Programmieren für ELO

ELO Flows Komponenten

Inhaltsverzeichnis

Grundlagen

ELO Flows Komponentenentwicklung

Im folgenden Abschnitt zu den *Grundlagen - ELO Komponentenentwicklung* lernen Sie die ELO Flows Komponentenentwicklung kennen.

Sie erfahren hier mehr über:

- ELO Flows
- den Mehrwert für die Entwicklung
- die Voraussetzungen, die zur Umsetzung von ELO FLows Komponenten nötig sind
- ELO Flows Komponenten im Allgemeinen

Einleitung

Mit ELO Flows können auch Nicht-Techniker Automatisierungs- und Integrationsaufgaben in ELO vornehmen. Als Grundlage hierzu dient die ELO Flows Komponentenentwicklung. Sie ermöglicht es den ELO Business Partnern modulare Anpassungen in ELO zu gestalten und diese als ELO Flows zur Verfügung zu stellen. Vertiefende Informationen zu ELO Flows finden Sie in der Dokumentation ELO Flows.

Mehrwert

Häufig werden bei ELO Businesspartnern kundenindividuelle Lösungen für ELO angefragt. Mit ELO Flows Komponentenentwicklung können Sie Kosten und Zeit der programmtechnischen Anpassungen stark reduzieren:

Die auf Java basierende Programmierung individueller ELO Flows Komponenten ermöglicht eine Erweiterung des Umfangs von ELO Standard Funktionen. Eigene Flows Komponenten führen zu wesentlich geringeren QS- und Support-Kosten gegenüber herkömmlichen Schnittstellen.

Voraussetzungen

Für die ELO Flows Komponentenentwicklung gelten folgende Voraussetzungen:

- mindestens ELO 21
- mindestens ELO Flows 21
- Visual Studio Code (kurz VS Code)
- das VS-Code-Plug-in: Framework für ELO Flows Komponentenentwicklung
- Java Runtime Environment (ab Java 16)
- mindestens Gradle 7
- aktivierter SSH-Zugang in Apache Karaf

Nähere Informationen finden Sie in den nachfolgenden Abschnitten.

Quelltext-Editor: Visual Studio Code (VS Code)

Verwenden Sie Visual Studio Code als Quelltext-Editor.

VS Code läuft unter den Betriebssystemen Windows, Linux und macOS. Der Editor ist keine echte integrierte Entwicklungsumgebung (IDE), dennoch beinhaltet er alle gängigen Programmierwerkzeuge (Versionsverwaltung, Debugging, Code-Vervollständigung oder Syntax-Hervorhebung).

VS Code arbeitet nicht auf Projektbasis, d. h. Implementierungsarbeiten werden in Arbeitsumgebungen und Ordnern organisiert. Durch Plug-ins kann VS Code nahezu für jede Programmiersprache eingesetzt, individuell angepasst und erweitert werden.

VS Code Plug-in: Framework für ELO Flows Komponentenentwicklung

Das ELO Dev Plug-in für VS Code sorgt für die optimale Arbeitsumgebung bei der ELO Flows Komponentenentwicklung. Es liefert ein vorkonfiguriertes Framework zur ELO Flows Komponentenentwicklung mit. Dieses vereinfacht die Entwicklung und schafft einheitliche Richtlinien für die Implementierung. Sie können nach der Installation ein Projekt starten und mit der Programmierarbeit beginnen. Es sind keine weiteren Konfigurationen bzw. Erweiterungen nötig. Das Plug-in beinhaltet zudem eine Beispielkomponente, die den Einstieg in die Umsetzung erleichtert.

Sie finden das ELO Plug-in in der ZIP-Datei ELO Flows Komponentenentwicklung.

Java Runtime Environment (ab Java 15)

Die Implementierung von ELO Flows Komponenten erfolgt mit der Programmiersprache Java. Hierfür benötigen Sie eine Java Laufzeitumgebung (JRE) ab Java 15. Für umfangreichere Implementierung ist ein Java Development Kit (JDK) zu empfehlen. Dieses stellt weitere Programmierwerkzeuge zur Verfügung. Das Java-Runtime-Environment (JRE) ist bereits im JDK enthalten.

Gradle 6

Das Build-Management-Projekt-Tool Gradle (mindestens Version 6) wird bei ELO Flows Komponentenentwicklung zum Einbinden von externen Java Libraries (JAR-Dateien) und Frameworks wie JUnit eingesetzt.

Aktivierter SSH-Zugang in Apache Karaf

Die Entwicklung mit dem VS-Code-Plug-in bedingt die Erreichbarkeit von Apache Karaf per SSH. Seit dem finalen Release von ELO 21 installiert das ELO Server Setup Apache Karaf mit deaktiviertem SSH Zugang. Aktivieren Sie zunächst den SSH Zugang in Apache Karaf.

ELO Flows Komponente

ELO Flows basiert auf dem Prinzip der Komponentenprogrammierung. ELO Flows Komponenten sind in sich abgeschlossene Software-Elemente, die die Standardfunktionen von ELO erweitern. Die Datenkapselung der Flows Komponenten ermöglicht eine hohe Portabilität. Sie können unabhängig von Kundensystemen programmiert und bei Bedarf über standardisierte Schnittstellen in das jeweilige Kundensystem eingebunden werden. Eine ELO Flows Komponente wird als JAR-Datei ausgegeben und zur weiteren Verwendung im ELO Flows Worker bereitgestellt. ELO Flows Worker basiert auf Apache Karaf und stellt den Container für ELO Flows Komponenten bereit.

Eine ELO Flows Komponente besteht aus grafischen Elementen. Diese werden über Java Annotationen realisiert. ELO Flows Komponenten können Trigger, Dienste oder eine Kombination aus beidem ausliefern. Außerdem können in ELO Flows Informationsfelder eingebunden werden, deren Inhalte als MD-Dateien hinterlegt sind.

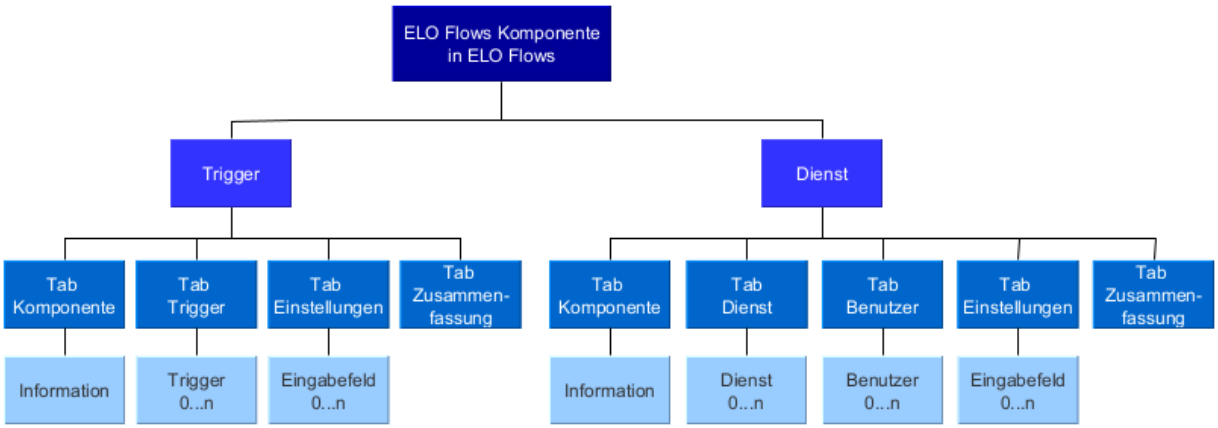

Abb. Grafischer Aufbau einer ELO Flows Komponente

ELO Flows Komponenten sind nach einem festen Schema aufgebaut, das durch ELO vorgegebene wird. Die Komponentenentwicklung von ELO Flows Komponenten findet im *Framework zur ELO Flows Komponentenentwicklung* in VS Code statt. Das Framework erhalten Sie über das ELO Dev Plug-in.

Mit ELO Flows werden Standardkomponenten mitgeliefert, die Sie im Verlauf der Komponentenentwicklung wiederverwenden oder weiterentwicklen können.

Die Funktion der Standardkomponenten wird in der Hauptdokumentation zu ELO Flows beschrieben.

Die jeweiligen Definitionen der Standardkomponenten und Annotationen finden Sie innerhalb der in ELO implementierten API-Doku.

Einrichtung

Umgebung einrichten

Im folgenden Abschnitt erfahren Sie, wie Sie *ELO Flows Komponentenentwicklung* auf ihrem System vorbereiten und die Umgebung optimal einrichten.

Sie erfahren hier mehr darüber:

- Wie Sie SSH in Apache Karaf aktivieren
- Wie Sie das ELO Dev Plug-in in VS Code installieren
- Wie Sie ein neues Projekt zur ELO Flows Komponentenentwicklung anlegen

Zusätzlich erhalten Sie kleinere Tipps und nützliche Zusatzinfos bei der Komponentenentwicklung.

SSH in Apache Karaf aktivieren

1. Erweitern Sie die Datei *users.properties* (`%karaf%/etc/users.properties`) um folgende Einträge:

   ```
   karaf = karaf,_g_:admingroup
   _g_\:admingroup = group,admin,manager,viewer,systembundles,ssh
   ```

2. Ergänzen Sie die Datei *org.apache.karaf.features.cfg* (`%karaf%/etc/org.apache.karaf.features.cfg`) unter *featureBoot* um das SSH Feature `,ssh/4.3.0`:

   ```
   featuresBoot = ...,ssh/4.3.0
   ```

3. Starten Sie abschließend Apache Karaf einmalig mit dem Flag `clean`.

   Alternativ:

   1. Löschen Sie bei gestoppten Apache Karaf das Verzeichnis `%karaf%/data` komplett.
   2. Starten Sie Apache Karaf neu.

ELO DEV Plug-in installieren

Vorbereitung: Umgebungs- und Path-Variable setzen

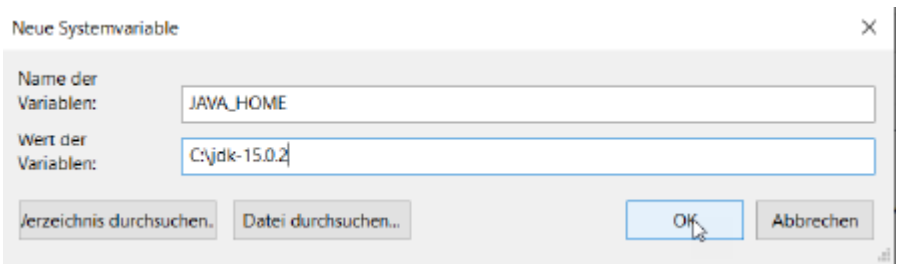

Abb.: Neue Systemvariable JAVA_HOME

1. Setzen Sie für Java die Umgebungsvariable `JAVA_HOME`. Tragen Sie den jeweils passenden Pfad ein.

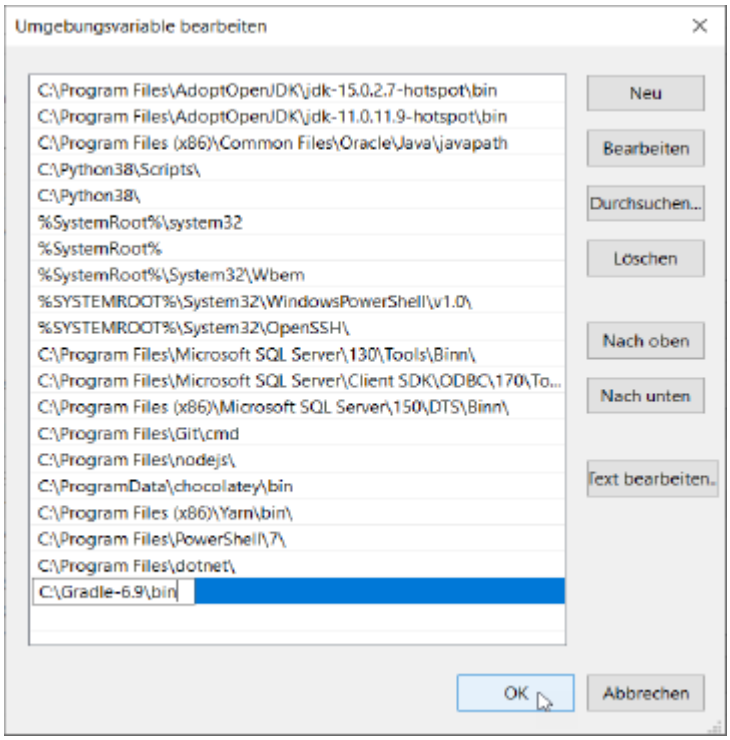

Abb.: Path-Variable ergänzen

2. Ergänzen Sie für Gradle die Path-Variable. Tragen Sie den jeweils passenden Pfad ein.

Download ELO Dev Plug-in für VS Code

1. Laden Sie das ELO Plug-in *Framework zur Komponentenentwicklung* herunter. Sie finden das ELO Plug-in in der ZIP-Datei ELO Flows Komponentenentwicklung.

