



EMBERFROST

System Test Report

Client

Vladimir Klimkiv

Team Mocha

Andrey Kurudimov

Christian Suarez

Ilya Indik

Jared Petersen

Justin Singh

Monica Abigail Suase

Date: 04/29/2022

Table of Contents

1.0 Cart

- 1.1 User Interaction - Add Item to Cart
- 1.2 User Interaction - Increase Item Amount
- 1.3 User Interaction - Decrease Item Amount
- 1.4 User Interaction - Remove Item from Cart

2.0 Subscribe Page

- 2.1 User Interaction - First Time Subscribing
- 2.2 User Interaction - Submitting an Invalid Email (Error)
- 2.3 User Interaction - Already Subscribed

3.0 Contact Page

- 3.1 User Interaction - Successfully Filled out the Form
- 3.2 User Interaction - Missing Field

4.0 FAQ/Landing and Product Page

- 4.1 Scrolling will load data.
- 4.2 Going to FAQ, loads questions
- 4.3 About Us Page Loading
- 4.4 Navbar responding

5.0 Test Order Creation

- 5.1 Going from cart to stripe
- 5.2 Going from cart to order page

6.0 Frontend Test Suites

- 6.0.1 Requirements
- 6.0.2 Setup
- 6.1 Banner Component
 - 6.1.1 Component should show featured item when user loads home page
 - 6.1.2 Component should add to cart when the user pressed button
 - 6.1.3 Component should show error message if request fails
 - 6.1.4 Test Results
- 6.2 Navbar Component
 - 6.2.1 Component should be visible
 - 6.2.2 Component should render links
 - 6.2.3 Component should render cart link
 - 6.2.4 Component should remove item from cart when user clicks remove icon
 - 6.2.5 Test Results
- 6.3 Product Item Component

[6.3.1 Component should render product image](#)
[6.3.2 Component should label of product](#)
[6.3.3 Component should render product price](#)
[6.3.4 Component should render the add to cart button](#)
[6.3.5 Component should add product to users cart](#)
[6.3.6 Test Results](#)
[6.4 Cart Page Component](#)
[6.4.1 Component should remove item from cart](#)
[6.4.2 Component should update quantity of item in cart](#)
[6.4.3 Test results](#)
[6.5.0 All frontend unit and integration tests](#)

[7.0 Back End Unit and Integration Tests Suites](#)

[7.0.1 Requirements](#)
[7.0.2 Setup](#)
[7.1.0 Running Test Suites](#)
[7.1.1 HTTP Controller Suite](#)
[7.1.2 Error Handling Suite](#)
[7.1.3 Services Test Suite](#)
[7.1.4 Sample Test Code](#)
[7.1.5 Dealing With Failing Tests](#)

1.0 Cart

The cart allows a user of the site to see their items which have been added into a central location to then purchase whenever they please. They are given various abilities from the cart page such as being able to add, remove, increase quantity, and decrease quantity.

The screenshot displays the Emberfrost website's cart interface across three panels:

- Panel 1 (Left):** Shows a product card for "Ahri". It includes a thumbnail image of a woman in a beret smoking a cigarette, the title "Ahri", a "BEST SELLER" badge, a short product description, and a blue "Add To Cart" button. A red box highlights the "Add To Cart" button, and a circled "1" is placed above it.
- Panel 2 (Bottom Left):** A modal window titled "Successfully added." shows the product thumbnail, the name "Ahri", and the quantity "1 x \$122.34". A green checkmark icon is on the left, and a red box surrounds the entire modal, with a circled "2" placed below it.
- Panel 3 (Right):** Shows the "Emberfrost" cart page with a header indicating 1 item. The cart table lists the product "Ahri" at \$122.34 with a quantity of 1. Below the table, the "CART TOTALS" section shows a subtotal of \$122.34 and a total of \$122.34. A blue "PROCEED TO CHECKOUT" button is at the bottom. A circled "3" is placed above the cart table.

Figure 1.1.1

1.1 User Interaction - Add Item to Cart

Test Description: This test is to verify that the “Add to Cart” button will add the product item to the cart. To perform this test, click the “Add to Cart” button shown in Figure 1.1.1: Box 1. A pop-up will appear showing the cart has successfully been added to, shown in Figure 1.1.1: Box 2. The “My Cart” items should increase as well, Figure 1.1.1: Box 3.

Tester: Christian Suarez

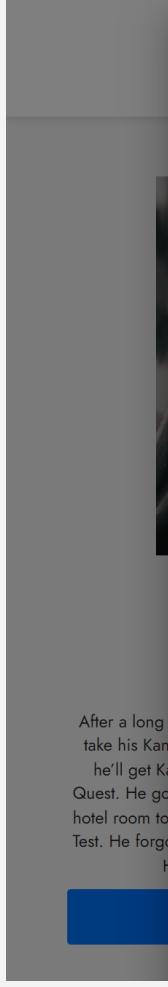
Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: Product Ahri

Expected output: Figure 1.1.1: Box 2 - 3

- Verified
- Not Verified



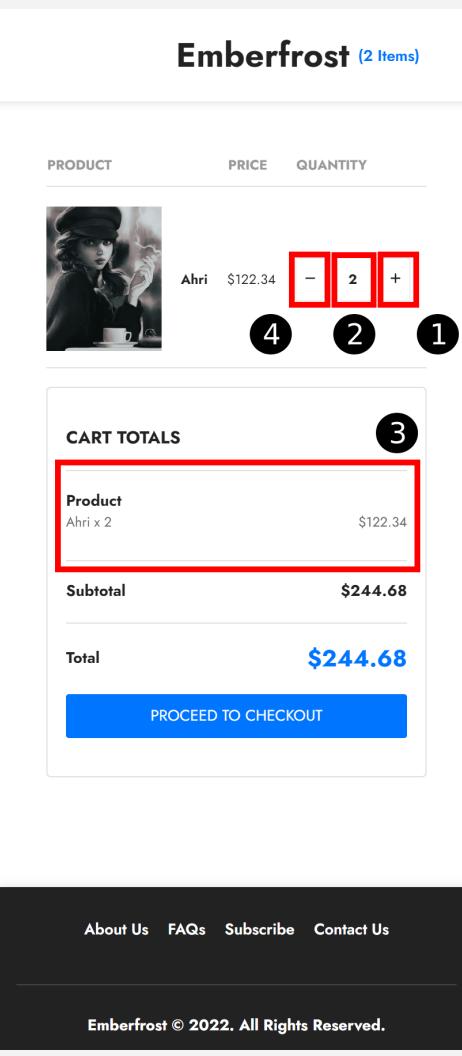
SHOPPING CART

CLOSE →

	Ahri	X
1 X \$122.34		
Subtotal:	\$122.34	

GO TO CART

After a long take his Kan he'll get K Quest. He go hotel room to Test. He forgo



Emberfrost (2 Items)

PRODUCT	PRICE	QUANTITY
	\$122.34	- 2 +
4	2	1

CART TOTALS

Product
Ahri x 2 \$122.34

Subtotal \$244.68

Total \$244.68

PROCEED TO CHECKOUT

About Us FAQs Subscribe Contact Us

Emberfrost © 2022. All Rights Reserved.

Figure 1.2.1

Figure 1.2.2

1.2 User Interaction - Increase Item Amount

Test Description: This test is to verify that the “+” button will increase the quantity of the item in the cart. To perform this test, click on the items drop-down menu, Figure 1.1.1: Box 3. Then, click on the “GO TO CART” button to get to the cart page, Figure 1.2.1. Once in the cart page, click on the “+” button to increase the amount of the item until satisfied, Figure 1.2.2: Box 1. The product item amount and product total will update automatically if successful, Figure 1.2.2: Box 2 and Box 3.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: Product Ahri

Expected output: Figure 1.2.2: Box 2 - 3

- Verified
- Not Verified

1.3 User Interaction - Decrease Item Amount

Test Description: This test is to verify that the “-” button will decrease the quantity of the item in the cart. To perform this test, click on the items drop-down menu, Figure 1.1.1: Box 3. Then, click on the “GO TO CART” button to get to the cart page, Figure 1.2.1. Once in the cart page, click on the “-” button to decrease the amount of the item until satisfied, Figure 1.2.2: Box 4.

The product item amount and product total will update automatically if successful, Figure 1.2.2: Box 2 and Box 3.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: Product Ahri

