D° Policysystem V1

Versionen

Datum	Notiz	Version
30.09.2019	Erste Veröffentlichung Vertikaler Prototyp des D° Policysystems	1.0

Autoren

Name	Institution	Email
Fabian Bruckner	Fraunhofer ISST	fabian.bruckner@isst.fraunhofer.de

Über das D° Policysystem

Eines der zentralen Ziele von D° ist es, die korrekte Verwendung von Policies zu vereinfachen und somit einer größeren Menge von Anwendern (Entwicklern) zu ermöglichen. Aus diesem Grund sieht D° eine Trennung der Anwender in zwei unterschiedliche Gruppen vor. Dabei ist zu beachten, dass diese Trennung nicht verpflichtend ist und Personen durchaus zu beiden Gruppen zugehörig sein können. Dennoch soll die Verwendung bzw. Erweiterung von D° auch möglich sein, wenn man nur zu einer der beiden Gruppen zugehörig ist.

Sprachanwender

Der Sprachanwender ist am ehesten vergleichbar mit einem Entwickler bei anderen Programmiersprachen. Er verwendet die Konstrukte und Elemente, welche in D° zur Verfügung stehen, um individuelle Data Apps zu erzeugen und auszuführen. Dabei benötigt der Sprachanwender keinerlei Kenntnisse über andere Programmiersprachen, oder die Mechanismen zur Erweiterung von D°.

Auch wenn Kenntnisse aus anderen Programmiersprachen den Einstieg in D° erleichtern können, sind ausschließlich Kenntnisse über die Entwicklung von Data Apps notwendig, um diese Rolle auszuführen. Soweit dies möglich ist, wird die Komplexität, welche durch die in D° verwendeten Konzepte ensteht vom Sprachanwender weg abstrahiert.

Sprachentwickler

Eine komplexere Rolle als der Sprachanwender haben die Sprachentwickler inne. Ihm kommen mehrere Aufgaben zu, welche eine unterschiedliche Komplexität aufweisen.

- Definition neuer Datentypen
    - Neben dem notwendigen Wissen über Semantik und Syntax des verwendeten Nukleus-Typsystems ist entweder eine Spezifikation der zu definierenden Datentypen notwendig, oder aber ein ausreichendes Verständnis der jeweiligen Fachdomäne, um passende Datentypen zu definieren.
- Definition und Implementation von Aktivitäten
    - Eine D°-Aktivität besteht aus zwei Komponenten. Zum einen ist eine Definition im Nukleus-Typsystem vorzunehmen und zum anderen muss eine Implementierung in Java oder Kotlin (- andere JVM Sprachen sind nicht getestet -) entwickelt werden, welche die eigentliche Funktionalität bereitstellt. Somit ist Wissen über die Verwendung des Nukleus-Typsystems (sowohl Definition neuer Elemente, als auch die Verwendung in der verwendeten Programmiersprache) notwendig. Desweiteren müssen die von D° gelieferten Annotationen und Schnittstellen bekannt sein, um eine korrekte Implementation sicherzustellen. Des Weiteren ist es je nach gewünschter Funktionalität der Aktivität notwendig interne Systeme von D° zu kennen um sie korrekt verwenden zu können. Dies trifft beispielsweise dann zu, wenn eine Aktivität nachverfolgbar Daten persistieren soll, die in der gesamten Data App zur Verfügung stehen und von anderen Aktivitäten und Policies verwendet werden können. Dies trifft auch dann zu, wenn Funktionalitäten verwendet werden sollen, welche einen Schutz durch den D°-SecurityManager erlauben, da in diesen Situationen zwingend die von D° bereitgestellten Schnittstellen verwendet werden müssen und die direkte Verwendung der verfügbaren JDK-Methoden nicht möglich ist.
- Definition und Implementation von Policies
    - Neben dem Wissen, das notwendig ist um Aktivitäten zu definieren und implementieren sind zusätzliche Kenntnisse zur Entwicklung neuer Policies notwendig. Es ist ein vollständiges Verständnis des D°-Policysystems notwendig, um die Zusammenhänge zwischen den einzelnen Evaluierungszuständen einer Policy zu erfassen. Unterschiedliche Zustände können unterschiedliche Eigenschaften überprüfen und durchsetzten und müssen in manchen Situationen kombiniert werden, um zum gewünschten Ergebnis zu kommen.

Es zeigt sich somit, dass der Sprachentwickler weitergehende Kenntnisse als der Sprachanwender benötigt, da für die Entwicklung mancher Konstrukte tiefgehendes Wissen über die interne Funktionalität von D° notwendig ist. Dieses Dokument gibt tiefgehende Informationen über das D° Policysystem und dient damit der Gruppe der Sprachentwickler als Hilfestellung bei der Entwicklung neuer Policies.

Die D°-Policy

Bevor die Funktionsweise des D°-Policysystems betrachtet wird, ist es wichtig den Begriff der Policy im Kontext von D° zu definieren.

Hierfür soll als erstes betrachtet werden, an welchen Stellen von Data Apps Policies verwendet werden können.

Zum einen ist es möglich Policies mit Aktivitäten zu verbinden. Eine Evaluierung findet dann bei jedem Aufruf der entsprechenden Aktivität statt. Falls die Evaluierung fehlschlägt wird die Ausführung der Data App abgebrochen. Bei Data Apps die dauerhaft laufen und über eine Schnittstelle (bspw. REST) von außen aufgerufen werden, wird nicht die komplette Data App abgebrochen sondern nur die Ausführung des aktuellen Requests.

Darüber hinaus können Policies mit Data Apps verknüpft werden und dienen dann als sogenannte Start-up-Policies. Diese Start-up-Policies werden vor dem Start der Applikation einmalig überprüft und verhindern vollständig die Ausführung der Data App, falls sie nicht erfüllt sind.

Als nächstes soll betrachtet werden, aus welchen Bestandteilen, die für die Durchsetzung relevant sind, eine D°-Policy besteht. Anschließend werden die verschiedenen Arten von Policies, die in D° unterschieden werden, betrachtet.

Aufbau

Policies in D° besitzen drei Bestandteile, welche an der Durchsetzung beteiligt sind und zu unterschiedlichen Zeitpunkten ausgewertet werden.

Precondition

Damit eine Policy ausgeführt wird, muss sie mit einem anderen Konstrukt von D° verknüpft werden, beispielsweise einer Aktivität. Die Precondition erlaubt es Vorbedingungen zu überprüfen, da die Precondition in jedem Fall vor dem verknüpften Konstrukt ausgewertet wird. Die Precondition kann für die Durchsetzung vieler verschiedener Policies verwendet werden und stellt, neben der Security Manager Intervention, den zentralen Durchsetzungspunkt für Policies dar.

Die Precondition eignet sich gut dazu, um Policies durchzusetzen, welche bei jeder Ausführung des verknüpften Elements evaluiert werden müssen. Beispiele hierfür wären Zugriffszähler, Einhaltung von erlaubten Ausführungs-Zeiträumen und Überprüfungen von Nutzer- und Rollenberechtigungen.

Postcondition

Die Postcondition einer D°-Policy ist weniger als eine Nachbedingung zu verstehen, sondern vielmehr als Möglichkeit für die Policy ihren internen Status anzupassen oder eventuell anfallende Daten zu persistieren bzw. für andere Teile der Data App verfügbar zu machen. Es ist nicht für jede Situation sichergestellt, dass die Postcondition nach der Ausführung des verknüpften Konstrukts ausgewertet wird. Dies hängt davon ab, mit welcher Art von Konstrukt die Policy verknüpft ist.

Es ist lediglich für Policies, die mit Aktivitäten verknüpft sind sichergestellt, dass die Postcondition nach der erfolgreichen Ausführung der Aktivität ausgewertet wird.

Aus diesem Grund sollten (kritische) Policies sofern möglich in Preconditions und Security Manager Interventions überprüft werden. Eine Ausnahme bilden Fälle, in denen das Ergebnis der Evaluation vom Berechnungsergebnis der verknüpften Aktivität abhängen und eine frühere Entscheidung nicht getroffen werden kann. Bspw. kann erst dann überprüft werden, ob eine Policy, welche ein bestimmtes Format für einen Rückgabewert vorschreibt, erfüllt ist, wenn der entsprechende Rückgabewert auch zur Verfügung steht.

