Gymnasium Leoninum Handrup | Hestruper Straße 1 | 49838 Handrup

Erstellung einer datenbankorientierten Web-Applikation

mit Ruby on Rails
am Beispiel eines Echtzeit-Browserspiels

Facharbeit im Seminarfach
„Webseiten selbst erstellen“
bei Ulrich Tönnies

von
Elias Kuiter

Webseite: elias-kuiter.de/facharbeit

Kurs sf256 | Schuljahr 2013/14

Inhaltsverzeichnis

1. Einleitung 1

2. Vorstellung der eingesetzten Technologien 2

2.1 Server-Programmierung 2

2.1.1 MySQL als Datenbank 2

2.1.2 Ruby 2

2.1.3 Ruby on Rails und MVC-Architektur 3

2.2 Client-Programmierung 4

2.2.1 Kommunikation mit dem Frontend: AJAX und JSON 4

2.3 Versionsverwaltung und Deployment 4

2.3.1 Gut organisiert: Git, GitHub und Gource 4

2.3.2 Bereitstellung mit Heroku 5

3. Schrittweise Entwicklung eines Browserspiels 5

3.1 Erste Schritte 5

3.2 Die Model-Ebene: Datenbankschema und Beziehungen 6

3.2.1 Die Spielidee 6

3.2.2 Modelle generieren 7

3.2.3 Beziehungen 8

3.2.4 Validierungen 9

3.2.5 Benutzung von Modellen 9

3.3 Die Controller-Ebene: JSON-Schnittstelle zum Spieler 10

3.3.1 Routing: Schnittstelle zur Außenwelt 10

3.3.2 Actions: Das Innenleben der API 11

3.4 Implementierung des Ressourcenkonzepts 12

3.4.1 Passive Rohstoffgewinnung in Echtzeit 12

3.4.2 Aktive Rohstoffnutzung 14

4. Bewertung 14

5. Anhang 16

5.1 Literatur 16

5.2 Anmerkungen 16

5.3 Zur Anfertigung des Browserspiels 19

1. Einleitung

In der heutigen Internetwelt, dem „Web 2.0“[1] sind interaktive und dynamische Web-Applikationen[2] stark vertreten[3]. Es zeichnet sich ein Trend ab, der von statischen Webangeboten Abstand nimmt, selbst eine zunächst rein informative Enzyklopädie wie Wikipedia stellt Möglichkeiten zur Kollaboration zur Verfügung. Aus diesem Grund sollte sich jeder Webentwickler mit der Erstellung dynamischer Webseiten vertraut machen. Dazu bieten sich Skriptsprachen wie PHP, Python oder Ruby an. Da ich besonderen Fokus auf den Aspekt „Datenbank“ lege und es sich für diese Art einer Web-Applikation sehr gut eignet, nutze ich in dieser Arbeit Ruby mit dem Framework „Ruby on Rails“ (dazu mehr in 2.1).

Um die Grundzüge der Rails-Programmierung an zunächst einfachen Beispielen zu erläutern, die im Detail aber eine steile Lernkurve aufweisen, habe ich mich für die Entwicklung eines serverbasierten[4] Browserspiels entschieden: Das bedeutet, dass mehrere Spieler zugleich auf demselben Server spielen und sich ihre Aktionen auf andere Spieler auswirken können. Für das hier entworfene Browserspiel gilt folgendes Aufrufmodell:

Demnach greift der *Spieler* auf unser Spiel namens *tranzfiction* über ein so genanntes **Frontend**[5] zu. Dies realisiert die grafische Darstellung und Illustration für den Spieler. Die Entwicklung und Dokumentation des Frontends übernimmt Simon Schröer, mit dem ich das Konzept des Spiels entwickelt habe. Das Frontend hat vor allem die Aufgabe, dem Nutzer ein gut aussehendes Spielerlebnis zu ermöglichen, die eigentliche Funktionalität wird jedoch an die *API*[6] und damit an das **Backend**[7] delegiert, welches dann die angeforderte Aktion ausführt, indem es mit der Datenbank kommuniziert, die die konkreten Spielerdaten enthält. Die API ist nichts anderes als eine Vereinbarung zwischen Front- und Backend darüber, wie bestimmte Aktionen ausgeführt werden.

Diesen Ablauf möchte ich kurz beispielhaft erläutern: Der Spieler öffnet das Frontend (tranzfiction.com). Das Frontend soll dem Spieler eine Liste seiner Städte anzeigen (zum Spielkonzept mehr in 3.2.1). Diese Liste wird nun geladen, indem das Frontend dem Backend sagt: „Gib mir eine Liste aller Städte dieses Spielers“. Technisch gesehen ruft das Frontend dabei die API-Funktion `/city`[8] auf. Ist der Nutzer eingeloggt, schlägt das Ba-

ckend die Städte in der Datenbank nach und gibt sie in einer für das Frontend lesbaren Form aus. Das Frontend kann nun mit diesen Daten die Städteliste für den Nutzer sichtbar befüllen. Nach diesem Prinzip laufen alle Aktionen im Spiel ab, sodass eine klare Trennung zwischen Geschäftslogik (Backend) und Präsentation (Frontend) zustande kommt.
Bevor das eigentliche Spiel entwickelt wird, folgt die Vorstellung einer Reihe von Technologien, die von *tranzfiction* genutzt werden.

2. Vorstellung der eingesetzten Technologien

2.1 Server-Programmierung

Die folgenden Technologien machen das *tranzfiction*-Backend aus, da sie direkten Zugriff auf die Datenbank haben und Daten manipulieren können.

2.1.1 MySQL als Datenbank

Für das Browserspiel wird ein Datenspeicher benötigt – schließlich müssen später Benutzerkonten, Städte, Gebäude etc. verwaltet werden. Als Speicher bietet sich eine relationale Datenbank an. Diese zeichnen sich dadurch aus, dass Informationen in Tabellen organisiert sind, deren Spalten verschiedene Daten angeben und deren Zeilen Datensätze sind:

id	email	password
1	1@test.com	e22[...]e37
2	2@test.com	0a4[...]c88

Diese Tabelle könnte „users“ heißen. Jeder *user* wird durch eine eindeutige *ID* angesprochen. Passwörter werden i. d. R. verschlüsselt.
Als Abfragesprache eignet sich SQL, im Speziellen wird hier das RDBMS[9] MySQL genutzt. Um jetzt den Nutzer mit der E-Mail-Adresse test@test.com herauszufiltern, schreibt man in SQL: `SELECT * FROM users WHERE email=„test@test.com“;` Nicht so in Ruby on Rails. Dort genügt `User.where email: „test@test.com“`. Noch deutlicher wird das an dem folgenden Beispiel: SQL: `SELECT * FROM users WHERE id=1;` Rails: `User.find 1`
(Das Befüllen der eigentlichen Datenbank folgt in 3.2.5.)

