WBA 2: Phase 2

Sommersemester 2013

Max Bobisch 11070440

Robert Schumann 11070745

Sticker-App

Dokumentation

Gliederung

1. Vorstellung der Idee

Das System ermöglicht, abfotografierte Sticker zu erkennen als entweder bereits vorhandene Sticker oder als neu anzulegende Sticker.

Zu einem Sticker wird eine Ressource angelegt, die z.B. folgende Daten enthält: Fundort (GPS-Daten, z.B. mit Verwendung der google maps-API), Aufnahmezeit, Nutzer, Stickerhersteller, Bezeichnung, Preis, Anbieter (Shops), Kommentare und etwaige quantitative Angaben (Häufigkeit der/s Sticker/s im System, in einer bestimmten Region, etc.).

Somit ergeben sich für das System verschiedene Stakeholder. Hier die primären: Die Endnutzer, Stickeranbieter (Shops, die die Produkte anbieten) und deren Hersteller.

News der Stakeholder können auf Wunsch abonniert werden. Asynchrone Kommunikation ist hierbei gegeben in Form von publish-Möglichkeiten seitens der Hersteller (neue Produkte, bestimmte Angebote, …), entsprechend auch seitens der Stickershops (sale, special offers, etc.) und der Kunden (Favoritenlisten, eigens eingestellte unvollständige Datensätze, Stickersammlungen, …). Jeder kann dabei jedem Nutzer oder auch einzelnen Stickern folgen (subscribe), so ist er/sie stets über alle Sticker-Informationen informiert, für die er sich interessiert.

2. Aufgaben zum Praktikum

1. Projektbezogenes XML Schema / Schemata

- Die Planung und Konzeption für das Projekt nötiger valider XML Schemata soll durchgeführt werden. Hinsichtlich der späteren Verwendung von JAXB sollte hier in jedem Fall darauf geachtet werden, dass die XML Schemata valide sind.

Als Ausgangspunkt für die Struktur der XML Schemata nutzen wir die Kernideen der Anwendung:

User, Sticker, Sammlungen von Stickern, Angebote von Usern (shop) und Warenkörbe.

Diese bilden die Primärressourcen. (Auch wenn aus strenger REST-Sicht alle Ressourcen einfach nur Ressourcen sind dient es hier zum Verständnis.)

Alle anderen Daten werden nicht als eigenständige Ressourcen angelegt.

Struktur der xml-Schemata (xsd)

Legende:

Ressource

- Elementname (Typ)
  [Attributname (Typ)]
  optionen / restrictions

User [UserID (int), Self(anyURI)]

- Nickname (String)
- Gender (String {male, female, not specified})
- HomeAdress (Adress *optional)*
- ShopAdress (Adress *optional)*
- StickerContainer
- CollectionContainer
- PhotoContainer
- OfferContainer
- ShoppingCartContainer
- Liker
- Follower
- Comments

Typ „**Adress**“ setzt sich aus FamilyName, FirstName, Street, Number, PostalCode, City, Province (*optional*), Country, Telephone (*optional*) und email (*optional*) zusammen (alle vom Typ String, Das Element „email“ beinhaltet zudem eine Restriction zur Prüfung auf Validität der ggf. angegebenen Emailadresse).

Home und- ShopAdress werden nur im Falle einer Transaktion benötigt und sind daher optional.

Die Container werden zum Zeitpunkt der Erstellung erstellt, beinhalten aber noch keine Items, diese können zu einem späteren Zeitpunkt hinzugefügt werden.
Die Container der einzelnen Ressourcen (Sticker-, Collection-, Photo-, Offer- und ShoppingCartContainer) beinhalten Links zu den Ressourcen die der jeweilige User besitzt, bzw. welche er hochgeladen oder erstellt hat. Ggf. sind diese noch angereichert mit sinnvollen Feldern, die auch in der jeweiligen Ressource vorhanden sind. Wenn der zuständige Service der Ressource auf die verwiesen wird, nicht funktionieren sollte, ist so gewährleistet, dass der Link stets mit wenigstens den minimalen Informationen angereichert ist. [1]
So beinhalten Photo-Verweise außer dem Link zur Ressource noch den Photolink, wo das Photo hinterlegt ist und ggf. den Titel (wenn vorhanden), ShoppingCart-Verweise stellen auch die Anzahl der Artikel, den Gesamtpreis und die Währung dar.
Follower- und Like-Container beinhalten nur die URIs der User, die Ihnen folgen bzw. sie liken. Konsequenterweise wäre hier auch die Darstellung des Usernamens denkbar.
Comments ist ein Container für die Comments zu einem User.
Ein Comment setzt sich aus dem Text (String), der Datetime (String, Zeitpunkt der Erstellung) einem Liker-Container (so kann man auch Comments liken) zusammen und ist außerdem mit einer eigenen ID, der ID und der URI des Erstellers angereicht.

