Doku

Für das Projekt haben wir uns tatsächlich an die Doku von JetBrains gehalten.

IntelliJ Platform SDK | IntelliJ Platform Plugin SDK

Die Struktur des Plugins basiert auf dem offiziellen Template und Markdown-Plugins von JetBrains.

1. GitHub - JetBrains/intellij-platform-plugin-template: Template repository for creating plugins for IntelliJ Platform
2. GitHub - JetBrains/projector-markdown-plugin: An IntelliJ plugin which provides a delegating Markdown preview

Converter

Der Converter selbst wendet das Strategy-Pattern an. Wir haben zwei generische Converter, auf die später in mehr Detail eingegangen wird. Diese generischen Converter haben dann jeweils 3 konkrete Converter (auch wenn sich diese Anzahl leicht erweitern lässt).

Wir haben ein Enum EFileType, in dem alle akzeptierten Dateitypen aufgezählt sind. Es ist wichtig zu erwähnen, dass einige Typen nur Import oder nur Export sind. Dies wird nicht im EFileType angegeben.

Der Converter bekommt die originale Datei und eine Instanz von EFileType übergeben. Basierend auf dem Typen der Datei wird ein konkreter IConvertFrom ausgewählt und basierend auf der EFileType-Instanz wird ein konkreter IConvertTo ausgewählt.

Außerdem bekommen wir Optionen aus dem OptionsPanel. Momentan ist dies einfach nur ein Boolean, welcher angibt, ob wir die originale Datei behalten oder löschen möchten. In der Zukunft könnte man dies durch ein Options-Objekt ersetzen oder einfach nur weitere Variablen hinzufügen.

Internal

Tree

Tree ist eine unserer internen Klassen. Ein Tree speichert die root `Node`, welche die äußerste Instanz von Daten darstellt und alle anderen Daten in sich einschießt.

Node

Die `Node` ist der Hauptteil unserer Zwischenschicht und unterteilt in drei Teile.

1. Name Jede `Node` hat einen Namen. (z. B. `<Haus>`)
2. Properties

   Eine `Node` speichert Daten, wenn möglich, als Properties. Dies setzt voraus, dass diese Daten keine eigenen Properties oder Children haben. (z. B. `<Haus quadratmeter=120>`)
3. Children Ein Child ist eine untergeordnete `Node`, die der übergeordneten `Node` zugeordnet wird. (z. B. `<Haus><Raum farbe=blau></Haus>`)

Die Darstellung der Beispiele ist in XML, da die `Nodes` keine Notation haben.

TreeBuilder

Beim TreeBuilder handelt es sich um eine Implementierung des Builder-Patterns.

Der `TreeBuilder` kümmert sich um das Verwalten der einzelnen Instanzen von `Node` während der `Tree` aus den Daten aufgebaut wird.

Da die `Node`s hierarchisch angeordnet sind, baut der TreeBuilder einen `Stack`, auf dem neue `Node`s abgelegt und wenn wieder auf den Parent zugegriffen werden möchte, wird die aktuelle `Node` vom `Stack` entfernt.

Hierfür stellt der `TreeBuilder` Funktionen für das einfachere Erstellen von bereit. (mehr dazu im Code)

Um das Arbeiten mit dem `TreeBuilder` möglichst einfach zu gestalten, wird immer (außer bei der `build()` Funktion) der `TreeBuilder` zurückgegeben, sodass Aufrufe aneinander gereiht werden können.

```
builder.newNode("NodeName")
    .addProperty("PropertyName", PropertyValue)
    .build()
```

XMLTreeBuilder

`XMLTreeBuilder` erbt vom `TreeBuilder` und fügt die im `ConverterFromXML`-Kapitel erwähnte Überprüfung auf Properties durch, welche ausgeführt wird, wenn die `build()` Funktion aufgerufen wird.

ConverterFrom

IConvertFrom

Das IConvertFrom Interface gibt zwei Funktionen vor:

1. readFile Bekommt den Pfad zur Datei als `String` gegeben und gibt sie als `File` zurück.
2. convert Bekommt die Datei als `File` übergeben und gibt den Inhalt als `Tree` zurück.

