ELO Indexserver Plug-in-Entwicklung

[Stand: 11.09.2019 | Programmversion: 12.00.000]

Mithilfe von OSGi-Plug-ins kann die Funktionalität des ELO Indexservers über die bisherigen Skripting-Möglichkeiten hinaus erweitert werden.

Inhalt

1 Überblick 5
1.1 Kurzbeschreibung OSGi 5
2 Plug-in-Entwicklung mit Eclipse 7
2.1 Voraussetzungen 7
2.1.1 ELO Serverinstallation 7
2.1.2 Java JDK 8 installieren 7
2.1.3 Eclipse installieren 7
2.1.4 Tomcat Server Runtime einrichten 7
2.2 Konfiguration für weiteren ELO Indexserver vorbereiten 8
2.2.1 Zusätzliche IX-Optionen 8
2.2.2 Konfigurationsdateien duplizieren 9
2.3 Workspace einrichten 10
2.3.1 JDK 8 einrichten 11
2.3.2 Server Runtime einrichten 15
2.3.3 Server-Runtime für den Workspace auswählen 19
2.3.4 ELO Indexserver in Workspace aufnehmen 25
2.3.5 ELO Indexserver deployen und starten 30
2.3.6 Projektverzeichnisse als Plug-in-Verzeichnisse festlegen 32
2.3.7 Target Platform für die Plug-in-Entwicklung im Eclipse definieren 33
2.3.8 Quellcode und Dokumentation der EloixClient.jar bereitstellen 36

2.4 Plug-in erstellen ... 42
2.4.1 Neues Plug-in-Projekt erstellen ... 42
2.4.2 Activator-Klasse anpassen ... 47
2.4.3 Schnittstelle für Server-Ereignisse IXServerEvents implementieren ... 48
2.4.4 Registrierte Funktionen bereitstellen ... 50
2.5 Plug-in debuggen ... 51
2.5.1 Kunstgriff ... 51
2.5.2 Prüfung, dass das Plug-in geladen wird ... 53
2.5.3 Quellcode zuweisen ... 55
2.6 Abhängigkeiten einbinden ... 58
2.6.1 Abhängiges Bundle referenzieren ... 58
2.6.2 JAR-Bibiliothek in das Plug-in einbetten ... 68
2.7 Plug-in fertigstellen ... 72
2.7.1 Vorbereitung ... 72
2.7.2 Plug-in-Verzeichnis als Option des ELO Indexservers einstellen ... 73
2.7.3 Plug-in exportieren ... 73
3 Services eines Plug-ins ... 76
3.1 Plug-in-Service InitializePlugin ... 77
3.2 Plug-in-Service WorkflowNodeEventsFactory ... 78
3.3 Plug-in-Service ‚DynamicKeywordFactory' ... 79
3.4 Plug-in-Service im Skripting aufrufen ... 81
3.5 Plug-in-Service in einem anderen Plug-in verwenden ... 82
3.6 HTTP-Anfragen im Plug-in verarbeiten ... 83
3.6.1 URI ohne Authentifizierung ... 84
3.6.2 URI mit Single-Sign-On ... 84
3.6.3 URI mit Tomcat-Authentifizierung ... 84
3.6.4 Servlet-Funktionen ... 84
3.6.5 Beispiel: Dokument zu gegebener GUID lesen ... 85
3.6.6 Beispiel: Zusammenspiel mit JSON-API ... 88
3.6.7 Hinweis: Browser-Anwendung sollte mit Session-Cookie statt ELO-Ticket arbeiten 94

4 Services des ELO Indexservers ... 96
4.1 Datenbankzugriff ... 97
5 Rest-Plug-in ... 99
5.1 Vorbereitung ... 99
5.2 Hello-World-Beispiel ... 100
5.3 IXConnection in Ressource-Klassen ... 102
5.4 Authentifizierung ... 104
5.5 Konvention zu den URIs der Ressource-Klassen ... 105
5.6 Weitere Funktionalitäten der JAX-RS-Spezifikation ... 105
5.6.1 Serialisierung von Objekten in XML oder JSON ... 105
5.6.2 Anforderungen an eine serialisierbare Klasse ... 105
5.6.3 Klassen als Rückgabewerte von Funktionen ... 106
5.6.4 Klassen als Parameter von Funktionen ... 107
5.6.5 Datei über eine Webseite hochladen ... 108
5.7 Debugging in den Quellcode ... 109
6 Proxy-Plug-in ... 110
6.1 Konfiguration ... 110
6.2 Übergabe der Benutzer-Session ... 111
6.2.1 Ticket-Cookie ... 111
6.2.2 Session-Transformation ... 111
6.2.3 Location-Header, URL Redirect ... 111
6.2.4 Weitere Cookies und Header ... 111
6.3 Debugging in den Quellcode ... 111
7 Pug-ins über die API des ELO Indexservers verwalten ... 112
7.1 Beispiel zum Auflisten der Plug-ins ... 112
7.2 Beispiel zum Stoppen und Starten eines Plug-ins ... 112
7.3 Beispiel zum Installieren und Deinstallieren eines Plug-ins ... 112
7.4 Beispiel zum Lesen und Ändern der Plug-in-Konfiguration ... 113
8 Relevante Log-Einträge ... 114
8.1 Verzeichnisse aus denen Plug-ins geladen werden ... 114
8.2 OSGi-Framework startet ... 114

8.3 Vom ELOix bereitgestellte Bundles werden installiert ... 114
8.4 Suche nach zu installierenden Plug-ins ... 115
8.5 Start der Plug-ins ... 115
8.6 Start des OSGi-Frameworks ... 115
9 FAQ ... 117
9.1 ELO Indexserver startet nicht ... 117
9.1.1 Fehlermeldung im ix-log „java.lang.NoClassDefFoundError: org/osgi/framework/FrameworkListener“ ... 117
9.1.2 Fehlermeldung im ix-log „java.lang.ClassNotFoundException: com.microsoft.sqlserver.jdbc.SQLServerDriver ... 117
9.2 Plug-in wird nicht geladen ... 117
9.2.1 Fehlermeldung im ix-log „org.osgi.framework.BundleException: Unable to resolve …: missing requirement…“: ... 117
9.2.2 Fehlermeldung „OSGi framework error…. Not found: …Activator“ ... 117
9.3 Plug-in-Services nicht verfügbar ... 117
9.4 Compiler-Fehler ... 118
9.4.1 PluginActivator cannot be resolved to a type ... 118
9.4.2 Nach Eclipse-Neustart Fehlersymbol auf dem Plug-in-Projekt ... 118
9.4.3 Plug-ins „out of sync“ ... 119
9.5 Eclipse hängt nach dem Laden des Workspace ... 119
10 Literaturverzeichnis ... 120

1 Überblick

In den Indexserver-Versionen bis ELO 10 können serverseitige Ereignisse (bspw. das Betreten eines Workflow-Knotens) ausschließlich über JavaScript behandelt werden. Dadurch ergeben sich die folgenden Nachteile:

- Der Skript-Quellcode liegt offen und kann von jedem (Rechte im Archiv vorausgesetzt) kopiert und möglicherweise fehlerhaft verändert werden.
- Wenn Code aus einer Java-Bibliothek aufgerufen werden soll, muss die Bibliothek in die Indexserver-Anwendung eingefügt werden – entweder in die WAR-Datei oder ins Tomcat/webapps-Verzeichnis. Zum einen ist dieses Vorgehen umständlich,
- zum anderen können Standard-Bibliotheken hinzugefügt werden, die einen Versionskonflikt mit den im Indexserver vorhandenen Bibliotheken auslösen.

Um diese Nachteile auszugleichen, bietet die Indexserver-Version 11 die Entwicklung von Java-Plug-ins an. Dabei handelt es sich im Kern um OSGi-Bundles [1]. Sie werden im Indexserver im Apache Felix Framework [2] ausgeführt.

Die Indexserver-Version 11 erlaubt ferner eine Erweiterung der Schnittstelle um selbstdefinierte HTTP- und REST-Aufrufe. Für solche Aufrufe kann optional die SSO-Authentifizierung des Indexservers genutzt werden. In den Aufrufen steht die gesamte Indexserver-API zur Verfügung.

1.1 Kurzbeschreibung OSGi

OSGi ist eine Softwareplattform auf Basis von Java, die die Entwicklung von modularen Anwendungen erleichtert. Diese Technik gibt es bereits seit dem Jahr 2000 und hat sich beispielsweise in der Entwicklungsumgebung Eclipse bewährt, um verschiedene Editoren, Debugger, Versionsverwaltungen uvm. in einem Anwendungsrahmen auszuführen.

Ein OGSi-Bundle stellt ein Modul im OSGi-Framework dar, das Dienste anbietet oder benutzt. Im Manifest ist beschrieben, welche Java-Packages von anderen Bundles importiert und welche Packages anderen Bundles zur Verfügung gestellt (exportiert) werden. Klassen in nicht-exportierten Packages sind nur innerhalb des Bundles sichtbar.

Auf der Festplatte ist ein Bundle eine (einzige) JAR-Datei mit erweiterten Einträgen in der *MANIFEST.MF*. Dazu gehören neben den bereits erwähnten, importierten und exportierten Packages (Bundle-ExportPackages), beispielsweise eine Liste von abhängigen Bundles (Require-Bundle), eine Versionsnummer (Bundle-Version) und ein symbolischer Name (Bundle-SymbolicName), der die eindeutige ID eines Bundles festlegt und die einzige obligatorische Eigenschaft ist. Benötigt das Bundle gewöhnliche Java-Bibliotheken, können diese in die Bundle-JAR eingebettet werden.

Beispiel für eine *MANIFEST.MF*:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: First-plugin
Bundle-SymbolicName: my.namespace.firstplugin
Bundle-Version: 1.0.0.0
Bundle-Activator: my.namespace.Activator
Bundle-Vendor: My Plugin Company
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: org.osgi.framework;version="1.3.0"
Require-Bundle: de.elo.ix.client;bundle-version="10.99.1",
 org.apache.httpcomponents.httpclient;bundle-version="4.5.2",
 org.apache.httpcomponents.httpcore;bundle-version="4.4.6",
 org.apache.commons.codec;bundle-version="1.9.0",
 org.apache.commons.logging;bundle-version="1.1.1"
Bundle-ClassPath: lib/foo.jar
```

Im Folgenden wird mit „Plug-in“ ein OGSi-Bundle bezeichnet, das Services (z. B. eine Implementierung von `IXServerEvents`) für den Indexserver anbietet.

In den nächsten Abschnitten wird an einem Beispiel die Entwicklung eines Plugins schrittweise gezeigt und das nötige Wissen vermittelt, sodass der Leser in der Lage ist, eigene Plug-ins zu schreiben.

Wenn darüber hinaus ein tieferes Verständnis der OSGi-Technik gewünscht ist, empfiehlt es sich, mit einem der Tutorials im Web zu starten, beispielsweise Java World, Bundles for beginners [3]. Zum Experimentieren mit OSGi kann die Apache Felix Framework Distribution [4] hilfreich sein.

Fundiertes OSGi-Wissen ist aber keine notwendige Voraussetzung für die Entwicklung von Indexserver Plug-ins.

2 Plug-in-Entwicklung mit Eclipse

Indexserver Plug-ins können prinzipiell mit jeder Entwicklungsumgebung für Java entwickelt werden. Mit Eclipse ist das Debugging jedoch besonders einfach, weil das Plug-in direkt im Debugging-Prozess ausgeführt werden kann.

Mit Netbeans ist nur das Remote-Debugging handhabbar, weil es den Tomcat nicht zu jeder Debug-Sitzung neu startet sondern ein Hot-Deployment durchführt. Benötigt der Indexserver länger zum Stoppen, kommt es zu Fehlern beim Deployment.

In den folgenden Abschnitten wird deshalb eine Entwicklungsumgebung auf der Basis von Eclipse aufgebaut. Sie startet einen separaten Indexserver in einem separaten Tomcat. Der Indexserver der ELO-Installation wird nicht für die Plug-in-Entwicklung benötigt.

2.1 Voraussetzungen

2.1.1 ELO Serverinstallation

- Es wird eine ELO Serverinstallation ab Version 11 benötigt. Das Installationsverzeichnis wird im Folgenden `ELODir` genannt. Das zum Debugging verwendete Testarchiv wird `archive1` genannt.

2.1.2 Java JDK 8 installieren

- Aktuelle Version des JDK der Version 8 von Oracle [5] herunterladen und installieren. Java Version 9 wird noch nicht unterstützt.

2.1.3 Eclipse installieren

- Aktuelle Version von Eclipse [6] herunterladen.
- Bei der Installation die „Eclipse IDE for Java **EE** Developers“ wählen.

2.1.4 Tomcat Server Runtime einrichten

- Aktuellen Tomcat 8.0 oder 8.5 von der Webseite Apache [7] unter „Binary Distributions, Core“ laden. Für Entwicklung unter Windows die Windows 64bit ZIP-Datei (nicht MSI) laden.
- ZIP in ein Verzeichnis für ausführbare Programme entpacken, `TomcatDir`.
- JDBC-Treiber kopieren: z. B. aus einer vorhandenen ELO-Server-Installation die benötigten JDBC-Treiber kopieren und in das Verzeichnis `TomcatDir/lib` einfügen. Die Treiber heißen `sqljdbc*.jar, ojdbc*.jar, db2*.jar, postgresql*.jar`.

- OSGi Framework Bibliotheken von Apache Felix, `org.apache.felix.main*.jar` und `org.apache.felix.framework*.jar`, in das Verzeichnis `TomcatDir/lib` einfügen: Die benötigten Dateien aus einer vorhandenen ELO-Installation kopieren oder im Maven Central Repository [8] herunterladen.

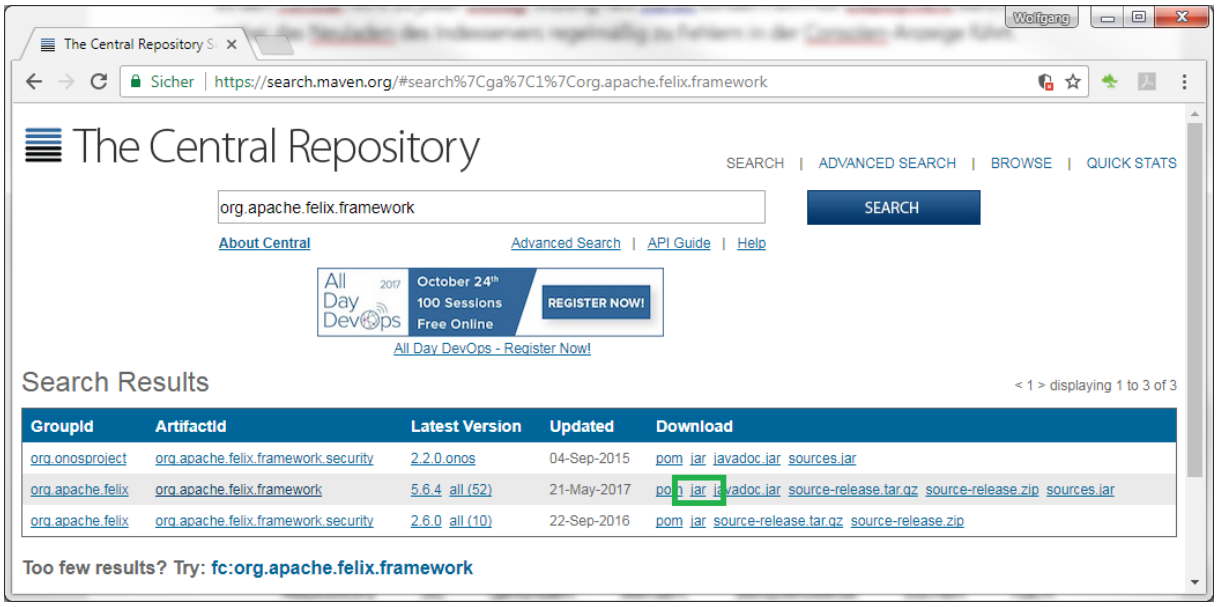


Abb. 1: Downlowad aus Maven Central Repository

2.2 Konfiguration für weiteren ELO Indexserver vorbereiten

Weil für das Debugging ein weiterer ELOix betrieben wird, muss eine zusätzliche ELOix Konfiguration für das Testarchiv eingerichtet werden. Der zusätzliche ELOix wird hier mit dem Instanznamen *DEBUG* belegt.

2.2.1 Zusätzliche IX-Optionen

Über die Statusseite des ELOix erreichen Sie unter dem Link *Configure Options* die Management-Seite des ELO Indexservers, in der die folgenden Optionen ergänzt werden. Beispiel:

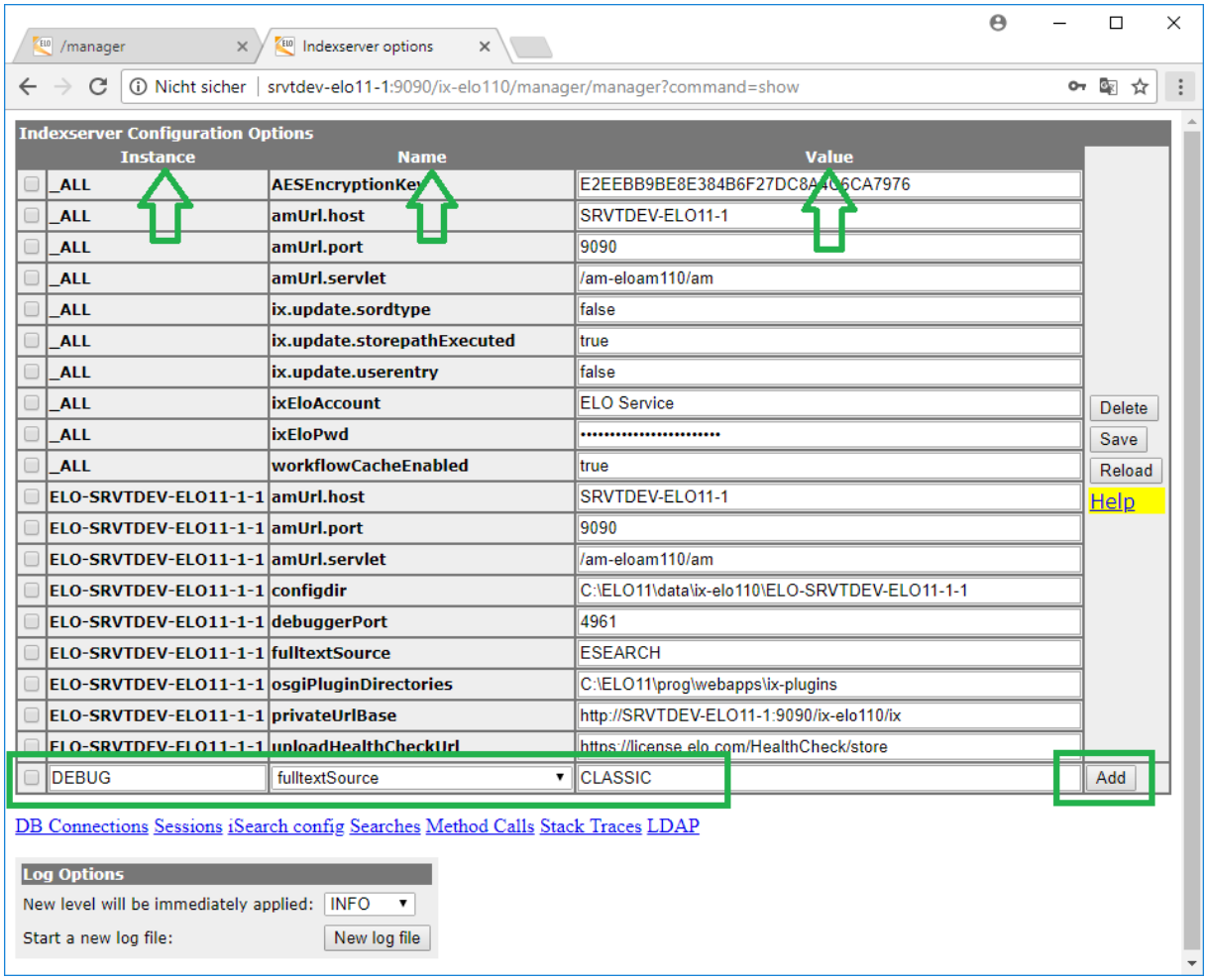


Abb. 2: Seite 'Indexserver options'

- Weil der ELOix während der Entwicklung häufig abgebrochen wird, kann die Steuerung der Volltextindizierung hängen bleiben. Deshalb ist es hilfreich, die Volltext-Indizierung komplett abzuschalten:

```
Instance=DEBUG
Name=fulltextSource
Value=CLASSIC
```

- Verzeichnis der Plug-ins bekanntgeben, z. B. C:\ELO11\prog\webapps\ix-plugins

