Datenbankzugriff aus Eclipse Rich-Client-Anwendungen über das Internet

Johannes Michler
PROMATIS software GmbH
Ettlingen

Schlüsselworte:

Eclipse RCP, OSGi, JPA, EclipseLink, Webservices, Apache CXF

Einleitung

Der folgende Artikel soll anhand eines Beispiels aufzeigen, wie sich der Datenbankzugriff einer Eclipse Rich-Client-Anwendung über das Internet realisieren lässt. Dazu wird zunächst Eclipse und dessen OSGi-Komponentenframework Equinox vorgestellt und anschließend kurz der Datenbankzugriff mittels der JPA-Implementierung EclipseLink erläutert. Darauf aufbauend werden Möglichkeiten aufgezeigt, den Zugriff auf die Datenbank über Webservices abzuwickeln. Das Komponentenmodell hilft hierbei, eine gute Wiederverwendbarkeit der Datenbankzugriffs-komponenten zu erreichen (Single Sourcing).

Eclipse als (Java) Entwicklungsumgebung

Zunächst von IBM und ab 2001 als Open Source Projekt unter der Führung der Eclipse Foundation entwickelt (kostenlos gemäß der Eclipse Public License (EPL)), war Eclipse lange Zeit vor allem als integrierte Entwicklungsumgebung für Java bekannt. Aufgrund des modularen Aufbaus gibt es für die Eclipse-Plattform eine große Anzahl von Erweiterungen („Plugins“). Damit eignet sich Eclipse außer für die Entwicklung von Anwendungen in der Java Standard Edition auch für die Enterprise Version (JEE) sowie eine Vielzahl weiterer Programmiersprachen (z. B. PHP, C/C++). Darüber hinaus stehen Erweiterungen zur Modellierung (EMF) inklusive der Erstellung grafischer Editoren (GMF), für Reporting (BIRT), den Datenbankzugriff (EclipseLink) und vieles mehr zur Verfügung [1]. Die Entwicklung von Plugins wird durch die offene Architektur der Eclipse-Plattform begünstigt.

Offene Architektur der Eclipse-Plugins

Seit Version 3 basiert Eclipse auf dem dynamischen Java-Komponentenframework OSGi (Open Service Gateway initiative) und besteht zunächst nur aus dem leichtgewichtigen Kern „Eclipse Equinox“. Eine Anwendung, wie z. B. die Java-Entwicklungsumgebung selbst, besteht nun jeweils aus einer bestimmten Zusammenstellung von Plugins. Einem Plugin in der Eclipse-Terminologie entspricht dabei exakt eine OSGi Komponente, die auch Bundle genannt wird. Jede Komponente spezifiziert in ihrer Beschreibungsdatei "MANIFEST.MF" - neben Versionierungs- und Namenseigenschaften und der benötigten Laufzeitumgebung - explizit ihre Abhängigkeiten zur Außenwelt. Dabei muss einerseits spezifiziert werden, welche anderen Java-Pakete oder OSGi-Bundles in welcher Version von der Komponente benötigt werden und anderseits, welche Java-Pakete die Komponente selbst für andere bereitstellt. Diese Metadatei wird anschließend mit den übersetzten Java-Paketen sowie evtl. weiteren Ressourcen in ein Java-Archiv (JAR) gepackt. Die Laufzeitumgebung Equinox sorgt dann beim Laden der Komponente zunächst dafür, dass die benötigten anderen Komponenten geladen werden. OSGi erlaubt dabei auch das dynamische Laden oder Entladen von Bundles. Darüber hinaus kann ein Bundle gleichzeitig mehrmals innerhalb einer Laufzeitumgebung in verschiedenen Versionen geladen sein. Dies ermöglicht zumindest theoretisch