2.1.2 Ruby

Ruby ist eine „sehr mächtige, hochgradig objektorientierte Skriptsprache mit einer einfachen Syntax“, entwickelt 1995 von Yukihiro Matsumoto (Morsy/Otto, S. 57). Was an Ruby sofort hervorsticht, ist die unkonventionelle Syntax, vergleicht man mit C-ähnlichen Sprachen wie bspw. PHP:

Ruby

```ruby
(0..9).each do |number|
  print number
end
```

PHP

```php
for ($i = 1; $i <= 10; $i++) {
  echo $i;
}
```

Beide Beispiele geben 0123456789 aus. Allerdings ist der Ruby-Code wesentlich besser lesbar. Dies zieht sich durch die gesamte Programmiersprache und vereinfacht das Programmieren immens. Durch diese einfache Syntax lassen sich sogar ganze englische Sätze nachbilden, bspw. `expect(false).not_to be_true` oder `something.should be_a Object`[10].

2.1.3 Ruby on Rails und MVC-Architektur

Ruby on Rails (kurz „Rails“ genannt) von David Heinemeier Hansson ist ein so genanntes „Framework“, spezialisiert auf Webentwicklung (Morsy/Otto, S. 21). Das bedeutet, dass Rails viele typische Anforderungen und Vorgänge bereits mitbringt und dem Entwickler viel Arbeit abnimmt; gerade im Bereich der Datenbank (siehe 2.1.1). Rails verfolgt verschiedene Konzepte, die die Entwicklung erleichtern, darunter das Prinzip „Konvention statt Konfiguration“ – falls nicht anders angegeben, werden viele automatische Annahmen über die Applikation getroffen, was zu sehr kurzen und einfachen Applikationen führt (ebenda, S. 25)[11].

Wichtiger jedoch ist das MVC-Entwurfsmuster, welches eine Applikation in drei verschiedene Teile trennt (ebenda, S. 23):

Model	Controller	View
Das Model verwaltet die Daten in der Datenbank. Es sorgt auch dafür, dass Daten validiert werden („Passwort zu kurz“).	Der Controller ruft Daten vom Model ab und gibt sie der View. Über die URL können verschiedene Aktionen ausgelöst werden.	Die View zeigt eine Webseite an und bindet dazu dynamisch die Daten vom Controller ein.

Für *tranzfiction* sind Views eher irrelevant, da das Backend keine vollständigen Webseiten, sondern nur JSON (siehe 2.2.1) ausgeben soll. Die Rolle der View wird somit gewissermaßen vom Frontend übernommen. Hier wird der eingangs angesprochene Unterschied zwischen Geschäftslogik (Model und Controller) und Präsentation (View) deutlich.

2.2 Client-Programmierung

2.2.1 Kommunikation mit dem Frontend: AJAX und JSON

Oben wurde bereits angedeutet, dass das Frontend mit dem Backend kommuniziert. Dabei gibt es eine wichtige Anforderung: Diese Kommunikation muss im Hintergrund ablaufen, damit der Spieler in seinem Spielfluss nicht unterbrochen wird. Für diese Art der *asynchronen Kommunikation* wird eine Technik namens AJAX verwendet. Diese JavaScript-basierte Technologie ermöglicht, dass Front- und Backend Nachrichten austauschen[12]. JSON ist das Datenformat, mit dem das Backend auf eine Anfrage des Frontends antwortet. Wenn man bspw. eine Stadt mit der ID 1 mit der Funktion `/city/1` abruft, schickt das Backend als Antwort: `{"id":1,"name":"Teststadt","build_speed":1,"resources":{"silicon":2010, "plastic":565,"graphite":1860}}` In diesem Format ist – für JavaScript einfach verwertbar – alles Wissenswerte über eine Stadt hinterlegt.

2.3 Versionsverwaltung und Deployment

2.3.1 Gut organisiert: Git, GitHub und Gource

Jedes größere Programmierprojekt will organisiert und versioniert werden. Dazu existieren verschiedenste Systeme für Versionsverwaltung (VCS). Das meiner Meinung nach modernste und praktischste ist Git, entwickelt von Linus Torvalds zur Versionierung des Linux-Kernels (Chacon, S. 4). Immer wenn eine Änderung am Programm erfolgt, zeichnet Git diese auf. So können alle Änderungen an einem Projekt nachvollzogen und, wenn nötig, auch rückgängig gemacht werden. Git ist vor allem für kollaboratives Arbeiten gedacht, kann aber auch einem einzelnen Entwickler die Arbeit erleichtern.

In Kombination mit der Webseite GitHub spielt Git seine Stärken aus: GitHub ist eine bekannte Plattform zum Publizieren von (meist) Open Source-Programmcode. Auf GitHub kann jeder Nutzer Code einsehen, herunterladen oder mitgestalten. Deshalb nutzen viele bekannte Open Source-Projekte GitHub, darunter auch Ruby on Rails selbst[13]. Auch *tranzfiction* ist auf Github vertreten[14]. Nicht nur kann man dort den aktuellen Zustand eines Projektes einsehen, sondern auch die Entwicklung desselben nachvollziehen[15]. Als kleines Anwendungsbeispiel für Git dient das Projekt Gource[16], welches ein mit Git verwaltetes Projekt durch ein animiertes Video visualisiert. Als Beispiel dient auch hier *tranzfiction*[17].

2.3.2 Bereitstellung mit Heroku

Zunächst läuft die Rails-Entwicklung allein auf dem eigenen Rechner ab. Was aber, wenn man eine Rails-Applikation für die ganze Welt zugänglich machen will?

Es gibt zahlreiche Hosting-Seiten, die dies anbieten – der einzige Haken: Diese stellen meist eine enorme Belastung für die Geldbörse dar. Nicht so der Hoster Heroku: Hier gibt es die Möglichkeit, kleine Projekte kostenlos zu hosten (Morsy/Otto, S. 526). Der einzige Nachteil ist, dass die so gehosteten Webseiten nur wenige Nutzer gleichzeitig bedienen können, ohne stark zu verlangsamen; für *tranzfiction* soll dies jedoch zunächst genügen. Heroku arbeitet eng mit Git zusammen. Das bedeutet, dass zum Hochladen eines Projektes das Kommando `git push` genutzt wird. So wird man zum Nutzen der Versionsverwaltung Git gezwungen, was allerdings sehr praktisch ist, weil man so ein Versionschaos vermeiden kann[18].

3. Schrittweise Entwicklung eines Browserspiels

In diesem Kapitel wird die Programmierung des Browserspiels *tranzfiction* gezeigt. Den Anfang machen zunächst die Grundlagen, dann widme ich mich dem Kern des Spiels, der Datenbank, und schließlich folgt die Erläuterung der Schnittstelle zum Frontend.

