



Universität Bremen

Fachbereich 3: Mathematik und Informatik

Diplomarbeit

clide

**Eine webbasierte Entwicklungsumgebung für den interaktiven
Theorembeweiser Isabelle**

Martin Ring

Matrikel-Nr. 221 590 8

16. Januar 2013

- 1. Gutachter:** Prof. Dr. Christoph Lüth
- 2. Gutachter:** Dr. Ing. Dennis Krannich
- Betreuer:** Prof. Dr. Christoph Lüth

Martin Ring

clide

Eine webbasierte Entwicklungsumgebung für den interaktiven Theorembeweiser Isabelle

Diplomarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, Januar 2013

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 16. Januar 2013

Martin Ring

Zusammenfassung

2010 wurde für den interaktiven Theorembeweiser Isabelle die Isabelle/Scala Schnittstelle eingeführt sowie die Beispielanwendung Isabelle/jEdit als Entwicklungsumgebung für Isabelle entwickelt. Dabei fiel die Wahl auf jEdit, weil der Aufwand für die Integration gering gehalten werden sollte. In diesem Projekt konzentrieren wir uns nun auf die Entwicklungsumgebung. Dabei werden modernste Techniken miteinander verknüpft und dadurch eine ganz neue Art des Umgangs

Zusammenfassung

mit

Inhaltsverzeichnis

Inhaltsverzeichnis	i
1 Einführung	1
1.1 Motivation	1
1.2 Aufgabenstellung	1
1.3 Anmerkungen	1
1.3.1 Systemanforderungen	1
2 Grundlagen	3
2.1 Scala	3
2.1.1 Sprachkonzepte	4
2.1.1.1 Traits	4
2.1.1.2 Implizite Parameter	4
2.1.1.3 Implizite Konvertierungen	5
2.1.1.4 Typklassen	5
2.1.1.5 Dynamische Typisierung	6
2.1.2 SBT	7
2.1.3 Akka	7
2.1.4 Play Framework	8
2.2 Isabelle	9
2.2.1 Asynchrones Beweisen	9
2.2.2 Isabelle/Scala	9
2.3 HTML5	11
2.3.1 Dokumentenobjektmodell	11
2.3.2 Cascading Styles Sheets	11
2.3.2.1 CSS3	12
2.3.2.2 LESS	12
2.3.3 JavaScript	13
2.3.3.1 CoffeeScript	13
2.3.4 HTTP	14

2.3.4.1	AJAX	14
2.3.4.2	WebSockets	14
2.3.5	JavaScript-Bibliotheken	16
2.3.5.1	jQuery	16
2.3.5.2	Backbone und Underscore	16
2.3.5.3	RequireJS	16
3	Anforderungen und Entwurf	19
3.1	Server	20
3.1.1	Wahl des Webframeworks	20
3.1.2	Authentifizierung	21
3.1.3	Persistenz	21
3.1.4	Bereitstellung von Ressourcen	21
3.1.5	Isabelle/Scala Integration	22
3.2	Kommunikation	22
3.2.1	WebSockets	23
3.2.2	Protokoll	24
3.3	Client	25
3.3.1	Browserkompatibilität	25
3.3.2	Benutzeroberfläche	26
3.3.2.1	Login	26
3.3.2.2	Projektübersicht	27
3.3.2.3	Die Sidebar	27
3.3.2.4	Die Editor-Komponente	27
3.3.2.5	Beweiszustände	30
3.3.3	Client-Modell	30
4	Implementierung	31
4.1	ScalaConnector / JSConnector	31
4.2	Clientseitiges Syntaxhighlighting	31
4.3	Synchrone Repräsentation von Dokumenten	31
5	Bewertung	33
6	Ausblick	35
7	Zusammenfassung	37
A	Appendix	39
A.1	Abbildungsverzeichnis	39
A.2	Tabellenverzeichnis	39
A.3	Literatur	39
A.4	Liste der Abkürzungen	40

List of TODOs

remove \listoftodos again	
■ Zusammenfassung	4
■ remove \listoftodos again	iii
■ Einführung	1
■ Motivation	1
■ Aufgabenstellung	1
■ Anmerkungen	1
■ Systemanforderungen	1
■ Isabelle/Scala auf dem Server	22
■ Kompatibilitäts-Tabelle	25
■ Client-Modell	30
■ Implementierung	31
■ Bewertung	33
■ Ausblick	35
■ Zusammenfassung	37

Einführung

Einführung

1.1 Motivation

Motivation

1.2 Aufgabenstellung

Aufgabenstellung

1.3 Anmerkungen

Anmerkungen

1.3.1 Systemanforderungen

Systemanforderungen

Grundlagen

Da die im Rahmen dieser Diplomarbeit entwickelte Anwendung vor allem die Verknüpfung sehr vieler Techniken, Konzepte und Standards aus normalerweise getrennten Bereichen der Informatik erfordert, ist es umso wichtiger diese für das Verständnis zu kennen. Im Folgenden sollen die relevanten Begriffe geklärt werden, sowie jeweils ein kurzer Einblick in wichtige Teilbereiche geliefert werden um den Leser auf die folgenden Kapitel vorzubereiten.

2.1 Scala

Die Programmiersprache Scala ist eine an der École polytechnique fédérale de Lausanne von einem Team um Martin Odersky entwickelte statisch typisierte, objektorientierte, funktionale Sprache. In Scala entwickelte Programme laufen sowohl in der Java Virtual Machine (JVM) als auch in der Common Language Runtime (CLR). Die Implementierung für die CLR hängt jedoch stark hinterher und ist für diese Arbeit auch nicht von Interesse. Die aktuelle Sprachversion ist Scala 2.10 welche auch im Rahmen dieser Arbeit verwendet wird.

Scala versucht von Anfang an den Spagat zwischen funktionaler und objektorientierter Programmierung herzustellen. Hierbei ist es sowohl möglich rein objektorientierten als auch rein funktionalen Code zu schreiben. Dadurch entstehen für den Programmierer sehr große Freiheitsgrade und es ist beispielsweise auch möglich imperativen und funktionalen Code zu mischen. Diese Freiheit erfordert eine gewisse Verantwortung von Seiten des Programmierers um lesbaren und wartbaren Code zu erstellen.

Seit 2011 wird Scala von der durch Martin Odersky ins Leben gerufenen Firma Typesafe¹ zusammen mit den Frameworks *Akka* (Abschnitt 2.1.3) und *Play* (Abschnitt 2.1.4) sowie des Buildtools *sbt* (Abschnitt 2.1.2) Kommerziell im sogenannten *Typesafe Stack* weiterentwickelt und unterstützt. Dadurch wird Scala zu einer ernstzunehmenden Sprache, welche auch in Zukunft noch

¹<http://www.typesafe.com>

schnell weiterentwickelt wird. Scala nicht zuletzt durch die Akteure-Bibliothek Akka geeignet um Hochskalierbare verteilte Systeme zu entwickeln. Firmen wie beispielsweise Twitter, LinkedIn, Siemens, TomTom, Sony, Amazon und die NASA treiben die Entwicklung immer schneller voran und sorgen dafür, dass eine große Infrastruktur um Scala herum entsteht. Siehe auch [al11].

2.1.1 Sprachkonzepte

Im folgenden sollen die für diese Arbeit relevanten Konzepte der Sprache Scala kurz vorgestellt werden. Dabei verzichten wir allerings auf Grundlagen der Objektorientierten und Funktionalen Programmierung, sowie auf bereits aus Java bekannte Konzepte.

2.1.1.1 Traits

Traits sind ein besonders wertvoller und wichtiger Bestandteil von Scala. Sie sind ein Mittelweg zwischen einer abstrakten Klasse und einem Interface. Dabei ermöglichen sie wie Interfaces in Java Mehrfachvererbung, können jedoch auch genau wie abstrakte Klassen schon implementierte Funktionen enthalten. Zudem können Traits in beliebige Klassen mit dem Konstruktor eingemischt werden (Sogenannte Mixins).

So können Teile der Funktionalität welche immer wieder verwendet werden, aber nicht von konkreten Klassen abhängig sind getrennt implementiert werden. Das fördert einen aspektorientierten Programmierstil und schafft Übersichtlichkeit.

2.1.1.2 Implizite Parameter

Implizite Parameter werden in Scala verwendet um Parameter die sich aus dem Kontext eines Funktionsaufrufs erschließen können nicht explizit übergeben zu müssen. Eine Funktion f besitzt hierbei zusätzlich zu den normalen Parameterlisten auch eine implizite Parameterliste:

```
f(a: Int)(implicit x: T1, y: T2)
```

In dem Beispiel hat die Funktion einen normalen Parameter a und zwei implizite Parameter x und y . Der Compiler sucht bei einem Funktionsaufruf, welcher die beiden oder einen der impliziten Parameter nicht spezifiziert nach impliziten Definitionen vom Typ $T1$ bzw. $T2$. Diese Definitionen werden im aktuell sichtbaren Scope nach bestimmten Prioritäten gesucht. Dabei wird zunächst

im aktuellen Objekt, dann im zu den Typen **T1** und **T2** gehörenden Objekten und dann in den importierten Namensräumen gesucht. Implizite Definitionen haben die Form **implicit def/val/var x: T = ...** wobei der Name **x** keine Rolle spielt.

2.1.1.3 Implizite Konvertierungen

Des Weiteren existiert das Konzept der impliziten Konvertierungen (implicit conversion). Hierbei werden bei Typfehlern zur Kompilierzeit Funktionen mit dem Modifizierer **implicit** gesucht, die den gefundenen Typen in den nötigen Typen umwandeln können. Die Priorisierung ist geschieht hierbei genauso wie bei impliziten Parametern. Ein Beispiel:

```
implicit def t1tot2(x: T1): T2 = ...  
def f(x: T2) = ...  
  
val x: T1 = ...  
f(x)
```

In dem Beispiel wird eine implizite Konvertierung von **T1** nach **T2** definiert. Bei dem Aufruf **f(x)** kommt es zu einem Typfehler, weil **T2** erwartet und **T1** übergeben wird. Dieser Typfehler wird gelöst indem vom Compiler die Konvertierung eingesetzt wird. Der Aufruf wird also intern zu **f(t1tot2(x))** erweitert.

