Software Architektur Dokument
Projekt BierIdee

Danilo Bargen, Christian Fässler, Jonas Furrer

7. Mai 2012

Inhaltsverzeichnis

1 Einführung **4**
1.1 Zweck . . . 4
1.2 Gültigkeitsbereich . . . 4
1.3 Referenzen . . . 4
1.4 Übersicht . . . 4

2 Systemübersicht **5**
2.1 Komponenten . . . 5
2.1.1 Datenbank . . . 5
2.1.2 REST API . . . 6
2.1.3 Clientanwendung . . . 6
2.2 Schnittstellen . . . 6
2.2.1 Datenbankzugriff . . . 6
2.2.2 REST API / HTTP . . . 6

3 Architektonische Ziele & Einschränkungen **6**
3.1 Safety / Security . . . 6

4 Logische Architektur **7**

5 Backend Architektur **9**
5.1 Klassendiagramm . . . 9
5.2 Typische Systemoperationen . . . 10
5.3 Wichtige Design Aspekte . . . 11

6 Frontend Architektur **12**
6.1 Klassendiagramm . . . 13
6.2 Typische Systemoperationen . . . 14

7 Prozesse und Threads **15**
7.1 Datenbank . . . 15
7.2 Server API . . . 15
7.3 Android App . . . 15

8 Deployment **16**
8.1 Frontend . . . 16
8.2 Backend . . . 16

9 Datenspeicherung **17**

10 Grössen und Leistung **20**

Änderungshistorie

Version	**Datum**	**Änderung**	**Person**
v1.0	02.04.2012	Dokument erstellt	dbargen
v1.1	05.05.2012	Frontendarchitekturdiagramm	dbargen
v1.2	05.05.2012	Dokument überarbeitet	dbargen
v1.3	05.05.2012	SSDs hinzugefügt	cfaessler
v1.4	07.05.2012	Wichtige Designaspekte	cfaessler, jfurrer, dbargen

1 Einführung

1.1 Zweck

Dieses Dokument beschreibt die Softwarearchitektur des Projektes BierIdee.

1.2 Gültigkeitsbereich

Die Gültigkeit des Dokumentes beschränkt sich auf die Dauer des SE2-Projekte Modules FS2012.

1.3 Referenzen

- REST.Interface.pdf
- Evaluation.Datenbank.pdf
- Authentication.pdf
- Systemtests.pdf
- Externes.Design.pdf

1.4 Übersicht

Das System der Bieridee kann in drei Kategorien/Teilbereiche unterteilt werden: Die Datenbank, die REST API und die Clientanwendung. Zwischen der API und der Android-Clientanwendung wird ausschliesslich via HTTP kommuniziert. Um auf die API zuzugreifen, muss man sich mit einem API-Token authentifizieren und die Requests mit einem kryptografischen Verfahren signieren. Logisch werden im Backend die Datenbank- und API-bezogenen Klassen in einem gemeinsamen Package geführt. Das Frontend wird nach dem MVC Pattern und gemäss Android Best Practices implementiert. Beim Deployment wird die API direkt über den eingebauten Webserver von Restlet gestartet, ohne Servlet-Container. Es wird aber aus Performance- und Optimierungsgründen ein Reverse Proxy davorgesetzt. Alle Daten werden im Backend persistent in einer Graphendatenbank gespeichert und durch Traversierung und eingebaute Algorithmen abgefragt. Das System soll in der Testphase bis zu 50 gleichzeitige Nutzer bedienen können.

2 Systemübersicht

Unsere Systemarchitektur gliedert sich in drei Teilbereiche: Die Datenbank, die REST API und die Clientanwendung.

