Aaron Gockley

Argonath Inventory Management System Documentation

codename: Anduin



# Executive Summary

Argonath Inventory Management Software is a point-of-sale program integrated with inventory management software for small businesses that focuses on businesses that do not need receiving processes, it currently lacks payment options, but shows inventory and processes orders.

My project is focused on small businesses, the idea came about because of the small business my dad and brother-in-law are in the process of starting. They are selling beef, and I want to build a system that not only lets orders be entered but also automatically updates their inventory. So the target audience is small companies with a lower number of items. That is the target audience for the current scope of the project anyway. This is designed for businesses that manufacture their own inventory. Whether it be someone who makes crafts, an ice cream company, or a small family-owned meat business.

The way the system works is like this. It is split into two parts. There is the public-facing storefront page, with all the items in the database. Underneath the picture and name of a product, there will be a cost, as well as the remaining quantity in inventory. When you click on the item, it takes you to a page with the item and a form field to add items to the cart, as long as inventory still exists. There will then be a link back to the main inventory. In the cart page, you can alter the amounts in the cart. When you submit the cart, it presents a customer field. When it is submitted it generates a pick ticket. Due to time constraints, this will not have a checkout feature at the moment. This moves to the non-public-facing part of the program.

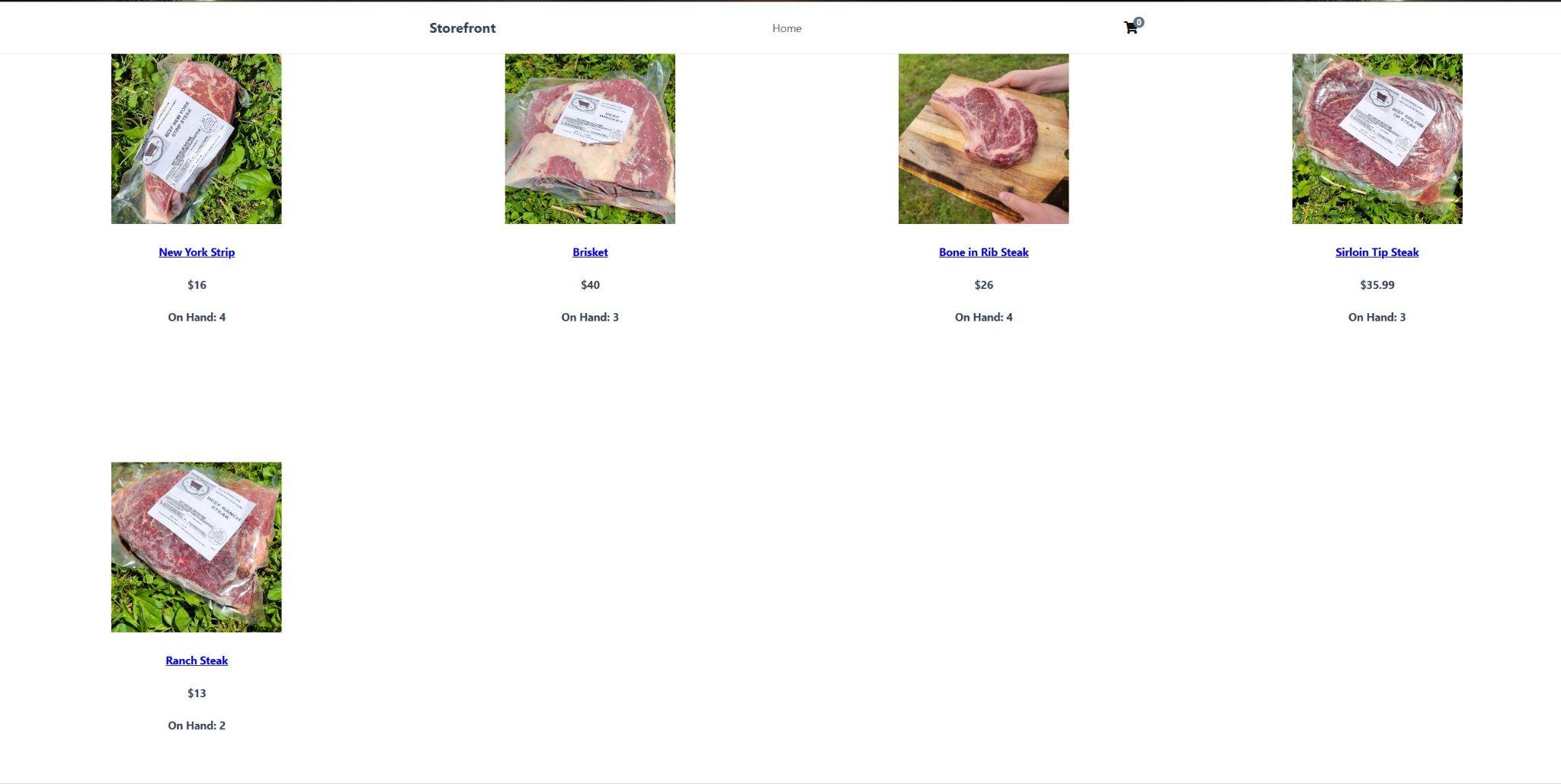
It first checks to see if the customer exists, if it does not it creates a new entry in the customer table, and connects the order to that new customer ID. If it does exist it ties the new order to that ID. The new order is marked as not picked. Inventory gets moved from uncommitted to committed. The order shows up on a screen that generates a PDF with the amount to be picked. Once an order is marked as picked the committed inventory is subtracted from the committed total.

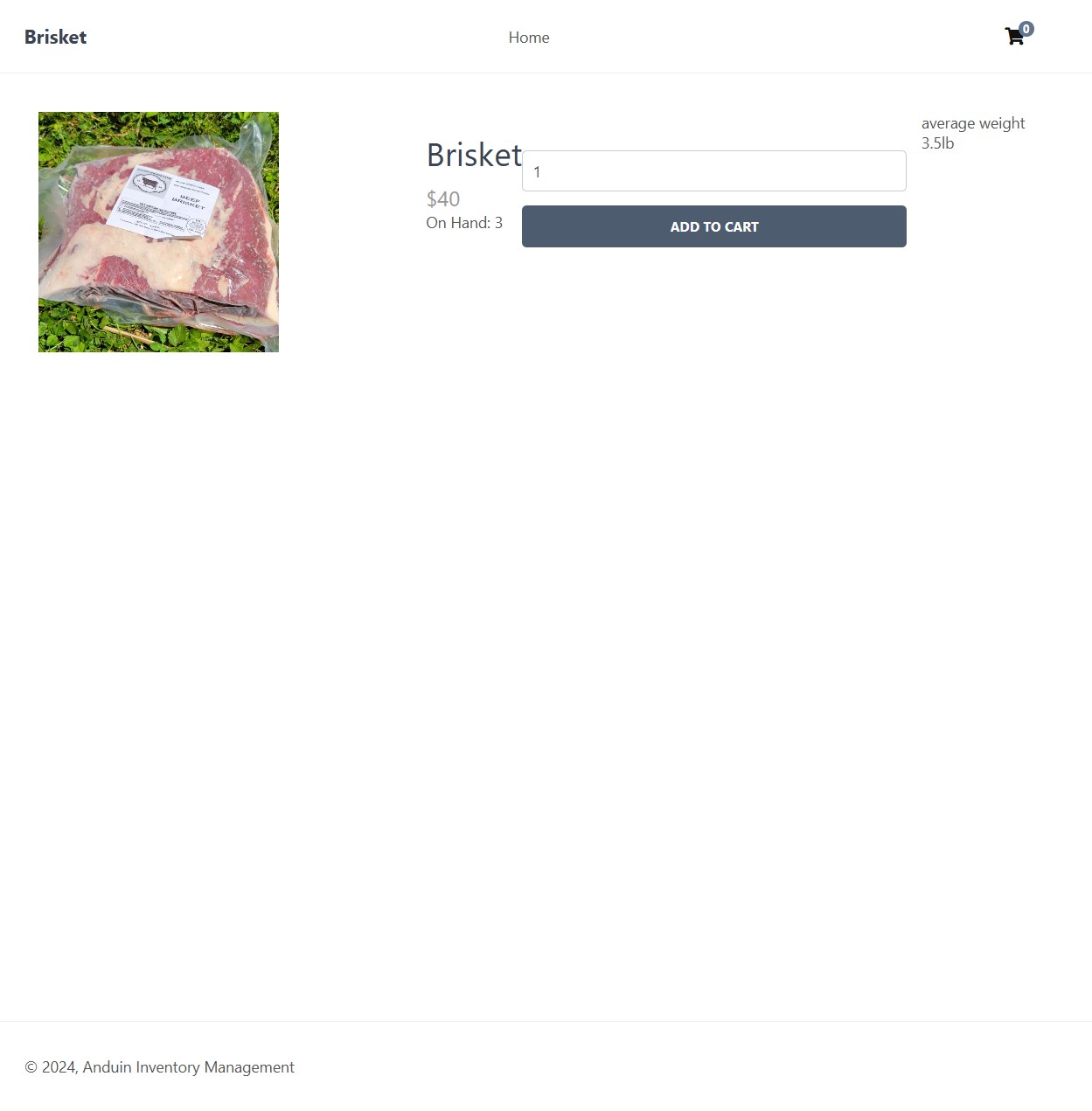
To break down the order to be picked page, it will be generated from all orders that are not picked. It will allow you to cancel an order, which will completely delete it from the database as well as return the inventory to uncommitted. As stated above the system will generate a PDF with a pick list that can be printed. Then you can mark the order as picked. Picked orders do not show up on this page.

