

Milestone project 4 additional write-up

Table of Contents

[User Stories 3](#_Toc3417580)

[Stories 3](#_Toc3417581)

[As a Lego enthusiast, I… 3](#_Toc3417582)

[As owner of the store, I… 4](#_Toc3417583)

[Derived stories yet to be implemented 5](#_Toc3417584)

[Mockups and UX considerations 5](#_Toc3417585)

[Mockups 5](#_Toc3417586)

[UX considerations 6](#_Toc3417587)

[Testing 11](#_Toc3417588)

[Manual Testing Table 11](#_Toc3417589)

[Manual Testing Scenarios 12](#_Toc3417590)

[Unregistered user 12](#_Toc3417591)

[Unregistered user / Registered user (not signed in) 12](#_Toc3417592)

[Unregistered user / Registered user (signed in) 12](#_Toc3417593)

[Regular registered user (signed in) 12](#_Toc3417594)

[Staff registered user (signed in) 13](#_Toc3417595)

[Django Admin access 13](#_Toc3417596)

[Automated Testing 14](#_Toc3417597)

[Cart 14](#_Toc3417598)

[Checkout 14](#_Toc3417599)

[Infopages 14](#_Toc3417600)

[Products 15](#_Toc3417601)

[Travis CI 15](#_Toc3417602)

[Deployment 16](#_Toc3417603)

[Deployment on local environment 16](#_Toc3417604)

[Environment variables 17](#_Toc3417605)

[Group creation in Django Admin 18](#_Toc3417606)

[Running on hosted environments 19](#_Toc3417607)

[Heroku 19](#_Toc3417608)

[AWS 21](#_Toc3417609)

[Bullet list deployment 22](#_Toc3417610)

# User Stories

The user stories here after are accompanied by the resulting developer tasks.

The overall Epic for this project is:

*As a user, I want a place where I can get stickers to complete the Lego sets I have built, but am missing the stickers for.*

This overall ‘desire’ was then broken down to the following stories and their accompanying developer tasks related to them.

## Stories

### As a Lego enthusiast, I…

1. Want to have a landing page, where I can see the latest stickers
   1. Build a landing page with some general information
   2. Create a back end able to hold a catalog of products
   3. Show the most recent additions to the catalog on the landing page
2. Want to be able to search for specific sticker sets
   1. Create a search field
      1. Allow free text searches
      2. Have searches cover categories and descriptions for maximum response
      3. Return the results, along with the search string they’re based on
   2. Show the search field on the products page
3. Want to be able to view a single Lego category for sticker sets
   1. Create a pulldown menu with categories
      1. Populate the list from the unique categories in the database
      2. When clicked, return the results specific to that category and show the category selected
4. Want to know how the stickers are made
   1. Create an FAQ section covering various questions
   2. Base the FAQ page on database entries, allowing easy adding of items, and no need to update page HTML
5. Want some way to be kept updated on new items or any other news
   1. Allow users to register an account, so their email address can be used to send out newsletters to them
   2. Upon registration, directly direct users to the main page
   3. Create an option for users to view their information
   4. Create a way for users to sign out of their account
   5. Create a way for users to delete their own account
6. Want to know how my personal data is treated
   1. Create a page with a privacy statement
7. Want to have a way to stop receiving news from the site
   1. This is covered in 5c.
8. Want to know how I can go about issuing questions
   1. Create a page where customers can contact the site’s owner through a contact form
   2. Provide a directly visible link to the customer to the contact page
   3. When submitted, email the form contents to the site owner
9. Want a notification when I place an order with information on my order
   1. When a user places an order, send them a confirmation mail containing their order information

### As owner of the store, I…

1. Want to have a way to add products without having to go into the admin pages
   1. Create an admin section where staff members can add items to the database
   2. Shield the page from non-staff users that manipulate the URL to the page
   3. Redirect unwanted individuals to the main page
   4. Throw an alert box when successfully adding an item or on error
2. Want to be able to quickly expand the FAQ area when certain questions are asked often
   1. Create the same overall solution for the addition of FAQ items as was made for new products to maintain look and feel
3. Don’t want regular customers to be able to add products, or FAQ items
   1. Create views to check for staff membership and render view based on true or false
4. Want to be able to process order from within the site
   1. Create a view showing all orders that are not yet processed
   2. Allow checking an order as processed to remove it from the view
   3. Make sure orders are not deleted, just marked as processed
5. Want customers to be able to leave comments on products
   1. Create a product detail page where customers can leave comments on the product
   2. Show the comments from newest to oldest
6. Want customers to get the content from their sent contact form as well
   1. Expand the view for the contact form to also send an email to the customer on the address they entered
7. Want customers that registered to have the option to change their password
   1. Provide the user with links on the login page to a password reset page
   2. Have users receive an email from which they can reset their password
   3. Upon successful change, give users option to log in, or go straight to the store
8. Don’t want staff members to be able to remove their accounts
   1. Create a check in the profile page to disable the delete button for staff members
9. Want to have an easy way of getting all email addresses of registered users for the newsletter
   1. Create a download button for a csv file with mail addresses

## Derived stories yet to be implemented

The stories below came about as the site was being built. Due to time constraints, these have not yet made the cut and are effectively on the backlog at the time of writing.