Alle Converter, die von einer Datei zu unserer Zwischenschicht konvertieren, müssen dieses Interface implementieren.

ConverterFromUtils

`ConvertFromUtils` stellt dem konkreten ConverterFrom die Helferfunktion `toTypeOrDefault` zur Verfügung. `toTypeOrDefault` hilft, die als `String` ausgelesenen Daten wieder in die richtigen Datentypen umzuwandeln.

ConverterFromJSON

`ConverterFromJSON` war recht einfach umzusetzen, da Kotlin JSON unterstützt. Erst wandeln wir den Inhalt der Datei zu einem Json-Objekt um. Dann gehen wir rekursiv durch alle Subelemente und übergeben diese in den `TreeBuilder`, welcher um Ende einen `Tree` zurückgibt.

ConverterFromXML

`ConverterFromXML` war etwas schwieriger und benötigte einen eigenen `XMLTreeBuilder`, da wir zwei ganze Durchläufe brauchen, um die Daten umzuwandeln. Im ersten Durchlauf werden die XML-Tags und deren Daten in `Nodes` umgewandelt. Im zweiten Durchlauf wird dann überprüft, ob `Nodes` auch in der übergeordneten `Node` als Property dargestellt werden können und wenn möglich geändert werden.

ConverterFromCSV

`ConverterFromCSV` war wieder einfacher, da es sich CSV sehr einfach auseinanderbauen lässt, indem man Zeile für Zeile durchgeht. Die erste Zeile enthält die Spaltennamen.

Neuen ConverterFrom hinzufügen

Um einen neuen ConverterFrom zu erstellen, müssen folgende Schritte beachtet werden.

1. `IConvertFrom` implementieren
2. Falls nötig, Typen zu `EFileType` hinzuzufügen
3. In `Converter` zur Enum-Auflösung hinzufügen
4. In `ConvertToActionGroup` den Type zu den `validTypes` hinzufügen

ConverterTo

IConvertTo

Das IConvertTo Interface enthält einfach nur die Funktion convertAndWrite, welche unseren Tree und den Zielpfad (als String) übergeben bekommt. Diese Funktion wird vom Converter aufgerufen, um die Daten aus der Zwischenebene (also dem Tree) zu konvertieren und als Datei zu speichern.

ConverterToJSON

Der konkrete Converter zu JSON ist der simpelste, da Kotlin bereits standardmäßig JSON-Support hat. Alles, was dieser Converter macht, ist rekursiv durch die Nodes vom Tree zu gehen und dabei die Nodes als JSON-Objekt zu speichern. Die Kinder der jeweiligen Node werden alle in einem JSON-Array gespeichert.

Für weitere Informationen kann man in den Code und die Kommentare schauen.

ConverterToXML

Der konkrete Converter zu XML ist etwas aufwendiger, da Kotlin keine eingebauten Funktionen zum Schreiben oder Bearbeiten von XML hat. Das Prinzip ist aber immer noch das gleiche, wie bei JSON; wir gehen rekursiv durch die Nodes im Tree und konvertieren sie zu XML. Der Haupt-Unterschied ist, dass wir die Struktur von XML selbst schreiben müssen, anstatt einfach eine Funktion von Kotlin aufzurufen.

Die Kinder werden einfach als Interne Nodes geschrieben. Da Whitespace in XML nicht wichtig für die Struktur sind, ist die Output-Datei komplett unformatiert. Allerdings kann man die Datei manuell von IntelliJ formatieren lassen, indem man sie öffnet und STRG + ALT + L drückt.

Außerdem muss man bei XML noch darauf achten, dass bestimmte Charaktere escaped werden, wofür ich eine Helferfunktion geschrieben habe.

Für weitere Informationen kann man in den Code und die Kommentare schauen.

ConverterToYAML

Der konkrete Converter zu YAML ist dem XML-Converter sehr ähnlich. Die einzigen Unterschiede sind, dass Whitespaces wichtig sind, für die YAML-Syntax und mehrere Felder mit dem gleichen Namen illegal sind.