The utilities beyond just the order picking fields are I will include a form to add inventory to the store, both adding quantities to existing items as well as adding new items to the database. The last tool is for looking up order history. It lets you search for an email, or click on a customer's name, and then it gives you a printout of all the customer's orders as well as which items were with which order.

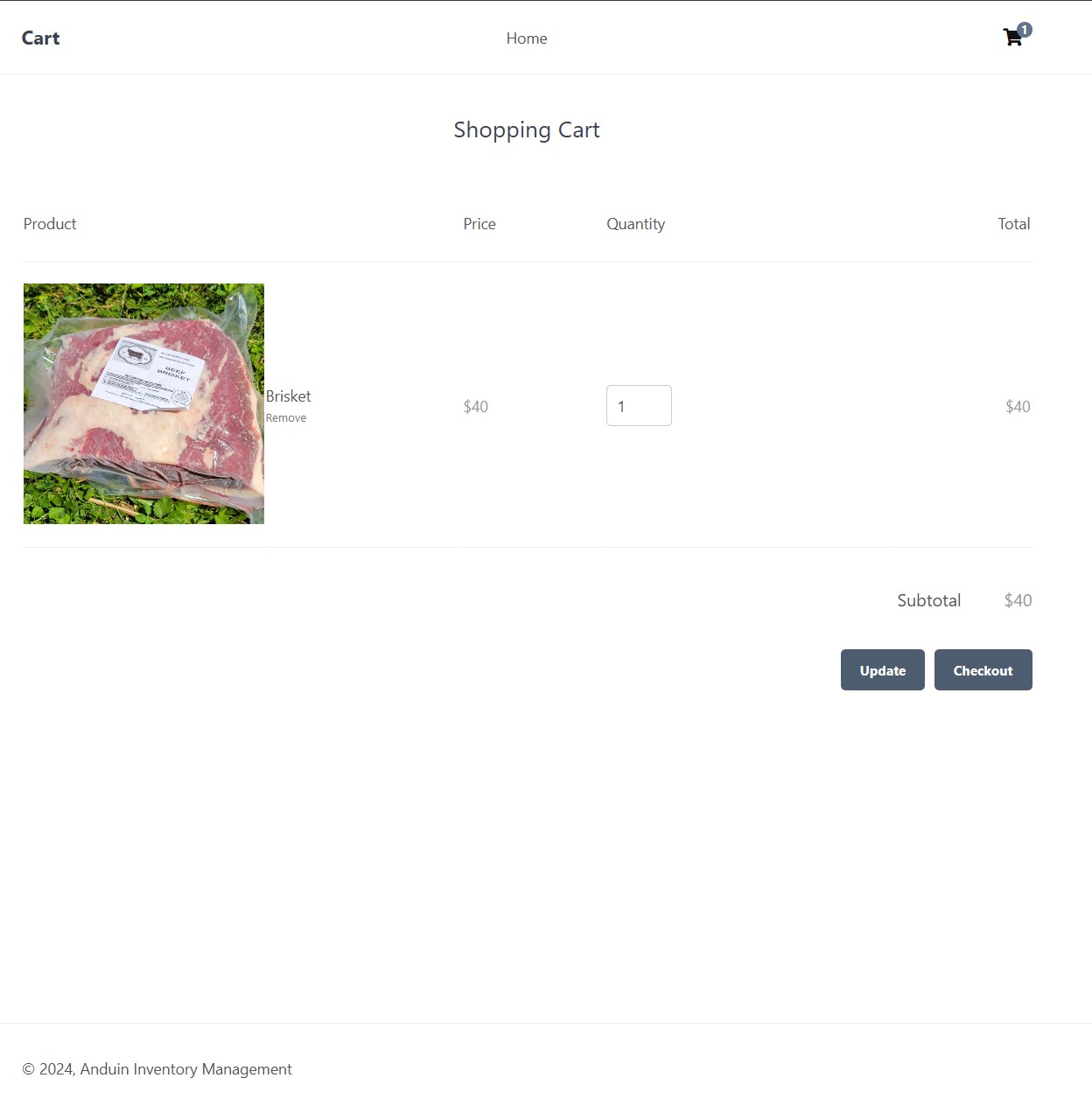
# Placing Order | Customer Places Order

This is the landing page for a customer, it loads all inventory data from the database.

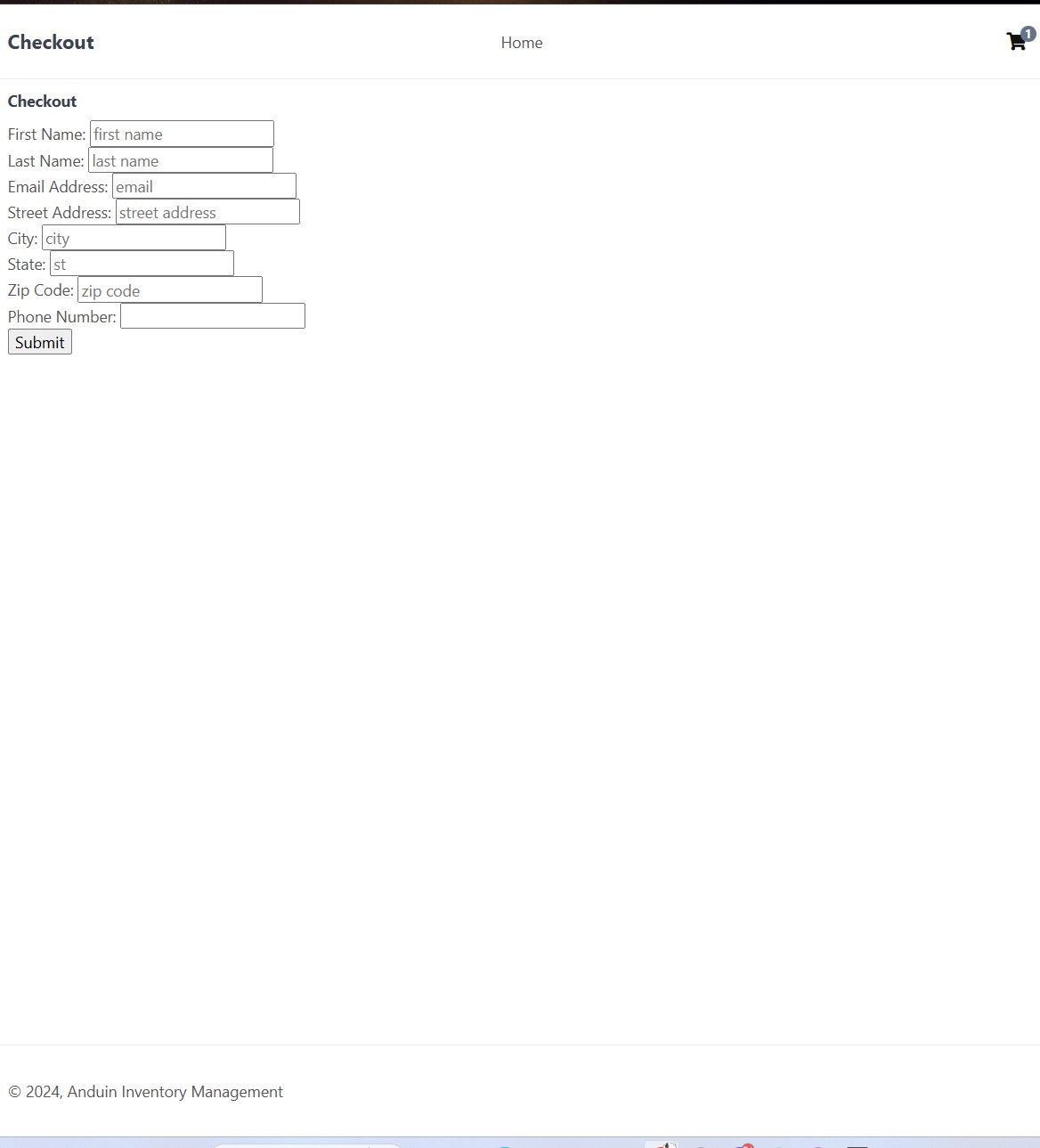


They click on Brisket, and it loads the brisket data to the product page.

They then click on add to cart.

