JAVA Mag
Java magazin™
Java | Architektur | Software-Innovation

JAX Programminfos ab Seite 51!

JAVA und die BLOCKCHAIN

Was ist die Blockchain? Grundlagen verständlich erklärt

web3j Integration von Java mit Ethereum

Corda Open-Source-Blockchain für Unternehmen

www.javamagazin.de
Ausgabe 4.2019

SAP S/4HANA und Java-Individualentwicklungen nahtlos integrieren

Auf zu mehr Hybridität!

„Individuell“ und „standardisiert“ zugleich – das scheint ein Widerspruch zu sein. Doch mit dem richtigen Ansatz lassen sich Standard- mit individuellen Lösungen fast nahtlos zu einer gemeinsamen Lösung zusammenführen. Wir zeigen hier aus technischer Sicht, wie sich mit Java auf der SAP-Plattform eine hybride Architektur umsetzen lässt, die ohne zusätzliche Abhängigkeiten von der Standardsoftware auskommt.

von Jakob Boos und Manjinder Singh

Fast jeder IT-Verantwortliche kennt den folgenden Balanceakt: mit der Einführung einer ERP-Standardsoftware eine weitgehende Abdeckung von allen notwendigen Funktionen zu erzielen und gleichzeitig auch individuelle Anforderungen optimal zu berücksichtigen. Häufig stößt dieses Vorgehen an Grenzen. Schließlich gilt es, eine wirtschaftliche Pflege und Wartung sicherzustellen und die volle Releasefähigkeit der Standardsoftware beizubehalten. Daher sind Abstriche bei der Individualität oft nicht zu vermeiden, führt die Erfüllung spezieller Anforderungen in der Regel doch zu mehr Aufwand bei der Umstellung auf neue Releases. Das gilt insbesondere, wenn ein tiefer Eingriff in die Standardsoftware notwendig ist. Muss also stets eine Entscheidung zu Lasten entweder der Handhabbarkeit oder der Individualität getroffen werden? Nicht unbedingt, denn eine hybride Architektur bietet die Möglichkeit, beide – scheinbar widersprüchlichen – Anforderungen zu erfüllen.

Handlungsbedarf: SAP-R3-Support endet

Es ist bekannt, dass SAP den Support für bestehende R3-Lösungen ab 2025 einstellen wird. SAP-R3-Nutzer stehen damit aktuell vor der Notwendigkeit, ihre bestehenden Investitionen auf die neuen SAP-Lösungen wie SAP S/4HANA und weitere zu überführen. Das bietet aber auch eine Chance: Individuelle Erweiterungen, die sich nicht eins zu eins auf die neuen, in der Regel Cloud-basierten Lösungen überführen lassen, könnten nämlich zukünftig auch unabhängiger von SAP-Lösungen implementiert werden. Trotzdem ist es hierbei möglich, ein homogenes Look-and-feel auf der Benutzeroberfläche beizubehalten und gleichzeitig eine ausufernde Plattformvielfalt zu vermeiden.