ELO Dev Plug-in in VS Code installieren

1. Öffnen Sie VS Code

2. Wählen Sie den Tab *Erweiterungen* in der Menüleiste aus.

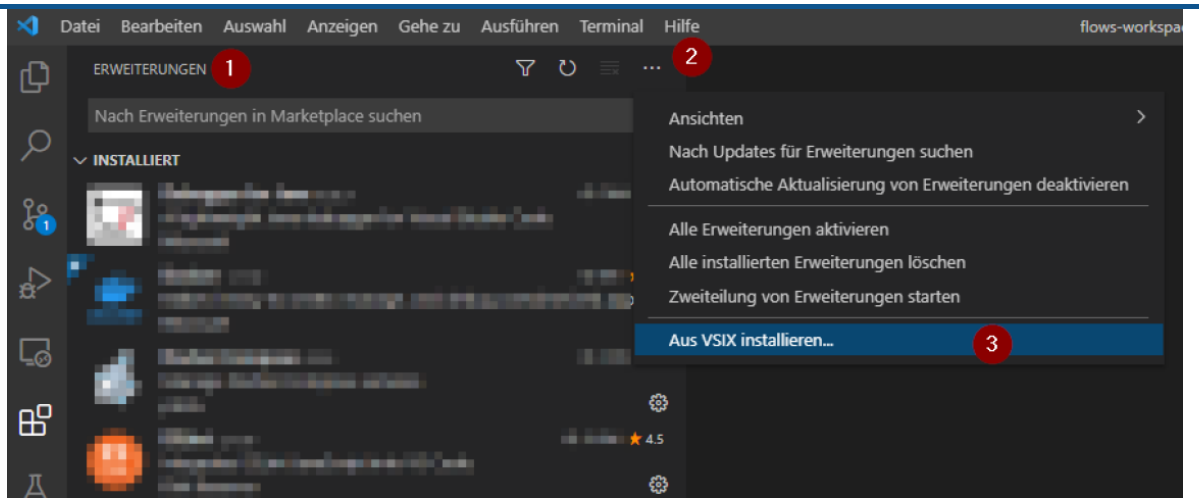

Abb.: Plug-in Installation Framework zur Komponentenentwicklung

3. Wählen Sie im Drop-Down-Menü *Aus VSIX installieren ...* aus.

4. Wählen Sie ELO Dev Plug-in in VSIX-Format aus.

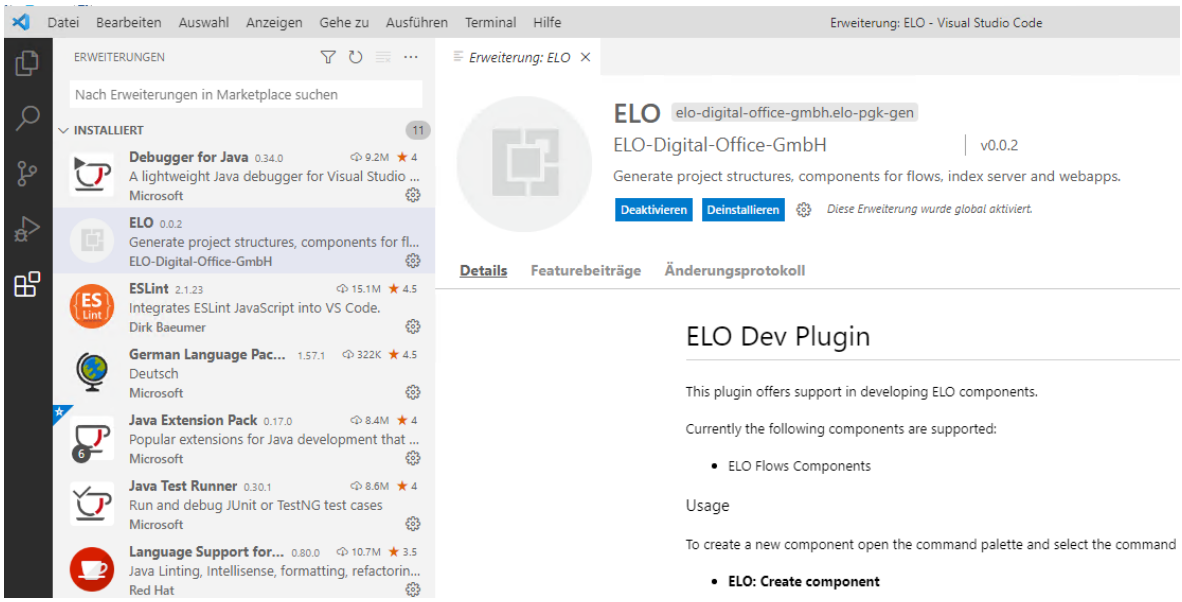

Abb.: ELO Dev Plug-in in VS Code

Nachdem die Installation abgeschlossen ist, erhalten Sie in VS Code weitere Informationen zum ELO Dev Plug-in.

Neues Projekt anlegen

ELO Dev Plug-in stößt eine automatisierte Projektstruktur-Erzeugung in VS Code an und richtet dabei das *Framework für ELO Flows Komponentenentwicklung* ein.

Das *Framework für ELO Flows Komponentenentwicklung* bietet ein vollständiges, direkt einsatzfähiges Grundgerüst, um Komponenten zu gestalten. Dieses erleichtert durch mitgelieferte Beispielkomponenten den Erstellungsprozess.

Im folgenden Abschnitt erfahren Sie:

-

Wie Sie eine neue Projektstruktur einer ELO Flows Komponente in VS Code anlegen.

- Wie Sie das Projekt als Java-Projekt initialisieren.

Projektstruktur einer ELO Flows Komponente anlegen

1. Starten Sie VS Code.

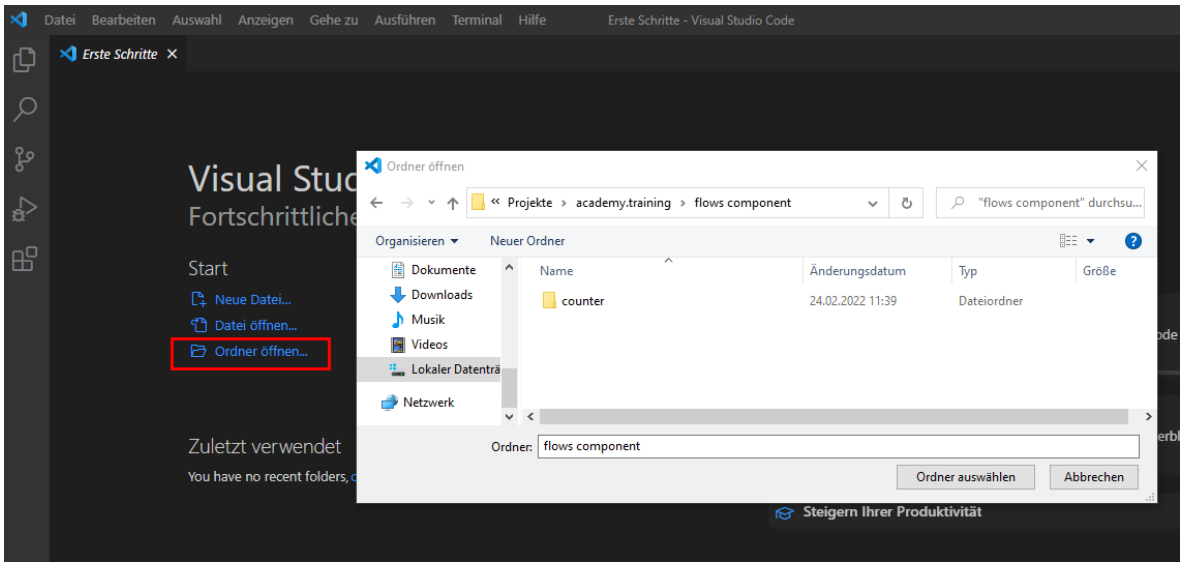

Abb.: Projektordner im Dateisystem öffnen

2. Wählen Sie einen Ordner aus, in welchem die neue Komponente angelegt werden soll:

   Datei > Ordner öffnen...

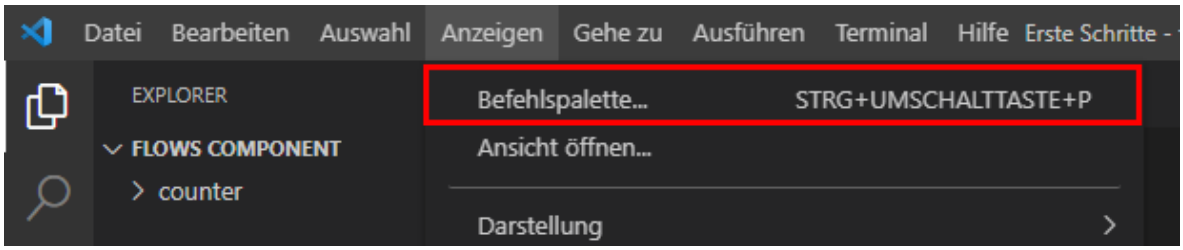

Abb.: Befehlspalette auswählen

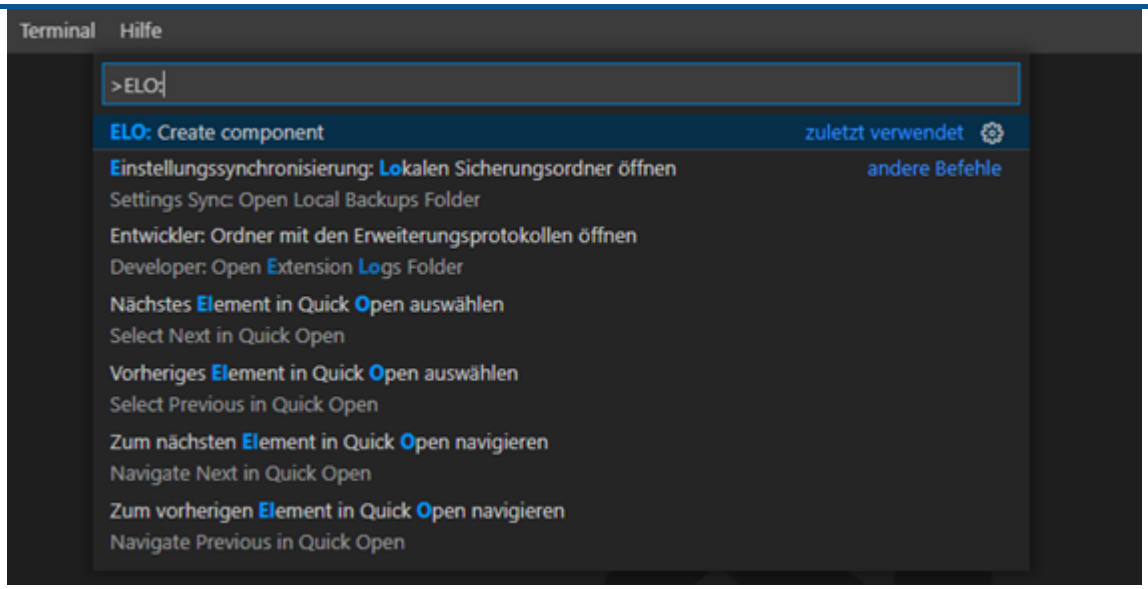

Abb.: 'ELO: Create component' ausführen

3. Führen Sie über die Befehlspalette den Befehl *ELO: Create component* aus:

   Anzeigen > Befehlspalette...

