Lab Report - Exercise 8: Rails Dash

2017-03-02

Simon Fecke (s0544318)

Metawee Langka (s0551423)

Marina Stürböth (s0534873)

Lab Report

Table of Contents

1. Setting up the Database
2. Adding test data via rake
3. Pages and the Comment section
4. CSS and the Application layout
5. Uploading images
6. Evaluation
7. Screenshots, Use cases and Testing

---

1. Setting up the Database

After installing Ruby on Rails and MySQL, we first created a new project using the MySQL database.

Console

```
$ rails new webshop -d mysql
```

Then, from inside the webshop directory we installed all dependencies the project has.

Console

```
$ bundle install
```

The next step was setting up the database connection.

/config/database.yml

```
default: &default
  adapter: mysql2
  encoding: utf8
  pool: 5
  username: root
  password: mysql
  host: localhost
  socket: MySQL

development:
  <<: *default
  database: webshop_development
```

Then we used scaffolding to implement the first modules, which in our case are ingredients, breadtypes and sandwiches. Ingredients and breadtypes have a name and a description, while sandwiches (we didn't use the word "hoagie" because rails would build the plural "hoagies" and then name some parts "hoagy" instead, which lead to inconsistencies), only has its ID and a reference to the breadtype used, which we implemented manually later.

Console

```
$ rails generate scaffold sandwich

$ rails generate scaffold ingredient name:string
description:string

$ rails generate scaffold breadtype name:string
description:string
```

Now it was time to set up the database and run a migration.

Console

```
$ rake db:create

$rake db:migrate
```

Looking into mysql directly showed that the tables were there, but of course there was still no connection between them. We had to tell the modules to connect and add these to the

database. In the case of breadtype and sandwich, this was easy, as it is a one-to-many relationship.

/app/models/sandwich.rb

```
class Sandwich < ApplicationRecord
     belongs_to :breadtype
end
```

/app/models/breadtype.rb

```
class Breadtype < ApplicationRecord
     has_many :sandwiches
end
```

And then we added the foreign-key-column to the migration file of sandwiches.

/db/migrate/20170131165158_create_sandwiches.rb

```
class CreateSandwiches < ActiveRecord::Migration[5.0]
  def change
    create_table :sandwiches do |t|
      t.belongs_to :breadtype, index: true
      t.timestamps
    end
  end
end
```

The last thing to do was to tell the controller about the additional parameter.

/app/controllers/sandwiches_controller.rb

```
def sandwich_params
      params.require(:sandwich).permit(:breadtype_id)
    end
```

The connection between sandwich and ingredient however was a little trickier to implement, as it is a many-to-many relationship. While Ruby on Rails provides the method `has_and_belongs_to_many` for this case, we decided to take a different, more flexible approach. So we created our own many-to-many model and called it “toppings”, but it only stores the ids of sandwiches and ingredients. This however would allow us to add further

columns at a later stage in development, such as quantities or extra wishes from the customer (e.g. “evenly spread out”, “only on one half”). So instead of has_and_belongs_to_many, we used has_many with the parameter through.

Console

```
$ rails generate model topping
```

/db/migrate/20170131165551_create_toppings.rb

```
class CreateToppings < ActiveRecord::Migration[5.0]
  def change
    create_table :toppings do |t|
      t.belongs_to :sandwich, index: true
      t.belongs_to :ingredient, index: true
      t.timestamps
    end
  end
end
```

/app/model/topping.rb

```
class Topping < ApplicationRecord
     belongs_to :sandwich
     belongs_to :ingredient
end
```

So this was the middle part of the relationship between ingredient and sandwich. Now we only had to tell those models about the connection as well.

/app/model/sandwich.rb

```
class Sandwich < ApplicationRecord
     has_many :toppings
     has_many :ingredients, :through => :toppings

     belongs_to :breadtype
end
```

/app/models/ingredient.rb

```
class Ingredient < ApplicationRecord
      has_many :toppings
      has_many :sandwiches, :through => :toppings
end
```

Since we won't query from the ingredient side ("On which sandwich are onions?" is not relevant at this point) we don't need to change anything there, but we do have to tell the sandwich controller to take care of the ingredients which will now be added to each sandwich via the topping-model.