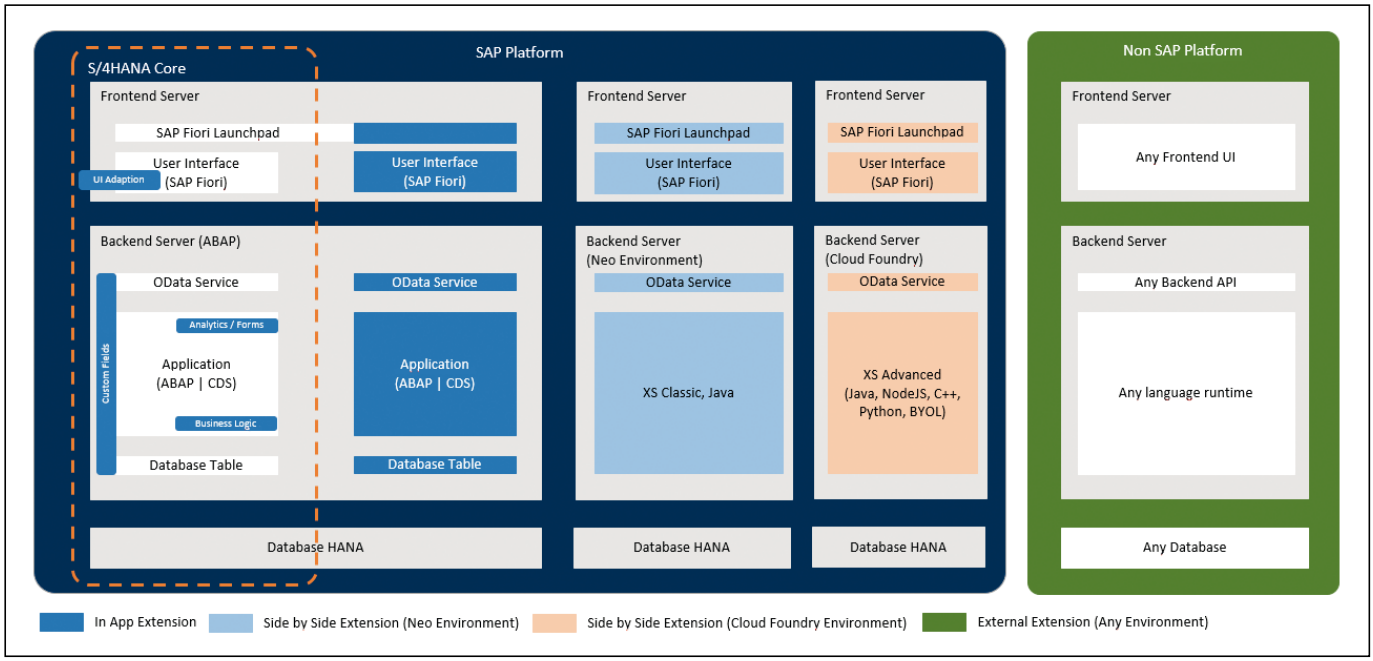

Abb. 1: Erweiterungsmöglichkeiten für SAP-Standardlösungen

So kombiniert die für eine hybride Architektur ausgelegte Capgemini-Architekturblaupause „MPSA – Multi Pillar S/4HANA Architecture“ SAP-S/4HANA-Standardkomponenten mit individuell mit Java entwickelten Lösungsbausteinen. Diese werden unter der homogenen Oberfläche von SAP UI5 oder Open UI5 integriert und auf der bestehenden SAP-Plattform betrieben. Der Ansatz baut auf State-of-the-Art-Entwicklungsparadigmen wie Java, Serviceorientierung/Microservices, loser Kopplung und Integrationsmustern auf. Das Vorgehen und die Architektur setzen auf der frei zugänglichen Open-Application-Standard-Plattform (OASP/DevonFw [1]) auf, bei der es sich um Architekturmuster, Bausteine und Werkzeuge zur Entwicklung von Java-basierten IT-Lösungen handelt.

Zu Demonstrationszwecken wurde eine konkrete Aufgabenstellung aus der öffentlichen Verwaltung als Proof of Concept umgesetzt. Es handelt sich hierbei um eine individuell entwickelte Haushaltsplanung, die neben dem unter S/4HANA betriebenen Haushaltsvollzug nebst Rechnungslegung betrieben wird. An dieser Lösung lassen sich exemplarisch die Vorgehensweise sowie die Vorteile der hybriden Architektur im SAP-Umfeld aufzeigen.

Handlungsoptionen für Architekten

Die auf einer In-Memory-Datenbank aufbauende SAP-HANA-Plattform ermöglicht es, Standard und Individualität ohne die genannten negativen Nebenwirkungen miteinander zu verbinden. Die hierfür entwickelte hybride IT-Lösung setzt sich aus lose gekoppelten Standard- und individuellen Bausteinen zusammen. Dieser Ansatz bietet mehrere Vorteile:

- Die Standardsoftwarekomponenten von S/4HANA werden weitgehend genutzt.
- Individuelle Softwarekomponenten werden auf Basis einer einheitlichen Architekturblaupause entwickelt und mit praxiserprobten Werkzeugen aus dem Capgemini-Framework devonfw umgesetzt.
- Standard- und Individualbausteine lassen sich flexibel verknüpfen.
- Das SAP-Development-Toolkit SAP UI5 sorgt für eine homogene Nutzeroberfläche ohne Brüche im Look-and-Feel zwischen Standard- und individuellen Bausteinen.
- SAP HANA bietet eine einheitliche Betriebsplattform und ermöglicht den Zugriff auf einen gemeinsamen Datenbestand.

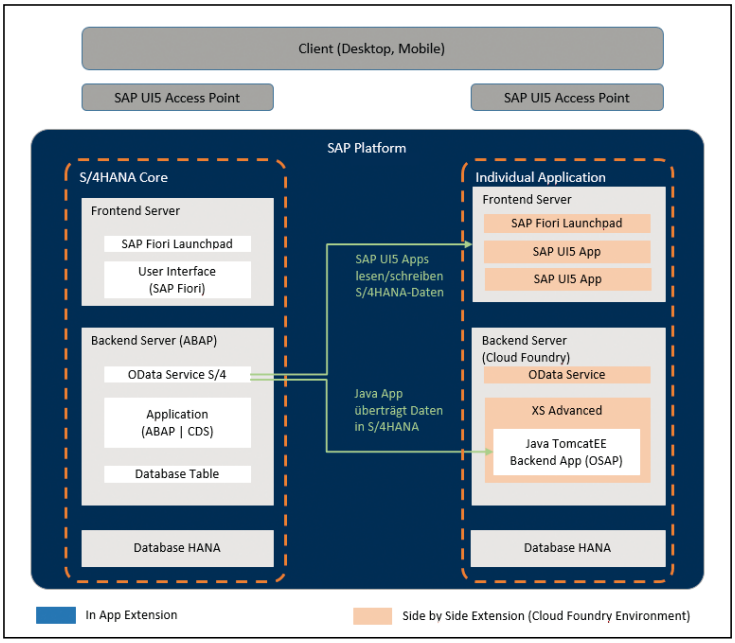

Abb. 2: Architekturblaupause Individualentwicklung Java auf der SAP-HANA-Plattform