Sticker [StickerID (int), Self(anyURI), OwnerID (int), Owner(anyURI)]

- Title (String *optional*)
- Description (String *optional*)
- Datetime (dateTime)
- RelatedPhotos
- Liker
- Follower
- Comments

Sticker bestehen bei der Erstellung erstmal nur aus dem Datetimeelement (Zeitpunkt des Anlegens der Ressource) und können später noch mit weiteren Informationen angereichert werden, vom User, der die Ressource erstellt hat aber vor allem auch von anderen Usern.
RelatedPhotos ist ein Container, der mit Links zu Photos, die diesen Sticker zeigen, gefüllt werden kann. Auch hier wäre konsequenterweise die zusätzliche Angabe des Photolinks eine gute Designentscheidung gewesen.

Collection [CollectionID (int), Self(anyURI), OwnerID (int), Owner(anyURI)]

- Title (String)
- Description (String *optional*)
- Item (anyURI *unbounded*)
- Liker
- Follower
- Comments

Collections stellen eine Möglichkeit dar, Sticker in Sammlungen zusammenzufassen. Sie beinhalten außer einem Titel und einer Beschreibung eine beliebige Anzahl an Items. Items sind Verweise auf die Sticker, die Teil dieser Collection sind. Dabei ist es möglich, dass ein Sticker Teil mehrerer Collections ist (m:n-Beziehung).

Photo [PhotoID (int), Self(anyURI), OwnerID (int), Owner(anyURI)]

- Title (String)
- Description (String *optional*)
- Datetime (dateTime)
- Photolink (anyURI)
- Location (*optional)*
- RelatedSticker (*optional unbounded*)
- Liker
- Follower
- Comments

Photos tragen in jedem Fall einen Titel, den Zeitpunkt des Uploads, die URI des Photos und wie für fast alle Ressourcen üblich, einen Liker-, Follower- und einen Comments-Container (bloß ShoppingCarts besitzen diese nicht).
Ein Locationelement (Breiten- und Längengrad vom Aufzeichnungsort), Verweise auf verwandte Sticker (einfach Element, Eine Designentscheidung wäre auch hier alternativ gewesen, erstmal einen Container zu kreieren, welcher wiederum die Links beinhaltet) können später hinzugefügt werden.

Offer [Currency(String), OfferID (int), Self(anyURI), OwnerID (int), Owner(anyURI)]

- Title (String)
- Description (String)
- Item (*optional unbounded*)
- Price (decimal)
- Liker
- Follower
- Comments

Angebote oder Offer können User einstellen, wenn sie Sticker oder Sammlungen verkaufen wollen. Auf selbige wird in inem Item-Element verwiesen. Weiter werden Titel, Beschreibung, Preis (mit resctriction int >= 0.0) und die Währung aufgezeichnet.

ShoppingCart [State(String {created, received, accepted, rejected, cancelled, fultilled}), Currency(String), OfferID (int), Self(anyURI), OwnerID (int), Owner(anyURI)]

- Products
- ShipTo (Adress)
- Price (decimal)
- Payment (String {Bank Account, Credit Card, Paypal, n.a.})

Ein ShoppingCart trägt als zusätzliches Attribut (gegenüber anderen Ressourcen) noch State, welcher den Zustand des Einkaufswagens widerspiegelt. Wurde die Bestellung angenommen, bearbeitet, ausgeliefert oder storniert? Dies wird hier gespeichert. Später kann man dahingehend die Ressourcen über die Funktionalität des Systems auch per Queryparam abfragen.
Products ist ein Container für Items, welche Links auf Angebote sind, die Teil dieser Komplettbestellung sind.
Außerdem werden noch die Auslieferungsadresse, der Gesamtpreis und die Art der Zahlung festgehalten. Für einen Einsatz im echten Leben wären für den Zahlungsverkehr noch weitere Informationen festzuhalten (Kontonummer, Kreditkartennummer, paypaladresse). Für die Zahlung als solche müsste noch eine sichere Anbindung an Drittanbieter (also insbesondere Banken bzw. paypal) von Nöten.

2. Ressourcen und die Semantik der HTTP-Operationen für das Projekt

- Eingangs soll eine theoretische Auseinandersetzung mit HTTP-Operationen und Ressourcen im Kontext RESTful Webservices erfolgen. Das erworbene Wissen ist Grundlage um die nächsten Schritte erfolgreich absolvieren zu können.

