Aptos CLI Setup

Part 1: Automated Installation for Windows/Linux

Prerequisites:

- Ensure Python 3.6+ is installed. You can check this by running python3 --version in your terminal.

  For macOS / Linux / Windows Subsystem for Linux (WSL):

```
curl -fsSL "https://aptos.dev/scripts/install_cli.py" | python3
```

Or with wget:

```
wget -qO- "https://aptos.dev/scripts/install_cli.py" | python3
```

  For Windows (NT):

  - In Powershell

```
iwr "https://aptos.dev/scripts/install_cli.py" -useb | Select-Object
-ExpandProperty Content | python3
```

Updating the CLI:

- To update the Aptos CLI, run aptos update in the terminal.

Part 2: Installation with Homebrew for macOS

Prerequisites:

- Ensure Homebrew is installed on your macOS. If not, install it from brew.sh.

Installation Process:

```
  brew update        # Gets the latest updates for packages
  brew install aptos # Installs the Aptos CLI
```

Part 3: Aptos CLI Binaries

If everything above dont work for you then do following or you can skip this part.

For Linux:

Downloading the Binary:

- Visit the Aptos CLI release page on GitHub.
- Under the latest release, click the "Assets" menu and download the aptos-cli-<version>-<platform>.zip file appropriate for your platform.
- Unzip the file to extract the aptos CLI binary.

Setting Up:

- Move the extracted aptos binary to a local folder, e.g., ~/bin/aptos.
- Make it executable with chmod +x ~/bin/aptos.
- To access the CLI easily, add ~/bin to your PATH in .bashrc or .zshrc.
- Verify the installation by typing ~/bin/aptos help in your terminal.

For Windows:

Downloading the Binary:

- Follow the same process as for Linux by visiting the Aptos CLI release page on GitHub and downloading the relevant .zip file for Windows.

Setting Up:

- Extract the aptos CLI binary into a folder, e.g., \Users\user\Downloads.
- Move the binary to a preferred location on your system.
- Open PowerShell and run .\Downloads\aptos-cli-<version>-Windows-x86_64\aptos.exe help to verify the installation.
- Add path to environment variables.

Smart Contract

1. Initial Setup

Navigate to your project directory and create a new move directory.Initialize the Move project:

```
cd my-first-dapp
```

```
mkdir move
```

```
cd move
```

```
aptos move init --name my_todo_list
```

This command creates a Move.toml file and a sources/ directory inside the move directory.

2. Understanding Key Components

- Move.toml: A manifest file for your Move package, containing metadata like the package name and dependencies.
- sources Directory: Contains your .move module files.

3. Creating a Move Module

1. In our move directory, run `aptos init --network devnet`. Press enter when prompted.
   This creates for us a .aptos directory with a config.yaml file that holds our profile information. In the config.yaml file, we now have our profiles list that holds a default profile. If you open that file, you will see content resembling:

```
default:
    private_key:
"0xee8f387ef0b4bb0018c4b91d1c0f71776a9b85935b4c6ec2823d6c0022fbf5cb"
    public_key:
"0xc6c07218d79a806380ca67761905063ec7a78d41f79619f4562462a0f8b6be11"
    account:
cbddf398841353776903dbab2fdaefc54f181d07e114ae818b1a67af28d1b018
    rest_url: "https://fullnode.devnet.aptoslabs.com"
    faucet_url: "https://faucet.devnet.aptoslabs.com"
```

From now on, whenever we run a CLI command in this move directory, it will run with that default profile. We use the devnet network flag so eventually when we publish our package it will get published to the devnet network.

As mentioned, our sources directory holds our .move module files; so let's add our first Move file.