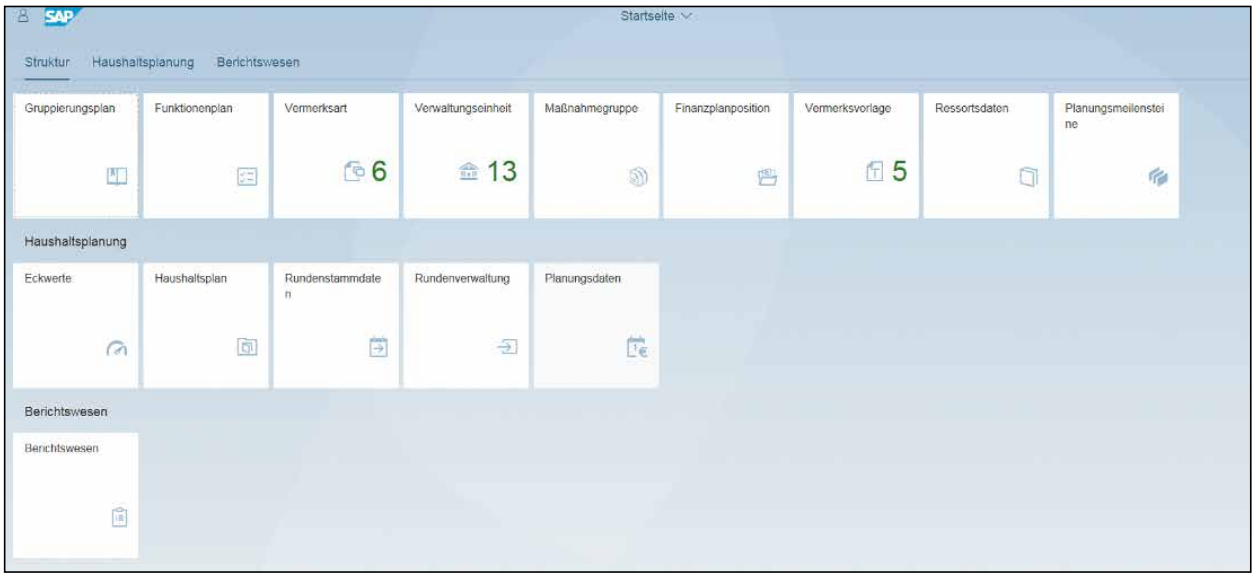

Abb. 3: Für den Benutzer kein Unterschied

Der Ansatz erlaubt grundsätzlich den Einsatz von verschiedenen Programmiersprachen und Werkzeugen für die Individualbausteine. **Abbildung 1** zeigt, welche Erweiterungsmöglichkeiten (Extensions) auf der SAP-Plattform bereitgestellt werden und welche Erweiterungsmöglichkeiten darüber hinaus verfügbar sind.

In-App-Extensions sind ABAP-basierte Modifikationen und Erweiterungen im S/4HANA Core, die auch bei Releaseupgrades laut SAP keine Probleme verursachen werden. Side by Side Extensions sind Erweiterungen, die in anderen, auch von der SAP-Plattform unterstützten Umgebungen implementiert werden. Diese Side by Side Extensions nutzen als Benutzeroberfläche ebenfalls SAP UI5/Fiori, sofern sie eine Oberfläche haben müssen, und HANA als Datenbank, sofern sie eine Datenbank benötigen. SAP bietet hier im Backend zwei verschiedene Umgebungen an: Neo und Cloud Foundry. Unabhängig davon können Erweiterungen auch vollständig außerhalb der SAP-Plattform implementiert werden (External Extension), was dann dazu führt, dass aus Betriebssicht neben der SAP-Plattform mindestens noch eine weitere Plattform betrieben werden muss.

Java-basierte SAP-Erweiterungen

Die Architekturblaupause „MPSA – Multi Pillar S/4HANA Architecture“ für eine hybride Architektur führt zu einem harmonischen Miteinander von SAP-S/4HANA-Lösungselementen und individuell entwickelten Lösungselementen. Das hier vorgestellte Beispiel einer Haushaltsplanung für die öffentliche Verwaltung wird als individuelle Applikation in der Cloud-Foundry-Umgebung implementiert (**Abb. 2**).

Beide Komponenten (Individual und Standard) werden auf der bestehenden SAP-Infrastruktur-Umgebung der SAP-HANA-Plattform integriert betrieben. Das sorgt wiederum dafür, dass eine unnötige Vielfalt unterschiedlicher Plattformen im IT-Betrieb vermieden und damit die Effizienz gesteigert wird.

Mit einem Prototyp wurde die Erreichbarkeit der gesteckten Ziele validiert. Dazu zählen insbesondere die homogene Oberfläche auf Basis von SAP UI5 und die Nutzung moderner Entwicklungsparadigmen wie beispielsweise Java, lose Kopplung und Integration. Der Prototyp präsentiert sich dem Benutzer als Lösung aus einem Guss: Die Bedienung und die Oberfläche für die Haushaltsplanung und den Haushaltsvollzug sind homogen, während in der Architektur Standard- und Individualkomponenten nebeneinanderstehen und unabhängig voneinander gewartet werden können (**Abb. 3**). Die S/4HANA-Lösungselemente basieren auf einem Backend-Server (ABAP), der auf die Funktionalitäten der SAP-HANA-Plattform zugreift. Analog dazu werden die individuell entwickelten Lösungselemente durch eine Java-Applikation bereitgestellt. Als Betriebsplattform für die Java-Applikation dient ein Applikationsserver XSA der HANA-Plattform. Die Java-Applikation ist wiederum als lauffähige Spring-Boot-Applikation mit einem Embedded Tomcat realisiert. Die Integration von Geschäftsprozessen über beide Systeme kann bidirektional durch ODATA als REST-Protokoll umgesetzt werden.

Im Folgenden werden wir uns auf einen Teilaspekt der Haushaltsplanung konzentrieren, um eine Java-basierte SAP-Erweiterung beispielhaft vorzustellen. Die Haushaltsplanung in der öffentlichen Verwaltung zählt zu den Kernkompetenzen der Finanzministerien, die mit Eckwerteverfahren, Abstimmungsschleifen zwischen Ressorts, Anforderungen der erweiterten Kameralistik und weiteren Faktoren einen komplexen Geschäftsprozess abbildet. Solch individuelle Anforderungen sind auf den ersten Blick wenig für eine Standardlösung geeignet. Gleichzeitig sollen der Haushaltsvollzug und die Rechnungslegung so weit wie möglich über eine Standardlösung wie zum Beispiel S/4HANA realisiert werden. Zur Erfüllung der scheinbar widersprüchlichen Anforderungen ist eine hybride Umsetzung eine geeignete Variante, um Standard und Individualität sowohl aus Benutzer-

sicht als auch aus Prozesssicht zu vereinen. Der Use Case ist dabei nicht auf die Haushaltsplanung beschränkt. Auch viele Unternehmen stehen vor der Aufgabe, spezielle, vom Kunden gestellte Anforderungen individuell auf Basis von SAP zu entwickeln.

