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

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.

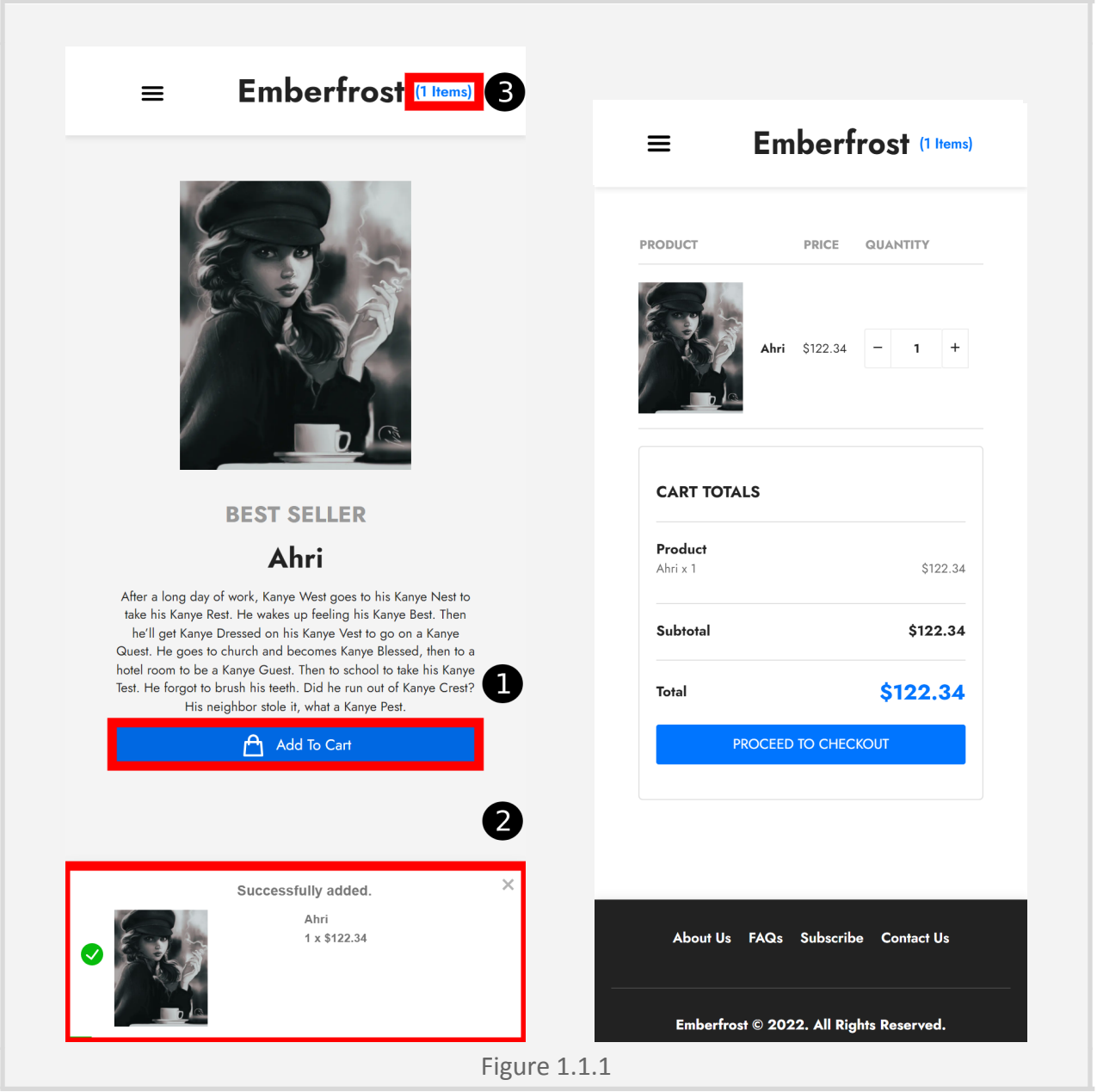


Figure 1.1.1

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

- [x] Verified
- [ ] Not Verified

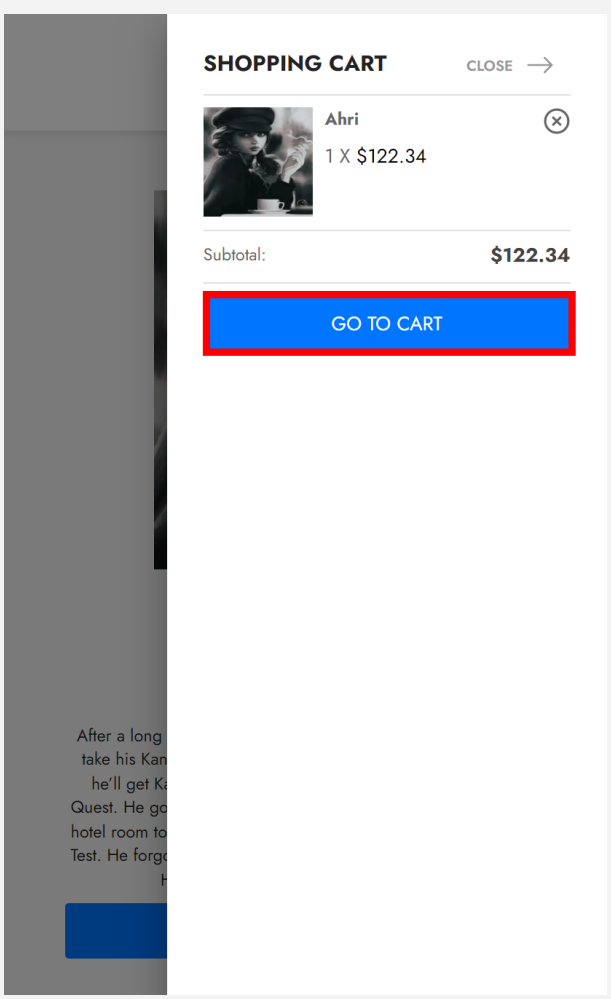


Figure 1.2.1

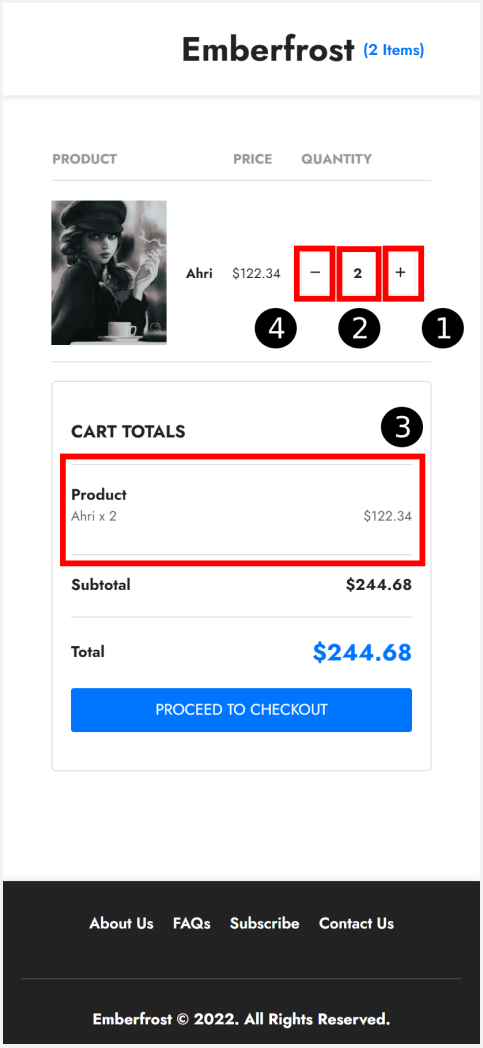


Figure 1.2.2

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

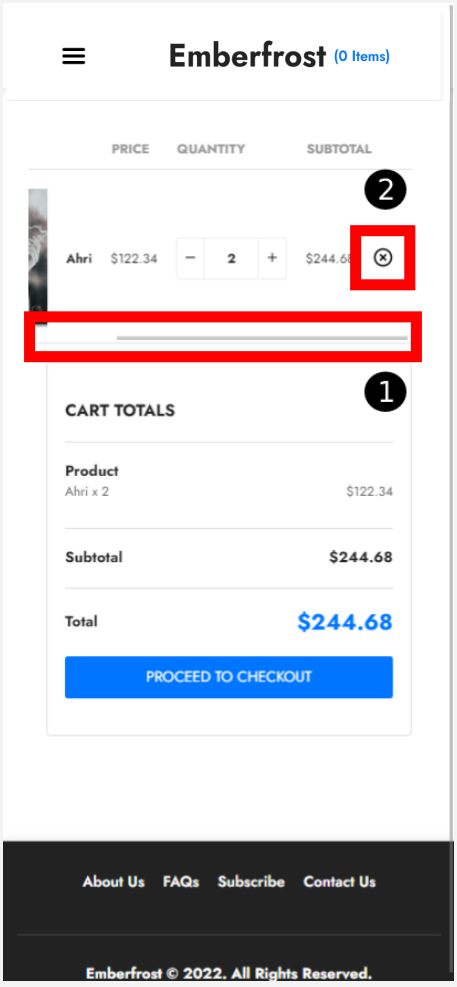


Figure 1.4.1

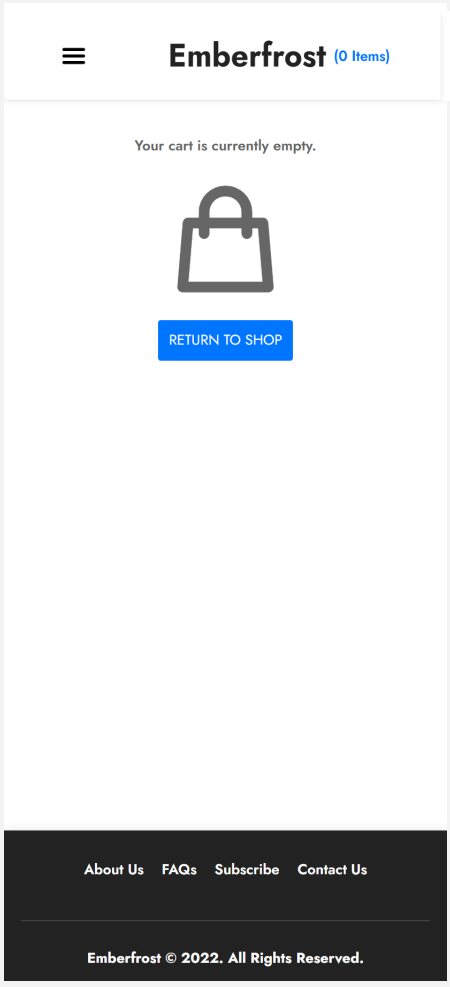


Figure 1.4.2

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 “⊗” 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.