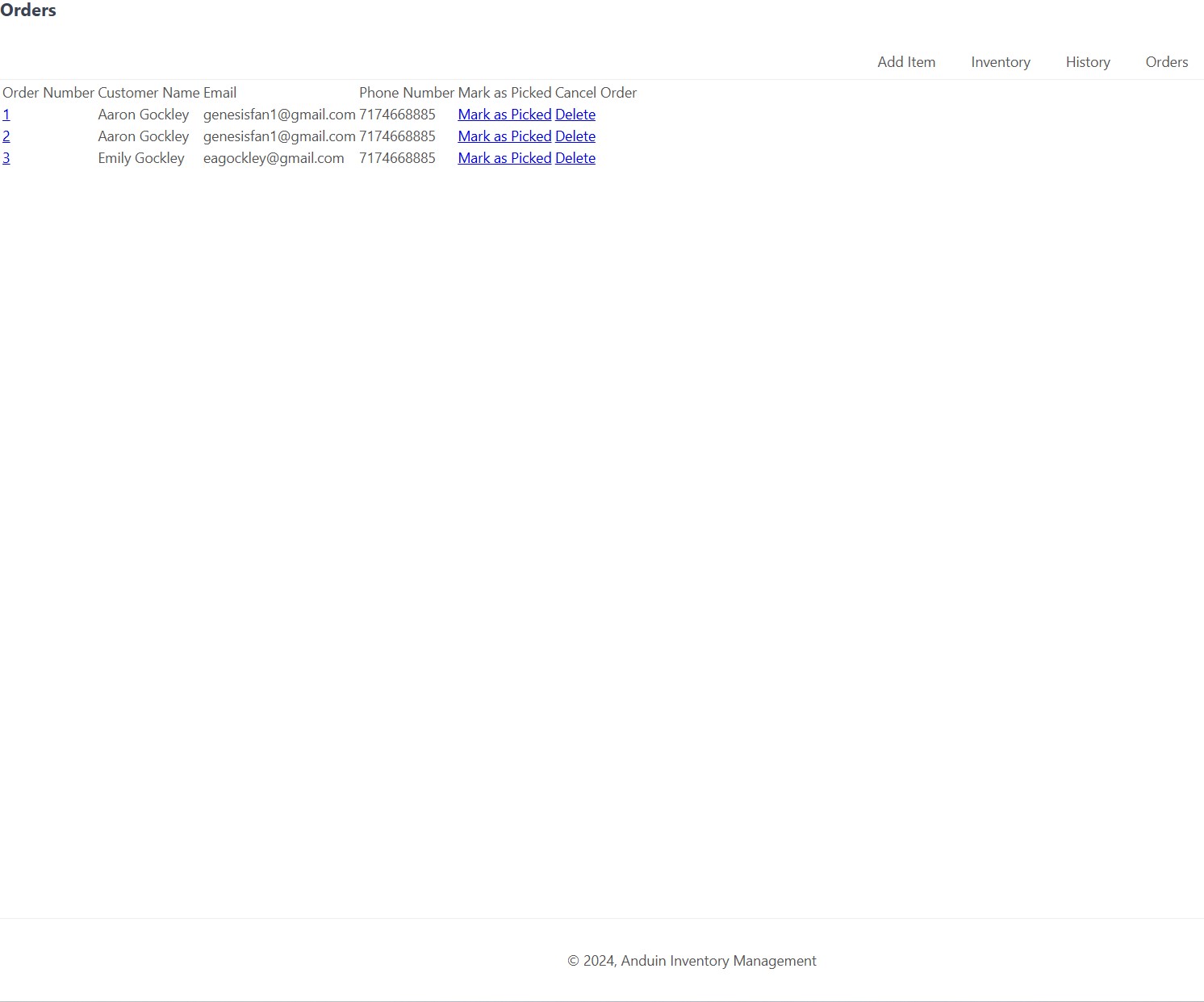


This was all they wanted to order, so they clicked on checkout. This loads the customer information form, which lets them put in their information. If the email address matches one already in the system, it attaches the pre-existing customer ID to the order, otherwise, it creates a new customer entry and ties that customer ID to the order.

