|  |
| --- |
| Universität Potsdam |
| Tool.UP |
| Dokumentation |

|  |
| --- |
| Thiemo Belmega  15.06.2016 |

Inhalt

[Einleitung 3](#_Toc454189993)

[Userstories 3](#_Toc454189994)

[Userstory 1 3](#_Toc454189995)

[Userstory 2 4](#_Toc454189996)

[Userstory 3 4](#_Toc454189997)

[Architektur 5](#_Toc454189998)

[Technologien und Bibliotheken 6](#_Toc454189999)

[Datenmodell und Datenbankschema 6](#_Toc454190000)

[Backend 7](#_Toc454190001)

[de.uni-potsdam.cs.toolup.model 7](#_Toc454190002)

[de.uni-potsdam.cs.toolup.database 7](#_Toc454190003)

[de.uni-potsdam.cs.toolup.ws.beans 8](#_Toc454190004)

[de.uni-potsdam.cs.toolup.ws.resource 8](#_Toc454190005)

[Rest-API 9](#_Toc454190006)

[GET-Anfragen 9](#_Toc454190007)

[POST-Anfragen 10](#_Toc454190008)

[Front-End 11](#_Toc454190009)

[Tests 12](#_Toc454190010)

[Lokale Tests (UnitTests) 12](#_Toc454190011)

[Webservice-Tests 12](#_Toc454190012)

[GUI-Tests 13](#_Toc454190013)

[Anhang I: Set up 14](#_Toc454190014)

GitHub:

<https://github.com/tbelmega/Tool.UP>

<https://github.com/tbelmega/Tool.UP-frontend-backbone>

Kontakt:

t.belmega@gmx.de

# Einleitung

Der Webservice Tool.UP präsentiert strukturierte und durchsuchbare Informationen über E-Learningwerkzeuge und ihre Funktionalität.

Der Nutzer navigiert zu den gewünschten Informationen über das Web-Frontend, das Abfragen zur REST-Schnittstelle des Backends schickt. Die Daten werden vom Betreiber von Tool.UP in einer SQL-Datenbank gepflegt, von wo das Backend die Daten abfragt und weiterverarbeitet.

Die vorliegende Dokumentation beschreibt die einzelnen Komponenten von Tool.UP und ihren groben Aufbau. Darüber hinaus folgt Tool.UP dem Ideal von *selbstdokumentierendem Code*: Durch beschreibende Variablen- und Methodennamen sowie kurze Methoden und Klassen mit klaren Verantwortlichkeiten ist der Code so gehalten, dass er möglichst lesbar und verständlich ist. Die mitgelieferten Unit-Tests und Integrationstests dienen gleichzeitig als Code-Beispiele, die dem Entwickler zeigen, wie die Klassen und Methoden des Produktiv-Codes zu verwenden sind. Eine vollständige Dokumentation des Codes durch Diagramme oder JavaDoc wird vermieden, da diese Art Dokumentation schnell veraltet und fehleranfällig wird, wodurch sie mehr Schaden als Nutzen verursacht (vgl. Robert C. Martin 2008, „Clean Code“).

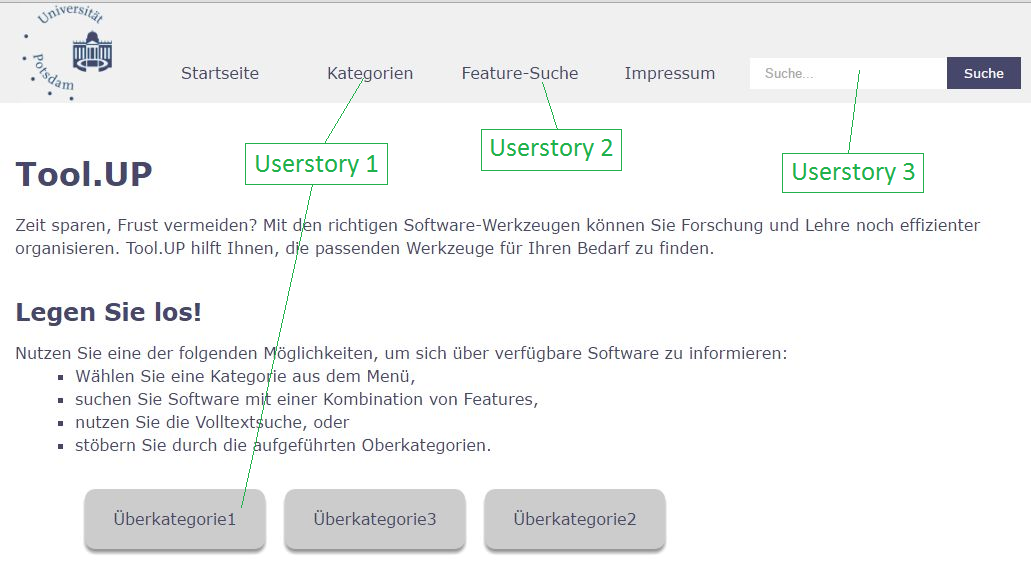
# Userstories

Tool.UP deckt folgende Userstories ab:

## Userstory 1

Absicht: Der Nutzer möchte sich einen Eindruck von verfügbaren E-Learning-Werkzeugen machen.