2.1.1.4 Typklassen

Mit Hilfe von impliziten Definitionen ist es möglich, die aus der Sprache Haskell bekannten Typklassen in Scala nachzubilden.

Eine Typklasse bietet die Möglichkeit Ad-hoc-Polymorphie zu implementieren. Damit ist es möglich ähnlich wie bei Schnittstellen Funktionen für eine Menge von Typen bereitzustellen. Diese werden jedoch nicht direkt von den Typen implementiert sein und können so auch Nachträglich beispielsweise für Typen aus fremden Bibliotheken definiert werden.

In Scala werden Typklassen als generische abstrakte Klassen oder Traits implementiert. Instanzen der Typklassen sind implizite Objektdefinitionen welche für einen spezifischen Typen die Typklasse bzw. die abstrakte Klasse implementieren. Eine Funktion für eine bestimmte Typklasse kann durch eine generische Funktion realisiert werden. Diese ist dann über einen oder mehrere Typen parametrisiert und erwartet als implizites Argument eine Instanz der Typklasse für diese Typen, also eine implizite Objektdefinition. Wenn diese im Namensraum existiert, wird sie automatisch vom Compiler eingesetzt.

Als Beispiel betrachten wir die Ordnung eines Typs. Zunächst definieren wir einen generischen Trait `Ord`, welcher über eine `compare`-Funktion zum Vergleich zweier Werte.

```
trait Ord[T] {  
  def compare: (a: T, b: T): Int  
}
```

Wollen wir nun für einen beliebigen bestehenden Typ eine Ordnung definieren, müssen wir lediglich ein implizites Objekt bereitstellen, welches `Ord` implementiert.

```
implicit object FileOrd extends Ord[File] {  
  def compare: (a: File, b: File): Int = ...  
}
```

Eine generische Funktion, welche die Funktion `compare` aus der Typklasse `Ord` verwendet kann nun definiert werden, indem eine Instanz von `Ord` als impliziter Parameter erwartet wird.

```
def sort[T](elems: List[T])(implicit ord: Ord[T]): List[T] = ...
```

Die Funktion `sort` kann nun verwendet werden um Dateien zu sortieren, solange das implizite Objekt `FileOrd` beim Aufruf sichtbar ist.

Das Konzept der Typklassen ist vor allem dort sehr hilfreich wo es darum geht fremde Bibliotheken um eigene Funktionen zu erweitern.

2.1.1.5 Dynamische Typisierung

Seit Scala 2.9 ist es möglich Funktionsaufrufe bei Typfehlern dynamisch zur Laufzeit auflösen zu lassen. Damit die Typsicherheit nicht generell verloren geht ist es nötig den Trait `Dynamic` zu importieren um einen Typ als dynamisch zu markieren. Wenn die Typüberprüfung dann bei einem Aufruf auf einem als dynamisch markierten Objekt fehlschlägt wird der Aufruf auf eine der Funktionen `applyDynamic`, `applyDynamicNamed`, `selectDynamic` und `updateDynamic` abgebildet. Diese Übersetzung geschieht nach folgendem Muster:

```
x.method("arg") => x.applyDynamic("method")("arg")  
x.method(x = y) => x.applyDynamicNamed("method")(("x", y))  
x.method(x = 1, 2) => x.applyDynamicNamed("method")(("x", 1), ("", 2))  
x.field          => x.selectDynamic("field")  
x.variable = 10 => x.updateDynamic("variable")(10)  
x.list(10) = 13 => x.selectDynamic("list").update(10, 13)  
x.list(10)       => x.applyDynamic("list")(10)
```

Die Dynamische Typisierung ist ein Sprachkonstrukt welches in Scala nur in Ausnahmefällen verwendet werden sollte. Es Erweist sich aber als sehr Praktisch in der Interaktion mit Dynamischen Sprachen sowie bei der Arbeit mit externen Daten, bei denen keine Typinformationen vorliegen.

2.1.2 SBT

Das Simple Build Tool (sbt)² wird seit 2011 von Typesafe weiterentwickelt und ist das Standard Werkzeug zur automatischen Projekt- und Abhängigkeitsverwaltung in der Scala Programmierung. Da es selbst in Scala geschrieben wurde und auch die Konfiguration in Scala Objekten stattfindet, ist es leicht Erweiterungen dafür zu entwickeln. Sbt ist mit Maven, einem sehr verbreiteten Build Tool für Java, kompatibel, sodass auch Javabibliotheken, welche in einem Maven-Repository liegen als Abhängigkeiten definiert werden können. Typesafe stellt zudem eine sehr große Auswahl von Scala-Bibliotheken in einem eigenen Repository zur Verfügung. Es können aber auch beispielsweise öffentliche Git-Repositories als Abhängigkeit definiert werden.

2.1.3 Akka

Akka³ war ursprünglich eine Implementierung des aus Erlang bekannten Aktoren Modells, ist mittlerweile jedoch zu einem umfangreiches Framework zur Entwicklung von Hochperformanten Verteilten Systemen gewachsen. Die Grundlage bilden nach wie vor Aktoren, wenn auch in, gegenüber anderen Implementierungen, leicht Veränderter Form. [Hal12]

Aktoren habe sich als eine sehr wertvolle Abstraktion zur Modellierung von Verteilten Sytemen herausgestellt. Dabei wird ein System in mehrere parallel agierende Aktoren aufgeteilt, welche sich untereinander Nachrichten senden. Um ungewollte Nebenläufigkeitseffekte auszuschließen, müssen die Nachrichten unveränderbar sein, was in Scala bislang leider noch nicht überprüfbar ist und somit in der Verantwortung des Entwicklers liegt.

In Akka sind Aktoren Ortsunabhängig, und können an einer beliebigen Stelle ausgeführt werden. Ein Aktor, kann auf dem selben Prozessor, einem anderen Prozessor im selben Rechner, auf einem anderen Rechner im lokalen Netz oder auch auf einem beliebigen über das Internet erreichbaren Rechner irgendwo auf der Welt ausgeführt werden. In dieser Eigenschaft liegt der Schlüssel zur Skalierbarkeit: In Akka entwickelte Systeme können ohne Veränderungen auf einem oder tausenden Rechnern gleichzeitig ausgeführt werden, ganz im Sinne des *Cloud Computing*.

²<http://www.scala-sbt.org/>

³<http://www.akka.io/>

Akka bietet über die Aktoren und deren Verwaltung hinaus noch viele weitere hilfreiche Abstraktionen. Besonders erwähnenswert sind hierbei noch die sogenannten *Iteratees*. Iteratees sind eine Möglichkeit einen Datenstrom zu verarbeiten, ohne dass alle Daten verfügbar sind. Das ist dann besonders Wichtig, wenn es um nicht blockierende kommunizierende Prozesse geht, wie es bei Webanwendungen üblich ist. (Siehe auch [Sur12])

2.1.4 Play Framework

Das *Play Framework*⁴ ist ein Rahmenwerk zur Entwicklung von Webanwendungen auf der JVM in Java mit einer speziellen API für Scala. Play ist ein sehr effizientes Zustandsfreies Framework welches auf Akka aufbaut um hohe Skalierbarkeit zu gewährleisten. Damit wird es leichter verteilte hochperformante Webanwendungen zu realisieren.

Die Struktur einer Play Anwendung ähnelt der bewährten Struktur von Ruby on Rails⁵. Es existieren Modelle, Views und Controller. Auch der Workflow ist ein ähnlicher, und so ist es möglich während der Entwicklung der Anwendung durch auffrischen der Seite im Browser immer die neuste Version zu sehen. Play nutzt sbt als Build System.

Views werden als spezielle HTML-Templates, die auch Scala Code zulassen, definiert. Durch die Ausdrucksorientiertheit von Scala ist es damit möglich die Views Typsicher zu halten und somit schon zur Kompilierzeit über sehr viele Klassen von Fehlern informiert zu werden. Das ist ein klarer Vorteil gegenüber den meisten andern modernen Webframeworks.

Modelle können natürlicherweise als beliebige Scala Klassen bzw. Datenstrukturen repräsentiert werden. Play beschränkt den Entwickler auch nicht auf eine bestimmte Möglichkeit der Datenpersistenz.

Die Controller gruppieren Aktionen, welche als Antworten auf Anfragen agieren und sind prinzipiell als Unterklassen von `play.mvc.controller`, welche in der Akka Infrastruktur leben, realisiert. Aktionen sind nicht mehr als Scala Funktionen, die die Parameter einer Anfrage verarbeiten und daraus eine Antwort produzieren, welche an den Browser zurückgesendet wird.

Das Routing geschieht über eine Konfigurationsdatei `conf/routes` in welcher von URLs auf Aktionen abgebildet wird. Es ist möglich sogenanntes *reverse routing* zu betreiben und von einer Scala Funktion zu einer Uniform Resource Locator (URL) zu gelangen. Des weiteren besteht die Möglichkeit JavaScript (JS) Code zu generieren welcher im Browser zum reverse routing verwendet werden kann.

Siehe auch Abschnitte 2.3.2.2, 2.3.3.1 sowie 2.3.5.3.

⁴<http://www.playframework.org>

⁵<http://rubyonrails.org/>

2.2 Isabelle

Isabelle ist ein generisches System welches vor allem zum interaktiven Beweisen von Theoremen unter der Nutzung von Logiken höherer Ordnung eingesetzt wird. Isabelle ist in Standard ML (SML) implementiert und stark davon abhängig. In Beweisen kann die volle Mächtigkeit von SML an jeder Stelle benutzt werden. Dadurch ist es schwer eine Echtzeitverarbeitung wie sie für eine Entwicklungsumgebung nötig wäre zu realisieren.

