Bootware Entwicklerhandbuch

Inhaltsverzeichnis

1 Einleitung

Die Bootware ist ein System aus mehreren Komponenten, die zusammen on-demand Workflow Middleware und Provisioning Engines in Cloud-Umgebungen zur Verfügung Stellen können. Die Bootware an sich ist generisch angelegt und wird durch spezifische Adapterkomponenten und Plugins an die jeweilige Einsatzumgebung angepasst. Dieses Entwicklerhandbuch gibt einen Überblick über die technischen Einzelheiten der Bootware. Es beschränkt sich dabei auf die Zusammenarbeit mit dem SimTech System.

2 Komponenten

Das komplette Bootwaresystem besteht aus 5 Teilen:

1. **Das modifizierte Eclipse BPEL UI Plugin:** Dieses Plugin bindet die Bootware in den bereits bestehenden Deploy Prozess ein.
2. **Das Bootware Eclipse Plugin:** Dieses Plugin wird durch das modifizierte Eclipse BPEL UI Plugin aufgerufen und integriert die Bootware in Eclipse.
3. **Die Local Bootware:** Der lokale Teil der Bootware, der durch das Bootware Eclipse Plugin gestartet wird und den Bootstrappingprozess einleitet indem es die Remote Bootware provisioniert.
4. **Die Remote Bootware:** Das serverseitige Gegenstück zur Local Bootware. Es wird von der Local Bootware z.B. in einer Cloud-Umgebung deployed und kann dann die Provisioning Engines sowie die Workflow Middleware provisionieren.
5. **Das Bootware Repository:** Wird von der Local und Remote Bootware aufgerufen wenn diese Plugins benötigen oder eine Anfrage auf einen Context abbilden müssen.

3 SVN Repository

Der Code für die Bootware Komponenten ist im SVN Repository in zwei Unterordner verteilt. Das modifizierte Eclipse BPEL UI Plugin befeindet sich in *BPELDesigner/org.eclipse.bpel.ui*. Alle restlichen Bootware Komponenten sind im Ordner *Bootware* untergebracht. Dieser enthält wiederum mehrere Unterordner die im Folgenden genauer beschrieben werden.

- **/core**: Enthält den Code der Bootware Core Library, die von den anderen Komponenten genutzt wird. Hier befindet sich der Großteil des Codes der Bootware.
    - **/events**: Enthält diverse Event Klassen, die Events definieren die über den Event Bus ausgetauscht werden können.
    - **/exceptions**: Enthält diverse Exception Klassen, die von der Bootware benutzt werden
    - **/plugins:** Enthält die Interfaces der verschiedenen Plugintypen die von der Bootware geladen werden können und die abstrakte Grundimplementierung.
    - **AbstractStateMachine.java:** Definiert den Großteil der State Machines die von der Local und Remote Bootware benutzt werden. Die hier implementierten Methoden werden

von beiden Bootwares genutzt. Sie können diese bei bedarf in ihrer eigenen Implementierung überschreiben oder erweitern.

    - **...Wrapper.java:** Wrapper Klassen, die das Marshalling von Maps in XML vereinfachen (jaxb kann Java Maps nicht direkt in XML umwandeln).
    - **ContextMapper.java:** Eine Klasse, die User Context auf Request Context abbildet, indem sie das Bootware Repository aufruft.
    - **EventBus.java:** Ein Event Bus der PubSub zur Verfügung stellt, z.B. für die Event Plugins.
    - **PluginManager.java:** Eine Wrapper Klasse um die OSGi Funktionalität. Sie lädt und entlädt Plugins und holt sich diese aus dem Bootware Repository, falls diese lokal noch nicht existieren.

- **/distribution**: Enthält Regeln die die finale Distribution erstellen. Die finalen Ergebnisse eines Kompiliervorgangs landen hier im Unterordner */target*.
- **/eclipse**: Enthält den Code des Bootware Eclipse Plugins. Es startet die Local Bootware, ruft sie auf um das SimTech SWfMS zu provisionieren und konfiguriert Eclipse mit den Rückgabewerten. Es dient als Mittelsmann zwischen der generischen Local Bootware und dem speziellen SimTech Modeler.
- **/local**: Enthält den Code der Local Bootware. Diese wird auf dem lokalen Rechner des Benutzers ausgeführt und ist hauptsächlich dafür zuständig die Remote Bootware in einer Cloud zu provisionieren.
- **/mappings**: Enthält eine Ordnerstruktur mit verschiedene Request Context XML Dateien, die vom Repository für die Abbildung von Anfragen auf Request Contexte genutzt wird.
- **/plugins**: Enthält diverse Plugins verschiedener Typen, die das Repository der Local und Remote Bootware zur Verfügung stellt und die diese während des Bootstrappingprozesses laden und nutzen.
    - **Application Plugins** installieren eine bestimmte Anwendung, hier z.B. die Remote Bootware oder OpenTOSCA.
    - **Communication Plugins** bauen einen Kommunikationskanal zu einem Rechner auf, hier z.B. mit SSH.
    - **Event Plugins** können auf Events am Event Bus reagieren.
        - **ConsoleLogger** gibt die Events auf der Konsole aus.
        - **FileLogger** schreibt die Events in eine Textdatei.
        - **ZeroMQ-Publisher** veröffentlicht Events an einem bestimmten Port.
        - **ZeroMQ-Subscriber** empfängt Events von einem bestimmten Port. Wird zusammen mit **ZeroMQ-Publsiher** benutzt um Events von der Remote Bootware an die Local Bootware zu schicken.
    - **Provision Plugins** provisionieren ein Service Package mit einer bestimmten Provisioning Engine (hier z.B. das SimTech SWfMS mit OpenTOSCA).

