Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 3

Erweiterte Referenzarchitektur für Multi-Plattformhomogenisierung von Datenhaltungsschichten

Autor: B. Klatt (FZI)
Co-Autoren: K. Krogmann (FZI)
Fertiggestellt am: 02. 01. 2013
Schlagworte: Referenzarchitektur, Multi-Plattformhomogenisierung

Projektpartner

GEFÖRDERT VOM

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

Inhalt

Abbildungen

1 Einleitung

Das vorliegende Dokument beschreibt eine Referenzarchitektur für eine homogenisierte Datenhaltung für die Entwicklung einer Multi-Plattformanwendung. Unter einer Multi-Plattformanwendung ist ein Softwaresystem zu verstehen, das verteilt auf unterschiedlichen Teilsystemen mit unterschiedlichen Anforderungen betrieben wird. Generell gibt es zwei Arten von Teilsystemen. Zum einen unterschiedlichste Anwendungen, die von Endbenutzern bedient werden, wie zum Beispiel verschiedene Mobile Apps (z.B. Android und iOs), Desktop Anwendungen oder Web-Anwendungen, die über einen Browser angesprochen werden. Zum anderen gibt es reine Serverseitige Anwendungen, die eine zentrale Datenverwaltung- und -verarbeitung übernehmen. In solchen komplexen, verteilten Anwendungen, tauschen die unterschiedlichen Endbenutzeranwendungen (Client-Anwendungen) Daten mit den zentralen Serveranwendungen aus, was diverse Anforderungen an die Koordination der Datenverarbeitung und des Datenaustauschs verlangt.

Die vorgestellte Referenzarchitektur spiegelt den aktuellen Stand der Technik bei der Entwicklung solcher Softwaresysteme wider und berücksichtigt dabei die Anforderungen, die zu Beginn des MOHITO Projektes von den Projektpartnern identifiziert und in dem Dokument „AP2 Anwendungsszenarios und Anforderungsanalyse“ festgehalten wurden.

Die Referenzarchitektur beschreibt die Grundlage für die Entwicklung und Planung im Rahmen des Projektes und liefert die Basis für weitere Designentscheidungen.
Gegenüber der initialen Referenzarchitektur (L3.1) beinhaltet die hier vorliegende, erweiterte Referenzarchitektur vor allem die Trennung zwischen Plattformspezifischen Lösungskomponenten, allgemein nutzbaren, technischen MOHITO Framework Komponenten und fachlichen Lösungskomponenten, die mit den MOHITO Werkzeugen modelliert und generiert werden.
Das vorliegende Dokument wurde bewusst als Fortführung der initialen Dokumentation der Referenzarchitektur geschrieben, so dass es in sich vollständig ist.
Die Architektur stellt einen fertigen, in sich kompletten Stand dar. Sollten neue Erkenntnisse im Laufe des Projektes gewonnen werden, so wird die Architektur auch im Laufe des Projektes weiterentwickelt.

2 Multi-Plattform Systeme

Unter einem Multi-Plattform System ist generell ein Software-System zu verstehen, dessen Bestandteile verteilt auf unterschiedlichen Hardwaresystemen und/oder mit unterschiedlichen Betriebssystemen betrieben werden. Wichtig sind hierbei auch die unterschiedlichen Rollen der Teilsysteme innerhalb des Gesamtsystems. Hierbei ist vor allem zwischen mobilen und nicht mobilen Endanwendersystemen und zentralen Serversystemen zu unterscheiden, die in der Regel räumlich voneinander getrennt sind.