Die Intelligible semi-automated reasoning (Isar)-Plattform bietet eine zusätzliche Abstraktion vom nackten SML Code die dem Benutzer eine komfortablere Umgebung zum formulieren von *Beweisdokumenten* liefert. Darüber hinaus ermöglichen die strukturierten Dokumente eine Menschenlesbare Veröffentlichung der Beweise. Das ist ein klarer Vorteil gegenüber Beweisen in SML Skripten, welche eher Maschinenbezogen sind.

Isabelle/Isar erlaubt das Veröffentlichen in verschiedene Formate, wie HTML und LaTeX. Dabei werden Bestimmte Konstrukte besonders Dargestellt. Solche Symbole werden als `\<...>` im Code repräsentiert. Es gibt theoretisch unendlich viele dieser Symbole, allerdings wird nur eine kleine Menge von Symbolen in [al12] genau spezifiziert. Des weiteren existieren Kontrollzeichen in der Form `\<^...>` welche genutzt werden können um Sub- und Superskript zu repräsentieren bzw. Zeichen Fett darzustellen. Da die konkrete Benutzung der Isabelle-Plattform selbst für diese Arbeit eine eher untergeordnete Relevanz hat, wird an dieser Stelle für weitere Informationen auf die Isabelle Referenz [al12] verwiesen.

2.2.1 Asynchrones Beweisen

Seit Version 2009 von Isabelle wurde es möglich Beweisdokumente, bzw. Theorien nebenläufig zu überprüfen. Das macht es realistisch Echtzeitinformationen während der Bearbeitung verfügbar zu machen. [Wen09]

2.2.2 Isabelle/Scala

Seit 2010 existiert mit Isabelle/Scala eine neue Schnittstelle zur Isabelle-Plattform welche auf der Sprache Scala basiert. Isabelle/Scala stellt eine API zur Arbeit mit Isabelle bereit, welche die zur Nutzung relevanten Teile der SML Implementierung in Scala abbilden. [Wen10]

Über statisch typisierte Methoden können die Dokumente modifiziert werden. Dafür wurde ein internes XML-Basiertes Protokoll eingeführt, welches die Scala API mit der SML API verknüpft. ÜDementsprechend sind auch die Informationen welche von Isabelle geliefert werden typisiert.

Das macht Isabelle/Scala in der Nutzung recht robust, da ein Großteil der Fehler bereits zur Übersetzungszeit gefunden werden kann.

Isabelle/Scala wurde für und zusammen mit der Anwendung Isabelle/jEdit entwickelt. JEdit wurde unter anderem deswegen gewählt, weil es eine sehr einfache API hat und somit das Projekt nicht zu sehr auf den Editor konzentriert ist.

2.3 HTML5

Für die Implementierung der Browseranwendung wird auf den aktuellen Entwurf des zukünftigen HTML5 Standards zurückgegriffen. [Smi12]

Hypertext Markup Language (HTML) ist eine Sprache die der strukturierten Beschreibung von Webseiten dient. Die Sprache wurde in ihrer ursprünglichen Form von 1989-1992, lange vor dem sogenannten Web 2.0, von Wissenschaftlern des Europäischen Kernforschungsinstitut CERN entwickelt. Sie war der erste nicht proprietäre globale Standard zur digitalen Übertragung von Strukturierten Dokumenten. Die Sprache HTML allein ist nicht geeignet um dynamische Inhalt wie sie heute praktisch auf allen modernen Webseiten vorkommen, zu beschreiben.

Der heutige HTML5-Standard geht weit über die Sprache HTML selbst hinaus und umfasst vor allem auch die Scriptsprache JS und die darin verfügbaren Bibliotheken, sowie das sogenannte Document Object Model (DOM), auf welches in Scripten zugegriffen werden kann um den angezeigten Inhalt dynamisch zu verändern.

2.3.1 Dokumentenobjektmodell

Das DOM ist eine Schnittstelle, die es erlaubt HTML- bzw. Extensible Markup Language (XML)-Dokumente zu modifizieren und bildet damit die Grundlage für die Realisierung dynamischer Webseiten.

Der Einstiegspunkt in das DOM ist der `document` Knoten, welcher von in JS global verfügbar ist und von welchem aus die gesamte Baumstruktur erreichbar ist. Jeder HTML-Tag, jedes Attribut und jeder Text wird als ein Objekt, bzw. ein Knoten im Baum repräsentiert. Über verschiedene Methoden ist es möglich die Kinder-, Geschwister- und Elternknoten zu erhalten. Durch Funktionen wie z.B. `appendChild` ist es schließlich möglich neue Elemente in das DOM zu integrieren bzw. bestehende zu modifizieren. [Kes12]

2.3.2 Cascading Styles Sheets

Cascading Style Sheets (CSS) ist eine Sprache die der Definition von Stilen bzw. Stilvorlagen für die Anzeige von Webinhalten dient.

Durch die Trennung von HTML und CSS wird erreicht, dass HTML-Dokumente sich auf den Inhalt einer Seite beschränken, während alle die grafische Anzeige betreffenden Aspekte in die sogenannten Stylesheets in CSS Dateien ausgelagert werden.

2.3.2.1 CSS3

In der kommenden CSS Version 3.0, welche bereits zu großen Teilen von den meisten aktuellen Browsern unterstützt wird kommen einige interessante Neuerungen hinzu welche es vor allem Ermöglichen, Anwendungen welche man Bisher in Frameworks wie Flash oder Silverlight implementiert hat nun in reinem HTML+CSS zu verwirklichen. Die für die Realisierung der browserbasierten Entwicklungsumgebung relevanten Neuerungen umfassen im speziellen

- Einbetten von Schriftarten,
- Animationen und Übergänge,
- Verhindern von Textmarkierungen und
- Festlegen der Sichtbarkeit von Elementen für den Mauszeiger.

2.3.2.2 LESS

Auch CSS3 hat immernoch einige konzeptionell Einschränkungen welche die Arbeit damit stark erschweren:

- Es ist nicht möglich Variablen zu definieren um Eigenschaften, welche an vielen Stellen vorkommen nur einmal zu definieren.
- Es fehlen Funktionsdefinitionen um ähnliche oder abhängige Definitionen zusammenzufassen und zu parametrisieren.
- Die Hierarchie einer CSS-Datei ist flach obwohl die Definitionen geschachtelt sind. Das reduziert die lesbarkeit der Dateien.
- Wenn aus Gründen der Übersichtlichkeit CSS in mehrere Dateien aufgeteilt werden, müssen alle Dateien einzeln geladen werden, was zu längeren Ladezeiten führt.

LESS ist eine Erweiterung von CSS welche unter anderem Variablen- und Funktionsdefinitionen, verschachtelte Definitionen sowie Dateiimports erlaubt. Damit werden die oben genannten Einschränkungen von CSS aufgehoben.

Das Play Framework (Siehe Abschnitt 2.1.4) ermöglicht es die Stylesheet-Sprache *LESS* zu verwenden ohne, dass diese auf Browserseite unterstützt werden muss. Hierfür werden die in *LESS* definierten Stylesheet auf Serverseite in *CSS* übersetzt und dem Browser zur Verfügung gestellt. Dafür müssen die Dateien an einem vorher konfigurierten Ort liegen. Nach dem übersetzen werden sie an der selben Stelle zur Verfügung gestellt wie normale *CSS* Dateien.

2.3.3 JavaScript

JS ist eine dynamisch typisierte, klassenlose objektorientierte Scriptsprache welche aktuell der Standard in der Entwicklung von clientseitigem Code für dynamische Webinhalte ist. Der Kern von JS wurde von der Ecma International als ECMAScript normiert. Durch JS ist es möglich Webseiten dynamisch zu verändern. Siehe auch [Fla97].

2.3.3.1 CoffeeScript

JavaScript wurde in Eile entwickelt und normiert, da zur Zeit der Entstehung ein schneller Bedarf an einer normierten Sprache für das Web bestand. Dadurch sind jedoch auch einige Unschönheiten in den Sprachkern gedrungen. So ist beispielsweise die C-artige Syntax für eine eher funktional angehauchte Sprache wie JS ungeeignet, da Funktionsdefinitionen durch geschweifte Klammern, das Schlüsselwort **function** und ein **return** Statement unnötig aufgeblasen werden und damit unleserlicheren Code erzeugen. Darüber hinaus fehlt in JS jede Möglichkeit der Modularisierung. Da noch nicht einmal Klassen existieren, führt das bei reinem JavaScript schnell zu unwartbarem Code. Ein weiterer Stolperstein ist das Schlüsselwort **this** welches nicht immer klar zu verstehen ist, da es in Funktionsaufrufen seine Bedeutung wechseln kann.

*CoffeeScript*⁶ ist eine neue Scriptsprache mit dem Ziel die Unschönheiten von JS auszumärzen. CoffeeScript hat eine eher an funktionale Sprachen wie Haskell erinnernde Syntax mit Verschachtelungen über Whitespace und einem `->` Operator zur Funktionsdefinition. Darüber hinaus bietet CoffeeScript die Möglichkeit Klassen zu definieren und führt das Konzept der Vererbung ein. Das **this**-Schlüsselwort kann an Instanzen von Klassen gebunden werden. CoffeeScript Dateien werden zu optimiertem JS Code kompiliert, welcher den Vorgaben des *JS Linter*⁷ entspricht. Im einzelnen sind die Verbesserungen gegenüber JS:

- Vereinfachte Syntax für Funktionsdefinitionen, Arrays, Blöcke,
- automatisches initialisieren von Variablen (*lexical scoping*),
- *Spats* (Variable Parameterlisten),
- universellere Iterationsschleifen (**for**),
- vereinfachtes *slicing* und *splicing* (Arrayoperationen),
- Ausdrucksorientiertheit,
- Klassen und Vererbung,
- ein Existenzoperator,
- destrukurierende Zuweisungen (z.B. `[a,b] = [c,d]`),
- Bindung von **this** an Klasseninstanzen sowie
- mehrzeilige Strings und reguläre Ausdrücke mit Kommentaren

⁶<http://www.coffeescript.org>

⁷<http://www.javascriptlint.com/>

Genauso wie für *LESS* existiert im Play Framework (Siehe Abschnitt 2.1.4) die Serverseitige Unterstützung für *CoffeeScript*. Die in CoffeeScript geschriebenen Dateien werden ebenfalls an gleicher Stelle wie normale JS-Dateien dem Browser als JS zur Verfügung gestellt.

