Fortgeschrittene Funktionale Programmierung in Haskell

Universität Bielefeld, Sommersemester 2015

Jonas Betzendahl & Stefan Dresselhaus

Übersicht I

Wir wollen mal so richtig Dampf ablassen und bauen uns dafür eine Ranting-Platform (RantR).

Wir wollen mal so richtig Dampf ablassen und bauen uns dafür eine Ranting-Platform (RantR).

Wir möchten eine Platform, auf der jeder (nach Anmeldung) einen Text posten und ihn der Welt zeigen kann. Außerdem möchten wird, dass die User ihre Äußerungen auch wieder löschen können.

Wir wollen mal so richtig Dampf ablassen und bauen uns dafür eine Ranting-Platform (RantR).

Wir möchten eine Platform, auf der jeder (nach Anmeldung) einen Text posten und ihn der Welt zeigen kann. Außerdem möchten wird, dass die User ihre Äußerungen auch wieder löschen können.

Das Ganze wird dann chronologisch sortiert (neueste Rants zuerst) angezeigt.

Natürlich beginnen wir damit, uns erstmal Gedanken über die Datentypen zu machen.

Natürlich beginnen wir damit, uns erstmal Gedanken über die Datentypen zu machen.

Wir brauchen:

Natürlich beginnen wir damit, uns erstmal Gedanken über die Datentypen zu machen.

Wir brauchen:

- Einen Rant, bestehend aus
  - Titel
  - Text
  - Datum

Natürlich beginnen wir damit, uns erstmal Gedanken über die Datentypen zu machen.

Wir brauchen:

- Einen Rant, bestehend aus
  - Titel
  - Text
  - Datum
- Einen User, der sich einloggen kann

Natürlich beginnen wir damit, uns erstmal Gedanken über die Datentypen zu machen.

Wir brauchen:

- Einen Rant, bestehend aus
  - Titel
  - Text
  - Datum
- Einen User, der sich einloggen kann
  Dies stellt `Yesod` schon automatisch zur Verfügung

Natürlich beginnen wir damit, uns erstmal Gedanken über die Datentypen zu machen.

Wir brauchen:

- Einen Rant, bestehend aus
  - Titel
  - Text
  - Datum
- Einen User, der sich einloggen kann
  Dies stellt `Yesod` schon automatisch zur Verfügung

Dieses reicht für unsere Demo-Applikation.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.
Die **Route** gibt einen Endpukt der Applikation an. So ist z.B. `/auth` für die Authentifizierung verantwortlich.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.
Die **Route** gibt einen Endpukt der Applikation an. So ist z.B. `/auth` für die Authentifizierung verantwortlich.
Routen sind die Teile der URL, die hinter der Domain stehen. Somit hat `http://techfak.de/webmail` die Route `/webmail`.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.
Die **Route** gibt einen Endpukt der Applikation an. So ist z.B. `/auth` für die Authentifizierung verantwortlich.
Routen sind die Teile der URL, die hinter der Domain stehen. Somit hat `http://techfak.de/webmail` die Route `/webmail`.
Routen sind also die Adressen mit denen der User auf unsere Applikation zugreift. Wird keine passende Route gefunden, bekommt der User ein 404.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.
Die **Route** gibt einen Endpukt der Applikation an. So ist z.B. `/auth` für die Authentifizierung verantwortlich.
Routen sind die Teile der URL, die hinter der Domain stehen. Somit hat `http://techfak.de/webmail` die Route `/webmail`.
Routen sind also die Adressen mit denen der User auf unsere Applikation zugreift. Wird keine passende Route gefunden, bekommt der User ein 404.
Jede Route in `Yesod` unterstützt zwei Modi: **GET** und **POST**. GET ruft hierbei eine Seite ab, währen POST dem Server Daten schickt (z.B. ein Formular) und das Ergebnis abruft.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.
Die **Route** gibt einen Endpukt der Applikation an. So ist z.B. `/auth` für die Authentifizierung verantwortlich.
Routen sind die Teile der URL, die hinter der Domain stehen. Somit hat `http://techfak.de/webmail` die Route `/webmail`.
Routen sind also die Adressen mit denen der User auf unsere Applikation zugreift. Wird keine passende Route gefunden, bekommt der User ein 404.
Jede Route in `Yesod` unterstützt zwei Modi: **GET** und **POST**. GET ruft hierbei eine Seite ab, währen POST dem Server Daten schickt (z.B. ein Formular) und das Ergebnis abruft.
Die **Handler** sind die Funktionen innerhalb unserer Applikation, die aufgerufen werden, wenn der User eine Route gewählt hat.