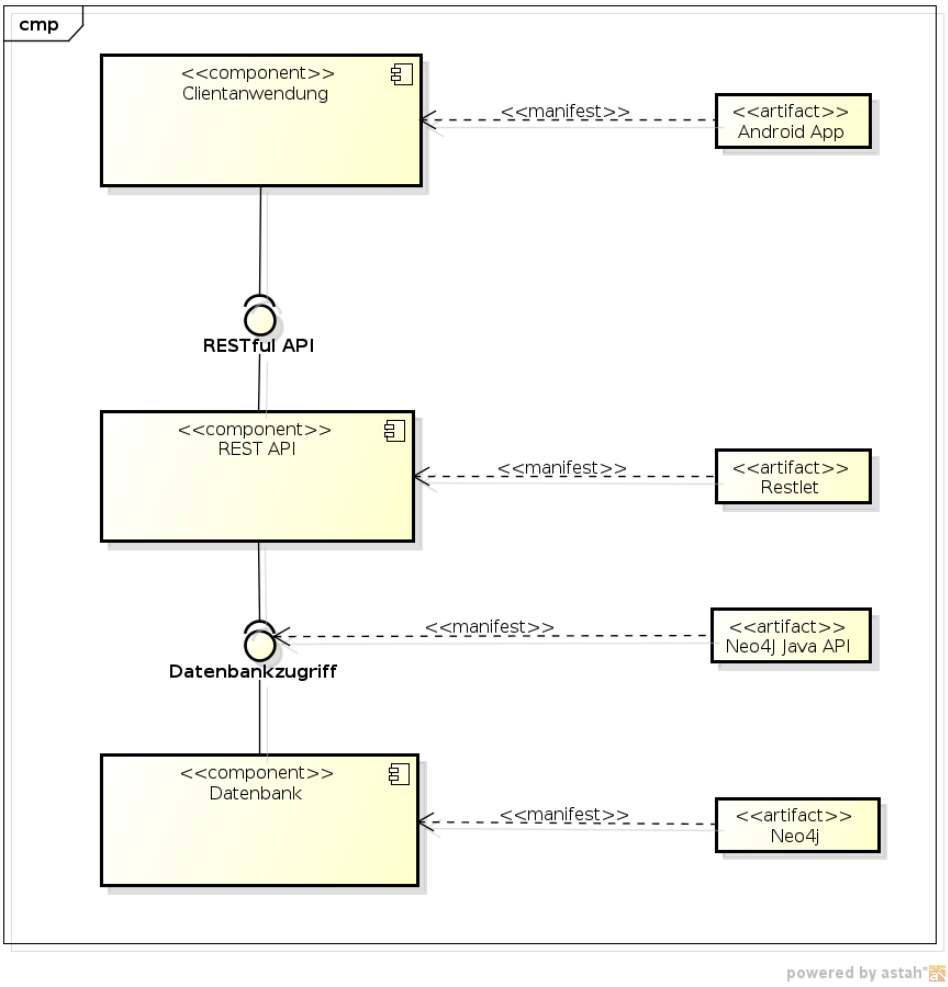

Abbildung 1: Komponentendiagramm

2.1 Komponenten

2.1.1 Datenbank

Als Datenbank kommt die Graphendatenbank Neo4j[1] zum Einsatz. Eine Graphendatenbank hat den Hauptvorteil, dass man sehr gut mit den Relationen arbeiten kann und so beispielsweise die Bier-Empfehlungen durch Traversierung realisieren

[1] `http://neo4j.org/`

kann. Näheres zum Entscheid zur Verwendung einer Graphendatenbank siehe Dokument Evaluation.Datenbank.pdf

2.1.2 REST API

Für das Zwischenglied zwischen Daten und Client kommt ein RESTful[2] Web Service zum Zuge. Diese REST API wird mithilfe von Restlet[3] realisiert. Die Domainobjekte werden in Form von abstrakten Ressourcen mit entsprechenden Repräsentationen zur Verfügung gestellt.

Die REST-Ressourcen werden im Dokument *REST.Interface.pdf* näher spezifiziert.

2.1.3 Clientanwendung

Die Android Clientanwendung greift auf die REST API zu und stellt die Informationen auf sinnvolle Art und Weise dar.

2.2 Schnittstellen

2.2.1 Datenbankzugriff

Der Datenbankzugriff geschieht über die eingebaute Java-Schnittstelle von Neo4j. Dadurch wird die Verwendung von JDBC oder eines OR Mappers unnötig. Datenbanknodes können entweder via objektorientierter Syntax oder mithilfe einer domainspezifischen Abfragesprache namens *Cypher* abgefragt werden.

2.2.2 REST API / HTTP

Die RESTful API wird von Restlet bereitgestellt. Die Domainobjekte werden in Ressourcen gegliedert und via HTTP im JSON- oder XML-Format ausgegeben.

3 Architektonische Ziele & Einschränkungen

3.1 Safety / Security

Die API wird nicht öffentlich/frei zugreifbar sein. Das Sicherheitskonzept ist zweistufig aufgebaut. Einerseits erzeugt die API sogenannte API-Tokens, mit welchem sich die App als „trustful“ App identifiziert.

[2] `http://en.wikipedia.org/wiki/Representational_state_transfer#RESTful_web_services`
[3] `http://www.restlet.org/`

In einem zweiten Schritt authentifiziert und autorisiert sich der Benutzer der Android App mittels Benutzernamen und Passwort-Hash. Die Daten werden jedoch nicht direkt übertragen, sondern werden nur benutzt um die Requests mithilfe des HMAC-SHA256 Verfahrens zu signieren.

Das genaue Authentifizierungs bzw. Signierungsverfahren ist im Dokument Authentication.pdf dokumentiert.

Security-Steps:

1. App Authentifizierung mittels App-Token
2. User-Authentifizierung mittels Benutzername und Passwort
3. Autorisierung für gewünschte Operationen

Alle Schritte werden bei jeder Anforderung einer Ressource (http Request) durchgeführt, um das Prinzip der *Statelessness* von RESTful APIs zu gewährleisten.

Auf Datenbankebene wird kein Rechtesystem implementiert. Die Datenbank ist auch nicht direkt ansprechbar, daher kann das ganze Berechtigungssystem in der REST API realisiert werden.

4 Logische Architektur

Die grundsätzliche Aufteilung in Subsysteme in diesem Projekt wird bereits im Kapitel *Systemübersicht* beschrieben, deshalb wird hier primär auf die Package-Architektur eingegangen.

Die logische Architektur des Gesamtprojektes wird in zwei Hauptkategorien bzw. Packages unterteilt – *android* und *back*. Beide Packages sind Unterpackages von *ch.hsr.bieridee*.

Im nachfolgenden Diagramm sieht man die Aufteilung der Projektteile in Packages.