3.1 Erste Schritte

Bevor ich zum eigentlichen Spielkonzept komme, möchte ich den Sprung ins kalte Wasser wagen und zeigen, wie eine Rails-Applikation erstellt, versioniert und ins Netz gestellt wird. Um zu beginnen, müssen auf dem Rechner diverse Programme installiert sein: Die Programmiersprache Ruby[19], das Framework Ruby on Rails[20], die Versionsverwaltung Git[21] sowie der Heroku Toolbelt[22]. Sind diese Tools alle installiert, kann es losgehen. Dazu müssen die folgenden Kommandos in einem Terminal ausgeführt werden[23]:

Das Rails-Projekt mit dem Namen *tranzfiction* wird erstellt: `rails new tranzfiction`. Es wird in das entstandene Verzeichnis gewechselt: `cd tranzfiction`. Ein Git-Repository wird erstellt: `git init`. Alle Projektdateien werden zum Git-Repository hinzugefügt: `git add -A`. Der jetzige Zustand wird in Git als neue Version festgehalten: `git commit -m 'Rails-Projekt erstellt'` (ebenda, S. 94/228f). Auf Heroku wird eine neue Webseite angelegt (mit Serverstandort in der EU): `heroku create --region eu`. Die noch

zufällig benannte Webseite wird nach *tranzfiction* umbenannt: `heroku apps:rename tranzfiction`. Die jetzige Version wird zu Heroku übertragen und ist damit auf tranzfiction.herokuapp.com erreichbar: `git push heroku master` (Morsy/Otto, S. 527f). (Mit DNS[24] habe ich dann eine Weiterleitung von api.tranzfiction.com auf tranzfiction.herokuapp.com eingerichtet.) Ruft man jetzt die genannte Webseite auf, erscheint eine Testseite von Rails.

Damit man zum Testen von Programmänderungen nicht ständig zu Heroku hochladen muss, bringt Rails einen Testserver mit, der mit `rails server` im Terminal aufgerufen wird. Dann kann man das Projekt im Webbrowser unter localhost:3000 erreichen (das Prinzip ähnelt dem von XAMPP für PHP) (ebenda, S. 215).

3.2 Die Model-Ebene: Datenbankschema und Beziehungen

3.2.1 Die Spielidee

Jetzt, wo das Fundament für das Backend steht, stehen einige theoretische Überlegungen zum Spielkonzept an. Bei *tranzfiction* soll es sich um einen vereinfachten Klon des bekannten Onlinespiels travian.de handeln.

Das Setting ist, dass jeder *Spieler* sich auf einem eigenen Planeten befindet, den er nach und nach mit *Städten* besiedeln kann. Jede Stadt enthält *Gebäude*, welche dem Spieler verschiedene Vorteile bescheren. Um neue Städte oder Gebäude zu errichten, sind *Rohstoffe* nötig. Es gibt drei unterschiedliche Rohstoffarten: Silizium, Kunststoff und Graphit. Diese können mit entsprechenden Gebäuden abgebaut werden. Als erschwerender Faktor kommt dazu, dass jedes Gebäude Energie benötigt, die ebenfalls durch bestimmte Gebäude bereitgestellt wird. Gebäude können, genug Rohstoffe vorausgesetzt, aufgestuft werden (*Upgrade*), wodurch der jeweilige Gebäudevorteil verbessert wird. Mit dieser groben Vorstellung im Hinterkopf kann nun ein ungefähres Datenbankmodell aufgestellt werden:

„Hat n“ drückt eine 1:n-Beziehung (eins zu n) aus. Das bedeutet, dass *ein* Spieler *mehrere* Städte hat (wobei die Zahl der Städte durchaus begrenzt ist). *Eine* Stadt hat *beliebig viele* Gebäude (vgl. ebenda, S. 296f).

Die schwarz beschrifteten Kästen werden *Models* genannt (siehe 2.1.3). Models werden in Datenbanktabellen gespeichert. Für das Modell Spieler (*User*) wird also eine Tabelle *users* angelegt. Jedes Model muss nun ausgearbeitet werden, daraus ergeben sich dann die Spalten für die Tabellen.

3.2.2 Modelle generieren

Welche Eigenschaften zeichnen einen *User* aus?

Ein User soll zunächst die Möglichkeit haben, sich einzuloggen. Dies habe ich mit dem Authentifizierungssystem *Devise* realisiert. Devise speichert verschiedene Nutzerdaten ab, vereinfacht kann man aber von `email` und `password` sprechen. Außerdem kann ein Spieler ein Administrator sein (oder eben nicht). Zudem befindet sich jeder Spieler auf einem eigenen Planeten. Aus diesen Überlegungen ergeben sich für das Model *User* die folgenden Spalten: `email` (`string`), `password` (`string`), `admin` (`boolean`), `planet` (`string`). *String* bedeutet, dass es sich um eine Zeichenkette wie *test@example.com* handelt. Ein *boolean* ist ein Wahrheitswert, es kann sich also um einen Admin handeln (`true`) oder nicht (`false`). Dieses Model kann nun auf der Konsole angelegt werden: `rails generate model User email:string password:string admin:boolean planet:string`. Der Doppelpunkt trennt Spaltennamen und Datentypen voneinander. Zusätzlich zu diesen Spalten erstellt Rails auch eine `id`-Spalte, die jeden User eindeutig identifiziert. Analog verfährt man nun mit allen anderen Models.

Eine Stadt (*City*) hat zunächst einen Namen. Außerdem ist jede Stadt genau einem Spieler zugeordnet (es handelt sich um eine 1:n-Beziehung). Dieser Bezug wird durch eine Spalte `user_id` realisiert. Diese verweist auf den User, dem die Stadt gehört. Zudem hat eine Stadt eine bestimmte Anzahl Rohstoffe (von jeweils Silizium, Kunststoff und Graphit), diese *Resources* werden allerdings in ein eigenes Model ausgelagert (das erleichtert später das Rechnen mit Rohstoffbeträgen). Das Resources-Model ist relativ einfach aufgebaut: Es gibt die Spalten `silicon`, `plastic` und `graphite`, die jeweils angeben, wie viel vom entsprechenden Rohstoff vorhanden ist. Außerdem gibt es noch eine Spalte, die auf eine Stadt verweist, nämlich `city_id` (ähnlich der `user_id` oben). Diese Spalte sorgt dafür, dass jedes Resources-Model einer bestimmten Stadt zugeordnet ist. Daraus ergeben sich die folgenden beiden Anweisungen: `rails generate model City name:string user_id:integer` sowie `rails generate model Resources silicon:decimal plastic:decimal graphite:decimal city_id:integer`.