Ressourcen

Als Ressourcen werden irgendwelche Daten(-sätze) oder ganz allgemein Dinge bezeichnet, die (für den Datenverkehr zwischen Anwendungen) identifizierbar sind, also zum Beispiel im Web mittels einer URI(Uniform Resource Identifier) adressierbar ist. Sie werden meist als xml/json oder ähnlichen kontextfreien Sprachen zur Datenrepräsentation bereitgestellt. Auf jeden Fall muss es aber Repräsentationen geben, die die Ressource der Außenwelt bereit stellt und über die sie bearbeitet wird. Wir sehen nie die Ressource selbst, sondern immer nur ihre Repräsentation(en). Tatsächlich kann eine Ressource sogar so definiert werden: als eine durch eine gemeinsame ID zusammengehaltene Menge von Repräsentationen. Wird jetzt zum Beispiel ein http GET mit 'Accept:application/xml' auf eine solche Ressource angewendet, so wird eine xml-Entität dieser Ressource zurückgegeben, nicht die Ressource selbst. Mit PUT wird sie aktualisiert mit DELETE gelöscht. Reduziert man die Ideen hinter REST auf einen einzigen Satz, ergibt sich „Eine eigene URI für jedes

Informationselement.“[1]

HTTP-Operationen

Auseinandersetzung mit den Operationen:
Es gibt verschiedene HTTP-Operationen,namentlich GET, HEAD, PUT, POST, DELETE, OPTIONS, TRACE und CONNECT. Außerdem ist es möglich selbst VERBEN zu definieren. Davon ist aus REST-Sicht allerdings abzuraten, da diese nur in einem begrenzten Umfeld(dort, wo sie bekannt gemacht wurden) verwendet werden, bzw. (richtig) interpretiert werden können.

GET: holt die Informationen, die durch eine URI spezifiziert werden in Form einer Entity (Repräsentation) ab. Also ein lesender Zugriff auf die Daten.
HEAD: ähnlich dem GET, allerdings werden nur die METADATEN (also vor allem die Header) zurückgeliefert.
PUT: Aktualisierung einer bestehenden Ressource, oder Erzeugung dieser, falls sie noch nicht existiert.
POST: Erzeugung einer Ressource unter Angabe der für das Anlegen zuständigen Ressource (Häufig Einträge in Listen).
DELETE: Löschen einer Ressource.
OPTIONS: Liefert Metadaten über eine Ressource, unter anderem Angaben über die unterstützten Methoden.
TRACE: Dient zur Diagnose von HTTP-Verbindungen.
CONNECT: Dient der Initialisierung einer Ende-zu-Ende-Verbindung bei der Verwendung von SSL durch einen Proxy.

Die HTTP-Operationen unterscheiden sich in ihrer Sicherheit und in ihrer Idempotenz.
Die Operationen GET, PUT, POST und DELETE werden als relevante Operationen genauer betrachtet.
Sicherheit bezeichnet hier, dass eine Operation keine Seiteneffekte hat, das bedeutet, eine Operation, die sicher ist kann vom Client ausgeführt werden, ohne eine Zustandsänderung anzufordern. Dies bedeutet nicht, dass serverseitig nicht doch Änderungen erfolgen(zum Beispiel Informationen darüber, wie oft eine Ressource angefragt wurde), diese sind aber für den Client unerheblich.
Idempotenz aus Rest-Sicht bedeutet, dass eine Operation mehrfach ausgeführt werden kann, ohne, dass dies ein anderes Ergebnis liefert. So ist es beispielsweise irrelevant, wie oft ein PUT auf die gleiche Ressource angewendet wird. Entweder ist die Ressource schon vorhanden und wird überschrieben, oder sie ist noch nicht vorhanden und wird erzeugt. In beiden Fällen wird das gleiche Ergebnis erzielt.

GET: ist sicher und idempotent.
PUT: ist nicht sicher aber idempotent.
DELETE: ist nicht sicher aber idempotent.
POST: ist weder sicher noch idempotent.