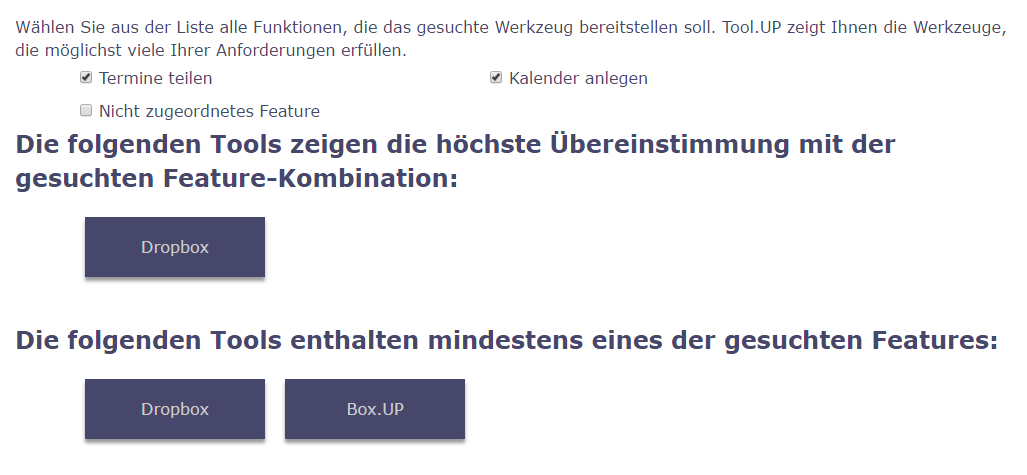
Umsetzung: Der Nutzer beginnt mit einer Liste von Kategorien und navigiert durch den Baum von Ober- und Unterkategorien und den dazugehörigen Applikationen. Zum Einstieg nutzt er entweder auf der Startseite alle Kategorien ohne weitere Oberkategorie, oder auf der Kategorien-Seite alle Kategorien mit direkt zugeordneten Applikationen.



## Userstory 2

Absicht: Der Nutzer sucht nach einem E-Learning-Werkzeug, das bestimmte Funktionen erfüllt.

Umsetzung: In der Feature-Suche legt der Nutzer eine Kombination von Features fest, die seinen Wünschen entspricht. Tool.UP zeigt ihm auf derselben Seite die besten Suchergebnisse sowie alle Applikationen mit mindestens einem der gesuchten Features.



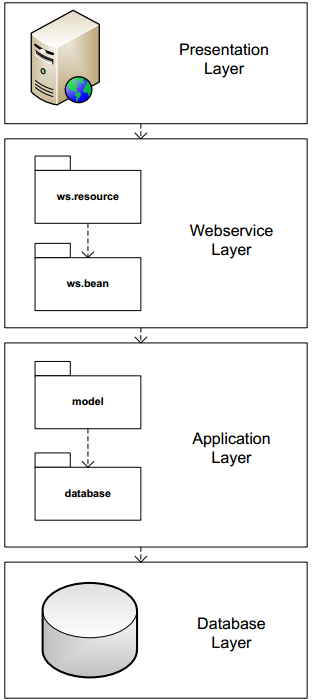
## Userstory 3

Absicht: Der Nutzer möchte eine Applikation finden, die er dem Namen nach kennt oder die er bei einer früheren Nutzung von Tool.UP gesehen hat.

Umsetzung: Der Nutzer gibt ein oder mehrere Worte in die Textsuche ein. Er erhält eine Liste aller Applikationen, deren Name, Beschreibung oder Features zu der Suchanfrage passen.

# Architektur

Tool.UP ist in vier Schichten organisiert. Kennzeichen einer Schichten-Architektur ist, dass die Komponenten jeder Schicht nur aufeinander und auf die direkt darunterliegende Schicht zugreifen. Keine Komponente greift auf eine höher- oder noch tieferliegende Schicht zu. Diese Architektur stellt eine möglichst lose Kopplung der Komponenten sicher, sodass man Teile des Systems verändern oder austauschen kann, ohne das gesamte System anpassen zu müssen.