Expected output: Figure 1.2.2: Box 2 - 3

- Verified
- Not Verified

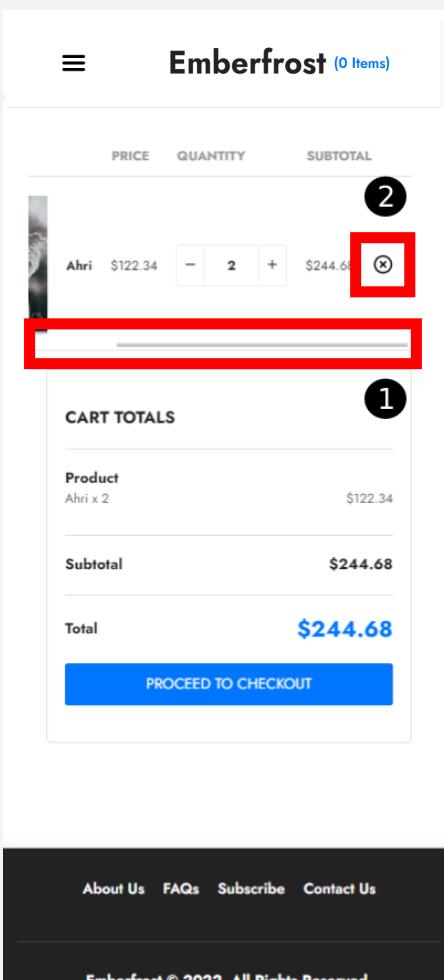


Figure 1.4.1

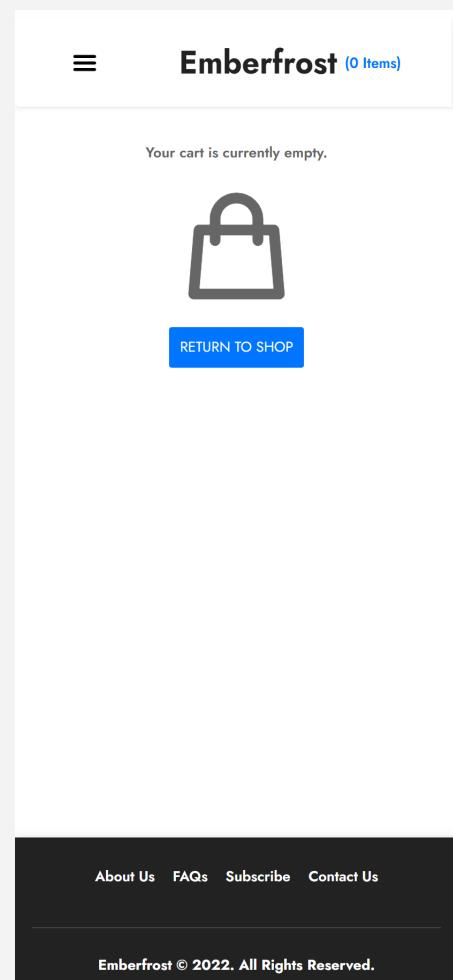


Figure 1.4.2

1.4 User Interaction - Remove Item from Cart

Test Description: This test is to verify that an item can be removed from the cart. To perform this test, click on the items drop down menu, Figure 1.1.1: Box 3. Then, click on the “GO TO CART” button to get to the cart page, Figure 1.2.1. Once in the cart page, use the scrollbar located under the item to slide it all the way left, Figure 1.4.1: Box 1, then click the “X” button to remove the item Figure 1.4.1: Box 2. If the test is successful, the item will disappear from the cart and the cart totals will automatically update. In this case, the cart becomes empty, Figure 1.4.2.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: Product Ahri

Expected output: Figure 1.4.2

- Verified
- Not Verified

2.0 Subscribe Page

The subscription page allows a user of the site to subscribe for newsletters of the client. They are allowed to subscribe as many times as they like, however they will not get spammed with newsletters.

The figure consists of two side-by-side screenshots of the Emberfrost website. The left screenshot, labeled Figure 2.1.1, shows the main homepage with sections for 'SHIPPING' and 'OTHER'. The 'SHIPPING' section contains four expandable questions: 'What shipping cost do you charge?', 'What is my shipment number?', 'Can I track my package?', and 'How can I update my delivery address?'. The 'OTHER' section contains four expandable questions: 'Do you guys do discounts?', 'What is EmberFrost's Cancelation policy?', 'What is EmberFrost's Return policy?', and 'What do you sell?'. Below these sections is a footer with four buttons: 'About Us', 'FAQs', 'Subscribe', and 'Contact Us'. The 'Subscribe' button is highlighted with a red border. The right screenshot, labeled Figure 2.1.2, shows the 'Subscribe to EmberFrost' page. It features a form with an 'Email' input field and a 'Submit' button, both of which are highlighted with red boxes. To the right of the form is a bulleted list of three benefits: 'Receive most recent updates about our products.', 'Get notified of new developments.', and 'Discounts on items that we have in store.' A circled '1' is placed next to the list. Below the form is a success message: 'You have successfully subscribed.' with a checkmark icon. A circled '2' is placed next to the message. A circled '3' is placed at the top right of the success message area. The entire right screenshot is also enclosed in a red box.

Figure 2.1.1

Figure 2.1.2

2.1 User Interaction - First Time Subscribing

Test Description: This test is to verify that when a user subscribes, an email is sent to them. To perform this test, navigate to the Subscribe page by scrolling to the footer. Then click on the link to the Subscribe page, Figure 2.1.1: Box 1. Once on the Subscribe page, use the subscribe form to enter an email, Figure 2.1.2: Box 1, then click the "Submit" button to submit the form, Figure 2.1.2: Box 2. Once the form is submitted, a pop up should appear, Figure 2.1.2: Box 3. You can then check the inbox of the email entered. The test is successful if the pop up is shown and an email is received matching Figure 2.1.3.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: User Email Address

Expected output: Figure 2.1.2: Box 3 and Figure 2.1.3

- Verified
- Not Verified

The screenshot shows a mobile-style website for 'Emberfrost'. At the top, there's a navigation bar with three horizontal lines on the left and the brand name 'Emberfrost' in bold, followed by '(0 items)' in parentheses. Below this is a section titled 'Subscribe to EmberFrost' with a descriptive paragraph about receiving updates and notifications. To the right of the text is a bulleted list of benefits: '• Recieve most recent updates about our products.', '• Get notified of new developments.', and '• Discounts on items that we have in store.' A red circle with the number '1' is positioned next to the list. Below the list is a form field labeled 'Email' containing 'isatpotatoes@me'. A red box surrounds this entire section. Underneath the email field, the text 'Invalid email address' is displayed in red. Below the form is a blue 'Submit' button, also enclosed in a red box. A red circle with the number '2' is positioned next to the 'Submit' button. At the bottom of the page is a dark footer bar with white text links for 'About Us', 'FAQs', 'Subscribe', and 'Contact Us'. The footer also includes the copyright notice 'Emberfrost © 2022. All Rights Reserved.'

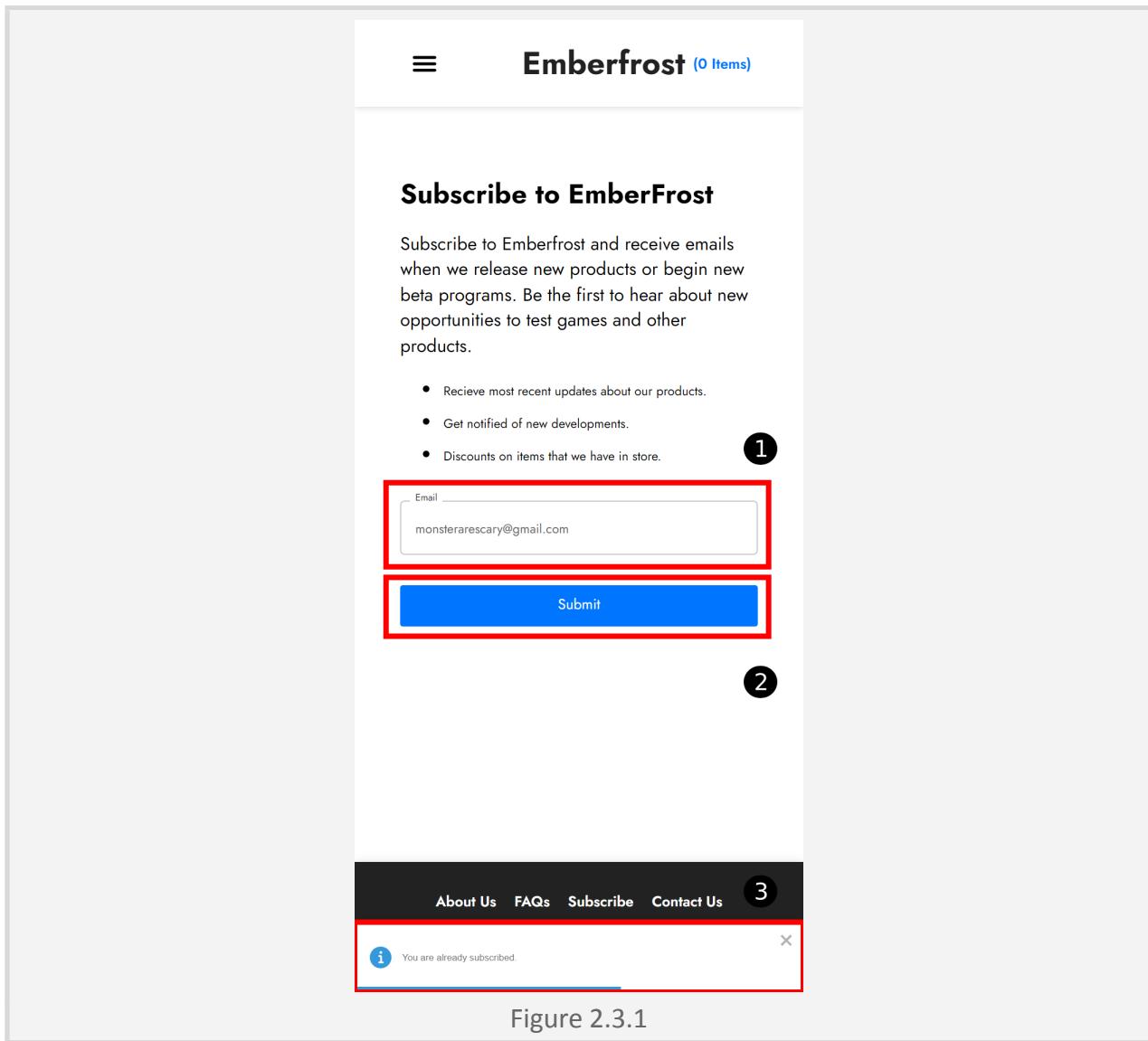
Figure 2.2.1

2.2 User Interaction - Submitting an Invalid Email (Error)

Test Description: This test is to verify that an invalid email input cannot be submitted. To perform this test, navigate to the Subscribe page by scrolling to the footer. Then click on the link to the Subscribe page, Figure 2.1.1: Box 1. Once on the Subscribe page, use the subscribe form to enter an invalid email, Figure 2.1.2: Box 1, then click the "Submit" button to submit the form, Figure 2.1.2: Box 2. Once the button is clicked, the test is successful if the form becomes red and says invalid email address underneath, Figure 2.2.1: Box 2.

Tester: Christian Suarez
Recorder: Monica Abigail Suase
Date: 04/29/2022
Specify Test input: hellotest
Expected output: Figure 2.2.1

- Verified
 Not Verified



2.3 User Interaction - Already Subscribed

Test Description: This test is to verify that an email address already on the subscribe list cannot subscribe again. To perform this test, navigate to the Subscribe page by scrolling to the footer. Then click on the link to the Subscribe page, Figure 2.1.1: Box 1. Once on the Subscribe page, use the subscribe form to enter an email, Figure 2.1.2: Box 1, then click the "Submit" button to submit the form, Figure 2.1.2: Box 2. Once the form is submitted, use that same form to submit

the same email. The test is successful if a pop-up saying you have already subscribed appears,
Figure 2.3.1: Box 3.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: User email address

Expected output: Figure 2.3.1

- Verified
- Not Verified

3.0 Contact Page

The contact page allows a user of the site to contact the client directly without logging into their email account. They must provide required field data in order to submit the form.