1. As a user I want to be able to see my order history
2. As a user I want to be able to save my cart for later
3. As a user I want to be able to talk with others about Lego
4. As a user I want my address information to be saved for later use
5. As site owner I want to modify products outside of Django Admin
6. As site owner I want to download a full order history for administrative purposes
7. As site owner I want to have more payment options available
8. As site owner I want to be able to process the newsletter within the site

## Mockups and UX considerations

### Mockups

As with every project so far, mocking up was done the old-school way using pen and paper. Mostly because of the extremely gratifying feeling of crumpling up any mistakes and drawing again. Maybe that’s just me though ☺

<IMAGES>

### UX considerations

Initially the idea was to create the site using the Bootstrap 4 framework. This progressed up to the point where I personally felt a distinct dislike for the way Bootstrap presents on screen.

At that point I decided to redo the entire site using MaterializeCSS. The original Bootstrap versions of the site pages, as well as a write-up of how to go back to using those, can be found in the root of the repository.

The initial idea, based on the user stories thought up, was to create a site with a category list on the left hand side of the page, and the items per category to the right of that. However, while that would work well enough on desktops, my experience is that sites using that tend to scale less well.

With that in mind, I decided to have a simple navigation bar that scales well across devices, providing the most important links for the site.

These links are: **Home**, **Products**, **Contact**, **Sign In / Sign Out** and of course, the **Cart**. Limiting the numbers of items directly in the navbar this way allows for there not being a need to collapse the menu, making it identical across devices (albeit less wide on mobile devices).

The colours used throughout the site are based on the original logo for the actual shop currently out there, with additions in milder tones so not to make the site scream at you.

To avoid too much white on the pages, an off-white is used as well to be easier on the eyes.

The colour red used throughout is #c9190a, which is the same red as in the logo.

The colour yellow used is #faec06, which is the same yellow as in the logo.

The off-white is #f7f7f7

The lighter yellow is the lighter tone of the yellow used, #fcf69c

The light blue was selected as alternative for what would be a sickly pink, # b6eafc.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| #c9190a  Red | #faec06  Yellow | #f7f7f7  Off-white | #fcf69c  Light Yellow | #b6eafc  Light Blue |

These colours persist throughout the site, only adding plain white and black into the pages.

Links in the navbar are accompanied by an icon, the standard composition is yellow for the icon, white for the text. The same applies to the buttons on pages:

Red background, yellow icon and white text.

In order to create a sharp contrast, cards used throughout the site are given a 1px red border.

Pulldown menus for categories and user menu, change colour when hovering over items in the list, to the lighter yellow. This is less intrusive than the hard yellow, but still clear to see.

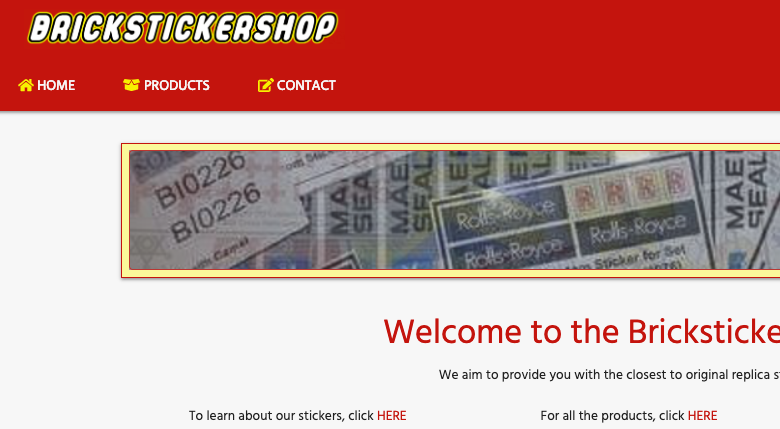
Overall, the idea was to have a site that is as straight-forward as possible, where adding items to the cart changes the button to show the item is in the cart.

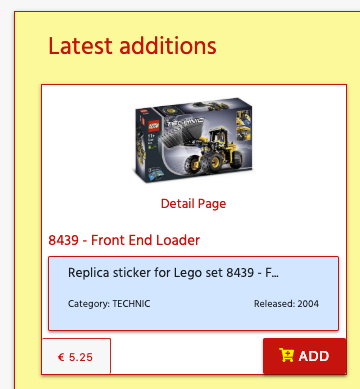
In the cart itself, items can be changed for quantity, or just deleted from the cart.

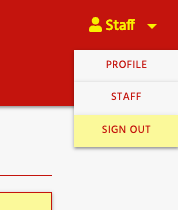
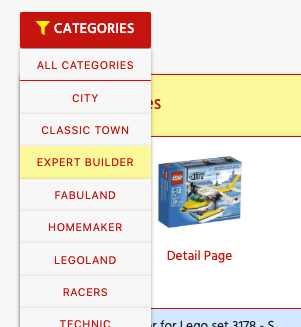
This extends to the checkout, where customers can still modify the contents of the cart.

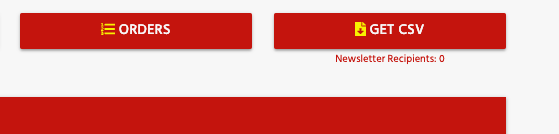
The footer, present on every page, allows access to general information at all times.