Die **Präsentationsschicht** umfasst das Web-Fontend von Tool.UP, aufgebaut als Single-Page-Application mit dem Framework backbone.js. Sie umfasst eine HTML-Seite, ein CSS, und jeweils einer JavaScript-Datei für jede Ansicht (Backbone View) und jedes Datenmodell (Backbone Models und Collection). Backbone lädt die Daten für jedes benötigte Model über eine Http-Anfrage ans Backend.

Im Backend wird die Http-Anfrage vom REST-Framework Jersey entgegengenommen und als Methodenaufruf einer Webressource ausgeführt (**Webservice-Schicht**).

Webressourcen sind annotierte Methoden im Paket *resource*. Jersey erwartet als Antwort Java-Objekte im JavaBean-Format, die es dann in JSON[[1]](#footnote-1) umwandelt und in eine Http-Antwort packt.

Die JavaBeans sind im Paket *bean* definiert. Sie dienen als Wrapperklassen für die Domänenobjekte aus der Anwendungsschicht.

Die Anwendungsschicht ist unabhängig vom Einsatz als Webservice. Es ist daher möglich, statt der Webservice-Schicht alternative Benutzerschnittstellen auf die Anwendungsschicht aufzubauen ohne sie zu verändern, zum Beispiel eine lokale grafische Benutzeroberfläche in Swing.

Die **Anwendungsschicht** umfasst die Java-Pakete *database* für den Zugriff auf die Datenbank, und das Paket *model* für die Datenhaltung im Backend und die Anwendungslogik. Tool.UP setzt einen Objektcache ein, um die Anzahl der Datenbankzugriffe zu reduzieren. Dabei werden Objekte nach dem Laden aus der Datenbank im Cache gehalten und bei erneutem Zugriff auf dieselbe ID direkt aus dem Cache verwendet.

Die **Datenbankschicht** besteht aus einer SQL-Datenbank zur Datenhaltung. Über einen externen Client können die Daten hier auch verändert werden. Das Backend von Tool.UP kann zwar bereits Daten in die Datenbank einfügen, updaten und löschen, vorerst wird diese Funktionalität aber nur von den UnitTests genutzt; es gibt in Tool.UP keine Nutzeroberfläche zum Verändern von Daten.

# Technologien und Bibliotheken

Die folgende Tabelle listet die verwendeten Technologien und Bibliotheken von Drittanbietern auf:

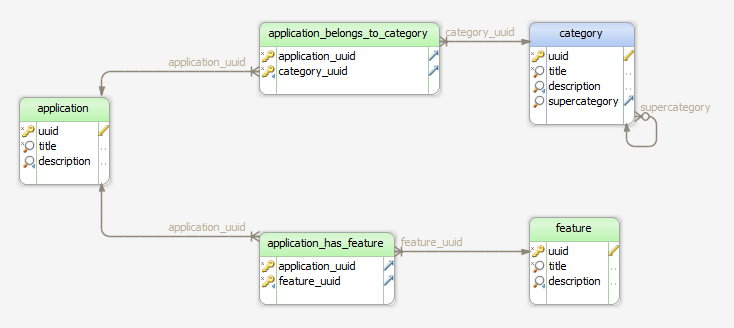
|  |  |
| --- | --- |
| **Verwendungszweck** | **Technologie** |
| Datenbankserver | MySQL Server 5.6 |
| Datenbankkonnektor im Backend | jDBC |
| Backend | Java 8 |
| Backend Test-Framework | TestNG |
| GUI Test-Framework | Selenium WebDriver |
| Build-Tool | Apache Maven |
| Backend-Bibliothek | Apache Commons Lang |
| Java-Framework für REST-Schnittstellen | Jersey |
| Verarbeitung von JSON-Daten in Java | org.json |
| Objektcache | EHCache |
| Frontend | HTML, CSS, Javascript |
| Frontend Bibliotheken | jquery, underscore.js, backbone.js |

# Datenmodell und Datenbankschema

Die in Tool.UP betrachteten Entitäten sind E-Learning-Programme und ihre Funktionalität. Außerdem werden die E-Learning-Programme in eine Hierarchie von Kategorien eingeteilt.

Im Datenbankschema werden diese Entitätsklassen repräsentiert durch die Tabellen Application, Feature und Category. Applications und Features stehen in viele-zu-viele-Beziehung (n:m), Applications und Categories ebenso. Dadurch werden zwei Beziehungstabellen notwendig. Außerdem kann jede Category eine Category als Überkategorie besitzen, dadurch steht die Tabelle Category in eins-zu-viele-Beziehung (1:n) zu sich selber.

Das vollständige Datenbankschema ist im Diagramm abgebildet.



**Konvention**: Die ID-Werte der Datensätze müssen mit den Teilstrings „application-“, „category-“ oder „feature-“ beginnen, entsprechend ihrer jeweiligen Tabelle. Dieses Präfix wird vom Java-Backend benutzt, um den Datensatz zu einer gegebenen ID aus einer Tabelle zu laden.