Emberfrost (0 Items)

What payment options do you take in? +

How do I pay with Venmo or Cash app? +

SHIPPING

What shipping cost do you charge? +

What is my shipment number? +

Can I track my package? +

How can I update my delivery address? +

OTHER

Do you guys do discounts? +

What is EmberFrost's Cancelation policy? +

What is EmberFrost's Return policy? +

What do you sell? +

4 3 1 2

About Us FAQs Subscribe Contact Us

Emberfrost © 2022. All Rights Reserved.

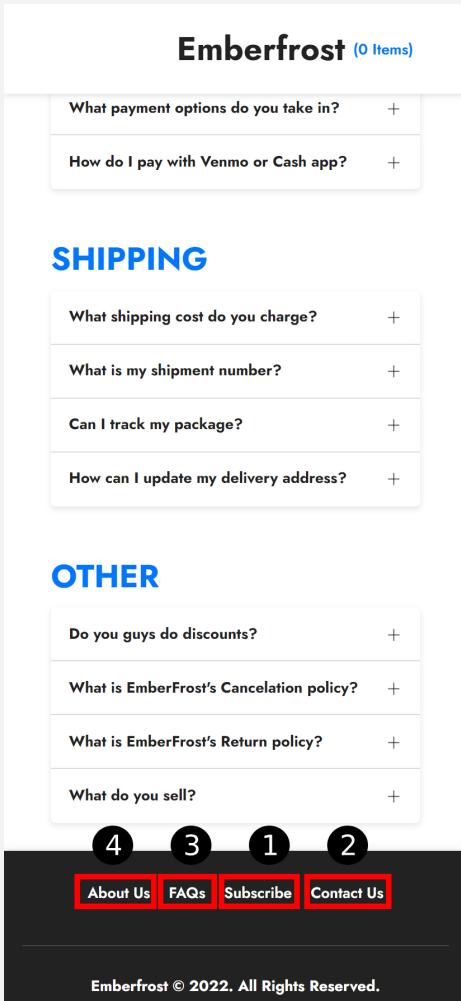


Figure 3.1.1

Emberfrost (0 Items)

Owner
Vlad Klimkiv

Phone Number
1.800.458.56

Support
support@emberfrost.com

Contact Us

1 Write your message here

Name

Email

Submit

2

3

About Us FAQs Subscribe Contact Us

Your message has been sent.

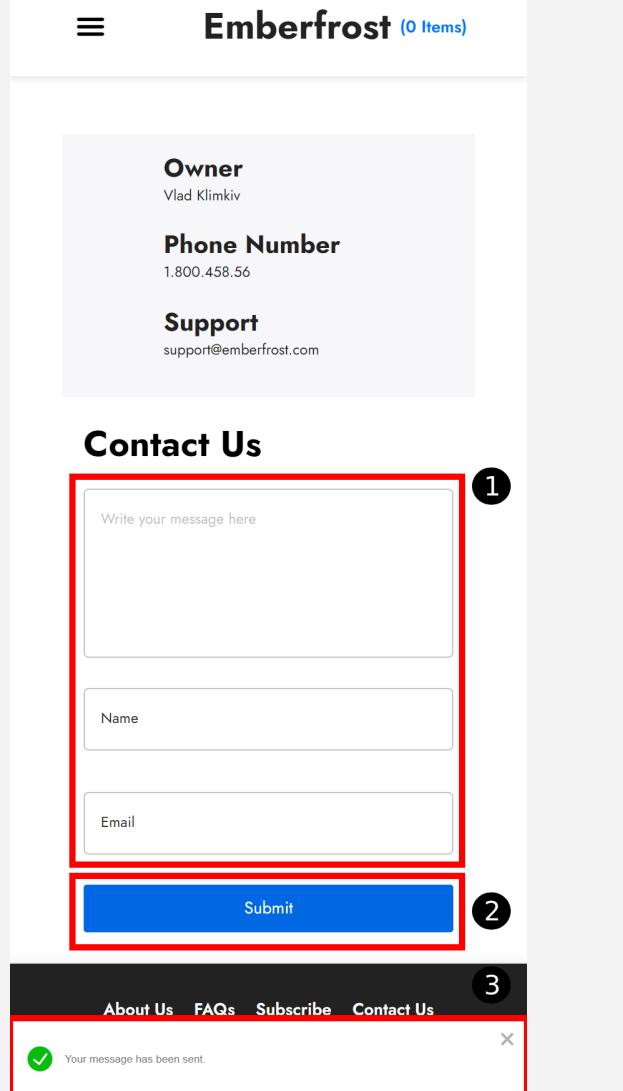


Figure 3.1.2

3.1 User Interaction - Successfully Filled out the Form

Test Description: This test is to verify that a user can use the contact form. To perform this test, navigate to the Contact Us page by scrolling to the footer. Then click on the link to the Contact Us page, Figure 3.1.1: Box 2. Once on the Contact Us page, fill out the contact us form with the appropriate data, Figure 3.1.2: Box 1, then click the "Submit" button to submit the form, Figure 3.1.2: Box 2. If the form is submitted and a pop up appears, the test is successful, Figure 3.1.2: Box 3.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: small message, name, email address

Expected output: Figure 3.1.2: Box 3

- Verified
- Not Verified

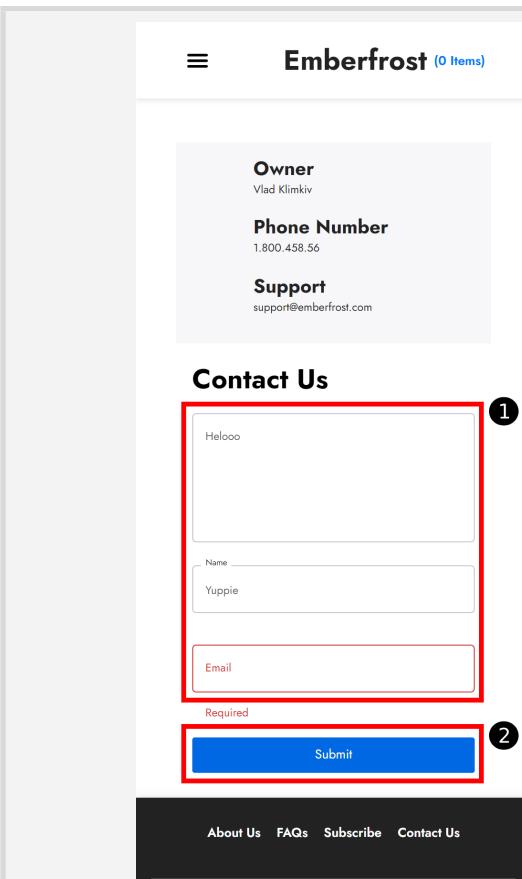


Figure 3.2.1

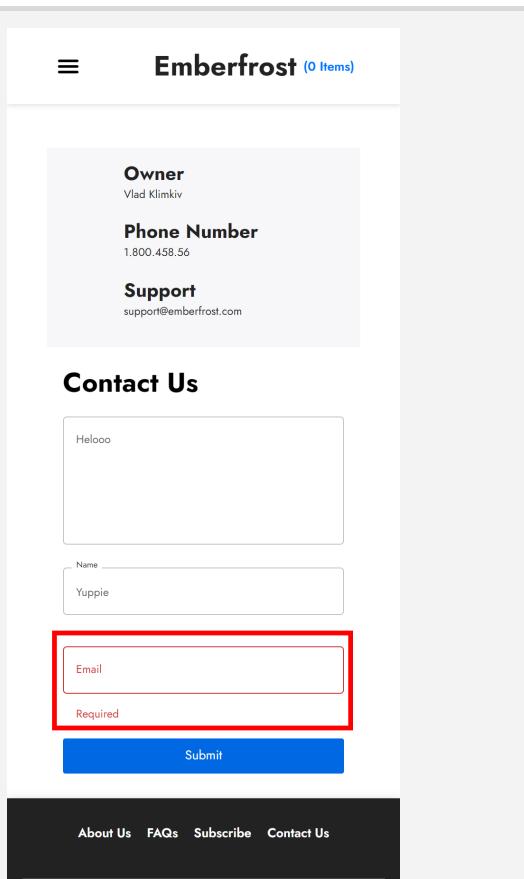


Figure 3.2.2

3.2 User Interaction - Missing Field

Test Description: This test is to verify that a user can use the contact form. To perform this test, navigate to the Contact Us page by scrolling to the footer. Then click on the link to the Contact Us page, Figure 2.1.1: Box 2. Once on the Contact Us page, fill out the contact us form with the appropriate data, excluding one of the fields, Figure 3.2.1: Box 1. In this case, do not input an email. Then click the “Submit” button to submit the form, Figure 3.2.1: Box 2. Once the button is clicked, the missing field form becomes red and Required is shown underneath for the test to be successful, Figure 3.2.2. The form cannot be submitted with a missing field.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: small message and name

Expected output: Figure 3.2.2

- Verified
- Not Verified

4.0 FAQ/Landing and Product Page

The following section focuses on landing page which hold the feature item. FAQ page contains all questions the client thought were necessary to answer. The product page will contain the list of products that are currently being sold in the store.

☰ Emberfrost (0 Items)



BEST SELLER

Ahri

After a long day of work, Kanye West goes to his Kanye Nest to take his Kanye Rest. He wakes up feeling his Kanye Best. Then he'll get Kanye Dressed on his Kanye Vest to go on a Kanye Quest. He goes to church and becomes Kanye Blessed, then to a hotel room to be a Kanye Guest. Then to school to take his Kanye Test. He forgot to brush his teeth. Did he run out of Kanye Crest? His neighbor stole it, what a Kanye Pest.

