Microservices Arrived at Your Home

by Martin Večeřa · May. 11, 16 · Integration Zone

The Integration Zone is brought to you in partnership with 3scale. Take control of your APIs and get a free t-shirt when you complete the 3step Challenge.

As there are more and more things being connected to the Internet, necessarily there is a need to integrate these devices together. We have some great opportunities to be really productive in partitioning huge problems into small and even smaller and solve them one by one. We can easily develop a simple service, put it into a Docker container and deploy it to any cloud solution. Later we can connect the services together and let them do a huge job.

The services are being developed in worldwide spread teams and integrated together as needed. The same good old service oriented architecture principles apply here as well. However, the integration part is the one that has changed from the past. We no longer put the services together into a single application container. We rather deploy them standalone. This freedom allows us to spawn more instances of the same service to handle higher load, it is more failure resilient (one failed deployment does not necessarily break the others when we use circuit breakers), we can use less powerful virtual machines for hosting the services, and we believe you can come up with even more advantages.

However, how do we develop these so called microservices? Or even better, how do we reuse our existing code and convert it to microservices? How could our developers use the skills they already have to develop the microservices? And how do we leverage all the great enterprise solutions deployed behind the gateway? It seems like we might use some smart glue. Fortunately, there is one, it is called SilverWare.

By an example, we would like to show you how you can easily develop, integrate and deploy microservices using the skills you already know - Java and CDI.

The scenario is built around controlling an intelligent home. We will be initiating actions from a mobile device (phone, tablet), processing them in a Business Rules Management System, creating commands and passing them through a workflow to individual things/actuators, and monitoring the intelligent home status that will be displayed on the mobile device and in the rules system to derive further commands. The status of the Things is going to be cached for later processing by the Rules and for the case the Things are offline or cannot provide their status. All this gets along with the Reference architecture of the Internet of Things.

The software pieces used in the example are:

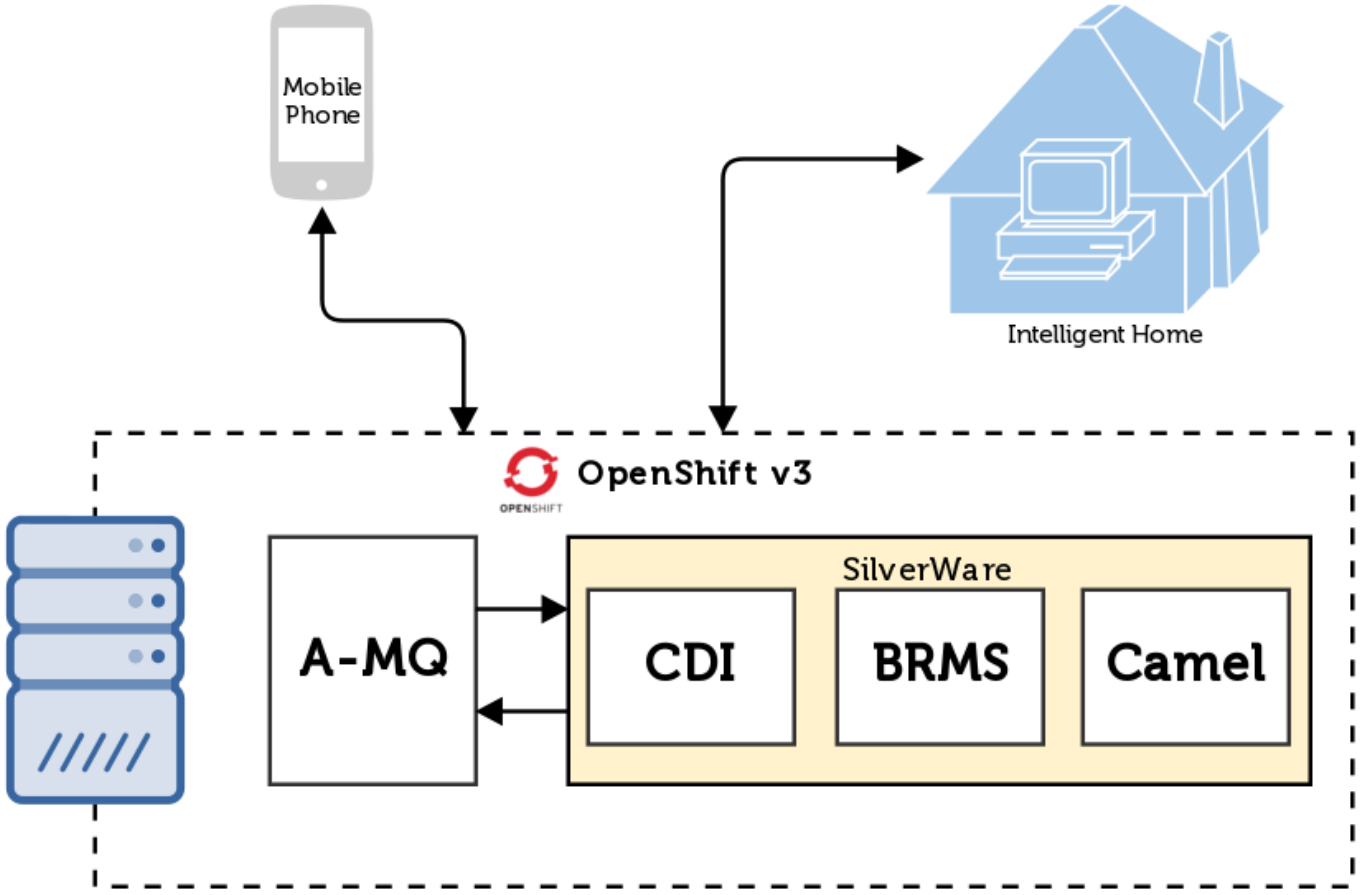
- JBoss Business Rules Management System(with its core component called Drools) for the decision engine
- JBoss A-MQ for MQTT messaging and topics to pass events/messages between services