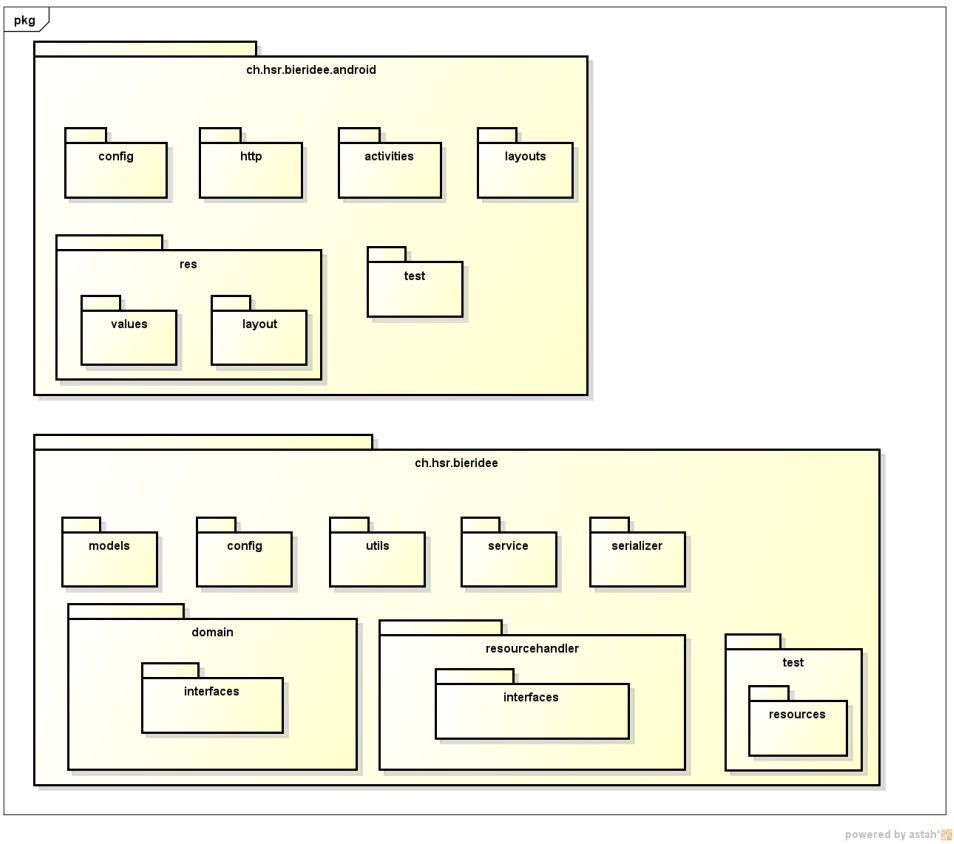

Abbildung 2: Package Diagramm

Selbstverständlich wird die Android-Applikation nach dem Model-View-Controller Modell implementiert. Dies ist jedoch zusammen mit dem Ressourcen-Management (Templates, Strings, Grafiken etc) ein integraler Bestandteil des Android SDKs, deshalb wird es hier nicht genauer erörtert.

5 Backend Architektur

Nachfolgend werden wichtige architektonische Klassen und deren Zusammenhang im Backend mittels Klassendiagramm beschrieben. Ebenso wird ein für die Architektur typischer Ablauf, das Erfassen einer Bewertung, mit den wichtigsten Systemoperationen anhand von einem Systemsequenzdiagramm aufgezeigt.

5.1 Klassendiagramm

Nachfolgend ein detailliertes Backend-Architektur-Klassendiagramm am Beispiel eines Bier-Objektes.

