Generierung unterschiedlicher Arten von Data Apps

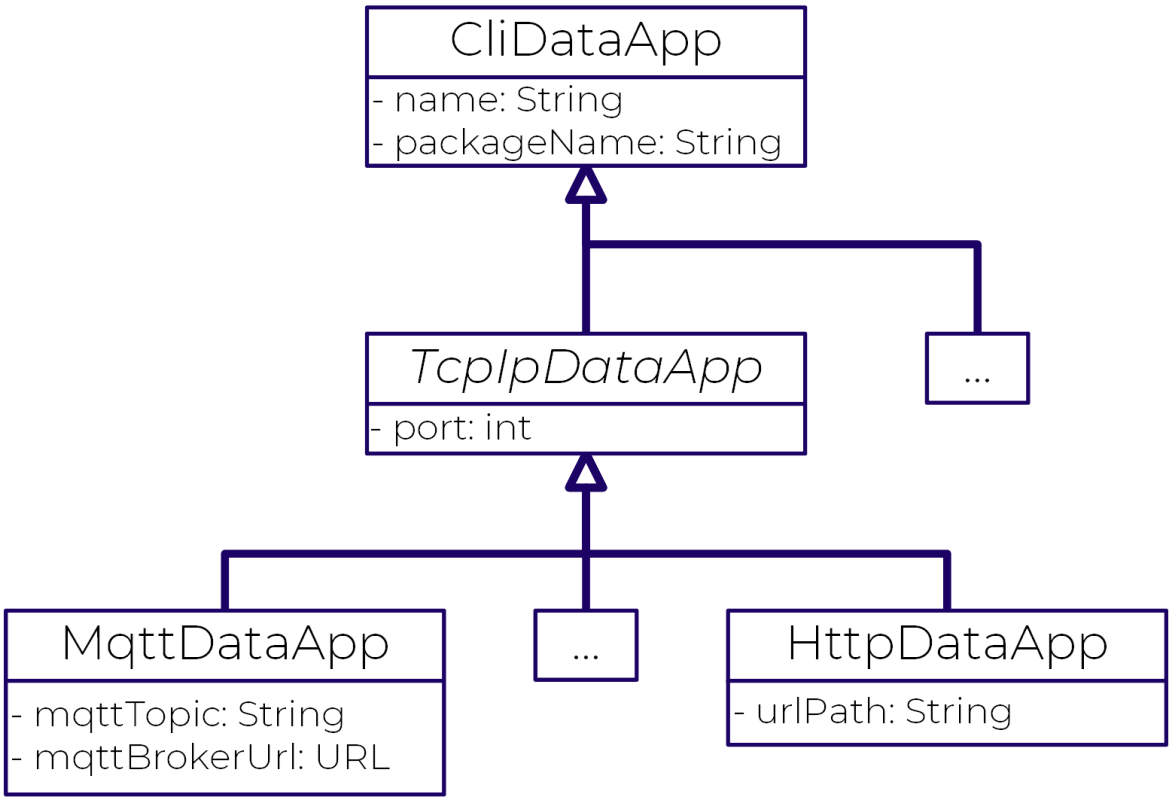
Versionen

Datum	Notiz	Version
30.06.2020	Erstveröffentlichung	1.0

Autoren

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

Für D° wurde ein Verfahren zur Kategorisierung von unterschiedlichen Data App Typen definiert. Dabei werden die einzelnen Typen von Data Apps durch einen möglichst kleinen Satz an Einstellungen definiert, welche vorhanden sein müssen, um den entsprechenden Data App Typen zu beschreiben. Die nachfolgende Abbildung zeigt ein Beispiel für eine mögliche Struktur von Data App Typen als Klassendiagramm. Die Darstellung als Klassendiagramm ist ein direktes Ergebnis des definierten Prozesses, welcher eine möglichst einfache Überführung der erstellten Data App Typen in nutzbaren Programmcode erlaubt.