/app/controllers/sandwiches_controller.rb

```
def sandwich_params
      params.require(:sandwich).permit(:breadtype_id,
ingredient_ids: [])

end
```

2. Adding test data via rake

We added some test values to the seed file.

/db/seeds.rb

```
Ingredient.create(name: 'Bacon', description: 'tasty')
Ingredient.create(name: 'Lettuce', description: 'healthy')
Ingredient.create(name: 'Tomato', description: 'juicy')
Ingredient.create(name: 'Pickles', description: 'funny')
Ingredient.create(name: 'Onions', description: 'roasted')
Ingredient.create(name: 'Jalapeno', description: 'spicy')
Breadtype.create(name: 'Toast', description: 'simple')
Breadtype.create(name: 'Italian', description: 'herbs')
```

And then invoked it to populate our database with these values.

Console

```
$ rake db:seed
```

3. Pages and the Comment section

With that all set up, we wanted to add some more pages to the webshop. First was a controller called pages which would serve three different views: Home, About and Admin. The first two simply being there to make the whole application seem more like a real website and give the user something to actually navigate and the last one being kind of a hub for pages that would not be accessible for normal users, especially the form pages ‘new’ and ‘edit’ for breadtypes and ingredients.

Console

```
$ rails generate controller pages home about admin
```

Then we also wanted to add the comment section (or guest book). This however would again need to store something in our database, so we needed a scaffold.

Console

```
$ rails generate scaffold comment name:string post:text
```

Then we also added some comments to the seed file.

/db/seeds.rb

```
Comment.create(name: 'Marcus', post: 'I had a BLT and it was
awesome!!!!')
Comment.create(name: 'Vanessa', post: 'Srsly?!?!?!? No
cheese????? Plz fix!!!11')
```

4. CSS and the Application layout

With all that we could start the face-lift. We changed the application layout so it would include a navigation bar visible on all pages and views. This really improved the usability of our application.

/app/views/layouts/application.html.erb

```
 <body>
     <div id="wrapper">

       <header>

           <%= link_to(
                image_tag('/assets/logo-100.png'),
             url_for({:controller => 'pages', :action => 'home'}),
             :class => 'websiteLogo') %>

           <%= link_to 'Hoagie To-Go Webshop', {:controller => 'pages', :action
=> 'home'},
              {:class => 'websiteTitle'} %>

       </header>

       <nav id="horizontalNav">
         <p>
            <%= link_to 'Home', {:controller => 'pages', :action => 'home'} %>
            <%= link_to 'Order Hoagie', {:controller => 'sandwiches', :action =>
'index'} %>
            <%= link_to 'Guestbook', {:controller => 'comments', :action =>
'index'} %>
            <%= link_to 'Admin', {:controller => 'pages', :action => 'admin'} %>
            <%= link_to 'About', {:controller => 'pages', :action => 'about'} %>
         </p>
       </nav>

       <main id="content">

        <%= yield %>

      </main>

     </div>
  </body>
```

Then we added a whole bunch of css to make everything look better. As for colors, we used the colors from http://corporatedesign.htw-berlin.de/schrift-farbe/markenfarben/ (last visited 2017-03-02), because of the webshop name (HTW - “Hoagie To-Go Webshop”).
Besides the colors, we also improved on spacing by adding some margin and padding attributes, especially to the navbar and the tables for the index sites.
We added classes for odd and even rows to alternate between to different grey background colors to improve the readability as well.