- **Resource Plugins** erstellen eine Bestimmte Ressource (hier z.B. EC2 Instanzen).

- **/remote**: Enthält den Code der Remote Bootware. Sie wird von der Local Bootware aufgerufen um das SimTech SWfMS zu provisionieren. Später wird sie vom Provisioning Manager des SimTech SWfMS aufgerufen um Provisioning Engines zu provisionieren.
- **/repository**: Enthält den Code des Bootware Repository. Es wird von der Local und Remote Bootware aufgerufen wenn diese einen User Context auf einen Request Context abbilden müssen oder eine Plugin benötigen.

4 Kompilieren

4.1 Vorbereitung

Bevor die Bootwarekomponenten kompiliert werden können gibt es folgendes vorzubereiten:

1. Maven 3.0 sollte installiert sein.
2. Java JDK 1.6 sollte installiert sein.

```
32  ————<plugin>
33  ————————<groupId>org.apache.maven.plugins</groupId>
34  ————————<artifactId>maven-compiler-plugin</artifactId>
35  ————————<version>3.1</version>
36  ————————<configuration>
37  ————————————<source>1.6</source>
38  ————————————<target>1.6</target>
39  ————————————<compilerArguments>
40  ————————————————<bootclasspath>/usr/lib/jvm/java-1.6.0-openjdk-i386/jre/lib/rt.jar</bootclasspath>
41  ————————————</compilerArguments>
42  ————————————<compilerArgument>-Xlint:all</compilerArgument>
43  ————————————<showWarnings>true</showWarnings>
44  ————————————<showDeprecation>true</showDeprecation>
45  ————————</configuration>
46  ————</plugin>
47
```

Abbildung 1: Angepasster bootclasspath in pom.xml

Nun muss in der *pom.xml* im Ordner *Bootware* noch der *bootclasspath* angepasst werden, damit dieser auf die *rt.jar* des Java JDK 1.6 zeigt. Ist das JDK beispielsweise unter */usr/lib/jvm/java-1.6.0-openjdk-i386* installiert, so muss hier */usr/lib/jvm/java-1.6.0-openjdk-i386/jre/lib/rt.jar* eingetragen werden (siehe Abbildung 1).

4.2 Kompilieren

Nun kann im Ordner *Bootware* der Befehl *mvn install* ausgeführt werden, um alle Komponenten der Bootware die sich in diesem Ordner befinden zu kompilieren. Das Ergebnis ist im *distribution* Ordner zu finden, unter *…/Bootware/distribution/target.* Hier sollten zwei Unterordner erstellt worden sein:

- **/bootware**: Enthält das Bootware Eclipse Plugin sowie die Local und Remote Bootware. Dieser Ordner kann so in den Eclipse *plugins* Ordner kopiert werden um die Bootware zu installieren (Siehe Benutzerhandbuch).

- **/repository**: Enhält das Bootware Repository sowie alle Plugins und Mappings. Dieser Ordner kann so auf einen Server kopiert werden um dort das Repository auszuführen (siehe Benutzerhandbuch).

Das modifizierte Eclipse BPEL UI Plugin muss separat kompiliert werden. Dazu wird im Ordner *BPELDesigner/org.eclipse.bpel.ui* der Befehl *mvn install* ausgeführt. Da das Plugin von anderen SimTech Komponenten abhängig ist, müssen diese Artefakte bereits im Maven Repository verfügbar sein, oder vorher kompiliert werden. Dazu kann im SVN Haupttrunk einmal das komplette *SimTech-EclipsePlugIns* Projekt kompiliert werden. Dazu muss *mvn install* im Ordner *SimTech-EclipsePlugIns/de.ustutt.simtech.master* ausgeführt werden. Danach sollten alle nötigen Artefakte im Maven Repository vorhanden sein und das modifizierte Eclipse BPEL UI Plugin kann wie oben beschrieben einzeln kompiliert werden. Das Ergebnis ist das *jar* Archiv *org.eclipse.bpel.ui-0.8.0-SNAPSHOT.jar* unter *…/BPELDesigner/org.eclipse.bpel.ui/target.* Dieses Archiv kann dann in den Eclipse *plugins* Unterordner kopiert werden um das alte Plugin zu überschreiben und die Bootware in den bestehenden Eclipse Prozess einzubinden (siehe Benutzerhandbuch).

5 Funktion

Im folgenden wird die Funktionalität der Bootware Schritt für Schritt beschrieben. Voraussetzung: Die Bootware wurde wie im Benutzerhandbuch beschrieben installiert und konfiguriert.

Anmerkung: An vielen Stellen in diesem Prozess wird überprüft, ob ein Prozess/Thread/Web Service/etc. bereits existiert und entsprechend anders reagiert. Zur Vereinfachung werden hier nicht alle dadurch mögliche Wege beschrieben, sondern nur der Standardweg, bei dem in der Regel noch nichts existiert/aufgerufen wurde.