2. Open the Move.toml file.
3. Add the following code to that Move file, substituting your actual default profile account address from .aptos/config.yaml:

```
[addresses]
todolist_addr='<default-profile-account-address>'
```

4. Contract

Create a new todolist.move file within the sources directory and add the following to that file:

```
module todolist_addr::todolist {

  use aptos_framework::account;
  use std::signer;
  use aptos_framework::event;
  use std::string::String;
  use aptos_std::table::{Self, Table};
  #[test_only]
  use std::string;

  // Errors
  const E_NOT_INITIALIZED: u64 = 1;
  const ETASK_DOESNT_EXIST: u64 = 2;
  const ETASK_IS_COMPLETED: u64 = 3;

  // Key ability allows struct to be used as a storage identifier.
  struct TodoList has key {
    tasks: Table<u64, Task>,
    set_task_event: event::EventHandle<Task>,
    task_counter: u64
  }
```

```
  // Store: Task needs Store as its stored inside another struct(TodoList)
  // Copy: Value can be copied
  // Drop: Value can be dropped by end of scope
  struct Task has store, drop, copy {
    task_id: u64,
    address:address,
    content: String,
    completed: bool,
  }

  // entry function is a func that can be called via transactions.
  // &signer is like address who signed a tx
  public entry fun create_list(account: &signer){
    let todo_list = TodoList {
      tasks: table::new(),
      set_task_event: account::new_event_handle<Task>(account),
      task_counter: 0
    };
    // move the TodoList resource under the signer account
    move_to(account, todo_list);
  }

  public entry fun create_task(account: &signer, content: String) acquires
TodoList {
    // gets the signer address
    let signer_address = signer::address_of(account);
    // assert signer has created a list
    assert!(exists<TodoList>(signer_address), E_NOT_INITIALIZED);
    // gets the TodoList resource
    let todo_list = borrow_global_mut<TodoList>(signer_address);
    // increment task counter
    let counter = todo_list.task_counter + 1;
    // creates a new Task
    let new_task = Task {
      task_id: counter,
      address: signer_address,
      content,
      completed: false
    };
    // adds the new task into the tasks table
    table::upsert(&mut todo_list.tasks, counter, new_task);
    // sets the task counter to be the incremented counter
    todo_list.task_counter = counter;
```

```
    // fires a new task created event
    event::emit_event<Task>(
      &mut borrow_global_mut<TodoList>(signer_address).set_task_event,
      new_task,
    );
  }

  public entry fun complete_task(account: &signer, task_id: u64) acquires
TodoList {
    // gets the signer address
    let signer_address = signer::address_of(account);
    // assert signer has created a list
    assert!(exists<TodoList>(signer_address), E_NOT_INITIALIZED);
    // gets the TodoList resource
    let todo_list = borrow_global_mut<TodoList>(signer_address);
    // assert task exists
    assert!(table::contains(&todo_list.tasks, task_id),
ETASK_DOESNT_EXIST);
    // gets the task matched the task_id
    let task_record = table::borrow_mut(&mut todo_list.tasks, task_id);
    // assert task is not completed
    assert!(task_record.completed == false, ETASK_IS_COMPLETED);
    // update task as completed
    task_record.completed = true;
  }
```

5. Testing the Contract

We can write test cases in contract module itself with

```
aptos move test
```

```
#[test(admin = @0x123)]
  public entry fun test_flow(admin: signer) acquires TodoList {
    // creates an admin @todolist_addr account for test
    account::create_account_for_test(signer::address_of(&admin));
    // initialize contract with admin account
    create_list(&admin);

    // creates a task by the admin account
```

```
    create_task(&admin, string::utf8(b"New Task"));
    let task_count =
event::counter(&borrow_global<TodoList>(signer::address_of(&admin)).set_tas
k_event);
    assert!(task_count == 1, 4);
    let todo_list = borrow_global<TodoList>(signer::address_of(&admin));
    assert!(todo_list.task_counter == 1, 5);
    let task_record = table::borrow(&todo_list.tasks,
todo_list.task_counter);
    assert!(task_record.task_id == 1, 6);
    assert!(task_record.completed == false, 7);
    assert!(task_record.content == string::utf8(b"New Task"), 8);
    assert!(task_record.address == signer::address_of(&admin), 9);

    // updates task as completed
    complete_task(&admin, 1);
    let todo_list = borrow_global<TodoList>(signer::address_of(&admin));
    let task_record = table::borrow(&todo_list.tasks, 1);
    assert!(task_record.task_id == 1, 10);
    assert!(task_record.completed == true, 11);
    assert!(task_record.content == string::utf8(b"New Task"), 12);
    assert!(task_record.address == signer::address_of(&admin), 13);
  }

  #[test(admin = @0x123)]
  #[expected_failure(abort_code = E_NOT_INITIALIZED)]
  public entry fun account_can_not_update_task(admin: signer) acquires
TodoList {
    // creates an admin @todolist_addr account for test
    account::create_account_for_test(signer::address_of(&admin));
    // account can not toggle task as no list was created
    complete_task(&admin, 2);
  }
```

