DAPP Front End II: Solutions

Question 1

Run the provided commands (also listed below). When prompted, select create empty hardhat project.

```
npm install ethers
npm install --save-dev hardhat @nomiclabs/hardhat-ethers ethers
npx hardhat init
```

Question 2

Create the `contracts` folder, copy the smart contract from last week in the new folder, and compile it using hardhat. Compile the smart contract using the following command:

```
npx hardhat compile
```

Note: Hardhat projects work with a specific file structure, so this will work as long as you have the smart contract in the `contracts` folder and the hardhat project is set up correctly.

Question 3

To configure the deployment to the Sepolia testnet, we need to add a new network to the hardhat config. Your `hardhat.config.js` file should look something like this:

```
require("@nomicfoundation/hardhat-toolbox");

// Go to https://infura.io, sign up, create a new API key
// in its dashboard, and replace "KEY" with it
const INFURA_API_KEY = vars.get("INFURA_API_KEY");

// Replace this private key with your Sepolia account private key
// To export your private key from Coinbase Wallet, go to
// Settings > Developer Settings > Show private key
// To export your private key from Metamask, open Metamask and
// go to Account Details > Export Private Key
// Beware: NEVER put real Ether into testing accounts
const SEPOLIA_PRIVATE_KEY_1 = vars.get("SEPOLIA_PRIVATE_KEY_1");
const SEPOLIA_PRIVATE_KEY_2 = vars.get("SEPOLIA_PRIVATE_KEY_2");;

module.exports = {
  solidity: "0.8.24",
  networks: {
    sepolia: {
      url: `https://sepolia.infura.io/v3/${INFURA_API_KEY}`,
      accounts: [SEPOLIA_PRIVATE_KEY_1, SEPOLIA_PRIVATE_KEY_2]
    }
  }
};
```

To add your private and infura keys, use the configurations variables functionality of hardhat. For example, to add your infura key, you can run the following command:

```
npx hardhat vars set INFURA_API_KEY
```

This will prompt you to enter your infura key. You can then access it in your `hardhat.config.js` file using `vars.get("INFURA_API_KEY")`.

Then, create a new folder called `scripts` in your project. Add a new file called `deploy.js` to this folder. Your `deploy.js` file should look something like this:

```
async function main() {
    const [deployer] = await ethers.getSigners();

    console.log("Deploying contracts with the account:", deployer.address);

    const coin = await ethers.deployContract("CoinFlip");
    await coin.waitForDeployment();

    console.log("Contract address:", await coin.getAddress());
  }

  main()
    .then(() => process.exit(0))
    .catch((error) => {
      console.error(error);
      process.exit(1);
    });
```

Finally, run the following command to deploy your smart contract to the testnet:

```
npx hardhat run scripts/deploy.js --network sepolia
```

Make a note of the contract address, as you will need it later.

