**Application Execution Guide**

For this practical I used Google Chrome to access and test my shop.

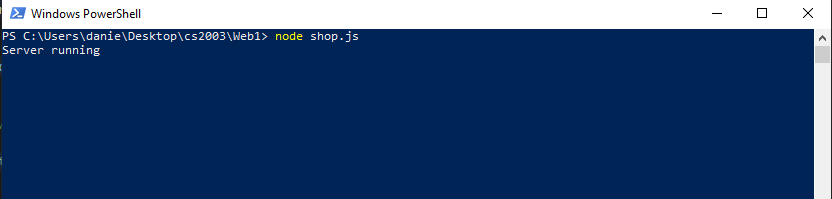
Accessing shop locally

1. In a terminal, navigate to the folder where shop.js is stored



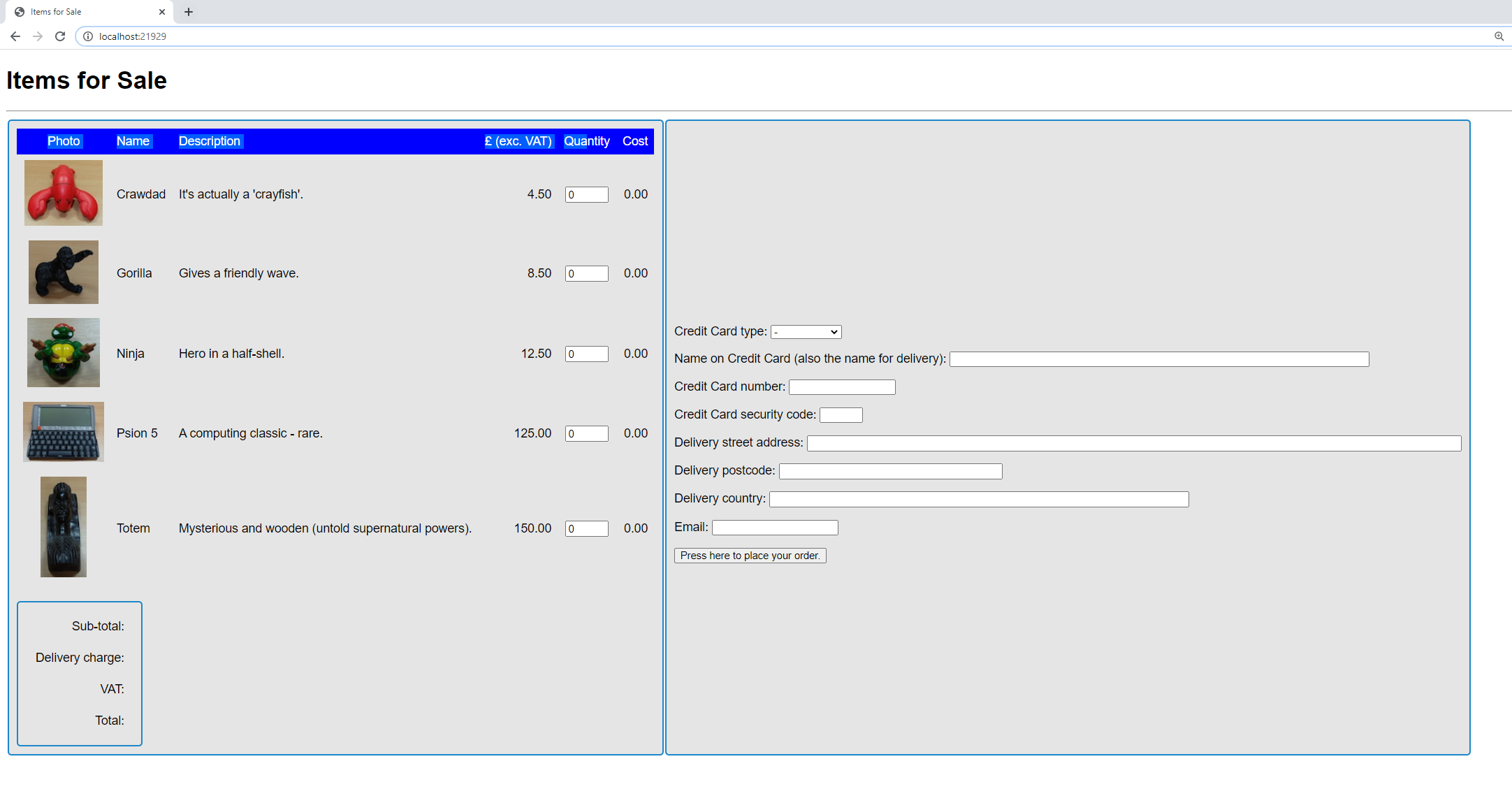
1. Using node, run the shop.js file in a terminal