/app/assets/stylesheets/aplication.css

```
a:hover {
      color: #76B900;
      background-color: #FFF;
}

body {
      background: #76B900;
}

div#wrapper{
      margin: auto;
      background: #FFF;
      width: 1000px;
      height: 2000px;
}

header {
      display: block;
      padding: 10px 10px 30px 10px;
}

.websiteLogo {
      float: left;
      text-decoration: none;
}

.websiteTitle {
      font-size: 50px;
      text-decoration: none;
      font-weight: bold;
      display: inline;
      line-height: 90px;
      padding-left: 20px;
}

#horizontalNav {
      display: block;
      padding: 10px 20px 10px 10px;
      background: #AFAFAF;
}

#horizontalNav a {
      font-size: 16px;
      text-decoration: none;
      margin-left: 40px;
      font-weight: bold;
      color: #FFF;
}
```

```
#horizontalNav a:hover {
      color: #76B900;
      background-color: #AFAFAF;
}

main {
      padding: 10px 20px 20px 20px;
}

table {
      border-collapse: collapse;
}

thead {
      text-align: left;
      background: #76B900;
      color: #FFF;
}

thead th {
      padding: 10px;
}

table tr td {
      padding: 10px;
}

.tr_even {
      background: #CCC;
}

.tr_odd {
      background: #EEE;
}
```

5. Uploading images

We used the paperclip gem for this feature. (https://github.com/thoughtbot/paperclip, last visited 2017-03-02). To be able to use Paperclip the dependency ImageMagick must be installed and Paperclip must have access to it.

config/environments/development.rb

```
Paperclip.options[:command_path] = "/usr/local/bin/"
```

In the model we added.

/app/models/comment.rb

```
has_attached_file :image, styles: { large: "600x600>", medium:
"400x400>", thumb: "150x150#" }
validates_attachment_content_type :image, content_type:
/\Aimage\/.*\z/
```

This allow us to upload and use an image file and we can also call this file in different sizes. Next, we have to add a migration for this file to the database.

Console

```
$ rails generate paperclip comment image
```

/db/migrate/20170203085442_add_attachment_image_to_comments.rb

```
class AddAttachmentImageToComments < ActiveRecord::Migration
  def self.up
    change_table :comments do |t|
      t.attachment :image
    end
  end

  def self.down
    remove_attachment :comments, :image
  end
end
```

Now the migration file has been created, we only have to invoke it with our database. As usual we wrote `$ rake db:migrate` in the console. Next, we have to change something in our view to be able to upload and edit the image file.

/view/comments/_form.html.erb

```
<%= form_for @comment, html: { multipart: true }) do |f| %>
.
.
<div class="field">
    <%= f.label :image %>
    <%= f.file_field :image %>
</div>
```

We made a change in the first line and added another field for the upload button. Now we can upload an image to the website, but it still doesn't show on the page yet. So the next thing we have to do to `show.html.erb`, `index.html.erb` , or wherever we want this image to display, is just adding `<%= image_tag @comment.image.url(:large) %>` to the code.

Lastly, we have to grant the permission in the controller to the image that we uploaded.

controllers/concern/comments_controller.rb

```
private
    # Use callbacks to share common setup or constraints between
actions.
    def set_comment
      @comment = Comment.find(params[:id])
    end
    # Never trust parameters from the scary internet, only allow
the white list through.
    def comment_params
      params.require(:comment).permit(:name, :post, :image)
    end
end
```

We repeated the same process with ingredients and bread types.

6. Evaluation

This last exercise, while very difficult, was also a lot of fun! We learned quite a lot about Ruby, Ruby on Rails and web development in general. From time to time there were some basic errors we had a hard time fixing.

For example, if a user created a sandwich with three toppings, and then went to the edit page and removed all toppings, nothing would change. It took us some time to find out, that zero checkboxes meant zero inputs and therefore zero changes to the database, which could be fixed by adding another hidden checkbox with a NIL value.