5.1 Der Benutzer startet ein Prozessmodell

Durch Drücken des Start-Knopfes wird der komplette Prozess gestartet.

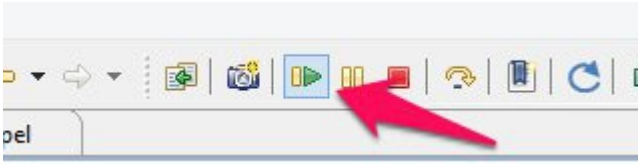

Abbildung 2: Starten eines Prozessmodells

Dies führt zuerst dazu, dass das modifizierte Eclipse BPEL UI Plugin aufgerufen wird, genauer *org.eclipse.bpel.ui.agora.actions.StartAction.run()*. Diese Methode wurde so modifiziert, das zuerst versucht wird das Bootware Eclipse Plugin zu laden, falls dieses existiert. Ist dies nicht der Fall, so wird das Prozessmodell wie zuvor in der unmodifizierten Version des Plugin deployed und gestartet, d.h. die das SimTech SWfMS muss manuell installiert und die Endpunkte in Eclipse konfiguriert werden. Ist das Bootware Eclipse Plugin allerdings vorhanden, so passiert folgendes:

1. Das Bootware Eclipse Plugin wird ausgeführt.
2. Es wird gewartet bis der Bootstrappingprozess beendet ist.

5.2 Das Bootware Eclipse Plugin wird ausgeführt

Hat das modifizierte Eclipse BPEL UI Plugin das Bootware Eclipse Plugin gefunden, so ruft es dessen *execute()* Methode auf. Dabei passiert folgendes:

1. Die Local Bootware wird gestartet und deren Output an die Bootware Konsole in Eclipse weitergeleitet. (*org.simtech.bootware.eclipse.BootwarePlugin.executeLocalBootware()*)

2. Der User Context und die Standardkonfiguration werden aus den XML Dateien eingelesen, die zuvor konfiguriert wurden (siehe Benutzerhandbuch). Abbildung 3 Zeigt einen exemplarischen User Context, der beschreibt, dass das in Amazon's Cloud (*aws-ec2*) die Provisioning Engine OpenTosca (*opentosca*) installiert werden soll, um dann mit dieser das Service Package *simtech.csar* (in diesem Fall das SimTech SWfMS) zu provisionieren.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<context>
    <resource>aws-ec2</resource>
    <application>opentosca</application>
    <servicePackageReference>http://.../simtech.csar</servicePackageReference>
</context>
```

Abbildung 3: Exemplarischer User Context

3. Die *setConfiguration()* Operation des Web Service Interface der Local Bootware wird aufgerufen um die Standardkonfiguration an diese zu übergeben. Diese Standartkonfiguration könnte wie in Abbildung 4 aussehen, die die Zugangsdaten für die Amazon Cloud enthält.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configurationListWrapper>
    <configurationList>
        <entry>
            <key>aws-ec2</key>
            <value>
                <configuration>
                    <entry>
                        <key>secretKey</key>
                        <value>aU7cD26+grgtluyqk0/YeMaRag5tmOZIUcvN+2Us</value>
                    </entry>
                    <entry>
                        <key>accessKey</key>
                        <value>AKIAJYIAG2HYG23MDSZQ</value>
                    </entry>
                    <entry>
                        <key>region</key>
                        <value>ec2.eu-west-1.amazonaws.com</value>
                    </entry>
                </configuration>
            </value>
        </entry>
    </configurationList>
</configurationListWrapper>
```

Abbildung 4: Exemplarische Standardkonfiguration

4. Die *deploy()* Operation des Web Service Interface des Local Bootware wird mit dem User Context aufgerufen um das SimTech SWfMS zu provisionieren.

5. Es wird auf die Antwort des *deploy()* Aufrufs gewartet.

5.3 Die Local Bootware erhält den *deploy()* Aufruf

Die Local Bootware hat nun vom Bootware Eclipse Plugin den Auftrag erhalten, das SimTech SWfMS in einer bestimmen Cloud Umgebung (hier AWS) mit einer bestimmten Provisioning Engine (hier OpenTOSCA) zu provisionieren. Wir befinden uns jetzt also in *org.simtech.bootware.local.LocalBootwareImpl.deploy()*. Sie wird dies allerdings nicht direkt tun, sondern den Aufruf an die Remote Bootware weiterleiten. Dabei passiert folgendes:

1. Die Remote Bootware existiert noch nicht, also wird diese durch die Local Bootware deployed.
    1. Die Local Bootware erstellt dafür einen User Context und ruft damit ihre eigene *deploy()* Operation auf.
    2. Der User Context wird an das Bootware Repository gesendet, um zu erfahren, welche Plugins für diesen *deploy()* Aufruf nötig sind. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.readContext())*
    3. Die nötigen Plugins werden aus dem Bootware Repository geladen falls sie lokal noch nicht existieren. Dann werden sie in die Local Bootware geladen und initialisiert. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.loadRequestPlugins())*
    4. Die *provision()* Operation des Resource Plugins wird aufgerufen, um die Ressource (hier eine AWS EC2 Instanz) zu provisionieren. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.provisionResource())*
    5. Die connect() *Operation* des Communication Plugins wird aufgerufen, um eine Verbindung zur Ressource herzustellen (hier SSH). (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.connect())*
    6. Die *provision()* Operation des Application Plugins wird aufgerufen, um die Applikation (hier die Remote Bootware) auf der Ressource zu installieren. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.provisionApplication())*
    7. Die *start()* Operation des Application Plugins wird aufgerufen, um die Remote Bootware zu starten. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.startApplication())*
    8. Die Plugins werden wieder entladen. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.unloadRequestPlugins())*
2. Die Remote Bootware existiert nun.
3. Die Standardkonfiguration wird an diese weiter geleitet.
4. Der ursprüngliche *deploy()*Aufruf wird an diese weiter geleitet.
5. Es wird auf die Antwort des *deploy()* Aufrufs gewartet.