6. Deployment

Deployment is fairly easy just make sure contract is compiling and passing your test cases with

```
aptos move compile && aptos move test
```

When we run this we get response something like this

```
Running Move unit tests
[ PASS    ]
0xcbddf398841353776903dbab2fdaefc54f181d07e114ae818b1a67af28d1b018::todolist:
:account_can_not_update_task
[ PASS    ]
0xcbddf398841353776903dbab2fdaefc54f181d07e114ae818b1a67af28d1b018::todolist:
:test_flow
Test result: OK. Total tests: 2; passed: 2; failed: 0
{
 "Result": "Success"
}
```

The address you can see above is your **Module address** copy it somewhere cause we need it while building frontend.

If everything is working fine we can publish contract with

```
aptos move publish
```

And you will get this…


```
{
 "Result": {
   "transaction_hash":
"0x96b84689a53a28db7be6346627a99967f719946bc22766811a674e69da7783fa",
   "gas_used": 7368,
   "gas_unit_price": 100,
   "sender":
"cbddf398841353776903dbab2fdaefc54f181d07e114ae818b1a67af28d1b018",
   "sequence_number": 2,
   "success": true,
   "timestamp_us": 1674246585276143,
   "version": 651327,
   "vm_status": "Executed successfully"
 }
}
```


You can check Tx in block explorer.

Frontend

1. React setup

Create the React App: In your project's root folder, run
`npx create-react-app client --template typescript`
to create a new client directory.

Start the App:

- Navigate into the client folder: cd client.
- Start the app: npm start.
- Your app will run on http://localhost:3000.

Set Up the UI:

- Install Ant Design with `npm i antd@5.1.4`.
- Update App.tsx to include the UI layout using Ant Design components.
- Import necessary components from Ant Design.

Run the App: Use `npm start` to see the updated UI in your browser.

Update app.tsx with following

```
return (
 <>
   <Layout>
     <Row align="middle">
       <Col span={10} offset={2}>
         <h1>Our todolist</h1>
       </Col>
       <Col span={12} style={{ textAlign: "right", paddingRight: "200px"
}}>
         <h1>Connect Wallet</h1>
       </Col>
     </Row>
   </Layout>
 </>
);
```

Dont forgot the imports

```
import { Layout, Row, Col } from "antd";
```

2. Wallet support

Install Necessary Packages:

- Stop your local server.
- Run the following commands in the client folder:

```
npm i @aptos-labs/wallet-adapter-react@1.0.2
npm i @aptos-labs/wallet-adapter-ant-design@1.0.0
npm i petra-plugin-wallet-adapter
```

Update index.tsx:

- Add imports for the wallet adapter and the Petra wallet.

```
import { PetraWallet } from "petra-plugin-wallet-adapter";
import { AptosWalletAdapterProvider } from
"@aptos-labs/wallet-adapter-react";
```

- Define the array of wallets and wrap your app with the wallet adapter provider.

```
const wallets = [new PetraWallet()];
// ...
<AptosWalletAdapterProvider plugins={wallets} autoConnect={true}>
 <App />
</AptosWalletAdapterProvider>
```

Update App.tsx:

- Import the wallet connect UI package.

```
import { WalletSelector } from "@aptos-labs/wallet-adapter-ant-design";
import "@aptos-labs/wallet-adapter-ant-design/dist/index.css";
```

- Replace the <h1>Connect Wallet</h1> with the WalletSelector component.

```
<Col span={12} style={{ textAlign: "right", paddingRight: "200px" }}>
 <WalletSelector />
</Col>
```

- Run the App: Start your local server with npm start and open the app in the browser to see the wallet connect button and selector modal.

