Die Kunst der Konfiguration (JSR 382)

Anatole Tresch, Trivadis AG

Im Oktober letzten Jahres wurde der JSR 382 [1] gestartet, der Konfiguration standardisieren soll. Höchste Zeit, uns den JSR etwas näher anzuschauen.

Was lange währt, wird endlich gut ...

Einen so zentralen Aspekt zu standardisieren, ist nicht einfach, denn die Meinungen darüber, wie Applikationen sinnvollerweise zu konfigurieren sind, gehen weit auseinander. Bereits die Repräsentation von Konfiguration als Schlüssel-Wertepaare („Map<String,String>“) gegenüber der Repräsentation als Baum (wie „java.util.Preferences“) kann unter Experten leicht zu abendfüllenden Diskussion führen.

Nimmt man dann noch bestehende System- und Umgebungsvariablen, Programm-Argumente, Speicherorte, -formate, Prioritäten und Übersteuerungsmechanismen hinzu, kann man erahnen, welche Vielfalt an Fragestellungen sich auftun. Entsprechend steht der aktuelle JSR am Ende einer Reihe von Versuchen, einen Standard für diese Problem-Domäne zu etablieren.

Im Alltag sind vermutlich viele mit Spring *[2]* und Java EE bereits in Berührung: Spring kennt beispielsweise mit „Environment“ und „PlaceHolderConfigurer“ aufgrund der Historie gleich mehrere SPIs und Java EE lässt standardmäßig nur Konfiguration zur Deployment-Zeit zu. Zudem existieren zahlreiche Projekte, die jedes für sich interessante Ansätze aufzuweisen haben. Die vielversprechendsten Ansätze, die auch gewisse Verbreitung gefunden haben, sind Apa-

che Tamaya *[3]* und MicroProfile Config *[4]*. Konsequenterweise finden sich Ideen aus beiden Projekten auch im JSR wieder. Weitere Details können dem Blog *[5]* entnommen werden.

Was ist Konfiguration?

Wie wird nun Konfiguration modelliert? Ganz einfach: als String-basierte Schlüssel-Wertepaare. Dieser Ansatz bringt gleich mehrere Vorteile mit sich:

- Das Konzept ist denkbar einfach
- Umgebungs- und System-Variablen sowie Programm-Argumente lassen sich einfach damit abbilden
- Der Ansatz ist kompatibel mit „java.util.Properties" und weitgehend auch mit den meisten anderen gängigen Formaten („xml", „ini", „yaml" und „json")
- Konfigurationswerte lassen sich einfach vergleichen und können problemlos serialisiert, gecacht, gespeichert oder übertragen werden
- Die Repräsentation als Text ist Plattform- und Sprach-unabhängig

Im neuen Standard werden Konfigurations-Properties über eine Instanz von „Config" abgefragt, die über das „ConfigProvider"-Singleton bezogen werden kann. Dabei kann auch ein „ClassLoader" übergeben werden, was vor allem in komplexeren Szenarien wie klassischem Java EE wichtig ist *(siehe Listing 1)*.

Konkrete Konfigurationswerte lassen sich mit drei Methoden abfragen: „getValue(String,Class)" setzt das Existieren eines entsprechenden Konfigurationseintrages voraus und wirft eine „NoSuchElementException", wenn kein Eintrag gefunden werden konnte. Ist ein Wert nicht zwingend, so sollte man „getOptionalValue(String,Class)" benutzen. Die Methode „access(String)" wird später im Zusammenhang mit dynamischer Konfiguration diskutiert. Möchte man wissen, welche Schlüssel aktuell definiert sind, lassen sich diese mit „getPropertyNames()" abfragen. Allerdings können nicht alle Konfigurationsquellen Auskunft über die definierten Schlüssel geben, sodass die zurückgelieferte Schlüsselliste nicht zwingend vollständig ist *(siehe Listing 2)*.