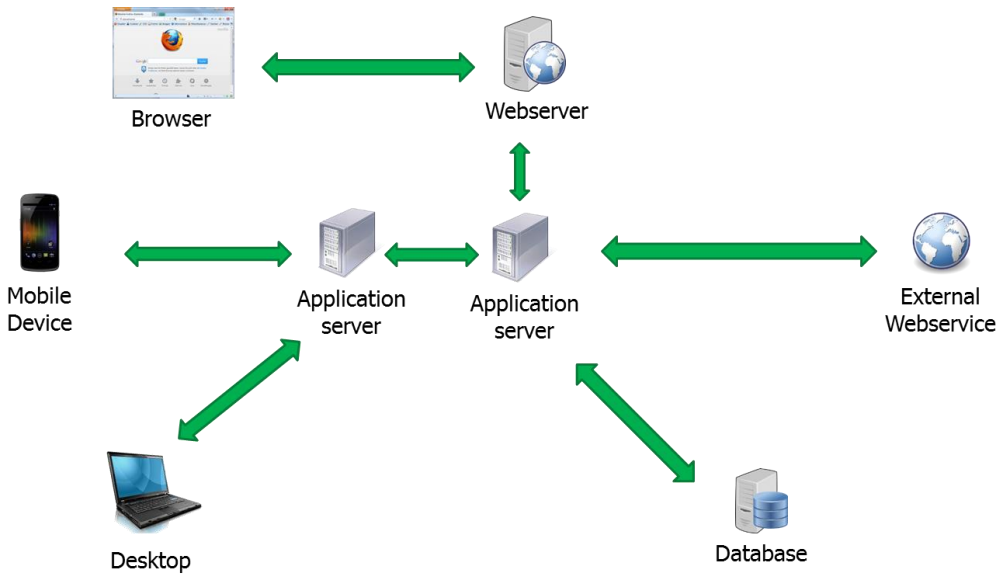

Abbildung 1 Konzeptuelle Bestandteile des Multi-Plattform Systems

Abbildung 1 zeigt die konzeptuellen Bestandteile eines Multi-Plattformsystems, wie es im Rahmen des MOHITO-Projekts behandelt wird. Zwischen den gezeigten Bestandteilen besteht eine Netzwerkverbindung, die entweder im lokalen Rechenzentrum und Unternehmen, oder über das Internet oder eine andere externe Infrastruktur, einschließlich Funkverbindungen, stattfindet.
Die Application- und Database-Server in der Abbildung stellen eine zentrale Infrastruktur dar, die Daten speichert, verarbeitet und für die verbundenen Client-Systeme zur Verfügung stellt. Für diese zentrale Rolle sind Fähigkeiten zur Koordination und intelligenten Vorverarbeitung und Bereitstellung der Daten notwendig.
Mobile Geräte können beliebig viele Endgeräte mit unterschiedlichen Betriebssystemen, wie Apples iOS, Googles Android oder Microsofts Windows Phone sein. Wichtig hierbei ist, dass unterschiedliche Geräte mit unterschiedlichen Fähigkeiten (z.B. Kamera vorhanden oder nicht) und unterschiedlichen Anwendungsplattformen (z.B. Android SDK versus Apple

xcode) bedient werden müssen. Hinzu kommt, dass die Geräte selbst in der Regel über eine unbeständige Internetverbindung zu dem zentralen Serversystem verfügen und somit Anforderungen an Offline-Fähigkeit und den Umgang mit Systemabbrüchen bestehen, die sich je nach Gesamtsystem unterschiedlich ausprägen. Zusätzlich muss die Art der Datenhaltung konfigurierbar sein, um sich konkreten Projektanforderungen anpassen zu können. Dies resultiert in entsprechenden Variationspunkten der Referenzarchitektur.
Bei den Desktops handelt es sich ebenfalls um beliebig viele Client-Systeme, die jedoch auf einem deutlich leistungsfähigeren Arbeitsplatzrechner betrieben werden und in der Regel über eine beständige und zuverlässige Netzwerkverbindung verfügen. Unter solchen Systemen werden typischerweise sogenannte Rich-Client-Anwendungen eingeordnet, die aus einer zu installierenden Software bestehen und viele Freiheiten bei der Entwicklung bieten.
Im Gegensatz zu den Desktop-Anwendungen stehen Web-Anwendungen, die über einen Internetbrowser bedient werden („Browser“ in Abbildung 1). Hier können wieder beliebig viele Clients beteiligt sein, die ebenfalls potentiell über das Internet mit dem zentralen Gesamtsystem verbunden sind. Sie werden von einem speziellen Webserver bedient, der die Web-Anwendung betreibt, die Browser-fähige Inhalte an den selbigen ausliefert und dessen Anfragen entgegennimmt und verarbeitet. Der Webserver selbst, bzw. die darin betriebene Web-Anwendung kommuniziert ihrerseits wieder mit dem zentralen Serversystem.
Zu guter Letzt enthält Abbildung 1 noch einen externen Webservice mit dem das zentrale Server-System kommuniziert. Hierbei handelt es sich um einen oder mehrere externe Dienste, die die Client-Anwendungen in der Regel nicht explizit wahrnehmen. Sie werden im Hintergrund genutzt um Funktionalitäten auszulagern und sie nicht als Teil des eigenen Systems realisieren zu müssen.
Generell sind alle beschriebenen Bestandteile des Gesamtsystems als optional anzusehen. Die in MOHITO geplanten Ergebnisse sind prinzipiell auch so einsetzbar, dass sich bereits bei der Entwicklung einzelner, für sich existierender Teilsysteme ein Vorteil bietet. Nichtsdestotrotz steckt die maßgebliche Komplexität der in MOHITO behandelten Szenarien in dem Zusammenspiel der unterschiedlichen Client-Anwendungen und dem zentralen Server-System. Allen voran das Zusammenspiel zwischen den Mobilen Clients und dem zentralen Serversystem inklusive Ihrer unzuverlässigen Datenverbindungen und der Diversität ihrer Betriebssysteme und Hardwareausstattung.
Der Fokus im MOHITO-Projekt liegt auf Clients für mobile Endgeräte und hier primär für Android und iOS und auf server-seitige Anwendungen.

3 Referenzarchitektur

3.1 MOHITO Programmier-Schnittstellen (APIs)

Das Hauptziel von MOHITO ist die einheitliche und zentrale modellgetriebene Entwicklung des Datenmodells einer Anwendung und die Erzeugung von Programmierschnittstellen, sogenannten „Application Programming Interfaces“ (API), die von einem Entwickler bei der Erstellung eines Softwaresystems genutzt werden können.