5.4 Die Remote Bootware erhält den *deploy()* Aufruf

Die Remote Bootware hat nun von der Local Bootware den *depoy()* Aufruf erhalten. Wir befinden

uns jetzt in *org.simtech.bootware.remote.RemoteBootwareImpl.deploy()*. Hier passiert folgendes:

Die Schritte 1.2 – 1.7 des vorherigen Unterkapitels werden von der Remote Bootware ausgeführt, mit dem Unterschied, dass jetzt statt der Remote Bootware die Provisioning Engine (OpenTOSCA) deployed wird. Nach Schritt 1.7 existiert nun also eine neue EC2 Instanz auf der OpenTOSCA installiert ist. Nun kommt ein zusätzlicher Schritt hinzu:

> … (siehe oben)
>
> 8. Die *provision()* Operation des Provision Plugin wird aufgerufen. Diese übergibt die Service Package Reference aus dem User Context an die PE (OpenTosca) um das Service Package (hier das SimTech SWfMS) mit der PE zu provisionieren. Nach diesem Schritt steht also das SimTech SWfMS in der Amazon Cloud zur Verfügung. (*org.simtech.bootware.remote.RemoteBootwareImpl.Machine.provisionMiddleware()*)
>
> 9. Die Plugins werden wieder entladen. (*org.simtech.bootware.core.AbstractStateMachine.AbstractMachine.unloadRequestPlugins())*

2. Die Antwort mit Informationen über das SimTech SWfMS wird an die Local Bootware zurück gegeben. Diese leitet die Antwort an das Bootware Eclipse Plugin weiter.

5.5 Das Bootware Eclipse Plugin erhält die Antwort des *deploy()* Aufrufs

Das Bootware Eclipse Plugin hat nun die Antwort des *deploy()* Aufrufs erhalten, den sie zuvor an die Local Bootware geschickt hatte. Nun passiert folgendes:

1. Mit den Informationen über das SimTech SWfMS aus der Antwort werden die Eclipse Einstellungen für das SimTech System angepasst. (*org.simtech.bootware.eclipse.BootwarePlugin.setSimTechPreferences()*)
2. Der Shutdown-Trigger wird gestartet. Dieser hört die Events der Workflow Engine des Simtech SWfMS ab und erfährt so, wenn keine Prozessmodelle mehr deployed sind. Tritt dieser Fall ein, fragt er den Benutzer, ob der Shutdown-Prozess der Bootware gestartet werden soll. Dieser Prozess würde dann das SimTech SWfMS, alle PEs, die Remote Bootware, die Local Bootware und alle dazugehörenden Ressourcen wieder entfernen. (*org.simtech.bootware.eclipse.BootwarePlugin.initializeShutdownTrigger()*)

Damit ist der Bootstrappingprozess beendet und die Ausführung geht weiter im modifizierte Eclipse BPEL UI Plugin.

5.6 Das Prozessmodell wird deployed und gestartet

Das modifizierte Eclipse BPEL UI Plugin hat gewartet bis der Bootstrappingprozess abgeschlossen ist. Nun kann es das Prozessmodell auf dem gerade provisionierten SimTech SWfMS deployen und starten.

6 Architektur

Im folgenden wird die Architektur des Bootware Systems grob beschrieben. Eine Ausführliche Be-

schreibung kann in der *Diplomarbeit 3616 - Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments* nachgelesen werden.

6.1 Integration in das SimTech System

Abbildung 5 zeigt die Integration des Bootware Systems in das bestehende SimTech System. Die Komponenten des Bootware Systems sind dabei schwarz eingefärbt. Zu sehen sind das Bootware Plugin, das (mit Hilfe des hier nicht gezeigten modifizierten Eclipse BPEL UI Plugins) die Local Bootware und den SimTech Modeler verbindet. Die Local Bootware ist dafür zuständig die Remote Bootware in der passenden Cloud-Umgebung zur Verfügung zu stellen. Die Remote Bootware wiederum provisioniert Provisioning Egnines und mit Hilfe dieser auch die Workflow Middleware (hier das SimTech SWfMS). Sowohl die Local, als auch die Remote Bootware, kommunizieren mit dem Bootware Repository, wenn diese Plugins benötigen oder einen User Context auf einen Request Context abbilden müssen.