# Backend

Das Backend von Tool.UP ist in Java 8 geschrieben.

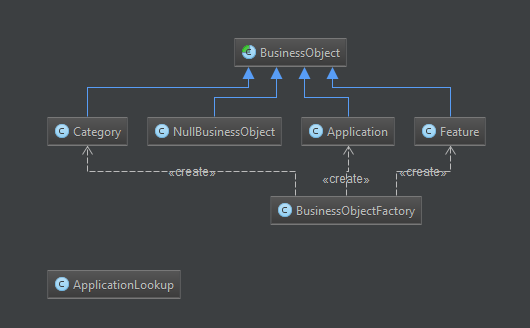
Es kommuniziert mit der Datenbank über den JDBC-Datenbankonnektor. Dabei werden vorbereitete SQL-Abfragen aus der Datei SQL\_Statements.xml gelesen und ausgeführt.

Mit der Frontend kommuniziert das Backend über eine REST-Schnittstelle, die mit Hilfe der javax.rs-Referenzimplementierung Jersey gestaltet ist.

Der Java-Code des Backends besteht aus mehreren Paketen, die im Folgenden beschrieben sind.

## de.uni-potsdam.cs.toolup.model

Das Paket **model** stellt die Klassen für die Domänenobjekte von Tool.UP bereit (Application für E-Learning-Programme, Feature für deren Funktionen und Category für die Kategorien-Hierarchie), sowie die Methoden zu ihrer Verarbeitung.



Die Klassen Application, Category und Feature entsprechen den Enitätsklassen des Datenbankmodells. Ihre gemeinsame Oberklasse BusinessObject enthält die Funktionalität, um eine JSON-Repräsentation der Java-Objekte zu erzeugen. Die BusinessObjectFactory erzeugt Java-Instanzen auf Grundlage der Datenbank-Antworten. Das NullBusinessObject repräsentiert das Nichtvorhandensein eines BusinessObjects, zum Beispiel als Ergebnis eine Datenbankabfrage (vgl. das Entwurfsmuster NullObject in der Literatur).

Die Klasse ApplicationLookup repräsentiert eine Menge von Applications als Ergebnis einer Feaure-Suche. Eine Instanz von ApplicationLookup nimmt eine Liste von Feature-IDs entgegen und identifiziert alle Applications aus der Datenbank, die eine größtmögliche Menge der gegebenen Features abdecken. Im besten Fall liefert der ApplicationLookup also alle Applications, die alle gewünschten Features beinhalten; wenn es keine solche Applications gibt, liefert er alle Applications, die höchstens eines der gesuchten Features nicht aufweisen, und so weiter.

## de.uni-potsdam.cs.toolup.database

Das Paket **database** ist für die Kommunikation mit der Datenbank verantwortlich.

Die Klasse ToolUpProperties lädt Einstellungen wie Serveradresse, Nutzername und Passwort aus der Datei Tool.UP\_cfg.xml.

Die Klasse SqlStatementFactory erzeugt SQL-Abfragen als JavaObjekte, indem sie den benötigten SQL-Code aus der Datei SQL\_Statements.xml einliest.

Der DatabaseController (Singleton-Entwurfsmuster) bildet den Adapter zur Datenbank für das übrige Backend. Er stellt alle Methoden zur Datenbankabfrage zur Verfügung, holt die PreparedStatement-Objekte dafür aus der SqlStatementFactory und führt sie aus.

## de.uni-potsdam.cs.toolup.ws.beans

Das Paket **beans** enthält Wrapperklassen für die Domänenobjekte des Pakets model. Eine BusinessObjectBean umhüllt also jeweils ein BusinessObject. Die Bean-Klassen arbeiten dadurch als Adapter (Adapter-Entwurfsmuster) zwischen den Model-Klassen und dem Jersey-Framework. Jersey erwartet nämlich Objekte im JavaBean-Format und konvertiert diese automatisch in JSON.

**Wichtig!** Brächte man die Model-Klassen selbst ins Bean-Format und ließe Jersey darauf arbeiten, wäre Jersey in einer Endlosschleife gefangen: Um eine Application in JSON abzubilden, müssten zunächst alle zugehörigen Features in JSON umgewandelt werden. Um ein Feature umzuwandeln, müssten zunächst alle zugehörigen Applications umgewandelt werden. Hier entsteht eine endlose Rekursion.

Aus diesem Grund enthält z.B. eine ApplicationBean eben keine Sammlung von FeatureBeans oder Feature Objekte als Felder; sondern Sammlungen von Strings, die bereits ins JSON-Format umgewandelte Features repräsentieren. Diese String-Umwandlung wird aber nicht von Jersey vorgenommen, sondern von der selbst implementierten convertToJson()-Methode der Klasse BusinessObject. Dabei wird die Rekursion verhindert, indem die verknüpften Objekte nur noch als ID dargestellt werden.