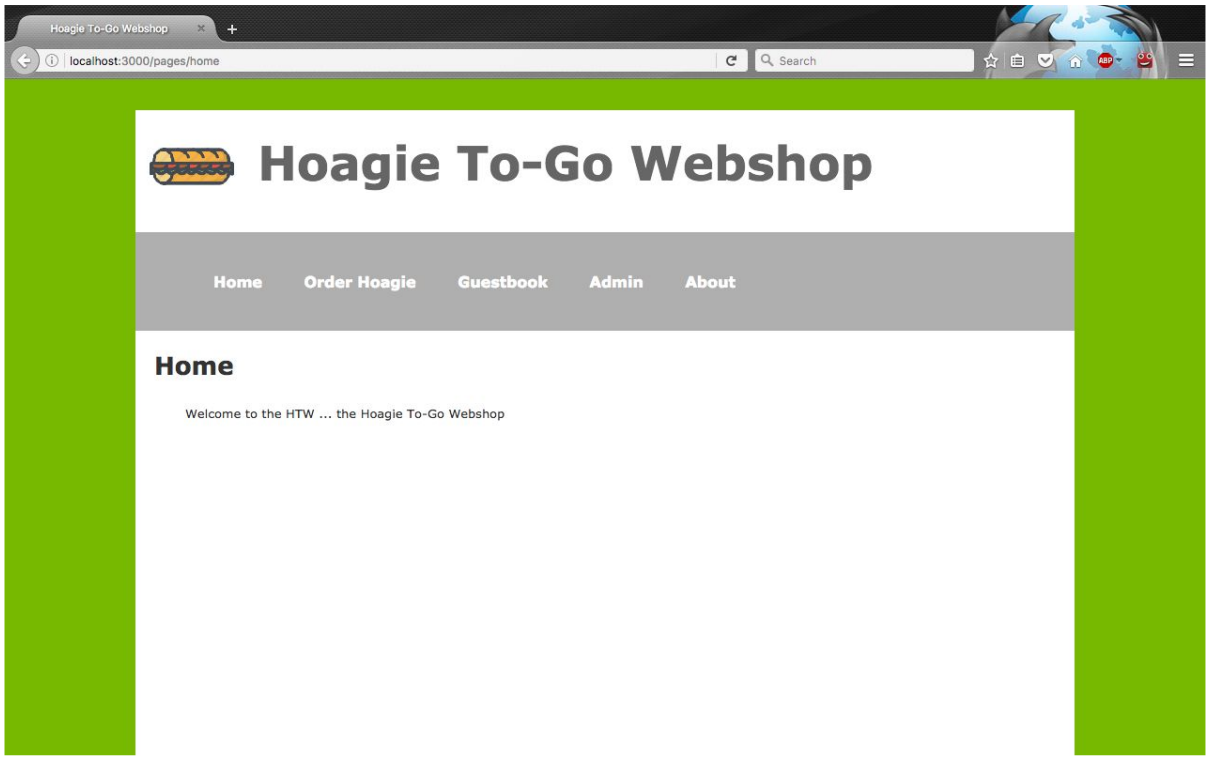
Also a lot of general mistakes kept us busy, which would mostly occur after enhancing a module without making all necessary changes in both the corresponding view, the model as well as the controller.

A lot of this could have been prevented by having a lab with Ruby on Rails earlier this semester. This could have been something short and simple, but it would have introduced us to the framework earlier and would have incentivised us to play around and experiment with it during the semester. All our group members agreed, that this would have made this big project, especially during the exams, go a little more fluid.