Nachfolgend wird das Verfahren zur Generierung unterschiedlicher Data App Typen beschrieben und unterstützte Data App Typen aufgelistet.

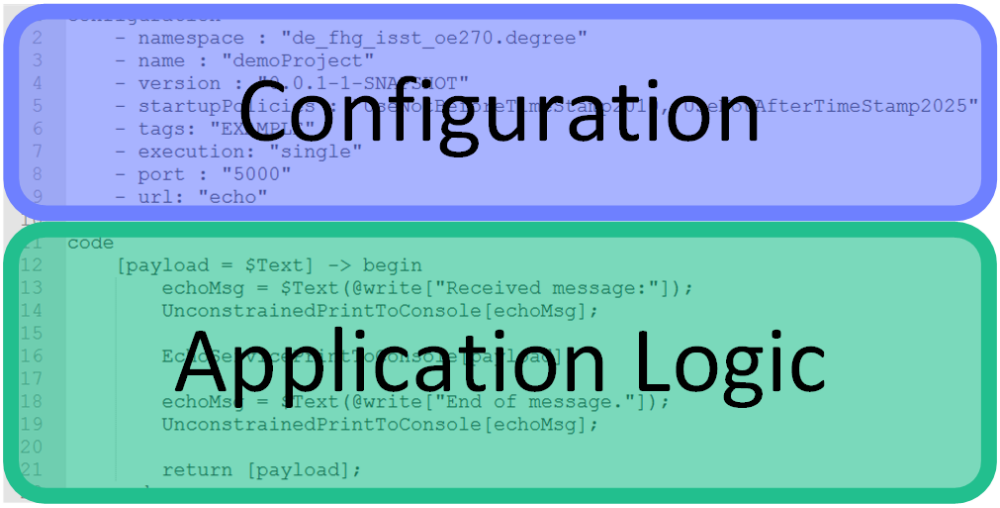
Verfahren

Um ein umfassendes Verständnis für die Generierung von unterschiedlichen Data App Typen zu erhalten ist eine vorherige Betrachtung des Aufbaus von Data Apps hilfreich. Zu diesem Zweck zeigt die nachfolgende Abbildung den Programmcode für eine einfache Applikation, welche in D° implementiert wurde. Diese bietet eine echo-Funktionalität, welche eine erhaltene Nachricht zuerst auf die Konsole ausgibt und anschließend unverändert an den Aufrufer zurück sendet.

```
configuration
    - namespace : "de_fhg_isst_oe270.degree"
    - name : "demoProject"
    - version : "0.0.1-1-SNAPSHOT"
    - startupPolicies : "UseNotBeforeTimeStamp2010, UseNotAfterTimeStamp2025"
    - tags: "EXAMPLE"
    - execution: "single"
    - port : "5000"
    - url: "echo"

code
    [payload = $Text] -> begin
        echoMsg = $Text(@write["Received message:"]);
        UnconstrainedPrintToConsole[echoMsg];

        EchoServicePrintToConsole[payload];

        echoMsg = $Text(@write["End of message."]);
        UnconstrainedPrintToConsole[echoMsg];

        return [payload];
    end
```

Betrachtet man diesen Programmcode wird eine Eigenschaft erkennbar, welche alle Applikationen, die mit D° implementiert werden gemeinsam haben: Der Programmcode besteht aus den zwei Teilen Konfiguration und Applikationslogik. Dieser Aufbau ist identisch für jede Applikation, die mit D° entwickelt wird. Der Vorteil bei dieser Strukturierung ist, dass eine Applikation (abgesehen von eventuellen Spracherweiterungen, welche Datentypen, Aktivitäten, etc. enthält) in sich geschlossen ist und sämtliche Informationen enthält, um übersetzt werden zu können. Gleichzeitig wird die Konfiguration der Applikation von der eigentlichen Applikationslogik entkoppelt, was die Übersichtlichkeit steigert.