2.3.4 HTTP

Das Hypertext Transfer Protocol (HTTP) ist das im Internet verwendete Standardprotokoll zur Übertragung von Daten. Siehe auch [al99]. Leider ist HTTP für die Implementierung von dynamischen Webapplikationen nur bedingt geeignet, da Anfragen immer vom Browser gestellt werden müssen, aber keine Möglichkeit vorgesehen ist in die andere Richtung initiativ zu kommunizieren. Auf Grund der Einschränkungen des Protokolls kam es in der Vergangenheit zu vielen Tricks um Server-Pushes zu realisieren. Die Bekanntesten sind das *Polling*, wobei in kleinen Abständen Anfragen an den Server gestellt werden, und *Comet* welches auf verzögerten Antworten vom Server basiert. Beide Lösungen werfen neue Probleme auf. Zum einen eine deutlich erhöhte Nutzung von Verbindungskapazität bei Polling und zum anderen das Ausbleiben von Antworten und damit die fehlende Freigabe von Threads bei Comet Anfragen.

2.3.4.1 AJAX

Asynchronous JavaScript and XML (AJAX) ist keine Bibliothek und auch kein Standard sondern ein sehr weit verbreitetes Konzept zur Übertragung von Daten zwischen Browser und Webserver. Hierbei wird das JS-Objekt `XMLHttpRequest` verwendet um während der Anzeige einer Webseite Daten vom Server nachzuladen, bzw. dem Server Daten zu senden, ohne dass die Webseite neu geladen werden muss wie es bei klassischen Webseiten der Fall war. Ursprünglich und Namensgebend wurde für die Übertragung der Daten XML verwendet. Mittlerweile ist es wegen der guten Unterstützung in JS und den Meisten Webframeworks auch üblich JavaScript Object Notation (JSON) zur Übertragung zu nutzen. Siehe auch [Jäg07]

2.3.4.2 WebSockets

Websockets sind ein in HTML5 neu eingeführter Standard zur bidirektionalen Kommunikation zwischen Browser und Webserver. Hierbei wird anders als bei AJAX eine direkte TCP-Verbindung hergestellt. Diese Verbindung kann sowohl von Browser- als auch von Serverseite aus gleich verwendet werden. Das macht es unnötig, wie bei AJAX wiederholte Anfragen oder Anfragen ohne Zeitbegrenzung zu stellen um Informationen vom Server zu erhalten wenn diese Verfügbar werden. Ein weiterer Vorteil gegenüber HTTP-Anfragen ist, dass durch die direkte

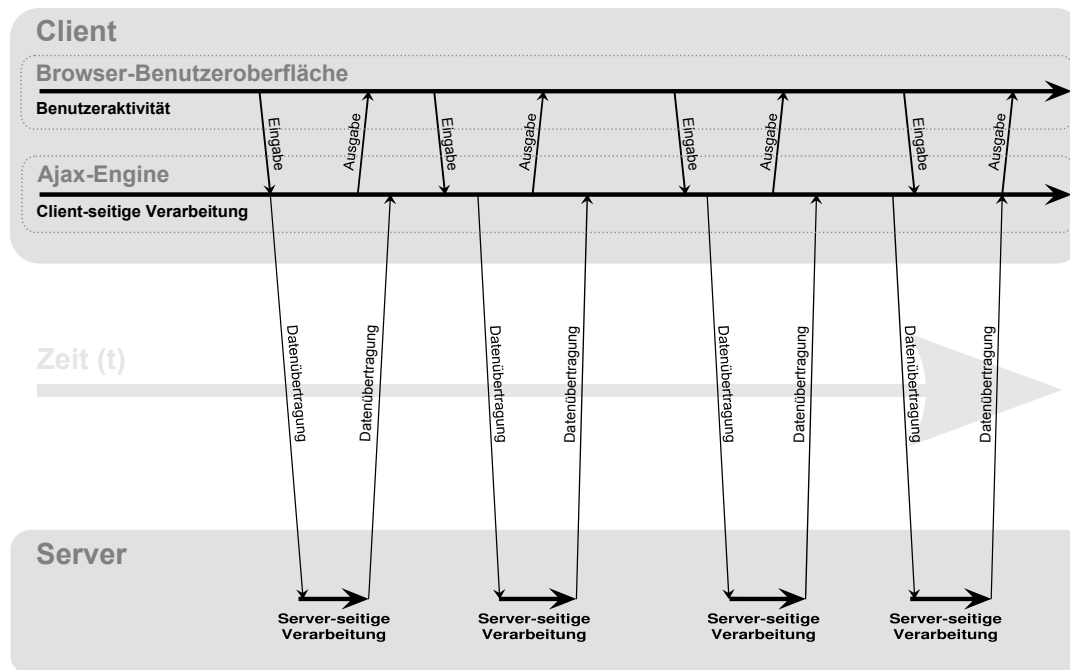


Abbildung 2.1 Ajax Modell einer Web-Anwendung (asynchrone Datenübertragung)

Quelle: Wikipedia

permanente Verbindung kein Nachrichtenkopf mehr nötig ist. Das macht es deutlich effizienter viele kleine Nachrichten zu versenden.

Zum Aufbau einer WebSocket Verbindung wird einmalig zu Beginn eine HTTP Anfrage vom Browser gestellt, welche der Server dann im Erfolgsfall mit der Eröffnung des WebSockets beantwortet. Siehe auch [Hic12]

2.3.5 JavaScript-Bibliotheken

Über den HTML5 Standard hinaus werden für die Strukturierung der Anwendung einige JS-Bibliotheken benötigt, welche im folgenden kurz erläutert werden.

2.3.5.1 jQuery

Die Bibliothek *jQuery* ist heute ein defacto Standard in der Webentwicklung. In erster Linie erleichtert es den Zugriff auf das und die Manipulation des DOM, bietet aber darüber hinaus zahlreiche Vereinfachungen im Umgang mit alltäglichen Aufgaben der Webentwicklung. Besonders erwähnenswert ist hierbei die AJAX Abstraktion. Eine gute Einführung in jQuery bietet [Fra11].

2.3.5.2 Backbone und Underscore

*Backbone*⁸ ist eine Bibliothek, die der Strukturierung von sogenannten *single-page* acrjs Anwendungen dient. Backbone baut auf der allgemeinen *Underscore*⁹-Bibliothek auf, die einige Erleichterungen im Umgang mit Daten in JS bietet.. Backbone führt das Model-View- Controller Konzept in Browseranwendungen ein und bietet hierfür einige Prototypen von welchen abgeleitet werden kann um eigene Modelle und Views zu implementieren, welche dann über Events deklarativ miteinander verknüpft werden können.

Besonders interessant ist die Möglichkeit die im HTML5 Standard neu eingeführte *History API* in Backbone zur Navigation zu verwenden. Dadurch ist es möglich in einer JS Anwendung, welche die Seite nicht neu aufbaut sondern immer nur Teile verändert Navigation einzuführen, sodass der Benutzer zwischen den Zuständen Navigieren kann. Hierfür bietet Backbone die sogenannten **Router** in denen von URLs auf Zustände der Anwendung und umgekehrt abgebildet werden kann. Siehe auch [Osm12].

2.3.5.3 RequireJS

Da JavaScript von Haus aus keine Möglichkeit der Modularisierung bietet, komplexe Anwendungen jedoch ohne Modularisierung kaum wartbar bleiben, haben sich unterschiedliche Lösungsansätze für dieses Problem entwickelt. Einer der umfassendsten ist die Bibliothek *RequireJS*.

Mit der Funktion **define** können Module in Form von Funktionsdefinitionen definiert werden. Alle lokalen Variablen in dem Modul sind anders als bei normalen Scripten außerhalb nicht mehr

⁸backbonejs.org

⁹underscorejs.org

sichtbar, da sie innerhalb einer Funktion definiert wurden. Das Funktionsergebnis ist das was nach außen sichtbar ist. Das kann ein beliebiges Objekt (also auch eine Funktion) sein.

Die so definierten Module können Abhängigkeiten untereinander spezifizieren indem der **define**-Funktion eine Liste von Modulen übergeben wird, die das aktuelle Modul benötigt. Die *RequireJS*-Bibliothek sorgt dann dafür, dass diese Module geladen wurden bevor das aktuelle Modul ausgeführt wird.

RequireJS bietet die Möglichkeit den JS-Code für den Produktiveinsatz zu optimieren. Dafür gibt es das sogenannte *r.js*-Script welches unter andern alle Abhängigkeiten in eine Datei zusammenfasst und den Code durch das Entfernen von Whitespaces und Kommentaren sowie dem Umbenennen von Variablennamen verkürzt. Zur Entwicklungszeit ist dieser nicht mehr lesbare Code nicht erwünscht. Deswegen bietet das Play Framework (Siehe Abschnitt 2.1.4) eine integrierte Version von RequireJS, welche automatisch den lesbaren Code zur Entwicklungszeit bereitstellt, im Produktiveinsatz jedoch den optimierten.