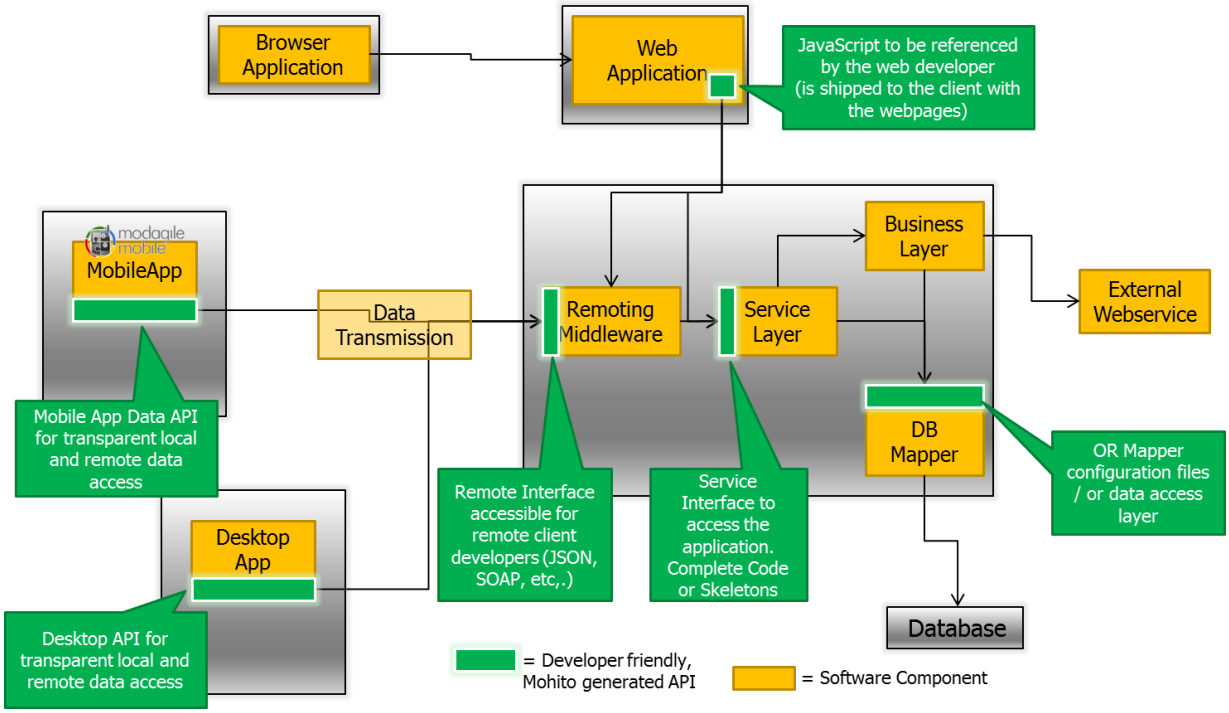

Abbildung 2 Mögliche MOHITO Datenzugriffs APIs in der Referenzarchitektur

Abbildung 2 zeigt eine erweiterte Darstellung der konzeptuellen Komponenten der Referenzarchitektur und die darin befindlichen Datenzugriffs-APIs, die potentiell durch MOHITO realisiert werden können.

Im Vergleich zu der Darstellung in Abbildung 1 ist vor allem zu bemerken, dass sich gerade auf Server-Seite das System in mehrere Teilkomponenten aufgliedert, die jeweils über eigene APIs verfügen. Die Aufteilung auf Teilkomponenten realisiert die Flexibilität zur Anpassung der Referenzarchitektur an Projektanforderungen. Beispielsweise können in der Referenzimplementierung Komponenten für konkrete Realisierungen entfernt werden.

Die Remoting Middleware stellt die Infrastruktur für die Kommunikation mit verteilten Komponenten zur Verfügung. Dies kann sowohl über die Data Transmission-Verbindung mit MobileApps und Desktops, als auch über eine direkte Verbindung mit einem im gleichen Rechenzentrum betriebenen Webserver beziehungsweise der darin betriebenen Web-Anwendung (WebApplication) der Fall sein. Die hier bereitgestellte API dient den Clients als

Zugriffsschnittstelle auf die serverseitige Anwendung. Für die Remoting Middleware selbst existieren heutzutage ausgereifte Lösungen sowohl im OpenSource als auch im kommerziellen Umfeld, die in der Architektur integriert werden.
Die Data Transmission-Verbindung stellt in diesem Fall eine Kommunikation über ein Übertragungsprotokoll und ein Übertragungsformat zur Verfügung. Typische Beispiele hierfür sind das Hyper Text Transfer Protocol (HTTP) (1) als Protokolle, SOAP (Simple Object Access Protocol) (2) oder JavaScript Object Notation (JSON) (3) als Datenformate und Representational State Transfer (REST) als Architekturstil (4).
Der Service Layer stellt den ankommenden Anfragen, sei es durch die Remoting Middleware oder direkt von der Web-Anwendung, Systemfunktionalitäten in Form einer angebotenen API zur Verfügung. Dabei kann er aufgrund von vordefinierten oder manuell implementierten Regeln entscheiden, ob ein Aufruf direkt durch einen Datenbankzugriff über die Database Mapper-Komponente (DB Mapper) bedient oder an die Business Layer-Komponente weitergereicht werden sollte.
Die Business Layer-Komponente implementiert wie der Name sagt die Anwendungsspezifische Geschäftslogik. Daher gibt es an dieser Stelle auch keine durch MOHITO unterstützte API.
Der DB Mapper stellt ein Mapping zwischen der objekt-orientierten Anwendungswelt und der relationalen Datenbank Welt zur Verfügung. Hierfür gibt es sowohl im OpenSource als auch im kommerziellen Umfeld eine Vielzahl ausgereifter Produkte. Sie werden in der Regel durch Konfigurationsdateien gesteuert und stellen daraufhin selbst Programmierschnittstellen und qualitative Fähigkeiten zur Verfügung.
Externe Webservices besitzen ebenso wie die Business Layer-Komponente eine eigene, individuelle Geschäftslogik und Schnittstelle. Sie werden daher von der Business Layer-Komponente direkt angesprochen und die entsprechenden Aufrufe individuell programmiert.

