λ IHP: Integrated Haskell Platform

Productive, Type-Safe and Fun!

IHP is a modern batteries-included Web Framework, built on top of Haskell and Nix.

Get Started Now

Grab a coffee and build your first haskell app in 20 minutes :)

Hi there! ✋

It's time we talk a bit about modern web development.

Back in the early days of the web, when server-side rendering was still a thing, javascript was sprinkled here and there to

make a few parts of your web app *just a little bit* smoother.

Today it's different: The whole rendering logic and data management has moved into the browser. Instead of delivering readable HTML, the server is just there to send the javascript over-the-wire, which then needs to make a few api calls before displaying anything useful.

This comes with lots of complexity, slow applications, frustated developers, and bugs. It leads us to the ironic situation that javascript, an interpreted language, actually taking more time being compiled in webpack for local development than e.g. haskell, which is supposed to be slower because it's far more high-level.

Somehow we have forgotten that the browser is actually pretty good at displaying HTML.

Luckily, in 2020 we are rediscovering the power of server side rendering.

With this technology shift it's time for a new, modern take on serverside web application development.

With IHP we did that. It's our new take on this. A new, modern framework to show how productive and stable serverside rendering can be.

We poured together the best ideas from frontend tooling and mixed them with strong type safety, a functional programing approach, and the best package manager.

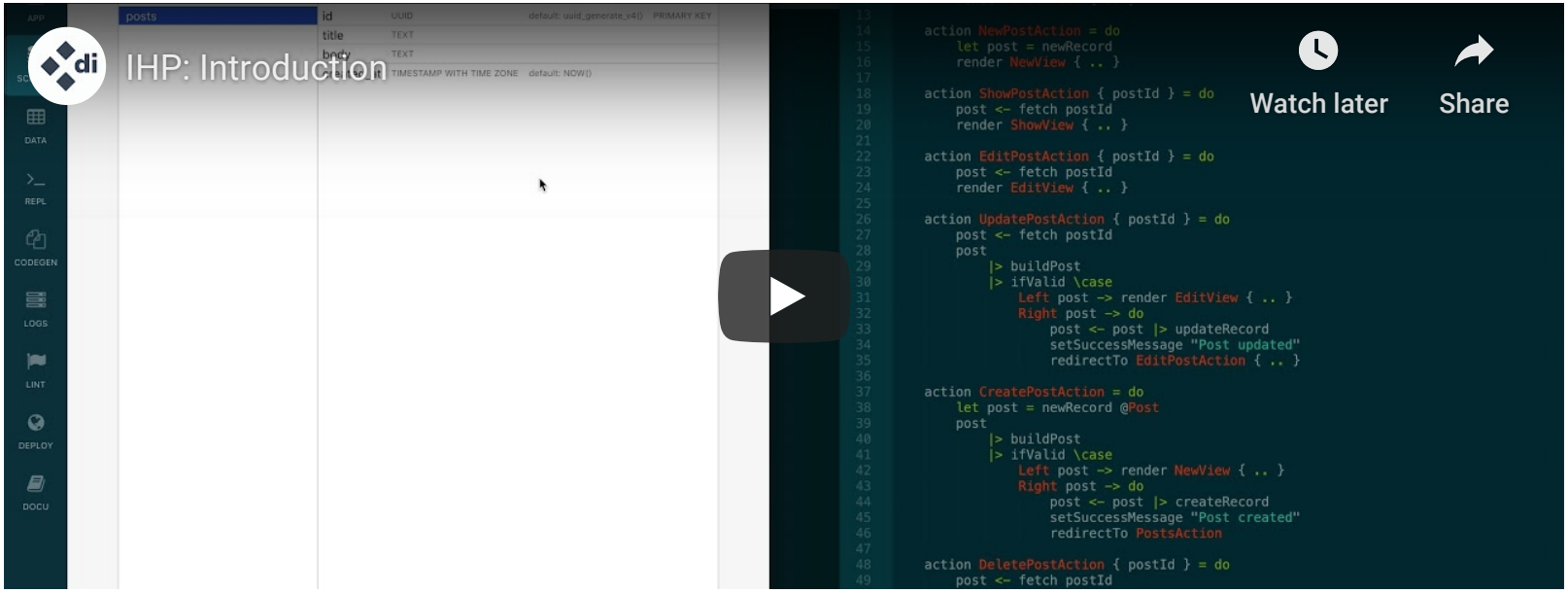
IHP is the result of two thoughts: highest quality software is built on solid foundations and the future of web applications is going to be strongly focused on server side rendering.

We're happy to give you an early look at it. While it's still in beta, IHP is already used in production by digitally induced and several of our friends and partners :-)

Let us know what you think on twitter, on GitHub, or via mail at hi@digitallyinduced.com*.

* We can also send you some IHP stickers. Just ask :)

Watch the Demo-Video:



Stephen Diehl

@smdiehl

This is *very* impressive work. A whole new Haskell web framework based on Warp server, with a non-TH ORM and type-safe templating framework. 🚀

10:22 AM · Jun 23, 2020 · Twitter Web App

33 Retweets **132** Likes

Will Ricketts
@willricketts

The more I look at IHP, the more I think this is Haskell's

killer framework. Really excellent work.

3:03 PM · Aug 27, 2020 · Twitter for Android

1 Retweet **4** Likes

λames
@ratherforky

lHP is ridiculously good. I hope it makes Haskell a bigger part of the web dev industry because wow

11:17 PM · Sep 9, 2020 · Twitter for Android

1 Quote Tweet **4** Likes

Lars Lillo Ulvestad
@larsparsfromage

Day 4 of #100DaysOfHaskell / #100DaysOfCode:

Haskell is a weirdo at times, but lots of fun. Made some more controllers in #IHP. I think it could become the killer framework Haskell has been lacking. Will see if I can figure out many-to-many by tomorrow.

6:58 PM · Aug 23, 2020 · Twitter Web App

8 Retweets **9** Likes

Yeong Sheng (永胜)
@yeongsheng

If you've enjoyed the safety and pleasureable development experience on developing in #elmlang, and enjoyed the power, speed and of server-side rendered front-end with #phoenix_liveview, here comes #ihp (Integrated Haskell Platform) @digitallyinduce

5:49 AM · Jul 28, 2020 · Twitter Web App

1 Retweet **5** Likes

Luc Tielen
@luctielen

Replying to @smdiehl

This looks really good! The ORM gives me some Ecto/Elixir vibes 😄

11:42 AM · Jun 23, 2020 · Twitter Web App

2 Likes

Soham Chowdhury
@evertedsphere

Replying to @smdiehl

tbh what I'm most excited about/impressed by is the schema/controller/model boilerplate handling and the admin interface in general

1:59 PM · Jun 23, 2020 · Twitter for Android

2 Likes

hendi 26 hours ago | link

IHP is supposed to become the Django/Rails/Phoenix of Haskell.

I've been using Django professionally for since 2013, but have started using IHP a couple of weeks ago. It's still quite early but with surprisingly few rough edges, i.e. the developer ergonomics are much better than I expected. It has great documentation that is improving rapidly (as opposed to many other Haskell libraries, which provide little more than API docs or even just the typed function definitions) and offers a refreshing take on database management and migrations.

Some of its killer features:

- HSX, a JSX-like template language that looks like HTML while providing type safety
- Auto live reloading without the need to setup anything
- Documentation with examples: it lets you query the database without learning about monads
- it defines |> for you ;-)
- type-safe, composable SQL queries:

```
projects <- query @Project
  |> filterWhere (#userId, userId)
  |> filterWhere (#deleted, False)
  |> orderBy #createdAt
  |> fetch
```

Martin
@martinmcwhorter

I think IHP might be the killer framework for Haskell. Hashell on Rails.

11:53 AM · Jul 17, 2020 from Dun Laoghaire-Rathdown, Ireland · Twitter for Android

1 Retweet **3** Likes

Daniel Vianna
@dmlvianna

Granted I did not build a SPA this afternoon. But using @nixpkg I have a functioning database-webserver-client USING HASKELL.

It is like suddenly all the effort learning Nix, Haskell and Web finally clicked into something that's not a promise, but REAL, NOW.

2:09 PM · Jun 28, 2020 · Twitter Web App

1 Retweet **2** Likes

Matt Parsons 🏳️‍🌈
@mattoflambda

It's Rails. It's Haskell on Rails.

You thought Yesod was Rails? Nope. Yesod is a highly modular library for developing web applications with a few opinions and some scaffolds.

IHP is Rails - all the components are glued together in a way that makes modularity difficult.

6:43 PM · Jun 23, 2020 · Twitter Web App

2 Retweets **19** Likes

Εκάτη
@TechnoEmpress

Welcome to 2020, web dev #haskell world!

12:05 PM · Jun 23, 2020 · Twitter Web App

2 Retweets **11** Likes

revskill 20 hours ago | parent | favorite | on: Show HN: IHP, a batteries-included web framework b...

So as i see, this is like a Rails framework with static type checking. Best of both worlds !

I'd love to see a guide on integrating with webpack or any frontend tech in the stack.

Avi Press
@avi_press

This is a really impressive project, super excited to try it. All this time I've been doing Haskell web development I've been wishing for features like an admin UI and all the other things rails/django have had for a long time

7:20 PM · Jun 25, 2020 · Twitter Web App

2 Retweets **8** Likes

Lars Lillo Ulvestad
@larsparsfromage

Day 7 of #100DaysOfHaskell / #100DaysOfCode: Managed to make the registration view for my authentication on #IHP. Starting to get the hang of the basics of this framework. Some things are actually much simpler to do in #Haskell/IHP compared to Elixir/Phoenix.

9:56 PM · Aug 26, 2020 · Twitter Web App

11 Retweets **9** Likes

IHP: Integrated Haskell Platform

What makes it different?

Productive & Fun:

IHP comes with everything you need to build great web applications out of the box. Combined with the unique mix of technologies and a fast development process, IHP makes it very pleasant to build applications.

Type-safe and reliable:

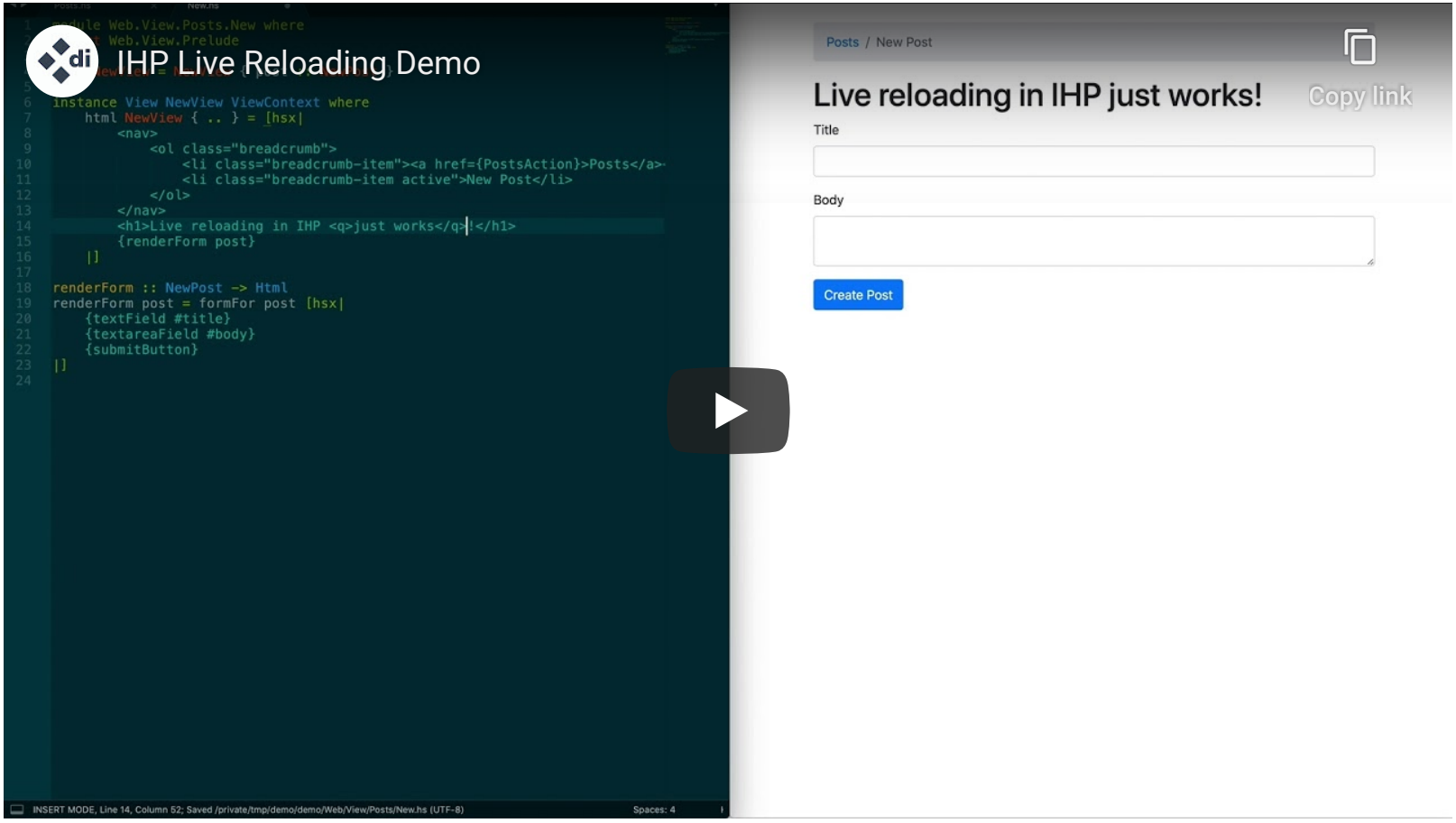
With Haskell and Nix we use the most reliable technologies available, to make sure your application will never crash because of Null Pointer Exceptions.

If you like TypeScript, you will love IHP.

Fast dev enviroment:

While haskell is a compiled language, the built-in dev server automatically reloads your code changes using the fastest way possible. Changes are reflected instantly. Just like good old PHP.

Watch it in action:

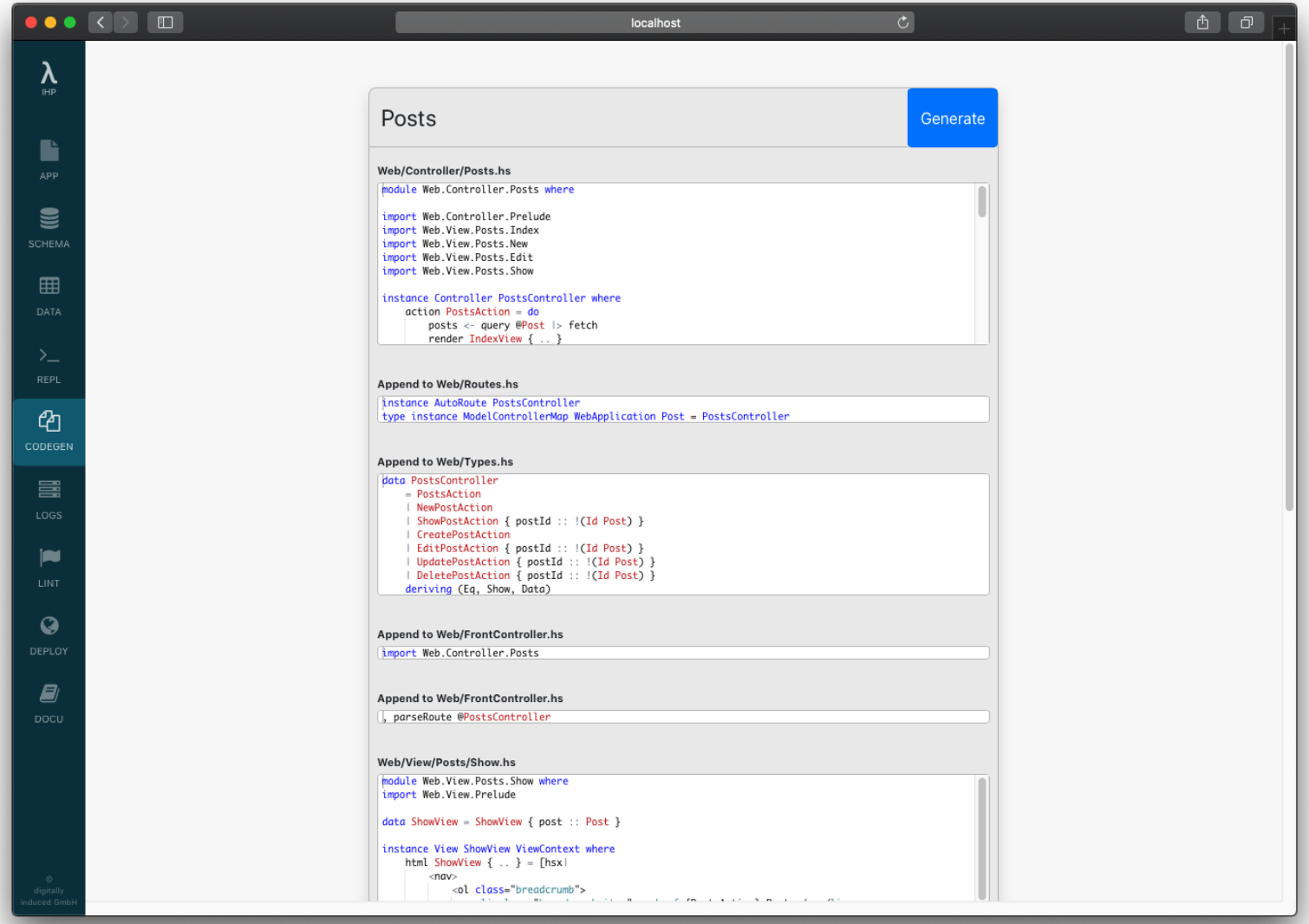


Accessible:

Setup of the fully-managed dev environment takes just 5 minutes. All dependencies (even database and compiler) are managed using the nix package manager. This means dependency problems just cannot occur anymore. Also everything is guaranteed to be same for all developers in your team.

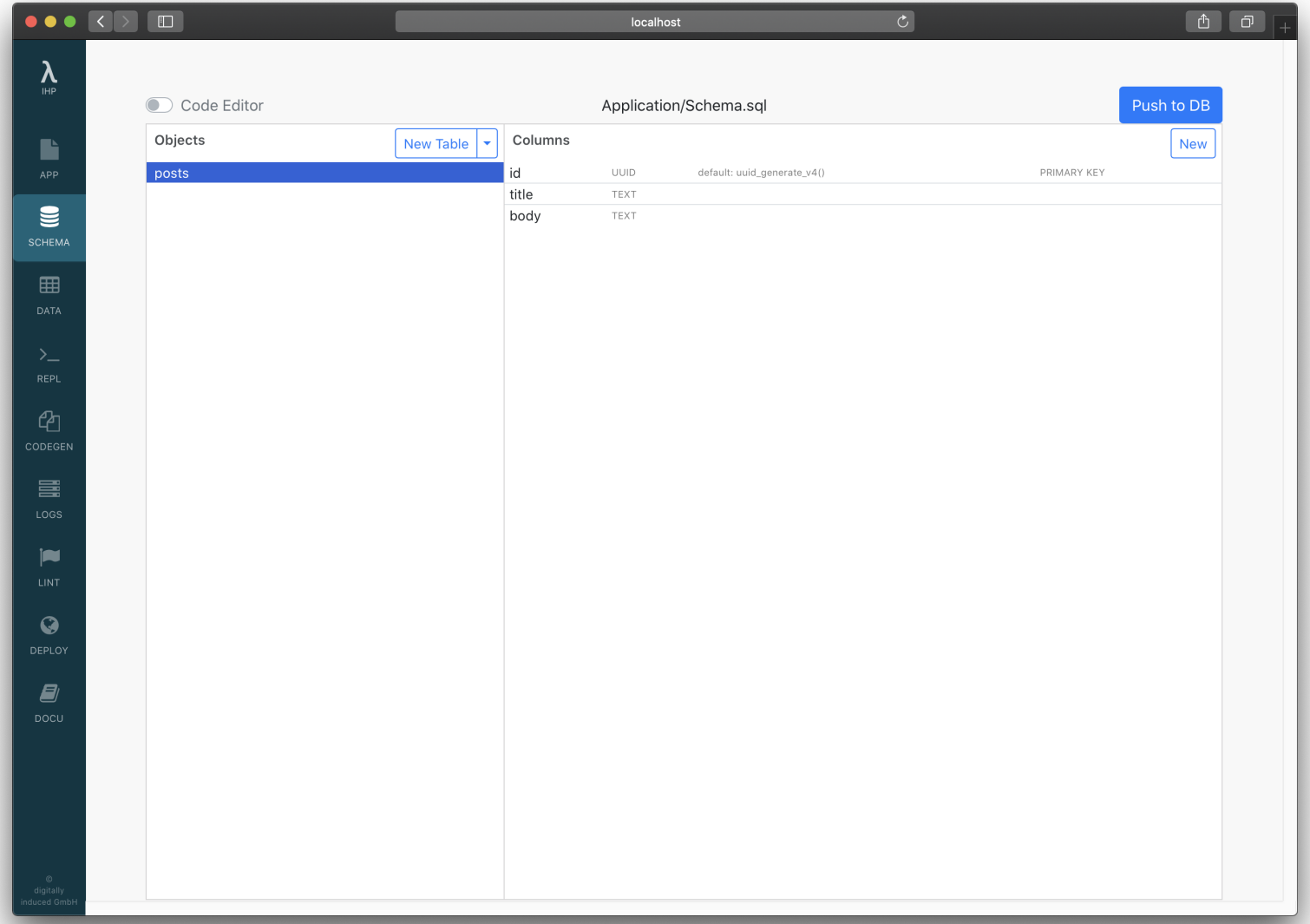
No Haskell Experience required:

Code Generators will help you to quickly build things even when you have no professional haskell experience yet. Pick up haskell by building real world applications.



Integrated Dev Tooling:

To speed up your development process, IHP comes with a full set of web based dev tools.
Including: a database schema designer, a web-based code generator, a web-based repl, ...



Major Operating Systems Supported:

Windows (via Linux Subsystem), macOS, NixOS, Debian, Ubuntu

HSX

Like React's JSX. Write html code in your haskell files. This will be transformed to actual type-checked haskell code at compile time.

```
instance View IndexView ViewContext where
    html IndexView { .. } = [hsx|
        <h1>Nutzer einladen</h1>

        <p>
            Hier kannst du neue Nutzer zu deinem Team <strong>{get #name company}</strong> einladen.
        </p>

        <div>
            {renderForm newInvite}
        </div>

        <table class="table table-responsive">
            <thead>
                <tr>
                    <th>Email</th>
                    <th>Datum</th>
                    <th>Status</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>{forEach invites renderInvite}</tbody>
        </table>
    |]
```

Longterm Roadmap

Lot's of frameworks are already gone a year after launch. Especially in the fast moving JS world. But don't worry about IHP. We have been using it at digitally induced since 2017. It's actively used by us and our friends and partners. Even without external contributors we will build new features and do periodic maintaince releases in the future. We have big plans for IHP and as a profitable and independent software company we have the ability to actually execute them over the longterm.

IHP is professionally used by companies such as:

Build your first haskell app in 20 minutes

Get Started with IHP Now

IHP Casts

Stay in the loop

Newsletter

To get notified about the latest updates, subscribe to the IHP newsletter.

You can also follow digitally induced on Twitter.

GitHub

Leave a star at the GitHub repo: digitallyinduced/ihp

Questions, or need help with haskell type errors? Join us at Gitter (IRC Bridge available) or on the IHP Forum

Want to see some demo code? We got you!

Build your first IHP app now!

Blog

21.08.2020: v21082020 released

10.08.2020: IHP Live Reloading: How it works

07.08.2020: v07082020 released 🎉

24.07.2020: v24072020 released 🎉

22.07.2020: Why IHP is Using The Nix Package Manager

13.07.2020: “Haskell Web Framework IHP Aims to Make Web Development Type-Safe and Easy” on InfoQ.com

10.07.2020: v10072020 released 🎉

23.06.2020: IHP has been released to the public

Getting Started With Integrated Haskell Platform

ihp.digitallyinduced.com/Guide/index.html

IHP is a full stack framework focused on rapid application development while striving for robust code quality.

We believe that functional programing is the future of software development and want to make functional programing with Haskell and nix available to anyone. We try to offer a solution which can be used by developers who have not worked with Haskell yet. IHP comes with everything you need to build great web applications with Haskell and nix. We have made a lot of pragmatic decision to get you started faster. This way you can just pick up Haskell along the way :-)

Feature Overview

- **Fully managed dev environment:** Works on any machine because all dependencies (including PostgreSQL) are managed using the nix package manager. You only need to know a single command to start the app.
- **Auto live reload using virtual dom in dev mode:** Each code change automatically triggers the web page to refresh. The refresh uses a diff based patch to avoid resetting the page state. This means: Changes are reflected instantly.
- **Build robust, type-safe applications:** With the power of Haskell your applications are going to be a lot more robust. Pretty much no runtime errors in production. You can refactor with confidence.
- **Fast dev enviroment:** While we use a compiled language, the built-in dev server automatically reloads your code changes using the fastest way possible. Usually changes are reflected in less than 50ms (alot faster than your average webpack setup).
- **Faster production environment:** Production response times are around 30ms. With help of instant click it is faster than most single page applications.
- **Integrated dev tooling:** You only need a text editor, everything else is taken care of.
- **Scalable Application Architecture:** IHP is the result of building lots of real world applications with Haskell.

Professional Use

Before open sourcing, IHP has already been used in production by digitally induced since 2017. Therefore you can expect continuous support and development in the future.

Additionally IHP is used by companies such as Prokapi and CodeKarussell (rausgegangen.de).

Example Project

Take a look at the example blog application project to get to see some code. Follow the documentation to build this application yourself in the section “Your First Project”.

Next: Installing IHP

Installing IHP

ihp.digitallyinduced.com/Guide/installation.html

1. Dependency: Nix Package Manager

The framework uses the nix package manager to manage the whole set of dependencies of your application

For example, postgresql and the Haskell compiler are both dependencies of your app, as well as all the Haskell or javascript packages you want to use. We use nix to make sure that these dependencies are available to the app - in development, as well as in production.

That's why we first need to make sure that you have nix installed.

▼

Mac

Run this command in your terminal to install nix on your machine.

```
sh <(curl -L https://nixos.org/nix/install) --darwin-use-unencrypted-nix-store-volume
```

After this restart your terminal.

If you get an error like `error: refusing to create Nix store volume because the boot volume is FileVault encrypted, but encryption-at-rest is not available.` , follow the steps described in this GitHub Issue. We're working on improving this step.

▼

Linux

Before installing nix and IHP, you need `curl` and `git` if you don't have them already. If you are unsure, run this:

```
sudo apt update
sudo apt upgrade
sudo apt install git curl make -y
```

For NixOS Users: If you're on NixOS, of course you don't need to install nix anymore :-) Just skip this step.

Install nix by running the following command in your shell and follow the instructions on the screen:

```
curl -L https://nixos.org/nix/install | sh
```

Due to Linux not loading the `.profile` file, nix will not be loaded. To fix that, we need to add this line to the rc file of your shell (usually `.bashrc`). Open it, and add this line

```
. ~/.nix-profile/etc/profile.d/nix.sh
```

There are also other ways to install nix, take a look at the documentation.

▼

Windows

Running nix on Windows requires the Windows Subsystem for Linux, which first needs manual activation via **Powershell with Administrator Privileges**:

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
```

Enabling this Feature needs a restart.

To download a Linux Distribution, open the Microsoft Store and search for Ubuntu or Debian. We recommend Ubuntu, since it works best with nix on Windows.

Note: You **do not** need a Microsoft account to download. You can simply cancel or close the login forms and the download will continue.

With the Distro downloaded, run it and update it using your package manager. In Ubuntu you would use:

```
sudo apt update
sudo apt upgrade
sudo apt install git curl make -y
```

WSL will add your Windows System Paths in your Linux Subsystem. These tend to generate errors due to spaces and brackets in folder names. Also, due to Linux not loading the `.profile`, we need to add the nix.sh manually. To fix these two issues, just add these lines to the end of your `.bashrc`

```
PATH=$(/usr/bin/printenv PATH | /usr/bin/perl -ne 'print join(":", grep { !/\/mnt\/[a-z]/ } split(/:/));')
. ~/.nix-profile/etc/profile.d/nix.sh
```

Now, create a folder for nix:

```
sudo mkdir -p /etc/nix
```

To make nix usable on Windows, we need to create and add the following lines to the file `/etc/nix/nix.conf`:

```
sandbox = false
use-sqlite-wal = false
```

After saving the file, you can now install nix:

```
curl -L https://nixos.org/nix/install | sh
```

When the installation finishes successfuly, you will be prompted to either reload your environment with the given command, or restart your shell.

If in doubt, just close and reopen Ubuntu/Your Distro.

NOTES FOR WINDOWS USERS:

Windows Firewall

When using Windows, you will be asked if tasks like ghc-iserv or rundevserver should be allowed by the firewall. This is needed to access the devserver interface and the webapp itself.

Installing nix for IHP was done using this guide.

2. Installing IHP

You can now install IHP by running:

```
nix-env -f https://ihp.digitallyinduced.com/ihp-new.tar.gz -i ihp-new
```

If you don't already use cachix, you will be prompted to install it. You don't need it, but it is highly recommended, as it reduces build time dramatically. Learn more about cachix here.

NixOS specific

If you get the following error during this step on NixOS:

```
MustBeRoot "Run command as root OR execute: $ echo \"trusted-users = root $USER\" | sudo tee -a /etc/nix/nix.conf && sudo pkill nix-daemon"
```

You have to add this to your NixOS configuration:

```
nix.trustedUsers = [ "root" "USERNAME_HERE" ];
```

See the documentation for nix.trustedUsers to learn more about what this is doing.

3. Next

It's time to start your first "hello world" project now :)

Next: Your First Project

Creating Your First Project

1. Project Setup

This guide will lead you through creating a small blog application. To set up the project, open a terminal and type:

```
$ ihp-new blog
```

The first time you set up IHP, this command might take 10 - 15 minutes to install. Any further projects after that will be a lot faster because all the packages are already cached on your computer. While the build is running, take a look at "What Is Nix" by Shopify to get a general understanding on how Nix works.

In case some errors appear now or in later steps, check out the troubleshooting section to get a quick solution. You can also join our awesome gitter community and ask a question there. We're happy to help!

The new `blog` directory now contains a couple of auto-generated files and directories that make up your app.

Here is a short overview of the whole structure:

File or Directory	Purpose
Config/	
Config/Config.hs	Configuration for the framework and your application
Config/nix/nixpkgs-config.nix	Configuration for the Nix package manager
Config/nix/haskell-packages/	Custom Haskell dependencies can be placed here
Application/	Your domain logic lives here
Application/Schema.sql	Models and database tables are defined here
Web/Controller	Web application controllers
Web/View/	Web application html template files

File or Directory	Purpose
Web/Types.hs	Central place for all web application types
static/	Images, css and javascript files
.ghci	Default config file for the Haskell interpreter
.gitignore	List of files to be ignored by git
App.cabal, Setup.hs	Config for the cabal package manager (TODO: maybe move to Config/App.cabal)
default.nix	Declares your app dependencies (like package.json or composer.json)
Makefile	Default config file for the make build system

2. Hello, World!

You now already have a working Haskell app ready to be started.

Switch to the `blog` directory before doing the next steps:

```
$ cd blog
```

Start the development server by running the following in the `blog` directory:

```
$ ./start
```

Your application is starting now. The development server will automatically launch the built-in IDE. The server can be stopped by pressing CTRL+C.

By default, your app is available at `http://localhost:8000` and your development tooling is at `http://localhost:8001`. The dev server automatically picks other ports when they are already in use by some other server. For example it would pick `http://localhost:8001` and `http://localhost:8002` if port 8000 is used.

In the background, the built-in development server starts a PostgreSQL database connected to your application. Don't worry about manually setting up the database. It also runs a websocket server to power live reloads on file saves inside your app.

3. Data Structures & PostgreSQL

Schema Modeling

For our blog project, let's first build a way to manage posts.

For working with posts, we first need to create a `posts` table inside our database. A single post has a title, a body and of course an id. IHP is using UUIDs instead of the typical numerical ids.

This is how our `posts` table might look like for our blog:

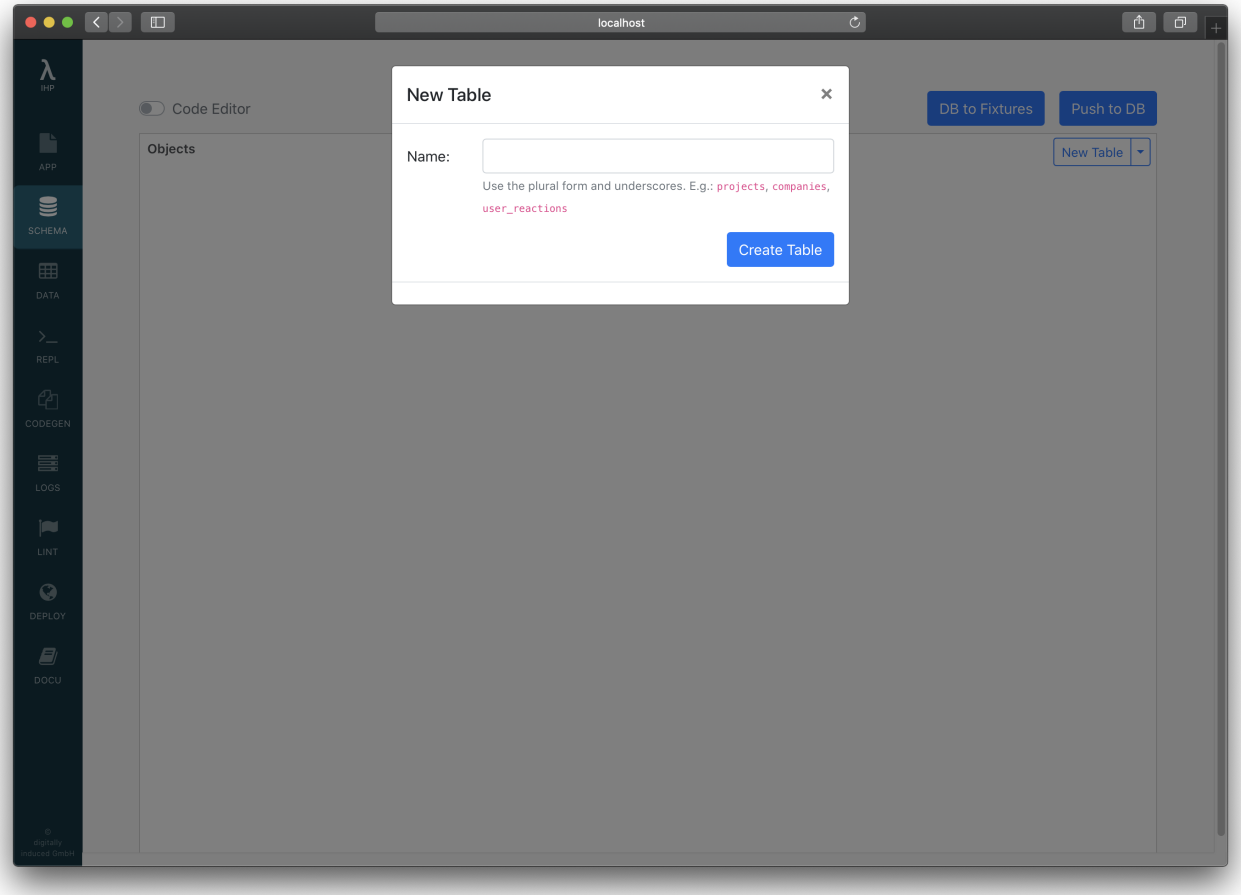
id :: UUID	title :: Text	body :: Text

id :: UUID	title :: Text	body :: Text
8d040c2d-0199-4695-ac13-c301970cff1d	My Experience With Nix On OS X	Some time ago, I've moved this jekyll-based blog …
b739bc7c-5ed1-43f4-944c-385aea80f182	Deploying Private GitHub Repositories To NixOS Servers	In a previous post I've already shared a way to deploy private git repositories to a NixOS based server. While …

To work with posts in our application, we now have to define this data schema.

For the curious: IHP has a built-in GUI-based schema designer. The schema designer will be used in the following sections of this tutorial. The schema designer helps to quickly build the DDL statements for your database schema without remembering all the Postgresql syntax and data types. But keep in mind: The schema designer is just a GUI tool to edit the `Application/Schema.sql` file. This file consist of DDL statements to build your database schema. The schema designer parses the `Application/Schema.sql`, applies changes to the syntax tree and then writes it back into the `Application/Schema.sql`. If you love your VIM, you can always skip the GUI and go straight to the code at `Application/Schema.sql`. If you need to do something advanced which is not supported by the GUI, just manually do it with your code editor of choice. IHP is built by terminal hackers, so don't worry, all operations can always be done from the terminal :-)

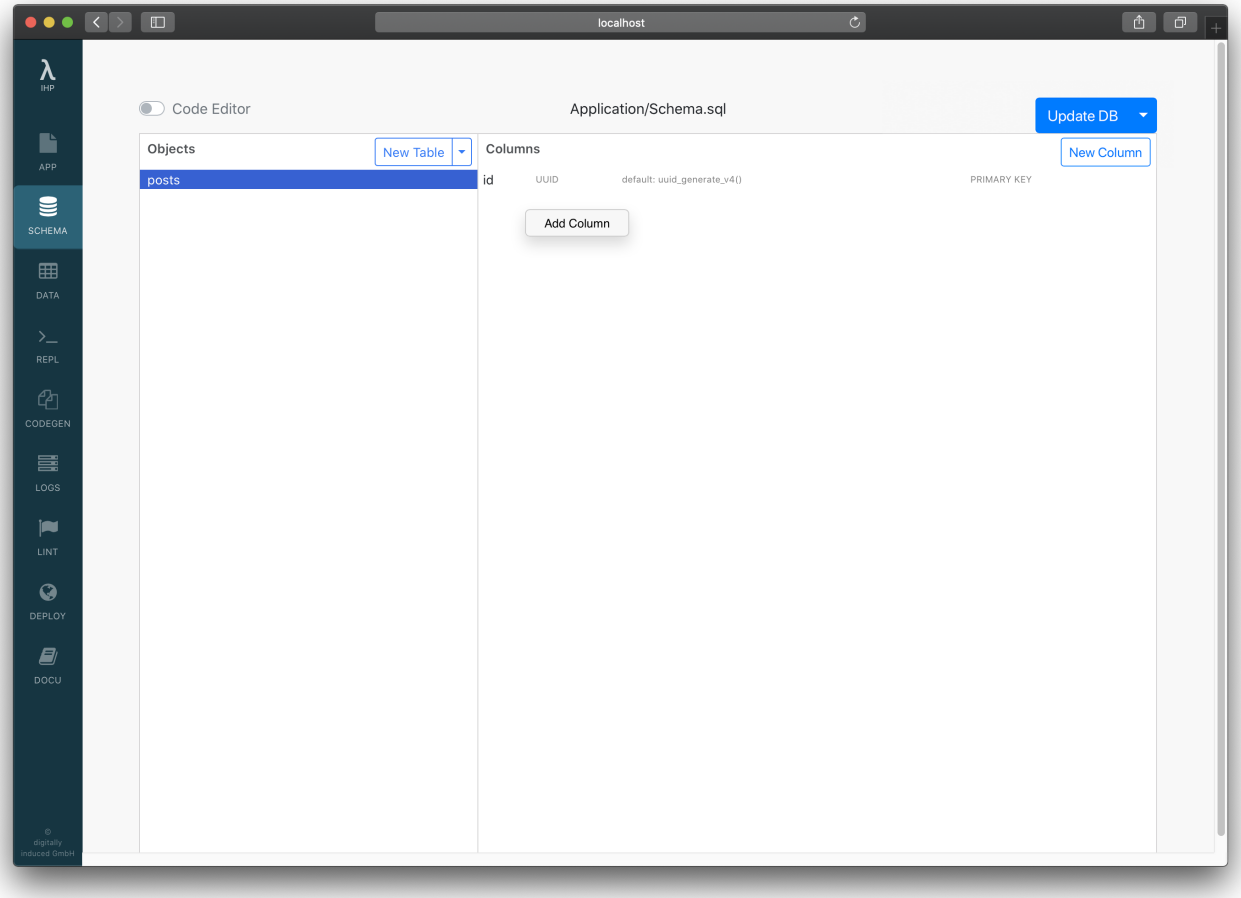
Open the IHP Schema Designer and add a new table with `title` and `body` as text columns. To do this click on the `New` button in the table view.



Enter the table name `posts` and click on `Create Table`.

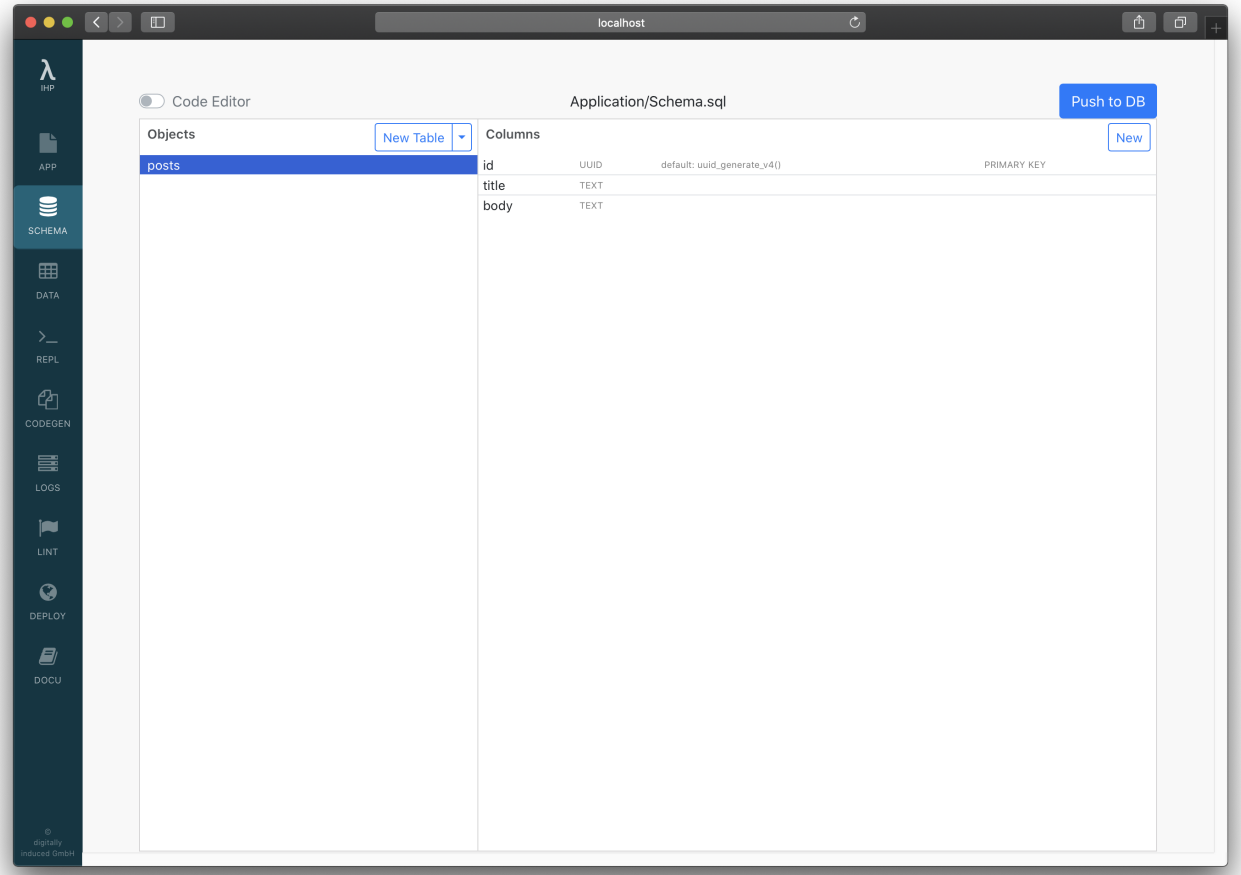
In the right pane, you can see the columns of the newly created table. The id column has been automatically created for us.

Right click into the `Columns` pane and select `Add Column`:



Use this dialog to create the title and body columns.

After that, your schema should look like this:



Loading the Schema

Next we need to make sure that our database schema with our `posts` table is imported into the local postgresql database. Don't worry, the local development postgresql server is already running. The dev server has conveniently already started it.

Open the `Application/Schema.sql` in your code editor to see the SQL queries which make up the database schema:

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE posts (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    title TEXT NOT NULL,
    body TEXT NOT NULL
);
```

To load the table into our local postgres server, we need to click `Update DB` in the Schema Designer (use `make db` from the command line).

The `posts` table has been created now. Let's quickly connect to our database and see that everything is correct:

```
$ make psql

psql (9.6.12)
Type "help" for help.

-- Let's do a query to check that the table is there

app=# SELECT * FROM posts;

 id | title | body
----+-------+------
(0 rows)
```

```
-- Looks alright! :)



-- We can quit the postgresql console by typing \q

app=# \q
```

Now our database is ready to be consumed by our app.

Record Types

By specificing the above schema, the framework automatically provides several types for us. Here is a short overview:

Type	**`Post`**
Definition	`data Post = Post { id :: Id Post, title :: Text, body :: Text }`
Description	A single record from the `posts` table
Example	`Post { id = cfefdc6c-c097-414c-91c0-cbc9a79fbe4f, title = "Hello World", body = "Some body text" } :: Post`

Type	**`Id Post`**
Definition	`newtype Id Post = Id UUID`

Type	`Id Post`
Description	Type for the `id` field
Example	`"5a8d1be2-33e3-4d3f-9812-b16576fe6a38" :: Id Post`

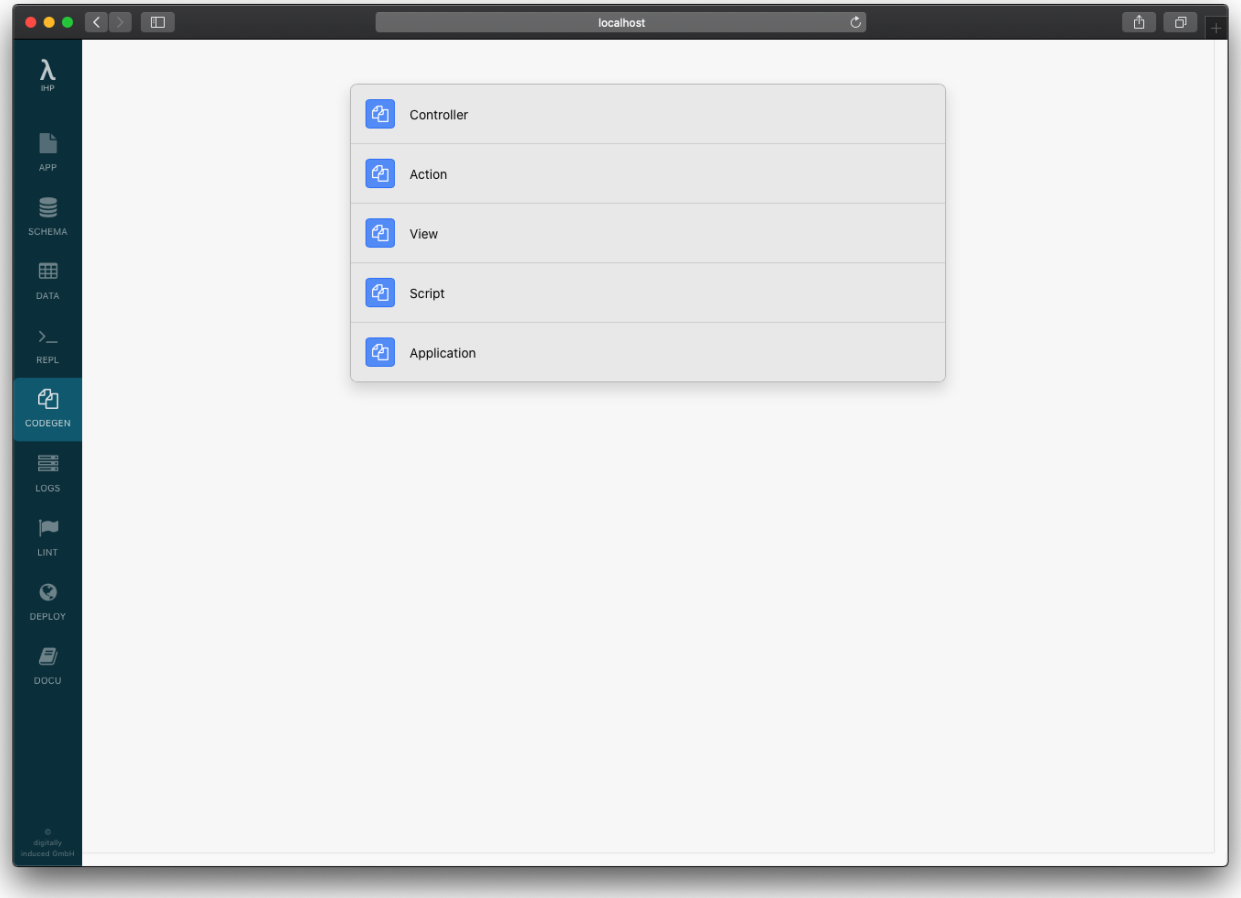
For the curious: To dig deeper into the generated code, open the Schema Designer, right-click a table and click `Show Generated Haskell Code` or look into `build/Generated/Types.hs`.

4. Apps, Controllers, Views

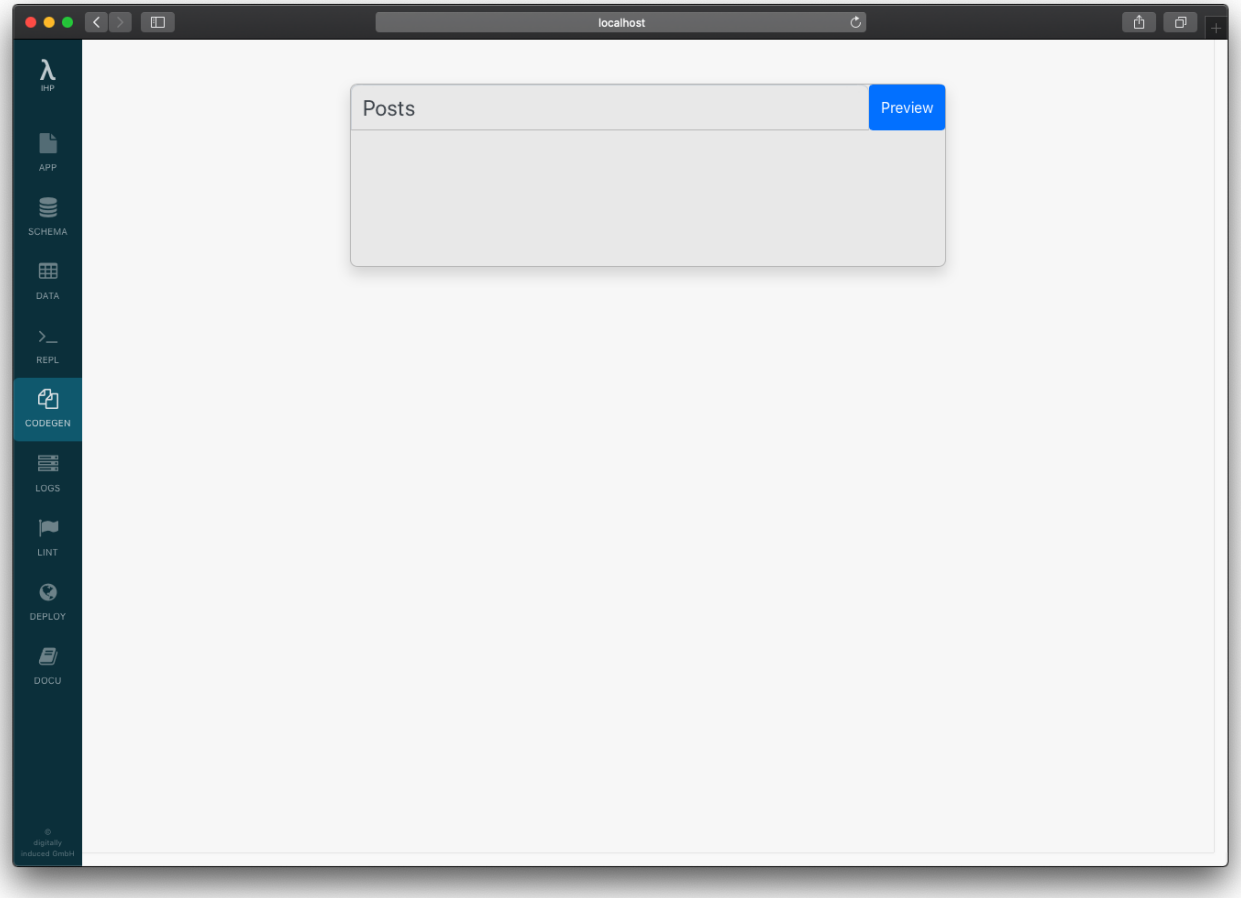
IHP follows the well-known MVC (Model-View-Controller) structure. Controllers and actions are used to deal with incoming requests.

A controller belongs to an application. The default application is called `Web` (that's why all controller and views are located there). Your whole project can consist of multiple sub-applications. Typically your production app will need e.g. an admin backend application next to the default web application.

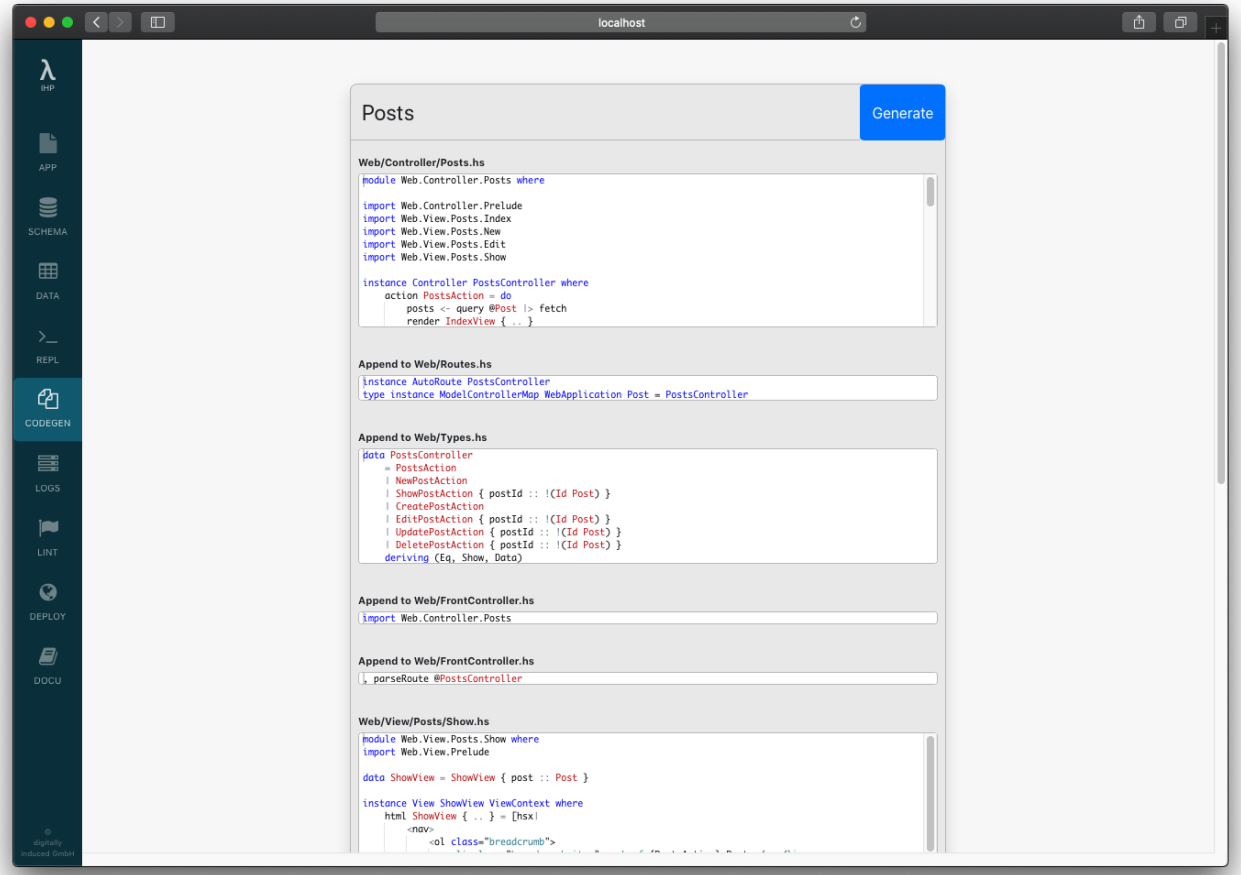
We can use the built-in code generators to generate a controller for our posts. Inside the dev server, click on `CODEGEN` to open the Code Generator. There you can see everything that can be generated. Click on `Controller`:



You need to enter the controller name. Enter `Posts` and click `Preview`:

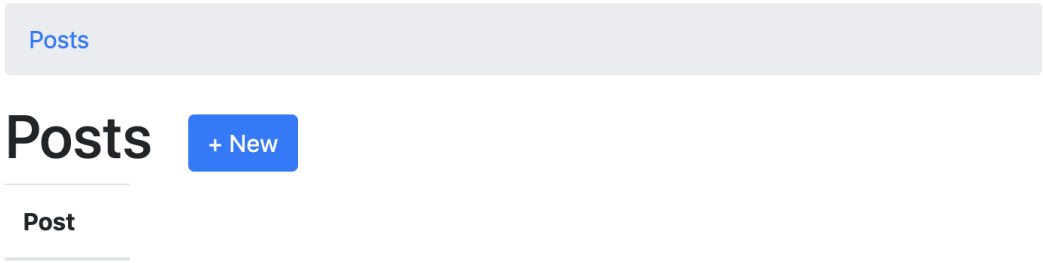


The preview will show you all the files which are going to be created or modified. Take a look and when you are ready, click on `Generate`.



After the files have been created as in the preview, your controller is ready to be used. Open your browser at http://localhost:8000/Posts to try out the new controller. The generator did all the initial work we need to get our usual CRUD (Created, Read, Update, Delete) actions going.

Here's how the new `/Posts` page looks like:



Next we're going to dig a bit deeper into all the changes made by the controller generator.

New Types

Let's first take a closer look at the changes in `Web/Types.hs`. Here a new data structure was created:

```
data PostsController
    = PostsAction
    | NewPostAction
    | ShowPostAction { postId :: !(Id Post) }
    | CreatePostAction
```

```
    | EditPostAction { postId :: !(Id Post) }
    | UpdatePostAction { postId :: !(Id Post) }
    | DeletePostAction { postId :: !(Id Post) }
    deriving (Eq, Show, Data)
```

We have one constructor for each possible action. Here you can see a short description for all the constructors:

Action	Request	Description
`PostsAction`	`GET /Posts`	Lists all posts
`NewPostAction`	`GET /NewPost`	Displays the form to create a post
`ShowPostAction { postId = someId }`	`GET /ShowPost?postId={someId}`	Shows the posts with id $someId
`CreatePostAction`	`POST /CreatePost`	Endpoint to create a Post
`EditPostAction { postId = someId }`	`GET /EditPost?postId={someId}`	Displays the form to edit a post
`UpdatePostAction { postId = someId }`	`POST /UpdatePost?postId={someId}`	Endpoint to submit a post update
`DeletePostAction { postId = someId }`	`DELETE /DeletePost?postId={someId}`	Deletes the post

A request like “Show me the post with id `e57cfb85-ad55-4d5c-b3b6-3affed9c662c`“ can be represented like `ShowPostAction { postId = e57cfb85-ad55-4d5c-b3b6-3affed9c662c`

`}`. Basically, the IHP router always maps an HTTP request to such an action data type. (By the way: The type `Id Post` is just a UUID, but wrapped within a newtype, `newtype Id model = Id UUID`).

Controller Implementation: `Web/Controller/Posts.hs`

The actual code that is run, when an action is executed, is defined in `Web/Controller/Posts.hs`. Let's take a look, step by step.

Imports

```
module Web.Controller.Posts where

import Web.Controller.Prelude
import Web.View.Posts.Index
import Web.View.Posts.New
import Web.View.Posts.Edit
import Web.View.Posts.Show
```

In the header we just see some imports. Controllers always import a special `Web.Controller.Prelude` module. It provides e.g. controller helpers and also the framework specific functions we will see below. The controller also imports all its views. Views are also just "normal" Haskell modules.

Controller Instance

```
instance Controller PostsController where
```

The controller logic is specified by implementing an instance of the `Controller` type-class.

Index Action

This is where the interesting part begins. As we will see below, the controller implementation is just an `action` function, pattern mattching over our `data PostsController` structure we defined in `Web/Types.hs`.

```
    action PostsAction = do
        posts <- query @Post |> fetch
        render IndexView { .. }
```

This is the index action. It's called when opening `/Posts`. First it fetches all the posts from the database and then passes them along to the view. The `IndexView { .. }` is just shorthand for `IndexView { posts = posts }`.

New Action

```
action NewPostAction = do
    let post = newRecord
    render NewView { .. }
```

This is our endpoint for `/NewPost`. It just creates an empty new post and then passes it to the `NewView`. The `newRecord` is giving us an empty `Post` model. It's equivalent to manually writing `Post { id = Default, title = "", body = "" }`.

Show Action

```
action ShowPostAction { postId } = do
    post <- fetch postId
    render ShowView { .. }
```

This is our show action at `/ShowPost?postId=postId`. Here we pattern match on the `postId` field of `ShowPostAction` to get the post id of the given request. Then we just call `fetch` on that `postId` which gives us the specific `Post` record. Finally we just pass that post to the view.

Edit Action

```
    action EditPostAction { postId } = do
        post <- fetch postId
        render EditView { .. }
```

Our `/EditPost?postId=postId` action. It's pretty much the same as in the `action ShowPostAction`, just with a different view.

Update Action

```
    action UpdatePostAction { postId } = do
        post <- fetch postId
        post
            |> buildPost
            |> ifValid \case
                Left post -> render EditView { .. }
                Right post -> do
                    post <- post |> updateRecord
                    setSuccessMessage "Post updated"
                    redirectTo EditPostAction { .. }
```

This action deals with update requests for a specific post. As usual we pattern match on the `postId` and fetch it.

The interesting part is `buildPost`. It is a helper function defined later in the controller: `buildPost = fill @["title", "body"]`. The `fill` call inside `buildPost` reads the `title`

and `body` attributes from the browser request and fills them into the `post` record. The `buildPost` is also the place for validation logic.

`ifValid` returns `Either Post Post`. `Left post` means that e.g. the `title` or `body` did not pass validation. `Right post` means that all parameters could be set on `post` without any errors.

In the error case (`Left post ->`) we just re-render the `EditView`. The `EditView` then tells the user about validation errors.

In the success case (`Right post ->`) we save the updated post to the database (with `updateRecord`). Then we set a success message and redirect the user back to the edit view.

Create Action

```
action CreatePostAction = do
    let post = newRecord @Post
    post
        |> buildPost
        |> ifValid \case
            Left post -> render NewView { .. }
            Right post -> do
                post <- post |> createRecord
                setSuccessMessage "Post created"
                redirectTo PostsAction
```

Our create action, dealing with `POST /CreatePost` requests.

It's pretty much like the update action. When the validation succeeded, it saves the record to the database using `createRecord`.

Delete Action

```
    action DeletePostAction { postId } = do
        post <- fetch postId
        deleteRecord post
        setSuccessMessage "Post deleted"
        redirectTo PostsAction
```

The last action is dealing with `DELETE /DeletePost?postId=postId` requests. It's pretty much like the other actions, we just call `deleteRecord` here.

Routes

The router is configured in `Web/Routes.hs`. The generator just places a single line there:

```
instance AutoRoute PostsController
```

This empty instance magically sets up the routing for all the actions. Later you will learn how you can customize the urls according to your needs (e.g. "beautiful urls" for SEO).

Views

We should also quickly take a look at our views.

Let's first look at the show view in `Web/View/Posts/Show.hs`:

```
module Web.View.Posts.Show where
import Web.View.Prelude

data ShowView = ShowView { post :: Post }

instance View ShowView ViewContext where
    html ShowView { .. } = [hsx|
        <nav>
            <ol class="breadcrumb">
                <li class="breadcrumb-item"><a href={PostsAction}>Posts</a>
                <li class="breadcrumb-item active">Show Post</li>
            </ol>
        </nav>
        <h1>Show Post</h1>
    |]
```

We can see that the `ShowView` is just a data definition. There is also a `View ShowView` instance. The HTML-like syntax inside the `html` function is `hsx` code. It's similar to React's JSX. You can write HTML code as usual there. Everything inside the `[hsx|...|]` block is also type-checked and converted to Haskell code at compile-time.

Now that we have a rough overview of all the parts belonging to our `Post`, it's time to do some coding ourselves.

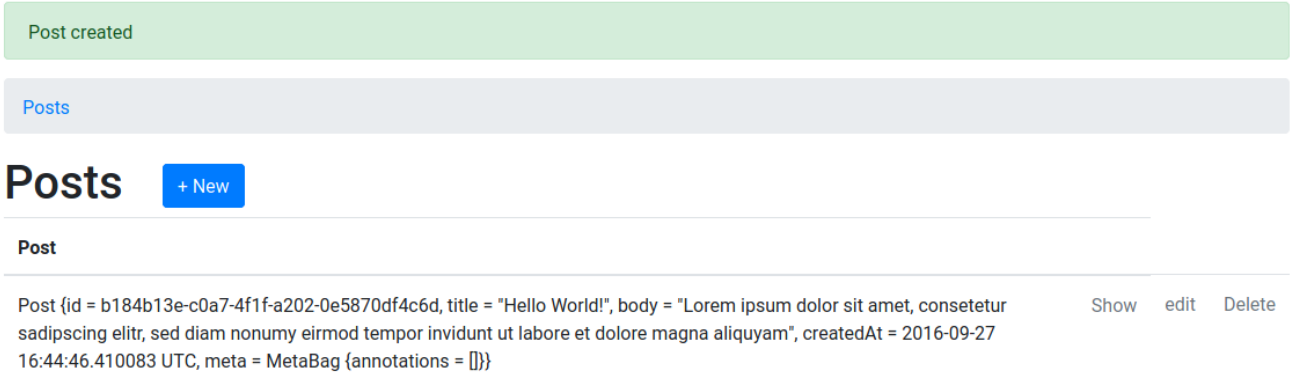
5. Extending the Blog

The generated controller already feels close to a super simple blog. Now it's time to make it more beautiful.

Creating a Post

First we quickly need to create a new blog post. Open http://localhost:8000/Posts and click on `+ New`. Then enter `Hello World!` into the "Title" field and `Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam` into the "Body".

Click `Save Post`. You should now see the new post listed on the `index` view.



Displaying a Post

Let's first improve the `show` view. Right now the headline is "Show Post", and the actual post body is never shown.

Open the `Web/View/Posts/Show.hs` and replace `<h1>Show Post</h1>` with `<h1>{get #title post}</h1>`. Also add a `<div>{get #body post}</div>` below the `<h1>`.

The `Web/View/Posts/Show.hs` file should look like this:

```
module Web.View.Posts.Show where
import Web.View.Prelude

data ShowView = ShowView { post :: Post }

instance View ShowView ViewContext where
    html ShowView { .. } = [hsx|
        <nav>
            <ol class="breadcrumb">
                <li class="breadcrumb-item"><a href={PostsAction}>Posts</a>
                <li class="breadcrumb-item active">Show Post</li>
            </ol>
        </nav>
        <h1>{get #title post}</h1>
        <div>{get #body post}</div>
    |]
```

After you saved the changes, you should see that the changes have been reflected in the browser already. In the background the page has been refreshed automatically. This refresh is using a diff based approach by using morphdom.

Display all posts

After creating your post, you should have already seen that posts list is right now displaying all the post fields. Let's change it to only display the post's title.

Open the `Web/View/Posts/Index.hs` and replace `<td>{post}</td>` with `<td>{get #title post}</td>`.

Let's also make it clickable by wrapping it in a link. We can just put a `<a href={ShowPostAction (get #id post)}>` around it. The line should now look like:

```
<td><a href={ShowPostAction (get #id post)}>{get #title post}</a></td>
```

Now we can also remove the "Show" link. We can do that by removing the next line `<td><a href={ShowPostAction (get #id post)}>Show</a></td>`.

Adding Validation

Let's make sure that every post has at least a title. Validations can be defined inside our controller `Web/Controller/Posts.hs`.