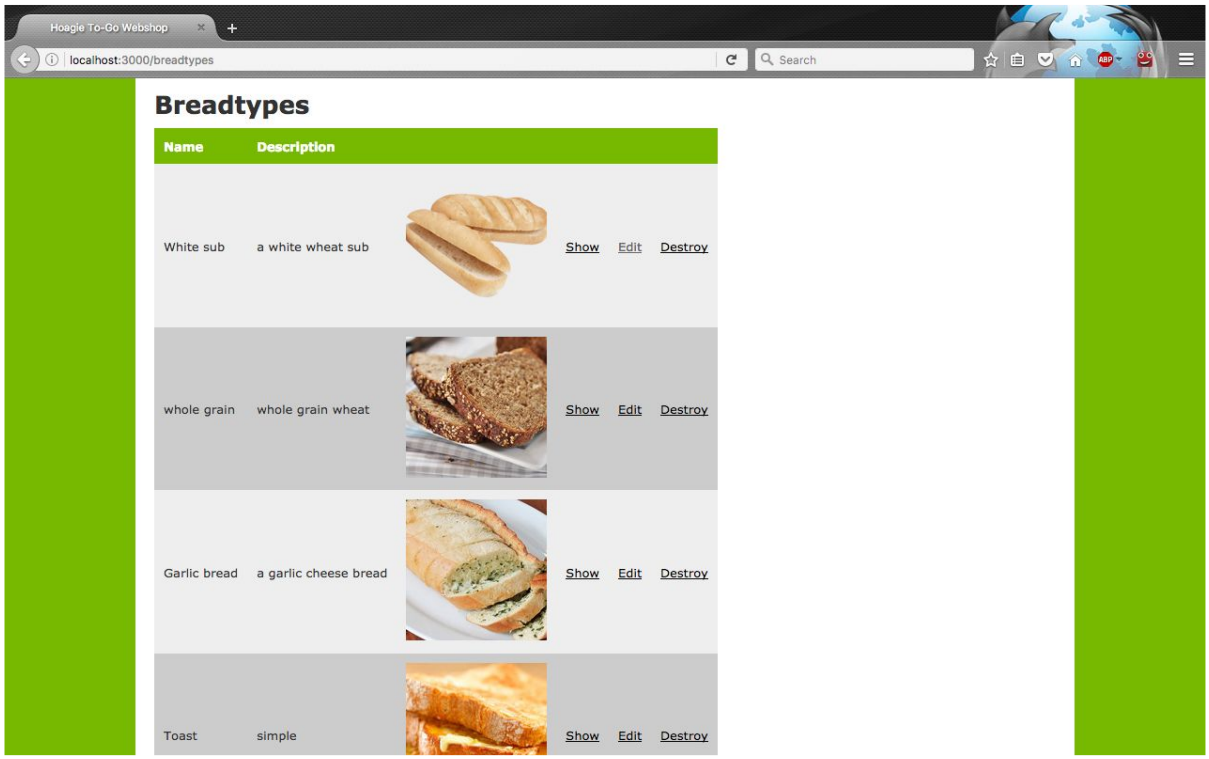
All in all this project required 32 man hours to complete.