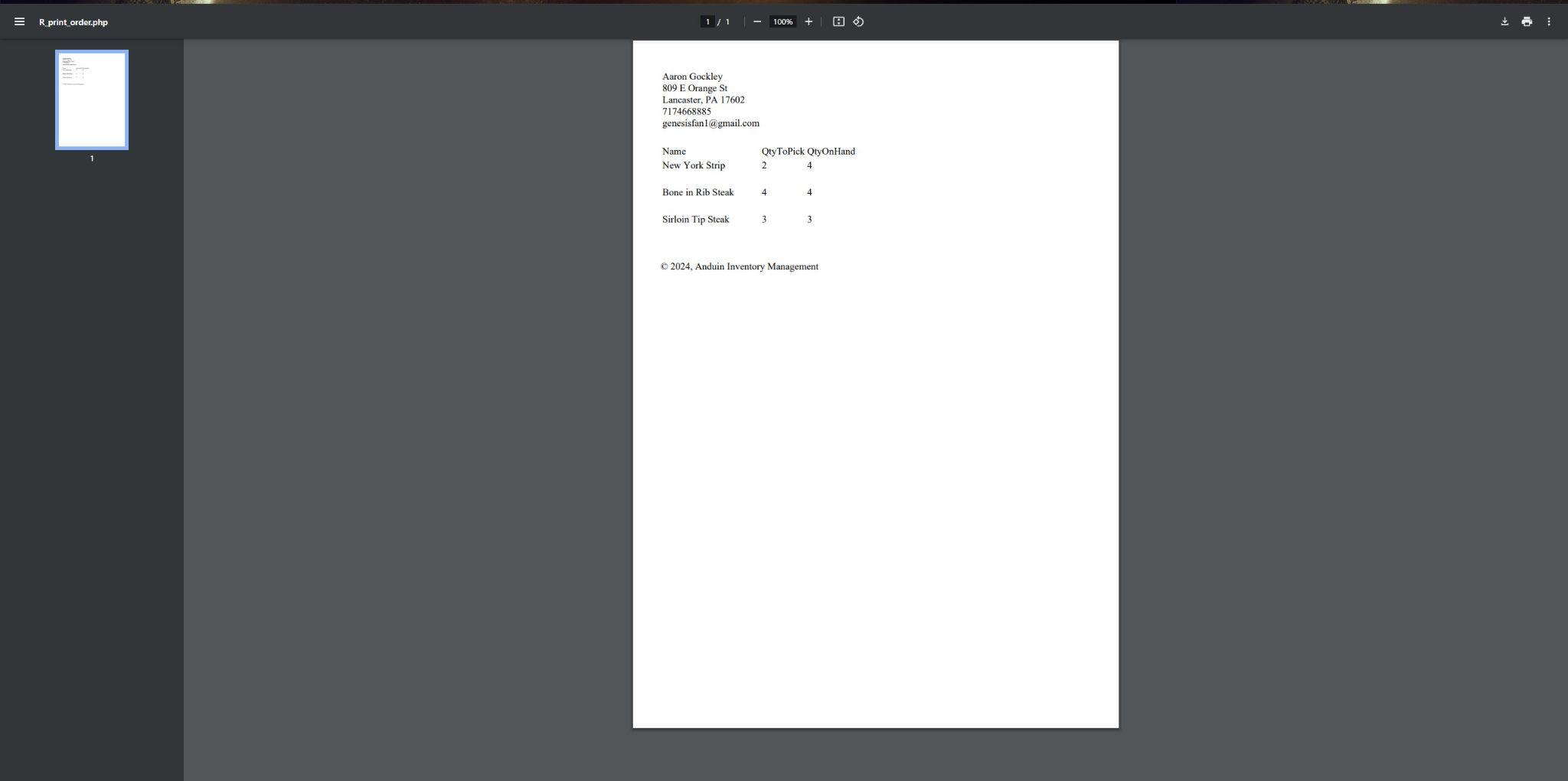


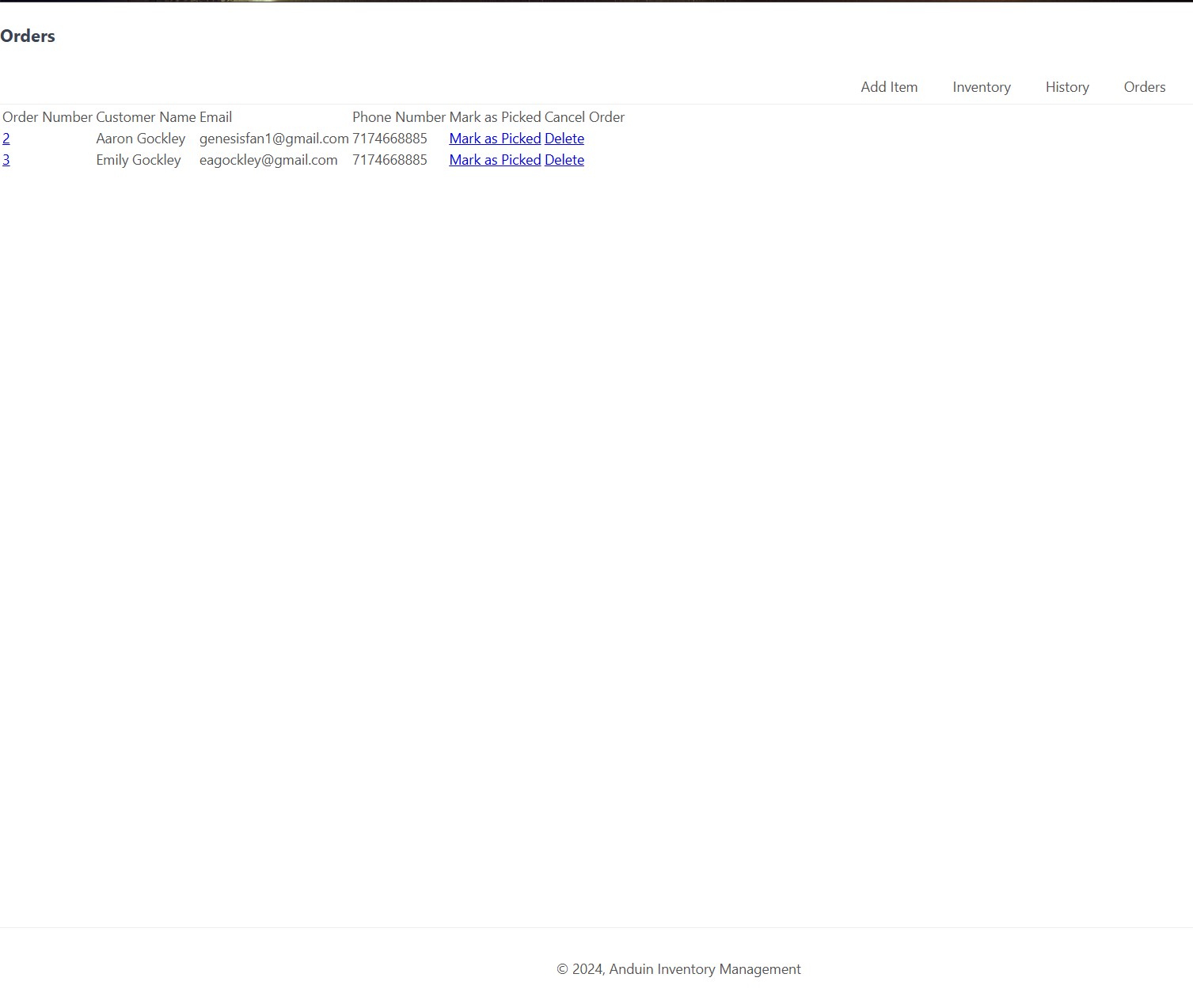
# Processing an Order | The Owner of Store Processes Order

This is the index page for the rear information, it is a list of all of the unpicked orders. You can click on the order number on the left side to print out the order.



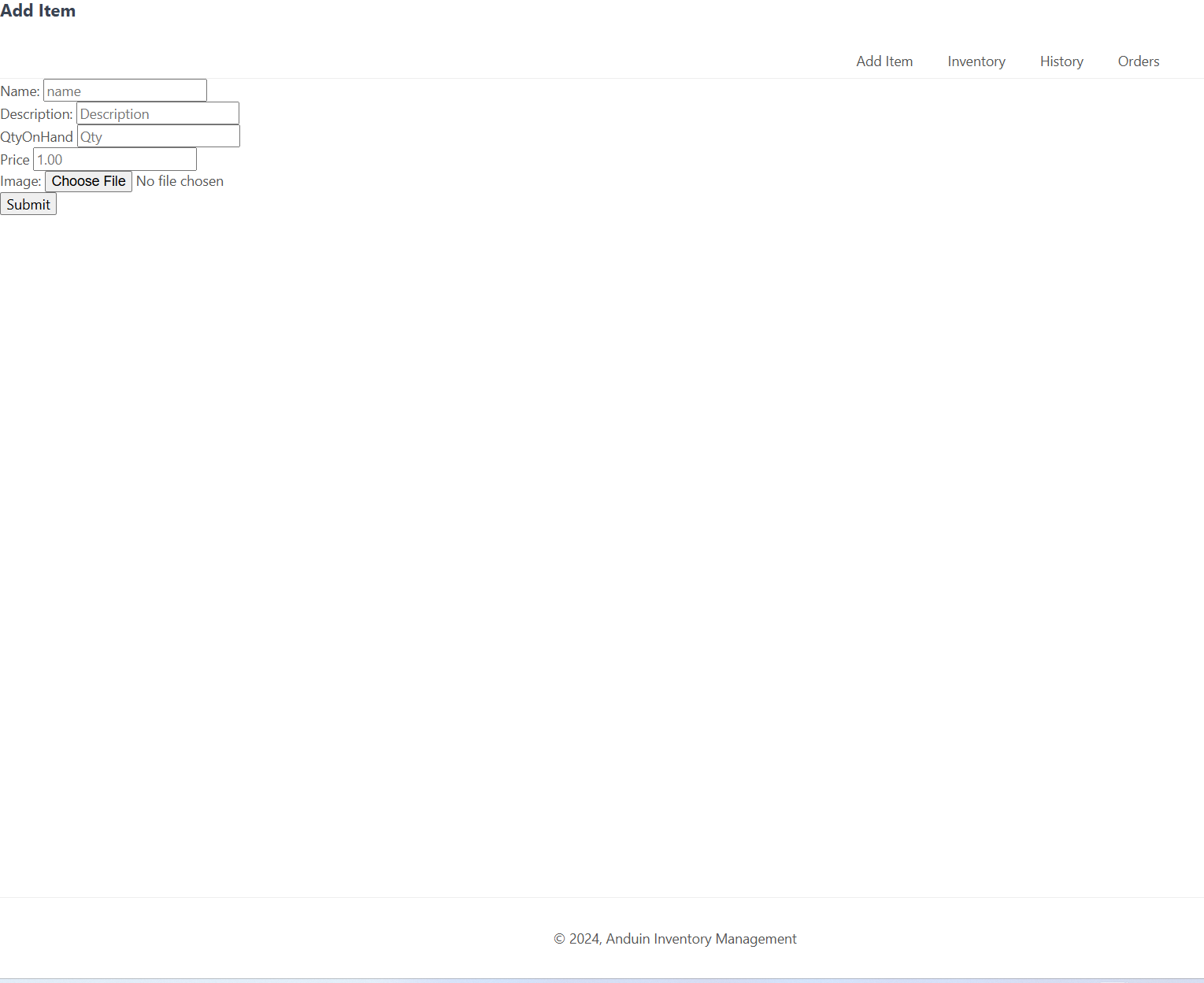
The user then clicks on the order number of order number 1



After printing out the order and picking it, the user then hits back, and then marks the order as picked, after clicking marked as picked the page looks like this. Order number 1 is gone. 

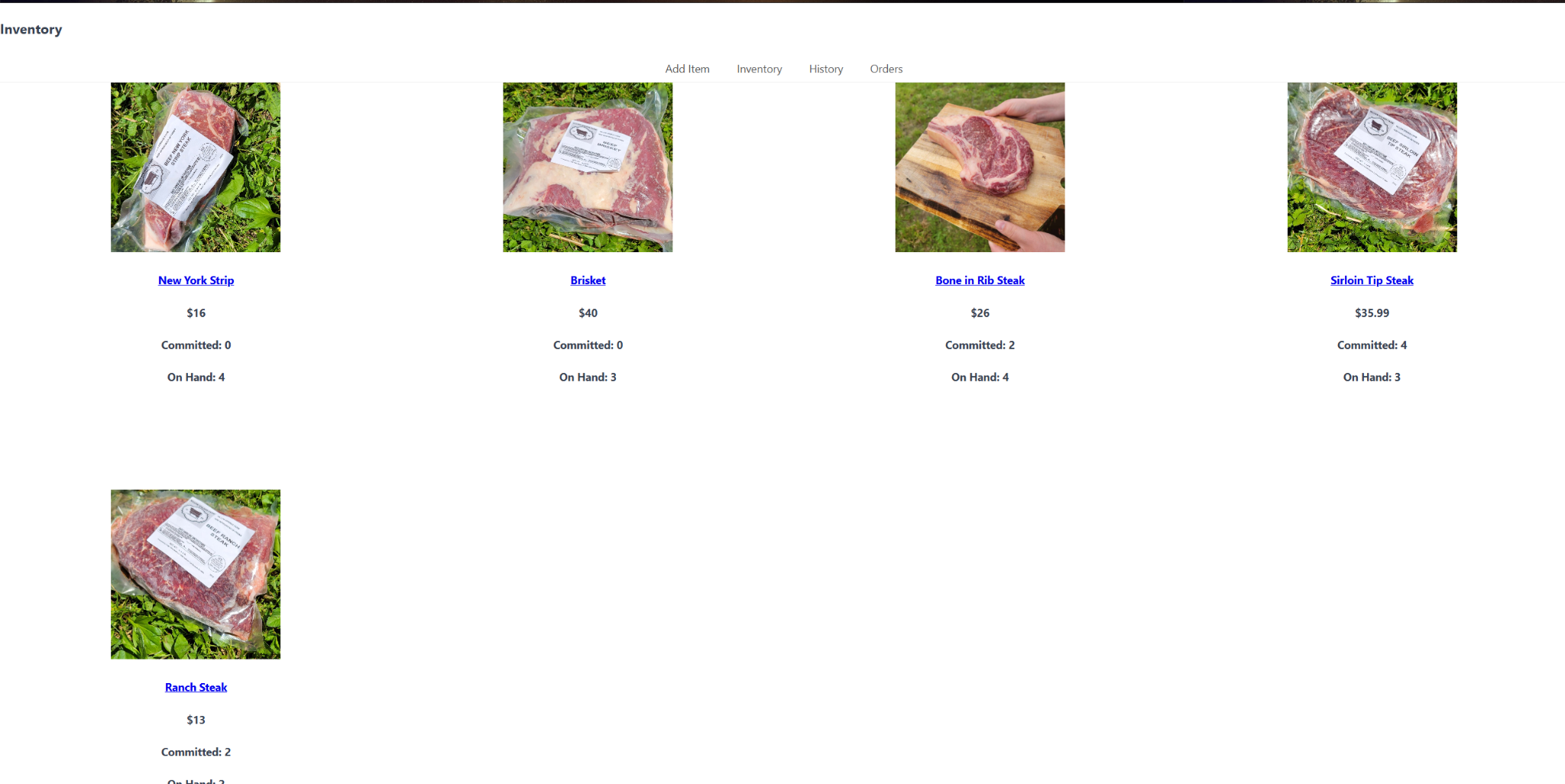
# Adding New Item | User Adds a New Item