die Aktualisierungen von Plugins ohne Neustart der Umgebung und damit minimale bis keine Ausfallzeiten.
Vor allem bei der Entwicklung von Anwendungen mit grafischen Oberflächen kann durch Verwendung der Eclipse-Plattform von vielen bereits vorhandenen Mechanismen aus den eclipse.ui.* Komponenten profitiert werden. Damit lassen sich oft mit wenig eigenem Programmcode komplexe und dennoch komfortable Fensteraufbauten, Menüs usw. im Stil der Eclipse-IDE erstellen. Besonders stark macht sich dies bei der Implementierung von Anwendungen aus dem Modellierungsumfeld bemerkbar. Hier können dank der Eclipse-Modellierungs-Projekte EMF und GMF recht einfach eigene Modellierungswerkzeuge für bestimmte Anwendungsdomänen entwickelt werden. Mit "server-side equinox" gibt es aber auch bereits Ansätze dafür, das Equinox-Framework bei Serveranwendungen einzusetzen. Trennt man strikt in „für grafische Oberfläche zuständige“ und „sonstige“ Plugins, lassen sich, wie im Folgenden gezeigt wird, auf einfache Art und Weise dieselben Komponenten sowohl Server- als auch Client-seitig einsetzen.

Datenbankzugriff aus Java-Anwendungen
Für den Zugriff einer Java- und damit auch einer Eclipse RCP-Anwendung auf eine Datenbank existieren verschiedene Möglichkeiten: Ein bekannter Weg ist es, über JDBC eine direkte Verbindung zur Datenbank herzustellen und über diese Verbindung übliche SQL Select, Update und Insert Anweisungen auszuführen. Dieser Weg bietet über den Aufruf beliebiger SQL Anweisungen oder PL/SQL Prozeduren eine hohe Flexibilität und bei richtiger Verwendung eine hohe Performance. Allerdings ist es oft mühsam, die SQL-Anweisungen manuell zu erstellen (ohne dabei SQL Injection oder anderen Angriffen ausgesetzt zu werden). Die SQL-Anweisungen werden dabei außerdem oft datenbankabhängig.
Objektrelationale Abbildungen bieten hingegen eine automatische Abbildung zwischen Java-Objekten und relationalen Tabellen. Im Java-Umfeld hat sich dabei die Java Persistence API (JPA) - welche neuerdings in Version 2.0 vorliegt - als offizielle Brückentechnologie etabliert. Der Standard besitzt mehrere Implementierungen, beispielsweise Hibernate, Apache OpenJPA oder EclipseLink. Letzteres ging als Open-Source-Variante aus Oracle TopLink hervor und stellt auch die Referenzimplementierung von JPA 2.0 dar. Der nächste Abschnitt geht auf JPA genauer ein.

Java Persistence API: Entitäten und der Zugriff auf sie
Zentraler Punkt der JPA-Spezifikation sind sogenannte Entitäten. Diese kennzeichnen eine spezielle Form von Klassen, nämlich solche, die persistente Instanzen besitzen können. Eine Klasse kann als Entität deklariert werden, indem sie mit der Annotation @Entity markiert wird. Dabei verfolgt JPA an vielen Stellen das Prinzip "Configuration by exception": Für viele Einstellungen sind Standardwerte voreingestellt, die bei Bedarf jedoch überschrieben werden können. So führt obige @Entity Annotation automatisch zu einer Abbildung der Klasse auf eine Tabelle gleichen Namens, wobei die primitiven Eigenschaften auf Spalten gleichen Namens in der Tabelle abgebildet werden. Über weitere Annotationen lässt sich dieses Verhalten jedoch auch manuell beeinflussen. Beziehungen zwischen Entitäten werden in Java meist über Referenzen oder Behältertypen wie Felder, Listen oder Mengen realisiert. Auch diese können mit JPA über Annotationen auf die Datenbankentsprechungen wie "Fremdschlüssel-Beziehung" oder "Join-Tabelle" abgebildet werden.
Der Zugriff auf Entitäten erfolgt über einen sogenannten EntityManager. Über diesen können bestehende Entitäten gesucht oder gelöscht, aber auch neue Entitäten erstellt werden. Ein EntityManager stellt dabei eine Art Datenbanksitzung dar. Über ihn kann auch ein transaktionaler Kontext aufgebaut werden. Zur Erstellung eines EntityManagers dient die EntityManagerFactory. Diese existiert genau einmal pro Persistenzkontext und besitzt unter anderem einen Verbindungspool zur Datenbank sowie einen gemeinsamen Cache.