#### Sample site images



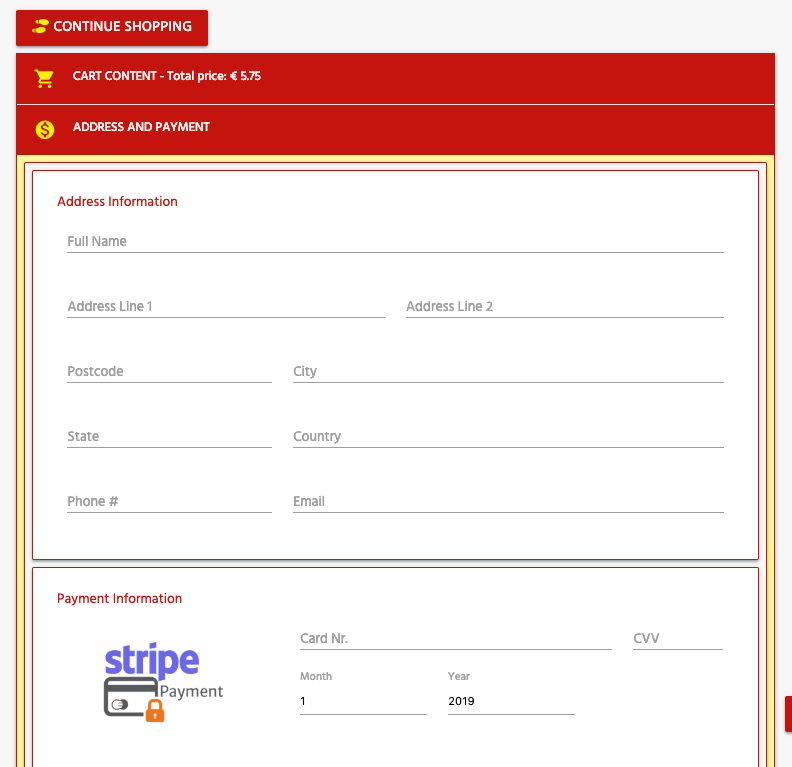


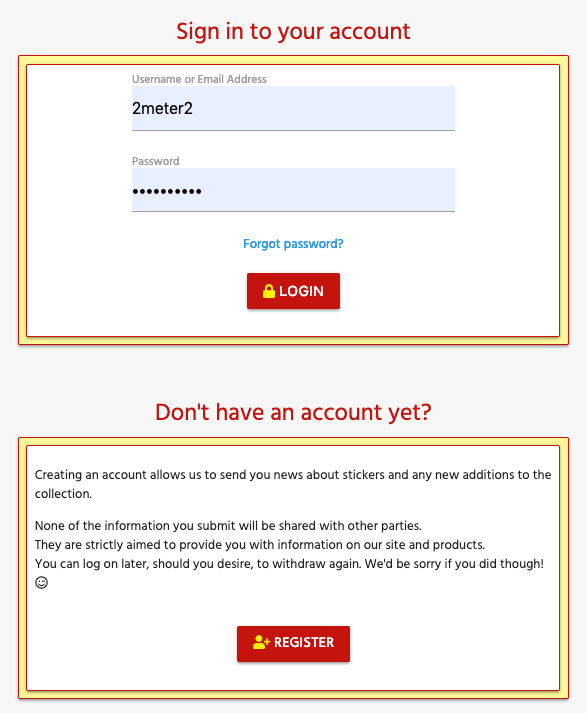
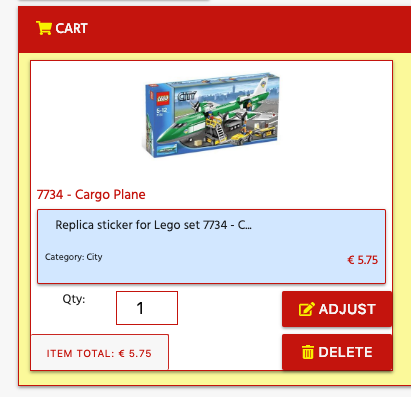


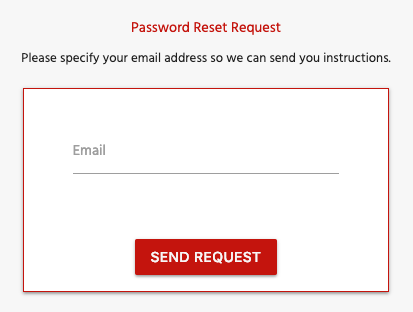
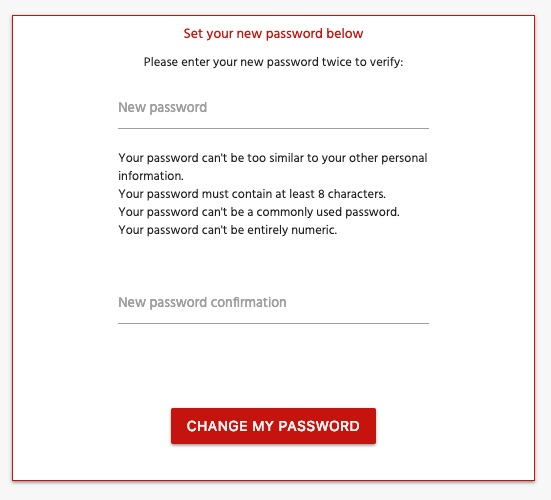