Security Manager Intervention

Diverse Schnittstellen des JDK (bspw. Datei- und Netzwerkoperationen) erlauben es, vor der Ausführung bestimmter Methoden beim `SecurityManager` die entsprechende Berechtigung anzufragen. Die entsprechenden Methoden werden dann nur ausgeführt, sofern die notwendigen Berechtigungen gewährt werden. Der `SecurityManager` ist seit dem JDK 1.0 ein Bestandteil des JDK.

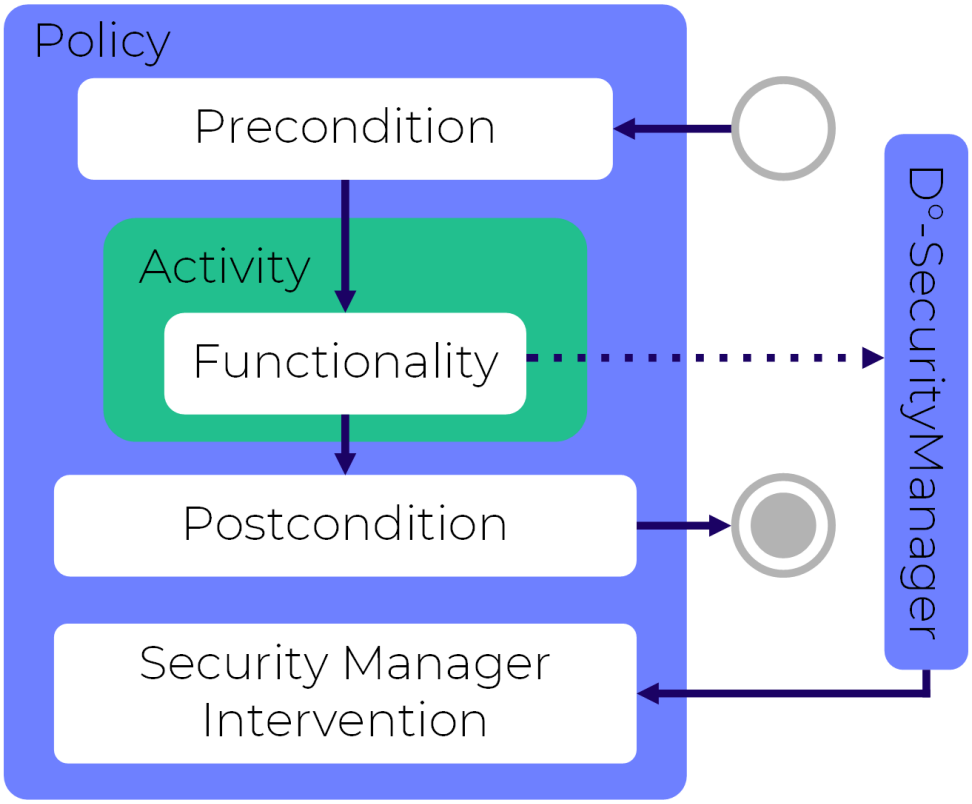
Für D° ist ein eigener Security Manager implementiert worden, welcher diese Rolle für D° einnimmt. Er definiert, welche Berechtigungen zur Ausführung der im JDK entsprechend instrumentierten Methoden notwendig sind. Diese Anfragen geschehen ausschließlich bei der Ausführung von D°-Aktivitäten, welche die entsprechenden Schnittstellen verwenden. Und es wird auch nur dann eine Anfrage gestellt, wenn die Schnittstelle tatsächlich aufgerufen wird. Enthält eine Aktivität verzweigten Programmcode, kann es passieren, dass eine solche Anfrage nicht gestellt wird, auch wenn der Code geschützte Schnittstellen auf anderen Ausführungspfaden enthält.

Sobald der Security Manager definiert hat, welche Berechtigungen für den Aufruf der Schnittstelle notwendig sind, wird allen Policies, die an der Aktivität hängen, die Möglichkeit gegeben im Rahmen der Security Manager Intervention bestimmte Zugriffe zu erlauben bzw. zu verweigern. Diese Sammlung von notwendigen Berechtigungen, Erlaubnissen und Verboten wird in einem nachgelagertem Schritt evaluiert und entscheidet darüber, ob der Aufruf der Schnittstelle durchgeführt wird, oder die Ausführung abgebrochen wird.

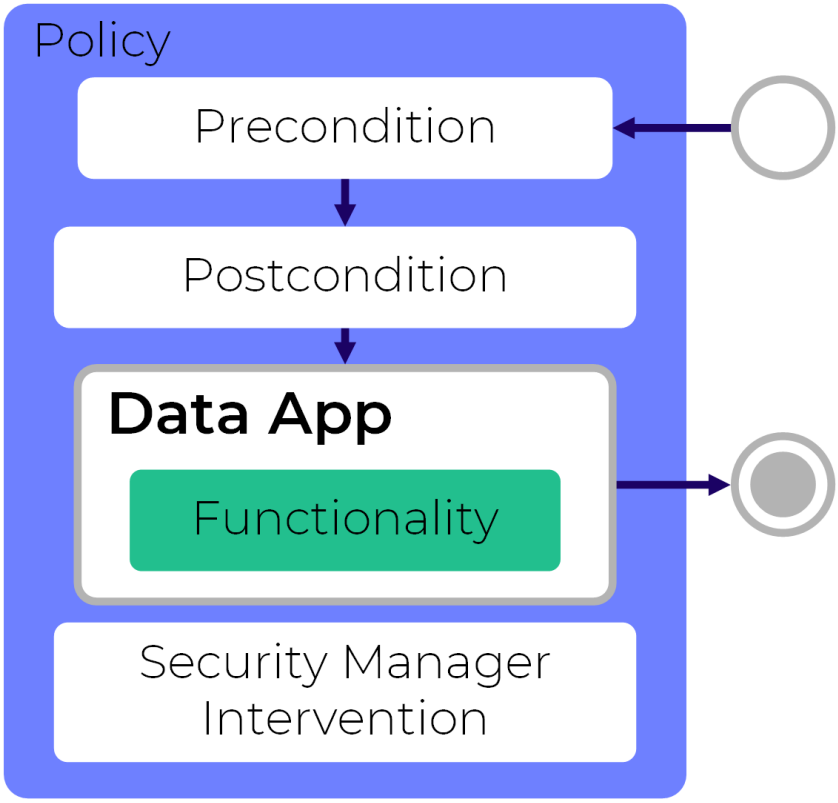
Security Manager Interventions unterscheiden sich in den folgenden Punkten von Preconditions, wodurch sie auch eine andere Menge von Policies durchsetzten können:

- Eine Security Manager Intervention findet immer nur dann statt, wenn die entsprechend geschützte Schnittstelle tatsächlich aufgerufen wird.
- Im Rahmen der Security Manager Intervention stehen bereits Informationen über die durchzuführende Aktion durch. Beispielsweise ist bekannt welche Datenmenge bei einem lesenden Zugriff auf eine Datei tatsächlich gelesen wird, obwohl die Funktion noch nicht ausgeführt wurde. Hierdurch lassen sich Anfragen verhindern, welche von der Precondition initial erlaubt wurden, aber während der Ausführung ungültig werden würden. Wird bspw. versucht eine 10 MB Datei zu lesen, aber es steht nur noch eine Quota von 1 MB zur Verfügung, würde die Precondition den Zugriff erlauben, da die Quota noch nicht ausgeschöpft ist und die Dateigröße nicht bekannt ist. Die Security Manager Intervention kennt die Dateigröße und kann den Zugriff ablehnen, da er die genehmigte Quota übersteigen würde.
    - Es wäre in einigen Fällen möglich diese Informationen bereits bei der Evaluation der Precondition zu erlangen, dies würde aber bei Aktivitäten, welche über mehrere Ausführungspfade verfügen unter Umständen nicht korrekt funktionieren, da die lesende Dateioperation aus obigem Beispiel nicht zwingend auf jedem Ausführungspfad vorhanden sein muss. Hierdurch kann ungewolltes verhalten entstehen, bspw. indem Quota verbraucht wird obwohl die Quota verbrauchende Operation gar nicht ausgeführt wurde.
- Während Pre- und Postcondition durch ihren Rückgabewert direkt über die weitere Ausführung der aktuellen Ausführung entscheiden, geben Security Manager Interventions eine Menge von Berechtigungen und Verboten zurück, welche im Anschluss mit den notwendigen Berechtigungen abgeglichen werden. Der direkte Abbruch der Ausführung kann in Fällen, in denen sicher ist, dass die Policy nicht erfüllt ist, über die Verwendung der `DegreeForbiddenSecurityFeatureException` erreicht werden.
- Wird die Ausführung aufgrund einer Security Manager Intervention abgebrochen, ist die Aktivität bereits partiell ausgeführt worden. Da D°-Aktivitäten keine Transaktionen sind, kann die bisherige Ausführung nicht zurückgerollt werden.

