Die Leichtigkeit des Seins

Bindings für Eclipse SmartHome entwickeln

Moritz Kammerer

Eclipse SmartHome ist eine Open-Source-Plattform, die sich durch Bindings genannte Plug-ins um weitere Geräte erweitern lässt. Der Artikel zeigt am Beispiel des Nest-Thermostats, wie einfach es ist, Geräte in Eclipse SmartHome zu integrieren. Als Laufzeitumgebung dient openHAB 2. Am Ende des Artikels kann das entwickelte Binding verwendet werden, um zum Beispiel eine Philips-Hue-Lampe mittels des Thermostats zu steuern.

Was ist Eclipse SmartHome?

Eclipse SmartHome ist ein unter dem Dach der Eclipse Foundation entwickeltes Framework für das intelligente Zuhause, das *Smarthome*. Das Framework ermöglicht es, herstellerübergreifend Geräte miteinander zu vernetzen, Geräte einheitlich zu steuern und Automatisierungsregeln zu definieren.

Dieser Artikel zeigt am Beispiel des Nest-Thermostats, wie man Geräte über ein sogenanntes *Binding* in Eclipse SmartHome integriert. Das Nest-Thermostat ist eine von der Firma Nest entwickelte Heizungssteuerung. Sie kann die aktuelle Raumtemperatur auslesen und die Zieltemperatur in einem Raum regeln. Das entwickelte Binding kann am Ende dieses Artikels in openHAB 2, einer Laufzeitumgebung auf Basis von Eclipse SmartHome, ausgeführt werden. Den kompletten Quelltext des Bindings finden Sie unter [GitH].

Anatomie eines Eclipse-SmartHome-Bindings

Ein Binding bindet Geräte in Eclipse SmartHome [ECL1] ein. Ein Gerät wird dabei als *Thing* bezeichnet. Ein Binding kann mehrere unterschiedliche Things anbinden. Ein Thing enthält mehrere *Channels*. Ein solcher Channel repräsentiert eine bestimmte Funktionalität des Geräts. Im Fall des Nest-Thermostats gibt es zwei Channels: einen Channel für die eingestellte Zieltemperatur und einen für die aktuelle Umgebungstemperatur. Jeder Channel ist von einem bestimmten Typ, im Falle der Zieltemperatur eine Zahl. Es gibt aber auch andere Typen, wie Schalter, Dimmer und Farben [ECL2].

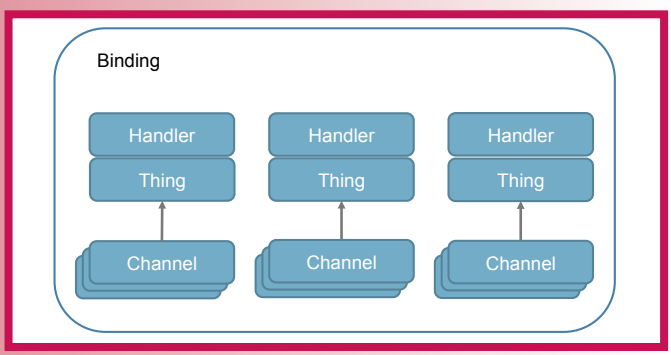

Abb. 1: Anatomie eines Bindings

Ein Channel kann einer *Kategorie* zugeordnet werden, damit die Smarthome-Benutzungsoberfläche spezifische Icons dafür anzeigen kann. Channels können les- und schreibbar oder nur lesbar sein. Im Fall des Nest-Thermostats lässt sich zum Beispiel der Channel für die Zieltemperatur lesen und schreiben, der Channel für die Umgebungstemperatur hingegen nur lesen.

Um ein Thing zu verwalten, wird ein *ThingHandler* implementiert. Dieser kümmert sich um das Aktualisieren der Channels und reagiert auf Kommandos. Abbildung 1 zeigt die einzelnen Bestandteile eines Bindings in einer Übersicht.