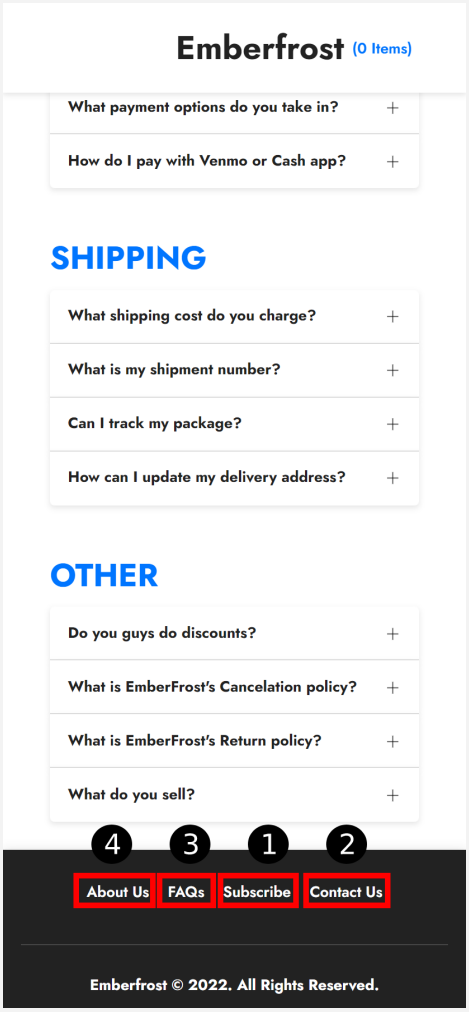


Figure 2.1.1

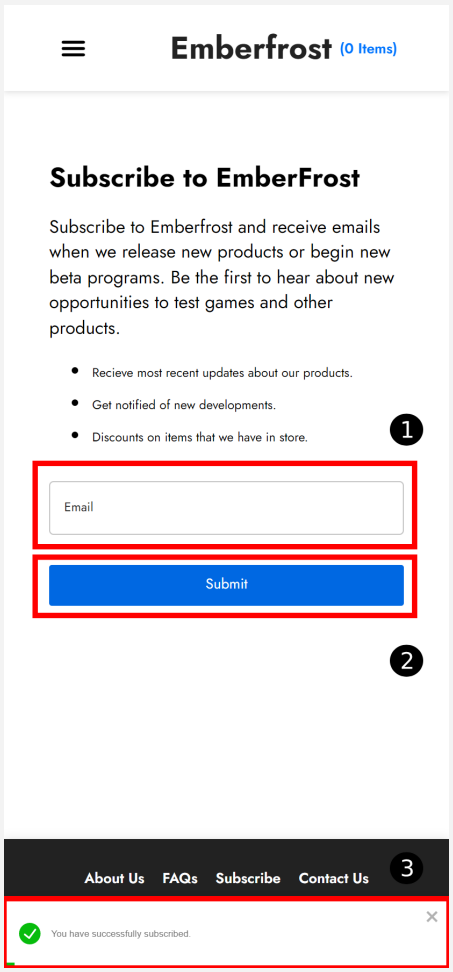


Figure 2.1.2

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

- [x] Verified
- [ ] Not Verified

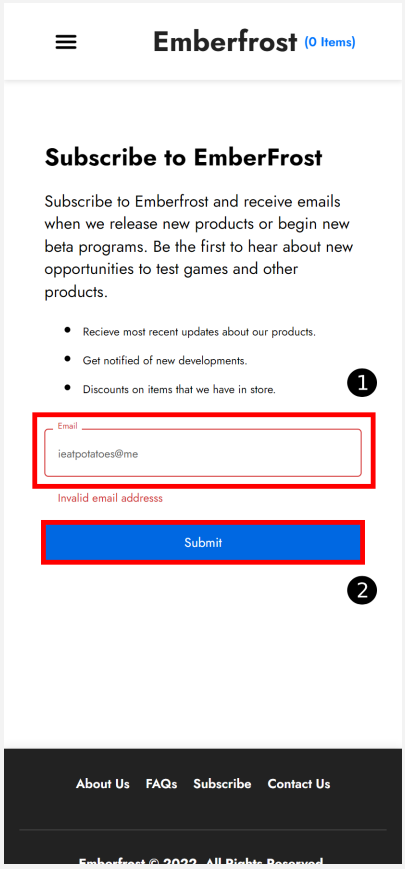


Figure 2.2.1

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

- [x] Verified
- [ ] Not Verified

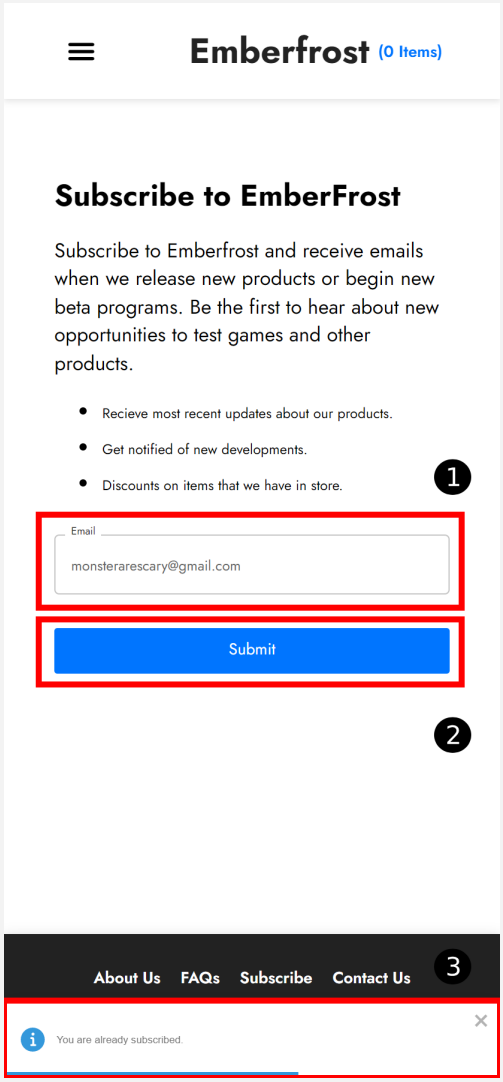


Figure 2.3.1