Im Web-Kontext begegnet man sogenannten **Routen**. Zu jeder Route gehört ein **Handler**.
Die **Route** gibt einen Endpukt der Applikation an. So ist z.B. `/auth` für die Authentifizierung verantwortlich.
Routen sind die Teile der URL, die hinter der Domain stehen. Somit hat `http://techfak.de/webmail` die Route `/webmail`.
Routen sind also die Adressen mit denen der User auf unsere Applikation zugreift. Wird keine passende Route gefunden, bekommt der User ein 404.
Jede Route in `Yesod` unterstützt zwei Modi: **GET** und **POST**. GET ruft hierbei eine Seite ab, währen POST dem Server Daten schickt (z.B. ein Formular) und das Ergebnis abruft.
Die **Handler** sind die Funktionen innerhalb unserer Applikation, die aufgerufen werden, wenn der User eine Route gewählt hat.
Für jede Route gibt es somit 2 Handler: `getRouteR` und `postRouteR`, die wir implementieren müssen.

Wir benötigen für unsere Applikation die folgenden Routen:

Wir benötigen für unsere Applikation die folgenden Routen:

- `Home` (GET) zum Sehen der letzten Posts

Wir benötigen für unsere Applikation die folgenden Routen:

- `Home` (GET) zum Sehen der letzten Posts
- `Rant` (GET) zum Anzeigen des Rant-Formulars

Wir benötigen für unsere Applikation die folgenden Routen:

- `Home` (GET) zum Sehen der letzten Posts
- `Rant` (GET) zum Anzeigen des Rant-Formulars
- `Rant` (POST) zum Speichern des Rants in der Datenbank

Wir benötigen für unsere Applikation die folgenden Routen:

- `Home` (GET) zum Sehen der letzten Posts
- `Rant` (GET) zum Anzeigen des Rant-Formulars
- `Rant` (POST) zum Speichern des Rants in der Datenbank
- `DelRant/#id` (GET) zum Anzeigen: „Wollen Sie Rant `#id` löschen?“

Wir benötigen für unsere Applikation die folgenden Routen:

- `Home` (GET) zum Sehen der letzten Posts
- `Rant` (GET) zum Anzeigen des Rant-Formulars
- `Rant` (POST) zum Speichern des Rants in der Datenbank
- `DelRant/#id` (GET) zum Anzeigen: „Wollen Sie Rant `#id` löschen?“
- `DelRant/#id` (POST) zum Löschen des Rants

Wir benötigen für unsere Applikation die folgenden Routen:

- `Home` (GET) zum Sehen der letzten Posts
- `Rant` (GET) zum Anzeigen des Rant-Formulars
- `Rant` (POST) zum Speichern des Rants in der Datenbank
- `DelRant/#id` (GET) zum Anzeigen: „Wollen Sie Rant `#id` löschen?“
- `DelRant/#id` (POST) zum Löschen des Rants
- `Auth` (GET/POST) zum Einloggen (wird von `Yesod` gestellt)

Im Folgenden nehmen wir an, dass `Yesod` bereits richtig installiert ist. Für Rückfragen hierzu stehen wir in den Tutorien gerne zur Verfügung.

Yesod bietet ein sogenanntes **Scaffolding** an, welches einem eine rudimentäre Hello-World-App erstellt. Von diesem Punkt aus starten wir und bauen unsere Plattform.

`Yesod` bietet ein sogenanntes **Scaffolding** an, welches einem eine rudimentäre Hello-World-App erstellt. Von diesem Punkt aus starten wir und bauen unsere Plattform.
Der Befehl hierzu lautet `yesod init`. Nach der Beantwortung der Fragen hat man einen Unterordner mit seinem Projektnamen.

Yesod bietet ein sogenanntes **Scaffolding** an, welches einem eine rudimentäre Hello-World-App erstellt. Von diesem Punkt aus starten wir und bauen unsere Plattform.
Der Befehl hierzu lautet `yesod init`. Nach der Beantwortung der Fragen hat man einen Unterordner mit seinem Projektnamen. Hier drin muss man nun noch die Applikation bauen. Dies geht einfach durch die folgenden Befehle:

```
cabal sandbox init
cabal install --only-dependencies
yesod devel
```

Yesod bietet ein sogenanntes **Scaffolding** an, welches einem eine rudimentäre Hello-World-App erstellt. Von diesem Punkt aus starten wir und bauen unsere Plattform.
Der Befehl hierzu lautet `yesod init`. Nach der Beantwortung der Fragen hat man einen Unterordner mit seinem Projektnamen. Hier drin muss man nun noch die Applikation bauen. Dies geht einfach durch die folgenden Befehle:

```
cabal sandbox init
cabal install --only-dependencies
yesod devel
```

Letzteres startet den Development-Server und wir können auf `http://localhost:3000` die Hello-World-App bewundern.

Demo

`Yesod` folgt einem Model-View-Controller-Prinzip.