Erste Schritte auf dem Weg zum Binding

Eclipse SmartHome ist in Java entwickelt und wird in einer OSGi-Laufzeitumgebung ausgeführt. Jedes Binding ist dabei ein eigenständiges OSGi-Bundle. Am komfortabelsten lässt sich ein Binding mit der Eclipse Java IDE entwickeln. Unter [ECL3] findet sich eine Anleitung, um Eclipse für die Entwicklung eines Bindings einzurichten.

Nach dem Setup-Prozess befindet sich unter `$INSTALLDIR/git/smarthome` das aktuelle Eclipse-SmartHome-Framework. Zu aller erst führen wir darin `mvn clean install` aus. Dies setzt ein installiertes Maven voraus. Dieser Schritt ist wichtig, da er unter anderem die Benutzungsoberfläche des Eclipse SmartHome baut, die wir später noch verwenden werden. Der Setup-Prozess legt auch einen Eclipse Workspace unter `$INSTALL_DIR/ws` an.

Unter `extensions/binding` befindet sich ein Skript namens `create_binding_skeleton` (Endung `.bat` bzw. `.sh`), welches Code für ein Binding generiert. Dieses Skript muss mit dem Namen des Bindings und dem Namen des Autors aufgerufen werden (z. B. `create_binding_skeleton.cmd Nest "Moritz Kammerer"`).

Das Skript erzeugt nun zwei neue Ordner mit dem Quelltext- beziehungsweise dem Testfall-Gerüst unseres Bindings. Im Quelltext-Ordner werden dabei die folgenden Dateien und Unterordner generiert:

- ESH-INF/:
  - `binding/binding.xml` – Metadaten des Bindings: Name, Autor, Beschreibung
  - `thing/thing-types.xml` – Definition von Things und Channels
  - `i18n` – Internationalisierung der Channelbeschreibungen usw.

- src/…/:
  - `NestBindingConstants.java` – Konstanten, die den Namen des Bindings, der Channel usw. enthalten. Stellt Verbindung zwischen dem Java-Code und `thing-types.xml` her
  - `handler/NestHandler.java` – Der `ThingHandler` des Bindings
  - `internal/NestHandlerFactory.java` – Factory für den `ThingHandler`
- META-INF/MANIFEST.MF – OSGi-Manifest
- OSGI-INF/NestHandlerFactory.xml – Registrierung der NestHandlerFactory als OSGi-Komponente
- target/ – Artefakte des Builds
- build.properties – Einstellungen für den Build (vom Maven-Tycho-Plug-in verwendet)
- pom.xml – Maven Project Model

Das neu angelegte Binding können wir nun in Eclipse importieren.

Kommunikation mit dem Nest-Thermostat

Um von Eclipse SmartHome auf das Nest-Thermostat zuzugreifen, muss der Nest-Webservice in der Cloud genutzt werden. Das Thermostat ist über das lokale Netzwerk nicht direkt zugreifbar. Der Webservice ist mit OAuth 2 vor unberechtigten Zugriffen geschützt. Für den Zugriff müssen wir uns zunächst unter [NES1] als Entwickler registrieren und einen neuen Client anlegen.

Dieser Client definiert zwei URLs: die Authorization- und die Access-Token-URL. Zuerst öffnen wir die Authorization-URL im Browser. Nach einem Login in den Nest-Account erhalten wir einen PIN-Code. Dieser Code kann nun in die Access-Token-URL eingefügt und mittels einem HTTP POST an den OAuth-Server gesendet werden. Dies können wir zum Beispiel mit `curl` erledigen. Der Server antwortet mit einem OAuth2-Access-Token, das wir nun verwenden können, um mit dem Nest-Webservice zu kommunizieren. Den Access-Token hinterlegen wir in der Konstanten `ACCESS_TOKEN` in der Klasse `NestBindingConstants`.