Beispiel: Erfassungsdialog für Verwaltungseinheit

Exemplarisch sollen hier an einer konkreten Beispielanwendung aus der Haushaltsplanung – „Verwaltungseinheit pflegen" – die einzelnen Entwicklungsschritte in Java für eine Java-basierte SAP-Erweiterung verdeutlicht werden. Basierend auf dem Java-Anwendungskern verfügt die Applikation über eine Benutzerschnittstelle, die mit SAP UI5 implementiert wird. SAP UI5 ist eine von SAP entwickelte JavaScript-Bibliothek, die auch als Open UI5 auf GitHub veröffentlicht ist [2]. Technisch gesehen läuft der Java-Anwendungskern auf einem Applikationsserver innerhalb der HANA-Plattform (XSA). In die lauffähige Spring-Boot-Applikation ist ein Embedded Tomcat als Webserver schon integriert. Der XSA-Applikationsserver verwendet für die Ausführung der Java-Applikation eine zertifizierte SAP Java Virtual Machine (JVM) und ein Java Development Kit (JDK), das auf dem Java-SE-8-Standard aufbaut. Technologisch basiert die SAP JVM auf OpenJDK [3]. Die Benutzeroberfläche wird auf einem Frontend-Server bereitgestellt. Der Benutzer bekommt einen webbasierten Zugang in Form eines Launchpads, über das er die Applikation „Verwaltungseinheit pflegen" startet (**Abb. 4**).

Im Verwaltungsbereich ist ein Haushalt typischerweise anhand verschiedener Ressorts geplant, die die obersten Planungseinheiten bilden. Diese Entität wird hier als Verwaltungseinheit bezeichnet. Der Erfassungsdialog für eine Verwaltungseinheit soll die normalen CRUD-Operationen (Create, Read, Update, Delete) unterstützen (**Abb. 5**).

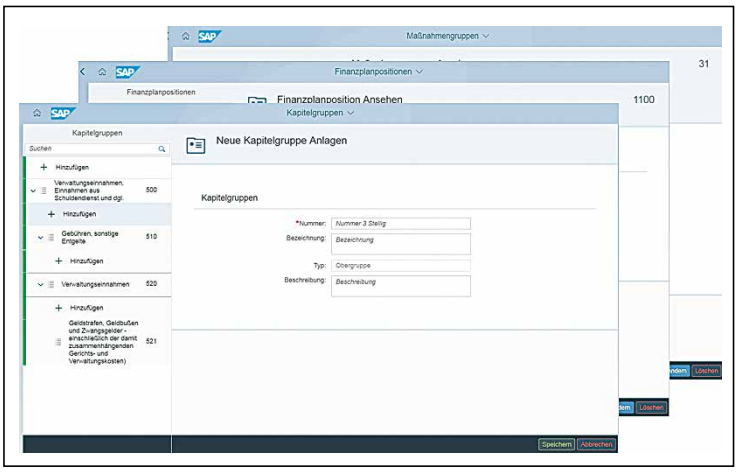

Abb. 4: Sowohl Standard- als auch Individualentwicklung integriert

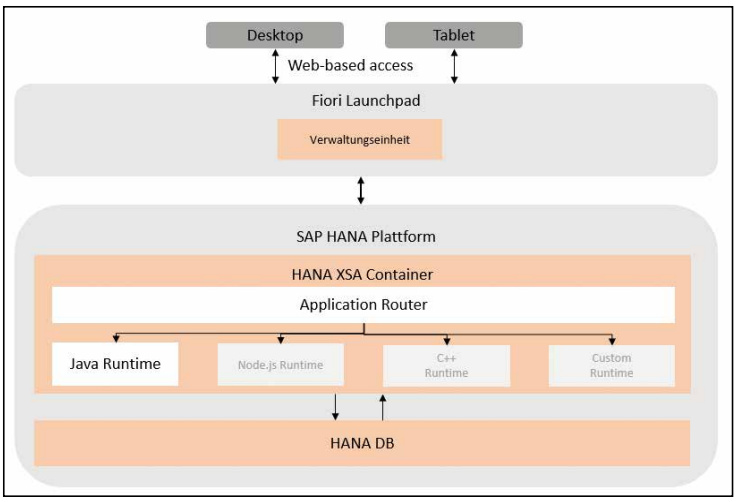

Abb. 5: Technische Infrastruktur für eine Java-Lösung auf Basis der SAP-HANA-Plattform

Application Server (XSA)

Mit der Einführung einer Cloud-ähnlichen On-Premise-Applikationsinfrastruktur versucht SAP die HANA-Plattform für SAP-Erweiterungen zu positionieren. Mit HANA XSA wurden die PaaS-Konzepte von Cloud Foundry auf die HANA-Plattform portiert (12-Factor-App [4]). Da die SAP-Cloud-Plattform zunehmend auch auf Cloud Foundry aufbaut, sind Applikationen, die innerhalb von XSA lauffähig sind, auch auf der SAP-Cloud-Plattform lauffähig und vice versa.