Da POST nicht idempotent ist ergibt sich für die Verwendung ein Problem: Wenn der Client keine Antwort auf seine POST-Operation erhält kann dies mehrere Ursachen haben, die unterschiedliche Vorgehensweisen erfordern. Entweder hat der Server die Anfrage gar nicht erst erhalten, oder bei der Verarbeitung ist ein Fehler aufgetreten und die Ressource wurde nicht angelegt. In diesem Fall müsste der Client die Anfrage erneut senden.
Oder der Client hat lediglich die Antwort nicht erhalten, in diesem Falle muss er nichts tun.
Der Client kann dies aber nicht wissen und riskiert nun, dass entweder keine Verarbeitung stattfindet, oder die Verarbeitung mehrfach ausgeführt wird und zum Beispiel zwei statt einer Ressource angelegt werden (z.B. zweimal die gleiche Bestellung).
Da POST keinerlei Garantien liefert sollten also reine POST vermieden werden, da sie nicht zuverlässig sind.
Alternativen gibt es mehrere. Hier drei häufig angewandte:
-PUT statt POST:
Bei dieser Vorgehensweise wird komplett auf die Verwendung von POST-Operationen verzichtet. Dies funktioniert so, dass das Erstellen der URI dem Client übertragen wird, zum Beispiel über ein UUID-Verfahren, bei dem der Client einen eindeutigen 128-bit-Wert auf eine Basis-URI des Servers abbildet und so eine neue URI erzeugt. Dann wird ein PUT auf diese URI ausgeführt. Da PUT idempotent ist, ist hier die Verlässlichkeit gewährleistet. Der einzige Nachteil ist, dass die Erstellung der URI Sache des Client, und dies nicht elegant ist.
-POST und PUT:
Hier wird weiterhin POST verwendet, allerdings nur, um die URI anzulegen. Die eigentlichen Informationen werden dann per PUT auf diese URI abgebildet. Es wird also mit POST eine leere Ressource erstellt, die dann mit PUT lediglich upgedatet wird. So wird gewährleistet, dass die sensiblen Daten idempotent, nämlich mit PUT übermittelt werden. Zwar wird ein POST verwendet, aber sollte der Client keine Antwort erhalten, so kann er einfach ein neues POST durchführen. Im schlimmsten Fall hat er so eine nicht benötigte leere Ressource erzeugt. Dies ist auch ein Nachteil dieser Vorgehensweise, der es eventuell nötig macht, diese leeren Ressourcen regelmäßig zu löschen. Ein weiterer Nachteil gegenüber der PUT-statt-POST-Variante ist der erhöhte Bedarf an Operationen und eventueller damit einhergehender Performance-Einbußen.
-POE:
POST-One-Exactly ist eine Spezifikation, bei der der Client dem Server mitteilt, dass er POE versteht und von diesem eine Liste der POE unterstützenden URIs zurückerhält.
Ein POST auf eine dieser URIs folgt dann den POE-Prinzipien, das heißt, dass das erste POST wie jedes andere auch behandelt wird. Erhält der Client keine Antwort kann er es noch einmal versuchen. War bereits das erste POST erfolgreich, antwortet der Server mit 405 Method not allowed, andernfalls mit 200 OK. Somit kann auf jede POE-Ressource nur ein POST ausgeführt werden.

- Es sollen projektbezogene Beschreibungen der für das Projekt benötigten

Ressourcen und Operationen inklusive Begründung der Entscheidungen erstellt werden.

Für unser Projekt wählen wir zur Sicherstellung der Verlässlichkeit von POST-Operationen die POST-PUT-Kombination, da wir so sowohl PUT- als auch POST-Operationen verwenden, wie es in der Aufgabe gefordert wird.

Auflistung der möglichen Ressourcen und deren unterstützte Methoden:

Ressource	**URI**	**Methoden**
Liste aller Nutzer	/users/	GET, POST
Einzelner Nutzer	/users/{id}	GET, POST, PUT, DELETE
Liste aller Sticker	/stickers/	GET, POST
Einzelner Sticker	/stickers/{id}	GET, POST, PUT
Liste aller Kommentare	/kommentare/	GET, POST
Einzelner Kommentar	/kommentare/{id}	GET, POST, PUT, DELETE
Liste aller Angebote	/angebote/	GET, POST
Einzelnes Angebot	/angebote/{id}	GET, POST, PUT, DELETE
Liste aller aktiven Angebote	/stickerangebote/?state=aktiv/	GET
Liste aller Stickersammlungen	/stickersammlungen/	GET, POST
Einzelne Stickersammlung	/stickersammlungen/{id}	GET, POST, PUT, DELETE
Liste aller Fotos	/fotos/	GET, POST
Einzelnes Foto	/fotos/{id}	GET, POST, PUT, DELETE
Liste aller Hersteller	/hersteller/	GET, POST
Einzelner Hersteller	/hersteller/{id}	GET, POST, PUT
Liste aller Adressen	/adressen/	GET, POST
Liste aller Lieferadressen	/adressen/lieferadressen/	GET, POST
Einzelne Lieferadresse	/adressen/lieferadressen/{id}	GET, POST, PUT, DELETE
Liste aller Rechnungsadressen	/adressen/rechnungsadressen/	GET, POST
Einzelne Rechnungsadresse	/adressen/rechnungsadressen/{id}	GET, POST, PUT, DELETE
Liste aller Kontaktadressen	/adressen/kontaktadressen/	GET, POST