3.2 Zuordnung von Fähigkeiten zu APIs

Verteilte Multi-Plattformsysteme, wie sie im Rahmen von MOHITO betrachtet und unterstützt werden zeichnen sich, neben den diversen zu unterstützenden Plattformen, maßgeblich durch ihre verteilte Kommunikation und ihre Anforderungen an die Übertragung und Speicherung von Daten aus.

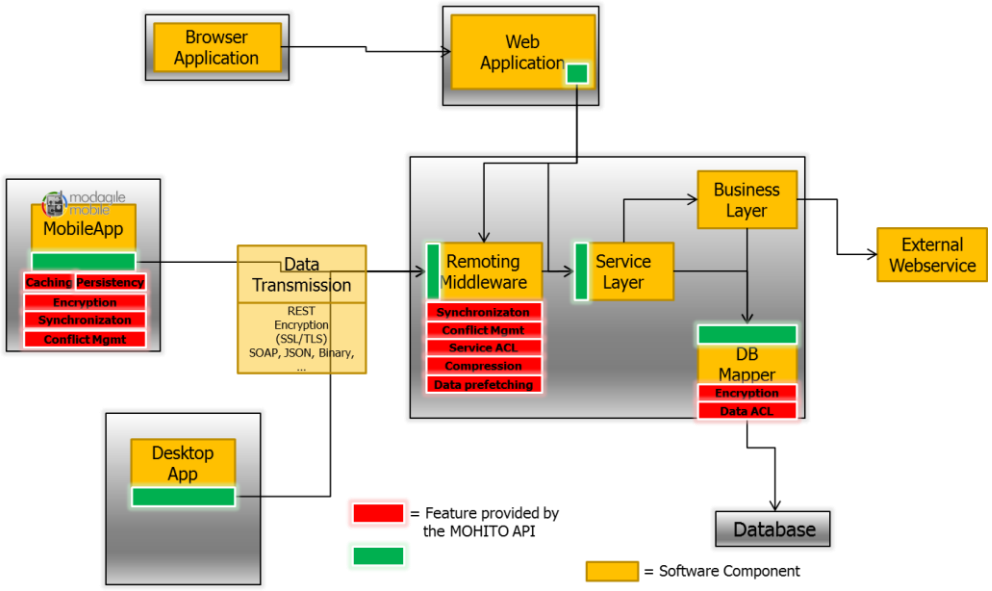

Abbildung 3 Datenmanagement-Fähigkeiten zugeordnet zu Systemkomponenten

Abbildung 3 zeigt die Zuordnung von speziellen Datenmanagement-Fähigkeiten, wie sie während der Anforderungserhebung identifiziert und in dem Dokument „AP2 Anwendungsszenarios und Anforderungsanalyse“ dokumentiert wurden.

Auf dem mobilen Endgerät ist es wichtig, dass die MobileApp von der API die Möglichkeit geboten bekommt Daten für die Offlinefähigkeit dauerhaft zu persistieren und/oder ein zeitlich begrenztes Caching durchzuführen. Bei sicherheitsrelevanten Anwendungen muss dies zudem mit einer Verschlüsselung (Encryption) einhergehen. Da es sich um eine Remote-Anwendung handelt, ist außerdem eine Synchronisierung der Daten mit den server-seitigen Systemteilen und in diesem Zuge gegebenenfalls ein Konfliktmanagement notwendig, falls Daten auf mehreren Teilsystemen parallel geändert wurden.

Seitens der Remoting Middleware sind ebenfalls Funktionalitäten zur Synchronisierung und zum Konfliktmanagement notwendig. Darüber hinaus benötigen viele Anwendungen eine Datenkompression zur Reduktion der zu übertragenden Datenmenge, sowie ein intelligentes Prefetching, das bereits im Voraus nicht nur die explizit angefragten Daten, sondern auch

zusätzlich relevante und gegebenenfalls später im Offline Betrieb des mobilen Clients notwendige Daten mit überträgt. Außerdem wird je nach Anwendung eine Zugriffssteuerung, auch als Access Control List (ACL) bezeichnet, speziell für die Remote Services notwendig. Diese ACL steuert, ob eine client-seitige Anwendung, bzw. der damit arbeitende Benutzer überhaupt einen Service nutzen darf.