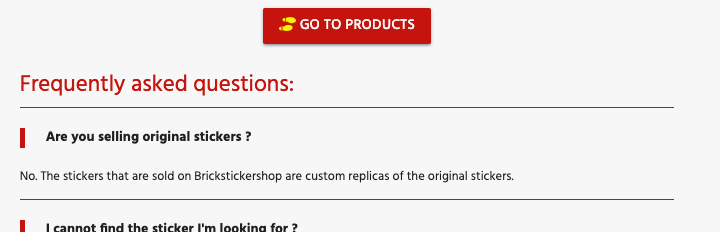


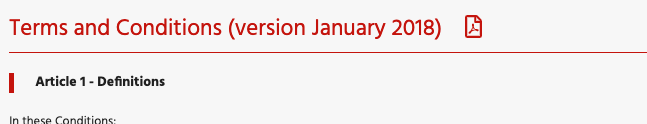










# Testing

## Manual Testing Table

The table below describes the tests that were performed manually, and extensively. In order to test the staff and superuser actions, see [deployment](#_Deployment) on how to create a development environment. I will not give out user, staff and admin keys. A standard user can be created on the deployed application: [Register an account](https://bss-msp-4.herokuapp.com/accounts/register/) That way you can use your own mail address. This will also allow you to test password resets.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Actions | <anonymous>  unregistered visitor | testuser1  registered user | testuser2  staff member | 2meter2  superuser |
| view opening page | x | x | x | x |
| add most recent items to cart | x | x | x | x |
| use nav items | x | x | x | x |
| view footer pages | x | x | x | x |
| view cart | x | x | x | x |
| adjust cart | x | x | x | x |
| check out | x | x | x | x |
| send contact form | x | x | x | x |
| add product comment | x | x | x | x |
| go to sign in | x | x | x | x |
| register account \* | x |  |  |  |
| perform sign in |  | x | x | x |
| user.is\_active FALSE sign in \*\* |  |  |  |  |
| view profile page |  | x | x | x |
| perform sign out |  | x | x | x |
| delete account \*\*\* |  | x |  |  |
| view staff page |  |  | x | x |
| view add product page |  |  | x | x |
| view add FAQ page |  |  | x | x |
| view orders page |  |  | x | x |
| add product |  |  | x | x |
| add FAQ |  |  | x | x |
| mark order processed |  |  | x | x |
| download csv |  |  | x | x |
| use Django Admin |  |  | x | x |
| perform user management |  |  |  | x |
| edit FAQ |  |  | x | x |
| edit product |  |  | x | x |
| edit order |  |  | x | x |
| delete order |  |  |  | x |
| delete order-item |  |  | x | x |
| delete product |  |  |  | x |
| delete FAQ |  |  | x | x |
| edit product comment |  |  | x | x |
| delete product comment |  |  | x | x |

\*: once authenticated, if users try to manipulate the URL, they’ll get directed with message

\*\*: users marked as inactive will receive an error

\*\*\*: initial button disabled, user must agree to make it active. Staff members will not be able to use this, not even through manipulation of page HTML

## Manual Testing Scenarios

The following are more extensive descriptions of manual tests that were performed throughout development.

### Unregistered user

* Gets redirected to sign in page when trying to access pages viewable to signed in users only, e.g.:
  + Staff page
  + Profile page

### Unregistered user / Registered user (not signed in)

* Can view sign in page
* Can access forgotten password link
* Can request password reset (no mail sent to unregistered user)
* Can complete password reset (registered user)
* Can view registration page
* Receives feedback on sign in errors
* Receives feedback on registration errors

### Unregistered user / Registered user (signed in)

* Can access all information pages
* Can access Products page
* Can filter by category
* Can filter by search words
* Can add product to cart
* Gets visual confirmation that item is in the cart
* Can view product detail
* Can go to cart from product detail page if item is in the cart
* Can view cart items
* Can adjust quantity of cart items
* Can delete cart items
* Can move to checkout
* Can adjust quantity of cart items during checkout
* Can delete cart items during checkout
* Can fill out forms
* Receives feedback on errors in fields
* Can submit order
* Receives email confirmation of order
* Can fill out contact form
* Receives feedback on errors in fields
* Can send contact form
* Receives transcript of submitted contact form in mail

### Regular registered user (signed in)

* Gets re-directed when trying to access staff only pages with error message
* Can access pulldown menu in navbar
* Can choose between profile and sign out
* Can view profile
* Can agree to delete account
* Can delete account
* Can’t log on after account deletion

### Staff registered user (signed in)

* Has added option ‘staff’ in navbar pulldown menu
* Can access staff page
* Can process orders
* Can add products
* Receives feedback on form errors for product
* Receives feedback on success or failure of addition
* Can add FAQ items
* Receives feedback on form errors
* Receives feedback on success or failure of addition
* Can download a csv file with registered users’ email addresses

### Django Admin access

* Unregistered users / registered users (not signed in) are redirected to Django login page
  + Registered users (not staff) receive error message on sign in attempt
  + Registered users (staff) are logged in
* Staff members do not see ‘Authentication and Authorization’ section
* Superuser sees ‘Authentication and Authorization’ section
* Staff members can:
  + View Orders
  + Edit Orders
  + Edit Order Items
  + Can**not** add Orders in Django Admin
  + Can**not** delete Order Items
  + Can**not** delete Orders
* Staff members can:
  + View Faqs
  + Add Faqs
  + Edit Faqs
  + Delete Faqs
* Staff members can:
  + Add Products
  + View Products
  + Edit Products
  + Add Product comments
  + Edit Product comments
  + Delete Product comments
  + Can**not** delete Products
* Superuser(s) can:
  + Add/View/Change/Delete groups
  + Add/View/Change/Delete users
  + Add/View/Change/Delete orders and order-items in them
  + Add/View/Change/Delete Faqs
  + Add/View/Change/Delete Products

For testing credit cards, the following credit card data can be used:

|  |  |  |  |
| --- | --- | --- | --- |
| Card Number | Card Type | Payment? | Shows |
| 4242424242424242 | Visa | Yes | Yes, success banner |
| 5555555555554444 | Mastercard | Yes | Yes, success banner |
| 378282246310005 | AMEX | Yes | Yes, success banner |
| 4000000000000127 | Visa | No, cvc error | No, error banner |
| 4000000000009995 | Visa | No, funds error | No, error banner |
| 4242424242424241 | Visa | No, invalid card number | No, form error Stripe |
| expiration date in past | Any | No, invalid date error | No, form error Stripe |
| missing fields | Any | No, missing data error | No, form error HTML |

## Automated Testing

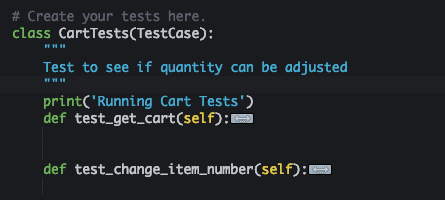
The following is a list of apps and automated tests in them:

### Cart

The cart app has two tests added to it.

