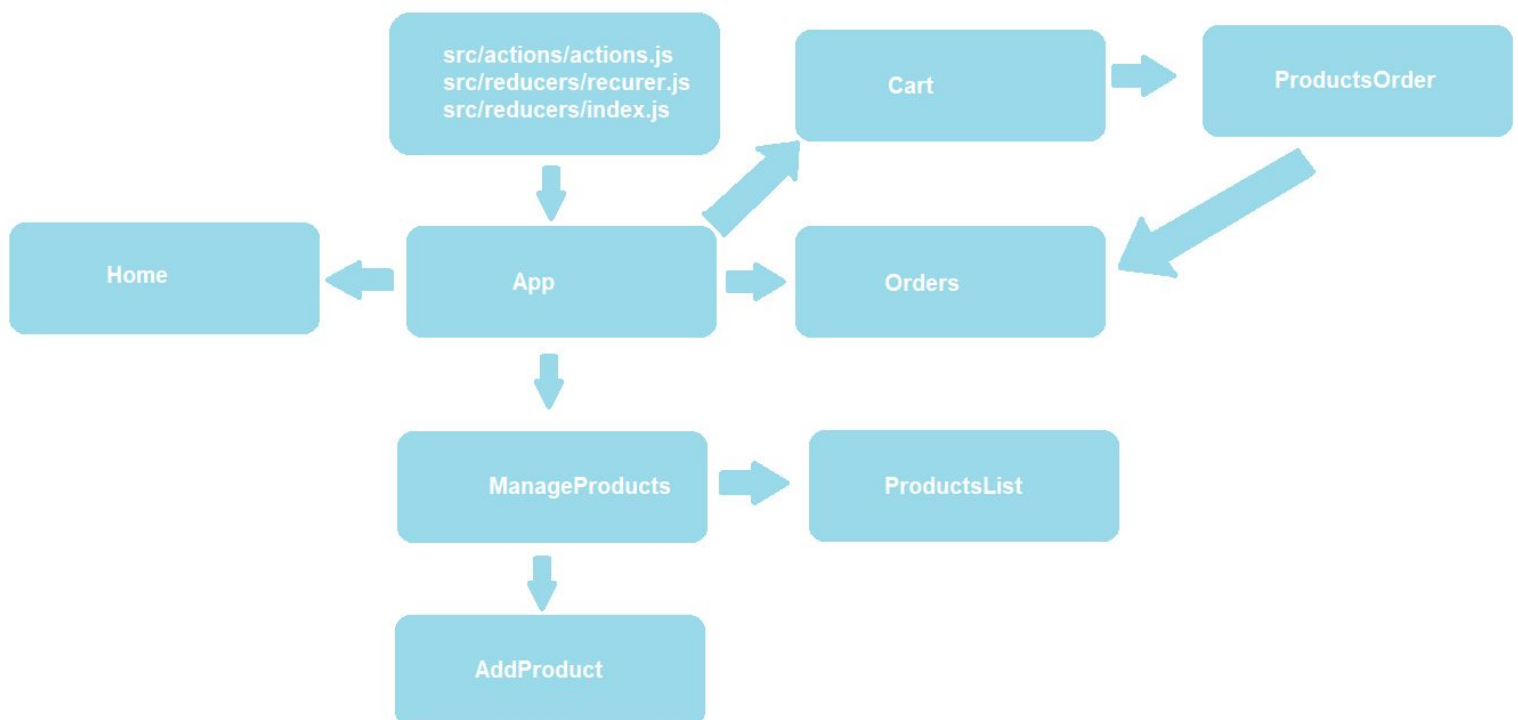


Wydział Informatyki Politechniki Białostockiej Aplikacje internetowe oparte o komponenty	Data oddania: 08.12.2019 r.
Dokumentacja projektu Temat: Sklep internetowy Skład grupy: <ol style="list-style-type: none"> 1. Jasiński Piotr 2. Zaorski Michał 	Prowadzący: dr inż. Urszula Kuźelewska

Dokumentacja

1. Wizualizacja architektury



2. Struktura danych

W aplikacji występują dwie struktury danych:

- a) **products** - jest to tablica obiektów będących produktami w sklepie, które mają następujące pola:
- **id** - number
 - **nazwa** - string
 - **producent** - string
 - **cena** - number
 - **img** - string
 - **ilosc** - number
- b) **orders** - jest to tablica obiektów do przechowywania złożonych zamówień, które posiadają następujące pola:
- **imie** - string
 - **nazwisko** - string
 - **adres** - string
 - **suma** - number
 - **orderedProducts** - Array<Object> - tablica obiektów, będąca produktami z zamówienia. Każdy obiekt z tej tablicy ma takie same pola jak obiekty z tablicy **products**