Dieser Ablauf ist nochmals in der nachfolgenden Darstellung abgebildet.



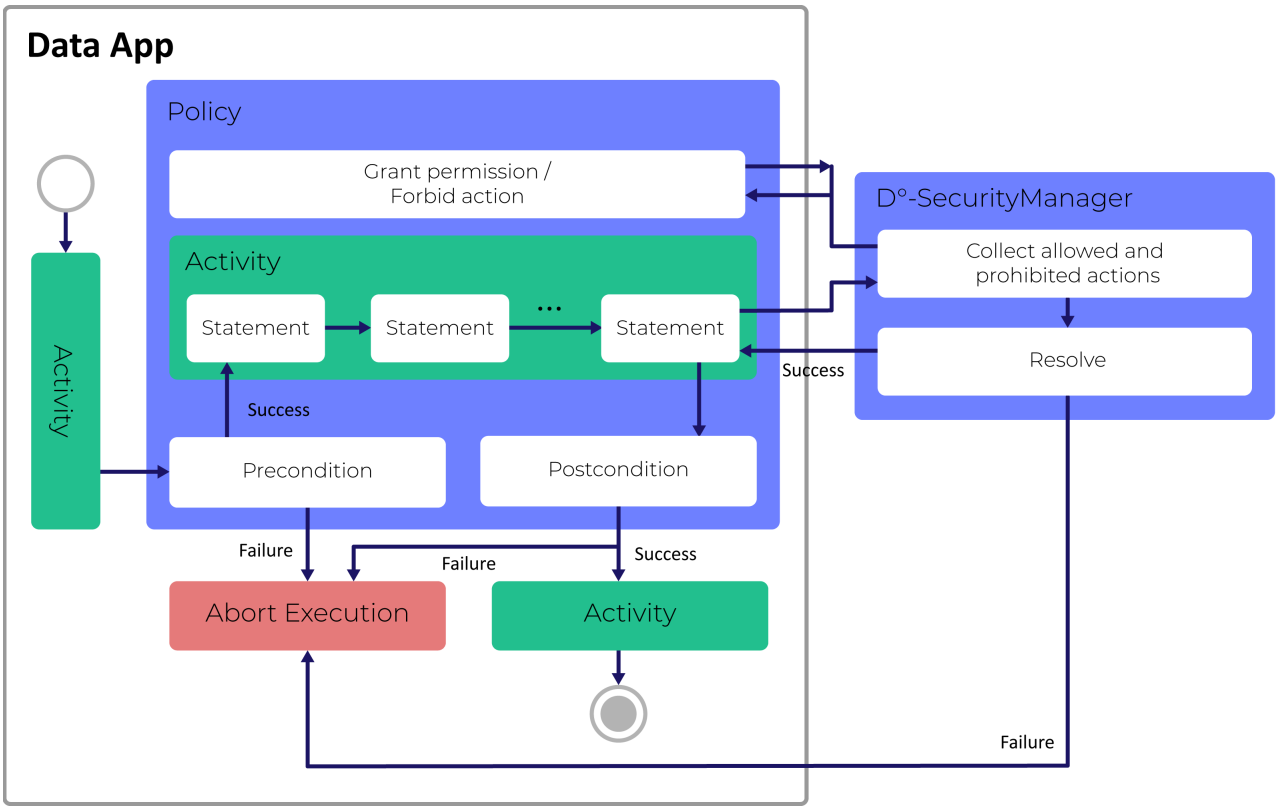
Wie bereits zuvor erwähnt, treten Security Manager Interventions nur bei Policies auf, welche mit Aktivitäten verknüpft sind. Bei Start-up-Policies können solche Anfragen nicht auftreten. Stattdessen beschränkt sich die Evaluierung der Policies auf Pre- und Postcondition. Dabei werden Pre- und Postcondition direkt hintereinander evaluiert und im Erfolgsfall die Data App gestartet. Dieser Ablauf ist in der folgenden Abbildung dargestellt.



In der nachfolgenden Abbildung ist die Ausführung einer Beispiel Data App schematisch dargestellt. Die Data App verfügt über drei Aktivitäten und keine Start-up-Policies. Die zweite Aktivität ist mit einer Policy verknüpft welche entsprechend in den Ausführungspfad eingewoben wird. Das letzte Statement der durch die Policy geschützten Aktivität verwendet eine der geschützten Schnittstellen, was zu einer Security Manager Intervention führt.

Im Rahmen dieser Intervention holt der D°-SecurityManager von der Policy Erlaubnisse und Verbote ein und gleicht diese anschließend mit der/den notwendige/n Bedingung/en ab.

Es ist sichtbar, dass die Data App drei Stellen hat, an denen die Ausführung vorzeitig abgebrochen werden kann. Diese drei Stellen lassen sich alle zu den einzelnen Phasen der einzigen Policy der Data App zurückverfolgen.



Constraint

Zum einen existieren Constraints, welche eine Prüfbedingung enthalten, welche während der Programmausführung überprüft wird. Jede Constraint besteht aus einer Definition im Nukleus-Typsystem und einer dazugehörigen Implementation, welche die eigentliche Überprüfung durchführt.

Bei der Implementierung muss die entsprechende Schnittelle verwendet werden, um zu gewährleisten, dass die Constraints während der Ausführung verwendet werden können. Die nachfolgende Abbildung zeigt die Definition der Schnittstelle für Constraints in der Programmiersprache Kotlin.

```kotlin
package de.fhg.isst.oe270.degree.core.policies.api

import de.fhg.isst.oe270.degree.core.policies.execution.PolicyInputScope
import de.fhg.isst.oe270.degree.registry.instances.api.DegreeJavaApiImplementation
import de.fhg.isst.oe270.degree.runtime.java.security.resolving.EvaluationCondition

/**
 * Uniform API which all Policies and Constraints must provide for functionality.
 *
 * @author <a href="mailto:fabian.bruckner@isst.fraunhofer.de">Fabian Bruckner</a>
 */
interface EmbeddedPolicyApi : DegreeJavaApiImplementation {

    fun acceptPrecondition(policyInput : PolicyInputScope) : Boolean

    fun evaluateSecurityManagerIntervention(input: PolicyInputScope): Collection<EvaluationCondition>

    fun acceptPostcondition(input : PolicyInputScope) : Boolean

}
```

Es ist erkennbar, das drei Methoden implementiert werden müssen, welche sich einfach dem Aufbau von D°-Policies zuordnen lassen. Darüber hinaus ist es notwendig, dass die Implementation mit der Java-Annotation `PolicyAnnotation` annotiert wird. Diese Annotation erhält einen Parameter, welcher angibt zu welcher Definition die annotierte Klasse die Implementierung liefert. Dies ist zwingend notwendig, um die Definitionen im Nukleus-Typsystem mit der Implementierung zu verbinden.

Die nachfolgende Abbildung zeigt die Implementation der Annotation in der Programmiersprache Kotlin.

```kotlin
package de.fhg.isst.oe270.degree.core.policies.annotations

/**
 * Simple annotation which is used during runtime to find annotated D°-Constraints.
 *
 * @author <a href="mailto:Fabian.Bruckner@isst.fraunhofer.de">Fabian Bruckner</a>
 */
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
@Repeatable
annotation class PolicyAnnotation(val qualifiedName: String)
```

Beispiel

Nachfolgend sollen die zuvor genannten Schnittstellen und Konzepte an einem Beispiel verdeutlicht werden. Als Beispiel wurde dabei eine Constraint gewählt, welche sicherstellt, dass beim Schreiben einer Datei eine gegebene Quota eingehalten wird.

As erstes wird die Definition im Nukleus-Typsystem betrachtet, welche im folgenden Codeblock erkennbar ist. Die Definition liegt im yaml-Format vor. An dieser Stelle soll nicht näher auf die einzelnen Parameter oder die Struktur der Definition eingegangen werden, da sich dies außerhalb des Rahmens dieses Dokuments befindet.