3. Fetch data from chain

Install Aptos SDK:

```
npm i aptos
```

Import and Initialize Provider in App.tsx:

```
import { Provider, Network } from "aptos";
const provider = new Provider(Network.DEVNET);
```

Use Wallet Adapter:

```
import { useWallet } from "@aptos-labs/wallet-adapter-react";
const { account } = useWallet();
```

Fetch TodoList Resource:

- Create a state to track if the account has a list:

```
const [accountHasList, setAccountHasList] = useState<boolean>(false);
```

- Use useEffect to call fetchList whenever the account address changes:

```
useEffect(() => {
 fetchList();
}, [account?.address]);
```

- Define fetchList function:

```
const fetchList = async () => {
 if (!account) return;
 const moduleAddress = "<Your Module Address>";
 try {
 await provider.getAccountResource(account.address,
`${moduleAddress}::todolist::TodoList`);
 setAccountHasList(true);
 } catch (e: any) {
```

```
 setAccountHasList(false);
 }
};
```

Update UI Based on State:

- Conditionally render the "Add new list" button:

```
return (
 <>
   <Layout>
     <Row align="middle">
       <Col span={10} offset={2}>
         <h1>Our todolist</h1>
       </Col>
       <Col span={12} style={{ textAlign: "right", paddingRight: "200px"
}}>
         <WalletSelector />
       </Col>
     </Row>
   </Layout>
   {!accountHasList && (
     <Row gutter={[0, 32]} style={{ marginTop: "2rem" }}>
       <Col span={8} offset={8}>
         <Button block type="primary" style={{ height: "40px",
backgroundColor: "#3f67ff" }}>
           Add new list
         </Button>
       </Col>
     </Row>
   )}
 </>
);
```

Run the App: Use `npm start` to see the updates in your app.

4. Submit data to chain

Extract signAndSubmitTransaction from Wallet Adapter:

```
const { account, signAndSubmitTransaction } = useWallet();
```

Add an onClick Event to the Button:

```
<Button onClick={addNewList} block type="primary" style={{ height: "40px",
backgroundColor: "#3f67ff" }}>
 Add new list
</Button>
```

Define the addNewList Function:

- Build and submit a transaction payload.
- Use signAndSubmitTransaction to submit the transaction.
- Wait for the transaction to complete.
- Update state based on the transaction's success.

```
const addNewList = async () => {
 if (!account) return [];
 // build a transaction payload to be submitted
 const payload = {
   type: "entry_function_payload",
   function: `${moduleAddress}::todolist::create_list`,
   type_arguments: [],
   arguments: [],
 };
 try {
   // sign and submit transaction to chain
   const response = await signAndSubmitTransaction(payload);
   // wait for transaction
   await provider.waitForTransaction(response.hash);
   setAccountHasList(true);
 } catch (error: any) {
   setAccountHasList(false);
 }
};
```

In our fetchList function, find the line:

```
// replace with your own address
const moduleAddress =
"0xcbddf398841353776903dbab2fdaefc54f181d07e114ae818b1a67af28d1b018";
```

And move it to outside of the main App function, right beneath our const provider declarations.

```
export const provider = new Provider(Network.DEVNET);
// change this to be your module account address
export const moduleAddress =
"0xcbddf398841353776903dbab2fdaefc54f181d07e114ae818b1a67af28d1b018";
```

Let's go over the addNewList function code.
First, we use the account property from our wallet provider to make sure there is an account connected to our app.
Then we build our transaction payload to be submitted to chain:

```
const payload = {
 type: "entry_function_payload",
 function: `${moduleAddress}::todolist::create_list`,
 type_arguments: [],
 arguments: [],
};
```

- type is the function type we want to hit - our create_list function is an entry type function.
- function- is built from the module address, module name and the function name.
- type_arguments- this is for the case a Move function expects a generic type argument.
- arguments - the arguments the function expects, in our case it doesn't expect any arguments.

Next, we submit the transaction payload and wait for its response. The response returned from the signAndSubmitTransaction function holds the transaction hash. Since it can take a bit for the transaction to be fully submitted to chain and we also want to make sure it is submitted successfully, we waitForTransaction. And only then we can set our local accountHasList state to true.