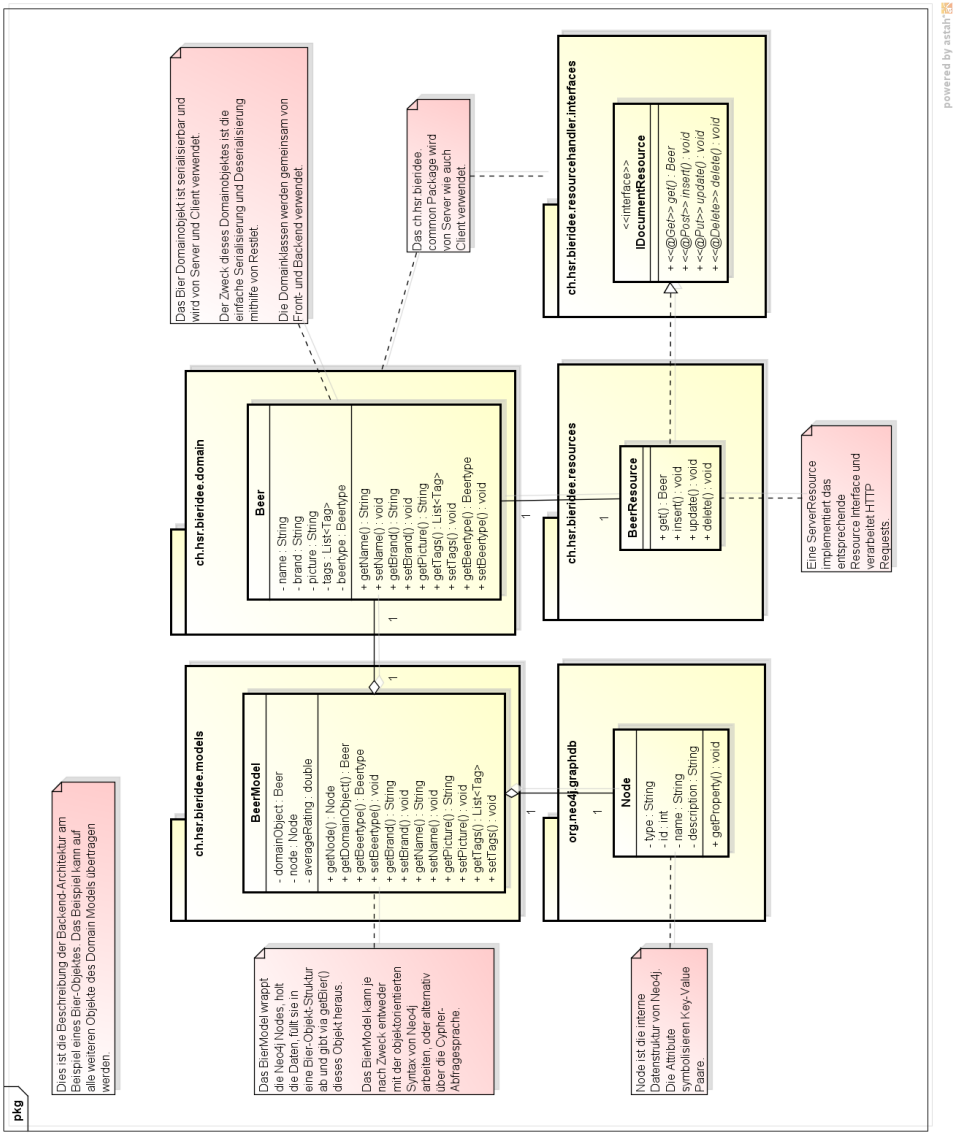

Abbildung 3: Backend Architektur

5.2 Typische Systemoperationen

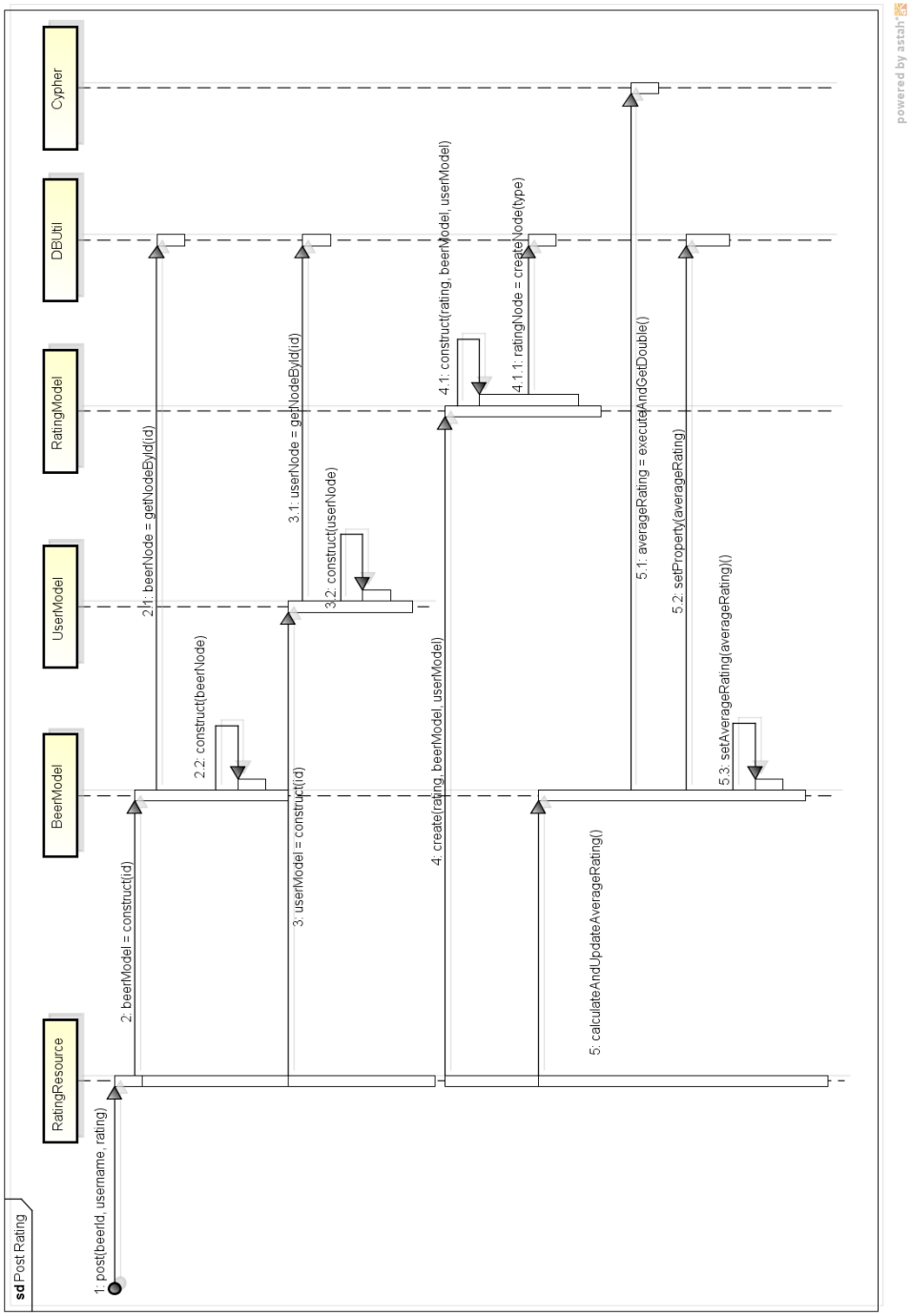

Abbildung 4: Post Rating SSD

Das SSD PostRating beschreibt den Vorgang einer Bier-Bewertung. Der Einstiegspunkt in diesen Vorgang ist die Rating Ressource, welche auf einen POST Request

reagiert und daraufhin den Vorgang startet. Um nicht direkt von der Ressource auf die Datenbank zuzugreifen, werden für das betroffene Bier sowie für den betroffenen Benutzer die entsprechenden Models instanziert. Der Zugriff auf die Datenbank passiert über eine Utility Klasse. Wenn dann alle die Informationen zum Bier und zum Benutzer zur Verfügung stehen, wird ein neues Rating Model instanziert und daduch auch – wieder mithilfe der DBUtil Klasse – die neue Rating-Node erstellt. Wenn nun das Rating in der Datenbank exisiert muss noch die durchschnittliche Gesamtbewertung des betroffenen Bieres neu berechnet werden. Dies passiert direkt auf der Datenbank mithilfe eines Cypher-Queries. Dieses wird über das Bier Model initiiert und über ein Cyher Utility ausgeführt, um auch in diesem Fall die Ressource von der Datenbank getrennt zu halten. Wenn alle Vorgänge die in diesem SSD beschrieben sind erfolgreich durchgeführt wurden, wurde eine neue Rating Node in der Datenbank erstellt und korrekt mit den benötigten Index-Nodes verbunden. Desweiteren wurde die durchschnittliche Bewertung des betroffenen Bieres auf der dazugehörenden Daten-Node in der Datenbank aktualisiert.

5.3 Wichtige Design Aspekte

Trennung Datenhaltung Domain

Um eine saubere Trennung zwischen Persistenz (Datenbank) und der eigentlichen Domain zu erhalten, wurde das DAO (Data Access Object) Pattern verwendet, um so die Persistenz von der Domain zu trennen. Einerseits erreicht man durch die klare Aufteilung der Zuständigkeiten eine höhere Kohäsion und auch eine gewisse Flexibilität in datenbankspezifischen Spezialitäten. Sprich, es ist somit einfacher die Datenbank durch eine andere zu ersetzen. Auch die übrige Applikationslogik, (so zum Beispiel die Resource-Handler) besitzt keine Kenntnis über die Datenbank. Das zentrale Interface für sämtliche Datenbankoperationen sind die Models. Daraus resultiert auch eine erstrebenswerte Verringerung der Kopplung.

