

Annotation Processor für die Generierung einer Webapp mit Datenobjekte

Sven Eric Panitz

30. Mai 2021

Inhaltsverzeichnis

1	Annotation Processor	1
1.1	Zu berücksichtigende Annotation	2
1.2	Beispielklassen	2
1.3	Prozessor	3
1.3.1	Die eigentliche Prozessormethode	4
1.3.2	Generierung von Setter-Hilfsmethoden	5
1.3.3	Check auf korrekte Feldtypen	7
1.4	Generierung eines Webservices mit Java Servlets	8
1.4.1	Servlet Container	9
1.4.2	Servletkonfiguration	9
1.4.3	Einrichten eines Servlets auf dem Tomcat	12
2	Aufgaben	13

1 Annotation Processor

Ziel dieses Lehrbrief soll es zeigen, wie man den Javacompiler über einen Prozessor für im Quelltext gemachte Annotationen um weitere Funktionalität erweitern kann. Anders als im voran gegangenen Lehrbrief, in dem Annotationen genutzt wurden, um zur Laufzeit über Reflexion Informationen dynamisch auszuwerten, werden nun die Annotationen statisch während der Kompilierung ausgewertet.

Hier gibt es zwei unterschiedliche Arten von Funktionalitäten, um die man den Compiler mit einem Prozessor für Annotationen erweitern kann:

- weitere statische Überprüfungen, die zu weiteren Fehlermeldungen und Warnungen während der Kompilierung führen und

- die Generierung von weiteren Quelltext, der dann direkt kompiliert wird.

Wir werden in diesem Lehrbrief beides zeigen und dieses anhand der Datenobjekte, wie sie im vorangegangenen Lehrbrief definiert wurden.

Ein zweiter Aspekt dieses Lehrbrief beschäftigt sich mit einem Webdienst. Es geht darum für einen Webserver Javaklassen zu schreiben, sogenannte Servlets, die für Http-Anfragen die Antworten generieren.

Damit erhalten wir Einblick in die Programmierung eines Frameworks. Mit dem vorhergehenden Lehrbrief erhalten wir ein Framework, dass für beliebige Record-Klassen mit der entsprechenden Annotationen eine Webapplikation erstellt, in der Objekte wird als REST Api bezeichnet.

1.1 Zu berücksichtigende Annotation

Als erstes benötigen wir eine Annotation, für die unser Prozessor im Compiler weitere Checks durchführen soll und weiteren Quelltext generieren und kompilieren soll. Es reicht aus, eine leere Annotationsschnittstelle zu definieren.

```
java: CheckDataObject

package name.panitz.util;

public @interface CheckDataObject {}
```

Anders als im letzten Lehrbrief die Annotation `VARCHAR` brauchen wir keine zusätzliche Angabe zur sogenannten **Retention** zu machen. Diese Annotation wird nur während der Kompilierung ausgewertet und steht zur Laufzeit nicht mehr zur Verfügung.

1.2 Beispielklassen

Als durchgehendes Beispiel nehmen wir wieder die beiden kleinen Beispielklassen aus dem letzten Lehrbrief. Jetzt sind sie allerdings zusätzlich mit der eben definierten Annotation `CheckDataObject` markiert.

Wir hatten einmal eine kleine Datenklasse für literarische Epochen:

java: Epoche

```
package name.panitz.util;

@CheckDataObject
public record Epoche(UID uid, String name, int jahrhundert)
    implements DataObject {
    public Epoche(String name, int jahrhundert){
        this(new UID(), name, jahrhundert);
    }
}
```

Und als zweites die Klassen um Autoren als Personen zu speichern, wobei hier eine Referenz auf die Epoche in einem Feld existiert:

java: Person

```
package name.panitz.util;

@CheckDataObject
public record Person(UID uid, @VARCHAR(200) String name
    ,String vorname, int gebJahr,Epoche epoche)
    implements DataObject {
    public Person(String name,String vorname,int gebJahr,Epoche epoche){
        this(new UID(), name, vorname, gebJahr, epoche);
    }
}
```