QuotaWriteFile - Definition

```
QuotaWriteFile:
  degree.Constraint@QuotaWriteFile:
    name:
      Identifier: "QuotaWriteFile"
    attribute:
      degree.Parameter:
        - name:
            Identifier: "quota"
          type:
            Type: "UnsignedInteger"
        - name:
            Identifier: "unit"
          type:
            Type: "ByteUnit"
        - name:
            Identifier: "path"
          type:
            Type: "Path"
        - name:
            Identifier: "matchingStrategy"
          type:
            Type: "PathMatchingStrategy"
```

Als erstes soll die Signatur der Klasse betrachtet werden, welche in der nachfolgenden Abbildung zu sehen ist. Es ist erkennbar, dass die notwendige Annotation an der Klasse vorhanden ist. Dies sorgt dafür, dass die Klasse mit der korrekten Definition verknüpft wird. Darüber hinaus wird das Interface `EmbeddedPolicyApi` implementiert, welches die notwendigen Schnittstellen enthält.

```kotlin
@PolicyAnnotation("QuotaWriteFile")
class QuotaWriteFile: EmbeddedPolicyApi {
```

Als nächstes wird die Precondition der Constraint betrachtet. Wie zuvor dargelegt stehen während der Evaluation der Precondition noch keine detaillierten Informationen über die tatsächliche Ausführung zur Verfügung, da die Auswertung vor der Ausführung der verknüpften Aktivität stattfindet.

Aus diesem Grund wird in der Precondition nur überprüft, ob die genehmigte Quota bereits überschritten wurde. Die beiden Datentyp-Instanzen `quotaInstance` und `unitInstance` sind beide im `policyInput` enthalten, welcher an die Constraint übergeben wird.

```
25  override fun acceptPrecondition(policyInput: PolicyInputScope): Boolean {
26      Parameter validation
56
57      val availableQuota = ByteUnitUtils.toByte(
58              quotaInstance.read().toLong(),
59              unitInstance.read()
60      )
61
62      return if (readBytes < availableQuota) {
63          true
64      } else {
65          logger.error("Validation failed. Granted quota is ${quotaInstance.read()}${unitInstance.read()} but " +
66                  "${ByteUnitUtils.toBytePrefix(readBytes, unitInstance.read())}${unitInstance.read()}.")
67          false
68      }
69  }
```

Sobald der schreibende Zugriff auf die entsprechende Datei stattfinden soll, wird die Ausführung unterbrochen und die Constraint kann ihre Security Manager Intervention durchführen. An dieser Stelle stehen auch die notwendigen Informationen über den Zugriff zur Verfügung und können verwendet werden. In diesem Beispiel basiert `quotaUsedByThisRequest` auf diesen Informationen und wird verwendet um zu entscheiden, ob der Aufruf die gewährte Quota überschreiten würde.

Falls dem so ist wird die Ausführung mit einer Exception beendet. Andernfalls wird eine `EvaluationCondition` zurück gegeben, welche den schreibenden Zugriff auf diese eine Datei erlaubt.

Eine `EvaluationCondition` ist ein internes Konstrukt aus dem D°-Policysystem, welches fein granular Verbote und Erlaubnisse für Aktionen erteilen kann.

Die erzeugten `EvaluationConditions` werden anschließend mit den notwendigen Berechtigungen abgeglichen, um zu entscheiden ob die Ausführung erlaubt wird. Diese Auswertung findet nicht in der Constraint selber statt.

```
69  override fun evaluateSecurityManagerIntervention(input: PolicyInputScope): Collection<EvaluationCondition> {
70      Parameter validation
101
102     // we need to check if this action would exceeds the granted quota
103     return if ((quotaUsedByThisRequest + readBytes) > givenQuota) {
104         val exceedValue = quotaUsedByThisRequest + readBytes - givenQuota
105         throw DegreeForbiddenSecurityFeatureException("The execution is aborted because a requested file operation " +
106                 "operation exceeds the granted quota of ${quotaInstance.read()}${unitInstance.read()}" +
107                 " by ${exceedValue}B.")
108     } else {
109         listOf(
110             EvaluationCondition(
111                 getPermissionType(),
112                 input.get("path")!!.read(),
113                 matchingStrategy,
114                  forbid: false
115             )
116         )
117     }
118 }
```

Als letztes wird die Postcondition ausgeführt, sobald die Ausführung der Aktivität abgeschlossen ist. Dabei wird in diesem Beispiel im Rahmen der Postcondition keine Evaluation vorgenommen. Dies ist in diesem Szenario auch nicht notwendig, da bereits eine Prüfung durch Precondition und Security Manager Intervention stattgefunden hat. Stattdessen wird aktualisiert, welcher Anteil der Quota bereits aufgebraucht ist.

In der Abbildung ist ein Zugriff auf den `PermissionScope` zu erkennen. Hierbei handelt es sich um eine zentrale Datenstruktur des D°-Policysystems, welche aktuell notwendige Berechtigungen sowie erteile Verbote und Erlaubnisse enthält. Darüber hinaus kann die Struktur verwendet werden, um Daten zwischen den einzelnen Evaluierungsphasen von Policies auszutauschen, oder um zusätzliche Informationen für die Auswertung von Policies bereitzustellen. In diesem Beispiel wird im `PermissionScope` hinterlegt wie groß die zu schreibende Datenmenge ist. Hierdurch stehen die Daten ab dem Zeitpunkt, an dem die SecurityManagerIntervention stattfindet, zur Verfügung.

```
122     override fun acceptPostcondition(input: PolicyInputScope): Boolean {
123         Parameter validation
149
150         readBytes +=  PermissionScope.getInstance().evaluationData[getByteOptionName()] as Long
151         return true
152
153     }
```

Policy

Im Gegensatz zu Constraints, besitzen Policies im Kontext von D° keine Implementierung. Eine Policy in D° wird lediglich zur Strukturierung verwendet. Eine D°-Policy enthält eine beliebige Anzahl weiterer Policies und Constraints. Während der Evaluation einer Policy werden alle enthaltenen Elemente nacheinander evaluiert und das Evaluationsergebnis ist die Boolesche Und-Verknüpfung der einzelnen Elemente. Sobald für ein enthaltenes Element die Evaluation fehlschlägt, schlägt die Evaluation der gesamten Policy fehl und wird abgebrochen.

In den meisten Szenarien in D° können Policies und Constraints ohne Anpassung miteinander ausgetauscht werden. Beispielsweise können sowohl Constraints als auch Policies mit Aktivitäten verknüpft werden. Aus diesem Grund werden die beiden Begriffe in diesem Dokument häufig synonym verwendet.

Somit werden in D° komplexe Policies erstellt, indem fein granulare, überprüfbare Bedingungen (Constraints) miteinander verknüpft, um die gewünschte Aussagekraft zu erhalten.

Whitelisting vs. Blacklisting

Eine wichtige Entscheidung für jedes Policysystem ist, ob Whitelisting oder Blacklisting betrieben werden soll. Die beiden Ansätze sind für unterschiedliche Einsatzgebiete unterschiedlich gut geeignet.

Blacklisting ist gut geeignet, wenn nur wenige Ausprägungen einer Eigenschaft verboten werden sollen, beispielsweise weil es sich um eine wenig restriktive Funktionalität handelt, oder das Einsatzgebiet nicht besonders sicherheitskritisch ist. Beim Blacklisting kann es unter Umständen schwierig werden alle ungewollten Werte zu verbieten, da hier abhängig von der Situation gegebenenfalls sehr viele Fälle zu betrachten sind.

Whitelisting auf der anderen Seite ist gut geeignet für Umgebungen die prinzipiell erst mal keine Zugriffe erlauben und nur für wenige, ausgewählte Werte einen Zugriff zulassen. Beim Whitelisting besteht ein geringeres Risiko, dass ein ungewollter Zugriff erlaubt wird, welcher in der Policy vergessen wurde. Dafür steigt die Komplexität von Whitelisting stark mit der Anzahl der erlaubten Werte bzw. Zugriffe an, da hier viele einzelne (Grenz-) Fälle entstehen können.

Um eine maximale Flexibilität bei der Verwendung von D° zu ermöglichen wird eine Kombination von Black- und Whitelisting verwendet, welche Greylisting genannt wird. Dabei ist der Begriff Greylisting im Kontext von D° nicht mit dem Greylisting-Verfahren zu verwechseln, welches zur Spambekämpfung bei E-Mails verwendet wird. Stattdessen wird damit das Verfahren beschrieben, welches D° verwendet, um Whitelisting mit Blacklisting auf eine definierte Art und Weise zu verbinden.