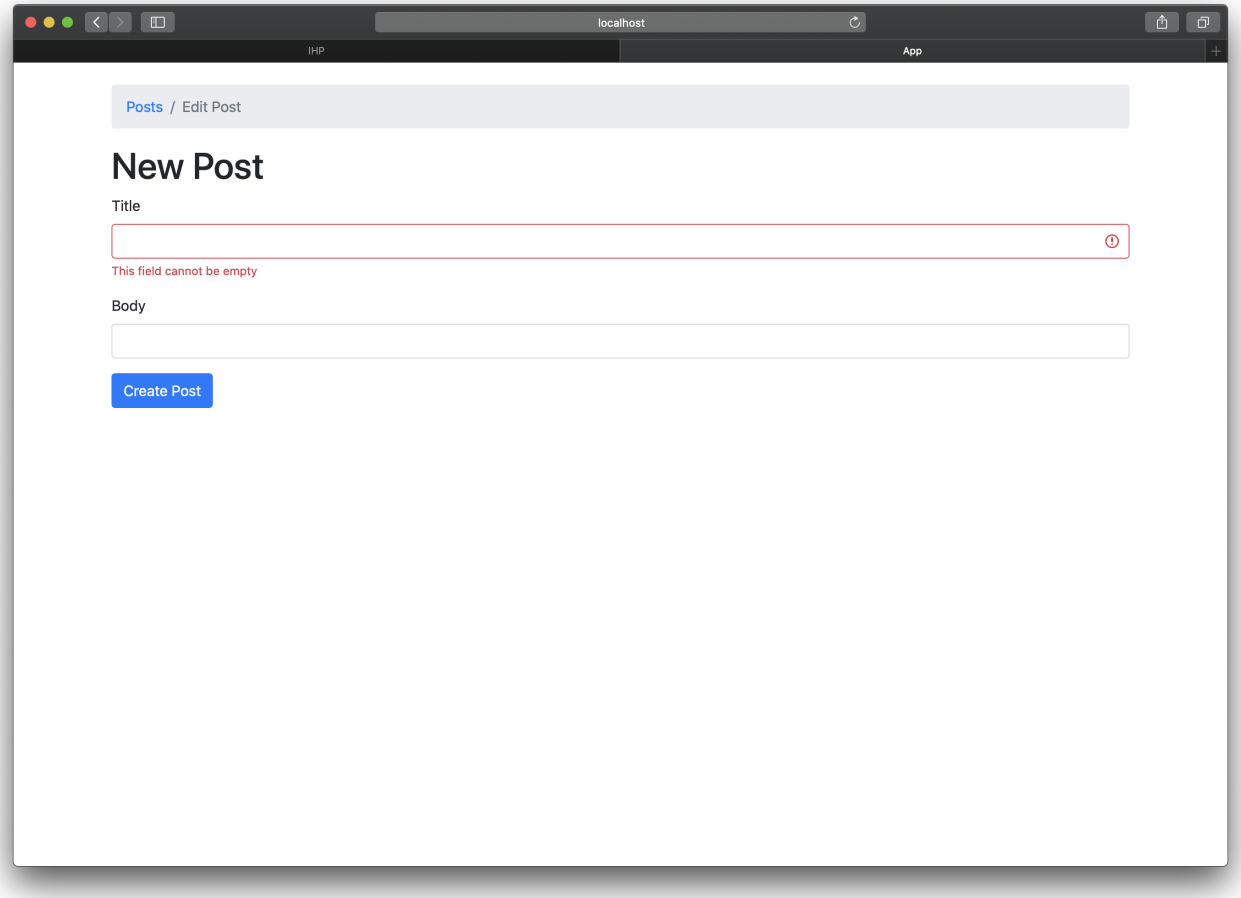
Right now at the bottom of the file we have this:

```
buildPost post = post
    |> fill @["title","body"]
```

Replace the implementation with this:

```
buildPost post = post
    |> fill @["title","body"]
    |> validateField #title nonEmpty
```

Now open http://localhost:8000/NewPost and click `Save Post` without filling the text fields. You will get a “This field cannot be empty” error message next to the empty title field.



You can find a list of all available validator functions in the API Documentation.

Timestamps

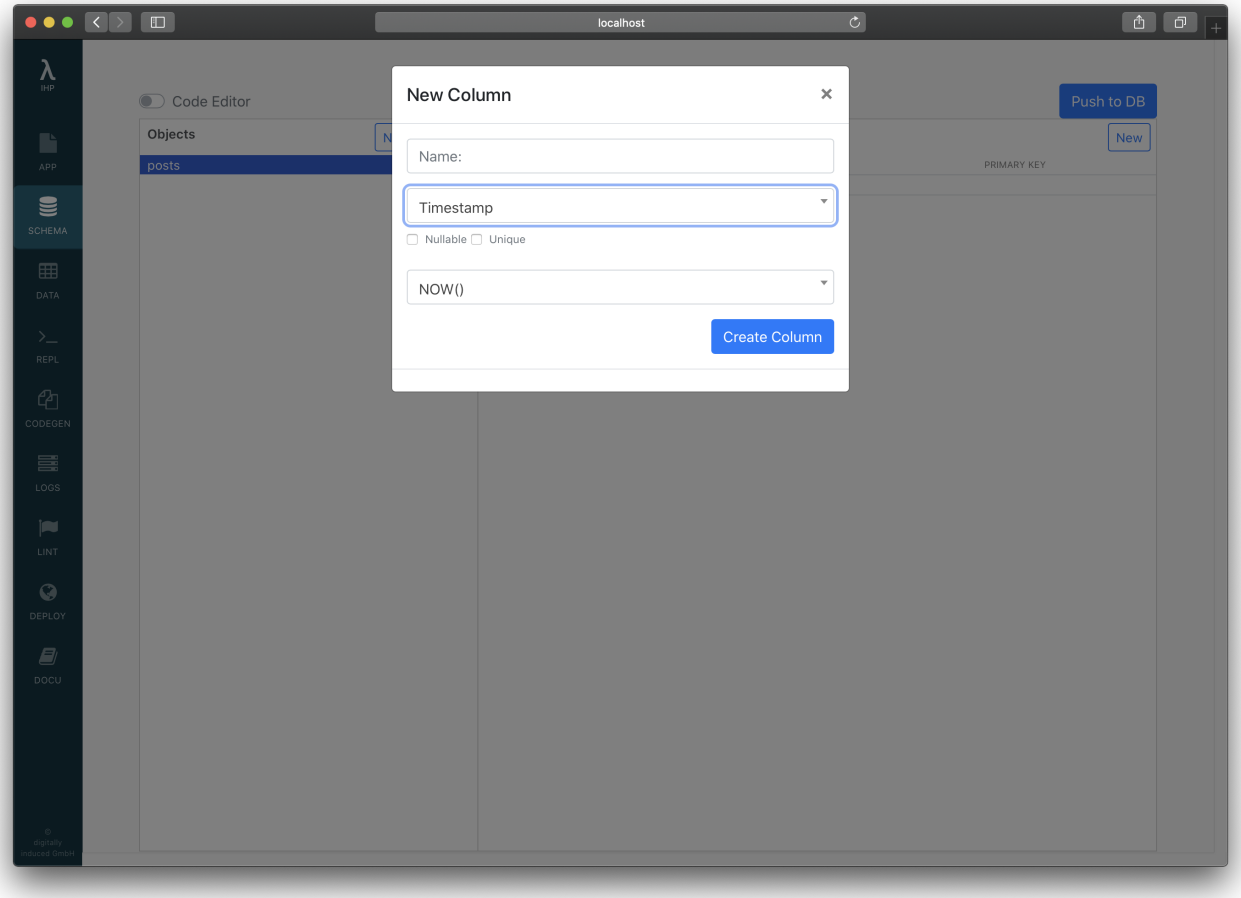
It would be nice to always show the latest post on the index view. Let's add a timestamp to do exactly that.

Take a look at `Application/Fixtures.sql`. The file should look like this:

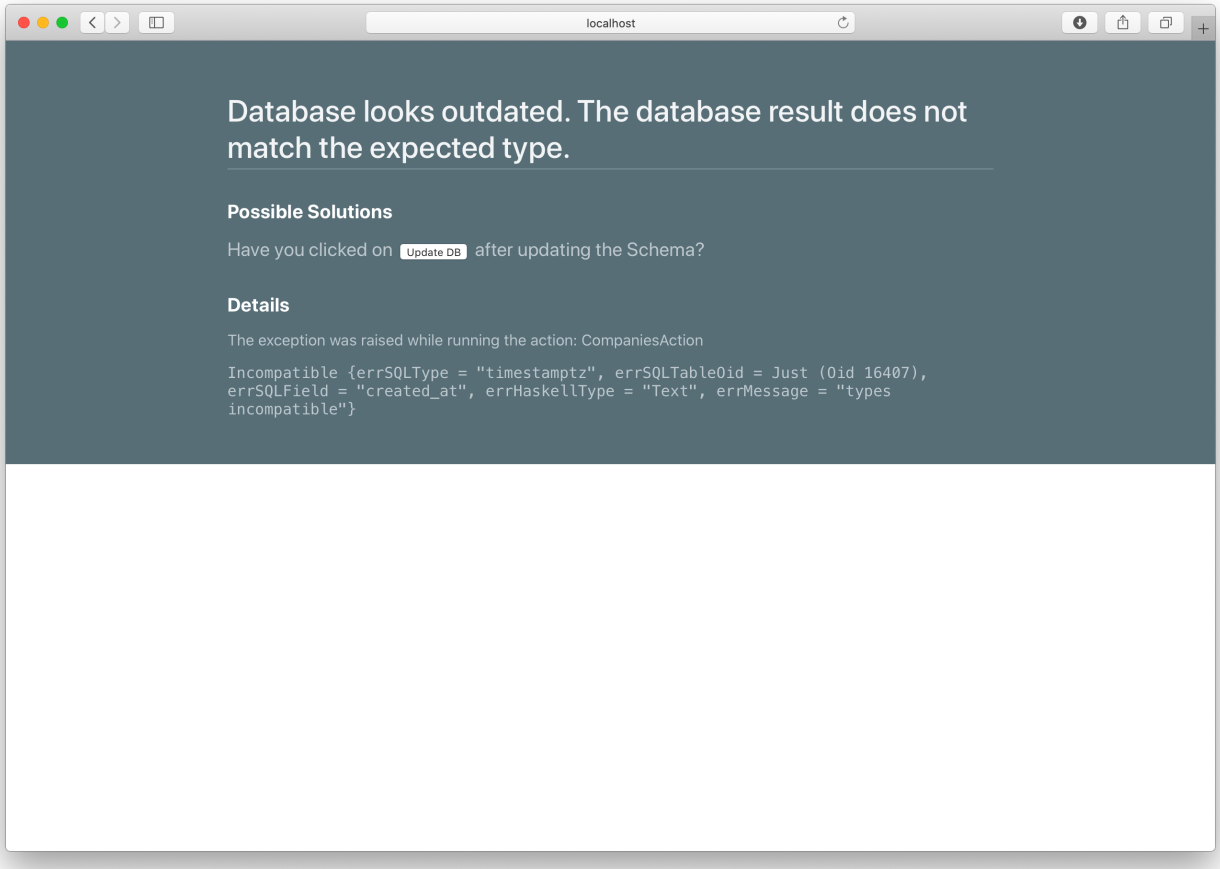
```
-- .......
-- .......

INSERT INTO public.posts VALUES ('fcbd2232-cdc2-4d0c-9312-1fd94448d90a', '
```

All our existing posts are saved here. You should also commit this file to git to share your fixtures with your team mates. We will need these saved fixtures in a moment, when we want to update the database schema.

Let's add a new `created_at` column. Open http://localhost:8001/Tables, enter `created_at` and select Timestamp for the type. Also set the default value to `NOW()`.



Now open the `/Posts` again inside your browser. You will see this error:



This happens because we only added the `created_at` column to the `Application/Schema.sql` file by using the Schema Designer. But the actual running Postgres server still uses the older database schema.

To update the local database, open the Schema Designer and click the `Update DB` button. This button will save the current database content to the fixtures, destroy the database, reload the schema and then insert the fixtures.

In general the workflow for making database schema changes locally is: Make changes to the `Schema.sql` and update Database with `Update DB`.

You can open http://localhost:8000/Posts again. The error is gone now.

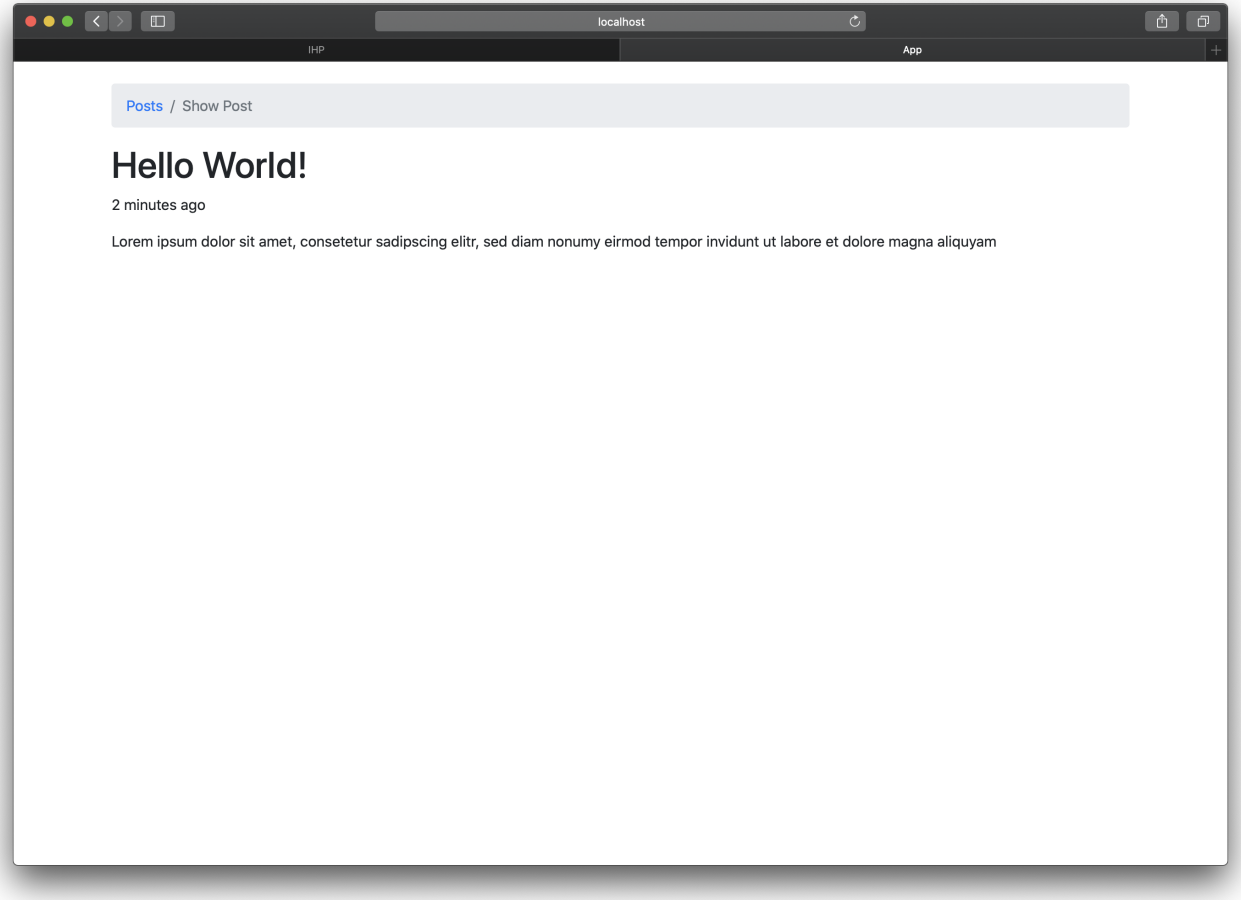
Now we can order the posts by our new `created_at` field. Open `Web/Controller/Posts.hs` and add `orderByDesc #createdAt` like this inside the `action PostsAction`:

```
action PostsAction = do
    posts <- query @Post
        |> orderByDesc #createdAt
        |> fetch
    render IndexView { .. }
```

Let's also show the creation time in the `ShowView` in `Web/View/Posts/Show.hs`. There we add `<p>{get #createdAt post |> timeAgo}</p>` below the title:

```
<nav>
    <ol class="breadcrumb">
        <li class="breadcrumb-item"><a href={PostsAction}>Posts</a></li>
        <li class="breadcrumb-item active">Show Post</li>
    </ol>
</nav>
<h1>{get #title post}</h1>
<p>{get #createdAt post |> timeAgo}</p>
<div>{get #body post}</div>
```

Open the view to check that it's working. If everything is fine, you will see something like `5 minutes ago` below the title. The `timeAgo` helper uses a bit of JavaScript to automatically display the given timestamp in the current time zone and in a relative format. In case you want to show the absolute time (like `10.6.2019, 15:58`), just use `dateTime` instead of `timeAgo`.



Markdown

Right now our posts can only be plain text. Let's make it more powerful by adding support for Markdown.

Adding a Markdown Library

To deal with Markdown, instead of implementing our own Markdown parser, let's just use an existing package. There's the excellent `mmark` package we can use.

To install this package, open the `default.nix` file and append `mmark` to the `haskellDeps` list. The file will now look like this:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/ihp.git";
        rev = "0d2924bcd4cde09e9f219f5e7eca888ad473094a";
    };
    haskellEnv = import "${ihp}/NixSupport/default.nix" {
        ihp = ihp;
        haskellDeps = p: with p; [
            cabal-install
            base
            wai
            text
            hlint
            p.ihp
            mmark # <--------- OUR NEW PACKAGE ADDED HERE
        ];
```

```
        otherDeps = p: with p; [
            # Native dependencies, e.g. imagemagick
        ];
        projectPath = ./.;
    };
in
    haskellEnv
```

Stop the development server by pressing CTRL+C. Then update the local development environment by running `make -B .envrc`. This will download and install the mmark package. Now restart the development server by typing `./start` again.

Markdown Rendering

Now that we have `mmark` installed, we need to integrate it into our `ShowView`. First we need to import it: add the following line to the top of `Web/View/Posts/Show.hs`:

```
import qualified Text.MMark as MMark
```

Next change `{get #body post}` to `{get #body post |> renderMarkdown}`. This pipes the body field through a function `renderMarkdown`. Of course we also have to define the function now.

Add the following to the bottom of the show view:

```
renderMarkdown text = text
```

This function now does nothing except return its input text. Our Markdown package provides two functions, `MMark.parse` and `MMark.render` to deal with the Markdown. Let's first deal with parsing:

```
renderMarkdown text = text |> MMark.parse ""
```

The empty string we pass to `MMark.parse` is usually the file name of the `.markdown` file. As we don't have any Markdown file, we just pass an empty string.

Now open the web app and take a look at a blog post. You will see something like this:

```
Right MMark {..}
```

This is the parsed representation of the Markdown. Of course that's not very helpful. We also have to connect it with `MMark.render` to get html code for our Markdown. Replace the `renderMarkdown` with the following code:

```
renderMarkdown text =
    case text |> MMark.parse "" of
        Left error -> "Something went wrong"
        Right markdown -> MMark.render markdown |> tshow |> preEscapedToHtm
```

The `show` view will now show real formatted text, as we would have expected.

Forms & Validation

Let's also quickly update our form. Right now we have a one-line text field there. We can replace it with a text area to support multi line text.

Open `Web/View/Posts/Edit.hs` and change `{textField #body}` to `{textareaField #body}`. We can also add a short hint that the text area supports Markdown: Replace `{textareaField #body}` with `{(textareaField #body) { helpText = "You can use Markdown here"} }`.

```
renderForm :: Post -> Html
renderForm post = formFor post [hsx|
    {textField #title}
    {(textareaField #body) { helpText = "You can use Markdown here"} }
    {submitButton}
|]
```

After that, do the same in `Web/View/Posts/New.hs`.

We can also add an error message when the user tries to save invalid Markdown. We can quickly write a custom validator for that:

Open `Web/Controller/Posts.hs` and import `MMark` at the top:

```
import qualified Text.MMark as MMark
```

```

```

Then add this custom validator to the bottom of the file:

```
isMarkdown :: Text -> ValidatorResult
isMarkdown text =
    case MMark.parse "" text of
        Left _ -> Failure "Please provide valid Markdown"
        Right _ -> Success
```

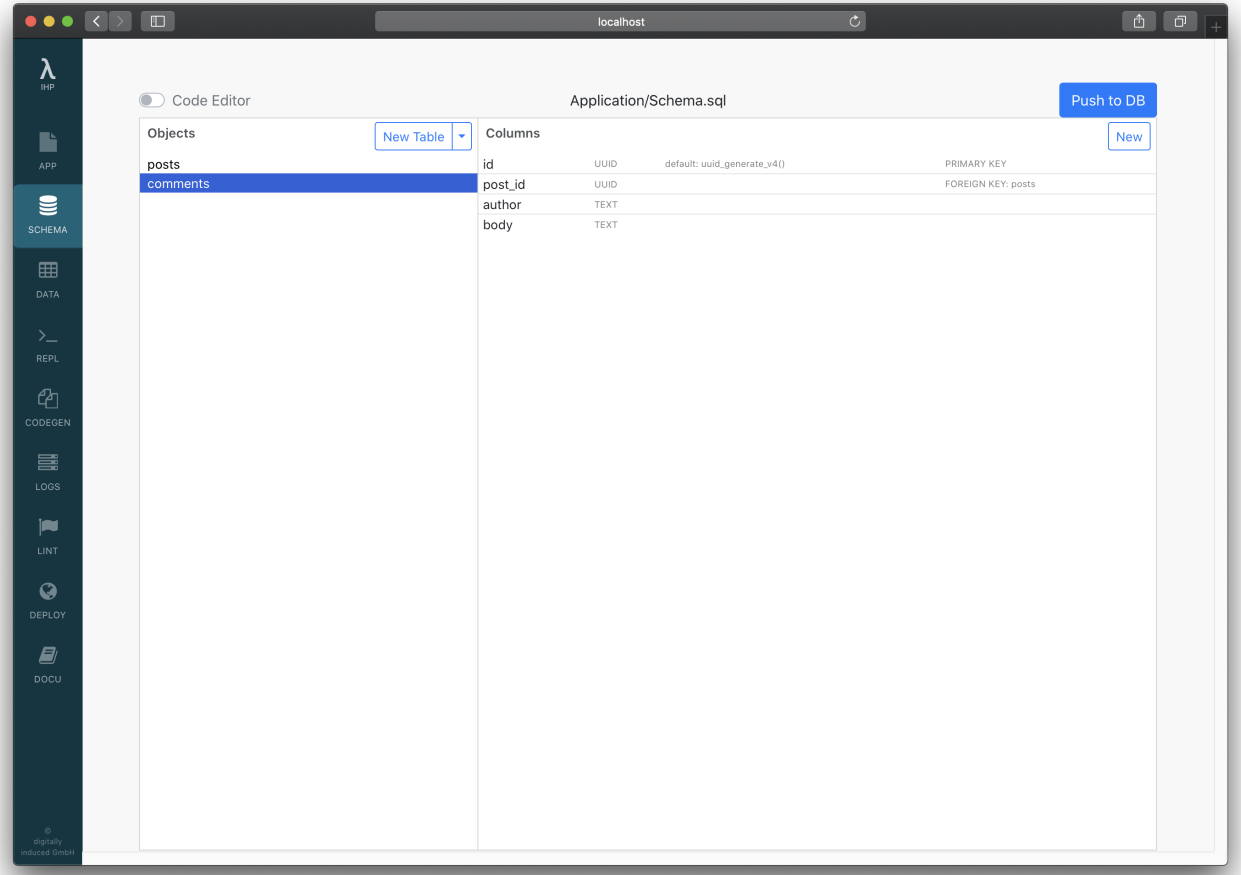
We can use the validator by adding a new `validateField #body isMarkdown` line to the `buildPost` function:

```
buildPost post = post
    |> fill @["title","body"]
    |> validateField #title nonEmpty
    |> validateField #body nonEmpty
    |> validateField #body isMarkdown
```

Create a new post with just `#` (a headline without any text) as the content to see our new error message.

6. Adding Comments

It’s time to add comments to our blog. For that open the Schema Designer and add a new table `comments` with the fields `id`, `post_id`, `author` and `body`:



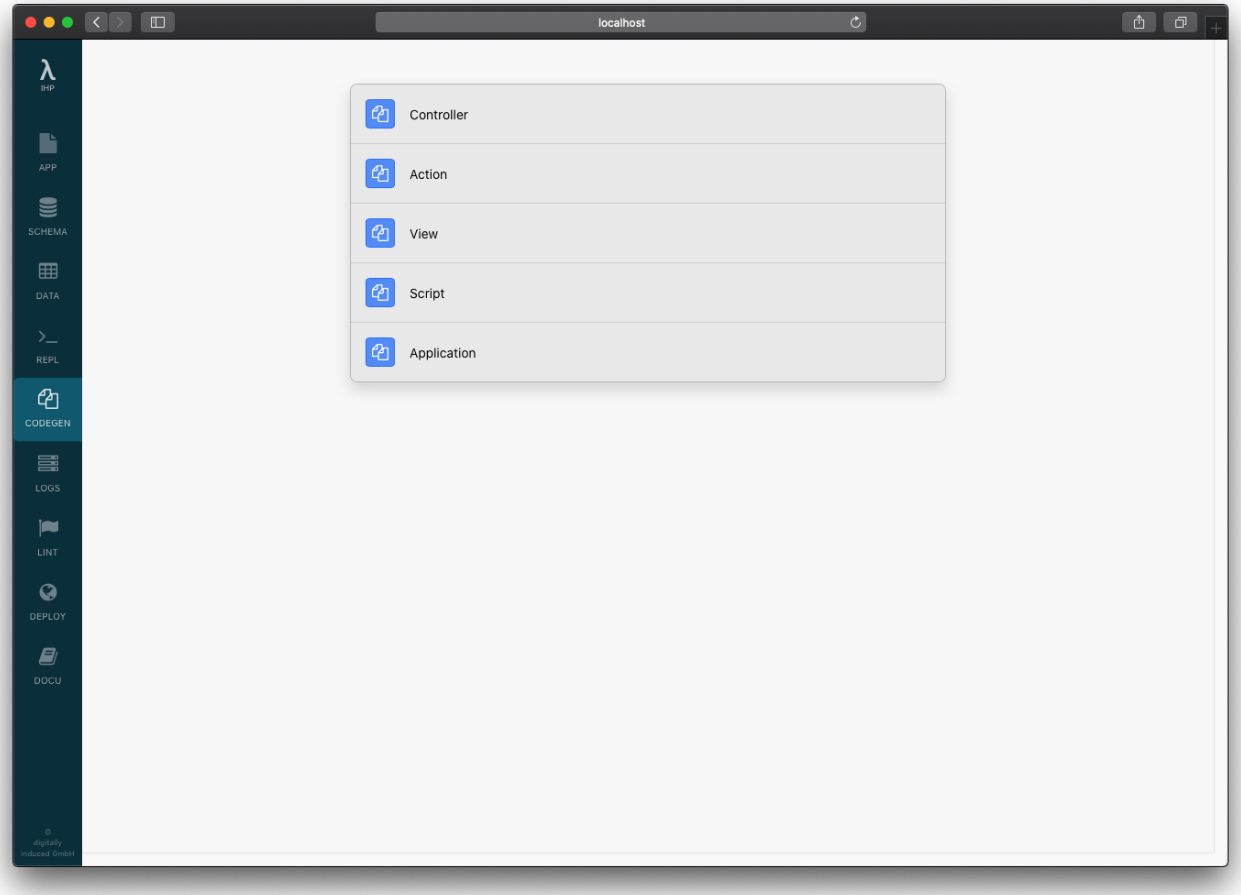
When adding the post_id column, it will automatically set the type to UUID. Unless you unselect the checkbox `References posts` it will also automatically create a foreign key constraint for this column:

By default the foreign key constraint has set its `ON DELETE` behavior to `NO ACTION`. To change the `ON DELETE`, click on the `FOREIGN KEY: posts` next to the `post_id` field.

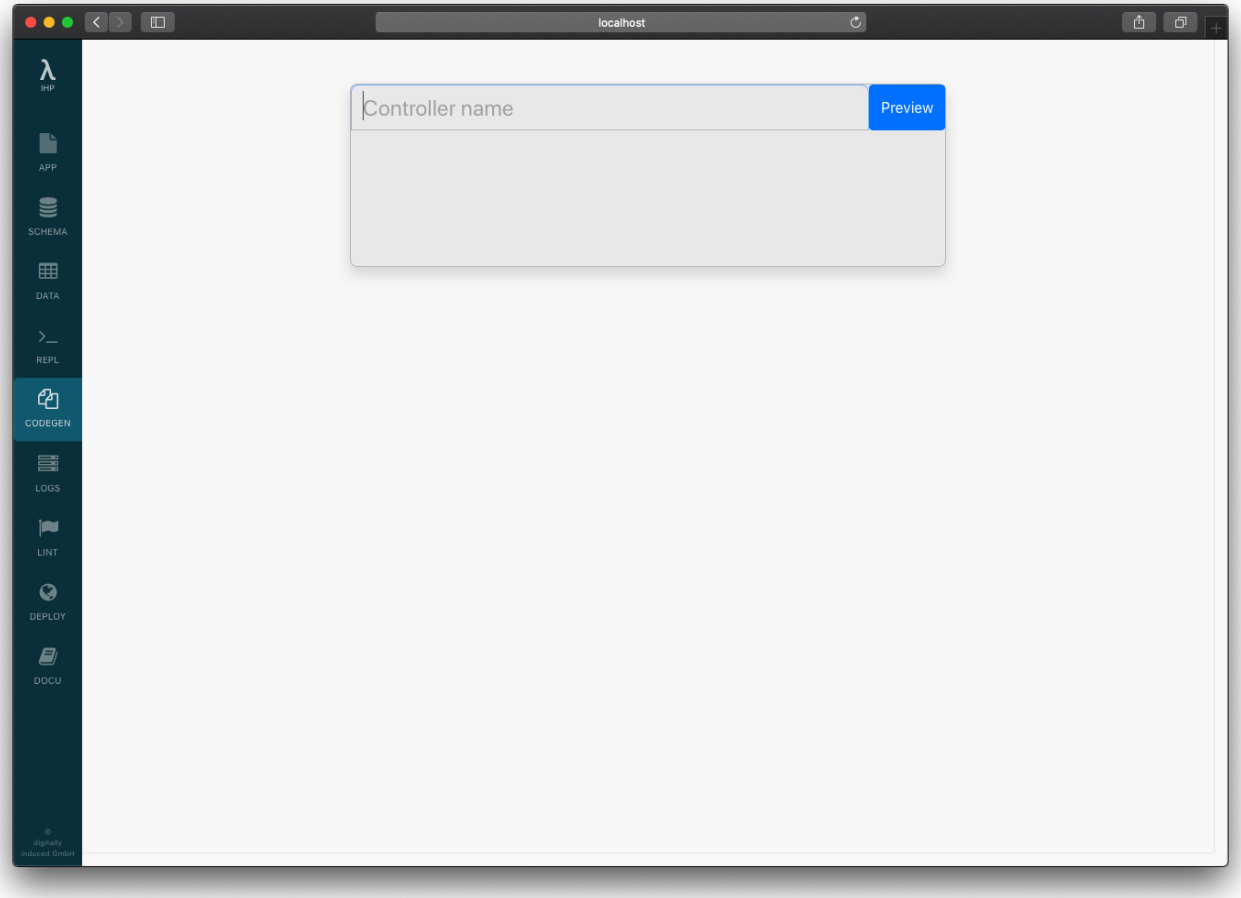
Press the `Update DB`-button to save our current posts to `Application/Fixtures.sql`, rebuild the database and reload the `Fixtures.sql` to add our new `comments` table.

The Controller

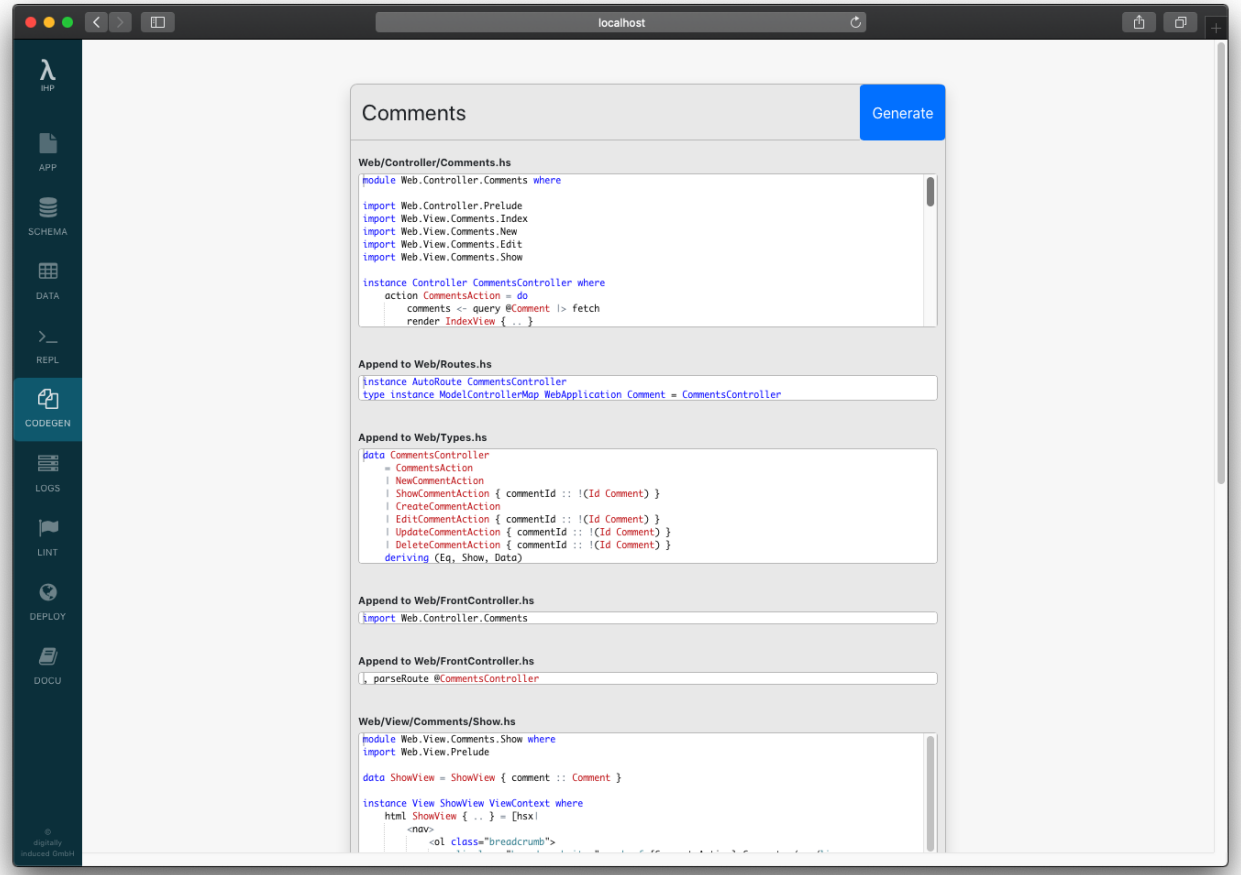
Let's add a controller for our comments. We can use the visual code generator for this:



Use `Comments` as the controller name:



Click `Generate`:



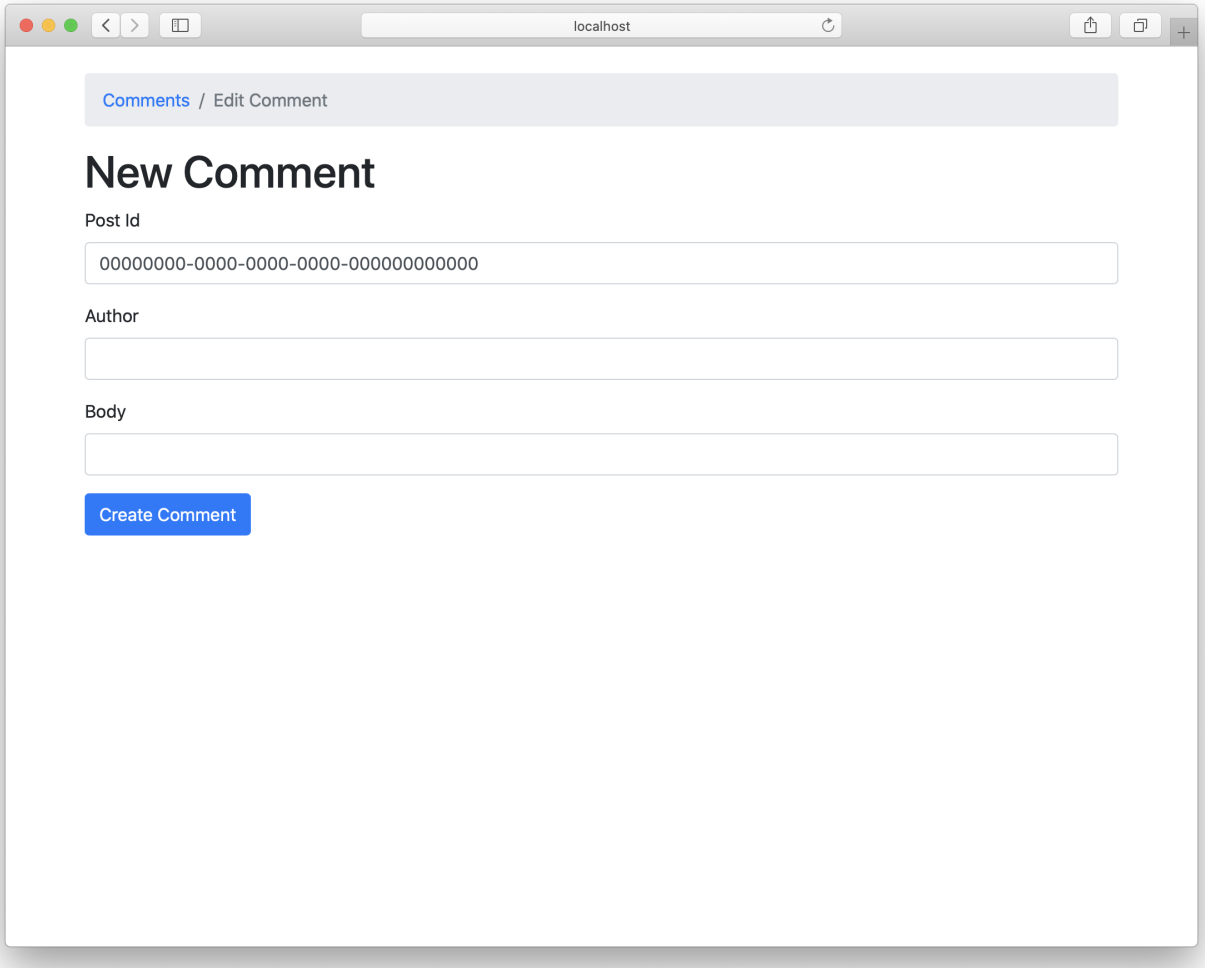
The controller is generated now. But we need to do some adjustments to better integrate the comments into the posts.

“Add comment”

First we need to make it possible to create a new comment for a post. Open `Web/View/Posts/Show.hs` and append `<a href={NewCommentAction}>Add Comment</a>` to the HSX code:

```
instance View ShowView ViewContext where
    html ShowView { .. } = [hsx|
        <nav>
            <ol class="breadcrumb">
                <li class="breadcrumb-item"><a href={PostsAction}>Posts</a>
                <li class="breadcrumb-item active">Show Post</li>
            </ol>
        </nav>
        <h1>{get #title post}</h1>
        <p>{get #createdAt post |> timeAgo}</p>
        <div>{get #body post |> renderMarkdown}</div>

        <a href={NewCommentAction}>Add Comment</a>
    |]
```

This creates an `Add Comment` link, which links to the New Comment Form we just generated. After clicking the `Add Comment` link, we can see this:



We can see, there is a post id field which is field with a lot of 0s. When we try to submit this form, it will fail because there is no post with this id. Let's first make it possible, that the post id is automatically set to the post where we originally clicked on "New Comment".

For that, open `Web/Types.hs`. We can see the definition of `CommentsController`:

```
data CommentsController
    = CommentsAction
    | NewCommentAction
    | ShowCommentAction { commentId :: !(Id Comment) }
    | CreateCommentAction
    | EditCommentAction { commentId :: !(Id Comment) }
    | UpdateCommentAction { commentId :: !(Id Comment) }
    | DeleteCommentAction { commentId :: !(Id Comment) }
    deriving (Eq, Show, Data)
```

Let's add an argument `postId :: !(Id Post)` to `NewCommentAction`:

```
data CommentsController
    -- ...
    | NewCommentAction { postId :: !(Id Post) }
    -- ...
```

After making this change, we can see some type errors in the browser. This is because all references to `NewCommentAction` now need to be passed the `postId` value. Think of these type errors as a todo list of changes to be made, reported to us by the compiler.

Open `Web/View/Posts/Show.hs` and change `<a href={NewCommentAction}>Add Comment</a>` to:

```
<a href={NewCommentAction (get #id post)}>Add Comment</a>
```

After that, another type error can be found in `Web/View/Comments/Index.hs`. In this auto-generated view we have a `New Comment` button at the top:

```
<h1>Comments <a href={pathTo NewCommentAction} class="btn btn-primary ml-4"
```

Let's just remove this button by changing this line to:

```
<h1>Comments</h1>
```

Now we see another type error:

```
Web/Controller/Comments.hs:14:12: error:
    • The constructor ‘NewCommentAction’ should have 1 argument, but has be
    • In the pattern: NewCommentAction
      In an equation for ‘action’:
          action NewCommentAction
            = do let comment = ...
                 render NewView {..}
      In the instance declaration for ‘Controller CommentsController’
   |
14 |     action NewCommentAction = do
   |            ^^^^^^^^^^^^^^^^
```

Open `Web/Controller/Comments` and add the missing `{ postId }` in the pattern match at line 14:

```
    action NewCommentAction { postId } = do
        let comment = newRecord
        render NewView { .. }
```

Now all type errors should be fixed.

Open http://localhost:8000/Posts and open the Show View of a post by clicking its title. Now Click `Add Comment`. Take a look at the URL, it will something like:

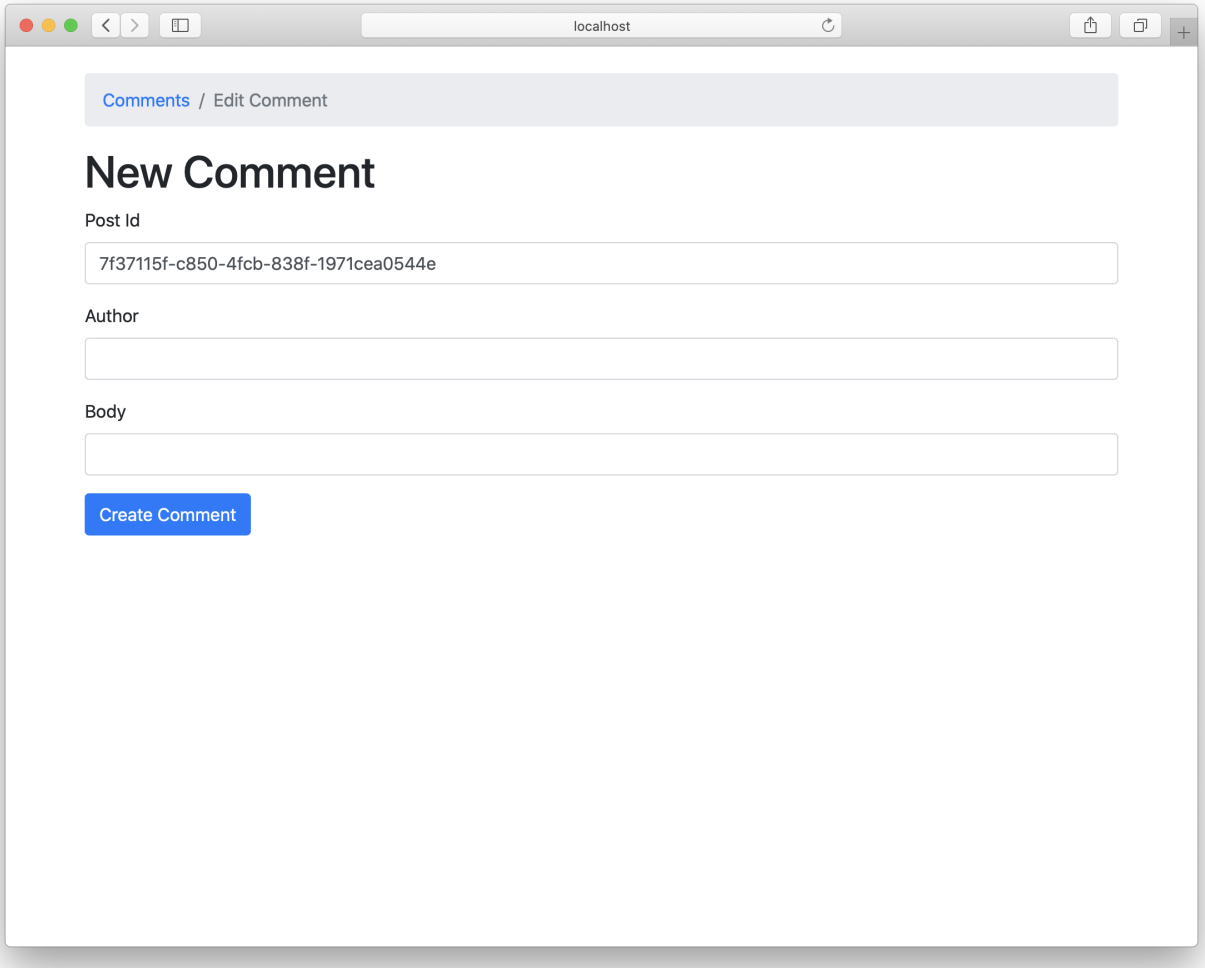
```
http://localhost:8000/NewComment?postId=7f37115f-c850-4fcb-838f-1971cea054
```

You can see that the `postId` has been passed as a query parameter. In the form, the post id field is still filled with 0s. Let's fix this. Open `Web/Controller/Comments.hs` and change the `NewCommentAction` to this:

```
    action NewCommentAction { postId } = do
        let comment = newRecord
                |> set #postId postId
        render NewView { .. }
```

This will set the `postId` of our new comment record to the `postId` given to the action.

Now take a look at your form. The `postId` will be prefilled now:



Of course, seeing the UUID is not very human-friendly. It would be better to just show the post title to our users. For that, we have to fetch and pass the post to our form and then make the `postId` a hidden field.

Append `post <- fetch postId` to fetch the post to the `NewCommentAction`:

```
    action NewCommentAction { postId } = do
        let comment = newRecord
                |> set #postId postId
        post <- fetch postId
        render NewView { .. }
```

Because the error view is rendering our `NewView` in an error case, we also have to update the `CreateCommentAction`:

```
    action CreateCommentAction = do
        -- ...
                Left comment -> do
                    post <- fetch (get #postId comment) -- <---- NEW
                    render NewView { .. }
                Right comment -> - ....
```

Inside the `Web/View/Comments/New.hs` retrieve the `post` variable from the action by updating the `NewView`:

```
data NewView = NewView
    { comment :: Comment
    , post :: Post
    }
```

This way the post is passed from the action to our view.

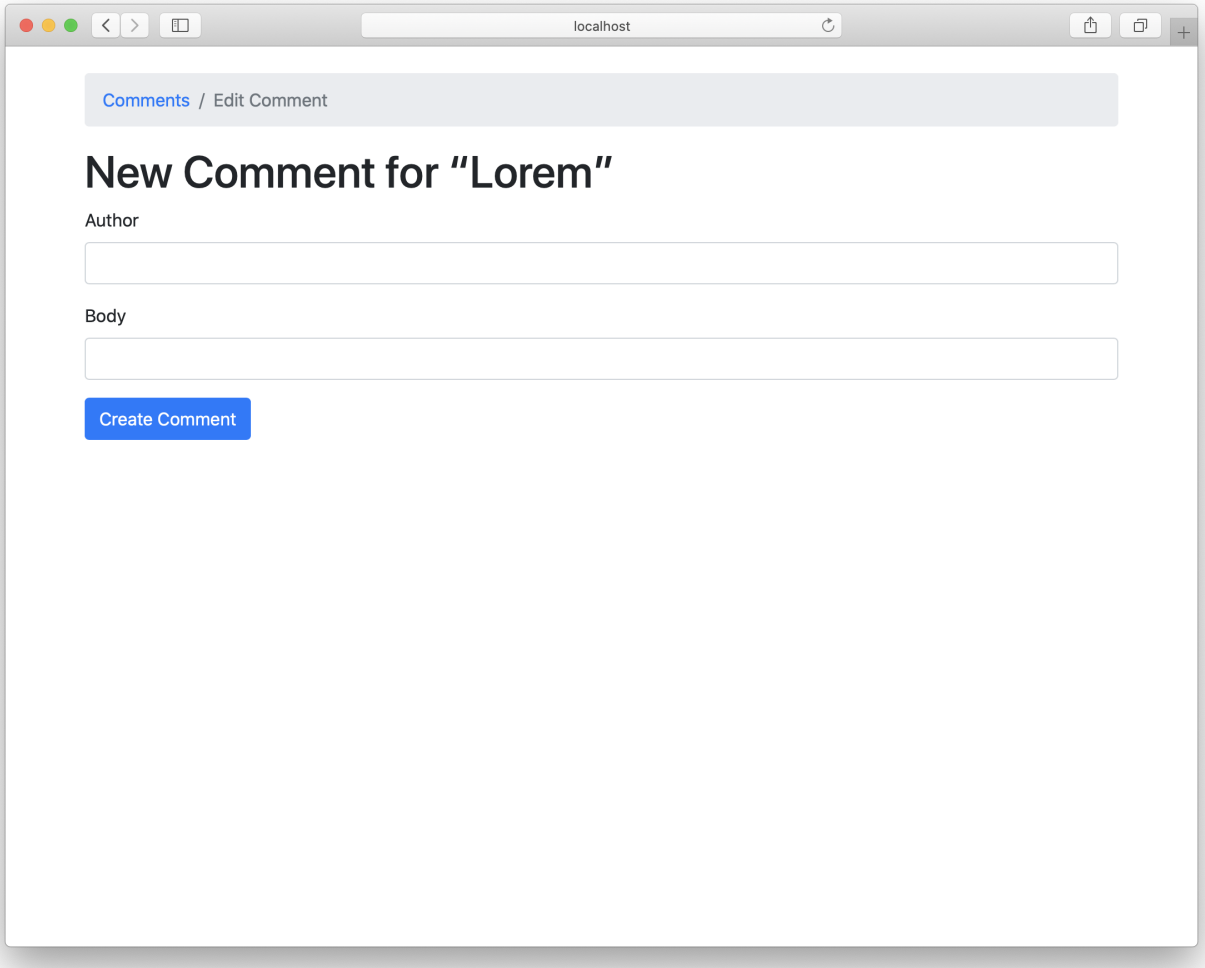
Now we can use the `post` variable to show the post title. Change `<h1>New Comment</h1>` to:

```
<h1>New Comment for <q>{get #title post}</q></h1>
```

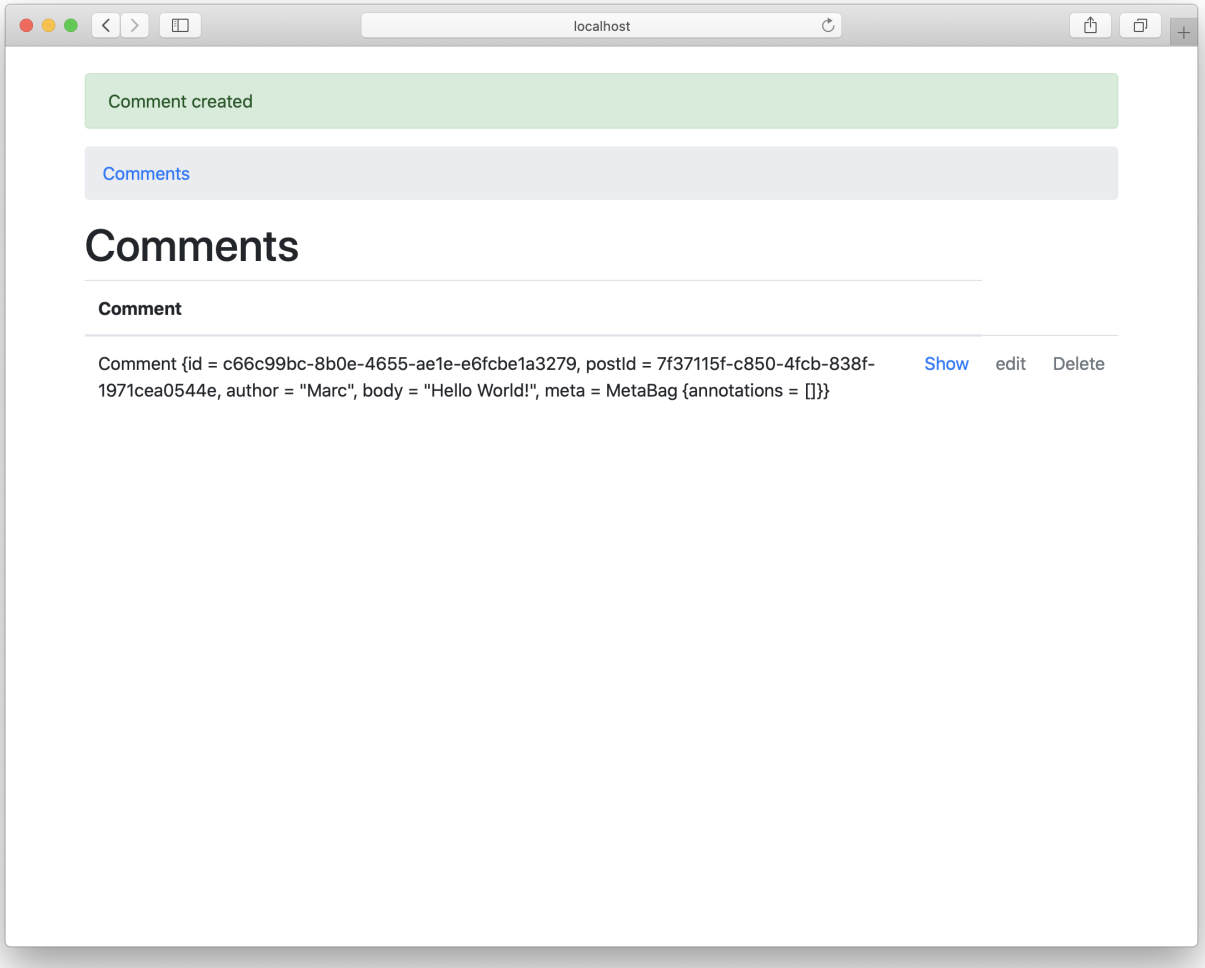
Let's also make the text field for `postId` a hidden field:

```
renderForm :: Comment -> Html
renderForm comment = formFor comment [hsx|
    {hiddenField #postId}
    {textField #author}
    {textField #body}
    {submitButton}
|]
```

Our form is complete now :-) Time to take a look:



Great, let’s add our first comment:



It works. We're redirected to the `CommentsAction`. If you look at the table, we can see that our `postId` has been set successfully.

Let's change our `CreateCommentAction` to make it redirect back to our Post again after commenting. Open `Web/Controller/Comments.hs` and take a look at the

`CreateCommentAction`.

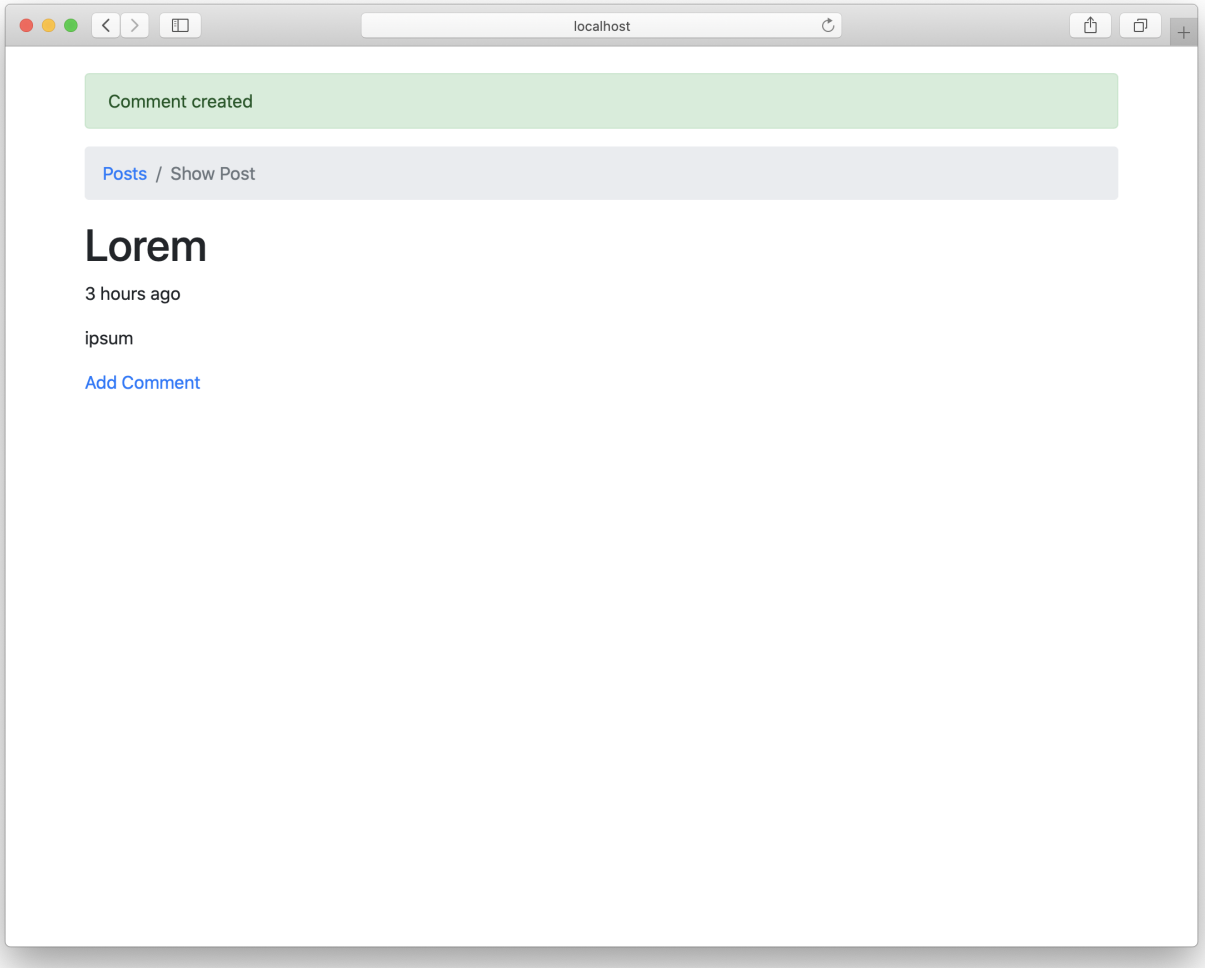
In the success case (`Right comment -> ...`) we see:

```
redirectTo CommentsAction
```

Change this to:

```
redirectTo ShowPostAction { postId = get #postId comment }
```

Open the browser and create a new comment to verify that this redirect is working:

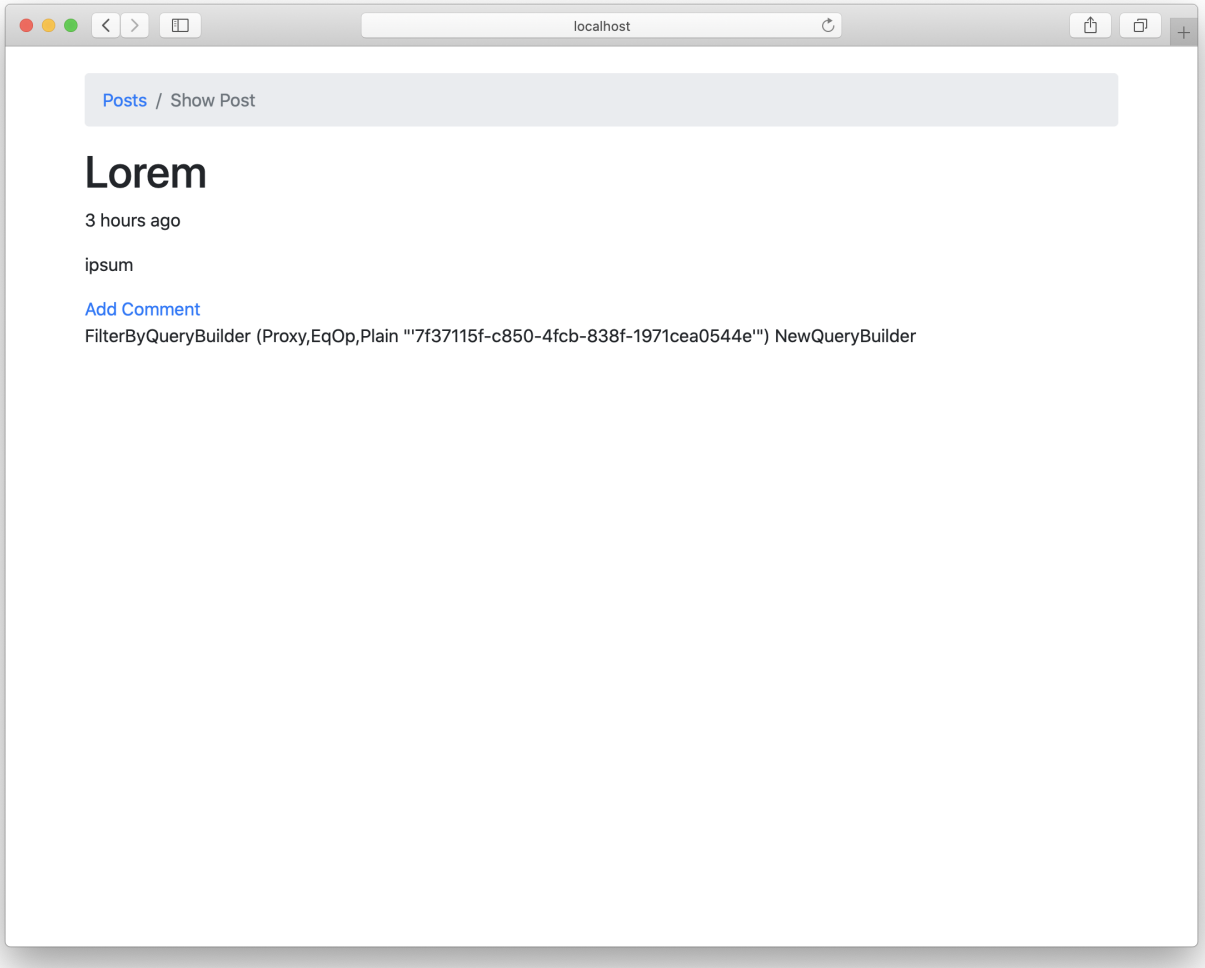


Show Comments of a Post

Next we're going to display our comments below the post. Open `Web/View/Posts/Show.hs` and append the following code to the HSX block:

```
<div>{get #comments post}</div>
```

It will display something like this:



What is shown is the technical representation of a query like `query @Comment |> filterWhere (#id, "'7f37115f-c850-4fcb-838f-1971cea0544e")` (this representation changes a bit between versions still, so don't worry if yours does not look exactly like this). But we don't want just the query, we want the actual comments. We cannot do this from our

view, because views should be pure functions without IO. So we need to tell the action to acutally fetch them for us.

Inside the `Show.hs` we need to update the type signature to tell our action what we want. Right now we have:

```
data ShowView = ShowView { post :: Post }
```

Add an `Include "comments"` like this:

```
data ShowView = ShowView { post :: Include "comments" Post }
```

This specifies that our view requires a post and should also include its comments. This will trigger a type error to be shown in the browser because our `ShowPostAction` is not passing the comments yet.

To fix this, open `Web/Controller/Posts.hs` and take a look at the`ShowPostAction`. Right now we have a `fetch` call:

```
post <- fetch postId
```

We need to extend our fetch to also include comments. We can use `fetchRelated` for this:

```
post <- fetch postId
    >>= fetchRelated #comments
```

```
```

The type of `post` has changed from `Post` to `Include "comments" Post`. In general, when you're dealing with has-many relationships, use `Include "relatedRecords"` and `fetchRelated` to specify and fetch data according to your needs.

The type error is fixed now. When opening the Show View of a post, you will see that the comments are displayed. When you take a look at the `Logs` in the Dev tools you can see, that when opening a Post, two sql queries will be fired:

```
("SELECT posts.* FROM posts WHERE id = ?  LIMIT 1",[Plain "'7f37115f-c850-4
("SELECT comments.* FROM comments WHERE post_id = ?  ",[Plain "'7f37115f-c8
```

Right now the view is displaying the comments as a string. Let's make it more beautiful. Open `Web/View/Posts/Show.hs`.

Let's first change the `{get #comments post}` to make a `<div>` for each comment:

```
<div>{forEach (get #comments post) renderComment}</div>
```

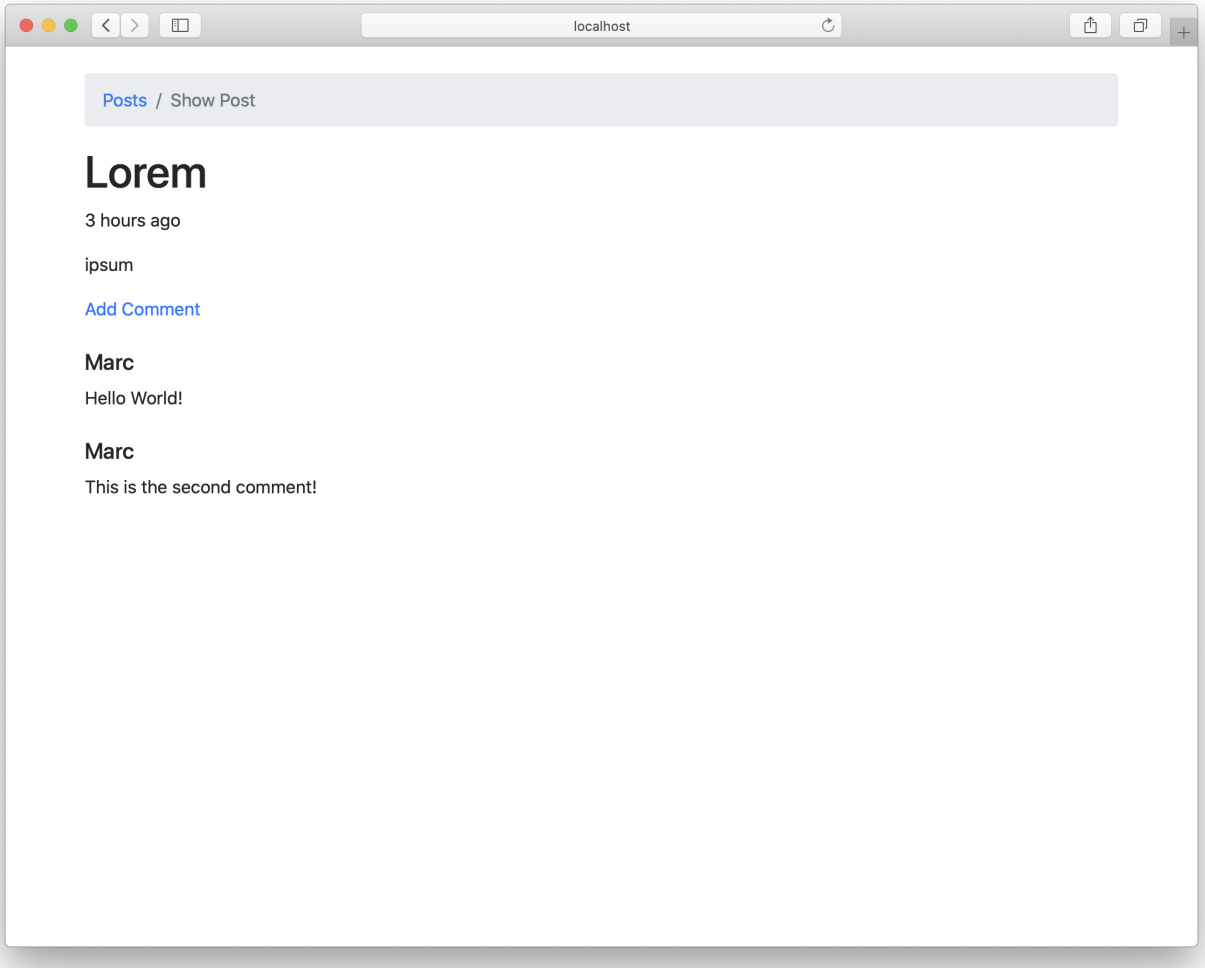
We also need to define the `renderComment` at the end of the file:

```
renderComment comment = [hsx|<div>{comment}</div>|]
```

Let's also add some more structure for displaying the comments:

```
renderComment comment = [hsx|
        <div class="mt-4">
            <h5>{get #author comment}</h5>
            <p>{get #body comment}</p>
        </div>
    |]
```

This is how it looks now:



Ordering Comments

Right now comments are displayed in the order they're stored in the database. So updating a comment will change the order. Let's change this, so that the newest comment is always displayed first.

Open the Schema Designer. Select the `comments` Table. Right click in the Columns Pane, Click `Add Column`. Enter `created_at`. The column type will be auto selected, and the default value automatically sets to `NOW()`. Click `Create Column`.

Now click `Update DB` to save the fixtures and then rebuild the database.

Now we have a `created_at` timestamp we can use for ordering. Open `Web/Controller/Posts.hs` and change the action from:

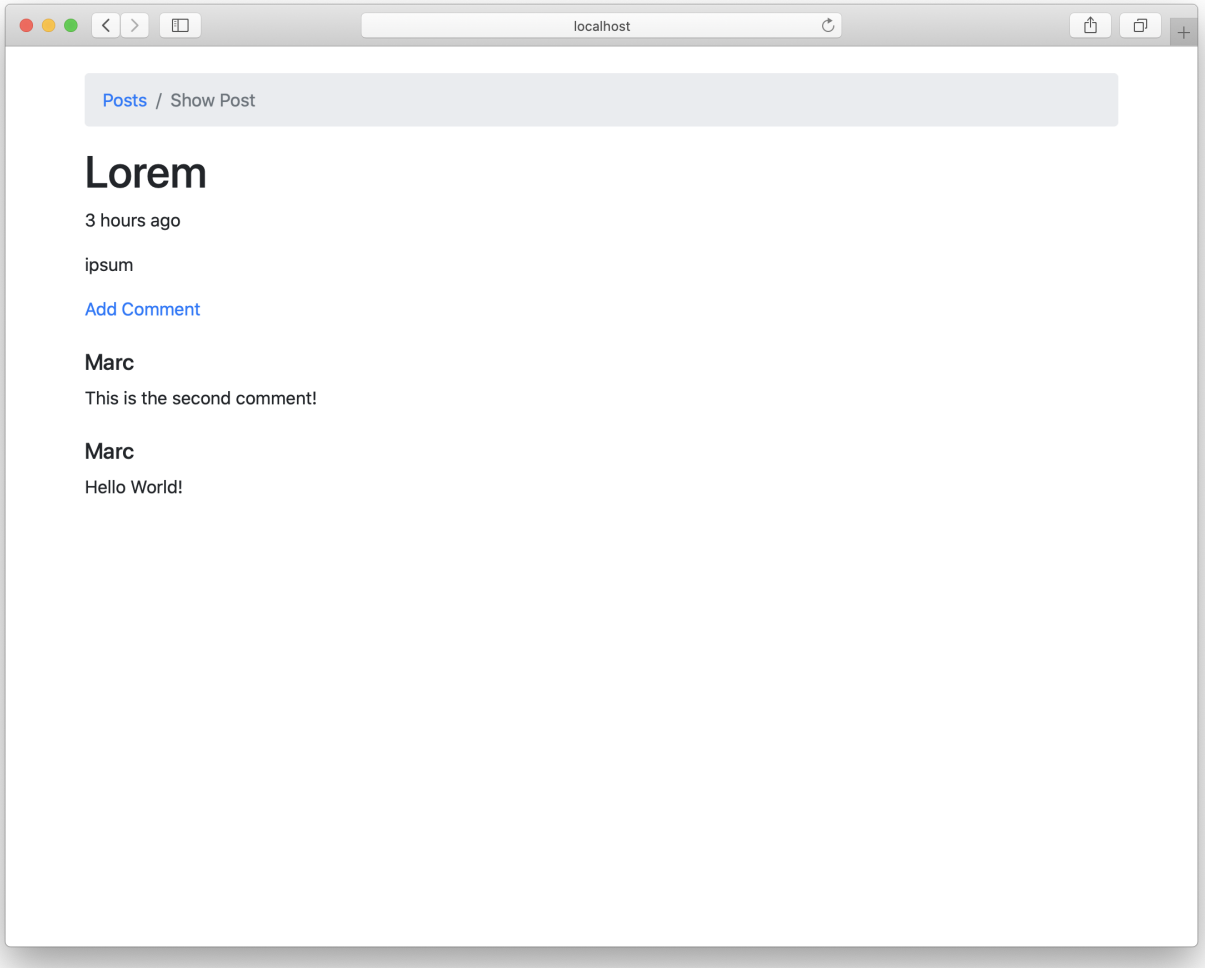
```
    action ShowPostAction { postId } = do
        post <- fetch postId
            >>= fetchRelated #comments
        render ShowView { .. }
```

To the following:

```
    action ShowPostAction { postId } = do
        post <- fetch postId
            >>= pure . modify #comments (orderByDesc #createdAt)
            >>= fetchRelated #comments
        render ShowView { .. }
```

The `modify #comments (orderByDesc #createdAt)` basically just does a `|> orderByDesc #createdAt` to the query builder inside the `#comments` field. Then it just writes it back to the field. The `fetchRelated #comments` will then use the query builder stored inside `#comments` to fetch the comments, thus using the `ORDER BY` we added to the query.

That's it already. Taking a look at our post, we can see that the newest comment is shown first now.



Have Fun!

You should have a rough understanding of IHP now. The best way to continue is to start building things. Take a look at the `The Basics` Section to learn more about all the provided modules.

Leave a Star on the IHP-GitHub repo and join the IHP community to work on the future of typesafe, FP-based software development.

To stay in the loop, subscribe to the IHP release emails.

Questions, or need help with haskell type errors? Join our Gitter Chat:

(IRC available)

Architecture

This section tries to answer common questions on where to place your code. These are recommendations found by digitally induced to be working well.

In general remember that all specific web app logic should stay in the `Web/` space. The `Application/` space is for sharing code across all your different applications. E.g. code shared between your web application and your admin backend.

Directory Structure

File or Directory	Purpose
Config/	
Config/Config.hs	Configuration for the framework and your application
Config/nix/nixpkgs-config.nix	Configuration for the nix package manager
Config/nix/haskell-packages/	Custom Haskell dependencies can be placed here
Application/	Your domain logic lives here
Application/Schema.sql	Models and database tables are defined here
Web/Controller	Web application controllers
Web/View/	Web application html template files
Web/Types.hs	Central place for all web application types
static/	Images, css and javascript files
.ghci	Default config file for the Haskell interpreter
.gitignore	List of files to be ignored by git

File or Directory	Purpose
App.cabal, Setup.hs	Config for the cabal package manager (TODO: maybe move to Config/App.cabal)
default.nix	Declares your app dependencies (like package.json or composer.json)
Makefile	Default config file for the make build system

FAQ

Where to place a function I want to use in all my views?

If the function is only used in a single application and is a building block for your layout, place it in `Web/View/Layout.hs`. The module is already imported in all your views (just don't forget to add the function to the export list).

If the function is used across multiple applications or more like a helper function, place it in `Application/Helper/View.hs`. This module is also already included in your view files.

Where to place a function I want to use in all my controllers?

Place it in `Application/Helper/Controller.hs`. This module is already imported into your controllers.

Where to place a custom type?

Place it in `Web/Types.hs`.

Next to my main web application, I'm building an admin backend application. Where to place it?

A IHP project can consist of multiple applications. Run `new-application admin` to generate a new admin application. The logic for the new application is located in the `Admin/` directory. On the web you can find it at `http://localhost:8000/admin/` (all actions are prefixed with `/admin/`).

How to structure my CSS?

CSS files, as all your other static assets, should be placed in the `static` directory.

Create a `static/app.css`. In there use CSS imports to import your other stylesheets. An example `app.css` could look like this:

```
@import "/layout.css";
@import "/widget.css";
@import "/form.css";
@import "/button.css";
@import "/users.css";
```

Page-specific CSS rules

Place page-specific CSS used by e.g. views of the `Web.Controller.Users` controller in `users.css`. Use currentViewId to scope your css rules to the view.

Given the view:

```
module Web.View.Projects.Show where
```

```
render = [hsx|
    <div id={currentViewId}>
        <h1>Hello World!</h1>
    </div>
|]
```

This will render like

```
<div id="projects-show">
    <h1>Hello World!</h1>
</div>`
```

So in your `projects.css` you can just do rules like

```
#projects-show h1 { color: blue; }
```

SASS & Webpack

We discourage the use of tools like SASS or webpack because they have too much overhead.

Library CSS

CSS files from external libraries or components should be placed in `static/vendor/`.

How to structure my Javascript Code?

JS files, as all your other static assets, should be place in the `static` directory.

In general we follow an approach where most of the business logic resides on the Haskell server. Only for small interactions we try to use a small isolated bit of javascript.

Your global, non-page specific, javascript code can be placed in `app.js`.

E.g. the `app.js` could look like this:

```
$(function () {
    initNavbarEffects();
});

function initNavbarEffects() {
    // ...
}
```

In your `Web.View.Layout` just import the `app.js`:

```
<script src="/app.js"></script>
```

Page-specific JS

Place page-specific JS used by e.g. views of the `Web.Controller.Users` controller in the `users.js`.

In the views, just import the javascript with `<script src="/users.js"></script>`.

Webpack

We discourage the use of webpack or any other bundler because they have too much overhead. Of course this advice only applies if you follow the approach to use as little javascript as possible.

Library JS

JS files from external libraries or components should be placed in `static/vendor/`. For simplicity it might make sense to just download the javascript bundle of the library you want to use, and then just commit it into git instead of using NPM.

For more complex use-cases with lots of javascript, you should not follow this advice and just use NPM instead.

Where to place static images?

Place your images in the `static` folder. We recommend to use SVG images.

Naming Conventions

Introduction

The code of applications powered by IHP should always feel like it's the same kind of application. This helps to quickly switch between projects. Also the end user of your application does not care whether something uses camel case or snake case, but it's important that this decision is used consistently in the project. Therefore IHP comes with a set of rules on how to name certain things.

Database

Table names should always use lowercase, snake case and be in the plural form.

Here are some examples of good names:

```
users
projects
companies
user_projects
company_admins
reactions
invites
comments
```

Here are some examples of bad names:

```
post -- Not in plural form, should be `posts`
UserProjects -- Not in snake case, should be `user_projects`
Posts -- Should be lowercase `posts`
company -- Should be `companies`
```

View

Module Names:

View modules should always follow this naming schema:

```
module <app>.View.<controller>.<actionVerb>
```

The `<controller>` should not end in `Controller`. The `<actionVerb>` should not end in `View`.

Here are some examples of good names:

```
module Web.View.Users.Show
module Web.View.Users.Edit
module Web.View.Companies.New
module Admin.View.Users.New
module Admin.View.UserProjects.Show
```

Here are some examples of bad names:

```
module Web.View.PostsController.Show -- Should not end in "Controller"
module Web.View.Posts.EditView -- Should not end in "View"
module Web.View.Post.New -- Should most-likely be plural (unless the contr
```

Data Structure:

The `View` data structure should always end in `View`, as it's usually imported into the controller module unqualified.

Here are some examples of good data structures:

```
data EditView = EditView
data NewView = NewView
data NewView = NewView
```

```
data IndexView = IndexView
data PostView = SimplePostView | AdvancedPostView
data QuestionTemplatesView = QuestionTemplatesView
```

Here are some examples of bad data structures:

```
data Edit = Edit -- Missing the View suffix
data NewView = NewPost -- The view constructor should be named the same as
```

Controller

Module Names:

Controller modules should always follow this naming schema:

```
module <app>.Controller.<controller>
```

The `<controller>` should not end in `Controller`. It should usually be in plural form, unless it's only working on a single entity. E.g. you might have a `UserController` when only dealing with the current user, because from the users point of view, there is only a single user resource they can interact with.

Here are some examples of good names:

```
module Web.Controller.Companies
module Web.Controller.Users
module Web.Controller.Static
module Admin.Controller.Sessions
```

Here are some examples of bad names:

```
module Web.Controller.PostsController -- Should not end in Controller
```

Data Structure:

The `Controller` data structure should always end in `Controller`, as it's usually imported into other modules unqualified.

Here are some examples of good data structures:

```
data UsersController
    = UsersAction
    | NewUserAction
    | ShowUserAction { userId :: !(Id User) }
    | CreateUserAction
    | EditUserAction { userId :: !(Id User) }
    | UpdateUserAction { userId :: !(Id User) }
    | DeleteUserAction { userId :: !(Id User) }
    deriving (Eq, Show, Data)

data CompaniesController
```

```
    = CompaniesAction
    | NewCompanyAction
    | ShowCompanyAction { companyId :: !(Id Company) }
    | CreateCompanyAction
    | EditCompanyAction { companyId :: !(Id Company) }
    | UpdateCompanyAction { companyId :: !(Id Company) }
    | DeleteCompanyAction { companyId :: !(Id Company) }
    deriving (Eq, Show, Data)

data AdminsController
    = AdminsAction
    | NewAdminAction
    | ShowAdminAction { adminId :: !(Id Admin) }
    | CreateAdminAction
    | EditAdminAction { adminId :: !(Id Admin) }
    | UpdateAdminAction { adminId :: !(Id Admin) }
    | DeleteAdminAction { adminId :: !(Id Admin) }
    deriving (Eq, Show, Data)

data SessionsController
    = NewSessionAction
    | CreateSessionAction
    | DeleteSessionAction
    deriving (Eq, Show, Data)
```

Here are some examples of bad data structures:

```
data CompanyController -- Type should be plural in this case
    = CompaniesAction
    | NewCompanyAction
```

```
    | ShowCompanyAction { companyId :: !(Id Company) }
    | CreateCompanyAction
    | EditCompanyAction { companyId :: !(Id Company) }
    | UpdateCompanyAction { companyId :: !(Id Company) }
    | DeleteCompanyAction { companyId :: !(Id Company) }
    deriving (Eq, Show, Data)
```

Controller & Actions

Routing Basics

In your project routes are defined in the `Web/Routes.hs`. In addition to defining that route, it also has to be added in `Web/FrontController.hs` to be picked up by the routing system.

The simplest way to define a route is by using `AutoRoute`, which automatically maps each controller action to an url. For a `PostsController`, the definition in `Web/Routes.hs` will look like this:

```
instance AutoRoute PostsController
type instance ModelControllerMap WebApplication Post = PostsController
```

Afterwards enable the routes for `PostsController` in `Web/FrontController.hs` like this:

```
instance FrontController WebApplication where
    controllers =
        [ -- ...
        , parseRoute @PostsController
        ]
```

Now you can open e.g. `/Posts` to access the `PostsAction`.

Changing the Start Page / Home Page

You can define a custom start page action using the `startPage` function like this:

```
instance FrontController WebApplication where
    controllers =
        [ startPage ProjectsAction
        -- Generator Marker
        ]
```

In a new IHP project, you usually have a `startPage WelcomeAction` defined. Make sure to remove this line. Otherwise you will still see the default IHP welcome page.

Url Generation

Use `pathTo` to generate a path to a given action:

```
pathTo ShowPostAction { postId = "adddfb12-da34-44ef-a743-797e54ce3786" }
-- /ShowPost?postId=adddfb12-da34-44ef-a743-797e54ce3786
```

To generate a full url, use `urlTo`:

```
urlTo NewUserAction
-- http://localhost:8000/NewUser
```

AutoRoute

Let's say our `PostsController` is defined in `Web/Types.hs` like this:

```
data PostsController
    = PostsAction
    | NewPostAction
    | ShowPostAction { postId :: !(Id Post) }
    | CreatePostAction
    | EditPostAction { postId :: !(Id Post) }
    | UpdatePostAction { postId :: !(Id Post) }
    | DeletePostAction { postId :: !(Id Post) }
```

Using `instance AutoRoute PostsController` will give us the following routing:

```
GET /Posts                          => PostsAction
GET /NewPost                        => NewPostAction
GET /ShowPost?postId={postId}       => ShowPostAction { postId }
POST /CreatePost                    => CreatePostAction
GET /EditPost?postId={postId}       => EditPostAction { postId }
POST /UpdatePost?postId={postId}    => UpdatePostAction { postId }
PATCH /UpdatePost?postId={postId}   => UpdatePostAction { postId }
DELETE /DeletePost?postId={postId}  => DeletePostAction { postId }
```

The urls are very close to the actual action which is called. Action parameters are taken automatically from the request query. This design helps you to always know which action is actually called, when requesting an url.

AutoRoute & Beautiful Urls

Lots of modern browser don't even show the full url bar anymore (e.g. Safari and most mobile browsers). Therefore AutoRoute doesn't aim to generate the "most" beautiful urls out of the box. It's rather optimized for the needs of developers. If you need beautiful urls for SEO reasons, instead of using AutoRoute you can use the more manual APIs of IHP Routing. See the section "Beautiful URLs" for details.

Multiple Parameters

An action constructor can have multiple parameters:

```
data PostsController = EditPostAction { postId :: !(Id Post), userId :: !(
```

This will generate a routing like:

```
GET /EditPost?postId={postId}&userId={userId} => EditPostAction { postId, u
```

Parameter Types

AutoRoute by default only works with UUID arguments (and Id types like `Id Post`). If you have a controller like `data HelloWorldController = HelloAction { name :: Text }` where you have a text parameter, you have to configure AutoRoute like this:

```
instance AutoRoute HelloWorldController where
    parseArgument = parseTextArgument
```

This way the `name` argument is passed as `Text` instead of `UUID`.

This also works with integer types:

```
instance AutoRoute HelloWorldController where
    parseArgument = parseIntArgument
```

This will support a controller like `data HelloWorldController = HelloAction { page :: Int }`.

Right now AutoRoute supports only a single type for all given parameters. E.g. an action which takes an UUID and a Text is not supported with AutoRoute right now:

```
data HelloController = HelloAction { userId :: !(Id User), name :: Text }
instance AutoRoute HelloController -- This will fail at runtime
```

This is a technical problem we hope to fix in the future. Until then consider using `param` for the `Text` parameter.

Request Methods

When a an action is named a certain way, AutoRoute will pick a certain request method for the route. E.g. for a `DeletePostAction` it will only allow requests with the request method `DELETE` because the action name starts with `Delete`. Here is an overview of all naming patterns and their corresponding request method:

```
Delete_Action => DELETE
Update_Action => POST, PATCH
Create_Action => POST
Show_Action   => GET, HEAD
otherwise     => GET, POST, HEAD
```

If you need more strong rules, consider using the other routing APIs available or overriding the `allowedMethodsForAction` like this:

```
instance AutoRoute HelloWorldController where
    allowedMethodsForAction "HelloAction" = [ GET ]
```

Application Prefix

When using multiple applications in your IHP project, e.g. having an admin backend, AutoRoute will prefix the action urls with the application name. E.g. a controller `HelloWorldController` defined in `Admin/Types.hs` will be automatically prefixed with `/admin` and generate urls such as `/admin/HelloAction`.

This prefixing has special handling for the `Web` module, so that all controllers in the default `Web` module don't have a prefix.

Advanced: Custom parseArgument

It's possible to use a custom data type as a routing parameter with AutoRoute. Now might be a good point to switch to a custom routing implementation (described later in this Guide) instead of hacking this into AutoRouting. If this warning can't stop you, go ahead.

A `parseArgument` function has the signature:

```
parseArgument :: forall d. Data d => ByteString -> ByteString -> d
```

The first bytestring argument is the field name of the action argument we are dealing with. The second argument is the value of the query string:

```
MyAction?{firstArgument}={secondArgument}
```

The last argument `d` cannot be implemented in a typesafe way. This is implemented by calling `unsafeCoerce` on our result value before returning it. The result of this is later used with `fromConstrM`. Therefore misusing `parseArgument` can result in a runtime crash. Again, consider not using this API.

Given we have a custom argument type in the format `ID-{numeric}` like `ID-0`, `ID-1`, etc, we can define a custom `parseCustomIdArgument` like this:

```
import qualified Data.Attoparsec.ByteString.Char8 as Attoparsec
import qualified Data.ByteString.Char8 as ByteString

parseCustomIdArgument :: forall d. Data d => ByteString -> ByteString -> d
parseCustomIdArgument field value =
    value
    |> ByteString.stripPrefix "ID-"
    |> fromMaybe (error "Failed to parse custom id, ID- missing")
    |> Attoparsec.parseOnly (Attoparsec.decimal <* Attoparsec.endOfInput)
    |> \case
        Right value -> unsafeCoerce value
        Left _ -> error "AutoRoute: Failed to parse custom id, numeric part
```

Custom Routing

Sometimes you have special needs for your routing. For this case IHP provides a lower-level routing API on which `AutoRoute` is built.

Let's say we have a controller like this:

```
data PostsController = ShowAllMyPostsAction
```

We want requests to `/posts` to map to `ShowAllMyPostsAction`. For that we need to add a `CanRoute` instance:

```
instance CanRoute PostsController where
    parseRoute' = string "/posts" <* endOfInput >> pure ShowAllMyPostsActi
```

The `parseRoute'` function is a parser which reads an url and returns an action of type `PostsController`. The router uses attoparsec. See below for examples on how to use this for building beautiful urls.

Next to the routing itself, we also need implement the url generation:

```
instance HasPath PostsController where
    pathTo ShowAllMyPostsAction = "/posts"
```

Beautiful URLs

Let's say we want to give our blog post application a beautiful url structure for SEO reasons. Our controller is defined as:

```
data PostsController
    = ShowPostAction { postId :: !(Id Post) }
```

We want our urls to look like this:

```
/posts/an-example-blog-post
```

Additionally we also want to accept permalinks with the id like this:

```
/posts/f85dc0bc-fc11-4341-a4e3-e047074a7982
```

To accept urls like this, we first need to make some changes to our data structure. We have to make the `postId` optional. Additonally we need to have a parameter for the url slug:

```
data PostsController
    = ShowPostAction { postId :: !(Maybe (Id Post)), slug :: !(Maybe Text)
```

This will also require us to make changes to our action implementation:

```
action ShowPostAction { postId, slug } = do
    post <- case slug of
            Just slug -> query @Post |> filterWhere (#slug, slug) |> fetch
            Nothing   -> fetchOne postId
    -- ...
```

This expects the `posts` table to have a field `slug :: Text`.

Now we define our `CanRoute` instance like this:

```
instance CanRoute PostsController where
    parseRoute' = do
        string "/posts/"
        let postById = do id <- parseId; endOfInput; pure ShowPostAction {
        let postBySlug = do slug <- remainingText; pure ShowPostAction { po
        postById <|> postBySlug
```

Additionally we also have to implement the `HasPath` instance:

```
instance HasPath PostsController where
    pathTo ShowPostAction { postId = Just id, slug = Nothing } = "/posts/"
    pathTo ShowPostAction { postId = Nothing, slug = Just slug } = "/posts/
```

Real-World Example

Here is a real world example of a custom routing implementation for a custom Apple Web Service interface implemented at digitally induced:

```
instance CanRoute RegistrationsController where
    parseRoute' = do
        appleDeviceId <- string "AppleWebService/v1/devices/" *> parseText
        passType <- parseText

        let create = do
            string "/"
```

```
            memberId  <- parseId
            endOfInput
            pure CreateRegistrationAction { .. }
        let show = do
            endOfInput
            pure ShowRegistrationAction { .. }

        choice [ create, show ]

instance HasPath RegistrationsController where
    pathTo CreateRegistrationAction { appleDeviceId, memberId } = "/AppleW
    pathTo ShowRegistrationAction { appleDeviceId } = "/AppleWebService/v1/
```

Method Override Middleware

HTML forms don't support special http methods like `DELETE`. To work around this issue, IHP has a middleware which transforms e.g. a `POST` request with a form field `_method` set to `DELETE` to a `DELETE` request.

Controller & Actions

Introduction

In IHP an action is a place for request handling logic. A controller is just a group of related actions.

You can think about an action as a message sent to your application, e.g. a `ShowPostAction { postId :: Id Post }` is basically the message "Show me the post with id $postId".

Each action needs to be defined as a data structure inside `Web/Types.hs`. Therefore you can see an overview of all the messages which can be sent to your application just by looking at `Web/Types.hs`.

Creating a new Controller

We recommend to use the code generators for adding a new controller. Using the GUI, you can open http://localhost:8001/NewController. Using the CLI run `new-controller CONTROLLER_NAME`.

The following section will guide you through the manual process of creating a new controller.

Let's say we want to create a new controller with a single action `ShowPostAction`. First we need to define our types in `Web/Types.hs`:

```
data PostsController
    = ShowPostAction { postId :: !(Id Post) }
    deriving (Eq, Show, Data)
```

This defines a type `PostsController` with a data constructor `ShowPostAction { postId :: !(Id Post) }`. The arguent `postId` will later be filled with the `postId` parameter of the request url. This is done automatically by the IHP router. IHP also requires the controller to have `Eq`, `Show` and `Data` instances. Therefore we derive them here.

After we have defined the "interface" for our controller, we need to implement the actual request handling logic. IHP expects to find this inside the `action` function of the `Controller` instance. We can define this instance in `Web/Controller/Posts.hs`:

```
module Web.Controller.Posts where

import Web.Controller.Prelude

instance Controller PostsController where
    action ShowPostAction { postId } = renderPlain "Hello World"
```

This implementation for `ShowPostAction` responds with a simple plain text message. The `action` implementation is usually a big pattern match over all possible actions of a controller.

Reading Query and Body Parameters

Inside the action you can access request parameters using the `param` function. A parameter can either be a url parameter like `?paramName=paramValue` (*this is also called a query parameter*), or given as a form field like `<form><input name="paramName" value="paramValue"/></form>` (*in that case we're talking about a body parameter*). The `param` function will work with query and body parameters, so you don't have to worry about that (in case a query and body parameter is set with the same name, the body parameter will take priority).

Given a request like `GET /UsersAction?maxItems=50`, you can access the `maxItems` like this:

```
action UsersAction = do
    let maxItems = param @Int "maxItems"
```

An alternative request to that action can use a form for passing the `maxItems`:

```
<form action={UsersAction} method="POST">
    <input type="text" name="maxItems" value="50"/>
    <button type="submit">Send</button>
</form>
```

The value is automatically transformed to an `Int`. This parsing works out of the box for Ids, UUID, Bools, Timestamps, etc. Here are some more examples:

```
action ExampleAction = do
    let userName = param @Text "userName"
    let timestamp = param @UTCTime "createdAt"
    let userId = param @(Id User) "userId"
```

Missing parameters

In case there is a problem parsing the request parameter, an error will be triggered.

When the parameter is optional, use `paramOrDefault`:

```
action UsersAction = do
    let maxItems = paramOrDefault @Int 50 "maxItems"
```

When this action is called without the `maxItems` parameter being set (or when invalid), it will fall back to the default value `50`.

There is also `paramOrNothing` which will return `Nothing` when the parameter is missing and `Just theValue` otherwise.

Multipe Params With Same Name (Checkboxes)

When working with checkboxes sometimes there are multiple values for a given parameter name. Given a form like this:

```
<h1>Pancake</h1>

<input name="ingredients" type="checkbox" value="milk" /> Milk

<input name="ingredients" type="checkbox" value="egg" /> Egg
```

When both checkboxes for Milk and Egg are checked, the usual way of calling `param @Text "ingredients"` will only return the first ingredient `"Milk"`. To access all the checked `ingredients` use `paramList`:

```
action BuildFood = do
    let ingredients = paramList @Text "ingredients"
```

When this action is called with both checkboxes checked `ingredients` will be set to `["milk", "egg"]`. When no checkbox is checked it will return an empty list.

Similiar to `param` this works out of the box for Ids, UUID, Bools, Timestamps, etc.

Passing Data from the Action to the View

A common task is to pass data from the action to the view for rendering. You can do this by extending the view data structure by the required data.

Given an action like this:

```
action ExampleAction = do
    render ExampleView { .. }
```

And an `ExampleView` like this:

```
data ExampleView = ExampleView { }

instance View ExampleView ViewContext where
    html ExampleView { .. } = [hsx|Hello World!|]
```

Now we want to pass the user's firstname to the view, to make the hello world a bit more personal. We can just do it by adding a `firstname :: Text` field to the `ExampleView` data structure and then adding a `{firstname}` to the HSX:

```
data ExampleView = ExampleView { firstname :: Text }

instance View ExampleView ViewContext where
    html ExampleView { .. } = [hsx|Hello World, {firstname}!|]
```

By now, the compiler will already tell us that we have not defined the `firstname` field inside the action. So we also need to update the action:

```
action ExampleAction = do
    let firstname = "Tester"
    render ExampleView { .. } -- Remember: ExampleView { .. } is just shor
```

This will pass the firstname `"Tester"` to our view.

We can also make it act more dynamically and allow the user to specify the firstname via a query parameter:

```
action ExampleAction = do
    let firstname = paramOrDefault @Text "Unnamed" "firstname"
    render ExampleView { .. }
```

This will render `Hello World, Unnamed!` when the `ExampleAction` is called without a `firstname` parameter.

Advanced: Working with Custom Types

Rarely you might want to work with a custom scalar value which is not yet supported with `param`. Define a custom `ParamReader` instance to be able to use the `param` functions with your custom value type. For that, take a look at the existing instances of `ParamReader`.

Records

When working with records, use `fill` instead of `param`. Fill automatically deals with validation failure when e.g. a field value needs to be an integer, but the submitted value is not numeric.

Here is an example of using `fill`:

```
action CreatePostAction = do
    let post = newRecord @Post
    post
        |> fill @'["title", "body"]
        |> ifValid \case
            Left post -> render NewView { .. }
            Right post -> do
                post <- post |> createRecord
                setSuccessMessage "Post created"
                redirectTo PostsAction
```

Lifecycle

The Controller instance provides a `beforeAction` function, which is called before the `action` function is called. Common request handling logic like authentication is often placed inside `beforeAction` to protect all actions of the controller against unprotected access.

Here is an example to illustrate the lifecycle:

```
instance Controller PostsController where
    beforeAction = putStrLn "A"
    action ShowPotsAction { postId } = putStrLn "B"
```

Calling the `ShowPostAction` will cause the following output to be logged to the server console:

```
A
B
```

Here is an example to illustrate authentication:

```
instance Controller PostsController where
    beforeAction = ensureIsUser
    action ShowPostAction { postId } = ...
```

Accessing the Current Action

Inside the `beforeAction` and `action` you can access the current action using the special `?theAction` variable. That is useful when writing controller helpers, because the variable is passed implicitly.

Accessing the Current Request

IHP uses the Haskell WAI standard for dealing with HTTP request and responses. You can get access to the Wai Request data structure by using `request`:

Take a look at the Wai documentation to see what you can do with the Wai `Request`:

```
action ExampleAction = do
    let requestBody = request |> getRequestBodyChunk
```

IHP provides a few shortcuts for commonly used request data:

```
action ExampleAction = do
    -- Use `getRequestPath` for accessing the current request path (e.g. /U
    putStrLn ("Current request url: " <> tshow getRequestPath)

    -- Use `getRequestPathAndQuery` for accessing the path with all parame
    putStrLn ("Current request url: " <> tshow getRequestPathAndQuery)

    -- Access the request body
    body <- getRequestBody

    -- Get a request
    let theRequest = request
```

Request Headers

Use `getHeader` to access a request header:

```
action ExampleAction = do
    let userAgent = getHeader "User-Agent"
```

In this example, when the `User-Agent` header is not provided by the request the `userAgent` variable will be set to `Nothing`. Otherwise it will be set to `Just "the user agent value"`.

The lookup for the header in the request is case insensitive.

Rendering Responses

Rendering Views

Inside a controller, you have several ways of sending a response. The most common way is to use the `render` function with a `View` data structure, like this:

```
render ShowPostView { .. }
```

The `render` function automatically picks the right response format based on the `Accept` header of the browser. It will try to send a html response when html is requested, and will also try to send a json response when a json response is expected. A `406 Not Acceptable` will be send when the `render` function cannot fullfil the requested `Accept` formats.

Rendering Plain Text

Call `renderPlain` to send a simple plain text response:

```
action ExampleAction = do
    renderPlain "My output text"
```

Rendering HTML

Usually you want to render your html using a view. See `Rendering Views` for details.

Sometimes you want to render html without using views, e.g. doing it inline in the action. Call `respondHtml` for that:

```
action ExampleAction = do
    respondHtml [hsx|<div>Hello World</div>|]
```

You will need to import `hsx` into your controller: `import IHP.ViewPrelude (hsx)`.

Rendering a Static File

Use `renderFile path contentType` to respond with a static file:

```
action ExampleAction = do
    renderFile "static/terms.pdf" "application/pdf"
```

Rendering a Not Found Message

Use `renderNotFound` to render a generic not found message, e.g. when a entity cannot be found:

```
action ExampleAction = do
    renderNotFound
```

Redirects

Redirect to an Action

Use `redirectTo` to redirect to an action:

```
action ExampleAction = do
    redirectTo ShowPostAction { postId = ... }
```

When you need to pass a custom query parameter, you cannot use the `redirectTo` function. See `Redirect to a Path` for that.

The redirect will use http status code `302`. The `baseUrl` in `Config/Config.hs` will be used. In development mode, the `baseUrl` might not be specified in `Config/Config.hs`. Then it will be set to localhost by default.

Redirect to a Path

Use `redirectToPath` when you want to redirect to a path on the same domain:

```
action ExampleAction = do
    redirectToPath "/blog/wp-login.php"
```

This can also be used to specify query parameter for actions:

```
action ExampleAction = do
    redirectToPath ((pathTo ShowPostAction { .. }) <> "&details=on")
```

Redirect to a Url

Use `redirectToUrl` to redirect to some external url:

```
action ExampleAction = do
    redirectToUrl "https://example.com/hello-world.html"
```

Action Execution

When calling a function to send the response, IHP will stop execution of the action. Internally this is implemented by throwing and catching a `ResponseException`. Any code after e.g. a `render SomeView { .. }` call will not be called. This also applies to all redirect helpers.

Here is an example of this behavior:

```
action ExampleAction = do
    redirectTo SomeOtherAction
    putStrLn "This line here is not reachable"
```

The `putStrLn` will never be called because the `redirectTo` already stops execution.

When you have created a `Response` manually, you can use `respondAndExit` to send your response and stop action execution.

Request Context

Actions have access to the special variable `?requestContext`.

The `?requestContext` provides access to the Wai request as well as infos like the request query and post params and the uploaded files. It's usually used by other functions to provide high level functionality. E.g. the `getHeader` function uses the `?requestContext` to access the request headers.

Controller Context

TODO

File Uploads

TODO

View

Introduction

IHP views are usually represented as HTML, but can also be represented as json or other formats.

The html templating is implemented on top of the well-known blaze-html Haskell library. To quickly build html views, IHP supports a JSX-like syntax called HSX. HSX is type-checked and compiled to Haskell code at compile-time.

To render a view, it has to be provided with a custom data structure called a ViewContext. A ViewContext provides the view with information it might need to render, without always explicitly passing it. This is usually used to pass e.g. the current http request, current logged in user, flash messages, the layout, etc..

Usually a view consist of a data structure and a `View` instance. E.g. like this:

```
data ExampleView = ExampleView { optionA :: Text, optionB :: Bool }

instance View ExampleView ViewContext where
    html ExampleView { .. } = [hsx|Hello World {optionA}!|]
```

Layouts

By default when rendering a HTML view, IHP uses the default application layout to render your view. It's defined at `defaultLayout` in `Web.View.Layout`.

A layout is just a function taking a view and returning a new view:

```
type Layout = Html -> Html
```

Adding a new layout

To add a new layout, add a new function to the `Web.View.Layout`:

```
appLayout :: Layout
appLayout inner = H.docTypeHtml ! A.lang "en" $ [hsx|
<head>
    <title>My App</title>
</head>
<body>
    <h1>Welcome to my app</h1>
    {inner}
</body>
|]
```

Now add `appLayout` to the export list of the module header:

```
module Web.View.Layout (defaultLayout, appLayout) where
```

Using a layout

To use the layout inside a view, set the layout attribute of the view context inside your view:

```
instance View MyView ViewContext where
    beforeRender (context, view) = (context { layout = appLayout }, view)
```

Changing the default layout

You can change the default layout of your application by updating `createViewContext` in `Web.View.Context`.

```
instance ViewSupport.CreateViewContext ViewContext where
    -- ...
    createViewContext = do
        -- ...
        let viewContext = ViewContext {
                -- ...
                layout = let ?viewContext = viewContext in appLayout
            }
        pure viewContext
```

Disabling the Layout for a View

You can disable the layout for a specific view by overriding the `beforeRender` function like this:

```
instance View MyView ViewContext where
    beforeRender (context, view) = (context { layout = \view -> view }, vi

    -- ...
```

Common View Tasks

Accessing the Request

Use `theRequest` to access the current WAI request.

Highlighting the current active link

Use `isActivePath` to check whether the current request url matches a given action.