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.

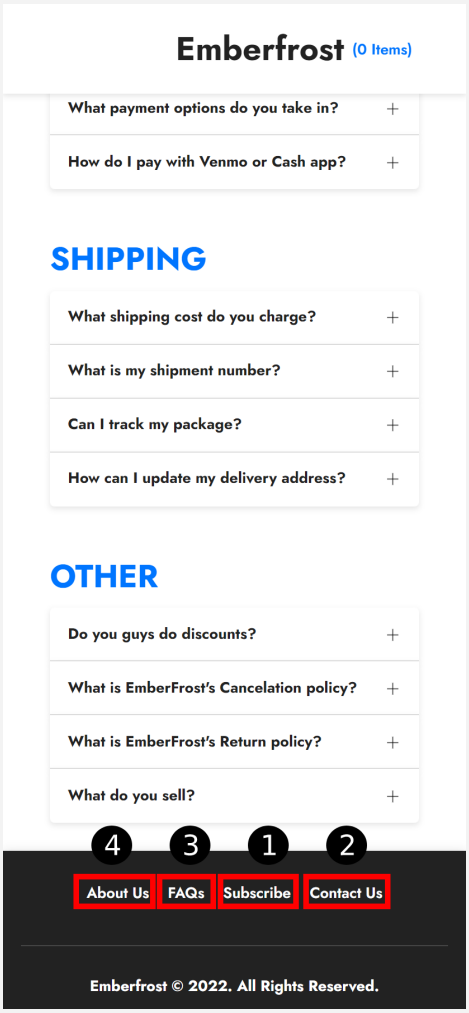


Figure 3.1.1

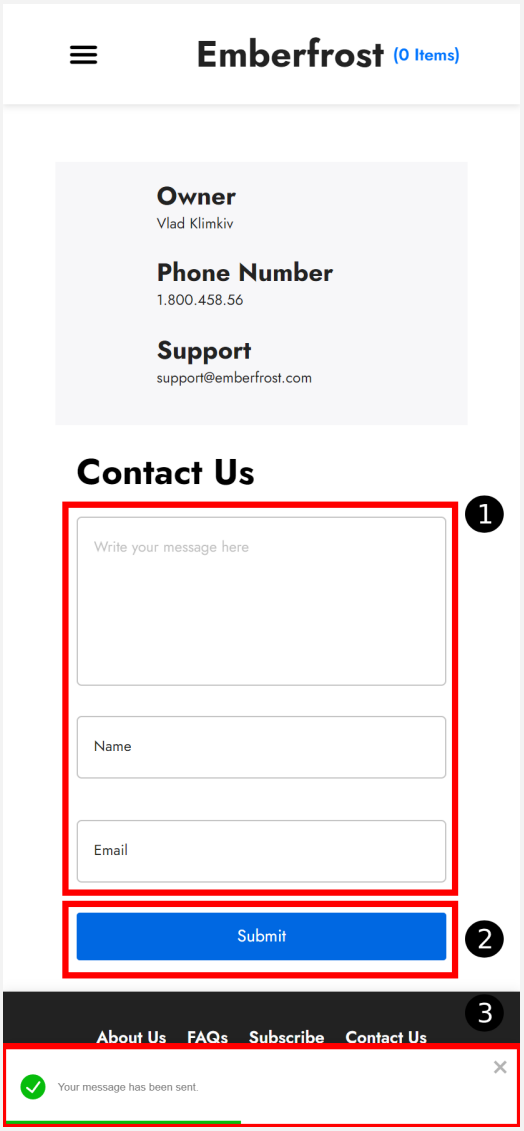


Figure 3.1.2

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

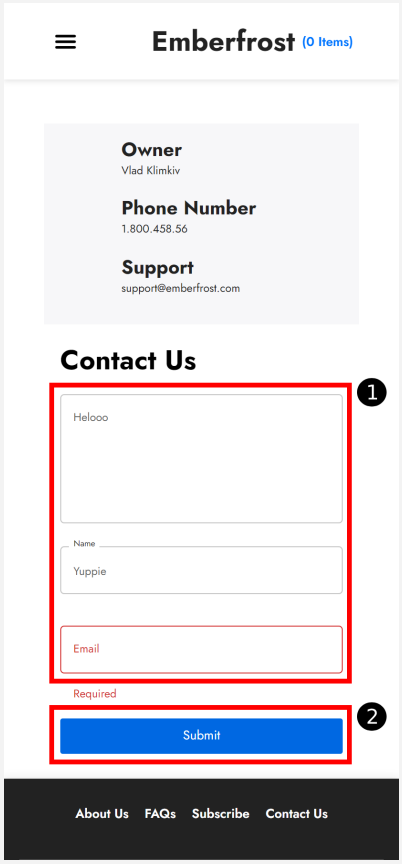


Figure 3.2.1

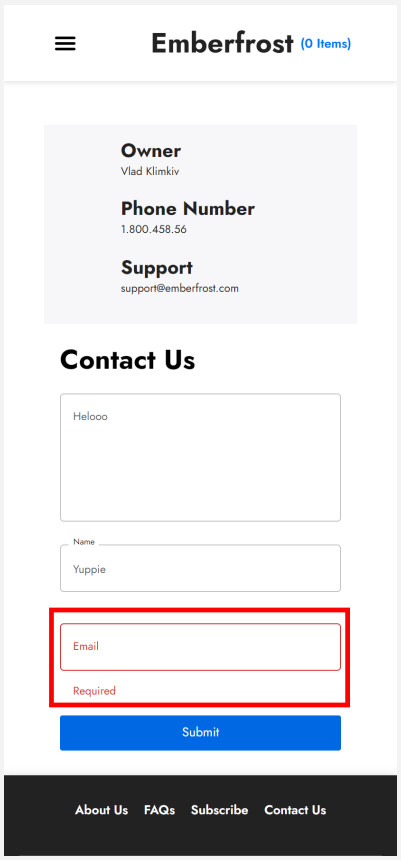


Figure 3.2.2

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

- [x] 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.