Den aufmerksamen Leserinnen und Lesern wird nun nicht entgangen sein, dass Werte als String repräsentiert werden, wir aber im obigen Beispiel die Konfiguration typisiert abfragen können. Hier fehlt also noch ein Baustein, der diese Lücke schließt, nämlich „Converter" *(siehe Listing 3)*. Alle gängigen Java-Basistypen sind bereits standardmäßig unterstützt. Will man weitere benutzerdefinierte Typen unterstützen, so lassen sich entsprechende Converter-Klassen ganz einfach mit den „ServiceLoader" registrieren.

Konfigurationsquellen und Ordinals

Wie setzt sich nun eine konkrete Konfiguration zusammen? Welche Konfigurationsquellen werden berücksichtigt, wie können Konfigurations-Properties einer Quelle die Properties einer anderen Quelle übersteuern? Dazu ist die „ConfigSource"-Schnittstelle genauer zu betrachten, die von allen Konfigurationsquellen implementiert werden muss *(siehe Listing 4)*.

```
// Access using current Thread classloader
Config config = ConfigProvider.getConfig();

// Access using explicit classloader
ClassLoader classloader = …
Config config = ConfigProvider.getConfig(classloader);
```

Listing 1

```
Iterable<String> getPropertyNames();

<T> T getValue(String key, Class<T> type);
<T> Optional<T> getOptionalValue(String key, Class<T>
type);
<T> ConfigValue<T> access(String key);
```

Listing 2

```
public interface Converter<T>{
    T convert(String value);
}
```

Listing 3

```
public interface ConfigSource {
   String CONFIG_ORDINAL = "config_ordinal";
   int DEFAULT_ORDINAL = 100;

   String getName();
   default int getOrdinal() {...}

   String getValue(String propertyName);
   default Set<String> getPropertyNames() {...}

   Map<String, String> getProperties();

   default void setOnAttributeChange(Consumer<Set<Str
ing>>
            reportAttributeChange) {...}
}
```

Listing 4

Wie zu erwarten, liefert eine „ConfigSource" Konfigurations-Properties. Wir sehen auch, dass die Properties wie erwähnt als „Strings" modelliert sind. Zudem besitzt jede Konfigurationsquelle einen Namen, der eindeutig sein sollte. Interessant ist die „getOrdinal"-Methode; sie definiert die Signifikanz einer Konfigurationsquelle.

Werte einer Quelle mit einem höheren Ordinal-Wert übersteuern Werte mit gleichem Schlüssel von Quellen mit tieferem Ordinal. Haben zwei Konfigurationsquellen denselben Ordinal-Wert, wird der vollqualifizierte Klassenname als zusätzliches Kriterium hinzugezogen, um immer eine definierte Reihenfolge garantieren zu können. Analog zu Convertern können eigene Konfigurationsquellen mit dem Java-„ServiceLoader" registriert werden. In vielen Fällen ist das jedoch gar nicht nötig, da standardmäßig bereits folgende Quellen mit entsprechenden Default-Ordinals automatisch zur Verfügung gestellt und registriert sind *(siehe Tabelle 1)*.

Da gewisse Betriebssysteme Umgebungsvariablen mit einem Punkt nicht unterstützen, sieht der JSR für Umgebungsvariablen (und nur für diese) einen erweiterten Lookup-Mechanismus vor. Der Schlüssel „com.ACME.size“ wird beispielsweise wie folgt aufgelöst (dabei wird der erste gefundene Wert als Resultat zurückgeliefert):

1. Exakter Match: „com.ACME.size“
2. Alle „.“ durch „_“ ersetzen: „com_ACME_size“
3. Alle „.“ durch „_“ ersetzen und zu Uppercase konvertieren: „COM_ACME_SIZE“

Nun könnte man den Eindruck erhalten, dass das alles doch sehr statisch ist. Dazu ein genauerer Blick auf die Default-Implementation der „ConfigSource#getOrdinal()“-Methode *(siehe Listing 5)*.

