den, auf eine Zeitachse. Dabei stehen hoch priorisierte, risikoarme und nutzenintensive Maßnahmen in vorderer Reihe.

- *Quantifizierung des Nutzens*
  Für die Roadmap in ihrer Gesamtheit werden die zu erwartenden Benefits geschätzt.
- *Produkt-Assessment*
  Verschiedene Optionen zur IT-Unterstützung werden aufgestellt und verglichen. Integrierte Systeme, die mit kleinen Initiativen über modulare Lösungen schrittweise erreicht werden können, sind dabei zu bevorzugen.
- *Kostenrechnung*
  Ausgehend von den Kosten der Projekt-Initiativen und den Kosten der identifizierten Produkte bildet die TCO-Bestimmung das Gegengewicht zur Nutzenrechnung im Business Case.
- *Aktionsplan und Change Plan*
  Change Management ist für umfassende Transformationen bereits zu Beginn als notwendiger Faktor einzuplanen. Gleiches gilt für sämtliche zusätzlichen Aufgaben der Mitarbeiter, die über ihre aktuellen Tätigkeiten hinausgehen. Beide Punkte müssen parallel zur Roadmap in einem Plan dargestellt werden.
- *Definition eines Quick-Win-Projekts*
  Für die erste mögliche Maßnahme zur Realisierung der CX-Strategie wird eine Entscheidungsvorlage erstellt, die nochmals Kosten und Nutzen nur für dieses erste Projekt ausweist.

Bei Berücksichtigung dieser Punkte wird das Risiko einer Fehl-Investition bei den ersten Schritten in Richtung Customer Experience Management reduziert. Das Unternehmen des Autors bietet ein einführendes CX Readiness Assessment von vier Wochen an, in dem alle genannten Aspekte in Zusammenarbeit mit einem CX-interessierten Unternehmen beleuchtet werden.

Earl-Bertram Kühne
e.kuehne@reply.de

Die E-Business Suite mit Apex erweitern

Jan Frikin, Marquard & Bahls AG

Jeder kennt Oracle Application Express (Apex) als einfaches und schnelles Toolset, um Standalone-Apps oder ganze Webseiten zu entwickeln. Das Framework ist aber auch für die Entwicklung von „Oracle E-Business Suite"-Erweiterungen geeignet. Der Artikel zeigt ein Projekt des Mineralölunternehmens Marquard & Bahls AG.

Wenn IT-Teams Anfragen von Anwendern oder Kunden in puncto Entwicklung oder Customizing im Rahmen der Oracle E-Business Suite (EBS) erhalten, dann lautet die erste Antwort meistens: „Okay, lass es uns in Oracle Forms machen oder ein paar Flex-Fields einsetzen". Dann kommen weitere Überlegungen ins Spiel: „Es gibt auch noch Oracle Application Framework (OAF) und Oracle Application Development Framework (ADF). Wir könnten etwas Raffiniertes machen, allerdings müssten wir Leute umschulen, zusätzliche Applicationsserver anschaffen etc. Und eine Sache noch: das ist alles Java." Spätestens ab diesem Zeitpunkt fängt ein Teamleiter an, sich Gedanken darüber zu machen, wie seine Mannschaft sich überhaupt auf eine solche Herausforderung vorbereiten kann.

Das waren auch bei der Marquard & Bahls AG die ersten Gedanken der IT-Abteilung. Im Laufe eines Upgrades von EBS 11*i* auf das Release 12 kamen Probleme im „Customer Modul" auf. Dieses sorgt für die Verwaltung von Kunden und Adressen. Das Entwicklungsteam hatte eigene FlexFields programmiert, die es im neuen Release nicht mehr zum Laufen brachte. Dahinter verbergen sich EBS-typische Felder.