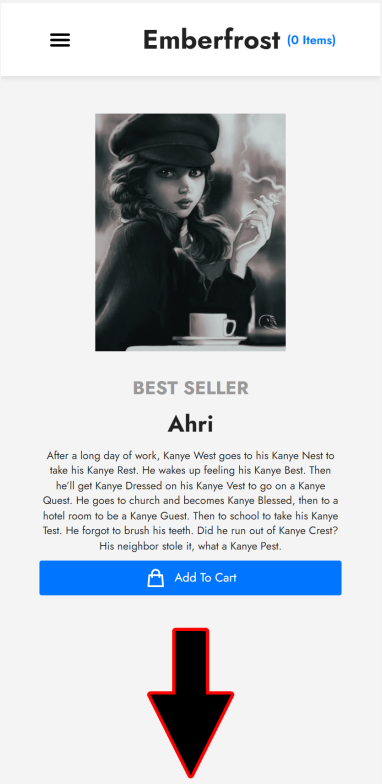


Figure 4.1.1

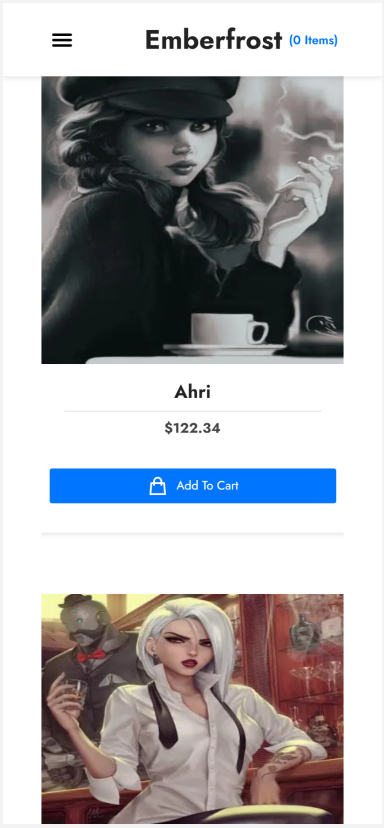


Figure 4.1.2

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.

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

- [x] Verified
- [ ] Not Verified

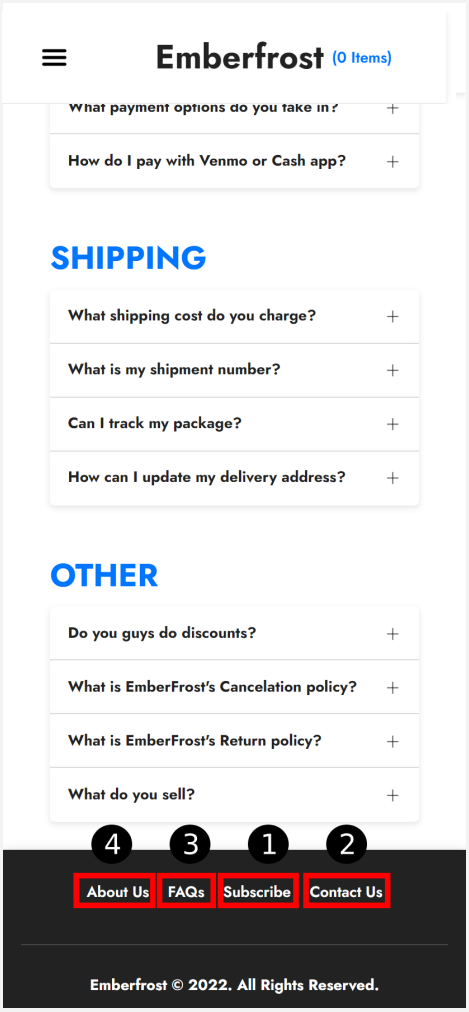


Figure 4.2.1

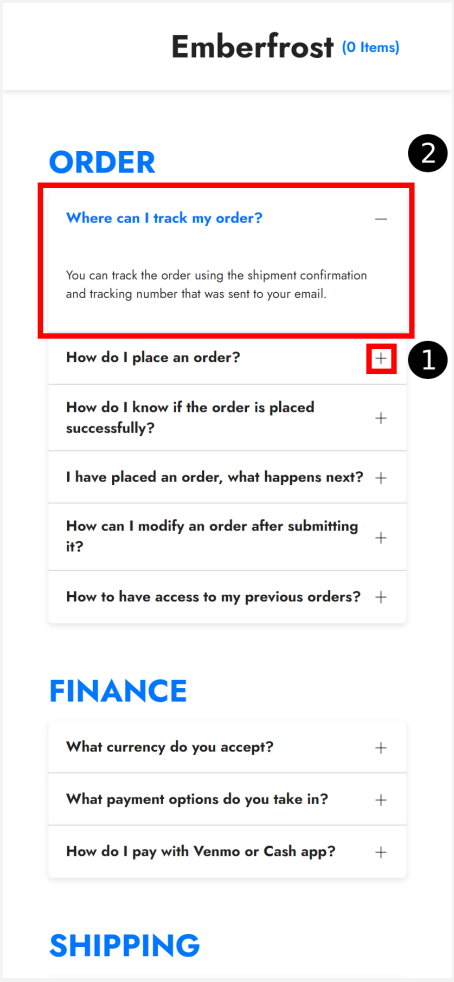


Figure 4.2.2

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

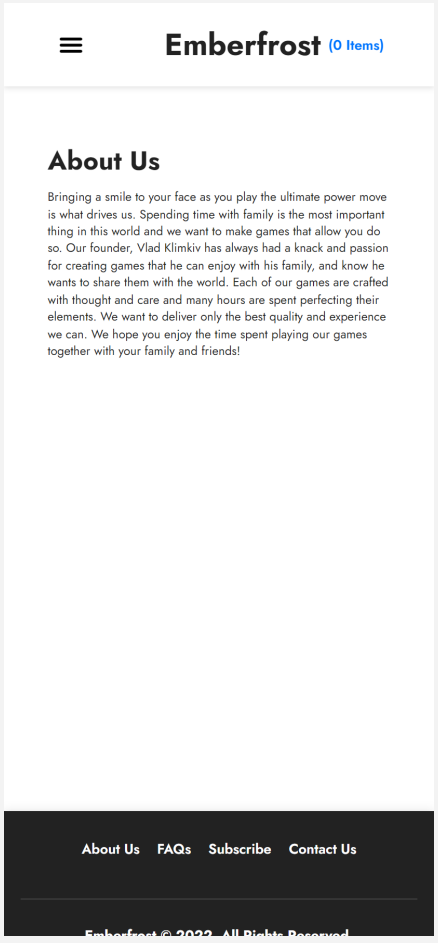


Figure 4.3.1

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.