Wir können also den Ordinal-Wert unserer Konfigurationsquelle beeinflussen, indem wir einen Eintrag unter dem Schlüssel „config_ordinal“ mit dem gewünschten ganzzahligen Ordinal-Wert ablegen. Möchten wir zum Beispiel die Signifikanz von System-Properties gegenüber Environment-Properties umkehren (etwa um in einem Docker-Container Environment-Properties zum Übersteuern zu nutzen), so genügt es, mit „java -Dconfig_ordinal=200 ...“ ein entsprechendes System-Property zu setzen. Der Mechanismus ist natürlich analog auch auf andere Konfigurationsquellen anwendbar.

Quelle	Ordinal
System-Properties	400
Environment-Properties	300
META-INF/javaconfig.properties (Classpath Resource)	100

Tabelle 1

```
default int getOrdinal() {
   String configOrdinal = getValue(CONFIG_ORDINAL);
   if(configOrdinal != null) {
       try {
           return Integer.parseInt(configOrdinal);
       }
       catch (NumberFormatException ignored) {

       }
   }
   return DEFAULT_ORDINAL;
}
```

Listing 5

```
ConfigValue<String> value = …;
ConfigValue<Integer> intVal = value.as(Integer.class);
```

Listing 6

```
// Default value setzen und abfragen
ConfigValue<Integer> intVal =
                    value(as(Integer.class).withDefault(8080);
intVal = value.as(Integer.class).withStringDefault(“8080”);
Integer defaultIntValue = intVal.getDefaultValue();

// Abfrage Schlüssel und Werte
System.out.println(“Key  : “ + intVal.getKey());
System.out.println(“Value: “ + intVal.getValue());

Optional<Integer> optIntValue = intValue.getOptionalValue();
```

Listing 7

Dynamische Konfiguration und Konfigurationsänderungen

In vielen Fällen ist Konfiguration eine relativ statische Angelegenheit. In verteilten Systemen muss allerdings manchmal sehr rasch auf Konfigurationsänderungen reagiert werden können. Es kann auch sein, dass man eine gewisse Zeit vor Änderungen geschützt sein will. In beiden Fällen kann ein „ConfigValue“ benutzt werden, der über die bereits vorher erwähnte Methode „Config#access(String)“ mit „ConfigValue<String> value = ConfigProvider.getConfig().access (“app.host.port”);“ bezogen werden kann. „ConfigValue“ implementiert ein Builder-Muster, das im Zusammenhang mit einem Konfigurationswert folgende Aktionen ermöglicht:

- Typisiert (optional mit eigenen Converter) oder als „Set“ beziehungsweise „List“ zugreifen
- Mit einen Default-Wert versehen
- Mithilfe von Platzhaltern auflösen
- Mithilfe einer Lookup-Chain über mehrere Schlüssel abfragen
- Cachen
- Callback-Listener fur Konfigurationsanderungen hinzufügen

Zuerst ein genauer Blick auf die Methoden für den typisierten Zugriff. Dazu startet man mit der „ConfigValue“-Instanz aus dem vorigen Beispiel und typisiert den Wert als Integer *(siehe Listing 6)*.

```
ConfigValue<List<Integer>> listVals = intVal.asList();
ConfigValue<Set<Integer>> setVals = intVal.asList();
```

Listing 8

```
Converter<PortList> portListConverter = …;
ConfigValue<PortList> configuredPortList =
                    value.useConverter(portListConverter);
```

Listing 9

```
ConfigValue<T> value = …;
value = value.evaluateVariables(true);
```

Listing 10

```
String tenant = getCurrentTenant();
Integer timeout = config.access("some.server.url")
               .withLookupChain(tenant, ${projectStage})
               .getValue();
```

Listing 11

Ähnlich wie mit „Config" kann man den aktuellen Schlüssel beziehungsweise Wert als erforderlichen oder optionalen Wert abfragen. Wie erwähnt lässt sich auch ein Default-Wert setzen, und zwar entweder als String oder typisiert *(siehe Listing 7)*. Sind in der Konfiguration mehrere Werte durch Komma getrennt konfiguriert, kann man auf einen Wert auch als „List" oder „Set" zugreifen *(siehe Listing 8)*. Man kann explizit für diesen Konfigurationswert einen eigenen „Converter" setzen *(siehe Listing 9)*.

