SWINBURNE UNIVERSITY OF TECHNOLOGY

2023-COS30049-Computing Technology Innovation Project

Workshop Guide

Workshop 09 - SE

Interact with Smart Contracts & Error Handling

Objective:

This workshop is designed for students who already have a basic understanding of Solidity and FastAPI. Students are guided through a comprehensive journey from understanding the separate roles of frontend and backend in web development, through to the practicalities of backend-blockchain interaction using FastAPI and web3.py. The workshop provided a balanced blend of theoretical knowledge and practical skills, offering participants insights into web development, blockchain interaction, error handling, and smart contract engagement. Practical code snippets and examples provided a foundation for participants to build upon, encouraging further exploration and experimentation in developing decentralized applications.

Workshop Structure:

Part 1: Recap on the Front-end & Back-end

Website development is typically divided into two main parts: frontend and backend, each responsible for the user interface (UI) and server-side logic, respectively.

Frontend

- **Definition**: The frontend is the part with which users directly interact, including the website's interface and functionality.
- **Tech Stack**: HTML, CSS, JavaScript, etc.
- **Frameworks**: React, Angular, Vue.js, etc.
- **Responsibilities**: Displaying data, user interaction, sending requests to the backend, etc.

Backend

- **Definition**: The backend processes requests sent from the frontend, interacts with the database, and then returns data to the frontend.
- **Tech Stack**: Python
- **Frameworks**: FastAPI

- **Responsibilities**: Handling business logic, data storage and retrieval, security, and authorization, etc.

Connecting React and FastAPI using Axios

1. Install Axios

In your React project, install Axios using npm.

npm install axios

2. Use Axios to Send Requests

In your React project, you can use Axios to send HTTP requests to the FastAPI backend.

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    axios.get('http://localhost:8000/data')
      .then((response) => {
        setData(response.data);
      })
      .catch((error) => {
        console.error("Error fetching data: ", error);
      })
  }, []);

  return (
    <div>
      {data ? (
        <div>Data: {data.message}</div>
      ) : (
        <div>Loading...</div>
      )}
    </div>
  );
}

export default App;
```

3. Configure FastAPI Backend

In your FastAPI project, you need to configure CORS middleware to allow requests from the React frontend.

```python
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

# Configure CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Allow all origins
    allow_credentials=True,
    allow_methods=["*"],  # Allow all methods
    allow_headers=["*"],  # Allow all headers
)