Anforderungen und Entwurf

Auf Grund der Komplexität und dem unvermeidbaren Rechenaufwand, welcher ein Theorembeweiser mitbringt, ist es aus aktueller Sicht nahezu ausgeschlossen diese Arbeit zu größeren Teilen im Browser zu verwirklichen. Die einzige von allen größeren Browsern unterstützte Scriptsprache ist zur Zeit immernoch JS, welche vor allem auf Grund der dynamischen Typisierung und der fehlenden Parallellisierbarkeit um einige Faktoren langsamer ausgeführt wird, als nativer Code.

Ein besonderer Vorteil, den die Anwendung gegenüber bisherigen Lösungen bringen soll, ist die Mobilität. Das bedeutet, dass es von jedem Rechner mit Internetzugang und einem modernen Webbrowser aus möglich sein soll, die Anwendung zu nutzen und auf eventuell bereits zu einem früheren Zeitpunkt an einem anderen Ort erstellten Theorien zugreifen zu können. Damit wird klar, dass die Projekte und Theorien nicht lokal auf den einzelnen Rechnern verwaltet werden können, sondern an einer zentralen von überall erreichbaren Stelle gespeichert sein müssen. Die Entscheidung zu einem Client-Server Modell ergibt sich bei einer Webanwendung ohnehin automatisch.

Da es sich bei der Webanwendung um eine Entwicklungsumgebung handelt, welche insbesondere durch Echtzeitinformationen einen Mehrwert bringen soll, ist einer der wichtigsten Aspekte des Entwurfs die effiziente Kommunikation zwischen Server und Browser. Da die Kommunikation bei einer Webanwendung generell sehr Zeitaufwändig ist - abhängig von der Internetanbindung kann es zu großen Verzögerungen kommen - muss abgewogen werden, welche Arbeit im Browser und welche auf dem Server erledigt werden soll. Als illustratives Beispiel kann das Syntax-Highlighting genannt werden: Isabelle verfügt über eine innere und eine äußere Syntax, die sich im analytischen Aufwand stark unterscheiden. Während die äußere Syntax relativ leicht zu parsieren ist und dabei bereits viele Informationen liefert, ist die innere Syntax sehr komplex, flexibel und stark vom jeweiligen Kontext abhängig. Somit liegt es nahe, das Syntaxhighlighting aufzuteilen: Um Übertragungskapazität zu sparen kann das Highlighting der äußeren Syntax bereits im Browser mittels JS stattfinden. Die feiner granulierten Informationen aus der inneren Syntax können dann auf dem Server ermittelt werden und mit kurzer Verzögerung in die Darstellung integriert werden.

3.1 Server

Der Webserver muss neben den normalen Aufgaben eines Webserver, wie der Bereitstellung der Inhalte, der Authentifizierung der Benutzer sowie der Persistierung bzw. Bereitstellung der nutzerspezifischen Daten (In unserem Fall Projekte / Theorien), auch eine besondere Schnittstelle für die Arbeit mit den Theorien bereitstellen. Vom Browser aus muss es möglich sein,

- die einzelnen Theorien in Echtzeit zu bearbeiten,
- Informationen über Beweiszustände bzw. Fehler zu erhalten,
- als auch Informationen über die Typen, bzw. Definitionen von Ausdrücken zu erhalten.

All diese Informationen müssen zuvor Serverseitig aufbereitet und bereitgestellt werden. Dabei ist es aus Sicht der Perfomanz wichtig, unnötige Informationen zu eliminieren und die Daten zu komprimieren.

Der Server stellt zum einen eine normale HTTP API zur Authentifizierung und zur Verwaltung der Projekte zur Verfügung, zum andern eine WebSocket Schnittstelle zur Arbeit mit den Theorien. Während bei der HTTP API auf bewährte Methoden aus der Literatur zurückgegriffen werden kann gibt es für die WebSocket Schnittstelle keine nennenswerten Erfahrungen auf die hier aufgebaut werden könnte.

3.1.1 Wahl des Webframeworks

Da wir die Isabelle/Scala Schnittstelle nutzen, liegt es nahe ein Webframework in Scala zu nutzen um den Aufwand für die Integration gering zu halten. Dafür existieren momentan zwei ausgereifte, bekannte Alternativen

- Das *Lift Webframework*¹ bietet viele neue Ansätze in der Webprogrammierung und kann als Experimentierkasten verstanden werden. Für jeden Anwendungsfall gibt es gleich mehrere Lösungen. Lift wird allerdings auf Grund des Rückzugs von David Pollack aus der Entwicklung seit einiger Zeit nicht mehr geordnet weiter entwickelt wird und ist zudem für unsere Zwecke überdimensioniert. Da die Webanwendung eher unkonventionelle Anforderungen an den Server hat, nutzen die meisten Funktionen von Lift nichts. Lift wurde in der Vergangenheit schnell in verschiedenste Richtungen weiterentwicklet, dabei ist die Dokumentation jedoch stets vernachlässigt worden.
- Das *Play Framework* (Siehe auch Abschnitt 2.1.4) ist dagegen bewusst Leichtgewichtig gehalten und eher auf hohe Performance ausgelegt, als auf die Lösung möglichst vieler Anwendungsfälle in verschiedenster ausgefallener Weise. Darüber hinaus wird Play mittlerweile kommerziell von Typesafe unterstützt und weiterentwickelt und verfügt über eine detaillierte und professionell gestaltete Dokumentation. **play**

¹<http://www.liftweb.org>

Erfahrungen aus früheren Projekten mit Lift scheinen an dieser Stelle nicht zu nützen, da die größte Hürde die Bereitstellung der WebSocket Schnittstelle bildet und diese in Lift nur sehr

3.1.2 Authentifizierung

Die Authentifizierung soll in diesem Projekt bewusst einfach gehalten werden, da es sich hierbei um eine Nebensächlichkeit handelt, welche ohne weitere Probleme aufgerüstet werden kann. Wir beschränken uns daher auf die Möglichkeit sich mit einem Benutzernamen sowie einem dazugehörigen Passwort einzuloggen, welche dann mit einer Konfigurationsdatei auf dem Server abgeglichen wird. Wir können dann auf die Möglichkeit des Play Frameworks zur sicheren Verwaltung von Session-Bezogenen Daten zurückgreifen um die Anmeldung aufrechtzuerhalten.

3.1.3 Persistenz

Als Besonderheit bei der Datenpersistenz sind die serverseitig zu verwaltenden Theorien zu nennen. Da jeder Benutzer eine von allen anderen Benutzern unabhängige Menge von Projekten mit Theorien besitzt, also eine hierarchische Struktur besteht, spricht nichts dagegen, die Daten Serverseitig im Dateisystem zu verwalten. Somit ist auch eine eventuelle spätere Integration eines Versionsverwaltungssystems wie *Mercurial* oder *Git* möglich. Da über diese Daten hinaus nur wenige Informationen vom Server verwaltet werden müssen, ist die Einrichtung und Anbindung einer Datenbank nicht von Nöten.

3.1.4 Bereitstellung von Ressourcen

Das Play-Framework bietet ausgefeilte Möglichkeiten sowohl statische als auch dynamische Ressourcen bereit zu stellen. Ein Hauptaugenmerk liegt hierbei auf der Bereitstellung der nötigen JS-, CSS- sowie HTML-Dateien.

Während der Entwicklung dieser Arbeit wurde ein Modul für das Play Framework entwickelt, um **CoffeeScript-Dateien** automatisch zu JS zu übersetzen, Abhängigkeiten mit RequireJS aufzulösen und eine Optimierte JS-Datei bereitzustellen. Da in der in Kürze erscheinenden Version 2.1 des Play Framework genau diese Funktionalität entwickelt wurde, liegt die Entscheidung nahe, diese neue Funktionalität zu nutzen und das eigene Modul wegfallen zu lassen, da so eine Weiterentwicklung bzw. die Kompatibilität mit zukünftigen Versionen des Frameworks gesichert ist. (Siehe Abschnitt 2.3.3.1 sowie 2.3.5.3)

Ebenfalls von Play unterstützt wird die Möglichkeit **Less CSS (LESS)-Dateien** mit ihren Abhängigkeiten zu einer CSS-Datei zu übersetzen. Da diese genau wie bei der CoffeeScript-Übersetzung zur Entwicklungszeit in lesbare und im Produktiveinsatz in optimierte Dateien übersetzt werden. Da die Oberfläche im Fall der Entwicklungsumgebung sehr vielschichtig und komplex ist, ist eine Modularisierung der Stilvorlagen eine willkommene Erleichterung und im Sinne der Wartbarkeit.

Statische Ressourcen wie in unserem Fall z.B. Font-Dateien oder fremde JS-Bibliotheken werden durch einen sogenannten *Asset*-Controller aus dem Play-Framework bereitgestellt. Dieser bietet die Möglichkeit alle Dateien in einem Ordner statisch bereitzustellen. In unserem Fall sind das die Dateien im Ordner `"/public"` welche unter der URL `"/assets"` bereitgestellt werden.

3.1.5 Isabelle/Scala Integration

Zunächst ist es für die Arbeit mit Isabelle/Scala wichtig, die

Isabelle/Scala auf
dem Server

3.2 Kommunikation

Da viele Daten in hoher Frequenz übertragen werden müssen, (Nach jeder Veränderung des Dokuments muss der Server informiert werden, der dann zu unbestimmten Zeitpunkten in mehreren Schritten die Zustands- Informationen zurücksendet) ist eine normale Datenübertragung wie bei Webapplikationen üblich über AJAX bzw. HTTP-Anfragen nicht gut geeignet: Bei normalem HTTP ist es zum einen immer nötig auf Browser Seite eine Anfrage zu stellen um Informationen vom Server zu erhalten, zum anderen hat jede Anfrage und jede Antwort zusätzlich einen Header, welcher mindestens einige hundert Bytes groß ist.