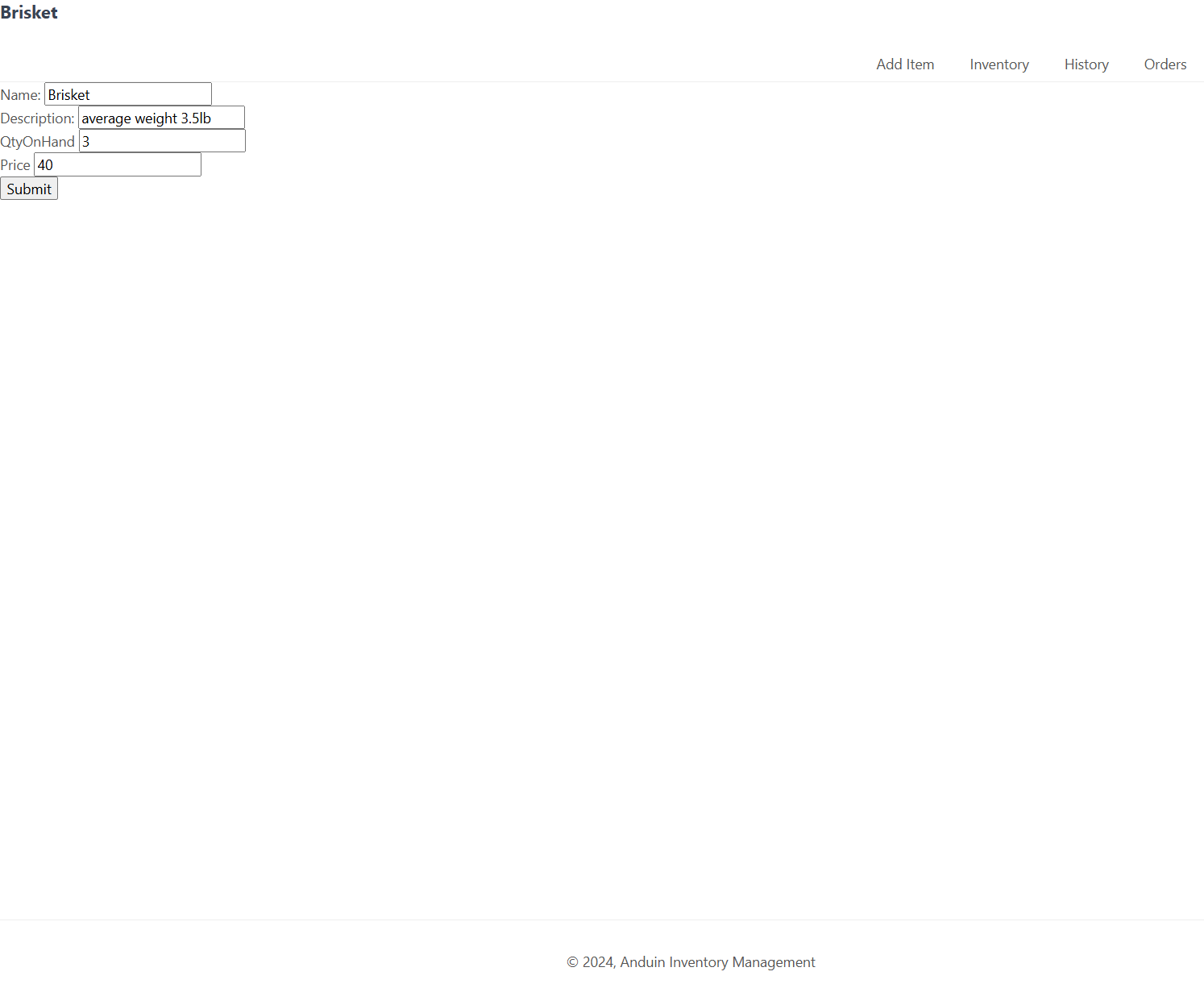
Here a user adds a new item to the database, it has several fields that need to be filled out, and then when you click submit, it adds the item to the database.



# Updating Item Information | Updating Item Information

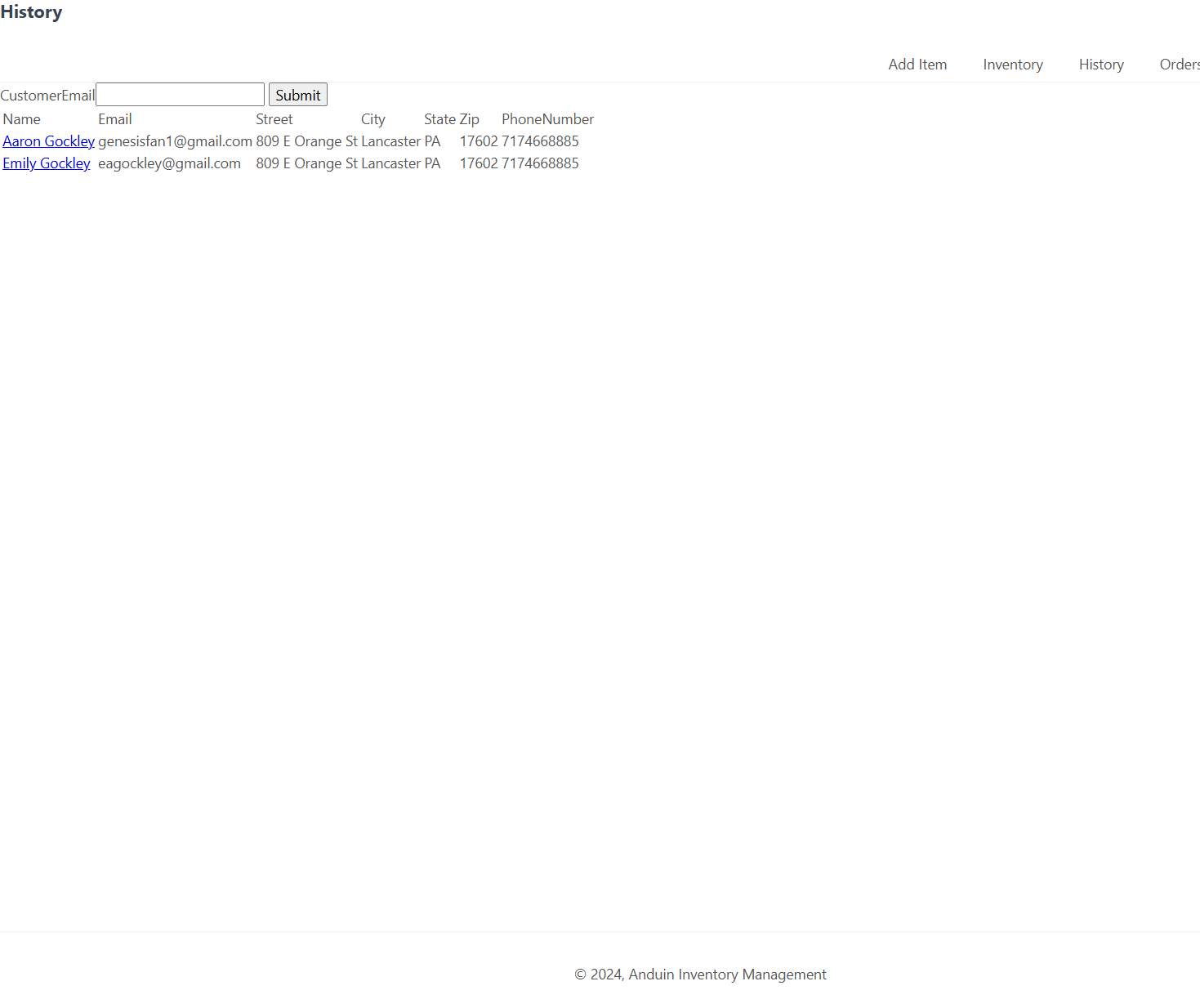
This is for updating information on an item. You begin on the inventory page.