7. Screenshots, Use cases and Testing

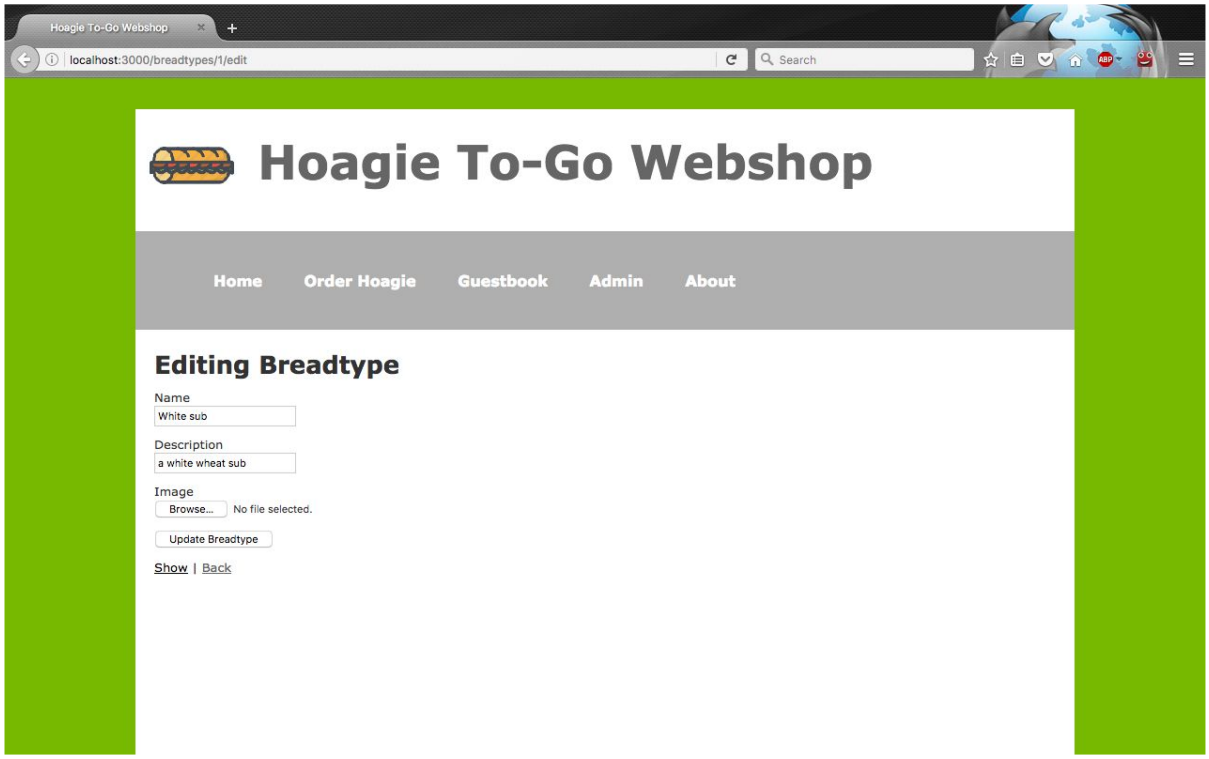
Homepage / Landing page - Clicking on the image or text in the website header as well as “Home” in the navbar will send the user here.



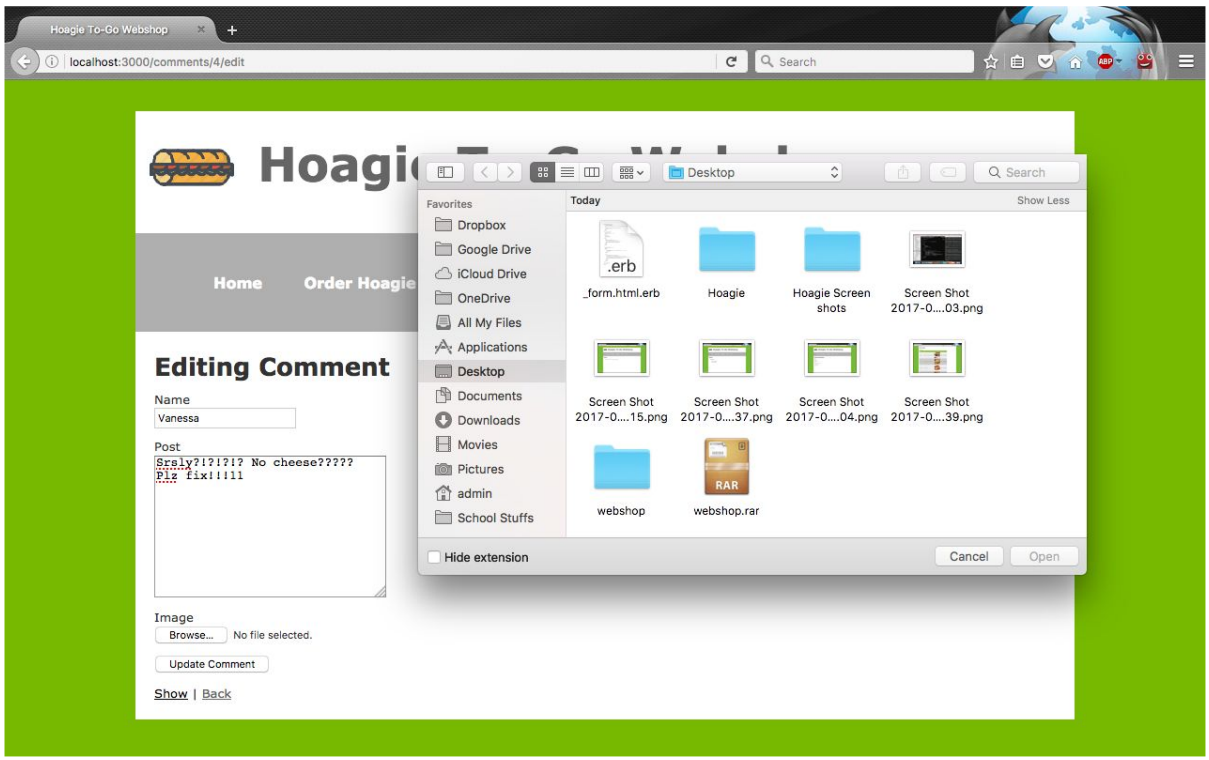
The list of all available bread types. This page can accessed by clicking on “Admin” in the navigation bar, then choose “Manage Bread Types”.