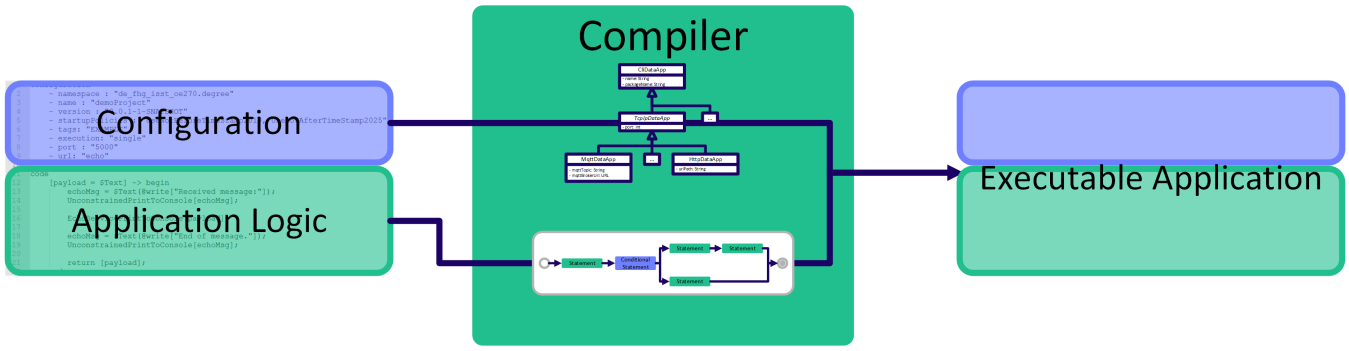
Die Nachfolgende Abbildung enthält die zuvor gezeigt Applikation und kennzeichnet die beiden Bereiche der Applikation farblich.



Ein weiterer großer Vorteil ergibt sich bei der Generierung unterschiedlicher Data App Typen: Um den generierten Typen einer D° Applikation zu ändern (beispielsweise von einer Kommandozeilen-Schnittstelle hin zu einer HTTP-API), ist es nicht notwendig die Applikationslogik zu modifizieren. Es reicht aus die Konfiguration der Applikation entsprechend anzupassen. Diese Trennung zwischen Konfiguration und Logik einer Applikation ermöglicht eine einfache Modifikation von Applikationen und verbessert die Übersichtlichkeit von D°-Applikationen erheblich.

Diese Trennung setzt sich in der Funktionsweise des D°-Compilers fort. Der D°-Compiler verarbeitet die Applikationslogik und die Konfiguration getrennt voneinander und kombiniert die beiden Ergebnisse abschließend in einer einzelnen ausführbaren Applikation. Der Codegenerator erzeugt den Code für die Programmlogik beinahe vollständig unabhängig vom generierten Data App Typen. Die einzige Ausnahme ist dabei die Art und Weise in der Rückgabewerte und Fehlermeldungen festgehalten werden. Dies ist Teil der Applikationslogik aber innerhalb des Compilers entsprechend gekapselt, das keine zusätzlichen Aufwände bei der Umsetzung zusätzlicher Data App Typen entstehen.

Die nachfolgende Abbildung stellt diesen Ablauf schematisch dar.



Somit ist es möglich den Typen einer Applikation allein durch Anpassung der Konfiguration anzupassen. Zu diesem Zweck führt der Compiler zum Startzeitpunkt eine Art Pattern Matching durch, welche den generierten Applikationstypen bestimmt. Diese Information wird dann entsprechend verwendet, um den korrekten Boilerplate-Code (bspw. Superklassen und Annotationen) zu erzeugen und die richtigen Konstrukte zu verwenden, beispielsweise zur Rückgabe von Fehlermeldungen.

Unterstützte Data App Typen

Nachfolgend werden die unterschiedlichen Data App Typen, welche vom D°-Compiler aktuell erzeugt werden können, näher betrachtet.