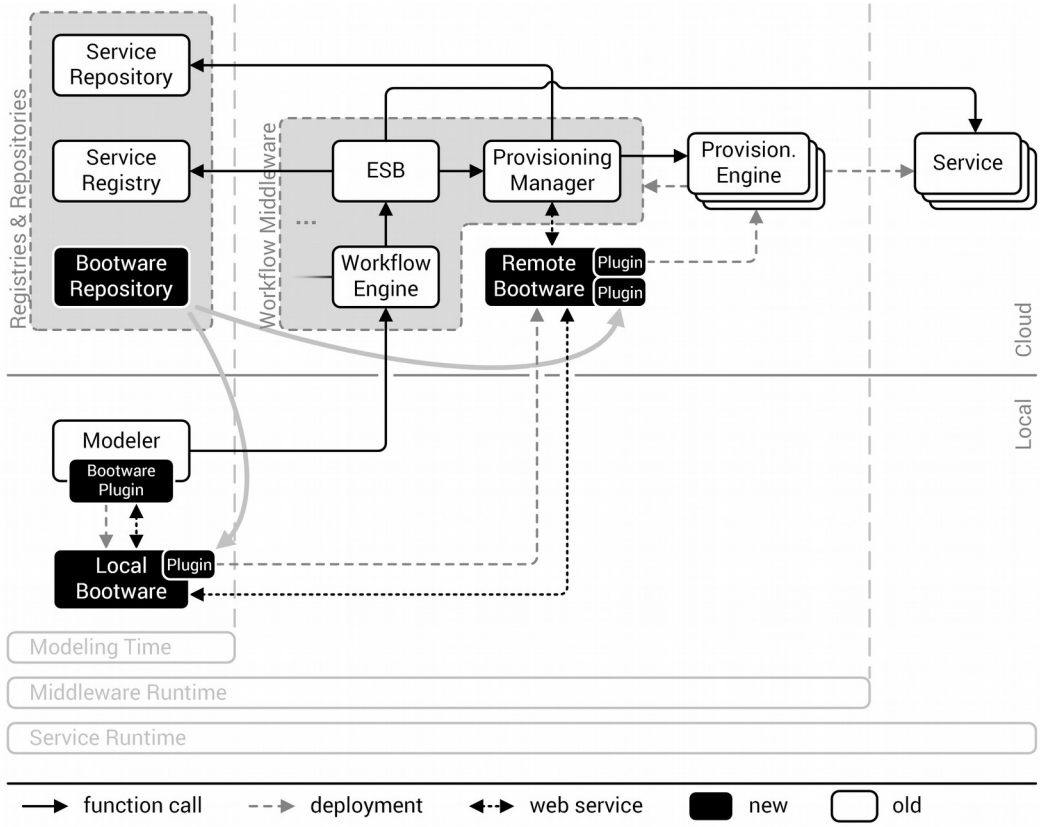

Abbildung 5: Die Integration des Bootware Systems in das bestehende SimTech System

6.2 Aufbau der Local und Remote Bootware Komponenten

Die Local und Remote Bootware Komponenten sind sehr ähnlich aufgebaut, da ihre Funktionalität in vielen Bereichen identisch ist. Abbildung 6 zeigt die Bestandteile der Local und Remote Bootware. Bei beiden handelt es sich um Web Services, die diverse Operationen zur Verfügung stellen. Diese Funktionen werden größtenteils durch Plugins realisiert. Deren Ausführungsreihenfolge wird wiederum durch eine State Machine implementiert. Während der Ausführung werden auch Events

generiert, die durch den Event Bus via PubSub verteilt werden. Geladen werden die Plugins durch den Plugin Manager. Event Plugins (unten) werden nur einmal beim Start der Bootware geladen, wohingegen die restlichen Plugins (oben) je nach Bedarf beim Bearbeiten eines Requests geladen (und danach wieder entladen) werden. Anzumerken ist hier, das die Provision Workflow Middleware Plugins (kurz Provision Plugins. Oben rechts) nur in der Remote Bootware vorkommen.