Einzelne Kontaktadresse	/adressen/kontaktadressen/{id}	GET, POST, PUT, DELETE
Liste aller Vorkasseadressen	/adressen/vorkasseadressen/	GET, POST
Einzelne Vorkasseadresse	/adressen/vorkasseadressen/{id}	GET, POST, PUT, DELETE
Liste aller Bestellungen	/bestellungen/	GET, POST
Einzelne Bestellung	/bestellungen/{id}	GET, POST, PUT
Status der Bestellung received	/bestellungen/?state=received/	GET
Status der Bestellung inoperation	/bestellungen/?state=inoperation/	GET
Status der Bestellung done	/bestellungen/?state=done/	GET
URI-Liste der likes	/likes/	GET, POST
Einzelne Likeliste	/likes/{id}	GET, POST, PUT, DELETE
URI-Liste der followers	/followers/	GET, POST
Einzelne Followerliste	/followers/{id}	GET, POST, PUT, DELETE
Liste der Zahlungsinformationen	/zahlungsinfos/	POST
Liste der Kontoinformationen	/zahlungsinfos/konten/	POST
Einzelne Kontoinformation	/zahlungsinfos/konten/{id}	GET, POST, PUT, DELETE
Liste der Paypalinformationen	/zahlungsinfos/paypal/	POST
Einzelne Paypalinformation	/zahlungsinfos/paypal/{id}	GET, POST, PUT, DELETE
Liste der Kreditkarteninformationen	/zahlungsinfos/kreditkarten/	POST
Einzelne Kreditkarteninformation	/zahlungsinfos/kreditkarten/{id}	GET, POST, PUT, DELETE
Liste der Vorkasseinformationen	/zahlungsinfos/vorkasse/	POST
Einzelne Vorkasseinformation	/zahlungsinfos/vorkasse/{id}	GET, POST, PUT, DELETE

Begründungen:
Wir haben die Informationen, die voraussichtlich persistent sind als eigene, stabile Ressourcen vorgesehen. Der Status der Bestellung wird als Ressource abgebildet, da es sich um eine serverseitige Statusverwaltung handelt und jeder Status als eigene Ressource angelegt wird.
Die Listenressourcen haben wir mit einer logischen ID für die jeweils enthaltenen Ressourcen ergänzt, um die Übersichtlichkeit zu erhöhen. Zum Beispiel beinhaltet die Liste aller User 'Users/' Referenzen zu den einzelnen Usern wie beispielsweise dem User mit der ID 1 als 'Users/1'.

3. RESTful Webservice

Ein RESTful Webservice soll in Java erstellt werden. Dazu gelten folgenden Bedingungen:

- Mindestens zwei Ressourcen müssen implementiert sein

Implementiert:
- Users („application/xml“)
- Stickers („application/xml“)
- Collections („application/xml“)
- Photos („application/xml“)
- Offers („application/xml“)
- ShoppingCarts („application/xml“)
- index („text/html“)

Außer den Repräsentationen der Datenressourcen als XML gibt es auch eine einfache index-Ressource als html-Dokument, in welchem man Links auf Listenrepräsentationen der anderen Ressourcen findet.

- JAXB soll für das marshalling / unmarshalling verwendet werden

JAXB wird verwendet.
Zum Beispiel wenn die jeweiligen XML-Dateien der Ressourcen ausgelesen werden, oder wenn in sie hineingeschrieben wird.

- Die Operationen GET, DELETE und POST müssen implementiert sein

GET: auf alle Listenressourcen möglich, und mittels ID (PathParam) auf Einzelressourcen. Bei den ShoppingCarts besteht zusätzlich die Möglichkeit, einzelne Kategorien (storniert, ausgeliefert, …) per QueryParam abzufragen.

DELETE: über den Parameter ID kann man Einzelressourcen löschen.

POST: Eine Createfunktion wurde für jede Ressource eingebunden, welche mittels POST-Operation angestoßen werden kann. Dabei wird dynamisch die ID

der neuen Ressource ermittelt.
Die Updatefunktionen, die Einzelelemente der Ressourcen manipulieren, steuern per ID als Parameter die jewelige Ressource an.

- Es sollen mindestens einmal PathParams und einmal QueryParams verwendet werden

PathParams: ID

QueryParams: State

4. Konzeption + XMPP Server einrichten

Es handelt sich hierbei um konzeptionellen Meilenstein für die genauere Planung der asynchrone Kommunikation. Dazu sollen folgende Aufgaben erledigt werden:

- Leafs (Topics) sollen für das Projekte definiert werden

Die Topics:

- Users – Listenressource aller Benutzer
- User – Einzelner Benutzer
- Stickers - Listenressource aller Sticker
- Sticker – Einzelner Sticker
- Shoppingcart – Einzelner Warenkorb
- Photos – Listenressource aller Photos
- Photo – Einzelnes Photo
- Offers – Listenressource aller Angebote
- Offer – Einzelnes Angebot
- Collections – Listenressource aller Sammlungen von Stickern
- Collection – Einzelne Sammlung von Stickern

- Wer ist dabei Publisher und wer Subscriber?
- Welche Daten sollen übetragen werden?