Der Nest-Webservice implementiert eine Firebase-Programmierschnittstelle. Um aus Java heraus mit dem Webservice kommunizieren zu können, benötigen wir den Firebase Client Connector. Dessen Fat-JAR, OSGi-ready mit allen Abhängigkeiten im Bauch, kann unter [Firebase] heruntergeladen werden. Es muss darauf geachtet werden, dass die JVM-Variante verwendet wird und nicht das Android-spezifische JAR. Zum Zeitpunkt dieses Artikels ist die Version 2.5.2 aktuell. Das JAR legen wir in einen neuen Ordner `lib`.

Nun öffnen wir `META-INF/MANIFEST.MF` in Eclipse, wechseln auf den Tab *Runtime* und fügen unter Classpath über die Schaltfläche *Add* die Datei `firebase-client-jvm-2.5.2.jar` hinzu. Auf dem Reiter *Build* aktivieren wir bei *Binary Build* die Checkbox beim Ordner `lib`.

Kommunikation mit dem Nest-Webservice

Da wir an verschiedenen Stellen des Bindings die Verbindung zum Nest-Webservice benötigen, bietet sich die Verwendung des Singleton-Entwurfsmusters an. Dazu legen wir die Klasse `NestWebservice` im Package `org.eclipse.smarthome.binding.nest` an, die einen Firebase-Client erzeugt und sich per OAuth 2 authentifiziert (s. Listing 1).

In Zeile 7 erstellen wir eine Instanz der Klasse `Firebase` mit der Webservice-URL als Argument. In Zeile 8 verwenden wir das OAuth2-Token, um uns beim Firebase-Webservice zu authentifizieren. Die Methode `onAuthenticationError` des `AuthResultHandlers` wird aufgerufen, wenn die Authentifizierung fehlgeschlagen ist, die Methode `onAuthenticated` wird aufgerufen, wenn die Authentifizierung erfolgreich war.

```
1 public class NestWebservice {
2  private final Logger logger = LoggerFactory.getLogger(getClass());
3  public static final NestWebservice INSTANCE = new NestWebservice();
4  private final Firebase client;
5
6  private NestWebservice() {
7    client = new Firebase("wss://developer-api.nest.com/");
8    client.authWithCustomToken(NestBindingConstants.ACCESS_TOKEN,
         new AuthResultHandler() {
9      @Override
10     public void onAuthenticationError(FirebaseError arg0) {
11       logger.error("Error while auth: {}", arg0.getMessage());
12     }
13
14     @Override
15     public void onAuthenticated(AuthData arg0) {
16       logger.info("Auth successful");
17     }
18   });
19 }
20
21 public Firebase getClient() {
22   return client;
23 }
24}
```

Listing 1: Kommunikation mit dem Nest-Webservice

Finden des Thermostats

Eclipse SmartHome hat das Konzept einer *Inbox*: Dies sind Things, die gefunden, aber noch nicht gekoppelt wurden. Die Inbox wird von sogenannten *Discovery Services* befüllt. Wie diese Services technisch Things finden, ist im dazugehörigen Binding geregelt. Das Hue-Binding von Eclipse SmartHome verwendet UPnP (Universal Plug and Play), um die Hue-Lampen im Netzwerk zu finden. Unser Binding verwendet die gerade geschaffene Klasse `NestWebservice`, um sich vom Nest-Webservice die verfügbaren Thermostate zu holen.

Um einen Discovery Service zu implementieren, legen wir eine neue Klasse namens `NestDiscoveryService` im Package `org.eclipse.smarthome.binding.nest.discovery` an und leiten von der Basisklasse `AbstractDiscoveryService` aus dem Eclipse-SmartHome-Framework ab. Der Konstruktor der Basisklasse erwartet zwei Argumente: Eine Liste der Things, die der Discovery Service finden kann, und einen Timeout in Sekunden, nachdem das Discovery spätestens beendet ist. Die abstrakte Methode `startScan()` muss von uns implementiert werden und wird vom Framework aufgerufen, wenn der Benutzer über die Oberfläche nach neuen Things sucht (s. Listing 2).