Zu guter Letzt muss auch in Schnittstellen des Database Mappers wieder eine Access Control List (ACL), diesmal jedoch speziell für die Daten, vorgesehen sein. Hierdurch wird beispielsweise gesteuert, welche Daten für den anfragenden Nutzer überhaupt geladen oder manipuliert werden dürfen. Ebenso kann eine Verschlüsselung (Encryption) je nach Art der Datenspeicherung notwendig sein. Beide Funktionalitäten werden möglicherweise bereits durch das verwendete DB Mapper Produkt angeboten und müssen nur durch die erzeugte Konfigurationsdatei aktiviert werden.

Wie in Abbildung 3 ersichtlich ist, sieht die Referenzarchitektur die Integrationsmöglichkeit mit Modagile Mobile-generierten Apps für mobile Endgeräte vor. Bei dieser Integration werden die benutzernahen Elemente einer mobilen App durch das Modagile Mobile-Framework[1] erzeugt.

3.3 Komponenten-Architektur

Bisher wurde die Referenzarchitektur aus einer konzeptionellen Sicht betrachtet. Verfeinert man diese in Software-Komponenten wird das Zusammenspiel zwischen MOHITO Komponenten, integrierten Produkten (3rd Party Products) und individueller Implementierung noch deutlicher.

[1] Siehe www.modagile-mobile.de

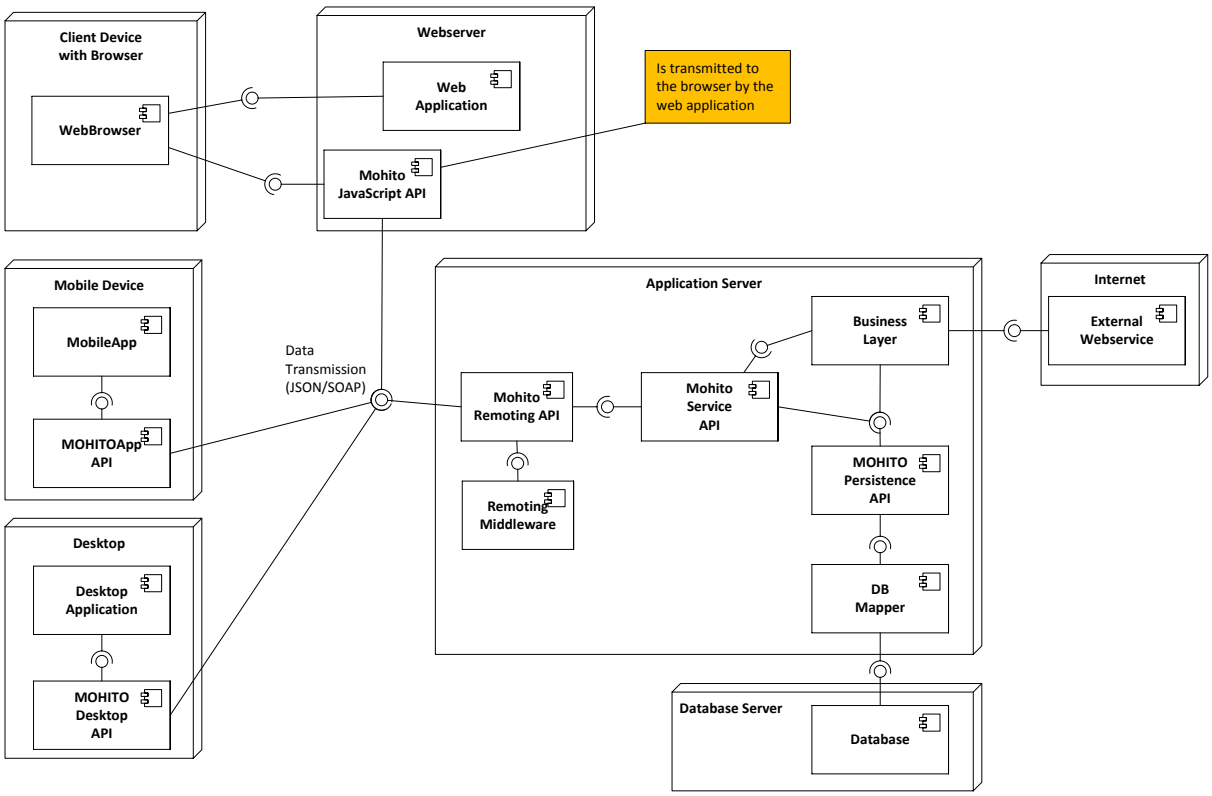

Abbildung 4 Software-Komponenten-Sicht der Referenzarchitektur

Abbildung 4 zeigt eine Sicht der Software-Komponenten aus der MOHITO Referenzarchitektur.

Bei den MobileApps und Desktop Applikationen besteht eine klare Trennung zwischen den client-seitigen Anwendung selbst und den MOHITO-APIs, die von den jeweiligen Anwendungsentwicklern nur über eine explizite Schnittstelle der API angesprochen werden. Diese Bibliotheken kapseln dann vollständig die in Kapitel 3.2 beschriebenen Fähigkeiten sowie die Kommunikation mit den server-seitigen Systemteilen.

Die WebApplication bindet die klar getrennte MOHITO JavaScript-API ein, die wie für JavaScript üblich an den Browser übertragen und dort ausgeführt wird. Hierdurch wird die Kommunikation mit dem server-seitigen System für den Entwickler nahezu transparent geregelt.