   Alternativ 1: STRG + SHIFT + P

   Alternativ 2: F1-Taste

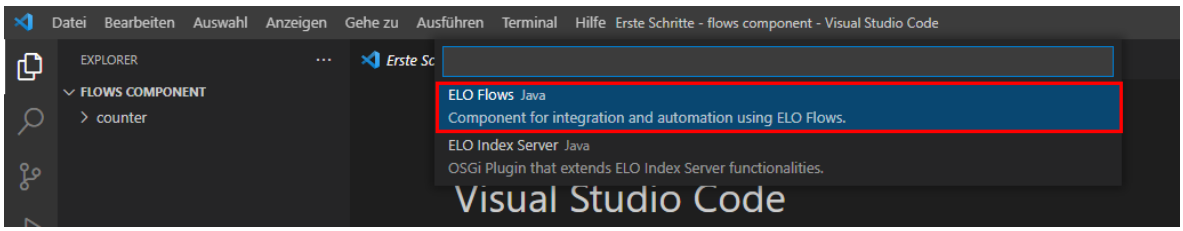

Abb.: 'ELO Flows Java' auswählen

4. Wählen Sie ELO Flows aus, um eine neue Komponente zu erzeugen.

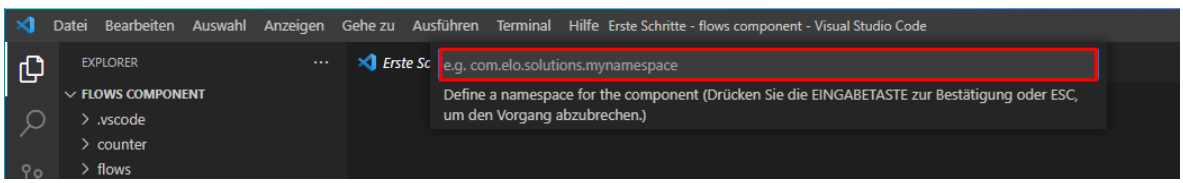

Abb.: Paketname eingeben

5. Geben Sie einen Namespace an.

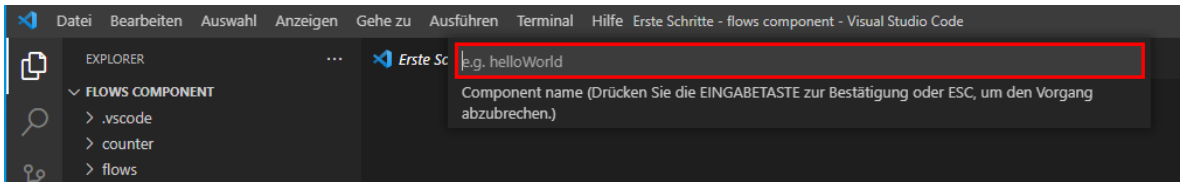

Abb.: Name der Komponente eintragen

6.

Geben Sie einen Namen für die Komponente an.

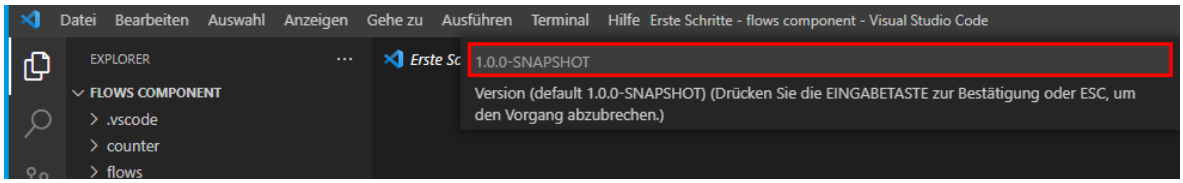

Abb.: Versionsnummer festlegen

7. Geben Sie eine Version an.

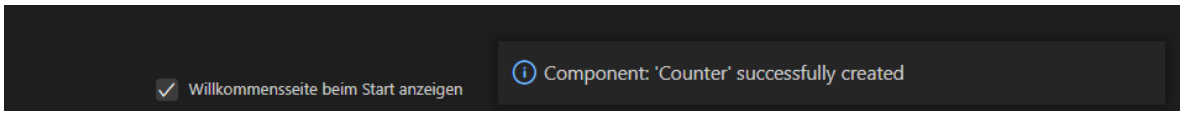

Abb.: Pop-up-Fenster bei erfolgreich angelegter Komponente

Rechts unten im VS Code zeigt ein Pop-up-Fenster an, dass die Komponente erfolgreich angelegt wurde.

Projekt als Java-Projekt initialisieren

Initialisieren Sie das Projekt als Java-Projekt, damit die Erweiterungen für die Java Entwicklung nachgeladen werden:

1. Navigieren Sie in der Projektstruktur bis zu einer beliebigen Java-Klasse.

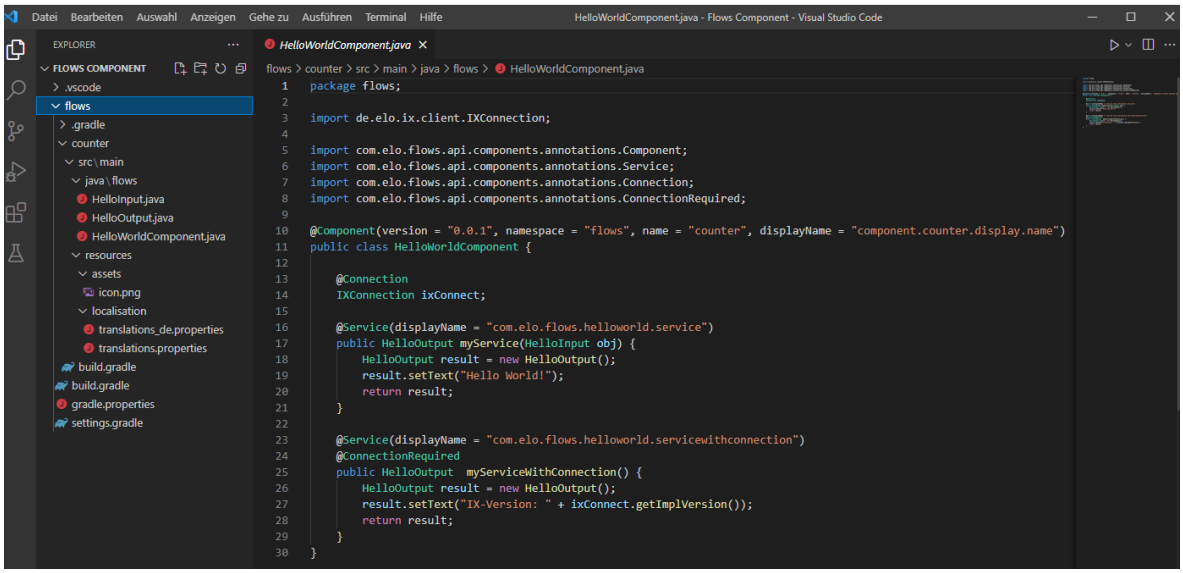

Abb.: Projektstruktur der Flows-Komponente

2. Wählen Sie die Java-Klasse via Mausklick aus.

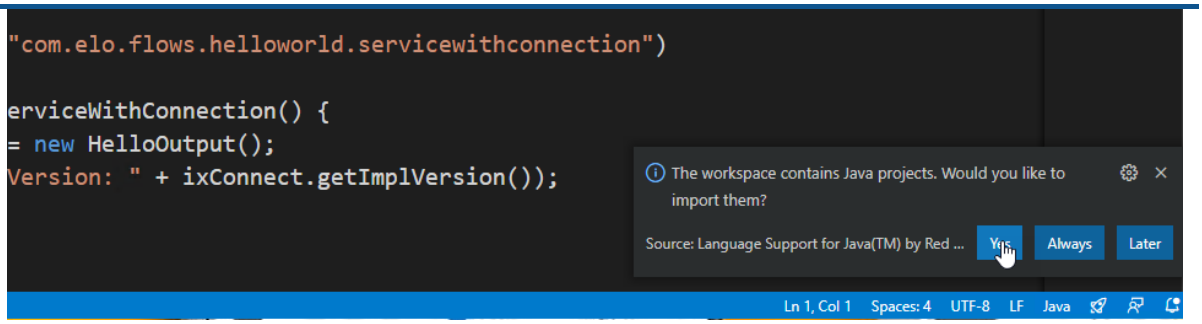

Abb.: Projekt als Java-Projekt initialisieren

3. Bestätigen Sie die jeweiligen Meldungen mit *Yes*.

> **Information**
>
> Führen Sie während des Ladens keine weiteren Tätigkeiten im VS Code aus.

Der Ladestatus kann durch Klicken auf den sich drehenden Kreis (rechts unten in VS Code) angezeigt werden. Das Laden ist beendet, wenn ein "Daumen nach Oben" Icon erscheint.

Tipps und Tricks

In diesem Abschnitt erhalten Sie zusätzliche Informationen.

Sie erfahren hier mehr über:

- Automatisch generierte Ordner-Struktur in VS Code
- Tastaturbefehle im Kontext mit ELO Dev Plug-in