Entfernter Zugriff über das Internet
Mit den beschriebenen Technologien lässt sich auf einfache Art und Weise eine Rich-Client (RCP)-Anwendung bauen, die ihre Daten über ein individuelles Persistenz-Bundle mit Hilfe von JPA und JDBC in einer relationalen Datenbank ablegt. Meist sollen dabei jedoch mehrere Benutzer gleichzeitig auch von entfernten Standorten auf den Datenbestand lesend und schreibend zugreifen. Dann gibt es zwischen Anwender (und daher der Rich-Client-Anwendung) und Datenbank eine Lücke, die – oft über das Internet – überbrückt werden kann bzw. muss.
Grundsätzlich wäre es natürlich möglich, die JDBC-Zugriffe, welche ohnehin über TCP abgewickelt werden, direkt über das Internet durchzuführen und die auf JPA basierende Persistierungslogik auf der Client-Seite zu belassen. In vielen Fällen führt dies jedoch zu deutlich mehr und in der Summe auch größeren Datenübertragungen über eine oft langsame Verbindung. Außerdem kann auf JPA-Ebene kaum gecacht werden: Es gibt in diesem Szenario dann viele verteilte und unabhängige Entity-Manager-Factories deren Caches synchron zu halten wären. Auch aus Sicherheitsaspekten ist es fragwürdig, einen direkten JDBC-Zugriff nach außen bereitzustellen, da oft auch in der Datenbankzugriffslogik überprüft wird, ob eine bestimmte Operation für den aktuellen Benutzer erlaubt ist. Es bietet sich also an, die JPA-Zugriffsschicht „auf dem Server“ und damit in der „Nähe“ der Datenbank zu belassen. Dann erfolgt die Kommunikation zwischen den die Oberfläche betreffenden Teilen der RCP-Anwendung und der Datenbankanbindung über das Internet. Um diese Lücke zu überwinden, bieten sich unter anderem die noch recht neuen OSGi Remote-Services aber auch etablierte Mechanismen wie Webservices an. Letztere gelten bereits als ausgereift und ermöglichen es, einfach weitere Clients mit anderen Technologien zu implementieren. Deshalb soll im Folgenden der Fokus auf Webservices liegen.

Webservices mit Java
Mit JAX-WS gibt es für Java einen weitverbreiteten Standard für die Nutzung und Bereitstellung von Webservices [2]. Dessen Implementierung Apache-CXF hat sich im praktischen Einsatz im Eclipse/OSGi-Umfeld bewährt. Der Webservice-Stack bietet umfassende Möglichkeiten um einerseits einen Dienst selbst als Webservice bereitzustellen und andererseits einen bestehenden Webservice aufzurufen. In Abbildung 1 ist ein Beispiel-Dienst ResourceManagerImpl gezeigt, welcher per Webservice bereitgestellt werden soll. Die öffentliche Schnittstelle dieses Dienstes ist in der Schnittstelle IResourceManager definiert.

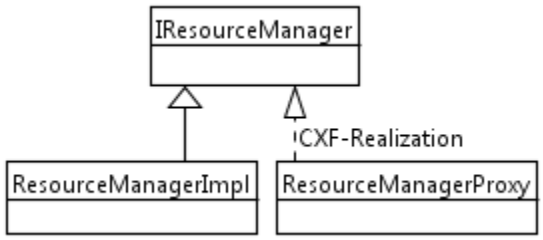

Abb. 1: Klassenhierarchie des per Webservice bereitgestellten Dienstes

Im Anwendungsfall wird sowohl auf der Server- als auch auf der Client-Seite CXF verwendet. Serverseitig stellt der Webservice-Stack unter anderem ein Servlet bereit, an dem Dienste registriert und damit bereitgestellt werden können. Das Servlet sorgt dabei automatisch dafür, dass anhand der Java-Schnittstelle der registrierten Dienst-Klasse dynamisch eine Webservice-Beschreibung (WSDL) generiert und bereitgestellt wird. Dadurch wird es möglich, die öffentlichen Methoden mittels SOAP anzusprechen. Das genaue Verhalten kann dadurch über JAX-WS-Annotationen an der Schnittstelle IResourceManager gesteuert werden. Durch diese können die Namen der Methoden und Parameter, aber auch z. B. die Art der Parameterübergabe gesteuert werden. Um einen so erstellten Dienst