Apache Camel (that is part of JBoss Fuse) for integrating services, topics and Things, and also for implementing the workflow to pass Commands to individual Things

- OpenShift Enterprise 3 to host all the previous parts and microservices
- SilverWare as the secret sauce that binds it all together

The cool part is a real 1:20 model of an intelligent home that contains the following “live” parts: LED lights, TV set (only audio), servo controlled door and window, A/C fan, fireplace (with a light bulb emulating the fire), and a RFID reader to find out who is at home.

We have three key players in the demo - the mobile phone invoking actions and monitoring the status of the whole system, the intelligent home model that receives commands and reports its state, and the OpenShift v3 installation hosting all the software components.

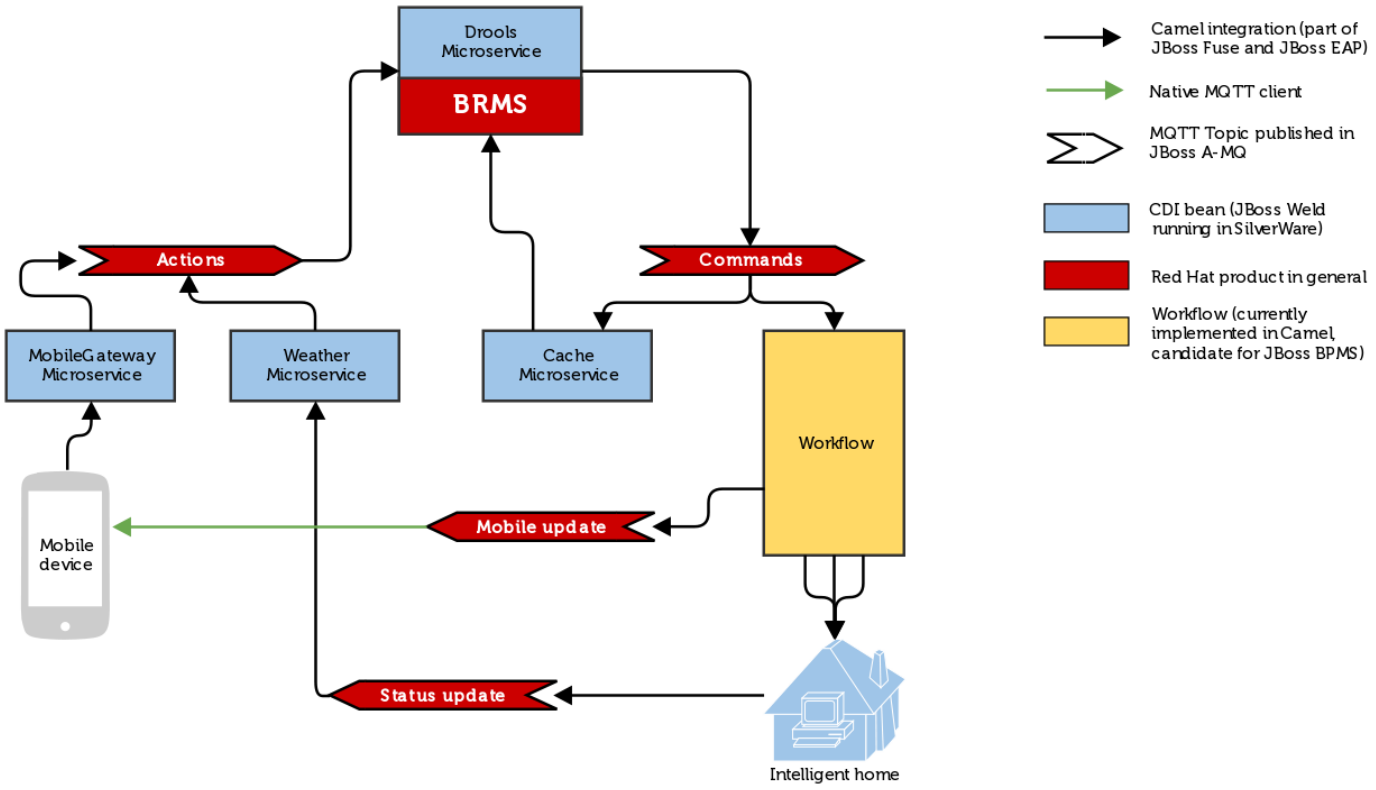


The home hardware is driven by Raspberry Pi running Bulldog, the universal Java library for accessing hardware pins on ARM boards, and SilverSpoon, the set of Camel components to communicate with various sensors. All the controls are available to the outside world via REST. There is JBoss A-MQ running on the Raspberry Pi hosting a MQTT topic where the information about the home are posted.

The mobile phone invokes Actions via REST requests to the microservices hosted on OpenShift. It also displays status about the home from a MQTT topic hosted on JBoss A-MQ running in OpenShift.

In the parts that we have implemented we stuck to the IoT Reference Architecture. The devices (mobile phone and the home) generate actions, these are processed through the Business Rules Management System (Drools) and the resulting Commands are passed through a workflow that sends them to the correct consumers represented by the individual Things. We also cache the state of the Things for case they do not report their state back or they go offline. Simply the latest state sent through a Command for the particular Thing is stored in the cache. For simplicity, we use a hash map as a cache. Later we plan to add a distributed cache like JBoss Data Grid.

Currently, the workflow is implemented using Camel routes, however, a full-blown Business Process Management System (likeJBoss BPM Suite) can be used here.



It is all implemented using four microservices written as CDI beans. The microservices are hosted in a single fat JAR file using the SilverWareplatform. SilverWare is just an integration layer that manages lifecycle of various frameworks (i.e. service providers) and their components. Such a service provider is Weld (CDI reference implementation) and its components are CDI beans. More details on SilverWare framework and its idea can be found in our previous article.

