Schriftliche Ausarbeitung der Besonderen Lernleistung im Fach Informatik

Helmholtz-Gymnasium Essen

Für das Abitur 2019

Feedback — Entwicklung und Gestaltung eines modernen Ticketsystems

Autor:
Julian Koch

Fachlehrer:
Dennis Großkamp

11. April 2019

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Über einen Zeitraum von ungefähr sechs Monaten habe ich zusammen mit einem Partner ein voll funktionsfähiges Ticketsystem bestehend aus einem WebSocket-Server und einem Client in Form einer Browser-Erweiterung entwickelt (im Folgenden "Feedback"). Dies geschah im Rahmen des GFOS Innovationsawards 2017, einem Wettbewerb zwischen mehreren teilnehmenden Schulen, welcher durch das Unternehmen GFOS Gesellschaft für Organisationsberatung und Softwareentwicklung mbH organisiert wurde.

Im Folgenden stelle ich den Entwicklungsprozess der Softwarelösung dar und dokumentiere dessen Architektur. Zusätzlich erläutere ich einige Code-Fragmente, die ich hervorheben möchte; beispielsweise solche, die ein ungewöhnlich schwierig zu lösendes Problem betreffen.

Hierzu werden einige Code-Beispiele im Text eingebunden. Aufgrund der Größe des Quellcodes ist es unmöglich, diesen komplett zu inkorporieren, jedoch liegen der digitalen Ausgabe dieses Dokuments der vollständige Code sowie alle weiteren genutzten Materialien bei.

1 Einleitung & Motivation

Feedback richtet sich an auf die Entwicklung von Softwarelösungen spezialisierte Unternehmen. Jegliche Art von Software ist im Zeitalter der rapiden digitalen Progression anfällig für Fehler (sogenannte *Bugs*), welche u.a. aus sich verändernden Einsatzbedingungen, unvorhergesehenen Use-Cases der Software oder während der Entwicklung nicht betrachteter Eventualitäten resultieren können.

Aufgrund der Komplexität vieler Softwarelösungen ist ein formaler Beweis der Korrektheit eines Produkts oft nicht realisierbar. Somit ist es möglich, dass Kunden, die die betroffene Software nutzen, ein Bug auffällt. Ein Ticketsystem bietet in solchen Fällen die Möglichkeit, den Fehler in Form eines *Bug Reports* bzw. *Tickets* zu melden. Diese können von Mitarbeitern des Unternehmens eingesehen, bearbeitet und schließlich gelöscht werden.

Feedback ist eine moderne Implementation eines solchen Ticketsystems mit vielen weiteren nützlichen Features. Hierzu zählen ein modernes User Interface (UI) orientiert an den von Google entwickelten Material Design Guidelines, eine flüssige und durchschaubare, doch reiche Nutzererfahrung (UX), ein hochmodernes, ausgereiftes *Ticket Dashboard* sowie ein sicherer und performanter Java-basierter Webserver.

2 Vorstellung des Projekts

Diese Sektion dient der Präsentation der Projektergebnisse in Form eine Kurzbeschreibung der wichtigsten UI-Elemente und Nutzerinteraktionen mit dem Ziel, dem Leser einen Überblick über die finale Version von Feedback zu gewähren.

2.1 Der Server

Ein Unternehmen, welches Feedback als Ticketsystem verwenden möchte, benötigt zunächst die `server.jar`-Datei sowie das OS-abhängige Startskript (`start.bat` für Windows oder `start.sh` für Unix-Systeme).

Um den Server nun zu starten, wird das Startskript ausgeführt, entweder durch einen Doppelklick oder im Terminal (Mac) bzw. Command Prompt (Windows). In der Folge öffnet sich ein Konsolenfenster, welches während der Laufzeit aktuelle Informationen anzeigt sowie als Eingabeaufforderung für Kommandos fungiert. Alle in dem Fenster erscheinenden Textzeilen sind ebenso in `log/latest.log` nachlesbar. Des Weiteren wird diese Datei nach Serverstopp komprimiert und in einen `.gz`-Archiv gespeichert, dessen Namen das aktuelle Datum beinhaltet.

Beim ersten Start erscheinen im Konsolenfenster sofort mehrere Nachrichten, welche in Abb. 1 gezeigt sind. Diese werden im Folgenden näher betrachtet.

```
1 [13.15.21.746+0200] [main/WARN]: Failed to find data
    configuration: data\products.txt
2 [13.15.21.748+0200] [main/WARN]: Using default values...
3 [13.15.21.750+0200] [main/INFO]: Starting server on port
    8080...
4 [13.15.21.890+0200] [Ticket-Loader-Thread/INFO]: Loading
    tickets from storage...
5 [13.15.21.892+0200] [Ticket-Loader-Thread/INFO]: Done loading
    (0) tickets!
6 [13.15.22.824+0200] [Server-Thread/INFO]: Server started, end
    with "end" (without quotation marks)
```

Abbildung 1: Nachrichten beim Erststart

1. Die Warnungen in Zeile 1 und 2 zeigen an, dass bisher keine Produkte konfiguriert worden sind.

2. Zeile 3 signalisiert die Initiierung des Startvorgangs des Servers auf dem angegebenen Port.

3. Zeilen 4 bis 5 geben Auskunft über den Abrufungsprozess etwaiger erstellter Tickets. Beim Erststart existieren natürlich keine; bei einer

großen Anzahl an Tickets kann dieser Prozess jedoch ein signifikantes Zeitintervall in Anspruch nehmen und wird daher asynchron ausgeführt.

4. Die letzte Zeile wird ausgegeben, wenn der Server mit allen seinen Funktionalitäten einsatzbereit ist und Verbindungen von Clients akzeptiert.

Der Server sollte nun mithilfe des `end`-Kommandos gestoppt werden. Hierbei werden einige Dateien erstellt, die nun bearbeitet werden können, um den Server zu konfigurieren.

2.2 Der Client—Ticketerstellung

Möchte ein Nutzer ein neues Ticket erstellen, benötigt er zunächst die Feedback Chrome Browser-Erweiterung. Nach der Installation kann er mit einem Klick auf das Feedback Icon in der oberen-rechten Ecke des Browsers das hierfür genutzte Pop-Up aufrufen. Nach erstmaliger Nutzung erscheint eine Fehlermeldung (Abb. 2), welche aussagt, dass keine Verbindung zu einem Feedback Server aufgenommen werden konnte.

Nun richtet der Nutzer einmalig den Server ein. Die hierfür benötigten Daten sollten auf der Website des Unternehmens zu finden sein, welchem er Feedback geben möchte.