3. Funkcjonalności

- 3.1. Dodawanie nowych produktów do głównej bazy produktów
- 3.2. Edytowanie produktów w głównej bazie produktów
- 3.3. Usuwanie produktów z głównej bazy produktów
- 3.4. Wyświetlanie listy produktów z głównej bazy produktów
- 3.5. Dodawanie produktów do koszyka z listy produktów
- 3.6. Usuwanie produktów z koszyka
- 3.7. Wyświetlanie listy produktów w koszyku
- 3.8. Edytowanie ilości produktów tego samego rodzaju w koszyku
- 3.9. Dodawanie zamówienia złożonego z przedmiotów w koszyku
- 3.10. Wyświetlanie listy zamówień

4. Elementy frameworka

4.1. Weryfikacja typu danych przekazywanych do komponentów

Wykorzystanie *propTypes* do walidacji typów danych.

Użycie:

- Komponent *ProductList*

```
ProductsList.propTypes = {  
  updateProduct: PropTypes.func,  
  deleteProduct: PropTypes.func,  
  products: PropTypes.arrayOf(PropTypes.object),  
  edit: PropTypes.bool,  
  idProdToEdit: PropTypes.number  
};
```

Sprawdzanych jest 5 typów w tym jeden złożony - *products*, gdzie przekazywana jest tablica obiektów.

- Komponent *AddProducts*

```
AddProduct.propTypes = {  
  fields: PropTypes.objectOf(PropTypes.array),  
  addProduct: PropTypes.func,  
  productsLength: PropTypes.number  
};
```

Sprawdzone są 3 typy przekazywanych props.

4.2. Komponenty

4.2.1. Klasowe

- App
- AddProduct
- ManageProducts
- ProductsList
- ProductsOrder

4.2.2. Funkcyjne

- Cart
- Home
- Orders

4.2.3. Kod wyświetlający menu w komponencie *App* przechowywany jest w pliku *Menu.js*

4.3. Komponenty typu Hook

Komponenty typu hook (komponent hook efektów) wykorzystywane są w komponentach funkcyjnych (bez stanowych): *Cart*, *Home*, *Orders*. Używane są do pobrania danych z serwera.

4.3.1. Home

```
function Home(props) {  
  useEffect(() => {  
    props.showAll();  
    // eslint-disable-next-line react-hooks/exhaustive-deps  
  }, []);  
}
```

4.3.2. Cart

```
function Cart(props) {  
  useEffect(() => {  
    props.showCart();  
    props.showSum();  
    // eslint-disable-next-line react-hooks/exhaustive-deps  
  }, []);  
}
```

4.3.3. Orders

```
function Orders(props) {  
  useEffect(() => {  
    props.showOrders();  
    // eslint-disable-next-line react-hooks/exhaustive-deps  
  }, []);  
}
```

4.4. Dwukierunkowa komunikacja pomiędzy komponentami

Komunikacja dwukierunkowa jest wykorzystywana w komponencie *ManageProducts* aby przesłać do komponentu *AddProduct* funkcję *addProduct* oraz do komponentu *ProductsList* funkcje *editProduct* i *removeProduct*.

4.4.1. Dodawanie produktu

Funkcja *add* w komponencie *ManageProducts*.

```
add(product) {  
  this.props.addProduct(product);  
}
```

Przesłanie funkcji przez *props* do komponentu *AddProduct*.

```
<AddProduct  
  fields={{  
    id: ["number", lastId],  
    nazwa: ["text"],  
    producent: ["text"],  
    cena: ["number"],  
    img: ["text"]  
  }}  
  addProduct={this.add.bind(this)}  
  productsLength={lastId}  
>
```

Funkcja *addProduct* w komponencie *AddProduct*. Wywołuje ona funkcję *addProduct* przesłaną w *props*.

```
addProduct() {  
  if (this.state.nazwa === null) return;  
  
  const product = {  
    id: this.props.productsLength,  
    nazwa: this.state.nazwa,  
    producent: this.state.producent,  
    cena: Math.round(this.state.cena * 100) / 100,  
    img: this.state.img,  
    ilosc: 0  
  };  
  this.props.addProduct(product);  
}
```