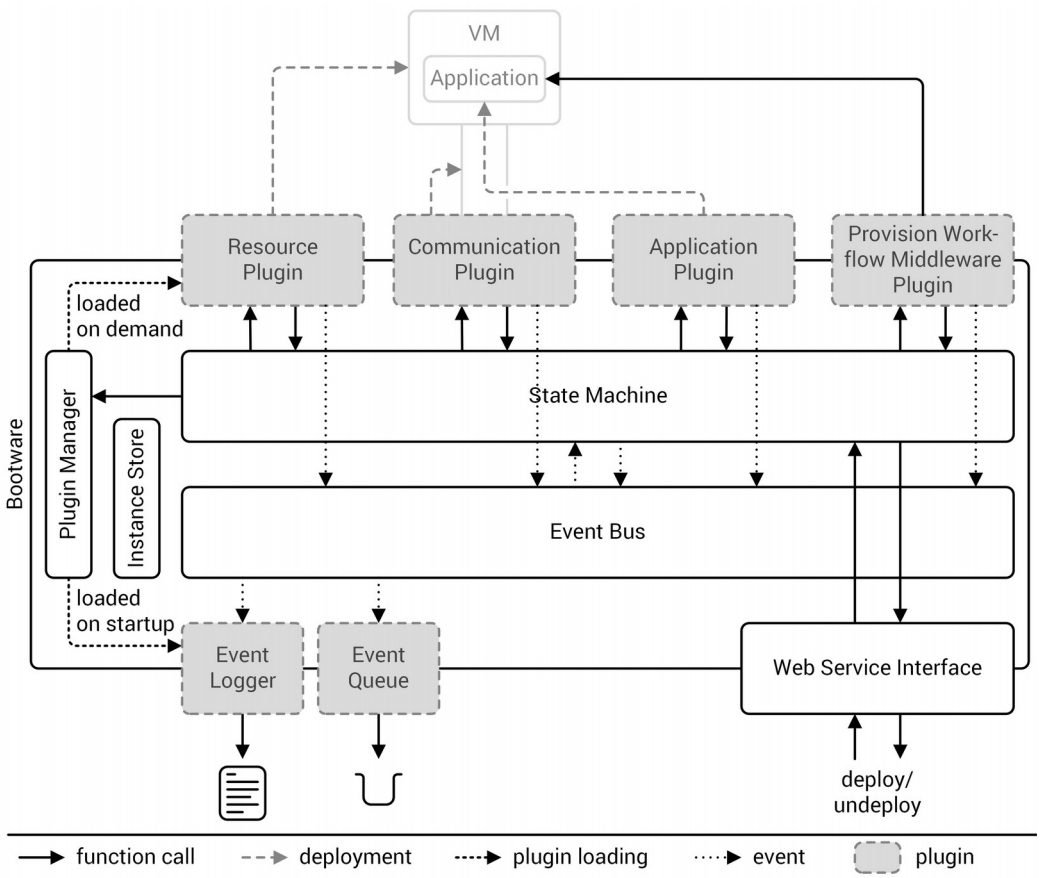

Abbildung 6: Architektur der Local und Remote Bootware Komponenten

Die Local und Remote Bootware bieten ihre Web Services über eingebettete Server an. Die Local Bootware ist nachdem sie gestartet wurde unter http://localhost:6007/axis2/services/Bootware zu erreichen, die Remote Bootware unter http://[RESOURCE_IP]:8080/axis2/services/Bootware.

Die Operationen, die über das Web Service Interface angeboten werden werden im Folgenden aufgelistet. Anmerkung: Die Local Bootware leitet diese Aufrufe immer an die Remote Bootware weiter.

- **IsReady():** Gibt an, ob die Bootware bereit ist Anfragen zu berarbeiten.
- **deploy(UserContext):** Führt den Deploy-Prozess der jeweiligen Bootware aus um die im UserContext angegebene Applikation (z.B. OpenTOSCA) auf der angegeben Ressource (z.B. AWS EC2) zu installieren und gegeben falls das Service Package zu provisionieren.
- **undeploy(UserContext):** Macht den Deploy-Prozess rückgängig, d.h. deprovisioniert die Applikationen und Ressourcen.
- **setConfiguration(ConfigurationListWrapper):** Legt die mitgeschickte Konfiguration als

Standardkonfiguration fest, die bei späteren *deploy()* und *undeploy()* Aufrufen immer als Grundlage verwendet wird. Anmerkung: Konfiguration aus einem Request Context und dem User Context überschreiben die Standardkonfiguration für den jeweiligen Aufruf.

- **shutdown():** Fährt die Bootware herunter. Dabei werden auch alle aktiven Applikationen und Ressourcen deprovisioniert.
- **getActive(UserContext):** Gibt Informationen über die angeforderte Ressourcen/Applikation Kombination zurück, falls diese aktiv ist. Anmerkung: Nur von der Remote Bootware unterstützt.

7 Repository, Plugins und Mappings

Ein Großteil der Funktionalität der Bootware wird durch Plugins realisiert. Wie bereits in Kapitel 3 beschrieben gibt es dafür verschiedene Plugintypen, die in verschiedenen Kombinationen benutzt werden. Der Benutzer gibt diese Kombinationen allerdings nicht direkt an, sondern beschreibt im User Context der einem *deploy()* Aufruf mitgegeben wird nur, welches Service Package mit welcher Provisioning Engine in welcher Cloud provisioniert werden soll (siehe Abbildung 7).

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <context>
3      <resource>aws-ec2</resource>
4      <application>opentosca</application>
5      <servicePackageReference>http://.../simtech.csar</servicePackageReference>
6  </context>
```

Abbildung 7: Exemplarischer User Context

Welche Kombination von Plugins diesen Aufruf umsetzen kann muss die Bootware erst herausfinden, indem sie eine Anfrage an das Bootware Repository stellt. Dazu schickt sie einen POST Request mit dem User Context als Payload an folgende URL:

http://[REPOSITORY_SERVER_DOMAIN]:8888/repository/mapContext

(Wo die Bootware das Repository findet wird über *properties* Dateien konfiguriert. Siehe Benutzerhandbuch)

Das (sehr einfache) Mock Repository ließt aus dem User Context die *resource* und *application* Werte und baut daraus einen Pfad zu einem potentiellen Request Context. Allgemein sieht dieser Pfad so aus:

mappings/[application]/[resource]/context.xml

Für den User Context aus Abbildung 7 dann entsprechend:

mappings/opentosca/aws-ec2/context.xml

Existiert diese Datei, so liefert das Repository diesen Request Context zurück, der die spezifischen Plugins und andere Konfigurationsparameter enthält (siehe Abbildung 8).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<context>
  <resourcePlugin>plugins/resource/aws-ec2-1.0.0.jar</resourcePlugin>
  <communicationPlugin>plugins/communication/ssh-1.0.0.jar</communicationPlugin>
  <applicationPlugin>plugins/application/opentosca-ec2-1.0.0.jar</applicationPlugin>
  <callApplicationPlugin>plugins/provision/opentosca-1.0.0.jar</callApplicationPlugin>
  <servicePackageReference></servicePackageReference>
  <configurationList>
    <entry>
      <key>aws-ec2</key>
      <value>
        <configuration>
          <entry>
            <key>username</key>
            <value>ubuntu</value>
          </entry>
          <entry>
            <key>ports</key>
            <value>22,1337,8080,9443,9763</value>
          </entry>
          <entry>
            <key>imageId</key>
            <value>ami-88f129ff</value>
          </entry>
          <entry>
            <key>instanceType</key>
            <value>m3.medium</value>
          </entry>
        </configuration>
      </value>
    </entry>
  </configurationList>
</context>
```