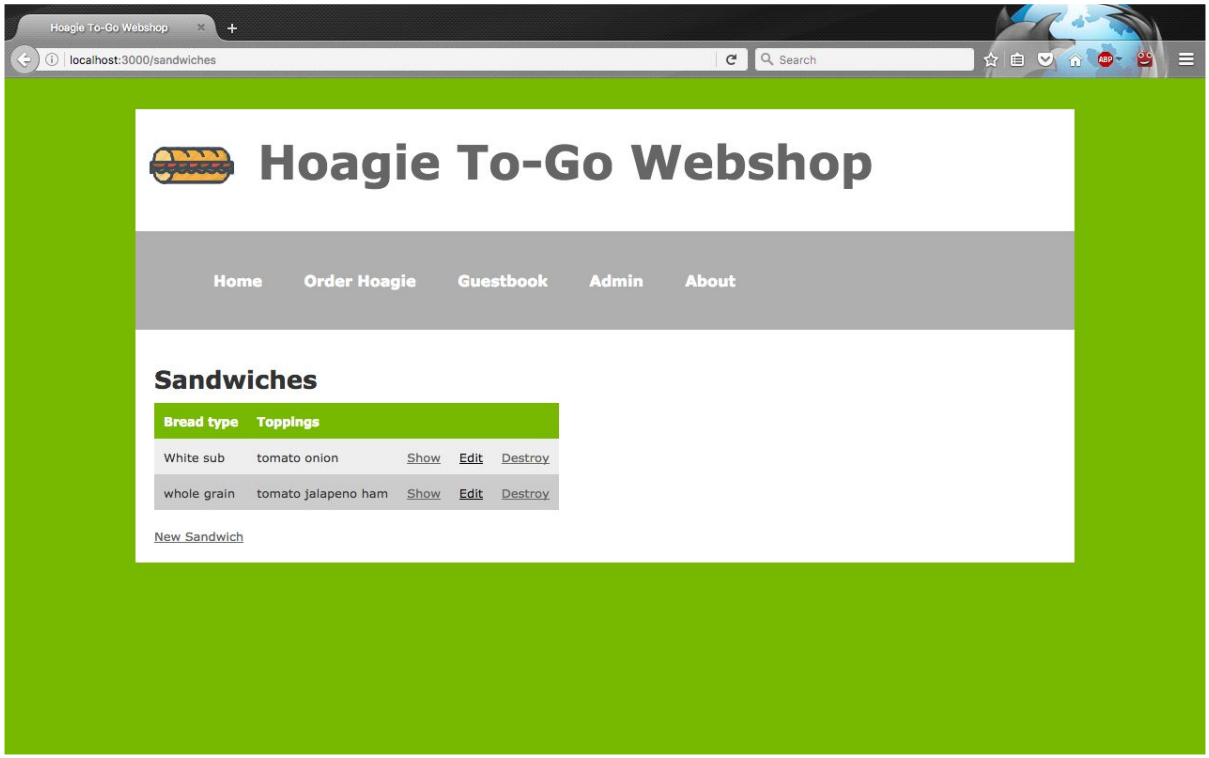
Editing a bread type. This page can be accessed from the previous screen by clicking “Edit” for the item in question.