- Users
  Publisher: eine neue Benutzerressource wird angelegt
  Subscriber: alle User
  Payload: URI der geänderten Liste der User
- User
  Publisher: eine Benutzerressource wird geändert
  Subscriber: alle an diesem Benutzer interessierten Benutzer
  Payload: URI der geänderten Userressource

- Stickers
  Publisher: eine neue Stickerressource wird angelegt
  Subscriber: alle User
  Payload: URI der geänderten Liste der Sticker
- Sticker
  Publisher: eine Stickerressource wird geändert
  Subscriber: alle an diesem Sticker interessierten Benutzer
  Payload: URI der erstellten Stickerressource
- Shoppingcart
  Publisher: eine Shoppingcartressource wird geändert
  Subscriber: alle an diesem Shoppingcart berechtigten Benutzer Käufer und Verkäufer
  Payload: URI der Warenkorbressource
- Photos
  Publisher: eine neue Photoressource wird angelegt
  Subscriber: alle User
  Payload: URI der geänderten Liste der Photos
- Photo
  Publisher: eine Photoressource wird geändert
  Subscriber: alle an dieser Photo interessierten Benutzer
  Payload: URI des geänderten Photos
- Offers
  Publisher: eine neue Offerressource wird angelegt
  Subscriber: alle an Offers interessierten User
  Payload: URI des geänderten Offers
- Offer
  Publisher: eine Offerressource wird geändert
  Subscriber: alle an diesem Offer interessierten Benutzer
  Payload: URI des geänderten Offers
- Collections
  Publisher: eine neue Collectionressource wird angelegt
  Subscriber: alle interessierten User
  Payload: URI der geänderten Liste der Collections
- Collection
  Publisher: eine Collectionressource wird geändert
  Subscriber: alle an dieser Collectionressource interessierten Benutzer
  Payload: URI des geänderten Collection

- Zusätzlich soll der XMPP Server installiert und konfiguriert werden (sofern auf eigenen Geräten gearbeitet wird)

XMPP Server installiert.

5. XMPP - Client

Es soll ein XMPP Client entwickelt werden, dafür soll folgendes berücksichtigt werden:

- Mittels der Anwendung soll es möglich sein Leafs zu abonnieren, Nachrichten zu empfangen und zu veröffentlichen.

Die Anwendung ermöglicht es Leafs zu abonieren. Hierzu wird die Subscriber.java-Datei verwendet (Menüwahl (s) : subscribe to node).
Um mit der Anwendung Payload zu publishen wird die Publisher.java verwendet (Menüwahl (p) : publish to node).
Die Listener zum Empfangen der Nachrichten sind in ItemEventCoordinator beschrieben und werden auch mittels des Subscribers (Menüwahl (l) : listen to node) verwendet.

- Ermöglichen Sie die Übertragung von Nutzdaten (Beispielsweise Simplepayload)

Um mit der Anwendung Payload zu publishen wird die Publisher.java verwendet (Menüwahl (p) : publish to node). Die Listener zum verarbeiten der ItemEvents sind in ItemEventCoordinator beschrieben und werden auch mittels des Subscribers (Menüwahl (l) : listen to node) verwendet.

- Leafs und ggf. mögliche Eigenschaften der Leafs sollen angezeigt werden können (Service Discovery)

ServiceDiscovery wird unterstützt bzw. genutzt. Hierzu verwendet man den Publisher (Publisher.java) und die Menüwahl i : information to the node.

6. Client - Entwicklung

Abschließend soll ein Client erstellt werden, welcher das Projekt repräsentiert. Dafür ist eine Nutzung des REST Webservices und des XMPP Servers erforderlich:

- Das System (RESTful Webservices sowie XMPP Server) muss über ein grafisches User Interface nutzbar sein

Es wird eine GUI (GUI.java) zur Verfügung gestellt, mittels der das Steuern möglich ist.

- Verwendung von Swing wird empfohlen
- Getrenntes Ausführen auf unterschiedlichen Systemen von Client und Server /Webservice sollte optimalerweise möglich sein

Getrenntes Ausführen auf unterschiedlichen Systemen ist möglich.

3. Vorgesehene Funktionalitäten

Vorgesehen für die Umsetzung im Projekt sind:

Das Subscriben an bestimmten nodes/topics.

Das Publishen von Payload(SimplePayload) und das Anzeigen der Änderungen seitens der Subscriber.

Get auf alle Listenressourcen und einzelne Ressourcen.

Post auf einzelne nicht vorhandene Ressource (ID wird vom System vergeben), was ein Kreieren dieser Ressource zur Folge hat.

Delete auf einzelne Ressource (mit ID).

Post (Update) auf einzelne Ressourcenelemente. Zum Beispiel das Hinzufügen und Löschen von Kommentaren bei einzelnen Ressourcen.

Features zum vereinfachten Testing:

Ändern des Users über die GUI.

Exploring der Daten über Browser (einfach links zu den Listenressourcen).

4. Werdegang des Projekts

Anders als in der Konzeption vorgesehen haben wir Kommentare, likers und followers nicht als eigenständige Ressourcen angelegt. Sie werden stattdessen als Items der jeweiligen Ressource realisiert. Die Granularität dieser Elemente ist uns zu fein, um sie als eigene Ressource abzubilden, da wir diese (für diese nur teilweise Realisierung des Systems) nicht einzeln adressierbar machen müssen. Dies ist nur eine mögliche Entwurfsentscheidung, die andere Lösung wäre nicht falsch gewesen.