```
Instance=DEBUG
Name= osgiPluginDirectories
Value=<instdir>/prog/webapps/ix-plugins
```

2.2.2 Konfigurationsdateien duplizieren

Duplizieren Sie das Verzeichnis `ELODir/config/ix-archive1`, z. B. `ELODir/config/ix-archive1-debug`

- Ändern Sie den Instanznamen des ELOix in der Datei `ELODir/config/ix-archive1-debug/config.xml`
  zu `<entry key="ixid">DEBUG</entry>`
- Ändern Sie den Log-Dateinamen in der Logger-Konfiguration : `ELODir/config/ix-archive1-debug/log4j.properties`
  zu einem Dateinamen in einem Verzeichnis mit Vollzugriff für Ihr Benutzerkonto. Unter Windows ist das Verzeichnis C:\Programme… ungeeignet. `log4j.appender.FI.File=…logdateiname…`

2.3 Workspace einrichten

Ein Eclipse-Workspace ist eine Sammlung von Java-Projekten zu einem Entwicklungsthema. Hier wird ein Workspace eingerichtet, der neben dem zu entwickelnden Plug-in, die EloixClient-Bibliothek als OSGi-Bundle und den Indexserver als Web-Anwendungsprojekt enthalten wird.

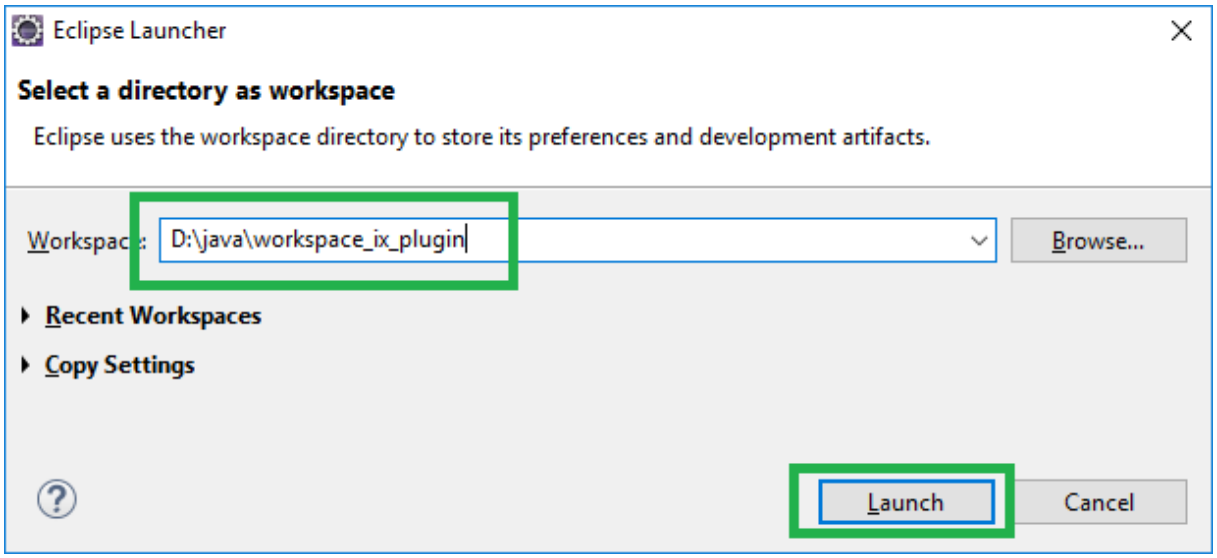


Abb. 3: Workspace-Verzeichnis angeben

- Eclipse starten und ein neues Workspace-Verzeichnis angeben, `WorkspaceDir`.

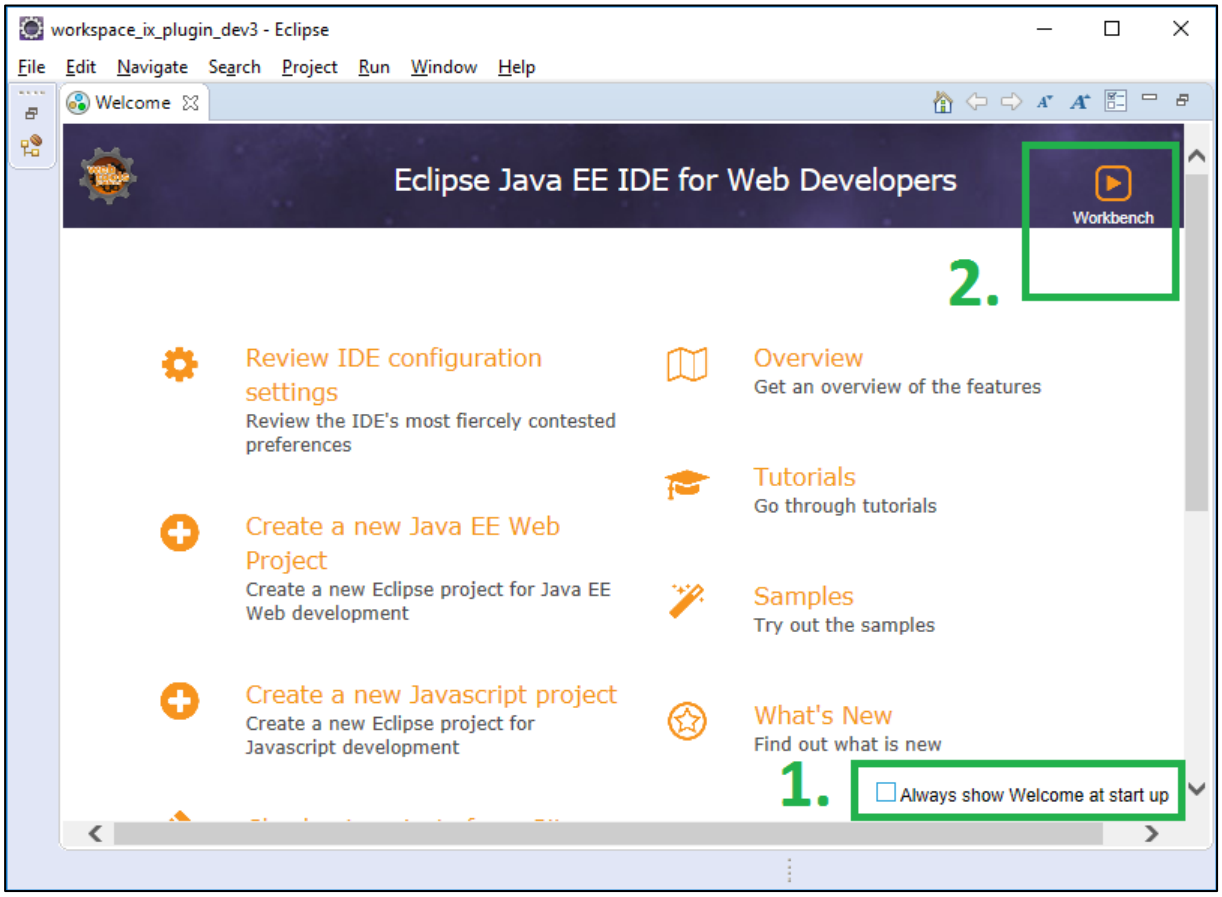


Abb. 4: Option 'Always show Welcome at start up'

2.3.1 JDK 8 einrichten

Die Plug-in-Entwicklung erfolgt unter der Java-Version 8. Wenn Eclipse nicht bereits unter Java 8 gestartet ist, richten Sie das Java ein unter:

- *Eclipse* > *Menü* > *Window* > *Preferences* wählen und den markierten Dialogelementen in den Screenshots folgen:

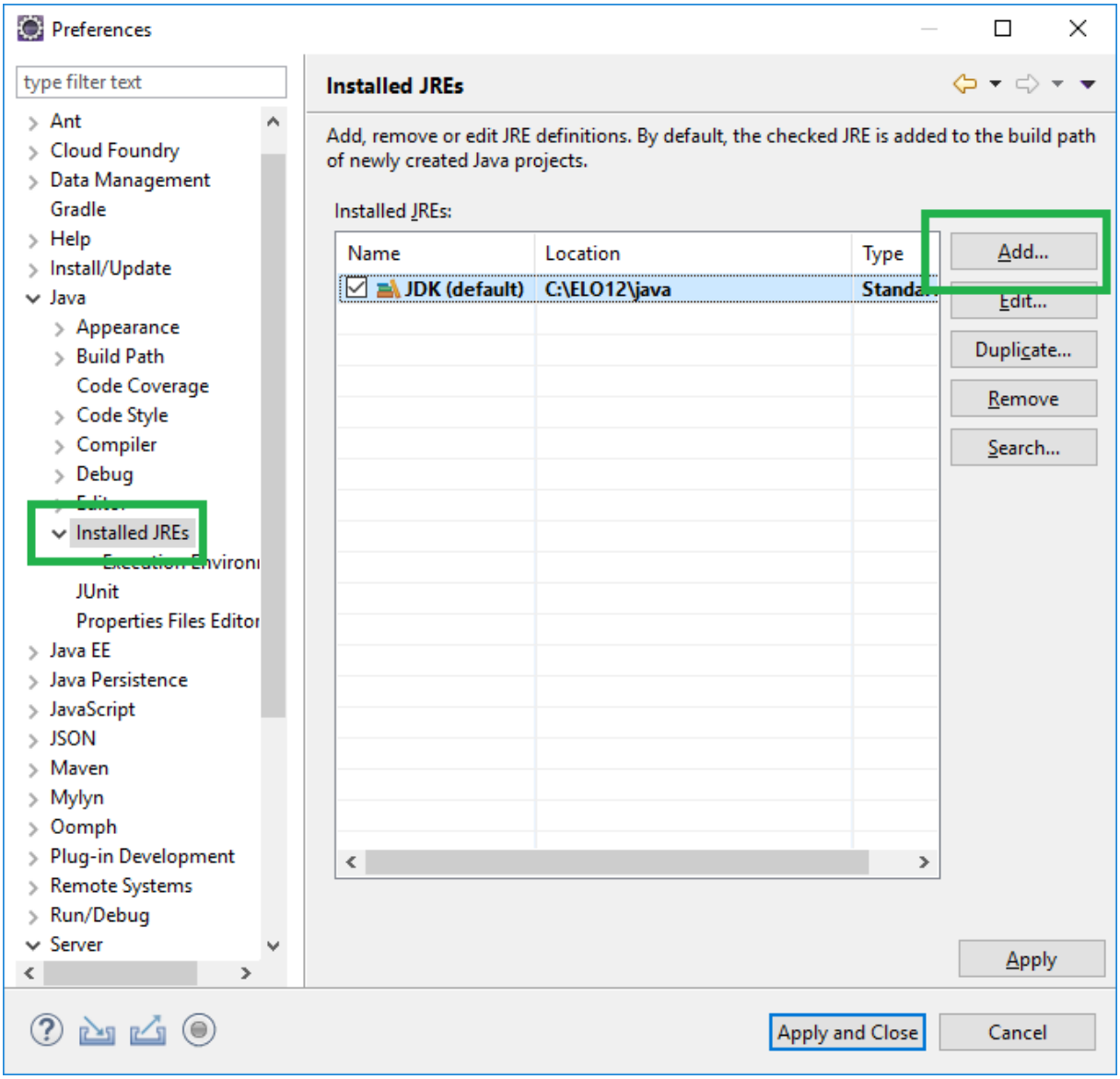


Abb. 5: Menüpunkt 'Installed JREs'

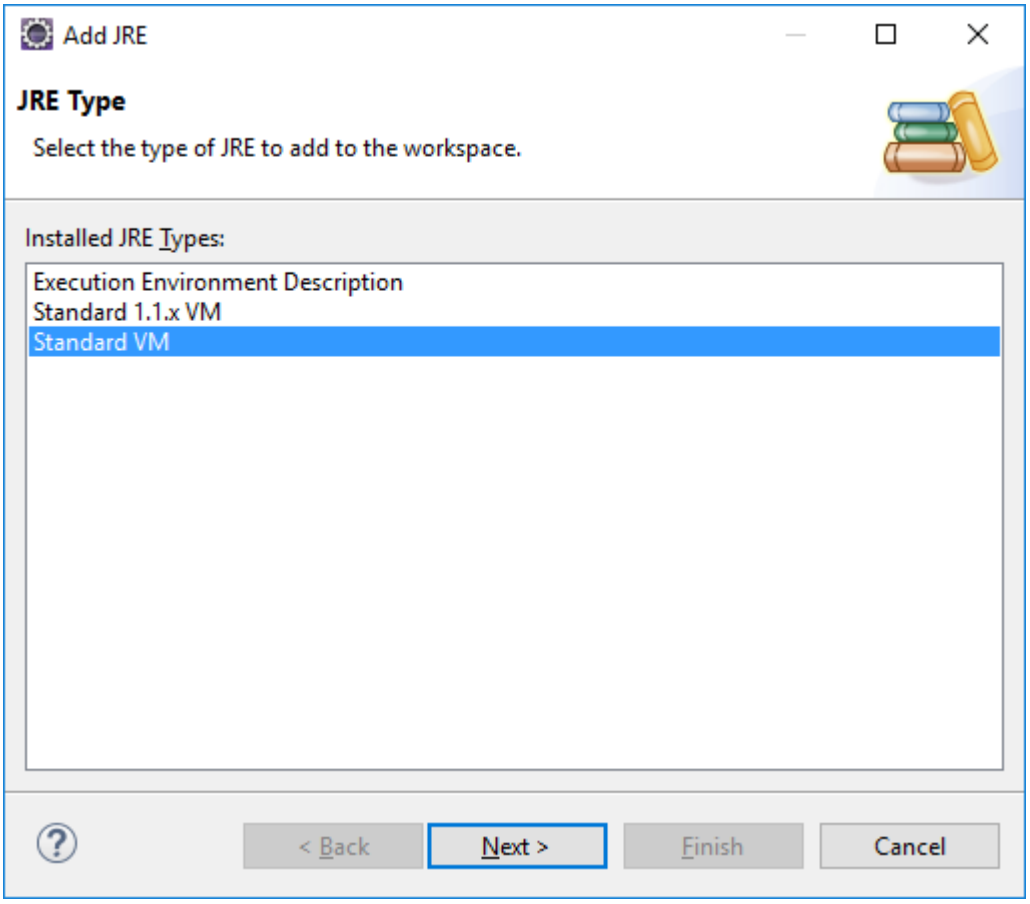


Abb. 6: Auswahl des JRE-Typs

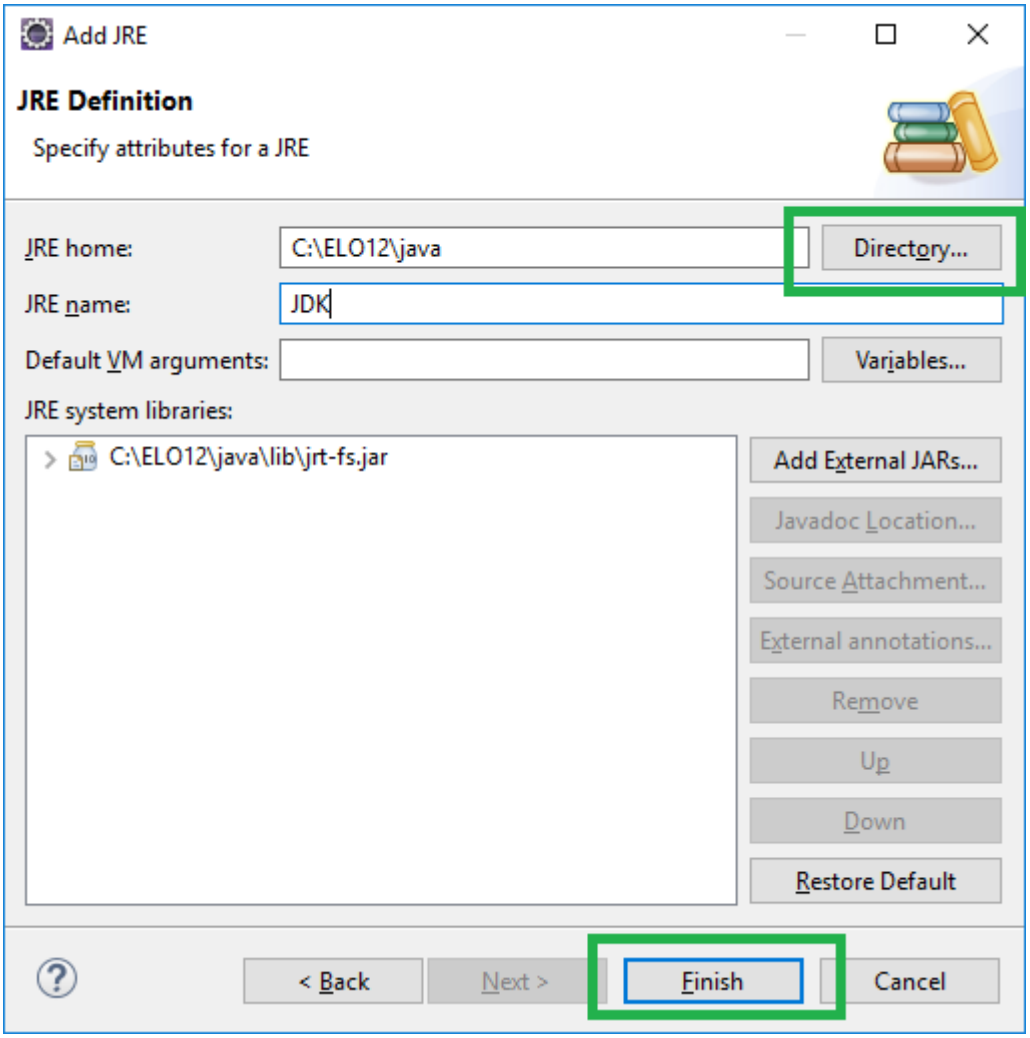


Abb. 7: Auswahl des Verzeichnisses und Abschluss

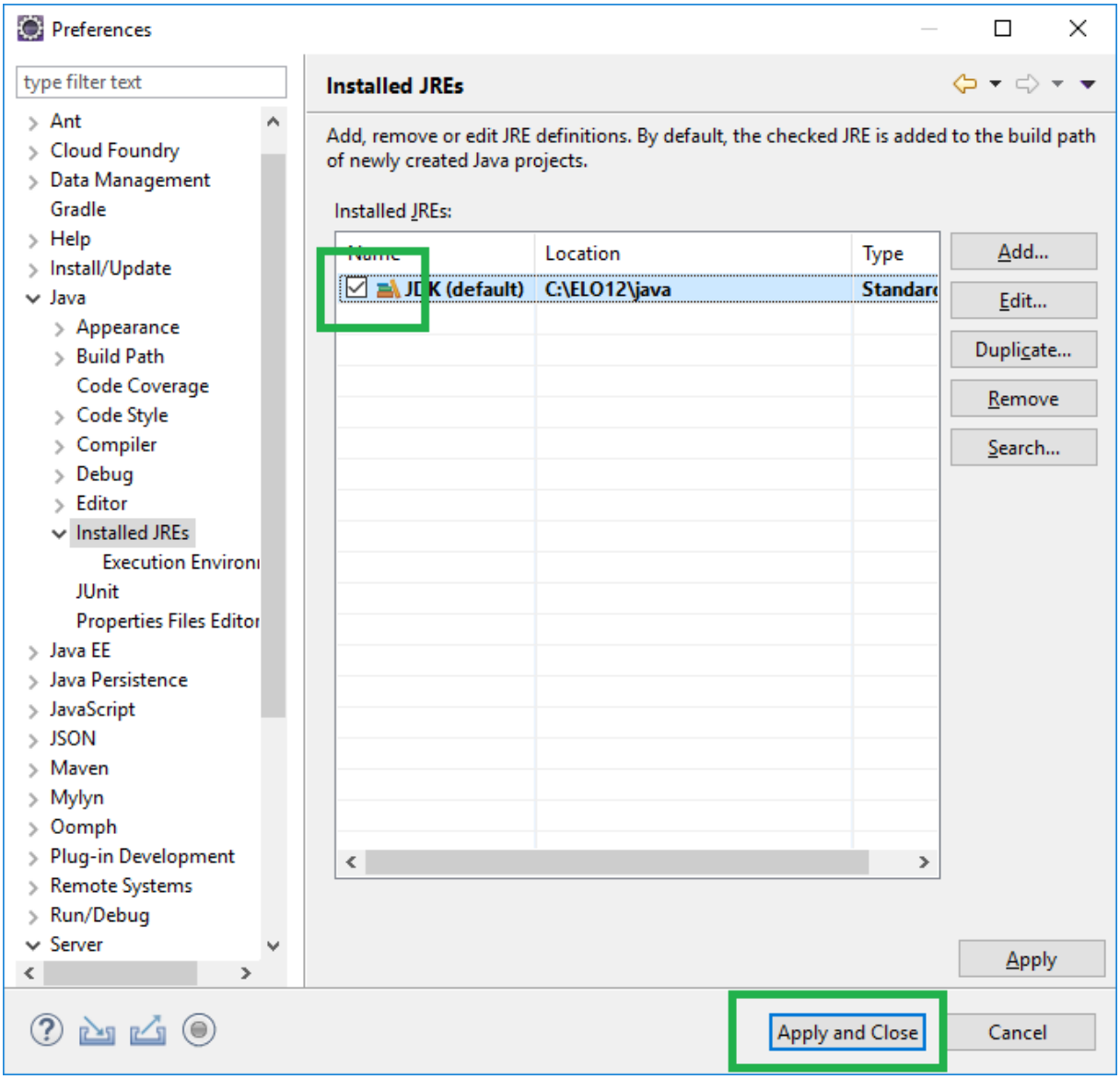


Abb. 8: Aktivierung des JREs

2.3.2 Server Runtime einrichten

Der im Abschnitt *2.1.4 Tomcat Server Runtime einrichten* installierte Apache Tomcat wird hier als Server-Runtime aufgenommen.

- *Eclipse* > *Menü* > *Window* > *Preferences* wählen und den markierten Dialogelementen in den Screenshots folgen:

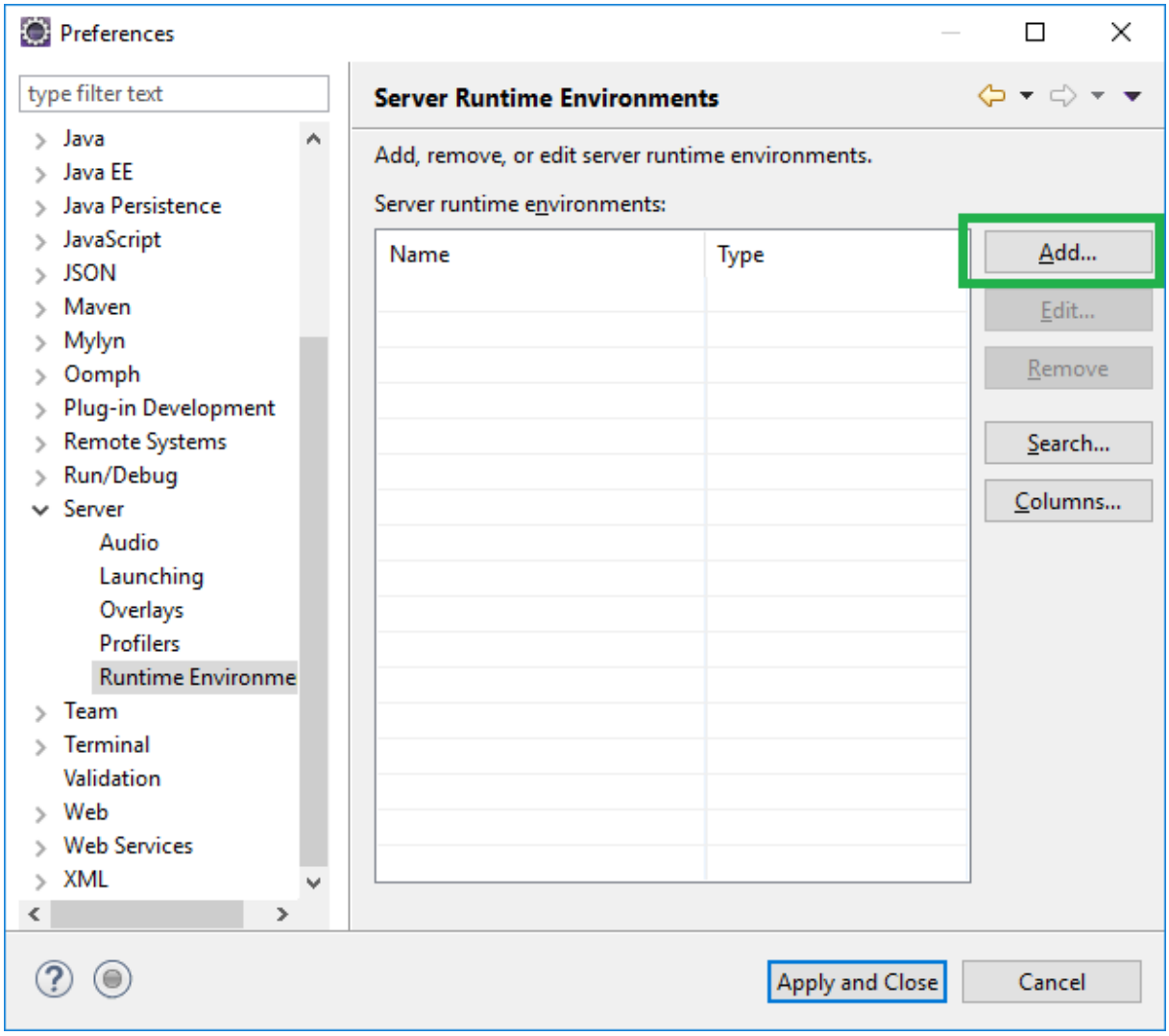


Abb. 9: Runtime Environment hinzufügen

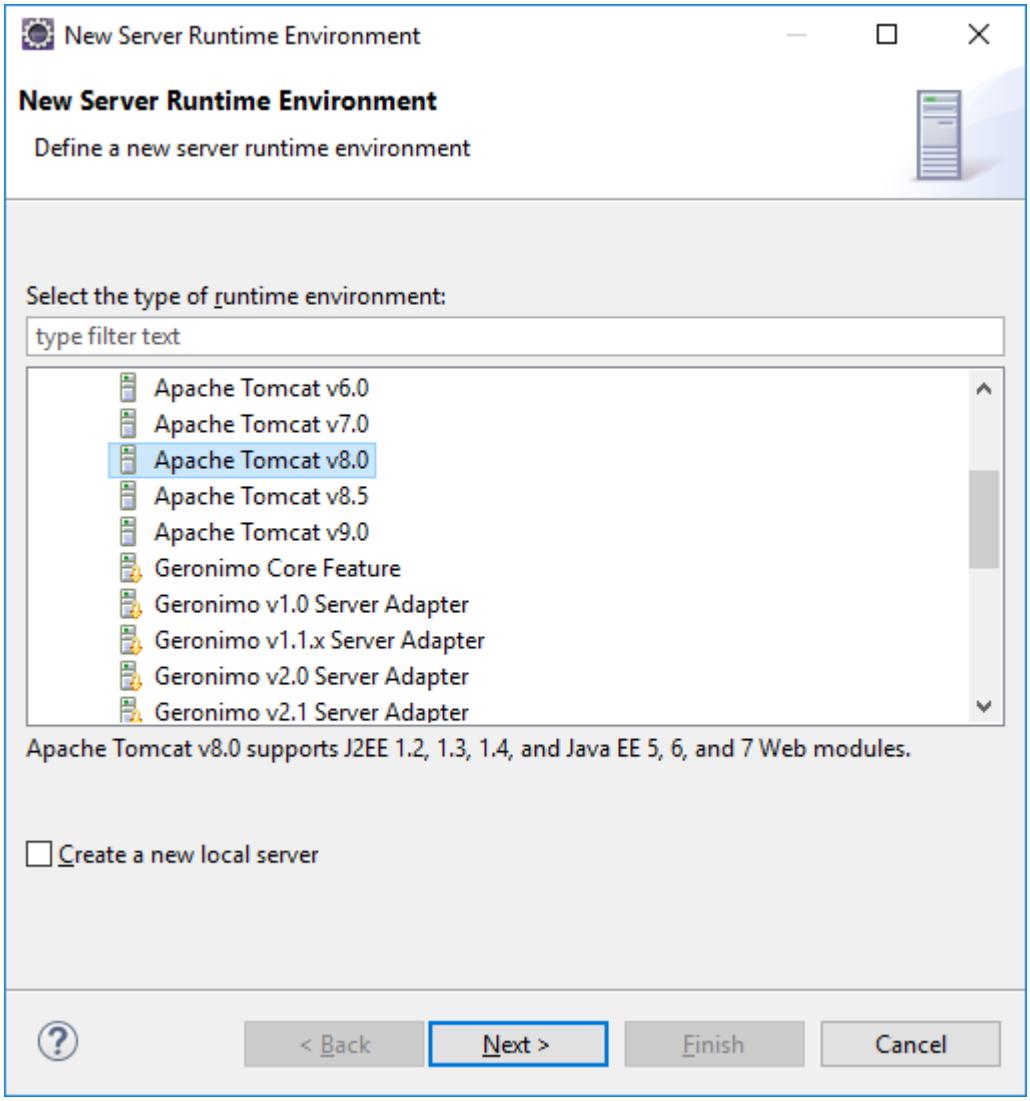


Abb. 10: Auswahl der passenden Apache-Tomcat-Version

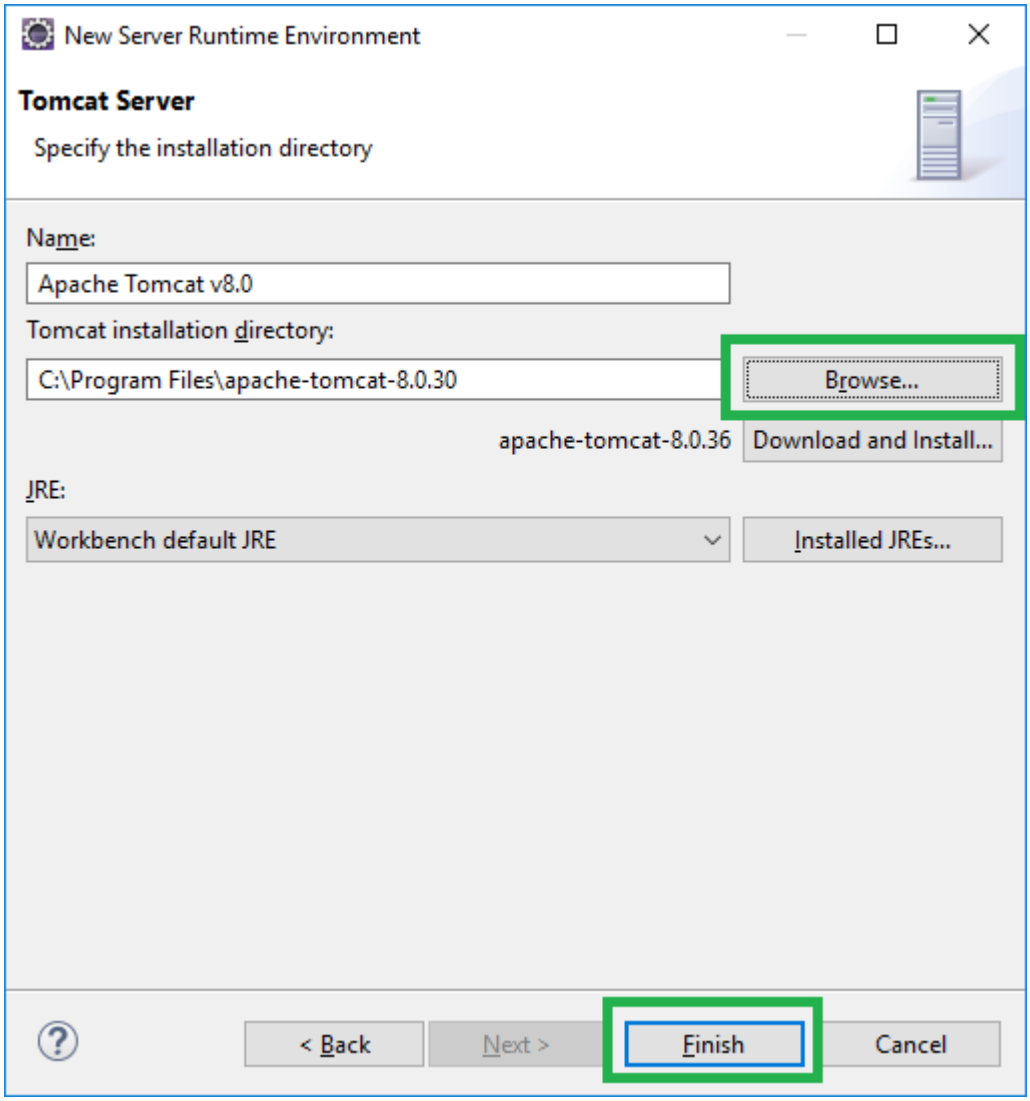


Abb. 11: Auswahl des Verzeichnisses und Abschluss

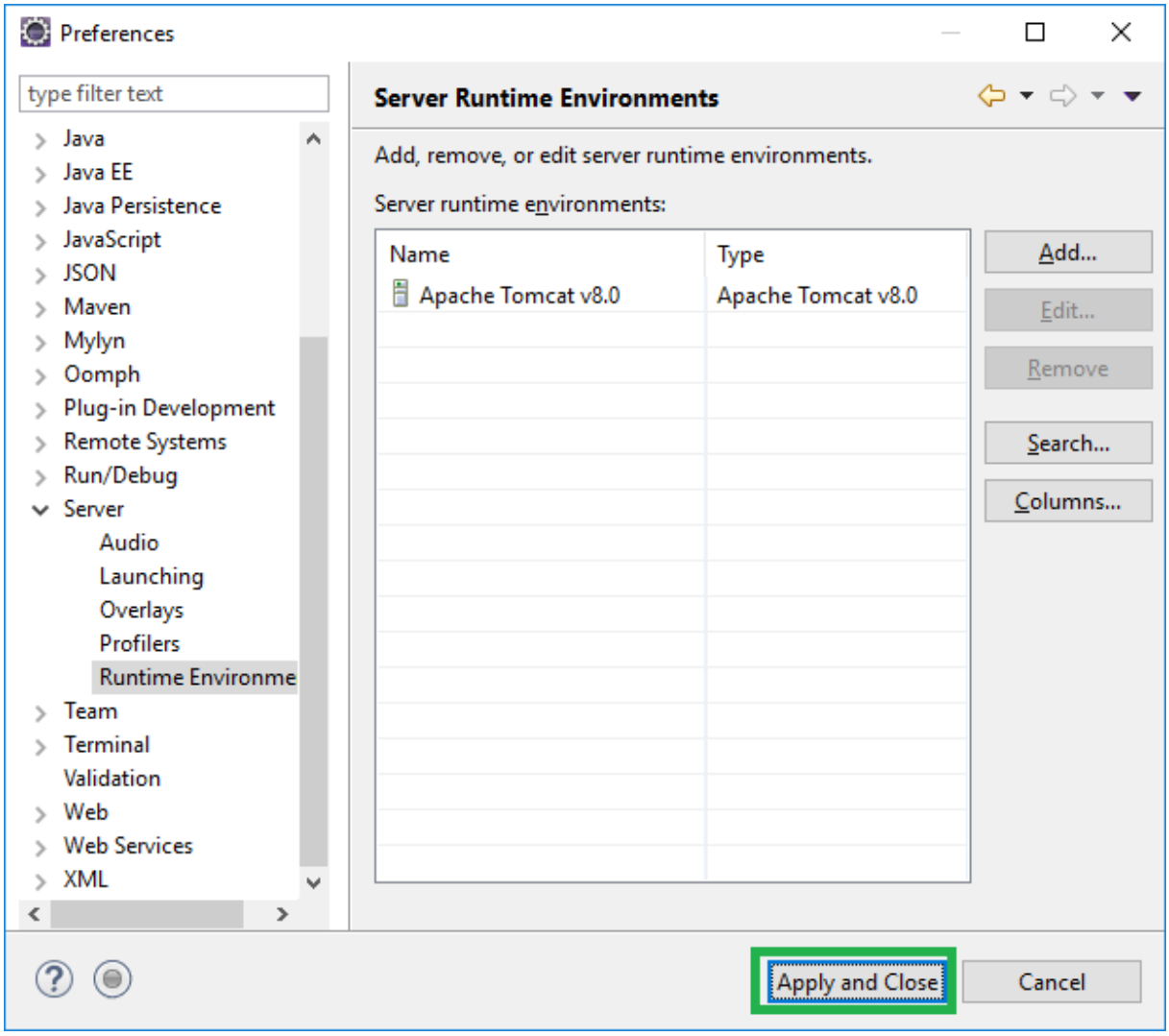


Abb. 12: Schaltfläche 'Apply and Close'

2.3.3 Server-Runtime für den Workspace auswählen

Um die zuvor angelegte Server-Runtime zum Starten und Deployen im Workspace auszuwählen, folgen Sie den nächsten Schritten.

- *Eclipse > Menü > File > New… > Other*

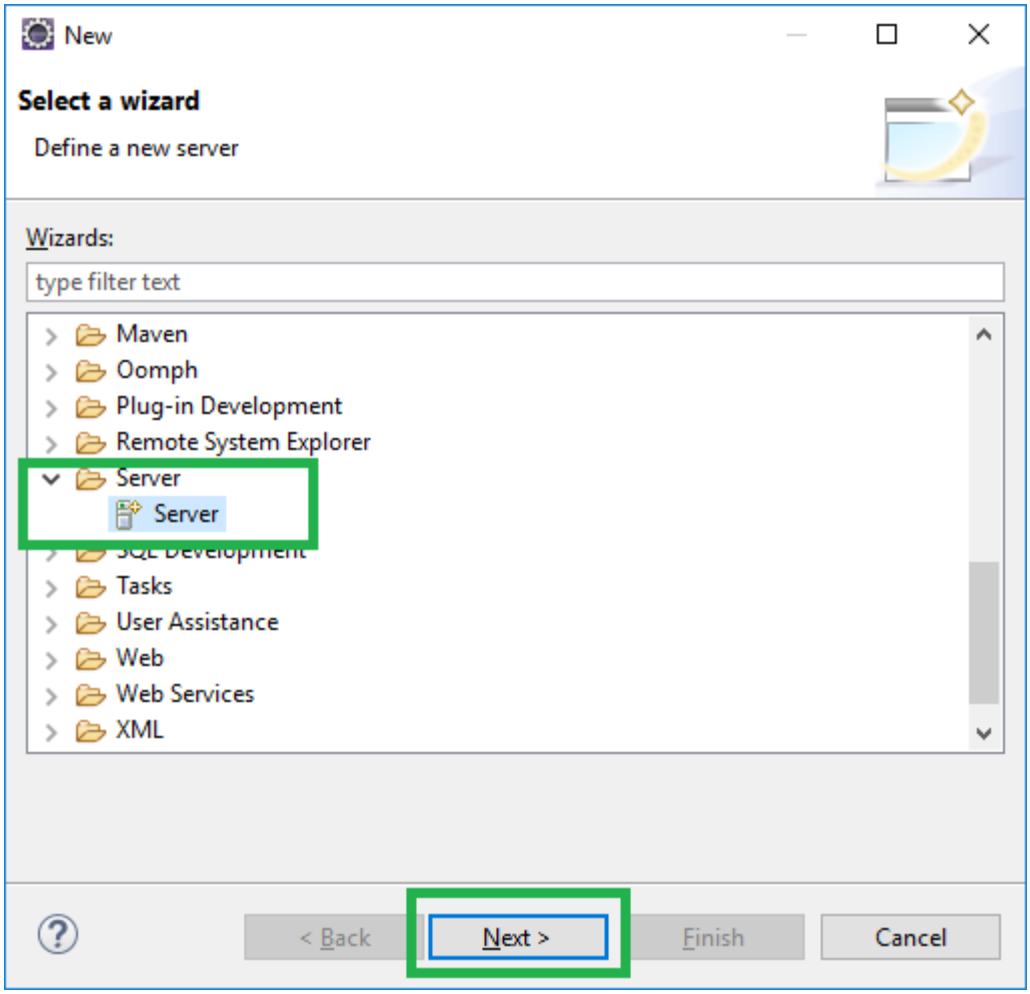


Abb. 13: Auswahl des Wizards

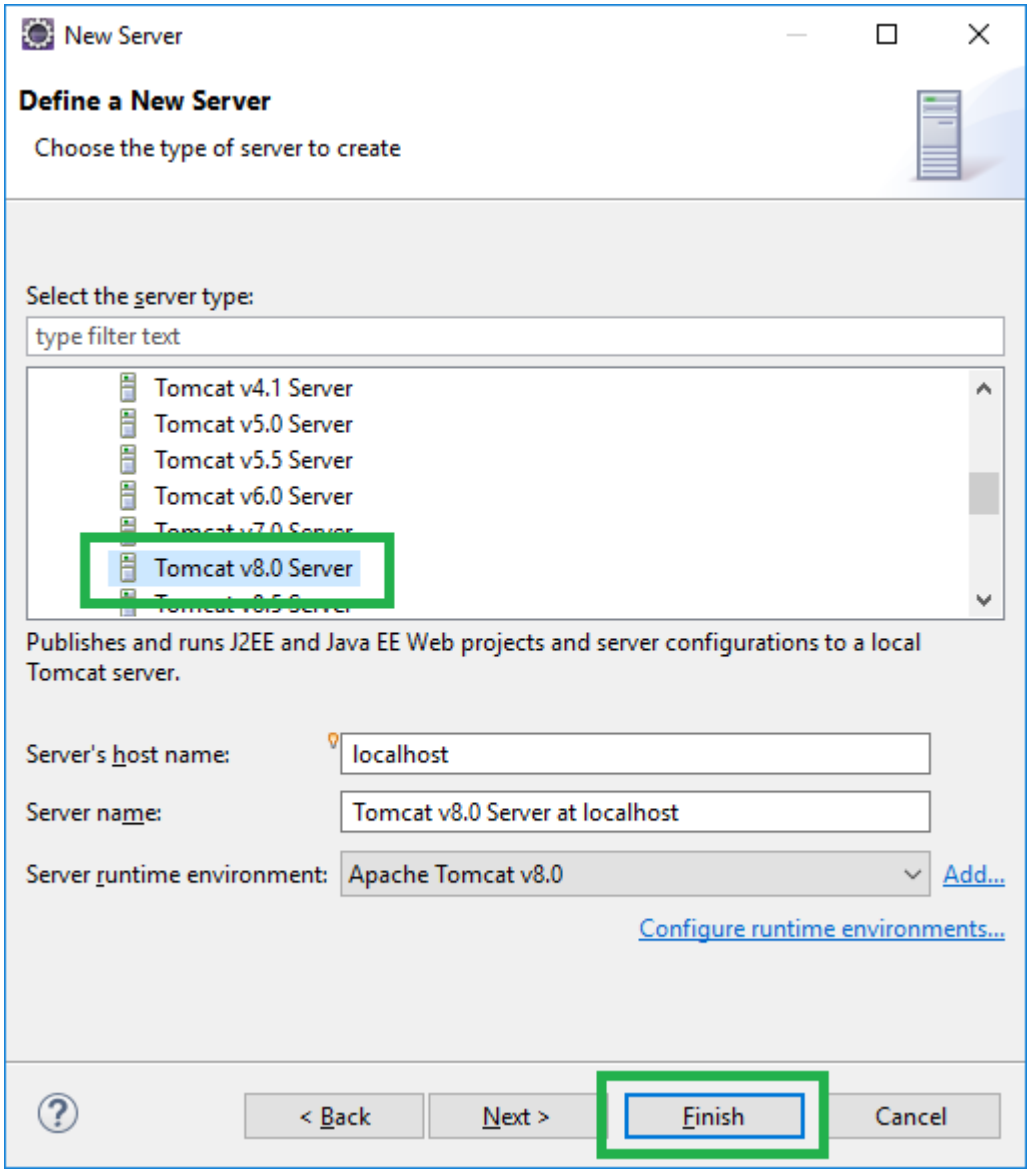


Abb. 14: Auswahl der passenden Version des Tomcat-Servers

Der Project-Explorer von Eclipse zeigt anschließend einen Servers-Knoten, unter dem ein Tomcat-Knoten liegt. Dieser Knoten beinhaltet die Konfiguration des für das Debugging ausgewählten Apache-Tomcat-Servers. Nachfolgend wird die Konfiguration angepasst.

2.3.3.1 TCP-Port für HTTP-Protokoll

Um die Standard-Ports des Apache Tomcat zu ändern, bearbeiten Sie die Datei *server.xml*.

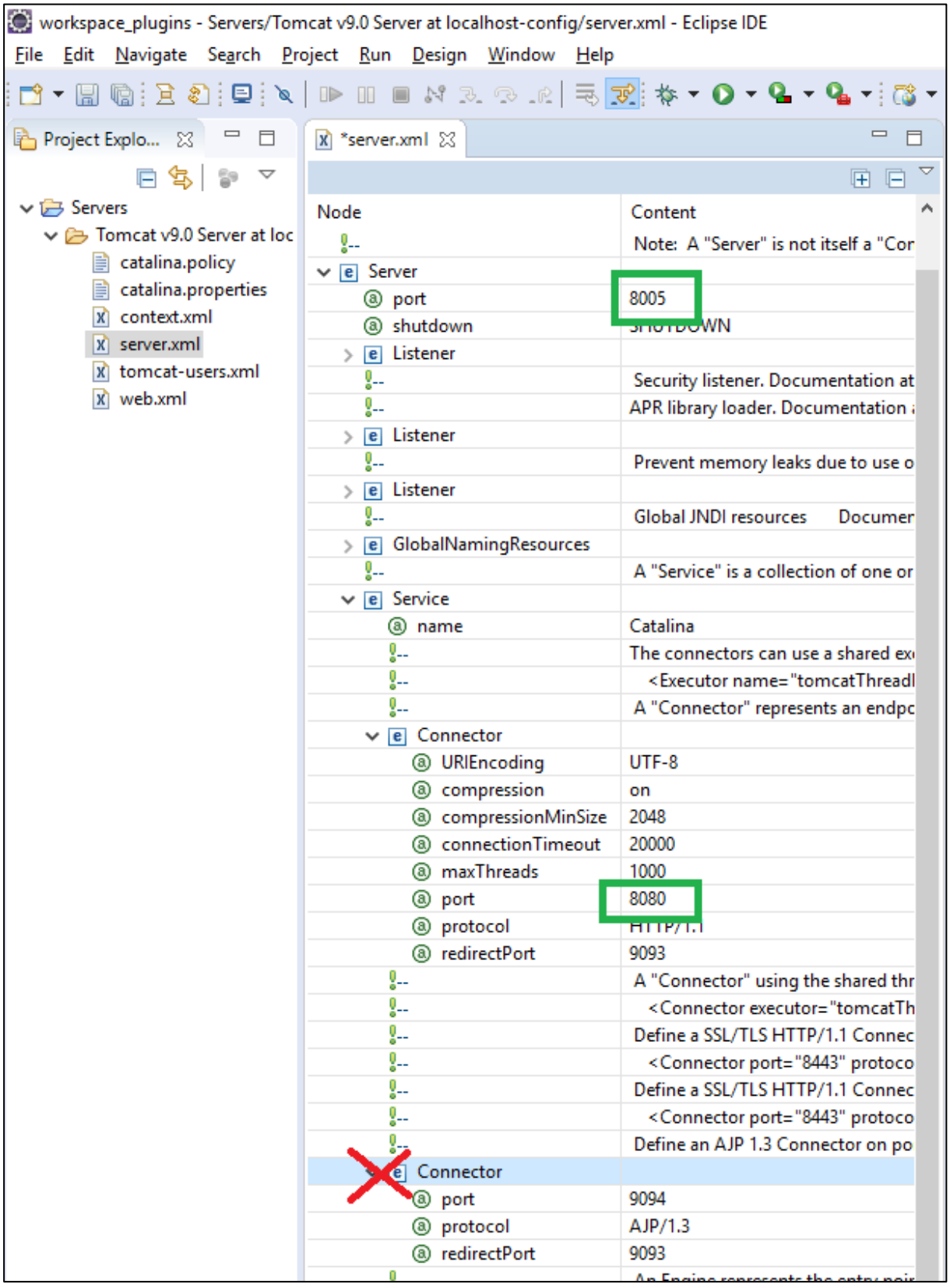


Abb. 15: Ports in der Datei 'server.xml'

2.3.3.2 Indexserver-Konfigurationsverzeichnis eintragen

- Indexserver-Konfigurationsverzeichnis aus 2.2 in der *context.xml* als Unterelement von `<Context>` eintragen:

```
<Environment name="webappconfigdir" override="false"
type="java.lang.String" value="ELODir/config/ix-archive1-debug" />
```

- `ELODir` und `archive1` durch die korrekten Werte ersetzten.

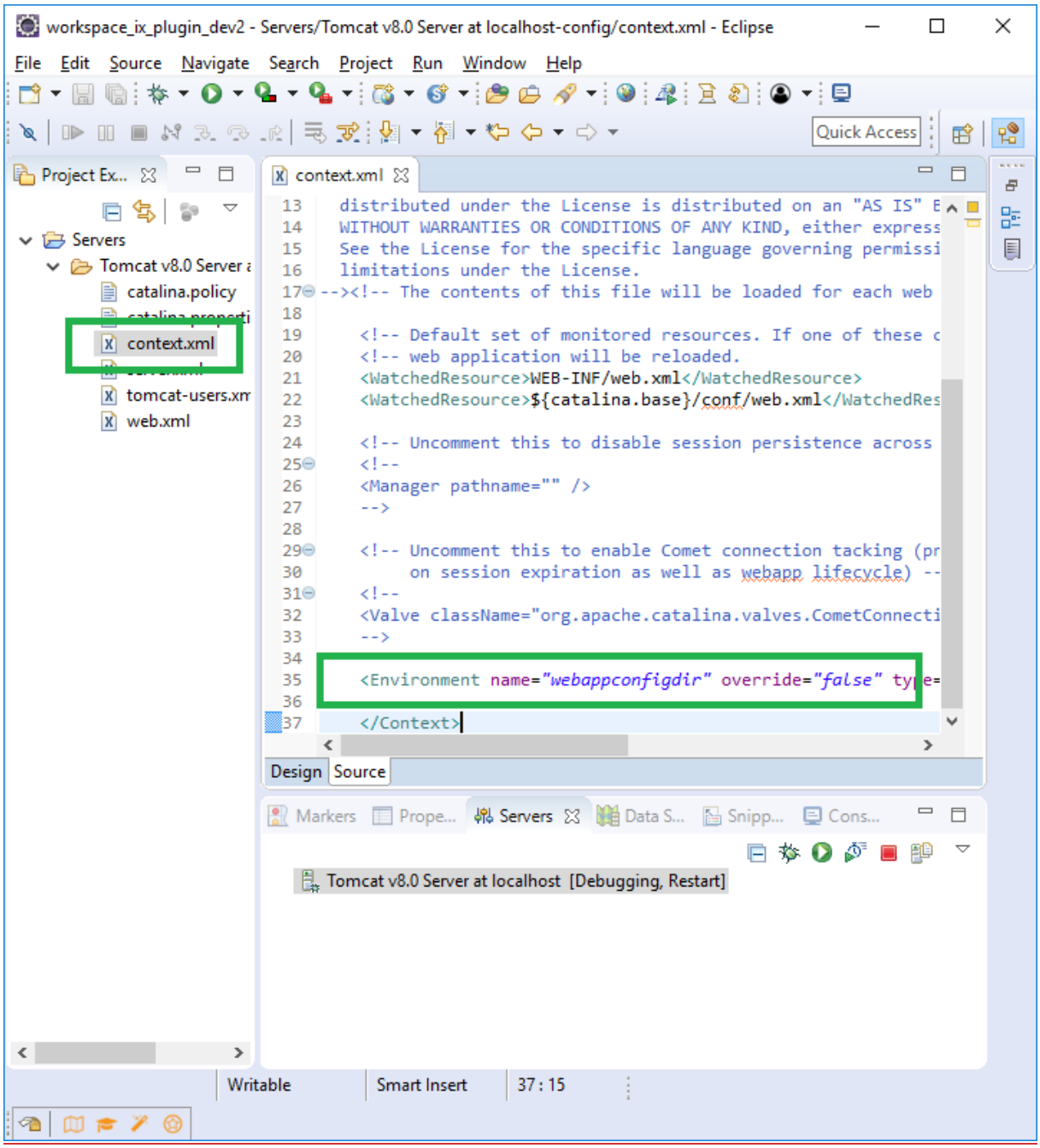


Abb. 16: Bearbeitung der Datei 'context.xml'

2.3.3.3 Tomcat-Anwenderkonto einrichten

Damit man sich später auf der Optionenseite des ELOix anmelden kann, muss ein Apache Tomcat-Konto als Unterelement von `<tomcat-users>` ergänzt werden:

```
<user username="admin" password="elo" roles="admin-gui,manager-gui"></user>
<role rolename="admin-gui"/>
<role rolename="manager-gui"/>
```

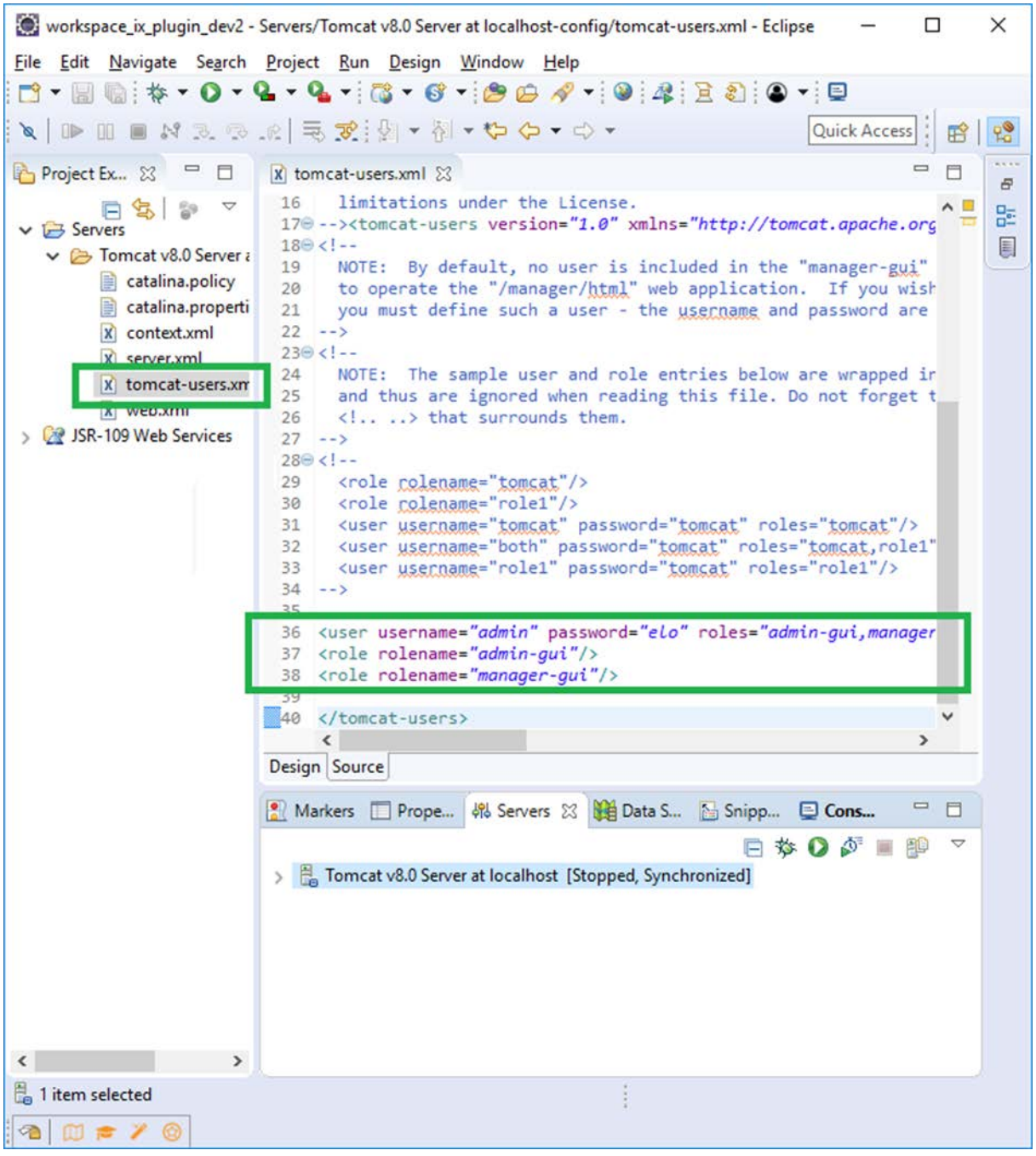


Abb. 17: Hinzufügen eines Tomcat-Kontos

2.3.3.4 Tomcat starten

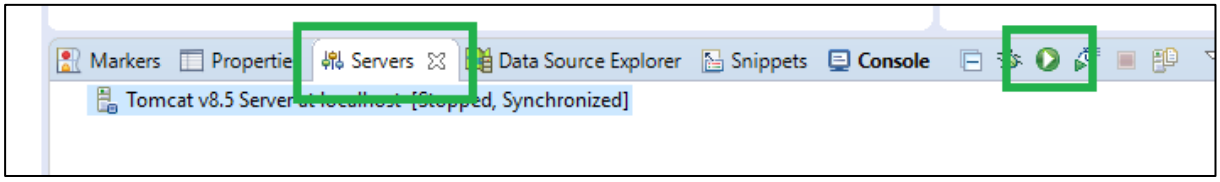
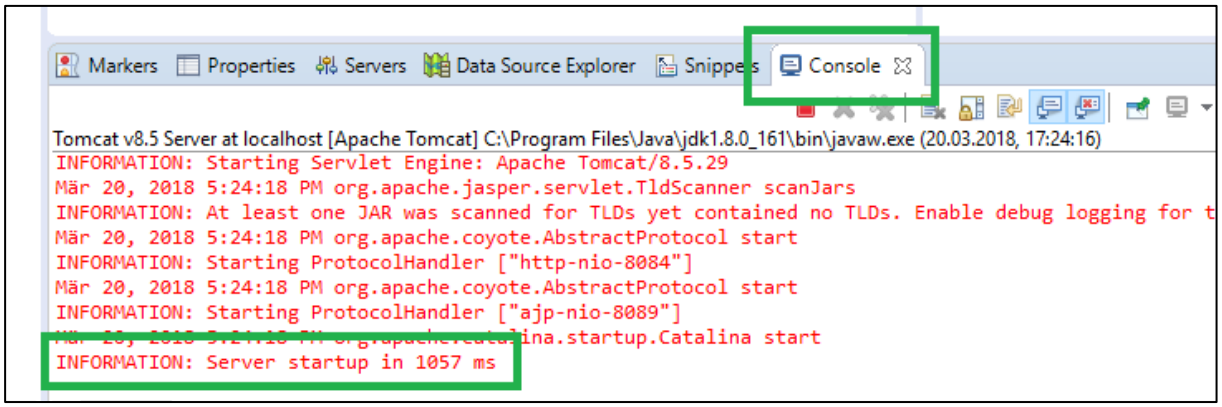


Abb. 18: Starten des Apache Tomcat

- Starten Sie zum Test den Apache Tomcat in der Ansicht *Servers*:



- Wechseln Sie in die Ansicht *Console*. Die letzte Zeile sollte „…Server startup…“ lauten.
- Stoppen Sie den Apache Tomcat in der Ansicht *Servers*.

2.3.4 ELO Indexserver in Workspace aufnehmen

Im Folgenden wird der ELO Indexserver als Web-Anwendungsprojekt in den Workspace importiert.

- Indexserver.zip aus dem Downloadbereich herunterladen in ein temporäres Verzeichnis entpacken. Oder im Folgenden die *ix.war* aus der ELO Server-Installation importieren.
    - Verzeichnis: *ELOdir/prog/webapps/ix.war*
- *Eclipse > Menü > File > Import…*

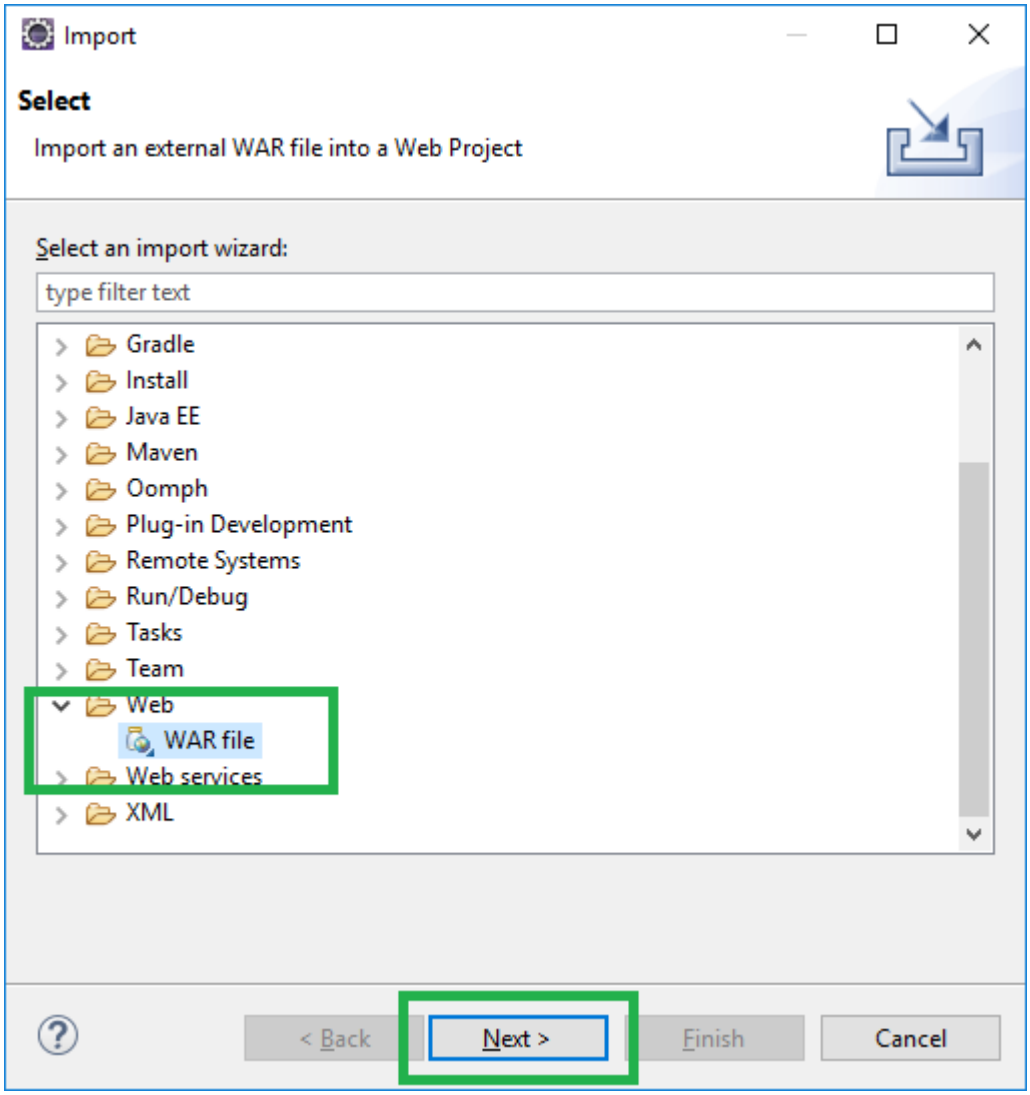


Abb. 19: Auswahl des Assistenten für WAR-Dateien

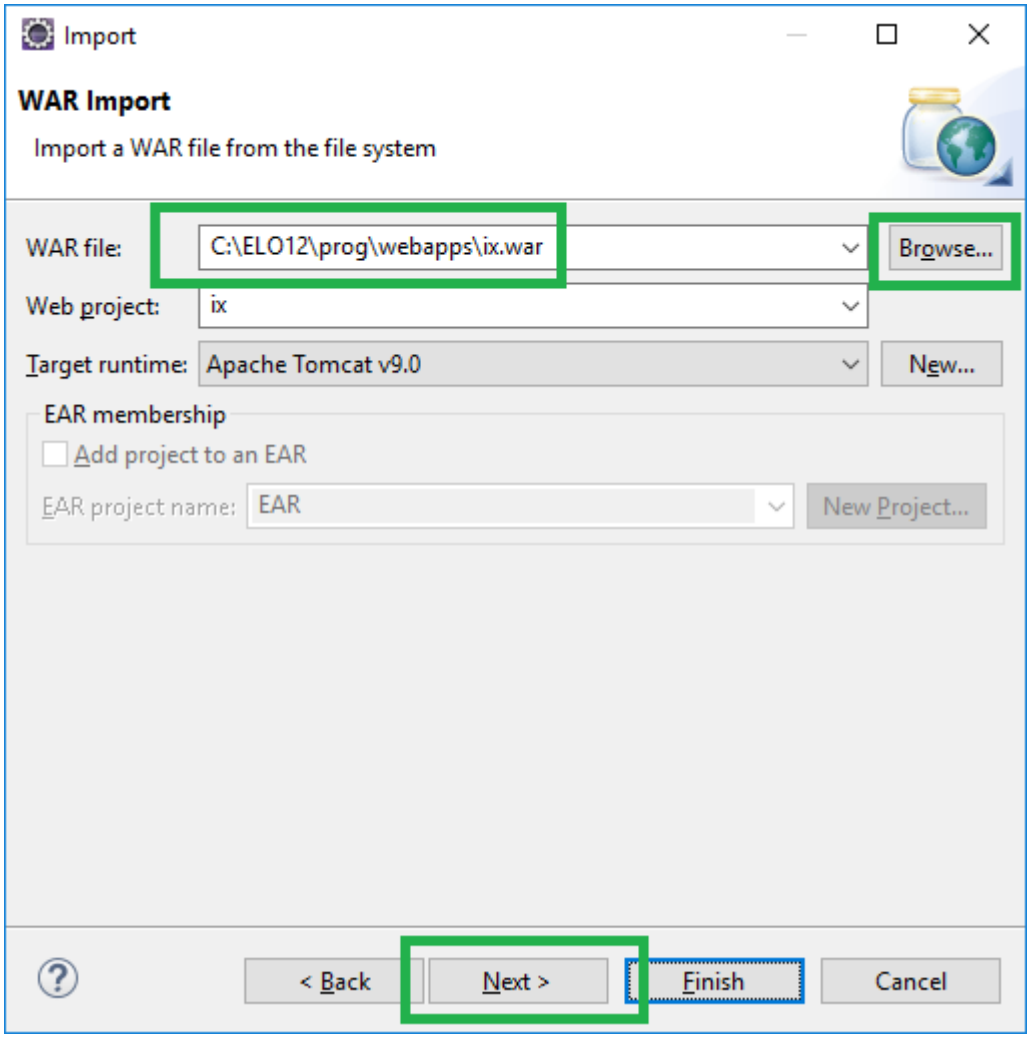


Abb. 20: Auswahl der WAR-Datei

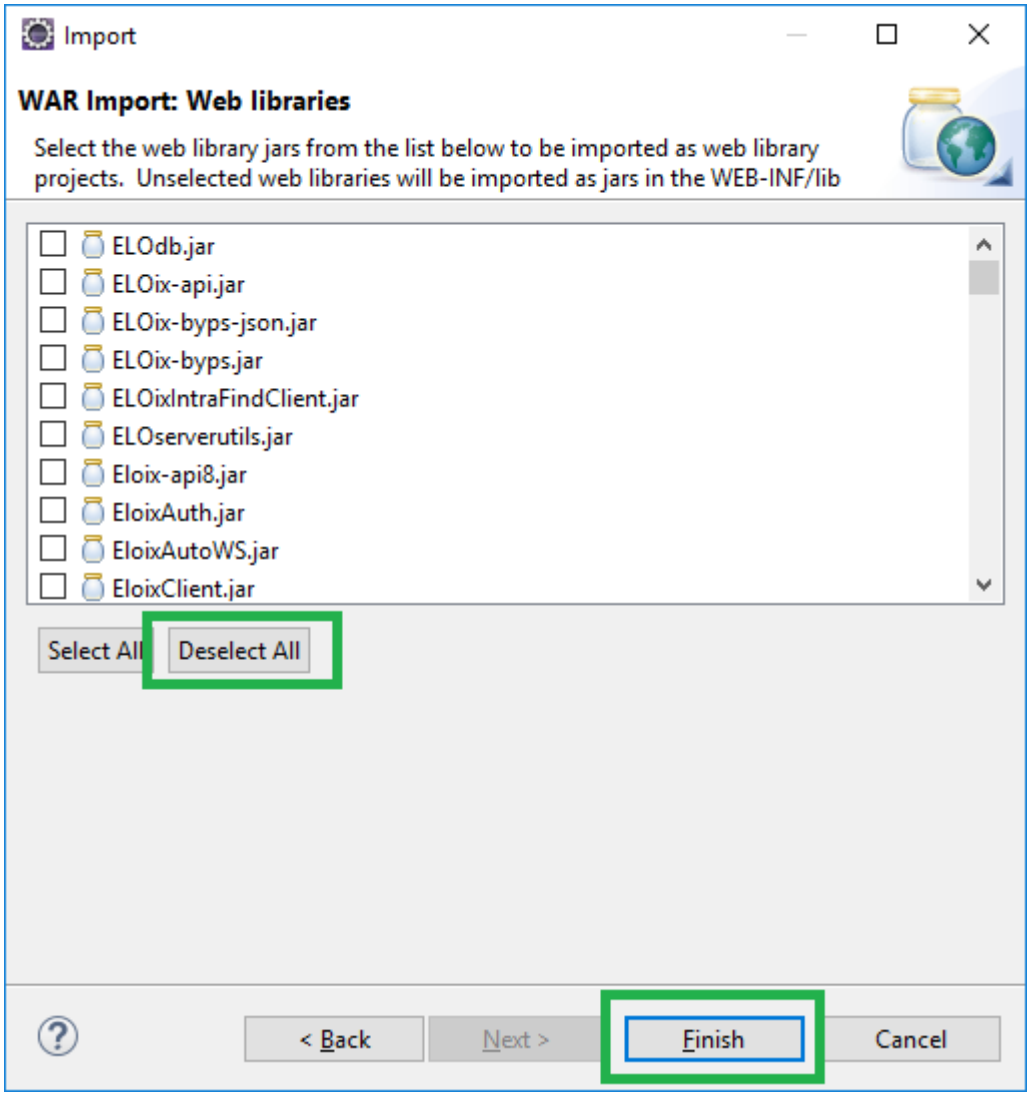


Abb. 21: Schaltfläche 'Deselect All'

2.3.4.1 Ergebnis:

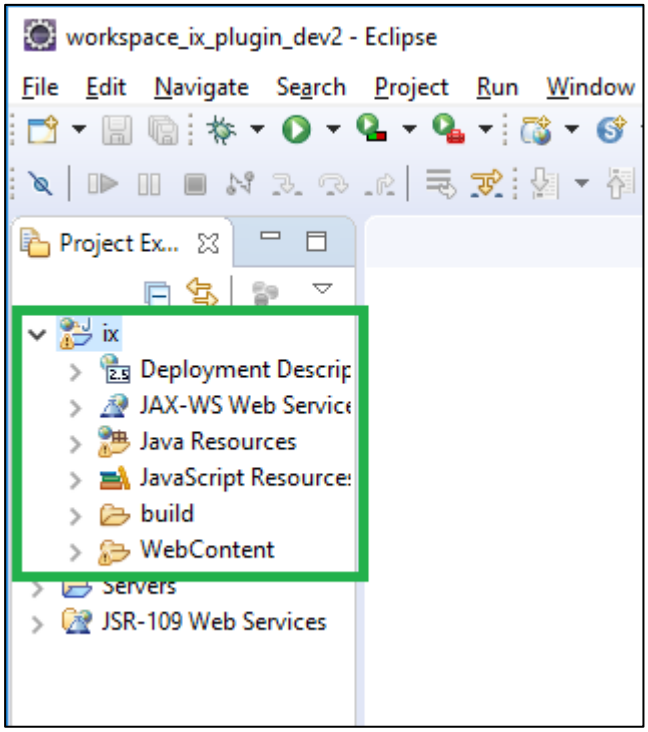


Abb. 22: Ergebnis in Eclipse

Information: Um auf eine neue Version des Indexserver zu wechseln, ist das ix.war-Projekt zu löschen (inklusive Dateien im Workspace) und neu anzulegen.

2.3.5 ELO Indexserver deployen und starten

Die nächsten Schritte spielen den ELO Indexserver auf den Apache Tomcat auf und prüfen, ob der ELO Indexserver korrekt startet.

- *Eclipse > Ansicht Servers > Kontextmenü >…*

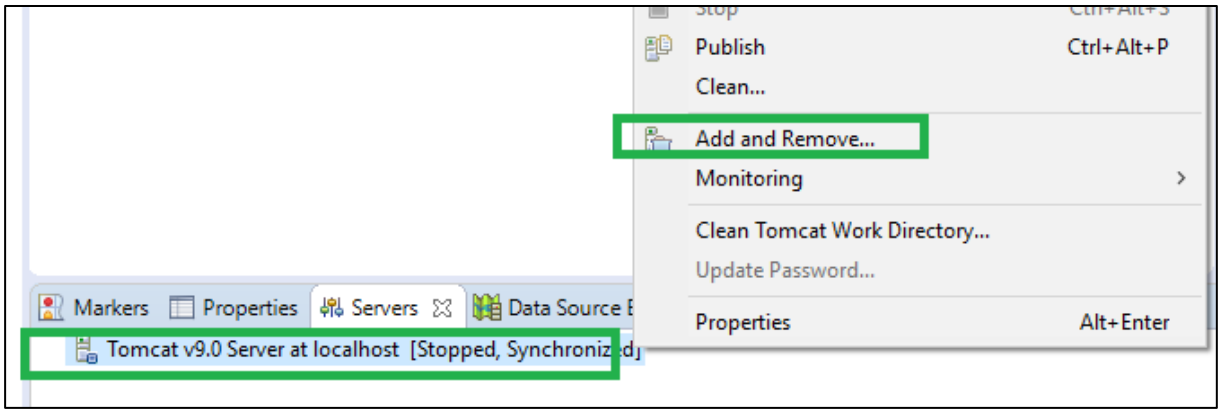


Abb. 23: Kontextmenü, Menüpunkt 'Add and Remove...'

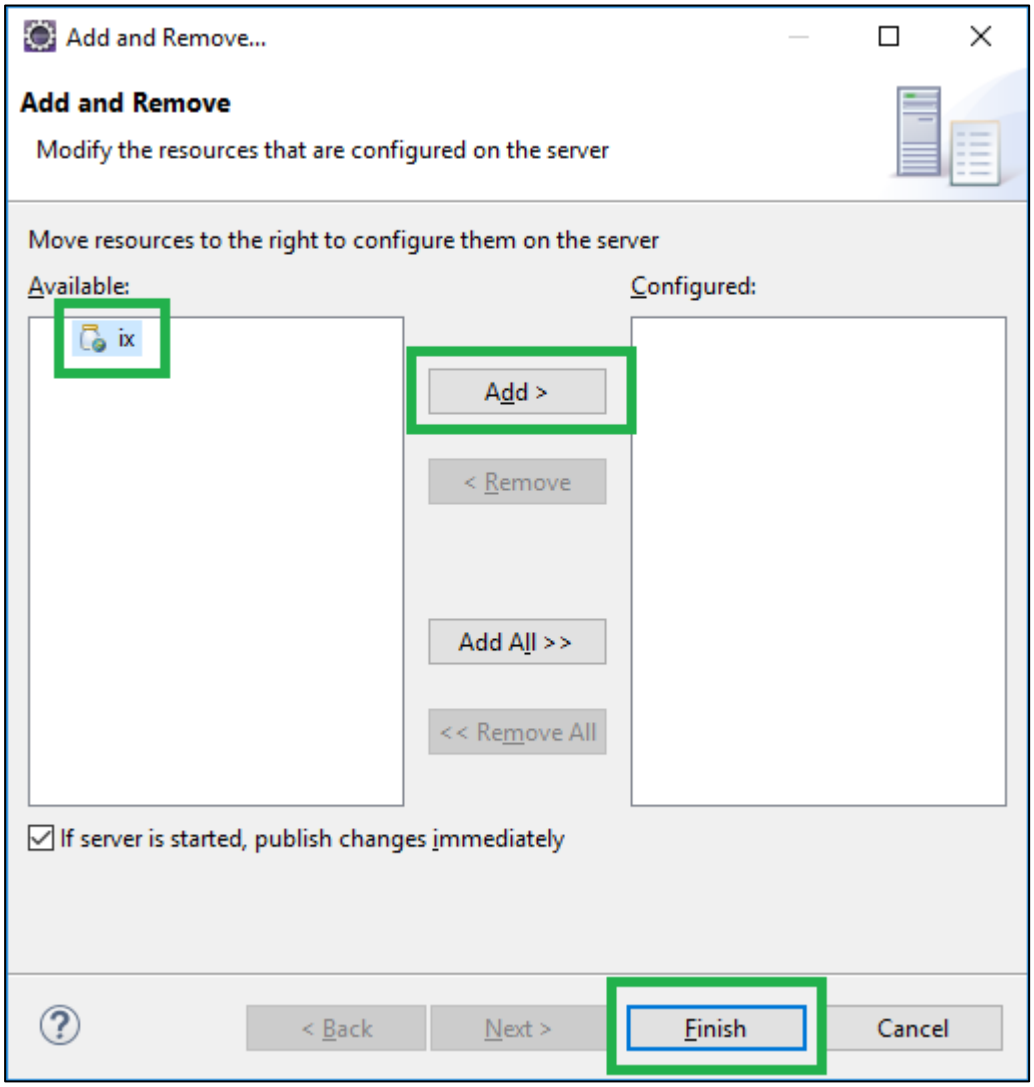


Abb. 24: ELOix zum Apache Tomcat hinzufügen

- Prüfen Sie, ob der Server startet.

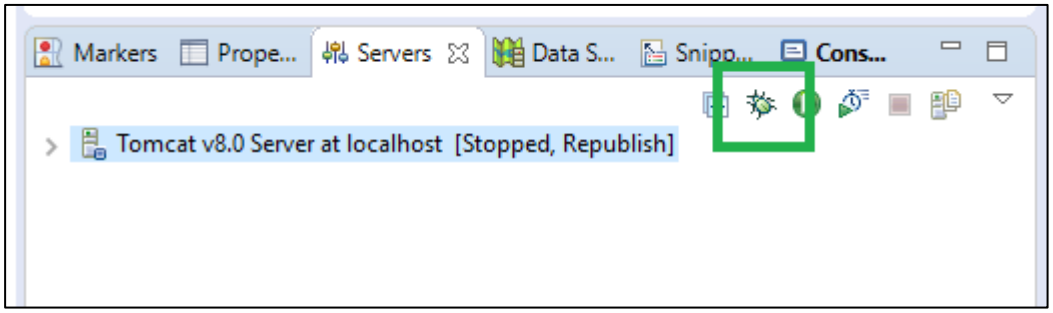


Abb. 25: Ansicht 'Servers', Käfersymbol

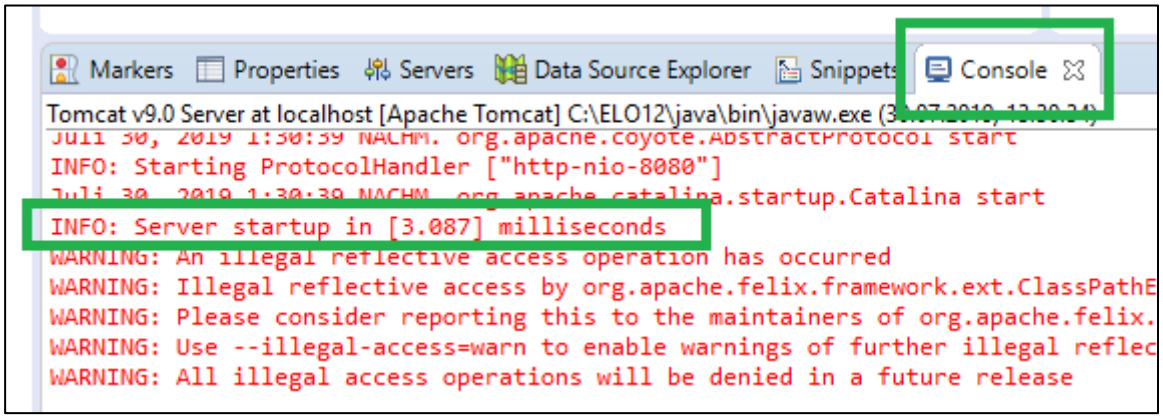


Abb. 26: Ansicht 'Console', Log-Datei

- Prüfen Sie die Log-Datei des ELOix auf Fehler.
- Prüfen Sie die Statusseite des ELOix: http://localhost:8080/ix/
- Stoppen Sie den Apache Tomcat in der Ansicht *Servers*.

2.3.6 Projektverzeichnisse als Plug-in-Verzeichnisse festlegen

Damit der ELO Indexserver die Plug-ins findet, die Sie im Workspace werden, wird dem Apache Tomcat eine Startoption mitgegeben, die alle Projektverzeichnisse des Workspace enthält.

- Wählen Sie im Eclipse Menü: *Run* > *Debug Configurations…*.
- Selektieren Sie links *Tomcat…* und rechts den Tab *Arguments*.
- Im Eingabefeld *VM arguments* ergänzen Sie am Ende der Argumentliste nach einem Leerzeichen (kein Zeilenvorschub) diesen Text: `-DosgiPluginDirectories=`
- Anschließend klicken Sie unter der Eingabebox auf die Schaltfläche *Variables…*
- Wählen Sie die Variable *workspace_project_locations*.
- Wenn diese nicht in der Liste enthalten ist, dann wählen Sie *workspace_loc* und erweitern den Variablenausdruck zu: `${workspace_loc:/}`

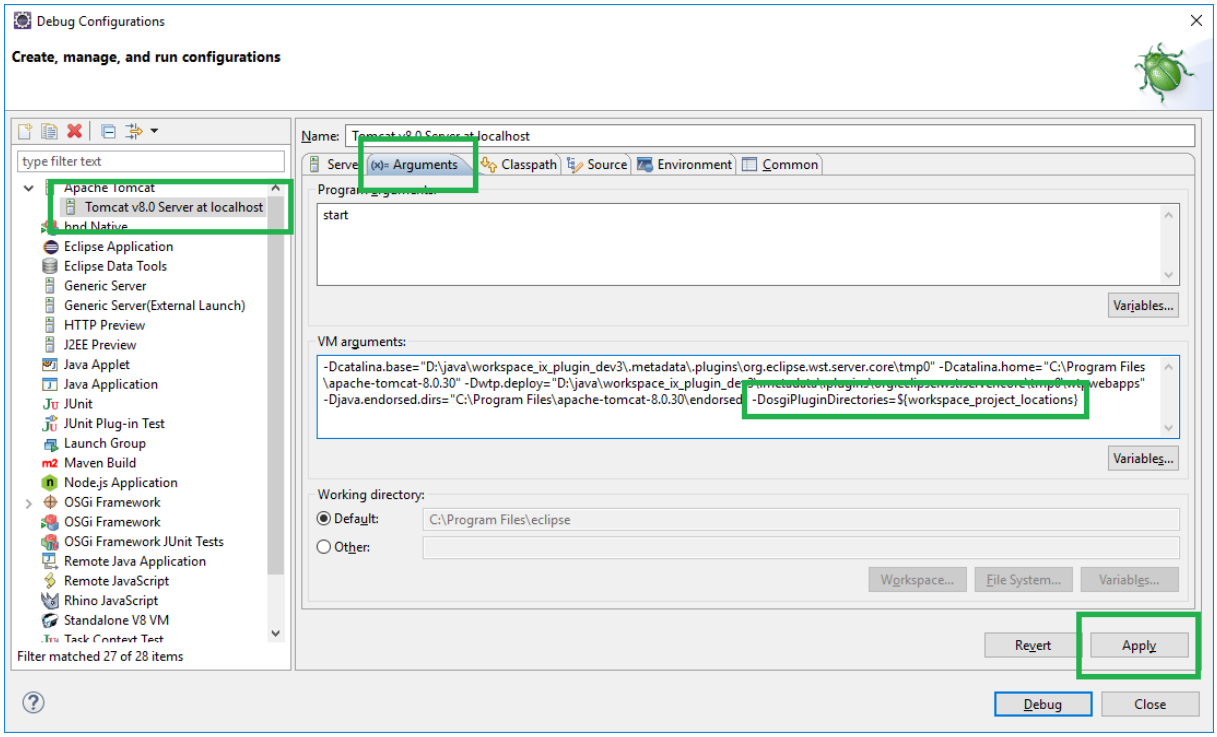


Abb. 27: Dialog 'Debug Configuration'

- Bestätigen Sie die Änderungen mit *Apply*, und schließen Sie den Dialog mit *Close*.
- Prüfen Sie, ob der Apache Tomcat mit dieser Änderung startet.
- Stoppen Sie den Apache Tomcat.

2.3.7 Target Platform für die Plug-in-Entwicklung im Eclipse definieren

Eine Target Platform konfiguriert im Wesentlichen alle Bundles, die zum Entwickeln und Ausführen von Plug-ins nötig sind. Der ELO Indexserver bringt eine Target Platform mit, die als Ausgangspunkt dienen kann. Es werden jedoch nicht alle von Eclipse angebotenen Optionen zu einer Target Platform unterstützt. Hauptsächlich soll die mitgelieferte Definition den Aufbau der Workspace erleichtern.

In den folgenden Schritten wird sie in den Eclipse-Workspace aufgenommen.

- Öffnen Sie unter dem Projekt *ix* den Pfad: *WebContent/resources/ix-osgi-platform*
- Öffnen Sie die Datei *ix-osgi-platform.target*. Sie enthält die Target-Platform-Konfiguration.

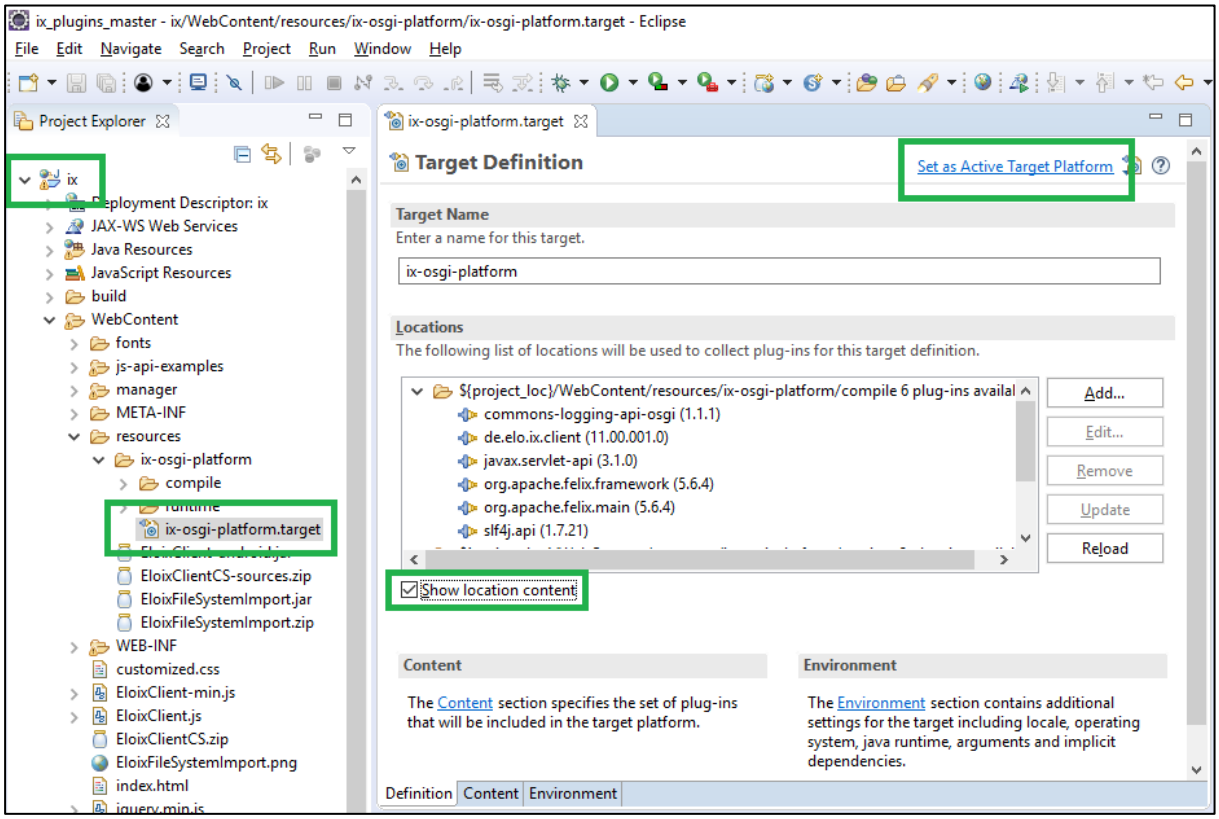


Abb. 28: Festlegen der Target Platform

- Um die enthaltenen Bundles zu sehen, aktivieren Sie die Option *Show location content*.
- Um die Definition in den Workspace aufzunehmen, klicken Sie auf den Link *Set as Active Target Platform*.
- Zur Kontrolle öffnen Sie im Eclipse-Menü *Window* > *Preferences…* > *Plug-in Development* > *Target Platform*.

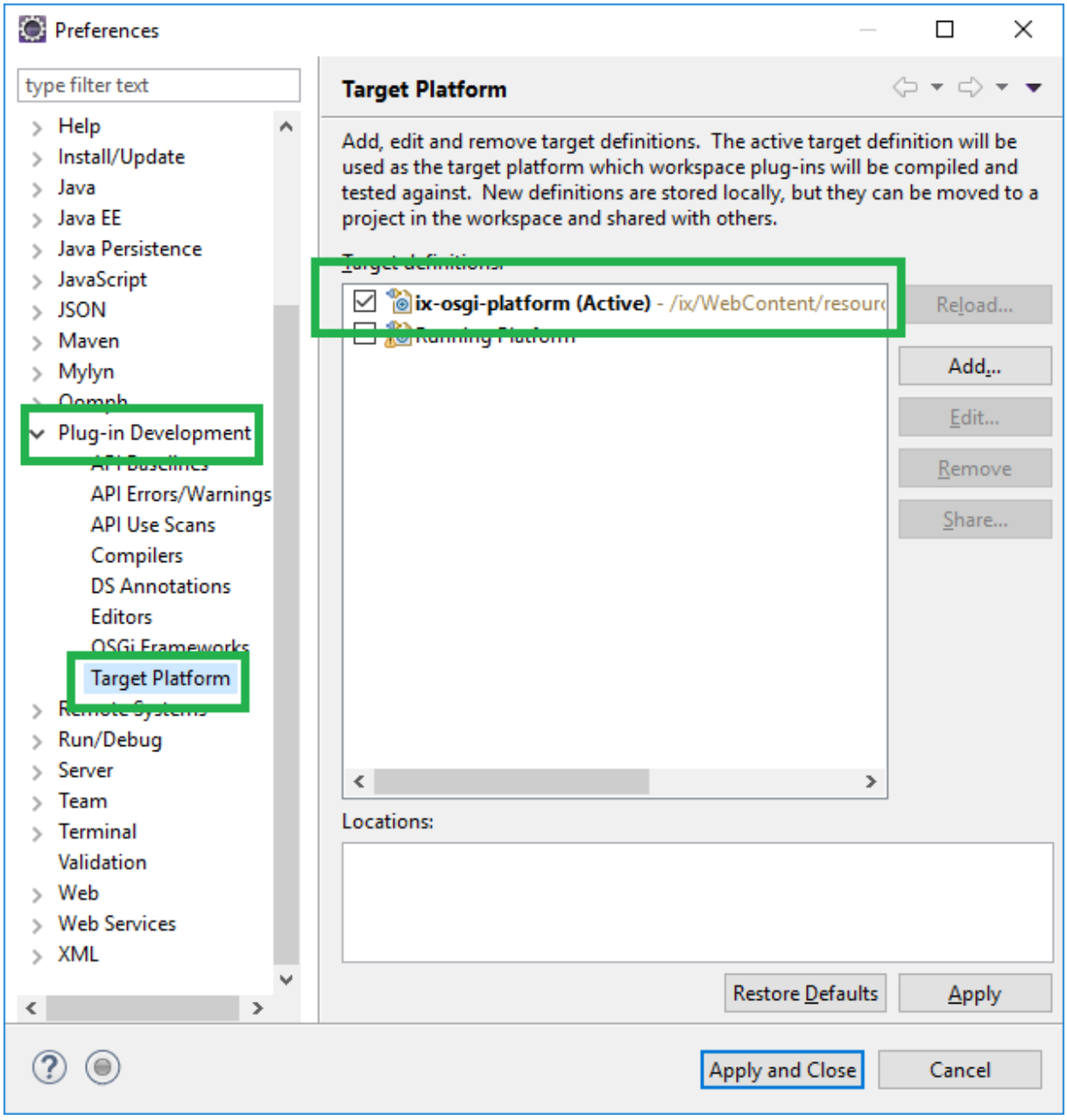


Abb. 29: Überprüfung der Target Platform

Die Target Platform *ix-osgi-platform* muss selektiert sein.

Information: Die Bundles im Verzeichnis *compile* der Target Platform werden nur zum Kompilieren bereitgestellt. Diese Bundles sind in den ELO Indexserver eingebaut und werden zur Laufzeit aus dem ELO Indexserver geladen.

Information: Wenn Sie auf eine neue Version des ELO Indexserver wechseln, wiederholen Sie die oben beschriebenen Schritte. Statt *Set as active Target Platform* heißt es dann *Reload Target Platform.*

2.3.8 Quellcode und Dokumentation der EloixClient.jar bereitstellen

Um die Quellen und Dokumentationen der Datei *EloixClient.jar* zur Verfügung stellen zu können, erstellen Sie mit den folgenden Schritten für die EloixClient.jar ein zusätzliches Eclipse-Projekt.

- Im Eclipse-Menü: *Window* > *Show View* > *Other…*

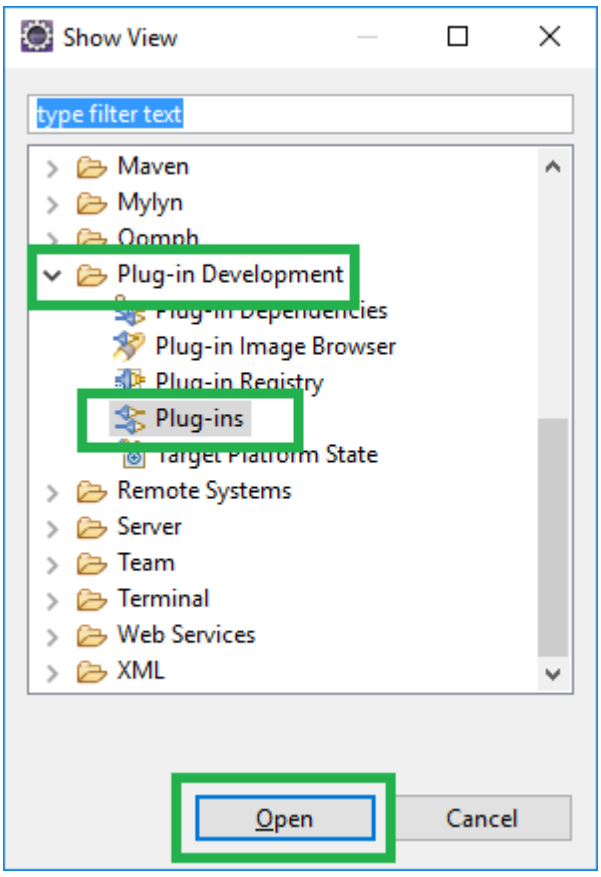


Abb. 30: Auswahl der Ansicht 'Plug-in'

- In der Ansicht *Plug-ins* folgen Sie den Angaben im nachfolgenden Bild:

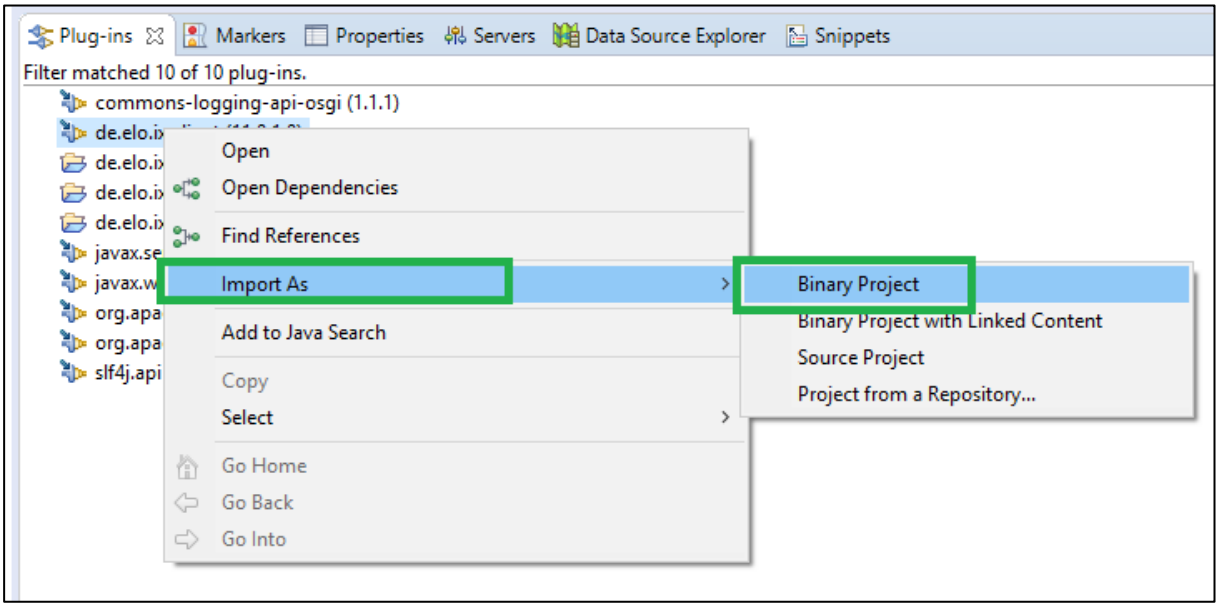


Abb. 31: Anlegen eines neuen Projekts

Sie erhalten dieses neue Projekt.

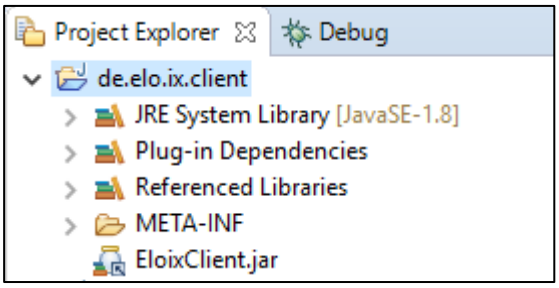


Abb. 32: Neues Projekt

- Wählen Sie: *Eclipse > Project Explorer > Kontextmenü > de.elo.ix.client > Properties*

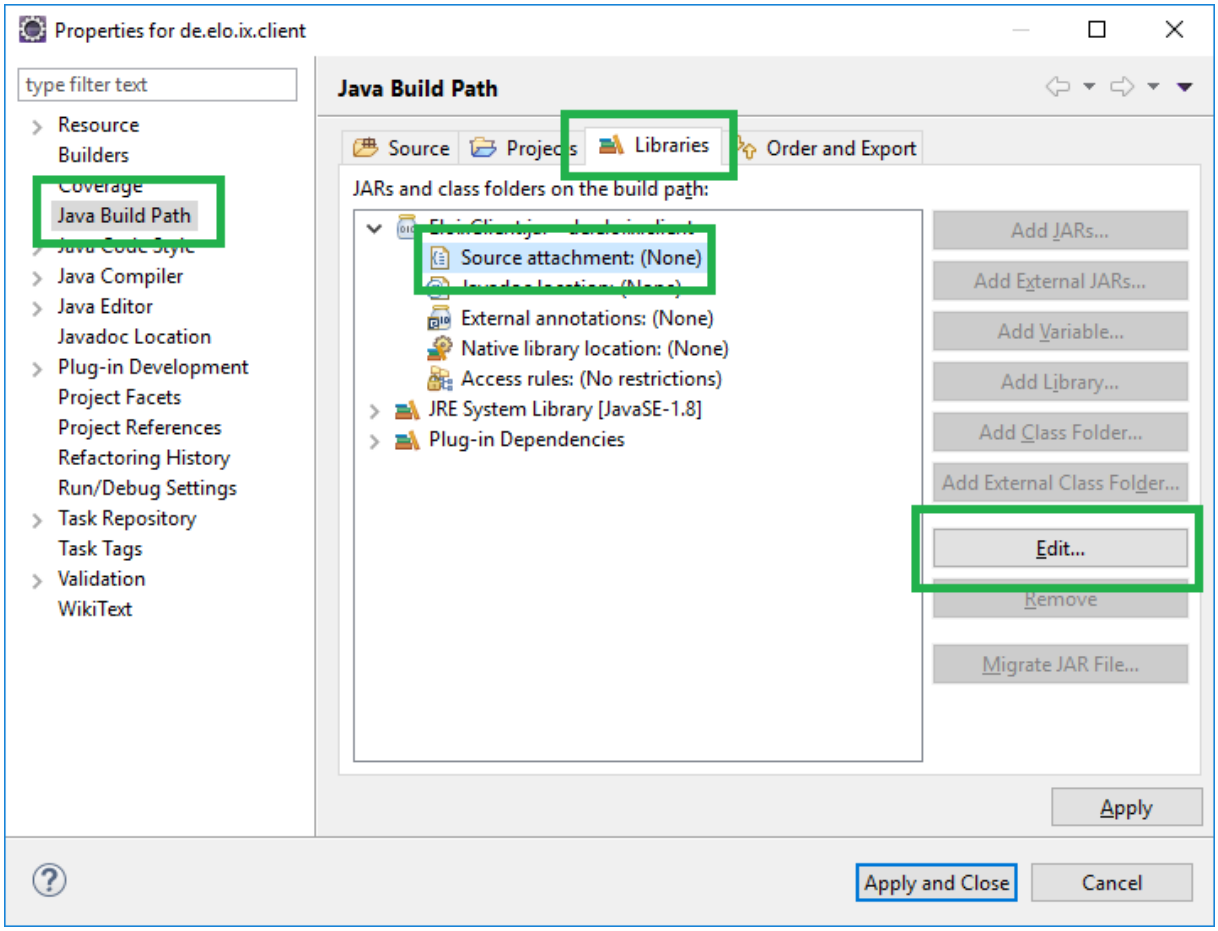


Abb. 33: Auswahl des Punktes 'Source attachment'

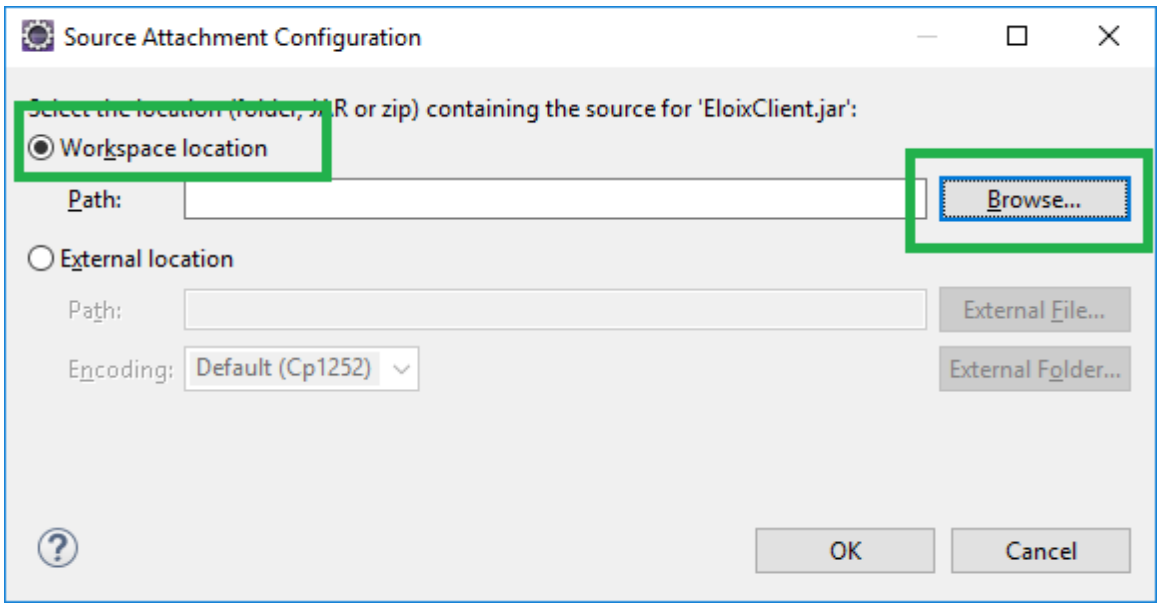


Abb. 34: Dialog 'Source Attachment Configuration'

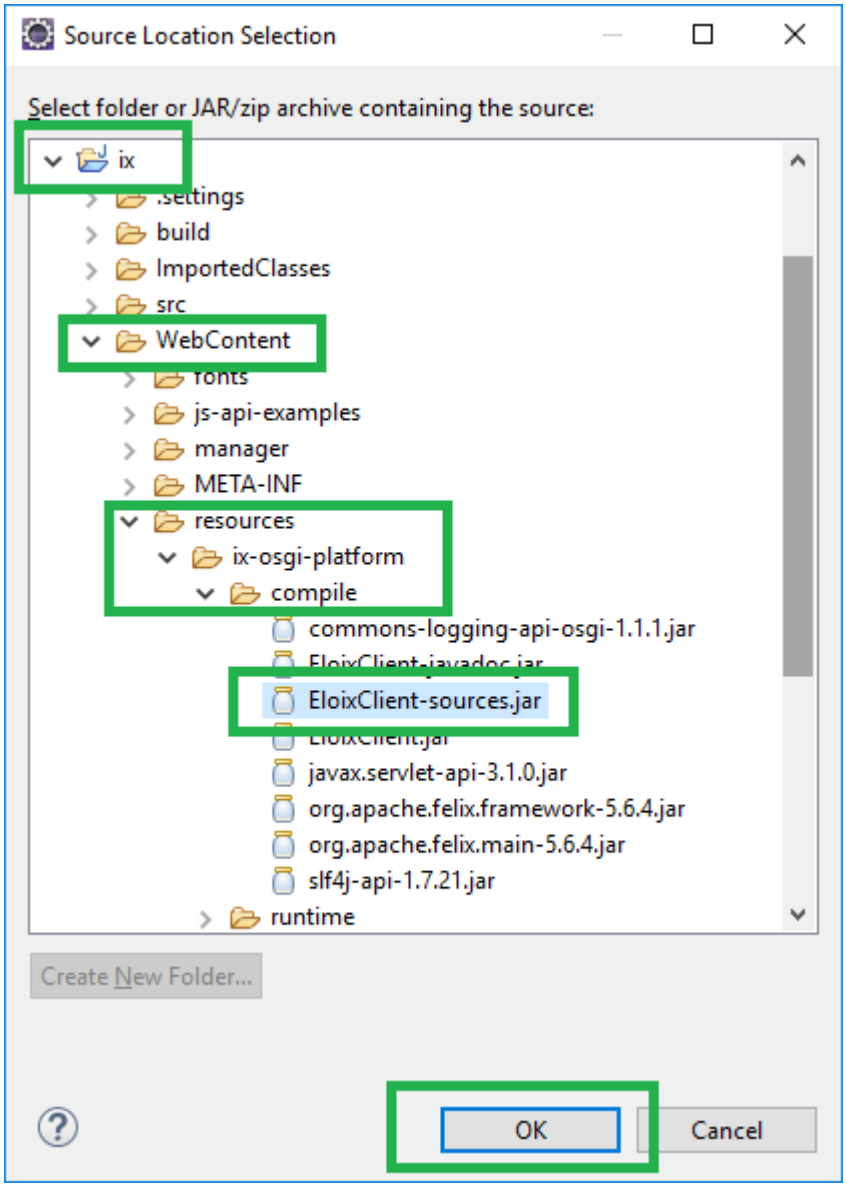


Abb. 35: Auswahl des Pfades

Nun richten Sie die Referenzdokumentation ein:

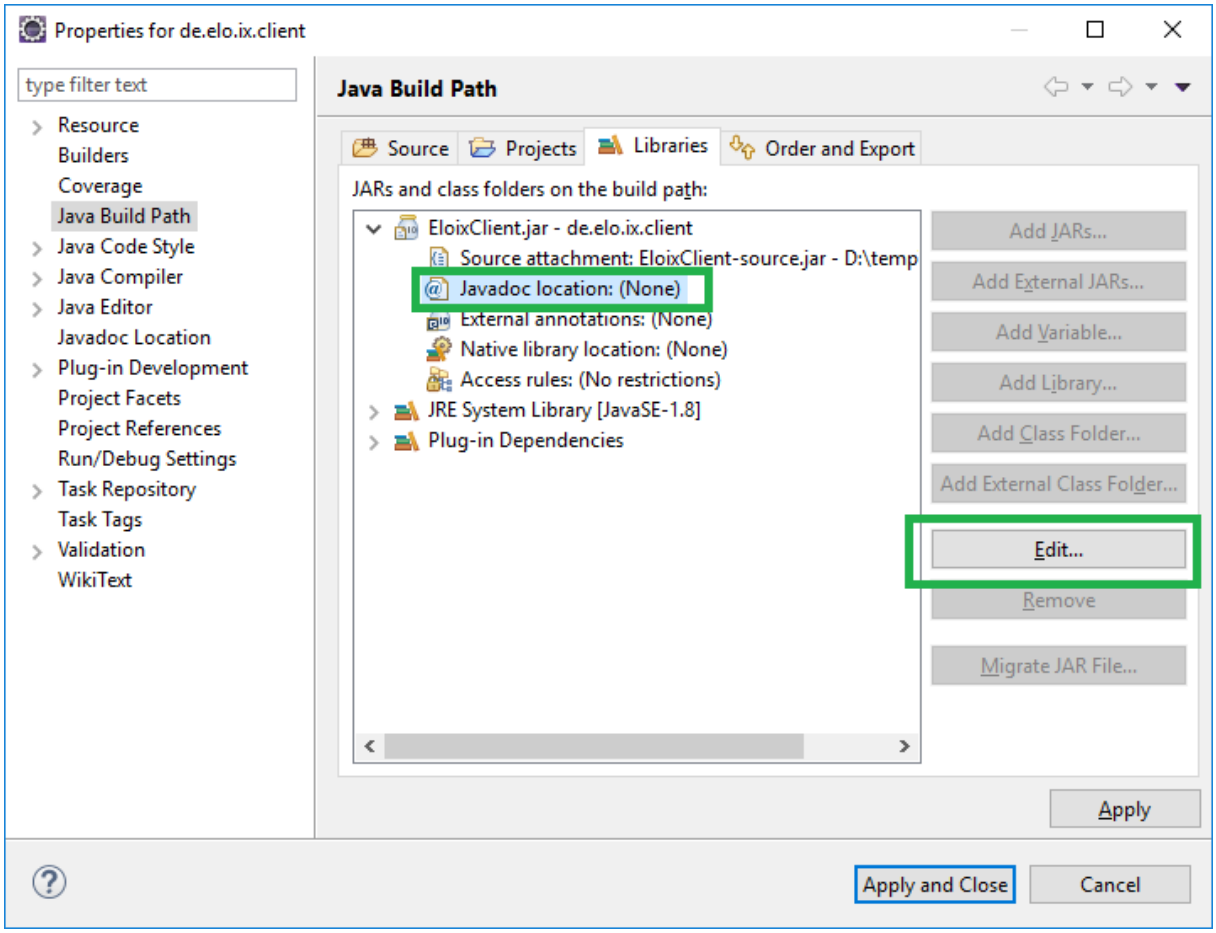


Abb. 36: Auswahl des Punktes 'Javadoc location'

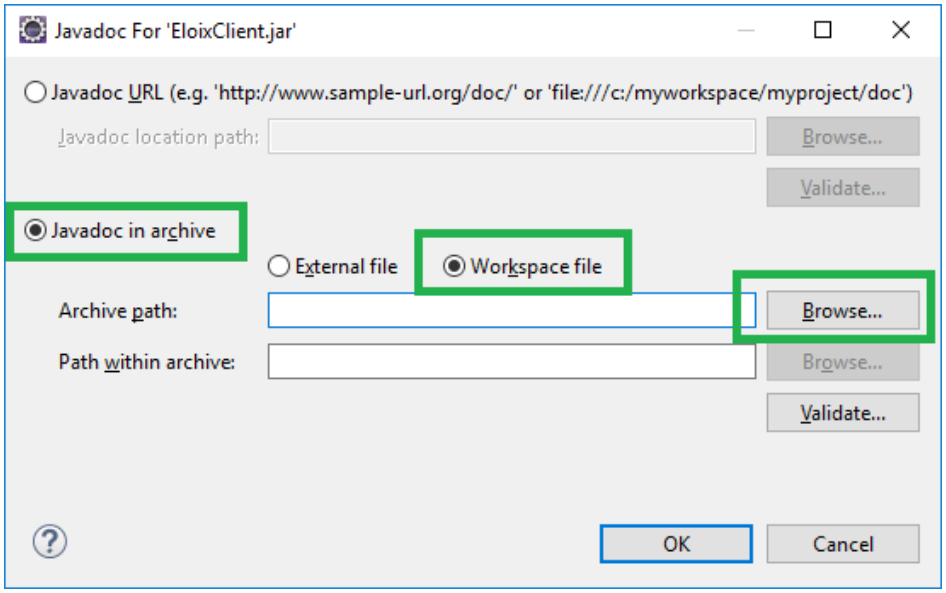


Abb. 37: Einstellungen für die Javadoc-Datei

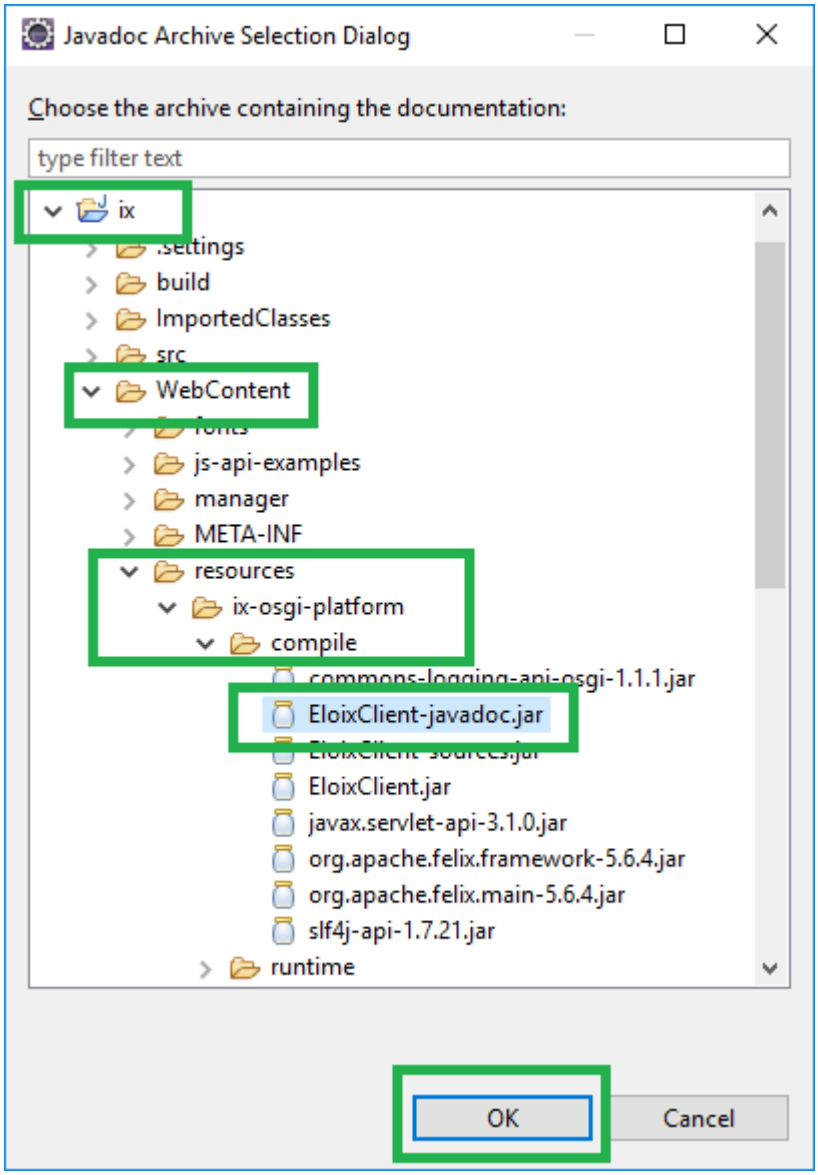


Abb. 38: Pfad zur Javadoc-Datei

Bestätigen Sie die offenen Dialoge mit *OK* oder *Apply and Close*.

Information: Um auf eine neue Version der *EloixClient.jar* zu wechseln, ist das de.elo.ix.client-Projekt zu löschen (inklusive Dateien im Workspace) und neu anzulegen.

2.4 Plug-in erstellen

2.4.1 Neues Plug-in-Projekt erstellen

- *Eclipse > Menü > File > New > Project…*

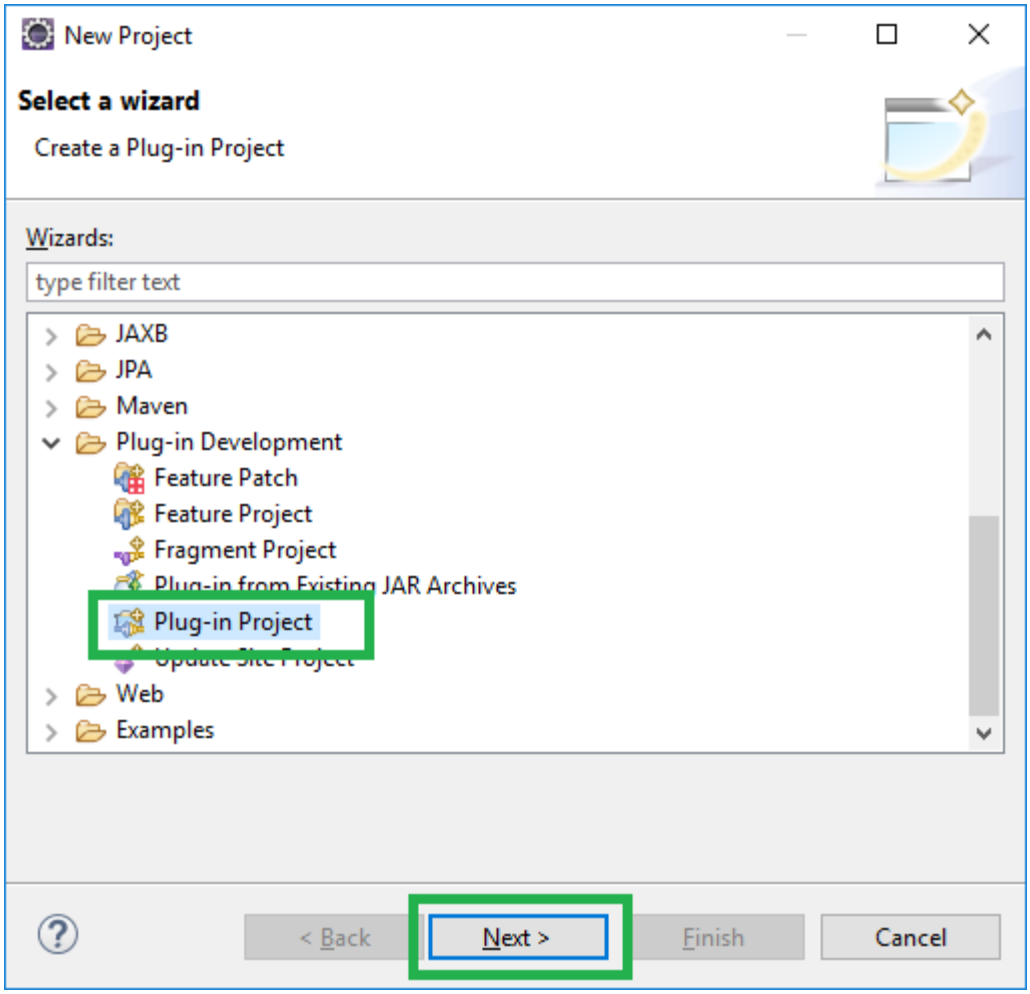


Abb. 39: Auswahl des Assistenten für Plug-in-Projekte

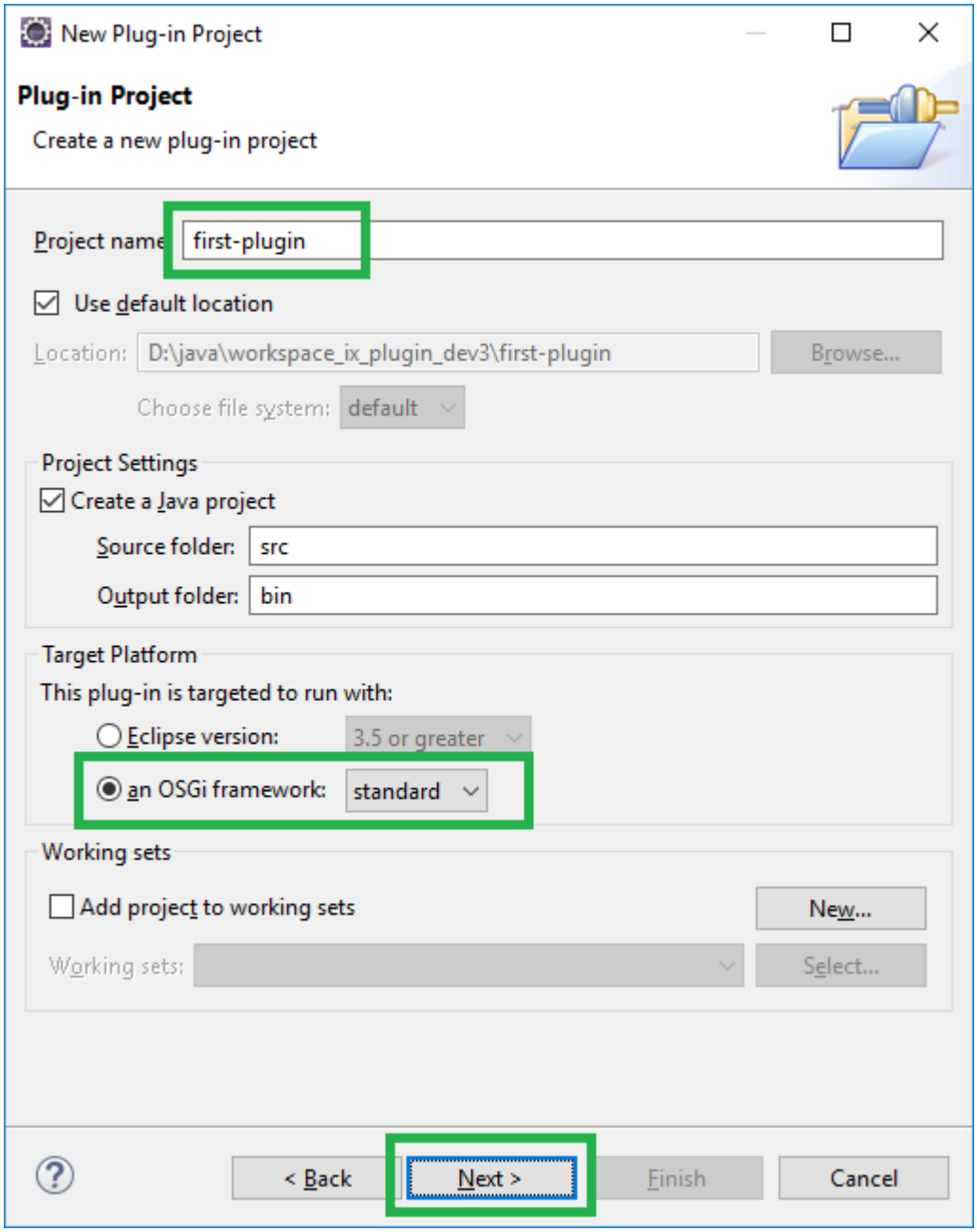


Abb. 40: Anlegen des Plug-in-Projekts

- Geben Sie eine ID für das Plug-in im Format von Java-Package-Namen an. Es sollte sich um eine eindeutige ID handeln. Beispielsweise kann die umgekehrte Adresse des Web-Auftritts des Herstellers verwendet werden.
- Die Präfixe *de.elo* und *com.elo* sind für die ELO Digital Office GmbH reserviert.

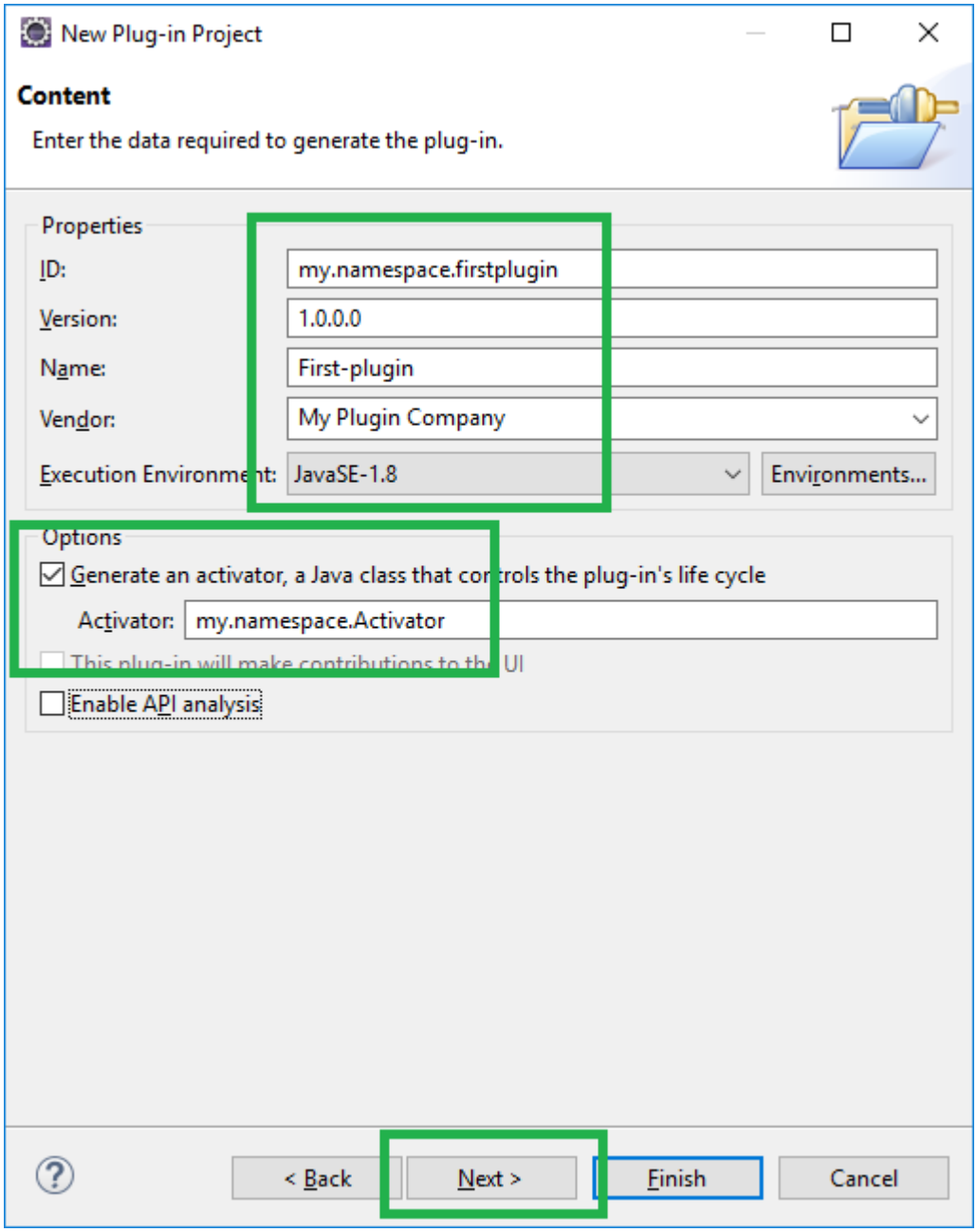


Abb. 41: Einstellungen für das Plug-in

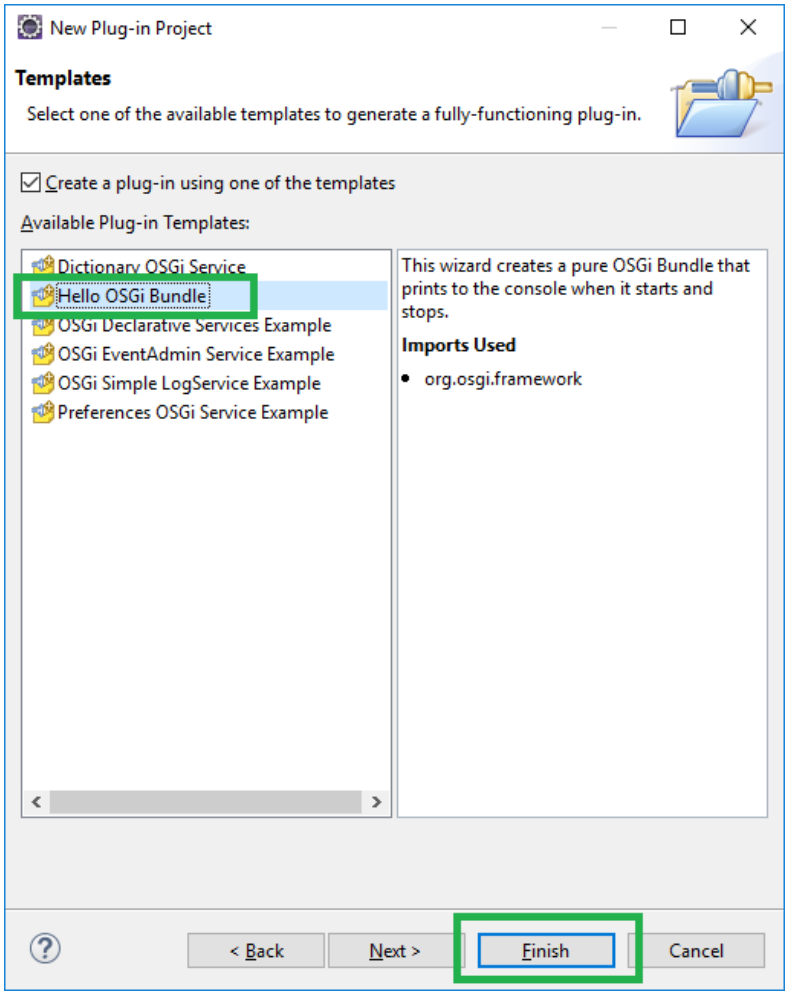


Abb. 42: Auswahl einer Vorlage

Zu empfehlen ist, die Plug-in-Ansicht im nachfolgenden Dialog nicht zu aktivieren. Dann bleibt das aktuelle Layout der Entwicklungsumgebung erhalten.

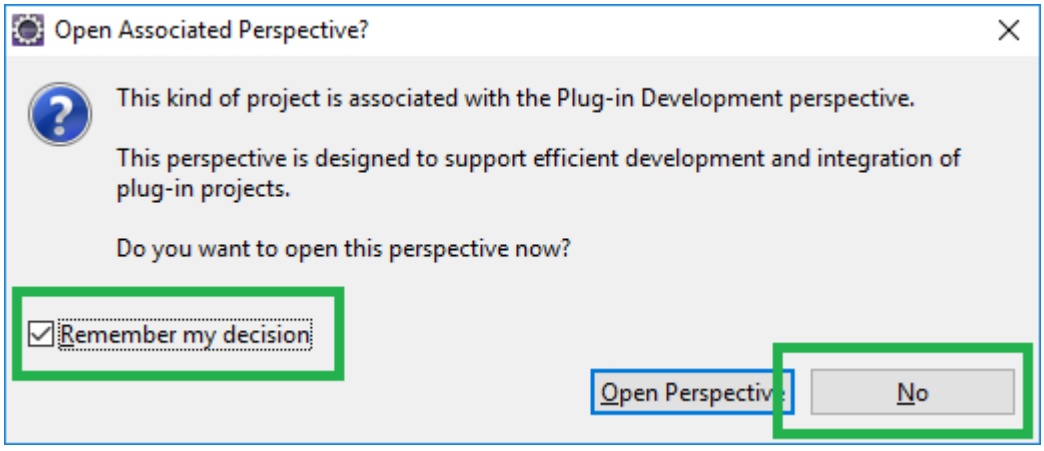


Abb. 43: Deaktivieren der Plug-in-Ansicht

Damit ist das Plug-in-Projekt erstellt. Nun definieren Sie das Bundle der *EloixClient.jar* als erforderliches Plug-in.

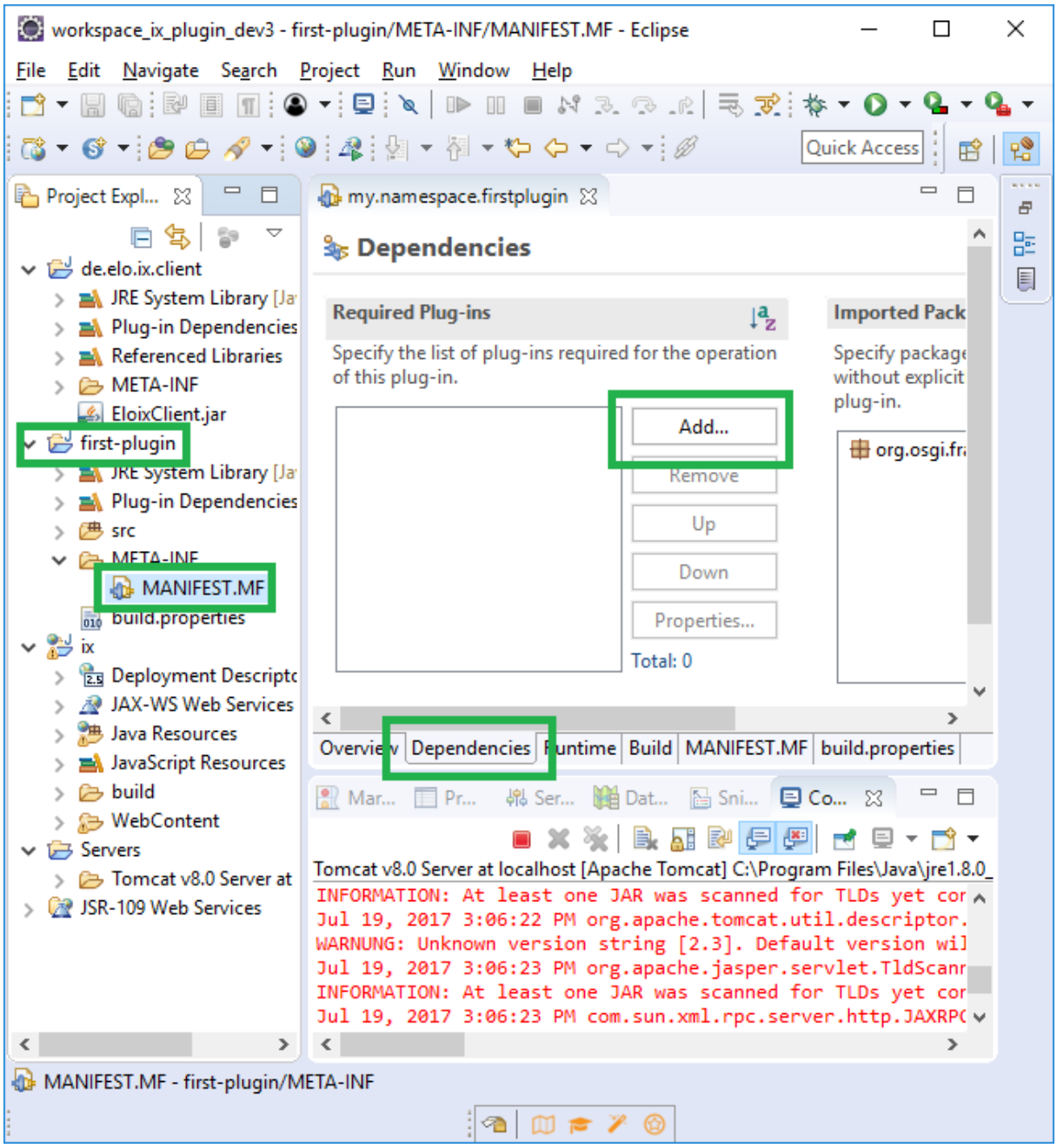


Abb. 44: Bearbeiten der Datei 'MANIFEST.MF'

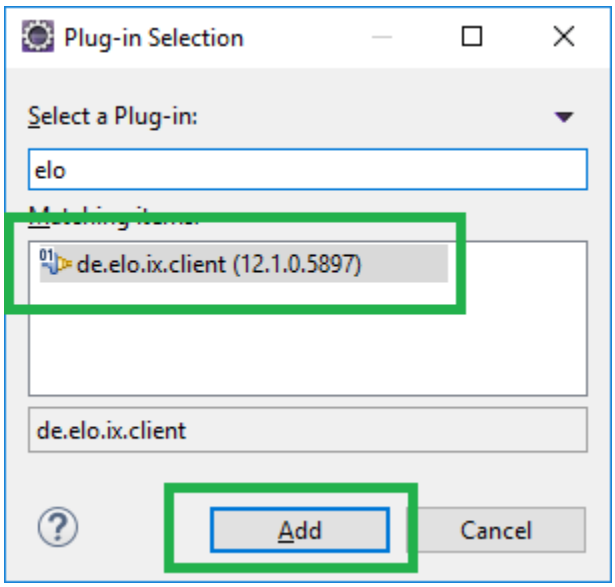


Abb. 45: Hinzufügen des Plug-ins

- Mit STRG + S speichern Sie die geänderte Datei *MANIFEST.MF*.

2.4.2 Activator-Klasse anpassen

Die vom Assistenten erstellte `Activator`-Klasse hat die Aufgabe, das Plug-in beim Start zu initialisieren. Wenn das Plug-in gestoppt wird, kann die `Activator`-Klasse intern verwendete Ressourcen aufräumen.

Die *EloixClient.jar* bringt eine `Activator`-Klasse mit, `PluginActivator`, die nützliches Verhalten bereits implementiert und den OSGi-spezifischen Code übernimmt. Die `Activator`-Klasse des neuen Plug-ins sollte davon abgeleitet werden.

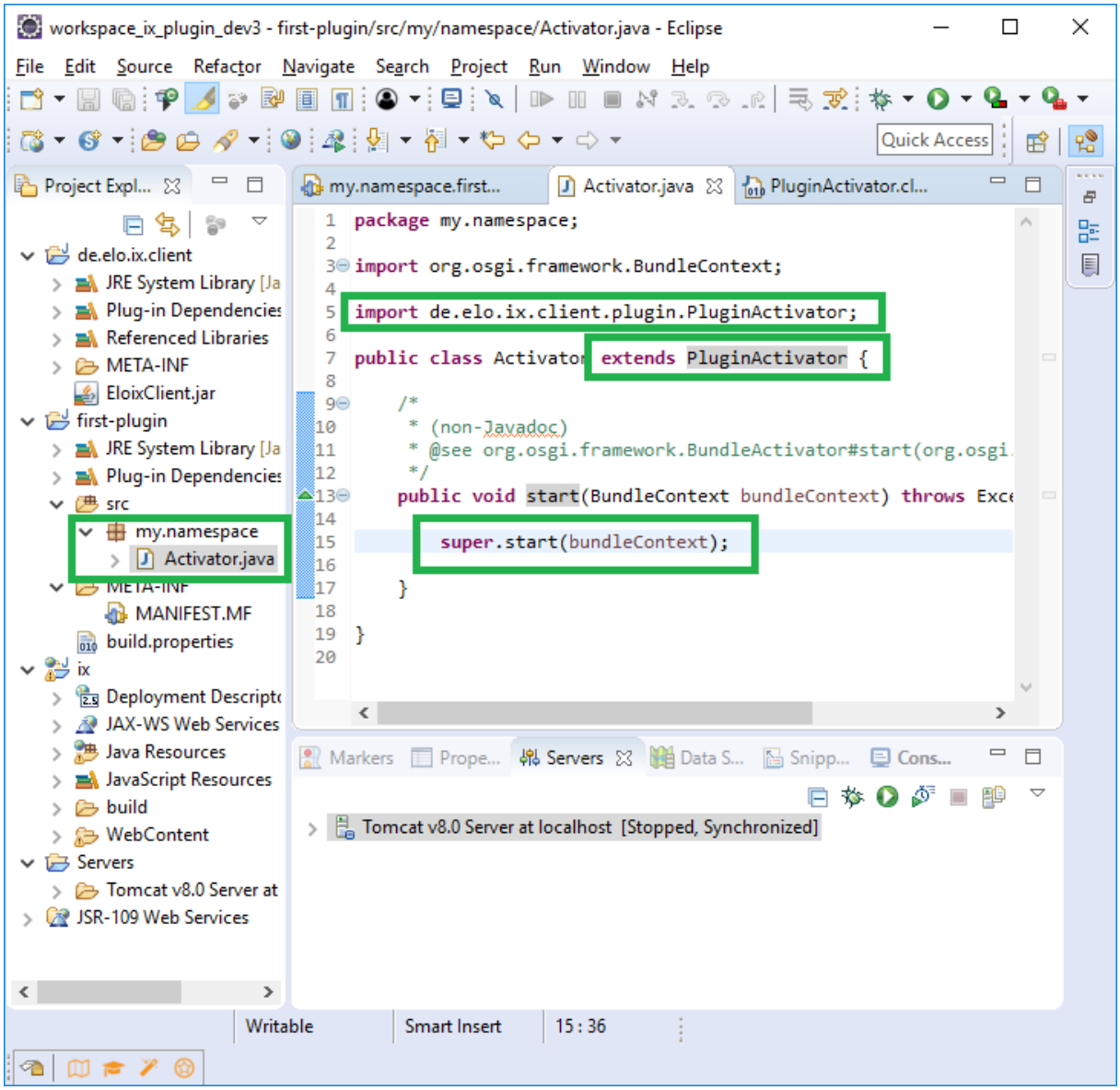


Abb. 46: Anpassen der Activator-Klasse

2.4.3 Schnittstelle für Server-Ereignisse IXServerEvents implementieren

Um die Server-Ereignisse abzufangen, ist eine Klasse zu erstellen, die das Interface `IXServerEvents` implementiert. Wie zur `Activator`-Klasse gibt es auch hier eine Basisklasse, die die Arbeit erleichtert. Sie heißt `IXServerEventsBase`. Im Konstruktor ermittelt sie die überladenen Methoden und kann dadurch in ihrer Funktion `IXServerEvents.hasMethod` die Prüfung bereitstellen, ob eine Methode vorhanden ist.

- Neue Klasse `IXServerEventsImpl` erstellen: Kontextmenü von Package `my.namespace` öffnen > *New* > *Class*.
- Name: `IXServerEventsImpl`

- Superclass: `de.elo.ix.client.plugin.IXServerEventsBase`
- Den erforderlichen Konstruktor mit Parameter hinzufügen. Z. B. Maus über rot unterstrichenes `IXServerEventsImpl` schieben, im Popup-Fenster wählen: *Add constructor….*
- Eine Funktion überschreiben. Z. B. STRG + LEERTASTE im Klassenrumpf, „`onbeforecheck`“ tippen und `onBeforeCheckinSord` auswählen.

Ergebnis:

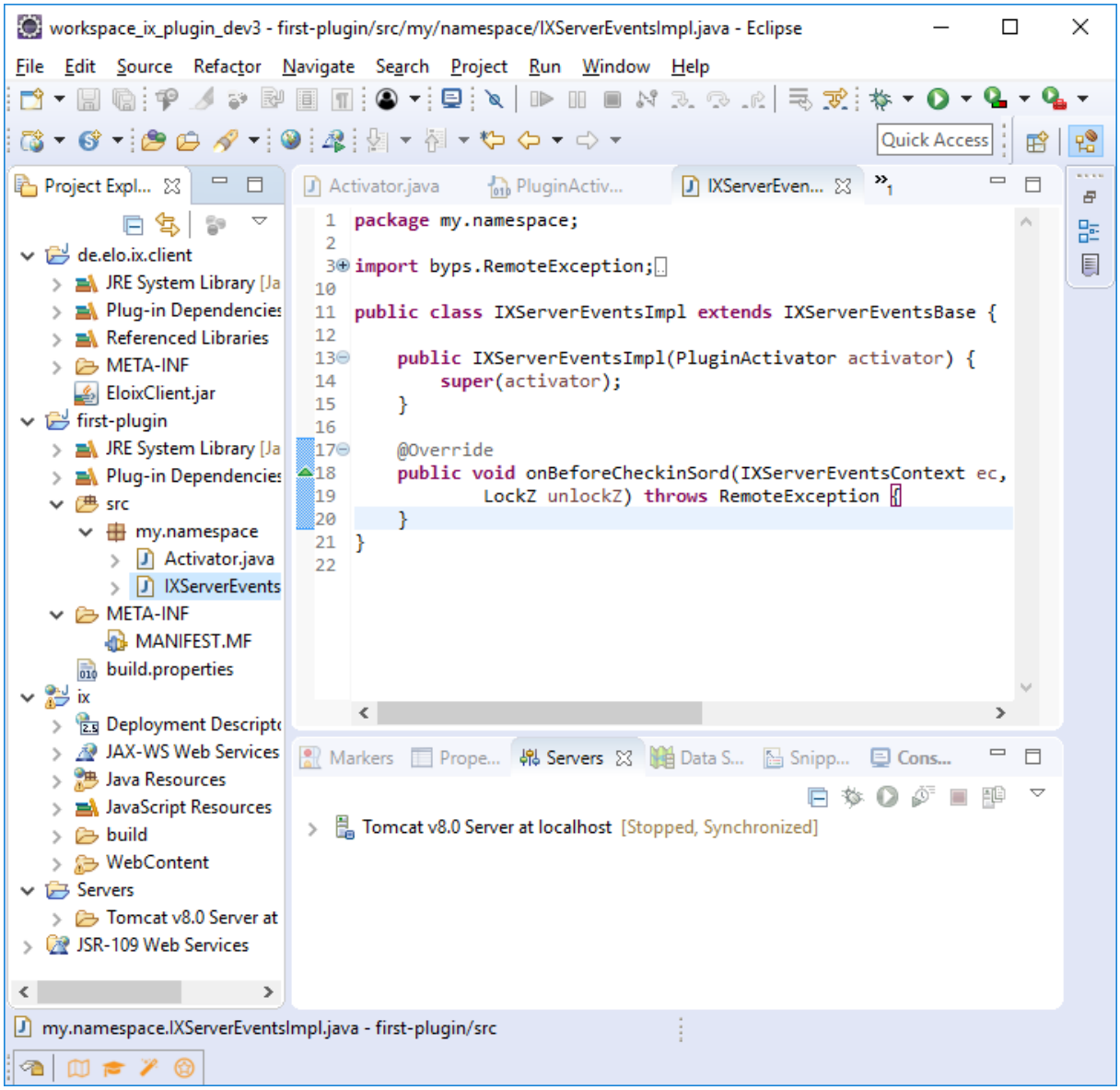


Abb. 47: Schnittstelle für Server-Ereignisse hinzufügen

- Diese Klasse nun im `Activator` (als OSGi-Service) registrieren. Dazu nach `super.start(bundleContext)` einfügen:

```
registerService(IXServerEvents.class, new IXServerEventsImpl(this));
```

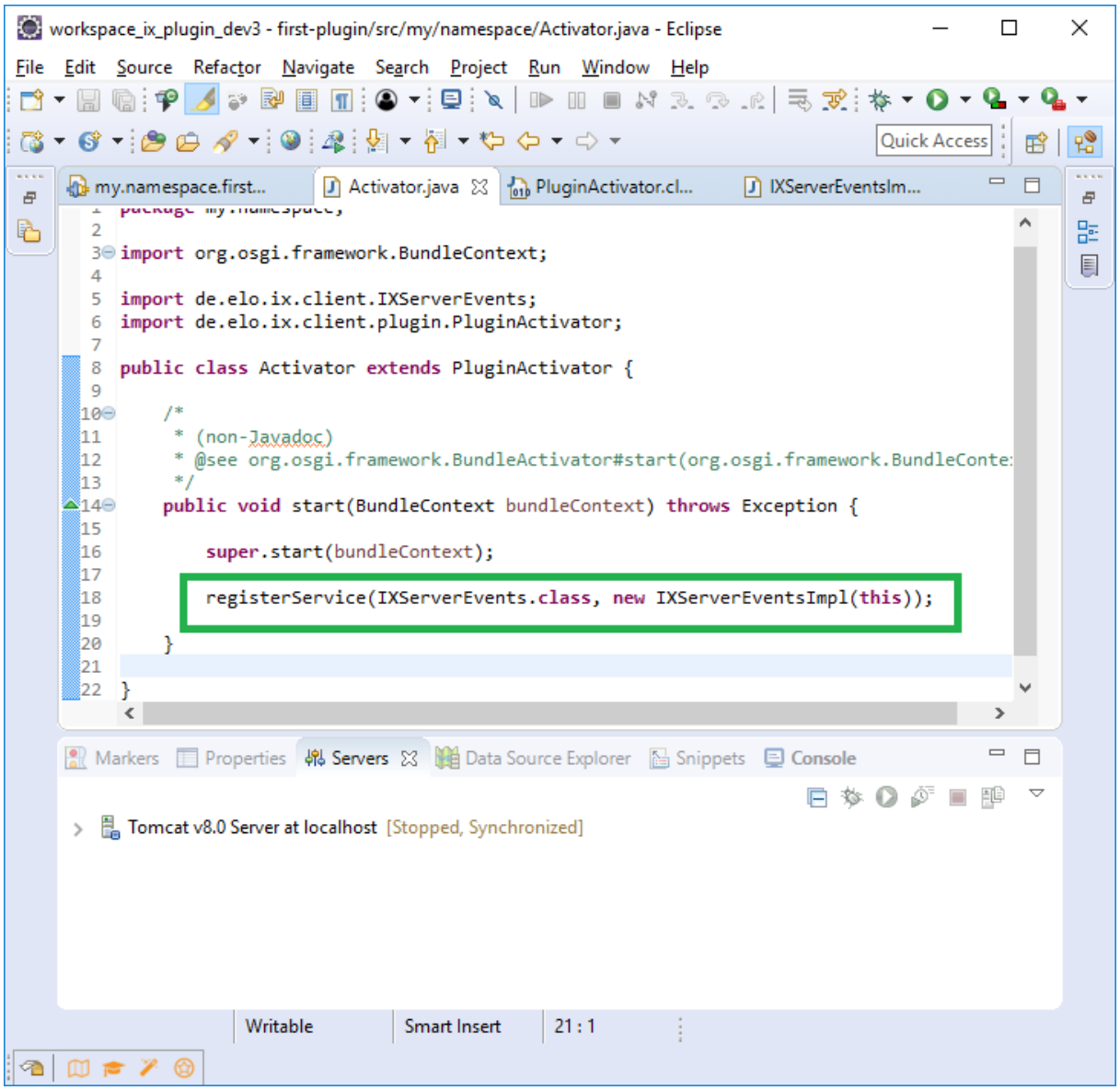


Abb. 48: Registrierung der Klasse im Activator

2.4.4 Registrierte Funktionen bereitstellen

Die in 2.4.3 vorgestellte Klasse `IXServerEventsBase` erleichtert auch die Implementierung von registrierten Funktionen. Sie können direkt als Elementfunktionen programmiert werden, Beispielsweise:

```
public class IXServerEventsImpl extends IXServerEventsBase {
…
  public String RF_StringFunction(IXServerEventsContext ec, String args)
      throws RemoteException {
    return args + " invoked";
```