Als Worst-Case Beispiel kann das Entfernen von Kommandos aus dem Dokumenten-Modell betrachtet werden: Um das Modell eines Dokuments auf Server und Client synchron zu halten müssen ab und zu Nachrichten vom Server gesendet werden, welche signalisieren, dass ein Kommando aus dem Modell entfernt wurde. Diese Nachricht enthält als Information die eindeutige ID des Kommandos. Bei der ID handelt es sich um eine 64-Bit Zahl. Zusätzlich dazu muss signalisiert werden, um welche Aktion es sich eigentlich handelt. Dafür reichen bei der überschaubaren Anzahl an möglichen Aktionen weitere 4 Byte mehr als aus. Das bedeutet, die eigentlichen Informationen die für diese Aktion relevant sind belaufen sich auf höchstens 12 Byte. Würde diese Aktion über HTTP laufen müsste zunächst eine Anfrage gestellt werden

```
GET /user/project/file.thy/remove-command HTTP/1.1
Host: www.clide.net
```

Diese Anfrage allein ist bereits 70 Zeichen lang ohne dass überhaupt relevante Informationen übertragen wurden. Die minimale Antwort sähe dann in etwa so aus:

```
HTTP/1.1 200 OK
Server: Apache/1.3.29 (Unix) PHP/4.3.4
Content-Length: 12
Content-Language: de
Connection: close
Content-Type: text/html
```

178

Das sind zusammen über 250 Zeichen um zu signalisieren, dass Kommando 178 entfernt werden soll. Damit wurde die Information um den Faktor 30 aufgeblasen. Zusätzlich kommt es zu Verzögerungen durch die zusätzlichen Anfragen.

3.2.1 WebSockets

Die im HTML5-Standard eingeführten WebSockets sind die ideale Lösung für dieses Problem. Bei WebSockets wird eine vollduplex Verbindung über TCP aufgebaut, welche ohne den HTTP-Overhead auskommt und lediglich ein Byte pro Nachricht benötigt um zu signalisieren, dass eine Nachricht endet. Außerdem ist es durch die duplex Verbindung möglich sogenannte Server-Pushes wie in dem vorangegangenen Beispiel ohne vorheriges Polling bzw. eine verzögerte Antwort auf eine Anfrage zu realisieren.

Dadurch, dass WebSockets ein relativ neues Konzept bilden, werden durch deren Anwendung die meisten älteren Browser von der Benutzung der Webapplikation ausgeschlossen. Ein Fallback auf HTTP wäre zwar relativ leicht zu implementieren, aber in der Benutzung kaum akzeptabel, da sich die zusätzlichen Verzögerungen bei teilweise weit über 1000 Nachrichten pro Minute so negativ auf die Ausführungsgeschwindigkeit auswirken würden, dass ein produktives Arbeiten mit dem System nicht mehr möglich wäre. Aktuell werden WebSockets von allen relevanten Browsern in der neusten Version unterstützt. Browser die seit einem Jahr nicht mehr aktualisiert wurden können mit dem Aufruf allerdings zu großen Teilen nichts anfangen. (Siehe 3.3.1) Das Play-Webframework unterstützt Serverseitige WebSocket-Verbindungen. Allerdings muss bei der Benutzung auf viel Funktionalität verzichtet werden. Da WebSockets allerdings so unabdingbar sind werden diese Einschränkungen akzeptiert.

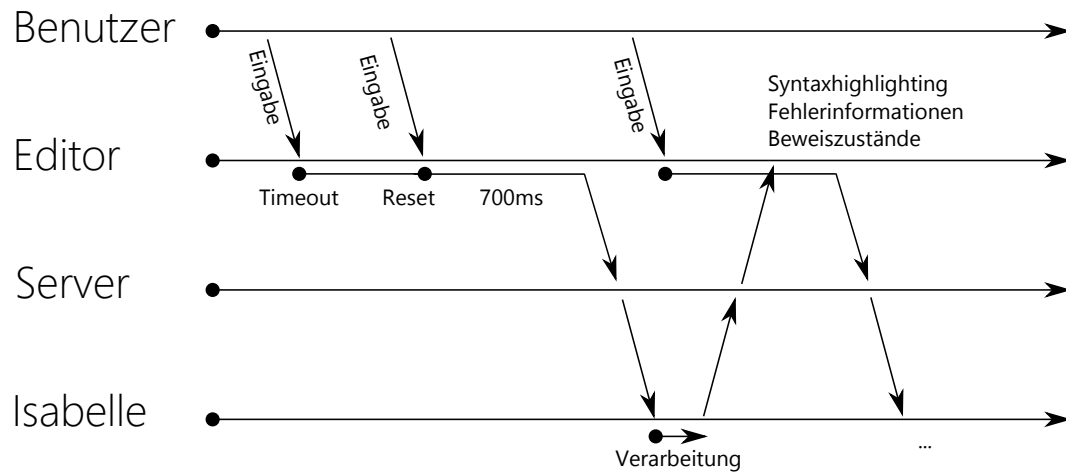


Abbildung 3.1 Datenfluss in clide

3.2.2 Protokoll

Für das Protokoll wurde zunächst der Einfachheit halber JSON gewählt mit der Möglichkeit im Hinterkopf, es zu einem späteren Zeitpunkt leicht durch das komprimierte Binary JSON (BSON)-Protokoll zu ersetzen. Um das möglich zu machen wird unter Verwendung von Dynamischer Typisierung in Scala (Siehe Abschnitt 2.1.1.5) vom Protokoll abstrahiert, (Siehe Abschnitt 4.1) damit es das Übertragungsprotokoll selbst ohne größeren Aufwand ausgetauscht werden kann.

3.3 Client

Zur Realisierung des Clients wird auf die etwas komfortablere Scriptsprache CoffeeScript zurückgegriffen. CoffeeScript-Code ist gegenüber JS-Code deutlich kürzer, (Etwa 30%) und hat eine an Funktionale Sprachen wie Haskell erinnernde Syntax. Da mit den neuen Möglichkeiten in Play 2.1 CoffeeScript problemlos verwendet werden kann (Der Browser erhält kompiliertes JS), entstehen hierdurch keine Nachteile.

Zur Strukturierung wird RequireJS sowie BackboneJS (und damit implizit auch UnderscoreJS) verwendet. Durch die vielfältigen Möglichkeiten dieser Bibliotheken ist es möglich den Code klar zu modularisieren.

3.3.1 Browserkompatibilität

Auf Grund der in 3.2.1 beschriebenen Notwendigkeit, kann auf WebSockets nicht verzichtet werden. Damit sind die meisten älteren Browser nicht mit der Anwendung kompatibel.

	Chrome	Safari	IE	Firefox	Opera
WebSockets	14.0	6.0	10.0	11.0	12.1
History API	5.0	5.0	10.0	4.0	11.5
WebWorkers	4.0	4.0	10.0	3.5	10.6
CSS Transitions	4.0	3.1	10.0	4.0	10.5

Tabelle 3.1 Kompatibilität der gängigsten Browser mit den Verwendeten Standards

Datein von caniuse.com

Kompatibilitäts-Tabelle

Aus Tabelle ?? ist zu entnehmen, dass alle weiteren in der Anwendung benutzten Standards eine geringere oder die gleiche Anforderung an die Aktualität des Browsers haben. Da WebSockets ein sehr neues Konzept sind, dienen sie als Orientierung: Alle Features, welche von jedem Browser, der WebSockets unterstützt, auch unterstützt werden, dürfen verwendet werden. Alle anderen schließen wir aus, da sonst die Zahl der potentiellen Nutzer weiter eingeschränkt würde. Die Anwendung ist damit im Standardbrowser auf allen Systemen mit einem aktuellen Betriebssystem (Windows 8, Ubuntu 12.10, OpenSUSE 12.2, OS X 10.8.2), sowie auf dem iPad, und aktuellen Windows RT Tablets benutzbar. Bei der Entwicklung wurde jedoch besonderes Augenmerk auf WebKit-basierte Browser, insbesondere Google Chrome gelegt und einige der anderen genannten Systeme sind ungetestet und damit ohne Gewähr.

3.3.2 Benutzeroberfläche

Bei der Gestaltung des User Interface (UI) wurde besonderes Augenmerk auf die User Experience gelegt um die Produktivität und die Zufriedenheit der Nutzer zu optimieren. [Nor03] Inspiriert wurde das Design durch die von Microsoft in Windows Phone 7 bzw. Window 8 eingeführte Design Sprache *Metro UI* (bzw. seit 2012 *Microsoft Design Language*). [Mic12] Dabei wird auf unnötige Grafiken verzichtet und die Typographie in den Vordergrund gestellt. Durch die Reduktion auf das Wesentliche entsteht eine neue Ästhetik welche eine willkommene Auffrischung der Icon- und Fensterbasierten Oberflächen wie sie seit jeher bestehen, bietet.

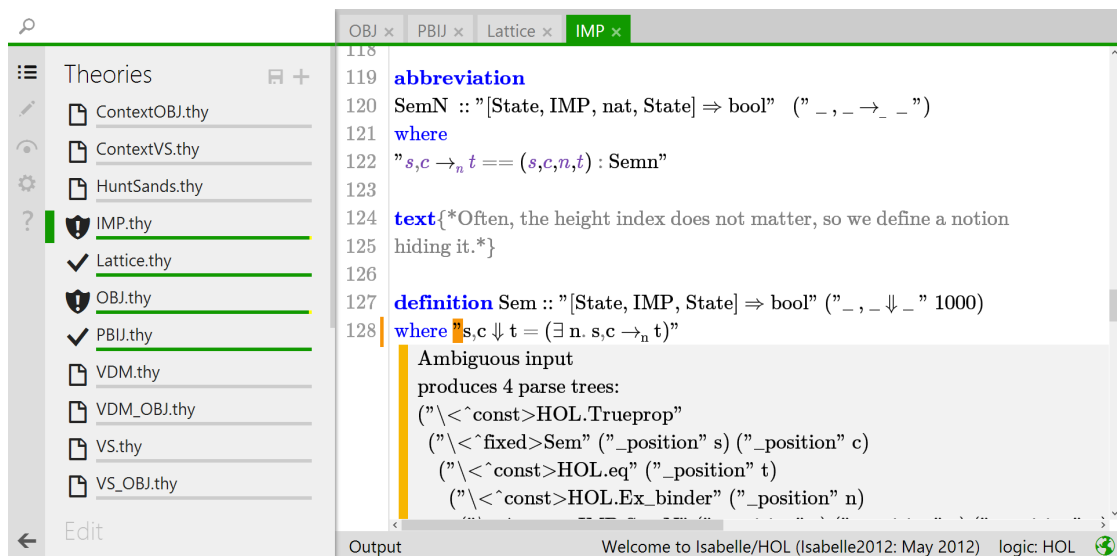


Abbildung 3.2 Die clide-Oberfläche

Besondere Herausforderungen sind das sinnvolle Unterbringen der Beweiszustände und Fehlerinformationen in der Darstellung sowie die Darstellung des Fortschritts, der einzelnen Theorien, welche nicht zwingend vom Nutzer geöffnet werden müssen sondern auch implizit als Abhängigkeiten anderer Beweisdokumente verarbeitet werden können.