Struktur des Frameworks für ELO Flows Komponentenentwicklung

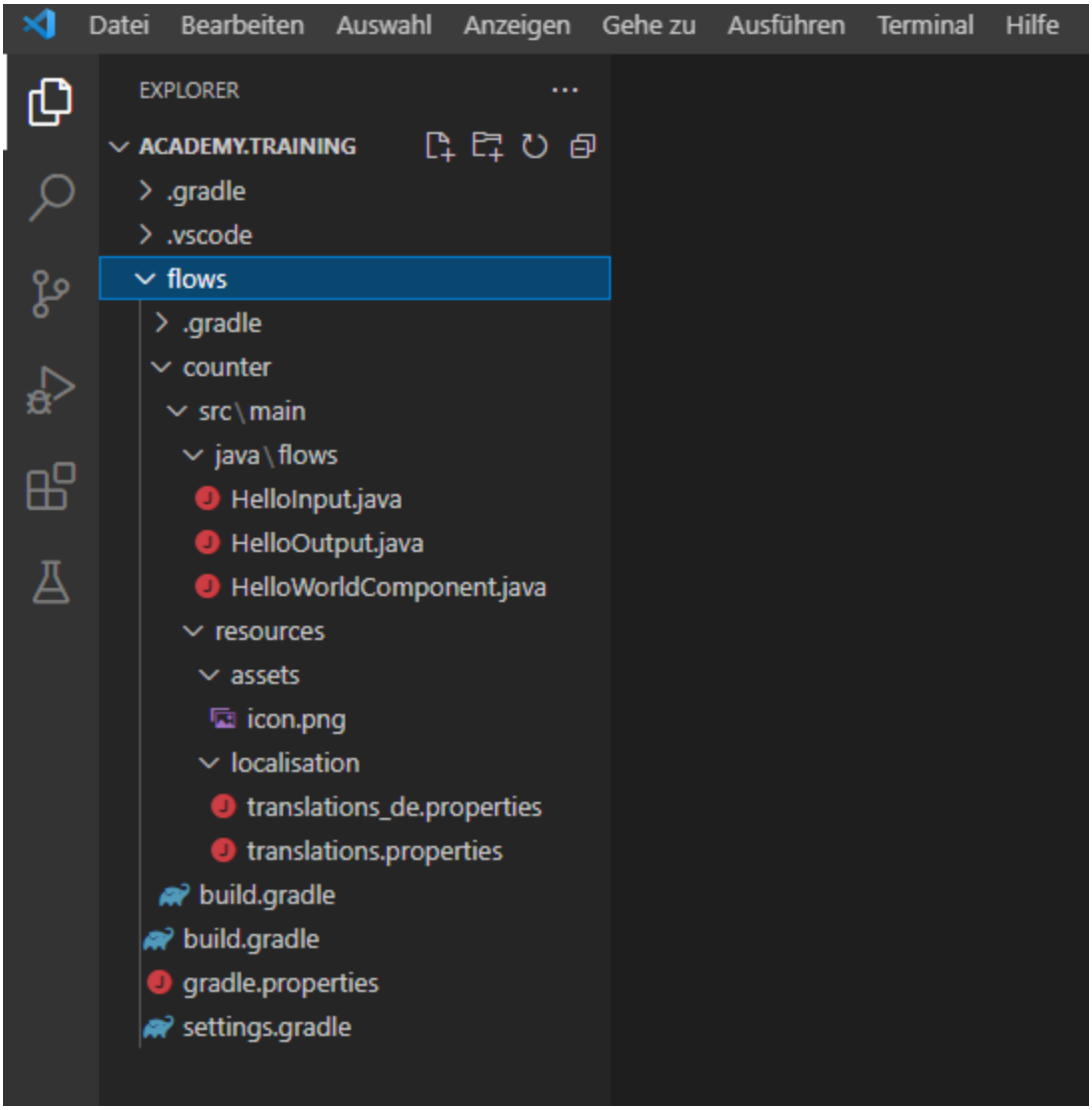

Abb.: Exemplarische Projektstruktur

Framework für ELO Flows Komponentenentwicklung besteht aus mehreren unterschiedlichen Bereichen, die für Java-Projekte typisch sind:

- src\main
- resources
- bin
- libs
- test

> **Information**
>
> Im Laufe des Entwicklungsprozesses können weiter Strukturbereiche nach und nach ergänzt werden. Dazu zählen zusätzliche Klassen oder die erstellte Bibliothek Ihres Projekts.

Quellcode-Ordner: src\main

Der Quellcode (Source) des Projekts befinden sich im Ordner *src\main*.

> **Information**
>
> Die Projektordner-Struktur entspricht keiner 1:1 Paketierung im Quellcode. Diese hängt von den Projekteinstellungen ab und kann später individuell konfiguriert werden.

Resourcen-Ordner: resources

In dem Projektordner *resources* befinden sich z. B. die Lokalisierungsdateien.

Binär-Dateien-Ordner: bin

Im Projektordner *bin* werden die vorübersetzten Java-Klassen sowie Ressourcen abgelegt. Hieraus werden später über den Build-Prozess die JAR-Dateien gebaut.

Bibliotheks-Ordner: libs

Nach dem ersten Build-Prozess erscheint ein neuer Ordner mit der JAR-Bibliothek.

Optionaler Test-Ordner: test

Es kann sinnvoll sein, wenn Sie zunächst einen Test-Ordner mit(Teststruktur) erstellen.

Tastaturbefehle für VS Code des ELO Dev Plug-in

Fuktion	Tastaturbefehle
'Befehlspalette' öffnen	STRG + UMSCHALT + P oder F1
Build-Aufgabe auswählen	STRG + UMSCHALT + B
'Hot Deployment to ELO Flows Worker' beenden	STRG + D ENTER
Debugger starten bzw. Debugger beenden	F5 bzw. STRG + F5

Komponentenentwicklung

Entwicklung einer ELO Flows Komponente

Dieser Abschnitt beschreibt den Entwicklungsprozess einer ELO Flows Komponente mit Fokus auf folgende Themen:

- Beispielkomponente *Dokumentenzähler*: Am Beispiel einer neuen Komponentenimplementierung durchlaufen Sie die wichtigsten Schritte der Komponentenentwicklung.
- Deployment: Das Framework zur ELO Flows Komponentenentwicklung ermöglicht zwei Varianten des Deployments: Einmaliges Ausrollen und kontinuierliches Ausrollen.
- Debugging: Sie erfahren mehr über den Debugging-Prozess innerhalb VS Code.
- Konfigurationsoptionen: Sie erhalten eine Liste der Konfigurationsoptionen, die ELO für den Entwicklungsprozess zur Verfügung stellt.

Besonderheiten von ELO Flows

ELO FlowFile-Handling

ELO weicht beim ELO FlowFile-Handling von der Java-Konvention ab.

> **Beachten Sie**
>
> Entgegen der häufig genutzten Java-Konvention beim Umgang mit File-Handling, darf bei einem ELO FlowFile der *InputStream* nicht geschlossen werden. Das ELO FlowFile-Proxy-Objekt kann einen geschlossenen *InputStream* nach dem Ausführen der Methode nicht mehr in den FileStore hochladen. Wenn Sie z. B. *try-with-resources* verwenden oder den Stream manuell im *finally* Block schließen, kann der *InputStream* nicht mehr hochgeladen werden.

Beispiel: ELO FlowFile spezifisches Vorgehen

Das folgende Snippet zeigt ein Beispiel, wie Sie ein ELO FlowFile mit einem *InputStream* zurückgeben. Das Proxy-Objekt kümmert sich nach dem Upload um das Schließen des *InputStreams*.

```
@Component(namespace = "com.elo.flows", name = "ELOExample",
version = "v1", displayName = "eloexamplecomponent.displayname")
public class FlowFileSpecialityExampleComponent {

    @Service(name = "LoadTxtFile")
    public FlowFile loadTxtFile(final String txtFileName) throws IOException {
        InputStream inputSteam = getInputStreamOfTxtFile(txtFileName);
```

```
        return FlowFile.of("example.txt", inputSteam);
    }
}
```

Beispielkomponente

Das folgende Beispiel implementiert einen Zähler für Dokumente, die im Repository abgelegt sind. Der Zähler garantiert, dass die Dokumente eines bestimmten Typs (z.B. festgelegt durch den BS SOL_TYPE oder die Metadatenmaske) eine eindeutige fortlaufende Kennung bekommen. Beispiele können Vertrags- oder Rechnungsnummern sein.

> **Information**
>
> Im folgenden Beispiel wird die Implementierung der Java-Klasse `CounterComponent.java` (bzw. `CounterException.java` und `CounterInput.java`) gezeigt. Die Java-Klasse `CounterService.java` muss zuvor implementiert bzw. als File bereitgestellt werden. Sie beinhaltet die Implementierung der Logik des Zählers und kann sehr individuell angepasst werden.
>
> Die Deklaration der folgenden Beispielkomponente basiert auf Inhalten der Academy-Schulung ELO Flows Development aus dem Jahr 2021. Daher spiegelt sich die Academy-Umgebung u.a. im Namespace der Java-Klassen wider.

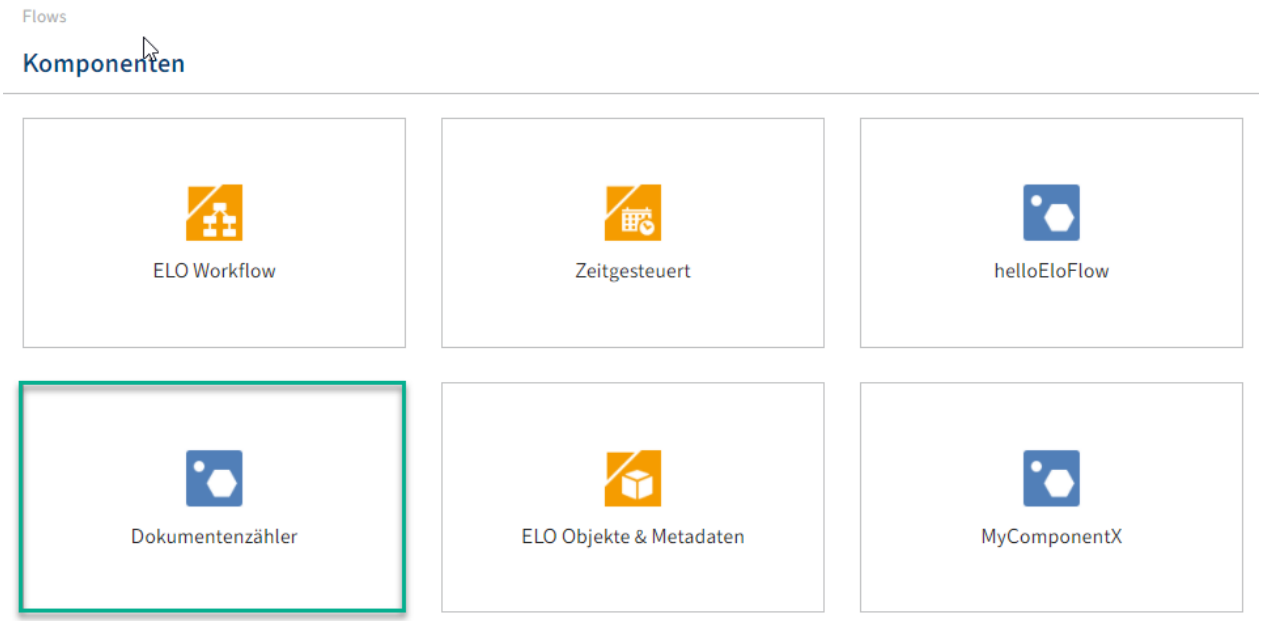

Abb. Komponentenübersicht mit 'Dokumentenzähler'