```
  }

  public Any RF_AnyFunction(IXServerEventsContext ec, Any args)
      throws RemoteException {
    Object obj = AnyToObject.toObject(args);
    String sret = obj + " invoked";
    Any ret = AnyToObject.fromObject(sret);
    return ret;
  }
}
```

Hierbei ist zu beachten, dass registrierte Funktionen die folgenden Methodensignaturen aufweisen müssen:

Clientseitiger Aufruf, `conn.ix()`…	Methodensignatur im Plug-in
`executeRegisteredFunctionString( "RF_Function1", "parameter1");`	`public String RF_Function1( IXServerEventsContext ec, String args)`
`executeRegisteredFunction( "RF_Function2", any);`	`public Any RF_Function2( IXServerEventsContext ec, Any args)`

2.5 Plug-in debuggen

Um das Plug-in aus dem Workspace heraus direkt im ELO Indexserver-Prozess debuggen zu können, benötigt man einen besonderen „Kunstgriff“. Dafür macht man sich zu Nutze, dass OSGi-Bundles aus einer entpackten JAR – also aus einem Verzeichnis – geladen werden können.

Nur entspricht das Projekt-Verzeichnis im Eclipse-Workspace nicht genau dem des ausgepackten OSGi-Bundles. Denn die Java-Class-Dateien liegen im Unterordner *bin* und nicht, wie von OSGi erwartet, direkt auf der Hauptebene.

2.5.1 Kunstgriff

Der Kunstgriff besteht nun darin, im Classpath des Bundles den Ordner *bin* anzugeben.



Beachten Sie: Vor dem Fertigstellen des Plug-ins (s. 2.7) muss dieser Verweis wieder entfernt werden. Andernfalls sind die Klassen doppelt in der JAR-Datei enthalten.

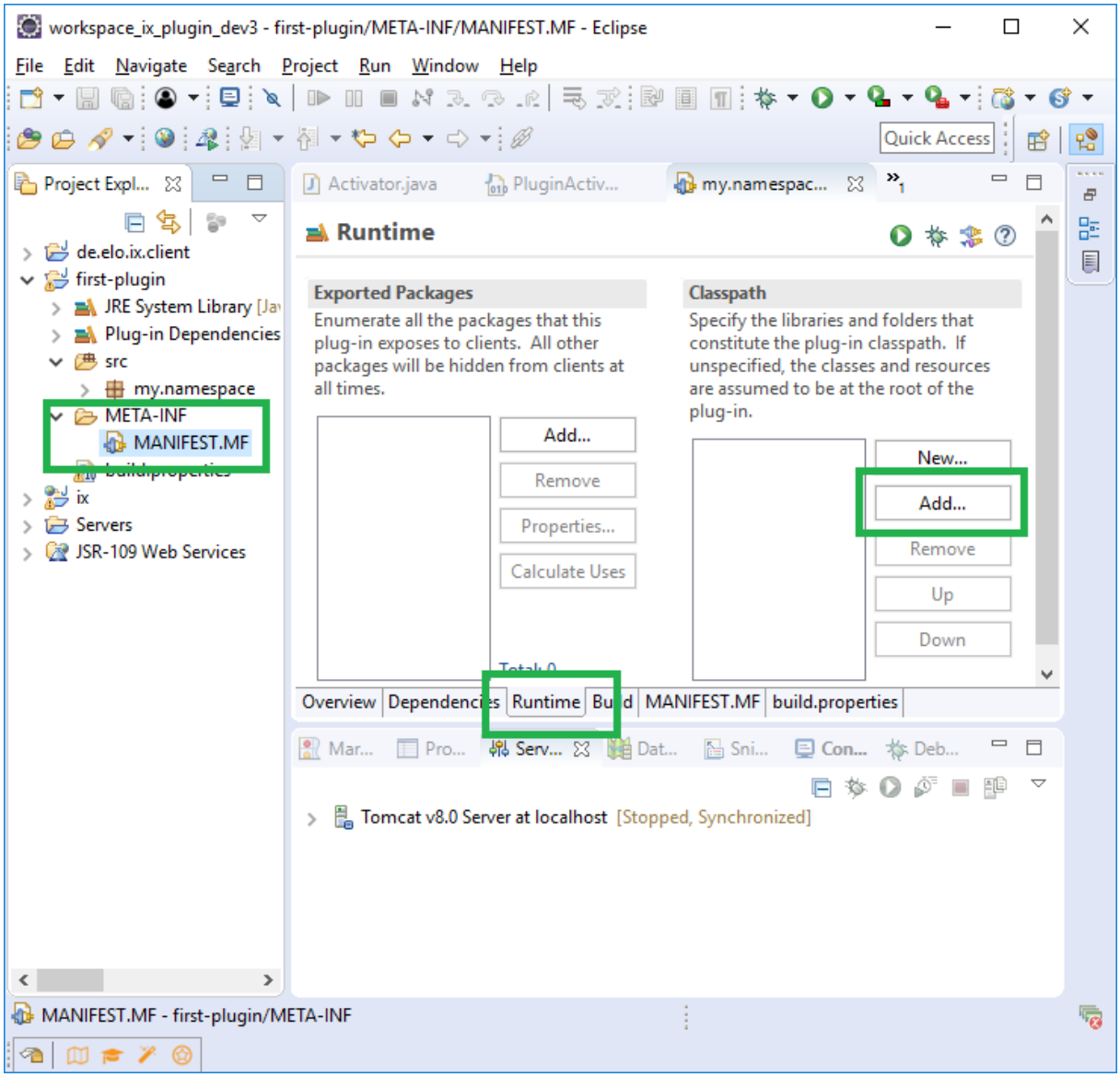


Abb. 49: Bereich 'Classpath', Schaltfläche 'Add'

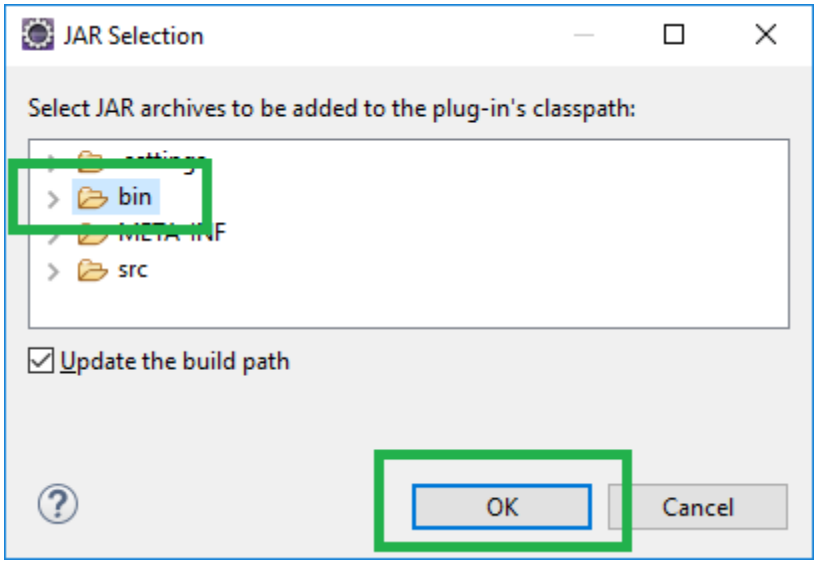


Abb. 50: Ordner 'bin'

- *MANIFEST.MF* speichern

2.5.2 Prüfung, dass das Plug-in geladen wird

Wenn man den Apache Tomcat über das Eclipse-Servers-Fenster startet, soll das Plug-in vom ELO Indexserver geladen werden.

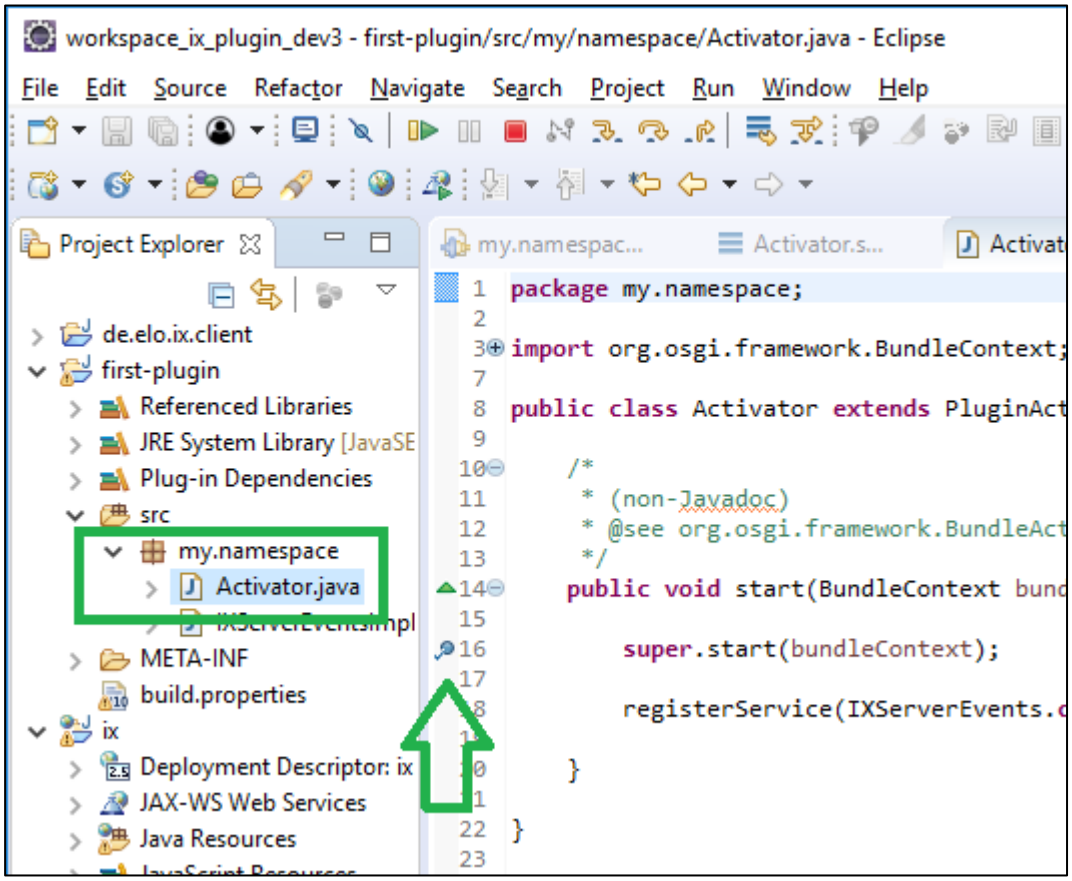


Abb. 51: Breakpoint

Dass dem so ist, kann man anhand eines Breakpoints z. B. in der `Activator`-Klasse nachweisen (Doppelklick am linken Rand des Editor-Fensters).

Anschließend den Apache Tomcat starten. Am Breakpoint wird die Ausführung angehalten. Allerdings findet Eclipse nicht von allein den Quellcode zur Activator-Klasse.

2.5.3 Quellcode zuweisen

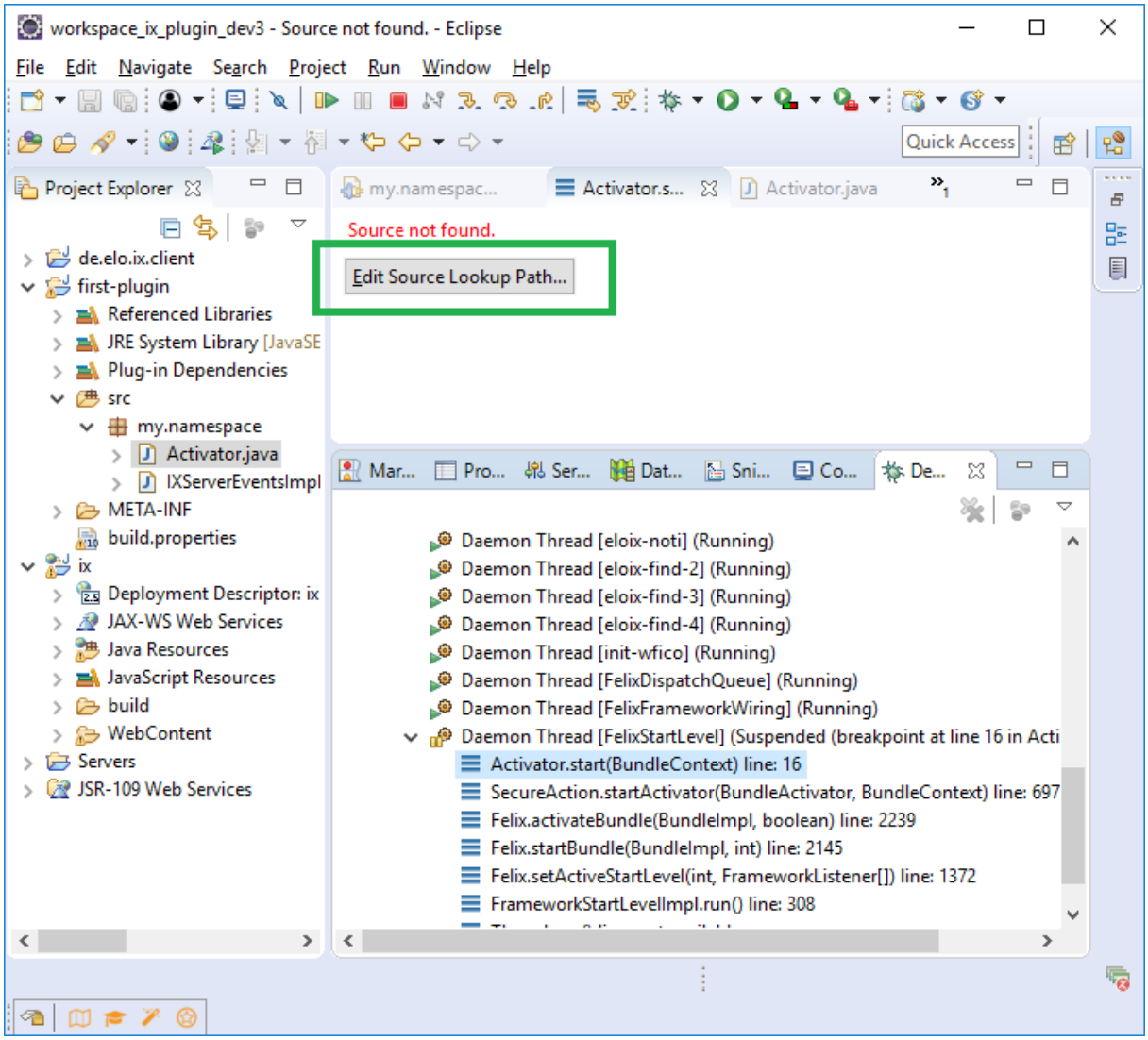


Abb. 52: Schaltfläche 'Edit Source Lookup Path...'

Im Editor-Fenster, wo der Quellcode angezeigt werden soll, klicken Sie auf die Schaltfläche *Edit Source Lookup Path…*.

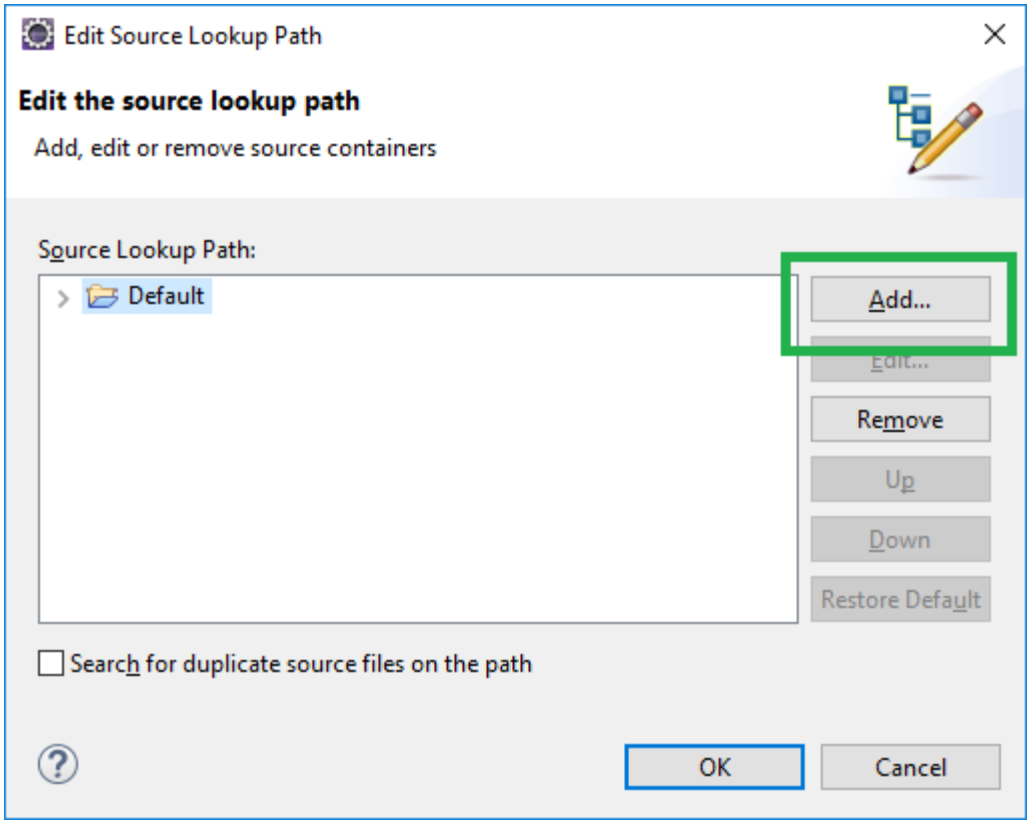


Abb. 53: Ordner hinzufügen

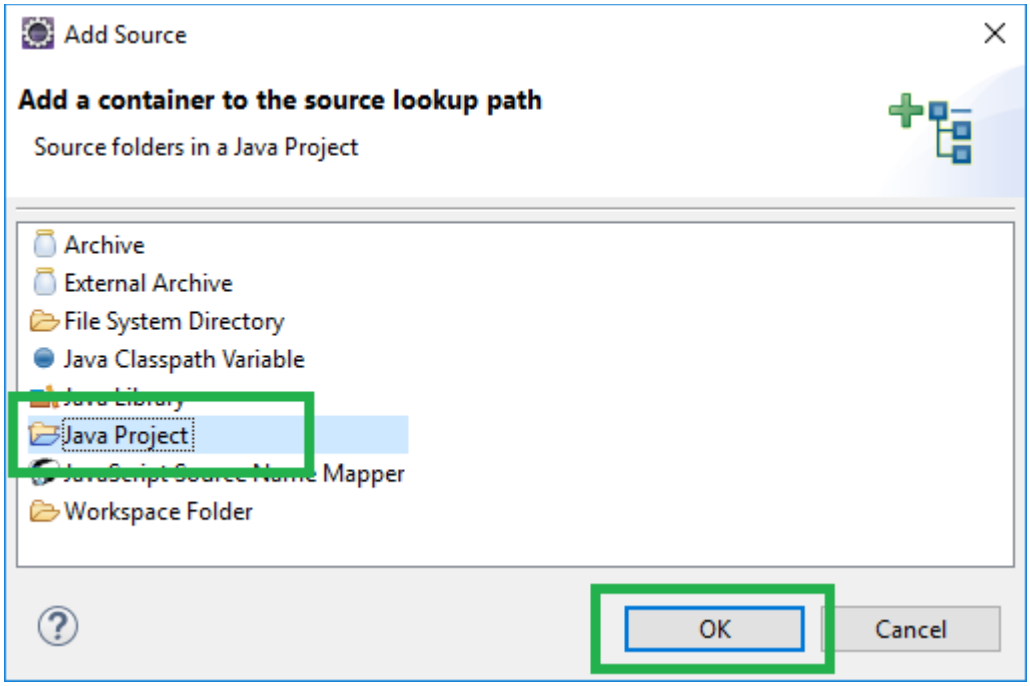


Abb. 54: Verzeichnis 'Java Project' auswählen

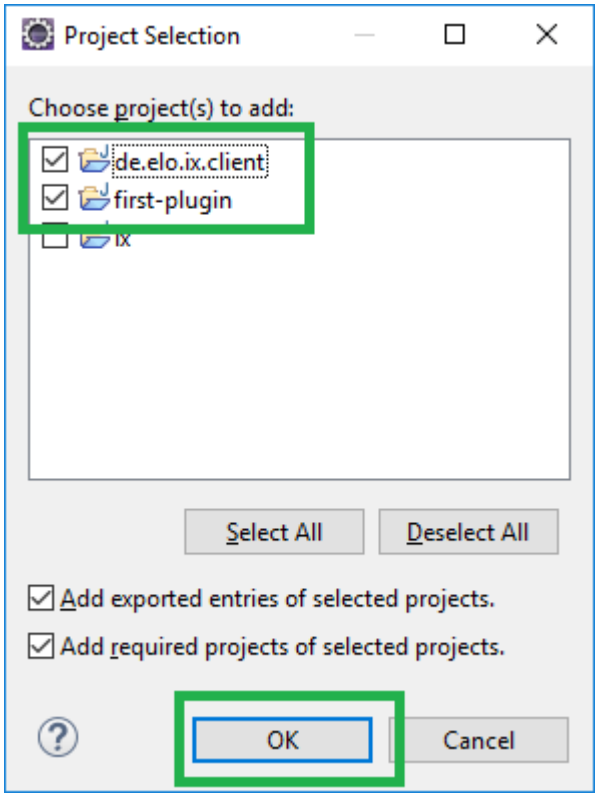


Abb. 55: Projekte auswählen

Nachdem alle offenen Dialoge mit OK geschlossen sind, wird die richtige Quellcodestelle angezeigt.

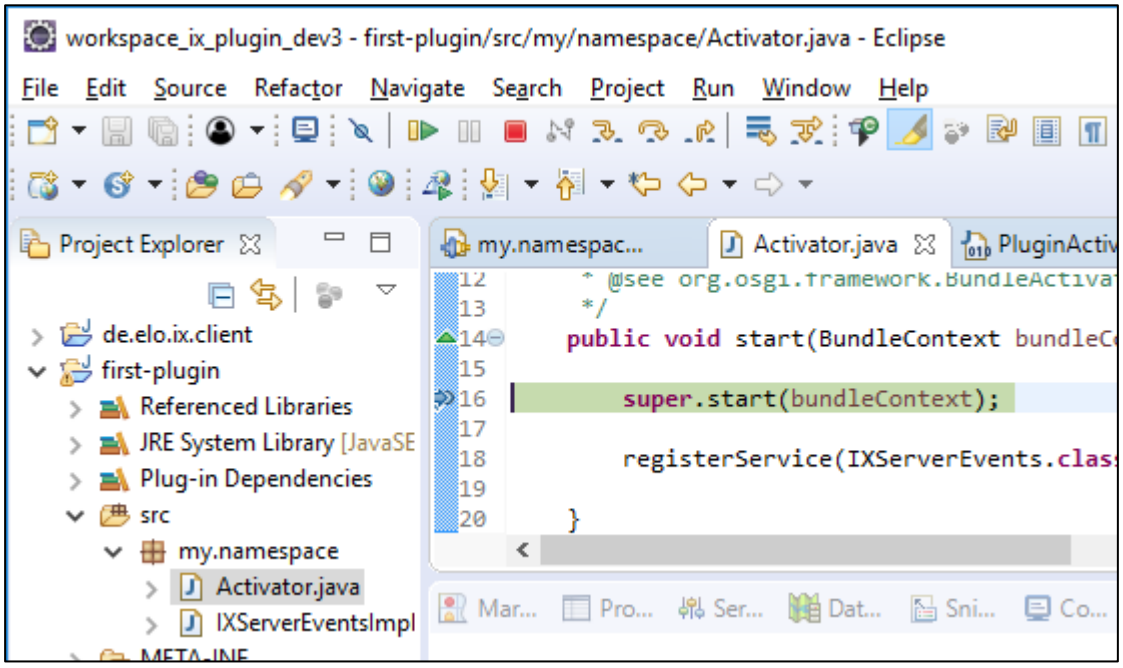


Abb. 56: Quellcode wurde zugewiesen



Information: Kleinere Änderungen können direkt während des Debuggings im Quellcode vorgenommen werden und sind nach dem Speichern sofort wirksam.

2.6 Abhängigkeiten einbinden

Ein Plug-in kann neben der Indexserver-Client-Bibliothek von weiteren OSGi-Bundles oder JAR-Bibliotheken abhängig sein. Diese Abhängigkeiten können auf zwei Arten aufgelöst werden.

Zum einen kann im Manifest des Plug-ins ein Eintrag unter *Require-Bundle* erfolgen. Das OSGi-Framework wird beim Laden des Plug-ins das genannte Bundle unter den installierten Bundles aufsuchen und seine exportierten Packages zugänglich machen. Falls es sich bei der Abhängigkeit um eine einfache JAR-Datei handelt, kann sie in wenigen Schritten in ein Bundle eingepackt werden (siehe *2.6.1 Abhängiges Bundle referenzieren*).

Der zweite Weg besteht darin, die Abhängigkeiten in das Plug-in einzubetten (siehe *2.6.2 JAR-Bibiliothek in das Plug-in einbetten*). Die daraus entstehende JAR wird häufig eine „Fat Jar" genannt. Der Vorteil dieser Technik liegt darin, dass nur eine Datei installiert wird. Es muss nicht darauf geachtet werden, dass weitere Bundles – in der richtigen Version – vorhanden sind. Ein Nachteil ergibt sich für Abhängigkeiten, die sich häufig ändern, und die man in weiteren Plug-ins verwendet: wenn eine Korrektur darin nötig ist, dann müssen mehrere Plug-ins ausgetauscht werden.



Information: Alle OSGi-Bundles von denen das Plug-in abhängig ist, müssen beim Veröffentlichen mitgeliefert und in eines der Verzeichnisse aus `osgiPluginDirectories` kopiert werden, damit der ELO Indexserver sie findet. (Siehe *2.7.2 Plug-in-Verzeichnis als Option des ELO Indexservers einstellen*)

2.6.1 Abhängiges Bundle referenzieren

Im Eclipse können diejenigen Bundles als Abhängigkeiten angegeben werden, die im Workspace oder in einem Verzeichnis der Target-Platform enthalten sind.

- *MANIFEST.MF* des Plug-ins öffnen

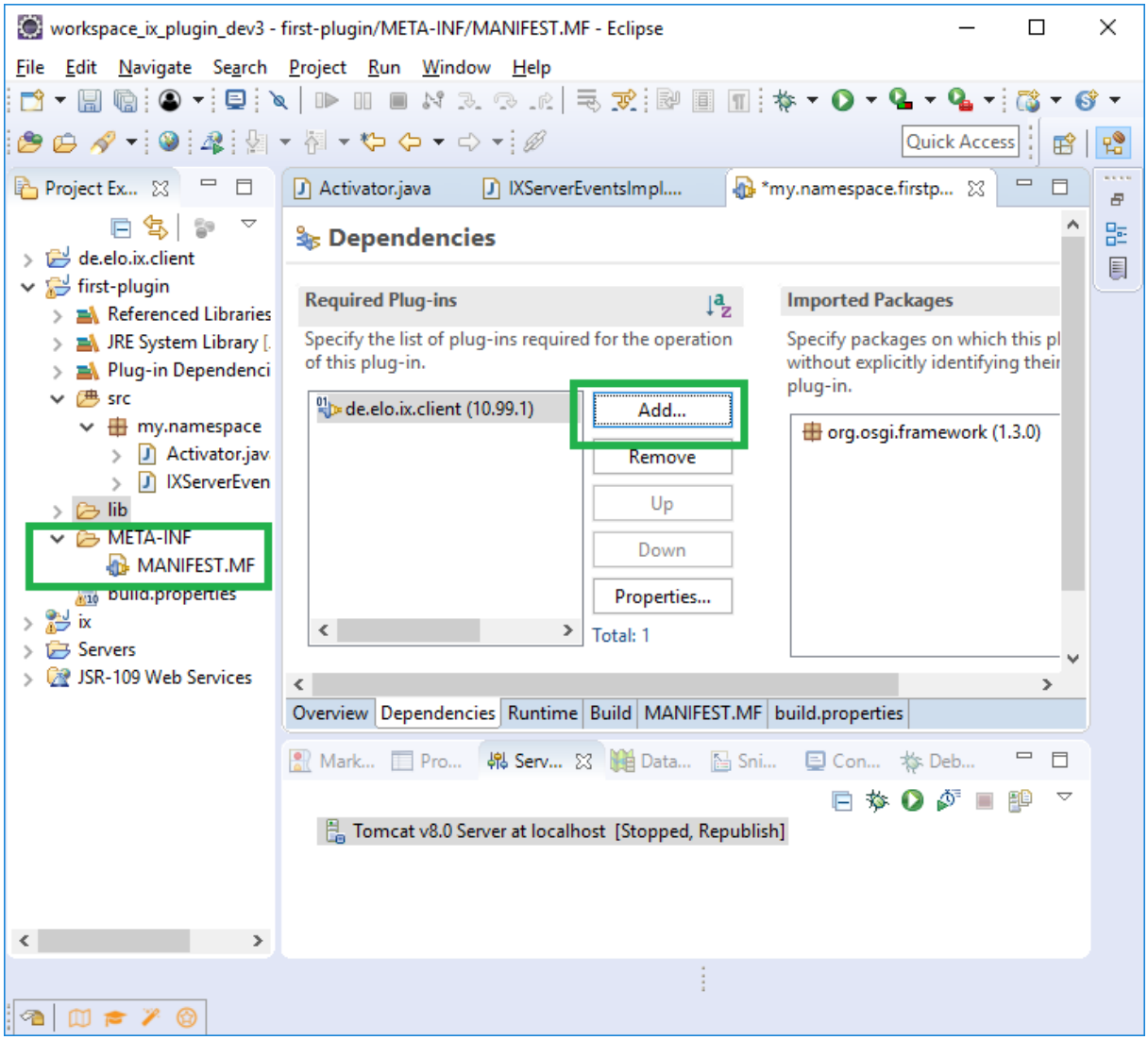


Abb. 57: Hinzufügen eines abhängigen Bundles

Wenn das Bundle nicht in der Target-Platform oder im Workspace vorhanden ist, gehen Sie wie folgt vor:

- Kopieren Sie alle benötigten Bundles in ein Verzeichnis.

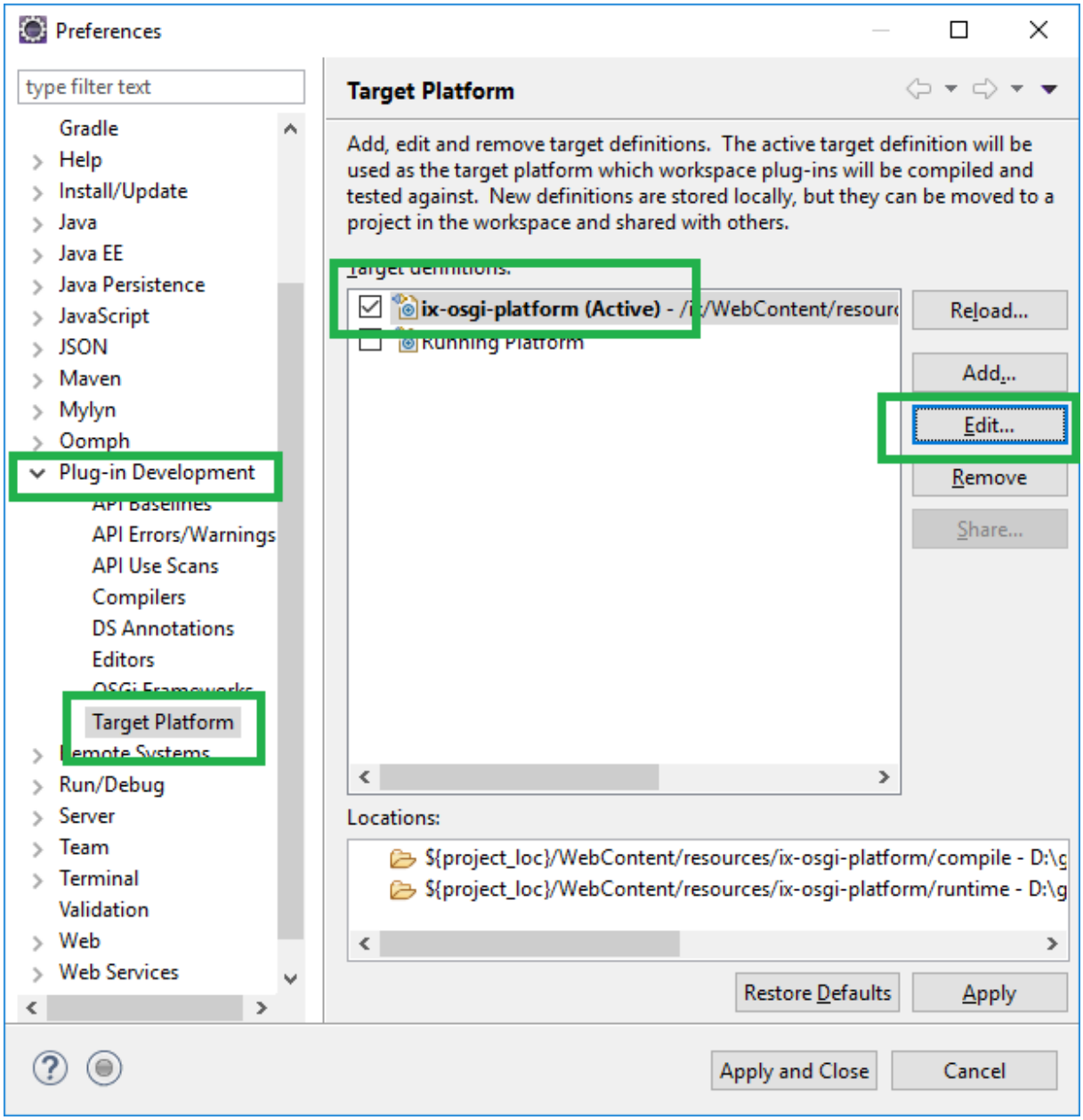


Abb. 58: Öffnen des Menüpunkts 'Target Platform'

- Öffnen Sie *Window* > *Preferences…* > *Plug-in Development* > *Target Platform*.

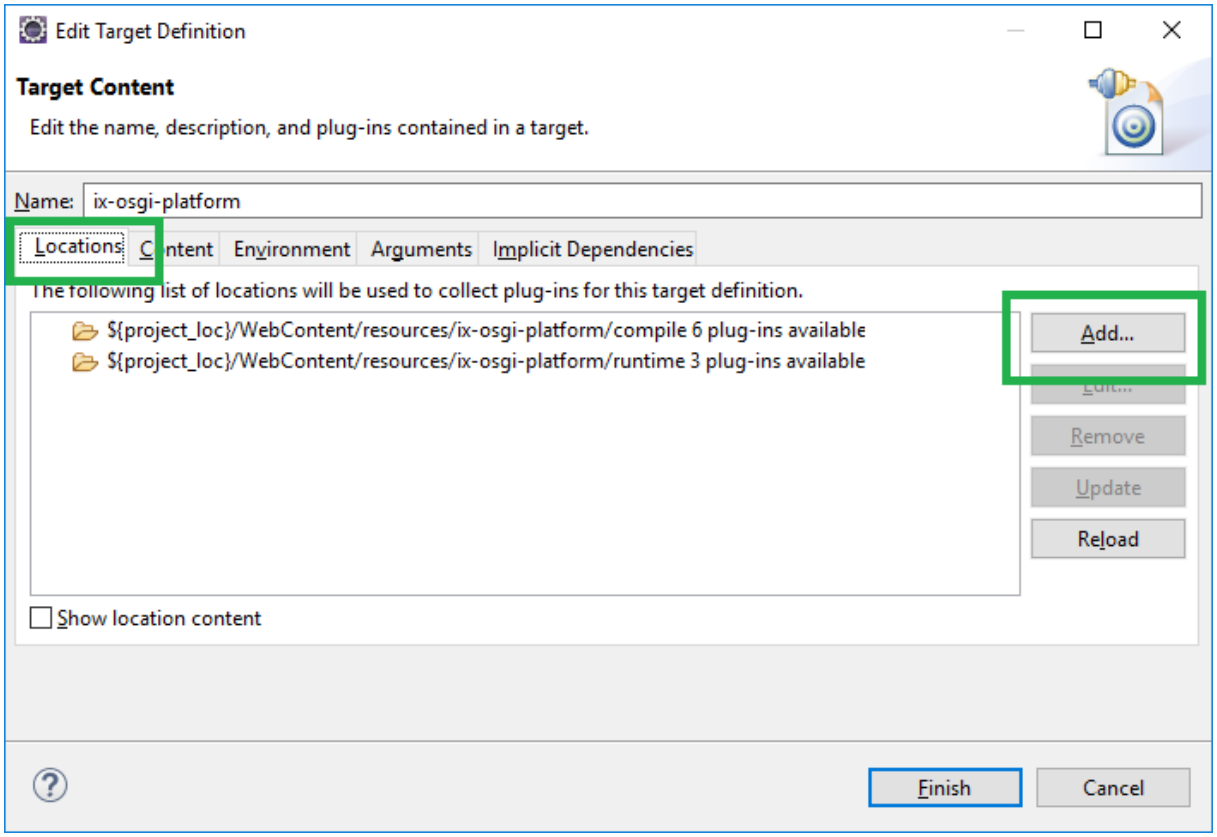


Abb. 59: Tab 'Location'

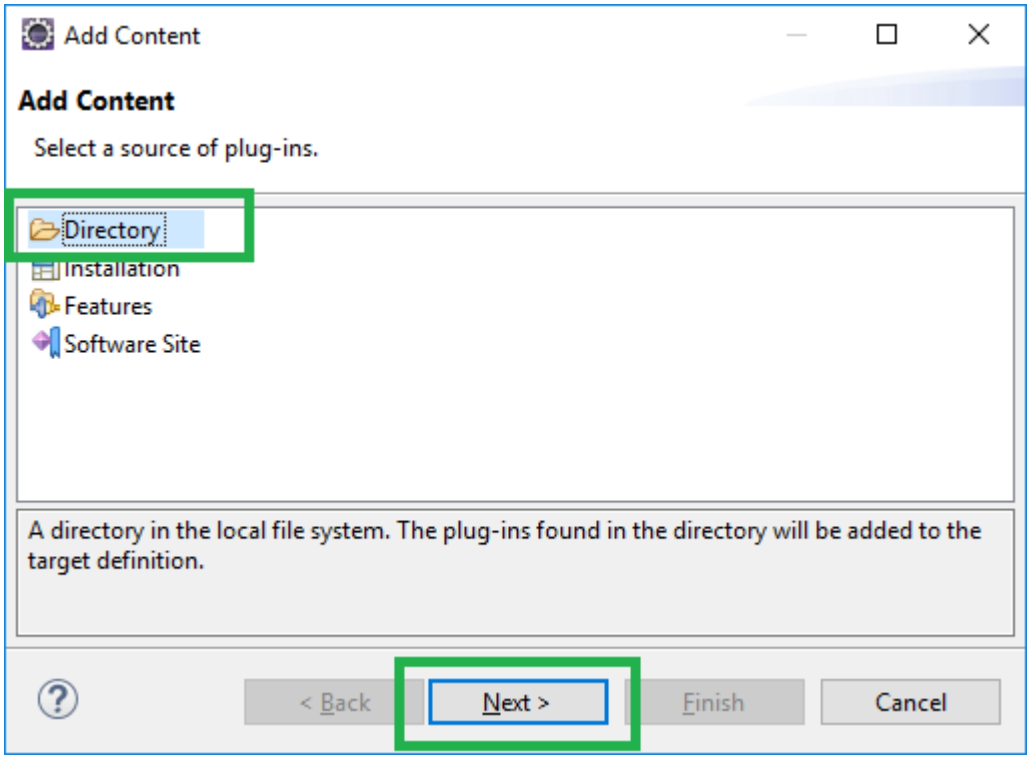


Abb. 60: Menüpunkt 'Directory'

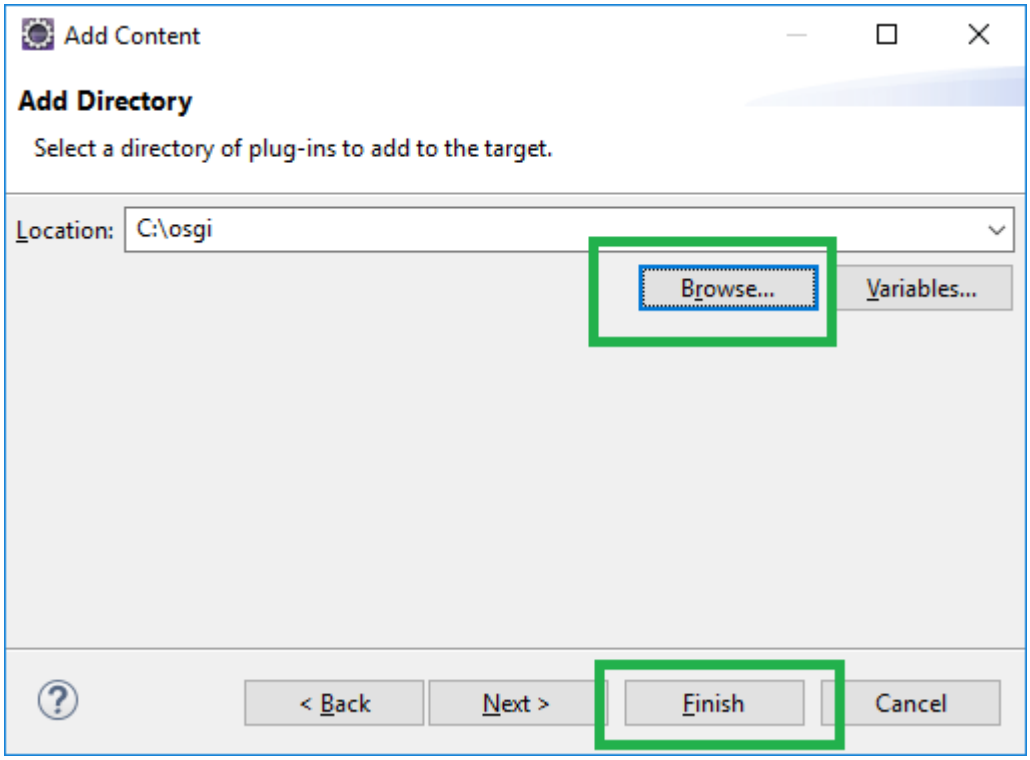


Abb. 61: Auswahl des Verzeichnisses

2.6.1.1 JAR-Bibliothek in OSGi-Bundle einpacken

Benutzt das Plug-in eine Java-Bibliothek, die nicht bereits ein OSGi-Bundle ist, kann sie mittels Eclipse mit den folgenden Schritten in ein OSGi-Bundle eingepackt werden.

- Eclipse > Menü > File > New > Other…

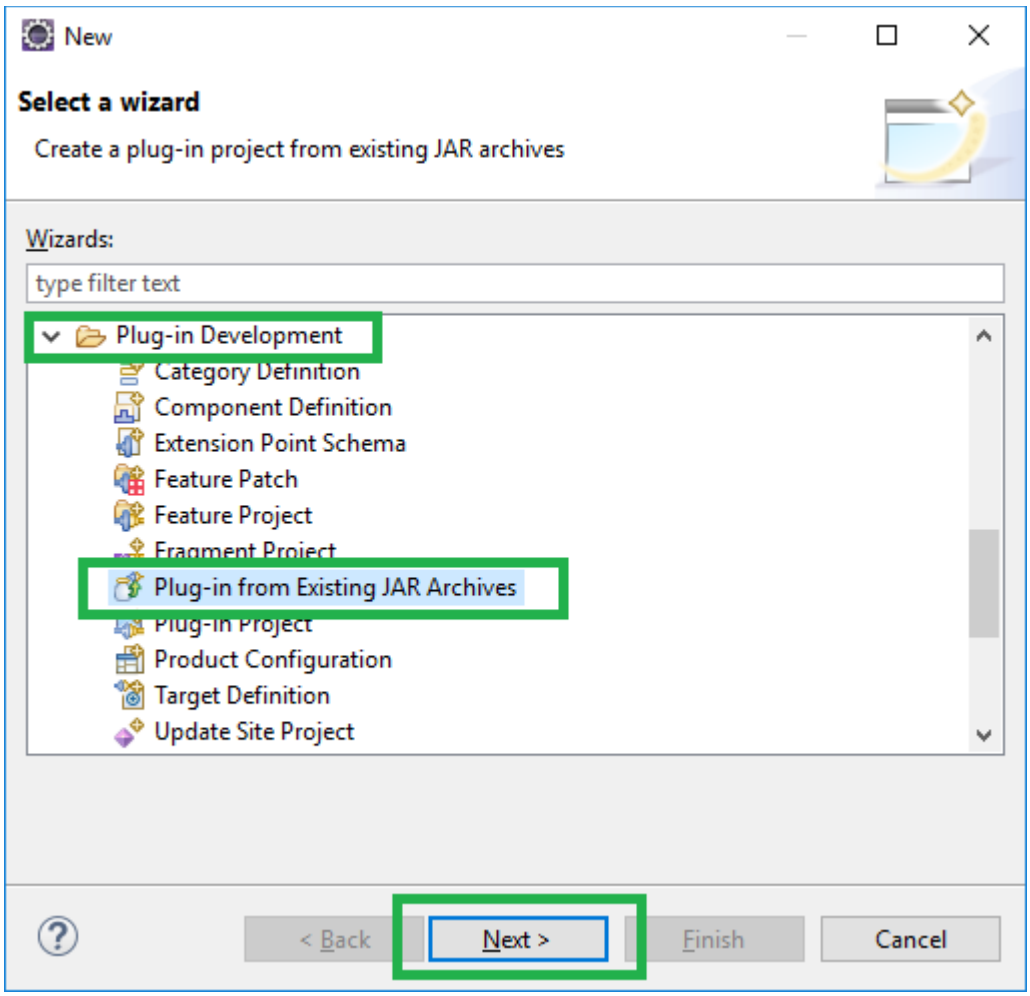


Abb. 62: Auswahl des passenden Assistenten

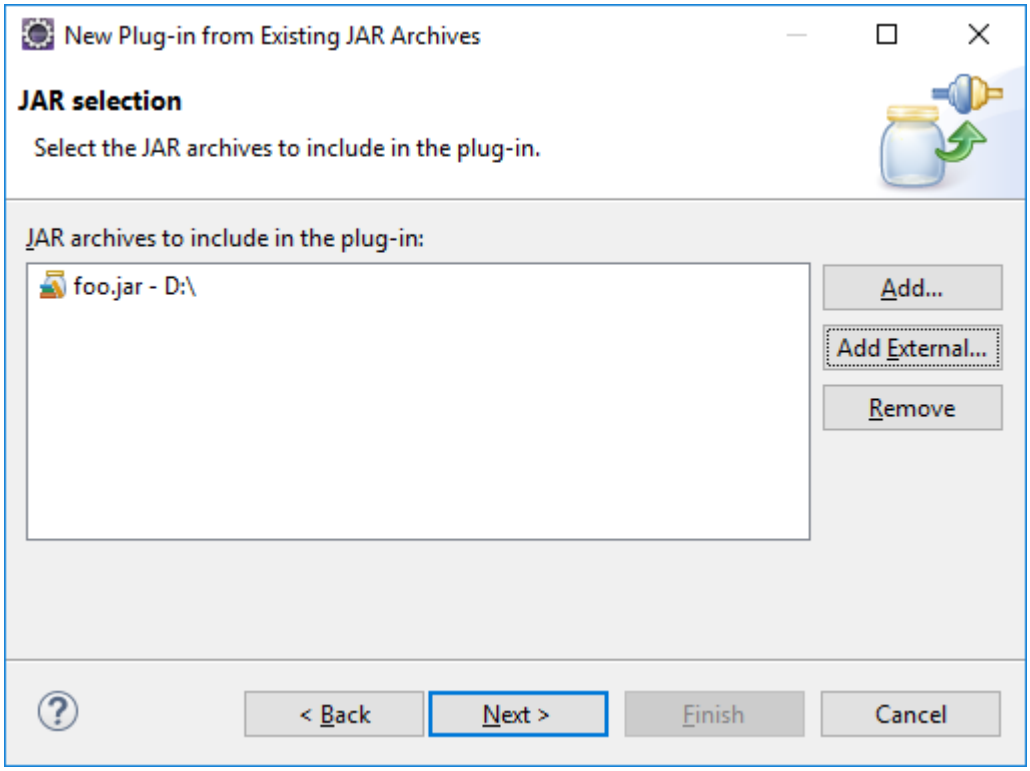


Abb. 63: Auswahl des JAR-Archivs

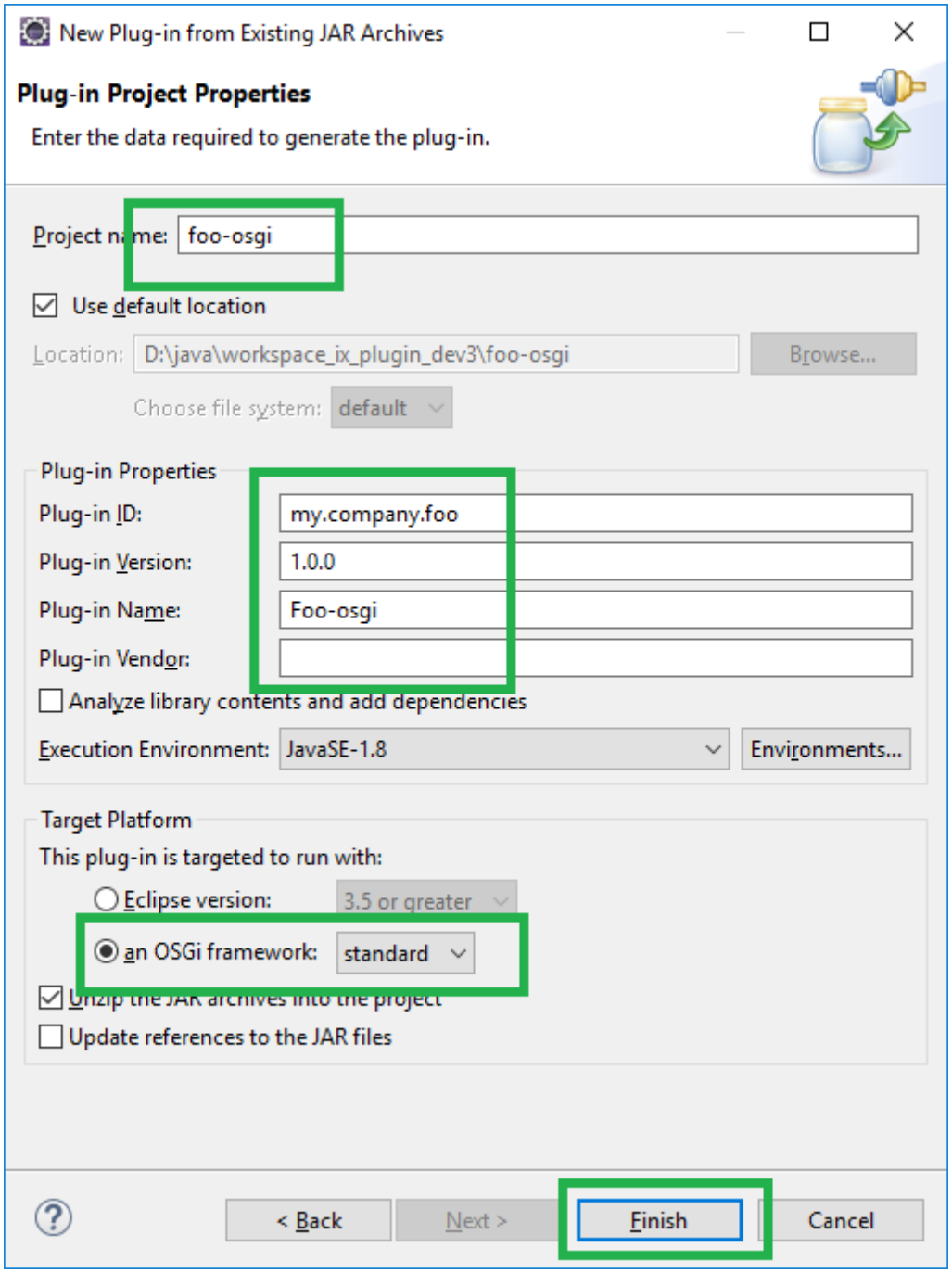


Abb. 64: Einstellungen für das neue Plug-in

- *MANIFEST.MF* des erstellten Bundles öffnen und Packages exportieren, die anderen Bundles zur Verfügung gestellt werden sollen.

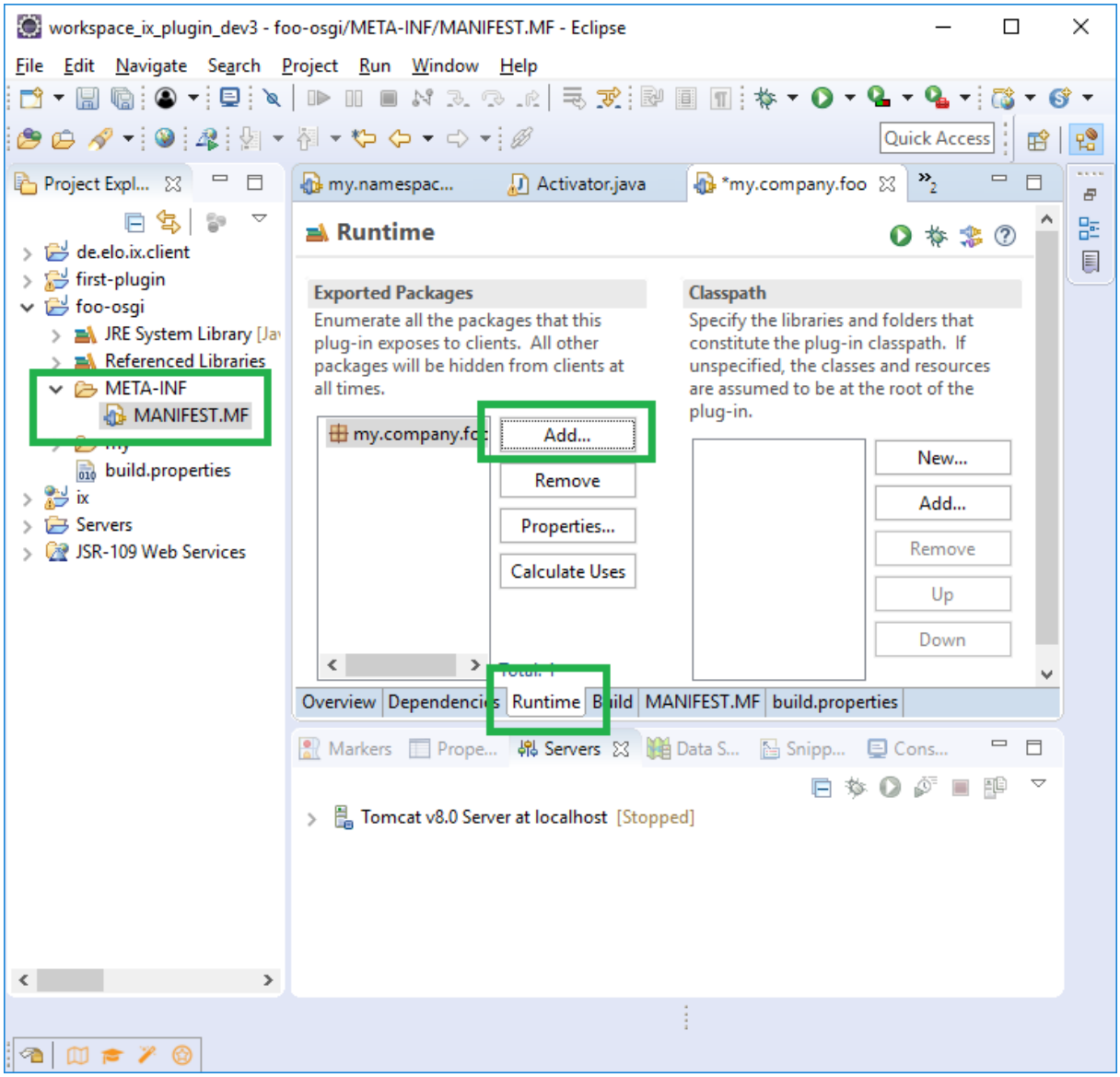


Abb. 65: Bearbeiten der Datei 'MANIFEST.MF'

- Mit STRG + S die *MANIFEST.MF* speichern
- Aus dem generierten Bundle-Projekt kann mithilfe des Export-Assistenten analog zu 2.7.3 eine JAR erstellt werden.
- *Eclipse > Menü > File > Export…*

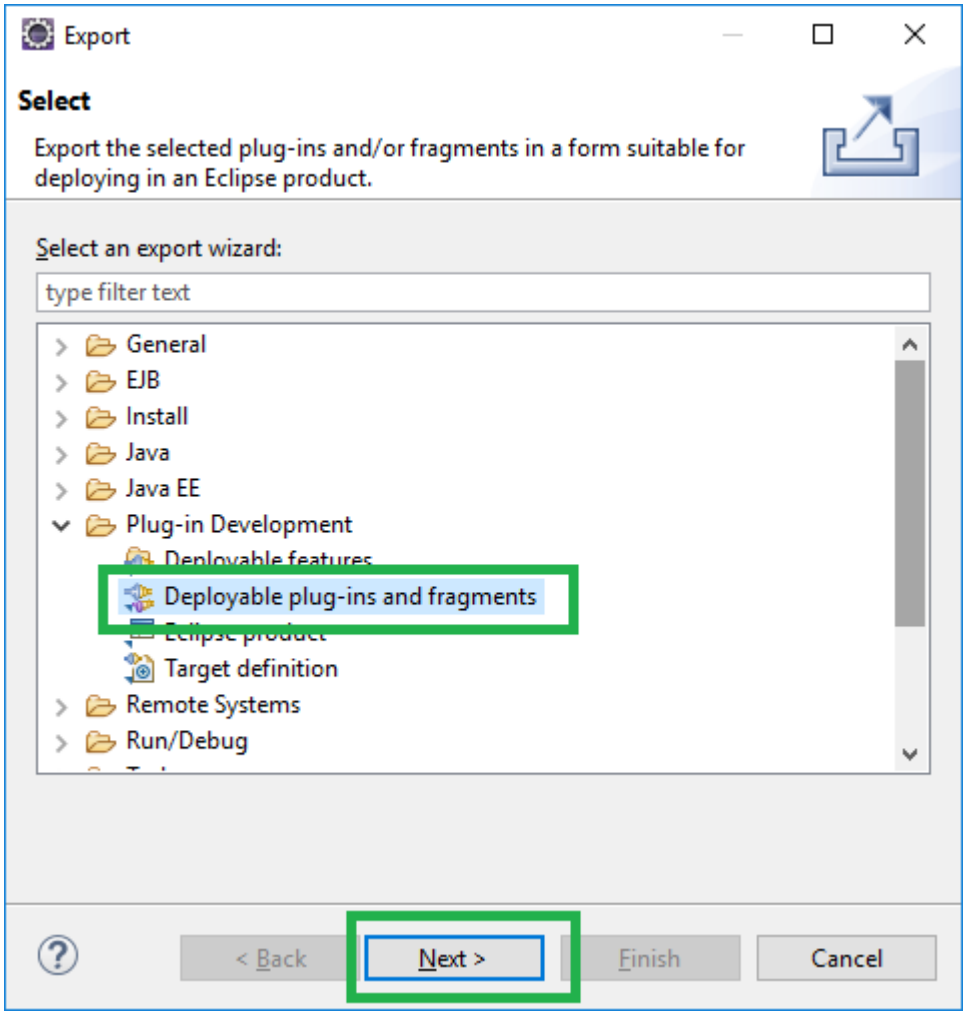


Abb. 66: Auswahl des Assistenten für den Export

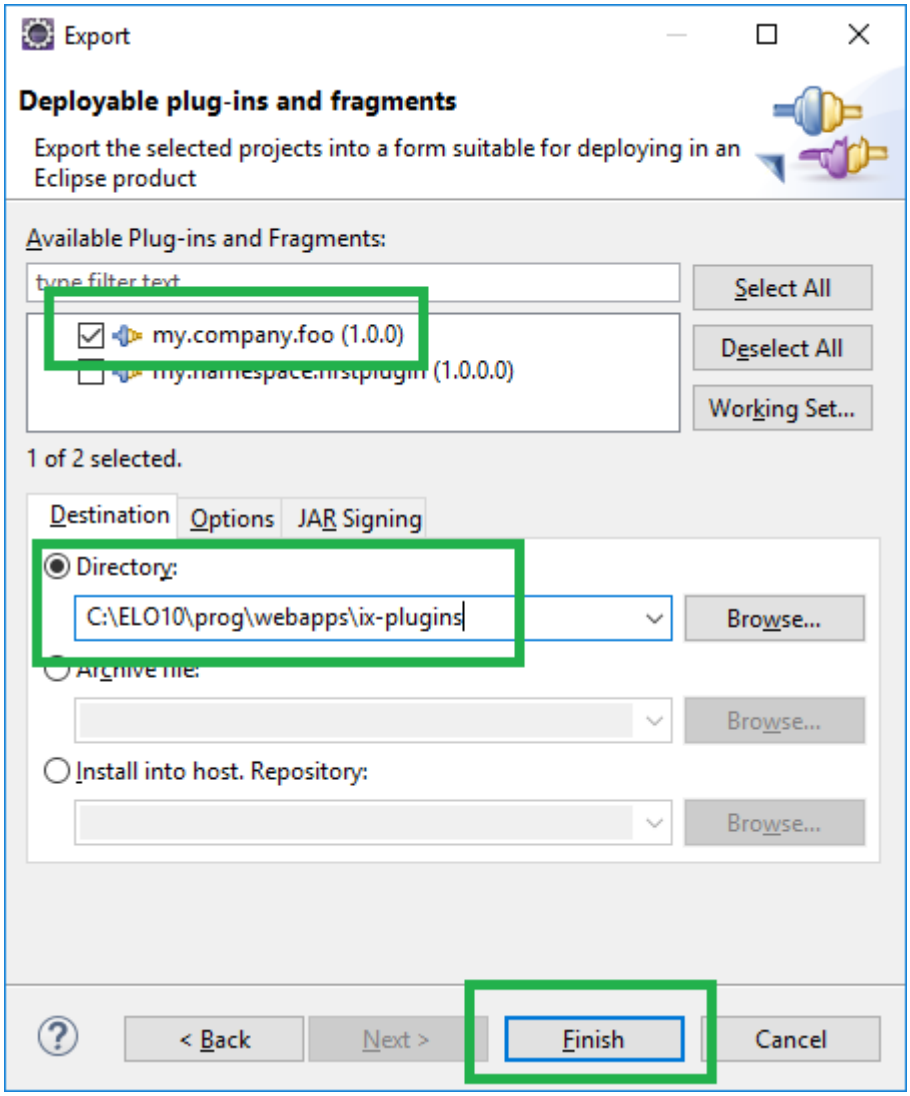


Abb. 67: Einstellungen für den Export

2.6.2 JAR-Bibiliothek in das Plug-in einbetten

Wenn eine Java-Bibliothek nur von einem oder wenigen Plug-ins benutzt wird, bietet es sich an, die JAR in das Plug-in einzufügen. Hierfür muss die JAR im Projektordner verfügbar sein.

- *Eclipse* > Kontextmenü zum Menüpunkt *first-plugin* > *New Folder*

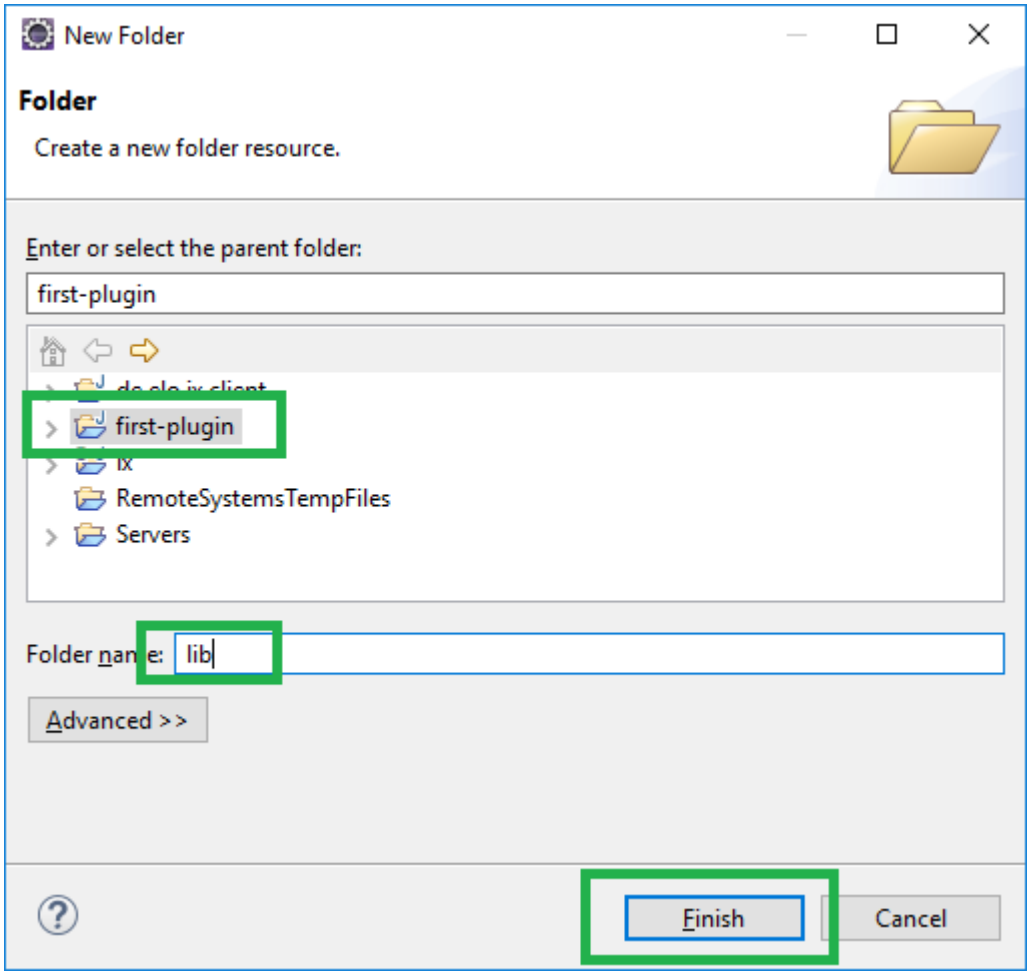


Abb. 68: Ordner 'lib' erstellen

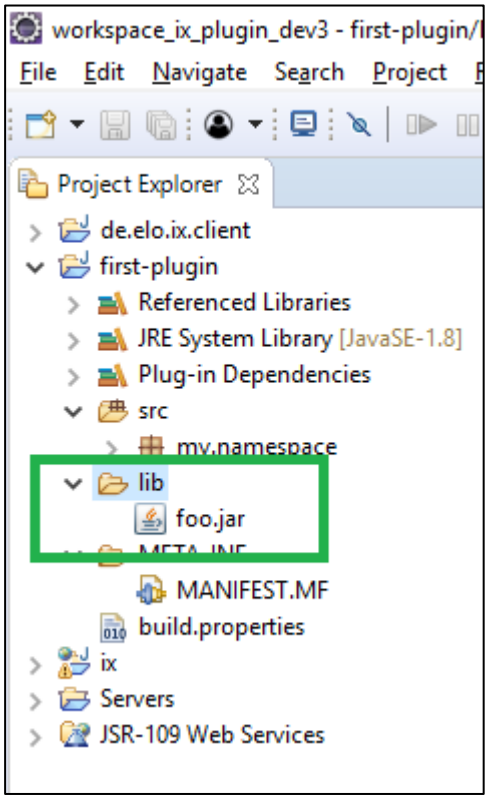


Abb. 69: JAR-Datei im Ordner 'lib'

- JAR-Datei z. B. per Drag-and-drop in den neuen Ordner *lib* kopieren.

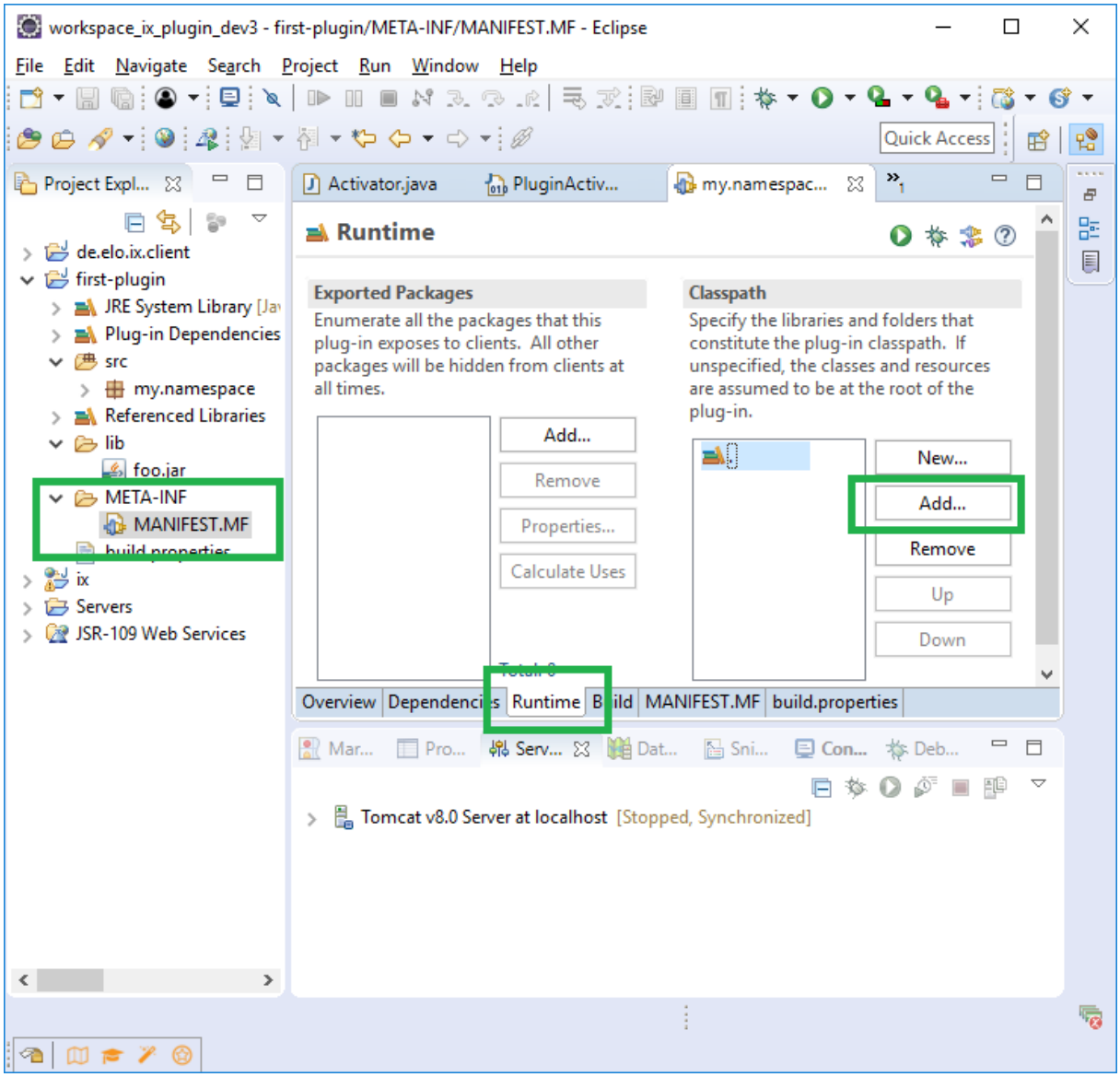


Abb. 70: Bereich 'Classpath'

- JAR-Datei in den Classpath des Bundles einfügen.

Ergebnis:

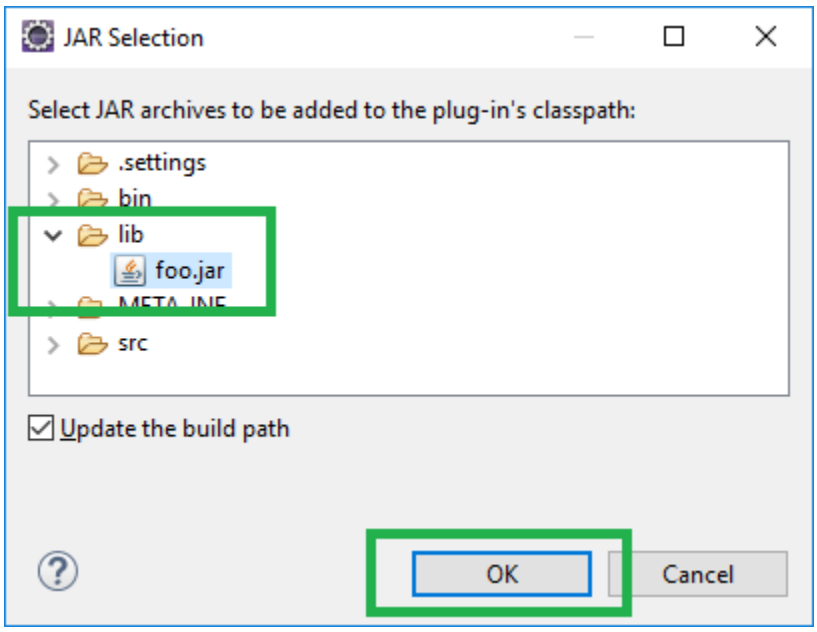


Abb. 71: Hinzufügen der JAR-Datei zum Classpath

- *MANIFEST.MF* speichern

2.7 Plug-in fertigstellen

Wenn die Entwicklung des Plug-ins abgeschlossen ist, kann es als OSGi-Bundle aus dem Workspace exportiert werden.

2.7.1 Vorbereitung

Vor dem Fertigstellen sollten diese Punkte beachtet werden:

> **Beachten Sie:** Das in 2.5.1 in den Classpath eingefügte Verzeichnis *bin* muss aus dem Classpath entfernt werden.

> **Beachten Sie:** Die Versionsnummer des Plug-ins sollte vor jedem Exportvorgang erhöht werden (*MANIFEST.MF* > *Overview* > *Version*).

2.7.2 Plug-in-Verzeichnis als Option des ELO Indexservers einstellen

Die Verzeichnisse, in denen die zu ladenden Plug-ins enthalten sind, werden dem ELO Indexserver mit der Option `osgiPluginDirectories` mitgeteilt (siehe *2.2.1 Zusätzliche IX-Optionen*). Es handelt sich um eine durch Semikolon getrennte Liste von Verzeichnissen. Sie werden rekursiv nach Plug-ins durchsucht.