D° erlaubt es Sprachentwicklern sowohl erlaubende, als auch verbietende Policies zu definieren, wodurch zum einen Blacklisting und zum anderen Whitelisting und Mischformen aus beiden Verfahren verwendet werden können. Dies erfordert die Aufmerksamkeit des Sprachentwicklers während der Entwicklung neuer Policies.

Greylisting in D°

Zunächst wird an dieser Stelle die Funktionsweise vom Greylisting in D° beschrieben, bevor zwei Sonderfälle beschrieben werden, welche sich durch das System ergeben uns es erlauben "konventionelles" White- und Blacklisting zu verwenden.

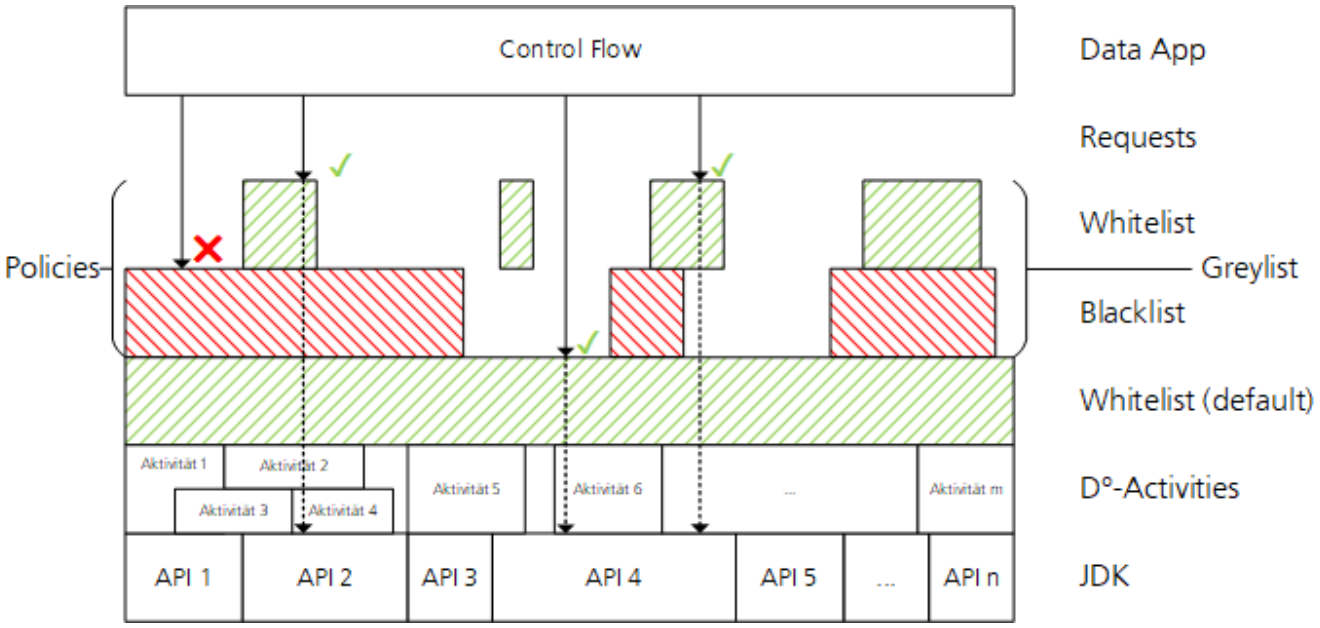
Das Ziel ist es bestimmte Aktionen unter bestimmten Umständen zu erlauben oder zu verbieten. Diese Aktionen werden in Java durch das JDK in Form von APIs bereitgestellt. Bei der Entwicklung von Data Apps ist keine direkte Verwendung von Java-Funktionalitäten möglich und nur die Verwendung von D°-Aktivitäten erlaubt es die gewünschten Funktionen auszuführen. Dabei können einzelne APIs von beliebig vielen Aktivitäten verwendet werden. So sind einzelne API Endpunkte von mehreren und andere von keiner Aktivität abgedeckt. Wird ein Teil einer API von keiner Aktivität verwendet kann er auch nicht in D° verwendet werden.

Dabei können diese Aktivitäten, welche von den Data Apps verwendet werden, mit Policies verknüpft werden, welche während der Ausführung überprüft werden, um zu entscheiden, ob die Ausführung erlaubt ist. Dabei wird bei der Überprüfung nach folgendem Verfahren vorgegangen:

1. Zuerst werden alle Policies, welche Aktionen erlauben überprüft. Diese werden im Folgenden "positive Policies" genannt. Sollte eine dieser positiven Policies auf die aktuelle Ausführung passen und die durchzuführende Aktion erlauben, wird die Überprüfung abgeschlossen und die Ausführung erlaubt.
2. Findet sich keine positive Policy, welche die geplante Ausführung erlaubt, werden alle Policies, welche Verbote enthalten (sogenannte "negative Policies") überprüft. Findet sich dabei eine negative Policy, welche die aktuelle Ausführung verbietet, wird die Ausführung mit einer Fehlermeldung abgebrochen.
3. Wird weder eine positive noch eine negative Policy gefunden, welche auf die Aktuelle Ausführung angewendet werden kann, wird die Ausführung erlaubt.

Aus diesem Verfahren ergibt sich der folgende Schluss, welcher bei der Entwicklung und Verwendung von Policies zwingend beachtet werden muss: Eine positive Policy überdeckt in jedem Fall eine negative Policy, sofern die Schnittmenge der Policies nicht leer ist und führt somit zu einer Erlaubnis für die Ausführung.

Die nachfolgende Abbildung illustriert das zuvor beschriebene Verfahren und zeigt deutlich auf, in welcher Reihenfolge die einzelnen Policyarten ausgewertet werden, um in jeder Situation zu eindeutigen Ergebnissen zu gelangen.



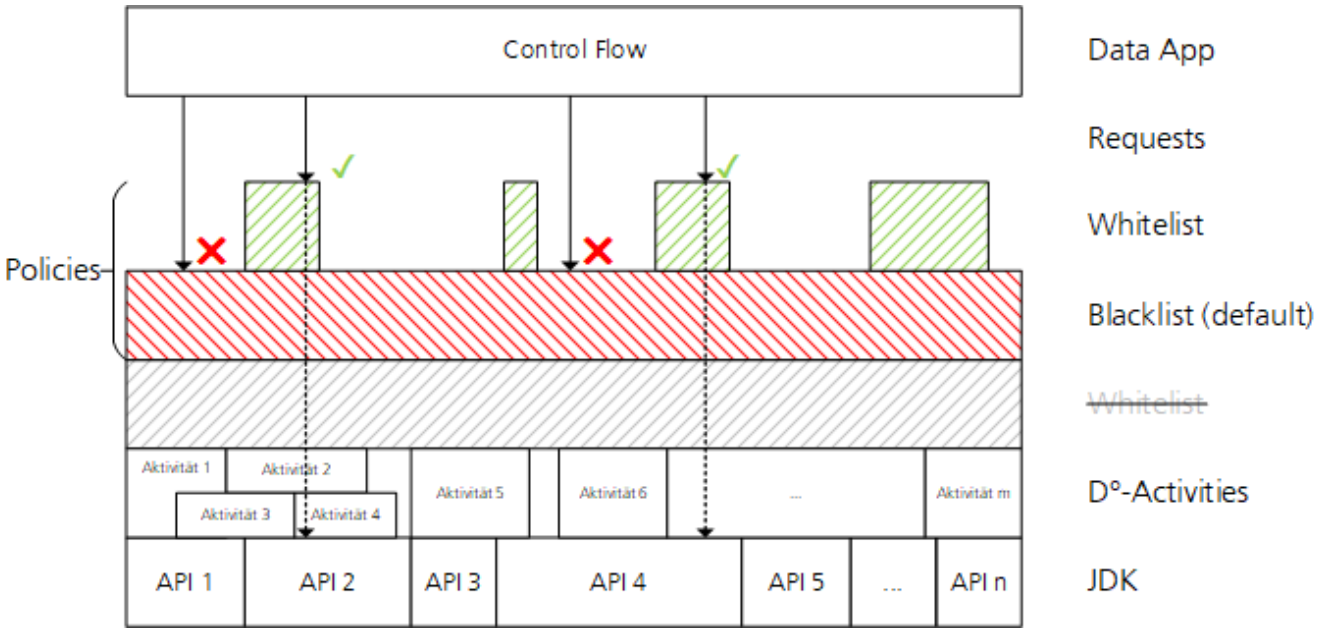
Nachfolgend werden noch zwei Sonderfälle betrachtet, welche sich durch die Ausgestaltung des Systems automatisch ergeben und es erlauben das D°-Policysystem als konventionelles White- bzw. Blacklisting System zu verwenden.