Before testing our app, let's tweak our UI a bit and add a Spinner component to show up while we are waiting for the transaction. Add a local state to keep track whether a transaction is in progress:

```
const [transactionInProgress, setTransactionInProgress] =
```

```
useState<boolean>(false);
```

Update our addNewList function to update the local state:

```
const addNewList = async () => {
 if (!account) return [];
 setTransactionInProgress(true);
 // build a transaction payload to be submited
 const payload = {
   type: "entry_function_payload",
   function: `${moduleAddress}::todolist::create_list`,
   type_arguments: [],
   arguments: [],
 };
 try {
   // sign and submit transaction to chain
   const response = await signAndSubmitTransaction(payload);
   // wait for transaction
   await provider.waitForTransaction(response.hash);
   setAccountHasList(true);
 } catch (error: any) {
   setAccountHasList(false);
 } finally {
   setTransactionInProgress(false);
 }
};
```

Update UI with Spinner:

- Show a spinner while the transaction is in progress.
- Display the "Add new list" button based on the accountHasList state.

```
return (
 <>
   ...
   <Spin spinning={transactionInProgress}>
     {!accountHasList && (
       <Row gutter={[0, 32]} style={{ marginTop: "2rem" }}>
         <Col span={8} offset={8}>
           <Button onClick={addNewList} block type="primary" style={{
height: "40px", backgroundColor: "#3f67ff" }}>
             Add new list
```

```
          </Button>
        </Col>
      </Row>
    )}
  </Spin>
 </>
);
```

We have covered how to fetch data (an account's todo list) from chain and how to submit a transaction (new todo list) to chain using Wallet.

Let's finish building our app by implementing fetch tasks and adding a task function.

5. Fetch tasks

1. Create a local state tasks that will hold our tasks. It will be a state of a Task type (that has the same properties we set on our smart contract):

```
type Task = {
 address: string;
 completed: boolean;
 content: string;
 task_id: string;
};

function App() {
  const [tasks, setTasks] = useState<Task[]>([]);
  ...
}
```

2. Update our fetchList function to fetch the tasks in the account's TodoList resource:

```
const fetchList = async () => {
 if (!account) return [];
 try {
   const TodoListResource = await provider.getAccountResource(
     account?.address,
     `${moduleAddress}::todolist::TodoList`
   );
   setAccountHasList(true);
       // tasks table handle
```

```
    const tableHandle = (TodoListResource as any).data.tasks.handle;
        // tasks table counter
    const taskCounter = (TodoListResource as any).data.task_counter;

    let tasks = [];
    let counter = 1;
    while (counter <= taskCounter) {
      const tableItem = {
        key_type: "u64",
        value_type: `${moduleAddress}::todolist::Task`,
        key: `${counter}`,
      };
      const task = await provider.getTableItem(tableHandle, tableItem);
      tasks.push(task);
      counter++;
    }
        // set tasks in local state
    setTasks(tasks);
  } catch (e: any) {
    setAccountHasList(false);
  }
};
```

This part is a bit confusing, so stick with us!

Tasks are stored in a table (this is how we built our contract). To fetch a table item (i.e a task), we need that task's table handle. We also need the task_counter in that resource so we can loop over and fetch the task with the task_id that matches the task_counter.

const tableHandle = (TodoListResource as any).data.tasks.handle;
const taskCounter = (TodoListResource as any).data.task_counter;

Now that we have our tasks table handle and our task_counter variable, lets loop over the taskCounter . We define a counter and set it to 1 as the task_counter / task_id is never less than 1.
We loop while the counter is less then the taskCounter and fetch the table item and push it to the tasks array:

```
let tasks = [];
let counter = 1;
while (counter <= taskCounter) {
  const tableItem = {
    key_type: "u64",
```

```
    value_type: `${moduleAddress}::todolist::Task`,
    key: `${counter}`,
  };
  const task = await provider.getTableItem(tableHandle, tableItem);
  tasks.push(task);
  counter++;
}
```

We build a tableItem object to fetch. If we take a look at our table structure from the contract: tasks: Table<u64, Task>,
We see that it has a key type u64 and a value of type Task. And whenever we create a new task, we assign the key to be the incremented task counter.

```
// adds the new task into the tasks table
table::upsert(&mut todo_list.tasks, counter, new_task);
So the object we built is:
{
 key_type: "u64",
 value_type:`${moduleAddress}::todolist::Task`,
 key: `${taskCounter}`,
}
```

Where key_type is the table key type, key is the key value we are looking for, and the value_type is the table value which is a Task struct. The Task struct uses the same format from our previous resource query:

- The account address who holds that module = our profile account address
- The module name the resource lives in = todolist
- The struct name = Task

The last thing we want to do is display the tasks we just fetched.