In Zeile 1 wird an die URL `/devices/thermostats` ein `ChildEventListener` gehängt. Firebase ruft diesen Listener auf, sobald ein neues Thermostat hinzugefügt wird. Der Listener wird auch für bereits vorhandene Thermostate aufgerufen. In Zeile 5 lesen wir aus dem `DataSnapshot` die eindeutige ID des Thermostats aus. In Zeile 6 erstellen wir eine ThingUID, einen eindeutigen Identifizierer für ein Thing bestehend aus der Typ-Bezeichnung und der ID des Things selbst (in diesem Fall der Einfachheit halber festkodiert auf „thermostat“). Über die Klasse `DiscoveryResultBuilder` wird in Zeile 8 ein `DiscoveryResult` für diese `ThingUID` erstellt. Es wird noch das Label des Things gesetzt (dieses wird dem Benutzer in der Inbox angezeigt) sowie die ID des Thermostats als Property in das `DiscoveryResult` gespeichert. Zu guter

Letzt wird in Zeile 11 das `DiscoveryResult` an die Inbox per `thingDiscovered()` aus der Basisklasse übergeben.

Damit die Discovery funktioniert, muss der Discovery Service noch beim Framework registriert werden. Dazu legen wir im Ordner `OSGI-INF` eine neue Datei namens `NestDiscovery.xml` an (s. Listing 3). Durch diese Datei registrieren wir unsere Klasse `NestDiscoveryService` als Discovery Service.

```
1 NestWebservice.INSTANCE.getClient().child("/devices/thermostats")
2   .addChildEventListener(new ChildEventListener() {
3     @Override
4     public void onChildAdded(DataSnapshot arg0, String arg1) {
5       String id = (String) arg0.child("device_id").getValue();
6       ThingUID uid =
7         new ThingUID(NestBindingConstants.THING_TYPE_SAMPLE,
                       "thermostat");
8       DiscoveryResult result =
          DiscoveryResultBuilder.create(uid)
          .withLabel("Nest thermostat")
9         .withProperty("thermostat-id", id).build();
10
11      thingDiscovered(result);
12    }
13  });
```