Darüber hinaus kritisierten die Anwender die vielen Dateneingabe- und Such-Masken, zwischen denen sie immer hin- und herspringen mussten. Sie plädierten für eine einzige Maske („One Screen Entry"), die für eine schnellere Erfassung der Daten sorgen sollte. Das Pflichtheft dazu war ziemlich kurz:

- Die Erweiterung sollte weniger Masken haben – am besten eine Einzige
- Eine Beschleunigung der Daten-Eingaben sollte erfolgen, indem Werte vom System vorgeschlagen werden, allerdings ohne auf Lists of Values (LOV) zurückzugreifen
- Die Sichtbarkeit von zusätzlichen Feldern sollte sowohl bei den in EBS bereits vorgesehenen FlexFields als auch bei den selbstprogrammierten Custom FlexFields erhöht werden
- Natürlich sollte ein besonderes Augenmerk auch auf die Cross-Kompatibilität zwischen Erweiterung und den Standardmasken von EBS R12 liegen

Neben Apex standen auch die Frameworks Forms 11*g*, ADF und OAF zur Wahl. Gegen OAF sprach die mangelnde Flexibilität beim Erstellen von komplexen Masken. Nach einer Gegenüberstellung der Technologien fiel die Entscheidung zugunsten von Apex.

Apex-Know-how

Das vorhandene Know-how und die bereits gesammelten Erfahrungen mit Apex 3 in den Jahren zuvor waren das wichtigste Entscheidungskriterium für die Wahl von Apex.

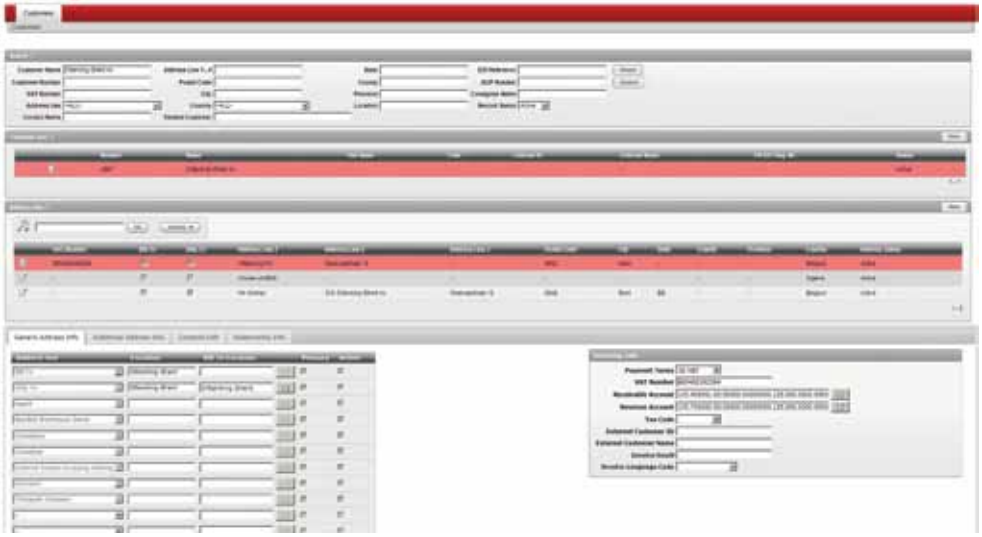

Abbildung 1: Eine einzige Maske mit Suchfeldern – so sieht die Anwendung aus

Vor allem die Verwendung von PL/SQL-Paketen innerhalb von Apex-Applikationen beherrschte das Team bereits. Die ersten Erfahrungen hatten die Entwickler mit einer Apex-Integration in EBS 11*i* im Bereich „Interactive Reports" gesammelt.

Das Apex-Standardmodul erlaubt eine flexible Darstellung von Daten aus einer oder mehreren Tabellen anhand von einfachen SQL-Statements. So können Anwender ihre Berichte selbst definieren und erstellen. Dabei werden die Daten aus Standard- und Custom-„Table Spaces" gelesen und in einer Apex-Applikation dargestellt. Die Funktionalität von „Interactive Reports" erlaubt sowohl die Suche als auch die Anpassung der Daten-Darstellung in verschiedenen Formularen. Zum Beispiel können ausgewählte Spalten oder Zeilen nach bestimmten Parametern addiert oder gruppiert optimal dargestellt werden. Großer Unterschied: Es kommen keine Daten-Eingaben vor.

Ein weiterer Vorteil von Apex: Mit dem Framework sind Funktionalitäten wie Type-ahead oder die Einbindung einer Suche in jedem Feld einfach realisierbar. Meistens basieren diese auf JQuery. Dort wo das Open-Source-Produkt keine Lösung bietet, lassen sich immer noch eigene Plug-ins erstellen, die dann in jedem Modul der Applikation zur Verwendung kommen können.