Um einen Konfigurationswert temporär zu cachen, kann die „ConfigValue#cacheFor(long,TimeUnit)"-Methode benutzt werden. Bei „ConfigValue<String> val = value.cacheFor(5,TimeUnit.MINUTES);" wird sich der Wert für die nächsten zehn Minuten nicht mehr verändern.

Platzhalter und Lookup-Chains

Der JSR unterstützt auch Platzhalter im UNIX-Stil, die ihrerseits wiederum auf andere Konfigurationseinträge zeigen können. Dazu ist auf einem „ConfigValue" die Methode „evaluateVariables(boolean)" anzuwenden, die einen neuen „ConfigValue" erzeugt *(siehe Listing 10)*. Bei Lookup-Chains werden mehrere Schlüssel in definierter Art und Weise evaluiert, um so typische Übersteuerungsmechanismen abzubilden. *Listing 11* zeigt dazu ein Beispiel.

Angenommen, „tenant" ist „myTenant" und „${projectStage}" evaluiert zu „Production", ergibt sich folgende Lookup-Reihenfolge:

1. „some.server.url.myTenant.Production"
2. „some.server.url.myTenant"
3. „some.server.url.Production"
4. „some.server.url"

Der effektiv aufgelöste Schlüssel lässt sich mit „ConfigValue #getResolvedKey()" abfragen.

```
interface ConfigChanged {
        <T> void onValueChange(String key, T oldValue,
T newValue);
}

ConfigValue<Integer> port = …;
port.onChange((k, o, n) -> { … });
```

Listing 12

```
ConfigBuilder builder = ConfigProvider.getInstance()
                              .getConfigBuilder();
Config config = builder
                        .withDefaultConverters()
                        .withDefaultSources()
                        .withDiscoveredSources()
                        .withSource(new MyConfig-
Source())
                        .build ;
```

Listing 13

```
@ConfigProperty
private Integer intVal;

@ConfigProperty(name="a.b.prop")
private Optional<Integer> optionalIntVal;

@ConfigProperty(name="a.b.prop", defaultValue="1234")
private Provider<Integer> providedIntVal;
```

Listing 14

Callbacks und „ConfigBuilder"

Schließlich kann auf einem „ConfigValue" auch ein Callback registriert werden, um über Konfigurationsänderungen informiert zu werden *(siehe Listing 12)*. In gewissen Szenarien möchte man den Lifecycle der „Config"-Instanzen nicht zentral verwalten lassen. Auch hierzu bietet der JSR eine Lösung an: den „ConfigBuilder". Mit diesem API können „Config"-Instanzen Schritt für Schritt definiert und geladen werden. Die erzeugten Konfigurationen sind vollkommen unabhängig und lassen sich nach Bedarf benutzen oder wieder dereferenzieren. Aus Platzgründen wird nicht weiter im Detail auf das API eingegangen, *Listing 13* zeigt allerdings ein kurzes Beispiel, wie es benutzt wird.

„@Config Injection"

Bisher haben wir uns auf das Java-SE-API konzentriert; der aktuelle JSR definiert jedoch auch ein Injection-API, das mit CDI entsprechend Java-EE-kompatibel ist. Die wichtigste Annotation ist dabei „@ConfigProperty". Mit ihr können Konfigurations-Properties in-

```
Config config = ConfigProvider.getConfig();
String host = config.getValue("service.host", String.class);
int port = config.getValue("service.port", int.class);
```

Listing 15

```
Consumer<Set<String>> configChangedListener = …;
config.registerConfigChangedListener(configChangedListener);
```

Listing 16

jiziert werden. Auch ist es möglich, Default-Werte zu setzen oder auf Werte als „Optional"- und „Provider"-Instanzen zuzugreifen. Ob auch „ConfigValue"-Instanzen injiziert werden können, ist aktuell noch offen *(siehe Listing 14)*.

Atomarität von Konfiguration

Zum Abschluss noch ein etwas weniger einfach ersichtlicher Aspekt: die Atomarität von Konfiguration beziehungsweise Konfigurationszugriffen. Dazu stellt man sich ein Szenario wie in *Listing 15* vor.