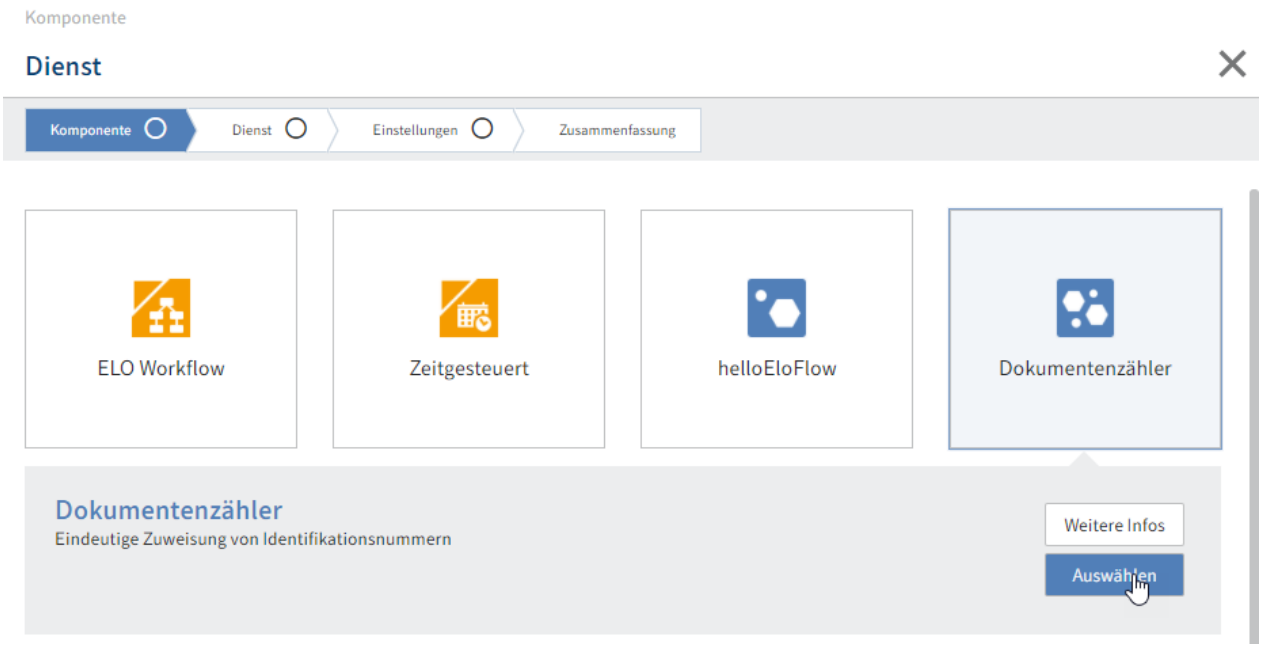

Abb. Dokumentenzähler als Dienst auswählen

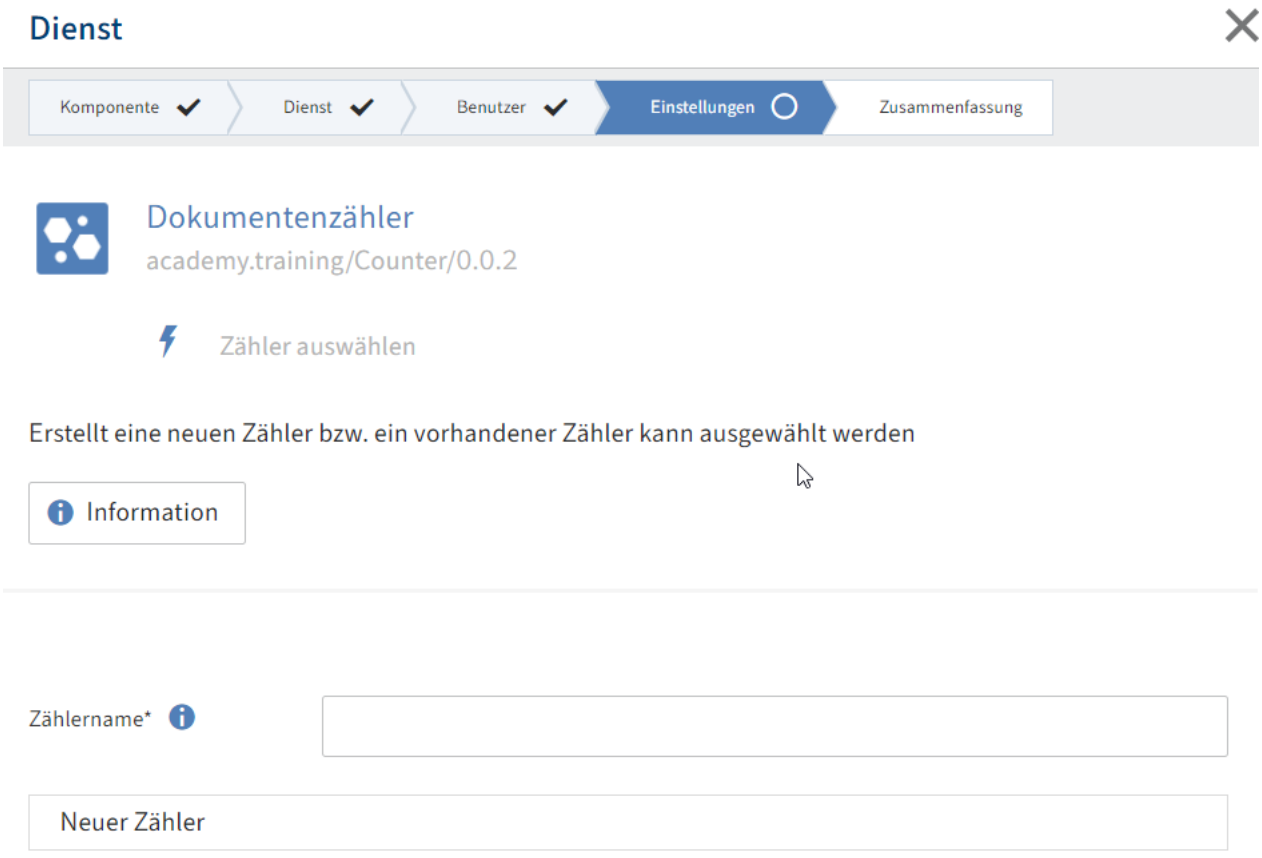

Abb. Eingabe eines neuen Zählers (Tab Einstellungen)

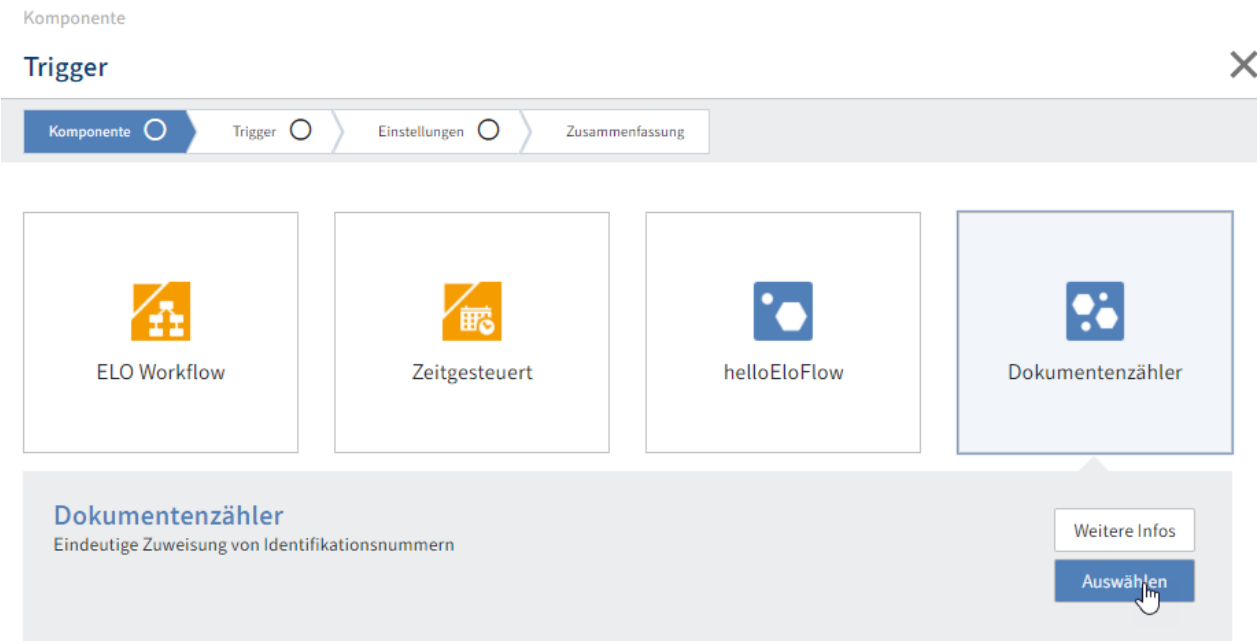

Abb. Dokumentenzähler als Trigger auswählen

Die fertige Komponente ermöglicht dem Benutzer folgendes:

- Auswahl eines Zählers (ein Zähler kann bereits vorhanden sein)
- Definition/Eingabe eines neuen Zählers (Prefix, Postfix - Anzahl der Stellen). Bsp. CounterInput.java.
- Erstellung eines neuen Zählers. Bsp. CounterComponent.java.
- Rückgabe des aktuellen Zählerstandes in weiteren Komponenten über einen Knotenschlüssel. Bsp. Schlüssel: `Counter.display.name`.
- Die neue Komponente als Trigger (REST) auswählen.

Framework für ELO Flows Komponentenentwicklung starten

1. Starten Sie VS Code und öffnen Sie den für die Umsetzung vorgesehenen Projektordner.
2. Erstellen Sie die Projektstruktur für die Flows Komponente.
3. Initialisieren Sie das Projekt als Java-Projekt, damit Erweiterungen für die Java-Entwicklung nachgeladen werden.
4. Bestätigen Sie bei der Initialisierung die jeweiligen Meldungen mit *Yes* und warten Sie, bis alle Erweiterungen nachgeladen sind.

> **Information**
>
> Vertiefende Informationen zum Anlegen der Komponente mit Projektstruktur und Java-Initialisierung finden Sie in der vorliegenden Dokumentation im Abschnitt Umgebung einrichten > Projekt anlegen.

Projektstruktur vorhandener Komponente anpassen

ELO hat in einer neu angelegten Flows Komponente eine Beispielkomponente implementiert. Sie können diese für neue Anforderungen anpassen oder durch vorheriges Löschen komplett neu aufsetzen. Im folgenden Beispiel gehen wir einen Mittelweg.

1.

Löschen Sie den Inhalt der Dateien im Ordner *localisation*:

> **Achtung**
>
> Löschen Sie keine Dateien. Löschen Sie ausschließlich die Inhalte der Dateien.

translations_de.properties und *translations.properties*.

2. Löschen Sie die Java-Klassen im Ordner *java\academy\training*:

   `HelloInput.java` und `HelloOutput.java

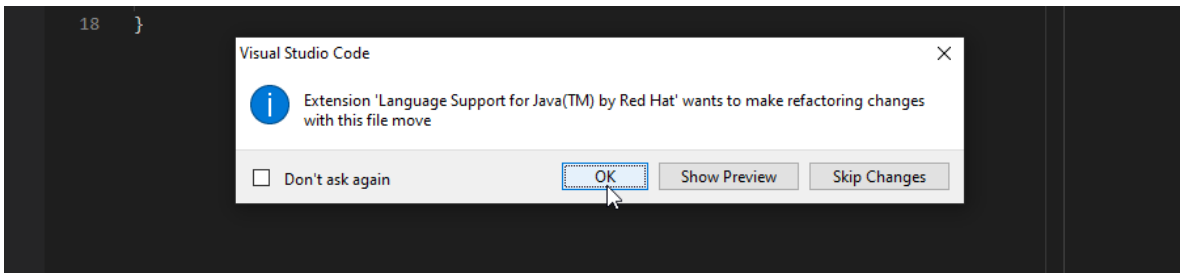

Abb.: Refactoring

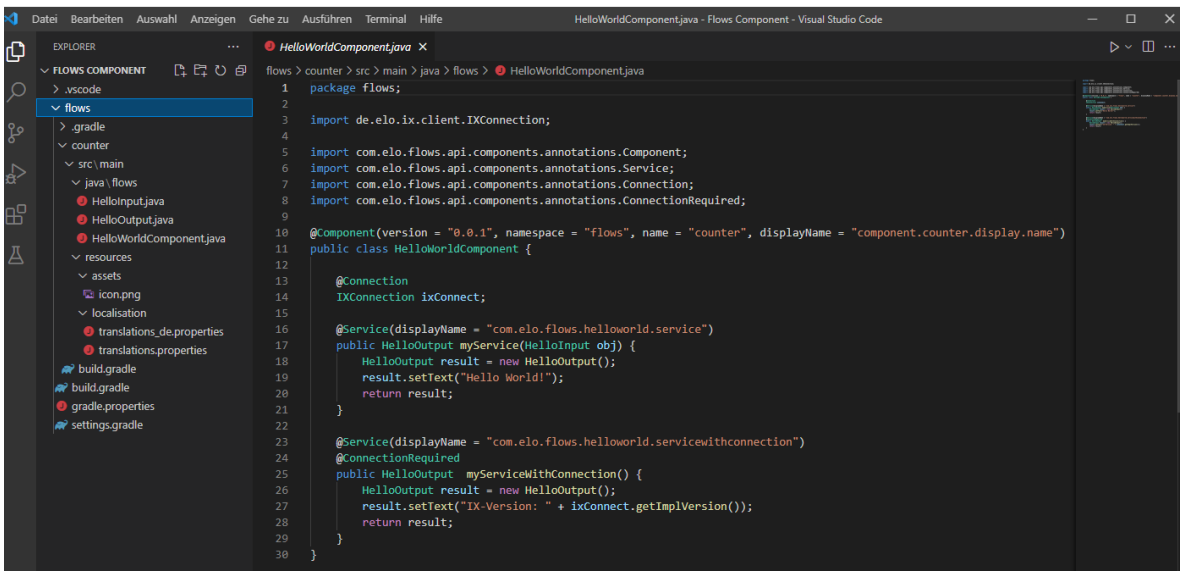

Abb.: Angepasste Java-Klasse

3. Passen Sie die Java-Klasse `HelloWorldComponent.java` an:

   Nutzen Sie hierzu die vom VS Code angebotene Refactoring-Möglichkeiten

translations_de.properties

flows > Counter > src > main > resources > localisation > translations_de.properties

```
1  Counter.display.name=Dokumentenzähler
```

translations.properties

flows > Counter > src > main > resources > localisation > translations.properties

```
1  Counter.display.name=Document Counter
```

Abb.: Lokalisierung

4. Tragen Sie folgendes in den Lokalisierungsdateien im Ordner *localisation* ein:

   `translations\_de.properties` und `translations.properties`

   Schlüssel: `Counter.display.name`

   Werte: `Dokumentenzähler` und `Document Counter`.