```
<a href={ShowProjectAction} class={classes ["nav-link", ("active", isActive
    Show Project
</a>
```

Check whether this view is called from a specific controller

Use `isActiveController`.

`timeAgo`

```
timeAgo (get #createdAt post) -- "1 minute ago"
```

`dateTime`

```
dateTime (get #createdAt post) -- "10.6.2019, 15:58"
```

Diff-Based DOM Updates

When in development, your views will automatically refresh on code changes. This works by re-requesting the view from the server via AJAX and then using morphdom to update the visible DOM.

TurboLinks

In production mode your application is using a custom integration of morphdom and TurboLinks together with InstantClick. TurboLinks makes navigating the application even faster because it's not doing a full page refresh. We've integrated TurboLinks with morphdom to only update the parts of your HTML that have actually changed. This was inspired by react.js's DOM patch approach and allows for e.g. CSS animations to run on a page transition. Using this makes your app feel like a SPA without you writing any javascript code.

To improve latency, TurboLinks is configured to prefetch the URL immediately on mouse-hover. Usually the time between a mouse-hover of a link and mouse click is 100ms - 200ms. As long as the server responds in less than 100ms, the response is already there when the click event is fired. This makes your app faster than most single page application (most SPAs still need to fetch some data after clicking).

This setup is designed as a progressive enhancement. Your application is still usable when javascript is disabled. Even when disabled, your application will still be amazingly fast.

You can disable this behavior by removing the following code from your `Web/Layout.hs`:

```
    when (isProduction FrameworkConfig.environment) [hsx|
            <script src="/vendor/turbolinks.js"></script>
            <script src="/vendor/morphdom-umd.min.js"></script>
            <script src="/vendor/turbolinksMorphdom.js"></script>
            <script src="/vendor/turbolinksInstantClick.js"></script>
        |]
```

JSON

Views that are rendered by calling the `render` function can also respond with JSON.

Let's say we have a normal HTML view that renders all posts for our blog app:

```
instance View IndexView ViewContext where
    html IndexView { .. } = [hsx|
        <nav>
            <ol class="breadcrumb">
                <li class="breadcrumb-item active"><a href={PostsAction}>Po
            </ol>
        </nav>
        <h1>Index <a href={pathTo NewPostAction} class="btn btn-primary ml
        <div class="table-responsive">
            <table class="table">
                <thead>
                    <tr>
                        <th>Post</th>
                        <th></th>
                        <th></th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>{forEach posts renderPost}</tbody>
            </table>
        </div>
    |]
```

We can add a JSON output for all blog posts by add a `json` function to this:

```
import Data.Aeson -- <--- Add this import at the top of the file

instance View IndexView ViewContext where
    html IndexView { .. } = [hsx|
        ...
    |]

    json IndexView { .. } = toJSON posts -- <---- The new json render funct
```

In the above code, our `json` function has access to all arguments passed to the view. Here we call `toJSON`, which is provided by the aeson haskell library. This simply encodes all the `posts` given to this view as json.

Additionally we need to define a `ToJSON` instance which describes how the `Post` record is going to be transformed to json. We need to add this to our view:

```
instance ToJSON Post where
    toJSON post = object
        [ "id" .= get #id post
        , "title" .= get #title post
        , "body" .= get #body post
        ]
```

The full `Index` View for our `PostsController` looks like this:

```
module Web.View.Posts.Index where
import Web.View.Prelude
```

```
import Data.Aeson

data IndexView = IndexView { posts :: [Post] }

instance View IndexView ViewContext where
    html IndexView { .. } = [hsx|
        <nav>
            <ol class="breadcrumb">
                <li class="breadcrumb-item active"><a href={PostsAction}>Po
            </ol>
        </nav>
        <h1>Index <a href={pathTo NewPostAction} class="btn btn-primary ml-
        <div class="table-responsive">
            <table class="table">
                <thead>
                    <tr>
                        <th>Post</th>
                        <th></th>
                        <th></th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>{forEach posts renderPost}</tbody>
            </table>
        </div>
    |]

    json IndexView { .. } = toJSON posts

instance ToJSON Post where
    toJSON post = object
        [ "id" .= get #id post
```

```
        , "title" .= get #title post
        , "body" .= get #body post
        ]

renderPost post = [hsx|
    <tr>
        <td>{post}</td>
        <td><a href={ShowPostAction (get #id post)}>Show</a></td>
        <td><a href={EditPostAction (get #id post)} class="text-muted">Edi
        <td><a href={DeletePostAction (get #id post)} class="js-delete tex
    </tr>
|]
```

Getting JSON responses

When you open the `PostsAction` at `/Posts` in your browser you will still get the html output. This is because IHP uses the browsers `Accept` header to respond in the best format for the browser which is usually html.

Javascript

From javascript you can get the JSON using `fetch`:

```
const response = await fetch('http://localhost:8000/Posts', { headers: { Ac
    .then(response => response.json());
```

curl

You can use `curl` to check out the new JSON response from the terminal:

```
curl http://localhost:8000/Posts -H 'Accept: application/json'

[{"body":"This is a test json post","id":"d559cd60-e36e-40ef-b69a-d651e325
```

Advanced: Rendering JSON directly from actions

When you are building an API and your action is only responding with JSON (so no html is expected), you can respond with your JSON directly from the controller using `renderJson`:

```
instance Controller PostsController where
    action PostsAction = do
        posts <- query @Post |> fetch
        renderJson (toJSON posts)

-- The ToJSON instances still needs to be defined somewhere
instance ToJSON Post where
    toJSON post = object
        [ "id" .= get #id post
        , "title" .= get #title post
```

```
        , "body" .= get #body post
        ]
```

In this example no content negotiation takes place as the `renderJson` is used instead of the normal `render` function.

The `ToJSON` instances has to be defined somewhere, so it's usually placed inside the controller file. This often makes the file harder to read. We recommend to not use `renderJson` most times and instead stick with a separate view file as described in the section above. Using `renderJson` makes sense only when the controller is very small or you already have a predefined `ToJSON` instance which is not defined in your controller.

HSX

Introduction

Introduction

HSX can be written pretty much like normal HTML. You can write a HSX expression inside your Haskell code by wrapping it with `[hsx|YOUR HSX CODE|]`. HSX expressions are just a

syntax for blaze html and thus are automatically escaped as described in the blaze documentation.

Because the HSX is parsed, you will get a syntax error when you type in invalid html.

Inline Haskell

HSX can access Haskell variables wrapped with `{}` like this:

```
let
    x :: Text = "World"
in
    [hsx|Hello {x}!|]
```

If the variable is another hsx expression, a blaze html element, a text or string: it is just included as you would expect.

If the variable is any other custom Haskell data structure: it will first be converted to a string representation by calling `show` on it. You can add a custom `ToHtml` (import it from `IHP.HtmlSupport.ToHtml`) instance, to customize rendering a data structure.

You can also write more complex code like:

```
let
    items :: [Int] = [ 0, 1, 2 ]
    renderItem n = [hsx|Hello {n}!|]
```

```
in
    [hsx|Complex demo: {forEach items renderItem}!|]
```

As the HSX expressions are compiled to Haskell code at compile-time, type errors inside these `{}` expressions will be reported to you by the compiler.

Dynamic Attributes

The variable syntax can also be used in attribute values:

```
let
    inputValue = "Hello World" :: Text
in
    [hsx|<input type="text" value={inputValue}/>|]
```

Boolean Attribute Values

HSX has special handling for boolean values to make it easy to deal with html boolean attributes like `disabled`, `readonly`, `checked`, etc.

You can write

```
<input disabled={True}/>
```

as a short form for:

```
<input disabled="disabled"/>
```

Writing `False`:

```
<input disabled={False}/>
```

This will not render the attribute:

```
<input/>
```

Spread Values

For dynamic use cases you can use `{...attributeList}`:

```
<div { ...[ ("data-my-attribute", "Hello World!") ] } />
<div { ...[ ("data-user-" <> tshow userId, tshow userFirstname) ] } />
<div { ...someVariable } />
```

Special Elements: `<script>` and `<style>`

For `<script>` and `<style>` tags HSX applies some special handling. This only applies to tags with inline scripts or styles:

```
No Special Handling:

<script src="/hello.js"/>
<link rel="stylesheet" href="layout.css"/>

Special Handling applies:

<script>alert("Hello");</script>
<style>h1 { color: blue; }</style>
```

Inside those tags using a haskell expression will not work:

```
<script>{myHaskellExpr}</script>
```

This will just literally output the string `{myHaskellExpr}` without evaluating the haskell expression itself. This is because javascript usually uses `{}` for object expressions like `{ a: "hello" }`. The same applies to inline CSS inside `<style>` elements.

So using `{haskellVariables}` inside your javascript like this will not work:

```
<script>
    var apiKey = "{apiKey}";
```

```
</script>
```

Instead use a `data-` attribute to solve this:

```
<script data-api-key={apiKey}>
    var apiKey = document.currentScript.dataset.apiKey;
</script>
```

Additionally HSX will not do the the usual escaping for style and script bodies, as this will make e.g. the javascript unusuable.

Syntax Rules

While most html is also valid HSX, there are some difference you need to be aware of:

Closing Tags

Tags always need to have a closing tag or be self-closing: so instead of `<br>` you need to write `<br/>`

JSX Differences

In JSX you have the restriction that you always have to have a single root tag. In HSX this restriction does not apply, so this is valid HSX (but not valid JSX):

```
[hsx|
<div>A</div>
<div>B</div>
|]
```

Whitespace

Spaces and newline characters are removed where possible at HSX parse time.

Comments

Html Comments are supported and can be used like this:

```
<div>
    <!-- Begin of Main Section -->
    <h1>Hello</h1>
</div>
```

Empty Attributes

HSX allows you to write empty attributes like these:

```
[hsx|
    <input disabled/>
|]
```

The underlying html library blaze currently does not support empty html attribute. Therefore empty attributes are implemented by setting the attribute value to the attribute name. This is valid html supported by all browsers.. Therefore the generated html looks like this:

```
<input disabled="disabled"/>
```

Form

Introduction

In IHP Forms are an essential way to interact with your application. Dealing with a lot of form markup can quickly become complex because of the need to deal with consistent styling and especially when dealing with lots of validation. IHP provides helpers to generate form markup to help you deal with the complexity.

By default forms in IHP follow the class names used by Bootstrap 4. Therefore the forms work with Bootstrap 4 out of the box. Of course the default form generation can be customized to support other CSS frameworks.

Unless javascript helpers have been deactivated, your form will be submitted using AJAX and TurboLinks instead of a real browser based form submission.

Simple Forms

Forms usually begin with a `formFor` expression. This is how a simple form can look like:

```
renderForm :: Post -> Html
renderForm post = formFor post [hsx|
    {textField #title}
    {textareaField #body}
    {submitButton}
|]
```

Calling this form from inside your HSX code will lead to the following HTML being generated:

```
<form method="POST" action="/CreatePost" id="" class="new-form">
    <div class="form-group" id="form-group-post_title">
        <label for="post_title">Title</label>
        <input type="text" name="title" id="post_title" class="form-contro
    </div>

    <div class="form-group" id="form-group-post_body">
        <label for="post_body">Body</label>
        <textarea name="body" id="post_body" class="form-control"></textar
    </div>

    <button class="btn btn-primary">Create Post</button>
</form>
```

You can see that the form is submitted via `POST`. The form action has also been set by default to `/CreatePost`.

All inputs have auto-generated class names and ids for styling. Also all `name` attributes are set as expected.

Form Controls

IHP has the most commonly-used form controls built in. In general the form control helpers just need to be passed the field name. Here is a list of all built-in form control helpers:

```
{textField #title}
{textareaField #body}
{colorField #brandColor}
{emailField #email}
{dateField #dueAt}
{passwordField #password}
{dateTimeField #createdAt}
{hiddenField #projectId}
{checkboxField #termsAccepted}
{selectField #projectId allProjects}
{submitButton}
```

A form control is always filled with the value of the given field when rendering. For example given a post

```
let post = Post { ..., title = "Hello World" }
```

Rendering `{textField #title}`, the input value will be set like

```
<input ... value="Hello World">
```

Validation

When rendering a record which has failed validation, the validation error message will be rendered automatically.

Given a post like this:

```
let post = Post { ..., title = "" }
    |> validateField #title nonEmpty
```

Rendering `{textField #title}`, the input will have the css class `is-invalid` and an element with the error message will be rendered below the input:

```
<div class="form-group" id="form-group-post_title">
    <label for="post_title">Title</label>
    <input type="text" name="title" placeholder="" id="post_title" class="
    <div class="invalid-feedback">This field cannot be empty</div>
</div>
```

Forms Are Also HSX

It's important to understand that while the form helpers like `{textField #title}` are called by `formFor`, you can still use HSX there. So you can just add any kind of HSX code inside your form:

```
renderForm :: Post -> Html
renderForm post = formFor post [hsx|
    <h1>Add a new post</h1>

    <div class="row">
        <div class="col">
            {textField #title}
        </div>

        <div class="col">
            Specify a title at the left text field
        </div>
    </div>

    {textareaField #body}

    <div style="background: blue">
        {submitButton}
    </div>
|]
```

Inside the HSX block of a form, you have access to the special `?formContext` variable. This variable keeps track of e.g. the current record (`post` in the above example), the form action,

as well as some other options. See the API Documentation on `FormContext` to learn more.

Customizing Inputs

The return values of the form control helpers are usually a value of type FormField. The `FormField` value is automatically rendered as HTML when used inside an HSX expression. Before this rendering happens, you can specifiy options to customize the rendering.

Help Texts

You can add a help text below a form control like this:

```
{(textField #title) { helpText = "Max. 140 characters"} }
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label for="post_title">Title</label>

    <input type="text" name="title" id="post_title" class="form-control">
    <small class="form-text text-muted">Max. 140 characters</small>
</div>
```

Custom Field Label Text

By default the field name will be used as a label text. The camel case field name will be made more human readable of course, so `contactName` will turn to `Contact Name`, etc. Sometimes you want to change this auto-generated input label to something custom. Use `fieldLabel` for that, like this:

```
{(textField #title) { fieldLabel = "Post Title"} }
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label for="post_title">Post Title</label>
    <input type="text" name="title" id="post_title" class="form-control">
</div>
```

Custom CSS Classes

You can add custom CSS classes to the input and label for better styling. Set `fieldClass` for adding a class to the input element and `labelClass` for the label element:

```
{(textField #title) { fieldClass="title-input", labelClass = "title-label"
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label class="title-label" for="post_title">Title</label>
    <input type="text" name="title" id="post_title" class="form-control ti
</div>
```

Of course, the CSS classes for validation are still set as expected.

Placeholder

You can specify an input placeholder like this:

```
{(textField #title) { placeholder = "Enter your title ..." } }
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label for="post_title">Title</label>

    <input type="text" name="title" id="post_title" placeholder="Enter you
</div>
```

Required Fields

You can mark an input as required like this:

```
{(textField #title) { required = True } }
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label for="post_title">Title</label>

    <input type="text" name="title" id="post_title" required="required" cla
</div>
```

Custom Submit Button Text

Customize it like this:

```
{submitButton { label = "Create it!" } }
```

This will render like:

```
<button class="btn btn-primary">Create it!</button>
```

When you want to use e.g. an icon inside your button, it might be easier to just write the HTML manually by hand then.

Custom Submit Button Class

Customize it like this:

```
{submitButton { buttonClass = "create-button" } }
```

This will render like:

```
<button class="btn btn-primary create-button">Create Post</button>
```

Advanced Customization Options

The following options are not commonly used, but are useful sometimes.

Adding custom attributes to the input element

Form rendering is built on top of blaze html. So you need to import the blaze html functions for this. Add this at the top of your module:

```
import qualified Text.Blaze.Html5 as H
import qualified Text.Blaze.Html5.Attributes as A
```

```
{(textField #title) { fieldInput = (\fieldInput -> H.input ! A.onclick "al
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label class="" for="post_title">Title</label>
    <input onclick="alert(1)" type="text" name="title" placeholder="" id="
</div>
```

Using the `fieldInput`, which is passed as an argument, you can access the other options of the form. Don't set the `class` attribute on your custom field input, as this will break rendering.

Custom name attribute

By default the field name is used for the `name` attribute of the input element. You can override it like this:

```
{(textField #title) { fieldName = "new-title" } }
```

This will render like:

```
<div class="form-group" id="form-group-post_title">
    <label for="post_title">Title</label>

    <input type="text" name="new-title" id="post_title" class="form-contro
</div>
```

Custom id attribute

You can override the auto-generated id value like this:

```
{(textField #title) { fieldInputId = "the-title-form-group" } }
```

This will render like:

```
<div class="form-group" id="the-title-form-group">
    <label for="post_title">Title</label>

    <input type="text" name="new-title" id="post_title" class="form-contro
</div>
```

Don’t render `<label>`

You can specifiy `disableLabel` to stop the label element from being generated:

```
{(textField #title) { disableLabel = True }
```

Will render as:

```
<div class="form-group" id="form-group-post_title">
    <input type="text" name="title" placeholder="" id="post_title" class="
</div>
```

Don’t render `<div class="form-group">`

You can specifiy `disableGroup` to stop the outer `<div class="form-group">` element from being generated:

```
{(textField #title) { disableGroup = True }
```

Will render as:

```
<input type="text" name="title" placeholder="" id="post_title" class="form
<label for="post_title">Title</label>
```

Don’t show validation error message

You can specify `disableValidationResult` to stop the validation error message being shown when the validation failed:

```
{(textField #title) { disableValidationResult = True }
```

This works out of the box for most Haskell data types. When you are working with a custom data type, e.g. a custom enum value, you need to add a `InputValue MyDataType` implementation. We will cover this later in this Guide.

Select Inputs

Select inputs require you to pass a list of possible values to select.

You can use the `selectField` helper for select inputs:

```
formFor project [hsx|
    {selectField #userId users}
```

```
|]
```

In the example above the variable `users` contains all the possible option values for the select.

You also need to define a instance `CanSelect User`:

```
instance CanSelect User where
    -- Here we specify that the <option>-value should contain a UserId
    type SelectValue User = UserId
    -- Here we specify how to transform the model into <option>-value
    selectValue = get #id
    -- And here we specify the <option>-text
    selectLabel = get #name
```

Given the above example, the rendered form will look like this:

```
-- Assuming: users = [User { id = 1, name = "Marc" }, User { id = 2, name =
<form ...>
    <select name="user_id">
        <option value="1">Marc</option>
        <option value="2">Andreas</option>
    </select>
</form>
```

Select Inputs with Custom Enums

You can use select fields with custom defined enums too.

Given an enum like this:

```
CREATE TYPE CONTENT_TYPE AS ENUM ('video', 'article', 'audio');
```

We need to define a `CanSelect ContentType` like this:

```
instance CanSelect ContentType where
    type SelectValue ContentType = ContentType
    selectValue value = value

    selectLabel Video = "Video"
    selectLabel Article = "Article"
    selectLabel Audio = "Audio"
    -- You can also use the following shortcut: selectLabel = tshow
```

Advanced Forms

You can get very far with the built-in form helpers. But sometimes you might need a very custom functionality which is not easily doable with the form helpers. In this case we highly recommend to not use the form helpers for that specific case. Don't fight the tools.

CSRF

IHP by default sets its session cookies using the Lax SameSite option. While `Lax` sounds not very secure, this protects against all common CSRF vectors. This browser-based CSRF protection works with all modern browsers, therefore a token-based protection is not used in IHP applications.

Javascript Helpers

By default your form will be submitted using AJAX and TurboLinks instead of a real browser based form submission. It's implemented this way to support SPA-like page transitions using TurboLinks and morphdom.

Additionally to integrate the form submission into TurboLinks, the javascript helpers will also disable the form submit button after the form has been submitted. Also any flash messages inside the form are removed.

When the IHP javascript helpers are included in a page, it will automatically hook into your form submissions. You can also call `window.submitForm(formElement)` to trigger a form submission from javascript.

The form helpers are designed to improve the User Experience for browsers where javascript is enabled. In case javascript is not enabled or blocked by a plugin, the form submission will still work as expected.

You can disable the form helpers by removing the IHP javascript helpers from your layout. In `Web/View/Layout.hs` remove the following line:

```
<script src="/helpers.js"></script>
```

This way no special behavior will be attached to your forms.

To dig deeper into the javascript, take a look at the source in helpers.js.

Working with other CSS Frameworks

TODO: This section still has to be implemented. The gist of how rendering can be completely overriden to support a different layout or CSS framework can be found in the implementation of horizontalFormFor (renders a bootstrap 4 form in a horizontal way).

Validation

Introduction

IHP provides a simple infrastructure for validation of incoming data. This guide will tell you more about validating new and existing records, as well as how to use more complex validations with database access.

Quickstart

Setting up the controller

Let's asume we have generated a `Posts` controller using the code generator. Our `Post` has a `title` and a `body`. The `CreatePostAction` looks like this:

```
action CreatePostAction = do
    let post = newRecord @Post
    post
        |> buildPost
        |> ifValid \case
            Left post -> render NewView { .. }
            Right post -> do
                post <- post |> createRecord
                setSuccessMessage "Post created"
                redirectTo PostsAction
```

This action is executed when a form like below is submitted:

```
module Web.View.Posts.New where
import Web.View.Prelude

data NewView = NewView { post :: Post }

instance View NewView ViewContext where
    html NewView { .. } = [hsx|
        <h1>New Post</h1>
        {renderForm post}
    |]

renderForm :: Post -> Html
renderForm post = formFor post [hsx|
    {textField #title}
    {textField #body}
    {submitButton}
|]
```

Now let's add some validation.

Adding Validation Logic

To make sure that the `title` and `body` are not empty, we can `validateField ... nonEmpty` like this:

```
    action CreatePostAction = do
        let post = newRecord @Post
```

```
        post
            |> buildPost
            |> validateField #title nonEmpty
            |> validateField #body nonEmpty
            |> ifValid \case
                Left post -> render NewView { post }
                Right post -> do
                    post <- post |> createRecord
                    setSuccessMessage "Post created"
                    redirectTo PostsAction

buildPost post = post
    |> fill @'["title", "body"]
```

The syntax to validate a record is always like this:

```
record
    |> validateField #fieldName validatorName
```

Common Validators

Here is a list of the most common validators:

```
-- works with Text fields
|> validateField #name nonEmpty
|> validateField #email isEma
```

```
  |> validateField #phoneNumber isPhoneNumber

  -- works with ints
  |> validateField #rating (isInRange 1 10)
```

You can find the full list of built-in validators in the API Documentation.

Fill Validation

When using `fill`, like `|> fill @'["title", "body"]`, any error parsing the input is also added as a validation error.

E.g. when `fill` fills in an integer attribute, but the string `"hello"` is submitted, an error will be added to the record and the record is not valid anymore. The record attribute will then keep its old value (before applying `fill`) and later re-render in the error case of `ifValid`. This applies to all arguments read via `fill`. It's very helpful when dealing with strings expected in a certain format, like date times.

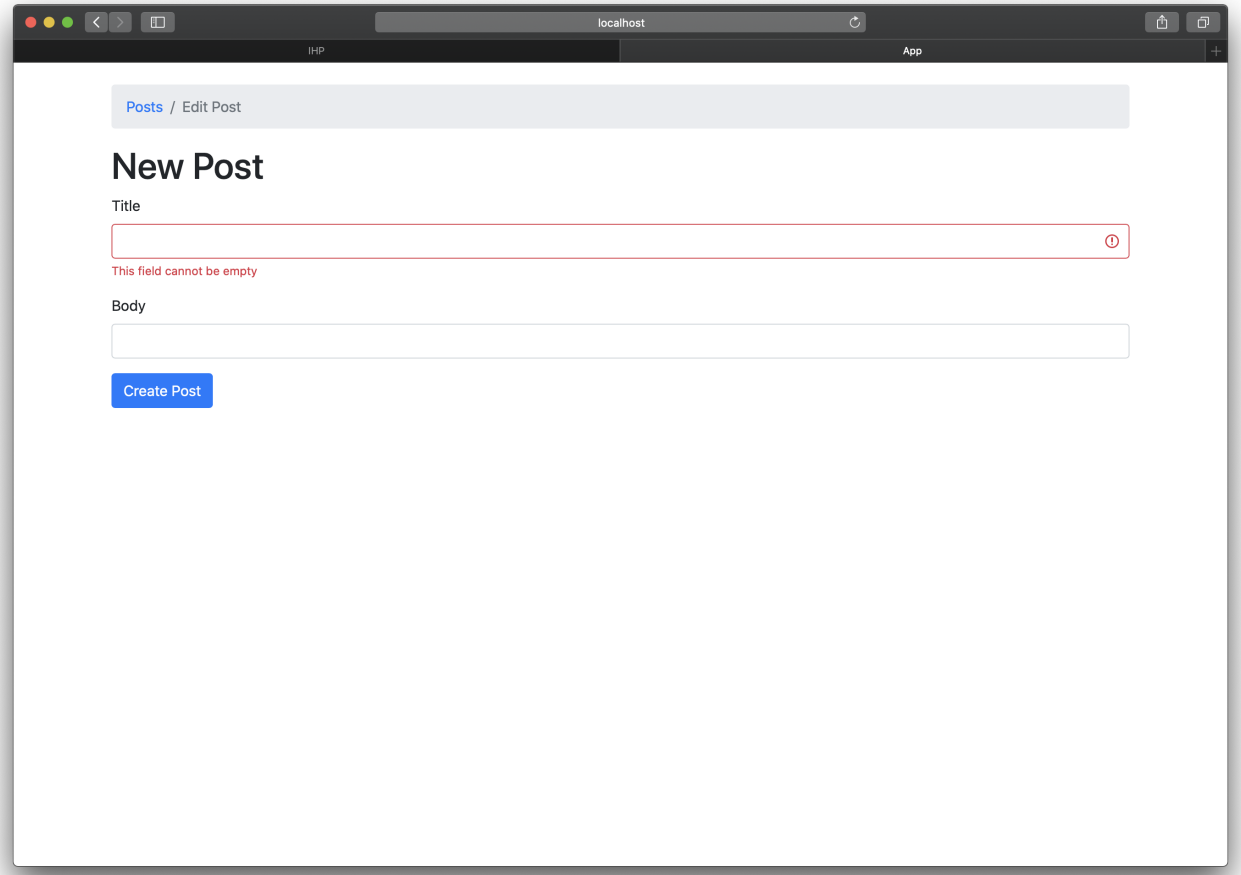
As IHP is never putting raw strings into the records, without previously parsing them, it does not provide validators like rails's `:only_integer`.

Rendering Errors

When a post with an empty title is now submitted to this action, an error message will be written into the record. The call to `|> ifValid` will see that there is an error, and then run the

`Left post -> render NewView { post }`, which displays the form to the user again. The form will be rendered, and errors will automatically pop up next to the form field.

The default form helpers like `{textField #title}` automatically render the error message below the field. Here is how this looks:



Validating An Email Is Unique

For example when dealing with users, you usually want to make sure that an email is only used once for a single user. You can use `|> validateIsUnique #email` to validate that an email is unique for a given record.

This function queries the database and checks whether there exists a record with the same email value. The function ignores the current entity of course.

This function does IO, so any further arrows have to be `>>=`, like this:

```
action CreateUserAction = do
    let user = newRecord @User
    user
        |> fill @'["email"]
        |> validateIsUnique #email
        >>= ifValid \case
            Left user -> render NewView { .. }
            Right user -> do
                createRecord user
                redirectTo UsersAction
```

Sharing Between Create and Update Action

Usually you have a lot of the same validation logic when creating and updating a record. To avoid duplicating the validation rules, you can apply them inside the `buildPost` function. This function is used by the create as well as the update action to read in the form values.

Here is how this can look:

```
    action CreatePostAction = do
        let post = newRecord @Post
        post
            |> buildPost
            -- <------------ Here we removed the `validateField ...`
            |> ifValid \case
                Left post -> render NewView { post }
                Right post -> do
                    post <- post |> createRecord
                    setSuccessMessage "Post created"
                    redirectTo PostsAction

buildPost post = post
    |> fill @'["title", "body"]
    |> validateField #title nonEmpty -- <---------- Here we added them
    |> validateField #body nonEmpty
```

In case a validation should only be used for e.g. updating a record or creating a record, just keep it there in the action only and don't move it to the `buildPost` function.

Creating a custom validator

You can just write your own constraint like this:

```
nonEmpty :: Text -> ValidatorResult
nonEmpty "" = Failure "This field cannot be empty"
nonEmpty _ = Success

isAge :: Int -> ValidatorResult
isAge = isInRange (0, 100)
```

Then just call it like:

```
user |> validateField #age isAge`
```

Checking If A Record Is Valid

Use `ifValid` to check for validity of a record:

```
post |> ifValid \case
    Left post -> do
        putStrLn "The Post is invalid"
    Right post -> do
        putStrLn "The Post is valid"
```

This will call the `Left post -> do ...` block when the record is not valid.

You can also use it like this:

```
let message = post |> ifValid \case
    Left post -> "The post is invalid"
    Right post -> "The post is valid"

putStrLn message
```

Customizing Error Messages

Use `withCustomErrorMessage` to customize the error message when validation failed:

```
user
    |> fill @'["firstname"]
    |> validateField #firstname (nonEmpty |> withCustomErrorMessage "Pleas
```

In this example, when the `nonEmpty` adds an error to the user, the message `Please enter your firstname` will be used instead of the default `This field cannot be empty`.

Internals

IHP's validation is built with a few small operations.

validateField

The primary operation is `validateField #field validationFunction record`.

This function does the follwing thing:

1. Read the `#field` from the record
2. Apply the `validationFunction` to the field value
3. When the validator returns an errors, store the errors inside the `meta` attribute of the record.

The `validateField` function expects the `record` to have a field `meta :: MetaBag`. This `meta` field is used to store validation errors.

Let's say we have an example data type `Post`:

```
data Post = Post { title :: Text, meta :: MetaBag }
```

A call to validateField will result in the following:

```
let post = Post { title = "" , meta = def } -- def stands for default :)

post |> validateField #title nonEmpty
-- This will return:
--
-- Post {
--     title = "",
--     meta = MetaBag {
--         annotations = [
```

```
--             ("title", "This field cannot be empty")
--         ]
--     }
-- }
```

As you can see, the errors are tracked inside the `MetaBag`. When you apply another `validateField` to the record, the errors will be appended to the `annotations` list.

Validation Functions

A validation function is just a function which, given a value, returns `Success` or `Failure "some error message"`.

Here is an example:

```
isColor :: Text -> ValidatorResult
isColor text | ("#" `isPrefixOf` text) && (length text == 7) = Success
isColor text = Failure "is not a valid color"
```

Calling `isColor "#ffffff"` will return `Success`. Calling `isColor "something bad"` will result in `Failure "is not a valid color"`.

It might be useful to take a look at the definition of some more validation functions to see how it works. You can find them in the API Docs.

Attaching Errors To A Record Field

You can attach errors to a specific field of a record even when not validating. These errors will then also show up when rendering a form.

Here is an example:

```
post
    |> attachFailure #title "This error will show up"
```

This record now has a validation error attached to its title.

Retrieving The First Error Message For A Field

You can also access an error for a specific field using `getValidationFailure`:

```
post
    |> validateField #name nonEmpty
    |> getValidationFailure #name
```

This returns `Just "Field cannot be empty"` or `Nothing` when the post has a title.

Retrieving All Error Messages For A Record

Access them from the `meta :: MetaBag` attribute like this:

```
record
    |> get #meta
    |> #annotations
```

This returns a `[(Text, Text)]`, e.g. `[("name", "This field cannot be empty")]`.

Session

Introduction

The session provides a way for your application to store small amounts of information that will be persisted between requests. It's mainly used from inside your controller actions.

In general you should not store complex data structures in the session. It's better to store scalar values in there only. For example: Store the current user id instead of the current user record.

The session works by storing the data inside a cryptographically signed and encrypted cookie on the client. The encryption key is generated automatically and is stored at `Config/client_session_key.aes`. Internally IHP uses the clientsession library. You can find more technical details on the implementation in the clientsession documentation.

Accessing the Session

Writing

In your controller action, use `setSession` to store a value:

```
action SessionExampleAction = do
    setSession "userEmail" "hi@digitallyinduced.com"
```

Right now, `setSession` only accepts `Text` values. Other types like `Int` have to be converted to `Text` using `tshow theIntValue`.

Reading

You can use `getSession` to retrieve a value:

```
action SessionExampleAction = do
    userEmail <- getSession "userEmail"
```

`userEmail` is set to `Just "hi@digitallyinduced.com"` when the value has been set before. Otherwise it will be `Nothing`.

For convience you can use `getSessionInt` to retrieve the value as a `Maybe Int`, and `getSessionUUID` to retrieve the value as a `Maybe UUID`:

```
action SessionExampleAction = do
    counter :: Maybe Int <- getSessionInt "counter"
    userId :: Maybe UUID <- getSessionUUID "userId"
```

Deleting

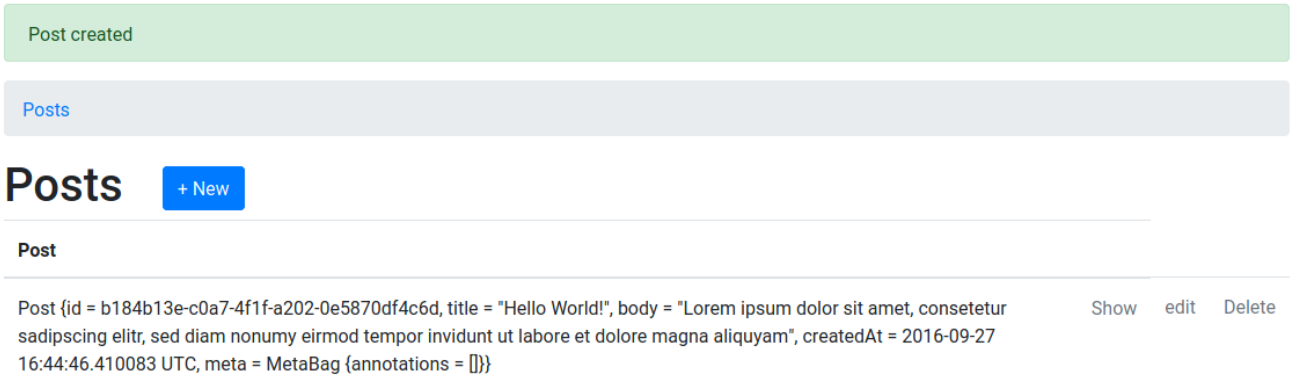
Set a value to an empty Text to remove it from the session:

```
action LogoutAction = do
    setSession "userId" ""
```

Flash Messages

Flash Messages provide a way to show success or error messages on the next request. After setting a success or error message, the message is only available on the next view rendered by the browser.

In the following view you can see a success flash message `Post created`:



.

Setting a Message

Use `setSuccessMessage` to set a success message:

```
action CreatePostAction = do
    ...
    setSuccessMessage "Your Post has been created succesfully"
    redirectTo ShowPostAction { .. }
```

Use `setErrorMessage` to set a failure message:

```
action CreatePostAction = do
    unless isAllowedToPost do
```

```
        setErrorMessage "You don't have the required permissions to create
        redirectTo NewPostAction

    ...
```

In both cases the messages are stored inside the session. The message value is automatically cleared after the next request (except redirects or when sending a JSON response).

Rendering a Message

You have to make sure that `{renderFlashMessages}` is displayed somewhere in your layout or view, otherwise the flash message is not visible.

Here is an example of a layout calling `renderFlashMessages`:

```
defaultLayout :: Html -> Html
defaultLayout inner = H.docTypeHtml ! A.lang "de" $ [hsx|
<head></head>
<body>
    <div class="container mt-2">
        {renderFlashMessages}
        {inner}
    </div>
</body>
|]
```

The rendered HTML looks like this:

```
<div class="alert alert-success">{successMessage}</div>
<div class="alert alert-danger">{errorMessage}</div>
```

To display the Flash Messages in custom way, you can always access them using `viewContext |> #flashMessages` in your views. This returns a list of `FlashMessage`. You can also take a look at the `renderFlashMessages` implementation and copy the code into your view, and then make customizations.

Session Cookie

The cookie max age is set to 30 days by default. To protect against CSRF, the SameSite Policy is set to Lax.

Scripts

Introduction

Scripts provide a way to run simple scripts inside the framework context, but outside of the usual web request response lifecycle.

Common use-cases include:

Sending periodic email reminders
Sending invoices
Anything to be run as a cronjob
Background job-queue processing

Creating a new script

Scripts are located in the `Application/Script/` directory. You can create a new script by running e.g. `new-script HelloWorldToAllUsers`. This will create a file at `Application/Script/HelloWorldToAllUsers.hs` like this:

```
#!/usr/bin/env run-script
module Application.Script.HelloWorldToAllUsers where

import Application.Script.Prelude

run :: Script
run = do
```

The `run` function is our entrypoint. There we can write our logic, just like inside an action. This means we can call other framework functions, access the database using the usual way, send emails, render views, etc.

Let's print out a hello world to all our users in the console:

```
#!/usr/bin/env run-script
module Application.Script.HelloWorldToAllUsers where

import Application.Script.Prelude

run :: Script
run = do
    users <- query @User |> fetch
```

```
    forEach users \user -> do
        putStrLn "Hello World, " <> get #firstname user <> "!"
```

This will fetch all users and then print out “Hello World, Firstname!”.

Running a script

Scripts are executable by default. You can just run them like a bash script:

```
./Application/Script/HelloWorldToAllUsers.hs
...
Hello World, A!
Hello World, B!
Hello World, C!
```

This is made possible because of the she-bang line `#!/usr/bin/env run-script` at the top of the task file.

In case you get a permission error, try to add the executable flag via `chmod +x Application/Script/HelloWorldToAllUsers.hs`.

Building a script

In production you might want to build a script to a binary for performance reasons. Use make like this:

```
make build/bin/Script/HelloWorldToAllUsers
```

This will produce a binary `build/bin/Script/HelloWorldToAllUsers` from the source file `Application/Script/HelloWorldToAllUsers.hs`.

Debugging

print-Style Debugging

print-Style Debugging

TODO: Show example using traceShowId

Updating IHP

Releases

During beta, there is a new release every two weeks on Friday. You can find a list of all releases on GitHub..

A new version is usually announced first via the IHP Newsletter and also on Gitter and Twitter.

IHP version numbers are assigned by the release date. For example the version `v10072020` stands for the release of the `10.07.2020`.

Updating to the Latest Release

To update to the current IHP version, follow the instructions in the release notes. It's recommended to only update a single release version at a time when there are major breaking changes between your current version and the targeted update version.

Updating to the Latest Git Commit

To test out the latest IHP features even when they are not released yet you can update to the latest IHP git commit. For that you need to copy the git commit hash from the latest commit on GitHub.

After that open the `default.nix` in your project folder. This will look like this:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/haskellframework.git";
        rev = "720a10ddfd500d5930c926c293aea5a9016acb29";
    };

...
```

Now change the git commit hash in line 4 next to `rev =` to your chosen git commit hash.