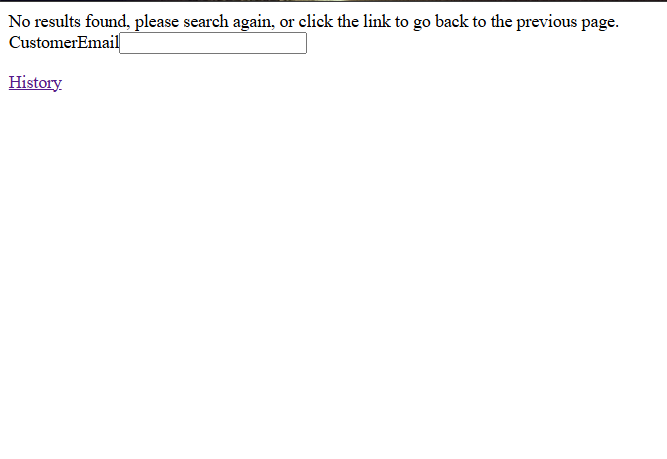
Once you click on the item you want to update, it loads all of its information into a form that allows you to update its information. Here is the brisket, I will be adding one more to inventory, once you click submit, if everything goes through properly, it will go back to the main inventory screen. 

# Customer History | User Searches for Customer History

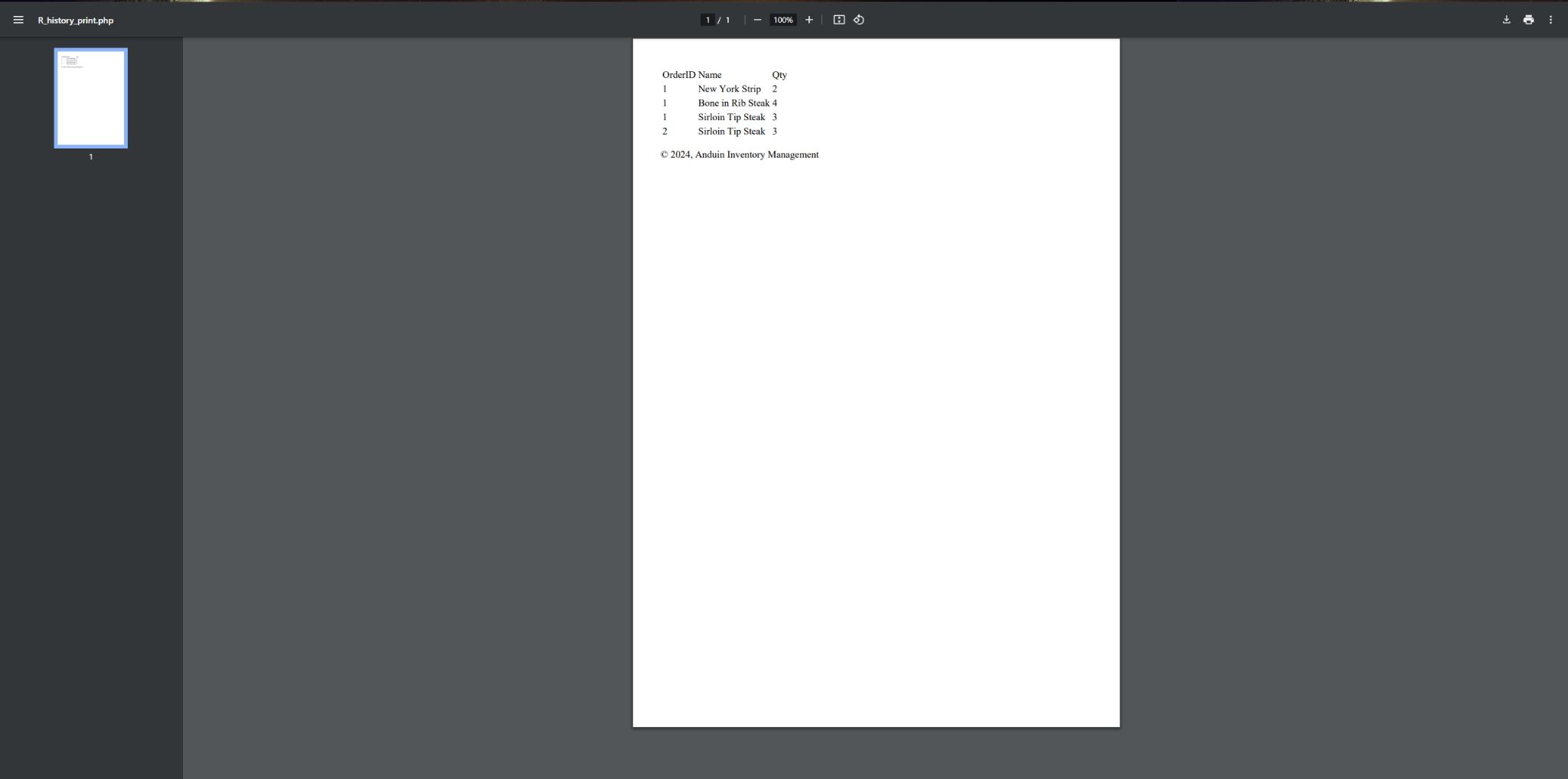
There are two ways to search for customer history, you can either click on their name or search for their email.



Searching for the email checks if the email exists if it does not it gives you the option to either search again or go back to the history page.

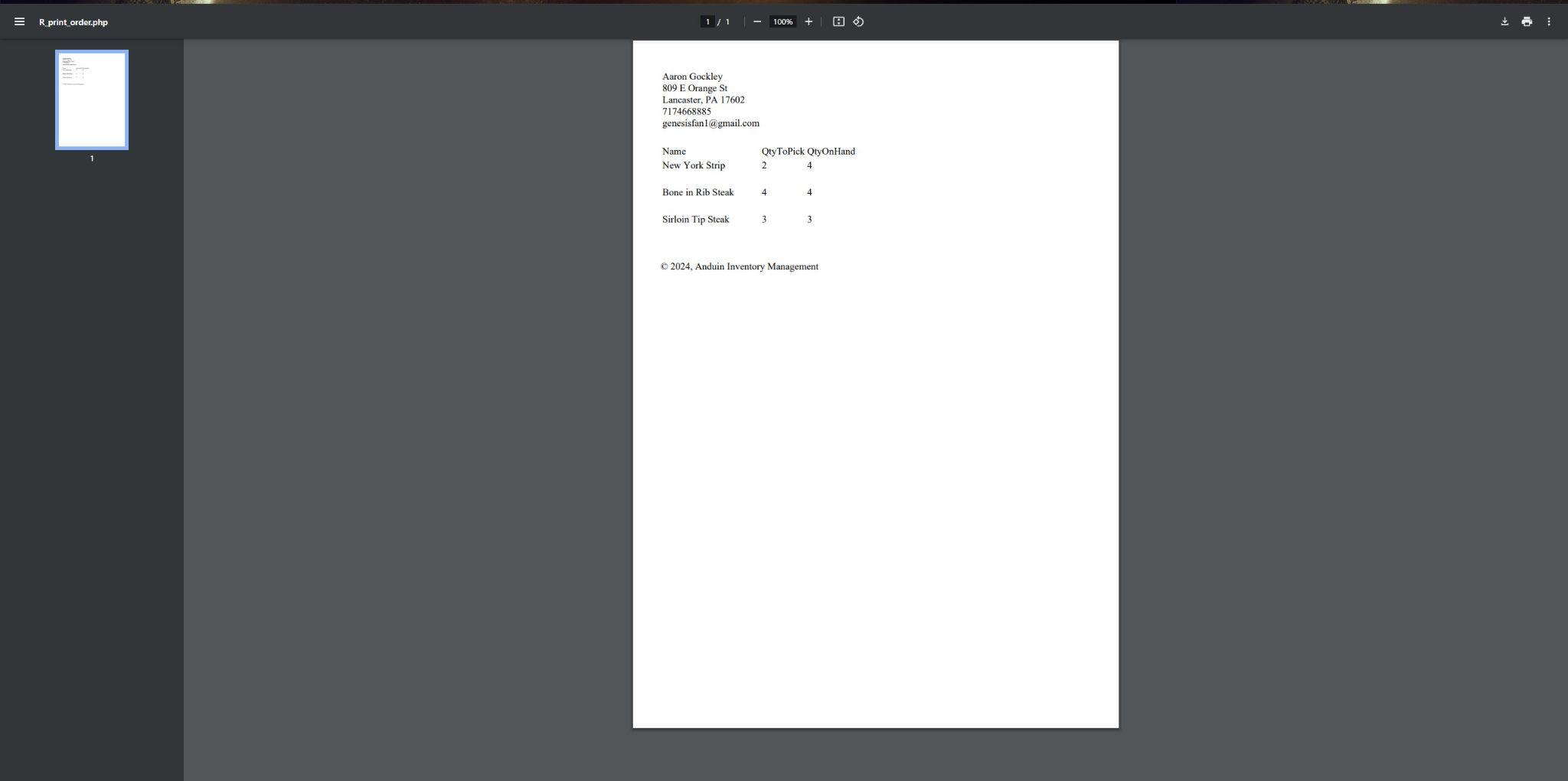


Once you actually check a real customer you get this page, which lets you print out all the items a customer ordered.