GlassFish als Applicationsserver

Die Frage des Applicationsservers war auch noch zu klären. Unter EBS 11*i* konnten Apex-Applikationen mittels des Oracle-„http"-Servers „mod_plsql" auf Oracle Internet Application Server (iAS) laufen. Da EBS R12 dies nicht mehr unterstützt, waren die erwähnten Interactive Reports zu migrieren, die von dem Development-Team entwickelt worden waren.

Oracle selbst empfiehlt den Einsatz von WebLogic- oder GlassFish-Servern in Ver-

Das ist ein FlexField

Unternehmen verwenden Codes, um Hauptbuch-Konten, Produkte, Kunden oder weitere Geschäftseinheiten zu identifizieren. Diese Codes sind in Segmenten unterteilt, die Detail-Informationen enthalten.

In der Oracle E-Business Suite sind diese Codes in sogenannten „FlexFields" gespeichert. Diese sorgen für eine durchwegs gleichbleibende Struktur der Daten. Es handelt sich dabei um Felder in der EBS-Maske, die vordefinierten Tabellen in der Datenbank entsprechen. Dabei kann sowohl das Code-Schema als auch die Art des Felds (zum Beispiel List of Values, Text) den Bedürfnissen des Unternehmens angepasst werden.

Man unterscheidet zwischen Pflichtfeldern (Key FlexFields) und optionalen Feldern (Descriptive FlexFields). Die descriptive FlexFields sind kontextabhängige Felder und werden genutzt, um unternehmensspezifische Informationen zu speichern. Pro Modul stellt Oracle fünfzehn FlexFields zur Verfügung, die frei konfigurierbar sind. Wer darüber hinaus weitere Felder braucht, ist auf „Custom FlexFields" angewiesen. Das sind selbstentwickelte Descriptive FlexFields. Diese Vorgehensweise ist allerdings von Oracle nicht gern gesehen.

FlexFields sind nicht sehr anwenderfreundlich: Um in Erfahrung zu bringen, aus welchen Informationen sich ein Code zusammensetzt, muss der Anwender auf den Code klicken. Die Detail-Informationen sind in einem Pop-up-Fenster angezeigt.

Abbildung 2: Kompakt und übersichtlich: Die Informationen sind auf vier Tabs verteilt

Abbildung 3: Von EBS zu Apex – so erfolgt die Übertragung der Daten von EBS nach Apex und umgekehrt

bindung mit iAS 10g, dem Standard-Applicationsserver für Oracle EBS R12. Nach einem R&D-Projekt sowie ersten Erfahrungen mit der „Oracle BI Suite 11*g*" stufte das Team WebLogic als weniger geeignet ein: Gegen den Applicationssserver sprach der hohe Installations- und Verwaltungsaufwand sowie das fehlende Know-how. Besonders die Verwaltung von User-Rechten (vor allem bei der Authentizierung) erschien mit WebLogic schwer realisierbar, da sie mittels Lightweight Directory Access Protocol (LDAP) erfolgt. Darüber hinaus ist es den Entwicklern nicht gelungen, mit WebLogic eine einfache Lösung zur Übertragung einer Web-Session zwischen EBS und Apex zu finden. Mit GlassFish indes ist eine verschlüsselte Cookie-Übertragung, die auch vom iAS-Server übernommen wird, einfach zu realisieren. Weil die Datenbank-Administratoren im Hause zudem positive Erfahrungen mit GlassFish gemacht hatten, folgte das Development-Team der Empfehlung.

Hinweis: Beide Applicationsserver benötigen „Application Express Listener", um mit der Apex-Engine zu kommunizieren. Dazu stellt Oracle eine gute Dokumentation [1] zur Verfügung.

Die Entwicklungsphase

Das Projekt lief von April 2011 bis Februar 2012. Es war zeitlich an einer Migration der EBS von 11*i* auf 12 gekoppelt. Erst im Dezember desselben Jahres veröffentlichte Oracle ein White Paper [2] genau zu dieser Thematik. Die Empfehlungen von Oracle unterscheiden sich etwas von dem Weg, den die Marquard & Bahls AG gegangen ist – allerdings nicht im Wesentlichen. Aber während das Mineralölunternehmen noch mit Apex 4.0 und 4.1 arbeitete, setzte Oracle auf Apex 4.2. Insbesondere das Log-in- und Responsibility-Konzept, wie von Oracle beschrieben, passte nicht zu den Anwendungsszenarien.