The first test simply tests if the is shown.

The second test tests modification of the quantity of items in the cart. Adding and removing items to the cart was tested manually extensively, this test ensures that changing quantities works throughout.

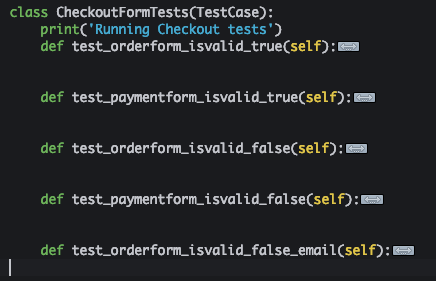


### Checkout

The checkout app has five tests. These are tests for validity of forms.

Two tests are done with all valid data, two tests are done with a field missing.

The fifth test presents a fully filled Orderform, but has an error in the mail-address (missing @) and thus fails the form (passing the test).

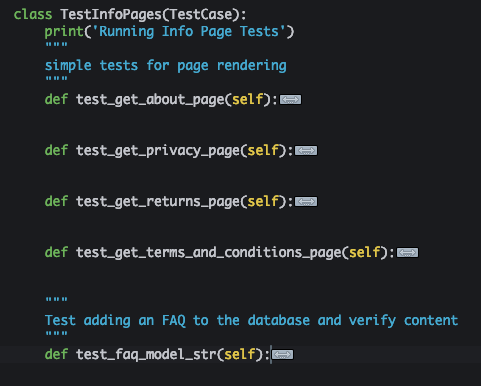


### Infopages

The infopages app has five tests.

Four tests concern themselves with page rendering.

The fifth test adds an FAQ item to the database and then reads back from the database checking the content.

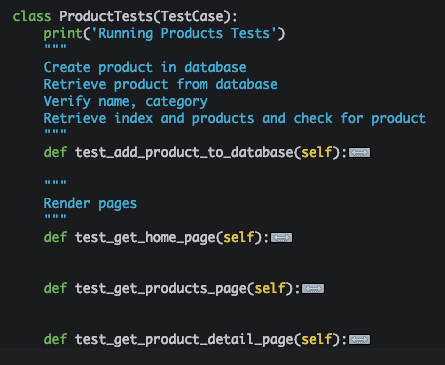


### Products

This app contains four tests. Three tests are only returning the page.

The fourth test (first in the file), performs several tasks:

* Add a product to the database
* Retrieve that product and check naming
* Verify product name
* Verify product category
* Render index page and check if product is there by name check
* Render products page and check if product is there by category check



## Travis CI

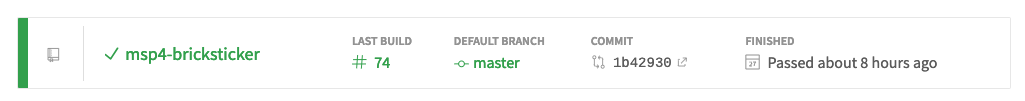
To aid in testing and verifying everyone works as intended, test are run on every deployment using Travis, regardless of whether the views and models were touched.

Any and all changes need to be tested on deployment, Travis deals with this.

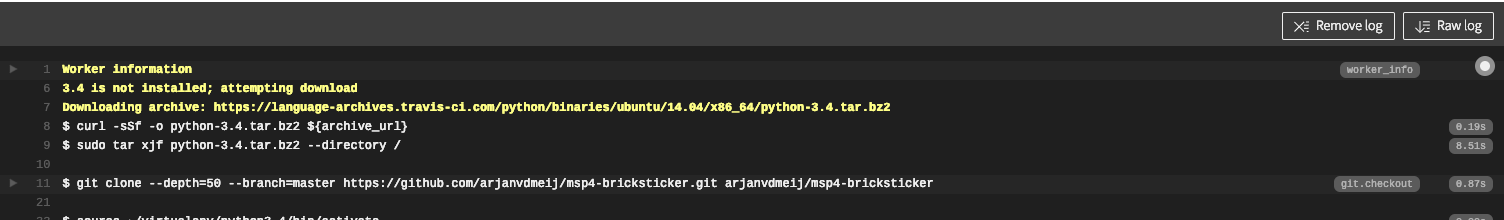
In order to have Travis perform the tests, the github repository has been linked to Travis and whenever changes are detected, Travis automatically runs the tests.

Whenever tests fail, Travis sends an email notification of this.

Current status in Travis:



Example of a Travis log:



…<break>…



# Deployment

## Detailed: deployment on local environment

To do an installation of the application perform the following steps:

* Download and unpack, or clone the repository from Github
* Set the environment variables below in **~/.bashrc** using your favorite editor   
  (be it vi or nano, I don’t judge) and of course providing real values.