3.3.2.1 Login

Der Anmeldebildschirm (Login) ist entsprechend einfach gehalten. Dem Nutzer wird lediglich ein Formular zur Eingabe von Benutzernamen und Passwort präsentiert. (Siehe Abbildung 3.3)

Die Kommunikation erfolgt an dieser Stelle noch über normales HTTP und bei fehlerhaften Eingaben wird auf einer neu geladenen Seite eine Fehlermeldung über dem Formular angezeigt. Das stört an dieser Stelle nicht und eine Nutzung von WebSockets würde die Sache hier nur ver-

komplizieren, da so nicht die bereits ausgereiften Verfahren zur Anmeldung über HTTP genutzt werden könnten.

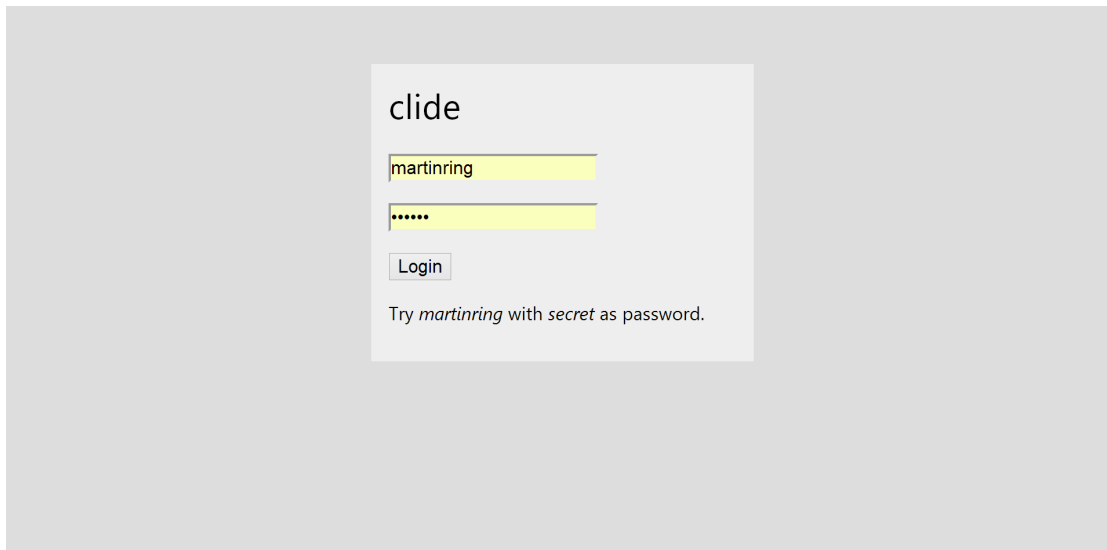


Abbildung 3.3 Das Anmeldeformular

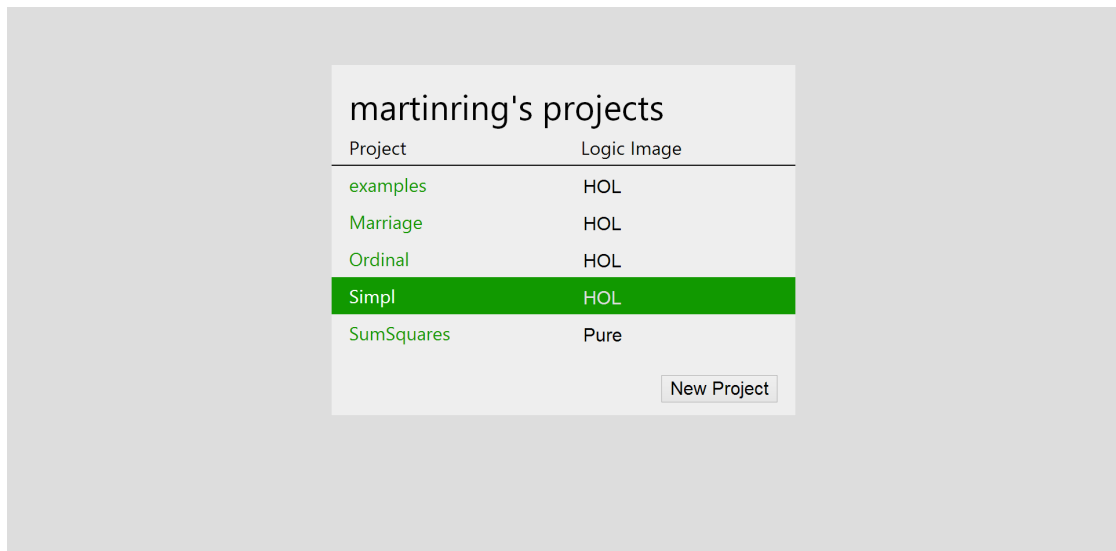
3.3.2.2 Projektübersicht

3.3.2.3 Die Sidebar

3.3.2.4 Die Editor-Komponente

Die wichtigste Benutzerkomponente einer Entwicklungsumgebung ist der Text-Editor. Ein Editor für Isabelle-Code hat hierbei besondere Anforderungen: Während in der Praxis bislang nur rudimentäre Unterstützung für die Darstellung von Isabelle-Sonderzeichen und insbesondere von Sub- und Superskript existierte, hat Isabelle/jEdit bereits eine stärkere Integration dieser eigentlich recht essentiellen Visualisierungen eingeführt. [Wen10] Da bei der HTML-Darstellung kaum Grenzen gesetzt sind und sich CSS-Formatierung sehr leicht dazu benutzen lässt bestimmte Text-Inhalte besonders darzustellen, ist es klar, dass unsere Entwicklungsumgebung an dieser Stelle besonders glänzen soll.

In einem ersten Prototypen war es möglich eine JS-Komponente zu entwickeln, welche es zuließ, Isabelle-Code zu bearbeiten, sodass Sub- und Superskript sowie die Sonderzeichen korrekt dargestellt wurden und bearbeitet werden konnten. Die besondere Anforderung bei ist hierbei nicht die Darstellung sondern vor allem der Umgang mit den variablen Breiten. Selbst wenn ein Monospace-Font verwendet würde, besteht das Problem, dass z.b. bei Sub- und Superskript



The screenshot shows a web interface titled "martinring's projects". It contains a table with two columns: "Project" and "Logic Image". The table lists five projects: "examples", "Marriage", "Ordinal", "Simpl", and "SumSquares". The "Simpl" row is highlighted in green. Below the table is a "New Project" button.

Project	Logic Image
examples	HOL
Marriage	HOL
Ordinal	HOL
Simpl	HOL
SumSquares	Pure

New Project

Abbildung 3.4 Die Projektübersicht

nach Typographischen Standards nur 66% der Textgröße verwendet wird und somit auch die Zeichenbreite geringer wird. Da aber eben die Visualisierung eine besondere Stärke der Anwendung sein soll, wollen wir zusätzlich auch nicht darauf verzichten Ähnliche Fonts zu verwenden, wie in der Ausgabe der LaTeX-Dateien, also auch Mathematische Sonderzeichen nicht in ein Raster quetschen.

Eine weitere besondere Anforderung, welche bislang relativ einmalig zu sein scheint, ist die Tatsache, dass das Syntax-Highlighting zu Teilen auf dem Server stattfinden soll und somit eine Möglichkeit bestehen muss diese zusätzlichen Informationen in die Darstellung zu integrieren.

Zusammenfassend können folgende besondere Anforderungen an die Editor-Komponente formuliert werden: Der Editor muss in der Lage sein

- Syntaxhighlighting zu betreiben,
- Externes Syntaxhighlighting verzögert zu integrieren,
- Schriftarten mit variabler Zeichenbreite anzuzeigen,
- Tooltips für Typinformationen o.ä. anzuzeigen und
- Isabelle-Sonderzeichen zu substituieren.

Da der Hauptsächliche Aufwand bei einer Editor-Komponente nicht darin liegt Text zu bearbeiten und darzustellen sondern vor allem in der Infrastruktur drumherum (Copy/Paste, Suche, Selektieren, Drag'n'Drop, usw.) ist es verlockend, eine fertige Komponente zu verwenden. Hier existieren mehrere ausgereifte Alternativen. Bei genauerer Betrachtung gibt es jedoch keine, welche optimal für unsere Zwecke geeignet ist.