Mittlerweile gab es auch den ersten Entwurf eines „Functional Designs". Er war gelungen, doch die Realität hatte das Team eingeholt: Von der Anwenderseite kamen neue Anforderungen. Die Applikation sollte neben den Standard-Masken von EBS R12 koexistieren, um einen nahtlosen Daten-Austausch gewährleisten zu können. Insbesondere für die Buchhaltung war es von großer Bedeutung, die Standard-FlexFields im Accounting (General-Ledger-Modul) und die von EBS verwendeten FlexFields weiterhin zu nutzen. Eine andere technische Herausförderung war die Erstellung von LOVs mit mehreren Spalten. Diese sind mit Forms einfach zu realisieren, in Apex ist der Aufwand an der Stelle deutlich höher.

Das Endergebnis

Im Gegensatz zu der Vorgehensweise in EBS wird der Anwender in der Apex-Applikation geführt. Der Einstieg erfolgt über eine einheitliche Maske mit Suchfeldern, die „Auto Complete"-Funktionalitäten beinhalten (siehe Abbildung 1). Bereits bei der Eingabe von Suchbegriffen macht die Applikation die entsprechenden Vorschläge. Der Anwender selektiert dann den entsprechenden Eintrag. Weiter unten ist der Haupteintrag mit dem Kundennamen dargestellt. Über die Auswahl der Adresse werden auch weitere Adressdaten angezeigt. Während alle diese Felder früher in EBS auf mehreren Custom FlexFields in verschiedenen Pop-up-Fenstern verteilt waren, sind nun in Apex alle Informationen übersichtlich in vier Tabs dargestellt (siehe Abbildung 2). Diese enthalten sowohl generische als auch spezifische Daten. Auch der „Electronic Data Interchange" (EDI) mit Partner-Unternehmen oder zollrelevante Informationen sind in einem Tab gespeichert.

Logik in PL/SQL-Paketen abgelegt

Das Team hat sich letztendlich für eine Apex-Lösung entschieden, bei der die gesamte Logik in separaten PL/SQL-Paketen abgelegt ist. Das Gleiche gilt für das „User Interface" und die Daten-Schnittstelle. Die Lesedaten werden mittels Views von der Datenbank in die Applikation geholt, während die Standard-TCA-API dafür sorgt, dass neue Daten von der Anwendung in die Datenbank wandern. Zwei PL-SQL Module er-

Abbildung 4: PL/SQL Modul für die Anwendungsmaske

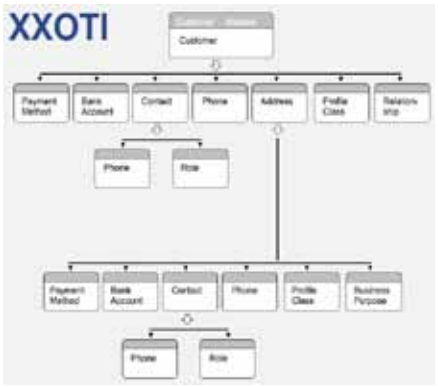

Abbildung 5: Apex-Datenmodell

möglichen, die Daten einzulesen (siehe Abbildung 3).

Es gibt hierzu zwei separate Daten-Schemata: eins für Apex („custom XX…"), das speziell für die Apex-Applikation entwickelt wurde, und eins für Oracle EBS („HZ…"). Diese Modelle sind mittels PL/SQL-Modulen („xxoti_customers_api_pk", siehe Abbildung 2) verlinkt und werden durch die jeweilige Applikation gestartet.

