Report on the CRUD Application

1. Overview

This project illustrates the implementation of a CRUD (Create, Read, Update, Delete) application on a decentralized, blockchain-based platform, specifically utilizing the Internet Computer (IC) ecosystem. The IC is a distributed computing environment that executes canister smart contracts, allowing developers to deploy WebAssembly binaries that expose public interfaces for stateful operations. This architecture enables data persistence and secure distributed processing without relying on traditional databases or servers.

This project is a full-stack CRUD application deployed on the Internet Computer (IC).

- Backend: Rust-based canister that stores and manages variables.
- Frontend: A React application that lets you create, view, update, and delete those variables.
- Key Idea: Treat variables as either immutable (`let`) or mutable (`let mut`), decided by the value's prefix.

---

2. Backend Architecture

Technologies

- **Rust** for backend logic.
- **ic_cdk** for canister integration.
- **candid** for data serialization.

Data Model

Each variable is an instance of the `Variable` struct:

- id (u64): Unique identifier.
- name (String): Variable name.
- value (String): Variable value (can be plain or prefixed with `"mut:"` to indicate mutable).

Storage

- Backend uses Thread-Local Storage with `RefCell<HashMap>` to hold variables.
- A separate `COUNTER` keeps track of the last used ID.

Backend Functions

1. **create_variable(name, value)** – Creates and stores a new variable.
2. **get_all_variables()** – Retrieves all variables as a list.
3. **update_variable(id, new_value)** – Updates the value if the variable exists.
4. **delete_variable(id)** – Deletes the variable if present.

Each function is tagged as `#[update]` or `#[query]` depending on whether it changes state (`update`) or just returns data (`query`).

---

3. Frontend Architecture

Technologies

- **React (JavaScript)** for building UI.

- Generated **JS bindings** (`crud_operations_backend`) to call canister methods.

Component State

- `variables`: An array of variables fetched from the backend.
- `name`, `value`: Input state for creating new variables.

Actions

- **Create Variable:**
  - Sends new variable to the canister if inputs are not empty.
- **Load Variables:**
  - Fetch all variables on page load and after each action.
- **Update Variable:**
  - Prompts for new value and updates the canister.
- **Delete Variable:**
  - Deletes the variable by ID.

---

4. Categorization of Variables

The React code splits the variables into two lists:

- **Immutable list:** Variables where value does not begin with `"mut:"`.
- **Mutable list:** Variables where value begins with `"mut:"`.
  This provides a clear visual distinction between two kinds of variables.

---

5. UI & UX

The interface is split into two columns:

- Left column: Immutable variables.
- Right column: Mutable variables.

Each variable is displayed in a list:

- Shows the variable's name and value.
- Includes Edit and Delete buttons for user interaction.

Styling is done inline for quick setup and basic visual appeal:

- Colored sections (`blue` for immutable, `green` for mutable).
- Rounded boxes and shadows for a clean, modern look.

Mutable vs Immutable:

- Borrowing this from Rust concepts (`let` and `let mut`), the app visually reinforces ownership and mutation principles.

---

6. Conclusion

This application showcases the fundamentals of building decentralized applications (DApps) on the Internet Computer. It emphasizes:

- Developed a complete end-to-end application on a local Internet Computer setup utilizing DFX and canister-based deployment.
- Clearly distinguished between immutable and mutable variables in the UI to reflect Rust's strict mutability model.

- Established seamless communication between the Rust backend canister and the interactive React interface for dynamic data handling.