aufzurufen, stellt CXF verschiedene Möglichkeiten zur Verfügung. Ein Weg ist es dabei, sich von CXF dynamisch einen Proxy erzeugen zu lassen (hier: ResourceManagerProxy). Dieser Proxy kann anschließend einer Variablen vom Typ des aufgerufenen Dienstes (IResourceManager) zugewiesen werden. Anschließend kann der entfernte Dienst beinahe wie ein lokales Objekt verwendet werden. Natürlich ist dabei unter anderem zu beachten, dass Methodenaufrufe an ein solches entferntes Objekt stets eine Call-By-Copy Semantik besitzen. Änderungen, die auf dem Server an den übergebenen Parametern vorgenommen werden, sind also anschließend beim Aufrufer nicht sichtbar. Die Java-Schnittstelle, der obiger Proxy zugewiesen wird, lässt sich entweder aus der WSDL automatisch generieren, oder sie kann, wenn Client und Server wie hier beschrieben beide mit CXF implementiert werden, einfach gemeinsam auf dem Client und dem Server verwendet werden.

Anwendungsszenario

Die beschriebenen Konzepte wurden konkret bei der Implementierung eines Prozessmodellierungswerkzeuges zur Speicherung der Modelle in einem gemeinsamen Repository verwendet. Dazu wird die in Abbildung 2 verwendete Architektur benutzt, die im Folgenden genauer erläutert werden soll.