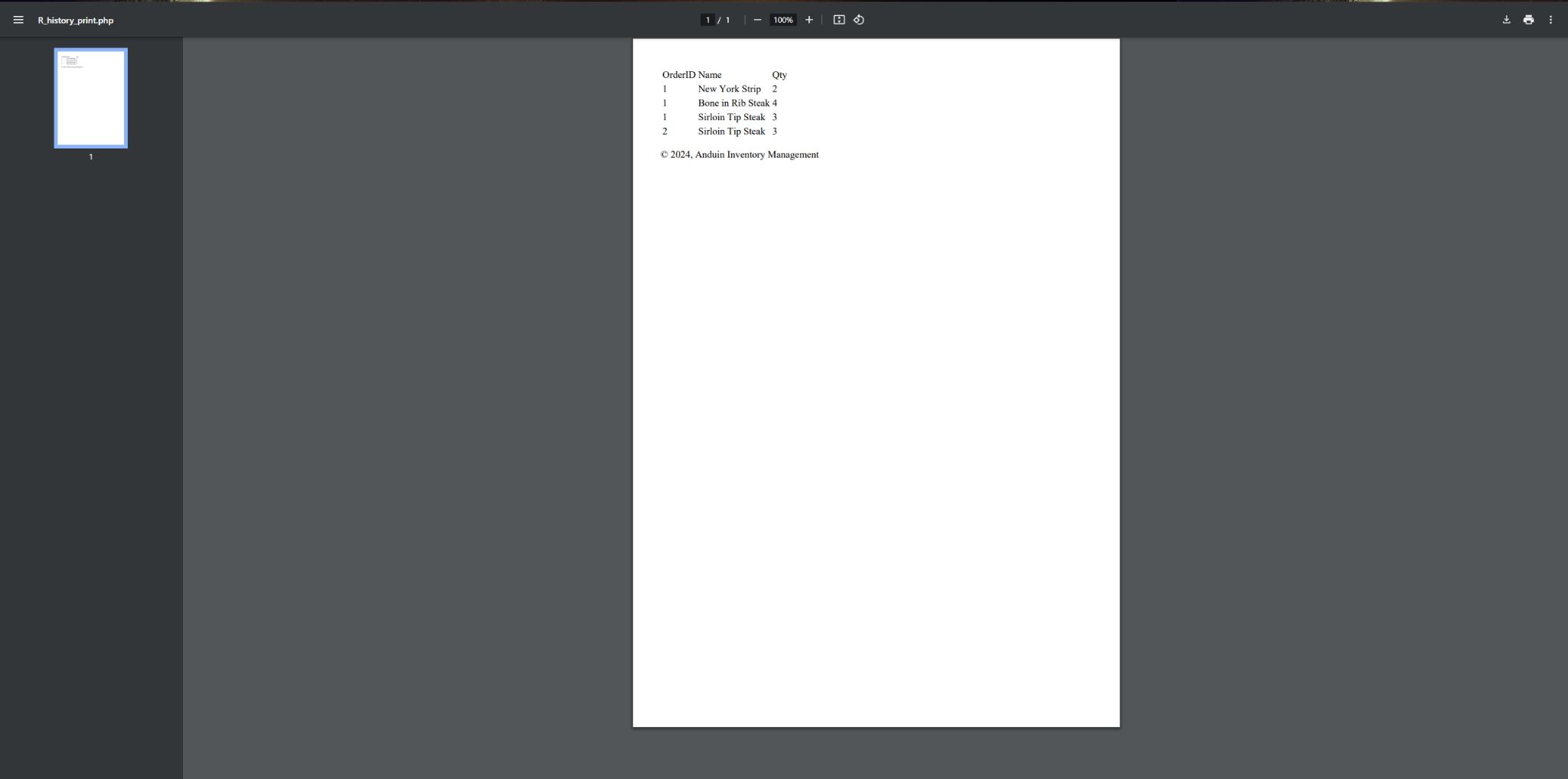


# Reports

There are two types of reports in this system. They are shown above, but the screenshots will be repeated here. The two reports are Orders and Customer History. Order gives you all the information that is available for that particular order.

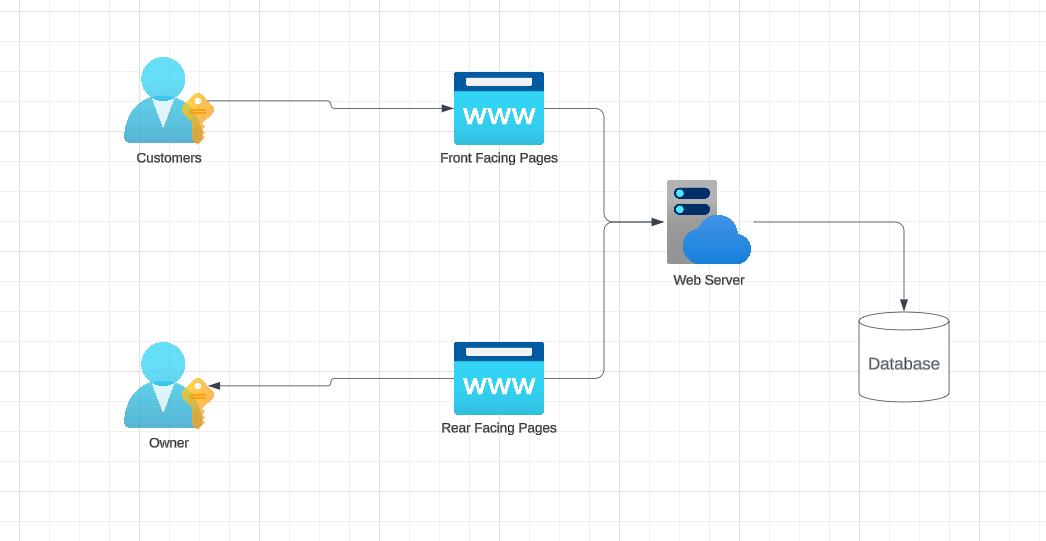


The other Report is the Customer History Report, which shows you all the orders and items for a particular customer.



# System Architecture

All of this runs on a web server, or in the case of where this project is, on a localhost. It is then accessed by users.



## 

## 

## 

## 

## Source Code Structure

Due to how this is structured, everything that is not a library or images is simply in the main Anduin directory, all other directories exist within the Anduin directory.

| **Code Directory** | |
| --- | --- |
| **Directory** | **Usage** |
| Anduin | This holds all of the pages for use in the project. |
| dompdf | This holds all of the files for the dompdf library |
| images | This holds all the images for the use of testing with the test databases. |
| owasp-php-filters | This holds all the files for the html sanitization library that is used. |
| db\_create | This holds a series of table create statements as well as a table constraint statement. |
| *Highlighted rows indicate directories containing source code.* | |

# Executables

Due to the nature of this being a web app, there are no executable files, everything involved is a PHP file, and when this is deployed it will be deployed as a static website.

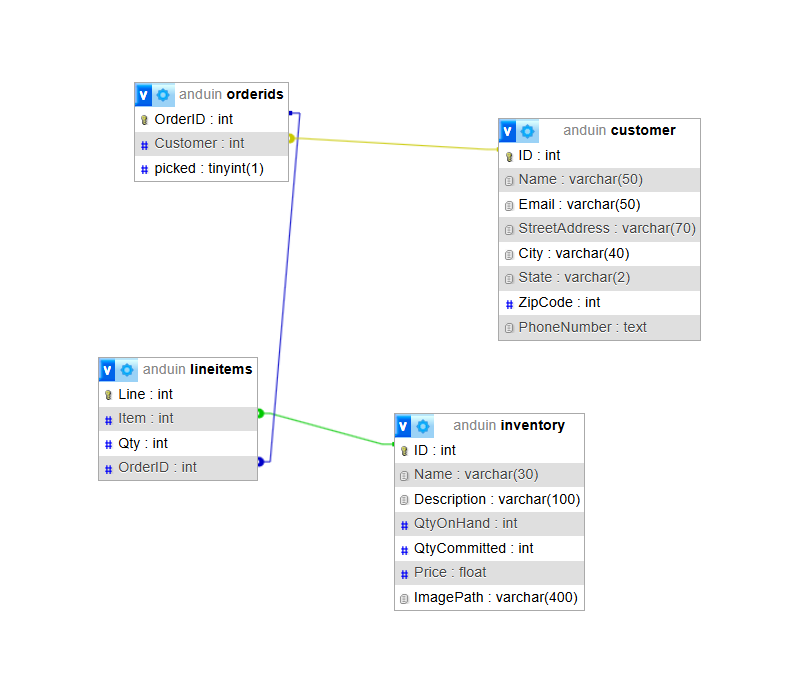
# Code Architecture

The code Architecture is laid out like this, you have two sets of pages, one of which is only accessed by customers, and the other that is only accessed by owners, they do not have any links to each other. You could even set up the admin pages to only be accessed on your network. Very limited information is shown to the customers, as shown above. The only information these people have access to is basic inventory information.