Add To Cart

☰ Emberfrost (0 Items)



Ahri

\$122.34

Add To Cart



Figure 4.1.1

Figure 4.1.2

4.1 Scrolling will load data.

Test Description: This test is to verify that with the scrollbar, the page populates. To perform this test, use the scroll bar to navigate down the page, Figure 4.1.1. The test is successful if items populate the page, Figure 4.1.2.

14

Tester: Christian Suarez
 Recorder: Monica Abigail Suase
 Date: 04/29/2022
 Specify Test input: No Input
 Expected output: Figure 4.1.2

- Verified
- Not Verified

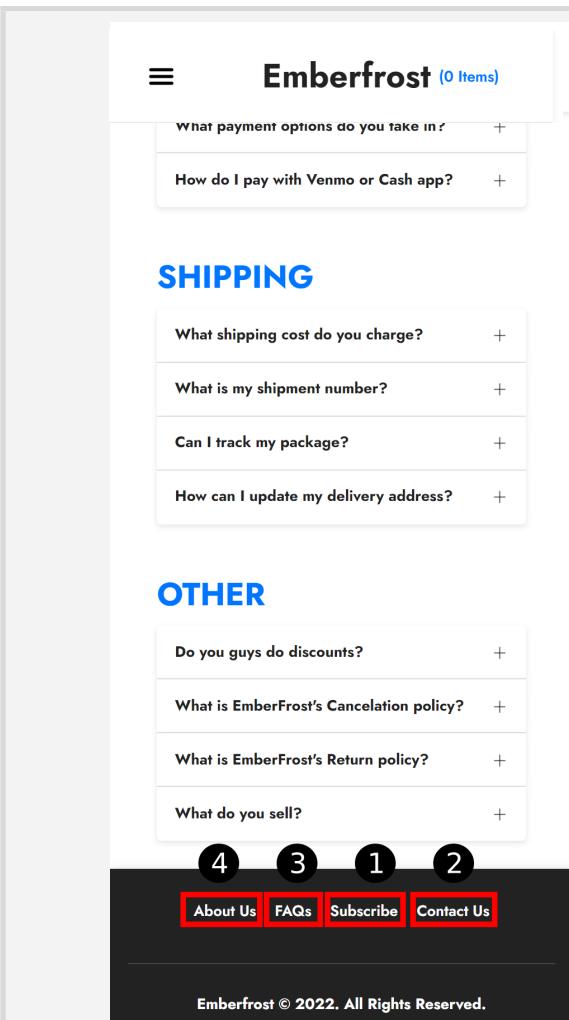


Figure 4.2.1 shows the Emberfrost FAQ page. At the bottom, there is a footer with four links: 'About Us', 'FAQs' (highlighted with a red border), 'Subscribe', and 'Contact Us'. Above the footer, there is a section titled 'SHIPPING' containing four questions. A red circle numbered 4 is over the 'About Us' link, and circles numbered 3, 1, and 2 are positioned above the 'FAQs', 'Subscribe', and 'Contact Us' links respectively.

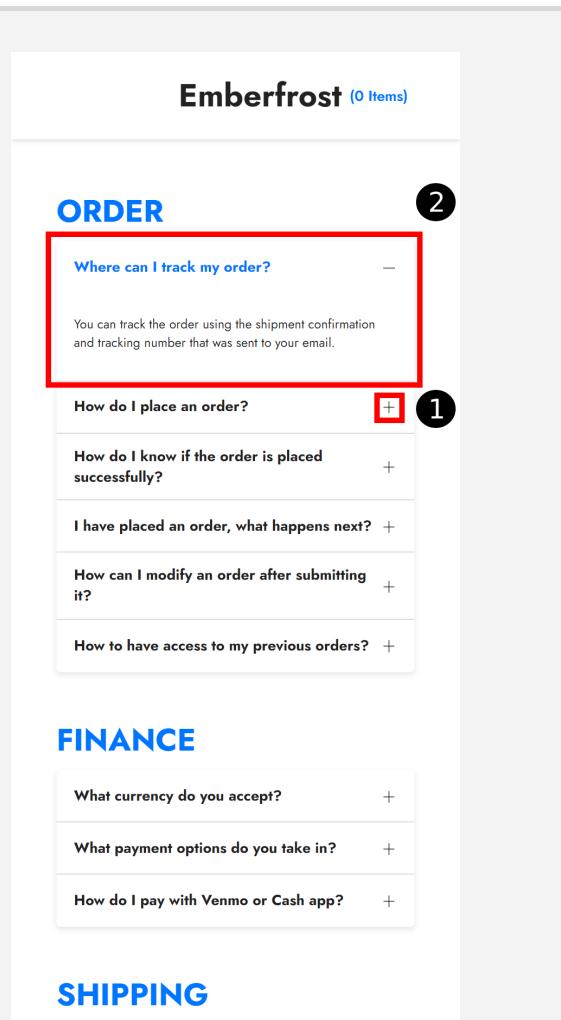


Figure 4.2.2 shows the same FAQ page after clicking on the 'FAQs' link. The 'Where can I track my order?' section has been expanded, revealing its answer. A red box surrounds this expanded section. A red circle numbered 2 is placed over the expanded section, and a red square with a plus sign is placed over the collapsed 'How do I place an order?' section. A black circle numbered 1 is placed over the collapsed 'How do I place an order?' section.

4.2 Going to FAQ, loads questions

Test Description: This test is to verify that the FAQs page loads the questions and answers. To perform this test, navigate to the FAQs page by scrolling to the footer. Then click on the link to the FAQs page, Figure 4.2.1: Box 3. Once on the FAQs page, expand the question by clicking on the "+" button, Figure 4.2.2: Box 1. The test is successful if the section expands and the answer appears, Figure 4.2.2: Box 2.