The source code for the intelligent home and all the microservices can be found atGitHub.

Let's have a look on a typical use case of the whole system. A user pushes a button on their mobile phone which creates a corresponding Action that is consumed by the MobileGateway Microservice. This microservice has a public REST API and converts the requests to Java objects that are serialized and sent to the Actions MQTT topic. All the Actions are picked up by the Drools Microservice and passed through the Business Rules Engine. Based on the

user defined rules, there are typically several Commands generated as a response to an Action. The Commands are sent to the Commands MQTT topic. The state changes carried in the Commands are stored in a cache for later use in the Rules Engine. At the same time the Commands are passed to the workflow which routes the Commands to the corresponding REST APIs installed in the intelligent home. Some of the Commands can also provide updates to the mobile phone. We'll get to this in a moment. The intelligent home periodically publishes status updates like indoor temperature, humidity and RFID tags present in the home. These are published to the Status update MQTT topic and consumed in the Weather Microservice (probably a candidate to be renamed) which converts them to Actions. Such status update actions generate Commands that provide status information back to the mobile phone through the Mobile update topic. By this we covered all the microservices and routes in the diagram above.

Now we can inspect some pieces of the implementation. For example the Drools Microservice that consumes Actions from the Action topic, passes them through Business Rules Engine and sends the resulting Commands to the Commands workflow.

```
@Microservice
public class DroolsMicroservice {

  private static final Logger log = LogManager.getLogger(DroolsMicroservice.class);

  // KieSession is not thread safe, we need to synchronize calls
  private static Semaphore sync = new Semaphore(1);

  @Inject
  @KSession
  private KieSession session;

  @Inject
  @MicroserviceReference
  private ProducerTemplate producer;

  @Inject
  @MicroserviceReference
  private CacheMicroservice cache;

  public void processActions(final List<Action> actions) throws InterruptedException {
     log.info("Firing rules for action {}", actions);

     sync.acquire();

     try {
        final EntryPoint entryPoint = session.getEntryPoint("actions");
        session.setGlobal("producer", producer);
        session.setGlobal("cache", cache.getCache());
        session.registerChannel("commands", cmd -> producer.asyncSendBody("direct:comma

        actions.forEach(entryPoint::insert);

        session.fireAllRules();

     } finally {
        sync.release();
     }
  }

}
```