Data App Schnittstellen

Eine Dimension, welche zuvor im Verfahren zur Kategorisierung unterschiedlicher Arten von Data Apps definiert wurde, ist die Schnittstelle, welche die Data App nach außen exponiert.

CLI Data App

Die CLI Data App ist die einfachste Art von Data App, welche vom D°-Compiler erzeugt werden kann. Es ist der Standardtyp, welcher generiert wird, sofern kein anderer Typ durch die gegebene Applikationskonfiguration generiert werden muss. Die Konfigurationselemente, welche für die CLI Data App notwendig sind, sind nicht für die eigentliche Schnittstelle notwendig, sondern werden zur Beschreibung von allen Data Apps verwendet.

	Name	Beschreibung
1	`name`	Eindeutiger Name für die gegebene Applikation.
2	`namespace`	Namensraum für die Applikation. Vergleichbar zum Java-Namespace.

Die CLI Data App wird über die Kommandozeile aufgerufen und bekommt dabei die notwendigen Eingabeparameter übergeben. Darüber hinaus bietet sie keine Schnittstelle zur Nutzerinteraktion, welche während der Ausführung zur Verfügung steht. Der vorzeitige Abbruch der Ausführung kann nur über externe Signale erfolgen.

HTTP Data App

Die HTTP Data App ist eine Unterklasse der (nicht instanziierbaren) TCP/IP Data App. Sie stellt eine Schnittstelle über HTTP bereit, welche es erlaubt die Ausführung mit einem übermitteltem Satz Eingabeparametern zu starten und als Antwort einen eindeutigen Bezeichner zu erhalten. Dieser Bezeichner wird verwendet um beim zweiten Verfügbaren HTTP-Endpunkt die Ausführungsergebnisse abzufragen.

	Name	Geerbt von	Beschreibung
1	`name`	CLI Data App	Eindeutiger Name für die gegebene Applikation.
2	`namespace`	CLI Data App	Namensraum für die Applikation. Vergleichbar zum Java-Namespace.
3	`port`	TCP/IP Data App	Port unter welchem die HTTP Schnittstelle verfügbar gemacht wird.
4	`url`	-	Pfad unter dem der Endpunkt zum starten der Ausführung erreichbar gemacht wird. Das Abrufen der Ergebnisse ist unter dem Pfad '${`url`}Result' möglich.

Die HTTP Data App wird über die Kommandozeile gestartet und stellt dann bis zum Stoppen der Applikation eine HTTP-Schnittstelle bereit, welche verwendet werden kann, um die Ausführung mit gegebenen Eingabeparametern zu starten und anschließend die dazugehörigen Ausführungsergebnisse abzuholen.

Ausführungsverhalten

Die zweite Dimension, welche bei der Definition von Data App Typen relevant ist, ist das Ausführungsverhalten einer Applikation, welche aktuell ausgeführt wird.

Einzelausführung

Das Standardverhalten für jede Data App ist die Einzelausführung. Bei diesem Verhalten wird die Applikationslogik beim entsprechenden Signal (bspw dem Aufruf der HTTP-Schnittstelle bei HTTP Data Apps) einmalig ausgeführt und das Ausführungsergebnis kann abgeholt werden bzw. wird zurückgeliefert (z.B. bei CLI Data Apps).

	Name	Beschreibung
1	`execution`	Wert "single". (Optional)

Periodische Ausführung

Ein weiteres mögliches Ausführungsverhalten ist die periodische Ausführung. Dabei bietet die Applikation wie gewohnt die Möglichkeit die Ausführung zu starten. Anschließend findet aber nicht eine einmalige Ausführung der Applikationslogik statt. Stattdessen wird die Ausführung in einer Endlosschleife und mit gegebener Pause zwischen den Ausführungen ausgeführt.

	Name	Beschreibung
1	`execution`	Wert "periodic".
2	`periodicTime`	Zeit in Millisekunden die vor dem Beginn der nächsten Ausführung gewartet wird, nachdem die aktuelle Ausführung beendet wurde. Defaultwert: 0; Optional