Die beiden neuen Datentypen `integer` und `decimal` sind Zahlentypen. Da jede Stadt bzw. jeder User eine eindeutige ID wie bspw. 1, 2 oder 3 zugewiesen bekommt, lautet der dazu passende Datentyp `integer` (Ganzzahl). Der Typ `decimal` ermöglicht Nachkommastellen für Rohstoffbeträge.

Das letzte bedeutende Model ist das Gebäude (*Building*). Jedes Gebäude hat einen Typ – es kann sich bspw. um ein Rathaus handeln, oder um ein Forschungslabor etc. Dieser Typ wird in der Spalte `type` festgehalten. Außerdem hat jedes Gebäude eine Stufe (`level`), die die Effizienz des Gebäudes bestimmt. Nach dem Bau eines Gebäudes hat es das Level 1, dieses kann aber im weiteren Spielverlauf noch erhöht werden.
Damit ein Gebäude einer Stadt zugeordnet werden kann, wird wieder eine Spalte `city_id` benötigt (diese Verweise auf andere Tabellen werden im Übrigen Fremdschlüssel genannt).
Der Bau oder das *Upgrade* eines Gebäudes fordert vom Spieler zweierlei: Einerseits muss die Stadt, in der das Gebäude steht, genug Rohstoffe zur Verfügung stellen. Andererseits nimmt der Bauvorgang eine gewisse Zeit in Anspruch. Während dieses Vorgangs kann das Gebäude nicht genutzt werden. Um festzustellen, ob gerade ein Bauvorgang stattfindet, gibt es eine weitere Spalte, nämlich `ready_at`. Diese ist vom Typ `datetime`, mit dem eine Datums- und Zeitangabe gespeichert werden, und sie enthält den Zeitpunkt, wann ein etwaiger Bauvorgang abgeschlossen sein wird.
Daraus geht hervor: `rails generate model Building type:string level:integer city_id:integer ready_at:datetime` (vgl. Morsy/Otto, S. 298).
Damit die generierten Modelle auch in die Datenbank übernommen werden, muss noch ein `rake db:migrate` ausgeführt werden, und die Datenbasis des Backends ist fertiggestellt. Mit dem Ausführen der `generate`-Befehle hat Rails auch Dateien im Unterverzeichnis `app/models` erstellt. Dort kann das Verhalten einzelner Modelle verändert werden. Im Folgenden erläutere ich einige der wichtigsten Rails-Konzepte im Bezug auf Models.

3.2.3 Beziehungen

Die wichtigste Anpassung betrifft die Beziehungen unter den Models, die Rails noch explizit mitgeteilt werden müssen: In der Datei `building.rb` wird durch die Anweisung `belongs_to :city` die Zugehörigkeit eines Gebäudes zu einer Stadt festgelegt. Analog dazu wird in `city.rb` folgendes hinzugefügt: `belongs_to :user`. Außerdem hat eine Stadt viele Gebäude: `has_many :buildings`. Bei diesen Anweisungen fällt sofort ein großer Vorteil von Ruby auf, nämlich die eingängige Syntax: Um Rails mitzuteilen, dass ein User viele Städte hat, schreibt man einfach `has_many :cities` in der Datei `user.rb`. Dies macht den Code sehr gut les- und wartbar (vgl. ebenda, S. 299).

3.2.4 Validierungen

Der Spieler soll die Möglichkeit zur Selbstregistrierung haben, d. h. er gibt seine E-Mail-Adresse, Passwort und einen gewünschten Planetennamen an, um sich zu registrieren. Diese müssen allerdings an bestimmte Regeln gebunden sein. So darf eine E-Mail-Adresse nur einmal registriert werden. Auch das Passwort sollte eine Mindestlänge haben. Und ein Planet muss angegeben werden, sonst erscheint eine Fehlermeldung. All diese Anforderungen können durch *Validations* ausgedrückt werden (Morsy/Otto, S. 272f). Diese sind ähnlich einfach zu notieren wie die obigen Beziehungen. Für den Planetennamen kann dies bspw. so aussehen (in `user.rb`):

```
validates :planet, presence: true, length: { minimum: 3, maximum: 50 }
```

Dieser Code sorgt dafür, dass ein Planet angegeben werden muss und dieser nicht kürzer als 3, aber auch nicht länger als 50 Zeichen sein darf. Diese Validierungen schützen die Datenbank davor, mit falschen oder inkonsistenten Daten gefüllt zu werden. Bspw. kann man dank Validierung sicherstellen, dass die `email` des Spielers auch tatsächlich eine E-Mail-Adresse ist (ihre Korrektheit kann man damit aber natürlich nicht prüfen).

3.2.5 Benutzung von Modellen

Sobald Beziehungen und Validierungen festgelegt sind, sind die Models nutzungsbereit. Da weder Controller noch Views existieren, werden die Models zunächst über die Rails-eigene Konsole getestet, zu öffnen via `rails console`. Nun wird eine Rails-Umgebung geladen, in der normaler Ruby-Code interaktiv eingegeben werden kann (ebenda, S. 216).

Wie kann man jetzt auf ein Model zugreifen? Oder konkreter: Wie kann man bspw. einen neuen Benutzer erstellen? Dank der gründlichen Vorarbeit reicht nun ein einfaches `User.create email: "test@example.com", password: "meinpasswort", admin: false, planet: "Alpha Centauri"` aus. Außerdem kann man versuchen, einen invaliden Spieler zu erstellen, indem man z. B. die Planetenangabe weglässt. Danach kann man durch `User.all` eine Liste aller in der Datenbank gespeicherten Nutzer abrufen. Dabei stellt man fest, dass der invalide User wegen der Validierungen nicht in die Datenbank übernommen wurde. Auch `WHERE`-Bedingungen sind einfach zu realisieren. So gibt der Befehl `User.where admin: true` alle Administratoren zurück. Einen bestimmten User macht man mit `mein_user = User.find 1` ausfindig, wobei `1` durch die entsprechende ID ersetzt wird. Mit `mein_user.admin = true; mein_user.save` kann ein normaler Nutzer zum

Admin hochgestuft werden. So können auch andere Eigenschaften verändert werden. Eine Anweisung wie `mein_user.planet = ""; mein_user.save`, würde dank Validierung fehlschlagen, da kein Planetenname vorhanden wäre. Das Löschen von Datensätzen ist ebenfalls sehr einfach: `mein_user.destroy` (vgl. Morsy/Otto, S.112-115).