In the CDI bean annotated with @Microservice, we inject BRMS Knowledge session, Camel message producer and Cache Microservice. Actions from the Actions topic are directly routed to the processActions() method. We synchronize the calls as KieSession is not thread-safe. The Actions are passed as facts through an entry point to the Knowledge Session as well as the Camel message producer and the current cache with state of individual Things. For

the output of newly created Commands we register a channel that sends the Commands to the corresponding Camel route connected to the Commands workflow.

A sample business rule that reacts to the mobile phone's button to set an evening mood in the home looks like this:

```
rule "Evening mood action"
when
  $mood: MoodAction(mood == MoodAction.Mood.EVENING) from entry-point "actions"
then
  System.out.println("Processing mood evening");
  channels["commands"].send(new BatchLightCommand(
     new LightCommand(LightCommand.Place.ALL, LedState.ON),
     new LightCommand(LightCommand.Place.LIVINGROOM_FIREPLACE, new LedState(10, 10, 10)
     new LightCommand(LightCommand.Place.LIVINGROOM_LIBRARY, new LedState(10, 10, 10)),
     new LightCommand(LightCommand.Place.LIVINGROOM_COUCH, new LedState(10, 10, 10))
  ));
  channels["commands"].send(new FireplaceCommand(FireplaceCommand.Fire.HEAT));
  channels["commands"].send(new MediaCenterCommand(MediaCenterCommand.Media.NEWS));
end
```

Upon the receival of the MoodAction with the Evening mood value, we create light commands to set light conditions, we set on the fireplace and turn on the media center with the news channel.

More details on Drools can be find in Drools Documentation.

Apache Camel is a great project to implement Enterprise Integration Patterns. Camel is very easy to use because the integration Camel routes are self describing. For example the route to read messages from the Actions topic and send them to the Drools Microservice looks like follows:

```
from("mqtt:inActions?subscribeTopicName=ih/message/actions&userName=mqtt&password=mqtt&
      .unmarshal().serialization().bean("droolsMicroservice", "processAction");
```

The source code is managed by Maven. This allows us to easily add the SilverWare secret sauce to bring all the components to life. This is done by adding a few dependencies to our project's pom.xml (in addition to the components we already use in the project).

```
    <dependency>
       <groupId>io.silverware</groupId>
       <artifactId>microservices</artifactId>
    </dependency>
    <dependency>
       <groupId>io.silverware</groupId>
       <artifactId>cdi-microservice-provider</artifactId>
    </dependency>
    <dependency>
       <groupId>io.silverware</groupId>
       <artifactId>camel-microservice-provider</artifactId>
    </dependency>
    <dependency>
       <groupId>io.silverware</groupId>
       <artifactId>camel-cdi-integration</artifactId>
    </dependency>
    <dependency>
       <groupId>io.silverware</groupId>
       <artifactId>drools-microservice-provider</artifactId>
    </dependency>
```

This automatically takes care of all the resources that are known to SilverWare and its providers. In this case, all CDI beans annotated with @Microservice, all Camel routes and Knowledge JAR files are discovered, made available to other components and started. As a result we get an executable JAR file with all the libraries in a separate lib directory. The overall size of the deployment is roughly 30MB.

It is now possible to deploy the application to OpenShift v3 and manage it there. For this reason we created a template to prepare everything that is needed for a successful deployment to OpenShift v3. One of the core and very interesting components isS2I(Source To Image) which is a tool for building “ready to run” Docker images from sources. However, given its complexity, we will follow up with another article describing how to deploy SilverWare Microservices to OpenShift v3.

In this demonstration we showed that we can easily develop simple microservices by using the SilverWare framework, we can invoke BRMS Knowledge sessions from them and link it all together by Camel and JBoss A-MQ. All that running in OpenShift v3 deployed as Docker containers. It does not need anything else than writing a few microservices and Camel routes - the components that provide the real business value and nothing else, no

boilerplate code was needed. We did not need to learn any new principles and tools.

The Integration Zone is brought to you in partnership with 3scale. Learn how API providers have changed the way we think about integration in The Platform Vision of API Giants.