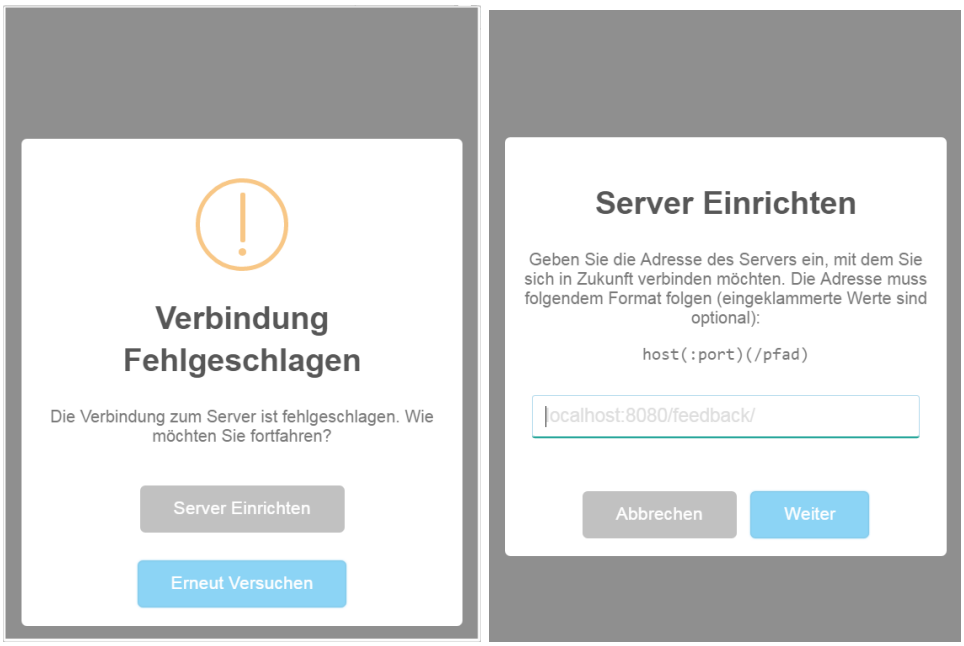


Abbildung 2: Links: Fehlermeldung nach fehlgeschlagenem Verbindungsversuch; rechts: UI zur Servereinrichtung

Ein neuer Server kann stets über das Einstellungsmenü eingerichtet werden, welches über das Zahnrad in der oberen-rechten Ecke des Pop-Ups erreichbar ist.

Im Anschluss ist es nun möglich, ein neues Ticket zu erstellen (alle weiteren Felder werden nach einer nichtleeren Eingabe in das erste Feld geöffnet) oder sich als Administrator anzumelden (Abb. 3).

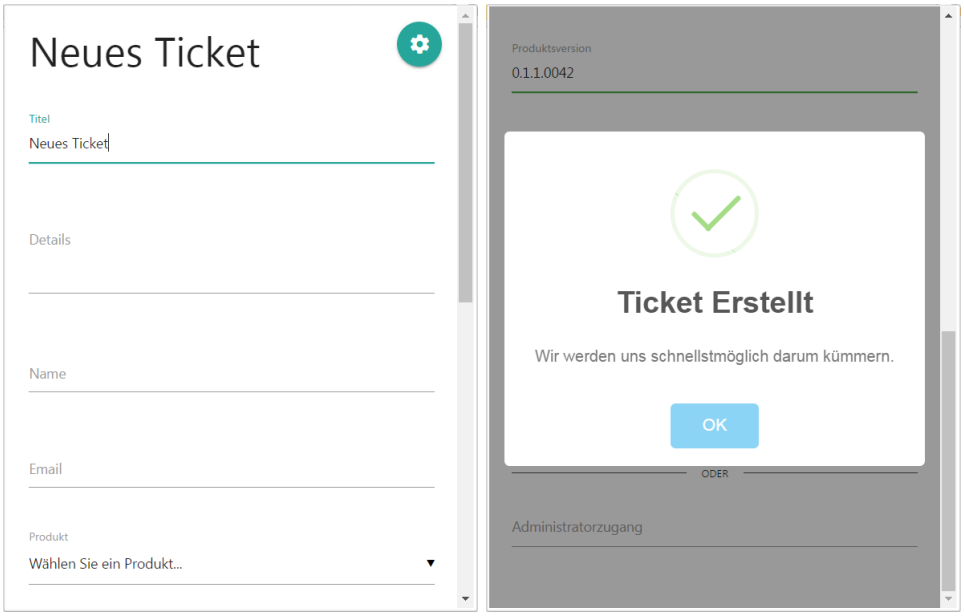


Abbildung 3: Links: UI zur Ticketerstellung; rechts: nachfolgende Bestätigung

2.3 Der Client—Administrator-Dashboard

Authentifizierten Administratoren steht ein Dashboard zur Ticketverwaltung zur Verfügung, das *Ticket Scrumboard*. Um dieses zu öffnen, muss der Administrator ebenfalls die Chrome-Erweiterung installiert haben. Um sich anzumelden, öffnet er das Pop-Up und gibt den Login-Code im zweiten sichtbaren Textfeld ein und bestätigt mit Enter.

Das nun gezeigte Scrumboard bietet eine Übersicht aller aktuellen Tickets. Es kann mithilfe des "Filter"-Feldes nach Tickets mit bestimmtem Inhalt, mithilfe des "Programme"-Dropdowns nach jenen mit einem der ausgewählten Programme assoziierten, und mithilfe des "Tags"-Feldes nach solchen mit spezifischen Tags gesucht werden. Per Drag-and-Drop kann jedes Ticket in eine andere Spalte verschoben und somit dessen Status geändert werden. Alle anderen Änderungen sind durch den Ticket Editor durchführbar, welcher sich durch das Anklicken eines Tickets öffnen lässt und in Abb. 4 gezeigt ist.

Alle Änderungen werden sofort dem Server übermittelt und gespeichert. In ähnlicher Weise werden auch alle externen Änderungen, die der Server etwa von anderen verbundenen Administratoren empfängt, an den Client übermittelt und das Scrumboard aktualisiert. Sobald die Internetverbindung unterbrochen wird, erscheint eine modale Nachricht, die das Scrumboard unnutzbar macht. Dies sorgt dafür, dass keine Ungewissheit über den Speicherstand des Scrumboards besteht.