Aber auch die Beziehungen lassen sich sehr einfach nutzen. Mit `mein_user.cities.create name: "Meine Stadt"` erstellt man eine neue Stadt, die mit dem User `mein_user` assoziiert ist. Mit `mein_user.cities` lassen sich dann alle Städte eines Benutzers auslesen. Allerdings ist auch die andere Richtung möglich: Zu einer gegebenen Stadt `meine_city` kann der assoziierte Nutzer wie folgt ermittelt werden: `meine_city.user`. Was ergibt dann `mein_building.city.user`? Den Spieler, dem `mein_building` gehört. Und `User.all.first.cities.first.buildings.first`? Dies gibt das erste Gebäude in der ersten Stadt des ersten Benutzers zurück. Dementsprechend wäre `mein_building.city.resources.silicon` die vorrätige Siliziummenge in der Stadt, in der `mein_building` steht (vgl. ebenda, S. 299f).

Diese Beispiele illustrieren, wie einfach es ist, mit Rails auf Datenbankobjekte zuzugreifen, gerade wenn Beziehungen unter Modellen im Spiel sind.

3.3 Die Controller-Ebene: JSON-Schnittstelle zum Spieler

Bisher können Veränderungen an der Datenbank nur mit der interaktiven Rails-Konsole durchgeführt werden. Um nun eine nutzbare Webschnittstelle für das Frontend bereitzustellen, sind zunächst Überlegungen zum sogenannten *Routing* nötig.

3.3.1 Routing: Schnittstelle zur Außenwelt

Eine *Route* legt fest, welche Aktion Ruby on Rails ausführen soll, wenn unterschiedliche URLs[25] angesteuert werden. Sie entscheidet somit über die Zuordnung URL <=> Aktion, wobei jeder *Action* ein sogenannter *Controller* übergeordnet ist (siehe 2.1.3) (ebenda, S. 351). Ein Beispiel: Passend zum Model Building gibt es einen Controller, den *BuildingController*, der verschiedene Operationen, genannt *Actions*, bereitstellt; darunter *create*, um Gebäude zu bauen und *upgrade*, um dieselben zu verbessern. Diese Aktionen können in der Konfigurationsdatei `config/routes.rb` wie folgt einer URL zugeordnet werden:

```
get "/city/:city_id/building/create", to: "building#create"
get "/city/:city_id/building/:building_id/upgrade", to: "building#upgrade"
```

Der Einfachheit halber nutzt die *tranzfiction*-API nur die GET-Methode, die durch Eingabe einer URL im Browser nachgestellt werden kann[26]. In Anführungsstrichen folgt das URL-Schema, bei dessen Eingabe im Browser die entsprechende Route ausgelöst wird; dabei werden veränderliche Bestandteile (*Parameter*) mit Doppelpunkt `:` gekennzeichnet (bspw. gehört jedes Building einer City an). Nach dem Pfeil `=>` folgt die Angabe, was bei Auslösen der Route geschehen soll, nach dem Schema `controller#action`. An dieser Stelle findet also die eigentliche äußere Formgebung der API statt, so dass eine Schnittstelle zum Frontend entsteht. Sind alle Routen definiert, müssen die dazugehörigen Aktionen programmiert werden.

3.3.2 Actions: Das Innenleben der API

Aktionen sind Methoden, die in Controllern enthalten sind. Dank der gründlichen Vorarbeit in 3.2 gestaltet sich dies nun recht trivial. Beispielhaft folgt die *upgrade-Action* des *BuildingControllers*[27]:

```
class BuildingController < ApplicationController
  def upgrade
     city = current_user.cities.find params[:city_id]
     building = city.buildings.find params[:building_id]
     render json: building.upgrade!
  end
end
```

Hier enthält der *BuildingController* die Action *upgrade*, die beim Routing bereits mit einer URL verknüpft wurde. Die folgenden drei Anweisungen sorgen nun für das Gebäude-Upgrade: Zunächst wird mit der in 3.2.5 vorgestellten `find`-Methode die Stadt, die das Gebäude enthält, in der Städteliste des aktuellen Nutzers (`current_user.cities`[28]) gesucht. Dabei wird der Parameter `city_id` genutzt, der beim Routing erstellt wurde. Innerhalb dieser Stadt wird nun ebenfalls mit `find` nach dem richtigen Gebäude gesucht. Dessen Level wird nun via `building.upgrade!`[29] um eins erhöht, und eventuelle Fehler- oder Erfolgsmeldungen dieses Funktionsaufrufs werden durch die Methode `render` als JSON zum Client geschickt.

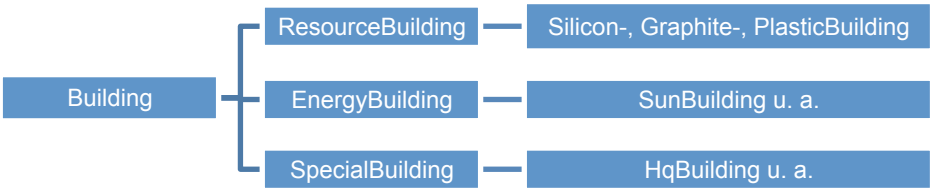
Nach ähnlichem Schema werden auch Aktionen wie *create* implementiert:

```
def create
  city = current_user.cities.find params[:city_id]
  building, success = Building.build city, params[:building]
  if success render json: building
  else       render json: building.errors, status: 400; end
end
```

Soll ein neues Gebäude gebaut werden, muss natürlich zunächst die beinhaltende Stadt ermittelt werden (s.o.). Anschließend wird mit `Building.build`[30] ein durch `params[:building]` charakterisiertes Gebäude in der Stadt `city` angelegt. Durch eine `if`-`else`-`end`-Fallunterscheidung wird für eine passende JSON-Ausgabe gesorgt. (Der Statuscode 400 steht dabei für *Bad Request*[31] und wird dann zurückgegeben, wenn bspw. kein korrekter Gebäudetyp in `params[:building]` angegeben wurde.)
Ein korrekter Aufruf dieser Action zum Bau eines Silizium-Werkes in Stadt 1 wäre demnach `/city/1/building/create?building[type]=SiliconBuilding`.

3.4 Implementierung des Ressourcenkonzepts

In der jetzigen Fassung liegt bereits eine nutzbare Rohfassung der API vor – diese kann durch Ausführen von `rails server` im Terminal unter localhost:3000 erreichbar gemacht werden und die API kann durch Eingabe der teilweise bereits vorgestellten URLs ausprobiert werden[32].
Allerdings fehlen noch einige Funktionen, die zum Spielgefühl beitragen sollen – so können in der aktuellen Version *bedingungslos* Gebäude gebaut werden, die allerdings noch keinerlei *Vorteile* bieten. Den Schlüssel zu diesen Konzepten stellen die *Gebäudetypen* dar – jedes Gebäude hat einen Typen, welcher sich in folgendes Schema einordnen lässt:



Daraus ergeben sich die beiden Aspekte Rohstoff- sowie Energiesystem; zusätzlich sollen die als *SpecialBuildings* gelisteten Gebäudetypen das Spielerlebnis auflockern. *ResourceBuildings* dienen zur Gewinnung der Rohstoffe Silizium, Graphit und Kunststoff. Diese können dann wieder beim Gebäudebau oder -upgrade ausgegeben werden. Die *EnergyBuildings* dienen dazu, den Energiebedarf der Gebäude zu decken. Als Teil der *SpecialBuildings* sorgt das *HqBuilding* (= Rathaus) für schnellere Bauzeiten. Es folgt die exemplarische Erläuterung der Rohstoffmechanik.