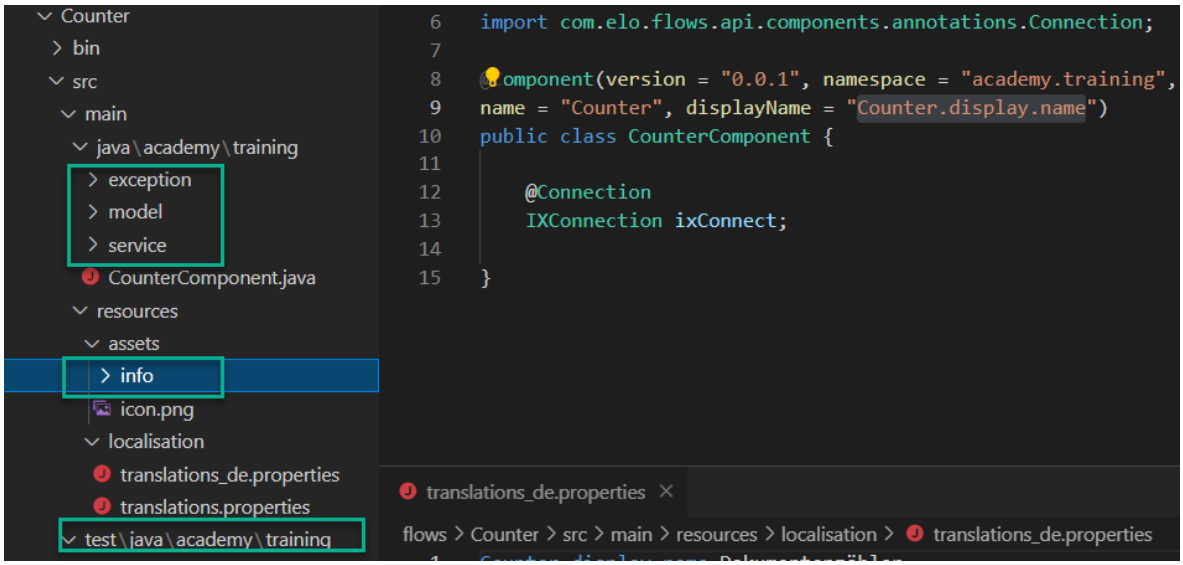

Abb.: Erweiterte Projektstruktur

5. Erweitern Sie die Projektstruktur um weitere Ebenen.

Verbindung zum ELO Indexserver herstellen

Die Annotation `@Connection` stellt automatisch eine Verbindung zum ELO Indexserver her. Über die Annotation `@Connection` wird die Verbindung zum ELOix in einer Klasse deklariert.

```
@Component(version = "0.0.1", namespace = "academy.training",
name = "Counter", displayName = "Counter.display.name")
public class CounterComponent {
  @Connection
  IXConnection ixConnect;
}
```

Sie können die Klasse über die Annotation `@ConnectionRequired` in jeder Methode verwenden.

```
public class CounterComponent {
   private static final Logger LOG =
   LoggerFactory.getLogger(CounterComponent.class);

@Connection
IXConnection ixConnect;

@LookupProvider("getCounters")
@ConnectionRequired
   public Map<String,String>; getCounters() throws CounterException {
   HashMap<String,String>; map = new HashMap<>;();
      try {
         CounterInfo[] counters = ixConnect.ix().checkoutCounters(null,
         false, LockC.NO);
         for (CounterInfo counterInfo : counters) {
            map.put(counterInfo.getName() + "["+ counterInfo.getValue() +"]", counterInfo.get
         }
      }
      catch (Exception e) {
         throw new CounterException(e.getMessage());
         }
      return map;
   }
}
```

Implementierung des Dokumentenzählers

Setzen Sie für die Deklaration folgende Implementierungsschritte um:

Logger für die Log-Ausgaben definieren

1. Verwenden Sie dazu folgenden Code:

```
private static final Logger LOG = LoggerFactory.getLogger(CounterComponent.class);
```

> **Information**
>
> Achten Sie darauf, dass die richtigen Imports ausgewählt werden.
>
> VS Code bietet Ihnen mehrere Import-Möglichkeiten für die Klasse Logger an (*import org.slf4j.Logger, org.slf4j.LoggerFactory*)

Methoden-Definition für den Dienst *Zähler* (Tab Dienst) implementieren

1.

Erstellen Sie eine neue Klasse (`CounterOutput.java`). Legen Sie die Klasse in der Projektstruktur unter *model* ab. Die Klasse gibt die Zählerinformationen über einen Schlüssel an die nachfolgende Komponenten in ELO Flows weiter.

```
package academy.training.model;
public class CounterOutput {
private String counterValue;
public String getCounterValue() {
   return counterValue;
}
public void setCounterValue(String counterValue) {
   this.counterValue = counterValue;
   }
}
```

2. Setzen Sie die Dienstimplementierung in der Klasse `CounterComponent.java` mithilfe der Annotation `@Service` und `@ConnectionRequired` um.

   Damit stellen Sie Folgendes sicher:

   - Der Dienst kann in der grafischen Benutzeroberfläche der Komponente ausgewählt werden
   - Der Dienst verfügt über die Verbindung zum ELO Indexserver

```
@Service(displayName = "Zähler auswählen")
@ConnectionRequired
public CounterOutput createCounter() {
   if (LOG.isDebugEnabled()) {
     LOG.debug("createCounter start");
   }
   CounterOutput counterOutput = new CounterOutput();
   String counterValue = "undefined";
   //TODO getCounterValue
   return counterOutput;
}
```

Eingabefelder für Zählername, Pre- und Postfix (Tab Einstellungen) erstellen

1. Für die Eingabefelder erstellen Sie eine neue Klasse (`CounterInput.java`).
2. Legen Sie diese in der Projektstruktur unter *model* ab.
3. In dieser Klasse weisen Sie via Annotationen die Elemente für die Zählerauswahl zu.

   Die Eingabefelder erscheinen später im Tab *Einstellungen*.

Annotation für Eingabefeld: @Property

Über die Annotation `@Property` definieren Sie die einzelnen Eingabefelder. Über `@PropertyGroups` und `@PropertyGroup` gruppieren Sie die Eingabefelder. Die Reihenfolge innerhalb der Gruppe legen Sie über `@DisplayOptions` fest.

Annotation für Vorschlagliste: @Lookup

Eine mögliche Lösung könnte wie folgt aussehen:

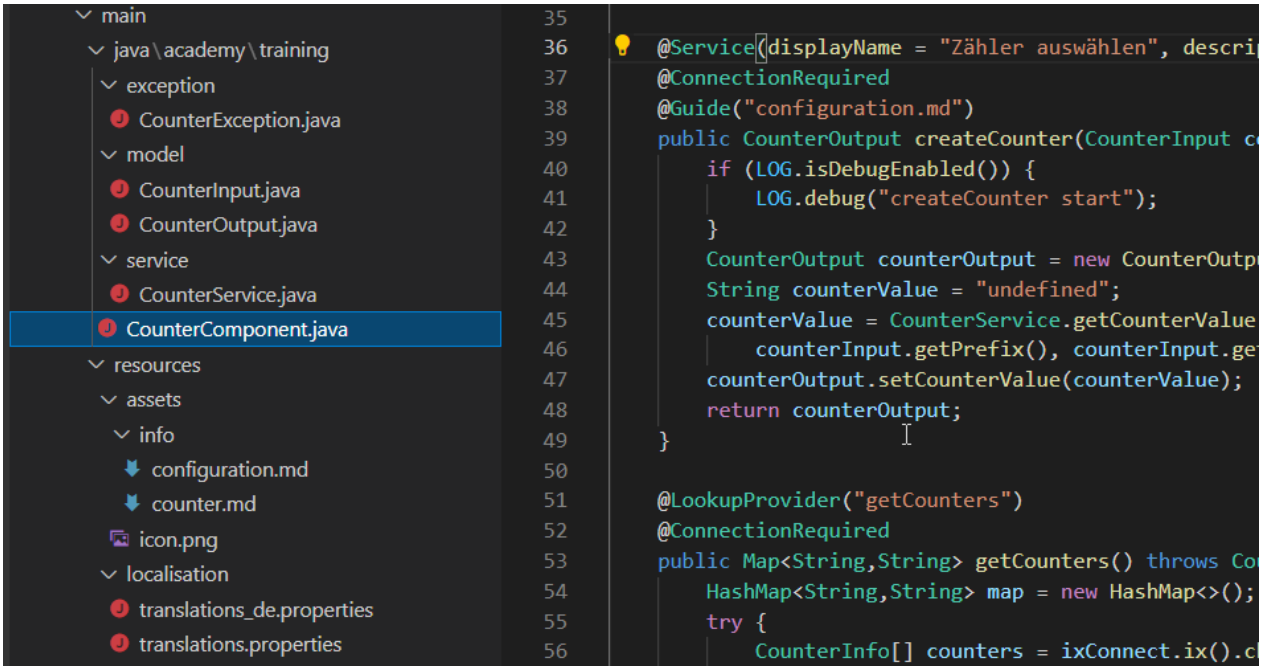

Abb.: Projektstruktur der Zählerlösung

Um eine Vorschlagliste mit bereits vorhandenen Zählern zu füllen, verwenden Sie die Annotation `@Lookup` und implementieren eine eigene Logik auf Basis der ELOix API.

> **Beachten Sie**
>
> Die Logik für die Vorschläge (`@Lookup`) müssen Sie unbedingt in der Komponentenklasse (`@Component`) programmieren.

1. Übergeben Sie innerhalb der Komponentenklasse (`@Component`) der Servicemethode (`@Service`) die neue Klasse (`CounterInput.java`) als Objekt.
2. Implementieren Sie die Logik für die Zählererstellung in der `CounterService.java` im Ordner *service*.

```
@Service(displayName = "Zähler auswählen")
@ConnectionRequired
public CounterOutput createCounter(CounterInput counterInput) {
…
}
```

REST-Trigger in der Komponente einbinden

Analog zu `@Service` implementieren Sie eine Trigger-Methode in der (`@Component`) Komponentenklasse. Die dazu notwendigen Annotationen sind `@Trigger` und `@WebHook`. Auch hier können Sie mit Input- und Output-Parametern arbeiten.

Beispielprogrammierung der Klassen 'CounterException', 'CounterInput', 'CounterComponent'

Die folgenden Code-Beispiele zeigen Beispielprogrammierungen der Java-Klassen *CounterException*, *CounterInput*, *CounterComponent*.