Question 4 & 5

Modify your front-end code in `src/app/page.js`. Your `page.js` file should look something like this (please note that the abi and address are absent below and should be replaced with your abi and address):

```
'use client'

import React, { useState } from 'react';
import { ethers } from "ethers";

const CoinFlipForm = () => {
```

```
const abi = [
  ...
];
const address = "...";
const bet = ethers.parseEther('0.01');

// State variables - will be used later
const [account, setAccount] = useState(null)
const [signer, setSigner] = useState(null)
const [contract, setContract] = useState(null);

const [connected, setConnected] = useState(false);
const [message, setMessage] = useState('No Bet');
const [selectedOption, setSelectedOption] = useState(0);
const [password, setPassword] = useState(0);

// Event handlers - will be used later
const handleOptionChange = (event) => {
  setSelectedOption(event.target.value);
};

const handlePasswordChange = (event) => {
  setPassword(event.target.value);
};

const handleConnect = async () => {
  // Logic to connect to wallet or DApp
  if (window.ethereum) {
    try {
      const accounts = await window.ethereum.request({ method: "eth_requestAccounts" });
      setAccount(accounts[0]);

      setConnected(true);

      const provider = new ethers.BrowserProvider(window.ethereum);
      const s = await provider.getSigner();

      setSigner(s);

      setContract(new ethers.Contract(address, abi, s));

    } catch (err) {
      console.log(err);
    }
  } else {
    setConnected(false);
  }
};

const handleMakeBet = async () => {
  // Logic to make a bet
  try {
    if (!password) {
      alert('Please add a password to make a bet');
      return;
    }

    const hash = await contract.getHash(selectedOption, password);
    console.log(hash);
    // "0xfa58b71a6fb66fb38c35354e640506789c3a5ca0c9abe3bb29fa65504a39d89c"
    const tx = await contract.makeBet(hash, { value: bet });
    await tx.wait();

    // console.log(ethers.solidityPackedKeccak256(['bool', 'uint256'], [selectedOption, password]));
    // const tx = await contract.makeBet(contract.getHash(selectedOption, password), { value: bet });
    // const tx = await contract.makeBet(ethers.solidityPackedKeccak256(['bool', 'uint256'], [selectedOption, password]), { value: bet });

    setMessage('Bet Made: ' + bet);
  } catch (err) {
    console.log(err);
  }
};

const handleGuess = async () => {
  // Logic to submit guess
  try {
    const tx = await contract.takeBet(selectedOption, { value: bet });
    await tx.wait();

    setMessage('Guess Submitted');
  } catch (err) {
    console.log(err);
  }
};

const handleReveal = async () => {
  // Logic to reveal result
  try {
    const tx = await contract.reveal(selectedOption, password);
    await tx.wait();

    setMessage('Result Revealed');
  } catch (err) {
    console.log(err);
  }
};

return (
  <div className="max-w-md mx-auto p-6 bg-white rounded-md shadow-md">
    <h1 className="text-2xl text-center font-bold mb-6">Coin Flip DApp</h1>
    <div className="mb-4">
      <label className="block mb-2">Account: {account ? account : "Not Connected"} </label>
    </div>
```

```
      <div className="mb-4">
        <label className="block mb-2">Status: {message}</label>
      </div>
      <div className="mb-4">
        <label className="block mb-2">Select Heads or Tails:</label>
        <select value={selectedOption} onChange={handleOptionChange} className="w-full px-4 py-2 border rounded-md">
          <option value={0}>Heads</option>
          <option value={1}>Tails</option>
        </select>
      </div>
      <div className="mb-4">
        <label className="block mb-2">Password:</label>
        <input type="text" value={password} onChange={handlePasswordChange} className="w-full px-4 py-2 border rounded-md" />
      </div>
      <div className="flex justify-center">
        {!connected && <button onClick={handleConnect} className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded mr-4">Connect</
        {connected && (
          <>
            <button onClick={handleMakeBet} className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded mr-4">Make Bet</button>
            <button onClick={handleGuess} className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded mr-4">Guess</button>
            <button onClick={handleReveal} className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">Reveal</button>
          </>
        )}
      </div>
    </div>
  );
};

export default CoinFlipForm;
```

Question 6

To deploy the front-end, first create a new empty github repository. To push your front-end code to it, run the following commands (replacing `[your-github-repo-url]` with the URL of your new repository):

```
git init
git add .
git commit -m "Initial commit"
git remote add origin [your-github-repo-url]
git branch -M main
git push -u origin main
```

Then, go to Vercel and sign in with your github account. Find the repository you just created and click the import button. Vercel should populate all fields automatically. Click on the deploy button and Vercel will automatically deploy your front-end. You can access it at the URL provided by Vercel.