Wie gesehen, ist die aktuelle Konfiguration durch eine nach Ordinal geordnete Liste von „ConfigSource"-Instanzen definiert. Diese Liste wird bei jedem Zugriff durchlaufen, um den aktuell richtigen, konkreten Wert zu evaluieren. Nun stellt sich die Frage, wie sich sicherstellen lässt, dass zwischen dem Zugriff auf „host" und dem auf „port" keine Konfigurationsänderung in einer „ConfigSource" stattfindet, die unter Umständen in einer inkonsistenten „host:port"-Kombination enden kann. Der JSR macht hierzu selbst keine Garantien, allerdings kann auf einer „Config"-Instanz ein Listener registriert werden, der bei Änderungen entsprechend informiert wird *(siehe Listing 16)*.

Somit kann betroffener Client-Code auf relevante Änderungen hören. Damit das funktioniert und während des Aufrufs eines Listeners selbst keine weiteren Konfigurationsänderungen möglich sind (der Zugriff während dieser Zeit also atomar ist), muss die Implementation der „Config"-Klasse transitiv auch von den registrierten „ConfigSources" Atomaritätsgarantien erhalten. Dies führt uns zur Methode „ConfigSource#onAttributeChanged(Consumer <Set<String>>)". Diese Methode erlaubt es, Änderungen in einer registrierten „ConfigSource" an die entsprechende „Config"-Instanz anzuzeigen.

Ob eine Änderung effektiv zu einer veränderten Konfiguration führt, hängt natürlich von Anzahl, Inhalt und Reihenfolge (Signifikanz/Ordinal) aller „ConfigSource"-Instanzen ab. Für die umgebende „Config"-Instanz ist dieser Mechanismus allerdings ausreichend. Wird nämlich während der Evaluation eines Wertes eine Änderung einer „ConfigSource" angezeigt, muss die „Config"-Instanz den evaluierten Wert verwerfen und zuerst alle registrierten Listener über eine allfällige Änderung informieren. Somit ist die Atomarität grundsätzlich gegeben. Dies erfordert aktuell einiges an Client-Code. Deshalb werden diese Fragestellungen derzeit im JSR intensiv diskutiert und es bleibt abzuwarten, ob hier noch das API erweitert wird oder man für diese Anwendungsfälle auf Vendor-spezifische APIs zurückgreifen muss.

Kritik und Ausblick

Grundsätzlich macht der Standard gute Fortschritte und es ist zu erwarten, dass in naher Zukunft ein „Early Draft Review" verfügbar sein wird. Das definierte API ist recht einfach gehalten und es kann eine Mehrheit der Anwendungsfälle unterstützt werden. Natürlich gibt es noch einiges an Feinarbeit zu leisten, und dazu ist auch ein breites Feedback der Community wichtig. Aber es scheint so, dass Java es nach vielen Startschwierigkeiten in naher Zukunft tatsächlich schafft, als erstes Ecosystem diesen wichtigen Cross-Cutting-Concern zu standardisieren.

Weiterführende Links

[1] *https://jcp.org/en/jsr/detail?id=382*
[2] *https://cloud.spring.io/spring-cloud-config/*
[3] Apache Tamaya: *http://tamaya.incubator.apache.org*
[4] *https://github.com/eclipse/microprofile-config*
[5] *http://javaeeconfig.blogspot.ch/2014/08/overview-of-existing-configuration.html*
[6] *https://github.com/eclipse/ConfigJSR*

Anatole Tresch
anatole.tresch@trivadis.com

Anatole Tresch war nach dem Wirtschaftsinformatik-Studium an der Universität Zürich mehrere Jahre lang als Managing Partner, Berater, Lead Engineer und technischer Architekt tätig. Er sammelte weitreichende Erfahrungen in allen Bereichen des Java-Ökosystems von IOT bis Java EE. Aktuell ist er Principal Consultant bei der Trivadis AG und beschäftigt sich dort mit Cloud-Architektur, Resilient Design, DevOps und verteilten Systemen allgemein. Zudem ist er JCP-Star-Specification-Lead und aktiver Apache-PPMC-Member.