*For development, the yellow lines are optional.*  
After, make sure to refresh using **. ~/.bashrc** from the prompt, or closing and opening the terminal again!

|  |
| --- |
| export SECRET\_KEY='I-am-secret-hear-me-roar'  export EMAIL\_HOST='smtp.example.com'  export EMAIL\_HOST\_USER='fake@example.com'  export EMAIL\_HOST\_PASSWORD='fake-password-this'  export STRIPE\_PK='pk\_test\_key'  export STRIPE\_SK='sk\_test\_key'  export DATABASE\_URL='fake-db-url'  export AWS\_ACCESS\_KEY\_ID='fake-AWS-key'  export AWS\_SECRET\_ACCESS\_KEY='fake-AWS-secret-key'  export AWS\_STORAGE\_BUCKET\_NAME='fake-bucket-name'  export AWS\_REGION\_NAME='fake-region-name'  export ENVTYPE='development or production' |

After that, add the following alias into **~/.bash\_aliases**, again using your favorite editor again:

|  |
| --- |
| alias run="python3 ~/workspace/manage.py runserver $IP:$PORT" |

*Substitute ~/workspace/ with whatever working directory you are using.*  
Make sure to refresh aliases as well: **. ~/.bash\_aliases**

* Run the following commands in succession and follow prompts where needed:

|  |
| --- |
| sudo pip3 install -r requirements.txt  python3 manage.py makemigrations  python3 manage.py migrate  python3 manage.py createsuperuser |

* At this point you have a very pristine working version of the site
* Running the server is as easy as entering **run** on the command prompt now. Logging in using the superuser you created allows you to add products to the database through the staff pages. The same goes for FAQ items.  
  Comments can then be added to the new products on the product detail pages as well.
* There will not be any orders. Placing orders will also not be possible unless the stripe credentials are added into the environment variables.

## Environment variables

The table below shows every environment variable used:

|  |  |
| --- | --- |
| SECRET\_KEY | Required by Django. For security reasons, this variable should never be present in files, so it was moved to live in an environment variable. Can be whatever your heart desires, but shouldn’t be too simple. |
| EMAIL\_HOST\_USER | Mail address used to send order confirmation, contactforms and password reset related mails. For the purpose of the course, I created a Gmail account and gave it the right access to be used by the application. |
| EMAIL\_HOST\_PASSWORD | Seems obvious enough. Don’t want that in a file |
| EMAIL\_HOST | Your chosen provider. Mine was smtp.gmail.com |
| STRIPE\_PK | Stripe Publishable Key, needs to be obtained at Stripe.com. For the project, the only keys used are the test keys provided by Stripe, as this site is not actually live in the sense of selling items. |
| STRIPE\_SK | Stripe Secret Key, same as above, obtained from Stripe.com. Again, for the projet, only the test keys are in use. |
| DATABASE\_URL | While not needed for development, it is required for production and referenced in the settings.py file. Contains the link to the PostGres database at Heroku for this project. |
| AWS\_ACCESS\_KEY\_ID | Not needed for development, required for production for communication with AWS |
| AWS\_SECRET\_ACCESS\_KEY | Same as above, only needed in production for communication with AWS |
| AWS\_STORAGE\_BUCKET\_NAME | Needed in production, and holds the actual bucket name as defined in AWS |
| AWS\_S3\_REGION\_NAME | Needed in production, holds the region your bucket was created in in AWS |

The colours are used to indicate the variables that are ‘sets’ when used.

*If accounts at Stripe and AWS aren’t readily available, payments can be tested on the Heroku hosted version of the project site.* [*https://bss-msp-4.herokuapp.com*](https://bss-msp-4.herokuapp.com/)

Get working keys to add to the environment variables:

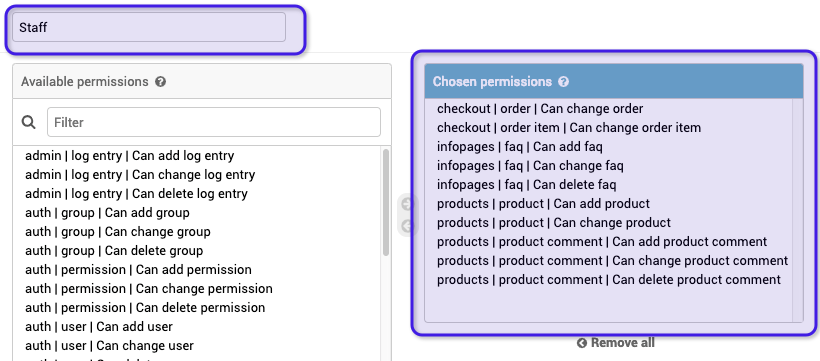
* In order to get mail working, provide host, user and password to be used. Gmail is an easy option.   
  \*\*Note that the port number is set in settings.py, do verify the port number when creating/adding an account to the application. Default is port 587\*\*
* In order to get Stripe working visit [Stripe.com](https://stripe.com/docs/development) and create a developer account.
* In order to get AWS up and running, visit [AWS](https://aws.amazon.com/) and create an account and/or bucket for the site to use.

Alternatively, test things out on the live URL <https://bss-msp-4.herokuapp.com>

The only thing you will not receive there, is the site admin mail for the contact form.