Abbildung von REST Archetypen

Um den REST Best Practices gerecht zu werden, sind für alle benötigten Archetypen[4] entsprechende Interfaces definiert. Die verschiedenen Resourcen implementieren diese Interfaces, dadurch wird sichergestellt, dass sämtliche Resourcen den geforderten REST Archetypen entsprechen.

[4]REST Archetypen sind gemäss dem Buch *REST API Design Rulebook* von O'Reilly Patterns für das Resource-Design einer API.

6 Frontend Architektur

Nachfolgend wird ein Ausschnitt aus der Frontendarchitektur am Beispiel der Bierliste aufgezeigt. Die Architektur ist grösstenteils durch das Android SDK und ihre Best-Practice Richtlinien vorgegeben. Anhand eines Systemsequenzdiagrammes wird eine systemtypische Interaktion zwischen Android App und Server API aufgezeigt. Es sind nur die für die Architektur wichtigsten Systemoperationen enthalten.

6.1 Klassendiagramm

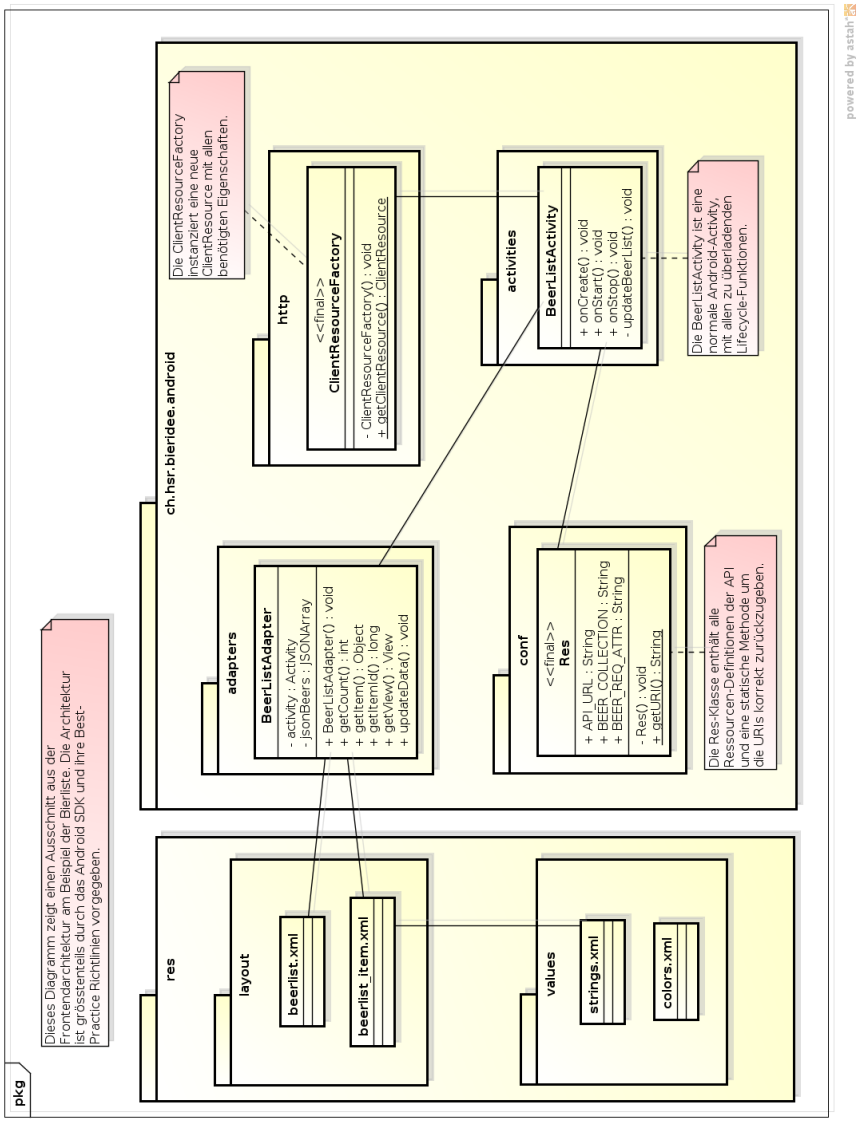

Abbildung 5: Frontend Architektur

6.2 Typische Systemoperationen

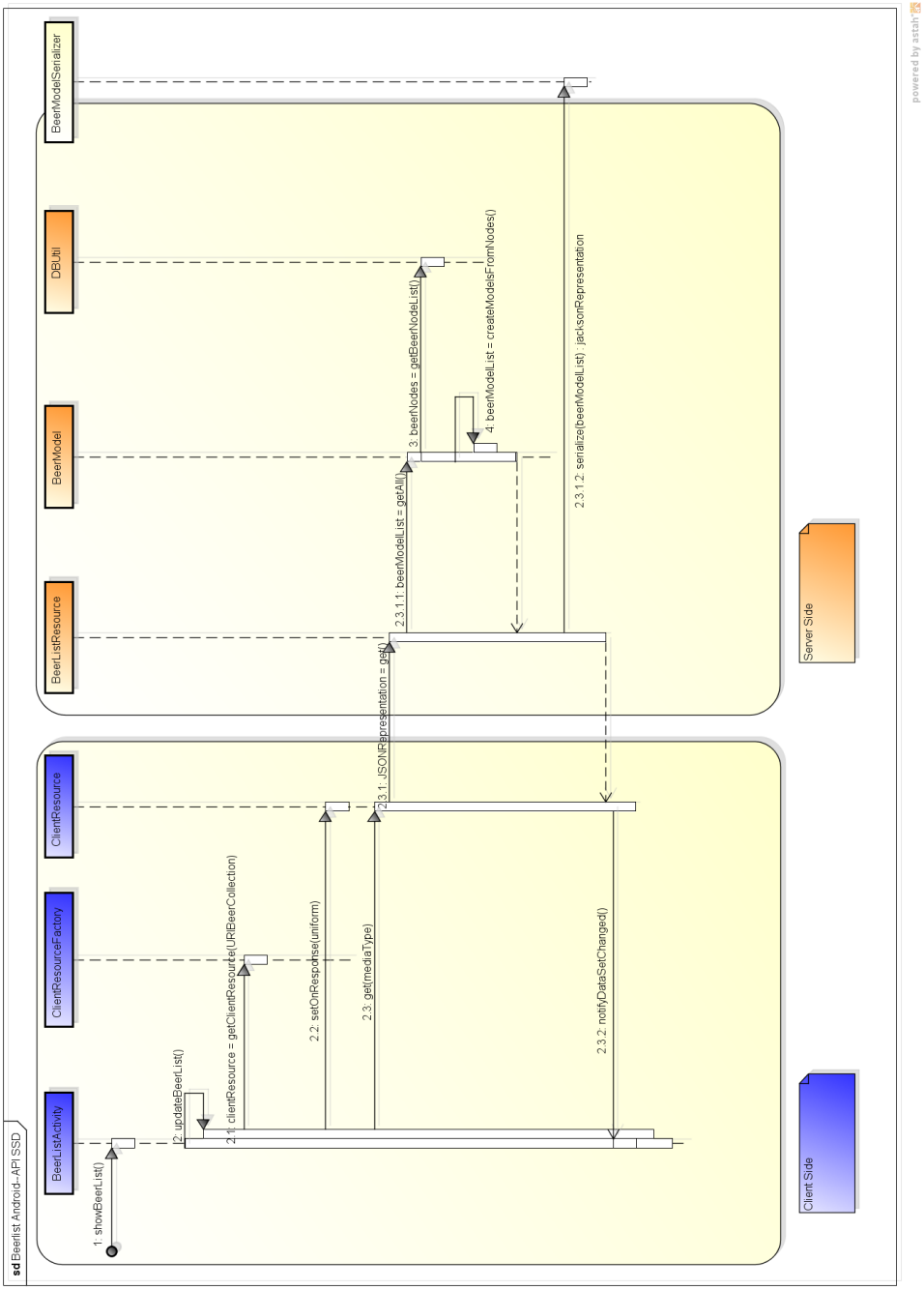

Abbildung 6: Frontend zu Backend Kommunikation SSD

Das SSD Beerlist beschreibt den prinzipiellen Ablauf der Kommunikation zwischen der Android App und der Server API. Einstiegspunkt ist das Starten des BeerList Screens. Als Erstes wird zur HTTP Kommunikation aus der `ClientResourceFactory`

eine `ClientResource` Instanz angefordert. Vor dem Absenden des Requests wird eine Completion Callback Methode definiert (`setOnResponse`). Somit kann der Request asynchron ausgeführt und das GUI später bei Erfolg aktualisiert werden. Als nächstes wird ein HTTP PUT Request auf die definierte URL abgesetzt. Serverseitig wird der Request verarbeitet; allenfalls werden wichtige Attribute wie Header, Query und Request Daten extrahiert und geparst. Der serverseitige Empfänger des Requests ist ein Resource Handler. Solche Resource Handler sind für jede URL im Dispatcher definiert. Der Resource Handler veranlasst die Instanzierung der notwendigen Models, diese wiederum kümmern sich um Abfragen in der Datenbank und erstellen die Domain Objekte. Die Models werden danach in einen JSON String serialisiert und mittels HTTP an den Client geschickt. Das GUI wird anschliessend über den aktualisierten Datenbestand informiert.