@app.get("/data")
async def get_data():
    return {"message": "Hello from FastAPI"}
```

Notes:
- **CORS**: Ensure your FastAPI backend allows cross-origin requests from your React frontend.
- **Environment Variables**: In a production environment, ensure sensitive information (e.g., API URL) is stored in environment variables, not hard-coded in the code.
- **Error Handling (Introduce Later in this workshop)**: Properly handle API request errors and exceptions in React.

Through the above steps, you can use Axios to communicate between the React frontend and FastAPI backend.

Part 2: Recap on the Back-end & Local Blockchain

Let's delve into the theoretical knowledge regarding the interaction between the backend and Ganache, and the role of web3.py and FastAPI.

Role of Ganache

Ganache is a popular Ethereum development tool that provides a private blockchain specifically for local development and testing. Key features of Ganache include:

- **Private Blockchain**: Ganache provides a local, private blockchain for development and testing, allowing developers to work without interacting with the mainnet or testnets.

- **Pre-funded Accounts**: Ganache provides pre-funded accounts with test Ether, enabling developers to conduct transactions and tests without spending real funds.

- **Graphical User Interface**: Ganache provides a GUI, allowing developers to visually inspect the state of the blockchain, account balances, transaction history, etc.

Role of web3.py

web3.py is a Python library for interacting with Ethereum blockchains. It allows developers to:

- **Send Transactions**: Including Ether transfers and smart contract function calls.

- **Query State**: Such as retrieving account balances, reading smart contract states, etc.

- **Deploy and Interact with Smart Contracts**: Including compiling, deploying, and calling methods on smart contracts.

Role of FastAPI

FastAPI is a modern, fast (high-performance) web framework for building APIs with Python. Based on standard Python type hints, it provides an intuitive and easy-to-use API development experience. FastAPI offers:

- **High Performance**: Comparable to NodeJS and Uvicorn (an ASGI server), almost as fast as NodeJS and Uvicorn (an ASGI server).

- **Ease of Use**: Simple and intuitive design, easy to learn, reducing development and debugging time.

- **Data Validation**: Automatically validates request and response data.

Backend Interaction with Ganache

In the development of blockchain applications (often referred to as DApps, decentralized applications), the backend often plays the role of interacting with the blockchain network. In this context, Ganache, serving as a local Ethereum test network, allows developers to test their applications without affecting the mainnet.

The backend interacts with Ganache using web3.py (or similar libraries in other languages) to:

- **Read Data**: Retrieve data from the blockchain, such as account balances, states of smart contracts, etc.

- **Write Data**: Send transactions to the blockchain, such as transfers, smart contract method calls, etc.

FastAPI Interaction with Blockchain

FastAPI can interact with the blockchain using web3.py, providing a middleware that manages all interactions with the blockchain for the frontend. This means that the frontend does not interact directly with the blockchain but interacts through the FastAPI backend, which handles all blockchain logic and data processing, then sends the necessary data to the frontend.

Advantages of this architecture include:

- **Security**: The frontend does not interact directly with the blockchain, reducing security risks (e.g., reducing the risk of private keys being stolen).

- **Flexibility**: The backend can manage interactions with the blockchain more flexibly, such as managing multiple blockchain accounts in the backend, handling complex transaction logic, etc.

Part 3: Error Handeling in Python

Error handling is a fundamental concept in programming that allows you to gracefully manage unexpected or erroneous situations that may occur during the execution of your code. Errors can take many forms, such as invalid inputs, unexpected conditions, or even system failures. Python provides a robust error handling mechanism to help you identify, handle, and recover from these errors, ensuring that your programs run smoothly and reliably.

Key components of error handling in Python

- ➔ Exceptions: Errors in Python are represented as exceptions, which are objects that encapsulate information about the error, including its type and context. When an error occurs, an exception is raised.
- ➔ Try-Except Blocks: Python allows you to use try and except blocks to catch and handle exceptions. The code within the try block is monitored for exceptions. If an exception occurs, the code within the corresponding except block is executed.
- ➔ Exception Types: Python has a wide range of built-in exception types, such as TypeError, ValueError, ZeroDivisionError, and FileNotFoundError. You can catch specific exceptions using multiple except blocks or a single except block that handles a more general exception type.
- ➔ Else Block: The else block can follow the except block(s) and contains code that runs when no exceptions are raised in the try block.
- ➔ Finally Block: The finally block is executed regardless of whether an exception occurred or not. It is often used for cleanup operations, such as closing files or releasing resources.

Basic Error Handling

In FastAPI, you can handle errors by creating exception handlers using the @app.exception_handler() decorator. This allows you to catch exceptions and return a suitable HTTP response.

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.exception_handler(HTTPException)
async def http_exception_handler(request, exc):
    return {"message": f"Oops! {exc.detail}"}

@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id <= 0:
        raise HTTPException(detail="Invalid item ID", status_code=400)
    return {"item_id": item_id}
```

Basic Try-Except Block

Here's a basic structure of a try-except block:

```python
try:
    # Code that might raise an exception
    pass
except SomeException as e:
    # Code that will run if the exception is raised
    pass
```

- The try block contains the code that might raise an exception.
- The except block contains the code that will be executed if an exception occurs.

Example of Try-Except Block

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    result = None
    print("You can't divide by zero!")
```

In this example, attempting to divide by zero will raise a ZeroDivisionError. The except block catches this exception and prints a message instead of allowing the program to crash.

Using Try-Except in FastAPI

In FastAPI, you might use try-except blocks to handle exceptions and provide HTTPException responses when an error occurs:

```python
@app.get("/divide")
async def divide(x: int, y: int):
    try:
        result = x / y
    except ZeroDivisionError:
        raise HTTPException(status_code=400, detail="Cannot divide by zero")
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
    else:
        return {"result": result}
```

Here is the code example:
https://codesandbox.io/p/sandbox/fastapi-demo-kkd6yv?file=%2Fweek9%2FerrorHandeling.py%3A31%2C1

Part 4: Try by yourself to get the value in smart contracts

In this section, you are required to reflect on how to obtain the return values of functions within a smart contract, based on the content discussed last week. We will provide some code hints, and you should integrate this code into your own program through debugging and experimentation to successfully retrieve the contract's return values.

After interacting with the smart contract and obtaining the transaction hash, querying it immediately through getTransactionReceipt may result in an error or no response. This is because transactions on the Ethereum blockchain take some time to be mined and recorded. Querying immediately may return empty results or an error.

Typically, waiting for about 10 to 20 seconds is a relatively safe wait time to ensure that the transaction has either completed or failed and the relevant information has been written to the blockchain. You can add a waiting period in Python as follows:

```
import time
from web3 import Web3

# Initialize the Web3 connection
w3 = Web3(Web3.HTTPProvider('https://your_ethereum_node_url_here'))

# Send the transaction
transaction_hash = w3.eth.sendTransaction({'to': 'receiver_address', 'value': w3.toWei(1, 'ether')})

# Wait for some time
time.sleep(15)  # Adjust the waiting time as needed

# Query the transaction receipt
receipt = w3.eth.getTransactionReceipt(transaction_hash)

if receipt is not None:
    if receipt['status'] == 1:
        print("Transaction successful")
    else:
        print("Transaction failed")
else:
    print("Transaction has not been confirmed yet")
```

Let's delve deeper into using events in Ethereum smart contracts and listening to these events using web3.py in a more detailed manner.

Here is the code sample:

https://codesandbox.io/p/sandbox/fastapi-demo-kkd6yv?file=%2Fweek9%2FcheckTxReciept.py%3A33%2C1

NOTE:

The provided code here is intended solely to assist you in debugging. You will need to incorporate this code into a FastAPI application for testing and execution. You can use the provided code as a reference and then attempt to write your own code to achieve the desired functionality.

Why Use Events?

In Ethereum, function calls to smart contracts are categorized into two types: read calls and transaction calls.

- **Read Calls**: They do not alter the state of the blockchain and are free and can return values. These calls are executed instantly on the local node and return results.
- **Transaction Calls**: They can alter the state of the blockchain (e.g., changing the value of variables or transferring tokens). They need to be mined and added to the blockchain, so they cannot return the result of execution immediately.

Since transaction calls cannot return values directly, we use events as a mechanism to obtain specific results generated by a transaction.

```
pragma solidity ^0.8.0;

contract MyContract {
    event MyEvent(uint256 indexed value);

    function doSomething(uint256 _value) external {
        // ... do something ...

        emit MyEvent(_value);
    }
}
```

In this example, MyEvent is an event that takes a parameter of type uint256. When the *doSomething* function is called, the event logs the value of the _value parameter in the transaction logs.

Listening to Events with web3.py

In the web3.py library in Python, you can listen for and retrieve data from smart contract events.

Create an Event Filter

You can create an event filter to listen for new instances of the event.

```
event_filter = contract.events.MyEvent.createFilter(fromBlock='latest')
```

Retrieve Event Logs

You can use the filter to retrieve new instances of the event and extract data from them.

```
event_logs = event_filter.get_new_entries()

for event in event_logs:
    print("New event:", event['args'])
```

Here is the code sample:

https://codesandbox.io/p/sandbox/fastapi-demo-kkd6yv?file=%2Fweek9%2FeventSample.py%3A30%2C1

NOTE:

The provided code here is intended solely to assist you in debugging. You will need to incorporate this code into a FastAPI application for testing and execution. You can use the provided code as a reference and then attempt to write your own code to achieve the desired functionality.

Part 5: Wrap-up

Congratulations on completing this workshop on web3.py and Ganache! We hope you found this learning experience valuable and gained a deeper understanding of working with local blockchain and web3.py.

Some useful Links:

web3.py official documentation

https://web3py.readthedocs.io/en/stable/

web3.py official Github

https://github.com/ethereum/web3.py

Deploy Smart Contract with Python

https://www.youtube.com/watch?v=zCAhMBedPjc&ab_channel=MammothInteractive