On the backend, the owner has access to all the tools for manipulating the database, with the update and add functions, as well as customer information and order information.

## Database or Data Store

# Database or Data Store Design

The structure of the database is this. The inventory table holds the inventory information, with each inventory item having an ID. The customer table holds all the customer information that is received when an email places its first order. The OrderIDs column sets up an order based on a customer's ID every time an order is placed and defaults the tinyint to 0. Then each Item in the order is added to the lineitems table, tied to both an inventory ID as well as the orderid. 

## Views, Stored Procedures and User Defined Functions

There are a number of functions that are used throughout the project, every header is a function that takes an argument that sets the title for the page, and the footer just generates a block of HTML.

On the save record page, there are a number of functions that are used to verify information from the HTML form. I have one for validating that something is positive, and I have one for redisplaying the form if something is incorrect. There is a function that checks if you are trying to input duplicate data, and there is a whole function that checks to see if an image is actually an image, and also uploads the image to the destination that is set in the file. I have a function that adds items to an order on the F\_place\_order page. It takes the orderID from earlier in the file and inserts items into a table tying them to that particular orderID.

External Files & Data

I have used two libraries, one is dompdf for my pdf generation, and the other is owasp-php-filters which is used for HTML form sanitation, both of those libraries required a little editing before they could be used.

The dompdf library is used across two pages, one for printing orders, and the other for printing customer history.

The owasp-php-filters library is used for any time a get is used to get information from an HTML form, this combined with parameterized queries helps protect against SQL injection attacks.

Programming Language | PHP/HTML/MySQL

The project is primarily written using PHP, though there is a lot of HTML mixed in. I used MySQL for the database and as for the approach to interacting with the database, I used

Project Classes

Due to how the project was made there are no classes, there are many objects, with the closest thing to a class being the elements of an associative array. It loops through the associative array and each element of that array is an object, which holds several properties that are set by the database. They are not technically classes, but some could argue that they are.

There is a product object, a line object, a customer object, and an order object. Each of these objects loads columns from the database as the data in their fields, and that data is called on throughout the loop. None of these has any functions attached to them, but they hold multiple properties. You can see all these properties when you look at the database layout.

Project Modules

Modules are used for procedural based code that does not require state data like class modules do. Complete the introduction to modules.

### Checks image information | R\_save\_record.php

This procedure uses a previously made function. There is a function called exif\_imagetype() that checks the metadata of the image, and if it is not an image it throws an error, this is called inside the CheckImage() function that first sets the target of where the image will be saved, and then after setting all the variables, it checks the image type, then it runs a few more checks, checking to see if the file already exists, checks the size, and makes sure it is only either a jpeg, png, or jpg. If any of the previous tests fail, it increases the error count, which affects the next procedure. If it all goes through the image is uploaded and the file path is returned.

### Saves a record if there are no errors | R\_save\_record.php

This procedure is split into two pieces, first, if there is an ID provided, it will simply run an update on the database. If the ID is 0 (as sent by the R\_add.php form) it creates a full new entry, this procedure calls several checks, first validating input, then checking if there is a duplicate, if all those pass, it will then run checkimage(), loading the file path into the $file variable. That $file variable loads the file path for the image into the database for the item, which allows the image to be displayed.

The biggest bug in the project is in this record, if everything works, there is no problem, but if any of the validate\_input functions fail, but the image is good it will still be uploaded. I have yet to come up with a good solution, but it will be worked on in the future.

### Place Order| F\_place\_order.php

This page is basically one long function, through a series of if…else statements, it checks to see if the email entered on checkout exists in the database, if it does, it enters a new line in the orderids table, then takes that id and runs that through the additem function. That function takes all the information from the $\_SESSION of the cart and loops through inserting each line from the cart into the lineitems table.

If the customer does not exist, it is very similar to the previous one, only before it enters the orderid, it will enter a new customer, pull that new id, and then from that point on it is very similar to if the customer did exist.

### Cart | F\_cart.php

This page is complicated. To start, if you are adding something to the cart from the product page, it will first load an array of all the information for an item from the database, then it will check to see if the product already exists in the cart and if it does, it will update the quantity for that item. To cut back on the number of pages, I was able to include a few flags. Under each item is a delete flag, and you can change the quantity on an item, and click update. If you click on remove on an item, it loads that id, and unsets the session instance of that particular part of the cart. If you instead update the cart, it will loop through each item in the cart, and if the value is different, it will update the value on that id using the new quantity. The final flag is placeorder, which directs you to the F\_place\_order.php page.

Program Start and End Flow

This is a static website that is split into two halves, the front and the backend. The flow of the front end is that someone navigates to the store using this project, sees an item they want, and adds it to their cart. They can then either checkout or add more items. When they checkout and the order is placed, it ends the session. Closing the browser also ends the session.

On the back end, you start on the index rear, which is the order list, from here you can go to the inventory page, which allows you to edit inventory, or go to the page to add an item. On the order page, you can choose to do one of three things, mark an order as picked, delete the order, or print it. Once it is printed, you can then either delete or mark it as picked. On the history page, you get all the customers in the system and can either search out a specific email or click on their line. When you search for an existing customer or click on a customer it prints out their order history.







Summary

In this documentation, I reiterated what this project is, an inventory management software, combined with a point-of-sale system. I walked through some usage scenarios, which showed how things work. I showed how a customer approaching the storefront would go about ordering, and then walked through the different backend features for the one who owns the project. I also showed how there are two reports from the system, the orders as well as the customer history reports. I also showed the system architecture and how all of this project is held on the web, with all the data stored on a mySQL server.

I went over the directories and what is included in them. Demonstrated that there are no executables since this is hosted on the internet. I showed how the code is separated into two sets of pages, one for a customer using the system, and the other for the owner. I showed the database design. I went over how there are many functions included in the project. I went over the two external libraries that are used. I went over that the project was written in PHP, with some mySQL and HTML intermixed. I covered how there are no classes in this project, but there are several complex objects. I then went into detail on some of the functions that procedurally manipulate data. I then presented the basic flow of both the front-end information as well as the back-end information. This was an overview of the project.

# APPENDIX A (DEPLOYMENT AND TESTING)

Deployment is still being considered, but I think currently I would set up a branch for each deployment, with alterations to fit each user's needs. Some pages need to be altered so that they will work outside of a local machine. At some point, I would want to convert this to follow a php framework like Yii or something like that, but as things stand, a branch for each deployment is all I can do.

The way I approached testing was by making sure each piece worked as I went along. If something was not working I took that piece of code and tested it by itself, making sure it worked before I put it back and moved on to the next thing. This systematic test as I go allowed me to limit the necessity of overall testing once things were finished.

# APPENDIX B (BUILD AND RELEASE PROCESS)

The way I would approach this is that I would set up multiple instances of the database, and would have separate branches on Git Hub. This would allow me to test things and alter the project, but not affect the live version. I am still looking into deployment methods, as this is practically a static website. I would probably simply have a branch of the project that is considered live, and when an update is ready to go, only the pieces that are updated will get changed.

I would not want to change any file paths, or change any database connection information, as each deployment would have its own file paths and database connections. I would look into how to do a merge push for each deployment of the project, excluding those particular pieces of information.

# APPENDIX C (CLIENT INSTALLATION INSTRUCTIONS)

Since this is web-hosted, there would be the need to have a site to host this information on. Something that would be a future goal is to have a central location for all the databases or host all of them on the cloud, each deployment would need its own database, and line 13 on utilities.php will need to be updated to match those databases.

Each deployment would also have its own folder, if I would in the future have a cloud server set up I would set the image path to that server, and simply have a different folder for each deployment and edit line 71 on the R\_save\_record.php page.

If you want to host the database locally, you can use wampserver, with the PhpMyAdmin being a great way to host a database locally, simply set the information on line 13 to match what the information is for your database. If you have a web host, simply look to see if there is a way to store images with that hosting site, and set their file path as the file path on line 13 in PHP.

# APPENDIX D (DEVELOPER SETUP INSTRUCTIONS)

You will need wampserver, or something similar with a MySQL database capability. Then you will need a PHP editor. The editor used for this particular project is PHPStorm. As stated above, you will need to set a file path for images on R\_save\_record.php line 71, and set your database information on line 13 of utilities.php. This will ensure that everything works as intended.