4.4.2. Usuwanie i edycja produktu

Przesłanie w props funkcji *editProduct* i *removeProduct*.

```
{products.loaded && (  
  <ProductsList  
    updateProduct={editProduct}  
    deleteProduct={removeProduct}  
    products={products.productsList}  
  />  
)}
```

Funkcja *finishEdit* z komponentu *ProductsList* wywołująca funkcję *editProduct* przesłaną w *props* jako *updateProduct*

```
finishEdit(e) {  
  this.setState({  
    edit: false,  
    idProdToEdit: -1  
  });  
  this.props.updateProduct(this.state.editedProduct, this.state.idProdToEdit);  
}
```

Funkcja *finishDelete* z komponentu *ProductsList* wywołująca funkcję *removeProduct* przesłaną w *props* jako *deleteProduct*.

```
finishDelete(e) {  
  this.setState({  
    edit: false,  
    idProdToEdit: -1  
  });  
  this.props.deleteProduct(this.state.editedProduct, this.state.idProdToEdit);  
}
```

4.5. Walidacja danych formularza

4.5.1. Przy dodawaniu produktu w komponencie *AddProduct*

Dodano blokadę przycisku w przypadku, gdy któreś z inputów jest pusty lub długość wpisanych danych jest zbyt długa. W celu przyjaźniejszego pokazywania błędów walidacji przy wpisywaniu danych przez użytkownika, wykorzystano bibliotekę *materializecss*.

```
<button
  className='waves-effect waves-light btn'
  disabled={
    !this.state.nazwa ||
    !this.state.producent ||
    !this.state.cena ||
    !this.state.img ||
    this.state.nazwa.length > 100 ||
    this.state.producent.length > 100 ||
    this.state.img.length > 1024
  }
  onClick={this.addProduct.bind(this)}
>
  Dodaj
</button>
```

4.5.2. Przy wypełnianiu danych do zamówienia

Walidację rozwiązano w identyczny sposób jak w przypadku 4.6.1

```
<Link to='/orders'>
  <button
    disabled={
      !this.state.imie ||
      !this.state.nazwisko ||
      !this.state.adres ||
      !this.state.numerKarty ||
      this.state.imie.length > 100 ||
      this.state.nazwisko.length > 100 ||
      this.state.adres.length > 512
    }
    onClick={this.addOrder.bind(this)}
  >
    Dodaj
  </button>
</Link>
```

4.6. Operacje modyfikacji danych (komunikacja z API)

Do przechowywania danych wykorzystano usługę serwera Node.js i modułu Express. Komunikujemy się z API za pomocą modułu Axios.

4.6.1. Typy żądań

- **GET**

- **Pobieranie listy produktów**

```
url: "http://localhost:8080/products"
```

Nie przekazujemy parametrów więc nie ma czego walidować.

- **Pobieranie listy zamówień**

```
url: "http://localhost:8080/orders"
```

Nie przekazujemy parametrów więc nie ma czego walidować.

- **Pobieranie produktów w koszyku**

```
url: "http://localhost:8080/cartProducts"
```

Nie przekazujemy parametrów więc nie ma czego walidować.

- **Pobieranie sumy produktów w koszyku**

```
url: "http://localhost:8080/suma"
```

Nie przekazujemy parametrów więc nie ma czego walidować.

- **POST**

- **Dodawanie produktu do bazy**

```
url: "http://localhost:8080/products"
```

Przekazujemy produkt do dodania, walidujemy czy nie jest duplikatem - duplikat -> kod błędu.

- **Dodawanie produktu do koszyka**

```
url: "http://localhost:8080/cartProducts"
```

Przekazujemy produkt do dodania, walidujemy czy nie jest duplikatem - duplikat -> zwiększamy ilość tego produktu w koszyku o 1.

- **Dodawanie sumy produktów**

```
url: "http://localhost:8080/suma"
```

Przekazujemy cenę produktu, walidujemy czy jest typu float - duplikat -> kod błędu i nie dodajemy.

- **Dodawanie nowego zamówienia do listy zamówień**

```
url: "http://localhost:8080/orders"
```

Przekazujemy zamówienie do dodania. Nic nie walidujemy.

- **PUT**

- **Edycja produktów w bazie danych**

```
url: `http://localhost:8080/products/${updatedProduct.id}`
```

Przekazujemy id produktu i nowe dane produktu do zaktualizowania. Sprawdzamy czy produkt jest w bazie.