node shop.js



1. Using a browser, preferably Google Chrome, access the shop at

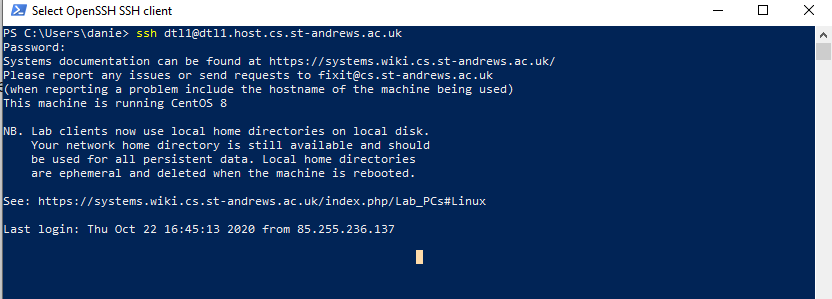
localhost:21929



Accessing shop on host server

1. ssh into host server in a terminal, and input password

ssh <username>@<username>@.host.cs.st-andrews.ac.uk



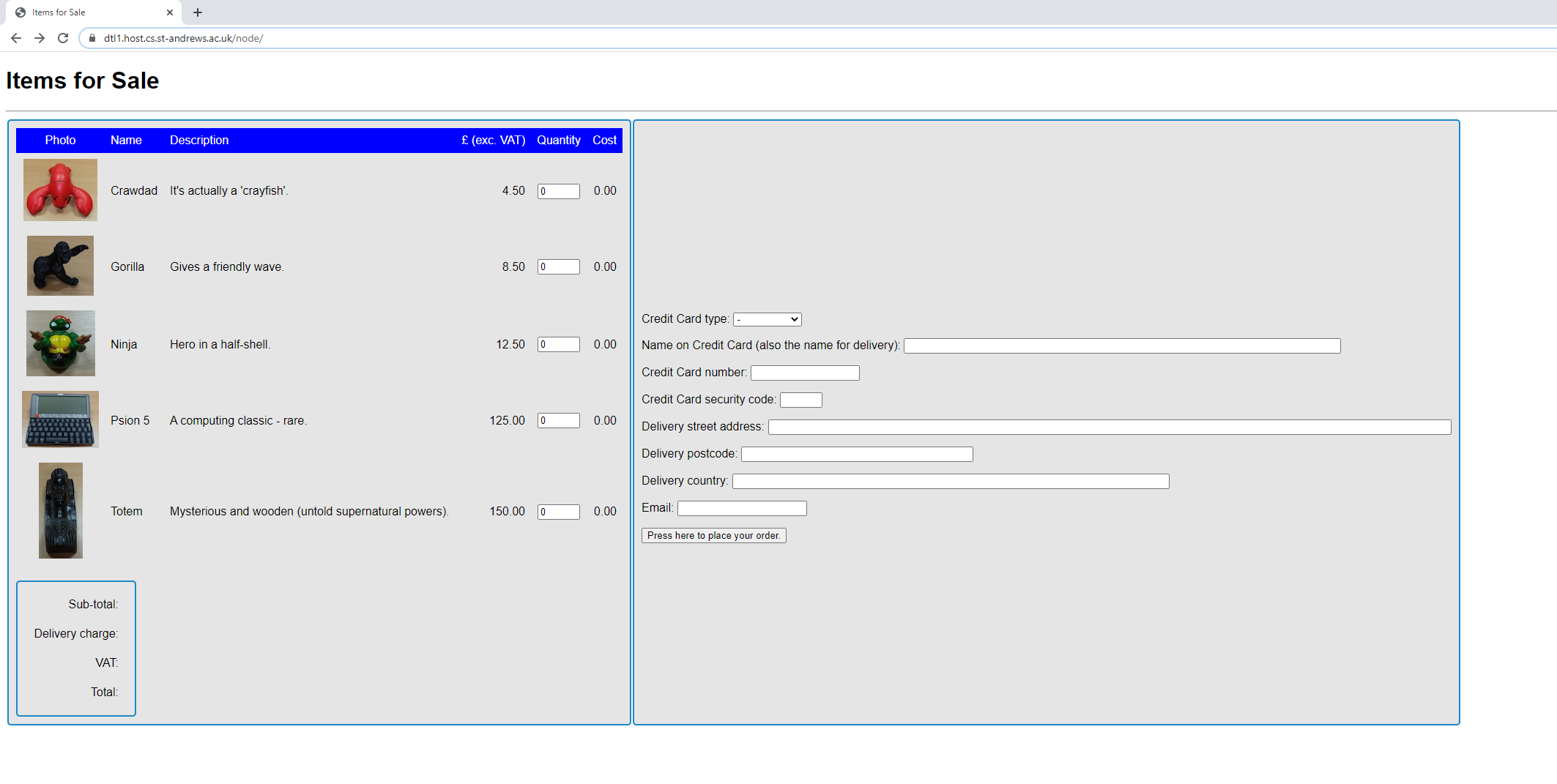
1. Navigate to the folder where shop.js is stored
2. Using node, run shop.js