Listing 2: Finden des Thermostats

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
                 immediate="true" modified="modified"
3                name=
  "org.eclipse.smarthome.binding.nest.discovery.NestDiscoveryService">
4   <implementation class=
  "org.eclipse.smarthome.binding.nest.discovery.NestDiscoveryService"/>
5   <service>
6     <provide interface=
        "org.eclipse.smarthome.config.discovery.DiscoveryService"/>
7   </service>
8 </scr:component>
```

Listing 3: Registrierung des Discovery Service

Der erste Probelauf

Nun können wir unser Binding das erste Mal ausprobieren. Dazu öffnen wir in Eclipse das Run-Menü, klicken auf *Run Configurations* und wählen die Konfiguration *SmartHome Runtime* aus. Unter dem Reiter *Plugins* muss bei unserem Nest-Binding das *Start Level* auf 0 und *Auto-Start* auf `true` stehen. Nun können wir über die Schaltfläche *Run* Eclipse SmartHome mit unserem Binding starten.

Die Web-Benutzungsoberfläche ist nun unter `http://localhost:8080/ui/index.html` erreichbar. Dort kann im Bereich Inbox eine Discovery gestartet und damit ein Nest-Thermostat gefunden werden. Damit wir die Nest-Funktionen nutzen können, müssen wir nun ein Thing mitsamt seiner Channels anlegen.

Anlegen des Things

Ein Thing besteht aus drei Dateien:

- der Definition des Things mit seinen Channels,
- einem Handler des Things und
- einer *Factory*, die das Thing erstellt.

Das Skript hat diese Dateien bereits angelegt. Als Erstes definieren wir die Channels unseres Things. Dazu öffnen wir die Datei `ESH-INF/thing/thing-types.xml`. In dieser wird ein Thing für das Nest-Thermostat definiert. Jedes Thing kann mehrere Channels haben, die im Element `<channels>` definiert werden. Jeder Channel hat einen Typ (`<thing-type>`-Element).

Wir passen nun die Definition unseres Things an (s. Listing 4). In Zeile 11 definieren wir den Channel `target_temperature`. Die Definition des Typs erfolgt ab Zeile 15. Der Typ des Channels ist `Number`, als Kategorie definieren wir `Temperature`. In Zeile 20 wird der Channel noch als les- und schreibbar angegeben, außerdem soll der Wert des Channels mit einer Nachkommastelle und mit Celsius dargestellt werden.

Nun muss noch die Datei `NestBindingConstants` entsprechend angepasst werden: Die Verknüpfung von XML und Java erfolgt über Strings, die den gleichen Wert haben müssen. Diese sind in der Klasse `NestBindingConstants` definiert (s. Listing 5).

Wenn der Anwender in der Inbox beim Thermostat auf den *Approve*-Button klickt, wird in der Klasse `NestHandlerFactory` die Methode `supportsThingType` mit dem Thermostat-Typen aufgerufen. Wenn die Methode `true` zurückgibt, wird der Handler des Things durch die Methode `createHandler` erstellt. Dieser Code wurde bereits generiert. Für das Thermostat wird eine Instanz der Klasse `NestHandler` erstellt.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <thing:thing-descriptions bindingId="nest"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:thing=
      "http://eclipse.org/smarthome/schemas/thing-description/v1.0.0"
5     xsi:schemaLocation=
"http://eclipse.org/smarthome/schemas/thing-description/v1.0.0
 http://eclipse.org/smarthome/schemas/thing-description-1.0.0.xsd">
6   <thing-type id="thermostat">
7     <label>Nest Thermostat</label>
8     <description>Binding for the Nest thermostat</description>
9
10    <channels>
11      <channel id="target_temperature" typeId="target_temperature"/>
12    </channels>
13  </thing-type>
14
15  <channel-type id="target_temperature">
16    <item-type>Number</item-type>
17    <label>Target temperature</label>
18    <description>The target temperature.</description>
19    <category>Temperature</category>
20    <state readOnly="false" pattern="%.1f C" />
21  </channel-type>
22 </thing:thing-descriptions>
```

Listing 4: Definition des Things

```
// List of all Channel ids
public final static String TARGET_TEMPERATURE = "target_temperature";
```

Listing 5: Verknüpfung zwischen XML- und Java-Code

Setzen der Zieltemperatur des Thermostats

Die Klasse `NestHandler` erbt von `BaseThingHandler` und enthält zwei Methoden: `initialize()` wird von Eclipse SmartHome aufgerufen, nachdem der Handler erstellt wurde. `handleCommand()` wird aufgerufen, wenn der Benutzer den Wert eines Channels ändert. Als Argumente erhält die Methode `handleCommand()` die ID des Channels und ein `Command`-Objekt, das den neuen Wert des Channels enthält (s. Listing 6).

In Zeile 2 wird überprüft, ob der Benutzer den `target_temperature`-Channel ändern möchte. Falls dies der Fall ist, wird aus

Architektur mit Kultur

Am 12. und 13. Mai versammelten sich in Hamburg Entwickler, Softwarearchitekten, Projektmanager und Entscheider. Die SEACON hat sich inzwischen als Branchentreff für IT-Profis etabliert. 180 Teilnehmerinnen und Teilnehmer in diesem Jahr können das bestätigen.

Architektur ist nicht mein Thema. Dachte ich. Doch *Johann Hartmann*, CTO („Chief Tailwind Office“) und Gründer der Mayflower GmbH, brachte im Eröffnungsvortrag Softwarearchitektur mit Organisationskultur in einen direkten Zusammenhang. Er hat mich mit Erfahrungen, Zusammenhängen und vor allem mit Einsichten überrascht. Und nicht nur mich. Das Publikum wünschte sich eine Vertiefung des Teilaspekts „Rollen statt Positionen“ (**siehe Abbildung 1**) im Open Space. Und so verriet uns Hartmann mehr über das Zusammenarbeitsmodell bei Mayflower. Die Teilnehmer zeigten großes Interesse daran, wie dort gearbeitet und gemanagt wird. Diese Session war besser besucht als so mancher Vortrag; offensichtlich ein spannender Stoff. Denn dass die Idee von New Work gut klingt, ist das eine. Dass sich dies irgendwo in der IT-Welt bewährt hat, davon hört man nur selten.