Im Kern stellt XSA einen Applikationsserver für verschiedene Laufzeitumgebungen dar. Aktuell werden neben Java und JavaScript (Node.js) auch C++ und Python unterstützt. In der Zukunft sollen gemäß des SAP-Slogans „Bring your own language" (BYOL) weitere Lautzeitumgebungen unterstützt werden. Die einzelnen XSA-Anwendungen laufen isoliert in eigenen Containern, technisch sogar in einem eigenen Prozess. Die XSA-Umgebung lässt sich von einem Benutzer über einen Client verwalten, beispielsweise Start und Stopp der Applikationen, Bereitstellung von Services und Skalierung der Applikationen. Obwohl eine enge Integration zwischen der HANA-Datenbank und dem HANA-XSA-Applikationsserver vorhanden ist, ist eine getrennte Installation auf unterschiedlichen Servern möglich. Zurzeit gibt es zudem einen getrennten Releasezyklus zwischen der HANA-Datenbank und XSA.

Unsere hier vorgestellte Beispielapplikation wurde als Java-Spring-Boot-Applikation auf dem XSA-Server ausgerollt. Für den Betrieb verwenden wir Spring Boot 2.0.4. Der Embedded Tomcat kommt in der Version 8.5.32 zum Einsatz.

Entwicklungsframeworks Devon, Spring, Olingo

Eine wichtige Komponente der Architekturblaupause ist die individuelle Umsetzung der Bausteine in Java. Hierfür verwenden wir das Open-Application-Standard-Platform-(OASP/devonfw-)Framework. Das Framework besteht aus einer Sammlung von Open-Source-Bibliotheken und Architekturpatterns in Form eines Architekturleitfadens. Dieser verdeutlicht, wie eine moderne Java-Applikation idealerweise aufgebaut werden sollte. Die Java-Applikation wird in Geschäftskomponenten aufgeteilt, die im Quellcode in Java-Paketen organisiert sind. Sie lassen sich in jeweils eine Service-,

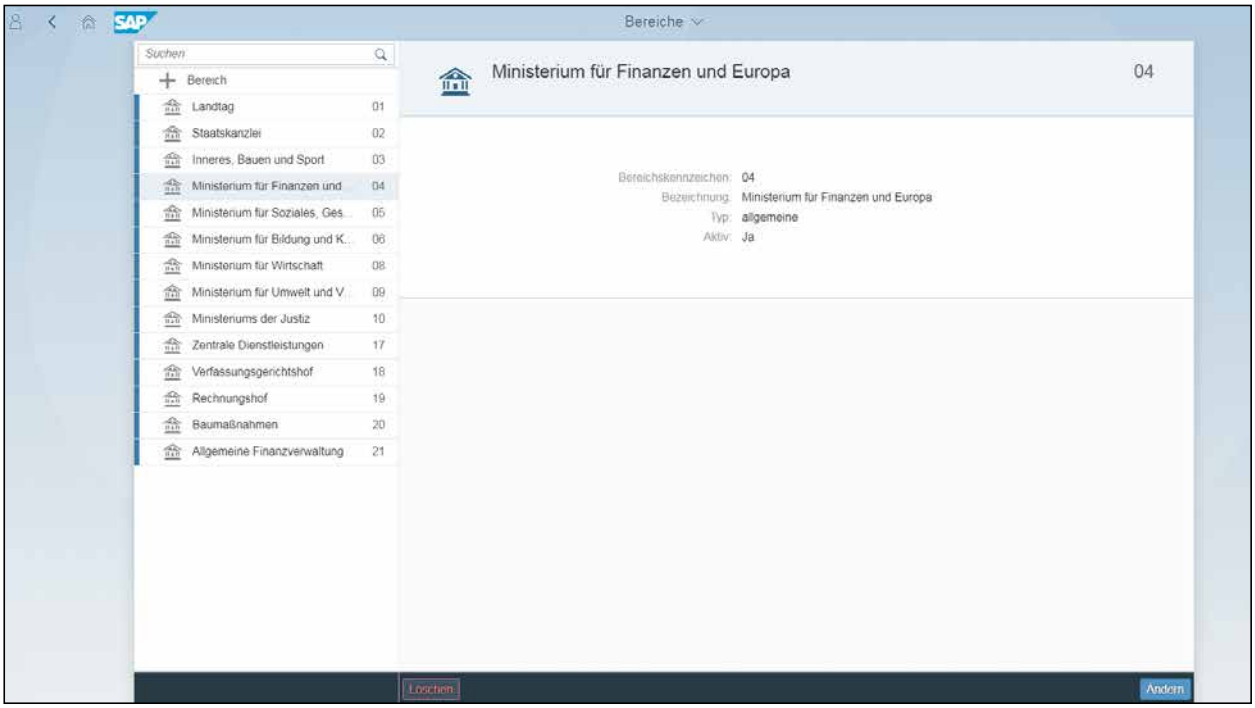

Abb. 6: Erfassungsdialog für Verwaltungseinheit

Logik- und Persistenzschicht unterteilen. In der Persistenzschicht wird gegen JPA mit Hilfe von Hibernate programmiert. Zusätzlich bietet devonfw Basiskomponenten an, um Themen wie Security, Logging und Transaction Handling abzudecken. Um am Ende einen OData-Service V2 [5] zu exponieren, wurde in Devon ein OData-Prozessor auf Basis von Apache Olingo [6] implementiert. Diesen benutzen wir, um die Entitäten in der Serviceschicht als REST-Schnittstelle für die Benutzeroberfläche zur Verfügung zu stellen (**Abb. 6**).

Das Framework OASP/devonfw baut überwiegend auf Spring [7] auf. Viele Konzepte, die aus der Spring-Welt bekannt sind – etwa Dependency Injection, DAOs oder Data Repositories – finden sich innerhalb von devonfw wieder. Um das devonfw-Framework als Java-Template zu benutzen, referenzieren wir auf das GitHub Repository mit allen zugehörigen Dokumentationen [8].

