

Christoph Lüth



Wintersemester 2020/21

I. Eine kurze Einführung in die Webentwicklung

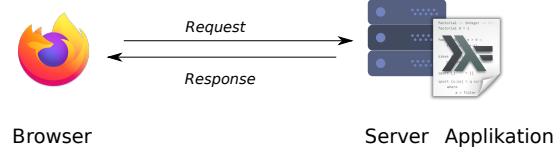
Kennzeichen einer Webanwendung

- ▶ **Zustandsfreiheit:** jeder Request ist ein neuer
- ▶ **Nebenläufigkeit:** ein Server, viele Browser (gleichzeitig)
- ▶ **Entkoppelung:** Serveranwendung und Browser weit entfernt

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
 - ▶ Aktionen und Zustände
 - ▶ Monaden als Berechnungsmuster
 - ▶ **Funktionale Webanwendungen**
 - ▶ Scala — Eine praktische Einführung
 - ▶ Rückblick & Ausblick

Wie funktioniert das Web?



Grober Ablauf

- ① Browser stellt **Anfrage**
 - ▶ *Gib mir Seite /home/carl/*
- ② Server nimmt Anfrage entgegen, **löst** Anfrage auf
 - ▶ */home/carl/, das muss die Datei /var/www/carl/index.html sein.*
- ③ Server sendet Antwort
 - ▶ *Hier ist die Seite: <h1>Hallo</h1><p>Foo ba...*

Verfeinerter Ablauf

- ▶ Das **Protokoll** ist HTTP (RFC 2668, 7540). HTTP kennt vier Arten von Requests: GET, POST, PUT, DELETE.
- ▶ Von der URL `http://www.foo.de/baz/bar/` löst der Server den **Pfad** (`/bar/bar/`) zu einer Resource auf (**Routing**). Das kann eine Datei sein (static routing), oder es wird eine Funktion aufgerufen, die ein Ergebnis erzeugt.
- ▶ HTTP kennt verschiedene Arten von **response codes** (100, 404, ...). Der Inhalt der Antwort ist **beliebig**, und nicht notwendigerweise HTML.

Architekturerwägungen

- ▶ Webanwendungen müssen **zustandsfrei** sein und **skalieren**
- ▶ Übertragung ist **unzuverlässig**.
- ▶ Dazu: **REST** (Representational State Transfer)
 - ▶ Sammlung von **Architekturprinzipien**
- ▶ Dazu: **CRUD** (create, read, update, delete)

Merkmale von REST-Architekturen

- ① Zustandslosigkeit — jede Nachricht in sich vollständig
- ② Caching
- ③ Einheitliche Schnittstelle:
 - Adressierbare Ressourcen — als URL
 - Repräsentation zur Veränderungen von Ressourcen
 - Selbstbeschreibende Nachrichten
 - *Hypermedia as the engine of the application state* (HATEOAS)
- ④ Architektur: Client-Server, mehrschichtig

Pi3 WS 20/21

9 [30]



Anatomie einer Web-Application

- **Routing:** Auflösen der Pfade zu Aktionen
- Eigentliche Aktion
- **Persistentes Backend**
- Erzeugung von HTML (meistens), JSON (manchmal)

Pi3 WS 20/21

10 [30]



Hands on!

Übung 12.1: HTTP handgemacht

Wenn nötig, installieren Sie das Programm `telnet` (oder schalten es, in Windows, frei). Starten Sie das Programm und verbinden Sie sich mit dem Web-Server der Uni Bremen:

`telnet> open www.uni-bremen.de 80`

Jetzt können Sie ein HTTP-Request von Hand eingeben (danach eine Leerzeile eingeben):

```
GET / HTTP/1.1  
Host: www.uni-bremen.de
```

Was passiert?

Pi3 WS 20/21

11 [30]



II. Web Development in Haskell

Ein erster Eindruck

```
{-# LANGUAGE OverloadedStrings #-}  
  
import Web.Scotty  
  
import Data.Monoid (mconcat)  
  
main = scotty 3000 $  
    get "/:word" $ do  
        beam ← param "word"  
        html $ mconcat ["<h1>Scotty, ", beam, "<h1>"]  
  
(Auch von der Webseite.)
```

Pi3 WS 20/21

15 [30]



Der Server antwortet

Lösung: Wahrscheinlich kommt die Antwort:

```
HTTP/1.1 301 Moved Permanently  
Date: Sun, 31 Jan 2021 01:52:40 GMT  
Server: Apache/2.4.29 (Ubuntu)  
Location: https://www.uni-bremen.de/  
Content-Length: 317  
Content-Type: text/html; charset=iso-8859-1
```

gefolgt von etwas HTML. Damit teilt uns der Server mit, dass er nur noch https-Anfragen annimmt (und sagt uns, wo). Eventuell kommt auch

```
HTTP/1.1 400 Bad Request
```

...

dann war die Anfrage nicht HTTP-konform (wahrscheinlich vertippt?).