6.3 Wichtige Design Aspekte

Model-View-Controller

Die Android-Applikation wird gemäss den Android-Developer-Richtlinien strikt nach dem MVC-Pattern aufgebaut. Alle GUI-Bezogenen Layouts werden als XML-Ressourcen erstellt, alle Strings in übersetzbare String-Ressourcen ausgelagert. Die Activities fungieren als Controller, die Daten werden durch Content Provider und entsprechende Adapter von der Controllerlogik getrennt.

Auslagerung des HTTP-Clientresource-Handling

Die HTTP Client-Ressourcen werden in eine entsprechende `ClientResourceFactoy` ausgelagert, um tiefe Kopplung und hohe Kohäsion zu erreichen. Die `ClientResourceFactoy` initialisiert die `ClientResource` Objekte und gibt die frisch erstellte Instanz zurück.

Mithilfe eines `ResourceSigningDecorator`s können existierende Client Ressourcen signiert werden. Dies wird im Decorator mithilfe des *Chain of Responsibility* Patterns gelöst. Der `ClientResource` werden sogenannte `next`-Objekte zugewiesen, welche man „chainen„ kann um mehrere Verarbeitungsschritte durchzuführen.

7 Prozesse und Threads

7.1 Datenbank

Die Neo4j Datenbank garantiert die ACID Prinzipien. Sämtliche Datenzugriffe erfolgen in Transaktionen, somit ist die Integrität der Datenbank sichergestellt. Neo4j ist komplett thread safe implementiert. Die Datenbank wird im embedded Modus verwendet und kann direkt über eine Java API angesprochen werden.

7.2 Server API

Die Server API ist multithreaded. Für jede Clientanfrage wird ein Worker Thread erstellt welcher den Request bearbeitet.

Die benötigten Ressourcen-Handler werden bei jeder Anfrage neu instanziert, somit müssen diese Komponenten nicht thread safe implementiert sein.

7.3 Android App

Die UI ist single threaded. Langlaufende Operationen werden in asynchrone Background-Threads ausgelagert und benachrichtigen das UI mithilfe von Callbacks.