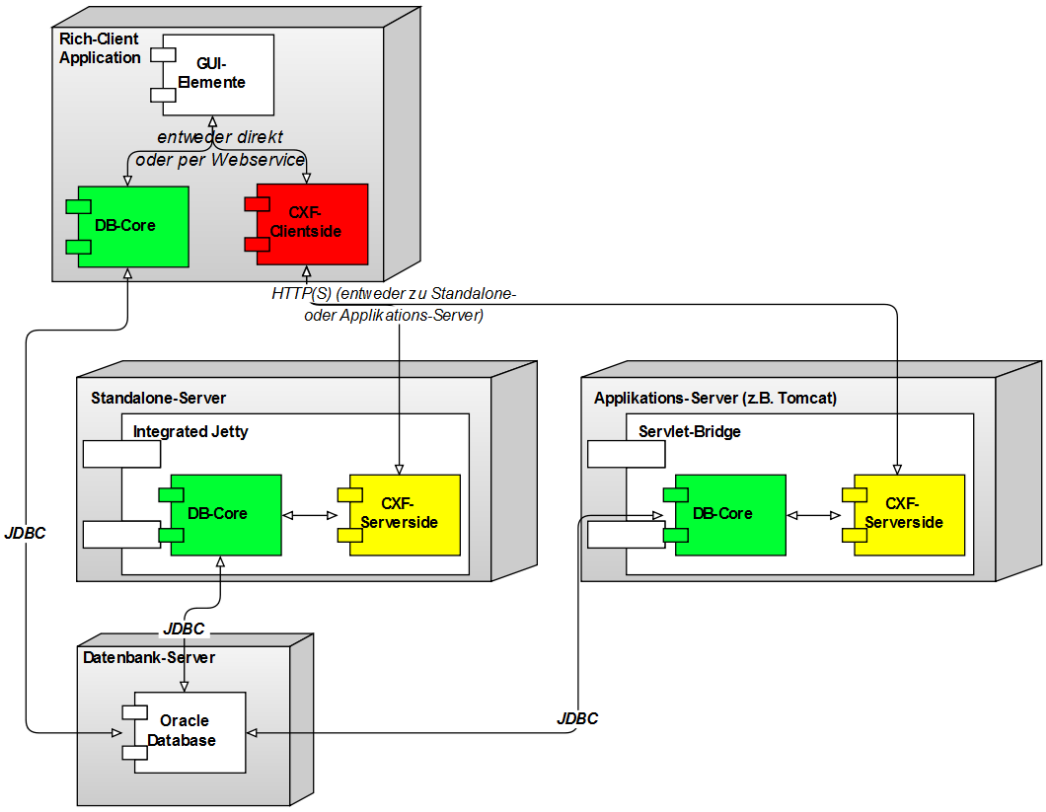

Abb. 2: Architektur im Fallbeispiel Horus Business Modeler

Die Kapselung der mit JPA implementierten Datenbankzugriffslogik im separaten Bundle "DB-Core" ermöglicht es, dass dieses Bundle sowohl bei der Anbindung der Datenbank direkt aus dem Rich-Client als auch bei der Nutzung der Webservice-Brücke ohne Änderung und sogar ohne neu zu übersetzen verwendet werden kann. Das Bundle stellt dazu die Implementierung ResourceManagerImpl der Schnittstelle IResourceManager bereit. Der entsprechende Teil der Anwendung greift dann entweder direkt auf "DB-Core" zu oder verwendet die von Apache CXF bereitgestellte Proxy-Klasse. Für die RCP-Anwendung ist dabei transparent, wie der Datenbankzugriff

bereitgestellt wird, da bei ihrer Implementierung bereits berücksichtigt wurde, dass sämtliche Aufrufe eine Call-By-Value Semantik erfahren.

Implementierung des Webservices

Serverseitig wurden zwei verschiedene Varianten zur Bereitstellung des Webservices implementiert. Für einfache Szenarien wurde - ebenfalls mit Eclipse RCP Mechanismen - eine Standalone Eclipse-Equinox-Anwendung erstellt (siehe Abb. 2 links). Diese startet beim Aktivieren ihres zentralen Bundles "CXF-Serverside" den als OSGi-Bundle bereitgestellten, leichtgewichtigen Servlet-Container Jetty und lädt in diesen das oben erwähnte Apache-CXF-Servlet. Alternativ bietet das Eclipse-Equinox-Projekt eine sogenannte Servlet-Bridge an. Mit dieser kann eine Equinox-OSGi-Umgebung als Webapplikation bereitgestellt werden (.WAR). Beim Laden dieser Applikation innerhalb eines Applikationsservers sorgt die Servlet-Bridge dafür, dass die leichtgewichtige OSGi-Umgebung mit ihren Plugins geladen wird (Abb. 2 rechts). Dies bietet den Vorteil, dass der Dienst auf einem existierenden Server mit einer im Umfeld der Anwendung gebräuchlichen URL über bereits freigeschaltete Ports bereitgestellt wird. Außerdem können die Authentisierungsmechanismen des zugrunde liegenden Applikationsservers verwendet werden. Über das OSGi-Services-Konzept steht dann den einzelnen geladenen Plugins der HTTP-Dienst des umgebenden Applikationsservers zur Verfügung. Das Bundle "CXF-Serverside" prüft deshalb bei seiner Aktivierung, ob von der Laufzeitumgebung bereits ein HTTP-Dienst bereitgestellt wird. Ist dies der Fall, so wird das CXF Webservice-Servlet bei diesem Dienst registriert. Falls nicht, wird wie oben beschrieben zunächst ein Jetty Webserver gestartet. Die anschließende Vorgehensweise unterscheidet sich dann bei beiden Varianten nicht mehr: Es wird eine Instanz der Datenbankzugriffslogik, die über eine Abhängigkeit zu diesem Plugin erreicht wird, erstellt und diese wird schließlich über das CXF Webservice-Servlet veröffentlicht:

```
ServiceReference sr=bndl.getServiceReference(HttpService.class.getName());
CXFNonSpringServlet srvlet = new CXFNonSpringServlet();
if (sr!=null) //Aktivierung in einem App-Server über Servlet-Bridge
 (HttpService)bndl.getService(sr).registerServlet("/",srvlet,null,null);
else{
  org.mortbay.jetty.Server server = new Server();
  Context jettyContext = new Context( server, "/");
  jettyContext.addServlet(srvlet,"/horus/*");
  server.start();
}
ServerFactoryBean factory = new JaxWsServerFactoryBean();
factory.setBus( srvlet.getBus());
JPA_DB_Access realService = new JPA_DB_Access(); //The service to expose
factory.setServiceBean( refManager);
factory.setAddress( "/dbaccess");
factory.create();
```