Pi3 WS 20/21

12 [30]



Scotty: ein einfaches Web-Framework

From the web-page <https://hackage.haskell.org/package/scotty>:

- Scotty is the cheap and cheerful way to write RESTful, declarative web applications.
- A page is as simple as defining the verb, url pattern, and Text content.
 - It is template-language agnostic. Anything that returns a Text value will do.
 - Conforms to WAI Application interface.
 - Uses very fast Warp webserver by default.

Pi3 WS 20/21

14 [30]



Ein erstes Problem

- Repräsentation von Zeichenketten als `type String = [Char]` ist elegant, aber benötigt Platz und ist langsam.
- Daher gibt es mehrere Alternativen:
 - `Data.Text` Unicode-Text, strikt und schnell
 - `Data.Text.Lazy`, Unicode-Text, String kann größer sein als der Speicher
 - `Data.ByteString` Sequenzen von Bytes, kein Unicode, kompakt
- Deshalb `mconcat` [...] oben (`class Monoid`)
- String-Literale können überladen werden (`LANGUAGE OverloadedStrings`)
- Mit `pack` und `unpack` Konversion von Strings in oder von `Text`.
- Potentielle Quelle der Verwirrung: Scotty nutzt `Text.Lazy`, Blaze nutzt `Text`.

Pi3 WS 20/21

16 [30]



HTML

- ▶ Scotty gibt nur den Inhalt zurück, aber wir wollen HTML erzeugen.
- ▶ Drei Möglichkeiten:
 - ① Text selber zusammensetzen: "<h1>Willkommen!</h1>\n"
 - ② Templating: HTML-Dokumente durch Haskell anreichern lassen (Hamlet, Heist)
 - ③ Zugrundeliegende Struktur (DOM) in Haskell erzeugen, und in Text konvertieren.

Pi3 WS 20/21

17 [30]



Erzeugung von HTML: Blaze

Selbstbeschreibung: <https://jaspervdj.be/blaze/>

BlazeHTML is a blazingly fast HTML combinator library for the Haskell programming language. It embeds HTML templates in Haskell code for optimal efficiency and compositability.

- ▶ Kann (X)HTML4 und HTML5 erzeugen.
- ▶ Dokument wird als Monade repräsentiert und wird durch Kombinatoren erzeugt:

```
numbers :: Int -> Html
numbers n = doctypeHtml $ do
    H.head $ do
        H.title "NaturalNumbers"
    body $ do
        p "A list of natural numbers:"
        ul $ forM_ [1 .. n] (li . toHtml)

image = img ! src "foo.png" ! alt "A foo image."
```

- ▶ Siehe Tutorial.

Pi3 WS 20/21

18 [30]



Persistenz

- ▶ Eine Web-Applikation muss **Zustände** verwalten können
 - ▶ Nutzerdaten, Warenbestand, Einkauf, ...
- ▶ Üblicher Ansatz: **Datenbank**
 - ▶ ACID-Eigenschaften garantiert, insbesondere Nebenläufigkeit
 - ▶ Aber: externe Anbindung nötig
- ▶ Hier: **Mutable Variables MVar** a (nicht durable, aber schnell und einfach)

Pi3 WS 20/21

19 [30]



Nebenläufige Zustände

- ▶ Haskell ist **nebenläufig** (hier ein Thread pro Verbindung)
- ▶ **MVar** α sind synchronisierte veränderliche Variablen.
- ▶ Kann **leer** oder **gefüllt** sein.
 - newMVar :: α -> IO (MVar α)
 - readMVar :: MVar α -> IO α — MVar bleibt gefüllt
 - takeMVar :: MVar α -> IO α — MVar danach leer
 - putMVar :: MVar α -> α -> IO () — Füllt MVar
- ▶ **readMVar** und **takeMVar** **blockieren**, wenn Variable leer ist
- ▶ Erlaubt einfache Synchronisation (vgl. **synchronized** in Java)

Pi3 WS 20/21

20 [30]



Zustand

- ▶ Wie können wir den Benutzer **identifizieren**?
- ▶ Ein Ansatz: **Cookies**
 - ▶ Widerspricht dem REST-Ansatz.
- ▶ Hier: über die URL — jeder Benutzer bekommt eine Resource

Pi3 WS 20/21

21 [30]



Erste Schritte.

Übung 12.2: Lecker Kekse!