Apex bietet hervorragende Möglichkeiten, jedes denkbare PL/SQL-Modul wiederzuverwenden. Derzeit sind nun mehrere API_PK-Module im Einsatz. Mithilfe dieser Module könnten beliebig viele Apex-Applikationen – beziehungsweise auch separate

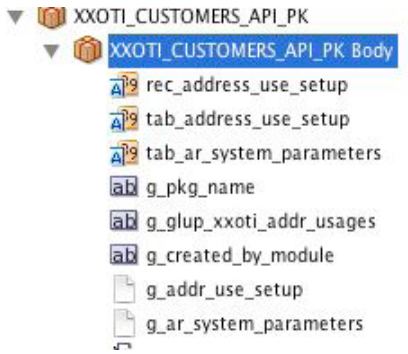

Abbildung 6: PL-SQL-Modul für die API

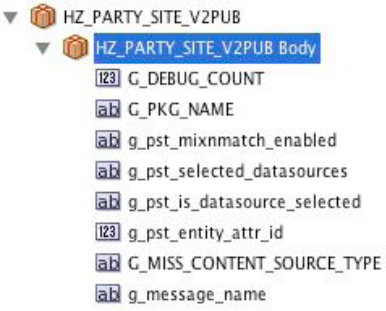

Abbildung 7: PL-SQL-Modul und EBS-Seite

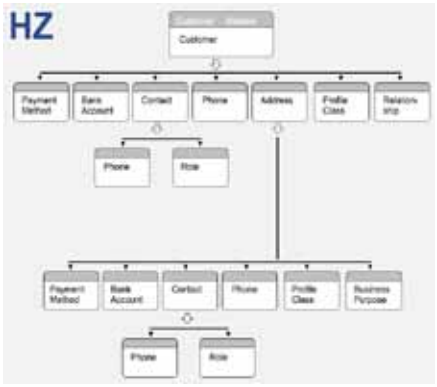

Abbildung 8: EBS-Datenmodell

Forms- oder Java-Anwendungen – direkt mit EBS verbunden werden. Deswegen ist die Business-Logik beziehungsweise die Schnittstellen einfach über einzige Module zu entwickeln oder anzupassen (siehe Abbildungen 4 – 8).

Anhand eines Beispiels kann man dieses Verfahren am besten veranschaulichen: In dem Auszug aus dem PL/SQL-Paket (siehe Abbildung 9) werden die Updates, die die Kunden-Adressen betreffen, in eine Tabelle geschrieben und mittels TCA ins „EBS HZ-Model“ übertragen.

Als erstes kommen die Daten aus dem Apex-Formular. „v.“, das in Apex für „Value“ steht, wird in die Tabelle geschrieben. Das Formular hat eine eigene Tabellen-Struktur.

```
PROCEDURE process_customer_4_maintain_pg
( p_customer_id    IN OUT NOCOPY NUMBER
, p_customer_number OUT NOCOPY VARCHAR2
, x_return_status OUT NOCOPY VARCHAR2)
IS
  c_source              CONSTANT VARCHAR2(60) := g_pkg_name ||
'.process_customer_4_maintain_pg';
  --
  l_customer_rec        xxoti_customers_api_pk.rec_customer;
  l_return_status       VARCHAR2(1);
  l_error_message       VARCHAR2(4000);
  l_save_success_msg    VARCHAR2(4000);
BEGIN
  log_message( p_text    => '> Begin ' || c_source
                          , p_level  => xxoti_logdb_pk.info
                          , p_source => c_source
                          );
  -- set default to success
  x_return_status := xxoti_customers_api_pk.g_return_sts_success;
  --
  IF v('P201_ACTION_ENTITY') = g_entity_customer
  THEN
    -- initialize customer record
    l_customer_rec.customer_id              := NVL(p_customer_id,
v('P201_CUSTOMER_ID'));
    l_customer_rec.party_id                 := v('P201_CUST_PARTY_ID');
    l_customer_rec.customer_number          := v('P201_CUSTOMER_NUMBER');
    l_customer_rec.NAME                     := v('P201_NAME');
    l_customer_rec.old_name                 := v('P201_OLD_NAME');
    l_customer_rec.short_name               := v('P201_SHORT_NAME');
    l_customer_rec.code                     := v('P201_CODE');
    l_customer_rec.external_id              := v('P201_EXTERNAL_ID');
    l_customer_rec.external_name            := v('P201_EXTERNAL_NAME');
    l_customer_rec.ph_637_reg_num           := v('P201_PH_637_REG_NUM');
    l_customer_rec.status                   := v('P201_CUSTOMER_STATUS');
    --l_customer_rec.vat_number               := v('P201_VAT_NUMBER');
    l_customer_rec.party_ovn                := v('P201_CUST_PARTY_OVN');
    l_customer_rec.customer_ovn             := v('P201_CUST_CUSTOMER_OVN');
    --
    -- CALL business logic
    xxoti_customers_api_pk.process_customer
      ( p_customer                  => l_customer_rec
      , x_return_status             => l_return_status
      , x_error_message             => l_error_message
      );
```