- [x] Verified
- [ ] Not Verified

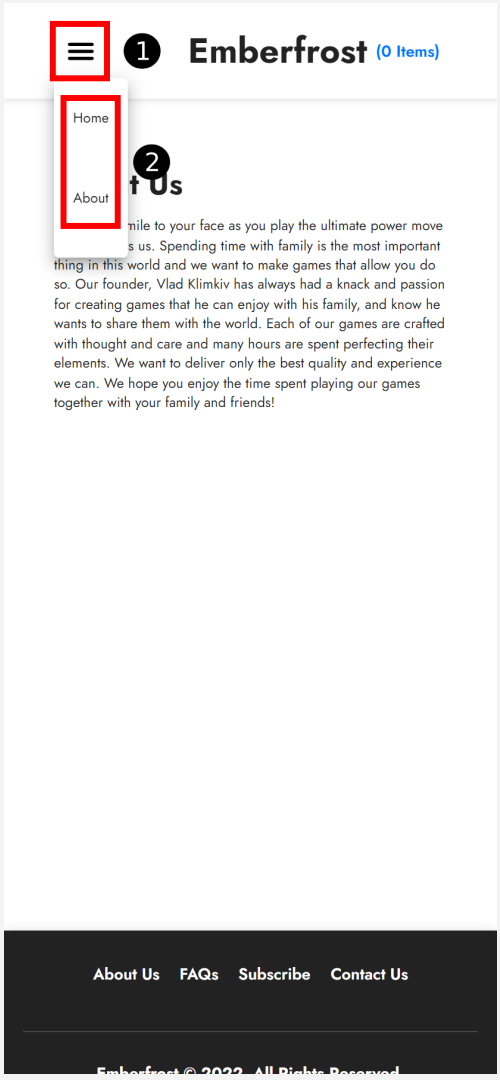


Figure 4.4.1

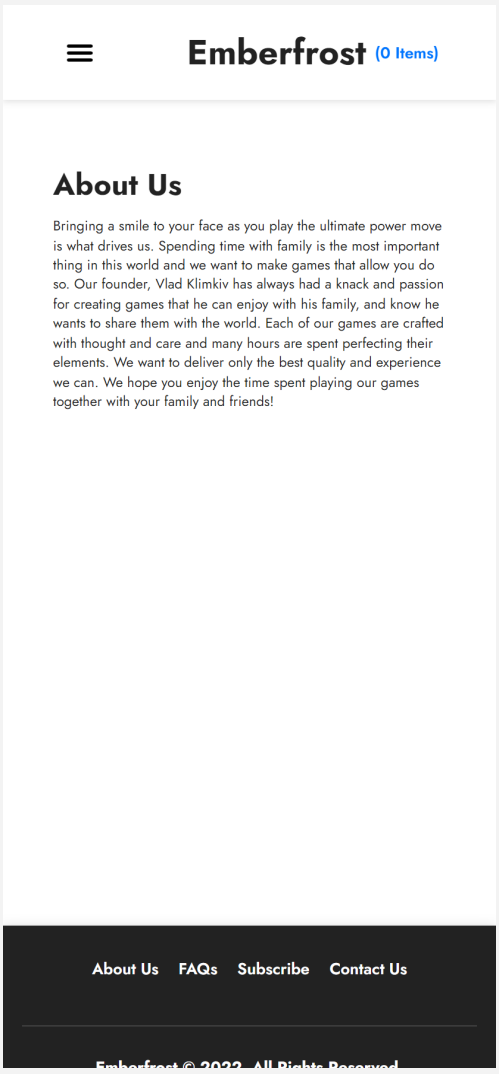


Figure 4.4.2

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.

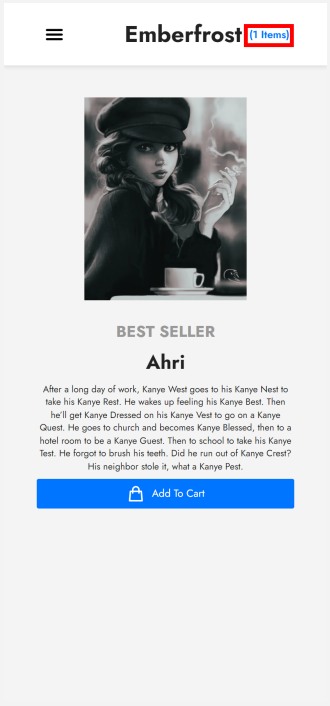


Figure 5.1.1

Figure 5.1.2

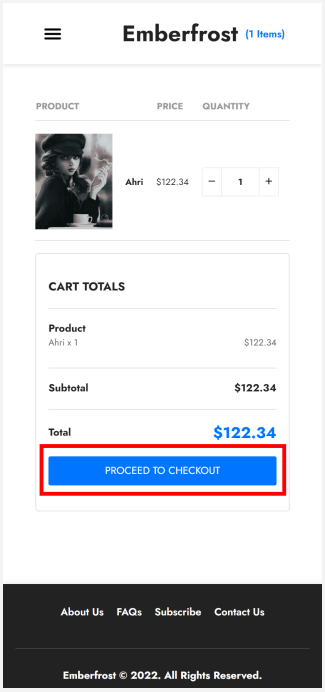


Figure 5.1.3

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

- [x] Verified
- [ ] Not Verified

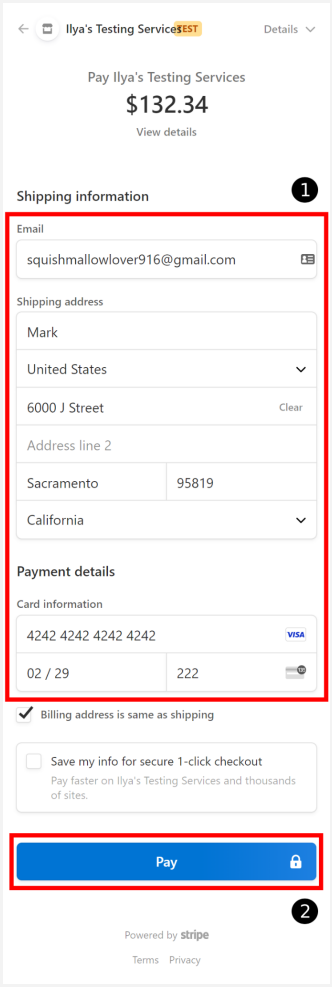


Figure 5.2.1

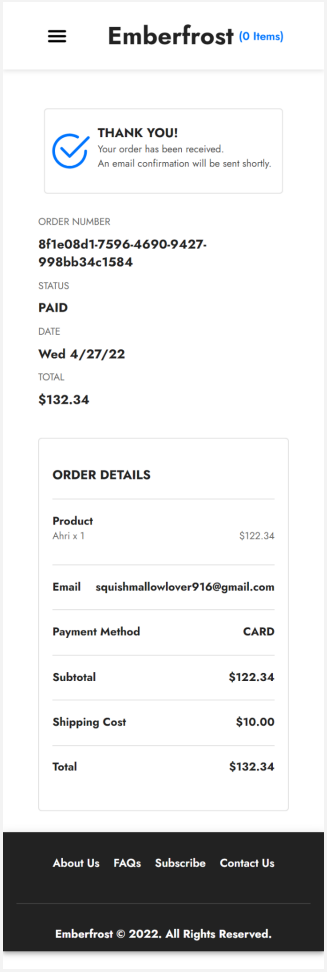


Figure 5.2.2

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:

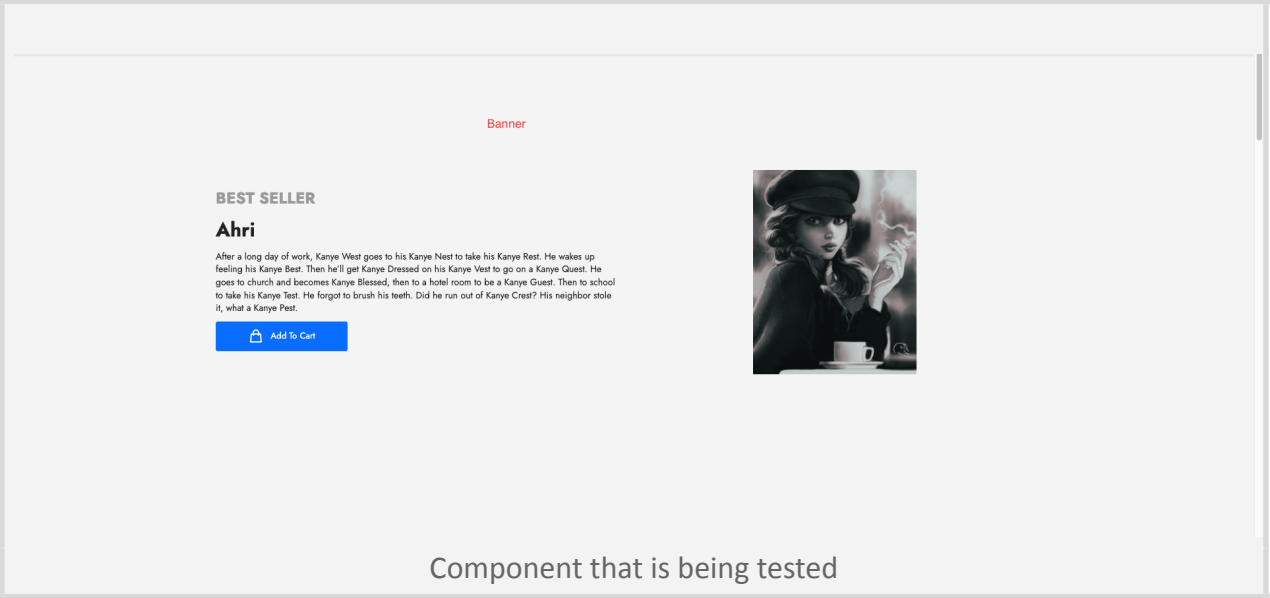
- ☐ **Node Version: v16.7.0**
- ☐ **Npm Version: v6.14.13**
- Npm version allows you to maintain the desired version for server-supported packages.