Laden Sie die Quellen für die Vorlesung von heute herunter, und betreten Sie das Verzeichnis `simple-cookies`. Dort finden Sie ein einfaches Programm, welches mit Cookies zählt, wie oft eine Seite aufgerufen wurde, und das obligatorische Hello World, welches ein einfaches HTML-Dokument mittels Blaze erzeugt. Erweitern Sie das Programm `SimpleCookies.hs` so dass es ein HTML-Dokument zurückgibt. Schreiben Sie dazu eine Funktion

`hitPage :: String -> Html`

welche über die Anzahl der bisherigen Aufrufe (`hits`) parametrisiert ist.

Pi3 WS 20/21

22 [30]



Eine mögliche Lösung

Lösung:

```
hitPage :: String -> Html
hitPage hits =
  doctypeHtml $ do
    H.head $ do
      H.title "SimpleCookies"
    H.body $ do
      H.h1 "HelloWorld."
      H.span $ do
        H.text "You have been here "
        H.text (T.pack hits)
        H.text " times."
```

Pi3 WS 20/21

23 [30]



III. Ein Web-Shop für Onkel Bob

Pi3 WS 20/21

24 [30]



Architektur des Web-Shop

Model-View-Controller-Paradigma (Entwurfsmuster):

- Das **Model** ist der eigentliche (und persistente) Teil der Anwendung, bestehend aus den Datentypen samt der Funktionen darauf.
- Die **Views** sind Funktionen, die Webseiten aufbauen.
- Der **Controller** übersetzt Anfragen von außen in die Aufrufe der Model-Funktionen, erzeugt aus den Ergebnissen mit den Views Webseiten und schickt diese wieder zurück.

Pi3 WS 20/21

25 [30]



Entwurf der Anwendung

Resource	Methode	Daten
/	GET	Home-Page: Angebote anzeigen. Link zu neuem Einkauf
/einkauf/neu	GET	Neuen Einkauf beginnt, Einkaufswagen wird zugewiesen. Dann Weiterleitung zu folgender: Einkaufswagen darstellen
/einkauf/:id	GET	Link zur Bezahlseite
/einkauf/:id	POST	Angegebene Produkte in den Einkaufswagen
/einkauf/:id/kasse	GET	Bezahlseite mit Rechnung. Link zur Home-Page
/einkauf/:id/kaufen	GET	Bezahlt, Einkaufswagen löschen
/einkauf/:id/abbruch	GET	Abgebrochen, Produkte zurück
/einkauf/lieferung	POST	Anlieferung von Artikeln
/einkauf/lager	GET	Lagerbestand als JSON-Objekt

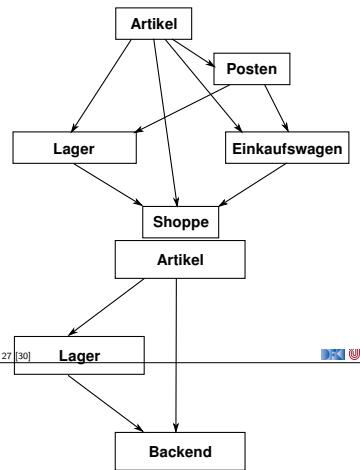
Pi3 WS 20/21

26 [30]



Model: der Shop

- Einheitliches Interface des Shop.
- Verwaltet Menge von **Einkäufen** (Einkaufswagen),
- Funktionen (Auszug):
 - Neuer Einkaufswagen
 - Produkt in Einkaufswagen
 - Einkauf abschließen/abbrechen
- Rein funktional, ADT **Shop** α
- Änderungen:
 - Einheitliche Mengen
 - Posten nicht mehr als ADT
 - Einkaufswagen nicht mehr als Modul



Pi3 WS 20/21

27 [30]



Controller

- Persistiert den **Zustand** des Shop (nur für Laufzeit des Servers)
- Nutzt **UUID** zur Zuordnung des Einkauf (garantiert eindeutige Bezeichner)
- Zugriff** auf den Shop:
 - Ändern** (muss synchronisieren)
 - Lesen** (ohne Synchronisation)

Pi3 WS 20/21

28 [30]



View

- Erzeugt Seiten (Templates):


```

homePage :: Text → [(Posten, Int)] → Html
shoppingPage :: String → String → [Text] → [(Posten, Int)]
          → Int → [Posten] → Html
checkoutPage :: String → String → [(Posten, Int)] → Int → Html
thankYouPage :: Text → Html
      
```
- Weitere Funktionen: Artikelname, Mengeneinheiten, Euros etc.
- Artikel werden über eine eindeutige Kennung ([articleId](#)) identifiziert.

Pi3 WS 20/21

29 [30]



Zusammenfassung

- Wichtige Prinzipien für Web-Anwendungen:
 - Nebenläufigkeit, Zustandsfreiheit, REST
- Haskell ist für Web-Development gut geeignet:
 - Zustandsfreiheit macht Nebenläufigkeit einfach
 - Bequeme Manipulation von Bäumen
 - Abstraktionsbildung
- Web-Programmierung ist **umständlich**.

Pi3 WS 20/21

30 [30]