Yesod folgt einem Model-View-Controller-Prinzip.
Das bedeutet, dass

- die Datenbank das Model ist

Yesod folgt einem Model-View-Controller-Prinzip.
Das bedeutet, dass

- die Datenbank das Model ist
  (Hier werden allen Informationen gespeichert)

Yesod folgt einem Model-View-Controller-Prinzip.
Das bedeutet, dass

- die Datenbank das Model ist
  (Hier werden allen Informationen gespeichert)
- die View die HTML-Ausgabe ist und

Yesod folgt einem Model-View-Controller-Prinzip.
Das bedeutet, dass

- die Datenbank das Model ist
  (Hier werden allen Informationen gespeichert)
- die View die HTML-Ausgabe ist und
- die App der Controller ist.

Yesod folgt einem Model-View-Controller-Prinzip.
Das bedeutet, dass

- die Datenbank das Model ist
  (Hier werden allen Informationen gespeichert)
- die View die HTML-Ausgabe ist und
- die App der Controller ist.
  (Hier werden Anfragen entgegen genommen und unter zuhilfenahme des Models eine Ausgabe in der View generiert.)

`Yesod` folgt einem Model-View-Controller-Prinzip.
Das bedeutet, dass

- die Datenbank das Model ist
  (Hier werden allen Informationen gespeichert)
- die View die HTML-Ausgabe ist und
- die App der Controller ist.
  (Hier werden Anfragen entgegen genommen und unter zuhilfenahme des Models eine Ausgabe in der View generiert.)

Da wir in `Haskell` sind, sind diese Bereiche strikt getrennt, indem man für jeden Bereich eine Monade nimmt.

Typischenweise schreibt man bei `Yesod` einfach nur die Handler, die einen Request nehmen und dann ein Ergebnis in der View-Monade liefern.

Typischenweise schreibt man bei `Yesod` einfach nur die Handler, die einen Request nehmen und dann ein Ergebnis in der View-Monade liefern.
Hierzu starten wir in der „App“-Monade. Von hier aus haben wir die Option

Typischenweise schreibt man bei `Yesod` einfach nur die Handler, die einen Request nehmen und dann ein Ergebnis in der View-Monade liefern.
Hierzu starten wir in der „App“-Monade. Von hier aus haben wir die Option

- über **runDB** eine Datenbank-Aktion zu starten,

Typischenweise schreibt man bei `Yesod` einfach nur die Handler, die einen Request nehmen und dann ein Ergebnis in der View-Monade liefern.
Hierzu starten wir in der „App“-Monade. Von hier aus haben wir die Option

- über **runDB** eine Datenbank-Aktion zu starten,
- mittels **liftIO** irgendetwas zu tun,

Typischenweise schreibt man bei Yesod einfach nur die Handler, die einen Request nehmen und dann ein Ergebnis in der View-Monade liefern.
Hierzu starten wir in der „App“-Monade. Von hier aus haben wir die Option

- über **runDB** eine Datenbank-Aktion zu starten,
- mittels **liftIO** irgendetwas zu tun,
- und am Ende über einen Layouting-Mechanismus (wie **defaultLayout**) eine Ausgabe erzeugen

Typischenweise schreibt man bei `Yesod` einfach nur die Handler, die einen Request nehmen und dann ein Ergebnis in der View-Monade liefern.
Hierzu starten wir in der „App“-Monade. Von hier aus haben wir die Option

- über **runDB** eine Datenbank-Aktion zu starten,
- mittels **liftIO** irgendetwas zu tun,
- und am Ende über einen Layouting-Mechanismus (wie **defaultLayout**) eine Ausgabe erzeugen
- oder einen anderen Handler aufzurufen (der dann ein Ausgabe erzeugt).

Um die interne Repräsentation in der Datenbank zu ändern, müssen wir lediglich die Datei `config/models` editieren.

Um die interne Repräsentation in der Datenbank zu ändern, müssen wir lediglich die Datei `config/models` editieren.
Zusätzlich zu dem Vorgegebenen fügen wir nun unsere Rant-Struktur ein:

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

Um die interne Repräsentation in der Datenbank zu ändern, müssen wir lediglich die Datei `config/models` editieren.
Zusätzlich zu dem Vorgegebenen fügen wir nun unsere Rant-Struktur ein:

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

Nach dem Speichern der Datei erkennt der Development-Server die Änderungen, kompiliert alles neu und passt die Datenbank an.

Um die interne Repräsentation in der Datenbank zu ändern, müssen wir lediglich die Datei `config/models` editieren.
Zusätzlich zu dem Vorgegebenen fügen wir nun unsere Rant-Struktur ein:

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

Nach dem Speichern der Datei erkennt der Development-Server die Änderungen, kompiliert alles neu und passt die Datenbank an.
Das war schon alles. Wir müssen uns nicht mit `SQL` oder ähnlichem herumschlagen.