Die Whitespaces, die vor jedem Eintrag stehen müssen, werden einfach rekursiv als Präfix übergeben.

Für die Duplikate habe ich einfach 3 Helferfunktionen geschrieben:

Eine, welche herausfindet, ob es Duplikate gibt, eine welche die Duplikate umbenennt, und eine welche so lange die zweite Funktion ausführt, bis es keine Duplikate mehr gibt.

Ansonsten muss man nur beachten, dass YAML andere spezielle Charaktere hat, welche escaped werden müssen.

Für weitere Informationen kann man in den Code und die Kommentare schauen.

Neuen Konkreten IConvertTo hinzufügen

Um einen neuen konkreten IConvertTo hinzuzufügen, muss man folgende Schritte befolgen:

1. Den neuen Type, zu dem man konvertieren möchte, zu EFileType hinzufügen.
2. Den Konkreten IConvertTo erstellen.
3. Die Cases im Converter um den neuen Typ erweitern.
4. Neue ConvertTo[Type]Action erstellen und Registrieren.

Man kann diese Schritte in beliebiger Reihenfolge durchführen, es ist allerdings am einfachsten, wenn man es wie oben beschrieben macht.

1. Um einen neuen Type in EFileType hinzuzufügen, muss man einfach nur die Dateiendung des Typens als neuen Eintrag in EFileType hinzufügen.
2. Um einen neuen konkreten IConvertTo zu erstellen, muss man einfach eine neue Klasse erstellen, welche IConvertTo implementiert. Wir empfehlen stark, diese Klasse ConverterTo[Type] zu benennen. Das tatsächliche Konvertieren ist für jeden Dateityp anders, aber man kann sich an den bestehenden Convertern orientieren.
3. Im zweiten when (Kotlin-Syntax für switch) muss ein weiterer case hinzugefügt werden, für den neuen Type. Dabei kann sich einfach an den bestehenden cases orientiert werden.
4. Hier kann man einfach eine der Bestehenden ConvertTo-Actions kopieren, umbenennen und alle Abfragen nach einem Typ auf den neuen Typ ändern.

Um die Action zu registrieren, kann man einfach eine der Actions in der plugin.xml kopieren und alle Referenzen auf die neue Action ändern.

Actions

Wir haben die ConvertToActionGroup, welche von der DefaultActionGroup erbt. Wir haben sie um eine Liste von validen Datentypen erweitert. Diese Liste repräsentiert alle Datentypen, welche einen konkreten IConvertFrom haben. Wenn man auf eine Datei, deren Typ in der Liste ist, rechts klickt, wird die Actiongroup angezeigt.

Des Weiteren haben wir eine Action für jeden konkreten IConvertTo, welche alle von AnAction erben. Diese sind der ConvertToActionGroup untergeordnet und werden deswegen nur angezeigt, wenn auch die ActionGroup angezeigt wird. Außerdem prüft die Action, ob die Datei, auf die gerechtsklickt wurde, denselben Typ hat wie die Action, und wird nicht angezeigt, wenn dies der Fall ist. (z. B. wird die ConvertToXMLAction nicht angezeigt bei der Datei test.xml).

Wenn die Action ausgeführt wird (also wenn man sie anklickt), dann öffnet sich ein OptionsPanel mit dem passenden EFileType und der PsiFile von der Datei, auf die geklickt wurde.

Für mehr Informationen zu Actions und ActionGroups kann man in der offiziellen JetBrains-Dokumentation dazu nachschlagen:

Actions | IntelliJ Platform Plugin SDK

UI

OptionsPanel

Diese Ansicht ist recht simpel mit Java-Swing erstellt und beinhaltet alle Optionen, die angezeigt werden, bevor eine Datei umgewandelt wird. Auch die Funktionen der Buttons sind hier hinterlegt.

OptionsPopupHandler

Der `OptionsPopupHandler` kümmer sich um das Anzeigen von Popups nach der offiziellen Dokumentation von JetBrains.

Popups | IntelliJ Platform Plugin SDK

Klassendiagramm