CounterException.java

```
public class CounterException extends Exception {
   public final static String COUNTERERROR = "Fehler in der Zählererstellung";
   public CounterException() {
      super(CounterException.COUNTERERROR);
   }
   public CounterException(String errorMessage) {
      super(errorMessage);
   }
}
```

CounterInput.java

```
public class CounterException extends Exception {
package academy.training.model;

import com.elo.flows.api.components.annotations.Guide;
import com.elo.flows.api.schema.annotations.DisplayOptions;
import com.elo.flows.api.schema.annotations.Lookup;
import com.elo.flows.api.schema.annotations.Property;
import com.elo.flows.api.schema.annotations.PropertyGroup;
import com.elo.flows.api.schema.annotations.PropertyGroupRef;
import com.elo.flows.api.schema.annotations.PropertyGroups;

@Guide("configuration.md")
@PropertyGroups({@PropertyGroup(displayName ="Neuer Zähler", name="groupNewCounter")})
public class CounterInput{

   @Property(displayName = "Zählername", required=true, description="Beim neuen Zähler bitte
   @Lookup("getCounters")
   private String counterName;

   @Property(displayName ="Präfix", description="Bitte den Zählername oben angeben")
   @PropertyGroupRef("groupNewCounter")
```

```
    @DisplayOptions(order = 1)
    private String prefix = "test";

    @Property(displayName ="Postfix", description="Bitte auch den Zählername oben angeben")
    @PropertyGroupRef("groupNewCounter")
    @DisplayOptions(order = 2, suggestValue = true)
    private int postfix = 6;
    public String getCounterName() {
        return counterName;
    }
    public void setCounterName(String counterName) {
        this.counterName = counterName;
    }
    public String getPrefix() {
        return prefix;
    }
    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }
    public int getPostfix() {
        return postfix;
    }
    public void setPostfix(int postfix) {
        this.postfix = postfix;
    }
}
```

CounterComponent.java

```
package academy.training;

import de.elo.ix.client.CounterInfo;
import de.elo.ix.client.IXConnection;
import de.elo.ix.client.LockC;

import java.util.HashMap;
import java.util.Map;

import com.elo.flows.api.components.annotations.Component;
import com.elo.flows.api.components.annotations.Connection;
import com.elo.flows.api.components.annotations.ConnectionRequired;
import com.elo.flows.api.components.annotations.Guide;
import com.elo.flows.api.components.annotations.LookupProvider;
import com.elo.flows.api.components.annotations.Service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import academy.training.exception.CounterException;
import academy.training.model.CounterInput;
import academy.training.model.CounterOutput;
import academy.training.service.CounterService;

@Component(version = "0.0.1", namespace = "academy.training",
name = "Counter", displayName = "Counter.display.name")
@Guide("counter.md")
public class CounterComponent {

   private static final Logger LOG = LoggerFactory.getLogger(CounterComponent.class);
   @Connection
   IXConnection ixConnect;
   @Service(displayName = "Zähler auswählen", description = "Wählen Sie einen Zähler für Ihr
   @ConnectionRequired
   @Guide("configuration.md")
   public CounterOutput createCounter(CounterInput counterInput) {
      if (LOG.isDebugEnabled()) {
        LOG.debug("createCounter start");
      }
      CounterOutput counterOutput = new CounterOutput();
      String counterValue = "undefined";
      counterValue = CounterService.getCounterValue(ixConnect, counterInput.getCounterName(),
        counterInput.getPrefix(), counterInput.getPostfix());
      counterOutput.setCounterValue(counterValue);
      return counterOutput;
   }

   @LookupProvider("getCounters")
   @ConnectionRequired
   public Map<String,String> getCounters() throws CounterException {
   HashMap<String,String> map = new HashMap<>();
   try {
        CounterInfo[] counters = ixConnect.ix().checkoutCounters(null, true, LockC.NO);
         for (CounterInfo counterInfo : counters) {
            map.put(counterInfo.getName(), counterInfo.getName());
         }
      }
      catch (Exception e) {
         throw new CounterException(e.getMessage());
      }
      return map;
   }
}
```

Weitere mögliche Erweiterungen der Komponente

Abb.: JUnit Bibliotheken einbinden

1. Test-Framework JUnit einbinden

   In der Konfigurationsdatei *build.gradle* folgenden Eintrag hinzufügen.

   > **Information**
   >
   > Achten Sie darauf, dass Sie die Einträge in der richtigen build.gradle ergänzen.

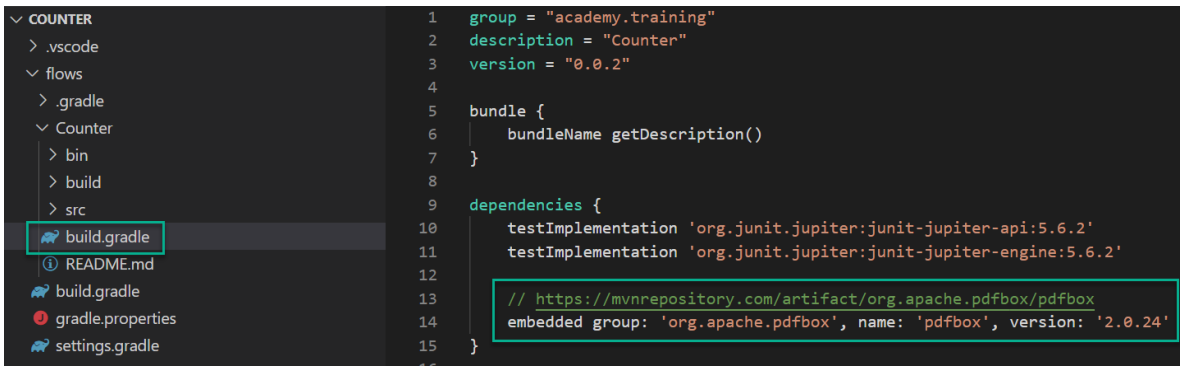

   Abb.: PDFBox über Maven einbinden

2. Neue Bibliotheken über Maven einbinden
3. Eigene Bibliotheken einbinden

Build-Prozess

ELO Dev Plug-in stellt die Funktion *Buildaufgabe* bereit, um die Komponente als JAR-Datei zu bauen. Diese kann anschließend im ELO Flows Worker (Apache Karaf) getestet werden.

Buildaufgabe ausführen

Gehen Sie wie folgt vor, um alle Komponenten im geöffneten Ordner zu bauen.

1. Führen Sie den Auswahldialog über Terminal > *Buildaufgabe ausführen...* aus.

   Alternativ: Drücken Sie STRG + SHIFT + B

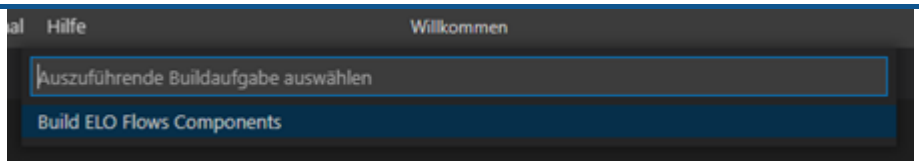

Abb.: Build ELO Flows Components

2. Wählen Sie *Build ELO Flows Components* aus.

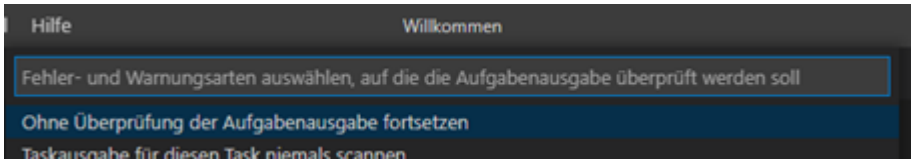

Abb.: Ohne Überprüfung der Aufgabenausgabe fortsetzen

3. Wählen Sie *Ohne Überprüfung der Aufgabenausgabe fortsetzen* aus.

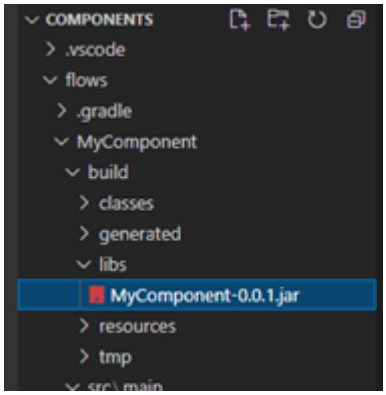

Abb.: Erstellte Komponente als JAR-Datei in Build-Ordner

Die erstellte Komponente liegt nun als JAR-Datei im Build-Ordner des Projektes.

Information

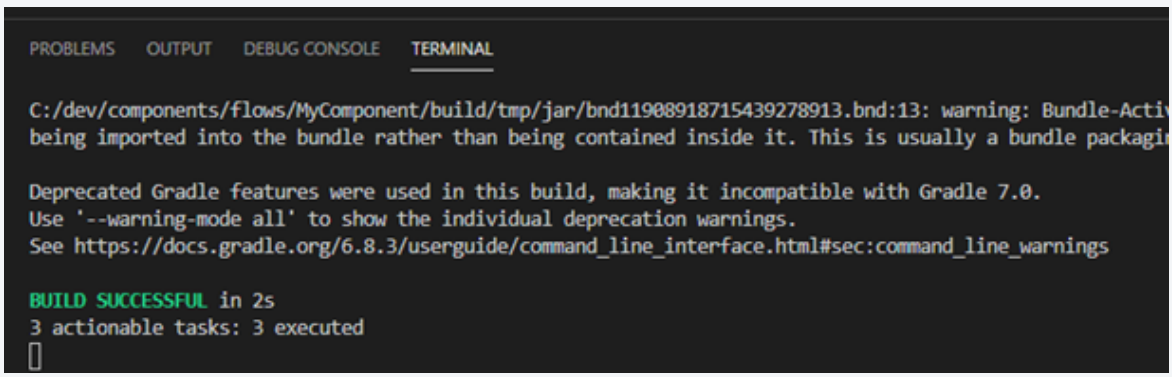

Abb.: Hinweis in Terminal: Build Successful

Der erste Build kann ein bisschen länger dauern. Sobald der Build-Prozess erfolgreich abgeschlossen ist, erscheint im Terminal der Hinweis "BUILD SUCCESSFUL".

Deployment

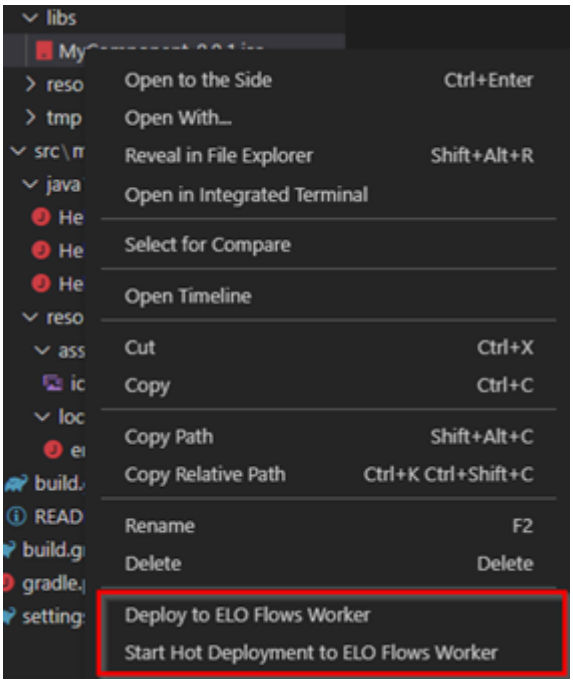

Abb.: Auswahl der Deployment-Optionen in VS Code

Stellen Sie Ihre neue Komponente in ELO Flows zur Verfügung. Rollen Sie hierzu nach dem Build-Prozess die JAR-Datei in ELO Flows Worker (Apache Karaf) aus.

ELO Dev Plug-in für VS Code stellt hierzu zwei Optionen bereit. In beiden Fällen muss Apache Karaf lokal laufen und erreichbar sein.

Einmaliges Ausrollen: Deploy to ELO Flows Worker

Mit der Option *Deploy to ELO Flows Worker* rollen Sie die neu gebaute JAR-Datei im ELO Flows Worker (Apache Karaf) einmalig aus. Diese Option empfiehlt sich vor allem am Anfang der Entwicklung, wenn man größere Änderungen macht und die Komponente höchstwahrscheinlich noch (Kompilierungs-) Fehler enthält.

Start von Deploy to ELO Flows Worker

1. Stoßen Sie einen Build-Prozess nach Ihren Änderungen an.
2. Klicken Sie mit der rechten Maustaste auf die JAR-Datei der neu gebauten Komponente.
3. Wählen Sie die Option *Deploy to ELO Flows Worker* aus. Die JAR-Datei wird nun im ELO Flows Worker ausgerollt.