## Group creation in Django Admin

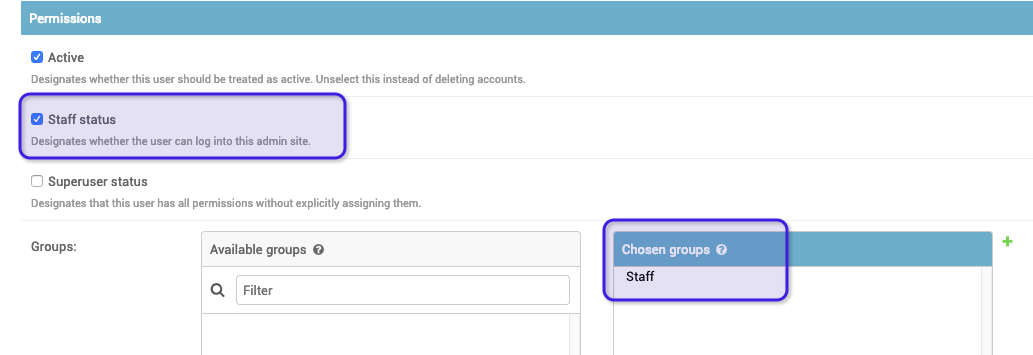
To easily add users to the staff group, create a group in Django admin with the following rights. Add entries as listed below:



Whenever a new user is made staff, they only need to be added to this group, to have their options available to them, outside of user management and other superuser tasks.

Deployment does not create this, and it will need to be performed in both development and production manually. This also allows to create more granular groups in a production setting, however for the purpose of the project the choice fell to one single group.

An example of a non-superuser staff member’s permissions then would be:



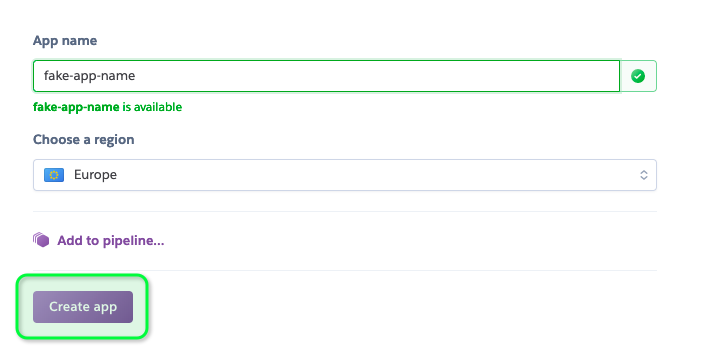
Currently there is no way to do this using the site interface itself, the superuser will need to do this through the Django Admin panel.

# Detailed: running on hosted environments

In order to run the site on a hosted environment some steps must be taken. For the project, Heroku was used, along with AWS for media and static file storage.

## Heroku

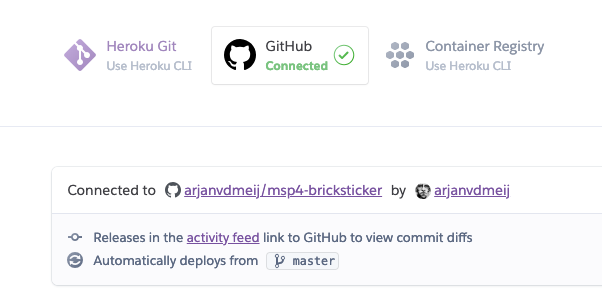
If not available, go to <https://www.heroku.com/> and create an account. Once verified, create a new app:

****

Choose any name you like, provided it is of course available.

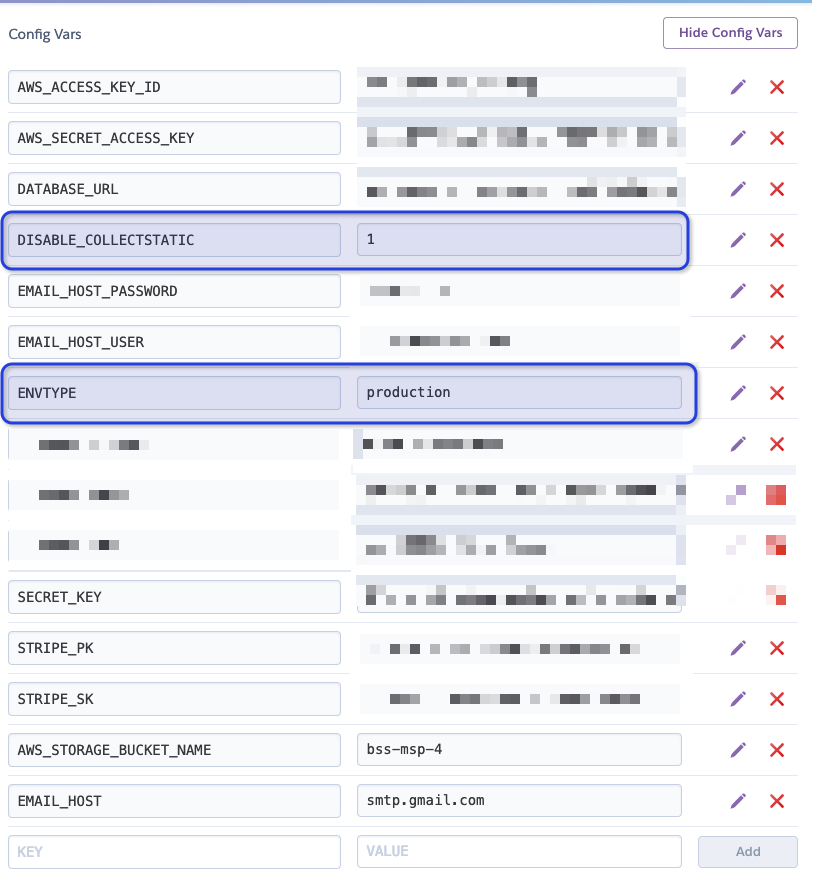
You can choose one of two ways to deploy an app, either directly with Heroku CLI, or using an existing Git repository.