Entgegen der vorangegangenen Überlegung wurde bei den Warenkörben auf ein Element Anmerkungen (String) verzichtet, da dieses nicht relevant genug erschien.

In der anfänglichen Planung waren Shops und Endnutzer noch eigenständige Ressourcen. Da jedoch beim Fortschreiten des Projekts deutlich wurde, dass diese beiden sowohl funktional als auch strukturell hinsichtlich der Daten nichts wesentlich unterscheidet wurde fortan darauf verzichtet, eine zusätzliche Ressource Shop einzuführen. Jeder User hat auch Shop-Funktionalität, sobald er Informationen wie Shop-Adresse und Informationen über den

Zahlungsverkehr in das System eingespeist hat.

Weiter wurde auch in Sammlungsangebote und Stickerangebote unterschieden, was aber redundant ist, da Angebote in der finalen einfach eine Sammlung an Verweisen auf Sticker *und* Angebote sind.

Außerdem war ein weiteres Element für eine Ressource Photo geplant: Das Aufzeichnungsdatum (nicht zu verwechseln mit dem Einstellungsdatum, also dem Uploaddatum). Sicher wäre dies eine gute Anreicherung des Photoelements gewesen, jedoch lassen sich diese Informationen auch in der Beschreibung eines Photos unterbringen. Keine dieser Entscheidung ist falsch oder richtig; die eine Lösung lediglich etwas schlanker, die andere dafür feingranularer in der Datenstruktur.

5. Überlegungen, Entscheidungen und Alternativen

In diesem Kapitel haben wir Gedanken und Entscheidungen zu diesem Projekt gesammelt. Diese sollen die Dokumentation lediglich anreichern. Viele der Gedanken werden nicht umgesetzt, liefern aber im Ganzen eine gute Ergänzung dazu, was das System leisten soll und wie dieses bewerkstelligt werden kann.

xs:element vs. Xs:attribute:
Meta-Daten werden als Attribute gespeichert, wie Ids. „...metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.“[2]

Sichtbarkeit:
Sichtbarkeitist Teil der Funktionalität, nicht des XML-Schemas, im XML-File werden allerdings die Rechte für ausgewählte Nutzer in einer Form einer Liste (Benutzer_ID und die dazugehörigen Rechte als Auswahl) gespeichert. Standardeinstellung sind vergleichbar mit den Rechten eines Gast-Accounts (z.B. in Foren). [Kommentare auf der eigenen Seite eines Nutzers nur als befreundeter User möglich, auch um Spam zu vermeiden, es sei denn man vergibt global mehr Rechte]. Man hat a) die Möglichkeiten, Rechte global anzugeben (alles zulassen, Standard, alles blocken) und b) sind auch user-spezifische Angaben möglich.
Ein shop hat nicht die Möglichkeit Rechte individuell zu vergeben, da er sich sonst strafbar machen kann sobald er bestimmte User bevorzugt (Rechtslage in den jeweiligen Staaten).
Bei der globalen Einstellung 'alles blocken' werden im Falle einer Transaktion allerdings die dafür notwendigen Daten übermittelt.

Speicherung von Passwörtern:
Passwortdaten und Daten zu Zahlungsinformation gehören auch nicht in das XML-Schema

Handling von Zahlungsinformationen:
Zahlungsinformation werden nur im Falle eines Kaufs angefordert und dann gespeichert oder verworfen (Temporärangaben möglich, die nach Abschluss der Transaktion wieder aus dem System gelöscht werden) [Sollen die Daten für zukünftige Transaktionen gespeichert werden?]
SWIFT und IBAN ermöglichen Transaktionen in und aus dem Ausland über mehrere Währungsräume hinweg.
Es werden jeweils nur die relevanten Daten erhoben, zum Beispiel die Kreditkartennummer bei Zahlung per Kreditkarte.

Der Shop als gewerblicher Kunde muss mindestens eine Zahlungsinformation für das System bereitstellen.
Ob ein Kunde als Shop angemeldet ist wird über ein Attribut festgehalten.
Der Kunde wird bei einer Transaktion gefragt, ob gegebenenfalls bestehende Anschriften übernommen werden sollen. Ist keine Anschrift im System hinterlegt, so werden Rechnungs- und Lieferanschrift angefordert.
Sowohl Privatkunden als auch gewerbliche, reine Shops müssen sich als Shop anmelden, wenn sie Sticker verkaufen wollen. Möchte man als Nutzer nur kaufen, ist das nicht nötig.

Aktiv/Inaktiv-Attribut von Angeboten:
Um Datensätze als Shop besser verwalten zu können haben Angebote einen Aktiv/Inaktiv-Attribut.

Verlinkung zu externen Shops:
Will ein Shop den URI eines vorhandenen Internetshops verlinken, so kann er das in Form eines links in der Shop-Beschreibung tun. Unsere Intention ist natürlich, dass über unser System gebucht wird(Prozente).