Then rebuild everything by running the following:

```
nix-shell -j auto --cores 0 --run 'make -B .envrc'
```

After that you can use the project as usual by using `./start`.

When the commit you're trying out is not merged into the master branch this will fail because nix is doing only a shallow-clone of the repo. Follow the steps in "Updating to a specific Git Commit" below to fix this.

Updating to a specific Git Commit

When the commit you're trying out is not yet merged into the master branch, follow these steps.

Open the `default.nix` in your project folder. This will look like this:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/haskellframework.git";
        rev = "720a10ddfd500d5930c926c293aea5a9016acb29";
    };

...
```

Now change the git commit hash in line 4 next to `rev =` to your chosen git commit hash. Also add a `ref = "*"` otherwise nix will only shallow-clone the master branch:

```
let
    ihp = builtins.fetchGit {
```

```
        url = "https://github.com/digitallyinduced/haskellframework.git";
        rev = "... my new commit hash ...";
        ref = "*";
    };

...
```

Then rebuild everything by running the following:

```
nix-shell -j auto --cores 0 --run 'make -B .envrc'
```

After that you can use the project as usual by using `./start`.

Package Management

IHP is using the Nix Package Manager for managing its dependencies, such as haskell packages, compilers and even postgres. You can find more about the motivation on using nix on the IHP blog.

Using a Haskell Package

To install a haskell package from hackage (the standard haskell package registry), open the `default.nix` file and append the package name.

Let's say we want to use mmark for rendering markdown in our project. The mmark library is not bundled with IHP, so we need to add this package as a dependency to our project. For

that open the `default.nix`. The file will look like this:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/ihp.git";
        rev = "c6d40612697bb7905802f23b7753702d33b9e2c1";
    };
    haskellEnv = import "${ihp}/NixSupport/default.nix" {
        compiler = "ghc865";
        haskellDeps = p: with p; [
            cabal-install
            base
            wai
            text
            hlint
            p.ihp
        ];
        otherDeps = p: with p; [
        ];
        projectPath = ./.;
    };
in
    haskellEnv
```

In the list following `haskellDeps` we can see a few haskell dependencies already. We have to append `mmark` to the list:

```
let
    ihp = builtins.fetchGit {
```

```
        url = "https://github.com/digitallyinduced/ihp.git";
        rev = "c6d40612697bb7905802f23b7753702d33b9e2c1";
    };
    haskellEnv = import "${ihp}/NixSupport/default.nix" {
        compiler = "ghc865";
        haskellDeps = p: with p; [
            cabal-install
            base
            wai
            text
            hlint
            p.ihp
            mmark
        ];
        otherDeps = p: with p; [
        ];
        projectPath = ./.;
    };
in
    haskellEnv
```

Run `nix-shell` to see that your project still builds fine. After that run `make -B .envrc` to rebuild the dev env used by IHP. Now you can run `./start` again to start the dev server. The `mmark` can now be used as expected:

```
nix-shell
# All good? Proceed
make -B .envrc
./start
```

Using a Native Dependency

Sometimes your project uses some other software tool which is not bundled with IHP by default. Because we're using nix, we can easily manage that dependency for our project.

Let's say we want to add imagemagick to transform and resize images uploaded by the users of our application.

All dependencies of our project are listed in `default.nix` at the root of the project directory. The file looks like this:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/ihp.git";
        rev = "c6d40612697bb7905802f23b7753702d33b9e2c1";
    };
    haskellEnv = import "${ihp}/NixSupport/default.nix" {
        compiler = "ghc865";
        haskellDeps = p: with p; [
            cabal-install
            base
            classy-prelude
            wai
            text
            hlint
            ihp
            wreq
```

```
        ];
        otherDeps = p: with p; [
        ];
        projectPath = ./.;
    };
in
    haskellEnv
```

We now just have to add `imagemagick` to `otherDeps`:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/ihp.git";
        rev = "c6d40612697bb7905802f23b7753702d33b9e2c1";
    };
    haskellEnv = import "${ihp}/NixSupport/default.nix" {
        compiler = "ghc865";
        haskellDeps = p: with p; [
            cabal-install
            base
            classy-prelude
            wai
            text
            hlint
            ihp
            wreq
        ];
        otherDeps = p: with p; [

            imagemagick # <-----------------------
```

```
        ];
        projectPath = ./.;
    };
in
    haskellEnv
```

If running, stop your development server. Now run `make -B .envrc`. This will install imagemagick locally to your project.

When you are inside the project with your terminal, you can also call `imagemagick` to see that it's available.

You can look up the package name for the software you depend on inside the nixpkgs repository. Just open it on GitHub and use the GitHub search to look up the package name.

Advanced

Using a Different Version of a Haskell Package

Let's say we want to use the google-oauth2 package from hackage to add Google OAuth to our application. We can install the package in our project by adding it to `haskellPackages` in our `default.nix`:

```
let
    ihp = builtins.fetchGit {
        url = "https://github.com/digitallyinduced/ihp.git";
```

```
        rev = "c6d40612697bb7905802f23b7753702d33b9e2c1";
    };
    haskellEnv = import "${ihp}/NixSupport/default.nix" {
        compiler = "ghc865";
        haskellDeps = p: with p; [
            cabal-install
            base
            wai
            text
            hlint
            p.ihp
            google-oauth2
        ];
        otherDeps = p: with p; [
        ];
        projectPath = ./.;
    };
in
    haskellEnv
```

This will install version 0.3.0.0 of the google-oauth2 package, as this is the latest version available in the package set used by IHP. For our specific application need we want to use version 0.2.2, an older version of this package.

To use the older version of the package we need to override the package definition. To do this you need to install install `cabal2nix` first:

```
nix-env -i cabal2nix
```

After cabal2nix is installed we can use it to get a nix package definition for the older version of google-oauth2:

```
cabal2nix cabal://google-oauth2-0.2.2
```

This will output a new nix package definition like this:

```
{ mkDerivation, aeson, base, bytestring, hspec, HTTP, http-conduit
, http-types, load-env, stdenv
}:
mkDerivation {
  pname = "google-oauth2";
  version = "0.2.2";
  sha256 = "0n408kh48d7ky09j9zw9ad4mhbv1v7gq6i3ya4f6fhkjqqgw8c1j";
  libraryHaskellDepends = [
    aeson base bytestring HTTP http-conduit
  ];
  testHaskellDepends = [
    base bytestring hspec http-conduit http-types load-env
  ];
  description = "Google OAuth2 token negotiation";
  license = stdenv.lib.licenses.mit;
}
```

Save this package definition code to a new file in `Config/nix/haskell-packages/google-oauth2.nix`. IHP projects are configured to automatically pick up any haskell package

definitions in the `Config/nix/haskell-packages` directory. So this package definition will be used automatically.

Go back to your project directory and run `nix-shell`. This will try to install the new `google-oauth2` package in the expected version `0.2.2`.

This step might fail with an error like `Encountered missing or private dependencies`:

```
$ nix-shell
these derivations will be built:
  /nix/store/9a0bnhr5fzj0a40g72j2sb1sdbcpfavj-google-oauth2-0.2.2.drv
  /nix/store/nngc65qyw3bdf6zn84ca0jpxz19mmcv6-ghc-8.8.3-with-packages.drv
building '/nix/store/9a0bnhr5fzj0a40g72j2sb1sdbcpfavj-google-oauth2-0.2.2.
setupCompilerEnvironmentPhase
Build with /nix/store/102df4ic8qmmc6qqq33wwppizk9pnj1s-ghc-8.8.3.
unpacking sources
unpacking source archive /nix/store/w028wlk7f2q5axsw94f5igdbyfq982pl-googl
source root is google-oauth2-0.2.2
setting SOURCE_DATE_EPOCH to timestamp 1479590674 of file google-oauth2-0.
patching sources
compileBuildDriverPhase
setupCompileFlags: -package-db=/private/var/folders/27/4cznr1bj5yd0mq5sbf3
[1 of 1] Compiling Main             ( Setup.hs, /private/var/folders/27/4c
Linking Setup ...
configuring
configureFlags: --verbose --prefix=/nix/store/ba7dpyhns6k6a9q6s9faxg3wc389
Using Parsec parser
Configuring google-oauth2-0.2.2...
CallStack (from HasCallStack):
  die', called at libraries/Cabal/Cabal/Distribution/Simple/Configure.hs:1
  configureFinalizedPackage, called at libraries/Cabal/Cabal/Distribution/
```

```
  configure, called at libraries/Cabal/Cabal/Distribution/Simple.hs:625:20
  confHook, called at libraries/Cabal/Cabal/Distribution/Simple/UserHooks.h
  configureAction, called at libraries/Cabal/Cabal/Distribution/Simple.hs:1
  defaultMainHelper, called at libraries/Cabal/Cabal/Distribution/Simple.hs
  defaultMain, called at Setup.hs:2:8 in main:Main
Setup: Encountered missing or private dependencies:
aeson >=0.8 && <0.12

builder for '/nix/store/9a0bnhr5fzj0a40g72j2sb1sdbcpfavj-google-oauth2-0.2
cannot build derivation '/nix/store/nngc65qyw3bdf6zn84ca0jpxz19mmcv6-ghc-8
error: build of '/nix/store/nngc65qyw3bdf6zn84ca0jpxz19mmcv6-ghc-8.8.3-with
```

This is usually caused by a version mismatch between what the package expects and what is given by nix. You can disable version checking by “jailbreaking” the package.

To jailbreak the package open `Config/nix/nixpkgs-config.nix` and append `"google-oauth2"` to the `doJailbreakPackages` list:

```
{ ihp }:

let
  dontCheckPackages = [
    "ghc-mod"
    "cabal-helper"
    "generic-lens"
    "filesystem-conduit"
    "tz"
    "typerep-map"
  ];
```

```
  doJailbreakPackages = [
    "ghc-mod"
    "filesystem-conduit"
    "http-media"

    "google-oauth" # <------- ADD THE NEW PACKAGE HERE
  ];

  dontHaddockPackages = [];

  nixPkgsRev = "da7ddd822e32aeebea00e97ab5aeca9758250a40";
  nixPkgsSha256 = "0zbxbk4m72psbvd5p4qprcpiadndq1j2v517synijwp2vxc7cnv6";

...
```

After that try to run `nix-shell` again. This will most likely work now.

When the run of `nix-shell` succeeds, you also need to run `make -B .envrc` to rebuild the `.envrc` file. Otherwise the new package might not be visible to all tools:

```
make -B .envrc
```

Stopping Nix From Running Tests for a Haskell Dependency

Nix will try to run a testsuite for a package when it's building it from source. Sometimes the tests fail which will stop your from installing the package. In case you want to ignore the failing tests and use the package anyway follow these steps.

Open `Config/nix/nixpkgs-config.nix` and append the package name to the `dontCheckPackages` list:

```
{ ihp }:

let
  dontCheckPackages = [
    "ghc-mod"
    "cabal-helper"
    "generic-lens"
    "filesystem-conduit"
    "tz"
    "typerep-map"

    "my-failing-package" # <------- ADD YOUR PACKAGE HERE
  ];

  doJailbreakPackages = [
    "ghc-mod"
    "filesystem-conduit"
    "http-media"
  ];

  dontHaddockPackages = [];

  nixPkgsRev = "da7ddd822e32aeebea00e97ab5aeca9758250a40";
  nixPkgsSha256 = "0zbxbk4m72psbvd5p4qprcpiadndq1j2v517synijwp2vxc7cnv6";
```

```
...
```

After that you can do `nix-shell` without running the failing tests.

Nixpkgs Pinning

All projects using IHP are using a specific pinned version of nixpkgs. You can find the version used in your project used in `Config/nix/nixpkgs-config.nix`. The definition looks like this:

```
nixPkgsRev = "da7ddd822e32aeebea00e97ab5aeca9758250a40";
nixPkgsSha256 = "0zbxbk4m72psbvd5p4qprcpiadndq1j2v517synijwp2vxc7cnv6";
```

All nix packages installed for your project are using this specific version of nixpkgs.

You can change the nixpkgs version by updating the `nixPkgsRev` and `nixPkgsSha256` to your custom values and then running `make -B .envrc` to rebuild the dev env.

We highly recommend to only use nixpkgs versions which are provided by IHP because these are usually verified to be working well with all the packages used by IHP. Additionally you will need to build a lot of packages from source as they will not be available in the digitally induced binary cache.

Binary Cache

When installing IHP, the `ihp-new` tool will add the `digitallyinduced.cachix.org` binary cache to your nix system. This binary cache provides binaries for all IHP packages and commonly used dependencies for all nixpkgs versions used by IHP.

When you are using packages which are not in the binary cache and thus are compiled from source very often, we highly recommend to set up your own cachix binary cache and use it next to the default `digitallyinduced.cachix.org` cache.

Editors & Tooling

Introduction

This place describes all the steps needed to get your code editors and other tooling working with IHP. When your favorite code editor is not described here, feel free to add your setup to this list.

Using IHP with Visual Studio Code / VSCode

When using VSCode you need to install a plugin which loads the `.envrc` in your project. Otherwise the wrong GHC will be picked up. Take a look at vscode-direnv.

Most likely you want to use haskell-language-server for smart IDE features like autocompletion and jump-to-symbol. This is not working with IHP yet. There is an active

issue to implement support for this.

Using IHP with Sublime Text

Works great already out of the box.

Recommended packages:

`Nix` for syntax highlighting of nix files
`Direnv` to load the `.envrc` file of the project.

Deployment

Deploying with IHP Cloud

The fastest way to share your app with the internet is by using IHP Cloud. We recommend following this approach as it's the most simple to get started with.

Account Setup

IHP Cloud is currently still in beta. To register a new account you need an invite from an existing user or you can join the waiting list at ihpcloud.com.

Creating the Project

Once you're logged in to IHP Cloud, follow the instructions to create a project for your application.

Private Git Repos: When you connect a private git repository make sure to provide a SSH git clone url. IHP Cloud will provide you with a SSH public key. You need to add this deploy key to your repository.

On GitHub you can do this by opening the repository settings and clicking on `Deploy keys`.

First Deployment

After the project setup is finished, click on `+ Deploy Now` to start your first deployment.

When your CSS or JS looks broken, take a look at the next section `CSS & JS Bundling`.

CSS & JS Bundling

In production all your CSS and JS is bundled into a single file.

By default the `Web.View.Layout` is configured to serve these `prod.css` and `prod.js` files.

Take a short look at `Web/View/Layout.hs`. You can see that depending on whether the app is running in `development` or `production` mode it includes your CSS files or the `prod.css` / `prod.js` file.

```
stylesheets = do
    when (isDevelopment FrameworkConfig.environment) [hsx|
        <link rel="stylesheet" href="/vendor/bootstrap.min.css"/>
        <link rel="stylesheet" href="/vendor/flatpickr.min.css"/>
        <link rel="stylesheet" href="/app.css"/>
    |]
    when (isProduction FrameworkConfig.environment) [hsx|
        <link rel="stylesheet" href="/prod.css"/>
    |]

scripts = do
    when (isDevelopment FrameworkConfig.environment) [hsx|
        <script id="livereload-script" src="/livereload.js"></script>
        <script src="/vendor/jquery-3.2.1.slim.min.js"></script>
        <script src="/vendor/timeago.js"></script>
        <script src="/vendor/popper.min.js"></script>
        <script src="/vendor/bootstrap.min.js"></script>
        <script src="/vendor/flatpickr.js"></script>
        <script src="/helpers.js"></script>
        <script src="/vendor/morphdom-umd.min.js"></script>
    |]
```

```
    when (isProduction FrameworkConfig.environment) [hsx|
        <script src="/prod.js"></script>
    |]
```

These bundles are generated by using make:

```
make static/prod.js
make static/prod.css
```

The bundling process is only concatenating the files (along the lines of `cat a.css b.css c.css > static/prod.css`). Currently there is no minifcation or transpiling applied.

Configuring the CSS & JS Bundling

The files that are bundled together to get our `prod.css` and `prod.js` are configured in your projects `Makefile`. Inside your projects `Makefile` you will have these statements:

```
CSS_FILES += ${IHP}/static/vendor/bootstrap.min.css
CSS_FILES += ${IHP}/static/vendor/flatpickr.min.css

JS_FILES += ${IHP}/static/vendor/jquery-3.2.1.slim.min.js
JS_FILES += ${IHP}/static/vendor/timeago.js
JS_FILES += ${IHP}/static/vendor/popper.min.js
JS_FILES += ${IHP}/static/vendor/bootstrap.min.js
JS_FILES += ${IHP}/static/vendor/flatpickr.js
JS_FILES += ${IHP}/static/helpers.js
```

```
JS_FILES += ${IHP}/static/vendor/morphdom-umd.min.js
JS_FILES += ${IHP}/static/vendor/turbolinks.js
JS_FILES += ${IHP}/static/vendor/turbolinksInstantClick.js
JS_FILES += ${IHP}/static/vendor/turbolinksMorphdom.js
```

You need to add your app specific CSS and JS files here as well. E.g. if you have a `app.css`, a `layout.css` and a `app.js` add them by appending this:

```
CSS_FILES += static/app.css
CSS_FILES += static/layout.css
JS_FILES += static/app.js
```

Run `make static/prod.js static/prod.css` to test that the bundle generation works locally.

You can also remove the JS and CSS files that are provided by IHP (like `${IHP}/static/vendor/bootstrap.min.css`) if you don't need them. E.g. if you don't use bootstrap for your CSS, just remove the `CSS_FILES` and `JS_FILES` statements for bootstrap.

DB Migrations

Currently IHP has no standard way of doing migrations. Therefore currently you need to manually migrate your IHP Cloud database after deploying.

Open the project the project in IHP Cloud, click `Settings`, then click `Database`. There you can find the database credentials for the postgres DB that is running for your application. Connect to your database and manually apply the migrations.

Deploying manually

You can build and deploy your IHP app yourself.

Make sure that the infrastructure you pick to buid your IHP app has enough memory. Otherwise the build might fail because GHC is very memory hungry. You can also set up a swap file to work around this.

Install nix on your server

Nix is needed to build your application. Install it the usual way:

```
curl -L https://nixos.org/nix/install | sh
```

We recommend to use the digitally induced cachix binary cache to avoid rebuilding the IHP dependencies and IHP itself:

```
cachix use digitallyinduced
```

In case you're on NixOS, you can skip this.

Copy your project folder to your server

Copy your application source code to the build server. If you're using `git` to clone it onto your server, we recommend you use `SSH agent forwarding`.

Configuration

Make required modifications to your `Config/Config.hs`:

1. Switch `environment = Development` to `environment = Production`
2. Set `appHostname = "https://YOUR_HOSTNAME"`
3. Configure any custom settings

(This includes ´make -B .envrc´ to download and build any extra Haskell packages, such as the mmark package in the tutorial)

To configure your database connection: Set the env var `DATABASE_URL` to your postgres connection url. Set the env var `PORT` to the port the app will listen on.

The database needs the UUID-extension which is enabled by running ´create extension if not exists "uuid-ossp";´

Building

Inside your project directory start a `nix-shell` for the following steps.

We can use `make` to build the application binary. The default IHP Makefile provides two target: `build/bin/RunUnoptimizedProdServer` and `build/bin/RunOptimizedProdServer`.

The first target runs with `-O0`. It's useful when you're setting up the deployment workflow for the first time. Otherwise you will always need to spend lots of time waiting for GHC to optimize your haskell code.

Both make targets will generate a binary at `build/bin/RunUnoptimizedProdServer` or `build/bin/RunOptimizedProdServer` and will create a symlink at `build/bin/RunProdServer` targeting to the binary.

Run `make static/prod.css static/prod.js` to build the asset bundles.

Starting the app

Now you should be able to start your app by running `build/bin/RunProdServer`.

More Resources

Check out this blog post on how to deploy IHP to EC2.

Database

Introduction

IHP provides a few basic functions to access the database. On top of Postgres SQL we try to provide a thin layer to make it easy to do all the common tasks your web application usually does.

The only supported database platform is Postgres. Focussing on Postgres allows us to better integrate advanced Postgres-specific solutions into your application.

In development you do not need to set up anything to use postgres. The built-in development server automatically starts a Postgres instance to work with your application. The built-in development postgres server is only listening on a unix socket and is not available via TCP.

When the dev server is running, you can connect to it via `postgresql:///app?host=YOUR_PROJECT_DIRECTORY/build/db` with your favorite database tool. When inside the

project directory you can also use `make psql` to open a postgres REPL connected to the development database (named `app`), or start `psql` by pointing at the local sockets file `psql --host=/PATH/TO/PROJECT/DIRECTORY/build/db app`. The web interface of the dev server also has a GUI-based database editor (like phpmyadmin) at http://localhost:8001/ShowDatabase.

Haskell data structures and types are generated automatically based on your database schema.

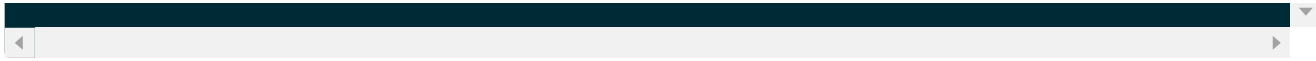
Schema.sql

Once you have created your project, the first step is to define a database schema. The database schema is basically just a SQL file with a lot of `CREATE TABLE ...` statements. You can find it at `Application/Schema.sql`.

In a new project this file will be empty. The UUID extension is automatically enabled for the database by IHP.

To define your database schema add your `CREATE TABLE ...` statements to the `Schema.sql`. For a users table this can look like this:

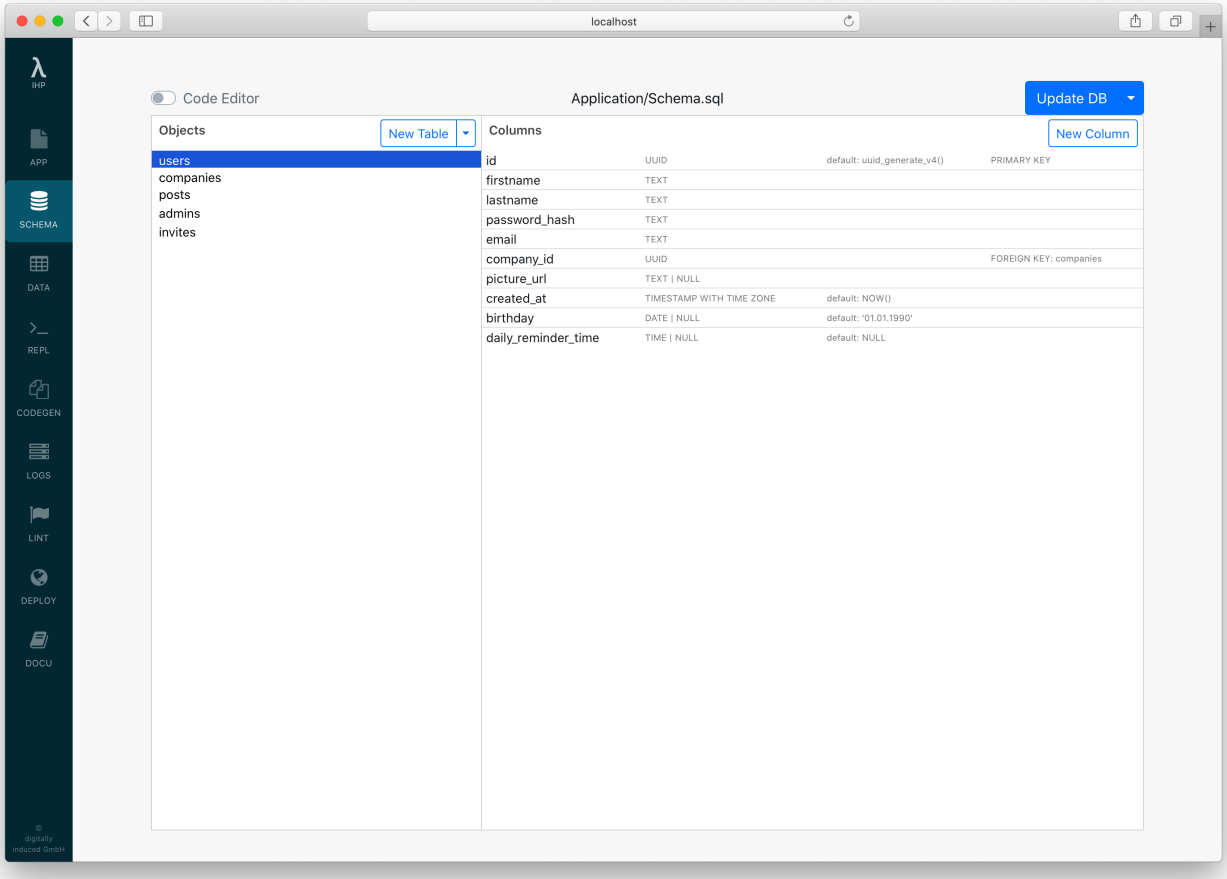
```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE users (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    firstname TEXT NOT NULL,
    lastname TEXT NOT NULL
);
```



Haskell data structures and types are automatically generated from the `Schema.sql` file. They are re-generated on every file change of the `Schema.sql`. We use the well-known `postgresql-simple` Haskell library to connect to the database.

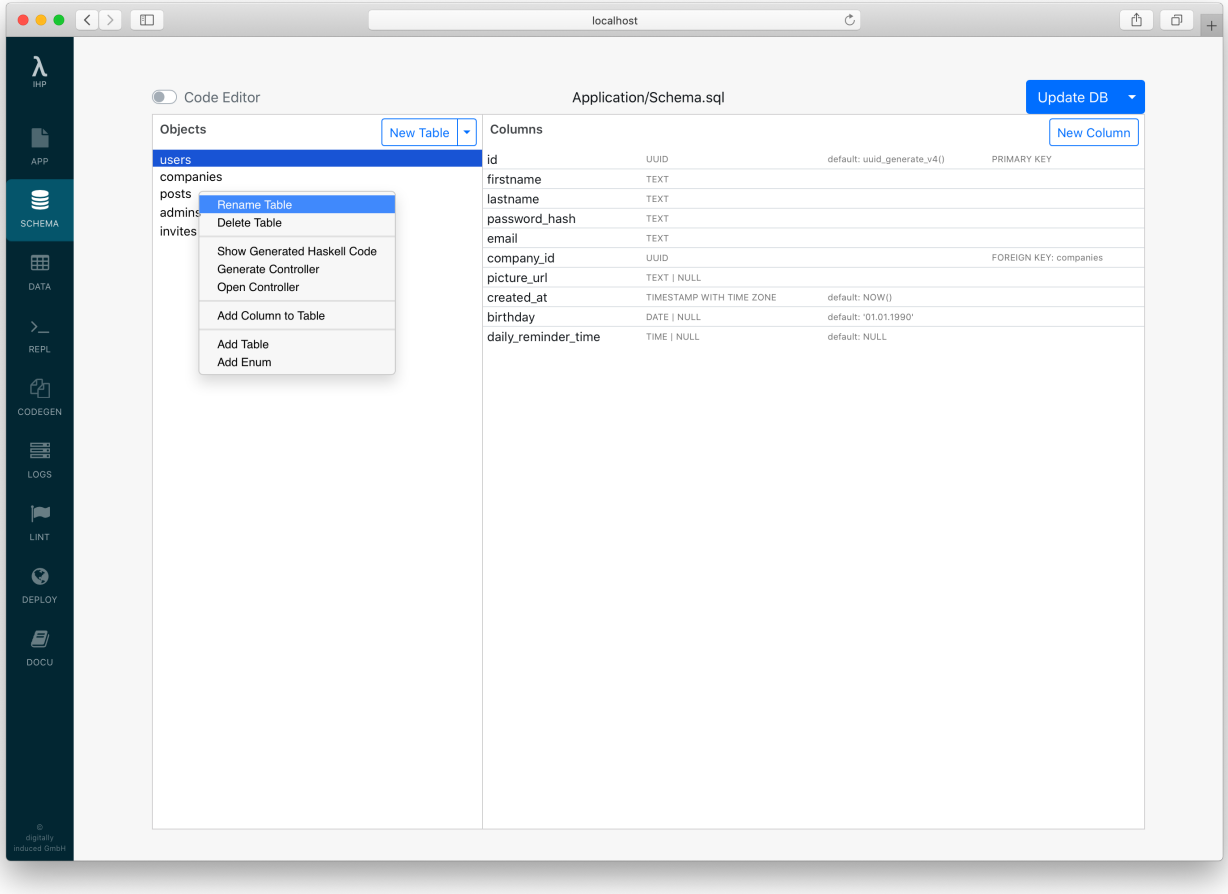
Schema Designer

Because the SQL syntax is sometimes hard to remember, the framework provides a GUI-based database editor called IHP Schema Designer. You can find the Schema Designer at `http://localhost:8001/Tables`.

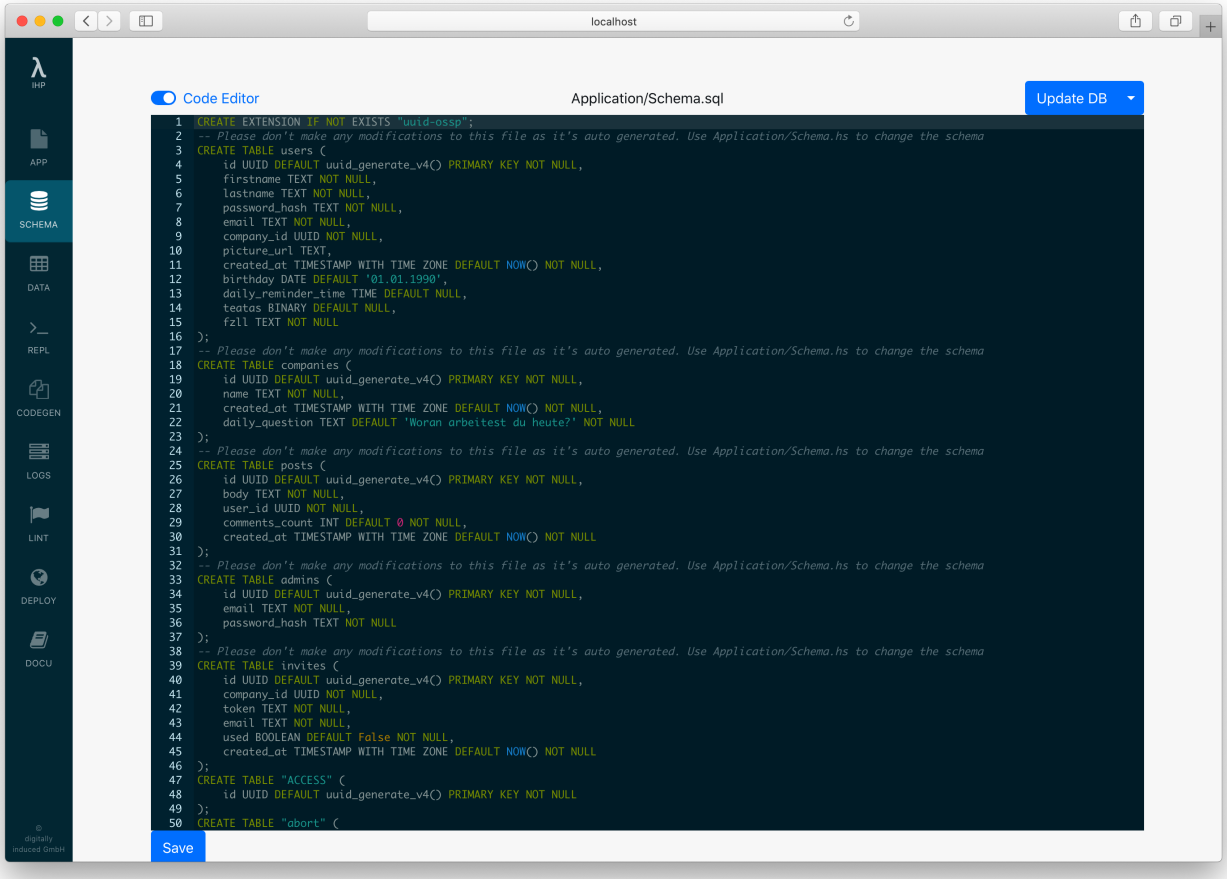


Keep in mind that the Schema Editor also only modifies the `Schema.sql`. This works by parsing the SQL DDL-statements and applying transformations on the AST and the compiling and writing it back to `Schema.sql`. When there is an syntax error in the `Schema.sql` file the visual mode will be unavailable and you have to work with the code editor to fix the problem.

You can add tables, columns, foreign key constraints, and enums. You can also edit these objects by right-clicking them. New tables have a `id` column by default. Lots of opinionated short-cuts for rapid application development like automatically offering to add foreign key constraints are built-in.