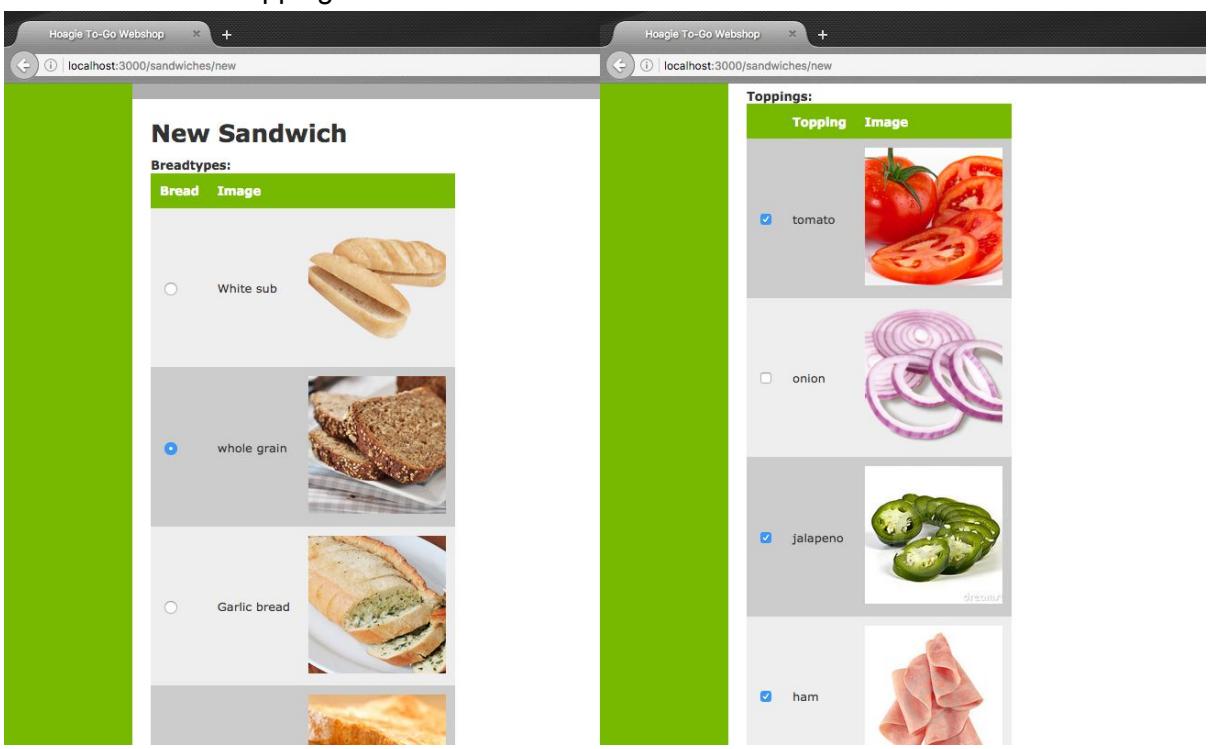
When editing a comment a user can add an image. This can also be used by an administrator to remove an image.



The sandwiches that a user buys are displayed in a list. In a more elaborate implementation this would be the content of an order / shopping cart. Clicking on “New Sandwich” will take the user to the sandwich form.



When creating a new sandwich all available breadtypes and ingredients are being listed. The input for choosing the bread is a radio button, as only one can be selected. The input for choosing the toppings however are checkboxes, as any number (even none) of ingredients can be chosen as toppings.



The new sandwich is created. The contents are displayed to the user. This is the sandwich view “show” with a success message after creating it.