Ähnliches thematisierte später *Bernd Oestereich* in der Keynote „Führung ist zu wichtig, um sie nur Führungskräften zu überlassen“. Er stellte vier Führungsmodelle vor. Vorgesetzte Führung (d. i. der klassische Command-and-Control-Stil), partizipative, dienende und kollegiale Führung. Vorgesetzte Führung scheitert, wenn der Kontext komplex wird. Um weg von diesem hin zu agileren Führungsstilen zu kommen, muss man den Unternehmen einfache Methoden anbieten. Denn Entscheider brauchen die Gewissheit, dass sie sich in kleinen Schritten auf den Weg zum Erfolg machen können – statt an großen Schritten zu scheitern. Sie wünschen sich Rezepte, an denen sie sich orientieren können. Als Beispiel nannte Oestereich eine Methode, bei der nicht die Zustimmung gezählt, sondern der Widerstand minimiert wird.

Benjamin Seidler, agiler Coach, führte hervorragend ins „Product Canvas“ ein. Das ist ein Schema, das die wichtigsten Fragen des Requirements Engineers enthält. Hier bündelt man Szenarien, Personas, Produktvision, Rollen usw. übersichtlich und für alle sichtbar. Als Beispiel stellte er eine App vor, die mir während eines Kinobesuchs die langweiligen Stellen meldet, damit ich meine WC-Pause planen kann, ohne den Clou zu verpassen. Sehr anschaulich!

Ralf Westphal, Clean Code Developer, stellte mit „Story Slicing“ vor, wie man die Kommunikation zwischen Product Owner und Entwicklungsteam verbessern kann, indem beide sich auf analogen Strukturebenen verständigen. Anders formuliert: Anforderungen werden so strukturiert im Code abgebildet, dass bei Änderungen oder Fehlern die fachliche Information „Applikation, Applikationsteil, Maske, Interaktionselement ...“ präzise zum Gegenstück im Code führt: „Service, Component, Class, f() ...“. Ein heißes Thema, bei dem die Meinungen im Publikum weit auseinander gingen. Mir selbst ging‘s nicht besser. Zuerst dachte ich: „Hmm, naja.“ Dann: „Geht gar nicht!“ Und schließlich: „Vielleicht ist doch was dran.“

Thomas Much, agiler Coach und Softwareentwickler, und *Stephan Kraus*, Abteilungsleiter E-Commerce bei OTTO, berichteten von ihren Erfahrungen, wie man Pair Programming erfolgreich einsetzt. Much berichtete vor allem davon, wie er selbst gemeinsam mit den Entwicklern im Zweierteam tief in den Code eintauchte, um die Methode zielführend zu etablieren.

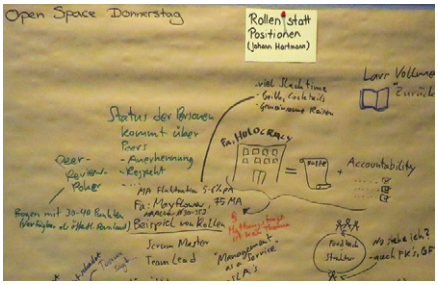

Abb. 1: Open-Space-Ergebnis „Rollen statt Positionen“ (Ausschnitt).

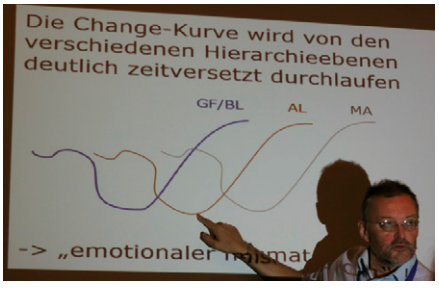

Abb. 2: Abgeleitet von Miss Marple: Zeitversetzte Wahrnehmungen verschiedener Beteiligter.