Tester: Christian Suarez

Recorder: Monica Abigail Suase

Date: 04/29/2022

Specify Test input: No Input

Expected output: Figure 4.2.2: Box 2

- Verified
- Not Verified

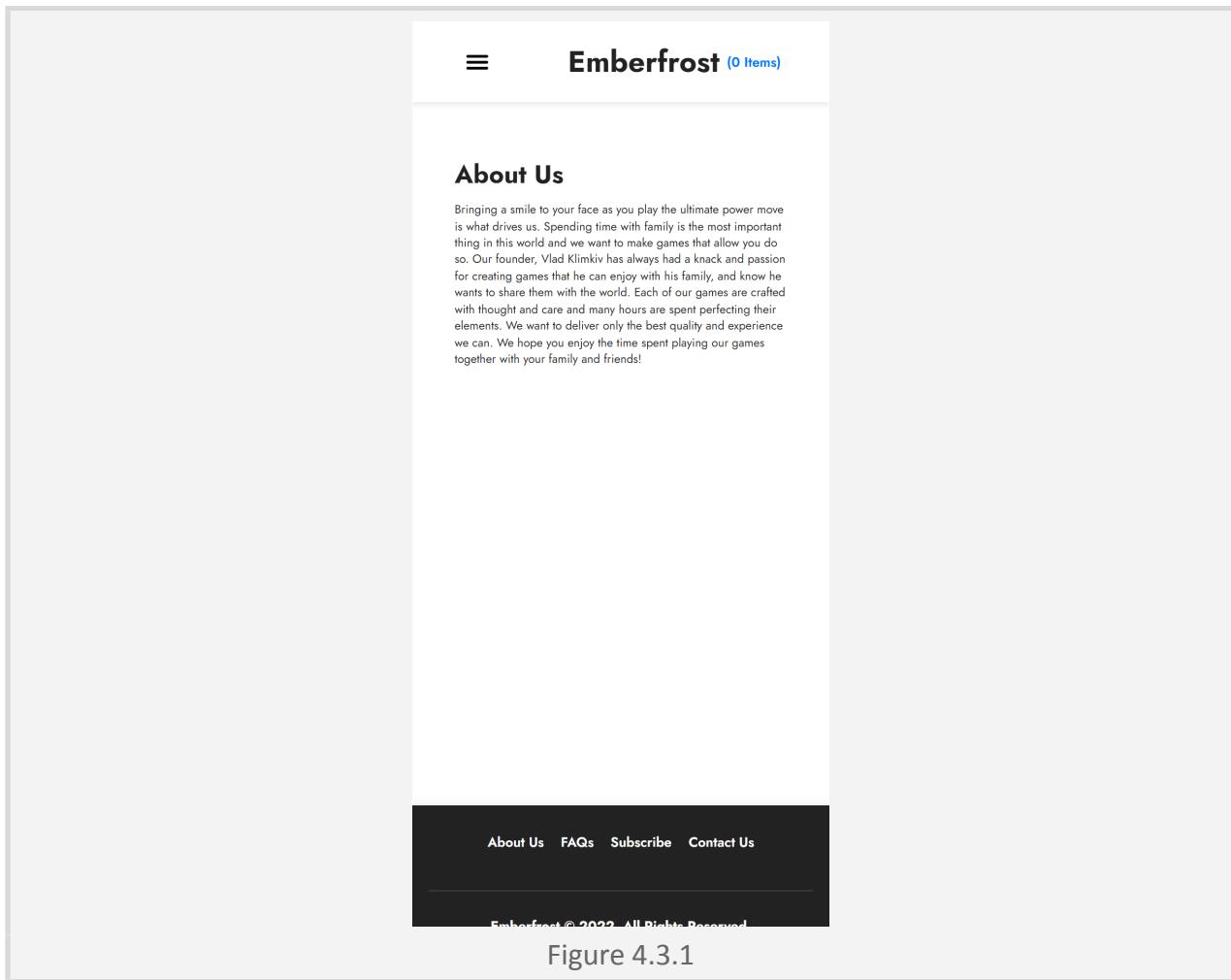


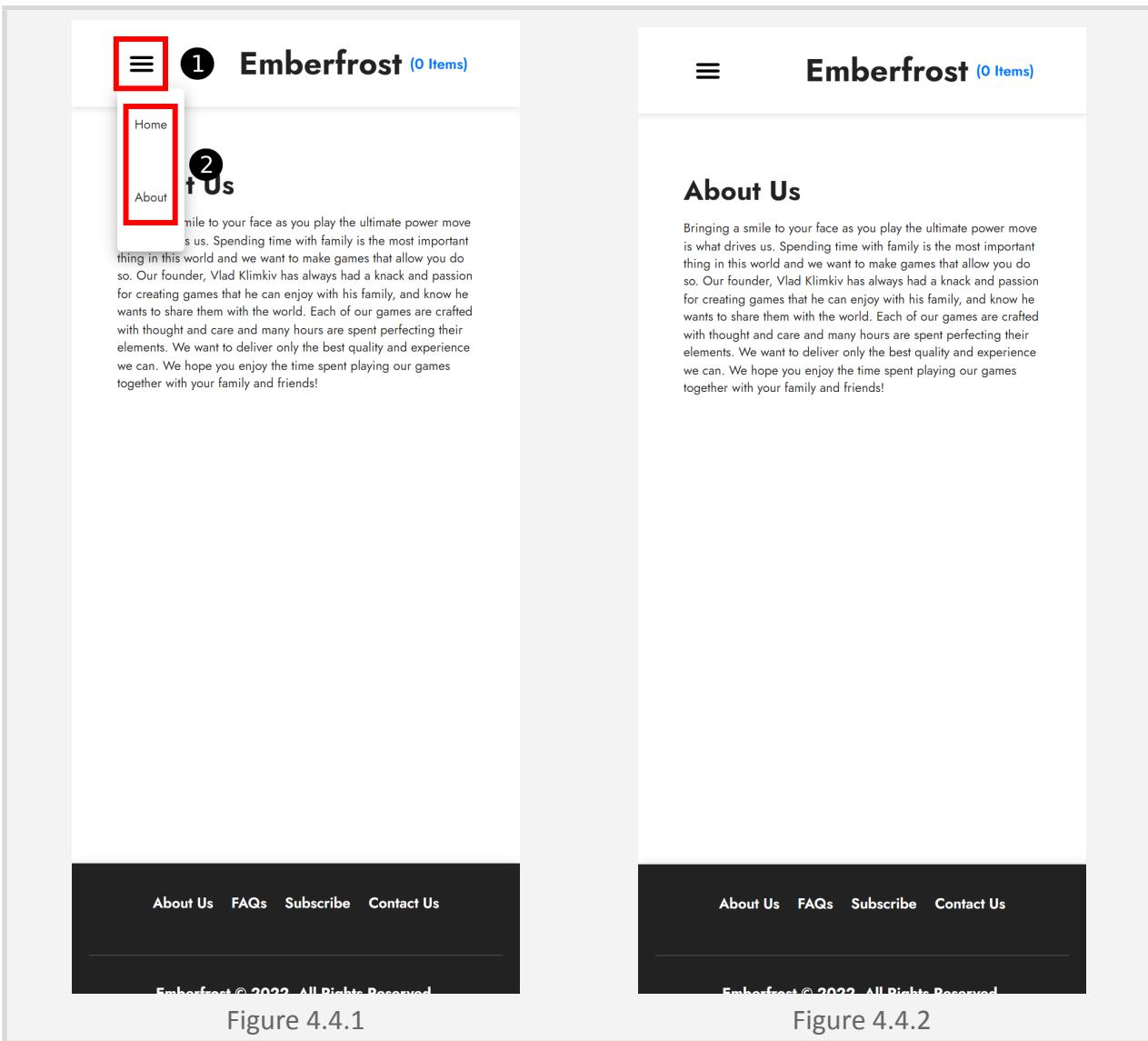
Figure 4.3.1

4.3 About Us Page Loading

Test Description: This test is to verify that the About page is loading. To perform this test, navigate to the bottom of the page using the scrollbar to hit the footer. Then click on the link to the FAQs page, Figure 4.2.1: Box 4. The test is successful if the About Us page renders, Figure 4.3.1.

Tester: Christian Suarez
Recorder: Monica Abigail Suase
Date: 04/29/2022
Specify Test input: No Input
Expected output: Figure 4.3.

- Verified
 Not Verified



4.4 Navbar responding

Test Description: This test is to verify that the Navbar is functional. To perform this test, click on the hamburger menu icon, Figure 4.4.1:Box 1. Then click on either the link to the Home or About Us page, Figure 4.4.1:Box 2. The test is successful if the page you click on renders. In this case, the About Us page renders, Figure 4.4.2.

Tester: Christian Suarez
Recorder: Monica Abigail Suase
Date: 04/29/2022
Specify Test input: No Input
Expected output: Figure 4.4.2.

- Verified
 Not Verified

5.0 Test Order Creation

The order creation section focuses on the process of creating an order. There is a flow to make sure the order is created out of the draft of the cart. Majority of it is on the backend, which includes Stripe and emailing the customer. Visual tests focus on making sure the front end UI process works.

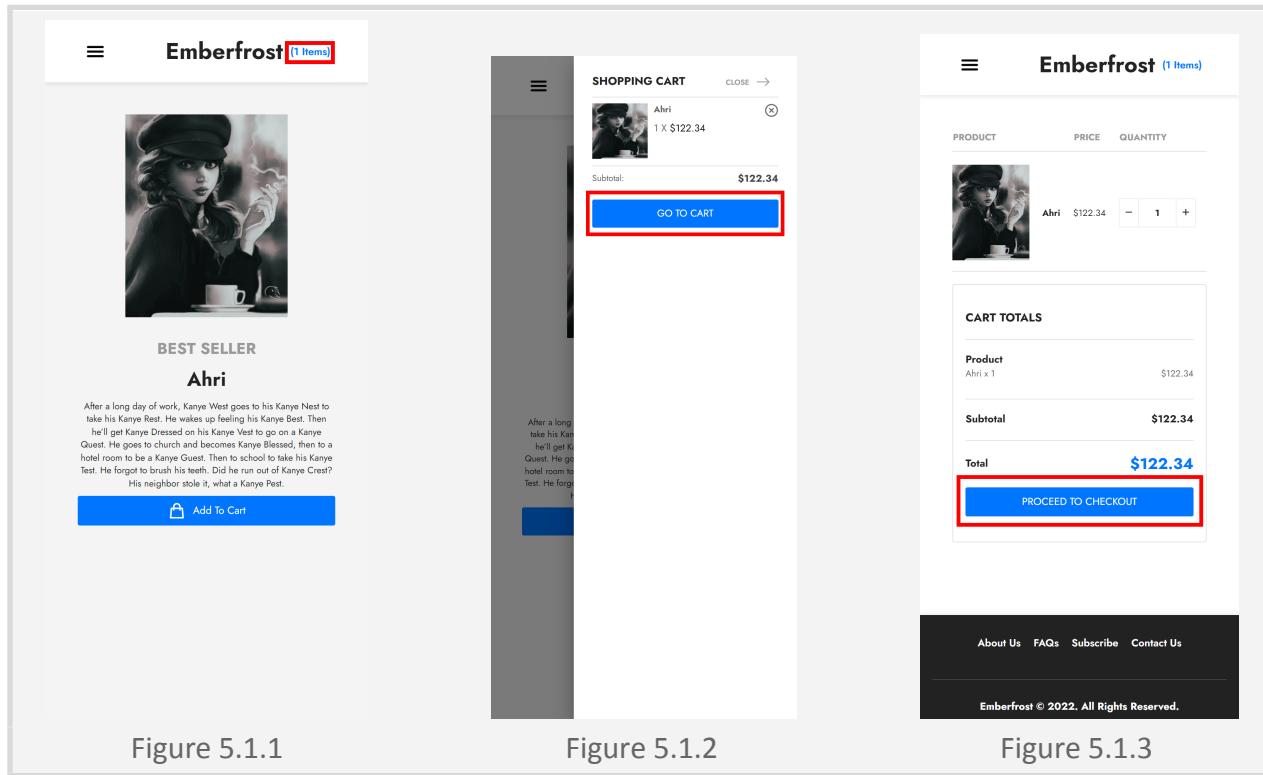


Figure 5.1.1

Figure 5.1.2

Figure 5.1.3

5.1 Going from cart to stripe

Test Description: This test is to verify that the Cart page can navigate to Stripe. To perform this test, click the “items” link to bring out the drawer menu, Figure 5.1.1. Then click on the “Go to Cart” button to navigate to the cart page Figure 5.1.2. Once on the Cart page, click on the “Proceed to Checkout” button to navigate to the Stripe checkout page, Figure 5.1.3. The test is successful if the Stripe checkout page loads, Figure 5.1.4

Tester: Christian Suarez
 Recorder: Monica Abigail Suase
 Date: 04/29/2022
 Specify Test input: Product Ahri
 Expected output: Figure 5.1.4