3.4.1 Passive Rohstoffgewinnung in Echtzeit

In 3.2.2 wurde bereits das Modell *Resources* vorgestellt, das mit den drei Spalten Silizium, Graphit und Kunststoff und einer Stadt-Zuordnung aufwartet. Dieses Modell ist vergleichbar mit einem Zahlentripel oder Vektor, denn es kann mit ihm gerechnet werden: `Resources.new(silicon: 200, gra-`

`phite: 200, plastic: 200) - Resources.new(silicon: 100, graphite: 75, plastic: 50)` ergibt `{"silicon":100,"graphite":125,"plastic":150}`.

Neben Addition und Subtraktion werden auch Vergleiche wie =, ≠, <, >, ≤ oder ≥ unterstützt. Dies wird durch die sogenannte *Operatorüberladung* in Ruby ermöglicht und es erleichtert die folgenden Rechen- und Vergleichsoperationen. Jede Stadt ist mit einem solchen Rohstoffmodell ausgestattet (durch eine `has_one`-Beziehung (Morsy/Otto, S. 304)). Die Idee ist, dass jedes gebaute *ResourceBuilding* zur Rohstoffproduktion beiträgt. So produziert ein *SiliconBuilding* auf der ersten Stufe 30 Silizium-Einheiten pro Stunde. Wird es auf Stufe 2 ausgebaut, sind es 40 Einheiten usw.

Entscheidend ist hier die Überlegung, dass die Rohstoffproduktion ein passiver Vorgang ist, der nicht an eine bestimmte Aktion des Spielers gekoppelt ist: Der Server soll in einem bestimmten Zeitintervall (bspw. jede Minute) die Städte aller Spieler durchgehen, um diesen die neu verdienten Rohstoffe anzurechnen. Dazu bietet es sich an, eine neue API-Funktion (`/gain`) anzulegen, die genau dies tut und minütlich aufgerufen wird[33]. Die im *WorkerController* abgelegte Action *gain* sieht folglich so aus:

```
def gain
  since_last_gain = ::Configuration.since_last_gain
  City.each do |city|
    city.buildings.where(type: "SiliconBuilding").each do |building|
  city.resources.silicon+=building.gain_per_hour/(3600.0/since_last_gain)
     end
  end
  ::Configuration.last_gain = Time.now
end
```

Interessant ist hier im Wesentlichen die mittlere Zeile, die für jedes *SiliconBuilding* ausgeführt wird[34]. Hier wird der Siliziumvorrat erhöht: Die bekannte stündliche Zunahme (`gain_per_hour`) wird so geteilt, dass für den verstrichenen Zeitraum in Sekunden (`since_last_gain`) passendes Silizium addiert wird. Am Ende wird noch die Zeit aktualisiert, so dass *tranzfiction* immer der Zeitpunkt des letztens Rohstoffabbaus bekannt ist. Die stündliche Zunahme hängt vom Gebäudelevel ab. Sie steigt exponentiell, also bietet sich zu ihrer Errechnung eine entsprechende Funktion an[35]:

```
def gain_per_hour
  23 * Math.exp(0.25 * level).round -1
end
```

Es handelt sich hierbei um die (auf Zehnerstellen gerundete) Ruby-Umschreibung der Exponentialfunktion $f(x) = 23 * e^{0.25x}$. Dadurch lassen

sich lange Rohstofftabellen vermeiden und man erhält eine passend ansteigende Funktion (Level 1: $f(1) \approx 30$, aber Level 20: $f(20) \approx 3410$).
Sind all diese Hürden erst einmal genommen, steht dem Rohstoffabbau nichts mehr entgegen; ohne einen Gegenspieler in Form des Rohstoffverbrauchs ist dies jedoch zunächst nicht besonders sinnvoll.

3.4.2 Aktive Rohstoffnutzung

Das Verbrauchen von Rohstoffen ähnelt dem Abbau, nur dass Rohstoffmengen subtrahiert werden. Was beide voneinander unterscheidet, ist der Zeitpunkt. Während der Abbau passiv und automatisch abläuft, wird der Verbrauch aktiv vom Nutzer angefordert und muss geprüft werden.
Es gibt zwei Situationen der Rohstoffnutzung, den Bau und das Upgrade von Gebäuden (`Building.build(...)` und `my_building.upgrade!`). In beiden Fällen wird überprüft, ob genug Rohstoffe für den Vorgang zur Verfügung stehen. Umgesetzt am Beispiel von `upgrade!` ergibt das folgenden Code:

```
def upgrade!
  if city.resources >= upgrade_resources
    level += 1
    city.resources -= upgrade_resources
  else
    errors.add :missing_resources, upgrade_resources.subtract_to_zero(city.resources)
  end
end
```

Falls die vorhandenen Rohstoffe die benötigten übersteigen, wird alles Nötige zum Upgrade veranlasst und die verbrauchten Rohstoffe entfernt; im Fehlerfall werden die noch fehlenden Rohstoffe ausgegeben. Die benötigten `upgrade_resources` errechnen sich auch hier durch von Gebäude zu Gebäude unterschiedliche Funktionen (bspw. das Rathaus mit $f(x) = \binom{55}{30}{45} * e^{0.25x}$), nur dass hier drei Funktionen für jeweils einen Rohstoff vorliegen. An dieser Stelle kann man mit Koordinaten und Exponenten experimentieren, um die gewünschte Spielbalance zu erreichen.