- **Edycja ilości produktu tego samego rodzaju w koszyku**

```
url: `http://localhost:8080/products/${updatedProduct.id}`
```

Przekazujemy id produktu i oraz informację czy dodać do ilości czy odjąć. Sprawdzamy czy produkt jest w koszyku.

- **DELETE**

- **Usuwanie produktu z bazy**

```
url: `http://localhost:8080/products/${removedProduct.id}`
```

Przekazujemy id produktu do usunięcia, sprawdzamy czy został usunięty.

- **Usuwanie produktu z koszyka**

```
url: `http://localhost:8080/cart/${id}`
```

Przekazujemy id produktu do usunięcia, sprawdzamy czy został usunięty.

4.7. Routing

Routing rozpoczyna się od komponentu App, gdzie wywoływany jest komponent Menu. Tam ustalane są elementy menu (komponent Link). Następnie w komponencie App używany jest komponent Switch, który sprawia, iż zostanie wywołany wyłącznie Route, który odpowiada wybranej ścieżce (wyklucza to możliwość wywołania wielu komponentów które pasują wybranej ścieżce).

Komponent App ze zdefiniowanym routingiem do innych komponentów.

```
<div className='App'>
  <BrowserRouter>
    <Menu />

    <Switch>
      <Route exact path='/' component={Home} />
      <Route
        path='/cart'
        render={() => <Cart products={products.addedItems} />}
      />
      <Route path='/manage' component={ManageProducts} />
      <Route
        path='/order'
        render={() => (
          <ProductsOrder
            fields={[
              imie: ["text"],
              nazwisko: ["text"],
              adres: ["text"]
            ]}
          />
        )}
      />
      <Route path='/orders' component={Orders} />
    </Switch>
  </BrowserRouter>
</div>
```

Komponent funkcyjny Menu, który ustala elementy menu.

```
const Menu = props => {
  return (
    <nav className='nav-wrapper green'>
      <div className='container' style={{ marginRight: 10 }}>
        <Link to='/' className='brand-logo center'>
          Spożywczak
        </Link>
        <ul className='right'>
          <li key='Management'>
            <Link to='/manage'>Management</Link>
          </li>
          <li key='Orders'>
            <Link to='/orders'>Zamówienia</Link>
          </li>
          <li key='Cart'>
            <Link to='/cart'>
              <i className='material-icons'>shopping_basket</i>
            </Link>
          </li>
        </ul>
      </div>
    </nav>
  );
};
```

4.8. Architektura Flux

Architektura Flux jest wykorzystywana do ustalenia jednokierunkowej komunikacji pomiędzy komponentami. Przykładowe wykorzystanie tej architektury występuje w komponencie funkcyjnym Home. Zostaje tam zmapowany stan i metody do props.

W pliku actions.js są zapisane akcje, które następnie są eksportowane z konkretnym typem i ładunkiem. Wykonywaną akcją przykładowo jest wysłanie do serwera żądania GET, aby pobrać wszystkie produkty z bazy.

W pliku reducers.js jest zapisana funkcja reducer przyjmująca poprzedni stan oraz akcję, która sprawdza typ przychodzących akcji i w zdefiniowany sposób zmienia stan aplikacji. Jeżeli typem jest SHOW_ALL wówczas zmieniany jest stan na nowy z listą produktów pobraną z serwera.

Mapowanie stanu oraz metod do props

```
const mapStateToProps = state => {  
  return {  
    products: state.products.productsList,  
    addedItems: state.products.addedItems,  
    suma: state.products.suma  
  };  
};
```

```
const mapDispatchToProps = dispatch => {  
  return {  
    showAll: () => {  
      dispatch(showAll());  
    },  
    showCart: () => {  
      dispatch(showCart());  
    },  
    showSum: () => {  
      dispatch(showSum());  
    },  
    addToCart: id => {  
      dispatch(addToCart(id));  
    },  
    addSuma: suma => {  
      dispatch(addSuma(suma));  
    }  
  };  
};
```

Akcja zdefiniowana w pliku actions.js do pobrania danych z serwera.

```
export const SHOW_ALL = "SHOW_ALL";  
  
export const showAll = () => dispatch => {  
  axios({ url: "http://localhost:8080/products" })  
    .then(res => {  
      dispatch(showAllAction(res.data));  
    })  
    .catch(error => {  
      throw error;  
    });  
};  
  
export const showAllAction = data => ({  
  type: SHOW_ALL,  
  products: data  
});
```

Funkcja reducera w pliku reducers.js przyjmująca poprzedni stan oraz akcję i definiująca zmianę stanu przy akcji SHOW_ALL