Zur Verdeutlichung derselbe Sachverhalt in anderen Worten:

1. Der Nutzer fragt über die REST-Schnittstelle ein Domänenobjekt an.
2. Tool.UP produziert eine Bean des angefragten Objekts, das von Jersey in JSON-Format umgewandelt und dem Nutzer per http-Response zurückgegeben werden kann.
3. Eine Bean enthält alle Daten des Domänenobjekts, inklusive der Verknüpfung mit anderen Domänenobjekten (Beziehung 1. Grades).
4. Dadurch enthält die Bean auch alle Daten der verknüpften Objekte 1. Grades in Form eines JSON-Strings.
5. Die Umwandlung der verknüpften Objekte 1. Grades erfolgt durch die Methode BusinessObject.convertToJson(). Die mit diesen Objekten verknüpften Objekte (Beziehung 2. Grades) werden nicht auch noch in JSON umgewandelt, um eine endlose Rekursion zu verhindern. Stattdessen fügt die Methode BusinessObject.convertToJson() der JSON-Repräsentation der Objekte 1. Grades lediglich eine Sammlung der IDs der Objekte 2. Grades hinzu.
6. Die http-Antwort an der Nutzer enthält damit
   1. Eine von Jersey hergestellte JSON-Form des angefragten Objekts mit all seinen Daten
   2. Alle Daten der verknüpften Objekte 1. Grades
   3. Nur die IDs der verknüpften Objekte 2. Grades

## de.uni-potsdam.cs.toolup.ws.resource

Das Paket **resource** stellt die Einstiegspunkte ins Backend bereit. Entsprechend der Spezifikation von Jersey wird über Annotationen jede gültige URL des Webservice einem Methodenaufruf von Tool.UP zugeordnet. Die Ressourcenklasse lässt von Bean-Paket das passende Antwortobjekt erzeugen, hüllt es in eine http-Response mit dem Status „200 OK“ und einem CORS-Header, und gibt dieses zurück.

Das Jersey-Framework übernimmt die Aufgabe, auf eingehende http-Requests mit dem passenden Methodenaufruf zu reagieren und die http-Antwort zu generieren.

# Rest-API

Die folgende Liste beschreibt alle gültigen Aufrufe an die REST-Schnittstelle. Die voranzustellende Base-URL ergibt sich dabei aus der Konfiguration des verwendeten Servlet-Containers.

Beispiel: <http://localhost:8080/toolup/> für einen lokalen Tomcat-Server auf Port 8080 mit Tool.UP deployed als toolup.war).

**Wichtig!** {id} steht dabei für die ID eines existierenden BusinessObjects. Das Tool.UP-Frontend übergibt niemals nicht-existente IDs, sondern ermöglicht nur Klicks auf Links, deren URL aus den IDs aus einer vorherigen Antwort vom Tool.UP-Backend erzeugt wurden.

## GET-Anfragen

**/application/**

Gibt ein JSON-Array mit allen Applications der Datenbank zurück.

**/application/{id}**

Gibt die angefragte Application zurück.

**/feature/**

Gibt ein JSON-Array mit allen Features der Datenbank zurück.

**/feature/{id}**

Gibt das angefragte Feature zurück.

**/category/**

Gibt ein JSON-Array mit allen Categories der Datenbank zurück.

**/ category /{id}**

Gibt die angefragte Category zurück.

**/category/withApplication**

Gibt alle Categories zurück, welche direkt mit Applications verknüpft sind. (Bottom-Level-Categories)

**/category/toplevel**

Gibt alle Categories zurück die selbst keine Superkategorie besitzen. (Top-Level-Categories)

## POST-Anfragen

**/search**

*Body-Parameter (x-www-form-urlencoded): search string: [text]*

Gibt ein JSON-Array mit allen Applications zurück, bei denen der Such-String zu ihrem Titel oder Beschreibung passt, oder zu Titel oder Beschreibung eines ihrer Features. Die praktische Bedeutung dieses „passt – zu“ (engl. „match“) hängt von der Implementierung der genutzten SQL-Datenbank ab und sollte in deren Dokumentation nachgelesen werden (Stichwort full text search).

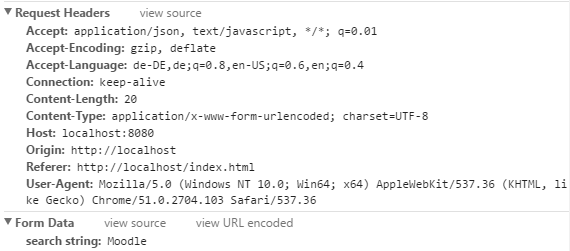


Abbildung : Beispiel für POST-Request an /search

**/lookup**

*Body-Parameter (x-www-form-urlencoded): features: [ids]*

Gibt das Ergebnis eines Feature-Lookup zurück. Das ist ein JSON-Object mit den beiden Properties „bestMatches“ und „singleMatches“.