Außerdem generiert Yesod aus

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

Außerdem generiert Yesod aus

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

die folgende Haskell-Datenstruktur,

```haskell
data Rant = Rant { rantTitel :: RantTitel
                 , rantInhalt :: RantInhalt
                 , rantErstellt :: RantErstellt
                 }
```

Außerdem generiert Yesod aus

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

die folgende Haskell-Datenstruktur,

```haskell
data Rant = Rant { rantTitel :: RantTitel
                 , rantInhalt :: RantInhalt
                 , rantErstellt :: RantErstellt
                 }
```

die automatisch über `import Import` importiert wird. Diese stellt natürlich die normalen record-accessor-Funktionen zur Verfügung.

Außerdem generiert Yesod aus

```
Rant
    titel Text
    inhalt Text
    erstellt UTCTime default=now()
```

die folgende Haskell-Datenstruktur,

```haskell
data Rant = Rant { rantTitel :: RantTitel
                 , rantInhalt :: RantInhalt
                 , rantErstellt :: RantErstellt
                 }
```

die automatisch über `import Import` importiert wird. Diese stellt natürlich die normalen record-accessor-Funktionen zur Verfügung. `RantTitel`, `RantInhalt` sind nur Aliase für `Text`, `RantErstellt` für `UTCTime`

Da wir auch HTML anzeigen wollen, müssen wir auch eine Möglichkeit haben HTML zu generieren.

Da wir auch HTML anzeigen wollen, müssen wir auch eine Möglichkeit haben HTML zu generieren.
Yesod macht dies über sogenannte **widgets**.

Da wir auch HTML anzeigen wollen, müssen wir auch eine Möglichkeit haben HTML zu generieren.
`Yesod` macht dies über sogenannte **widgets**.
Man kann verschiedenste Dinge in diese Widgets stecken (HTML, CSS, JS) und `Yesod` kümmert sich darum, dass diese an die richtige Position kommen (HTML an die Stelle, CSS in den `<head>`, JS am Ende des `<body>`

Da wir auch HTML anzeigen wollen, müssen wir auch eine Möglichkeit haben HTML zu generieren.
`Yesod` macht dies über sogenannte **widgets**.
Man kann verschiedenste Dinge in diese Widgets stecken (HTML, CSS, JS) und `Yesod` kümmert sich darum, dass diese an die richtige Position kommen (HTML an die Stelle, CSS in den `<head>`, JS am Ende des `<body>`
Wir begnügen uns damit, simples HTML zu schreiben. Die wichtigste Funktion hierbei ist der QuasiQuoter `[hamlet]`. Dieser wandelt (Pseudo-)HTML in richtiges HTML um, mit dem `Yesod` umgehen kann.

Beispiel:

```
let html = [hamlet|
    <h1>Hi!
    <p>
        Willkommen auf meiner Seite!
|]
```

Beispiel:

```
let html = [hamlet|
    <h1>Hi!
    <p>
        Willkommen auf meiner Seite!
|]
```

Dieses generiert:

```
<h1>Hi!</h1>
<p>Willkommen auf meiner Seite!</p>
```

Beispiel:

```
let html = [hamlet|
    <h1>Hi!
    <p>
        Willkommen auf meiner Seite!
|]
```

Dieses generiert:

```
<h1>Hi!</h1>
<p>Willkommen auf meiner Seite!</p>
```

Hamlet kümmert sich also darum, dass die Tags geschlossen sind (durch die Einrückungstiefe). Man kann die Tags auch manuell schliessen und mit Variablen arbeiten.

```
let greet name = [hamlet|
    <h1>Hi #{name}!
    <p>
        <a href=@AuthR>Log dich doch ein!</a>
|]
```

```
let greet name = [hamlet|
    <h1>Hi #{name}!
    <p>
        <a href=@AuthR>Log dich doch ein!</a>
|]
```

Dieses generiert für greet „Stefan“:

```
<h1>Hi Stefan!</h1>
<p><a href="/auth/login">Log dich doch ein!</a></p>
```

```
let greet name = [hamlet|
    <h1>Hi #{name}!
    <p>
        <a href=@AuthR>Log dich doch ein!</a>
|]
```

Dieses generiert für `greet „Stefan“`:

```
<h1>Hi Stefan!</h1>
<p><a href="/auth/login">Log dich doch ein!</a></p>
```

Wir können also keine Links mehr falsch setzen, weil Hamlet automatisch Routen (hier `@AuthR`) ersetzt und diese durch den Compiler(!) geprüft werden.

```
let greet name = [hamlet|
    <h1>Hi #{name}!
    <p>
        <a href=@AuthR>Log dich doch ein!</a>
|]
```

Dieses generiert für `greet „Stefan“`:

```
<h1>Hi Stefan!</h1>
<p><a href="/auth/login">Log dich doch ein!</a></p>
```

Wir können also keine Links mehr falsch setzen, weil Hamlet automatisch Routen (hier `@AuthR`) ersetzt und diese durch den Compiler(!) geprüft werden.
Auch sorgt Hamlet dafür, dass keine bösen Dinge in unsere Seite wandern, weil Sonderzeichen wie < und > escaped werden.