Abbildung 9: Der Weg von Apex zu EBS

```
-- ------------------------------------------------------------------------
-- initialize TCA records
l_cust_account_rec.cust_account_id          := p_customer.customer_id;
l_cust_account_rec.account_number           := NVL(p_customer.customer_number, fnd_api.g_miss_char);
l_organization_rec.organization_name        := NVL(p_customer.NAME, fnd_api.g_miss_char);
l_cust_account_rec.attribute10              := NVL(p_customer.old_name, fnd_api.g_miss_char);
l_cust_account_rec.attribute12              := NVL(p_customer.code, fnd_api.g_miss_char);
l_cust_account_rec.attribute13              := NVL(p_customer.external_id, fnd_api.g_miss_char);
l_cust_account_rec.attribute14              := NVL(p_customer.external_name, fnd_api.g_miss_char);
l_organization_rec.party_rec.party_id       := NVL(p_customer.party_id, fnd_api.g_miss_num);
l_cust_account_rec.attribute19              := NVL(p_customer.ph_637_reg_num, fnd_api.g_miss_char);
l_cust_account_rec.attribute20              := NVL(p_customer.short_name, fnd_api.g_miss_char);
l_cust_account_rec.status                   := NVL(p_customer.status, fnd_api.g_miss_char);
l_organization_rec.tax_reference            := NVL(p_customer.vat_number, fnd_api.g_miss_char);
--
IF l_cust_account_rec.cust_account_id IS NULL
THEN
  l_cust_account_rec.created_by_module := g_created_by_module;
  --
  hz_cust_account_v2pub.create_cust_account
    ( p_init_msg_list         => fnd_api.g_true
    , p_cust_account_rec      => l_cust_account_rec
    , p_organization_rec      => l_organization_rec
    , p_customer_profile_rec  => NULL --l_customer_profile_rec
    , x_cust_account_id       => p_customer.customer_id
    , x_account_number        => p_customer.customer_number
    , x_party_id              => p_customer.party_id
    , x_party_number          => l_party_number
    , x_profile_id            => l_profile_id
    , x_return_status         => l_return_status
```

Abbildung10: Das Modul wird nach jedem Update gerufen

Anschließend schreibt die Mapping-Logik die Daten in eine Memory-Tabelle. Danach werden die Values an TCA übergeben, wo entweder eine Neuanlage der Daten oder ein Update stattfindet. Auch in dieser Logik entstehen viele Defaults. Das letzte Modul wird automatisch nach jedem Update von Kunden-Strukturen gerufen (siehe Abbildung 10).