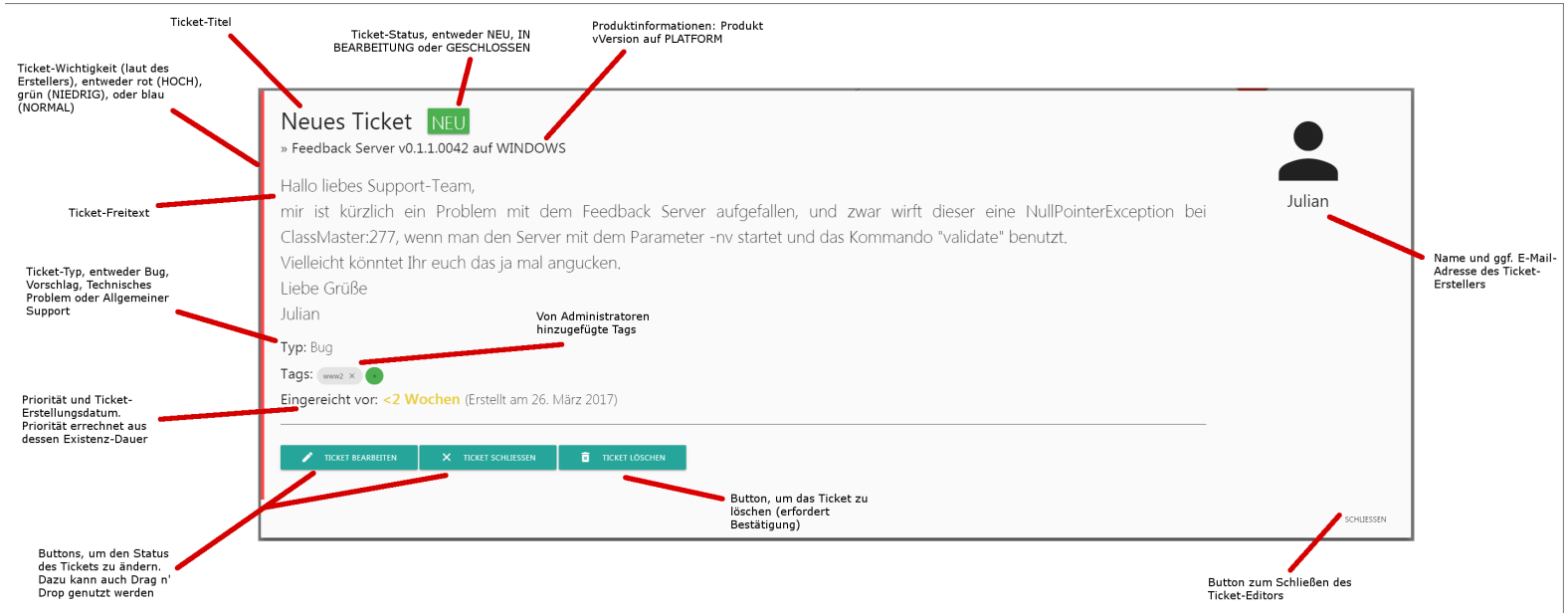


Abbildung 4: Ticket Scrumboard

3 Feedback: Design

3.1 Server & Client

Feedback besteht aus einem Webserver und einem Client in Form einer Browser-Erweiterung. Der Server ist so konzipiert, dass eine oder mehrere Instanzen gleichzeitig auf einer oder verschiedenen Maschinen laufen können, welche nach Möglichkeit nicht auszuschalten oder zu unterbrechen sind. Während der Lebensdauer einer Serverinstanz akzeptiert dieser eingehende Feedback-Verbindungen auf einem konfigurierten Port, authentifiziert Clients und bearbeitet und beantwortet Anfragen.

Konvers ist ein Client nur bei dessen Nutzung als aktiv zu betrachten. Wird das Pop-Up bei einem Klick auf das Feedback-Icon geladen, etabliert der Browser eine Verbindung mit dem eingestellten Server. Sollte dieser Versuch fehlschlagen, wird eine Fehlermeldung angezeigt und der Nutzer darauf hingewiesen, die korrekte vom jeweiligen Unternehmen bereitgestellte Adresse anzugeben[1].

Wurde die Verbindung zum Server hergestellt, beantragt der Client verschiedene Informationen wie eine Liste der vom Unternehmen konfigurierten Produkte, die mit Tickets assoziiert werden können. Diese Daten sind zur Erstellung des UIs des Pop-Ups nötig; nach Herunterladen dieser wird die Verbindung zum Server jedoch sofort wieder geschlossen, um diesen zu entlasten und somit zu einer geringeren Latenz (Verzögerung) anderer Clients beizutragen.

Der Nutzer kann nun ein neues Ticket erstellen, indem er die notwendigen Daten angibt und die Eingabe bestätigt. Dann wird erneut eine Verbindung zum Server aufgebaut und die Daten an diesen gesendet. Der Server verarbeitet diese und erstellt ein entsprechendes Ticket. Nach der erhaltenen Bestätigung schließt der Client die Verbindung und meldet dem Nutzer, dass das Ticket erstellt wurde.

Dieses Design steht im Gegensatz zu dem des Ticket Scrumboards: Da dieses im durchschnittlichen Fall häufig User-Aktionen in kurzen Zeitabständen registriert, ist es sinnvoll, eine persistente Verbindung aufzubauen, da das Versenden einzelner Aufträge vom Client weniger Ressourcen kostet als das wiederholte Herstellen einer Verbindung und somit die Latenz des Scrumboards verringert und im Effekt die UX verbessert wird.

3.2 Sicherheit

Ein weiterer wichtiger Vorteil einer bestehenbleibenden Verbindung ist die Fähigkeit des Servers, Informationen zu dem verbundenen Client zu speichern, wie dessen Login-Daten. Dies ermöglicht das Ausführen von Aktionen

[1] Diese Einstellung kann jederzeit auch bei erfolgreich hergestellter Verbindung in dem Pop-Up über das Rad-Icon oben rechts geändert werden.

ohne das erneute Versenden des Account-Passworts; stattdessen wird lediglich ein vom Server generierter *Session-Token* an jede Nachricht angehängt, der vor dem Nutzer verborgen bleibt und ein erhöhtes Maß an Sicherheit garantiert.

Das Administratorkennwort ist eine wichtige, sicherheitskritische Ressource, und Feedback folgt dem Prinzip, dessen Sicherheit als höchste Priorität anzuerkennen. Aus diesem Grund wird es nicht im *Plaintext* (in roher Form) gespeichert, sondern ausschließlich dessen Hashwert. Ein idealer Hashwert einer Datenmenge berechnet sich durch eine Funktion H, für die keine Umkehrfunktion existiert, von dem enkodierten Wert also nicht auf das ursprüngliche Passwort geschlossen werden kann. Um dieses dennoch zu berechnen, müsste es in einer *Brute-Force-Attacke* erraten werden. Es folgt, dass eine sichere Hashfunktion eine signifikante Berechnungszeit aufweisen muss, um solche Attacken undurchführbar zu gestalten oder zumindest extrem zu verlängern. Ebenso sollte diese mit der Entwicklung effizienterer CPUs gesteigert werden.

Des Weiteren ist die perfekte Hashfunktion injektiv, da mit der Anzahl der Kollisionen $|\mathbf{K}|$ ebenfalls die Wahrscheinlichkeit bei einer Brute-Force-Attacke steigt, mit einem Passwort x, das nicht das Originalkennwort $\bar{x}$ ist, trotzdem als Administrator authentifiziert zu werden. Die genaue Relation wird folgend hergeleitet:

$$
\begin{aligned}
\mathbf{X} &= \{x_1, x_2, \ldots\} &&, \\
\mathbf{M} &= \{(x_1, x_2) \mid x_1, x_2 \in \mathbf{X}\} &&, \\
\mathbf{K} &= \{(x_1, x_2) \mid x_1, x_2 \in \mathbf{X} \ \wedge \ H(x_1) = H(x_2)\} &&, \\
w &= P\left(H(x) = H(\bar{x})\right) \\
&= P\left((x, \bar{x}) \in \mathbf{K}\right) \\
&= \frac{|\mathbf{K}|}{|\mathbf{M}|} = \frac{|\mathbf{K}|}{|\mathbf{X}|^2}
\end{aligned}
$$

Gibt es keine Kollisionen, so ist $|\mathbf{K}| = |\mathbf{X}|$ und folglich $w = 1/|\mathbf{X}|$ gleich der Wahrscheinlichkeit, das Originalpasswort zu erraten; liegt eine schlechtestmögliche Hashfunktion vor, die für alle Passwörter denselben Hashwert liefert, ist $|\mathbf{K}| = |\mathbf{M}|$ und somit $w = 1$.