BestMatches ist ein Array aller Applikationen, die eine möglichst große Zahl der gesuchten Features enthalten.

SingleMatches ist ein Array aller Applikationen, die mindestens eins der gesuchten Features enthalten.

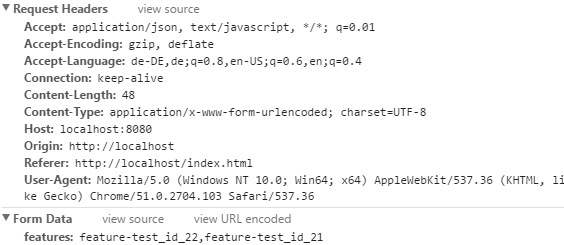


Abbildung : Beispiel für POST-Request an /lookup

# Front-End

Das Fontend von Tool.UP ist eine Single-Page-Application mit dem Framework backbone.js. Sie umfasst eine HTML-Seite, ein CSS, und jeweils einer JavaScript-Datei für jede Ansicht (Backbone View) und jedes Datenmodell (Backbone Models und Collection).

Die HTML-Seite index.html enthält mehrere Abschnitte:

* Die Templates. Sie sind bei Aufruf der Seite zunächst nicht sichtbar, sondern werden später vom JavaScript-Code aus in die Seite eingefügt, um die Seite dynamisch zu ändern.
* Die Navigationsleiste. Sie enthält Links auf die Startseite, die Seite mit den Bottom-Level-Kategorien, die Feature-Suche und das Impressum, sowie das Freitext-Suchfeld. Die Seiten (aus Nutzersicht) sind als Backbone-Routes umgesetzt, bestehen aus technischer Sicht also nur aus dynamischer Veränderung des Inhalts einer einzigen Seite.
* Den Inhalt der Startseite.
* Den AJAX-Prefilter. Hier ist definiert, wie jede ausgehende AJAX-Abfrage modifiziert wird. Standardmäßig ist hier die URL des Backends mit <http://localhost:8080/toolup> definiert und die Erlaubnis für Cross-Domain-Abfragen wird erteilt. Letzteres ist nötig, wenn Frontend und Backend unter verschiedenen Domains oder Ports erreichbar sind.
* Den Backbone-Router. Hier wird definiert, welche URLs auf welche Routes verweisen und wie die Webapp auf die Routes reagiert.

Das gesamte Frontend funktioniert nach folgendem Prinzip:

1. Der Nutzer navigiert auf eine URL.
2. Der Backbone-Router führt die Route zu dieser URL aus und rendert eine View.
3. Die View erzeugt ein Model bzw. eine Collection als Objekt und fragt die zugehörigen Daten asynchron vom Backend an.
4. Nach Erhalt der Antwort vom Backend wird eine Success-Callbackfunktion ausgeführt. Diese nutzt das dazugehörige Template aus index.html und fügt die Darstellung der Daten in die Seite im Browser ein.

# Tests

Das Backend von Tool.UP besitzt zur Zeit eine Test-Abdeckung von über 90% (Messung mit IntelliJ IDEA CodeCoverage). Die Tests sollen einerseits jederzeit die Funktionsfähigkeit des Codes prüfen; dadurch wird rasches und sicheres Verändern des Codes erst möglich, da nach jedem Änderungsschritt die Tests anzeigen, dass der Code noch funktioniert wie zuvor. Andererseits sollen die Testfälle aber auch dokumentieren, was der Produktivcode tut und als Code-Beispiele für die Benutzung dienen. Der Entwickler sei daher ermutigt, zum Verstehen einer Klasse des Produktivcodes vor allem ihre Testfälle zu betrachten.

Tool.UP enthält drei Arten von Tests:

## Lokale Tests (UnitTests)

Lokale Tests decken die Klassen der Anwendungsschicht ab, insbesondere werden die Datenbankoperationen getestet, sowie die Umwandlung von Domänenobjekten in JSON-Daten und Datenbank-Daten.

Die lokalen Tests benötigen keinen laufenden Applicationserver (Tomcat), sondern führen den Produktivcode direkt als Javaprogramm aus. Es ist daher nicht nötig, bei Änderungen im Code erst ein Build auszuführen und Tool.UP neu zu deployen. Schlagen lokale Tests fehl, wird ein Maven-Build von Tool.UP sogar abgebrochen und das Deployen verhindert.