1.3 Prozessor

Jetzt schreiben wir einen Prozessor für Annotationen. Beginnen wir mit den benötigten Importanweisungen:

Java: DataObjectProcessor

```
package name.panitz.util;

import java.util.Set;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import javax.tools.Diagnostic.Kind;

import javax.lang.model.type.*;
import java.io.*;
```

Ein Prozessor, der den Compiler um weitere Auswertung von Annotation erweitert, lässt sich am einfachsten als Unterklasse von `AbstractProcessor` definieren. Als Anno-

tationen können diesen wieder Informationen über die von dem Prozessor ausgewerteten Annotationen und über die bearbeitete Quelltextversion mitgegeben werden.

Java: DataObjectProcessor

```
@SupportedAnnotationTypes("name.panitz.util.CheckDataObject")
@SupportedSourceVersion(SourceVersion.RELEASE_16)
public class DataObjectProcessor extends AbstractProcessor {
```

1.3.1 Die eigentliche Prozessormethode

Die abstrakte Methode, die zu überschreiben ist, heißt `process`. Sie erhält als Argument eine Menge der gefundenen Annotationstypen und eine Umgebung.

Java: DataObjectProcessor

```
@Override
public boolean process(Set<? extends TypeElement> as, RoundEnvironment env){
    try{
```

Wir lassen uns über die Umgebung alle Konstrukte des Quelltextes geben, die eine Annotation des Typs `CheckDataObject` haben und iterieren über diese.

Java: DataObjectProcessor

```
for (var element : env.getElementsAnnotatedWith(CheckDataObject.class)) {
```

Hier gehen wir vorerst davon aus, dass das annotierte Element ein `TypeElement` ist, also eine Klassendefinition, eine Schnittstellendefinition, oder eine Aufzählungs- oder Record-Klasse. Wir gehen davon aus, dass keine innere Klasse annotiert wurde, sprich das umschließende Element ein Paket und nicht eine äußere Klasse ist.¹

Java: DataObjectProcessor

```
String packageName = ((PackageElement)element.getEnclosingElement())
    .getQualifiedName().toString();
var cs = ((TypeElement)element);
```

Im Rahmen dieses Lehrbriefs werden wir für die annotierten Record-Klassen drei zusätzliche Schritte im Compiler vornehmen. Wir werden jeweils eine kleine Utility-Klasse mit statischen Methoden generieren, werden dann zusätzliche Checks vornehmen, ob die Record-Klassen alle Einschränkungen für Datenhaltungsklassen einhalten und schließlich in Form der Aufgabe werden die Servlet-Klassen zum Verwalten der Objekte in einer Datenbank generiert.

¹ Alle diese Annahmen könnten wir auch erst noch überprüfen.

Java: DataObjectProcessor

```
generateSetter(packageName, cs);  
checkArgumentTypes(packageName, cs);  
generateServlet(packageName, cs);  
}
```

Die Methode `process` verlangt einen Wahrheitswert als Ergebnis. Dieser gibt an, ob der Prozessor die annotierten Objekte exklusiv behandeln möchte oder auch anderen Prozessoren offen sein soll.

Java: DataObjectProcessor

```
return false;  
} catch (Exception e) { throw new RuntimeException(e); }  
}
```

Shell

```
javac -processor name.panitz.util.DataObjectProcessor Epoche.java
```

1.3.2 Generierung von Setter-Hilfsmethoden

Jetzt können wir loslegen und die im Prozessor verwendeten Methoden definieren, die dann bei der Kompilierung für die annotierten Klassen ausgeführt werden.

