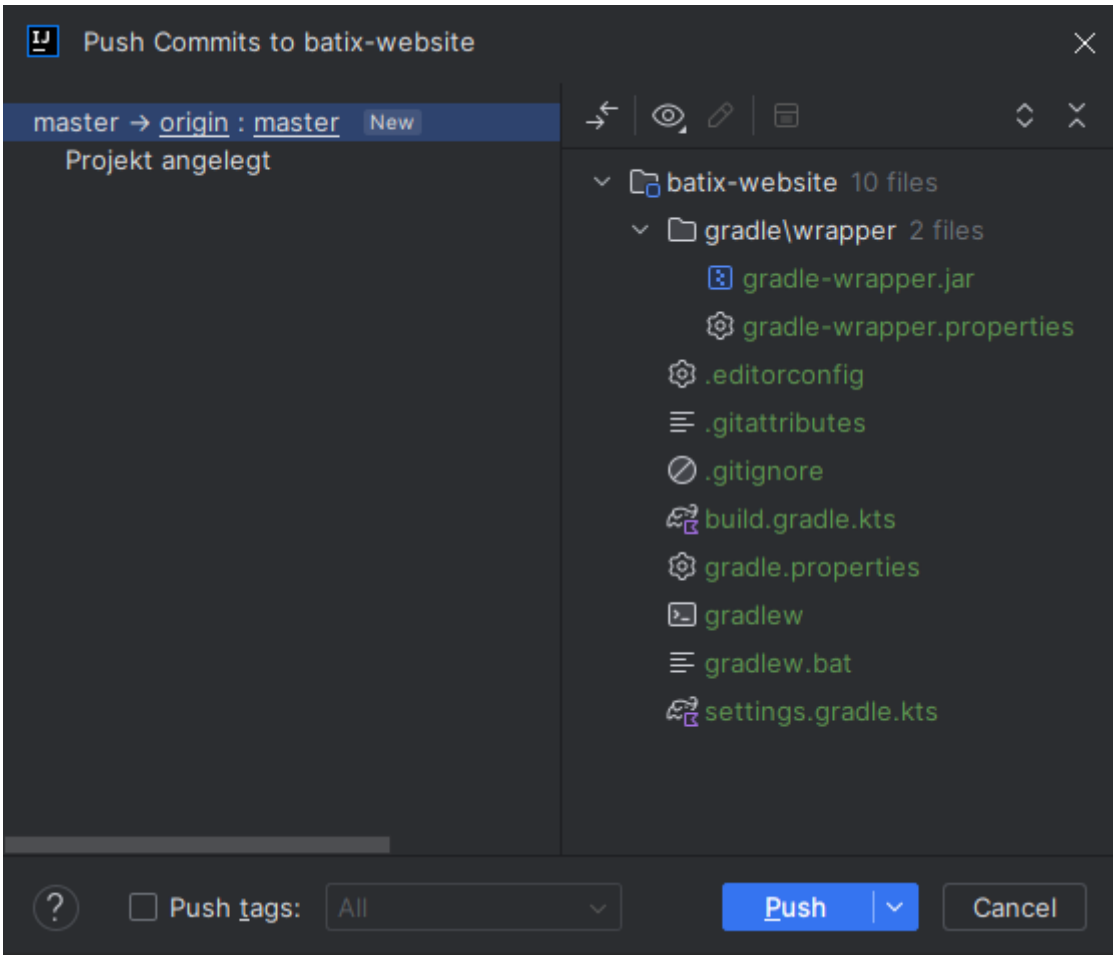


Man sollte nun regelmäßig weitere Commits machen, um später seine Änderungen noch nachverfolgen und zu früheren Code-Ständen zurückspringen zu können. Wann genau ein neuer Commit gemacht wird, ist Geschmackssache, es sollten aber nicht zu viele Änderungen in einen Commit einfließen - lieber kleine, in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



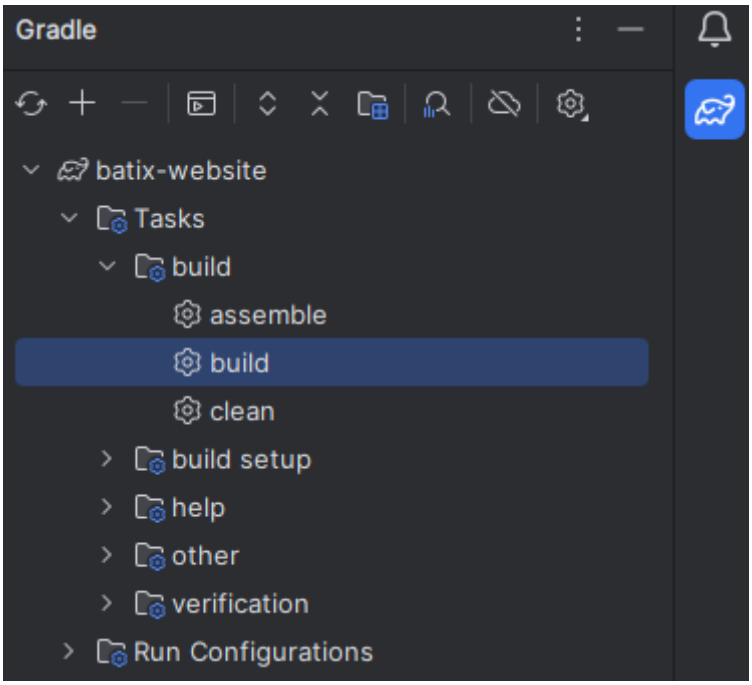
Ist dies erledigt oder wurde das Git Repo nicht lokal erzeugt, sondern initial vom Remote-Server geladen (*Clone*), werden die zu übertragenen Commits angezeigt.



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.