Abb. 5 zeigt einen beispielhaften Wahrscheinlichkeitsbaum zur Illustration dieser Berechnung.

Besteht ein Passwort $x = z^m$ aus der Verkettung von $m \in \mathbb{N}$ Zeichen $z \in \mathbf{Z}$, so ist $|\mathbf{X}| = |\mathbf{Z}|^m$ und in der Folge $|\mathbf{M}| = |\mathbf{Z}|^{2m}$.

bcrypt. Feedback nutzt bcrypt als Hashfunktion, um das Administratorpasswort zu verschlüsseln. Dieser Algorithmus liefert einen Hashwert mit einer Größe von 184 Bits, sodass es 2^{184} mögliche Werte gibt. Es wird davon ausgegangen, dass bcrypt perfekt *diffus* ist, also die verfügbaren Hashwerte

gleichmäßig an alle Input-Werte verteilt. Ferner wird von Passwörtern der Länge 10 ausgegangen, mit dem lateinischen Alphabet der Größe 26, sodass $|\mathbf{X}| = 26^{10}$.

In diesem Fall werden 26^{10} Werte 2^{184} möglichen Hashwerten zugeordnet, wobei $2^{184} \gg 26^{10}$, sodass es keine Kollisionen gibt und also $w = 26^{-10}$. Selbst bei Passwörtern der Länge 30 und einem Alphabet mit 64 Symbolen ist $64^{30} < 2^{184}$; die Wahrscheinlichkeit einer Kollision in bcrypt ist also extrem gering.

Allgemein folgt (unter Annahme fast perfekter Diffusion), da $|\mathbf{X}|$ Passwörter 2^{184} Hashwerten zugeteilt werden müssen[2]:

$$|\mathbf{K}| \approx \max\left\{ \frac{|\mathbf{X}|^2}{2^{184}}\,,\, |\mathbf{X}| \right\} \quad .$$

Für $|\mathbf{X}| > 2^{184}$ ist

$$w = \frac{|\mathbf{K}|}{|\mathbf{X}|^2} \approx \frac{|\mathbf{X}|^2/2^{184}}{|\mathbf{X}|^2} = \frac{1}{2^{184}} \quad .$$

Es ist erkennbar, dass die Wahrscheinlichkeit w, mit einem zufälligen Passwort als Administrator authentifiziert zu werden, unabhängig von der Menge der möglichen Passwörter ist. Dies folgt ebenfalls aus der Überlegung, dass mit der Anzahl der Passwörter die der zu probierenden Kombinationen sowie der Kollisionen steigt; diese beiden Effekte "gleichen sich aus".

Bei einer Brute-Force-Attacke, in welcher N Versuche gestartet werden, administrativen Zugang zu erhalten, kann die Erfolgswahrscheinlichkeit durch die Modellierung des Problems als Binomialverteilung errechnet werden (die Zufallsvariable S ist die Anzahl der *Erfolge*, also gelungenen Login-Versuche):

$$\begin{aligned} & P(S \geq 1) \\ &= 1 - P(S = 0) \\ &= 1 - \binom{N}{0} \cdot w^0 \cdot (1-w)^{N-0} \\ &= 1 - \left(1 - \frac{1}{2^{184}}\right)^N \end{aligned}$$

Um mit einer Wahrscheinlichkeit von mindestens 50% eingeloggt zu werden, müssen $N \geq 4{,}877\,59 \cdot 10^{13}$ (fast fünfzig Billionen) Versuche gestartet werden. Bei einer mittleren Versuchsdauer von einer Sekunde bedeutet dies ungefähr 15 000 Jahrhunderte.

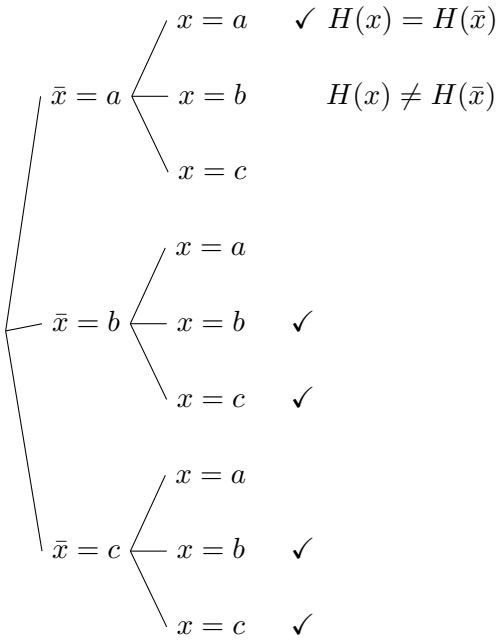
[2] All diese Folgerungen habe ich selbstständig hergeleitet, weshalb keine Quellenangabe platziert ist.

Wörterbuchangriffe. Das Passwort wird nur als Hashwert gespeichert, damit es bei einem Angriff und Exposition der Passwortdatei weiterhin geschützt bleibt. Hier reicht es jedoch nicht aus, die Hashfunktion *nur* auf das Passwort anzuwenden: es existieren zahlreiche frei verfügbare Tabellen (Bsp.[3]), welche die Hashwerte vieler Standardpasswörter wie "1234", "abc", etc. beinhalten. Da nicht davon ausgegangen werden kann, dass ein Nutzer ein sicheres Passwort wählt, welches nicht in einer solchen Liste enthalten ist, müssen weitere Sicherheitsmaßnahmen implementiert werden.

Salt. Beim *Salting* wird dem Plaintext-Passwort ein zufällig generierter String angehängt, welcher nebst dem hieraus resultierenden Hashwert gespeichert wird. Ferner hat dies zur Folge, dass die Hashwerte duplizierter Passwörter in einer Datenbank trotzdem verschieden sind, da das Salt für jedes Kennwort neu erstellt wird. In bcrypt is dieser Sicherheitsfaktor "automatisch" integriert.

[3] `md5decrypt.net/en/` enthält nach eigenen Angaben über zehn Milliarden Einträge des MD5-Algorithmus

x	a	b	c
$H(x)$	1	2	2



$$\mathbf{X} = \{a, b, c\}$$
$$\mathbf{M} = \{(a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c)\}$$
$$\mathbf{K} = \{(a,a), (b,b), (c,c), (b,c), (c,b)\}$$
$$w = \frac{|\mathbf{K}|}{|\mathbf{M}|} = \frac{5}{9}$$

Bei einem zufällig ausgewählten Passwort $\bar{x} \in \mathbf{X}$ beträgt die Wahrscheinlichkeit, dass ein weiteres zufälliges Passwort $x \in \mathbf{X}$ denselben Hashwert hat, $5/9 \approx 56\%$.