Bonus Question

Create a new folder called `test`. Add a new file (you can call it whatever you like) to this folder. Create some tests. This file should look something like this (please note that this is a simple example and you should improve and add more tests to cover all the functionality of your smart contract):

```
// This is an example test file. Hardhat will run every *.js file in `test/`,
// so feel free to add new ones.

// Hardhat tests are normally written with Mocha and Chai.

// We import Chai to use its asserting functions here.
const { expect } = require("chai");

// We use `loadFixture` to share common setups (or fixtures) between tests.
// Using this simplifies your tests and makes them run faster, by taking
// advantage of Hardhat Network's snapshot functionality.
const {
  loadFixture,
} = require("@nomicfoundation/hardhat-toolbox/network-helpers");

// `describe` is a Mocha function that allows you to organize your tests.
// Having your tests organized makes debugging them easier. All Mocha
// functions are available in the global scope.
//
// `describe` receives the name of a section of your test suite, and a
// callback. The callback must define the tests of that section. This callback
// can't be an async function.
describe("CoinFlip contract", function () {
  // We define a fixture to reuse the same setup in every test. We use
  // loadFixture to run this setup once, snapshot that state, and reset Hardhat
  // Network to that snapshot in every test.
  async function deployTokenFixture() {
    // Get the Signers here.
    const [owner, addr1, addr2] = await ethers.getSigners();

    // To deploy our contract, we just have to call ethers.deployContract and await
    // its waitForDeployment() method, which happens once its transaction has been
    // mined.
    const coin = await ethers.deployContract("CoinFlip");

    await coin.waitForDeployment();

    const selection = 1;
    const password = 123;
    const hash = await coin.getHash(selection, password);

    const p2Selection = 1;

    // Fixtures can return anything you consider useful for your tests
    return { coin, addr1, addr2, selection, password, hash, p2Selection };
  }

  // You can nest describe calls to create subsections.
  describe("Deployment", function () {
    // `it` is another Mocha function. This is the one you use to define each
```

```
    // of your tests. It receives the test name, and a callback function.
    //
    // If the callback function is async, Mocha will `await` it.
    it("Bet should be 0 initially", async function () {
      // We use loadFixture to setup our environment, and then assert that
      // things went well
      const { coin } = await loadFixture(deployTokenFixture);

      // `expect` receives a value and wraps it in an assertion object. These
      // objects have a lot of utility methods to assert values.

      expect(await coin.bet()).to.equal(0);
    });

    it("getHash should return correct values", async function () {
      const { hash } = await loadFixture(deployTokenFixture);
      expect(hash).to.equal("0x0aaa3ce42286152b719abbca9f317d4abc82039051363ec7758d508c8a88401a");
    });
  });

  describe("Betting", function () {
    it("Should allow player 1 to create a new bet", async function () {
      const { coin, addr1, hash } = await loadFixture(
        deployTokenFixture
      );

      await coin.connect(addr1).makeBet(hash, { value: 100 });

      expect(await coin.bet()).to.equal(100);
      expect(await ethers.provider.getBalance(coin.getAddress())).to.equal(100);
    });

    it("Should allow player 2 to accept a bet", async function () {
      const { coin, addr1, addr2, hash, p2Selection } = await loadFixture(
        deployTokenFixture
      );

      await coin.connect(addr1).makeBet(hash, { value: 100 });
      await coin.connect(addr2).takeBet(p2Selection, { value: 100 });

      expect(await coin.bet()).to.equal(100);
      expect(await ethers.provider.getBalance(coin.getAddress())).to.equal(200);
    });

    it("Should allow player 1 to reveal", async function () {
      const { coin, addr1, addr2, selection, password, hash, p2Selection } = await loadFixture(
        deployTokenFixture
      );

      await coin.connect(addr1).makeBet(hash, { value: 100 });
      await coin.connect(addr2).takeBet(p2Selection, { value: 100 });
      await coin.connect(addr1).reveal(selection, password);

      expect(await ethers.provider.getBalance(coin.getAddress())).to.equal(0);
      expect(await ethers.provider.getBalance(addr2.address)).to.greaterThan(await ethers.provider.getBalance(addr1.address));

    });
  });
});
```

Run the tests using the following command:

```
npx hardhat test
```