```
Instance: DEBUG
Name: osgiPluginDirectories
Value: C:\ELO11\prog\webapps\ix-plugins
```

2.7.3 Plug-in exportieren

Für das Erstellen einer OSGi-JAR-Datei stellt Eclipse einen Assistenten bereit:

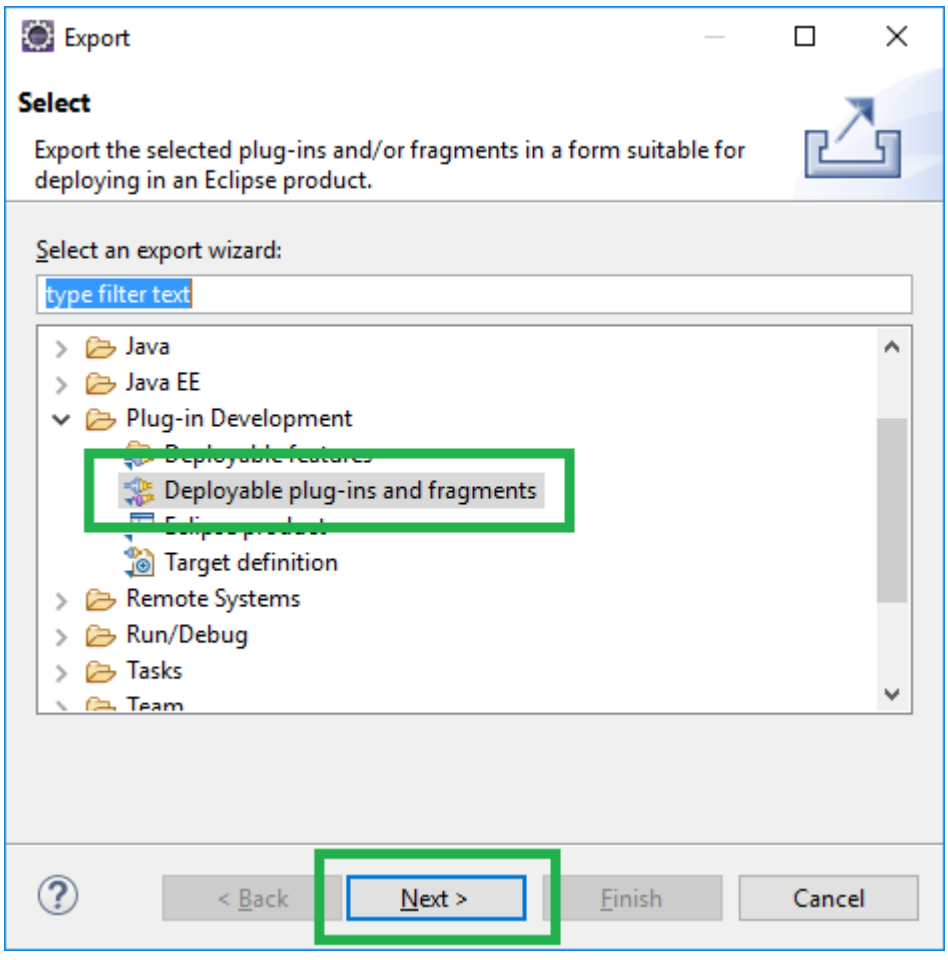


Abb. 72:Auswahl des Assistenten für den Export

- *Eclipse > Menü > File > Export…*

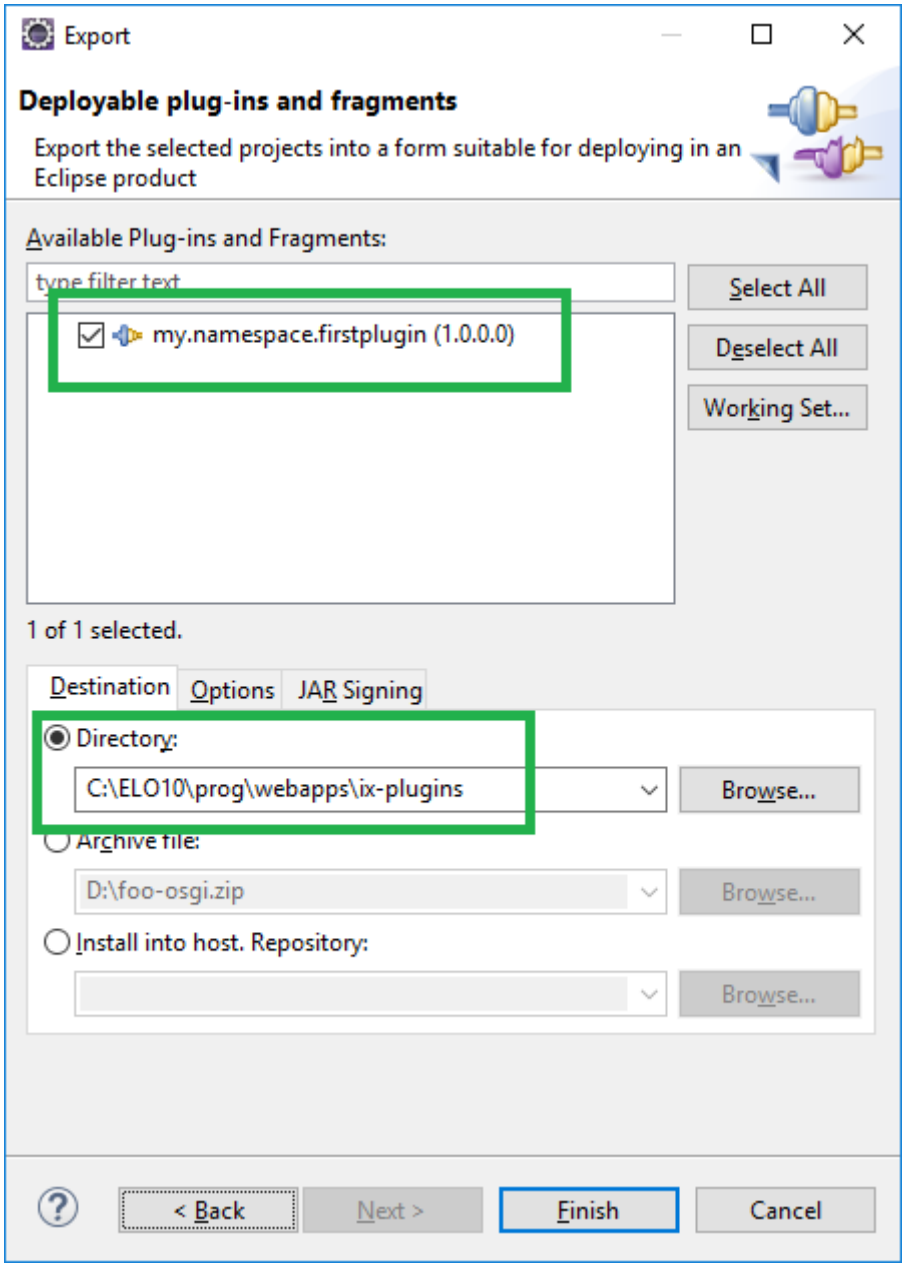


Abb. 73: Auswahl des Verzeichnisses

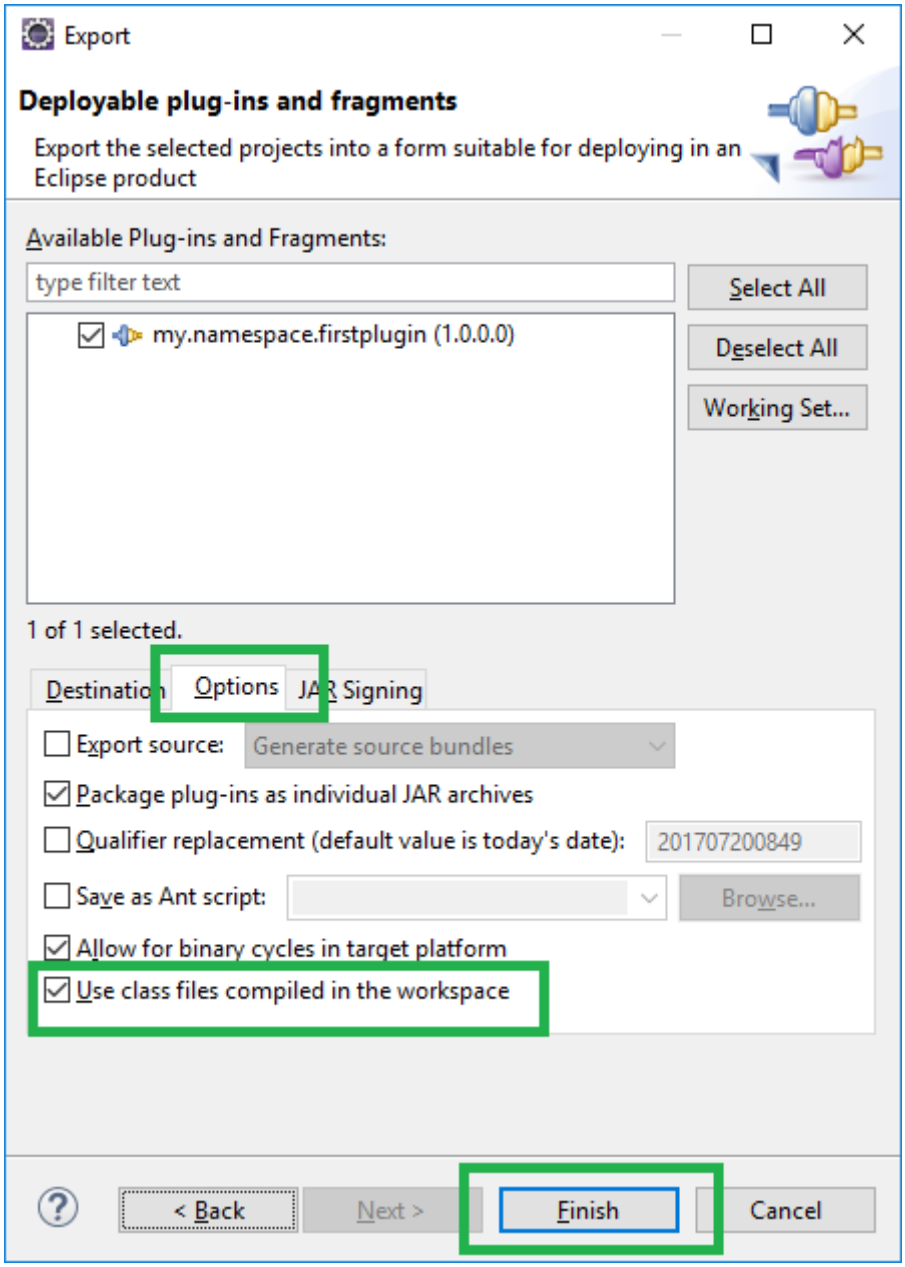


Abb. 74: Auswahl der Option 'Use class files compiles in the workspace'

Als Zielverzeichnis ist hier ein in der IX-Option `osgiPluginDirectories` angegebenes Verzeichnis zu wählen. Implizit ergänzt Eclipse ein weiteres Unterverzeichnis mit dem Namen *plugins* und speichert dort das Plug-in.

3 Services eines Plug-ins

Im OSGi-Framework stellt ein Bundle eine Interface-Implementierung als Service bereit. In der `Activator`-Klasse werden diese Services mit `registerService` im Framework angemeldet.

Der ELO Indexserver interessiert sich für die folgenden Services:

`de.elo.ix.client.plugin.InitializePlugin`	Initialisierungsfunktion, die während der Startphase des ELO Indexservers aufgerufen wird. (Siehe *3.1.Plug-in-Service InitializePlugin*)
`de.elo.ix.client.plugin.ConfigurePlugin`	Erlaubt die Konfiguration des Plug-ins über die API des ELO Indexservers (siehe `PluginService.configure)`
`de.elo.ix.client.plugin.VersionHistory`	Gibt die Versionsgeschichte des Plug-ins als Text zurück.
`de.elo.ix.client.IXServerEvents`	Allgemeine Ereignisse wie `onBeforeCheckinSord`, … (Siehe *2.4.3. Schnittstelle für Server-Ereignisse IXServerEvents implementieren*)
`de.elo.ix.client.plugin.WorkflowNodeEventsFactory`	Stellt ein Objekt bereit, dass Workflow-Knoten-Ereignisse verarbeitet. (Siehe *3.2. Plug-in-Service WorkflowNodeEventsFactory*)
`de.elo.ix.client.plugin.DynamicKeywordEventsFactory`	Stellt ein Objekt bereit, dass eine dynamische Stichwortliste füllt. (Siehe *3.3. Plug-in-Service ‚DynamicKeywordFactory*)
`javax.servlet.Servlet`	Verarbeiten von HTTP-Aufrufen. (Siehe *3.6. HTTP-Anfragen im Plug-in verarbeiten*)

Ein Beispiel für die Registrierung von Services:

```
public class Activator extends PluginActivator {

  public void start(BundleContext context) throws Exception {
```

```
    super.start(context);

    registerService(IXServerEvents.class,
                            new IXServerEventsImpl(this));

    registerService(DynamicKeywordEventsFactory.class,
                            new DynamicKeywordEventsFactoryImpl(this));

    registerService(WorkflowNodeEvents.class,
                            new WorkflowNodeEventsFactoryImpl(this));
  }
}
```



Information: Die Methoden `start` und `stop` der `Activator`-Klasse können über die Indexserver-API ausgelöst werden. Man muss folglich damit rechnen, dass sie mehrmals während der Laufzeit des Indexservers aufgerufen werden.



Information: Die Super-Klasse `PluginActivator` de-registriert in ihrer `stop`-Funktion alle mit `registerService` angemeldeten Dienste. Die eigenen Activator-Klasse kann folglich ohne `stop`-Funktion auskommen.

3.1 Plug-in-Service InitializePlugin

Während der Startphase ruft der ELO Indexserver für alle Plug-ins die `initialize`-Funktion der Schnittstelle `InitializePlugin` auf:

- nachdem alle internen Verwaltungsobjekte initialisiert sind
- bevor JavaScripte geladen und Aufrufe von außen zugelassen werden.

Innerhalb der Funktion können bereits API-Funktionen des ELO Indexservers über das administrative Verbindungsobjekt verwendet werden.

Beispiel:

```
import java.util.HashMap;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import byps.RemoteException;
import de.elo.ix.client.plugin.InitializePlugin;
import de.elo.ix.client.plugin.PluginActivator;

public class InitializePluginImpl implements InitializePlugin {

  private static Logger log = LoggerFactory.getLogger(
                                    InitializePluginImpl.class);
  private PluginActivator activator;

  public InitializePluginImpl(PluginActivator activator) {
    this.activator = activator;
  }

  @Override
  public void initialize(HashMap<String, String> indexserverOptions,
                         String webappConfigDir,
                         String webappDataDir) throws RemoteException {
    log.info("initialize(");
    log.info("webappConfigDir=" + webappConfigDir);
    log.info("webappDataDir=" + webappDataDir);


    IXConnection connA = activator.getIxConnectAdmin();
    EditInfo ed = connA.ix().createSord("1", "", EditInfoC.mbSord);
    …

    log.info(")initialize");
  }
}
```

3.2 Plug-in-Service WorkflowNodeEventsFactory

Ereignisse zu Workflow-Knoten können im Plug-in in `WorkfowNodeEvents`-Objekten abgefangen werden. Diese Objekte sind über die `create`-Methode des `WorkflowNodeEventsFactory`-Service dem ELO Indexserver bereitzustellen. Für jeden Ereignisaufruf aus dem ELO Indexserver wird mit `create` ein `WorkflowNodeEvents`-Objekt angefordert.

In der Workflow-Vorlage gibt man den symbolischen Namen des Plug-ins als Start- oder Ende-Skript ein. An den Namen kann ein „Pfad“ beginnend mit Schrägstrich angehängt werden, der der Funktion `create` übergeben wird. Auf diese Art kann ein Plug-in für verschiedene Knoten verschiedene Aufgaben erledigen.

Beispiel für die Zuweisung des Plug-in-Service an einen Workflow-Knoten:

```
WFDiagram wf = …
// Assign plugin event: Bundle-SymbolicName + path
wf.nodes[1].onEnter = "com.partner.invoice-solution/wf-incoming";
```

Beispiel für Factory-Objekt:

```
public class WorkflowNodeEventsFactoryImpl implements
                                              WorkflowNodeEventsFactory {
  private final Activator activator;

  public WorkflowNodeEventsFactoryImpl(Activator activator) {
    this.activator = activator;
  }

  @Override
  public WorkflowNodeEvents create(IXServerEventsContext ec, String path)
    throws RemoteException {
    WorkflowNodeEvents impl = null;
    if (path.equals("/wf-incoming")) {
       impl = new WorkflowNodeEventsImpl(activator);
    }
    else if (path.equals(…)) {
       impl = …
    }
    return impl;
  }
}
```

3.3 Plug-in-Service ‚DynamicKeywordFactory‘

Innerhalb eines Aufrufs von `checkoutKeywordsDynamic` wird über die `create` Methode dieses Service ein Objekt mit der Schnittstelle `DynamicKeywordEvents` zum Aufbau einer dynamischen Stichwortliste angefordert.

Der symbolische Name des Plug-ins wird in der Verschlagwortungsmaske als *DocMaskLine.serverScriptName* angegeben. An den Namen kann ein „Pfad“ beginnend mit Schrägstrich angehängt werden, der der Funktion `create` übergeben wird. Auf diese Art kann ein Plug-in für unterschiedliche Verschlagwortungsmasken verschiedene Aufgaben erledigen.

Beispiel für Zuweisung an ein Indexfeld:

```
DocMask dm = …
DocMaskLine dml = dm.lines[0];
dml.serverScriptName = "com.partner.invoice-solution/kw-incoming";
```

Beispiel für eine Factory-Implementierung:

```
public class DynamicKeywordEventsFactoryImpl implements
                                        DynamicKeywordEventsFactory {
  private Activator activator;

  public DynamicKeywordEventsFactoryImpl(Activator activator) {
    this.activator = activator;
  }

  @Override
  public DynamicKeywordEvents create(IXServerEventsContext ec, String path)
    throws RemoteException {
    DynamicKeywordEvents impl = null;
    if (path.equals("/kw-incoming")) {
        impl = new DynamicKeywordEventsImpl(activator);
    }
    else if (path.equals(…)) {
        impl = …
    }
    else {
        activator.getScriptObjectFactory(ec).throwException(
            IXExceptionC.INVALID_PARAM, "Unsupported path=" + path);
    }
    return impl;
  }
}
```

3.4 Plug-in-Service im Skripting aufrufen

Angenommen, ein Plug-in stellt einen Service bereit, der in einem ELO Indexserver Ereignisskript verwendet werden soll. Wie kann das Skript den Service aufrufen?

Für diesen Fall stellt der ELO Indexserver im Skripting das implizite Objekt `ixPlugins` bereit. Den gewünschten Service erhält man mit `ixPlugins.getService`. Wobei der vollqualifizierte Name des Service-Interfaces angegeben werden muss, unter dem der Service im Plug-in registriert ist.

Beispiel:

Registrierte Funktion, in der ein Service aufgerufen wird:

```
function RF_AccessPluginService(ec, args) {

  // Service interface name
  var MY_SERVICE = "de.elo.ix.osgi.test.MyServiceForScripting";

  // Lookup service.
  // Do not store the returned object in a global variable.
  // Use the object only for a short time.
  var service = ixPlugins.getService(MY_SERVICE);

  // Call service function add()
  var sum = service.add(123, 456);
  …

}
```

Plug-in, das den Service implementiert:

```
public class Activator extends PluginActivator {
  public void start(BundleContext context) throws Exception {
    super.start(context);
    registerService(MyServiceForScripting.class,
                    new MyServiceForScripting());
  }
}
```

```
package de.elo.ix.osgi.test;

public class MyServiceForScripting {

  public int add(int a, int b) {
    return a + b;
  }
}
```

Information: Im Skripting muss die Klasse des Service nicht bekannt sein, weil die Javascript-Engine über Reflection auf alle Elemente und Funktionen des Service zugreift.

Information: Das von `ixPlugins.getService` zurückgegebene Objekt sollte möglichst nur für eine kurze Zeit in einer Variablen gespeichert werden. Denn das Plug-in könnte während der Laufzeit des ELO Indexservers entladen und durch eine neue Version ersetzt werden.

Information: Die Funktion `ixPlugins.getService` liefert nie einen null-Wert. Wenn kein Plug-in geladen ist, das den angegebenen Service implementiert, wird eine Ausnahme ausgelöst mit der Fehlerkennung „[ELOIX:5023]" (Nicht gefunden).

Information: Die Plug-ins sind bereits verfügbar, wenn der Toplevel der JavaScripte ausgeführt werden – also direkt nach dem Laden der JavaSkripte und bevor der ELO Indexserver in den Status *Running* übergeht. Damit ist es einerseits möglich, den Plug-ins weitere Startwerte per Skripting zu übergeben. Andererseits können globale Variablen im Toplevel der Skripte durch Werte aus den Plug-ins initialisiert werden.

3.5 Plug-in-Service in einem anderen Plug-in verwenden

Um in einem Plug-in den Service eines anderen Plug-ins zu erreichen, benötigt man einen `ServiceTracker`. Er wird in der `start`-Methode der `Activator`-Klasse erstellt und in einer Instanzvariablen des Plug-ins gespeichert. Das Service-Interface erhält man aus der Methode `ServiceTracker.getService`. Wird das Plug-in gestoppt, schließt man den `ServiceTracker` mit der Funktion `close`.

Im Quellcode der EloixClient-Bibliothek kann die Handhabung in der Klasse `PluginActivator` nachgesehen werden.

Weil es sich beim `ServiceTracker` um ein Standardkonzept von OSGi handelt, findet man auch im Internet einige Beispiele, bspw [9].

Information: Das `ServiceTracker`-Konzept ist nicht unbedingt notwendig, um Funktionalitäten eines anderen Plug-ins zu verwenden. Mit der Bundle-Option `Import-Package` oder `Require-Bundle` kann man direkt die Klassen eines anderen Plug-ins benutzen. Wenn die benötigte Funktion nur in einem bestimmten Plug-in vorhanden ist, stellt der direkte Verweis im Vergleich zum `ServiceTracker` die einfachere Lösung dar.

Information: Um ServiceTracker zu verwenden, muss das Plug-in das Package *org.osgi.util.tracker* importieren:

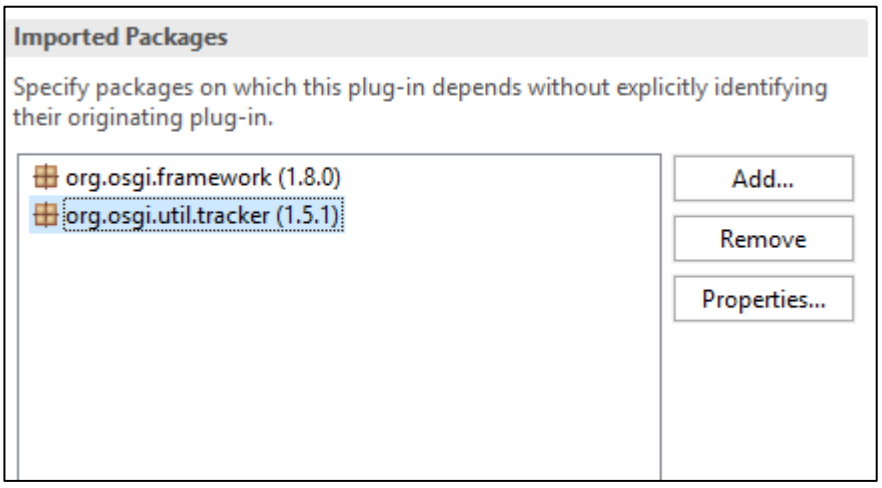


Abb. 75: Import von 'org.osgi.util.tracker'

3.6 HTTP-Anfragen im Plug-in verarbeiten

Ein Plug-in kann als Java-Servlet angesprochen werden, wenn es von `javax.servlet.HttpServlet` erbt und als OSGi-Service unter `javax.servlet.Servlet` registriert wird.

Das Plug-in ist unter verschiedenen URIs erreichbar, die sich in der Art der Authentifizierung unterscheiden. Innerhalb der URI wird das Plug-in entweder durch seinen eindeutigen Bundle-SymbolicName oder durch eine seiner registrierten Servcie-Interfaces spezifiziert. Letzteres ist nur dann erfolgreich, wenn genau ein Plug-in den angegebenen Service anbietet.

3.6.1 URI ohne Authentifizierung

```
/ix-archivename/plugin/symbolic-name-or-service/…
```

Über diese URI werden HTTP-Anfragen ohne vorige Authentifizierung gesendet. Beispielsweise könnte eine solche URI Statusinformationen zum Plug-in liefern. Ferner könnte sie ein Dokument zu einer in der Anfrage übergebene GUID zurückgeben. Weil eine GUID nicht erraten werden kann, würde dies kein Sicherheitsproblem darstellen.

3.6.2 URI mit Single-Sign-On

```
/ix-archivename/plugin-auth/symbolic-name-or-service/…
```

Bei Anfragen an diese URI wird eine Authentifizierung des Benutzers gegen die Betriebssystem-Domäne erzwungen. Das Authentifizierungsverfahren ist dasselbe wie beim Zugriff auf die API des ELO Indexservers (Basic, NTLM, Negotiate/Kerberos), und es gelten dieselben Voraussetzungen – bspw., dass der Benutzer in der Domäne angemeldet ist.

Der ELO Indexserver stellt nach erfolgreicher Anmeldung über

```
PluginActivator.getIxConnect(HttpSevletRequest)
```

eine `IXConnection` mit den Berechtigungen des Benutzers bereit. Weil sie in der Tomcat-Session gespeichert ist, sollte sie vom Plug-in geschlossen werden, bevor die HTTP-Session ausläuft.

3.6.3 URI mit Tomcat-Authentifizierung

```
/ix-archivename/manager/plugin/symbolic-name-or-service/…
```

Diese URI erfordert eine Basic-Authentifizierung gegen den Tomcat-Manager. Bei jeder Anfrage muss der Authentifizierungs-Header und das Session-Cookie mitgesendet werden, was Browser üblicherweise automatisch erledigen.

Dem Plug-in wird von `PluginActivator.getIxConnect(HttpSevletRequest)` eine `IXConnection` mit administrativen Rechten bereitgestellt.

3.6.4 Servlet-Funktionen

Aus der Servlet-Schnittstelle des Plug-ins werden vom ELO Indexserver die folgenden Funktionen aufgerufen:

```
public void init(ServletConfig config) throws ServletException;

public void destroy();

public void service(HttpServletRequest req, HttpServletResponse res) throws
                    ServletException, IOException;
```



Information: Das Plug-in wird nicht, wie man vielleicht vermuten könnte, als eigenständiges Servlet in den Tomcat eingehängt. Dies erlaubt der Tomcat nur beim Hochfahren der Web-Anwendung (hier der ELO Indexserver), was bereits vor der Initialisierung des OSGi-Frameworks und damit vor dem Laden der Plug-ins geschieht.

3.6.5 Beispiel: Dokument zu gegebener GUID lesen

Das folgende Beispiel-Servlet liefert ein Dokument zu einer gegebenen Objekt-ID aus dem ELO Archiv aus, nachdem der Benutzer über eine SSO-Anmeldung authentifiziert wurde (3.6.2). Die Aufruf-URI könnte bspw. so aussehen:

```
http://localhost:9090/ix-elo110/plugin-
auth/de.elo.plugin.example/readdoc?objid=(D88E3872-6026-9FED-AB16-
F407363CB8E2)
```

Zur Vorbereitung fügen Sie in den Abhängigkeiten des Plug-ins (wie in Abschnitt *2.4.1 Neues Plug-in-Projekt erstellen*) für die Indexserver-Client-Bibliothek gezeigt, eine Referenz zu *javax.servlet* hinzu.

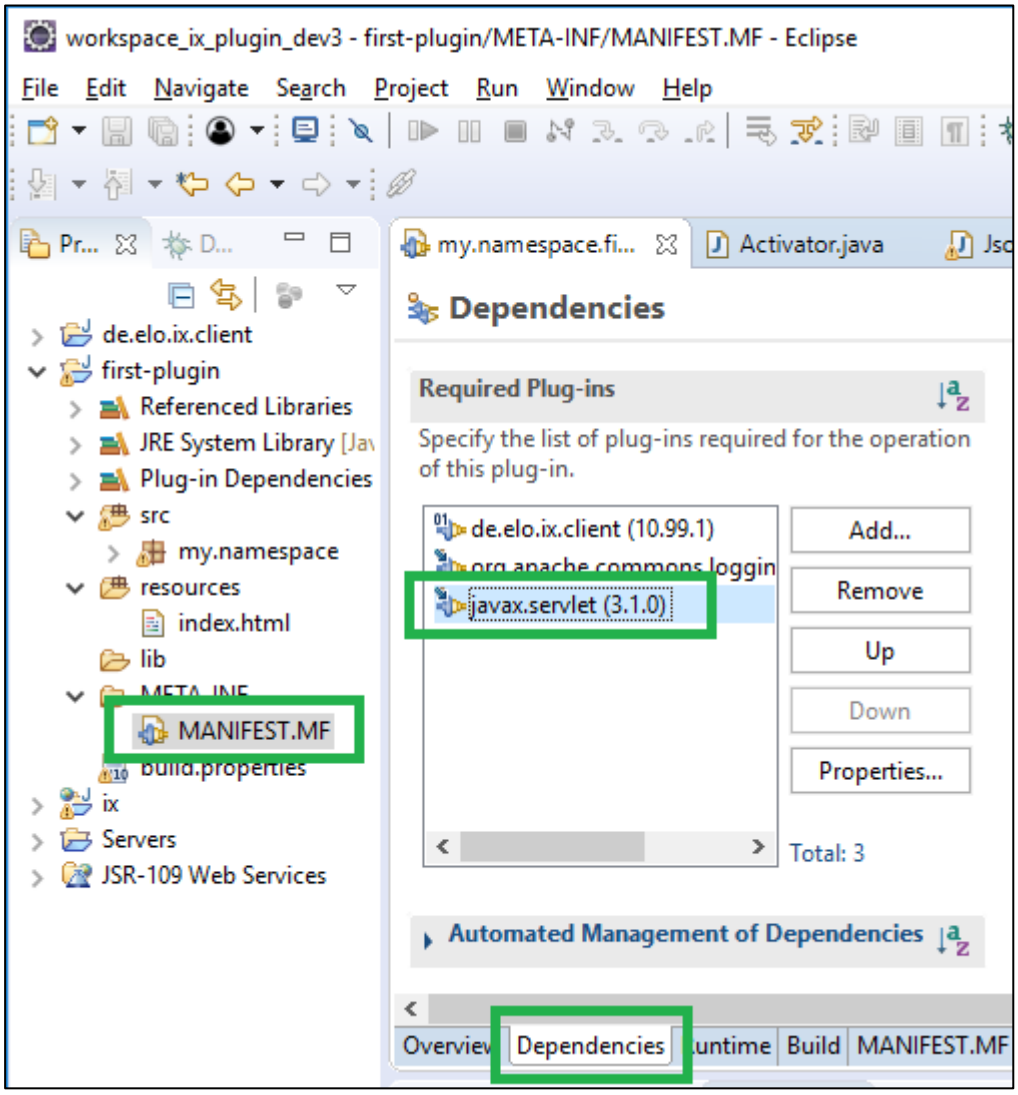


Abb. 76: Hinzufügen einer Referenz zu 'javax.servlet '

Erstellen Sie eine Klasse `ReadDocServlet` mit dem folgenden Code:

```
public class ReadDocServlet extends HttpServlet {

  private static final long serialVersionUID = 4119332353629228235L;

  PluginActivator activator;

  public ReadDocServlet(PluginActivator activator) {
    this.activator = activator;
  }

  @Override
```

```
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {

    // Check that request URI ends with /readdoc
    String uri = request.getRequestURI();
    String command = uri.substring(uri.lastIndexOf("/") + 1);
    if (!command.equals("readdoc")) throw new InvalidParameterException();

    // Get object ID from URI parameter.
    String objId = request.getParameter("objid");

    // Get IXConnection from authenticated user.
    IXConnection conn = activator.getIxConnect(request);

    // Get work version of document.
    SordZ sordZ = new SordZ(SordC.mbDocVersion|SordC.mbContentStream);
    Sord sord = conn.ix().checkoutSord(objGuid, sordZ, LockC.NO);
    DocVersion dv = sord.getDocVersion();

    // Set response headers: Content-Type, Content-Length
    response.setContentType(dv.getContentType());
    response.setHeader("Content-Length", Long.toString(dv.getSize()));

    // Copy document content to response stream.
    byte[] buf = new byte[10 * 1000];
    try (InputStream istream = dv.getFileData().getStream();
        OutputStream ostream = response.getOutputStream()) {
      int len = 0;
      while ((len = istream.read(buf, 0, buf.length)) != -1) {
        ostream.write(buf, 0, len);
      }
    }
  }
}
```

In der `Activator`-Klasse des Plug-ins wird das Servlet wie folgt registriert:

```
public class Activator extends PluginActivator {

  public void start(BundleContext context) throws Exception {
    super.start(context);
    registerService(Servlet.class, new ReadDocServlet(this));
  }
}
```



Information: Für dieses Beispiel ist eine Authentifizierung zwingend erforderlich, weil auch eine numerische ID übergeben werden kann. Andernfalls könnte ein Benutzer mit einer geratenen, numerischen ID ein Dokument erhalten, auf das er kein Leserecht besitzt.

Um das Beispiel so anzupassen, dass es ohne Authentifizierung auskommt, könnte man als Parameter nur eine Objekt-GUID zulassen. Unter dieser Voraussetzung könnte intern die administrative Verbindung `activator.getIxConnectAdmin()` verwendet werden.

3.6.6 Beispiel: Zusammenspiel mit JSON-API

Dieses Beispiel zeigt, wie ein Plug-in eine Single-Page-Web-Anwendung aufbauen kann, in der die JSON-API des ELO Indexservers eingesetzt wird.

Zur Vorbereitung ist (wie in Beispiel *3.6.4 Servlet-Funktionen*) eine Abhängigkeit zum Bundle *javax.servlet* einzurichten.

Erstellen Sie eine Klasse `JsonApiServlet` mit diesem Inhalt:

```
public class JsonApiServlet extends HttpServlet {
  private static final long serialVersionUID = 77096564761887475l3L;

  private final PluginActivator activator;

  public JsonApiServlet(PluginActivator activator) {
    this.activator = activator;
  }

  @Override
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    index(response);
  }

  /**
   * Copy index.html from resources to HTTP response.
   * @param response Servlet response
   * @throws IOException
   */
  private void index(HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    OutputStream ostream = response.getOutputStream();
    try (InputStream istream =
            getClass().getResourceAsStream("/index.html")) {
```

```
      byte[] buf = new byte[10*1000];
      int len = 0;
      while ((len = istream.read(buf)) != -1) {
        ostream.write(buf, 0, len);
      }
    }
  }
}
```

In der `Activator`-Klasse des Plug-ins wird das Servlet wie folgt registriert:

```
public class Activator extends PluginActivator {
  public void start(BundleContext context) throws Exception {
    super.start(context);
    registerService(Servlet.class, new JsonApiServlet(this));
  }
}
```

Die Startseite der Web-Anwendung befindet sich in einer statischen HTML-Datei, index.html, die im Plug-in enthalten ist und über eine HTTP-GET-Anfrage ausgeliefert wird. Ausgehend vom Plug-in-Projekt aus Abschnitt 2.4.1 wird dazu ein Ordner *resources* erstellt, in dem die index.html gespeichert wird.

Öffnen Sie das Kontextmenü zum Projekt *first-plugin* im Project Explorer. Wählen Sie *New* > *Folder.* Als *Folder name* geben Sie `resources` ein und bestätigen mit *Finish*.

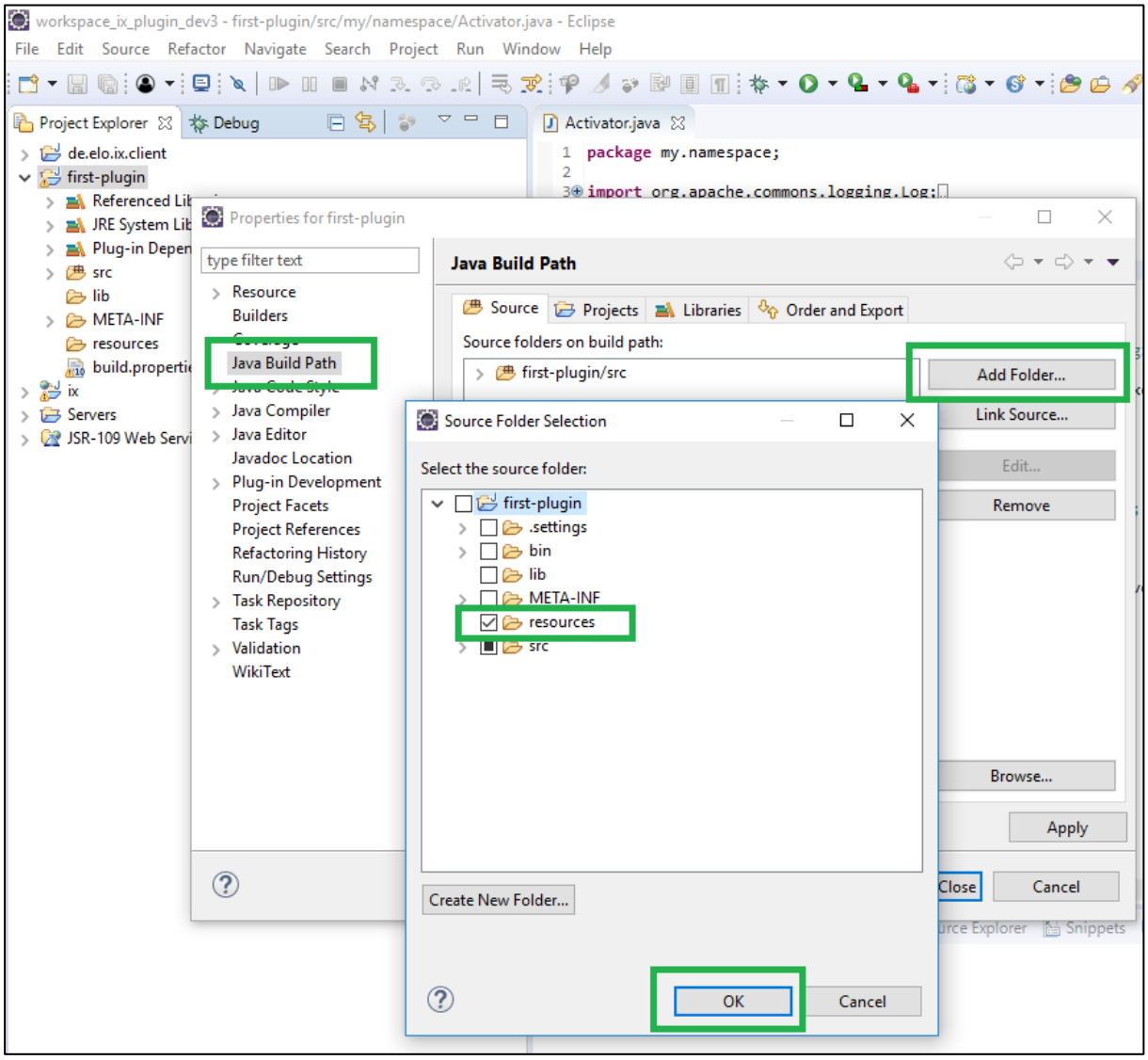


Abb. 77: Hinzufügen des Ordners 'resources'

Öffnen Sie erneut das Kontextmenü, wählen Sie *Properties*. Selektieren Sie links *Java Build Path* und rechts den Tab *Source*. Fügen Sie mit *Add Folder…* den neuen Ordner *resources* zum Build-Path hinzu.

Erstellen Sie im Ordner *resources* eine neue HTML-Datei: Kontextmenü zu *resources* > *New* > *Other…* > *Web* > *HTML File*. Als *File name* geben Sie `index.html` ein. Bestätigen Sie mit *Finish*.

Ersetzen Sie den Inhalt der Datei *index.html* mit dem folgenden Code:

```
<html>
<head>

<!-- Include Indexserver JSON API -->
<script src="../../EloixClient.js"></script>
```

```
<script type="text/javascript">

  // Initialize asynchronous processing using Promise objects.
  de.elo.ix.client.initPromiseDefaultImpl();

  // Simplify access to package de.elo.ix.client by alias IX
  var IX = de.elo.ix.client;

  // This promise will hold the IXConnection object.
  var _connPromise = null;

  // Build IX-URL /ix-archivename/ix and initialize connection factory.
  var url = [ "/", location.pathname.split("/")[1], "/ix" ].join("");
  var connFact = new IX.IXConnFactory(url, "JSON-API", "1.0");

  // Get or create the IXConnection object.
  function conn() {
    if (_connPromise) return _connPromise;

    // Create IXConnection.
    // Call cookieFactory to process authentication.
    var clientInfo = connFact.makeDefaultClientInfo();
    _connPromise = connFact.createFromCookie(clientInfo, null, null,
                                             cookieFactory);
    return _connPromise;
  }

  // Login
  function login() {
    conn().then(function(conn) {
      var CONST = conn.getCONST();
      conn.ix().createSord("", "", CONST.EDIT_INFO.mbSord)
         .then(function(ed) {
              alert("Connection from Cookie for user=" +
                     conn.getUserName());
      });
    });
  }

  // Logout
  function logout() {
    conn().then(function(conn) {
        conn.close();
    });
  }

  // Process authentication.
  // As result, the server sends a HTTP session cookie (JSESSIONID).
```

```
  var cookieFactory = function(cookieResult) {
    var uri = location.pathname.replace("/plugin/", "/plugin-auth/");
    xhr = new XMLHttpRequest();
    xhr.withCredentials = true;
    xhr.open("GET", uri, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState != 4) return;
        var ex = null;
        if (xhr.status != 200) {
          ex = new byps.BException(byps.BExceptionC.IOERROR,
                          "HTTP status " + xhr.status, xhr.responseText);
        }
        cookieResult(xhr.responseText, ex);
    };
    xhr.send();
  };

</script>
</head>
<body>
<input type="button" value="Login" onclick="login()" />
<input type="button" value="Logout" onclick="logout()" />
</body>
</html>
```

Starten Sie den ELOix. Geben Sie im Browser als Adresse ein: http://localhost:8080/ix/plugin/my.namespace.firstplugin/jsonapi. Anstelle von „jsonapi“ können Sie auch eine beliebige andere Zeichenfolge einsetzen. Denn sie wird in diesem Beispiel nicht ausgewertet. Sie erhalten die folgende Ansicht:

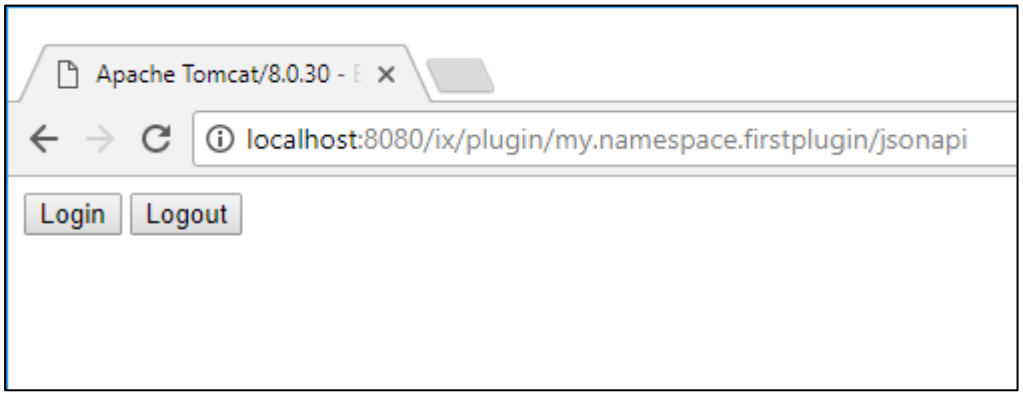


Abb. 78: Aufruf der Adresse im Browser

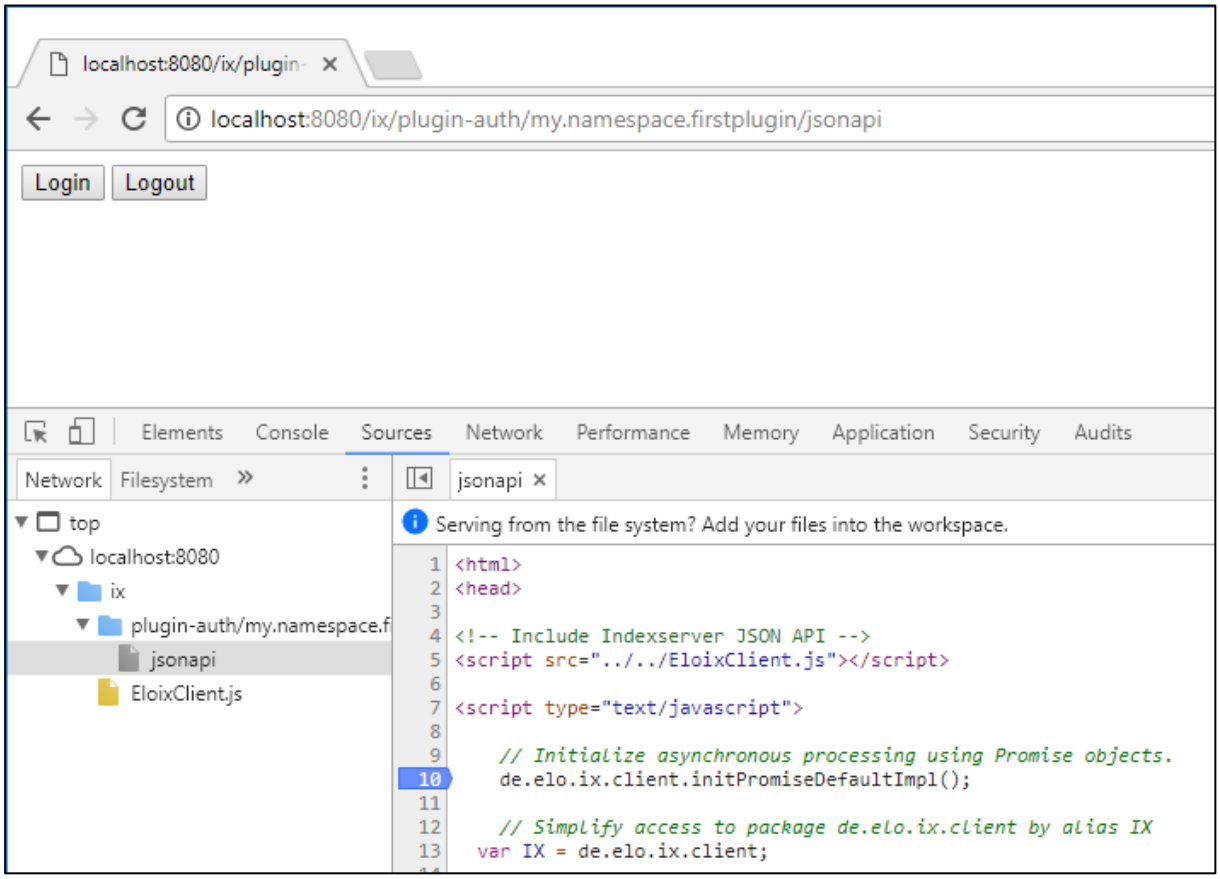


Abb. 79: Breakpoint

Öffnen Sie die Debugging Tools des Browsers, um den Programmfluss zu verfolgen. Den ersten Breakpoint setzten Sie in der Zeile `de.elo.ix.client.initPromiseDefaultImpl();`

Laden Sie die Seite neu. Der Debugger hält am ersten Breakpoint. Mit `initPromiseDefaultImpl` wird in der JSON-API des ELO Indexservers festgelegt, dass alle Aufrufe asynchron ausgeführt werden. Die Indexserver-Funktionen geben damit ein `Promise`-Objekt zurück, dass das Funktionsergebnis einhüllt [10].

Das `Promise`-Objekt für die `IXConnection` wird wenige Zeilen darunter deklariert: `_connPromise`. Der Unterstrich am Anfang des Variablennamens soll verdeutlichen, dass es sich um eine „interne" Variable handelt und nicht direkt darauf zugegriffen werden soll. Stattdessen ist die weiter unten definierte Funktion `conn()` zu verwenden.

Lassen Sie das Programm bis zum Setzen der Variablen `url` weiterlaufen. Hier wird aus der URI der aufgerufenen Seite „/ix/plugin/my.namespace.plugin/jsonapi" die URI zur Indexserver-API zusammengebaut: „/ix/ix". Anschließend erstellt der Code eine `IXConnFactory` mit dieser URI.

Führen Sie das Programm fort, und setzen Sie einen Breakpoint in der Funktion `conn()`. Dort erfolgt beim ersten Aufruf ein Login am ELO Indexserver. Klicken Sie auf die Schaltfläche *Login* der Webseite. Die Indexserver-Verbindung wird hier mit der Funktion `createFromCookie` (neu ab ELO 11) aufgebaut. Sie erlaubt es, ein `IXConnection`-Objekt allein aus dem Session-Cookie des Apache Tomcat zu erstellen.

Um den Server zu veranlassen, ein Cookie für diese Zwecke zu erstellen, muss eine Authentifizierung erfolgen. Hierzu dient die übergebene Funktion `cookieFactory`. Sie wird immer dann von der `IXConnection` aufgerufen, wenn noch keine Authentifizierung erfolgt ist oder diese aufgrund eines Zeitlimits abgelaufen ist.

Im Beispiel ruft die Funktion `cookieFactory` die Seite einfach noch einmal über das `XMLHttpRequest`-Objekt des Browsers auf – jedoch unter dem Pfad `/plugin-auth`, der eine Authentifizierung erzwingt. Bei erfolgreicher Anmeldung wird im Server zum Cookie das Ticket des ELO Access Managers gespeichert, auf das er in folgenden Anfragen zugreift, um im Context des Benutzers zu arbeiten.

Die Schaltfläche *Logout* meldet die IXConnection ab. Dabei wird auch das Session-Cookie ungültig. Für nachfolgende Aufrufe wird ein neues erzeugt, das zunächst kein Ticket enthält, bis eine neue Anmeldung erfolgt ist.

3.6.7 Hinweis: Browser-Anwendung sollte mit Session-Cookie statt ELO-Ticket arbeiten

Im Browser sollte das ELO Ticket nicht verwendet werden, damit bei einer erfolgreichen JavaScript-Injection-Attacke die Benutzer-Sitzung nicht entführt werden kann. Wenn die JSON-API eingesetzt wird, sollte zum einen mit der in 3.6.6 vorgestellten `createFromCookie`-Funktion gearbeitet werden. Zum anderen ist sicherzustellen, dass die vom Server erstellten Session-Cookies das HttpOnly-Attribut besitzen, damit sie nicht über JavaScript ausgelesen werden können.



Abb. 80: Attribut 'HttpOnly'

In den Entwickler-Tools des Browsers kann diese Eigenschaft überprüft werden.

```
<!-- ==================== Default Session Configuration ================ -->
<!-- You can set the default session timeout (in minutes) for all newly   -->
<!-- created sessions by modifying the value below.                       -->

  <session-config>
      <session-timeout>30</session-timeout>
      <cookie-config>
          <http-only>true</http-only>
          <!-- <secure>true</secure> -->
      </cookie-config>
  </session-config>
```

Abb. 81: Datei 'web.xml'

In der Datei *web.xml* des Apache Tomcat kann für alle Anwendungen eingestellt werden, dass Session-Cookies mit HttpOnly ausgegeben werden:

Wenn die Web-Anwendung nur über HTTPS erreichbar ist, kann zusätzlich die Einstellung *secure* vorgenommen werden. Der Browser sendet damit das Cookie nur über eine verschlüsselte Verbindung (HTTPS) zurück an den Server.

4 Services des ELO Indexservers

Der ELO Indexserver stellt den Plug-ins den `HostService` bereit. Der Service kann über die Funktion `PluginActivator.getHostService()` erhalten werden.

Darin findet man die Methode `getScriptObjectFactory`, die das Interface `ScriptObjectFactory` zum Erstellen von Objekten bereitstellt, die z. B. einen direkten Datenbankzugriff oder die Prüfung von Berechtigungen ermöglichen.

Diese Objekte entsprechen denen im JavaScript (`DBConnection`, `Access`, …). Nur können sie im Plug-in nicht mit dem Operator `new` erstellt werden, weil die Informationen zum Ausführungskontext nicht implizit im Thread-Kontext enthalten sind. Deshalb müssen die Objekte über `ScriptObjectFactory` erstellt werden.

Für häufig verwendete Funktionen aus `ScriptObjectFactory` gibt es im `PluginActivator` Komfortfunktionen für den einfacheren Zugriff, z. B. `PluginActivator.getIxConnect()`.

Beispiel:

```
import java.util.Date;

import byps.RemoteException;
import de.elo.ix.client.IXExceptionC;
import de.elo.ix.client.IXServerEventsContext;
import de.elo.ix.client.LockZ;
import de.elo.ix.client.Sord;
import de.elo.ix.client.SordZ;
import de.elo.ix.client.plugin.IXServerEventsBase;
import de.elo.ix.client.plugin.PluginActivator;

public class IXServerEventsImpl extends IXServerEventsBase {

  public IXServerEventsImpl(PluginActivator activator) {
    super(activator);
  }

  @Override
  public void onBeforeCheckinSord(IXServerEventsContext ec,
      Sord sord, Sord sordDB, Sord parentSord, SordZ sordZ, LockZ unlockZ)
      throws RemoteException {

    if (sord.getName().contains("admin-only")) {

      // Check, whether the current user has administrator permissions...
      try (ScriptObjectFactory sof = activator.getScriptObjectFactory(ec)){
        if (!sof.getAccess().isAdmin(ec.getUser())) {

        // ... if not, throw an access denied exception
```

```
          sof.throwException(
            IXExceptionC.ACCESS_DENIED, "throw exception", null);
        }
      }
    }

    if (sord.getName().contains("replace-with-current-date")) {
      String now = activator.getIxConnect(ec).dateToIso(new Date());
      sord.setName(now);
    }
  }
}
```



Information: Innerhalb der `Activator.start` Methode eines Plug-ins kann noch nicht auf die Services des ELO Indexservers zugegriffen werden. Dies würde ein Deadlock verursachen. Falls ein Plug-in mit Informationen aus dem Archiv initialisiert werden muss, kann es das Interface `de.elo.ix.client.plugin.InitializePlugin` implementieren. Die `initialize`-Funktion des Interfaces wird während der Startphase des ELO Indexservers aufgerufen, nachdem die Plug-ins gestartet sind und bevor die Skripte geladen werden. In dieser Funktion steht eine IXConnection mit Administrationsrechten bereit: `activator.getIxConnectAdmin`.



Information: `ScriptObjectFactory` Objekte müssen mit ihrer `close` Methode aufgeräumt werden. Im obigen Beispiel geschieht dies implizit im try-with-Resources Statement.



Information: Ein `ScriptObjectFactory` Objekt erhält man außerhalb einer Ereignisfunktion von der Methode `getScriptObjectFactoryAdmin`. Mit diesem Factory-Objekt werden Dienste im Kontext des Indexserver-Kontos erstellt.

4.1 Datenbankzugriff

Für den Zugriff auf eine beliebige Datenbank verwendet man das von `ScriptObjectFactory.getDBConnectionForDataSource` bereitgestellte `DBConnection` Objekt. Voraussetzung dafür ist die Konfiguration einer Resource in der context.xml des Tomcat, s. Referenzdokumentation der IX-API, `de.elo.ix.jscript.DBConnection`.

Um SQL-Statements auf die Archivdatenbank anzuwenden, liefert die `ScriptObjectFactory` mit der Methode `getDBConnection` ein entsprechendes `DBConnection` Objekt.

Beispiel für die Ausgabe, wie oft jede Verschlagwortungsmaske verwendet wird:

```
public class Activator extends PluginActivator {
  private static Logger log = LoggerFactory.getLogger(Activator.class);
…
  public void dumpUsageOfKeywordingForms() throws Exception {
    try (ScriptObjectFactory fact = getScriptObjectFactoryAdmin()) {