Eine Datenbankverbindung ist nötig. Wenn bei der Ausführung eines lokalen Tests eine Datenbankabfrage nötig ist, werden zuvor die passenden Daten in die Datenbank geschrieben und am Ende wieder gelöscht. Auf diese Weise ist der Test unabhängig davon, welche Daten in der Datenbank vorhanden sind, und hinterlässt keine dauerhaften Nebeneffekte. Für das Schreiben und Löschen der Testdaten ist die Klasse DatabaseControllerHelper zuständig.

## Webservice-Tests

Diese Tests decken die Klassen der Webservice-Schicht ab. Ein Teil der Webservice-Tests sind UnitTests und arbeiten genau wie bei den lokalen Tests beschrieben.

Die übrigen Tests sind Integrationstests. Sie setzten einen laufenden Applicationserver voraus, auf dem das Tool.UP-Backend deployed ist. Die Integrationstests befinden sich im Unterpaket „integration“ und erben von der Klasse AbstractTestWebSerivce, wo die URL des Servers als Konstante definiert sein muss.

Die Integrationstests schicken http-Abfragen an den Server und werten die Antwort aus. Sie testen damit die REST-API, genau wir sie vom Frontend benutzt wird, inklusive der Funktion des Jersey-Frameworks.

In Bezug auf die Testdaten arbeiten die Integrationstests ebenfalls anders als die Unit-Tests. Die Tests selber manipulieren die Datenbank nicht. Stattdessen setzen sie einen bestimmten Datenbestand voraus, um die erhaltenen Antworten mit den erwarteten zu vergleichen. Daher kann eine Änderung der Daten in der Datenbank zum Fehlschlagen der Integrationstests führen!

Das Skript aus der Datei TestCreateScript.sql legt eine Datenbank nach Tool.UP-Datenbankschema auf dem Datenbankserver an und enthält genau die benötigten Testdaten. Es empfiehlt sich daher, während der Entwicklung Tool.UP so zu konfigurieren, dass auf eine derart erzeugte Testdatenbank anstatt auf die Livedaten zurückgegriffen wird. Da die Livedaten veränderlich sein sollen, ist es nicht sinnvoll, die Tests auf die heutigen Livedaten zuzuschneiden.

## GUI-Tests

Das Frontend-Projekt enthält eine Suite von GUI-Tests. Im Gegensatz zu den anderen Testarten greifen diese über die grafische Benutzeroberfläche im Browser auf das zu testende Programm zu.

GUI-Tests haben zwei Besonderheiten:

* Sie versuchen den Nutzer nachzuahmen, weil sie Nutzerinteraktion testen. Daher sollten GUI-Komponenten möglichst so identifiziert werden, wie es für den Benutzer möglich ist, also z.B. über ihre Aufschrift anstatt über die unsichtbare ID. Auch sollte ihre Sichtbarkeit (auf dem Bildschirm) anstatt ihrer Existenz (als Datenobjekt im Speicher) getestet werden.
* Sie laufen ansynchron ab. Der Test kann nicht unterscheiden, ob ein erwartetes Ergebnis auf der dargestellten Seite wirklich fehlt, oder ob die Seite noch nicht fertig aufgebaut ist. Daher arbeiten die Tests mit wait-Anweisungen und Timeouts. Außerdem fügt das Frontend am Ende der Success-Funktionen <div id="ready"/> ein, was keinen anderen Zweck hat, als den Tests einen Anhaltspunkt für die Fertigstellung der Abfrage zu geben. Dennoch bleiben GUI-Tests insgesamt ungenauer als die übrigen Tests.

Die vorliegenden GUI-Tests wurden mit Selenium WebDriver implementiert. Sie können wie die übrigen Tests mit TestNG ausgeführt werden, setzen aber voraus, dass auf dem ausführenden System Firefox als Webbrowser installiert und in den PATH eingetragen ist.

Grundsätzlich lassen sich GUI-Tests in mehrere Kategorien unterscheiden:

**Navigationstests**

Sie testen, ob dem Nutzer nach Klick auf einen Link oder Button die erwartete Seite angezeigt wird. Die derzeit enthaltenen GUI-Tests von Tool.UP fallen in diese Kategorie. Für diese Tests muss definiert werden, aufgrund welcher Eigenschaft eine Seite identifiziert wird. Tool.UP identifiziert die Seiten über den Titel.

**Komponententests**

Sie testen, ob eine Seite alle erwarteten Komponenten enthält und ob diese funktionieren (z.B. anklickbar sind).

**Featuretests**

Sie testen, ob ein Feature aus Nutzersicht funktioniert (z.B. ob nach Anklicken mehrerer Features im Feature-Lookup die erwarteten Applications angezeigt werden). Ziel ist es aber nicht, die dazugehörige Logik im Backend erschöpfend zu testen (dies ist Aufgabe der Unit- und Integrationstests), sondern das Auslösen der richtigen Abfrage und richtige Darstellung der Antwort.