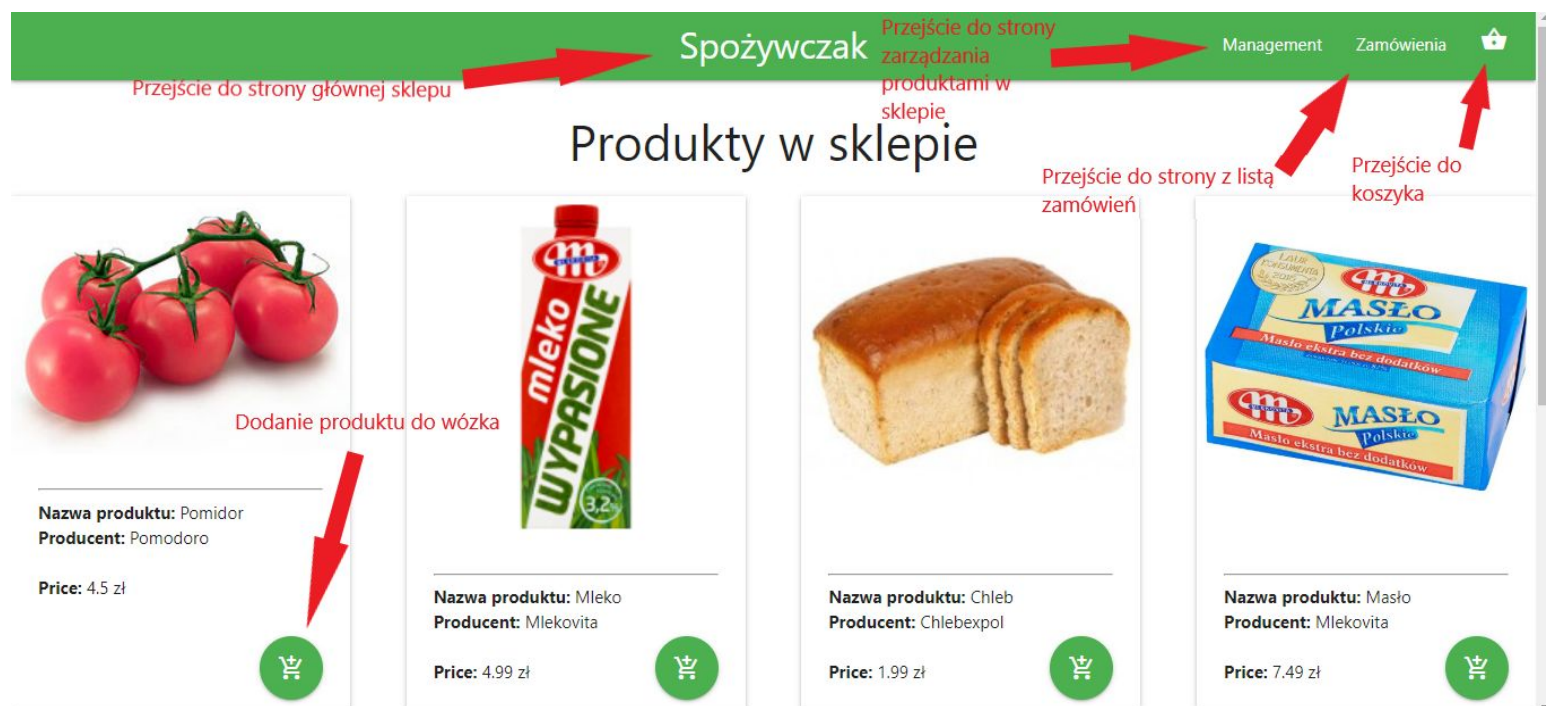
```
✓ export default function products(
  ✓ state = {
    productsList: [],
    addedItems: [],
    orders: [],
    loaded: false,
    suma: 0
  },
  action
) {
  let new_state;
  ✓ switch (action.type) {
    case "SHOW_ALL":
      new_state = Object.assign({}, state);
      new_state.productsList = action.products;
      new_state.loaded = true;
      return new_state;
```

5. Instrukcja użytkownika

Pierwszą stroną widoczną po odpaleniu aplikacji jest **strona główna**, na której znajdują się wszystkie produkty w sklepie.

Z tego miejsca możliwe jest **dodanie produktu do koszyka** poprzez kliknięcie przycisku widocznego na karcie produktu.

Górne menu umożliwia na każdej ze stron przejście na **stronę główną**, **stronę zarządzania produktami**, **stronę z listą zamówień** oraz **stronę koszyka**.