- Verified
- Not Verified

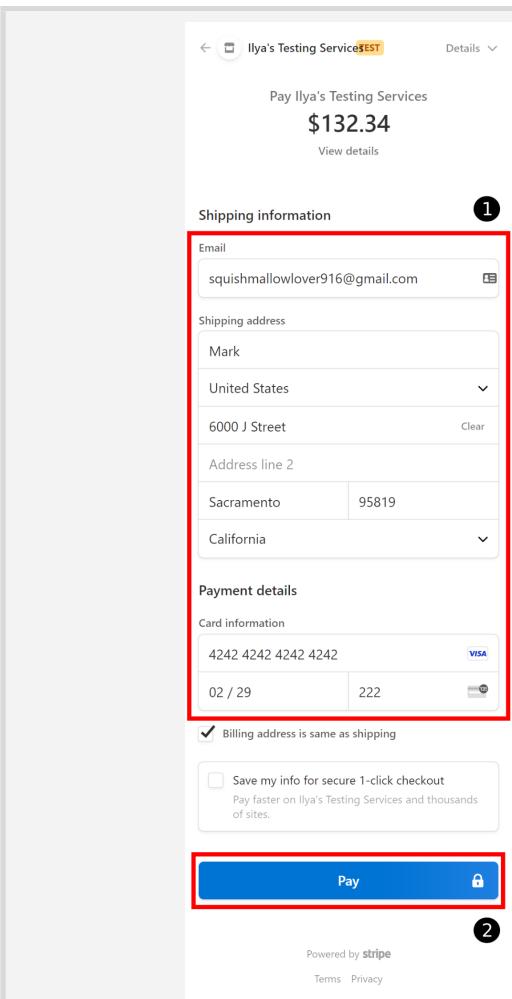


Figure 5.2.1

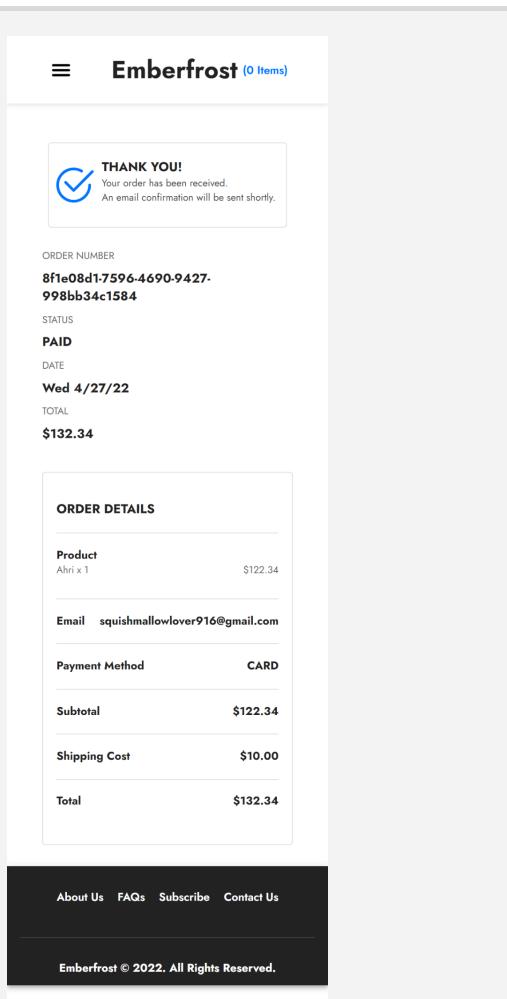


Figure 5.2.2

5.2 Going from cart to order page

Test Description: This test is to verify that the Cart page can navigate to Stripe. To perform this test, click the “items” link to bring out the drawer menu, Figure 5.1.1. Then click on the “Go to Cart” button to navigate to the cart page Figure 5.1.2. Once on the Cart page, click on the “Proceed to Checkout” button to navigate to the Stripe checkout page, Figure 5.1.3. Once on the Stripe checkout page, fill out the Stripe checkout form with a name, address, and supplied card number for testing, Figure 5.2.1: Box 1. Then click the “Pay” button to submit the form and complete the order, Figure 5.2.1: Box 2. The test is successful if the page redirects to the order page, Figure 5.2.2.

Tester: Christian Suarez
Recorder: Monica Abigail Suase
Date: 04/29/2022
Specify Test input: testing card: 4242 4242 4242 4242 date: 02/29 csv:222
Expected output: Figure 5.2.2

- Verified
- Not Verified

6.0 Frontend Test Suites

6.0.1 Requirements

Requirements: bash/powershell or zsh

- Both are available online and are free to use.

System Requirements:

- Node Version: v16.7.0
- Npm Version: v6.14.13
- Npm version allows you to maintain the desired version for server-supported packages.

6.0.2 Setup

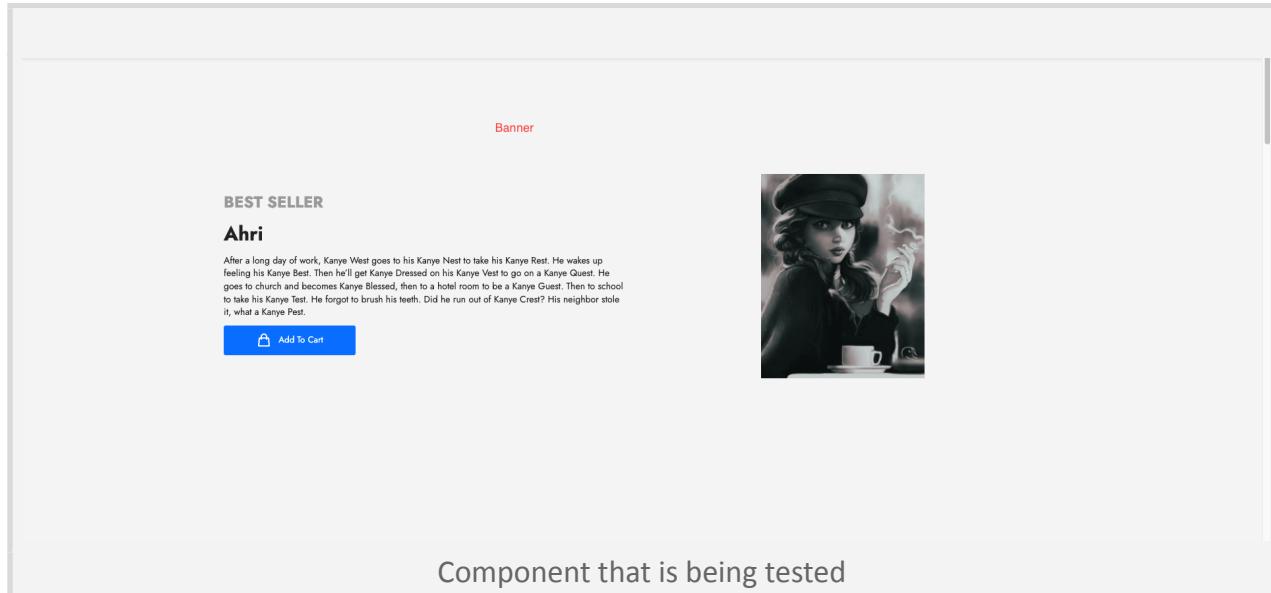
1. After pulling the repo to your local machine. The user has to install all of the required packages in order to run test.
2. In bash run `npm install`. This command will install are the required packages.
3. After the packages succesfully installed, make sure you are in the root directory before you run you test command. If you type linux command 'ls' and in the list of files see packages.json file. You are in the correct directoy.
4. Now run command “npm run test” in the command line.
 - a. This command will execute and run all suites at the same time.
5. If you want to test a specific file, you will have to run the test command and append the path to file.
 - a. Exmaple: npm run test ./somedir/somedfir/tests/testfile.test.ts
6. Now you have successfully run your tests.

6.1 Banner Component

Tester: Justin Singh

Recorder: Ilya Indik

Date: 04/29/2022



Component that is being tested

6.1.1 Component should show featured item when user loads home page

This test checks if the featured item is successfully queried from the backend and is rendered onto the screen. First we render our component. We then use our `useFeatueredItem` hook which makes a mock api call. We then check to see if the data we received from our mock is found in our component. If found the test passes.



6.1.2 Component should add to cart when the user pressed button

This test checks if we can add a product item to cart when the user presses a button in the banner component. We first render the component. We then use our useFeatueredItem hook which makes a mock api call. Once loaded the add to cart button should be visible so we query the dom to check if it exists. If it does we click the button and check to see if the success message was triggered. If triggered, our test passes.

```
1  test('should add to cart', async () => {
2    const spy = jest.spyOn(toast, 'success');
3    const { getByText } = render(<Banner />);
4    const featuredItemQuery = renderHook(() => useFeaturedItem(), {
5      wrapper: AllTheProviders,
6    });
7
8    await featuredItemQuery.waitFor(() => featuredItemQuery.result.current.isSuccess);
9
10   const shoppingCartServerState = renderHook(() => useShoppingCartServerState(), {
11     wrapper: AllTheProviders,
12   });
13
14   await shoppingCartServerState.waitFor(
15     () => shoppingCartServerState.result.current.itemsQuery.isSuccess
16   );
17
18   const addToCartButton = getByText('Add To Cart');
19
20   act(() => {
21     fireEvent.click(addToCartButton);
22   });
23
24   await waitFor(() => {
25     expect(spy).toHaveBeenCalled();
26   });
27 });
28
```

6.1.3 Component should show error message if request fails

This test checks if we are properly handling when the request fails. We first render the component. We then use our useFeaturedItem hook which makes a mock api call, but this time the mock should return an error instead of a successful response. We then check if the error message is being rendered onto the screen. If found, test passes.

```
1  test('should show error message', async () => {
2    server.use(...emberfrostApiFailedMockHandlers);
3    render(<Banner />);
4
5    const featuredItemQuery = renderHook(() => useFeaturedItem(), {
6      wrapper: AllTheProviders,
7    });
8
9    await featuredItemQuery.waitFor(() => {
10      expect(featuredItemQuery.result.current.isError).toBeTruthy();
11    });
12
13    await waitFor(() =>
14      screen.getByText(featuredItemQuery.result.current.error?.response?.data.error.message ?? '')
15    );
16  });
17});
```

6.1.4 Test Results

```
PASS components/Banner/index.test.tsx
<Banner/>
  ✓ should show featured item (81 ms)
  ✓ should add to cart (58 ms)
  ✓ should show error message (17 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.119 s
```

Evidence that all of our tests are passing

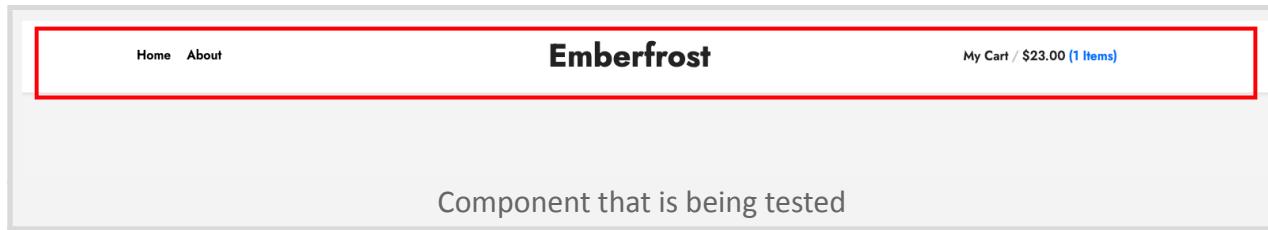
- Verified
- Not Verified

6.2 Navbar Component

Tester: Justin Singh

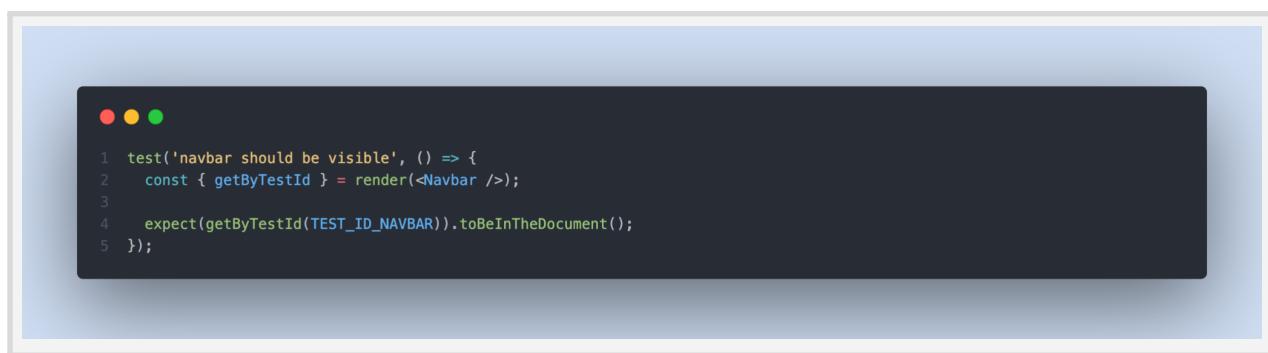
Recorder: Ilya Indik

Date: 04/29/2022



6.2.1 Component should be visible

This test checks if the navbar parent element is visible on the page. We first query the dom and find the parent element of our navbar. If found the test passes.