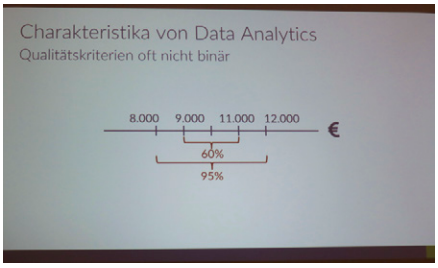

Abb. 3: Agile Data Analytics: Unschärfe vs. Schwankung.

Was gab‘s sonst noch?

- Wieso und was man von Miss Marple (**siehe Abbildung 2**, Peter Schnell und Kurt Jäger) über agiles Arbeiten lernen kann: Rollen, Tools und Fallstricke; schön mit Filmausschnitten.
- Wie das Gehirn uns reinlegt, zum Beispiel warum so oft die anderen Schuld sind (Vortrag „Brain Patterns“ von Jan Gentsch und Julia Dellnitz).
- Agile Data Analytics (**siehe Abbildung 3**, Collin Rogowski) schließlich führte in die Welt schwieriger Entscheidungen ein, zum Beispiel bei der Frage, ob ich eine Antwort mit kleiner Schwankung und viel Unschärfe oder eine mit beträchtlicher Schwankung und wenig Unschärfe bevorzuge.

Die nächste SEACON startet im Mai 2017, wie immer in Hamburg. ‖

Über die Konferenz berichtet

‖ Maria Oelinger .
(maria.oelinger@knh.de)
ist IT-Systemanalytikerin bei Kindernothilfe e. V. Sie ist unter anderem bewandert im Anforderungsmanagement.

dem Thing in Zeile 3 die ID des Thermostats extrahiert. Diese ID wurde im Discovery Service gesetzt. In Zeile 5 wird der neue Wert des Channels aus dem `Command`-Objekt extrahiert. Jeder Channel-Typ hat ein dazugehöriges `Command`-Objekt. In Zeile 6 wird der Nest-Webservice verwendet, um die Zieltemperatur des Thermostats zu setzen.

Wenn wir das Binding nun in Eclipse SmartHome starten, das Thermostat koppeln und unter dem Reiter *Control* den Channel *Target Temperature* ändern, übermittelt unser Binding den neuen Wert an den Nest-Cloudwebservice, welcher wiederum die Zieltemperatur auf dem Thermostat einstellt.

```
1 public void handleCommand(ChannelUID channelUID, Command command) {
2   if (channelUID.getId().equals(TARGET_TEMPERATURE)) {
3     String thermostatId =
4       getThing().getProperties().get("thermostat-id");
5     double value = ((DecimalType) command).doubleValue();
6     NestWebservice.INSTANCE.getClient().child("/devices/thermostats")
        .child(thermostatId)
7       .child("target_temperature_c").setValue(value);
8   }
9 }
```

Listing 6: Setzen der Zieltemperatur

Reagieren auf Änderungen der Zieltemperatur

Nun fehlt noch die umgekehrte Richtung: Wenn der Benutzer die Zieltemperatur am Thermostat ändert, soll sich auch der Wert des Channels im Binding ändern. Dies erreichen wir, indem wir uns in der Methode `initialize()` am Firebase-Webservice registrieren. Damit benachrichtigt uns der Nest-Webservice, wenn der Benutzer die Temperatur am Thermostat ändert (s. Listing 7).

```
1 String thermostatId = getThing().getProperties().get("thermostat-id");
2
3 NestWebservice.INSTANCE.getClient().child(
    "/devices/thermostats").child(thermostatId)
4     .child("target_temperature_c").addValueEventListener(
      new ValueEventListener() {
5   @Override
6   public void onDataChange(DataSnapshot arg0) {
7     double value = (Double) arg0.getValue();
8
9     ChannelUID channel = new ChannelUID(getThing().getUID(),
        TARGET_TEMPERATURE);
10    updateState(channel, new DecimalType(value));
11  }
12 });
```