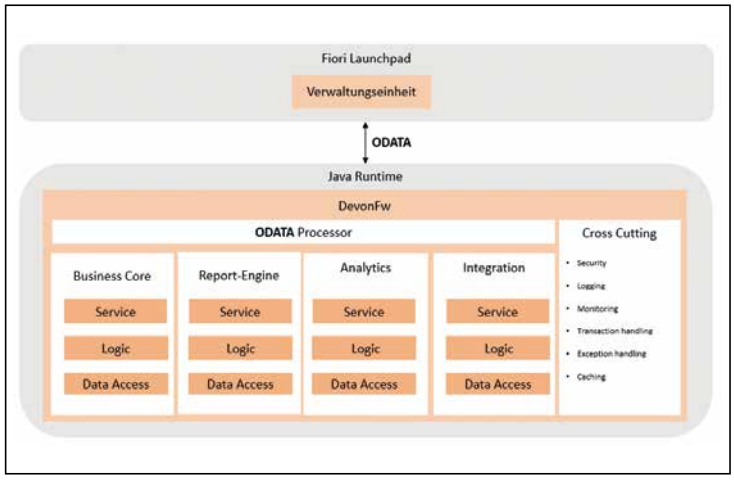

Abb. 7: Softwarearchitektur basierend auf devonfw

Beispiel: OData Service für Verwaltungseinheit

Anschließend definieren wir einen OData Service für die Entität „Verwaltungseinheit“. Wir fahren hier einen Bottom-up-Ansatz gemäß des Schichtenmodells in devonfw (**Abb. 7**). Die Entität „Verwaltungseinheit“ wird im Java-Quellcode mit dem Begriff „AdministrationUnit“ ins Englische übersetzt.

Persistenzschicht

In der Persistenzschicht wird gegen JPA programmiert. Als Persistenz-Framework gegen die HANA-Datenbank wird Hibernate verwendet. Innerhalb von devonfw kann die Datenbankkonfiguration (URL, User, Password) in der Spring-Konfigurationsdatei *application.properties* hinterlegt werden.

Um später über OData die Anwendungsdaten zu modifizieren, muss eine Java-JPA-Entität erstellt werden. Eine JPA-Entitätsklasse entspricht einer Tabelle in der Datenbank. Hibernate konvertiert die CRUD-Operationen automatisch über den JPA-Standard in HANA-SQL-Statements. In unserem Beispiel besteht die Verwaltungseinheit aus vier Attributen: *Identifier*, *KeyFigure*, *IsActive* und *IsDeleted* (Listing 1).

Zusätzlich zur JPA-Entität muss noch eine Wrapper-Klasse implementiert werden, um die CRUD-Operationen zu kapseln. Diese Isolation der Entitäten von den Businessoperationen wird über das Data-Access-Object-(DAO-)Pattern realisiert. Aufgrund der gleichen Mechanik können Data-Repository-Klassen in Spring verwendet werden, sodass herkömmliche Operationen nicht immer neu geschrieben werden müssen. Die abstrakte Klasse *JpaRepository* übernimmt die Implementierung für die CRUD-Operationen:

```
public interface AdministrationUnitEntityRepository
  extends JpaRepository<AdministrationUnitEntity, Long> {

}
```

Logikschicht

In der Logikschicht wird die Geschäftslogik implementiert, in diesem Fall einfache CRUD-Operationen. Gemäß einer devonfw-Richtlinie müssen zu jeder Entität ein API und die dazugehörige Implementierungsklasse erstellt werden. Da wir hier nur CRUD-Operationen über OData exponieren, können wir in devonfw auf die abstrakte Klasse *AbstractLogic* zurückgreifen. Der Aufruf über das Spring Data Repository erfolgt über die Vererbung der abstrakten Klasse automatisch. Sollten eigene Erweiterungen notwendig sein, können diese in der Implementierungsklasse überschrieben werden. Aufgrund der Isolation zwischen dem Datenbank- und dem Geschäftsmodell wird zusätzlich in der Logikschicht eine Konvertierung eines Transfer Objects (TO) in ein JPA-Objekt und umgekehrt realisiert. Wir bedienen uns hier des Dozer-Frameworks [9] (Listing 2).

```
public interface AdministrationUnitLogic
  extends LogicComponent<AdministrationUnitTo> {
}
```

Über die Spring-Annotation *@Component* kann das Objekt in der darüberliegenden Service-Schicht über Dependency Injection automatisch initialisiert werden.

Service-Schicht

Das Geschäftsmodell wird letztendlich in der Service-Schicht durch ein Transfer Object über OData zur Verfügung gestellt. In unserem Fall entspricht es der JPA-Entität (Listing 3).

Zusätzlich wird in der Service-Schicht analog zur Logikschicht eine API- und die zugehörige Implementierungsklasse erstellt. Das devonfw-OData-Modul bietet hierfür bereits die abstrakte Klasse *AbstractService* an, um einfache CRUD-Operationen zu verarbeiten (Listing 4).

Listing 1

```
@Getter
@Setter
@Entity
@Table(name = "\"hkrbudgetingdb.db.dbmodel::hkrbudgeting.
                                              AdministrationUnit\"")
public class AdministrationUnitEntity extends AbstractEntity {

  private static final long serialVersionUID = 1L;

  @Id
  private Long id;

  @Nationalized
  @Column(name = "\"Identifier\"", length = 256)
  private String identifier;

  @Column(name = "\"KeyFigure\"")
  private String keyFigure;

  @Column(name = "\"IsActive\"")
  private Boolean isActive;

  @ManyToOne
  @JoinColumn(name = "\"AssignedAdministrationUnitType_ID\"")
  private AdministrationUnitTypeEntity administrationUnitType;

  @Column(name = "\"IsDeleted\"")
  private Boolean isDeleted;
}
```