6. In our App.tsx file, update our UI with the following code:

```
{
 !accountHasList ? (
   <Row gutter={[0, 32]} style={{ marginTop: "2rem" }}>
     <Col span={8} offset={8}>
       <Button
         disabled={!account}
         block
         onClick={addNewList}
         type="primary"
         style={{ height: "40px", backgroundColor: "#3f67ff" }}
```

```
        >
          Add new list
        </Button>
      </Col>
    </Row>
  ) : (
    <Row gutter={[0, 32]} style={{ marginTop: "2rem" }}>
      <Col span={8} offset={8}>
        {tasks && (
          <List
            size="small"
            bordered
            dataSource={tasks}
            renderItem={(task: any) => (
              <List.Item actions={[<Checkbox />]}>
                <List.Item.Meta
                  title={task.content}
                  description={
                    <a

href={`https://explorer.aptoslabs.com/account/${task.address}/`}
                      target="_blank"
                    >{`${task.address.slice(0,
6)}...${task.address.slice(-5)}`}</a>
                  }
                />
              </List.Item>
            )}
          />
        )}
      </Col>
    </Row>
  );
}
```

That will display the Add new list button if account doesn't have a list or instead the tasks if the account has a list.
Go ahead and refresh your browser - see the magic!
We haven't added any tasks yet, so we simply see a box of empty data. Let's add some tasks!

6. Add tasks

1. Update our UI with an add task input:

```
{!accountHasList ? (
 ...
) : (
 <Row gutter={[0, 32]} style={{ marginTop: "2rem" }}>
       // Add this!
   <Col span={8} offset={8}>
     <Input.Group compact>
       <Input
         style={{ width: "calc(100% - 60px)" }}
         placeholder="Add a Task"
         size="large"
       />
       <Button
         type="primary"
         style={{ height: "40px", backgroundColor: "#3f67ff" }}
       >
         Add
       </Button>
     </Input.Group>
   </Col>
   ...
 </Row>
)}
```

We have added a text input to write the task and a button to add the task.

2. Create a new local state that holds the task content:

```
function App() {
 ...
 const [newTask, setNewTask] = useState<string>("");
 ...
}
```

3. Add an onWriteTask function that will get called whenever a user types something in the input text:

```
function App() {
 ...
 const [newTask, setNewTask] = useState<string>("");

 const onWriteTask = (event: React.ChangeEvent<HTMLInputElement>) => {
```

```
    const value = event.target.value;
    setNewTask(value);
  };
  ...
}
```

4. Find our <Input/> component, add the onChange event to it, pass it our onWriteTask function and set the input value to be the newTask local state:

```
<Input
 onChange={(event) => onWriteTask(event)} // add this
 style={{ width: "calc(100% - 60px)" }}
 placeholder="Add a Task"
 size="large"
 value={newTask} // add this
/>
Cool! Now we have a working flow that when the user types something on the
Input component, a function will get fired and set our local state with
that content.
Let's also add a function that submits the typed task to chain! Find our
Add <Button /> component and update it with the following
<Button
 onClick={onTaskAdded} // add this
 type="primary"
 style={{ height: "40px", backgroundColor: "#3f67ff" }}
>
 Add
</Button>
```

That adds an onClickevent that triggers an onTaskAdded function.
When someones adds a new task we:

- want to verify they are connected with a wallet.
- build a transaction payload that would be submitted to chain.
- submit it to chain using our wallet.
- wait for the transaction.
- update our UI with that new task (without the need to refresh the page).