Als erstes soll gezeigt werden, wie weiterer Quelltext generiert werden kann, der dann direkt kompiliert wird. Für unsere Datenhaltungsklassen, die wir bearbeiten, soll Hilfsfunktionen generiert werden, die ein neues Objekt mit einem Wert neu gesetzt erzeugt. Dieses ist notwendig, wenn wir von einem Objekt ein Objekt ableiten wollen durch Änderung eines Feldes. In Recordklassen können Felder nicht neu gesetzt werden, so dass wir im Falle des Falles ein neues Objekt erzeugen müssen. Wir wollen zum Beispiel für `Epoche` folgende Klasse automatisch generieren:

java: EpocheUtility

```
package name.panitz.util;
public class EpocheUtil{
    static public Epoche setuid(Epoche dies, name.panitz.util.UID v){
        return new Epoche(v, dies.name(), dies.jahrhundert());
    }
    static public Epoche setname(Epoche dies, java.lang.String v){
        return new Epoche(dies.uid(), v, dies.jahrhundert());
    }
    static public Epoche setjahrhundert(Epoche dies, int v){
        return new Epoche(dies.uid(), dies.name(), v);
    }
}
```

Für jedes Feld der Recorsklasse wir also eine statische Methode erzeugt, die ein Objekt der Recordklasse erhält, einen neuen Wert für das Feld und damit ein neues Objekt aus den ursprünglichen Feldwerten und neuen Wert erzeugt.

Zunächst einmal betrachten wir den Klassennamen der annotierten Recordklasse, erzeugen daraus den Namen der zu generierenden Klasse, indem wir das Wort Util dran hängen und errechnen den vollqualifizierten Namen der zu generierenden Klasse.

Java: DataObjectProcessor

```
private void generateSetter(String packageName,TypeElement cs)
                                throws Exception{
    var simpleName = cs.getSimpleName();
    var name = simpleName+"Util";
    var fqn = packageName != null?packageName+"."+name:name;
```

Für den vollqualifizierten Klassennamen lässt sich aus der Umgebung des Prozessors einen Writer für die zu generierenden Quelltextdatei erzeugen.

Java: DataObjectProcessor

```
var builderFile = processingEnv.getFiler().createSourceFile(fqn);

var out = new PrintWriter(builderFile.openWriter()) ;
```

Und nun können wir damit beginnen, den Quelltext zu erzeugen. Zunächst einmal die Paketdeklaration:

Java: DataObjectProcessor

```
if (packageName != null) {
    out.append("package ").append(packageName).append("; \n");
}
```

Dann die Klassendeklaration:

Java: DataObjectProcessor

```
out.append("public class ").append(name).append("{");
```

Und schließlich für jedes Feld der Recordklasse wird eine statische Methode generiert, die ein neues Objekt der Recordklasse generiert:

Java: DataObjectProcessor

```
for (var rc:cs.getRecordComponents()){
    out.write("\n static public "+simpleName);
    out.write(" set"+rc.getSimpleName());
    out.write("(" + simpleName + " dies, "+ rc.asType() + " v){");
    out.write("\n     return new "+simpleName+"(");
```

Die Argumente des neuen Objektes werden entweder aus dem übergebenen Objekt dies übernommen, oer sind für das zu setzende Feld der übergebene Wert:

Java: DataObjectProcessor

```
var first=true;
for (var rc2:cs.getRecordComponents()){
    if (first) first=false; else out.write(", ");
    if (rc2.getSimpleName().equals(rc.getSimpleName())) out.write("v");
    else out.write("dies."+rc2.getSimpleName()+"()");
}
out.append(");").append("\n }");
}
```

Und damit ist die Generierung der Klasse schon beendet.

Java: DataObjectProcessor

```
out.write("\n}");
out.close();
}
```

1.3.3 Check auf korrekte Feldtypen

Nachdem wir gesehen haben, wie wir neuen Quelltext generieren und gleichzeitig wieder kompilieren lassen können, soll jetzt noch gezeigt werden, wie man zusätzliche Checks in der Kompilierung vornehmen kann. Hierzu kann hat man in der Umgebung des Prozessor Zugriff auf die Meldungen des Compilers bekommen. Wir gehen durch die Felder der

Recordklasse und checken deren Typen. Wenn die sich nicht mit den erlaubten Typen einer DataObject-Klasse vertragen, so meldet der Compiler nun einen Fehler.