Listing 2

```
@Component
public class AdministrationUnitLogicImpl
  extends AbstractLogic<AdministrationUnitTo, AdministrationUnitEntity>
  implements AdministrationUnitLogic {

}
```

Listing 3

```
@Getter
@Setter
@EdmEntityType(name = "AdministrationUnit")
@EdmEntitySet(name = "AdministrationUnits")
public class AdministrationUnitTo extends AbstractTo{

  @EdmKey
  @EdmProperty
  private Long id;

  @EdmProperty
  private String identifier;

  @EdmProperty
  private String keyFigure;

  @EdmProperty
  private Boolean isActive;
}
```

Listing 4

```
@Service
public class AdministrationUnitServiceImpl
  extends AbstractService<AdministrationUnitTo>
  implements AdministrationUnitService {

}
```

Listing 5

```
@Service
@WebServlet(urlPatterns = {"/services/odata.svc/*"})
@ODataEntityScan(
 functionImportPackages = {
  "com.cg.hkrbudgeting.budgetingmanagement.service.api",
  "com.cg.hkrbudgeting.reportengine.service.api",
  "com.cg.hkrbudgeting.analytics.service.api",
  "com.cg.hkrbudgeting.integration.service.api"
 },
 entityPackages = {
  "com.cg.hkrbudgeting.budgetingmanagement.common.api.to",
   "com.cg.hkrbudgeting.reportengine.common.api.to",
  "com.cg.hkrbudgeting.analytics.common.api.to",
   "com.cg.hkrbudgeting.integration.common.api.to"
 }
)
public class ODataHkrServlet extends AbstractODataServlet {

}
```

Listing 6

```
ID: org.capgemini.hkrbudgeting
version: 1.0.0
applications:
 - name: hkrbudgeting-demo
   memory: 8192M
   type: java
   buildpack: sap_java_buildpack
   path: core/target/hkrbudgeting-core-bootified.jar
   env:
    JBP_CONFIG_COMPONENTS: [jres: ['com.sap.xs.java.buildpack.jdk.SAPJVMJDK']]
    JBP_CONFIG_JAVA_OPTS: '[from_environment: false, java_opts: "-Dfile.
encoding=UTF-8"]'
```

Listing 7

```
{
 ...
 "dataSources": {
  "hkrbudgeting-api": {
   "uri": "/services/odata.svc/",
   "type": "OData",
   "settings": {
    "odataVersion": "2.0"
   }
  }
 },
 ...
 "models": {
  "oDataModel": {
   "dataSource":
    "verwaltungseinheiten.datasource.
                          hkrbudgeting-api ",
   "type": "sap.ui.model.odata.
                          v2.ODataModel",
   "preload": true,
   "settings": {
    "useBatch": true
   }
  }
 },
 ...
}
```

```
public interface AdministrationUnitService
  extends ODataOperationService<AdministrationUnitTo> {

}
```

OData Servlet

Sobald die CRUD-Operationen über die drei Schichten hinweg implementiert sind, muss ein OData Web Service exponiert werden. Der Web Service muss die Anfragen über HTTP gemäß OData annehmen und zu den richtigen Service-Klassen orchestrieren. Hierfür haben wir einen OData-Prozessor basierend auf dem Olingo-Framework als devonfw-Modul implementiert. Mit Hilfe der Spring-Annotation *@Service* und *@WebServlet* und der Vererbung der abstrakten Klasse *AbstractODataServlet* wird ein Java Servlet erzeugt. Im Beispiel ist der ODATA-Service über den URL */services/odata.svc/* erreichbar (Listing 5). Über die Annotation *@ODataEntityScan* können wir Java-Pakete angeben, woraus das OData-Modell generiert wird. Zusätzlich zu den CRUD-Operationen können in OData auch Function Imports definiert werden, um zusätzliche Geschäftslogiken zu realisieren, die über eine Oberfläche angestoßen werden können. In diesem Anwendungsfall wurde auf Function Imports verzichtet.

Möchte man alle Verwaltungseinheiten über OData im Browser angezeigt bekommen, muss die folgende HTTP-GET-Anfrage gesendet werden: *GET /services/odata.svc/Verwaltungseinheiten*. Lesen einer konkreten Verwaltungseinheit mit dem Schlüssel *ID=1* ist über die folgende HTTP-GET-Anfrage möglich: *GET /services/odata.svc/Verwaltungseinheiten(1L)*.

Die *Create-* und *Update*-Operationen funktionieren analog mit *HTTP POST* oder *HTTP PUT* inklusive der Entitätsdaten als JSON im HTTP Body.

Deployment auf HANA XSA

Sobald die Java-Applikation als *.jar* oder *.war* (zum Beispiel mit Maven) gebaut wurde, kann ein Deployment auf der HANA-XSA-Infrastruktur durchgeführt werden (Listing 6). Aufgrund der starken Ähnlichkeit zu Cloud Foundry gibt es eine Clientsoftware (xs-client), mit deren Hilfe sich das Deployment automatisieren lässt. Hierfür müssen wir eine Deployment-Datei definieren. In dieser Datei legen wir die operationalen Parameter wie zum Beispiel Speicher, Anzahl der Instanzen oder Buildpacks der Java-Applikation fest.

Das Deployment kann mit dem folgenden Befehl durchgeführt werden: *xs push -f mta.yaml*. Um ein homogenes Benutzererlebnis innerhalb der SAP-Welt zu gewährleisten, liegt es nahe, auch das UI in der gleichen Technologie wie in der Standard-S/4HANA-Software zu implementieren.