**Szenariotests**

Sie testen, ob eine vollständige Userstory ausführbar ist. Dies schließt die Navigation und Nutzung von Features ein, was immer der Nutzer benötigt, um im Sinne der Story zu seinem Ergebnis zu kommen.

Komponententests, Featuretests und Szenariotests wurden bisher nicht implementiert. All diese Tests wären stark von den Daten in der Datenbank abhängig, sodass eine Veränderung der Daten die Tests brechen würde. Vor einer Implementierung müsste daher sorgfältig analysiert werden, auf welchen (Live- oder Test-) Datenbestand solche Tests aufbauen sollen. Aus diesem Grund werden Komponenten-, Feature und Szenariotests zur Zeit explorativ bei Hand durchgeführt.

# Anhang I: Set up

Folgende Schritte sind notwendig, um Tool.UP zu hosten:

* Aufsetzen und starten eines MySQL Server 5.6
* Um das Datenbankschema zu erzeugen, kann das Skript aus der Datei TestCreateScript.sql genutzt werden, oder ein Export der bestehenden Tool.UP-Datenbank
* Konfigurieren der Datenbankverbindung:
  + Datei Tool.UP\_cfg\_template.xml in Tool.UP\_cfg.xml umbenennen
  + Einträge für Passwort usw. an die neue Datenbank anpassen
  + Nach Eintragen des Passworts sollte die Datei nicht mehr comitted werden!
* Bauen des Backends:
  + Maven Build ausführen. Erzeugt die Datei toolup.war.
* Deployen des Backends: toolup.war auf einem Application Server deployen, z.B. Apache Tomcat oder Glassfish
* Testen des Backends:
  + URL des Backends eintragen in AbstractTestWebService.
  + Danach allTests.xml mit TestNG ausführen
* Konfigurieren des Frontends:
  + URL des Backends eintragen in index.html, Abschnitt ajaxPrefilter
* Deployen des Frontends:
  + Inhalt des src-Ordners in den Zielordner eines Webservers kopieren, z.B. Apache Http Server
* Testen des Frontends:
  + URL des Frontends in NavigationTest.java eintragen, (Standardmäßig ist <http://localhost/index.html> eingetragen).
  + Tests mit TestNG ausführen.

# Anhang II: TODO

Die hier aufgelisteten Punkte konnten im Projektumfang von Tool.UP nicht umgesetzt werden.

## Fehlerbehandlung bei nicht-existenter ID

Das Tool.UP-Frontend schickt nur IDs in Abfragen ans Backend, die zuvor vom Backend aufgelistet wurden. Wird die REST-Schnittstelle des Backends aber von einer dritten Partei aus angesprochen, könnten durchaus IDs abgefragt werden, die nicht existieren. Von der Anwendungschicht wird in diesem Fall ein NullBusinessObject zurückgegeben. Die Webserviceschicht behandelt diesen Fall zur Zeit nicht, sondern wird eine ClassCastException auslösen. Der Applicationserver antwortet daher mit einer HTML-Seite für einen Internal Server Error. Darüber hinaus ist diese Antwort nicht CORS-enabled, sodass der Fehler für den Aufrufer sogar wie ein CORS-Problem aussehen könnte.

Stattdessen sollte das Backend eine sinnvolle Antwort zurückgeben, z.B. den Status NOT FOUND und eine Fehlerbeschreibung im JSON-Format, wie {„msg“: „ID does not exist.“}

## Entkopplung der Tests von der Datenbank

UnitTests sollten immer nur kleinstmögliche Einheiten des Systems testen, indem sie erwartete und tatsächlich Ausgabe einer Methode vergleichen. Viele UnitTests sind dabei momentan vom DatabaseController abhängig. Fehler im Paket database würden daher Tests für die Pakete model und bean fehlschlagen lassen, was der Idee von UnitTests widerspricht. Stattdessen sollte der DatabaseController für diese Tests gemockt werden, z.B. mit Mockito, sodass tatsächlich nur die zu testende Methode getestet wird und nicht der DatabaseController.

## Live Search

Das Suchfeld in der Navigationsleiste sollte durch eine Live-Suche ergänzt werden. Ist unter Umständen nur dann sinnvoll, wenn der ganze Suchalgorithmus so abgeändert wird, dass er nicht mehr auf der MySQL Full Text Search basiert.

1. **J**ava**S**cript **O**bject **N**otation, ein gängiges Datenaustauschformat zwischen Programmiersprachen. Entspricht der Syntax einer Objektdefinition in JavaScript. [↑](#footnote-ref-1)