Whitelisting in D°

Sofern sich unter den negativen Policies eine befindet, welche alle möglichen Anfragen inkludiert, bzw. die Summe aller negativen Policies dieses Ziel erreichen, funktioniert das D°-Policysystem wie ein normales Whitelisting-System. Dies wird erreicht, da die negativen Policies ausgewertet werden, bevor auf das Standardverhalten (erlauben) zurückgefallen wird.

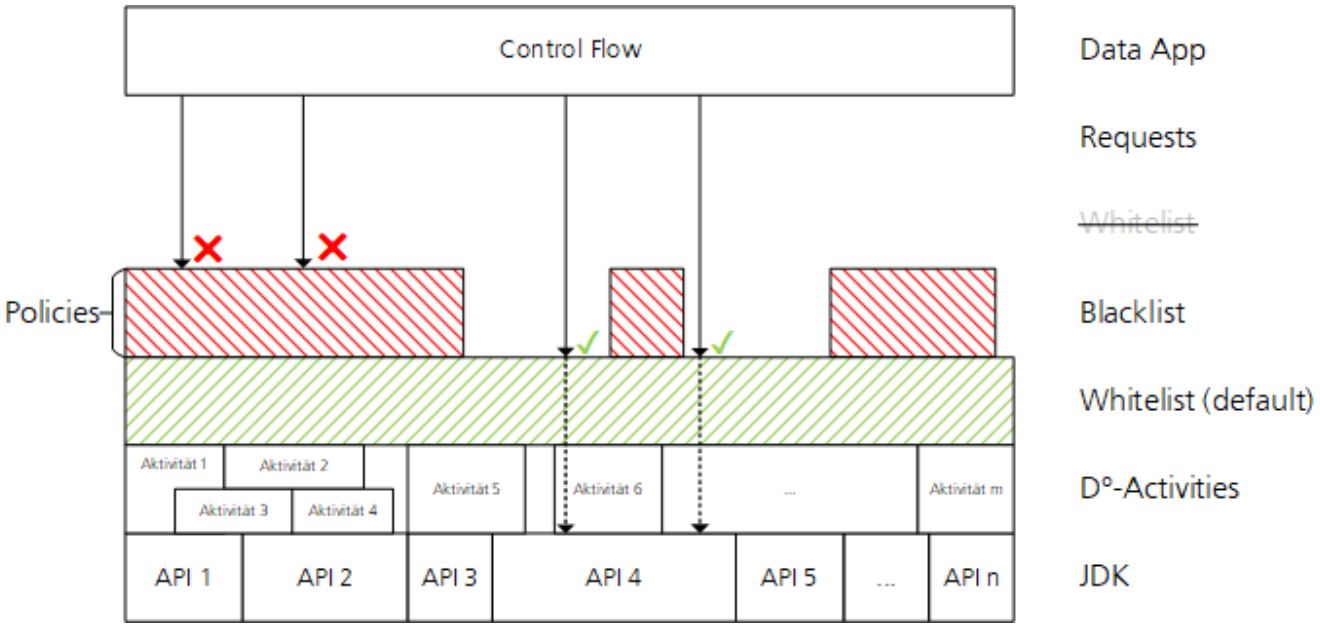
Da, abhängig von der/den API/APIs auf die sich eine Policy bezieht, unterschiedlich komplexe Angaben notwendig sind um eine Blacklist zu erzeugen, welche alle Anfragen abdeckt, bietet D° hierfür eine extra `WILDCARD`-Policy, welche genau dies leistet.

Die nachfolgende Abbildung illustriert dieses Szenario.



Blacklisting in D°

Für den Fall, dass keine positiven Policies verwendet werden, funktioniert das D°-Policysystem als Blacklisting-System.



Somit kann das D°-Policysystem sowohl als Greylisting, als auch als White- bzw. Blacklisting System verwendet werden. Hierdurch wird für Sprachentwickler eine maximale Flexibilität und Ausdrucksstärke erreicht.

Pre- & Postconditions

Die vorherigen Abschnitte beschreiben das Greylisting in D° für verwendete Schnittstellen des JDKs und beschreibt somit das Verfahren der Auswertung für Security Manager Interventions im Detail.

Für Pre- und Postconditions funktioniert die Auswertung anders. Für eine erfolgreiche Evaluation der Pre- und Postcondition-Phasen muss die Boolesche Und-Verknüpfung aller Policies den Wert `true` ergeben. Daraus folgt, dass bereits eine nicht erfüllte Pre- oder Postcondition die gesamte Evaluation scheitern lässt.

Dies liegt darin begründet, dass bei der Security Manager Intervention zunächst Erlaubnisse und Verbote erzeugt werden, welche anschließend ausgewertet werden. Dieses Konzept gibt es bei Pre- und Postconditions nicht. Jede einzelne Pre- und Postcondition wird für sich individuell betrachtet und direkt ausgewertet. Es sind eigenständige Teile einer Policy, welche nicht von anderen Elementen beeinflusst werden können.

Innerhalb der Pre- und Postconditions können beliebige Aussagen überprüft werden, d.h. auch hier ist eine Verwendung von Black- und Whitelisting möglich, dies findet aber in einem kleineren Umfeld statt, da sich die Pre- und Postconditions nicht gegenseitig beeinflussen können.

Somit ist es für jede Pre- und Postcondition möglich, eine eigene Whitelist und Blacklist zu definieren, welche während der Evaluation ausgewertet werden muss. Jede dieser Pre- und Postconditions wird einzeln ausgewertet und muss bei der Evaluation ein positives Ergebnis liefern, ansonsten wird die Ausführung abgebrochen.

Daraus folgt, dass Policies, welche alle mit dem gleichen Konstrukt verknüpft sind, widerspruchsfreie Pre- und Postconditions haben müssen, da die Evaluation sonst in jedem Fall fehl schlägt. Für die Erlaubnisse und Verbote der Security Manager Intervention gilt dies nicht, da das Vorgehen bei solchen Widersprüchlichen Erlaubnissen und Verboten klar definiert ist.

Durchsetzbarkeit

Es gibt einige Aspekte die zu beachten sind, um eine korrekte Funktionsweise des D°-Policysystems zu gewährleisten. Diese Punkte werden im Folgenden detailliert betrachtet.

Erfassung zusätzlicher Daten

Es wurde bereits zuvor erwähnt, dass im Falle von Security Manager Interventions innerhalb der betroffenen Policy zusätzliche Daten zur Verfügung gestellt werden, welche aus dem unterbrochenem Aufruf resultieren und zur Evaluierung verwendet werden können. Dies wurde am Beispiel einer Constraint aufgezeigt, welche eine Quota zum schreiben von Dateien gewährt.

Diese Daten existieren erstmal nicht und müssen zunächst gesammelt und in einer Form bereit gestellt werden, dass sie von den Policies erreicht und verwendet werden können.

Zu diesem Zweck wurden Schnittstellen entwickelt, welche auf die geschützten APIs des JDK abbilden. Diese Schnittstellen stehen Sprachentwicklern zur Verfügung, wenn sie D°-Aktivitäten entwickeln wollen. Dabei werden von dieser Schnittstelle die notwendigen Daten gesammelt, bereitgestellt und anschließend der Aufruf an die entsprechende geschützte API des JDK delegiert.

An dieser Stelle ist es notwendig, dass jeder Aufruf der geschützten APIs, welcher aus einer D°-Aktivität erfolgt, über die geschaffenen Schnittstellen erfolgen, da ansonsten die notwendigen Daten nicht gesammelt werden können.