Listing 7: Reagieren auf Änderungen der Zieltemperatur

In Zeile 1 lesen wir die ID des Thermostats aus den Properties des Things. In Zeile 3 verwenden wir den `NestWebservice`, um einen Listener auf die URL `/devices/thermostats/{id}/target_temperature_c` zu registrieren. Der Handler wird aufgerufen, wenn der Benutzer die Temperatur am Thermostat ändert. Über das `DataSnapshot`-Argument können wir per Methode `getValue()` in Zeile 7 auf den aktuellen Wert des Thermostats zugreifen. In Zeile 9 konstruieren wir die ID des Channels, diese besteht aus der ID des Things und dem Wert der Konstanten `TARGET_TEMPERATURE`, die die Verbindung mit dem in der XML-Datei definierten Channel herstellt. Über die Methode `updateState`, die wir von `BaseThingHandler` geerbt haben, können wir mit der Channel-ID in Zeile 10 den neuen Wert des Channels setzen.

Starten des Bindings in openHAB 2

Zuerst laden wir uns die neuste Version von openHAB 2 herunter [OPEN]. Dann muss das Binding per `mvn clean install` gebaut werden. Die dabei erzeugte JAR-Datei muss nun in den Ordner `addons` von openHAB 2 kopiert werden. Die Property `ui` in `conf/services/addons.cfg` muss auf den Wert `paper` gesetzt sein, damit die Benutzungsoberfläche aktiviert ist. Nun kann openHAB 2 über `start.bat` (bzw. `.sh`) gestartet werden.

Nach dem Start können Sie die Benutzungsoberfläche unter `http://localhost:8080/ui/index.html` erreichen und das Nest-Thermostat nutzen. Auch wenn Sie kein Nest-Thermostat besitzen, können Sie das Binding trotzdem ausprobieren: Nest stellt unter [NES2] einen Simulator bereit, der auch mit unserem Binding funktioniert.

Fazit

Eclipse SmartHome ist ein mächtiges Framework für das Smarthome, in das mit wenig Aufwand neue Geräte integriert werden können. Durch die aktive Open-Source-Entwicklung und die Schirmherrschaft der Eclipse Foundation hat es gute Chancen, das führende Framework im Bereich Smarthome zu werden.

Dank der stabilen und ausgereiften Technologien Java und OSGi müssen keine exotischen neuen Technologien gelernt werden. OpenHAB 2 ist eine Smarthome-Plattform auf Basis des Eclipse-SmartHome-Frameworks. Durch die Verwendung von Java lässt sich openHAB 2 auf verschiedensten Plattformen ausführen. Es läuft zum Beispiel auch problemlos auf einem Raspberry Pi und bietet damit einen kostengünstigen Einstieg in das vernetzte Zuhause.

Links

[ECL1] Eclipse SmartHome, Documentation Overview, `https://eclipse.org/smarthome/documentation/index.html`
[ECL2] Items, `https://eclipse.org/smarthome/documentation/concepts/items.html`
[ECL3] Setting Up a Development Environment, `https://eclipse.org/smarthome/documentation/development/ide.html`
[Firebase] `https://www.firebase.com/docs/android/quickstart.html`
[GitH] `https://github.com/qaware/building-iot-2016`
[NES1] Nest Developers, `https://developers.nest.com`
[NES2] `https://developers.nest.com/documentation/cloud/home-simulator`
[OPEN] Maven-Projekt openHAB-Distribution, `https://openhab.ci.cloudbees.com/job/openHAB-Distribution`

Moritz Kammerer ist Softwareentwickler bei der QAware GmbH. Er hat bereits diverse Bindings für Eclipse SmartHome entwickelt und interessiert sich von Anfang an für das Internet of Things. Nebenbei ist er Autor diverser Open-Source-Projekte.
E-Mail: moritz.kammerer@qaware.de