Po naciśnięciu **przycisku koszyka** w menu nastąpi przejście do koszyka. Wówczas ukaże się strona z listą produktów wcześniej wybranych. Każdy produkt ma swoją ilość w koszyku, którą można zwiększyć lub zmniejszyć klikając odpowiednio na przyciski **+1** lub **-1**.

Pod listą wybranych produktów do koszyka widnieje **suma pieniędzy do zapłaty** za dane produkty.

Natomiast pod nią widoczny jest przycisk, po którego kliknięciu nastąpi przeniesienie do **formularza składania zamówienia**.



Koszyk



Nazwa produktu: Mleko
Producent: Mlekovita
Cena: 4.99 zł
Ilość: 2
-1 +1

Zwiększenie lub zmniejszenie ilości produktu w koszyku

Usunięcie produktu z koszyka

Suma do zapłaty: 9.98 zł

ZAMÓW

Przejdźcie do formularza składania zamówienia na produkty w koszyku

Na stronie z formularzem do **składania zamówienia** widoczne są 3 pola: **imię**, **nazwisko** i **adres**, które należy uzupełnić aby zamówienie zostało utworzone. Dane te posłużą do realizacji zamówienia.



imię

nazwisko

adres

Dodaj

Pola do uzupełnienia danymi (imię, nazwisko, adres) osoby składającej zamówienie

Zatwierdzenie formularza i utworzenie zamówienia z podanymi danymi

Po poprawnym wypełnieniu pól i wciśnięciu przycisku pod formularzem nastąpi przeniesienie do **listy złożonych zamówień**. Tam można dostrzec **dane do zamówienia** oraz **listę produktów w zamówieniu**.

Spożywczak

Management Zamówienia

Zamówienia

Imie: Michał
Nazwisko: Zaorski
Adres: Politechnika Białostocka
Suma: 9.98 zł



Mleko

Nazwa produktu: Mleko
Producent: Mlekovita
Cena: 4.99 zł
Ilość: 2

Po naciśnięciu **przycisku Management** nastąpi przeniesienie do **strony do zarządzania produktami**. Na stronie tej pierwszą widoczną rzeczą jest **formularz do dodawania nowych produktów** do sklepu z następującymi polami: **id** (uzupełniane automatycznie), **nazwa produktu**, **producent**, **cena za produkt** oraz **img**, czyli adres url obrazu produktu. Po poprawnym uzupełnieniu pól należy kliknąć **przycisk Dodaj** aby dodać nowy produkt do sklepu.

Spożywczak

Management Zamówienia

id	6	Id produktu uzupełniane automatycznie	Formularz do dodawania nowych produktów do sklepu
nazwa		Pola do uzupełnienia danymi (nazwa, producent, cena) produktu który ma być dodany do sklepu	
producent			
cena			
img		Pole na wstawienie adresu URL obrazu produktu, który ma być dodany	
DODAJ		Zaakceptowanie formularza i dodanie nowego produktu do sklepu	

Pod formularzem widoczna jest **lista produktów** w sklepie. Po naciśnięciu na produkt zostaje pokazany **formularz do edycji danych** produktu. Możliwa jest edycja **nazwy produktu, producenta, ceny** oraz **adresu url** obrazu produktu. Natomiast nie jest możliwe edytowanie **id produktu** oraz jego **ilości w koszyku**.

Aktualne **dane produktu** widoczne są nad formularzem. Po zmianie któregoś z możliwych do edycji pól należy kliknąć **przycisk Update** aby zaakceptować zmiany danych. Możliwe jest dodatkowo wciśnięcie **przycisku Remove**, którego kliknięcie spowoduje usunięcie produktu ze sklepu.

Id 0: Pomidor Pomodoro, cena: 4.5

0

Pomidor

Pomodoro

4.5

https://upload.wikimedia.org/wikipedia/commons/thumb/6/66/Pomidory_-_tomato.jpg/250px-Pomidory_-_tomato.jpg

0

UPDATE REMOVE

Id 1: Mleko Mlekovita, cena: 4.99

Id 2: Chleb Chlebexpol, cena: 1.99

Id 3: Mleko Mlekovita, cena: 7.40

Aktualne dane klikniętego produktu

Pola do zmiany danych
(nazwa, producent, cena,
adres url obrazu) produktu

Przyciski do zaakceptowania zmian danych produktu
lub usunięcia produktu z bazy

Lista produktów w sklepie