# Trek Trill – Particular Code Documentation

- Browse Documentations

```
export default function BrowseAccommodations(props: { posts: any[] }) {
  const [searchQuery, setSearchQuery] = useState("");
  const [debouncedQuery] = useDebounce(searchQuery, 500);
  const [filteredPosts, setFilteredPosts] = useState([] as any[]);
  const router = useRouter();

  const handleChange = (e: any) => setSearchQuery(e.target.value.toLowerCase());

  const handleResultClick = (postId: string) => {
    router.push(`/display_accommodation/${postId}`);
  };

  useEffect(() => {
    setFilteredPosts(
      props.posts.filter((post) => {
        return (
          post.location.toLowerCase().includes(debouncedQuery) ||
          post.title.toLowerCase().includes(debouncedQuery)
        );
      })
    );
  }, [debouncedQuery, props.posts]);

  return (
    <div className="mb-10">
      <label className="text-lg mr-2" htmlFor="search">
        Browse by location or name:
      </label>
      <input
        type="text"
        name="search"
        placeholder="e.g. Iasi"
        id="search"
        value={searchQuery}
```

```
        onChange={handleChange}
        className="p-2 border rounded-md"
      />
      <ul
        className={`absolute border rounded-md ml-2 w-[300px] [&>li]:p-2
[&>li]:rounded-md [&>li]:bg-white
       ${debouncedQuery !== "" ? "inline-block" : "hidden"}`}
      >
        {filteredPosts.length === 0 && <li className="">No results.</li>}
        {filteredPosts.map((post) => {
          return (
            <li
              key={post.id}
              className="cursor-pointer flex justify-between hover:bg-gray-100"
              onClick={() => handleResultClick(post.id)}
            >
              <span>{post.title}</span>
              <span className="text-sm text-gray-500">
                <SlLocationPin className="inline-block" /> {post.location}
              </span>
            </li>
          );
        })}
      </ul>
    </div>
  );
}
```

The **`BrowseAccommodations`** component allows users to browse a list of accommodations based on a search query. The component takes an array of accommodation posts as a prop, each with **`id`**, **`title`**, and **`location`** fields. It manages three state variables: **`searchQuery`** for the current input, **`debouncedQuery`** for the debounced version of **`searchQuery`**, and **`filteredPosts`** for the filtered list of posts.

The **`handleChange`** function updates **`searchQuery`** with user input, converting it to lowercase. The **`handleResultClick`** function navigates to the selected accommodation's details page. The **`useEffect`** hook filters posts based on **`debouncedQuery`** and updates the filtered list whenever the search query or posts change.

The JSX structure includes an input field for search queries and a dynamically rendered list of filtered posts. The list shows "No results" if no posts match the query. Each post displays its title and location and navigates to its details page when clicked. The component is styled with Tailwind CSS for padding, borders, and rounded corners, and the results list is conditionally displayed based on the presence of **debouncedQuery**. This component offers a user-friendly interface for searching and browsing accommodations, with optimized performance and seamless navigation.

- Account and login (validation included)

```
onst handleSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();

    //validate username
    const regex = /^[0-0A-Za-z_]+$/;
    if (username.length < 3 || !regex.test(username)) {
      toast.error('Invalid username (minimum 3 characters, alphanumeric only)')
      return;
    }

    const userDoc = doc(db, 'users', user.uid);
    const usernameDoc = doc(db, 'usernames', username);

    // check if username exists
    const usernameSnapshot = await getDoc(usernameDoc);
    if (usernameSnapshot.exists()) {
      toast.error('Username already exists');
      return;
    }

    // create user and username documents
    const batch = writeBatch(db);
    batch.set(userDoc, {username: username, usertype: accountType});
    batch.set(usernameDoc, {uid: user.uid});

    await batch.commit();
    toast.success('Account created successfully');
  }
```

```
  return (
    <form className="px-8 py-4 mx-auto mt-10 bg-gray-200 shadow-md select-none
rounded-xl mb-4 w-96" onSubmit={handleSubmit}>
      <MdOutlineDriveFileRenameOutline className="mx-auto mb-6 w-28 h-28 text-
gray-800" />
      <p className="text-lg mb-2">Enter your username below:</p>
      <label className="flex items-center w-full p-2 mb-2 border-gray-700
rounded-lg bg-white">
        <span>
          <MdAccountCircle className="w-8 h-8 pr-2 text-gray-500 border-r" />
        </span>
        <input type="text" name="username" id="username" className="flex-1 px-3
text-lg bg-transparent focus:outline-none" placeholder="Your Username" required
value={username} onChange={(e) => setUsername(e.target.value)} />
      </label>
      <div className="flex items-center justify-between w-full p-2 mb-2 border-
gray-700 rounded-lg bg-white">
        <span>
          <FaPersonCircleQuestion  className="w-8 h-8 pr-2 text-gray-500 border-
r" />
        </span>
        <select name="accountType" id="accountType" className="flex-1 px-3 text-
lg bg-transparent focus:outline-none" value={accountType} onChange={(e) =>
setAccountType(e.target.value)}>
          <option value="guest">Guest</option>
          <option value="host">Host</option>
        </select>
      </div>
      <button className="w-full py-2 mt-4 text-xl font-bold text-white duration-
200 rounded-lg bg-gray-800 hover:bg-gray-600">Create account</button>
    </form>
  )
}
```

The **`handleSubmit`** function processes a form submission to create a user account. It validates the username, checks if it already exists, and if not, creates user and username documents in the database. The function prevents the default form submission behavior, ensuring custom validation and asynchronous operations can complete before any default actions.