Registrierung:
Bei einer neuen Registrierung wird eine email-Adressen-Validierung vorgesehen, ohne die man sich nicht registrieren kann (Aktivierungslink).

Ist ein User noch aktiv?:
Die Angabe über das letzte Einloggen („last seen“) werden auch im XML-File hinterlegt, so dass man nachverfolgen kann, ob ein User noch aktiv ist.

Logging/Versionierung:
Um Missbrauch des offenen Systems in der Form eines Wikis zu begegnen könnten bei Änderungen/Löschungen von Ressourcen die Änderungen (bzw. die vorherige Version) und wer diese verursacht hat gespeichert werden. Bei Missbrauch ist es so möglich, diese User zu sperren/löschen.

RelatedShop-Option:
Einzelne Sticker können auch related shops haben, in denen sie angeboten werden. Diese Einträge sind optional und können von den jeweiligen Anbietern erworben werden.

Is_a_shop_Attribut:
Da sich Shops und Endnutzer durchaus unterscheiden wird das Attribut is_a_shop eingeführt.

Rechtevergabe:
Die Rechtevergabe kann auf viele verschiedene Weisen realisiert werden. Vor allem in Hinsicht auf die Granulierung. Möglich wären zum Beispiel Unterscheidungen der Rechte für einzelne Medien in Hinsicht auf Kommentieren von bestimmten Nutzern/Gruppen. Außerdem sind Standardeinstellungen sinnvoll, so dass die bevorzugte Einstellung nicht immer wieder konfiguriert werden muss. Zur Verwaltung der Rechte werden sowohl eine Liste der „befreundeten" als auch der „geblockten" User-IDs für jeden User angelgt.

CHOICE:
Verfügbarkeiten, Geschlecht und ähnliche Auswahloptionen werden als choice, Mehrfachauswahlen wie Zahlungsinformationen der Shops als multiple choice (choice mit maxOccurs) angelegt.

Aktualität vs. Performance:
Aktualität und Konsistenz sind für uns wichtige Designkriterien, unter anderem weil die Anwendung soziale Komponenten beinhaltet. So werden zum Beispiel Ids statt Namen gespeichert für den Fall, dass ein User sich umbenennt. So wird gewährleistet, dass sein aktueller Name bei allen seinen Beiträgen angezeigt wird (Stabile URIs). Dies kann zu vielen Abfragen führen, die unter Umständen (z.B. Schlechte Übertragungsrate) zu Beeinträchtigungen in der Geschwindigkeit führen können.
Dies wird von uns aber in Kauf genommen, da die Aktualität und Konsistenz der Daten den Vorrang haben.

Struktur - breit vs. tief:
Um eine möglichst einfache Kommunikation mit den einzelnen Informationen des Systems(und derer untereinander) zu ermöglichen wird die Struktur möglichst breit angelegt. Jede logische Informationseinheit, die das System bereitstellen muss wird als eigene Ressource abgebildet. So ergeben sich viele „kleine" Ressourcen, die sich gegenseitig verlinken.

Feingranularität der XML-Dateien:
Macht es immer Sinn, die Feingranularität von XML-Dateien 1:1 in Ressourcen umzusetzen?
Um dies beantworten zu können stellen wir uns die Frage: Werden die Daten einzeln abgefragt?
Wenn ja, dann macht es Sinn diese auch als einzelne Ressource vorzusehen. Dabei gilt zu beachten: Es ist gut möglich, dass mit der Lebenszeit des Systems und den gesammelten Erfahrungen die Anforderungen was eine optimale Struktur angeht, sich im Laufe der Zeit sukzessive verändern.

Ressourcendesign:
Es wird mit vielen „kleinen" xml-Dateien gearbeitet, die sich untereinander

verlinken.
Es werden relative URIs verwendet.
Es werden technische Schlüssel verwendet. Fachliche Schlüssel können zusätzlich genutzt werden, wo sie Sinn ergeben, weil der Nutzer sie kennt/vermutet (z.B. nickname statt UserID). Hierzu werden dann einfach mehrere URIs erzeugt, die auf die gleiche Ressource verweisen.

6. Quellen

Hier sind sowohl Quellen angegeben, die zum Erstellen der Dokumentation als auch des Programmcodes und der Datenstrukturen herangezogen und verwendet wurden.

[1] Stefan Tilkov, REST und HTTP
[2] http://www.w3schools.com/dtd/dtd_el_vs_attr.asp
[3] https://github.com/metashare/META-SHARE/issues/527
[4] http://www.igniterealtime.org/builds/smack/docs/latest/javadoc/
[5] http://www.vogella.com/articles/JAXB/article.html
[6] http://www.oio.de/public/xml/rest-webservices.htm
[7] http://xmpp.org/extensions/xep-0060.html