Abbildung 8: Exemplarische Antwort des Bootware Repository für mapContext

Durch diesen Request Context weiß die Bootware nun, welche Plugins sie benötigt. Falls sie diese noch nicht heruntergeladen hat wird sie dies ebenfalls über das Repository tun. Dazu schickt sie ein GET Request an folgende URL:

http://[REPOSITORY_SERVER_DOMAIN]:8888/repository/plugins/[pluginType]/[pluginName]

Für das in Abbildung 8 gezeigt Resource Plugin *aws-ec2-1.0.0.jar* würde die URL so aussehen:

http://[REPOSITORY_SERVER_DOMAIN]:8888/repository/plugins/resource/aws-ec2-1.0.0.jar

Wichtig: *PluginName* sollte immer der vollständige Name des jar Archivs sein, also inklusive Versionsnummer und Dateiendung. Das Bootware Repository liefert auf diese Anfrage dann das entsprechende Plugin zurück (falls es existiert, wie hier z.B. unter *plugins/resource/aws-ec2-1.0.0.jar*).

7.1 Neue Plugins Hinzufügen

Sollen nun neue Plugins für die Bootware zugänglich gemacht werden, so ist wie folgt vorzugehen:

1. Plugin programmieren und kompilieren

2. Plugin im Repository im *plugins* Ordner im entsprechenden Unterordner ablegen
3. Ein bereits vorhandene Request Context modifizieren oder einen neuen erstellen, der auf das neue Plugin verweist

Wichtig: Wenn bereits vorhandene Plugins modifiziert werden dann sollte entweder jedes mal die Versionsnummer erhöht werden (dann natürlich auch in den jeweilige Request Contexts im Repository), oder man muss die alten Plugins aus dem lokalen Pluginordner der Local Bootware löschen, damit die neue Version aus dem Repository geladen wird. Bei der Remote Bootware ist dies nur nötig, falls sie zwischen den Änderungen an Plugins nicht neu deployed wurde.

7.2 Plugin Dependencies

```
<dependencies>

    <dependency>
        <groupId>commons-configuration</groupId>
        <artifactId>commons-configuration</artifactId>
        <version>1.10</version>
    </dependency>

    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk</artifactId>
        <version>1.7.5</version>
    </dependency>

    <dependency>
        <groupId>javax.mail</groupId>
        <artifactId>mail</artifactId>
        <version>1.4.7</version>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>

</dependencies>
```

Abbildung 9: Aws-ec-1.0.0.jar Resource Plugin Dependencies

Wenn ein Plugin Abhängigkeiten zu Bibliotheken hat, die nicht durch die JVM (java, javax, …) oder durch die Bootware selbst (org.simtech.bootware.core, net.engio.mbassy, …) bereitgestellt werden, so müssen diese in die Plugin .jar integriert werden. Dabei ist zu beachten, dass dies auch für transitive Abhängigkeiten gilt (also die Abhängigkeiten von Bibliotheken). Dies wird in der pom.xml

des jeweiligen Plugins gemacht. Ein gutes Beispiel ist das *aws-ec2-1.0.0.jar* Resource Plugin.

```
<!-- run mvn dependency:tree -->
<!-- list all artifacts that should be embedded
here -->
<!-- (the jars will be embedded in the lib folder)
-->
<Embed-Dependency>
commons-configuration,
commons-lang,
aws-java-sdk,
commons-logging,
httpclient,
httpcore,
commons-codec,
jackson-core,
jackson-databind,
jackson-annotations,
joda-time,
mail,
javax.servlet-api
</Embed-Dependency>
<!-- list all packages that are not embedded here
-->
<!-- (these have to be exported by the bootware
core) -->
<Import-Package>
javax.*,
org.xml.sax.*,
org.w3c.dom.*,
net.engio.mbassy.*,
org.simtech.bootware.core.*,
!*
</Import-Package>
<Embed-Directory>lib</Embed-Directory>
```

Abbildung 10: Eingebettete Abhängigkeiten

Diese Plugin benutzt unter anderem das AWS-Java-SDK, was in der *pom.xml* deshalb auch unter *<dependencies>* angegeben ist, wie in Abbildung 9 zu sehen ist. Dies reicht allerdings nicht aus. Zusätzlich müssen in der *pom.xml* in der Konfiguration des Maven Bundle Plugins unter *<Embed-Dependencies>* die Artefaktnamen angegeben werden, die später in das Pluginarchiv integriert werden sollen. Für das AWS-Java-SDK ist das sowohl *aws-java-sdk*, als auch dessen transitiven-Abhängigkeiten, wie in Abbildung 10 gezeigt (*httpclient, httpcore, commond-codec, jackson-core, …)*. Diese Abhängigkeiten lassen sich entweder übre die Maven Repository Seite herausfinden, oder, indem man (nach hinzufügen der Abhängigkeit unter *<dependencies>* in der *pom.xml*) im

Pluginordner den Befehl *„mvn dependency:tree“* ausführt. Dieser listet alle Abhängigkeiten auf.