Auch server-seitig ist die klare Trennung zwischen der MOHITO Remoting API-Komponente, die die Remoting Schnittstelle und die entsprechenden Fähigkeiten steuert, und der dafür eingebundenen Remoting Middleware Komponente zu erkennen.

Gleiches gilt auch für den DB Mapper-Teil. Zum einen gibt es die API, die klar durch Konfigurationsdateien, die durch MOHITO erzeugt wurden, spezifiziert und gegebenenfalls durch die MOHITO Plattform bereitgestellt werden. Zum anderen wird ein 3rd Party DB Mapper Produkt als Komponente integriert.

4 MOHITO API Muster

Alle Programmierschnittstellen und Bibliotheken, die dem Entwickler auf Basis seiner fachlichen Modellierung zur Verfügung gestellt werden, folgen einem einheitlichen Konzept, auch bezeichnet als *MOHITO API Muster*. Dieses Muster erlaubt es für die jeweiligen qualitativen und funktionalen Anforderungen des Entwicklers eine schlanke, anwendungsspezifische Schnittstelle zu generieren und technische Details, wie die Integration von existierenden Plattformkomponenten oder Framework Code für ihn transparent zu kapseln.

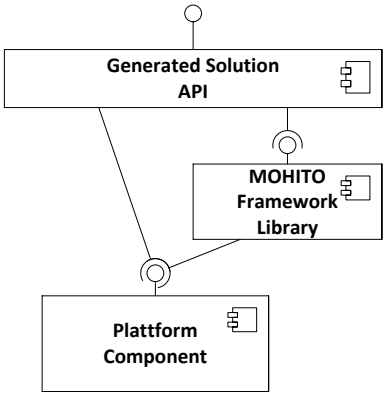

Abbildung 5 MOHITO API Muster

Abbildung 5 zeigt die involvierten Komponenten. Hierbei stellt die **„Generated Solution API“** die anwendungsspezifische Schnittstelle für den Entwickler bereit, die sich an seiner fachlichen Modellierung orientiert. Sie stellt ihm die notwendigen Datentypen und Operationen zur Verfügung um entsprechende Datenobjekte zu erzeugen und zu verarbeiten. Sie wird vollständig generiert. Zum einen orientiert sie sich an der Modellierung des Entwicklers, in der er die Datentypen und ihre Verarbeitung innerhalb der gesamten, verteilten Anwendung spezifiziert. Zum anderen orientiert sich die generierte API an gewissen Standards der MOHITO Werkzeugkette, um homogenisiert auf allen bedienten Plattformen eine einheitliche Schnittstelle anzubieten. Dies reicht von einer einheitlichen Methoden-Benennung, sowie deren Aufruf Konzept bis hin zu verfügbaren Schnittstellenfähigkeiten. Hierdurch wird die Anforderung bedient, dass Entwickler einheitliche Schnittstellen auf allen Plattformen verwenden können.

„Plattform Component“ repräsentiert eine oder mehrere, existierende Plattform-spezifische Komponenten, die bereits einen Teil oder die gesamte, geforderte Funktionalität für die entsprechende Plattform realisieren. Solche Komponenten werden durch die API gekapselt und dem Entwickler einheitlich zur Verfügung gestellt. Diese Kapselung ist vor allem dann maßgeblich, wenn auf unterschiedlichen Plattformen Komponenten für die gleiche Funktionalität bereitstehen, sie aber sehr unterschiedliche Schnittstellen bereitstellen und von dem Ent-

wickler einen sehr hohen Einarbeitungsaufwand sowie auch langfristig ständiges Umdenken verlangen würden.

Die **„MOHITO Framework Library“** beinhaltet technischen Code, der die Verwendung der „Plattform Component“, unabhängig von den anwendungsspezifischen fachlichen Datentypen ermöglicht. Die Library stellt vor allem anwendungsunabhängigen Code zur Verfügung, damit dieser nicht immer wieder neu generiert werden muss. Außerdem stellt er wenn nötig Funktionalität zur Verfügung, die nicht durch eine plattformspezifische Komponente angeboten werden.

Das Verhältnis zwischen MOHITO Framework Library und Plattform Component ist hierbei komplett uneingeschränkt. Wenn bereits eine Platform Component existiert, die alle benötigten Funktionalitäten bereit stellt, kann hier möglicherweise komplett ohne MOHITO Framework Library gearbeitet werden. Das andere Extrem hierzu ist, dass es keine passende Plattform Component gibt und sämtliche Funktionalität durch eine MOHITO Framework Library realisiert wird.

Durch das MOHITO API Muster wird es ebenso möglich die gleiche Schnittstelle, beispielsweise für die Server-seitige MOHITO Persistence API anzubieten und einmal mit dem freien Hibernate OR Mapper und einmal mit der CAS Open Plattform bereitzustellen. Zusammenfassung
Die MOHITO Referenzarchitektur stellt eine Grundlage für die Entwicklung verteilter Multi-Plattformanwendungen bereit. Die Aufteilungsaspekte und die in die Architektur integrierten Komponenten berücksichtigen dabei bereits die Anforderungen der Partner hinsichtlich technischer Vorarbeiten und technischer Rahmenbedingungen. Sie erlaubt eine nahtlose Integration von Architekturausprägungen der Referenzarchitektur für die bestehende Infrastruktur der Partner und stellt eine weitgehende Domänenunabhängigkeit sicher (bspw. keine Einschränkung auf reine CRM-Anwendungen).