node shop.js



1. In a browser, preferably Google Chrome, access the shop at

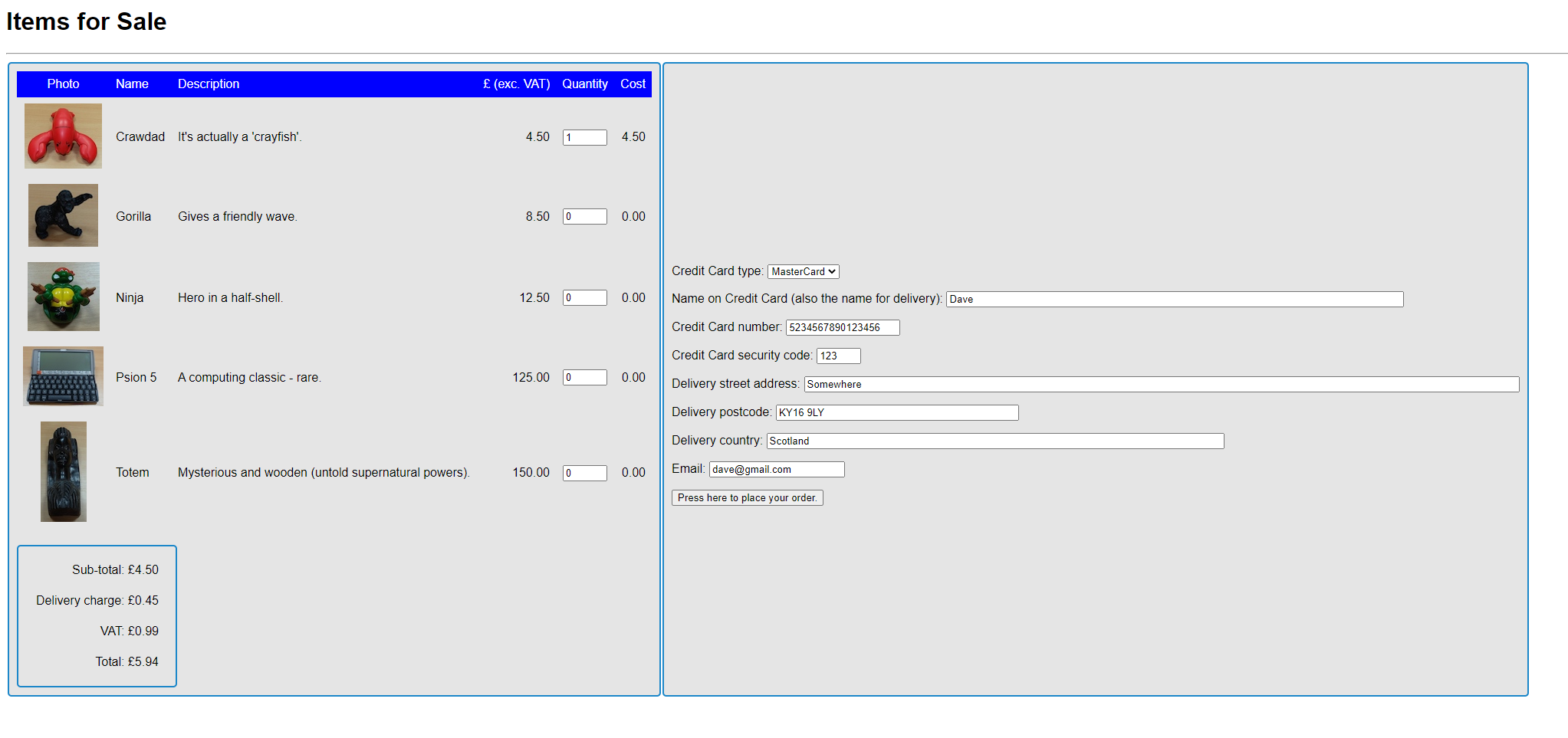
https://<username>.host.cs.st-andrews.ac.uk/node/