Abbildung 11: Apex-Applikation

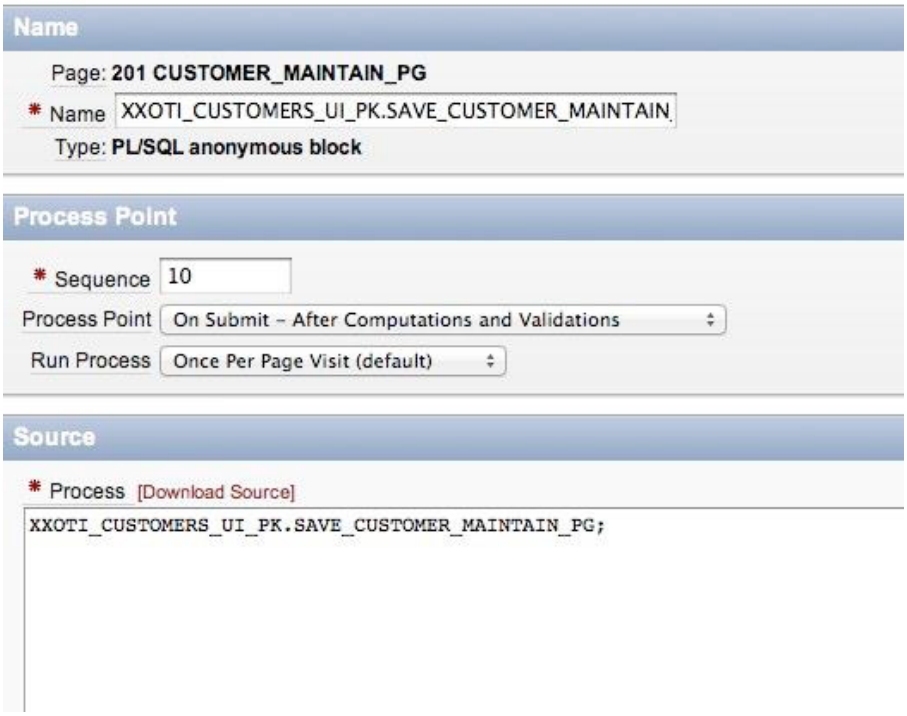

Abbildung 12: Aufruf eines PL-SQL Moduls innerhalb der Apex-Applikation

Hinweis: Im produktiven Einsatz wies die sogenannte „v.-Methode" Performance-Probleme auf. Der PL/SQL-Code soll an dieser Stelle angepasst werden. Die Lösung ist allerdings noch nicht implementiert. Laut Angaben von Oracle und Apex-Gruppen gibt es hierzu sehr unterschiedliche Meinungen. Zu diesem Zeitpunkt wird davon ausgegangen, dass diese Problematik bei jeder neuen Anwendung separat neu zu bewerten ist.

Innerhalb der Apex-Applikation selbst erfolgen also nur Aufrufe von PL/SQL-Paketen. Diese beinhalten den gesamten Code. Das vereinfacht die Anpassung und Wartung der Business-Logik (siehe Abbildungen 11 und 12).

Es wäre natürlich möglich und relativ einfach, den Code (insbesondere die Parameter) in die Apex-Applikation zu integrieren, anstatt separate Module zu erstellen. Einige Gründe sprachen in diesem spezifischen Fall dagegen. Zunächst einmal hatte das Team, wie bereits erwähnt, bereits Erfahrungen mit diesem Verfahren gesammelt.

Aber vor allem die Frage vom „Version Control" hätte auf dieser Weise nicht gelöst werden können: Da Apex-Applikationen komplett als Teil einer Datenbank gesehen werden, ist es nicht möglich, Teile von Applikationen in einem einfachen Verfahren ein- und auszuchecken. Vielmehr müsste die gesamte Applikation exportiert und importiert werden, was in dem Fall einer Entwicklung im Team nahezu unmöglich gewesen wäre.

Auch im Umgang mit Ajax gab es ein paar Bedenken: Da nicht klar war, wie Ajax in einer heterogenen Landschaft (basierend auf normalen PCs und abgespeckten Citrix-Clients) funktionieren würde, hat das Entwicklungsteam viel Code auf der Server-Seite gelassen. Apex-Native-Code ist insofern nur in einem geringen Umfang vorhanden.

Nach dieser einschlägigen Erfahrung realisiert das Entwicklungsteam der Marquard & Bahls AG fast jede weitere Entwicklung mit Apex. Die Resonanz ist bei den Anwendern aufgrund der Einfachheit der Maske sehr positiv. Einziger Wermutstropfen: der Lösungsansatz beansprucht den Server sehr. An der Stelle darf allerdings die Server-Performance kein Problem darstellen. Andererseits braucht iAS, worauf GlassFish läuft, mit EBS R12 viel mehr Ressourcen als irgendeine Apex-basierte Applikation mit noch so vielen Server-Calls.

Weitere Informationen

[1] Dokumentation: Oracle Application Express Listener Installation and Configuration Guide Release 2.0: http://docs.oracle.com/cd/E37099_01/doc/doc.20/e25066/toc.htm

[2] White Paper Extending Oracle E-Business Suite Release 12 using Oracle Application Express: http://www.oracle.com/technetwork/developer-tools/apex/learnmore/apex-ebs-extension-white-paper-345780.pdf

Jan Frikin
ian.frikin@mbholding.de