Using React Fast Refresh (dev mode only)

Fast Refresh allows near-instant feedback for changes in React components. This is meant for developers, so while it won't directly affect end users, this will improve our performance when developing the website. This will enable us to get to production faster and improve it in the future.

src > foodCard > FoodCardConstants.tsx > defaultFood > pickupTime

```tsx
export const defaultFood = {
  foodName: "Assorted Pastries",
  restaurantName: "Bella Bakery",
  imageUrl: "/api/placeholder/300/200",
  distance: 0.7,
  pickupTime: "Today, 5:00 PM - 6:30 PM",
  tags: ["Vegetarian", "Bakery"],
  active: true,
  isFavorite: true
};

export const foodCardButtonStyle = {
    backgroundColor: '#16a34a', // Tailwind green-600
  color: 'white',
  fontWeight: 'bold',
  padding: '0.5rem 1rem',
  borderRadius: '0.5rem',
  width: '100%',
  transition: 'background-color 200ms',
};

export const handleMouseEnter = (e) => e.target.style.backgroundColor = '#15803d';

export const handleMouseLeave = (e) => e.target.style.backgroundColor = '#16a34a';
```

React Fast Refresh requires there to be no separate constants in the same file as a React component. So we made a separate file with the constants used for the food cards.

Code splitting with React Router

Code splitting lets us load only the necessary code for a given part of the application, thereby improving performance.

```tsx
// React Router code splitting
const OrgView = lazy(() => import('./profile/OrgProfile.tsx'));
const UserView = lazy(() => import('./profile/UserProfile.tsx'));
```

```tsx
            />
            <Route path="/profile" Component={isOrg ? OrgView : UserView} />
          </Routes>
```

In this example we lazily load the views for the user and organization profiles separately. Then when the user goes to the profile path, we can use the lazily loaded view. Since the vast majority of users will only be interacting with the app as a user (ordering food) or organization (giving away food), this is much more efficient than loading both views in advance

No unnecessary empty `div`

React requires each function to only return one component. This can force us to nest a bunch of components under an empty div, but this reduces performance. While we do use top layer divs in our code, they only serve a styling purpose, so that the reduced performance is not without improvement.

```
return (
  <header className="app-banner">
    <div className="banner-content">
      {isOrg && <button onClick={foodListingContext.toggleOpen}>
        New <br></br>Listing
      </button>}
      {!isOrg && <div></div>}

      <div className="logo-container justify-center">
        {logoSrc && <img src={logoSrc} alt="Logo" className="app-logo" />}
        <div className="title-container">
          <h1 className="app-title">{name}</h1>
          <p className="app-description">{desc}</p>
        </div>
      </div>

      <div className="profile-container">
        {profileSrc && (
          <img
            src={profileSrc}
            alt="Profile"
            className="profile-image"
            onClick={handleProfileClick}
          />
        )}
      </div>
    </div>
  </header>
);
```

Our top level React components all have a className or some other form of styling.

React hooks with dependency arrays to prevent unnecessary rerendering.

The useEffect hook is only called when one of the attributes in the dependency array changes.

```
// update food cards whenever selectedTag changes
useEffect(() => {
  org.listings = sortCards();
}, [selectedTag, org.listings]);
```

In this example we only re-sort the cards when the tag changes, as a result of filtering, or the food items change (because an item was added or removed).

No spreading props on DOM elements:

Spreading properties into a DOM element adds an unknown HTML attribute, so we pass in the specific attribute that is needed from the prop into the DOM element.

```
const OrgFoodCard = ({ food }: {food: foodItemType}) => {
  // Allows us to have default values for the card
  const foodData = { ...defaultFood, ...food };

  const {
    foodName,
    restaurantName,
    imageUrl,
    distance,
    pickupTime,
    tags,
    active,
  } = foodData;

  return (
    // Used many Tailwind CSS classes to have the styling applied
    <div className="max-w-sm rounded-lg overflow-hidden shadow-lg bg-white hover:shadow-xl transition-shadow duration-300">
      <div className="relative">
        {FoodCardImage({ imageUrl, foodName, active })}
      </div>

      <div className="p-4">

        {FoodCardText({ foodName, restaurantName, distance, pickupTime, isUser: false })}

        {FoodCardTags({ tags })}
```

We split the prop beforehand and then pass them into the components, rather than passing in the whole prop with the spread operator

Make images load lazily

Lazy loading an image means that an image is only loaded when it is needed, rather than when the page is loaded.

```
export const FoodCardImage = ({ imageUrl, foodName, active }: { imageUrl: string; foodName: string; active: boolean }) => (
  <div>
    <img
      loading="lazy"
      src={"/src/assets/" + imageUrl}
      alt={foodName}
      className="w-full h-48 object-cover"
    />
    {active ?
    (<div className="absolute top-0 right-0 bg-green-500 text-white px-2 py-1 m-2 rounded-md text-sm font-bold">Active</div>)
    : (<div className="absolute top-0 right-0 bg-red-500 text-white px-2 py-1 m-2 rounded-md text-sm font-bold">Closed</div>)}
  </div>
);
```

We lazy load the food card images. If there are hundreds of listings, it would be very computationally expensive to load them all during the initial page load. We can improve performance by only loading them when the use goes to the page with the food item.