Using the shop

1. Now you have access to the shop, fill out all the fields on the right and make sure to order at least one item on the left.

Once you are done click the “Press here to place order button”



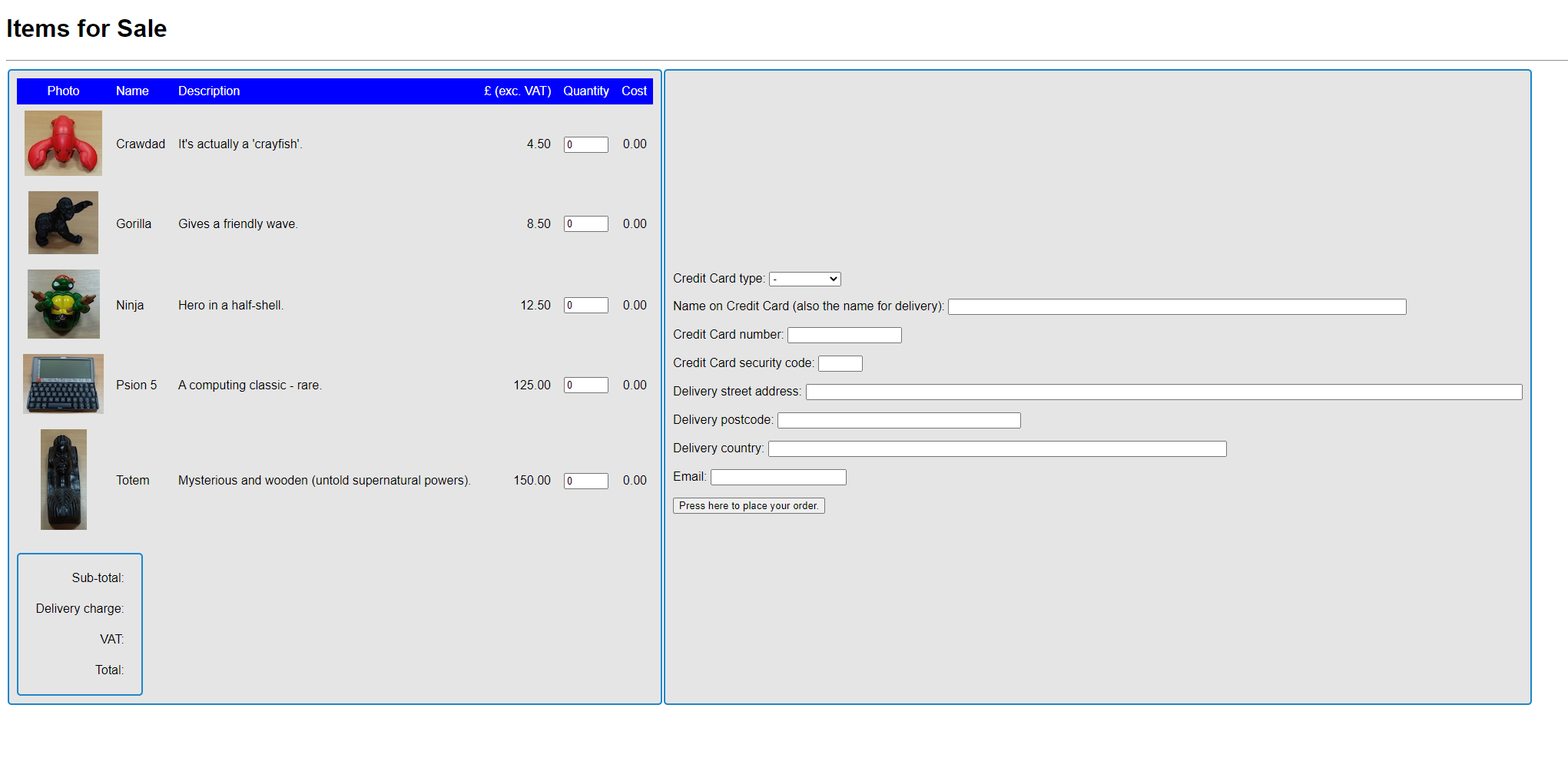
1. This takes you to the receipt page where you can either confirm your order or cancel and return to the shop.