      DBConnection db = fact.getDBConnection();
      Object[][] objUsages = db.query("select objmask, COUNT(*) from
                              objekte where objid > 1 group by objmask");

      for (int i = 0; i < objUsages.length; i++) {
        Object[] row = objUsages[i];
        int maskId = (int)row[0];
        int count = (int)row[1];
        log.info("{} {}", maskId, count);
      }
    }
  }
}
```

5 Rest-Plug-in

Zusammen mit dem ELO Indexserver wird das Plug-in *rest-plugin* installiert, in dem die JAX-RS-Referenzimplementierung *Jersey* [11] enthalten ist. Mithilfe dieses Plug-ins kann der ELO Indexserver auf einfachem Wege um eine selbstdefinierte REST-Schnittstelle erweitert werden.

5.1 Vorbereitung

Das *rest-plugin* befindet sich im ELO Installationsverzeichnis unter */prog/webapps/ix-plugins/private*. Damit es im Workspace referenziert werden kann, wird es mit den folgenden Schritten importiert.

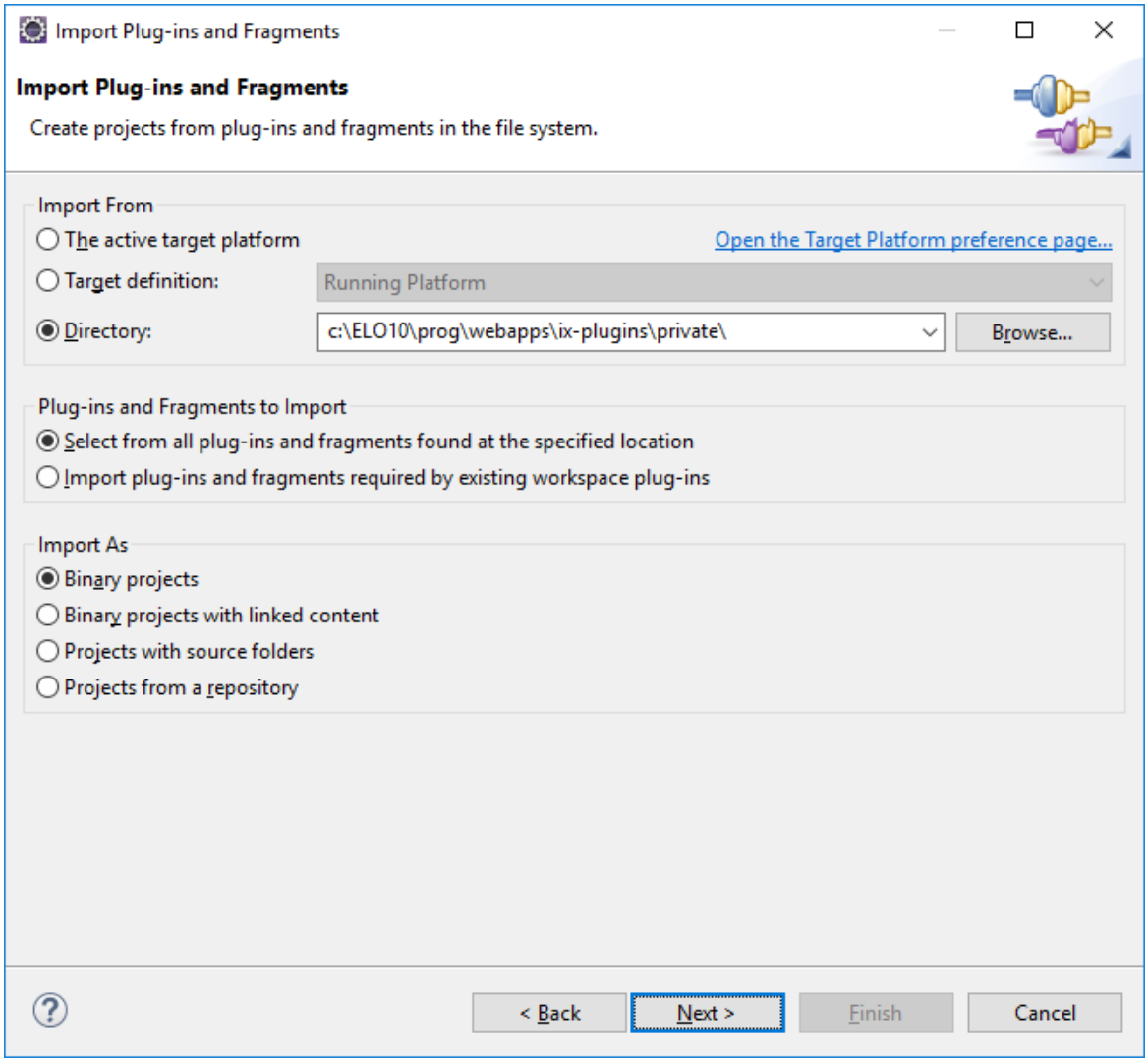


Abb. 82: Dialog 'Import Plug-ins and Fragments'

- *Eclipse > File > Import … > Plug-in Development > Plug-ins and Fragments > Next*

- Wählen Sie für *Directory* das Verzeichnis */prog/webapps/ix-plugins/private* im ELO Installationsverzeichnis.
- Im nachfolgenden Dialog wählen Sie *de.elo.ix.plugin.rest* aus.

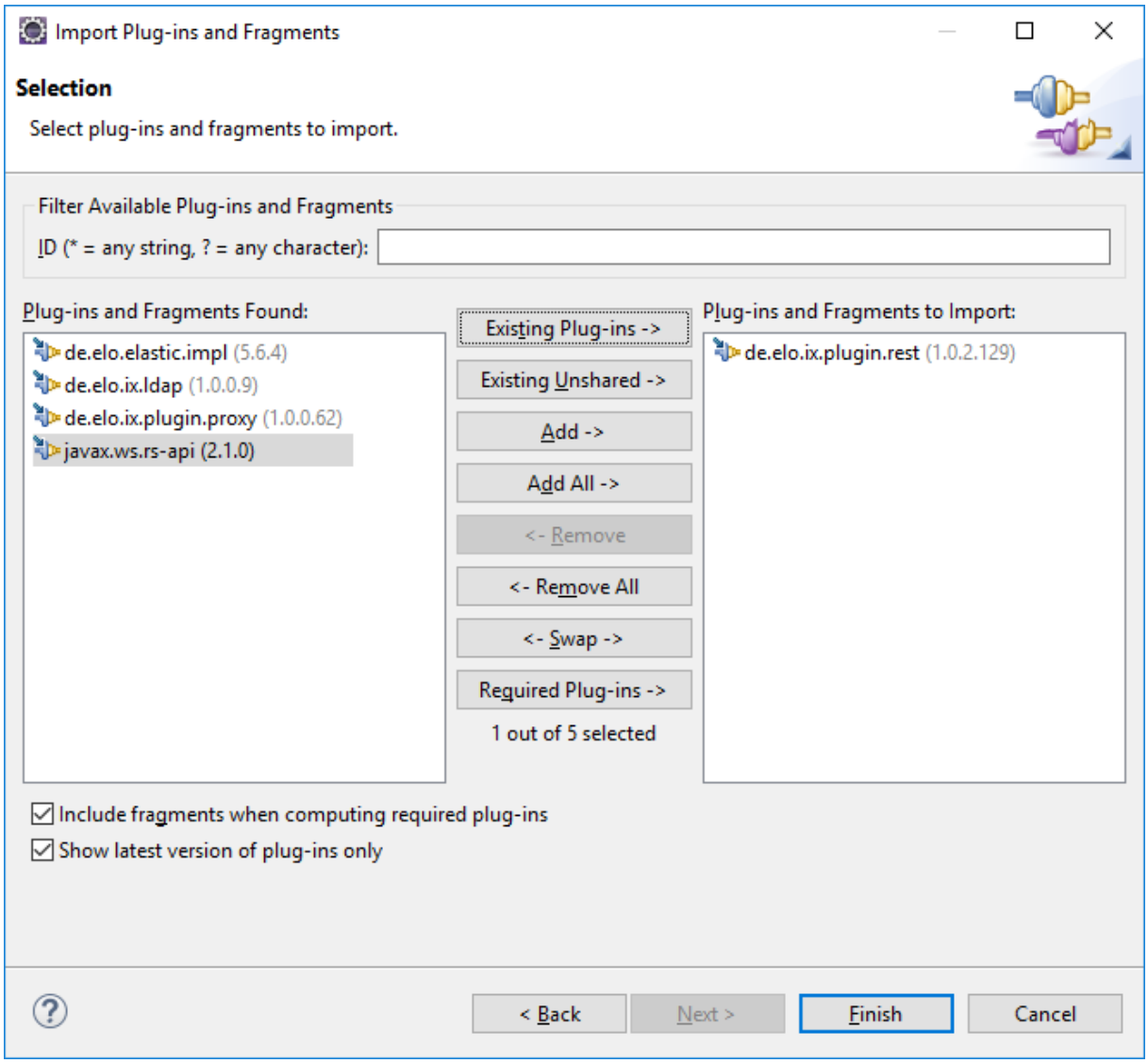


Abb. 83: Abschluss des Imports

- Mit *Finish* importieren Sie das Plug-in in den Workspace.

5.2 Hello-World-Beispiel

Für dieses Beispiel erstellen Sie zunächst ein Plug-in-Projekt (wie im Abschnitt *2.4 Plug-in erstellen* beschrieben).

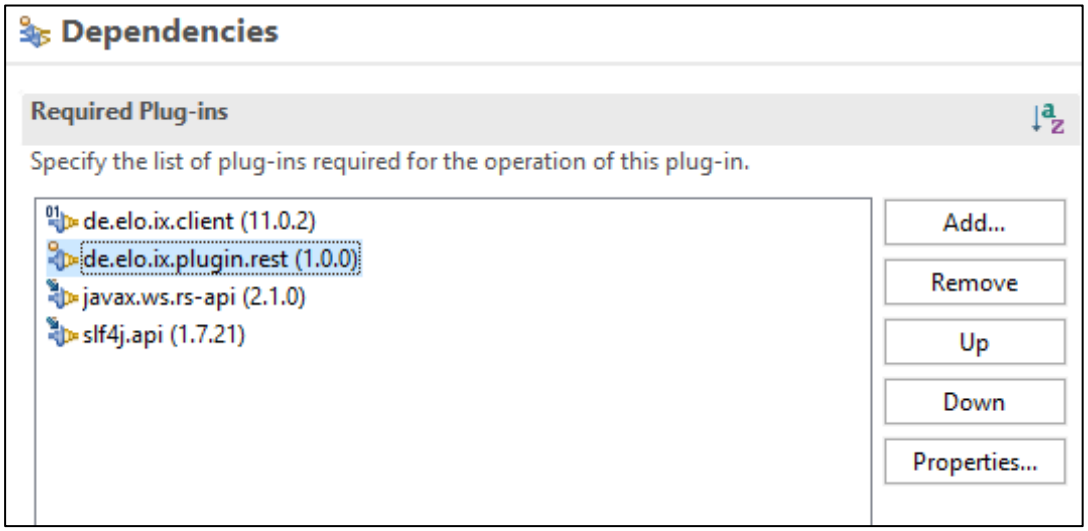


Abb. 84: Angabe notwendiger Plug-ins

Als zusätzliche Abhängigkeiten werden die Plug-ins *javax.ws.rs-api* und *de.elo.ix.plugin.rest* benötigt.

Definieren Sie eine Klasse, die als REST-Ressource dienen soll. Beispielsweise:

```
package de.elo.ix.rest.simple;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("simple-rest")
public class Hello {

  @Path("hello")
  @GET
  public String sayHello() {
    return "Hello!";
  }

}
```

In Ihrer `Activator`-Klasse registrieren Sie Ihre Ressource-Klasse wie folgt:

```
package de.elo.ix.rest.simple;

import org.osgi.framework.BundleContext;

import de.elo.ix.client.plugin.PluginActivator;
import de.elo.ix.plugin.rest.api.RestResource;

public class Activator extends PluginActivator {
```

```
  public void start(BundleContext context) throws Exception {
    super.start(context);
    registerService(RestResource.class, new RestResource(Hello.class));
  }

}
```



Information zur Erinnerung: Die `Activator`-Klasse muss von `PluginActivator` abgeleitet sein (2.4.2). In der `start`-Funktion muss `super.start` aufgerufen werden. Im Plug-in-Manifest muss unter Classpath das Verzeichnis *bin* eingetragen sein (2.5.1).

Starten Sie den ELO Indexserver, und öffnen Sie die folgende URL im Browser:

```
/ix-archivename/plugin/de.elo.ix.plugin.rest/simple-rest/hello
```

Der Browser sollte den Text „Hello!“ ausgeben.

5.3 IXConnection in Ressource-Klassen

In den Funktionen der Ressource-Klassen stehen zwei `IXConnection`-Objekte für den Aufruf von Indexserver-Funktionen zur Auswahl. Zum einen wird `IXConnection`-Objekt angeboten, das analog zum `ixConnect`-Objekt im Skripting eine benutzerbezogene Verbindung repräsentiert. Zum anderen besteht Zugriff auf ein `IXConnection`-Objekt mit administrativen Berechtigungen, wie `ixConnectAdmin` im Skripting.

Es ist im Rest-Plug-in vorgesehen, dass die beiden Verbindungsobjekte über Elementvariablen der Ressource-Klassen angeboten werden. Eine Übergabe in Funktionsparametern wird nicht unterstützt. Eine weitere Einschränkung besteht darin, dass Ressource-Klassen keine Singleton-Klassen sein können, wenn sie Verbindungsobjekte verwenden sollen.

Bedingt durch die interne Verarbeitungsreihenfolge im REST-Framework, müssen die `IXConnection`-Objekte in sogenannte Provider-Klassen eingefasst werden. Deren `get`-Methode liefert das `IXConnection`-Objekt zurück. Damit das REST-Framework die Elementvariablen der Provider-Objekte erkennt, müssen sie mit der Annotation `@Context` versehen werden.

Ein benutzerbezogenes Verbindungsobjekt kann nur in einer Funktion bereitgestellt werden, für die eine Benutzerauthentifizierung erfolgt ist. Um für eine Funktion eine Authentifizierung zu erzwingen, ist sie mit der Annotation `@Authenticated` zu markieren.

Das folgende Beispiel soll die beschriebenen Sachverhalte veranschaulichen.

```
package de.elo.ix.rest.simple;
```

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Context;

import de.elo.ix.client.IXConnectionIF;
import de.elo.ix.plugin.rest.api.Authenticated;
import de.elo.ix.plugin.rest.api.IXConnectionAdminProvider;
import de.elo.ix.plugin.rest.api.IXConnectionProvider;

@Path("simple-rest")
public class HelloWithConnection {

  /**
   * IXConnection provider for authenticated user.
   */
  @Context
  IXConnectionProvider ixConnectP;

  /**
   * IXConnection provider for administrative tasks.
   */
  @Context
  IXConnectionAdminProvider ixConnectAdminP;

  /**
   * Get name of authenticated user.
   * @return User name
   * @throws Exception
   */
  @Path("username")
  @Authenticated
  @GET
  public String getUserName() throws Exception {
    IXConnectionIF ixConnect = ixConnectP.get();
    return ixConnect.getUserName();
  }

  /**
   * Get IX instance name.
   * @return Instance name
   * @throws Exception
   */
  @Path("instance")
  @GET
  public String getInstanceNamme() throws Exception {
    IXConnectionIF ixConnectAdmin = ixConnectAdminP.get();
    return ixConnectAdmin.ix().getServerInfo().getInstanceName();
  }
}
```

Um den Code ausprobieren zu können, muss die Klasse in der Activator-Klasse registriert werden.

```
public class Activator extends PluginActivator {

  public void start(BundleContext context) throws Exception {
    super.start(context);
    registerService(RestResource.class,
            new RestResource(HelloWithConnection.class));
  }
}
```

Die Funktion `getInstanceName` steht unter der folgenden URL zur Verfügung:

```
http://server:port/ix-archivname/plugin/de.elo.ix.plugin.rest/simple-rest/instance
```

Im nachfolgenden Abschnitt erfahren Sie, wie Sie `getUserName` testen können.

5.4 Authentifizierung

Aufrufe der REST-Funktionen können zum einen über die bereits in 3.6.2 und 3.6.3 vorgestellten Verfahren authentifiziert werden. Daneben gibt es für REST-Plug-ins noch die Möglichkeit einer BASIC-Authentifizierung mit einem ELO Konto.

Die Funktion `getUserName` aus dem obigen Beispiel ist mit `@Authenticated` markiert und verlangt damit eine Authentifizierung. Sie kann über die folgenden URLs aufgerufen werden (s. 5.4):

Mit SSO-Authentifizierung:

```
http://server:port/ix-archivname/plugin-auth/de.elo.ix.plugin.rest/simple-rest/username
```

Mit Apache-Tomcat-Authentifizierung:

```
http://server:port/ix-archivname/manager/plugin/de.elo.ix.plugin.rest/simple-rest/username
```

Mit BASIC-Authentifizierung gegen ELO:

```
http://server:port/ix-archivname/plugin/de.elo.ix.plugin.rest/simple-rest/username
```



Information: Schließen Sie zwischen den Aufrufen alle Browserfenster, um bei jedem Aufruf eine separate Authentifizierung auszulösen.



Information: In einer mit `@Authenticated` markierten Funktion ist das benutzerbezogene Verbindungsobjekt immer gültig. Wenn die Authentifizierung scheitert, wird die Funktion nicht aufgerufen.

5.5 Konvention zu den URIs der Ressource-Klassen

Die Funktionen aller Ressource-Klassen des Systems sind unter der URI des REST-Plug-ins eingehängt. Damit es nicht zu Überschneidungen in den Funktionsaufrufen kommt, sollte jedes Plug-in eine Basis-URI definieren, die jeder Ressource-Klasse zugewiesen wird. Beispielsweise ist in den beiden vorangegangenen Klassendefinitionen die URI *simple-rest* die den Ressource-Klassen zugeteilte Basis-URI. In einem produktiv eingesetzten Plug-in ist hierfür eine globale Konstante zweckmäßig.

5.6 Weitere Funktionalitäten der JAX-RS-Spezifikation

Es ist Ziel der Entwicklung des Rest-Plug-ins möglichst viele Funktionalitäten der JAX-RS-Spezifikation zu unterstützen. Die Einschränkungen liegen hauptsächlich in dem Bereich, wo direkt auf die Objekte des Frameworks zugegriffen werden muss. Beispielsweise können keine selbstdefinierten JAX-RS-Komponenten erstellt werden, die mit der `@Context`-Annotation vom Framework in die Ressource-Klassen injiziert werden.

Eine weitere Einschränkung besteht darin, dass das Rest-Plug-in die Ressource-Klassen nicht automatisch findet. In vielen Beispielen im Web zur Konfiguration eigenständiger REST-Services findet man dagegen eine Liste von Packages, die nach Ressource-Klassen durchsucht werden. Für das Rest-Plug-in müssen jedoch alle Ressource-Klassen explizit im OSGi-Framework registriert werden, wie es im Beispiel 5.2 erfolgt.

5.6.1 Serialisierung von Objekten in XML oder JSON

Die Version 1.0.1. des Rest-Plug-ins erlaubt bereits die Serialisierung von Klassen ins XML-Format. Eine Übertragung im JSON-Format ist ab Version 1.0.2 möglich. Weil das JSON-Format in der Praxis weitaus häufiger zum Einsatz kommt als das XML-Format, behandeln die folgenden Beispiele nur die JSON-Serialisierung und setzen somit 1.0.2 voraus.

Wenn nur die Version 1.0.1 verfügbar ist und im JSON-Format serialisiert werden muss, kann die einfach zu verwendende GSON-Bibliothek [12] hilfreich sein.

5.6.2 Anforderungen an eine serialisierbare Klasse

Damit eine Klasse vom REST-Framework serialisiert werden kann, darf sie nicht abstrakt sein und benötigt einen Standard-Konstruktor. Weil dies für die Klassen der API des ELO Indexservers gilt, können sie als Rückgabewerte und Parameter von REST-Funktionen eingesetzt werden.

5.6.3 Klassen als Rückgabewerte von Funktionen

Beispielsweise kann die Hello-World-Klasse aus dem Abschnitt *5.3 IXConnection in Ressource-Klassen* um die folgende Funktion erweitert werden, um ein Sord-Objekt aus der Datenbank zu lesen:

```
@GET
@Path("sord/{id}")
@Authenticated
@Produces(MediaType.APPLICATION_JSON)
public Sord getSord(@PathParam("id") String id) throws Exception {
  IXConnectionIF conn = ixConnectP.get();
  Sord sord = conn.ix().checkoutSord(id, SordC.mbLean, LockC.NO);
  return sord;
}
```

Im Browser ruft man die Funktion mit der folgenden URL auf:

```
http://server:port/ix-archive/plugin/de.elo.ix.plugin.rest/simple-
rest/sord/1
```

Das Ergebnis sieht nach erfolgreicher Anmeldung z. B. wie folgt aus:

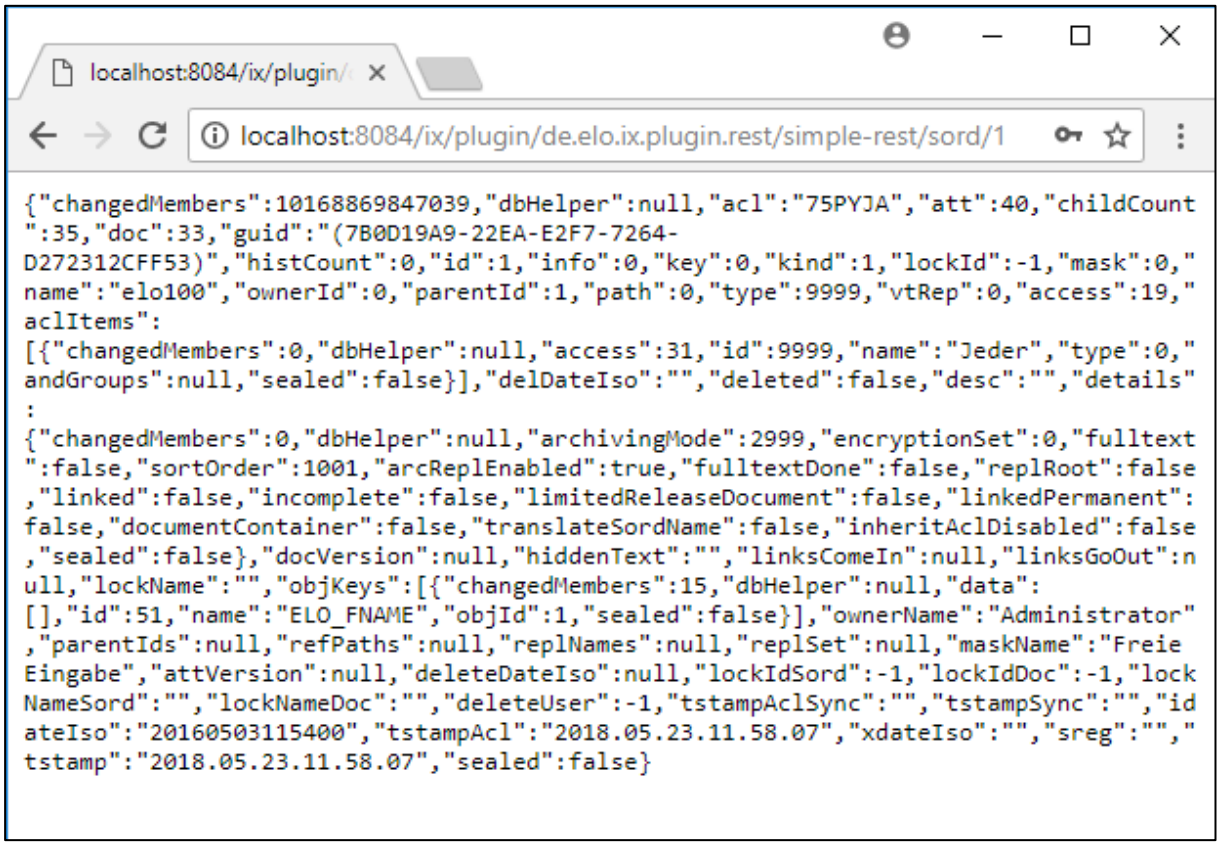


Abb. 85: Ausgabe im Browser

5.6.4 Klassen als Parameter von Funktionen

Im folgenden Beispiel soll ein SORD-Objekt über eine REST-Funktion eingefügt werden. Die Funktion ist wie folgt definiert:

```
  @POST
  @Path("sord")
  @Authenticated
  @Produces(MediaType.APPLICATION_JSON)
  @Consumes(MediaType.APPLICATION_JSON)
  public int updateSord(Sord sord) throws Exception {
    IXConnectionIF conn = ixConnectP.get();
    return conn.ix().checkinSord(sord, SordC.mbAll, LockC.NO);
  }
```

Um die Funktion aufzurufen, benötigt es etwas Code auf der Client-Seite. Am einfachsten geht dies mit JavaScript-Code in einer HTML-Seite, der vom Browser ausgeführt wird. Wegen der Same-Origin-Policy muss der JavaScript-Code allerdings vom selben Server kommen, für den die REST-Funktion aufgerufen wird. Dafür kann man die HTML-Seite ebenfalls von einem REST-Aufruf zurückliefern, wie es nachfolgend gezeigt wird.

```
  @GET
  @Path("form")
  @Produces(MediaType.TEXT_HTML)
  public String getUpdateForm() throws Exception {
    StringWriter sw = new StringWriter();
    try (Reader rd = new InputStreamReader(
           getClass().getResourceAsStream("UpdateSord.html"), "UTF-8")) {
      char[] buf = new char[10 * 1000];
      int len = 0;
      while ((len = rd.read(buf)) != -1) {
        sw.write(buf, 0, len);
      }
    }
    return sw.toString();
  }
```

Die HTML-Seite *UpdateSord.html* enthält den folgenden Inhalt und kann im Eclipse-Projekt im selben Source-Code-Ordner und Package liegen wie die REST-Klasse.

```
<html>
<head>
<title>Simple REST Application</title>
<script src="https://code.jquery.com/jquery-1.10.2.js"></script>
<script type="text/javascript">

  var ixurl = http://server:port/ix-archive;
  var url = ixurl + "/plugin/de.elo.ix.plugin.rest/simple-rest/sord";

  function updateSord() {
```

```
    var sord = {
      "id" : -1,
      "parentId" : 1,
      "mask" : 0,
      "type" : 1,
      "name" : "Simple REST",
      "details" : {
            "sortOrder" : 1001
      }
    };

    $.ajax({
      url : url,
      type : "POST",
      data : JSON.stringify(sord),
      contentType : "application/json; charset=utf-8",
      dataType : "json",
      success : function(objId) {
            alert(objId);
      },
      error : function(xhr, textStatus, error) {
            alert(xhr.status + " " + xhr.responseText);
      }
    });
  }
</script>
</head>
<body>
  <input type="button" value="Update Sord" onclick="updateSord()" />
</body>
</html>
```

Im Browser erhält man die Seite mit der URL:

```
http://server:port/ix-archive/plugin/de.elo.ix.plugin.rest/simple-rest/form
```

Sie zeigt die Schaltfläche *Update Sord*. Beim Klick auf die Schaltfläche wird ein SORD-Objekt auf oberster Ebene im Archiv eingefügt.

5.6.5 Datei über eine Webseite hochladen

Die offizielle REST-API-Spezifikation von Oracle sieht keine Annotationen und Klassen für das Hochladen von Dateien aus einem Web-Formular vor.

Zwar definiert das hier verwendete REST-Framework die nötigen Werkzeuge. Jedoch sollten sie als Implementierungsdetail nicht nach außen gegeben werden, um die entwickelten Plug-ins vom Framework unabhängig zu halten.

Deshalb bleibt nur der Weg übrig, die HTTP-Nachricht in der Ressource-Klasse komplett selbst zu behandeln. Hier kann die Bibiliothek Apache Commons FileUpload [13] sehr hilfreich sein.

Eine Upload-Funktion kann wie folgt definiert werden:

```
@POST
@Path("upload")
public Response upload(@Context HttpServletRequest request) {
  // Use Apache Commons FileUpload
  // ...
  return Response.status(200).build();
}
```

5.7 Debugging in den Quellcode

Beim Debugging kann es hilfreich sein, in den Quellcode des Rest-Plug-ins zu navigieren. Hierzu nehmen Sie das Plug-in als *Binary-Project* in den Eclipse-Workspace auf, wie es für die Datei *EloixClient.jar* im Abschnitt *2.3.8 Quellcode und Dokumentation der EloixClient.jar bereitstellen* gezeigt wird. Weil der Quellcode in der JAR enthalten ist, benötigen Sie keine Angabe für *Source attachment* in den Projekteigenschaften.

6 Proxy-Plug-in

Das mitgelieferte Proxy-Plug-in ermöglicht es, eine andere Web-Anwendung unter der Indexserver-URL anzusprechen. Hierdurch erzielt man die folgenden Vorteile:

- Der Konfigurationsaufwand auf der Client-Seite reduziert sich, weil die URL der Web-Anwendung nicht angegeben werden muss sondern aus der URL des ELO Indexservers abgeleitet werden kann.
- Ist die Client-Anwendung eine Browser-Seite (JavaScript-Anwendung) kommt sie ohne Cross-Site-Scripting-Aufrufe aus und eine serverseitige CORS-Konfiguration ist nicht nötigt.
- Die Übergabe der ID der Benutzer-Session (z. B. das Ticket des ELO Access Mangers) in die andere Web-Anwendung entfällt. Das Ticket wird vom Proxy-Plug-in bei jedem Aufruf server-seitig über ein Cookie an die Web-Anwendung weitergegeben. Die Benutzer-Session im Browser kann allein über das Session-Cookie des ELO Application-Servers identifiziert werden. Weil es nicht im JavaScript benötigt wird, kann es mit der Option *httpOnly* ausgeliefert werden, was die Sicherheit der Anwendung erhöht.

6.1 Konfiguration

Das Proxy-Plug-in liest beim Start seine Konfiguration aus der Datei *de.elo.ix.plugin.proxy.properties* im Konfigurationsverzeichnis des ELO Indexservers, also */ELO/config/ix-archivname/servername*.

Die Weiterleitungen werden in Form von Schlüssel-Wert-Paaren angegeben. Als Schlüssel dient der auf „de.elo.ix.plugin.proxy“ folgende Teil der URI, der umgeleitet werden soll.

Beispiel einer Konfigurationsdatei:

```
/web-elo100/=http://localhost:9090/web-elo100/
/wf-elo100/=http://localhost:9090/wf-elo100/
```

Damit werden Aufrufe zu

```
http://server:port/ix-archivname/plugin/de.elo.ix.plugin.proxy/web-elo100/
```

umgeleitet nach:

http://localhost:9090/web-elo100/



Information: Die hier verwendeten URLs für den ELO Web Client und ELOwf funktionieren zusammen mit dem Proxy-Plug-in voraussichtlich erst ab Version 12 von ELO Web Client und ELOwf.

6.2 Übergabe der Benutzer-Session

6.2.1 Ticket-Cookie

Vom Proxy-Servlet wird in jedem Aufruf das Session-Ticket des ELO Access Managers als HTTP-Cookie an die Web-Anwendung übergeben. Dies erlaubt es in der Web-Anwendung ein `IXConnection`-Objekt über `IXConnFactory.createFromTicket` zu erstellen. Das Ticket-Cookie hat den Namen *ticket*.

6.2.2 Session-Transformation

Wird von der Web-Anwendung ein Session-Cookie mit dem Namen *JSESSIONID* zurückgegeben, merkt sich das Proxy-Servlet dessen Wert zur Indexserver-Session und sendet ihn bei Aufrufen mit. Die Session zwischen Client und ELO Indexserver wird also in die Session zwischen ELO Indexserver und Web-Anwendung transformiert.

6.2.3 Location-Header, URL Redirect

Sendet die Web-Anwendung einen Location-Header und den HTTP-Status 301, 302, 303, 307 oder 308, wird die im Location-Header angegebene URL auf die Basis-URL des Proxy-Plug-ins transformiert. Dies setzt voraus, dass der Location-Header eine relative URL oder eine absolute URL zur Web-Anwendung (laut Konfigurationsdatei) enthält. Eine fremde URL wird zum Browser unverändert durchgereicht.

6.2.4 Weitere Cookies und Header

Alle weiteren von der Web-Anwendung gelieferten Cookies oder HTTP-Header werden über das Proxy-Plug-in an den Client unverändert weitergereicht, sofern sie nicht vom ELO Application-Server überschrieben werden (bspw. „Transfer-Encoding“). Gleiches gilt für die Header, die der Browser zum ELO Indexserver sendet.

6.3 Debugging in den Quellcode

Beim Debugging kann es hilfreich sein, in den Quellcode des Proxy-Plug-ins zu navigieren. Hierzu nehmen Sie das Plug-in als *Binary-Project* in den Eclipse-Workspace auf, wie es für die Datei EloixClient.jar im Abschnitt *2.3.8 Quellcode und Dokumentation der EloixClient.jar bereitstellen* gezeigt wird. Weil der Quellcode in der JAR-Datei enthalten ist, benötigen Sie keine Angabe für *Source attachment* in den Projekteigenschaften.

7 Pug-ins über die API des ELO Indexservers verwalten

Über die API des ELO Indexservers können die installierten Plug-ins aufgelistet, gestartet, gestoppt und deinstalliert werden. Darüber hinaus ist es möglich, neue Plug-ins zu installieren. Die zugehörigen Funktionen findet man im Interface `de.elo.ix.client.plugin.PluginService`.

7.1 Beispiel zum Auflisten der Plug-ins

```
IXConnection conn = …
List<PluginInfo> pluginInfos = conn.getPluginService().getPlugins();
for (PluginInfo pluginInfo : pluginInfos) {
  System.out.println("Plugin: "
    + "ID=" + pluginInfo.getId()
    + ", symbolicName=" + pluginInfo.getSymbolicName()
    + ", state=“ + pluginInfo.getState();
}
```

7.2 Beispiel zum Stoppen und Starten eines Plug-ins.

Es ist hierfür das Recht *Hauptadministrator* nötig.

```
IXConnection conn = …
PluginInfo pluginInfo = …
long pluginId = pluginInfo.getId();
conn.getPluginService().stop(pluginId);
conn.getPluginService().start(pluginId);
```

7.3 Beispiel zum Installieren und Deinstallieren eines Plug-ins

Installation kann zum einen mit dem Dateinamen des Plug-ins erfolgen. Der Dateiname muss für den ELO Indexserver sichtbar sein und sollte nicht in einem Verzeichnis liegen, das in den Optionen `osgiPluginDirectories` oder `osgiPluginDirectoriesDebug` angegeben ist.

```
IXConnection conn = …
String pluginFile = … absolute path from the viewpoint of IX
long pluginId = conn.getPluginService().install(pluginFile);
```

Zum anderen kann das Plug-in auch in einem Stream übergeben werden:

```
IXConnection conn = …
InputStream pluginStream = …
long pluginId = conn.getPluginService().upload(pluginStream);
```

Vor der Installation wird das Plug-in unter seinem symbolischen Namen plus Versionsnummer in dem bevorzugten Plug-in-Verzeichnis abgelegt. Es ist das erste in der Option `osgiPluginDirectories` angegebenen Verzeichnis.

Zum Deinstallieren ruft man die Funktion `uninstall` auf:

```
IXConnection conn = …
PluginInfo pluginInfo = …
long pluginId = pluginInfo.getId();
conn.getPluginService().uninstall(pluginId);
```

7.4 Beispiel zum Lesen und Ändern der Plug-in-Konfiguration

Die Konfiguration des Plug-ins kann über die API des ELO Indexservers vorgenommen werden, sofern das Plug-in den Serivce `de.elo.ix.client.plugin.ConfigurePlugin` anbietet.

```
IXConnection conn = …
long pluginId = …

// Read plugin configuration.
String configString = conn.getPluginService().configure(pluginId,
ConfigurePlugin.READ);

// Modify plugin configuration.
String previousConfigString = conn.getPluginService().configure(pluginId,
configString);
```

8 Relevante Log-Einträge

Zur Kontrolle, ob die erwarteten Plug-ins geladen werden, oder zur Fehlersuche können die folgenden Log-Einträge hilfreich sein.

8.1 Verzeichnisse aus denen Plug-ins geladen werden

Aus diesen Zeilen kann abgelesen werden, in welchen Verzeichnissen nach Plug-ins gesucht wird:

```
07:52:27,007 eloix-init-2 init-2 INFO  (EnvProperties.java:414) -
osgiPluginDirectories=c:\ELO11\prog\webapps\ix-plugins
07:52:27,008 eloix-init-2 init-2 INFO  (EnvProperties.java:414) -
osgiPluginDirectoriesDebug=D:\java\workspace_ix_plugin_dev3\first-
plugin;D:\java\workspace_ix_plugin_dev3\http-plugin;
```

8.2 OSGi-Framework startet

Die Startphase der OSGi-Integration beginnt im ELOix-Log mit dem Eintrag (Beispiel):

```
07:52:29,899 eloix-init-2 init-2 INFO  (ServerObject.java:1179) - Start
OSGi framework.
```

Darauf folgt die Ausgabe des Cache-Verzeichnisses, in das die Plug-ins kopiert werden.

```
07:52:29,899 eloix-init-2 init-2 INFO  (OSGiFrameworkService.java:275) -
OSGi cache directory=c:\ELO11\data\ix-elo100\ELO-PCIMIGW-ELO11-1\osgi-
8990289655060174449.cache
```

Anschließend startet die Initialisierung des OSGi-Frameworks:

```
08:38:37,737 eloix-init-2 init-2 INFO  (OSGiFrameworkService.java:125) -
Initialize Apache Felix
```

8.3 Vom ELOix bereitgestellte Bundles werden installiert

Vom ELO Indexserver werden implizit Bundles für das Logging bereitgestellt. Somit ist es möglich, dass Plug-ins in die Log-Datei des ELO Indexservers schreiben. Ferner wird die EloixClient-Bibliothek als Bundle installiert.

```
07:52:30,669 eloix-init-2 init-2 INFO  (ProvisionActivator.java:62) -
Install OSGi bundle=org.apache.commons.logging_1.1.1, location=Apache
Commons Logging from ix.war
07:52:30,696 eloix-init-2 init-2 INFO  (ProvisionActivator.java:62) -
Install OSGi bundle=org.slf4j_1.7.21, location=Slf4j from ix.war
07:52:30,708 eloix-init-2 init-2 INFO  (ProvisionActivator.java:62) -
Install OSGi bundle=de.elo.ix.client_10.99.999.022, location=ELO
Indexserver Client Library from ix.war
```

8.4 Suche nach zu installierenden Plug-ins

Die in den Optionen `osgiPluginDirectories` und `osgiPluginDirectoriesDebug` angegebenen Verzeichnisse (siehe *2.2 Konfiguration für weiteren ELO Indexserver* vorbereiten) werden rekursiv nach Plug-ins durchsucht. Als Plug-ins werden JAR-Dateien erkannt, die ein Manifest mindestens mit dem Attribut *Bundle-SymbolicName* enthalten.

Plug-ins aus dem Archiv (siehe *2.7 Plug-in fertigstellen*) werden in das Verzeichnis *data* ausgepackt (z. B. *c:\ELO11\data\ix-elo100\ELO-PCIMIGW-ELO11-1\Indexserver Scripting Base_ALL*) und von dort geladen.

Ein erkanntes Plug-in wird im Log ausgegeben mit:

```
08:29:44,738 eloix-init-2 init-2 INFO  (ProvisionActivator.java:204) -
Found plugin=org.apache.httpcomponents.httpcore_4.4.6.v20170210-0925 at
file:/c:/ELO11/prog/webapps/ix-
plugins/plugins/org.apache.httpcomponents.httpcore_4.4.6.v20170210-0925.jar
```

Anschließend werden die Plug-ins installiert und Log-Zeilen (wie in *8.3 Vom ELOix bereitgestellte Bundles werden installiert*) erzeugt.

8.5 Start der Plug-ins

Nachdem alle Plug-ins installiert sind, werden sie gestartet.

```
08:38:38,821 eloix-init-2 init-2 INFO  (ProvisionActivator.java:79) - Start
OSGi bundle, ID=5, org.apache.httpcomponents.httpcore_4.4.6.v20170210-0925
```

Mit dem Bundle-Namen wird die ID ausgegeben, die das OSGi-Framework beim Installieren zugeteilt hat. Die ID gilt nur solange der ELOix nicht neu gestartet und neu geladen wird.

8.6 Start des OSGi-Frameworks

Mit der Ausgabe

```
08:38:38,833 eloix-init-2 init-2 INFO  (OSGiFrameworkService.java:139) -
Start Apache Felix
```

erfolgt der Start des OSGi-Frameworks. Dabei werden die start-Methoden der Activator-Klassen der Plug-ins aufgerufen (siehe *2.4.2 Activator-Klasse anpassen*). In einer start-Methode erfolgt in der Regel die Registrierung der bereitgestellten Services:

```
08:38:38,989 FelixStartLevel   INFO  (Activator.java:22) - first-plugin
Activator startet
08:38:38,994 FelixStartLevel   INFO  (HostActivator.java:67) - OSGi service
of my.namespace.firstplugin changed: [de.elo.ix.client.IXServerEvents]
registered
```

Das OSGi-Framework ist erfolgreich gestartet, wenn diese Ausgabe erscheint:

```
08:38:39,003 eloix-init-2 init-2 INFO  (OSGiFrameworkService.java:142) -
Start Apache Felix OK
```

9 FAQ

9.1 ELO Indexserver startet nicht

9.1.1 Fehlermeldung im ix-log „java.lang.NoClassDefFoundError: org/osgi/framework/FrameworkListener“

Es fehlen die OSGi-Framework-Bibliotheken im Tomcat/Lib-Verzeichnis (siehe *2.1.4 Tomcat Server Runtime einrichten*).

9.1.2 Fehlermeldung im ix-log „java.lang.ClassNotFoundException: com.microsoft.sqlserver.jdbc.SQLServerDriver

Es fehlen die JDBC-Treiber im Tomcat/Lib-Verzeichnis (siehe *2.1.4 Tomcat Server Runtime einrichten*).

9.2 Plug-in wird nicht geladen

9.2.1 Fehlermeldung im ix-log „org.osgi.framework.BundleException: Unable to resolve …: missing requirement…“:

Es fehlt ein Bundle, von dem das Plug-in abhängig ist.

9.2.2 Fehlermeldung „OSGi framework error…. Not found: …Activator“

Das Plug-in benötigt zum Debuggen die Erweiterung des Classpath um *bin* (siehe *2.5.1. Kunstgriff*)

Weiterer Grund: Eclipse hat nach einem *Clean* die Java-Dateien noch nicht übersetzt und das Verzeichnis *bin* ist leer. Dann ein Neustart von Eclipse.

9.3 Plug-in-Services nicht verfügbar

Prüfen Sie die Activator-Klasse:

- Die Activator-Klasse muss von PluginActivator abgeleitet sein (`extends` statt `implements`)
- Die start-Funktion der Activator-Klasse muss die start-Funktion der Oberklasse aufrufen:

```
public void start(BundleContext context) throws Exception {
    super.start(context);
    registerService(…);
}
```

9.4 Compiler-Fehler

9.4.1 PluginActivator cannot be resolved to a type

Die Activator-Klasse kann nicht von PluginActivator abgeleitet werden. Die Bundle-Abhängigkeit zum Bundle *de.elo.ix.client* fehlt oder die Ableitung wird mit dem Schlüsselwort „implements“ statt „extends“ angegeben.

9.4.2 Nach Eclipse-Neustart Fehlersymbol auf dem Plug-in-Projekt

Beispiel:

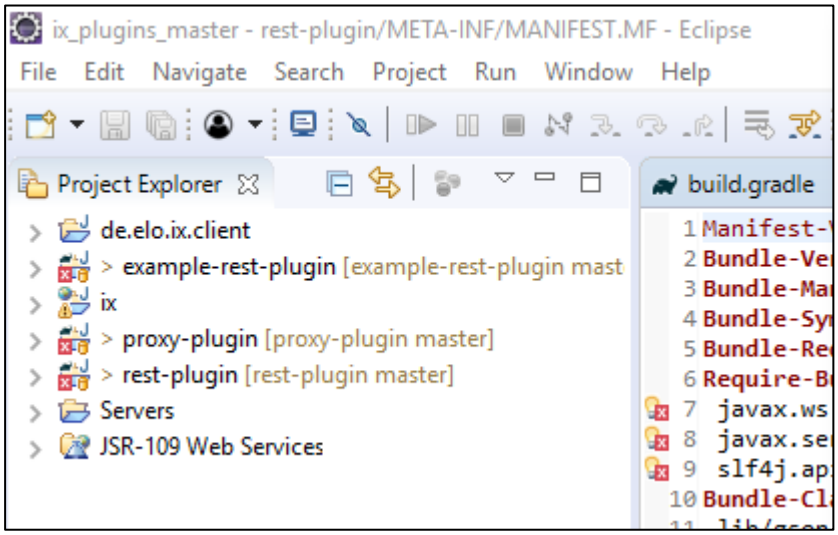


Abb. 86: Fehlersymbol

Öffnen Sie die Target-Platform-Definition im „ix“-Projekt (siehe *2.3.7 Target Platform für die Plug-in-Entwicklung im Eclipse definieren*). Klicken Sie den Link *Reload Target Platform.*

9.4.3 Plug-ins „out of sync“

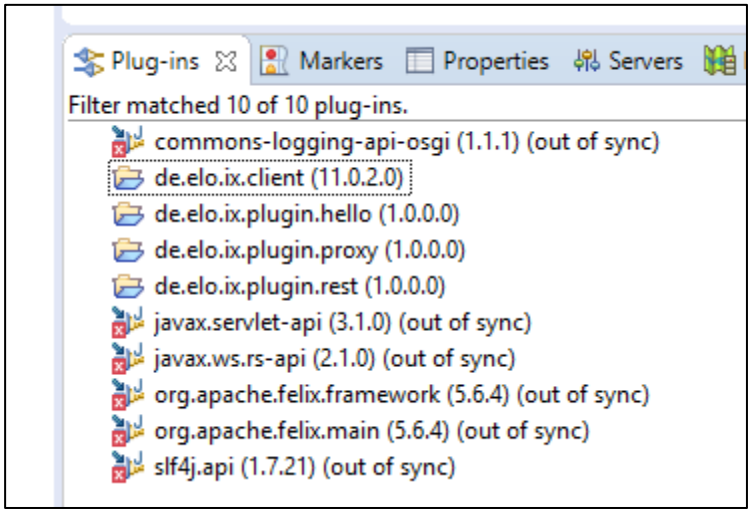


Abb. 87: Kennzeichnung mit '(out-of-sync)'

Wenn in der Plug-ins-Ansicht einige Plug-ins mit „(out-of-sync)“ gekennzeichnet sind, wie im Beispiel-Screenshot gezeigt, hilft ein Neustart von Eclipse.

9.5 Eclipse hängt nach dem Laden des Workspace

Beenden Sie Eclipse (ggf. im Windows Task-Manger). Öffnen Sie ein Kommandozeilenfenster im Installationsverzeichnis von Eclipse, und geben Sie ein:

```
eclipse -clean –clearPersistedState
```

siehe: https://stackoverflow.com/questions/8972034/eclipse-hangs-on-loading-workbench

10 Literaturverzeichnis

[1] WIKIPEDIA, „OSGi," 23 03 2017. [Online]. Available: https://de.wikipedia.org/wiki/OSGi.

[2] A. S. Foundation, „Apache Felix," 24 04 2016. [Online]. Available: http://felix.apache.org/.

[3] S. Patil, „Java World," 4 3 2008. [Online]. Available: http://www.javaworld.com/article/2077837/application-development/java-se-hello-osgi-part-1-bundles-for-beginners.html.

[4] A. S. Foundation, „Apache Felix Framework Distribution," 10 7 2017. [Online]. Available: http://felix.apache.org/downloads.cgi#framework.

[5] ORACLE, „JDK Download," [Online]. Available: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. [Zugriff am 26 02 2018].

[6] Eclipse, „Eclipse Download," 2017. [Online]. Available: https://www.eclipse.org/downloads/.

[7] A. S. Foundation, „Apache Tomcat," [Online]. Available: https://tomcat.apache.org/. [Zugriff am 01 07 2017].

[8] Maven, „Maven Central Repository," 2017. [Online]. Available: https://search.maven.org/.

[9] DZone, „DZone," 29 4 2008. [Online]. Available: https://dzone.com/articles/simple-osgi-service.

[10] Mozilla, „Promise," [Online]. Available: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Zugriff am 07 02 2018].

[11] J. P. Team, „Jersey," [Online]. Available: https://jersey.github.io/. [Zugriff am 25 05 2018].

[12] Google, „Maven Central," [Online]. Available: https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.5. [Zugriff am 29 05 2018].

[13] A. Group, „Apache Commons FileUpload," [Online]. Available: https://commons.apache.org/proper/commons-fileupload. [Zugriff am 30 05 2018].

[14] „Maven Central org.apache.httpcomponents," [Online]. Available: https://mvnrepository.com/artifact/org.apache.httpcomponents. [Zugriff am 2017].

[15] O. JDK. [Online]. Available: https://openjdk.java.net/. [Zugriff am 30 07 2019].