6. Add an onTaskAdded function with:

```
const onTaskAdded = async () => {
```

```
  // check for connected account
  if (!account) return;
  setTransactionInProgress(true);
  // build a transaction payload to be submited
  const payload = {
    type: "entry_function_payload",
    function: `${moduleAddress}::todolist::create_task`,
    type_arguments: [],
    arguments: [newTask],
  };

  // hold the latest task.task_id from our local state
  const latestId = tasks.length > 0 ? parseInt(tasks[tasks.length -
1].task_id) + 1 : 1;

  // build a newTaskToPush object into our local state
  const newTaskToPush = {
    address: account.address,
    completed: false,
    content: newTask,
    task_id: latestId + "",
  };

  try {
    // sign and submit transaction to chain
    const response = await signAndSubmitTransaction(payload);
    // wait for transaction
    await provider.waitForTransaction(response.hash);

    // Create a new array based on current state:
    let newTasks = [...tasks];

    // Add item to the tasks array
    newTasks.push(newTaskToPush);
    // Set state
    setTasks(newTasks);
    // clear input text
    setNewTask("");
  } catch (error: any) {
    console.log("error", error);
  } finally {
    setTransactionInProgress(false);
  }
```

```
 };
```

Let’s go over on what is happening.
First, note we use the account property from our wallet provider to make sure there is an account connected to our app.
Then we build our transaction payload to be submitted to chain:

```
const payload = {
 type: "entry_function_payload",
 function: `${moduleAddress}::todolist::create_task`,
 type_arguments: [],
 arguments: [newTask],
};
```

- type is the function type we want to hit - our create_task function is an entry type function.
- function- is built from the module address, module name and the function name.
- type_arguments- this is for the case a Move function expects a generic type argument.
- arguments - the arguments the function expects, in our case the task content.

Then, within our try/catch block, we use a wallet provider function to submit the transaction to chain and an SDK function to wait for that transaction. If all goes well, we want to find the current latest task ID so we can add it to our current tasks state array. We will also create a new task to push to the current tasks state array (so we can display the new task in our tasks list on the UI without the need to refresh the page).
TRY IT!
Type a new task in the text input, click Add, approve the transaction and see it being added to the tasks list.

7. Mark task as complete

Next, we can implement the complete_task function. We have the checkbox in our UI so users can mark a task as completed.

1. Update the <Checkbox/> component with an onCheck property that would call an onCheckboxChange function once it is checked:

```
<List.Item actions={[
 <Checkbox onChange={(event) => onCheckboxChange(event, task.task_id)}/>
]}>
```

2. Create the onCheckboxChange function (make sure to import CheckboxChangeEvent from antd - import { CheckboxChangeEvent } from "antd/es/checkbox";):

```
const onCheckboxChange = async (
   event: CheckboxChangeEvent,
   taskId: string
 ) => {
   if (!account) return;
   if (!event.target.checked) return;
   setTransactionInProgress(true);
   const payload = {
     type: "entry_function_payload",
     function:
       `${moduleAddress}::todolist::complete_task`,
     type_arguments: [],
     arguments: [taskId],
   };

   try {
     // sign and submit transaction to chain
     const response = await signAndSubmitTransaction(payload);
     // wait for transaction
     await provider.waitForTransaction(response.hash);

     setTasks((prevState) => {
       const newState = prevState.map((obj) => {
         // if task_id equals the checked taskId, update completed property
         if (obj.task_id === taskId) {
           return { ...obj, completed: true };
         }

         // otherwise return object as is
         return obj;
       });

       return newState;
     });
   } catch (error: any) {
     console.log("error", error);
   } finally {
     setTransactionInProgress(false);
   }
 };
```

Here we basically do the same thing we did when we created a new list or a new task.
We make sure there is an account connected, set the transaction in progress state, build the transaction payload, submit the transaction, wait for it and update the task on the UI as completed.

3. Update the Checkbox component to be checked by default if a task has already marked as completed:

```
<List.Item
 actions={[
   <div>
     {task.completed ? (
       <Checkbox defaultChecked={true} disabled />
     ) : (
       <Checkbox
         onChange={(event) =>
           onCheckboxChange(event, task.task_id)
         }
       />
     )}
   </div>,
 ]}
>
```

...
Try it! Check a task's checkbox, approve the transaction and see the task marked as completed.