Daher ist einer der ersten Schritt des Ablaufs einer Security Manager Intervention immer, das der D°-SecurityManager überprüft, ob der Aufruf aus einer der dafür geschaffenen Schnittstellen stammt. Ist dies nicht der Fall wird die Ausführung mit einer Fehlermeldung verweigert. Hierdurch wird sichergestellt, dass jeder Aufruf der geschützten APIs des JDK über die D°-Schnittstellen erfolgt. Somit ist sichergestellt, dass alle notwendigen Daten gesammelt werden können, sobald die Aufrufe stattfinden.

Diese Überprüfung findet dabei auf Basis des Java-StackTrace statt, welche die komplette Aufrufhierarchie des aktuellen Threads enthält. Da ein Request an eine Data App stets sequentiell in einem Thread ausgeführt wird ist sichergestellt, dass die aktuelle StackTrace bis zum Aufruf der Data App zurückverfolgt werden kann.

Wirksamkeitsbereich des D°-SecurityManagers

Wie im vorherigen Abschnitt beschrieben, ist es durch die speziellen Schnittstellen und Analysen des Stacks möglich zu bestimmen, ob ein Aufruf an eine geschützte API des JDK die dafür vorgesehenen Schnittstellen erfolgt. Dabei ist es aber nicht in sämtlichen Situationen erwünscht, dass Anfragen, die nicht diesen Weg gehen, abgelehnt wird.

Dies ist beispielsweise der Fall, wenn eine Data App grade hochgefahren wird. Beispielsweise basiert eine REST-Data App auf der Verwendung von Spring Boot. Der Startprozess einer solchen Data App resultiert daher in einer großen Anzahl von Aufrufen an die geschützten APIs des JDK. Dabei erfolgen diese Aufrufe nicht über die speziellen Schnittstellen von D°. Normalerweise würden diese Anfragen aus diesem Grund abgelehnt werden und es wäre unmöglich die Data App jemals zu starten. Ähnliches gilt für den Java-Code zwischen den einzelnen Anweisungen der Data App.

Dieser Gluecode/Boilerplate-Code ist notwendig um die korrekte Funktionsweise der Data App zu gewährleisten, soll aber nicht durch den D°-SecurityManager überprüft werden.

Das Ziel ist es die Security Manager Interventions ausschließlich für Aufrufe aus Aktivitäten auszuführen.

Um diese Problematik zu lösen werden die folgenden Regeln angewandt:

- Jede Data App hat eine zentrale Einstiegsmethode mit dem Namen `process`, welche den Startpunkt für die Ausführung des Workflows darstellt
- Jede Aktivität hat eine Ausführungsmethode mit den Namen `run`
- Befindet sich auf der aktuellen StackTrace ein Aufruf der `process`-Methode einer Klasse, die Zuweisungskompatibel zur Klasse `DataApp` ist, befinden wir uns in der Ausführung des Data App Workflows
    - Dies reicht alleine nicht aus, da grade ein Teil des Gluecodes ausgeführt werden könnte
    - Es kann so sichergestellt werden, dass der Startvorgang, sowie das ordnungsgemäße Beenden der Data App nicht durch den D°-SecurityManager beeinflusst bzw. gestört wird.
- Befindet sich auf der aktuellen StackTrace ein Aufruf der `run`-Methode einer Klasse, die Zuweisungskompatibel zur Klasse `ActivityApi` ist, befindet sich die Ausführung in der Logik einer Aktivität
    - Theoretisch ist es möglich eine Aktivität auch außerhalb des Workflows aufzurufen, weswegen diese Bedingung notwendig für die Verwendung des D°-SecurityManagers ist, aber nicht hinreichend
- Wenn sichergestellt ist, dass grade die Logik einer Aktivität, welche in einem Data App Workflow enthalten ist, stattfindet, wird der D°-SecurityManager für diesen Aufruf verwendet
- Ansonsten ignoriert der D°-SecurityManager den Aufruf

Filtern irrelevanter Aufrufe

Selbst innerhalb der Logik einer Aktivität können Aufrufe an die geschützten APIs des JDK erfolgen, welche nicht vom Sprachentwickler intendiert sind und auch nicht durch das D°-Policysystem verarbeitet werden sollen. Dies liegt an der Funktionsweise von Java bzw. der Ausführungsumgebung JRE. Beispielsweise werden während der Laufzeit Klassen nachgeladen, sobald diese das erste mal verwendet werden. Dies resultiert in lesenden Dateizugriffen durch einen Classloader.

Normalerweise würde in diesen Fällen der D°-SecurityManager die Ausführung verweigern, da nicht die D°-Schnittstelle für Dateioperationen verwendet wird.

Aus diesem Grund kann der D°-SecurityManager fein granular konfiguriert werden, um bestimmte Policies bei Aufrufen von spezifischen Methoden einzelner Klassen zu ignorieren. Diese Konfiguration ist in eine properties-Datei ausgelagert und enthält Einträge der folgenden Form:

```
degree.securitymanager.ignore[N] = POLICY,CLASS,METHOD
```

Dieser Mechanismus muss mit größter Vorsicht verwendet werden, da ansonsten die Gefahr besteht, dass ungewollt Aufrufe vom D°-SecurityManager ignoriert werden und somit das D°-Policysystem ausgehebelt werden könnte. Nachfolgend sind einige Beispiele für mögliche Einträge in diese Datei zu sehen.

ignored_elements.properties

```
degree.securitymanager.ignore[0] = READ_FILE,java.lang.ClassLoader,loadClass
degree.securitymanager.ignore[1] = READ_FILE,java.security.SecureClassLoader,loadClass
degree.securitymanager.ignore[2] = READ_FILE,java.net.URLClassLoader,loadClass
degree.securitymanager.ignore[3] = READ_FILE,org.springframework.boot.web.embedded.tomcat.
TomcatEmbeddedWebappClassLoader,loadClass
degree.securitymanager.ignore[4] = READ_FILE,de.fhg.isst.oe270.degree.runtime.java.security.functionality.
modules.DegreeFileOperations,getFileSize
```

Verfügbare D°-Schnittstellen

Nachfolgend werden die Schnittstellen, die bereits für die Verwendung der geschützten APIs des JDK vorhanden sind beschrieben. Diese werden Schrittweise und dem aktuellen Bedarf entsprechend (weiter-) entwickelt.

Schnittstelle	Methode	Eingabeparameter	Ausgabeparameter	Exceptions	Notiz
`DegreeFileOperations`	`readFileToString`	`String filepath`	`String`	IOException	Liest den Inhalt einer Datei als String ein und gibt ihn als String zurück. Als Encoding wird UTF-8 verwendet. Im Fehlerfall wird eine IOException geworfen.
	`readFileToString`	`String filepath` `String encoding`	`String`	IOException	Liest den Inhalt einer Datei als String ein und gibt ihn als String zurück. Dabei wird ein gegebenes Encoding verwendet. Im Fehlerfall wird eine IOException geworfen.
	`readFileToString`	`String filepath` `Charset encoding`	`String`	IOException	Liest den Inhalt einer Datei als String ein und gibt ihn als String zurück. Dabei wird ein gegebenes Encoding verwendet. Im Fehlerfall wird eine IOException geworfen.
	`writeStringToFile`	`String filepath` `String content`	void	IOException	Schreibt den Inhalt eines Strings in eine Datei. Die Datei wird dabei erzeugt, falls sie noch nicht existiert. Existiert die Datei bereits, wird an die Datei angehängt. Als Encoding wird UTF-8 verwendet. Im Fehlerfall wird eine IOException geworfen.

	`writeStringToFile`	`String filepath`
		`String content`
		`boolean append`
			void	IOException	Schreibt den Inhalt eines Strings in eine Datei.
					Die Datei wird dabei erzeugt, falls sie noch nicht existiert.
					Existiert die Datei bereits, entscheidet der Parameter `append`, ob an die Datei angehängt wird.
					Als Encoding wird UTF-8 verwendet.
					Im Fehlerfall wird eine IOException geworfen.

	`writeStringToFile`	`String filepath`
		`String content`
		`String encoding`
			void	IOException	Schreibt den Inhalt eines Strings in eine Datei.
					Die Datei wird dabei erzeugt, falls sie noch nicht existiert.
					Existiert die Datei bereits, wird an die Datei angehängt.
					Dabei wird ein gegebenes Encoding verwendet.
					Im Fehlerfall wird eine IOException geworfen.

	`writeStringToFile`	`String filepath`
		`String content`
		`String encoding`
			void	IOException	Schreibt den Inhalt eines Strings in eine Datei.
					Die Datei wird dabei erzeugt, falls sie noch nicht existiert.
					Existiert die Datei bereits, entscheidet der Parameter `append`, ob an die Datei angehängt wird.
					Dabei wird ein gegebenes Encoding verwendet.