4. Bewertung

Obwohl *tranzfiction* im aktuellen Stadium keineswegs als fertig bezeichnet werden kann, ist doch deutlich, wohin die Reise geht – eine einfach wartbare Datenbankapplikation mit einer vergleichsweise kleinen Codebasis. Die noch fehlenden Features (wie Energie- und Spezialgebäude sowie

Kompetition mit anderen Spielern) lassen sich dank der zuverlässigen Modellstruktur recht einfach ergänzen.
So kann man der API nach und nach immer mehr Aktionen hinzufügen, bis eine voll funktionsfähige Schnittstelle zum Frontend entsteht. Bei diesem Vorgehen wird deutlich, wo die Stärken von Ruby on Rails liegen: Rails' Fokus liegt eindeutig auf der Datenorganisation und -manipulation durch Modelle, während die Controller und das Routing die Schnittstelle zu diesen (im Idealfall) gut ausgearbeiteten Modellen darstellen. Dadurch kann sich der Programmierer stärker mit den für eine Datenbankapplikation entscheidenden Thematiken wie Validierungen und Relationen auseinandersetzen, was die Produktivität und Codequalität steigert.
Alle anderen Vorgänge, z. B. der eigentliche Auslieferungsvorgang der Webseiten sowie die Parameterübergabe werden von Rails stark vereinfacht, so dass überzeugende Ergebnisse schon mit wenig, dadurch zudem wartbaren Code erzielt werden können. Dies ist auch der Grund, warum Rails seine Stärken vorrangig in datenbankbasierten Applikationen ausspielt, da diese auf Modellen aufbauen.
Aus persönlicher Erfahrung kann ich nur zwei Nachteile bei der Nutzung von Rails nennen: Zum einen die unzureichenden Möglichkeiten, eine Applikation günstig im Internet bereitzustellen – in der Tat ist Heroku der einzig praktikable Anbieter – sowie die langsame Geschwindigkeit bei kostenlosen Angeboten[36]. Dies liegt nicht zuletzt daran, dass Ruby on Rails ein sehr umfangreiches Framework ist, das aus vielen ineinandergreifenden und dementsprechend langsamen Schichten besteht.
Ein weiterer Aspekt, den es zu beachten gilt, sind die Anforderungen an das eigene Projekt – wenn nur einige dynamische Inhalte in eine statische Webseite eingefügt werden sollen, ist sicherlich PHP besser als Rails geeignet. Gleiches gilt bspw. für Applikationen, die mit anderen Protokollen arbeiten, wie ein mit Node.js realisierter IRC-Chat. Skriptsprachen bzw. Frameworks sprechen unterschiedliche Zielgruppen an, deshalb sollte man das beste Werkzeug für jedes Projekt individuell bestimmen.
Wie so oft gilt es also, sich auf unterschiedliche Anforderungen flexibel einstellen zu können – sollten diese jedoch den in dieser Arbeit vorgestellten Charakter einer dynamischen Webapplikation mit Datenbankanbindung aufweisen, ist Ruby on Rails genau das richtige Framework zur Realisierung des Projektes.

5. Anhang

5.1 Literatur

Morsy, Hussein / Otto, Tanja (2012): Ruby on Rails 3.1. Das Entwickler-Handbuch. 2. Auflage.

Chacon, Scott (2009): Pro Git. Everything you need to know about the Git distributed source control tool. PDF von github.s3.amazonaws.com/media/progit.en.pdf (22.02.2014).

5.2 Anmerkungen

Die Anmerkungen enthalten neben Quellenangaben auch klärende Informationen zu technischen Begriffen. Bei Internetquellen ist auch der Zeitpunkt des Abrufs angegeben, außerdem stehen diese auf der Webseite der Facharbeit (elias-kuiter.de/facharbeit) als Screenshot zur Verfügung.

[1] Das Web 2.0 ist keine technische Revolution (wobei Skriptsprachen wie Perl, PHP oder Ruby natürlich Voraussetzung für diese Entwicklung waren). Es bezeichnet vielmehr eine Umorientierung des WWW von einem rein informativen Angebot hin zu aktiven Mitwirkungsmöglichkeiten für den Internet-Nutzer. Auch die neuen „Social Media“-Trends wie Facebook und Twitter verfolgen den Web-2.0-Gedanken.
Nach wirtschaftslexikon.gabler.de/Definition/web-2-0.html (M1, 08.02.14)

[2] Applikation ist ein Synonym für Programm oder Software. Eine Web-Applikation ist somit eine Software, die im Internet läuft, und grenzt sich durch ihre dynamischen Funktionalitäten wie Nutzer-Authentifizierung oder Datenbankmanipulation von normalen, statischen Webseiten ab.

[3] siehe alexa.com/topsites/countries/DE (M2, 08.02.14), wo die beliebtesten Webseiten Deutschlands aufgeführt werden, darunter nur Anwendungen, die mit dem Nutzer interagieren (Google) oder ihn mitwirken lassen (Facebook).

[4] Ein Server bezeichnet den Computer, der eine Webseite ausliefert. Der Computer, der die Webseite erhält, wird Client genannt. Server und Client ermöglichen also ein Zusammenspiel ähnlich dem der Paketzustellung mit Absender und Adressat.

[5] Ein Frontend ist die sichtbare Oberfläche einer Webapplikation. Das Frontend von *tranzfiction* ist auf tranzfiction.com zu finden.

[6] API = Application Programming Interface, also eine Schnittstelle, in diesem Fall zum Backend von *tranzfiction*.

[7] Wenn das Frontend die Karosserie ist, ist das Backend der Motor einer Applikation. Bei *tranzfiction* ist es auf api.tranzfiction.com zu erreichen.

[8] API-Funktionen werden in dieser Arbeit verkürzt als `/funktion` geschrieben. Ausgeschrieben bedeutet das: api.tranzfiction.com/funktion. Die API-Funktion `/city` wird also so aufgerufen: api.tranzfiction.com/city

[9] Relationales Datenbankmanagementsystem. MySQL ist sehr verbreitet, da sehr einfach zu benutzen; eine Alternative ist bspw. PostgreSQL. Aus praktischen Gründen nutze ich hier MySQL.

[10] Diese Art der Programmierung kommt beim Testen zum Tragen. Auch *tranzfiction* wurde mit dieser Methode getestet. Die entsprechenden Tests (genannt *Specs*) sind zu finden unter github.com/ekuiter/tranzfiction-backend/tree/master/spec.

[11] Weiterhin interessant ist das DRY-Konzept („Don't Repeat Yourself"), das besagt, dass Redundanzen, also doppelter Code vermieden werden sollen (à la: `x.color = "#fff"; y.color = "#fff"`, besser wäre: `white = "#fff"; x.color = y.color = white`) (Morsy/Otto, S.25).

[12] Wie AJAX en détail funktioniert, ist Thema meiner Arbeit *Interaktive Webanwendungen mit AJAX* (dev.elias-kuiter.de/ajax/Interaktive Webanwendungen mit AJAX.pdf), eine Übersicht findet sich auch bei Morsy/Otto, ab S. 505. Wie es beim Spiel *tranzfiction* zum Einsatz kommt, beleuchtet Simon Schröer in seiner Facharbeit.