Ein simpler Hello-World-Handler wäre zum Beispiel:

```
HomeR :: Handler HTML
HomeR = defaultLayout $ [whamlet|
        <h1>Hello World!
        |]
```

Ein simpler Hello-World-Handler wäre zum Beispiel:

```
HomeR :: Handler HTML
HomeR = defaultLayout $ [whamlet|
        <h1>Hello World!
        |]
```

Das `[whamlet| ... |]` steht hierbei für `toWidget [hamlet| ... |]`.

Demo

Da wir das Model eben schon hinzugefügt haben, fügen wir nun alle Handler hinzu, die wir in unserer Applikation haben wollen, mittels

```
yesod add-handler
Name of route (without trailing R): Rant
Enter route pattern (ex: /entry/#EntryId): /rant
Enter space-separated list of methods (ex: GET POST): GET POST
```

Da wir das Model eben schon hinzugefügt haben, fügen wir nun alle Handler hinzu, die wir in unserer Applikation haben wollen, mittels

```
yesod add-handler
Name of route (without trailing R): Rant
Enter route pattern (ex: /entry/#EntryId): /rant
Enter space-separated list of methods (ex: GET POST): GET POST
```

Analog machen wir dies für alle Handler, die wir haben wollen.

Da wir das Model eben schon hinzugefügt haben, fügen wir nun alle Handler hinzu, die wir in unserer Applikation haben wollen, mittels

```
yesod add-handler
Name of route (without trailing R): Rant
Enter route pattern (ex: /entry/#EntryId): /rant
Enter space-separated list of methods (ex: GET POST): GET POST
```

Analog machen wir dies für alle Handler, die wir haben wollen.
In diesem Fall werden automatisch folgende Dateien bearbeitet/erstellt:

- **config/routes** wird durch `/rant RantR GET POST` erweitert

Da wir das Model eben schon hinzugefügt haben, fügen wir nun alle Handler hinzu, die wir in unserer Applikation haben wollen, mittels

```
yesod add-handler
Name of route (without trailing R): Rant
Enter route pattern (ex: /entry/#EntryId): /rant
Enter space-separated list of methods (ex: GET POST): GET POST
```

Analog machen wir dies für alle Handler, die wir haben wollen.
In diesem Fall werden automatisch folgende Dateien bearbeitet/erstellt:

- **config/routes** wird durch `/rant RantR GET POST` erweitert
- **Application.hs** importiert unsere neue Route automatisch

Da wir das Model eben schon hinzugefügt haben, fügen wir nun alle Handler hinzu, die wir in unserer Applikation haben wollen, mittels

```
yesod add-handler
Name of route (without trailing R): Rant
Enter route pattern (ex: /entry/#EntryId): /rant
Enter space-separated list of methods (ex: GET POST): GET POST
```

Analog machen wir dies für alle Handler, die wir haben wollen.
In diesem Fall werden automatisch folgende Dateien bearbeitet/erstellt:

- **config/routes** wird durch `/rant RantR GET POST` erweitert
- **Application.hs** importiert unsere neue Route automatisch
- **project.cabal** gibt dieses Modul als „benutzt“ an

Da wir das Model eben schon hinzugefügt haben, fügen wir nun alle Handler hinzu, die wir in unserer Applikation haben wollen, mittels

```
yesod add-handler
Name of route (without trailing R): Rant
Enter route pattern (ex: /entry/#EntryId): /rant
Enter space-separated list of methods (ex: GET POST): GET POST
```

Analog machen wir dies für alle Handler, die wir haben wollen.
In diesem Fall werden automatisch folgende Dateien bearbeitet/erstellt:

- **config/routes** wird durch `/rant RantR GET POST` erweitert
- **Application.hs** importiert unsere neue Route automatisch
- **project.cabal** gibt dieses Modul als „benutzt“ an
- **Handler/Rant.hs** enthält das eigentliche Modul, welches die neue Route handeln soll.

Demo

Für die Generation von Formularen gibt es in `Yesod` zwei Wege:

Für die Generation von Formularen gibt es in `Yesod` zwei Wege:

- Applikativ (d.h. mit `Applicative`)

Für die Generation von Formularen gibt es in `Yesod` zwei Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.