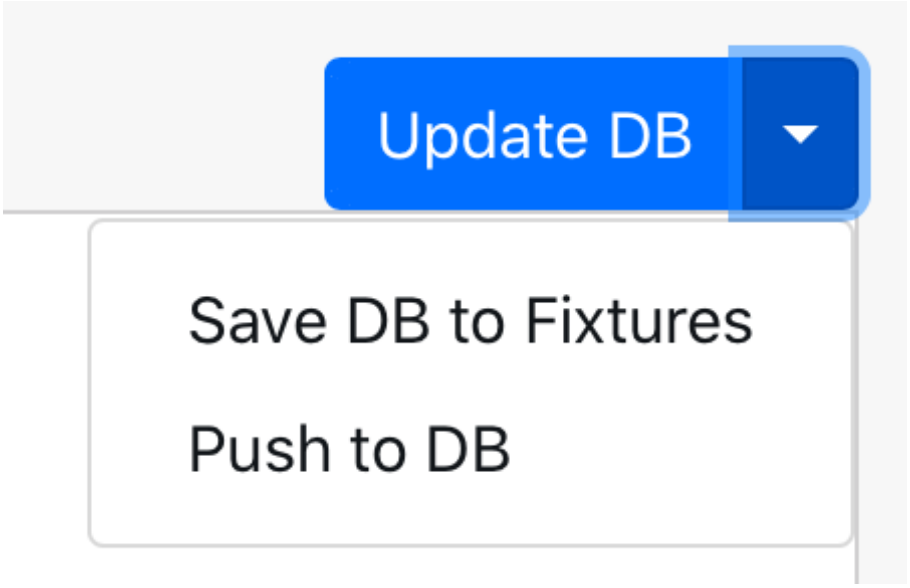
When the Visual Editor is not powerful enough, just switch back to the code editor. For convience, the Schema Designer also allows you to toggle to the Code Editor inside the web browser:



Push to DB

After we have added a few data structures to our `Schema.sql`, our running Postgres database is still empty. This is because we still need to import our database schema into the database.

In the Schema Designer: Click on `Push to DB`:



In the command line: Run `make db` while the server is running.

This will delete and re-create the current database and import the `Schema.sql`. After importing the Schema, it will also import the `Application/Fixtures.sql` which is used for prepopulating the empty database with some data. It's equivalent to running `psql < Schema.sql; psql < Fixtures.sql` inside an empty database.

When the dev server is started the first time, the `Schema.sql` and `Fixtures.sql` are automatically imported.

Fixtures.sql

The `Fixtures.sql` includes a lot of `INSERT INTO` statements to prefill your database once the schema has been created.

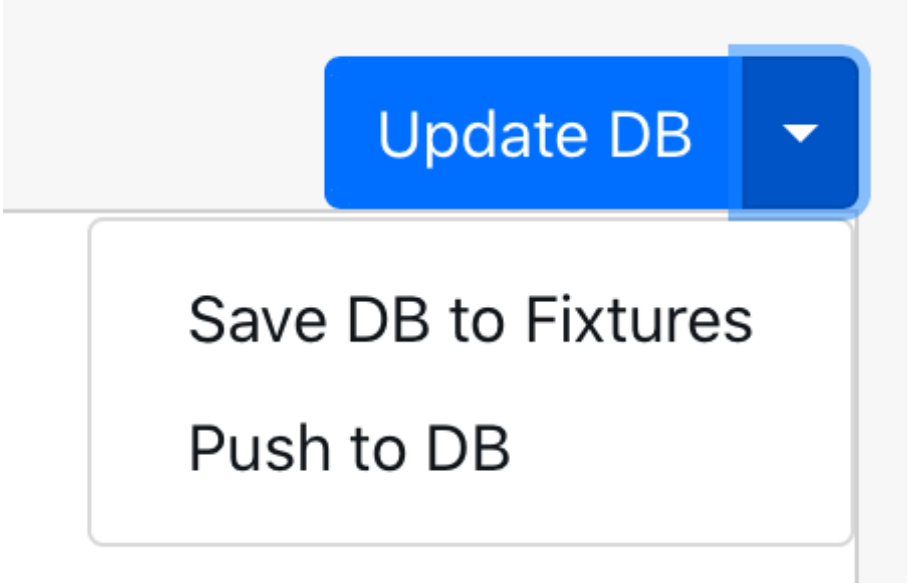
You can manually add `INSERT INTO` statements to this file. You can also *migrate* your fixtures by just making the required changes to this sql file.

You can dump your current database state into the `Fixtures.sql` by running `make dumpdb`. This way you can regularly commit the database state to git, so other developers have the same data inside their local development database as you have.

Update DB

You can also update the database while keeping its contents.

In the Schema Designer: Click on `Update DB`:



In the command line: Run `make dumpdb` and after that `make db`.

When dumping the database into the `Fixtures.sql` first and then rebuilding the database with the dump, the contents will be kept when changing the schema.

Haskell Bindings

Model Context

In a pure functional programming language like Haskell we need to pass the database connection to all functions which need to access the database. We use a implicit parameter `?modelContext :: ModelContext` to pass around the database connection without always specifying it. The `ModelContext` data structure is basically just a wrapper around the actual database connection.

An implicit paramter is a parameter which is automatically passed to certain functions, it just needs to be available in the current scope.

This means that all functions which are running database queries will need to be called from an function which has this implicit parameter in scope. A function doing something with the database, will always have a type signature specifing that it requires the `?modelContext` to be available, like this:

```
myFunc :: (?modelContext :: ModelContext) => IO SomeResult
```

All controller actions already have `?modelContext` in scope and thus can run database queries. Other application entry-points, like e.g. Scripts, also have this in scope.

This also means, that when a function does not specify that it depends on the database connection in its type signature (like `?modelContext :: ModelContext => ..`), you can be sure that it's not doing any database operations.

Haskell Data Structures

For every table in the `Schema.sql` a coresponding data structure will be generated on the Haskell side. For example given a table:

```
CREATE TABLE users (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    firstname TEXT NOT NULL,
    lastname TEXT NOT NULL
);
```

The generated Haskell data structure for this table will look like this:

```
data User = User
    { id :: Id User
    , firstname :: Text
    , lastname :: Text
    }
```

The `id` field type `Id User` is basically just a wrapper around `UUID` for type-safety reasons. All database field names are mapped from `under_score` to `camelCase` on the Haskell side.

When a sql field can be `NULL`, the Haskell field type will be contained in `Maybe`.

In the Schema Designer you can take a look at the generated Haskell code by right-clicking the table and clicking `Show Generated Haskell Code`.

Retrieving Records

Querying Records

You can retrieve all records of a table using `query`

```
do
    users <- query @User |> fetch
    forEach users \user -> do
        putStrLn (get #name user)
```

This will run a `SELECT * FROM users` query and put a list of `User` structures.

Fetching a single record

When you have the id of a record, you can also use `fetch` to get it from the database:

```
do
    let userId :: Id User = ...
    user <- fetch userId
    putStrLn (get #name user)
```

This will run the SQL query `SELECT * FROM users WHERE id = ... LIMIT 1`.

`fetch` knows only a single entity will be returned for the id, so instead of a list of users just a single user will be returned. In case the entity is not found, an exception is thrown. Use `fetchOrNothing` to get `Nothing` instead of an exception when no result is found

Fetching a list of ids

When have you a list of ids of a single record type, you can also just `fetch` them:

```
do
    let userIds :: [Id User] = ...
    users <- fetch userIds
```

This will run the SQL query `SELECT * FROM users WHERE id IN (...)`. The results in `users` have type `[User]`.

Fetching a `Maybe (Id record)`

Sometimes you have an optional id field, like e.g. when having a database schema like this:

```
CREATE TABLE tasks (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    description TEXT,
    assigned_user_id UUID
);
```

In this case the field `assigned_user_id` can be null. In our action we want to fetch user when it's not null, and return `Nothing` otherwise:

```
action ShowTask { taskId } = do
    task <- fetch taskId
    assignedUser <- case get #assignedUserId task of
            Just userId -> do
                user <- fetch userId
                pure (Just user)
            Nothing -> pure Nothing
```

This contains a lot of boilerplate for wrapping and unwrapping the `Maybe` value. Therefore you can just call `fetchOneOrNothing` directly on the `Maybe (Id User)` value:

```
action ShowTask { taskId } = do
    task <- fetch taskId
    assignedUser <- fetchOneOrNothing (get #assignedUserId task)
```

```
```

Raw SQL Queries

Use the function `sqlQuery` to run a raw sql query.

```
do
    result <- sqlQuery "SELECT * FROM projects WHERE id = ?" (Only id)
```

You might need to specify the expected result type, as type inference might not be able to guess it.

```
do
    result :: Project <- sqlQuery "SELECT * FROM projects WHERE id = ?" (O
```

You can query any kind of information, not only records:

```
do
    count :: Int <- sqlQuery "SELECT COUNT(*) FROM projects" []

    randomString :: Text <- sqlQuery "SELECT md5(random()::text)" []
```

Create

Creating a single record

To insert a record into the database, call `newRecord` to get an empty record value:

```
do
    let user = newRecord @User
    -- user = User { id = 0000-0000-0000-0000, firstname = "", lastname = "
```

The `newRecord` function does not insert the record, it just returns a new empty data structure we can fill with values and then insert into the database.

We can use `set` to fill in attributes:

```
do
    let user = newRecord @User
            |> set #firstname "Max"
            |> set #lastname "Mustermann"

    -- user = User { id = 0000-0000-0000-0000, firstname = "Max", lastname
```

We use `createRecord` to insert the above record into the `users` table:

```
do
    let user = newRecord @User
            |> set #firstname "Max"
```

```
        |> set #lastname "Mustermann"

    -- user = User { id = 0000-0000-0000-0000, firstname = "Max", lastname

    insertedUser <- user |> createRecord
    -- This will run INSERT INTO users (id, firstname, lastname) VALUES (D
    -- insertedUser = User { id = cf633b17-c409-4df5-a2fc-8c3b3d6c2ea7, fi
```

This can also be rewritten in a more compact form:

```
do
    user <- newRecord @User
            |> set #firstname "Max"
            |> set #lastname "Mustermann"
            |> createRecord
    -- user = User { id = "cf633b17-c409-4df5-a2fc-8c3b3d6c2ea7", firstnam
```

Creating many records

You can use `createMany` to insert multiple records with a single `INSERT` statement:

```
do
    let usersToBeInserted [ newRecord @User, newRecord @User, ... ]
    users <- createMany usersToBeInserted
```

This will run:

```
INSERT INTO users (id, firstname, lastname)
    VALUES (DEFAULT, "", ""), (DEFAULT, "", "") , (DEFAULT, "", "") ... ;
```

Update

The function `updateRecord` runs an `UPDATE` query for a specific record:

```
do
    user <- fetch ("cf633b17-c409-4df5-a2fc-8c3b3d6c2ea7" :: Id User)
    user
        |> set #lastname "Tester"
        |> updateRecord
```

This will set the lastname of user `cf633b17-c409-4df5-a2fc-8c3b3d6c2ea7` to `Tester` and run an `UPDATE` query to persist that:

```
UPDATE users SET firstname = firstname, lastname = "Tester" WHERE id = "cf
```

The `UPDATE` query will only update columns that have been changed using `|> set #someField someValue` on the record.

Delete

Deleting a single record

Use `deleteRecord` to run a simple `DELETE` query:

```
do
    user <- fetch ("cf633b17-c409-4df5-a2fc-8c3b3d6c2ea7" :: Id User)
    deleteRecord user
```

This will execute:

```
DELETE FROM users WHERE id = "cf633b17-c409-4df5-a2fc-8c3b3d6c2ea7"
```

Deleting many records

Use `deleteRecords` to run a `DELETE` query for multiple records:

```
do
    users :: [User] <- ...
    deleteRecords users
```

This will execute:

```
DELETE FROM users WHERE id IN (...)
```

Deleting all records

Use `deleteAll` to run a `DELETE` query for all rows in a table:

```
do
    deleteAll @User
```

This will execute:

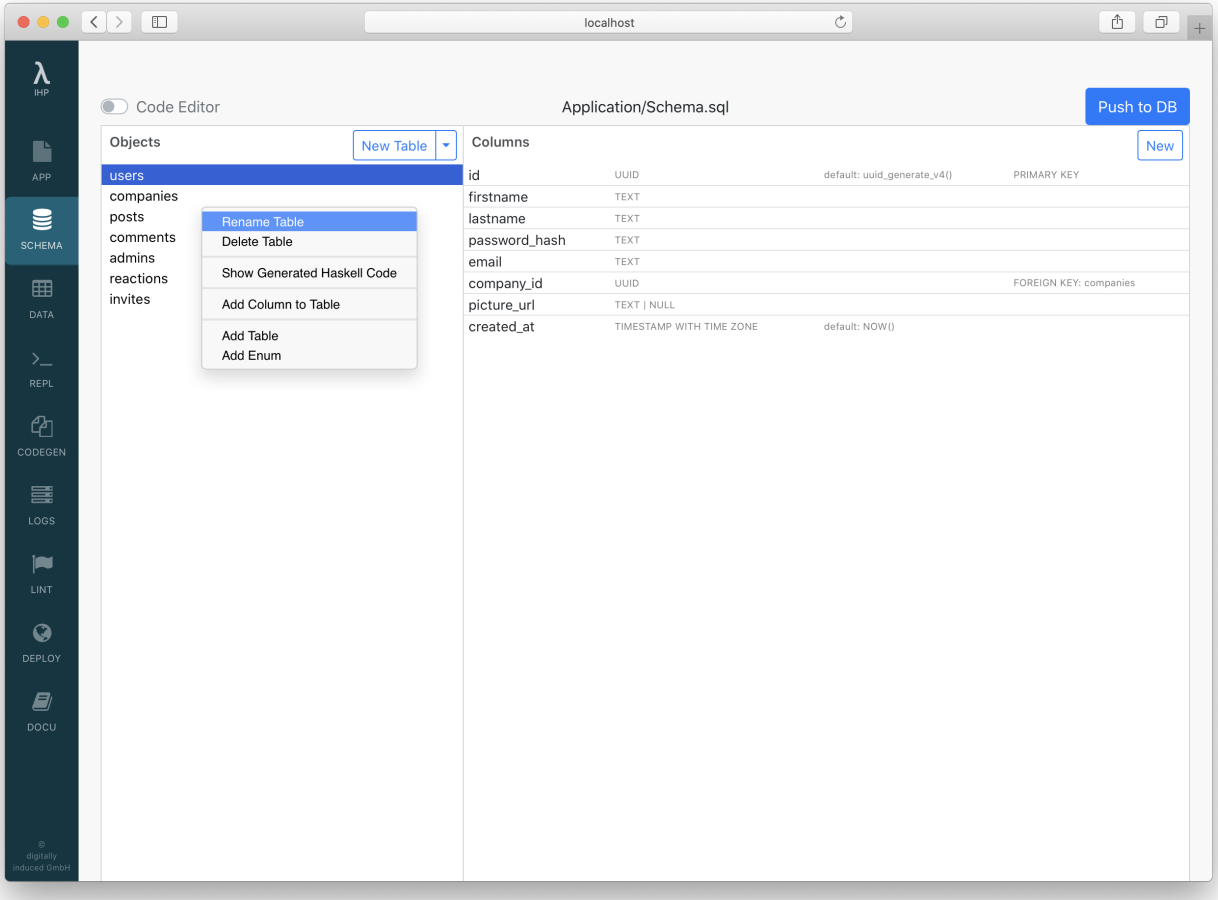
```
DELETE FROM users
```

Enums

It's possible to define and use custom enum types with IHP. A enum can be created using the Schema Designer. The process is pretty much the same as when creating a table.

Adding enums via the Schema Designer

Open the Schema Designer, right click into the `Objects` Pane and then select `Add Enum`:



You have to give a name to your enum type. The name should be in plural form, like with table names. E.g. we could name our enum `colors`.

Next add the enum values by right clicking into the `Values` pane and click on `Add Value`. Here we could add values such as `red`, `blue` and `yellow`.

Adding enums via SQL

Instead of using the Schema Designer you can also just add the required SQL statement manually into `Application/Schema.hs`:

```
CREATE TYPE colors AS ENUM ('blue', 'red', 'yellow');
```

Using enums in the Haskell Code

The above `colors` example will allow us to access the enum like this:

```
let blue :: Colors = Blue
let red :: Colors = Red
let yellow :: Colors = Yellow
```

You can use the enum as a field type for another record. E.g. we can have `posts` table and there give each post a color:

```
CREATE TABLE posts (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    body TEXT NOT NULL,
    color colors
);
```

You can use `fill` even with custom enums:

```
    action CreatePostAction = do
        let post = newRecord @Post
        post
            |> fill @["body", "color"]
            |> ifValid \case
                Left post -> render NewView { .. }
                Right post -> do
                    post <- post |> createRecord
                    setSuccessMessage "Post created"
                    redirectTo PostsAction
```

In your views, use `inputValue` to get a textual representation for your enum which works with `fill`:

```
[hsx|
    <input type="text" value={inputValue Blue}/>
|]
```

Database Refactoring

In the following section you will find best practise workflows to do changes to your database schema.

Adding a Column

When adding a new column to an existing table, it's best to add a default value so that existing values in your table can be re-inserted from your `Fixtures.sql` before using `Update DB`.

In case you don't have a default value and run `Update DB`, the import of `Application/Fixtures.sql` will be completed with errors. The table with the new column will most likely be empty. To keep the records in your table without setting a default value use the following process to work around this issue:

1. Add your column `new_col` in the Schema Designer
2. Click `Save DB to Fixtures` in the Schema Designer (Use the arrow next to the `Update DB` button to see this option)
3. Open the `Application/Fixtures.sql` in your editor and manually update all `INSERT INTO` lines for your changed table. Change the lines so that all columns are provided for the changed table.
4. Click `Push to DB` in the Schema Designer (Click the arrow on the `Update DB` button to see this option)

Renaming a Column

When you are renaming a column, the development process of using `Update DB` will not work be working. This is because the `Update DB` will save the old database state into the `Fixtures.sql`. There it still references the old column names.

In this case it's best to use the following approach:

1. Rename your column `col_a` to `col_b` in the Schema Designer
2. Click `Save DB to Fixtures` in the Schema Designer (Use the arrow next to the `Update DB` button to see this option)
3. Open the `Application/Fixtures.sql` in your editor and manually update references from the old column `col_a` to the new name `col_b`
4. Click `Push to DB` in the Schema Designer (Use the arrow next to the `Update DB` button to see this option)

Migrations In Production

IHP currently has no built-in migration system yet. We're still experimenting with a great way to solve this. Until then, the recommended approach used by digitally induced is to manually migrate your database using DDL statements.

Relationships

Introduction

The following sections assume the following database schema being given. It's basically the same as in "Your First Project".

```
CREATE TABLE posts (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    title TEXT NOT NULL,
    body TEXT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW() NOT NULL
);
```

```
CREATE TABLE comments (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    post_id UUID NOT NULL,
    author TEXT NOT NULL,
    body TEXT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW() NOT NULL
);

ALTER TABLE comments ADD CONSTRAINT comments_ref_post_id FOREIGN KEY (post_
```

Has Many Relationships

Given a specific post, we can fetch the post and all its comments like this:

```
let postId :: Id Post = ...

post <- fetch postId
    >>= fetchRelated #comments
```

This Haskell code will trigger the following sql queries to be executed:

```
SELECT posts.* FROM posts WHERE id = ?  LIMIT 1
SELECT comments.* FROM comments WHERE post_id = ?
```

In the view we can just access the comments like this:

```
[hsx|
    <h1>{get #title post}</h1>
    <h2>Comments:</h2>
    {post |> get #comments}
|]
```

The `post |> get #comments` returns a list of the comments belonging to the post.

The type of `post` is `Include "comments" Post` instead of the usual `Post`. This way the state of fetched nested resource is tracked at the type level.

Order by

When we want to order the relationship in a certain way, we can just apply our commonly used `orderBy` function:

```
let postId :: Id Post = ...

post <- fetch postId
    >>= pure . modify #comments (orderByDesc #createdAt)
    >>= fetchRelated #comments
```

This works because the `comments` field of a `Post` is just a `QueryBuilder` before we call `fetchRelated`.

This query builder is equivalent to:

```
query @Comment |> filterWhere (#postId, get #id post)
```

The call to `>>= pure . modify #comments (orderByDesc #createdAt)` just appends a `|> orderByDesc #createdAt` like this:

```
query @Comment |> filterWhere (#postId, get #id post) |> orderByDesc #creat
```

Then the `fetchRelated` basically just executes this query builder and puts the result back into the `comments` field of the `post` record.

Multiple Records

When we want to fetch all the comments for a list of posts, we can use `collectionFetchRelated`:

```
posts <- query @Post
    |> fetch
    >>= collectionFetchRelated #comments
```

This will query all posts with all their comments. The type of posts is `[Include "comments" Post]`.

The above Haskell code will trigger the following two sql queries to be executed:

```
SELECT posts.* FROM posts
SELECT comments.* FROM comments WHERE post_id IN (?)
```

Belongs To Relationships

Given a specific comment, we can fetch the post this comment belongs to. Like other relationships this is also using `fetchRelated`:

```
let comment :: Id Comment = ...

comment <- fetch comment
    >>= fetchRelated #postId
```

This Haskell code will trigger the following sql queries to be executed:

```
SELECT comments.* FROM comments WHERE id = ? LIMIT 1
SELECT posts.* FROM posts WHERE id = ?  LIMIT 1
```

In the view we can just access the comments like this:

```
[hsx|
    <h1>Comment to {comment |> get #postId |> get #title}</h1>
    <h2>Comments:</h2>
    {comment |> get #body}
|]
```

The type of `comment` is `Include "postId" Comment` instead of the usual `Comment`. This way the state of fetched nested resource is tracked at the type level.

Delete Behavior

Usually all your relations are secured at the database layer by using foreign key constraints. But that means e.g. deleting a post will fail when there still exists comments.

By default a new foreign key constraint created via the Schema Designer has no `on delete` behavior specified. Therefore the foreign key constraint will look like this:

```
ALTER TABLE comments ADD CONSTRAINT comments_ref_post_id FOREIGN KEY (post_
```

See the `NO ACTION` at the end of the statement? We have to change this do `CASCADE` to delete all comments when the related post is going to be deleted:

```
ALTER TABLE comments ADD CONSTRAINT comments_ref_post_id FOREIGN KEY (post_
```

Of course, you can change this using the Schema Designer by clicking on the foreign key next to the `post_id` column in the `comments` table.

QueryBuilder


```
projectId |> fetch
```

Introduction

The QueryBuilder module allows you to compose database queries in a type-safe way. Below you can find a short reference to all the commonly-used functions.

Creating a new query

To query the database for some records, you first need to build a query. You can just use the `query` function for that.

```
let myQueryBuilder = query
```

You can optionally specify the model you want to query:

```
let myProjectQueryBuilder = query @Project
```

Running a query

You can run a query using `fetch`, `fetchOneOrNothing` or `fetchOne`:

many rows: `fetch`

To run a query which will return many rows use `fetch`:

```
example :: IO [Project]
example = do
    projects <- query @Project |> fetch
    -- Query: `SELECT * FROM projects`
    return projects
```

maybe single row: `fetchOneOrNothing`

To run a query which will maybe return a single row use `fetchOneOrNothing`:

```
example :: IO (Maybe Project)
example = do
    project <- query @Project |> fetchOneOrNothing
    -- Query: `SELECT * FROM projects LIMIT 1`
    return project
```

single row: `fetchOne`

To run a query which will return a single row and **throw an error if no record is found** use `fetchOne`:

```
example :: IO Project
example = do
    project <- query @Project |> fetchOne
    -- Query: `SELECT * FROM projects LIMIT 1`
    return project
```

Where Conditions

To specify `WHERE` conditions, you can use `filterWhere`:

```
projectsByUser :: UserId -> IO [Project]
projectsByUser userId = do
    projects <- query @Project
            |> filterWhere (#userId, userId)
            |> filterWhere (#deleted, False)
            |> fetch
    -- Query: `SELECT * FROM projects WHERE user_id = <userId> AND deleted
    return projects
```

Order By

You can just use `orderBy #field`:

```
projects <- query @Project
        |> orderBy #createdAt
        |> fetch
-- Query: `SELECT * FROM projects ORDER BY created_at`
```

Or

```
projects <- query @Project
         |> queryOr
            (filterWhere (#userId, userId)) (filterWhere (#teamId, teamId))
        |> fetch
-- Query: `SELECT * FROM projects WHERE (user_id = ?) OR (team_id = ?)`
```

Union / Merging two queries

Two query builders of the same type can be merged like this:

```
-- SELECT * FROM projects WHERE team_id = ?
let teamProjects :: QueryBuilder Project = query @Project |> filterWhere (#

-- SELECT * FROM projects WHERE team_id IS NULL AND created_by = ?
let personalProjects :: QueryBuilder Project = query @Project |> filterWher
```

```
-- SELECT * FROM projects WHERE (team_id = ?) OR (team_id IS NULL AND creat
let projects :: QueryBuilder Project = queryUnion teamProjects personalPro
```

Shortcuts

`findBy #field value`

Just a shortcut for `filterWhere (#field, value) |> fetchOne`

```
-- Long version
project <- query @Project |> filterWhere (#userId, userId) |> fetchOne
-- Shorter version
project <- query @Project |> findBy #userId userId
```

`findMaybeBy #field value`

Just a shortcut for `filterWhere (#field, value) |> fetchOneOrNothing`

```
-- Long version
project <- query @Project |> filterWhere (#userId, userId) |> fetchOneOrNo
-- Shorter version
project <- query @Project |> findMaybeBy #userId userId
```

`findManyBy #field value`

Just a shortcut for `filterWhere (#field, value) |> fetch`

```
-- Long version
projects <- query @Project |> filterWhere (#userId, userId) |> fetch
-- Shorter version
projects <- query @Project |> findManyBy #userId userId
```

`projectId |> fetch`

Ids also have `fetch` implementations, that way you can just run:

```
let projectId :: ProjectId = ...
project <- projectId |> fetch
```

For convenience there is also a `fetch` implementation for `Maybe SomeId`:

```
let assignedUserId :: Maybe UserId = project |> get #assignedUserId
assignedUser <- assignedUserId |> fetchOneOrNothing
```

Authentication

Introduction

IHP provides a basic authentication toolkit out of the box.

The usual convention in IHP is to call your user record `User`. When there is an admin user, we usually call the record `Admin`. In general the authentication can work with any kind of record. The only requirement is that it has an id field.

To use the authentication module, your `users` table needs to have at least an `id`, `email`, `password_hash`, `locked_at` and `failed_login_attempts` field:

```
CREATE TABLE users (
    id UUID DEFAULT uuid_generate_v4() PRIMARY KEY NOT NULL,
    email TEXT NOT NULL,
    password_hash TEXT NOT NULL,
    locked_at TIMESTAMP WITH TIME ZONE DEFAULT NULL,
    failed_login_attempts INT DEFAULT 0 NOT NULL
);
```

The password authentication saves the passwords as a salted hash using the pwstore-fast library. By default, a user will be locked for an hour after 10 failed login attempts.

Setup

Currently the authentication toolkit has to be enabled manually. We plan to do this setup using a code generator in the future.

Controller Helpers

First we need to enable the controller helpers. Open `Application/Helper/Controller.hs`. It will look like this:

```
module Application.Helper.Controller (
    -- To use the built in login:
    -- module IHP.LoginSupport.Helper.Controller
) where

-- Here you can add functions which are available in all your controllers

-- To use the built in login:
-- import IHP.LoginSupport.Helper.Controller
```

Enable the import and re-export for `IHP.LoginSupport.Helper.Controller` and add a type instance `type instance CurrentUserRecord = User`:

```
module Application.Helper.Controller (
    module IHP.LoginSupport.Helper.Controller
) where

-- Here you can add functions which are available in all your controllers

import IHP.LoginSupport.Helper.Controller
import Generated.Types

type instance CurrentUserRecord = User
```

View Helpers

Additionally we also need to enable the view helpers. Open `Application/Helper/View.hs`. It will look like this:

```
module Application.Helper.View (
    -- To use the built in login:
    -- module IHP.LoginSupport.Helper.View
) where

-- Here you can add functions which are available in all your views

-- To use the built in login:
-- import IHP.LoginSupport.Helper.View
```

Enable the import and re-export for `IHP.LoginSupport.Helper.View`:

```
module Application.Helper.View (
    module IHP.LoginSupport.Helper.View
) where

-- Here you can add functions which are available in all your views

import IHP.LoginSupport.Helper.View
```

Activating the Session

Open `Web/FrontController.hs`. Add an import for `IHP.LoginSupport.Middleware` and `Web.Controller.Sessions`:

```
import IHP.LoginSupport.Middleware
import Web.Controller.Sessions
```

We then need to mount our session controller by adding `parseRoute @SessionController`:

```
instance FrontController WebApplication where
    controllers =
        [ startPage WelcomeAction
        , parseRoute @SessionsController -- <---------------- add this
        -- Generator Marker
        ]
```

At the end of the file there is a line like:

```
instance InitControllerContext WebApplication
```

We need to extend this function with the following code:

```
instance InitControllerContext WebApplication where
    initContext =
        initAuthentication @User
```

This will fetch the user from the database when a `userId` is given in the session. The fetched user record is saved to the special `?controllerContext` variable and is used by all the helper functions we have imported before.

Adding a Session Controller

In the last step we need to add a new controller which will deal with the login and logout. We call this the `SessionsController`.

First we have to update `Web/Types.hs`. The auth module directs users to the login page automatically if required by a view, to set this up we add the following to `Web/Types.hs`:

```
import IHP.LoginSupport.Types

instance HasNewSessionUrl User where
    newSessionUrl _ = "/NewSession"
```

You also need to add the type definitions for the SessionsController:

```
data SessionsController
    = NewSessionAction
    | CreateSessionAction
    | DeleteSessionAction
    deriving (Eq, Show, Data)
```

After that we need to set up routing for our new controller in `Web/Routes.hs`:

```
instance AutoRoute SessionsController
```

We also need to create a `Web/Controller/Sessions.hs` calling the IHP authentication module:

```
module Web.Controller.Sessions where

import Web.Controller.Prelude
import Web.View.Sessions.New
import qualified IHP.AuthSupport.Controller.Sessions as Sessions

instance Controller SessionsController where
    action NewSessionAction = Sessions.newSessionAction @User
    action CreateSessionAction = Sessions.createSessionAction @User
    action DeleteSessionAction = Sessions.deleteSessionAction @User

instance Sessions.SessionsControllerConfig User where
```

Additionally we need to implement a login view at `Web/View/Sessions/New.hs` like this:

```
module Web.View.Sessions.New where
import Web.View.Prelude
import IHP.AuthSupport.View.Sessions.New

instance View (NewView User) ViewContext where
```

```
    html NewView { .. } = [hsx|
        <div class="h-100" id="sessions-new">
            <div class="d-flex align-items-center">
                <div class="w-100">
                    <div style="max-width: 400px" class="mx-auto mb-5">
                        {renderFlashMessages}
                        <h5>Please login:</h5>
                        {renderForm user}
                    </div>
                </div>
            </div>
        </div>
    |]

renderForm :: User -> Html
renderForm user = [hsx|
    <form method="POST" action={CreateSessionAction}>
        <div class="form-group">
            <input name="email" value={get #email user} type="email" class=
        </div>
        <div class="form-group">
            <input name="password" type="password" class="form-control" pla
        </div>
        <button type="submit" class="btn btn-primary btn-block">Login</butt
    </form>
|]
```

Trying out the login

After you have completed the above steps, you can open the login at `/NewSession`. You can generate a link to your login page like this:

```
<a href={NewSessionAction}>Login</a>
```

Accessing the current user

In order to access the current user from your actions and templates you need to add it to the view context.

Update `Web/Types.hs` and add a `user` field to the ViewContext data type:

```
data ViewContext = ViewContext
    { requestContext :: ControllerSupport.RequestContext
    , flashMessages :: [IHP.Controller.Session.FlashMessage]
    , controllerContext :: ControllerSupport.ControllerContext
    , layout :: Layout
    , user :: Maybe User -- <--------------- add this
    }
```

and then uncomment it in `Web/View/Context.hs`:

```
let viewContext = ViewContext {
    requestContext = ?requestContext,
        user = currentUserOrNothing, -- <--------------- uncomment this li
```

```
        flashMessages,
        controllerContext = ?controllerContext,
        layout = let ?viewContext = viewContext in defaultLayout
}
```

Inside your actions you can then use `currentUser` to get access to the current logged in user:

```
action MyAction = do
    let text = "Hello " <> (get #email currentUser)
    renderPlain text
```

In case the user is logged out, an exception will be thrown when accessing `currentUser` and the browser will automatically be redirected to the `NewSessionAction`.

You can use `currentUserOrNothing` to manually deal with the not-logged-in case:

```
action MyAction = do
    case currentUserOrNothing of
        Just currentUser -> do
            let text = "Hello " <> (get #email currentUser)
            renderPlain text
        Nothing -> renderPlain "Please login first"
```

Additionally you can use `currentUserId` as a shortcut for `currentUser |> get #id`.

You can also access the user using `currentUser` inside your views:

```
[hsx|
    <h1>Hello {get #email currentUser}</h1>
|]
```

Logout

You can simply render a link inside your layout or view to send the user to the logout:

```
<a class="js-delete js-delete-no-confirm" href={DeleteSessionAction}>Logou
```

Creating a user

To create a user with a hashed password, you just need to call the hashing function before saving it into the database:

```
    action CreateUserAction = do
        let user = newRecord @User
        user
            |> fill @["email", "passwordHash"]
            |> validateField #email isEmail
```

```
            |> validateField #passwordHash nonEmpty
            |> ifValid \case
                Left user -> render NewView { .. }
                Right user -> do
                    hashed <- hashPassword (get #passwordHash user)
                    user <- user
                        |> set #passwordHash hashed
                        |> createRecord
                    setSuccessMessage "You have registered successfully"
```

Next: Authorization

Authorization

Restricting an action to logged in users

To restrict an action to a logged in user, use `ensureIsUser`:

```
action PostsAction = do
    ensureIsUser
    posts <- query @Post |> fetch
    render IndexView { .. }
```

When someone is trying to access the `PostsAction` but is not logged in, the browser will be redirected to the login page. After the login succeeded, the user will be redirected back to the `PostsAction`.

It's common to restrict all actions inside a controller to logged in users only. Place the `ensureIsUser` inside the `beforeAction` hook to automatically apply it to all actions:

```
instance Controller PostsController where
    beforeAction = ensureIsUser

    action PostsAction = do
        posts <- query @Post |> fetch
        render IndexView { .. }

    action ShowPostAction { postId } = do
        post <- fetch postId
        render ShowView { .. }
```

In this case `PostsAction` and `ShowPostAction` are only accessible to logged-in users.

Checking for Permissions

You can use `accessDeniedUnless` to allow certain things only for specific users. For example, to restrict a `ShowPostAction` only to the user who a post belongs to, use this:

```
    action ShowPostAction { postId } = do
        post <- fetch postId
        accessDeniedUnless (get #userId post == currentUserId)

        render ShowView { .. }
```