Java: DataObjectProcessor

```
private void checkArgumentTypes(String packageName, TypeElement cs)
    throws Exception{
    for (var rc:cs.getRecordComponents()){
        if (!isAllowedFieldType(rc.asType()))
            processingEnv.getMessager().printMessage
                (Kind.ERROR, "illegal field type in DataObject " +
                    + rc.asType()+ " for field "+rc, cs);
    }
}
```

Die folgende Methode checkt, ob der Feldtyp einer Recordklasse für DataObject-Klassen erlaubt ist. Erlaubt sind alle primitiven Typen außer char, der Standardtyp String, der spezielle Typ UID und alle Typen, die DataObject implementieren.

Java: DataObjectProcessor

```
public static boolean isAllowedFieldType(TypeMirror type) {
    if (type instanceof DeclaredType dt) {
        if (dt.asElement() instanceof TypeElement ce) {
            return
                ce.toString().equals("name.panitz.util.UID")
                || ce.toString().equals("java.lang.String")
                || ce.getInterfaces().stream()
                    .anyMatch(i->(" "+i).equals("name.panitz.util.DataObject"));
        }
    } else if (type instanceof PrimitiveType){
        return !type.toString().equals("char");
    }
    return false;
}
```

1.4 Generierung eines Webservices mit Java Servlets

In diesem Abschnitt betrachten wir die Möglichkeit, mit Java-Programmen die Funktionalität eines Webservers zu erweitern. Damit lassen sich dynamische Webseiten erzeugen oder Webdienste realisieren. Eine Webadresse wird vom Server umgeleitet auf ein Java-Programm, das die entsprechende Antwortseite erzeugt.

1.4.1 Servlet Container

Standardmäßig unterstützt ein Webserver nicht die Fähigkeit Javaprogramme für bestimmte URLs aufzurufen. Hierzu ist der Webserver zunächst um eine Komponente zu erweitern, die ihn dazu befähigt. Eine solche Komponente wird *servlet container* bezeichnet. Einer der gebräuchlichsten *servlet container* ist der *tomcat*². Dieser kann genutzt werden um z.B. einen Webserver zu erweitern, so dass auf ihm Servlets laufen können. Der *tomcat* selbst ist jedoch auch bereits ein eigener Webserver und kann, so wie wir es in diesem Kapitel tun, auch als eigenständiger Webserver betrieben werden.

1.4.2 Servletkonfiguration

Die wichtigste Konfigurationsdatei für eine Webapplikation ist die Datei `web.xml`. In ihr wird die Webapplikation mit ihren Servlets beschrieben. Für jedes Servlet wird ein Name vergeben. Dieser Name wird gebunden an eine Servletklasse, die die entsprechende Funktionalität zur Verfügung stellt und an die URL über den auf das Servlet zugegriffen wird. Das Format der Datei `web.xml` ist in einer DTD definiert. In unseren Beispiel sehen wir zwei Servlets vor:

²<http://tomcat.apache.org/>

xml: web

```
<web-app
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  version="3.1">

  <servlet>
    <servlet-name>PersonServlet</servlet-name>
    <servlet-class>name.panitz.util.PersonServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>PersonServlet</servlet-name>
    <url-pattern>/Person/*</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>EpocheServlet</servlet-name>
    <servlet-class>name.panitz.util.EpocheServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>EpocheServlet</servlet-name>
    <url-pattern>/Epoche/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Das Servlet `EpocheServlet` ist an die Klasse `name.panitz.util.EpocheServlet` gebunden, die aufgerufen wird, wenn eine Anfrage an den Webserver kommt, die auf die URL des Servlets landet. In diesem Fall die URL nach dem Muster: `/Epoche/*`, die also mit dem Namen `/Epoche` beginnt und dann noch weitere Bestandteile haben kann.

Jetzt sind die entsprechenden Klassen zu schreiben, d.h. in unserem Beispiel die Klassen `EpocheServlet` und `PersonServlet`.

Als erstes Beispiel geben wir ein minimales Beispiel für eine Servletklasse. Hierzu schreiben wir eine Unterklasse der abstrakten Klasse `HttpServlet`.

java: `ExampleEpocheServlet`