6.2.2 Component should render links

This test checks if the number of links visible on the page is the same as our array length of links. We first query the dom and find all elements that are links. If the number found is the same as our array length of links then the test passes.



6.2.3 Component should render cart link

This test checks if our cart link is visible on the page. We first query the dom and find the element that is the cart link. If found the test passes.

```
● ● ●  
1 test('cart link should be visible', () => {  
2   const { getByTestId } = render(<Navbar />);  
3  
4   expect(getByTestId(TEST_ID_CART)).toBeInTheDocument();  
5 });  
6
```

6.2.4 Component should remove item from cart when user clicks remove icon

This test checks if we can remove an item from the cart. First we query the dom for the button with the text “My Cart”. We then click the button to open the drawer to see items in our cart. We then wait for the items to be queried and rendered onto the screen. Once items are rendered we query the dom for the remove icon. Once we have the element we click it and check to see if the item is not in the document.

```
● ● ●  
1 test('should remove item from cart', async () => {  
2   window.matchMedia = jest.fn().mockImplementation(() => ({  
3     matches: true,  
4     addListener: jest.fn(),  
5     removeListener: jest.fn(),  
6   }));  
7  
8   const { getByText, getByTestId } = render(<Navbar />);  
9  
10  const drawerButton = getByText('My Cart');  
11  
12  act(() => {  
13    fireEvent.click(drawerButton);  
14  });  
15  
16  const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {  
17    wrapper: AllTheProviders,  
18  });  
19  
20  await waitForHook(() => result.current.itemsQuery.isSuccess);  
21  
22  const removeIcon = getByTestId(TEST_ID_CART_ITEM_REMOVE_ICON);  
23  
24  act(() => {  
25    if (removeIcon) {  
26      fireEvent.click(removeIcon);  
27    }  
28  });  
29  
30  await waitFor(() => expect(removeIcon).not.toBeInTheDocument());  
31});  
32
```

6.2.5 Test Results

```
PASS components/Navbar/index.test.tsx
  ✓ navbar should be visible (97 ms)
  ✓ links should be rendering (19 ms)
  ✓ cart link should be visible (15 ms)
  ✓ should remove item from cart (177 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
```

Evidence that all of our tests are passing

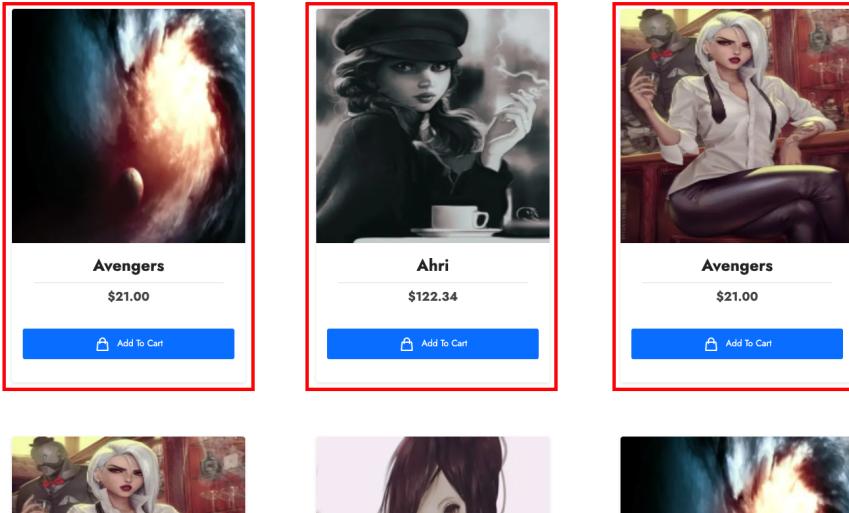
- Verified
- Not Verified

6.3 Product Item Component

Tester: Justin Singh

Recorder: Ilya Indik

Date: 04/29/2022



Component that is being tested

6.3.1 Component should render product image

This test checks to see if the product image is being rendered onto the page. We first render the component and then query the dom for the image element. If it is found the test passes.

```
● ● ●  
1 test('product item image is visible', () => {  
2   const data = productItemFactory.build();  
3   const { getByTestId } = render(<ProductItem data={data} />);  
4  
5   expect(getByTestId(TEST_ID_PRODUCT_ITEM_IMAGE)).toBeInTheDocument();  
6 });  
7
```

6.3.2 Component should label of product

This test checks to see if the product label is being rendered onto the page. We first render the component and then query the dom for an element with the following text of the product. If found the test passes.

```
● ● ●  
1 test('product item label is visible', () => {  
2   const data = productItemFactory.build();  
3   const { getByTestId } = render(<ProductItem data={data} />);  
4  
5   expect(getByTestId(TEST_ID_PRODUCT_ITEM_LABEL).textContent).toBe(data.name);  
6 });
```

6.3.3 Component should render product price

This test checks to see if the product price is being rendered onto the page. We first render the component and then query the dom for an element with the following text of the product price. If found the test passes.

```
● ● ●  
1 test('product item price is visible', () => {  
2   const data = productItemFactory.build();  
3   const { getByTestId } = render(<ProductItem data={data} />);  
4  
5   expect(getByTestId(TEST_ID_PRODUCT_ITEM_PRICE).textContent).toBe(  
6     currency(data.price, { fromCents: true }).format()  
7   );  
8 });
```

6.3.4 Component should render the add to cart button

This test checks to see if the add to cart button is being rendered onto the page. We first render the component and then query the dom for the button element. If found the test passes.

```
● ● ●
1 test('Add to cart button is visible', () => {
2   const data = productItemFactory.build();
3   const { getByTestId } = render(<ProductItem data={data} />);
4
5   expect(getByTestId(TEST_ID_PRODUCT_ITEM_BUTTON)).toBeInTheDocument();
6 });


```

6.3.5 Component should add product to users cart

This test checks to see if we can add a product to our cart from the product item component. We first render the component. We then use our useShoppingCartServerState which makes a series of mock requests to get items. We then query the add to cart button and then click it. If a success message is triggered the test passes.

```
● ● ●
1 test('Should add item to cart', async () => {
2   const spy = jest.spyOn(toast, 'success');
3   const data = productItemFactory.build({ guid: faker.datatype.uuid() });
4
5   const { getByTestId } = render(<ProductItem data={data} />);
6
7   const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
8     wrapper: AllTheProviders,
9   });
10
11  await waitForHook(() => result.current.initializeQuery.isSuccess);
12
13  const addToCartButton = getByTestId(TEST_ID_PRODUCT_ITEM_BUTTON);
14
15  act(() => {
16    fireEvent.click(addToCartButton);
17  });
18
19  await waitFor(() => {
20    expect(spy).toHaveBeenCalled();
21  });
22 });


```

6.3.6 Test Results

```
PASS components/ProductItem/index.test.tsx
✓ product item image is visible (49 ms)
✓ product item label is visible (13 ms)
✓ product item price is visible (11 ms)
✓ Add to cart button is visible (7 ms)
✓ Should add item to cart (66 ms)
```

```
Test Suites: 1 passed, 1 total
Tests:      5 passed, 5 total
```

Evidence that all the tests are passing

- Verified
- Not Verified

6.4 Cart Page Component

Tester: Justin Singh

Recorder: Ilya Indik

Date: 04/29/2022

The screenshot shows the Emberfrost application's cart page. At the top, there are navigation links for 'Home' and 'About'. The title 'Emberfrost' is centered above the main content. On the right, a link 'My Cart / \$719.89 (7 Items)' is visible. The main content area is divided into two sections: a list of items in the cart and a summary of the cart totals.