Abbildung 5: Beispielhafte Wahrscheinlichkeitsberechnung des Erfolgs einer Brute-Force-Attacke

4 Feedback: Architektur

Das Feedback-Projekt lässt sich im Groben in zwei Unterprojekte einteilen, die wiederum mehrere kleinere Komponenten inkorporieren.

Das erste Unterprojekt ist der in HTML, CSS und JavaScript geschriebene Client, welcher in Abschnitt 4.4 näher betrachtet wird.

Das zweite Unterprojekt ist der Webserver von Feedback (genannt Feedback Server). Ausgeliefert wird dieser in Form eines JAR-Archivs (eingepackt in einer ZIP-Datei nebst Installationsanleitung und Startskripts für Windows und Unix-Systeme). Hierin sind alle genutzten Bibliotheken (inklusive deren transitiven Dependenzen und Lizenzen) eingeschlossen, sodass keine Daten mehr heruntergeladen werden müssen. Der Quellcode des Servers ist in dem Archiv nicht beinhaltet, da er für Nutzer zumeist irrelevant ist.

Feedback Server ist hauptsächlich in der objektorientierten Programmiersprache Java (Version 1.8) geschrieben. Die wichtigste genutzte Bibliothek ist *Netty*, eine "low-level" Server-API, welche eingehende TCP-Verbindungen akzeptiert, das WebSocket-Protokoll implementiert und über dieses verschickte "Nachrichten" (*Frames*, siehe RFC 6455, Abschnitt 5) übersetzt, also die binären Daten in Java-Objekte umwandelt (oder umgekehrt).

Feedback Server arbeitet mit diesen Objekten, um sauberen und lesbaren Code zu ermöglichen und dessen Flexibilität und Erweiterbarkeit zu garantieren.

Der Quellcode des Servers besteht aus 95 *.java*-Dateien mit einer Summe von 8015 Zeilen. Davon sind 846 Zeilen Kommentare, also ca. 11%.

4.1 Das WebSocket-Protokoll

Einer der ersten Schritte in der Entwicklung war die Implementation des WebSocket-Protokolls. Dieses bietet einen großen Vorteil gegenüber traditionellen HTTP AJAX-Anfragen: Verbindungen werden dauerhaft aufrecht erhalten, und beide Parteien können unabhängig voneinander Nachrichten verschicken.

Anfangs war ein völlig selbstständig entwickelter WebSocket-Server angedacht. So beschäftigte ich mich intensiv mit RFC 6455, welches das Protokoll spezifiziert. Dieses wird im folgenden Absatz übersichtlich und vereinfacht dargestellt.

Der erste Schritt bei der Etablierung einer WebSocket-Verbindung ist ein *Connection Upgrade* initiiert durch den *Opening Handshake* gesendet von dem Client. Nun können dieser und der Server beliebige Frames an den anderen senden. Diese beinhalten einen *opcode*, welcher den weiteren Aufbau des Frames sowie die Art der im Frame gesendeten Informationen bestimmt. Es können *Control Frames* verschickt werden, welche den Verbindungspartner "anpingen" (um z.B. sicherzustellen, dass die Verbindung weiterhin besteht) oder anweisen können, die Verbindung zu schließen.

Nach etwa einer Woche konnte sich unser primitiver Test-Client mit unserem selbstgeschriebenen Server verbinden; jedoch wurde uns angesichts der bereits aufgewendeten Zeit bewusst, dass eine eigene Implementation des WebSocket-Servers außerhalb des zeitlichen Rahmens war. Somit entschieden wir uns nach Rücksprache mit einem Betreuer des Wettbewerbs zur Nutzung der Bibliothek *Netty*.

Netty assoziiert mit jeder bestehenden Verbindung (genannt *Channel*) eine *Pipeline*, in welcher eine Reihe von *Handler*n festgelegt ist[4] . Diese Handler werden bei jeder ein- und ausgehenden Nachricht aufgerufen. Es bestehen Unterklassen, welche nach der Richtung der zu bearbeitenden Nachricht "filtern", Frames (de-)komprimieren oder die zwischen binären Daten und POJOs (Plain Old Java Object) übersetzen.

Feedback Server konfiguriert Netty so, dass nach aufgebauter Verbindung ein standardmäßiger Handler der Pipeline hinzugefügt wird, welcher auf Authentifikation des Clients wartet und im Anschluss entweder durch einen normalen oder administrativen Handler ersetzt wird. Ersterer ist in der Lage, Anfragen wie das Anfordern der Produktliste oder das Erstellen eines neuen Tickets zu bearbeiten; letzterer beantwortet alle Anfragen, für die administrative Rechte (also das Administratorpasswort) benötigt werden. Der Channel Initializer ist in Abb. 6 dargestellt.

4.2 Die Logging-API

Feedback nutzt eine selbstgeschriebene Logging-API, welche mehrere Vorteile gegenüber der standardmäßig in Java integrierten bietet. Besonders hervorzuheben ist hier die Fähigkeit, den Namen des aktuellen Threads im Output anzuzeigen; diese Möglichkeit besteht bei dem Standard-Logger nicht und war ausschlaggebend für die Entscheidung, ein eigenes Logger-Framework zu nutzen.

Der Logging-Mechanismus ist grundlegend als Pipeline von `LoggingHandler` Instanzen strukturiert. Die Klasse `Logger` umfasst eine Liste solcher Objekte, die bei jedem Aufruf einer beliebigen Logging-Methode ein Nachrichtenobjekt erstellt und dieses an alle registrierten Handler weiterleitet.

Um die Implementation eines Handlers zu vereinfachen und redundantem/dupliziertem Code entgegenzuwirken, wird jede Nachricht in einem `LogRecord`-Objekt eingefasst, welches den aktuellen Zeitpunkt und Thread, die geloggte Nachricht als String, ggf. eine übergebene Exception sowie das `LogLevel` (die Wichtigkeitsstufe) der Nachricht einbindet.

Mehrere `LoggingHandler`-Implementationen transformieren und speichern eingehende Nachrichten auf verschiedene Weise.

- `FormattingLoggingHandler` ist eine abstrakte Unterklasse und konvertiert `LogRecord`s mithilfe eines konfigurierbaren `LoggingFormatter`s in Strings,

[4] Die Initialisierung erfolgt in einem *Channel Initializer*

```
(...).childHandler(new ChannelInitializer<SocketChannel>()
{
  protected void initChannel(SocketChannel ch)
      throws Exception
  {
    /* Initialisiere die ChannelPipeline jeden
     * neu verbundenen Channels mit folgenden
     * ChannelHandlern, in der Reihenfolge:
     */
    ch.pipeline().addLast(
      // Dekodiere ByteBufs zu HttpRequests und HttpContents
      new HttpRequestDecoder(),
      // Aggregiere HttpRequests und HttpContents
      // zu je einer ganzen FullHttpRequest
      new HttpObjectAggregator(65536),
      // Enkodiere ausgehende Nachrichten in ByteBufs
      new HttpResponseEncoder(),
      // Implementation des WebSocket Protokolls,
      // einschließlich Control Frames usw.
      new WebSocketServerProtocolHandler(getWebsocketPath()),
      // Haupt-Authentifikationshandler (fügt richtigen
      // weiterführenden Handler hinzu)
      new FeedbackAuthHandler(Server.this)
    )
      // Handler für ausgehende Nachrichten,
      // der Metadaten (u.a. Client IP) zu Debug-Zwecken loggt
      .addLast(
        // Handlername: Klassenname mit "_"-Suffix,
        // um Verwirrung vorzubeugen
        FeedbackLoggingOutboundHandler.class.getName() + "_",
        new FeedbackLoggingOutboundHandler()
      );
  }
});
```

Abbildung 6: Feedback Servers ChannelInitializer

die er wiederum seiner konkreten Implementation übergibt. Ein LoggingHandler heißt im Folgenden formatierend, wenn er von dieser Klasse erbt.