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.

```
test('should show featured item', async () => {
  render(<Banner />);

  const featuredItemQuery = renderHook(() => useFeaturedItem(), {
    wrapper: AllTheProviders,
  });

  await featuredItemQuery.waitFor(() => featuredItemQuery.result.current.isSuccess);

  await waitFor(() => {
    screen.getByText(featuredItemQuery.result?.current?.data?.name ?? '');
    screen.getByText(featuredItemQuery.result?.current?.data?.featureDescription ?? '');
  });
});
```

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.

```
test('should add to cart', async () => {
  const spy = jest.spyOn(toast, 'success');
  const { getByText } = render(<Banner />);
  const featuredItemQuery = renderHook(() => useFeaturedItem(), {
    wrapper: AllTheProviders,
  });

  await featuredItemQuery.waitFor(() => featuredItemQuery.result.current.isSuccess);

  const shoppingCartServerState = renderHook(() => useShoppingCartServerState(), {
    wrapper: AllTheProviders,
  });

  await shoppingCartServerState.waitFor(
    () => shoppingCartServerState.result.current.itemsQuery.isSuccess
  );

  const addToCartButton = getByText('Add To Cart');

  act(() => {
    fireEvent.click(addToCartButton);
  });

  await waitFor(() => {
    expect(spy).toHaveBeenCalled();
  });
});

```

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 useFeatueredItem 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.

```
test('should show error message', async () => {
    server.use(...emberfrostApiFailedMockHandlers);
    render(<Banner />);

    const featuredItemQuery = renderHook(() => useFeaturedItem(), {
      wrapper: AllTheProviders,
    });

    await featuredItemQuery.waitFor(() => {
      expect(featuredItemQuery.result.current.isError).toBeTruthy();
    });

    await waitFor(() =>
      screen.getByText(featuredItemQuery.result.current.error?.response?.data.error.message ?? '')
    );
  });
```

6.1.4 Test Results

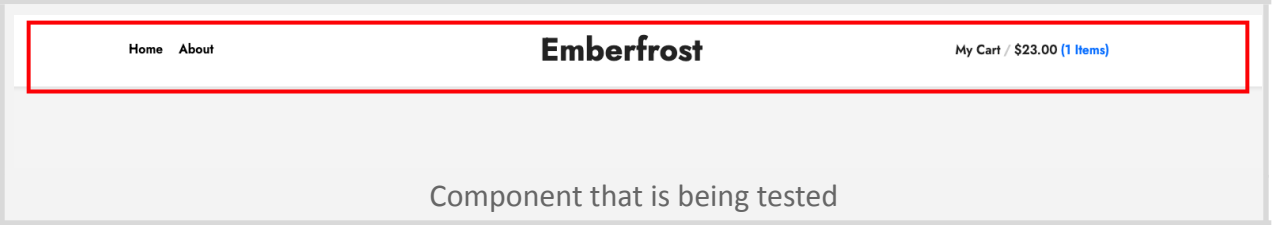
```
PASS components/Banner/index.test.tsx
  <Banner/>
    ✓ should show featured item (81 ms)
    ✓ should add to cart (58 ms)
    ✓ should show error message (17 ms)

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.

```
test('navbar should be visible', () => {
  const { getByTestId } = render(<Navbar />);

  expect(getByTestId(TEST_ID_NAVBAR)).toBeInTheDocument();
});
```

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.

```
test('links should be rendering', () => {
  const { getAllByTestId } = render(<Navbar />);

  expect(getAllByTestId(TEST_ID_LINKS)).toHaveLength(LINKS.length);
});
```

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.

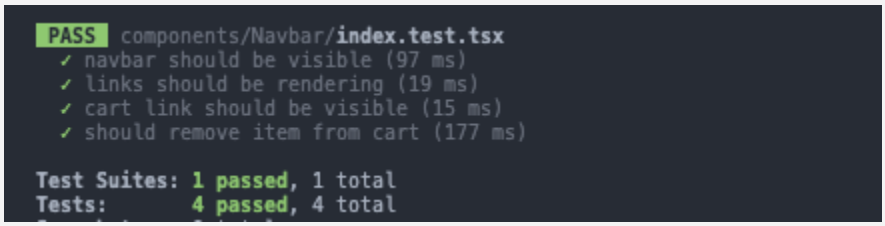
```
test('cart link should be visible', () => {
  const { getByTestId } = render(<Navbar />);

  expect(getByTestId(TEST_ID_CART)).toBeInTheDocument();
});

```

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.

```
test('should remove item from cart', async () => {
  window.matchMedia = jest.fn().mockImplementation(() => ({
    matches: true,
    addListener: jest.fn(),
    removeListener: jest.fn(),
  }));

  const { getByText, getByTestId } = render(<Navbar />);

  const drawerButton = getByText('My Cart');

  act(() => {
    fireEvent.click(drawerButton);
  });

  const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
    wrapper: AllTheProviders,
  });

  await waitForHook(() => result.current.itemsQuery.isSuccess);

  const removeIcon = getByTestId(TEST_ID_CART_ITEM_REMOVE_ICON);

  act(() => {
    if (removeIcon) {
      fireEvent.click(removeIcon);
    }
  });

  await waitFor(() => expect(removeIcon).not.toBeInTheDocument());
});

```

6.2.5 Test Results

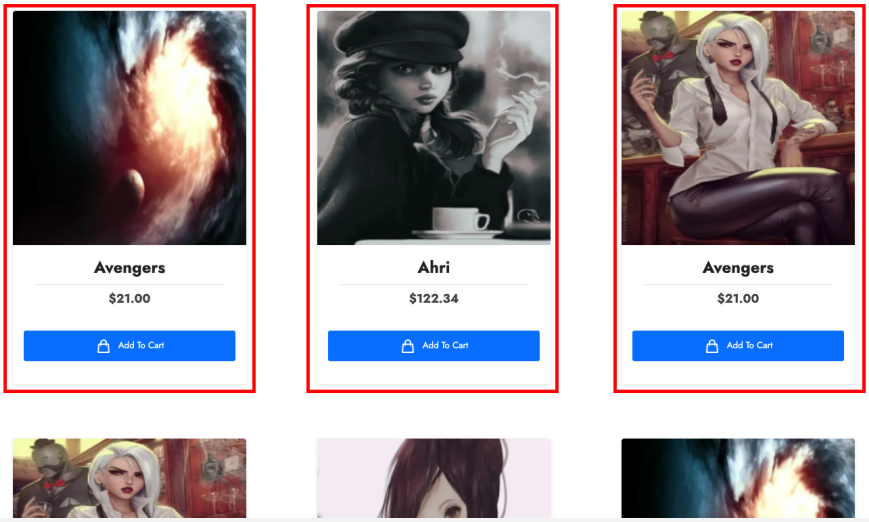


Evidence that all of our tests are passing

- ☑ 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.