[13] github.com/rails/rails

[14] github.com/ekuiter/tranzfiction-backend. Der Quelltext kann auch als komprimierte ZIP-Datei bezogen werden:
github.com/ekuiter/tranzfiction-backend/archive/master.zip

[15] Versionsübersicht von *tranzfiction*:
github.com/ekuiter/tranzfiction-backend/commits/master

[16] code.google.com/p/gource

[17] Das mit Gource animierte Video steht ebenfalls auf GitHub zur Verfügung: github.com/ekuiter/tranzfiction-backend/raw/gource/gource.mp4

[18] Bei PHP beispielsweise besteht das Deployment oft nur daraus, dass man seine PHP-Dateien auf einen FTP-Server kopiert. Aus eigener Erfah-

rung kann ich sagen, dass dies schnell zum Problem werden kann: Wenn man Dateien überschreibt, die eigentlich noch benötigt werden; oder wenn man ein Update durchgeführt hat, welches nur noch mehr Probleme verursacht, aber nun nicht mehr rückgängig zu machen ist – für all diese Probleme bietet Git entsprechende Lösungen an.

[19] Unter Windows bietet sich dazu der RubyInstaller (rubyinstaller.org) an, andere Möglichkeiten werden auf www.ruby-lang.org/de/installation geschildert.

[20] Dazu genügt, wenn Ruby korrekt installiert wurde, ein `gem update` gefolgt von `gem install rails` in einem Terminal (unter Windows die Eingabeaufforderung `cmd.exe`).

[21] git-scm.com/downloads

[22] toolbelt.heroku.com

[23] Bei Windows dazu die Tasten `Win+R` drücken, `cmd.exe` eingeben und die Enter-Taste betätigen.

[24] Das Domain Name System (DNS) sorgt dafür, dass Webadressen wie google.de in IP-Adressen wie `173.194.113.152` umgesetzt werden.

[25] URL = Uniform Resource Locator (ungenau „Webadresse“), diese werden bei *tranzfiction* zum Aufruf der API-Funktionen genutzt.

[26] In produktiven Webapplikationen ist eine andere Technik namens *REST* üblich – dort werden dann auch andere HTTP-Methoden wie POST, PUT oder DELETE genutzt (POST wird bspw. auch beim Absenden von Formularen genutzt.)

Die *tranzfiction*-API ist allerdings bewusst nicht *RESTful* gehalten (denn alle API-Aufrufe finden mit der HTTP-Methode GET statt): Ohne REST lassen sich die API-Aufrufe viel einfacher im Browser testen. Aus demselben Grund werden die Parameter für den jeweiligen Aufruf als GET-Parameter übermittelt. Dadurch wird die API anschaulicher und kann auch beispielhaft erläutert und ausprobiert werden. (In einer produktiven Web-Anwendung sollte man stattdessen auf REST zurückgreifen.)

[27] Dieses und die folgenden Programm-Listings sind stellenweise vereinfacht dargestellt (bzgl. Sicherheit, Codequalität- und duplikaten), um die Leitidee besser transportieren zu können. Die praktische Implementierung kann im Quelltext von *tranzfiction* nachvollzogen werden.

[28] `current_user` ist eine Hilfsmethode, die das Authentifizierungssystem *Devise* zur Verfügung stellt. Sie stellt das zum aktuell angemeldeten Nutzer gehörende *User*-Objekt bereit; falls keiner angemeldet ist, wird automatisch eine entsprechende Anmeldeseite angezeigt.

[29] Zur Implementierung der Methode `upgrade!` siehe 3.4.2.

[30] Die Implementierung von `Building.build(...)` ähnelt der von `upgrade!`.

[31] vgl. HTTP/1.1-Spezifikation: tools.ietf.org/html/rfc2616#section-10.4.1 (M3, 14.03.14)

[32] Eine vollständige Auflistung aller API-Funktionen steht, falls ein lokaler Testserver läuft (`rails server` auf der Konsole), unter localhost:3000/api zur Verfügung (M4).

[33] Realisiert wird dies durch einen sogenannten *Cronjob*, ein Dienst, der zeitgesteuert Webseiten abruft (M5).

[34] Beispielhaft wurde hier Silizium gewählt, für die anderen Rohstoffe ist das Vorgehen analog. Außerdem wurde in diesem Listing des Platzes wegen auf das Abspeichern (`save`) der Rohstoffe verzichtet.

[35] Diese und alle weiteren Exponentialgleichungen wurden per Regression mit den Datentabellen von t4.answers.travian.de/index.php?aid=189, 193, 195 (M6-8) als Vorlage ermittelt. Die Tabellen werden in einem Online-Regressions-Tool wie xuru.org/rt/ExpR.asp eingegeben und in eine Gleichung überführt (M9). Details dazu: github.com/ekuiter/tranzfiction-backend/blob/master/app/models/buildings/resource/silicon_building.rb

[36] Ähnliches gilt zwar für PHP, allerdings können bei PHP auch durchaus mehrere Projekte auf einem günstigen virtuellen Server betrieben werden, während bei Rails ein Server immer einem Projekt entspricht.

5.3 Zur Anfertigung des Browserspiels

Zur Implementierung des *tranzfiction*-Backends habe ich viele Internetquellen herangezogen, darunter hauptsächlich Code-Ausschnitte, die ich übernommen habe sowie Anleitungen für bestimmte Zusatzprogramme.

Zusatzsoftware

Authentifizierung: github.com/plataformatec/devise

Backend-Design: railsapps.github.io/twitter-bootstrap-rails.html

AJAX domainübergreifend zulassen: github.com/cyu/rack-cors, enable-cors.org, www.d-mueller.de/blog/cross-domain-ajax-guide
Unicorn-Webserver: devcenter.heroku.com/articles/rails-unicorn
u.a., siehe github.com/ekuiter/tranzfiction-backend/blob/master/Gemfile

Anleitungen

Administratoren und reguläre Nutzer unterscheiden: github.com/plataformatec/devise/wiki/How-To:-Add-an-Admin-role
Session domainübergreifend zulassen: excid3.com/blog/sharing-a-devise-user-session-across-subdomains-with-rails-3/
AJAX-Login mit Devise: natashatherobot.com/devise-rails-sign-in/, stackoverflow.com/questions/19707539

Code-Ausschnitte

Deutsche Übersetzung von Devise: gist.github.com/almays/3909491
Deutsche Rails-Übersetzung: github.com/svenfuchs/rails-i18n/blob/master/rails/locale/de.yml
Tooltips erstellen: qtip2.com/download, jsfiddle.net/craga89/L6yq3, github.com/jeremyfa/yaml.js

Testen mit Rspec

Sumner, Aaron (2014): Everyday Rails Testing with Rspec. A practical approach to test-driven development.
Tipps zum Testen: betterspecs.org
Specs mit Devise: github.com/plataformatec/devise/wiki/How-To:-Controllers-tests-with-Rails-3-(and-rspec)
Controller testen: blogs.evergreen.edu/jbolton/blog/2011/10/17/testing-controllers-with-rspec-in-rails
Zeiten vergleichen: stackoverflow.com/questions/3559924/rails-time-inconsistencies-with-rspec