Ergebnis von Deploy to ELO Flows Worker

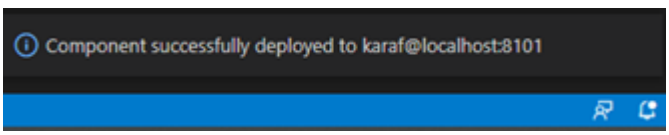

Abb.: Meldung in VS Code bei erfolgreichem einmaligen Ausrollen

Ein Pop-up-Fenster am unteren Bildschirmrand signalisiert das erfolgreiche Ausrollen.

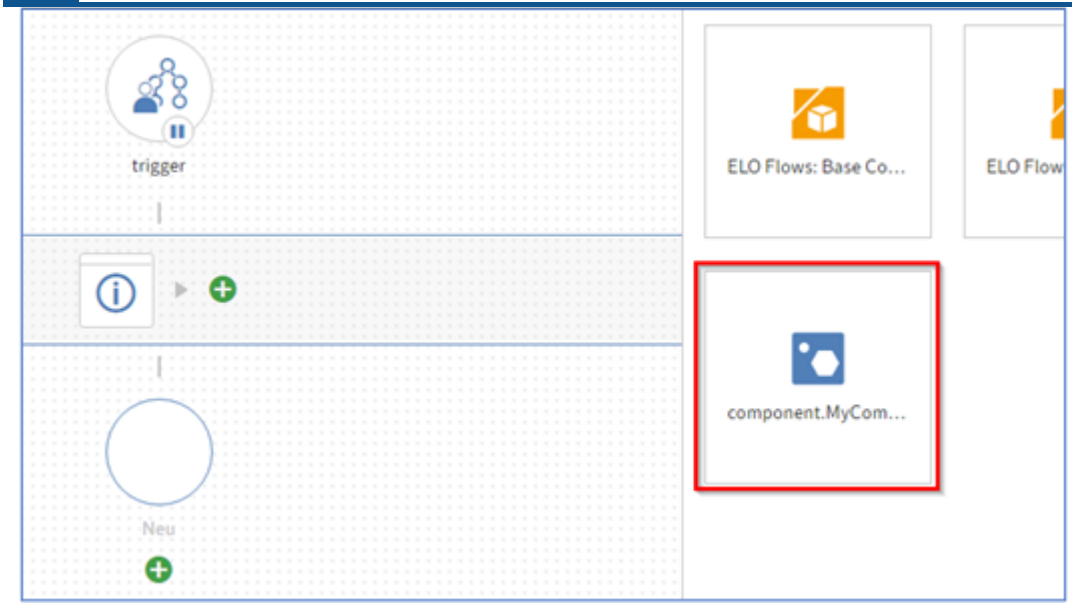

Abb.: Neue Komponente in der Oberfläche von ELO Flows nach Aktualisierung der Seite

1. Aktualisieren Sie die Oberfläche von ELO Flows.

Beenden von Deploy to ELO Flows Worker

1. Das einmalige Ausrollen der Komponente ist nach dem Vorgang abgeschlossen. Starten Sie den Deploy-Prozess erneut bei späteren Änderungen der Komponente.

Kontinuierliches Ausrollen: Hot Deployment to ELO Flows Worker

Die Option *Start Hot Deployment to ELO Flows Worker* startet einen Hintergrundprozess, welcher Änderungen an der Komponente überwacht. Nach dem Start erscheint im Terminal die Meldung, dass auf Änderungen gewartet wird. Wenn Sie etwas an der Komponente ändern, wird die Änderung nach dem Speichern automatisch gebaut und im Karaf ausgerollt.

> **Beachten Sie**
>
> Damit Sie kontrollieren können was für Änderungen ausgerollt werden, muss die AutoSave-Funktion von VS Code deaktiviert sein.

Start von Hot Deployment to ELO Flows Worker

1. Klicken Sie mit der rechten Maustaste auf die JAR-Datei der neu gebauten Komponente.
2. Wählen Sie die Option *Hot Deployment to ELO Flows Worker* aus.

Ergebnis von Hot Deployment to ELO Flows Worker

```
Waiting for changes to input files of tasks... (ctrl-d then enter to exit)
<-------------> 0% WAITING
> IDLE
```

Abb.: Meldung im Terminal bei aktivem kontinuierlichen Ausrollen

Die JAR-Datei wird nun bei jedem Speichervorgang automatisch im ELO Flows Worker ausgerollt.

1. Aktualisieren Sie die Oberfläche von ELO Flows.

Beenden von Hot Deployment to ELO Flows Worker

1. Drücken Sie im Terminal von VS Code STRG + D und anschließend ENTER, um das automatische Ausrollen zu beenden.

Debugging

Das Debugging erfolgt direkt über Java-Remote-Debugging in VS-Code. Dies ist möglich, da ELO Flows-Worker und die Java-Extensions für VS-Code Java-Remote-Debugging unterstützen.

Voraussetzung für Debugging

In der Standardinstallation von Flows über das ELO-Setup ist die Debugging-Funktion deaktiviert. Der Debug-Modus im Flows-Worker (Apache Karaf lokal) muss manuell aktiviert werden.

Flows Worker (Apache Karaf) muss lokal im Debug-Modus laufen und erreichbar sein.

Debug-Modus aktivieren

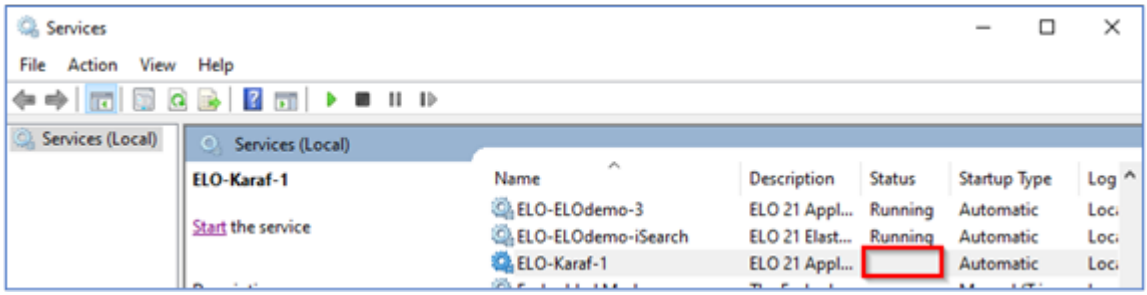

Abb.: Apache Karaf in ELO Server Setup beenden

1. Beenden Sie den von ELO Server Setup eingerichteten Service.

> **Information**
>
> Wurde Apache Karaf manuelle installiert, entfällt dieser Schritt.

2. Starten Sie anschließend Karaf über die Kommandozeile im Debug-Modus mit folgendem Befehl:

```
C:\ELOprofessional\servers\ELO-Karaf-1\bin> .\karaf debug
```

Debugger in VS Code starten

1. Aktuellste Version der Komponente im Apache Karaf deployen.
2. Apache Karaf im Debug-Modus starten.
3.

Drücken Sie F5

Abb.: Statusleiste in VS Code färbt sich orange

Abb.: Toolbar im Debugging-Modus

Sie erkennen an der orange eingefärbten Statusleiste und der Debugging-Toolbar, dass der Debugger korrekt starten und sich mit Karaf verbinden konnte.

Debuggen

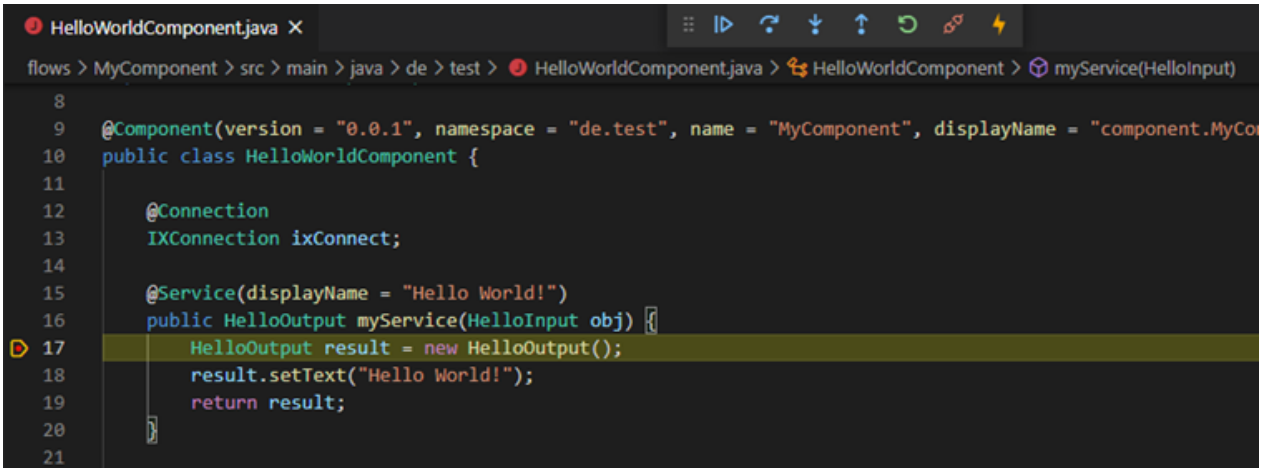

Abb.: VS Code beim Debugging

1. Setzen Sie wie gewohnt Breakpoints. Der Debugger hält dann beim Ausführen eines Flows mit der Komponente an der Stelle an.

Debugging in VS Code beenden

1. Beenden Sie den Debugger über die Funktion *Disconnect* in der Debugging-Toolbar.

   Alternativ: Drücken Sie SHIFT + F5

2. Beenden Sie das Karaf in der Konsole über STRG + D.

3. Starten Sie den zuvor beendeten Apache Karaf-Service wieder.

Konfigurationsoptionen der Komponente

Grafischen Elemente

Überblick über die grafischen Elemente, die in einer Komponente verwendet werden können:

- Komponente (`@Component`)
- Trigger (`@Trigger`, `@Webhook`, `@Config`, `@Scheduled`, `@Synchron`)
- Dienst (`@Service`, `@SchemaProvider`)
- Connection-Provider (`@Create`, `@Destroy`, `@Prepare`, `@Refresh`, `@Validate`)
-

Eingabefeld (String) (`@Property`)
- Eingabefeld (int) (`@Property`)
- Eingabefeld (boolean) (`@Property`)
- Eingabefeld (Object) (`@Property`)
- Eingabefeld (Array) (`@Property`)
- Eingabefeld [Pflichtfeld] (`@Property required=true`)
- Zusätzliche Konfigurationsmöglichkeiten eines Eingabefeldes (`@DisplayOptions`)
- Vorschläge (`@Lookup`)
- Gruppen in den Einstellungen (`@PropertyGroups`, `@PropertyGroup`, `@PropertyGroupRef`)
- Gruppieren von Diensten (`@ComponentServices`)

Annotation

Annotation, die zusätzlich mit den grafischen Elementen kombiniert werden können:

- Informationen (`@Guide`)
- Verbindung zum ELO Indexserver (`@Connection`, `@ConnectionRequired`)
- Verknüpfung mit Schema Provider (`@DynamicSchema`)
- Eingabefeld / Eigenschaft wird ignoriert (`@PropertyIgnore`)
- Überschreiben des Eigenschaftswerts, z.B. Übersetzung von Enum-Werten (`@PropertyValue`)

Werkzeugklassen/Utility-Classes

Werkzeugklassen bzw. Utility-Classes, die zusätzlich eingesetzt werden können:

- Wrapper-Klasse für das Ergebnis eines Triggers oder Dienstes, z.B. zum Abbrechen eines Flows (`ActionResponse`)
- Proxy-Objekt für das Senden und Empfangen von Dateien zwischen Flow-Knoten (Flow-File)
- Validierungsergebnis der mit `@Validate` annotierten Methode eines `ConnectionProviders` (`ValidationResult`)