```
test('product item image is visible', () => {
  const data = productItemFactory.build();
  const { getByTestId } = render(<ProductItem data={data} />);

  expect(getByTestId(TEST_ID_PRODUCT_ITEM_IMAGE)).toBeInTheDocument();
});
```

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.

```
test('product item label is visible', () => {
  const data = productItemFactory.build();
  const { getByTestId } = render(<ProductItem data={data} />);

  expect(getByTestId(TEST_ID_PRODUCT_ITEM_LABEL).textContent).toBe(data.name);
});
```

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.

```
test('product item price is visible', () => {
  const data = productItemFactory.build();
  const { getByTestId } = render(<ProductItem data={data} />);

  expect(getByTestId(TEST_ID_PRODUCT_ITEM_PRICE).textContent).toBe(
    currency(data.price, { fromCents: true }).format()
  );
});
```

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.

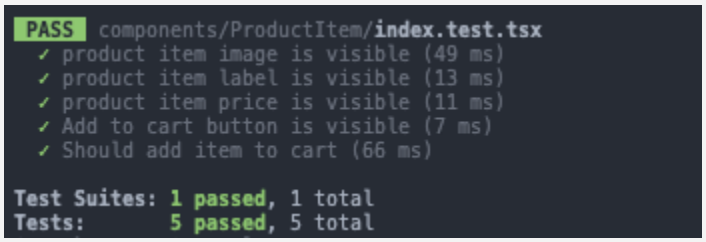
```
test('Add to cart button is visible', () => {
  const data = productItemFactory.build();
  const { getByTestId } = render(<ProductItem data={data} />);

  expect(getByTestId(TEST_ID_PRODUCT_ITEM_BUTTON)).toBeInTheDocument();
});
```

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.

```
test('Should add item to cart', async () => {
  const spy = jest.spyOn(toast, 'success');
  const data = productItemFactory.build({ guid: faker.datatype.uuid() });

  const { getByTestId } = render(<ProductItem data={data} />);

  const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
    wrapper: AllTheProviders,
  });

  await waitForHook(() => result.current.initializeQuery.isSuccess);

  const addToCartButton = getByTestId(TEST_ID_PRODUCT_ITEM_BUTTON);

  act(() => {
    fireEvent.click(addToCartButton);
  });

  await waitFor(() => {
    expect(spy).toHaveBeenCalled();
  });
});
```

6.3.6 Test Results

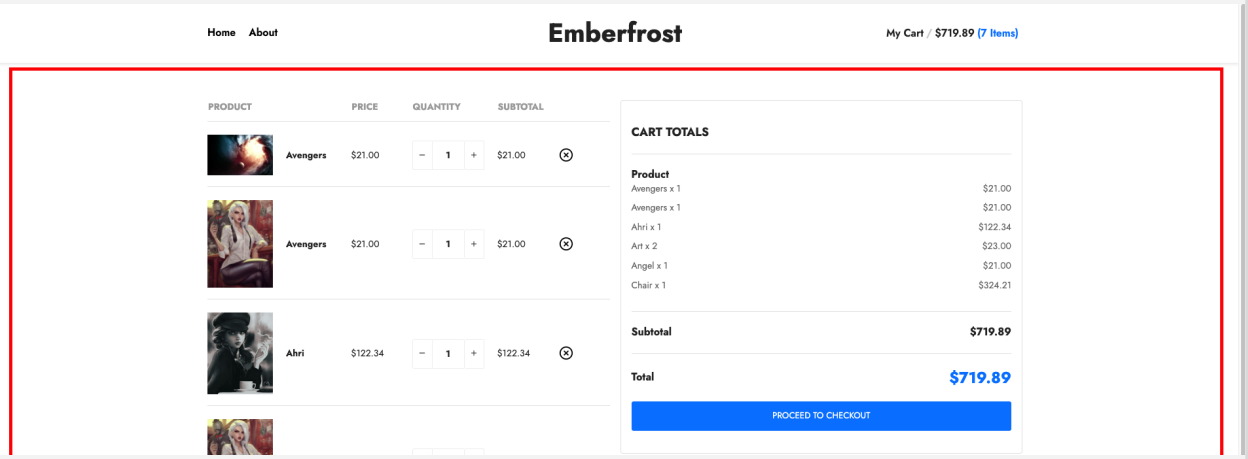


Evidence that all the tests are passing

- ☑ Verified
- ☐ Not Verified

6.4 Cart Page Component

Tester: Justin Singh
Recorder: Ilya Indik
Date: 04/29/2022



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.

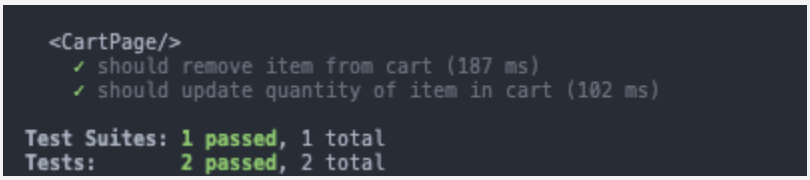
```
test('should remove item from cart', async () => {
    const { getByTestId } = render(<CartPage />);

    const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
      wrapper: AllTheProviders,
    });

    await waitForHook(() => result.current.itemsQuery.isSuccess);

    const removeIcon = getByTestId(TEST_ID_REMOVE_ITEM_ICON);

    act(() => {
      if (removeIcon) {
        fireEvent.click(removeIcon);
      }
    });

    await waitFor(() => expect(removeIcon).not.toBeInTheDocument());
  });
```

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.

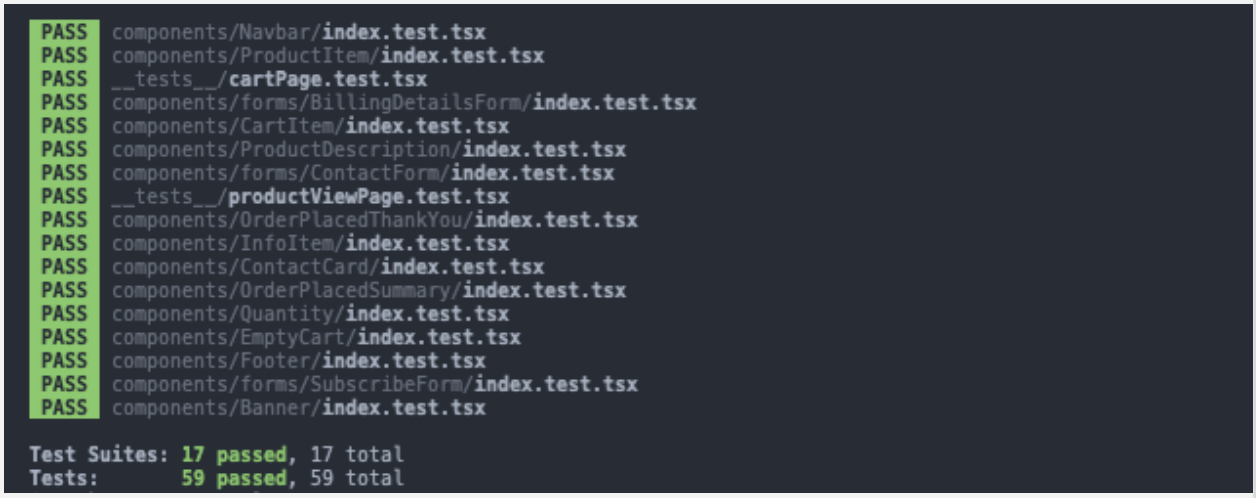
```
  test('should update quantity of item in cart', async () => {
    const { getByTestId } = render(<CartPage />);