8 Deployment

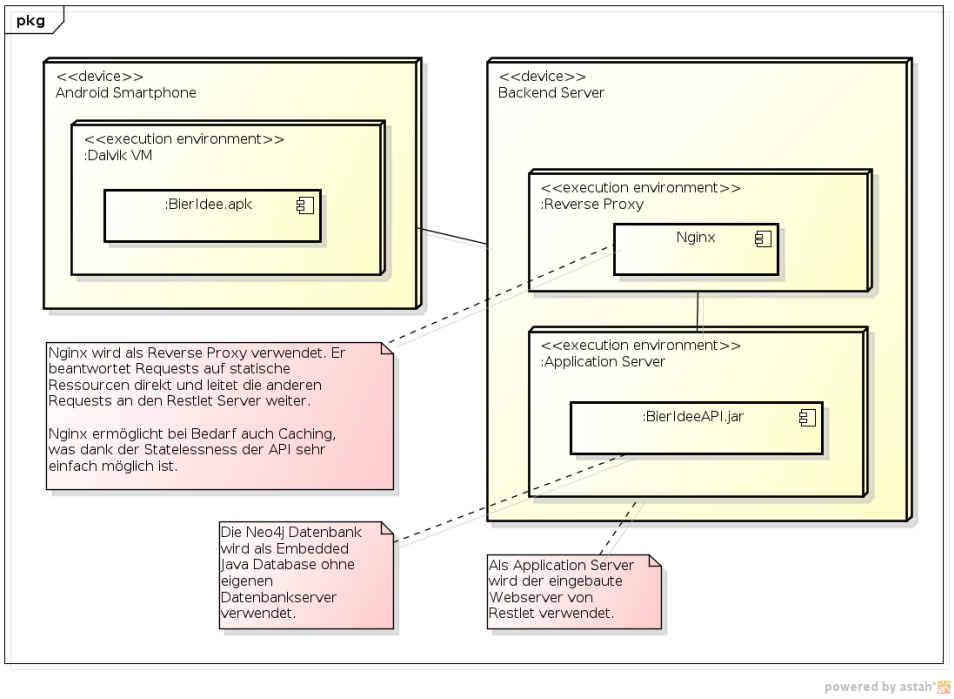

Abbildung 7: Deployment Diagramm

8.1 Frontend

Die Frontend-Applikation wird im Rahmen dieses Projektes manuell als `.apk`-Datei auf die Testgeräte kopiert werden. Falls das Projekt weitergezogen wird, wäre eine Publikation im Android Market gut denkbar, für den Moment wird aber darauf verzichtet.

8.2 Backend

Das Backend wird auf einem Debian Linux Server deployed. Die Application Server Instanz läuft mithilfe von Java 1.6 und Restlet ohne zusätzliche Software.

Zur Überwachung des Systemprozesses wird Supervisor verwendet (nicht im Deployment Diagramm erwähnt, da kein integraler Bestandteil). Supervisor überwacht den Systemprozess und startet ihn, falls er abstürzt, sofort neu.

Vor den Application Server wird ein Reverse Proxy – konkret Nginx – gesetzt. Dieser nimmt alle Requests entgegen und entscheidet, ob es sich dabei um statische oder um dynamisch generierte Daten handelt. Statische Daten – beispielsweise Bilder – werden direkt zurückgeliefert, während die anderen Requests an den Application

Server weitergegeben werden. So kann auch ein Caching problemlos implementiert werden, was den Application Server entlastet und die Skalierbarkeit stark erhöht.

9 Datenspeicherung

Die Daten werden in einer Graphendatenbank (konkret Neo4j) abgelegt. Diese besteht grundsätzlich nur aus zwei Arten von Objekten – Nodes und Relationen. Mithilfe von diesen zwei Objekten kann die gesamte Datenstruktur sehr flexibel abgebildet werden; Abfragen können mit gezielter Traversierung durch eine domainspezifische Abfragesprache namens *Cypher* ausgedrückt werden.

Der Evaluationsbeschrieb der Datenbank findet sich im Dokument *Evaluation.Neo4j.pdf*.

Nachfolgend das allgemeine Datenbankdiagramm sowie ein Diagramm eines Beispielfalles:

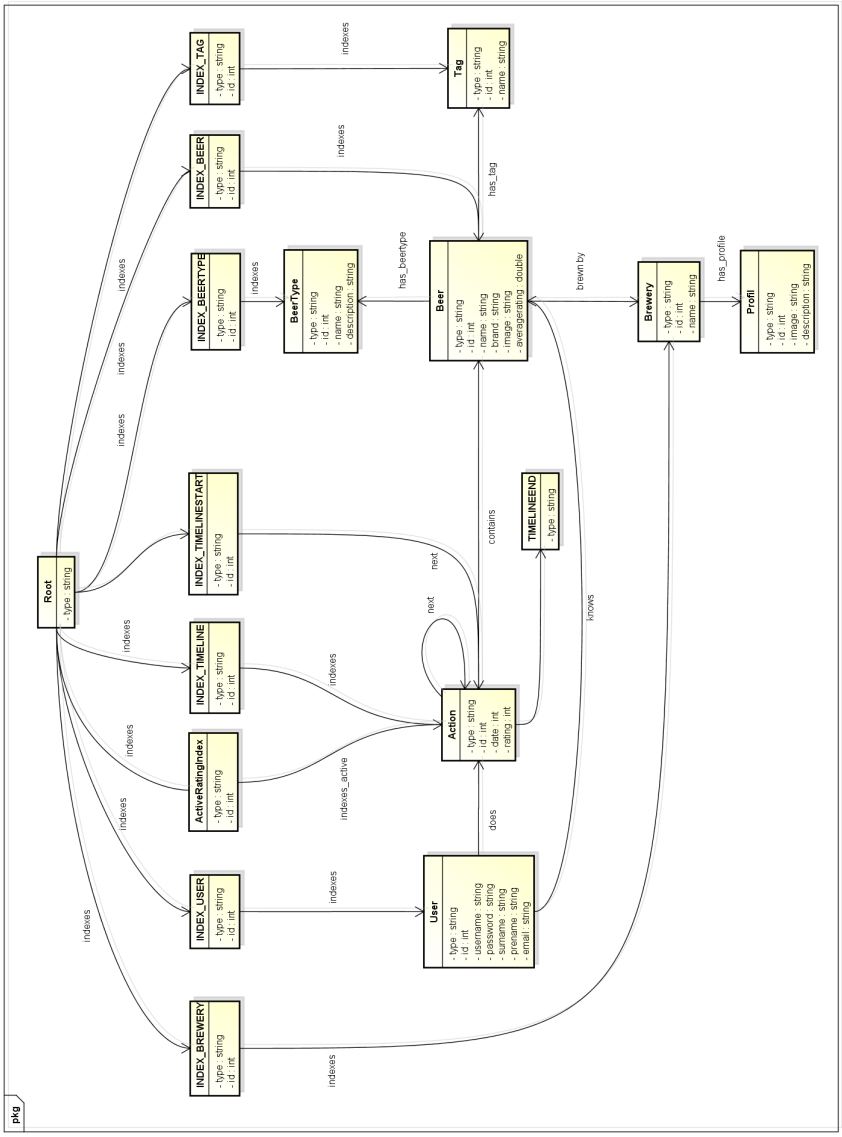

Abbildung 8: Allgemeines Datenbankdesign

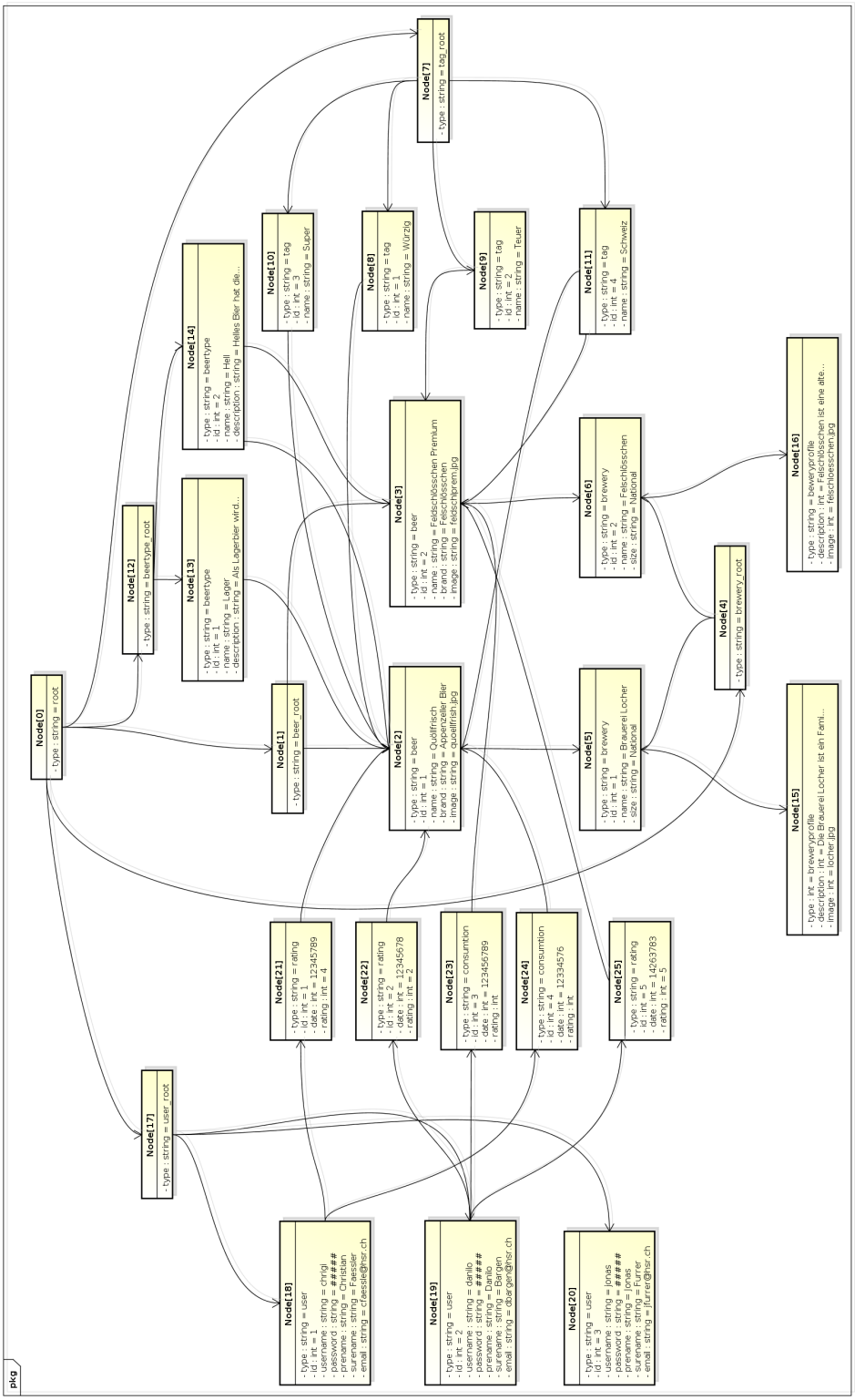

Abbildung 9: Datenbank Beispielfall

10 Grössen und Leistung

Im Rahmen dieses Projektes ist das System und die serverseitige Umgebung für den Einsatz mit bis zu 50 Benutzern, rund 100 Bieren und 20 Brauereien ausgelegt. Es sollte aber skalierbar sein, so dass mit besserer Serverumgebung erheblich mehr Benutzer bedient werden könnten.

Die Architektur wird so ausgelegt, dass so wenig Rechenleistung wie möglich auf der Clientseite benötigt wird. Das heisst, sämtliche Logik und Aufbereitung der Daten wird auf der Serverseite implementiert. Da clientseitig die Unterschiede in der verwendeten Hardware sehr gross sein können (Smartphones, Tablets, Netbooks), können keine einheitlichen Anforderungen zur Rechenkapazität auf den Geräten gestellt werden. Desweiteren muss heutzutage bei der Entwicklung von Mobile Apps stark auf den Energieverbrauch geachtet werden. Mit der zentralisierten Ausführung rechenintensiver Aufgaben sind die verfügbaren Ressourcen bekannt und die Software kann entsprechend adäquat entwickelt werden.