- `ConsoleLoggingHandler` ist formatierend und gibt Nachrichten in der Konsole aus.
- `FileLoggingHandler` ist formatierend und speichert Log-Nachrichten kontinuierlich in einer bestimmten Datei.
- `ArchivingFileLoggingHandler` delegiert alle Logging-Anfragen an einen `FileLoggingHandler` und komprimiert und speichert beim Schließen des Handlers die Datei in einem GZIP-Archiv.

Der `ArchivingFileLoggingHandler` implementiert einen Namensgebungsalgorithmus, um sicherzustellen, dass ein bereits bestehendes Archiv nicht überschrieben wird. In diesem befindet sich eine leere und somit scheinbar sinnfreie `while`-Schleife (siehe Abb. 7), welche jedoch ein interessantes Coding-Paradigma demonstriert.

```
1 int i = 0;
2 while ((archiveFile =
3     new File(archive + "_" + ++i + ".log.gz"))
4     .exists());
```

Abbildung 7: Namensgebungsalgorithmus des `ArchivingFileLoggingHandler`s

Es werden hier zwei kontraintuitive Eigenschaften der Sprache Java genutzt, um den Code kompakt und effizient zu halten, ohne erfahrenen Programmieren das Lesen zu erschweren. Erstens fällt die Präfixinkrementierung der Indexvariable `i` auf: `++i` bedeutet, dass der Wert von `i` um 1 erhöht und das Ergebnis zurückgegeben wird. Zweitens kommt die Eigenschaft der Sprache zum Vorschein, dass Variablenzuweisungen auch den zugewiesenen Wert zurückgeben, welcher in der `while`-Bedingung genutzt wird.

Dieser Algorithmus generiert also sukzessive Namen in der Form `..._1.log.gz`, `..._2.log.gz`, ...

4.3 Data Access Object–Pattern

Datenpersistenz ist ein kritischer Bestandteil jeder Applikation, die über einen Neustart hinweg Daten speichern muss, wie beispielsweise konfigurierte Werte oder angelegte Benutzerkonten. Hierzu existieren verschiedene Mechanismen. In Feedback kann zwischen zwei unterschiedlichen Datenspeicherungsmethoden gewählt werden: das Nutzen mehrerer lokaler Dateien ist performanter, jedoch auf eine einzige Serverinstanz begrenzt, während eine

(MySQL-)Datenbank eine höhere Latenz aufweist, allerdings auf einem externen Server liegen kann und oft weitere Sicherheitsfaktoren besitzt, wie automatische Backups der Daten.

Das Entwurfsmuster *Data Access Object* (DAO) entkoppelt die Implementation des Datenpersistenzmechanismus von der Use-Site. Dies ermöglicht, dem Liskovschen Substitutionsprinzip entsprechend, das Austauschen dieser Implementation, um beispielsweise eine virtuelle Persistenzebene in Testszenarien anzulegen oder zwischen einer Datenbank- und einer klassischen, dateibasierten Lösung während der Laufzeit wechseln zu können.

Ein DAO-Interface definiert zumeist "higher-level"–Funktionen, welche POJOs akzeptieren und zurückgeben, anstelle von simplen CRUD-Operationen (*Create, Read, Update, Delete*). Je abstrakter die vom Interface definierten Methoden, desto weniger Kopplung entsteht, und desto mehr Flexibilität besitzen die verschiedenen Implementationen, die Daten in für den jeweiligen Persistenzmechanismus angemessener Weise zu enkodieren.

In Feedback liegt eine leicht abgewandelte Version dieses Entwurfsmusters vor: ein DAO-Interface (die abstrakte Klasse `StorageAccessInfo`) agiert als Wrapper-Klasse für ein schließbares `StorageAccess`-Objekt, um die Verbindung zur Datenbank bzw. die Datei nicht unnötig offenzuhalten. (Eine bessere Implementation wäre jedoch die komplette Entkopplung dieses Mechanismus und dessen Umwandlung in eine intrinsisch in die DAO-Implementation integrierte Funktionalität.)

4.4 Die Chrome-Erweiterung

Die Entwicklung einer Chrome-kompatiblen Browser-Erweiterung stellte eine weitere Herausforderung dar. Dies liegt in den spezifischen Anforderungen an das Format der Erweiterungsdatei begründet. Ich nutzte die offiziellen Google Extension Development Ressourcen (Link [5]), um diese Anforderungen zu erlernen und implementieren zu können.

4.4.1 Format der Erweiterung

Abb. 8 zeigt die `manifest.json`-Datei der Chrome-Erweiterung, welche sich im Root-Verzeichnis dieser befindet. Hierin stehen Metainformationen, die die Erweiterung beschreiben, wie deren Name, Version, Autor, Sprache, usw. Des Weiteren werden die *permissions* (Berechtigungen) der Extension spezifiziert, also die Funktionsbereiche der Chrome-API, auf die während der Laufzeit zugegriffen werden darf. Bei der Installation jeder Erweiterung muss der Nutzer die einzelnen angeforderten Berechtigungen bestätigen.

Feedback benötigt lediglich zwei distinkte Berechtigungen: `management`, um die von Chrome zugewiesene ID der Erweiterung abfragen und in der Konsole ausgeben zu können, um vom Nutzer kopier- und einsetzbare Links

[5] Direktlink: `https://developer.chrome.com/extensions/getstarted`

```
{
    "manifest_version": 2,
    "name": "Feedback",
    "version": "0.1.0",
    "default_locale": "de",
    "description": "Feedback-Client zum Erstellen und
        Verwalten von Tickets",
    "author": "Julian B. Koch",
    "permissions": [
        "storage",
        "management"
    ],
    "browser_action": {
        "default_icon": "logo/16x.png",
        "default_popup": "popup.html"
    },
    "icons": {
        "16": "logo/16x.png",
        "48": "logo/48x.png",
        "128": "logo/128x.png"
    }
}
```

Abbildung 8: manifest.json

generieren zu können; sowie `storage`, welche Zugriff auf die internen Speicherungsmechanismen Chromes gewährt. So wird zum Beispiel der Destinationsserver persistent gespeichert, sodass er nicht nach jedem Neustart Chromes erneut eingegeben werden muss und außerdem zwischen einzelnen Chrome-Instanzen, in denen der Nutzer eingeloggt ist, synchronisiert werden kann.