CART TOTALS

Product	Quantity	Subtotal
Avengers	1	\$21.00
Avengers	1	\$21.00
Ahri	1	\$122.34
Angie	2	\$23.00
Chair	1	\$324.21
Subtotal		\$719.89
Total		\$719.89

PROCEED TO CHECKOUT

Component that is being tested

6.4.1 Component should remove item from cart

This test checks to see if we can remove an item from our cart on the cart page. We first render the component. We then use our `useShoppingCartServerState` which makes a series of mock requests to get items. We then query the remove item button and then click it. After click we check if the item is still in the document. If it is not, the test passes.

```

1 test('should remove item from cart', async () => {
2   const { getByTestId } = render(<CartPage />);
3
4   const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
5     wrapper: AllTheProviders,
6   });
7
8   await waitForHook(() => result.current.itemsQuery.isSuccess);
9
10  const removeIcon = getByTestId(TEST_ID_REMOVE_ITEM_ICON);
11
12  act(() => {
13    if (removeIcon) {
14      fireEvent.click(removeIcon);
15    }
16  });
17
18  await waitFor(() => expect(removeIcon).not.toBeInTheDocument());
19});

```

6.4.2 Component should update quantity of item in cart

This test checks to see if we can update an item from our cart on the cart page. We first render the component. We then use our useShoppingCartServerState which makes a series of mock requests to get items. We then query the add quantity button After clicking we check if the item's quantity increments. If it did, the test passes.

```

1 test('should update quantity of item in cart', async () => {
2   const { getByTestId } = render(<CartPage />);
3
4   const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
5     wrapper: AllTheProviders,
6   });
7
8   await waitForHook(() => result.current.itemsQuery.isSuccess);
9
10  const quantityInput = getByTestId(TEST_ID_QUANTITY_NUMBER_FIELD) as HTMLInputElement;
11
12  const previousValue = parseInt(quantityInput.value, 10);
13
14  const addButton = getByTestId(TEST_ID_QUANTITY_ADDITION_BUTTON);
15
16  act(() => {
17    fireEvent.click(addButton);
18  });
19
20  await waitFor(() => expect(parseInt(quantityInput.value, 10)).toBe(previousValue + 1));
21});

```

6.4.3 Test results

```
<CartPage/>
  ✓ should remove item from cart (187 ms)
  ✓ should update quantity of item in cart (102 ms)

Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
```

Evidence that all the tests are passing

6.5.0 All frontend unit and integration tests

```
PASS components/Navbar/index.test.tsx
PASS components/ProductItem/index.test.tsx
PASS __tests__/cartPage.test.tsx
PASS components/forms/BillingDetailsForm/index.test.tsx
PASS components/CartItem/index.test.tsx
PASS components/ProductDescription/index.test.tsx
PASS components/forms/ContactForm/index.test.tsx
PASS __tests__/productViewPage.test.tsx
PASS components/OrderPlacedThankYou/index.test.tsx
PASS components/InfoItem/index.test.tsx
PASS components/ContactCard/index.test.tsx
PASS components/OrderPlacedSummary/index.test.tsx
PASS components/Quantity/index.test.tsx
PASS components/EmptyCart/index.test.tsx
PASS components/Footer/index.test.tsx
PASS components/forms/SubscribeForm/index.test.tsx
PASS components/Banner/index.test.tsx

Test Suites: 17 passed, 17 total
Tests:      59 passed, 59 total
```

Evidence that all the tests are passing

7.0 Back End Unit and Integration Tests Suites

7.0.1 Requirements

Requirements: bash/powershell

- Both are available online and are free to use.

System Requirements:

- Node Version: v16.7.0**
- Since we are using Crypto, a Node.js module that provides cryptographic functionality for our random guid values in tests. Has to be version 16.7 and greater for tests to function correctly.
- Npm Version: v6.14.13**
- Npm version allows you to maintain the desired version for server-supported packages.

7.0.2 Setup

7. After pulling the repo to your local machine. The user has to install all of the required packages in order to run the test.
8. In bash run `npm install`. This command will install the required packages.
9. After the packages are successfully installed, make sure you are in the root directory before you run your test command. If you type linux command 'ls' and in the list of files see packages.json file. You are in the correct directory.
10. Now run the command “**npm run test**” in the command line.
- a. This command will execute and run all suites at the same time. **Figure 7.0.2.1**
11. If you want to test a specific file, you will have to run the test command and append the path to file. **Figure 7.0.2.2**
- a. Example: **npm run test ./somedir/somedfir/tests/testfile.test.ts**
12. Now you have successfully run your tests.

```
bash > emberfrost-estore-api (MOC-258): npm run test
> ember@0.0.2 test C:\Users\IlyaI\Desktop\Project\EmberFrostNew\emberfrost-estore-api
> jest [-no-cache --detectOpenHandles --forceExit

    ✓ userSubscribe (2443 ms)
    ✓ userSubscribe - Error (667 ms)

Test Suites: 17 passed, 17 total
Tests:    79 passed, 79 total
Snapshots: 0 total
Time:      40.316 s
Ran all test suites.
```

Figure 7.0.2.1

```

bash > emberfrost-estore-api (MOC-258): npm run test ./src/ErrorHandling/ErrorHandlingTests/ApiError.test.ts
> ember@0.0.2 test C:\Users\Ilya\Desktop\Project\EmberFrostNew\emberfrost-estore-api
> jest --no-cache --detectOpenHandles --forceExit "./src/ErrorHandling/ErrorHandlingTests/ApiError.test.ts"

PASS  src/ErrorHandling/ErrorHandlingTests/ApiError.test.ts
  ApiError
    ✓ should create an instance of ApiError (4 ms)
    ✓ API Error (2 ms)
    ✓ API Error - thrown (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        2.243 s
Ran all test suites matching ./src/ErrorHandling/ErrorHandlingTests/ApiError.test.ts/i.

```

Figure 7.0.2.2

7.1.0 Running Test Suites

7.1.1 HTTP Controller Suite

This suite of tests - tests the entry point functions for all of the implemented routes. Generally each method is tested 2 times. 1 time - expected to successfully return data and a 200 status. The second time it is expected to throw an error.

```

PASS  src/HTTPControllers/HTTPTests/CartController.test.ts
  ✓ getCart (4 ms)
  ✓ getCart - cart not found (2 ms)
  ✓ addItem (1 ms)
  ✓ addItem - cart not found (1 ms)
  ✓ clearCart (1 ms)
  ✓ clearCart - cart not found (1 ms)
  ✓ updateItem (1 ms)
  ✓ updateItem - item not found (1 ms)
  ✓ removeItem (2 ms)
  ✓ removeItem - item not found (2 ms)
  ✓ initCart (1 ms)
  ✓ initCart - DB Error (1 ms)

```

The image above shows the tests for all of the CartController. Each method has an expected normal response and also a test for a thrown error. Each HTTP Controller file has similar tests as above for its methods.

7.1.2 Error Handling Suite

Some examples of errors are 404 Not Found and 500 errors when something goes wrong on a database or application level. Additionally we can see package specific errors, like a SendGrid Error. Each custom error class is tested for 3 things. Instantiation, toJSON() and expected to be thrown. Below is an example of some error classes.

```
PASS src/ErrorHandling/ErrorHandlingTests/ApplicationError.test.ts
ApplicationError
  ✓ should create an instance of ApplicationError (2 ms)
  ✓ Application Error toJSON (2 ms)
  ✓ Application Error - thrown

PASS src/ErrorHandling/ErrorHandlingTests/AuthenticationError.test.ts
AuthenticationError
  ✓ should create an instance of AuthenticationError (1 ms)
  ✓ Authentication Error to JSON (1 ms)
  ✓ Authentication Error - thrown

PASS src/ErrorHandling/ErrorHandlingTests/NotFoundError.test.ts
NotFoundError
  ✓ should create an instance of NotFoundError (2 ms)
  ✓ NotFound Error to JSON (2 ms)
  ✓ NotFoundError - thrown (1 ms)
```

7.1.3 Services Test Suite

The service layer holds the bulk of the application business logic and is tested in a similar fashion as the HTTPControllers. Each service class method has one test that is expected to succeed without throwing any errors and one test that will throw an error. Below are some examples.

```
PASS src/Services/ServiceTests/SupportService.test.ts
  ✓ createSupportRequest (415 ms)
  ✓ createSupportRequest - MissingDataError (397 ms)

PASS src/Services/ServiceTests/UserService.test.ts
  ✓ userSubscribe (2139 ms)
  ✓ userSubscribe - Error (563 ms)
```

7.1.4 Sample Test Code

Below is a set of tests for the `createQuestion` method for the `FAQService` class.

```
18  test('createQuestions', async () => {
19    question = await FAQService.createQuestion({
20      question: 'test',
21      answer: 'test',
22      type: 'GENERAL'
23    });
24
25    expect(question.question).toBe('test');
26  });
27
28  test('createQuestion - Missing Payload', async () => {
29    try {
30      await FAQService.createQuestion({});
31    } catch (err) {
32      expect(err).toBeDefined();
33    }
34  });

```

As we can see there are two tests, the first calls the Service method with a payload and expects it to succeed and respond with the value that was sent in to begin with. The second doesn't pass in a proper payload and is wrapped in a try-catch block as it is expecting a `MissingDataError` to be thrown.

7.1.5 Dealing With Failing Tests

Below is an example of tests that failed and the output from the terminal regarding those tests.

```
Test Suites: 2 failed, 15 passed, 17 total
Tests:       16 failed, 63 passed, 79 total
Snapshots:   0 total
Time:        27.795 s
Ran all test suites.
.
→ 18 item = await PrismaClient.item.create(
  Error occurred during query execution:
  ConnectorError(ConnectorError { user_facing_error: None, kind: QueryError(Error { kind: Db, cause: SqlState("22P03"), message: "incorrect binary data format in bind parameter 5", detail: None, hint: None, datatype: None, constraint: None, file: Some("postgres.c"), line: Some(1895), routine: Some("39070 |      throw new PrismaClientRustPanicError(message, this.client._clientVersion);
39071 |    } else if (e instanceof PrismaClientUnknownRequestError) {
39072 |      throw new PrismaClientUnknownRequestError(message, this.client._clientVersion);
|      ^
|      ^
```

When this happens, this can be one or a combination of a few things. You might be testing against an outdated/not-up-to-date prisma Schema so certain calls fail since the code might be referencing fields that don't exist (as seen in the error above). Or it may be that there is a dependency that hasn't been installed on the local machine. If the developer has worked on the code and now the tests are breaking - it is now the developer's job to determine where they're failing and if it's his code that needs to be changed or the tests need to be updated to expect new outputs/inputs.

System Test Report

System testing is the process whereby the complete software application is tested. Testing intends to evaluate and verify the system's compliance with specified requirements.

Product Owner

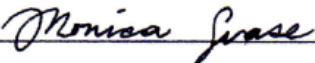
Vlad Klimkiv

CEO

EmberFrost Corp.

This document holds the signatures of every team member who participated in the testing of the application.

Team Members: System Test Verification Completed: Date: 04/29/2029

Name:	Signature:
Ilya Indik	
Andrey Kurudimov	
Justin Singh	
Christian Suarez	
Jared Petersen	
Monica Abigail Suase	

Client:

Vladimir Klimkiv



DATE: 05/06/2022