Für die Generation von Formularen gibt es in `Yesod` zwei Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.
- Monadisch (d.h. mit `Monad`)

Für die Generation von Formularen gibt es in `Yesod` zwei Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.
- Monadisch (d.h. mit `Monad`)
  Hier können wir einzelne Felder selektieren, müssen aber seperat das HTML generieren, welches im Browser angezeigt wird

Für die Generation von Formularen gibt es in `Yesod` zwei Wege:

- Applikativ (d.h. mit `Applicative`)
  Hier wird der Code für uns automatisch generiert, aber wir haben kaum Einfluss auf die Gestaltung.
- Monadisch (d.h. mit `Monad`)
  Hier können wir einzelne Felder selektieren, müssen aber seperat das HTML generieren, welches im Browser angezeigt wird

Wir werden vorerst nur die applikative Syntax benutzen. Die monadische Variante ist im Buch aber sehr gut erklärt und ggf. einfach zu adaptieren.

Kommen wir zunächst zu der Syntax für die applikative Schreibweise.

Kommen wir zunächst zu der Syntax für die applikative Schreibweise.
Ein Beispielformular für unseren Rant könnte in etwa so Aussehen:

```
rantForm :: Form Rant
rantForm = renderDivs $ Rant
    <$> areq textField "Titel" Nothing
    <*> areq textAreaField "Rant" (Just "Rant here")
    <*> lift (liftIO getCurrentTime)
```

Kommen wir zunächst zu der Syntax für die applikative Schreibweise.
Ein Beispielformular für unseren Rant könnte in etwa so Aussehen:

```
rantForm :: Form Rant
rantForm = renderDivs $ Rant
    <$> areq textField "Titel" Nothing
    <*> areq textAreaField "Rant" (Just "Rant here")
    <*> lift (liftIO getCurrentTime)
```

Hier generieren wir einen Rant aus

- einem benötigtem einzeiligem TextFeld ohne Default-Wert
- einer benötigten mehrzeiligen TextArea mit Default-Wert „Rant here“
- der aktuellen Uhrzeit (die nicht vom Client, sondern vom Server kommt).

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)
- Konstanten (`pure constant`)

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)
- Konstanten (`pure constant`)
- Daten aus der App-Monade (`lift`)

In der applikativen Notation haben wir vier Möglichkeiten Werte zu bekommen:

- Benötigte Werte vom Client (`areq`)
- Optionale Werte vom Client (`aopt`)
- Konstanten (`pure constant`)
- Daten aus der App-Monade (`lift`)

Natürlich können wir in der App-Monade dann durch `liftIO` beliebige Funktionen ausführen.

Nun haben wir ein Formular definiert. Dieses wird sowohl für das Anzeigen beim Client verwendet, als auch für die Validierung auf dem Server.

Nun haben wir ein Formular definiert. Dieses wird sowohl für das Anzeigen beim Client verwendet, als auch für die Validierung auf dem Server.
Hierzu gibt es die Funktion

```
(widget, enctype) <- generateFormPost rantForm
```

um ein Formular zu generieren.

Nun haben wir ein Formular definiert. Dieses wird sowohl für das Anzeigen beim Client verwendet, als auch für die Validierung auf dem Server.
Hierzu gibt es die Funktion

```
(widget, enctype) <- generateFormPost rantForm
```

um ein Formular zu generieren.

`widget` enthält den HTML-Teil, den wir mittels `defaultLayout` rendern können. `enctype` enthält den Encoding-Type, der im äußeren Formular angegeben werden muss.

Eine fertige Seite würde somit wie folgt aussehen:

```
getRantR :: Handler Html
getRantR = do
  (rantWidget, rantEnctype) <- generateFormPost rantForm
  defaultLayout $ do
    [whamlet|
      <h1>Rant
      <form method=post action=@{RantR} enctype=#{rantEnctype}>
        ^{rantWidget}
        <button>Rant!
    |]
```

Eine fertige Seite würde somit wie folgt aussehen:

```
getRantR :: Handler Html
getRantR = do
  (rantWidget, rantEnctype) <- generateFormPost rantForm
  defaultLayout $ do
    [whamlet|
      <h1>Rant
      <form method=post action=@{RantR} enctype=#{rantEnctype}>
        ^{rantWidget}
        <button>Rant!
    |]
```

Wir sehen hier, dass im Hamlet mittels `^{rantWidget}` das Widget direkt eingebunden werden kann. Wir müssen nur noch das äußere `<form>`-Konstrukt definieren und einen Button zum Absenden hinzufügen.

Eine fertige Seite würde somit wie folgt aussehen:

```
getRantR :: Handler Html
getRantR = do
  (rantWidget, rantEnctype) <- generateFormPost rantForm
  defaultLayout $ do
    [whamlet|
      <h1>Rant
      <form method=post action=@{RantR} enctype=#{rantEnctype}>
        ^{rantWidget}
        <button>Rant!
    |]
```

Wir sehen hier, dass im Hamlet mittels `^{rantWidget}` das Widget direkt eingebunden werden kann. Wir müssen nur noch das äußere `<form>`-Konstrukt definieren und einen Button zum Absenden hinzufügen.
Nach dem Abschicken wird die Route `RantR` aufgerufen, wo wir dann das Ergebnis abholen.