4.4.2 Clientseitige Bibliotheken

Zur konsistenten Anwendung des Material Designs nutzt Feedback die Materialize CSS-Library; jQuery bietet eine einfache und elegante Schnittstelle zur dynamischen Bearbeitung und Generierung von HTML. Eine weitere nützliche Library ist *SweetAlert*, die Browser-native Dialoge mit elegant gestalteten Elementen ersetzt. Die Nutzung von SweetAlert ist beispielhaft in Abb. 9 aufgeführt.

```
swal({
  title: "Passwort Fehlerhaft",
  text: "Das Passwort ist fehlerhaft. Sollten Sie das " +
        "Passwort vergessen haben, kontaktieren Sie bitte " +
        "einen Server-Administrator.",
  type: 'error'
}, function () {
  // Lösche Inhalt des Passwort-Felds
  $('#adminin-password').val('');
  // Weise Materialize an, Textfelder zu aktualisieren
  Materialize.updateTextFields();
});
```

Abbildung 9: Dialogfeld nach Antwort des Servers, die das eingegebene Passwort als falsch angibt

5 Der JSON-Parser—Compiler-Design

Feedback Server beinhaltet einen selbstgeschriebenen JSON-Parser, der Text im populären JSON-Format in eine programmatisch navigierbare Baumstruktur übersetzen kann.

Das Parsen ist in drei Phasen eingeteilt: Tokenisierung, Syntaktische Analyse und Übersetzung in POJOs.

5.1 Tokenisierung

In der ersten Phase, auch *Lexing* genannt, wird die auszuführende Datei Zeichen für Zeichen gelesen und in einen ausgehenden Fluss von *Tokens* übersetzt. Diese beinhalten folgende Informationen:

- **Lexem.** Die Zeichenkette, aus der dieser Token generiert wurde.
- **Typ.** Der Typ eines Tokens gibt Auskunft über die Natur des zugrundeliegenden Lexems. Beispielsweise beschreibt ein Token des Typs `STRING` eine beliebige, in Anführungszeichen eingeschlossene Zeichenkette, während `OBJECT_START` eine geöffnete geschweifte Klammer meint. Eine Liste aller Typen findet sich in der Klasse `EnumJsonTokenType`.
- **Wert.** Die Java-Repräsentation des Wertes, beispielsweise ein `String` für eine Zeichenkette oder ein `double`.

Hier werden nur wenige Fehler entdeckt, wie illegale Symbole oder nicht abgeschlossene Strings.

Im Folgenden wird zunächst der Lexing-Algorithmus allgemein spezifiziert und anschließend im Kontext dessen Implementation erläutert.

Der Lexer arbeitet nach dem *Maximal Munch*–Prinzip: es wird stets eine maximale Anzahl an Zeichen eingelesen, aus denen ein valider Token generiert werden kann. Bei hiernach weiterhin bestehenden Konflikten (wenn etwa ein Lexem zwei distinkten Typen zugeordnet werden kann, das Einlesen eines weiteren Zeichens jedoch zur Invalidierung beider Typzuweisungen führt) wird derjenige Token-Typ gewählt, der nach einer bestimmten Anordnungsfunktion zuerst definiert ist.

Formal wird dies folgend mathematisch beschrieben. Es seien Z das Alphabet der einzulesenden Zeichen; n die Zahl der bereits eingelesenen Zeichen; $\vec{s} \in Z^n$ der Vektor der eingelesenen Zeichen; T die Menge aller Token-Typen; $i : T \to \mathbb{N}$ eine injektive Abbildung mit $i^{-1}(i(t)) = t$ für alle $t \in T$, die jedem Token-Typen einen einzigartigen Index zuordnet; $b : (T, Z^n) \to \{0, 1\}$ eine Wahrheitsfunktion, die aussagt, ob aus einer Zeichenkette ein Token eines bestimmten Typs generiert werden kann; $f \in \{0, 1\}$ genau dann 1, wenn zu mindestens einem Zeitpunkt galt: $M \neq \emptyset$; und $M = \{t \mid t \in T \land b(t, \vec{s}) = 1\} \subseteq T$ die Menge aller Token-Typen, aus denen ein Token mit den bereits eingelesenen Zeichen generiert werden kann.

Anfangs sind $n = 0$, $\vec{s} = \vec{0}$, $f = 0$ und $M = \emptyset$. Iterativ werden nun n inkrementiert und $\vec{s}$, M und f aktualisiert, bis gilt: $f = 1 \wedge M = \emptyset$. Hierauf wird der vorausgegangene Zustand der Variablen wiederhergestellt, und es wird ein Token mit dem Typen $t \in T$ zurückgegeben, für den gilt: $t \in M \wedge i(t) = \min\{i(\bar{t}) \mid \bar{t} \in M\}$.

Mit diesem (oder einem Algorithmus, der nach demselben Prinzip funktioniert) ist es möglich, beinahe jede Sprache zu lexen[6].

Implementation in Feedback. Der Lexing-Algorithmus ist in der Klasse `JsonLexer` implementiert. Bei Aufruf der `nextToken`-Methode wird erst Whitespace (d.h. Leerzeichen, Tabs, Neuzeilen, etc.) entfernt.

Hierauf liest der Lexer das nächste Zeichen und entscheidet auf Basis dessen, welche Symbole als nächstes folgen dürfen und wie diese zu parsen sind. Beispielsweise muss auf den Buchstaben `f` der Rest des Wortes `false` folgen, da dies die einzige Möglichkeit ist, ohne einen Fehler zu produzieren.

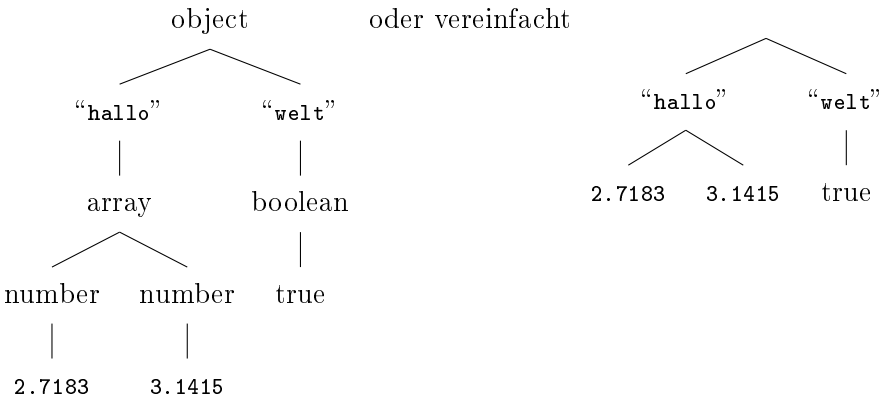
Diese Implementation entspricht zwar nicht exakt der hier formal definierten, ist jedoch beträchtlich performanter; bei komplexeren Grammatiken wäre eine allgemeinere Fassung allerdings angebrachter.