1. Confirming your order then takes you to this page where you receive a conformation message and then can return to the shop.

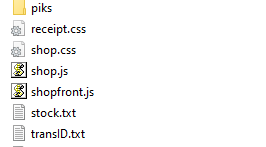


1. Returning to the shop at either the receipt or conformation stage will take you back to the shop page.



**Application operation summary**

My application consists of the following files:



* piks folder to store images for the shop
* 2 Stylesheets, one for the main shop page and one for the receipt and conformation page
* Shop.js, the shop file
* Shopfront.js, the script file
* Stock.txt, a CSV file holding information of items in the shop
* transID.txt, a txt file holding the transaction ID of the last confirmed order.

My application fully completes all the 7 requirements as listed in the specification.

It dynamically calculates and updates line costs and totals. Then sends the order data to the server where it is validated and then either a receipt page is served to confirm the order, or if the data is invalid, a pop-up error box is displayed.

I made use of a txt file to store the last transaction ID from the last confirmed order, so for new orders this last ID is read, incremented and displayed, this ensures that a unique ID is being served every time.

If the user confirms their order, then this new incremented ID is written to the file.

On the receipt page, the user can then either confirm the order or return to the shop.

I also styled and improved the presentation of the web pages, on the main page I moved the credit card and delivery order information to the right of the shop, as beforehand the page was too long and didn’t use up the horizontal space on the screen. I also styled the tables to have a grey background, so the white input boxers are easier to see and identify, and a blue border to match the blue table header already given to us.

On the receipt page I chose to display the information in tables so the names and then values of fields could be lined up into columns which makes it easier to read, I also kept the same grey and blue table style from the main page.

**Testing**

(see next page)

|  |  |  |  |
| --- | --- | --- | --- |
| Test | Expected Result | Actual Result | Screenshot |
| Trying to place an order with an empty field | Pop up saying to fill in the field | As expected |  |
| Changing the quantity of an item | The line cost and totals change | As expected |  |
| Trying to place an order for 0 items | Alert saying “you cannot order 0 items” | As expected |  |
| Not inputting a credit card type | Alert saying “please select a credit card type” | As expected |  |
| Inputting a card number with a wrong first digit for the type | Alert saying “Mastercard card numbers must start with a 5, and Visa card numbers must start with a 4” | As expected |  |
| Not matching the correct format for the inputs | Popup saying “please match the format requested” | As expected |  |
| Ordering with correct and valid data | Takes me to the receipt page.  Date is correct.  Unique Transaction ID is printed.  Only items that have orders are printed.  Only displays the first and last 2 digits of the card number  Displays all other relevant data as requested by the requirements for the order correctly | As Expected |  |
| Clicking the confirm order button | Takes me to the order conformation page | As expected |  |
| Clicking the return to shop button | Takes me back to the shop | As expected |  |
| Checking to see if a unique transaction ID is correctly calculated | Current ID in the txt file is 13, so new one should be 14 | As expected |  |
| Confirming the order should update the txt file to 14 | Txt file correctly updates | As expected |  |

**Security and Privacy**

Due to this application being a shop that requires a user to input personal data, there are several privacy issues that arise.

Firstly, the shop requires credit card data, this data is obviously sensitive and would have severe consequences if it ended up in the hands of someone with malicious intent. The credit card data is sent to the server via POST however instead of GET, this is the more secure and therefore right option as sending the data via GET would append the data in the URL string which could easily be intercepted by an attacker, whereas POST carries data in the message body.

On top of not only the credit card data there is other data sensitive to the user such as their full name and address. So, if the application were to store this kind of data and order transactions then it would need to do so in a secure manner that meets the requirements of the Data Protection Act.