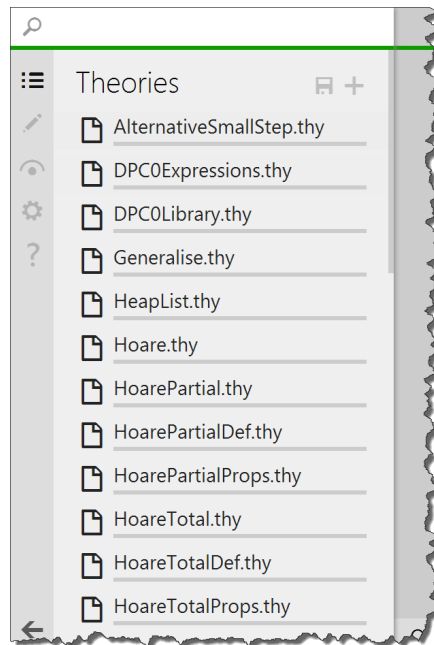


Abbildung 3.5 Die Sidebar

Der **MDK-Editor**² bietet viele Features, wird aber seit 2008 nicht mehr weiter entwickelt und scheidet damit sofort aus.

Der **AJAX.org Cloud9 Editor (ACE)**³ (Ehemals Mozilla SkyWriter) wird momentan sehr stark weiter entwickelt. ACE bietet bereits ein ausgeklügeltes Framework für das Syntaxhighlighting, welches sich in einem Prototyp relativ leicht an das Serverseitige Syntaxhighlighting anbinden ließ. ACE bietet alle Funktionen, welche man von einem Modernen Text-Editor erwartet, hat jedoch einen entscheidenden Nachteil: Zur Darstellung wird aus Performance-Gründen intern ein festes Raster verwendet. Dabei wird davon ausgegangen, dass ein Monospace Font verwendet wird. Von diesem wird einmalig eine Zeichenbreite ermittelt und diese feste Metrik wird dann für alle internen Operationen verwendet. Da diese Designentscheidung so tiefgreifend ist, scheint es nicht realistisch in akzeptabler Zeit, die Komponente so zu modifizieren, dass variable Breiten unterstützt werden können. Außerdem ist es nicht möglich Textstellen durch Sonderzeichen zu Substituieren. Somit scheidet auch ACE für die Verwendung in der Anwendung aus.

CodeMirror⁴ ist ebenfalls eine weit entwickelte Editor-Komponente, welche nicht ganz so umfangreich, wie `acrace` ist, jedoch um einiges flexibler erscheint. In einem Prototyp war es möglich einige eigene Modifikationen für die Darstellung (Sub- Superskript, Tooltips, Hyperlinks) zu integrieren. CodeMirror verwendet kein festes Raster, darunter leidet die

²<http://www.mdk-photo.com/Editor/>

³<http://ace.ajax.org/>

⁴<http://codemirror.net/>

Performanz. Da wir jedoch darauf angewiesen sind, müssen wir diese Einbußen in der Geschwindigkeit akzeptieren. Seit Version 3.0, welche am 10.12.2012 erschien, ist es möglich, Textteile durch HTML-Widgets zu substituieren. Dadurch ist es möglich, Isabelle-Sonderzeichen, welche durch ASCII-Sequenzen wie beispielsweise `\<rightarrow>` für das Zeichen \rightarrow repräsentiert werden, direkt im Editor zu ersetzen, sodass der bearbeitete Text valide Isabelle-Code bleibt, die Darstellung hingegen der eines LaTeX-Dokuments entspricht.

Weitere Editoren existieren zwar, scheiden aber alle aus, da die meisten nicht einmal die Hälfte der oben formulierten Anforderungen erfüllen. Es wird klar, dass CodeMirror der am besten geeignete Editor ist. Trotzdem müssen einige eigene Anpassungen in den Kern des Editors integriert werden. Diese sind unter anderem die Unterstützung von

- Sub- und Superskript (sowie angepassten Cursorpositionen und -höhen),
- Tooltips für einzelne Textabschnitte sowie
- die Darstellung von Hyperlinks im Text.

3.3.2.5 Beweiszustände

3.3.3 Client-Modell

Der Client muss zu jeder Zeit über ein konsistentes Abbild der relevanten Informationen verfügen. Das stellt sich als schwierige Herausforderung heraus. Insbesondere müssen der Inhalt des Texteditors mit dem Server synchron gehalten werden. Abbildung 3.1 zeigt ein vereinfachtes Abbild des Datenflusses in der Anwendung. Die besondere Schwierigkeit liegt darin, dass natürlich nicht jeder Tastendruck übertragen werden kann. Das würde auch wenig Sinn machen, da es nicht realistisch ist, jede einzelne Veränderung zu überprüfen. Den Nutzer würden Fehlermeldungen zu Zwischenzuständen stören und der Server wäre absolut überlastet. Also wird nach jedem Tastendruck ein *Timeout* von 700 Millisekunden gestartet. Wenn es abläuft, ohne dass eine Taste gedrückt wird, werden die Veränderungen im Dokument an den Server übertragen. In dem häufigen Fall eines weiteren Tastendrucks innerhalb der Zeitspanne, wird das Timeout neu gestartet (*Reset*) so lange, bis der Nutzer über den Zeitraum von 700ms keine Veränderung mehr vornimmt.

Die Zeitspanne von 700 Millisekunden hat sich in eigenen Experimenten als guter Wert herausgestellt und ähnliche Werte werden auch von anderen Entwicklungsumgebungen und Editoren (ACE, eclipse, Visual Studio, ...), standardmäßig verwendet, um den Präsentationskompiler zu entlasten.

Implementierung

Im folgenden werden einige interessante Gesichtspunkte der Implementierung herausgearbeitet.

Implementierung

4.1 ScalaConnector / JSConnector

4.2 Clientseitiges Syntaxhighlighting

`isabelle.coffee`

4.3 Synchrone Repräsentation von Dokumenten

`LineBuffer`

Kapitel 5

Bewertung

Bewertung

Kapitel 6

Ausblick

Ausblick

Kapitel 7

Zusammenfassung

Zusammenfassung

Appendix

A.1 Abbildungsverzeichnis

2.1	Ajax Modell einer Web-Anwendung (asynchrone Datenübertragung)	15
3.1	Datenfluss in clide	24
3.2	Die clide-Oberfläche	26
3.3	Das Anmeldeformular	27
3.4	Die Projektübersicht	28
3.5	Die Sidebar	29

A.2 Tabellenverzeichnis

3.1	Kompatibilität der gängigsten Browser mit den Verwendeten Standards	25
-----	---	----

A.3 Literatur

- [al11] Martin Odersky et al. *Programming In Scala*. 2. Aufl. Artima, 2011. ISBN: 0981531644.
- [al12] Makarius Wenzel et al. *The Isabelle/Isar Reference Manual*. 2012. URL: <http://isabelle.in.tum.de/doc/isar-ref.pdf>.
- [al99] R. Fielding et al. *Hypertext Transfer Protocol 1.1*. Techn. Ber. IETF, Juni 1999. URL: <http://tools.ietf.org/html/rfc2616>.
- [Fla97] David Flanagan. *JavaScript: The Definitive Guide*. Second. O'Reilly & Associates, Jan. 1997. ISBN: 1-56592-234-4. URL: <http://www.ora.com/catalog/jscrip2/noframes.html>.

- [Fra11] Maximilian Vollendorf Frank Bongers. *jQuery: Das Praxisbuch*. 2. Aufl. Galileo Computing, 2011. ISBN: 3836218100.
- [Hal12] Philipp Haller. *Actors in Scala*. 1. Aufl. Artima, 2012. ISBN: 0981531652.
- [Hic12] Ian Hickson. *The WebSocket API*. W3C Working Draft (work in progress). W3C, Sep. 2012. URL: <http://www.w3.org/TR/2012/CR-websockets-20120920/>.
- [Jäg07] Kai Jäger. *Ajax in der Praxis: Grundlagen, Konzepte, Lösungen*. 1. Aufl. Springer, 2007. ISBN: 3540693335.
- [Kes12] Anne van Kesteren. *DOM*. W3C Working Draft (work in progress). W3C, Dez. 2012. URL: <http://www.w3.org/TR/2012/WD-dom-20121206/>.
- [Mic12] Microsoft. *UX guidelines for Windows Store apps*. Nov. 2012. URL: <http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx>.
- [Nor03] Donald A. Norman. *Emotional Design: Why We Love (or Hate) Everyday Things*. 1. Aufl. Basic Books, 24. Dez. 2003. ISBN: 0465051359.
- [Osm12] Addy Osmani. *Developing Backbone.js Applications*. 1. Aufl. O'Reilly Media, 2012. ISBN: 1449328253.
- [Smi12] M. K. Smith. *HTML: The Markup Language (an HTML language reference)*. W3C Working Draft (work in progress). W3C, Okt. 2012. URL: <http://www.w3.org/TR/2012/WD-html-markup-20121025/>.
- [Sur12] Josh Sureth. *Iteratees*. Feb. 2012. URL: <http://jsuereth.com/scala/2012/02/29/iteratees.html>.
- [Wen09] Makarius Wenzel. *Parallel Proof Checking in Isabelle/Isar*. Techn. Ber. Technische Universität München, 2009.
- [Wen10] Makarius Wenzel. *Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit*. Techn. Ber. Université Paris Sud 11, LRI, Orsay France, 2010.

A.4 Liste der Abkürzungen

ACE AJAX.org Cloud9 Editor, S. 29, 30

AJAX Asynchronous JavaScript and XML, S. 14, 16, 22

BSON Binary JSON, S. 24

CSS Cascading Style Sheets, S. 11, 12, 21, 27

DOM Document Object Model, S. 11, 16

HTML Hypertext Markup Language, S. 11, 12, 16, 21, 23, 27

HTTP Hypertext Transfer Protocol, S. 14, 15, 22, 26, 27

Isar Intelligible semi-automated reasoning, S. 9

JS JavaScript, S. 8, 11, 13, 14, 16, 17, 19, 21, 25, 27

JSON JavaScript Object Notation, S. 14, 24

JVM Java Virtual Machine, S. 3

LESS Less CSS, S. 21

sbt Simple Build Tool, S. 7, 8

SML Standard ML, S. 9

UI User Interface, S. 26

URL Uniform Resource Loactor, S. 8, 21

XML Extensible Markup Language, S. 11