Wenn das Ausführen eines Plugin mit einer OSGi-Fehlermeldung fehlschlägt (Bundle wiring etc.) dann wird das wahrscheinlich daran liegen, das in der *pom.xml* nicht die richtigen Abhängigkeiten eingetragen wurden. Hier ist es auch immer hilfreich zu überprüfen, ob sich nach der Kompilierung eines Plugins alle nötigen Bibiliotheken im *lib* Ordner des Pluginarchivs befinden und ob im *MANIFEST.MF* (das durch Maven generiert wird) die richtigen Einträge vorhanden sind.

7.3 Plugin Ressourcen

Externe Dateien, die von Plugins für irgend eine Aufgabe benötigt werden, können dem Plugin auf zwei Arten zur Verfügung gestellt werden.

1. Sie werden auf einem externen Server gehostet und das Plugin lädt sich diese von dort herunter.
2. Die Datei wird in das Pluginarchiv integriert.

Um eine Datei in ein Pluginarchiv zu integrieren, muss diese in dem Ordner *src/main/resource* des jeweiligen Plugins abgelegt werden. Maven wird die Datei dann beim kompilieren mit in das .jar Archiv packen. Dies passiert inklusive der Ordnerstruktur unterhalb des *resource* Ordners.

Auf diese integrierten Ressourcen kann dann im Plugincode wie folgt zugegriffen werden:

final InputStream is = PluginKlasse.class.getResourceAsStream(“/unterordner/datei.txt“);

Zu Beachten:

- *PluginKlasse* ist die Klasse des jeweiligen Plugins
- *„/unterordner/date.txt“* ist der Pfad zur Datei innerhalb des .jar Archivs. Man beachte das „/“ am Anfang. In diesem Fall wurde die Datei also ursprünglich unter *src/main/resource/unterordner.datei.txt“* abgelegt.

8 Bekannte Probleme, Future Work

Im Folgenden werden einige bekannte Probleme, Verbesserungsmöglichkeiten und Potential für Future Work beschrieben. Siehe hierzu auch das Kapitel Future Work in der Diplomarbeit.

8.1 OpenTOSCA-SimTech Provision Plugin

Das OpenTOSCA-SimTech Provision Plugin, das OpenTOSCA aufruft um die SimTech.csar zu provisionieren, führt im Moment noch einige manuelle Konfigurationsschritte nach der Provisionierung durch OpenTOSCA aus. Diese beinhalten unter anderem das Setzen von einigen URLs in *.property* Dateien, das neu starten von Service Mix, sowie das hochladen des Provisioning Context.

In Zukunft sollten diese Aufgaben ebenfalls durch OpenTOSCA ausgeführt werden, entweder während der Ausführung des Buildplans oder danach durch einen zusätzlichen Managementplan. Die hierfür nötigen Parameter (URLs, Provisioning Context) müssten dann beim Aufruf des jeweiligen

Plans übergeben werden. Dann können die Klassen *SetupProvisioningManager.java* sowie *SshConnection.java* (Dependencies in der pom.xml nicht vergessen!) wieder aus dem Plugin entfernt werden.

8.2 Fragmento Initialisierung

Nach der Provisionierung des SimTech SWfMS durch die Bootware werden einige Optionen im Modeler aktualisiert, die ActiveMQ Verbindung neu gestartet, sowie Fragmente aus dem Fragmento Repository geladen (siehe *SimTechPreferences.java* im Eclipse Plugin Ordner). Das Update der Fragmente ist im Moment etwas fehleranfällig. Falls das Fragmento Fenster im Modeler nicht sichtbar ist, schlägt *updateFragmentoSettings()* fehl und der Bootwareprozess wird nicht richtig beendet. Dies sollte gefixt werden.

Zusätzlich wird das Fragmento Fenster nicht richtig aktualisiert, nachdem die Fragmento geladen wurden. Wie bei der manuellen Aktualisierung/Befüllung über das Fragmento Zahnrad-Symbol sollte nach dem herunterladen der Fragmente die Symbolleiste im Fragmentofenster erscheinen, was aber im Moment nicht der Fall ist. Durch eine manuelle Aktualisierung (nach der automatischen) lässt sich dies beheben.

8.3 Shutdown Trigger

Der Bootware Shutdown Trigger zählt jedes mal wenn eine Workflowinstanz gestartet wird einen Zähler hoch. Wenn eine Workflowinstanz terminiert zählt er den Zähler wieder runter. Erreicht der Zähler wieder 0, wird gefragt, ob der Shutdownprozess ausgelöst werden soll.

Wenn Workflowinstanzen nicht richtig terminiert werden, oder wenn Workflowmodelle undeployed werden, ohne vorher deren Instanzen zu terminieren, dann kann es vorkommen, das dieser Zähler nie mehr 0 erreicht und somit der Shutdownprozess nie ausgelöst wird. Das könnte verbessert werden.