First, the function validates the username using a regular expression and checks that it is at least 3 characters long. If the validation fails, an error message is shown and the function returns early. Next, it prepares references to the user and username documents in the database. It then checks if the username already exists by querying the username document. If the username exists, an error message is shown and the function returns.

If the username is valid and not already taken, the function creates a batch operation to write both the user and username documents to the database. After committing the batch, a success message is shown.

The JSX structure for the form includes input fields for the username and account type (guest or host), and a submit button. The form is styled with Tailwind CSS and includes icons from **`react-icons`** for a more user-friendly interface. The **`handleSubmit`** function is attached to the form's **`onSubmit`** event to handle the form submission.

This form component provides a streamlined and user-friendly way for users to create an account, with built-in validation and feedback to ensure a smooth registration process.

- Shopping Cart

```
const CheckoutPage: React.FC = () => {
  const [tab, setTab] = useState("checkout");
  const [totalPrice, setTotalPrice] = useState(0);
  const [selectedAccommodations, setSelectedAccommodations] = useState([] as
DocumentData[]);
  const {user} = useContext(UserContext)

  useEffect(() => {
    (async () => {
      if(!Object.keys(user) || !user.uid){
        return;
      }
      const userCartDoc = await getDoc(doc(db, 'cart', user.uid));
      if (!userCartDoc.exists()) {
        return;
      }

      const cartItems = userCartDoc.data().cart;
```

```
      let cartAccommodations = [];
      let cartPrice = 0;
      for(const item of cartItems) {
        const accommodationDoc = await getDoc(doc(db, 'accommodations',
item.accommodationId))
        if (!accommodationDoc.exists()) {
          continue;
        }
        cartAccommodations.push({...accommodationDoc.data(), checkIn:
item.checkIn, checkOut: item.checkOut, id: accommodationDoc.id})

        const reservationDuration = Math.floor((Date.parse(item.checkOut) -
Date.parse(item.checkIn)) / 86400000);
        cartPrice += accommodationDoc.data().price * reservationDuration
      }

      setSelectedAccommodations(cartAccommodations)
      setTotalPrice(computeCartTotal(cartAccommodations))
    })();
  }, [user])

  const computeCartTotal = (accommodations: any[]) => {
    let cartPrice = 0
    for(const acc of accommodations) {
      const reservationDuration = Math.floor((Date.parse(acc.checkOut) -
Date.parse(acc.checkIn)) / 86400000);
        cartPrice += acc.price * reservationDuration
    }
    return cartPrice;
  }

  const removeFromCart = async(accId:string) => {
    const accommodationDoc = doc(db, 'cart', user.uid)
    const newAccommodations = selectedAccommodations.filter(acc => {
      if (acc.id === accId) return;
      return {
        accommodationId: acc.id,
        checkIn: acc.checkIn,
        checkut: acc.checkOut
      }
    })
    updateDoc(accommodationDoc, {cart: newAccommodations}).then(() => {
      toast.success("Removed from cart!")
```

```
      setSelectedAccommodations(newAccommodations)
      setTotalPrice(computeCartTotal(newAccommodations))
    }).catch(err => {
      toast.error("Error removing from cart")
      console.error(err)
    })
  }

  return (
    <AuthCheck usertype="guest">
      {tab === "checkout" ?
        <CheckoutForm
          selectedAccommodations={selectedAccommodations}
          totalPrice={totalPrice}
          onPayment={() => setTab("payment")}
          onRemoveFromCart={(id) => removeFromCart(id)}
        />
        : tab === "payment" &&
        <Payment totalPrice={totalPrice} />
      }
    </AuthCheck>
  );
};
```

The **`CheckoutPage`** component manages the checkout process for a user, displaying selected accommodations and handling payment. It tracks the current tab (either "checkout" or "payment"), the total price of the selected accommodations, and the list of selected accommodations. The user context provides information about the currently logged-in user.

Upon mounting, the component fetches the user's cart data from the database. If the user is not logged in or the cart is empty, it exits early. Otherwise, it retrieves the accommodation details for each item in the cart, calculates the total price based on the reservation duration, and updates the state with the selected accommodations and the total price.

The **`computeCartTotal`** function calculates the total price of the accommodations based on their reservation durations. The **`removeFromCart`** function removes an accommodation from the cart in the database and updates the state accordingly. It filters out the specified accommodation and updates the database with the new list of accommodations. Upon

successful update, it shows a success message and updates the state; otherwise, it shows an error message.

The JSX structure renders different components based on the current tab. If the tab is "checkout", it renders the **`CheckoutForm`** component with the selected accommodations, total price, and functions to handle payment and remove items from the cart. If the tab is "payment", it renders the **`Payment`** component with the total price. The **`AuthCheck`** component ensures that only users with the "guest" user type can access this page.

This component provides a seamless and user-friendly checkout experience, handling the complexities of fetching and updating cart data, calculating prices, and transitioning between checkout and payment views.