Die MOHITO Referenzarchitektur ist modular ausgelegt, so dass durch das Hinzufügen oder Entfernen von Komponenten insbesondere die Anforderungen von CAS und B2M für spätere Produktentwicklungen erfüllt werden können. Die Integration mit Bestandskomponenten und verwandten Frameworks wie Modagile Mobile, MML und CAS Open erlaubt es, auf einem aktuellen technischen Stand aufzusetzen und eine Konzentration auf die plattformübergreifende Integrationsleistung für mehrere Domänen.

4.1 Caching Beispiel

Zur Veranschaulichung des Umgangs mit generierten, MOHITO Framework und plattformspezifischen Bestandteilen soll das Caching auf dem Mobilen Client mittels einer lokalen SQLite Datenbank dienen.

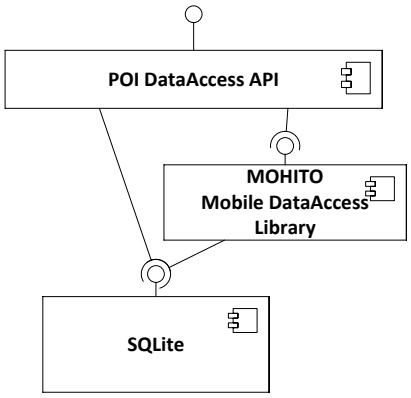

Abbildung 6 MOHITO Mobile Data Access

Abbildung 6 zeigt die Ausprägung des Musters für die DataAccess API, die dem Entwickler für das Speichern und Laden von Daten innerhalb seiner mobilen App erzeugt wird. Das konkrete Beispiel stammt aus der b2m Referenzanwendung „Here I AM“ und stellt einen Datenzugriff inklusive Caching Funktionalität speziell für Points of Interests (POI) dar. POIs repräsentieren Ortsangaben, die relevante Punkte aus Sicht des Benutzers beschreiben.

Die Komponentensicht zeigt, dass die DataAccess API intern eine plattformspezifische SQLite Datenbank mit der entsprechenden Android-Bibliothek nutzt („SQLite“). Außerdem wird von dem MOHITO Framework eine Bibliothek zur Verfügung gestellt („MOHITO Mobile DataAccess Library“), die fachlich unabhängige Funktionalitäten für Datenzugriff und Caching bereitstellt. Nur die „POI DataAccess API“ ist fachlich geprägt und wird daher dem Entwickler Anwendungsspezifisch, auf Basis seiner Modellierung generiert.

Abbildung 7 zeigt das Klassendiagramm zur „MOHITO Mobile DataAccess Library“ für Android. Die Klasse *DataAccess<T>* stellt über ihre Schnittstelle *IDataAccess<T>* generische Funktionalität für den Datenzugriff bereit. Diese Schnittstelle, bzw. die Implementierung, Kapselt intern das Management des Caches über die hier ebenfalls generische Klasse *Cache<T>* sowie den allgemeinen SQLite spezifischen *DatabaseHelper*.