Absichern der Kommunikation

Abschließend soll noch kurz auf Möglichkeiten zur Sicherung der Kommunikation zwischen Client und Server eingegangen werden. Im Webservice-Umfeld bieten sich hier einerseits das WS-Security-Protokoll und andererseits die Mechanismen des eingesetzten Transportprotokolls - in der Regel SOAP über HTTP - an. Beide Möglichkeiten werden von CXF unterstützt und mit beiden lässt sich sowohl die Vertraulichkeit als auch die Authentizität einer Anfrage sicherstellen. Einfacher - vor allem bei Verwendung eines bestehenden Applikationsservers - gelingt dabei jedoch die Absicherung des Transportprotokolls. Im ersten Schritt kann dazu der Server auf HTTPS umgestellt werden. Der Apache CXF Client kann einen solchen Server genauso wie einen ungesicherten verwenden. Gegebenenfalls ist dazu noch das Zertifikat des Servers als für die Java-VM vertrauenswürdig bekannt

zu machen. Die Authentisierung kann dann über Client-Zertifikate oder wie hier geschehen über HTTP-Basic-Auth (Benutzername und Passwort) erfolgen. Verwendet man einen Applikationsserver, so muss der Bereich, in dem das Servlet bereitgestellt wird, lediglich als "zu schützen" markiert werden. Zur Benutzerüberprüfung kann ein beliebiges Verfahren des Applikationsservers verwendet werden. Im Beispielszenario authentisierte ein Tomcat-Server gegen ein LDAP-Verzeichnis. Die Autorisierung, das heißt die Festlegung, welche Privilegien ein Benutzer besitzt, erfolgte dabei innerhalb des "DB-Core" Plugins wieder für alle drei Zugriffsformen auf die gleiche Art und Weise.

Fazit

Es hat sich gezeigt, dass sich Eclipse nicht mehr nur als Java-Entwicklungsumgebung, sondern spätestens seit Version 3 auch zur einfachen Erstellung von Rich-Client-Anwendungen eignet. Mit EclipseLink steht außerdem eine JPA-2-Implementierung für den objektrelationalen Datenbankzugriff zur Verfügung. Wie in dem Beispiel aufgezeigt wurde, ermöglicht es das Eclipse Komponenten-konzept dabei, dieselbe Datenbankzugriffslogik sowohl im Rich-Client als auch über einen Webservice angebunden auf einem (Applikations-)Server auszuführen. Dieses Single Sourcing reduziert dabei sowohl die Entwicklungsaufwände als auch die Fehleranfälligkeit der Datenbankzugriffsschicht. Zur einfachen Bereitstellung des Webservice-Client und -Servers hat sich dabei Apache CXF bewährt. Dieser Webservice-Stack unterstützt dabei eine Reihe fortgeschrittener WS-* Standards und ermöglicht so wie gezeigt zum Beispiel auch eine Authentisierung und Verschlüsselung der Zugriffe.

Quellen:

[1] Download unter http://www.eclipse.org
[2] http://wiki.apache.org/ws/StackComparison

Kontaktadresse:

Johannes Michler
PROMATIS software GmbH
Pforzheimer Str. 160
D-76275 Ettlingen

Telefon:	+49 (0) 7243 2179 0
Fax:	+49 (0) 7243 2179 99
E-Mail	johannes.michler@promatis.de
Internet:	www.promatis.de