Wenn wir ein Formular auswerten wollen, dann benutzen wir

```
((result,rantWidget), rantEnctype) <- runFormPost registerForm
```

Wenn wir ein Formular auswerten wollen, dann benutzen wir

```
((result,rantWidget), rantEnctype) <- runFormPost registerForm
```

`result` ist hierbei das Ergebnis des Formulars. Falls irgendetwas nicht stimmt, dann bekommen wir gleich auch noch das (teilausgefüllte) Widget und den Enctype zurück um dem User das Formular erneut anzuzeigen.

Wenn wir ein Formular auswerten wollen, dann benutzen wir `((result,rantWidget), rantEnctype) <- runFormPost registerForm`

`result` ist hierbei das Ergebnis des Formulars. Falls irgendetwas nicht stimmt, dann bekommen wir gleich auch noch das (teilausgefüllte) Widget und den Enctype zurück um dem User das Formular erneut anzuzeigen.
Außerdem ist `result` vom Typen `FormResult a`:

```
data FormResult a = FormMissing
                  | FormFailure [Text]
                  | FormSuccess a
```

Wenn wir ein Formular auswerten wollen, dann benutzen wir `((result,rantWidget), rantEnctype) <- runFormPost registerForm`

`result` ist hierbei das Ergebnis des Formulars. Falls irgendetwas nicht stimmt, dann bekommen wir gleich auch noch das (teilausgefüllte) Widget und den Enctype zurück um dem User das Formular erneut anzuzeigen.
Außerdem ist `result` vom Typen `FormResult a`:

```
data FormResult a = FormMissing
                  | FormFailure [Text]
                  | FormSuccess a
```

Für die drei möglichen Fälle:

- Keine Formulardaten vorhanden
- Fehlermeldungen
- Erfolg

Normalerweise macht man ein case über das result:

```
postRantR :: Handler Html
postRantR = do
  ((result,rantWidget), rantEnctype) <- runFormPost rantForm
  let again err = defaultLayout $ do
       [whamlet|
      <h1>Rant
      <h2>Fehler:
      <p>#{err}
      <form method=post action=@{RantR} enctype=#{rantEnctype}>
        ^{rantWidget}
        <button>Rant!
       |]
  case result of
    FormSuccess rant -> do --put into database
                           _ <- runDB $ insert rant
                           getHomeR --and redirect home

    FormFailure (err:_) -> again err
    _ -> again "Invalid input"
```

Wir haben eben schon gesehen, wie man Daten in die Datenbank einfügt: Man wechselt in die Datenbank-Monade und macht `insert object`, wobei über die Typen von `object` klar ist, was nun wo eingefügt werden soll und wir bekommen die ID des eingefügten Objektes wieder.

Wir haben eben schon gesehen, wie man Daten in die Datenbank einfügt: Man wechselt in die Datenbank-Monade und macht `insert object`, wobei über die Typen von `object` klar ist, was nun wo eingefügt werden soll und wir bekommen die ID des eingefügten Objektes wieder.

Aus der Datenbank können wir u.a. Daten abfragen über

Wir haben eben schon gesehen, wie man Daten in die Datenbank einfügt: Man wechselt in die Datenbank-Monade und macht `insert object`, wobei über die Typen von `object` klar ist, was nun wo eingefügt werden soll und wir bekommen die ID des eingefügten Objektes wieder.

Aus der Datenbank können wir u.a. Daten abfragen über

- `selectList`
  Alle Einträge in der Datenbank

Wir haben eben schon gesehen, wie man Daten in die Datenbank einfügt: Man wechselt in die Datenbank-Monade und macht `insert object`, wobei über die Typen von `object` klar ist, was nun wo eingefügt werden soll und wir bekommen die ID des eingefügten Objektes wieder.

Aus der Datenbank können wir u.a. Daten abfragen über

- `selectList`
  Alle Einträge in der Datenbank
- `selectFirst`
  Den ersten Eintrag (falls vorhanden)

Meistens wird `selectList` genommen. Dieses hat die folgende Signatur:

```
selectList :: (MonadIO m, PersistEntity val,
               PersistQuery backend,
               PersistEntityBackend val ~ backend)
           => [Filter val]
           -> [SelectOpt val]
           -> ReaderT backend m [Entity val]
```

Meistens wird `selectList` genommen. Dieses hat die folgende Signatur:

```
selectList :: (MonadIO m, PersistEntity val,
               PersistQuery backend,
               PersistEntityBackend val ~ backend)
           => [Filter val]
           -> [SelectOpt val]
           -> ReaderT backend m [Entity val]
```

Keine Panik!

Interessant für uns sind nur die ersten zwei Parameter. Wir können eine Liste von Filtern und eine Liste von Optionen angeben.

Interessant für uns sind nur die ersten zwei Parameter. Wir können eine Liste von Filtern und eine Liste von Optionen angeben.