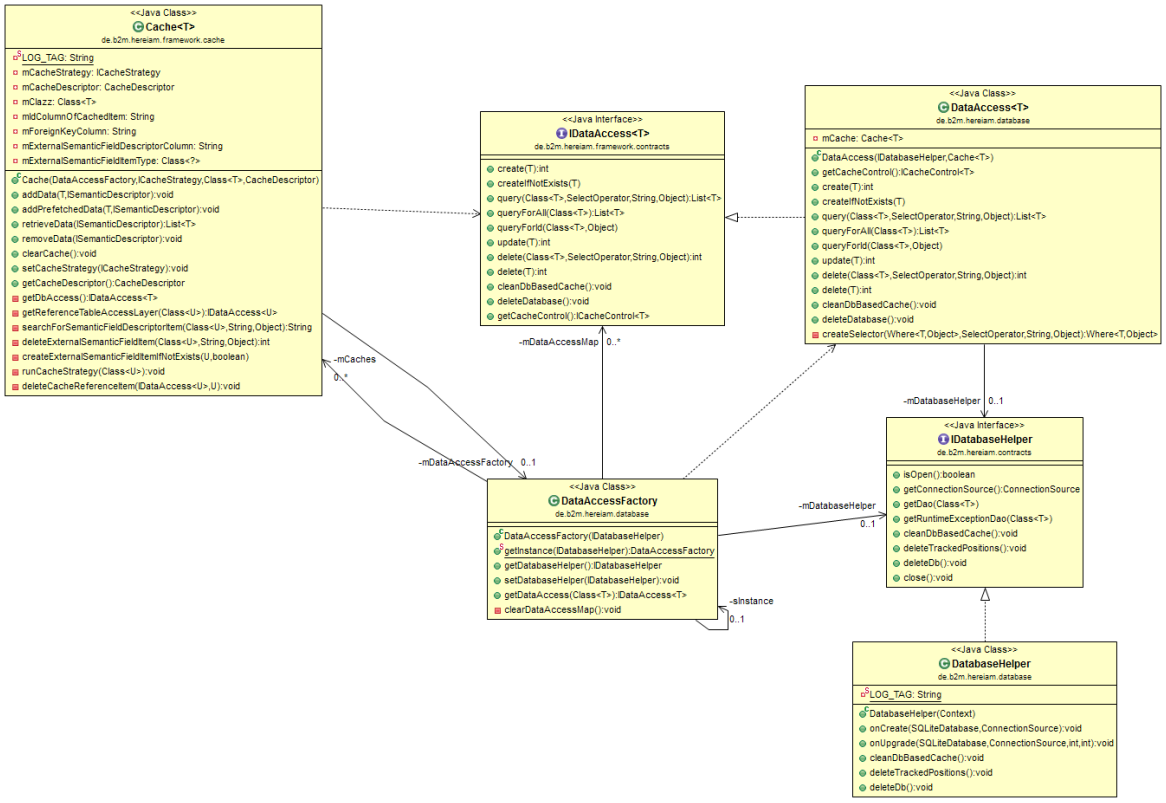

Abbildung 7 MOHITO Mobile Caching Library – Klassendiagramm

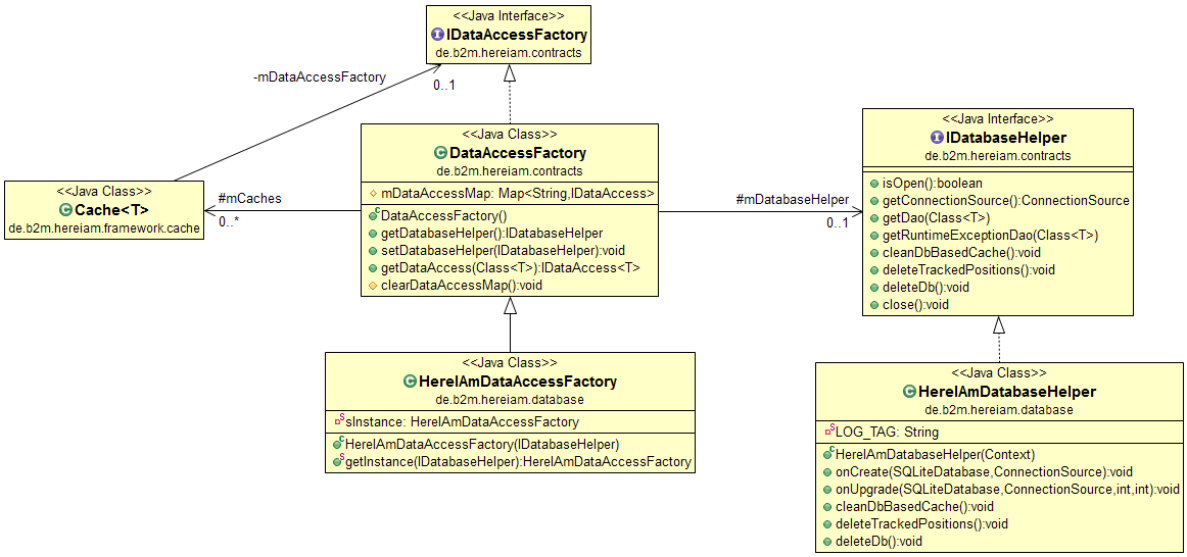

Abbildung 8 HereIAm DataAccess Klassendiagramm

Abbildung 8 zeigt das Klassendiagramm für die HereIAm DataAccess API und ihre Verbindung zur MOHITO DataAccess Library. Es gibt einen Anwendungsspezifischen DatabaseHelper (*HereIAmDatabaseHelper*), der beispielsweise das initialisieren und Management der Datenbank, spezifisch für die HereIAm Anwendungsdaten realisiert. Darüber hin-

aus gibt es eine Anwendungsspezifische *DataAccessFactory* (*HereIAmDataAccessFactory*), die sich um die Initialisierung der Datenbankschicht kümmert und *DataAccess* Klassen (*DataAccess<T>*) für die Anwendungsspezifischen DatenTypen bereitstellt (z.B. *DataAccess<POI>*). Die Anwendungsspezifische Cache Infrastruktur wird, wie in Abbildung 8 gezeigt ebenfalls durch die *DataAccessFactory* gekapselt. Wenn der Entwickler beispielsweise eine POI spezifische *DataAccess* Klasse von der Factory erstellen lässt, so verwendet diese intern bereits einen POI spezifischen Cache (*Cache<POI>*), so dass sich der Entwickler um das Caching selbst nicht mehr kümmern muss.

5 Literaturverzeichnis

1. w3c. HTTP Specification. [Online] [Zitat vom: 30. 10 2012.] http://www.w3.org/Protocols/rfc2616/rfc2616.html.

2. —. SOAP Specification. [Online] [Zitat vom: 30. 10 2012.] http://www.w3.org/TR/soap/.

3. ECMAScript Language Specification. [Online] [Zitat vom: 30. 10 2012.] http://www.json.org/.

4. Foundation, Wikipedia. REST Wikipedia Artikel. [Online] [Zitat vom: 30. 10 2012.] http://en.wikipedia.org/wiki/Representational_state_transfer.