```
package name.panitz.util;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ExampleEpocheServlet extends HttpServlet{
```

Nun lassen sich durch Überschreiben Methoden definieren, die auf die unterschiedlichen HTTP-Anfragetypen reagieren. Dieses `doGet`, `doPost`, `doPut` und `doDelete`, wobei die

letzten beiden seltener verwendet werden.

Überschreiben wir zunächst einmal die häufigst verwendete Methode `doGet`. Sie erhält zwei Argumente, die Objekte mit Informationen über die Anfrage und die zu gebende Antwort enthalten.

java: ExampleEpocheServlet

```
@Override
public void doGet
    (HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
```

Auf dem `HttpServletRequest` Objekt lassen sich viele Informationen über die Anfrage ermitteln. Zum Beispiel mit `getParameter`, ob in der Anfrage für einen Parameter ein Wert gesetzt wurde. Auch die URI und URL der Anfrage lassen sich hier abfragen. Wir lassen uns exemplarisch die URI geben:

java: ExampleEpocheServlet

```
String requestUrl = request.getRequestURI();
```

Auf dem Antwortobjekt, lässt sich zum Beispiel den Typ der Antwortdaten und ein Encoding setzen. Am wichtigsten ist aber, dass hier für das Schreiben der Antwortdaten mit `getOutputStream` ein Ausgabestrom und mit `getWriter` ein `Writer`-Objekt erhalten werden kann.

java: ExampleEpocheServlet

```
response.setContentType("text/xml");
response.setCharacterEncoding("UTF-8");
response.getWriter()

    ↪ .println("<info>request url was: "+requestUrl+" "+request.getRequestURL()+" "+request.
}
```

Häufig soll es keinen Unterschied machen, ob das Servlet per HTTP GET oder per HTTP POST angesprochen wurden und die Antwort auf ein POST verweist auf die Antwort auf einem GET.

java: ExampleEpocheServlet

```
@Override
public void doPost
    (HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    doGet(request, response);
}
}
```

1.4.3 Einrichten eines Servlets auf dem Tomcat

Jetzt kommen wir zu dem etwas kniffligeren Part: die kleine Webapplikation muss auf dem Tomcat bereitgestellt werden. Hierzu muss man wissen, wo die eigene Tomcat-Installation die Webseiten verwaltet. Dazu legt der Tomcat einen Ordner an. Im Falle von Linux ist das standardmäßig der Ordner:

`/var/lib/tomcat9/webapps.`

Hier kann man jetzt einen Ordner für die eigene Webapplikation erzeugen. Dieser soll dann Anwendername und Gruppenname tomcat haben:

Shell

```
panitz@px1:~$ sudo -u tomcat mkdir /var/lib/tomcat9/webapps/myApp
[sudo] password for panitz:
panitz@px1:~$ ls -l /var/lib/tomcat9/webapps/
total 7356
drwxr-xr-x 3 tomcat tomcat 4096 Mai 31 2020 ROOT
drwxrwxr-x 2 tomcat tomcat 4096 Mai 30 12:13 myApp
```

In diesem Ordner sind ein Unterordner WEB-INF, WEB-INF/classes und WEB-INF/lib zu erzeugen.

Shell

```
panitz@px1:~$ sudo -u tomcat mkdir /var/lib/tomcat9/webapps/myApp/WEB-INF
panitz@px1:~$ sudo -u tomcat mkdir
↪ /var/lib/tomcat9/webapps/myApp/WEB-INF/classes
panitz@px1:~$ sudo -u tomcat mkdir
↪ /var/lib/tomcat9/webapps/myApp/WEB-INF/lib
```

Die Datei `web.xml`, die wir oben vorgestellt haben, kommt in den WEB-INF Ordner.

Die class-Dateien unserer Applikation kommen in den `classes`-Ordner und eventuelle weitere verwendeten Javabibliotheken können als jar-Datei in den `lib`-Ordner abgelegt werden.

Shell