    const { result, waitFor: waitForHook } = renderHook(() => useShoppingCartServerState(), {
      wrapper: AllTheProviders,
    });

    await waitForHook(() => result.current.itemsQuery.isSuccess);

    const quantityInput = getByTestId(TEST_ID_QUANTITY_NUMBER_FIELD) as HTMLInputElement;

    const previousValue = parseInt(quantityInput.value, 10);

    const addButton = getByTestId(TEST_ID_QUANTITY_ADDITION_BUTTON);

    act(() => {
      fireEvent.click(addButton);
    });

    await waitFor(() => expect(parseInt(quantityInput.value, 10)).toBe(previousValue + 1));
  });
```

6.4.3 Test results



Evidence that all the tests are passing

6.5.0 All frontend unit and integration tests



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 package**s.**

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.

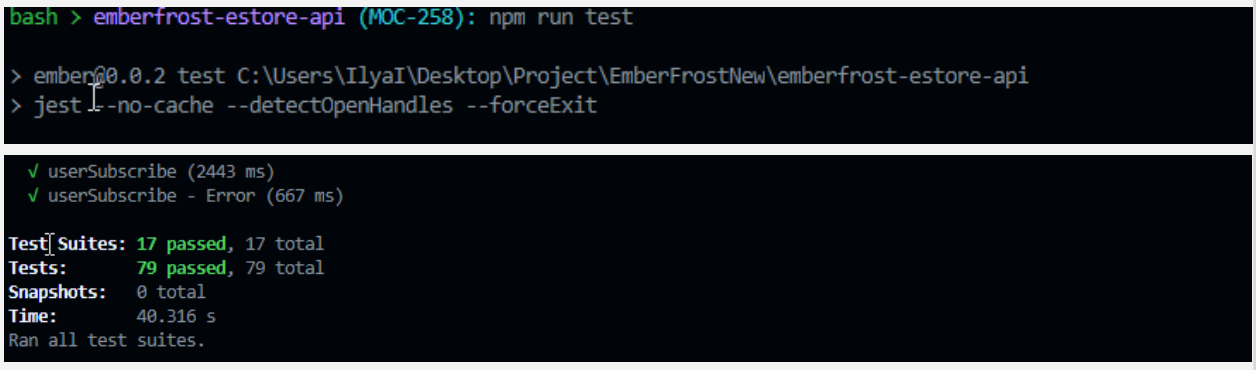


Figure 7.0.2.1

```
bash > emberfrost-estore-api (MOC-258): npm run test ./src/ErrorHandling/ErrorHandlingTests/ApiError.test.ts

> ember@0.0.2 test C:\Users\IlyaI\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

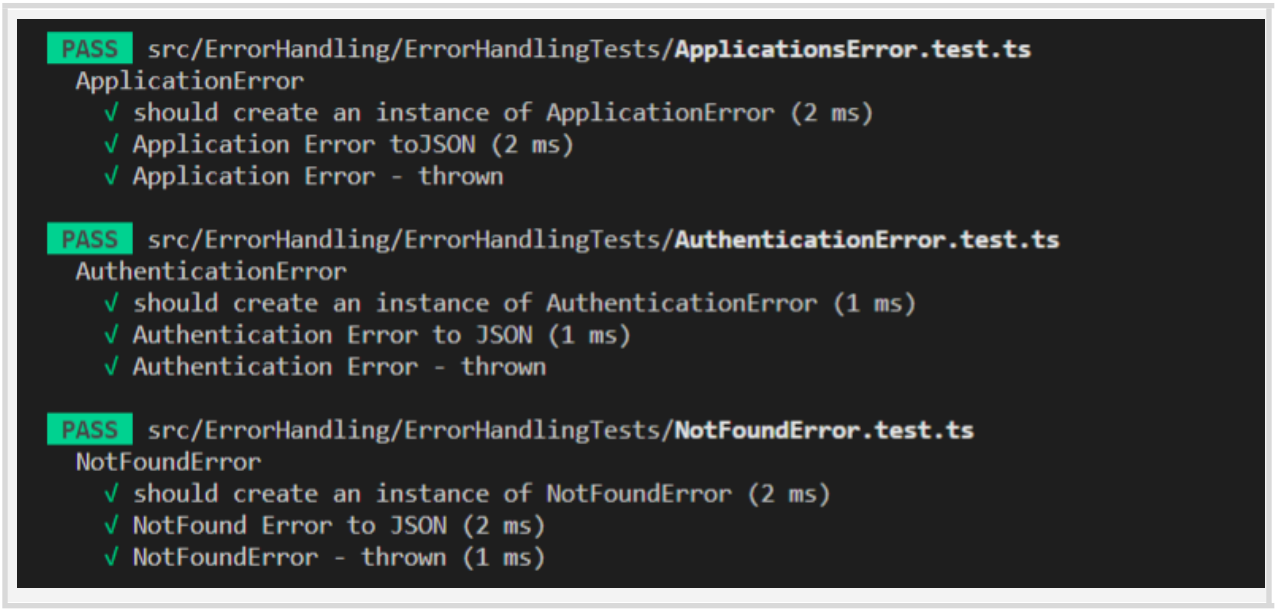
This suite of tests - tests the entry point functions for all of the implemented routes. Generally each method is tested 2 times. 1 time - expected to successfully return data and a 200 status. The second time it is expected to throw an error.

```
 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.

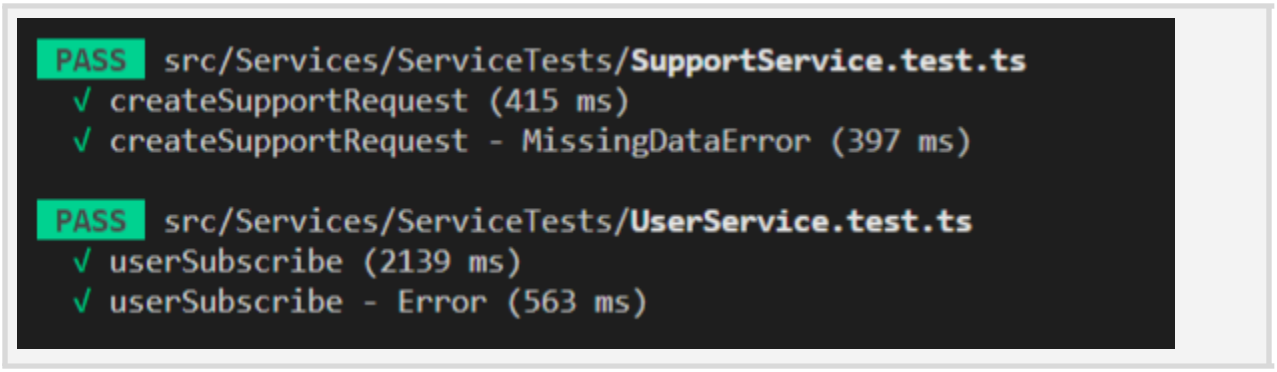
7.1.2 Error Handling Suite

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.



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.



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

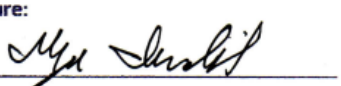
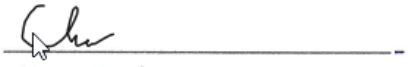
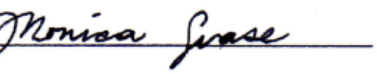
Vlad Klimkiv

CEO

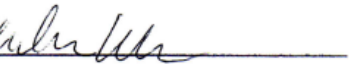
EmberFrost Corp.

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