		`boolean append`			Im Fehlerfall wird eine IOException geworfen.

	`writeStringToFile`	`String filepath`
		`String content`
		`Charset encoding`
			void	IOException	Schreibt den Inhalt eines Strings in eine Datei.
					Die Datei wird dabei erzeugt, falls sie noch nicht existiert.
					Existiert die Datei bereits, wird an die Datei angehängt.
					Dabei wird ein gegebenes Encoding verwendet.
					Im Fehlerfall wird eine IOException geworfen.

	`writeStringToFile`	`String filepath`
		`String content`
		`Charset encoding`
		`boolean append`
			void	IOException	Schreibt den Inhalt eines Strings in eine Datei.
					Die Datei wird dabei erzeugt, falls sie noch nicht existiert.
					Existiert die Datei bereits, entscheidet der Parameter `append`, ob an die Datei angehängt wird.
					Dabei wird ein gegebenes Encoding verwendet.
					Im Fehlerfall wird eine IOException geworfen.

Verfügbare Constraints & Policies

Abschließend werden die verfügbaren Constraints und Policies aufgelistet und ermöglichen so eine einfachere Verwendung. Neue Constraints & Policies werden bei Bedarf entwickelt und finden dann Einzug in diese Liste.

Constraints

Kategorie	Name	Eingabeparameter	Precondition	Security Manager Intervention	Postcondition	Notiz
`core.authorization`	`core.DenyUsername`	`core.Username username`	Blacklist	-	-	Verbietet dem Nutzer mit gegebenem Namen die Ausführung. Siehe Abschnitt "Subsystem für Kontextinformationen" in Compiler V2.
	`core.DenyUserrole`	`core.Userrole role`	Blacklist	-	-	Verbietet dem Nutzer mit gegebener Rolle die Ausführung. Siehe Abschnitt "Subsystem für Kontextinformationen" in Compiler V2.
	`core.RequireUsername`	`core.Username username`	Whitelist	-	-	Erlaubt nur dem Nutzer mit gegebenem Namen die Ausführung. Siehe Abschnitt "Subsystem für Kontextinformationen" in Compiler V2.
	`core.RequireUserrole`	`core.Userrole role`	Whitelist	-	-	Erlaubt nur dem Nutzer mit gegebener Rolle die Ausführung. Siehe Abschnitt "Subsystem für Kontextinformationen" in Compiler V2.
`core.authorization.jwt`	`core.DenyUsernameJwt`	`core.Username username`	Blacklist	-	-	Verbietet dem Nutzer, der im JWT des Requests defineirt ist, mit gegebenem Namen die Ausführung. Funktioniert nur bei REST-Data Apps.
	`core.DenyUserroleJwt`	`core.Userrole role`	Blacklist	-	-	Verbietet dem Nutzer, der im JWT des Requests defineirt ist, mit gegebener Rolle die Ausführung. Funktioniert nur bei REST-Data Apps.
	`core.RequireUsernameJwt`	`core.Username username`	Whitelist	-	-	Erlaubt nur dem Nutzer, der im JWT des Requests defineirt ist, mit gegebenem Namen die Ausführung. Funktioniert nur bei REST-Data Apps.
	`core.RequireUserroleJwt`	`core.Userrole role`	Whitelist	-	-	Erlaubt nur dem Nutzer, der im JWT des Requests defineirt ist, mit gegebener Rolle die Ausführung. Funktioniert nur bei REST-Data Apps.

core.authorization.os	core.DenyUsernameOs	core.Username username	Blacklist	-	-	Verbietet dem Nutzer des Betriebssystems mit gegebenem Namen die Ausführung.
	core.DenyUserroleOs	core.Userrole role	Blacklist	-	-	Verbietet dem Nutzer des Betriebssystems mit gegebener Rolle die Ausführung.
	core.RequireUsernameOs	core.Username username	Whitelist	-	-	Erlaubt nur dem Nutzer des Betriebssystems mit gegebenem Namen die Ausführung.
	core.RequireUserroleOs	core.Userrole role	Whitelist	-	-	Erlaubt nur dem Nutzer des Betriebssystems mit gegebener Rolle die Ausführung.
core.date	core.UseNotAfterTimestamp	core.DateTime timestamp	Blacklist	-	-	Verbietet Aufrufe, die nach einem bestimmten Zeitpunkt erfolgen.
	core.UseNotBeforeTimestamp	core.DateTime timestamp	Blacklist	-	-	Verbietet Aufrufe, die vor einem bestimmten Zeitpunkt erfolgen.
core.io.file	core.AllowDeleteFile	core.Path path core.PathMatchingStrategy matchingStrategy	-	Whitelist	-	Erlaubt es Dateien an gegebenem Pfad zu löschen. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.AllowReadFile	core.Path path core.PathMatchingStrategy matchingStrategy	-	Whitelist	-	Erlaubt es Dateien an gegebenem Pfad zu lesen. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.AllowWriteFile	core.Path path core.PathMatchingStrategy matchingStrategy	-	Whitelist	-	Erlaubt es Dateien an gegebenem Pfad zu schreiben. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.ForbidDeleteFile	core.Path path core.PathMatchingStrategy matchingStrategy	-	Blacklist	-	Verbietet es Dateien an gegebenem Pfad zu löschen. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.ForbidReadFile	core.Path path core.PathMatchingStrategy matchingStrategy	-	Blacklist	-	Verbietet es Dateien an gegebenem Pfad zu lesen. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.ForbidWriteFile	core.Path path core.PathMatchingStrategy matchingStrategy	-	Blacklist	-	Verbietet es Dateien an gegebenem Pfad zu schreiben. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.QuotaReadFile	core.UnsignedInteger quota core.ByteUnit unit core.Path path core.PathMatchingStrategy matchingStrategy	Whitelist	Whitelist	Internes Statusupdate	Erlaubt es Dateien an gegebenem Pfad zu lesen, bis die gewährte Quota aufgebraucht ist. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
	core.QuotaWriteFile	core.UnsignedInteger quota core.ByteUnit unit core.Path path core.PathMatchingStrategy matchingStrategy	Whitelist	Whitelist	Internes Statusupdate	Erlaubt es Dateien an gegebenem Pfad zu schreiben, bis die gewährte Quota aufgebraucht ist. Wie der Pfad mit der Anfrage abgeglichen wird, hängt von der `PathMatchingStrategy` ab.
core.tags	core.DenyTag	core.Tag tag	Blacklist	-	-	Verbietet die Ausführung, wenn bestimmte Tags in der Data App vorhanden sind.
	core.RequireTag	core.Tag tag	Whitelist	-	-	Erlaubt die Ausführung nur dann, wenn bestimmte Tags in der Data App vorhanden sind.

core.text	core.MaxLength	core.UnsignedInteger maxLength core.Text content	Whitelist	-	-	Erlaubt die Ausführung nur, wenn der gegebene Inhalt eine gegebene Länge nicht überschreitet.
core.wildcard	core.GrantAll	-	-	Whitelist	-	Erlaubt alle Anfragen und deaktiviert die Security Manager Intervention für den aktuellen Aufruf.
	core.UseWhitelist	-	-	Blacklist	-	Verbietet alle Aufrufe und sorgt dafür, dass der aktuelle Aufruf der Security Manager Intervention als reines Whitelisting ausgewertet wird.
ids.geolocation	IDS.RequireGeolocation	core.Text infomodel core.Text location	Whitelist	-	-	Erlaubt die Ausführung nur, wenn bei einer gegebenen Ausprägung des IDS Infomodells eine bestimmte Position enthalten ist.
	IDS.DenyGeolocation	core.Text infomodel core.Text location	Blacklist	-	-	Erlaubt die Ausführung nicht, wenn bei einer gegebenen Ausprägung des IDS Infomodells eine bestimmte Position enthalten ist.

Policies

Name	Enthaltene Policies & Constraints	Beschreibung
AllowedTimeInterval	Constraint UseNotBefore Constraint UseNotAfter	Erlaubt die Nutzung nur in einem gegebenen Zeitraum.
AllowReadOnlyFile	Constraint AllowReadFile Constraint ForbidWriteFile	Gibt lesenden Zugriff auf Dateien, aber verbietet das Schreiben.