5.2 Syntaktische Analyse

Zweitens wird aus den generierten Tokens ein *Abstract Syntax Tree* (AST) erstellt. Dieser speichert alle verfügbaren Informationen über die Struktur der eingelesenen Daten. Folgend wird der AST für einen Beispieltext dargestellt:

```
{
    "hallo": [
        2.7183,
        3.1415
    ],
    "welt": true
}
```

[6] Siehe "'Maximal-Munch' Tokenization in Linear Time", Thomas Reps, University of Wisconsin. `http://research.cs.wisc.edu/wpis/papers/toplas98b.pdf`



Der AST ist eine Baumstruktur von Java-Klassen. Jedes Baumelement erbt von `JsonValue`; diese abstrakte Klasse definiert Methoden und Variablen wie Referenzen auf das übergeordnete (*parent*) Element sowie Methoden, um den Wert in einen anderen Typen zu konvertieren.

Die Implementation in `JsonParser` erfolgte rekursiv. Dies vereinfacht u.a. die Verifizierung von Klammerausgeglichenheit deutlich. Beispielsweise wird nach einem `OBJECT_START`-Token die `expectObject`-Methode aufgerufen, welche erst nach einem `OBJECT_END` zurückgibt. Innerhalb dieser kann sie sich jedoch selbst aufrufen, wenn ein weiteres Objekt begonnen wird; somit ist die Balance dieser Klammern garantiert. `expectObject` findet sich in Abb. 11.

In dieser Phase werden alle Fehler gefunden[7].

5.3 Übersetzung in POJOs

Die programmatische Navigation des ASTs erlaubt die Untersuchung unbekannter oder dynamischer Daten. Oft ist das genaue Format der Daten jedoch bereits bekannt. In diesen Fällen ist es sinnvoll, die Daten in POJOs (*Plain Old Java Objects*) zu speichern, um die Interaktion mit diesen so flüssig und prägnant wie möglich zu gestalten.

Eine Klasse kann hierfür eine statische, mit `@JsonDeserializer` annotierte Funktion definieren, die als einzigen Parameter ein `JsonValue` annimmt und ein Objekt der umschließenden Klasse zurückgibt. Eine beispielhafte Implementation der `Ticket`-Klasse findet sich in Abb. 12.

[7] Ausgenommen sind semantische Fehler, also beispielsweise Diskrepanzen zwischen den Daten und einem vereinbarten Format oder fehlende Werte

$$T = \{\ \texttt{K\_FUN},\ \texttt{IDENT}\ \}$$

$t \in T$	$i(t)$	$b(t, \vec{s})$
K_FUN	1	$\vec{s} =$ “fun”
IDENT	2	1

n	$\vec{s}$	M	f	$i^{-1}(\min\{i(\bar{t}) \mid \bar{t} \in M\})$
0	“”	$\emptyset$	0	—
1	“f”	{IDENT}	1	IDENT
2	“fu”	{IDENT}	1	IDENT
3	“fun”	{K_FUN, IDENT}	1	K_FUN

Abbildung 10: Beispielhaftes Lexing des Schlüsselworts `fun` in einer fiktiven Programmiersprache zur Veranschaulichung des allgemeinen Prinzips des Lexers. Da `K_FUN` vor `IDENT` (*Identifier*) definiert ist (das kleinere i aufweist), wird der richtige Typ zurückgegeben. Zur Vereinfachung gilt: $\vec{s} =$ “$abc\ldots$” $= (a\ b\ c\ \ldots)^T$.

```
1  private JsonObject expectObject(JsonValueContainer parent)
2  {
3      JsonObject result = valueFactory.createObject(parent);
4
5      // {} ==> return sofort
6      if (next().getType() == EnumJsonTokenType.OBJECT_END)
7          return result;
8      // Lege { zurück (in if-Abfrage wurde Lexer vorgerückt)
9      pushBack();
10
11     // Für jedes Key-Value-Paar...
12     while (peek().getType() == EnumJsonTokenType.STRING)
13     {
14         String key = next().stringValue();
15
16         requireAndSkip(EnumJsonTokenType.COLON);
17
18         // expectValue liest einen beliebigen JSON-Wert
19         result.put(key, expectValue(result));
20
21         if (peek().getType() == EnumJsonTokenType.OBJECT_END)
22             break;
23
24         // Key-Value-Paare sind komma-separiert
25         requireAndSkip(EnumJsonTokenType.COMMA);
26     }
27
28     // Überspringe }
29     next();
30     return result;
31 }
```

Abbildung 11: `expectObject`-Methode im JsonParser

```java
@JsonDeserializer
public static Ticket fromJson(JsonValue json)
{
    // Construct a ticket from a json value

    // the json value must be a JsonObject
    if (!(json instanceof JsonObject))
        throw new IllegalArgumentException();
    JsonObject a = (JsonObject) json;

    // Build the ticket from values in the json
    return builder()
            .withCreationTime(
                a.getNumberAsLong("creationTime"))
            .withTitle(a.getString("title"))
            .withContactInfo(
                Json.fromJson(a.child("contactInfo"),
                    ContactInfo.class))
            .withProduct(a.getString("product"))
            .withProductVersion(a.getString("productVersion"))
            .withPlatform(
                EnumPlatform.valueOf(a.getString("platform")))
            .withDetails(a.getString("details"))
            .withImportance(EnumImportance.valueOf(
                a.getString("importance")))
            .withType(EnumTicketType.valueOf(
                a.getString("type")))
            .withTags(a.isArray("tags")
                      ? a.getArray("tags")
                         .toJavaCollection(ArrayList::new,
                             JsonValue::stringValue)
                      : new ArrayList<>())
            .withStatus(EnumStatus.valueOf(
                a.getString("status", "NEW")))
            .build();
}
```

Abbildung 12: Dekodierung eines `Ticket`-Objekts

6 Schlusswort

Feedback war ein sehr umfangreiches Projekt, dessen Entwicklung mehrere Monate in Anspruch nahm. Es war ein stetiger Lernprozess: Zu Anfang hatte ich mich noch nie mit dem WebSocket-Protokoll beschäftigt und wusste tatsächlich nicht von dessen Existenz; mein ursprünglicher Plan war eine Reihe an AJAX-Requests. Diese Vorgehensweise erwies sich schnell als nicht zielführend, weshalb ich mich nach alternativen Ansätzen umgesehen habe.

Die Aneignung der Details des Protokolls erforderte, sich intensiv mit der Materie auseinanderzusetzen. Hierbei erlernte ich methodisches wissenschaftliches Arbeiten. Darüber hinaus half mir die Aufbereitung des Projekts im Kontext der für den Innovationsaward verfassten Dokumentationen dabei, den Erstellungsprozess zu reflektieren und zu verinnerlichen, sodass ich auch mehr als ein Jahr später weiterhin mit der Thematik vertraut bin.

Ferner war die Präsentation der Ergebnisse vor mehr als Hundert Zuhören eine ausgezeichnete Vorbereitung auf spätere Vorstellungen im Arbeitsleben.

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel benutzt und Stellen, die im Wortlaut oder im Inhalt aus anderen Werken oder dem Internet entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe.

Essen, den 11. April 2019

_____

Julian Koch