Entwicklungsframework SAPUI5

SAP UI5 ist ein Model View Controller (MVC) JavaScript Framework, das auf HTML5 basiert. Es bringt eine große Anzahl an UI-Elementen für gängige Enter-

prise-Applikationen mit. Neben der SAP-UI5-Version gibt es eine frei verfügbare Open-Source-Variante namens Open UI5 [10]. Das Programmiermodell ähnelt Angular. Dabei werden UI-Elemente und die zugehörige Benutzerinteraktionslogik getrennt voneinander entwickelt. Die UI-Beschreibungssprache ist XML, die Benutzerlogik wird in JavaScript implementiert. Die Bereitstellung der Konfigurationseinstellungen erfolgt über *.json*-Dateien.

Beispiel: Anbindung des Erfassungsdialogs für die Verwaltungseinheit an OData

Eine der großen Stärken von SAP UI5 ist die gute Anbindungsmöglichkeit an OData. Über eine zentrale Konfigurationsdatei (*manifest.json*) können ein oder mehrere Modelle definiert werden, die anschließend überall in der Applikation als JavaScript-Objekte zur Verfügung stehen. Hierfür wird zuerst ein Quellsystem (siehe *dataSources*) mit dem jeweiligen URI zum OData Service definiert. Das Quellsytem wird mit Hilfe einer Java-Applikation auf HANA XSA exponiert. Anschließend wird ein Modell *oDataModel* erzeugt (Listing 7). Eine Anbindung an S/4HANA über OData ist ebenfalls möglich. Die UI-Applikationen können gleichzeitig auch auf S/4HANA und die individuell entwickelten Komponenten zugreifen.

Mit dem JavaScript-Objekt *oDataModel* können wir in der Applikation arbeiten. Entweder bindet man dazu die UI-Elemente direkt mit dem OData Service über das Modell ein oder man führt im Code Operationen darauf aus.

Als Beispiel ist hier eine *Create*-Operation aufgelistet. Wir rufen einen OData Path (hier */services/odata.svc/Verwaltungseinheiten*) auf und übergeben die Daten als POST im HTTP Body (Listing 8). Das jeweilige Java-Backend-System nimmt diese Anfrage an und speichert die Daten in einer HANA-Datenbank ab.

Fazit

Das hier vorgestellte Beispiel demonstriert, wie sich auf der neuen SAP-Plattform Standard- und individuelle Lösungen auf Basis der Programmiersprache Java nahtlos miteinander integrieren lassen. „Nahtlos" bedeutet zum einen, dass es für den Benutzer durch das homogene Look and Feel keine Brüche gibt und er nicht bemerkt, ob er mit einer Standard- oder einer individuell entwickelten Komponente arbeitet. Die Benutzerschnittstelle präsentiert sich also aus einem Guss. Zum anderen sind wir überzeugt, dass es sich aus Betriebssicht vielfach auszahlt, Standard- und Individuallösungen mit diesem Ansatz auf einer einzigen Plattform (in diesem Fall einer SAP-Plattform) zu betreiben.

Ein weiterer wesentlicher Vorteil ist, dass individuelle Anwendungskomponenten auf Basis offener Standardtechnologien entwickelt werden können, was einem Vendor Lock-in und einer damit verbundenen Abhängigkeit entgegenwirkt. Damit ist es zugleich möglich, diese Komponenten auch außerhalb der SAP-Plattform einzusetzen. Eine weitere Form des Vendor Lock-ins ist die Abhängigkeit von externen Dienstleistern. Auch diesem Risiko wird vorgebeugt, indem alle allgemeinen Blaupausen sowie die Werkzeuge über freie Lizenzen wie beispielsweise Apache v2.0 bereitgestellt werden. Das alles schließt natürlich nicht aus, auch auf eigene Frameworks für individuelle Java-Lösungen zurückzugreifen.

Jakob Boos ist Experte für IT-Strategie und -Architektur bei Capgemini. Seit über zwanzig Jahren berät er in seiner Managementfunktion Behörden und Organisationen der öffentlichen Verwaltung zum Einsatz von Standard- und individuell entwickelter Software. Seine Leidenschaft ist, neue Trendthemen kritisch zu hinterfragen und den tatsächlichen Nutzen für den öffentlichen Sektor zu ermitteln.

Manjinder Singh ist leitender Software Engineer bei Capgemini. Er berät Kunden im Bereich IT-Architektur, insbesondere zu SAP und Individualentwicklungen. Seine Schwerpunkte sind SAP HANA, Fiori und Cloud-Plattformen wie SAP Cloud und AWS sowie hybride Architekturen. Er verfügt über einen Master in Computer Science der RWTH Aachen und der École Polytechnique in Lausanne.

Links & Literatur

[1] Open-Application-Standardplattform: https://github.com/oasp/

[2] SAP UI5 auf GitHub: https://github.com/SAP/openui5/

[3] OpenJDK: https://openjdk.java.net

[4] 12-Factor-App: https://12factor.net

[5] OData Service V2: https://www.odata.org/documentation/odata-version-2-0/

[6] Apache Olingo: https://olingo.apache.org

[7] Spring: https://spring.io

[8] devonfw-Framework auf GitHub: https://github.com/devonfw/devon4j/

[9] Dozer-Framework: http://dozer.sourceforge.net

[10] Open UI5: https://openui5.org

Listing 8

```
...
saveEntity: function(context, sPath, oEntry, oModel) {
  let promise = jQuery.Deferred();
  oModel.create(sPath, oEntry, {
    success: (oData) => {
      oModel.refresh(true);
      MessageToast.show(I18nPropertyGetter.getProperty("success_on_create", context));
      promise.resolve(oData);
    },
    error: (oData) => {
      let errorMessage = this._prepareErrorMessage(oData, context);
      MessageToast.show(errorMessage, {
        duration: 10000
      });
      BusyDialog.hide(context);
    }
  });
  return promise;
}
...
```