Filter sind z.B.:

```
    [ RantTitel <-. "foo"        --title containing foo
    , PersonAge >=. 18 ]         --and person over 18
||. [ PersonIsSingle ==. True ] --or person is single
```

Interessant für uns sind nur die ersten zwei Parameter. Wir können eine Liste von Filtern und eine Liste von Optionen angeben.

Filter sind z.B.:

```
    [ RantTitel <-. "foo"          --title containing foo
    , PersonAge >=. 18 ]           --and person over 18
||. [ PersonIsSingle ==. True ] --or person is single
```

und Optionen:

```
-- only the first 50 rants
-- sorted by erstellt
-- in descending order
[Desc RantErstellt, LimitTo 50]
```

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`).

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`).
Eine `Entity` besteht aus

```
data Entity record = PersistEntity record =>
                     Entity { entityKey :: Key record
                            , entityVal :: record }
```

Wir erhalten damit eine Liste von Ergebnissen (vom Type `Entity`). Eine `Entity` besteht aus

```
data Entity record = PersistEntity record =>
                     Entity { entityKey :: Key record
                            , entityVal :: record }
```

Wir bekommen also die interne Id, die wir z.B. für Updates brauchen und unsere Datenstruktur selbst. Konkret könnte das also so aussehen:

```
getHomeR :: Handler HTML
getHomeR = do
  rants <- rundb $ selectList [] [Desc RantErstellt, LimitTo 50]
  defaultLayout $ [whamlet|
    <h1>last 50 rants
    <ul>
      $forall Entity rid (Rant t i ts) <- rants
        <li>#{t} (postet at: #{ts})<br><br>
            #{i}<br>
            <a href=@{DelRantR rid}>Delete!
|]
```

Demo

Yesod kommt schon mit einer Login und User-Verwaltung. Genutzt wird Standardmässig „Persona“ von Mozilla.

Yesod kommt schon mit einer Login und User-Verwaltung. Genutzt wird Standardmässig „Persona“ von Mozilla.
Alternativen sind z.B.:

- Email
  setzt einen gültigen Email-Server vorraus

Yesod kommt schon mit einer Login und User-Verwaltung. Genutzt wird Standardmässig „Persona“ von Mozilla.
Alternativen sind z.B.:

- Email
  setzt einen gültigen Email-Server vorraus
- Google+

Yesod kommt schon mit einer Login und User-Verwaltung. Genutzt wird Standardmässig „Persona“ von Mozilla.
Alternativen sind z.B.:

- Email
  setzt einen gültigen Email-Server vorraus
- Google+
- OpenId

Yesod kommt schon mit einer Login und User-Verwaltung. Genutzt wird Standardmässig „Persona“ von Mozilla.
Alternativen sind z.B.:

- Email
  setzt einen gültigen Email-Server vorraus
- Google+
- OpenId
- OAuth2
  wird genutzt von z.b. Twitter, Github, Spotify, etc.

Wie bringen wir Yesod nun bei, auf welcher Seite man autorisiert sein muss?

Wie bringen wir Yesod nun bei, auf welcher Seite man autorisiert sein muss?
In Foundation.hs finden wir die generierten Einstellungen:

```
-- Routes not requiring authentication.
isAuthorized (AuthR _) _ = return Authorized
-- Default to Authorized for now.
isAuthorized _ _ = return Authorized
```

Wie bringen wir Yesod nun bei, auf welcher Seite man autorisiert sein muss?
In Foundation.hs finden wir die generierten Einstellungen:

```
-- Routes not requiring authentication.
isAuthorized (AuthR _) _ = return Authorized
-- Default to Authorized for now.
isAuthorized _ _ = return Authorized
```

Der erste Parameter ist hier eine Route, der zweite Paremeter ein Bool, der für den Schreibzugriff steht.

Wie bringen wir Yesod nun bei, auf welcher Seite man autorisiert sein muss?
In Foundation.hs finden wir die generierten Einstellungen:

```
-- Routes not requiring authentication.
isAuthorized (AuthR _) _ = return Authorized
-- Default to Authorized for now.
isAuthorized _ _ = return Authorized
```

Der erste Parameter ist hier eine Route, der zweite Paremeter ein Bool, der für den Schreibzugriff steht.
Wir wollen, dass jeder eingeloggte User ranten und rants entfernen können. Also fügen wir hinzu:

```
isAuthorized (DelRantR _) _ = isUser
isAuthorized RantR _ = isUser
isUser = do
  mu <- maybeAuthId
  return $ case mu of
      Nothing -> AuthenticationRequired
      Just _  -> Authorized
```

Mehr ist nicht nötig um unsere Applikation sicher zu machen. Natürlich sollten wir beim Löschen des Rants prüfen, ob der Rant auch wirklich jemandem gehört, der das darf (z.B. der Ersteller oder der Admin).