```
panitz@px1:~$ sudo -u tomcat cp -r classes/*  
↪ /var/lib/tomcat9/webapps/myApp/WEB-INF/classes/
```

Eine letzte Falle sind die Javaversionen. Der tomcat wird meist für eine JDK Version installiert, die das Installationsskript auf dem Rechner findet. Da wird mit JDK 16 Features arbeiten wollen, ist dieses auch dem Tomcat bekannt zu machen.

Der Tomcat installiert auf Linux seine Startskripte auf `/usr/share/tomcat9/bin/`.

Hier kann man ein Skript `setenv.sh` schreiben, dass die gewünschte Javaversion setzt. In meinem Fall hat es folgenden Inhalt:

```
sh: setenv
```

```
CATALINA_OPTS=---enable-preview
```

```
JAVA_HOME=/home/panitz/jdk-16/
```

Damit der Tomcat diese Umgebung korrekt berücksichtigt, sollte er neu gestartet werden:

Shell

```
panitz@px1:~$ sudo service tomcat9 restart
```

Wenn jetzt alles gut gegangen ist, haben wir einen lokalen Webserver, der für die URI `myApp/Epoche`, das Servlet für die Antwort verwendet:



2 Aufgaben

In dieser Aufgabe geht es jetzt darum, die Servletklassen für die Datenobjekte generieren zu lassen. Damit erhalten wir für jede Record-Klasse, die mit der Annotation versehen sind, einen Webdienst, um die Daten in eine Datenbank zu speichern, aufzulisten, anzuzeigen und zu löschen.

Aufgabe 1 In dieser Aufgabe ist die Funktion `generateServlet`, die für jede `DataObject` Recordklasse ein Servlet erzeugt zu vervollständigen.

Wir erzeugen den Kopf der Klasse:

Java: DataObjectProcessor

```
private void generateServlet(String packageName, TypeElement cs)
    throws Exception {
    var simpleName = cs.getSimpleName();
    var name = simpleName + "Servlet";
    var builderFile = processingEnv.getFile().createSourceFile(name);

    var out = new PrintWriter(builderFile.openWriter());
    if (packageName != null) {
        out.append("package ").append(packageName).append(";\n");
    }
    out.write(
        """
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import static name.panitz.util.DataObject.*;
public class """;
        out.write(" "+name+" ");
        out.write(
            """
extends HttpServlet{

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        response.setContentType("text/xml");
        response.setCharacterEncoding("UTF-8");
        String requestUri = request.getRequestURI();
        String rest = requestUri.substring((request.getContextPath()+"/"+"
            +simpleName+"/").length());");
```

Jetzt geht es darum aus dem Rest `rest` der angesprochenen URI die

- a) Eine Auflistung aller Objekte

Java: DataObjectProcessor

```
        out.write(
            """
            if (rest.startsWith("list")){
                //TODO
            }
            """;
```

- b) Ein Objekt mit einer bestimmten UID

Java: DataObjectProcessor

```
        out.write(  
        ""  
        }else if (rest.startsWith("get")){  
            //TODO  
        """);
```

c) Das Erzeugen der Datenbanktabellen

Java: DataObjectProcessor

```
        out.write(  
        ""  
        }else if (rest.startsWith("createTable")){  
            //TODO  
        """);
```

d) Das Speichern eines im Parameter object als XML kodierten Objekts.

Java: DataObjectProcessor

```
        out.write(  
        ""  
        }else if (rest.startsWith("save")){  
            //TODO  
        """);
```

e) Das Lösen eines Objekts

Java: DataObjectProcessor

```
        out.write(  
        ""  
        }else if (rest.startsWith("delete")){  
            //TODO  
        """);
```

Java: DataObjectProcessor

```
        out.write(
        ""
        }else {
            response
                .getOutputStream()
                .println
                    ("<info>no match "+rest+" in "+requestUri+"</info>");
        }
    }

    @Override
    public void doPost(HttpServletRequest rq, HttpServletResponse rp)
        throws IOException, ServletException {
        doGet(rq,rp);
    }
}

""");
    out.close();
}
```

Java: DataObjectProcessor

```
}
```