For the project, I have linked Heroku to my Git repository:



I have also enabled Automatic deploys on detection of changes to github, in order to directly push changes live. This is entirely optional however.

Within the settings tab, the config vars entered are:



These variables are the same as they are in the development environment, with one addition, and one notable change:

**ENVTYPE** here is set to production

**DISABLE\_COLLECTSTATIC** is set to **1** to prevent Heroku from trying to collect static files.

The **DATABASE\_URL** is provided from within Heroku, by installing *PostGres* as an add-on to the app.

In the Resources tab, type postgres into the *Add-ons* quick search bar. Select **Heroku PostGres** and the plan as *Hobby Dev – Free*, then click **Provision**.

This will attach the database to your app.

This will automatically fill out the **DATABASE\_URL** environment variable for you, you don’t need to do anything there yourself. This is the only variable to be entered automatically however, the rest needs to be added manually.

The values for the AWS variables are those provided to you by AWS and your self-chosen bucket name.

\*\* *the three blurred variable names and variables are monitoring add-on related and not needed* \*\*

In the Heroku Heroku page, click ‘More’ and select ‘Run console’. This allows you to perform command line tasks. In the available box, enter **python manage.py createsuperuser** and click the Run button.

This will let you create your superuser in the PostGres version of your database. Follow the questions asked in the screen that’s loaded.

## AWS

In AWS, create a bucket that has public access, so uploading of images for products works without problems.

Within S3, create a bucket that holds public access and is accessible for static website hosting.

Follow the tutorials by Amazon to create the bucket, and its accompanying IAM groups, users and policies.

Set the variables in your environment to the values that match AWS. Refresh your terminal after!

From a command prompt, run the command **./deploy.sh** and follow the questions asked. This script will set the environment to make use of AWS temporarily, collect static files and upload them, then revert to the development environment. After that, a push to GitHub will be attempted. This can be cancelled if you wish by using <CTRL>-C when asked for credentials.

**Fallback if the staticfiles directory is created locally:**

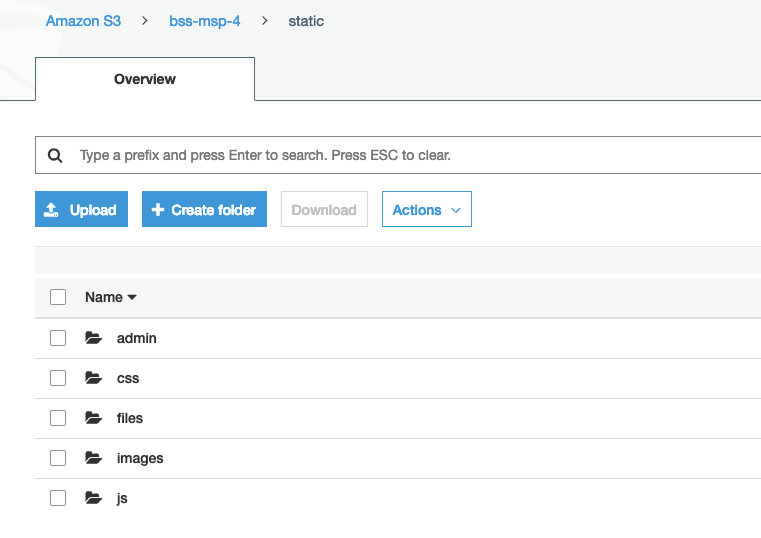
A *staticfiles* folder is created within the root of the project, containing everything that needs to go into AWS.

In the bucket, create two directories:

* *media* (and in it, a directory called *img*)
* *static*

In theory, the media directory would be created automatically when adding content, but given how little effort, it can be done immediately as well.

From the *staticfiles* directory locally, upload the entire contents into *static* on AWS resulting in a view like below:



Make sure your personal AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY are stored as variables for the environment.

The same goes for the AWS\_STORAGE\_BUCKET\_NAME variable and the AWS\_S3\_REGION\_NAME variable.

## Short: bullet list deployment

The following assumes working knowledge of Linux, git, AWS buckets, Heroku apps, GitHub, Django CLI, Django Admin and that you are aware of the environment variables

mentioned [here](#_Environment_variables)

Steps:

* Initialize git locally
* Download and unzip or clone the repository from [msp4-bricksticker](https://github.com/arjanvdmeij/msp4-bricksticker)
* Create a bucket at [AWS](https://aws.amazon.com/console/) with correct settings for public read access
* Create a suitable mail host account
* Create a [Heroku](https://id.heroku.com/login) app and install PostGres add-on on it
* Add Heroku remote to git
  + Alternatively, create remote at GitHub and connect Heroku app to that
* Set environment variables locally (refresh environment) and in Heroku
* Pip install requirements
* Run makemigrations (manage.py makemigrations)
* Run migrate (manage.py migrate)
* Create superuser (manage.py createsuperuser)
* Run server (manage.py runserver)
* Create [group](#_Group_creation_in) in Django Admin
* [Test](#_Testing) functionalities
* Stop server
* Collect static files (run deploy.sh, <CTRL>-C to abort GitHub push if using Heroku Git)
  + If staticfiles directory created locally:  
    manually upload content to ‘static’ directory in AWS bucket
* Deploy to Heroku if not using GitHub (git push heroku master)
* Either from Heroku cli or console on Heroku app page:
  + Create superuser (manage.py createsuperuser)
* Create [group](#_Group_creation_in) in Django Admin
* [Test](#_Testing) functionalities