LAB 6 & 7: Managing Complex Data with Structs and Mappings in Solidity and Deploy a smart contract on the Ethereum testnet using MetaMask

Objective:

In this lab, you will learn how to use **structs** and **mappings** in Solidity to manage complex data. Specifically, you will create a contract that can manage **student records**, **employee data**, or any other entity requiring multiple attributes. You will implement functions to **add**, **update**, and **retrieve** this data, using **structs** to store related data, and **mappings** to allow for efficient lookups by a key (e.g., ID).

By the end of this lab, you will:

- Implement **structs** to organize complex data.
- Use **mappings** to store and access structured data efficiently.
- Create functions to **add**, **update**, and **retrieve** data.
- Learn how to apply **modifiers** for validation and control.
- Deploy a smart contract on the **Ethereum testnet** using **MetaMask**

Real-World Scenario:

Imagine you are building a **Student Management System** for a school, where each student has a **unique ID**, **name**, **age**, and **grades**. This contract will store and manage these student records using **structs** and **mappings**.

Assignment Requirements:

Create a **Student Management Contract** with the following functionality:

1. **Student Struct**:
    - Create a Student struct with the following fields:
        - name (string)
        - age (uint)
        - grade (uint)
        - enrollmentDate (uint, representing the timestamp when the student was added).
2. **Mappings**:
    - Create a **mapping** from **student IDs** (uint) to **Student** structs. This will store the student data.
3. **Functions**:

- **Add a Student**:
  - A function addStudent(uint studentId, string memory name, uint age, uint grade) that adds a student to the system.
  - The function should ensure that the student does not already exist using require.

    Help: // Check if the student already exists

    require(students[studentId].enrollmentDate == 0, "Student already exists!");
- **Update Student Data**:
  - A function updateStudent(uint studentId, uint age, uint grade) to allow updating a student's age and grade.
  - Ensure the student exists before updating their data.
- **Retrieve Student Information**:
  - A function getStudent(uint studentId) to fetch and return a student's details (name, age, grade, enrollmentDate).
- **Check Student Existence**:
  - A helper function studentExists(uint studentId) that returns a boolean indicating whether the student exists in the system.

4. **Modifiers**:
   - **Only Existing Student Modifier**:
     - Create a modifier onlyExistingStudent to check if a student exists before performing actions like updating their data.

       // Modifier to check if the student exists

       modifier onlyExistingStudent(uint studentId) {

         require(students[studentId].enrollmentDate != 0, "Student does not exist!");

         _;

       }
5. **Testing the Contract**:
   - Test the contract by deploying it in **Remix IDE** and interacting with it by adding, updating, and retrieving student data.
   - Ensure the following validations:
     - Students cannot be added twice with the same ID.

- The data is correctly updated.
- Retrieve the correct details of any student using their ID.'

Steps to Deploy a Solidity Smart Contract on the Existing Sepolia Testnet Using MetaMask and Remix IDE

Step 1: Install MetaMask

1. **Download MetaMask**:
    - Install the **MetaMask browser extension** from MetaMask.
    - After installation, create a **new wallet** or restore an existing wallet using your seed phrase.

Step 2: Set Up MetaMask with Sepolia Testnet

1. **Add Sepolia Testnet in MetaMask**:
    - Open **MetaMask** and click on the **network dropdown** (top center, where it says "Ethereum Mainnet").
    - Select **"Add Network"**
    - Select Sepolia in testnet section
2. **Switch to Sepolia Network**:
    - After adding the Sepolia network, go to the **MetaMask** extension and select **"Sepolia Test Network"** from the network dropdown.

Step 3: Get Sepolia Testnet Ether

1. **Request Sepolia Testnet Ether**:
    - Go to a **Sepolia Faucet** to get some free Sepolia testnet Ether (needed for deployment and transaction fees).
    - Visit Sepolia Faucet or search for a faucet by typing "Sepolia Testnet Faucet" in your browser.
    - Enter your **MetaMask address** (the public address in MetaMask) and request some test Ether.
2. **Confirm the Transaction**:

- After requesting, it may take a few minutes for the faucet to send the test Ether to your MetaMask wallet. Check your MetaMask balance to confirm it has received the Sepolia Ether.

Step 4: Deploy the Smart Contract Using Remix IDE

1. **Open Remix IDE**:
   - Go to Remix IDE in your browser.
2. **Connect MetaMask to Remix**:
   - In Remix, open the **"Deploy & Run Transactions"** plugin (left panel).
   - Change the **Environment** to **Injected Web3**. This connects Remix to your MetaMask wallet.
   - You should see your **Sepolia network** listed in the network dropdown in Remix (ensure you are connected to the Sepolia Testnet in MetaMask).
   - If prompted, **connect Remix to your MetaMask wallet** by approving the connection in MetaMask.
3. **Compile the Contract**:
   - Go to the **Solidity Compiler** tab in Remix (left panel).
   - Select the appropriate **compiler version** (make sure it matches the version used in your Solidity contract).
   - Click **Compile** to compile your contract.
4. **Deploy the Contract**:
   - In the **"Deploy & Run Transactions"** tab, select the compiled contract.
   - Under the **Deploy** section, click **Deploy**.
   - MetaMask will pop up asking for your confirmation to deploy the contract. Confirm the transaction.
   - After the transaction is mined, you should see the contract deployed on the Sepolia Testnet.

Step 5: Interact with the Deployed Contract

1. **View the Contract in Remix**:
   - Once deployed, Remix will show you the **contract address**.

- You can interact with your contract directly in Remix using the **Deployed Contracts** section. Here you will see all the functions of your deployed contract.

2. **Test the Functions**:
    - Now that the contract is deployed, you can call functions like addStudent, updateStudent, and getStudent to test the contract and manage student records.
    - Make sure to test adding and updating student records to ensure everything works as expected.
3. **Confirm Transactions in MetaMask**:
    - For each interaction with the contract (e.g., adding students or updating records), you may need to confirm transactions in MetaMask to complete the action. You'll see the gas fees and transaction details in the MetaMask pop-up.

Step 6: Verify the Contract on Sepolia Etherscan

1. **Find Contract on Sepolia Etherscan**:
    - After deployment, you can visit **Etherscan** and search for your contract address to see its transaction history and any actions that have been performed on it.
2. **Verify Contract Code**:
    - You can verify the contract code on Etherscan by going to the contract page and clicking **Verify and Publish**.
    - Follow the steps on Etherscan to verify the source code for your contract.

Sample Test Case 1: Add Students

Let's add the three students to the contract using the addStudent function:

```
addStudent(1, "Aarav Kumar", 18, 85);
addStudent(2, "Priya Sharma", 19, 90);
addStudent(3, "Saanvi Patel", 17, 88);
```

- **Expected Output:**
    - The students will be added to the contract, and the following events will be emitted:
        - StudentAdded(1, "Aarav Kumar", 18, 85)
        - StudentAdded(2, "Priya Sharma", 19, 90)
        - StudentAdded(3, "Saanvi Patel", 17, 88)

Sample Test Case 2: Retrieve Student Information

Now, we will retrieve the information of each student using the getStudent function. For example:

```
getStudent(1);  // Aarav Kumar
getStudent(2);  // Priya Sharma
getStudent(3);  // Saanvi Patel
```

- **Expected Output:**
  - For getStudent(1) (Aarav Kumar):

    ("Aarav Kumar", 18, 85, [timestamp])

  - For getStudent(2) (Priya Sharma):

    ("Priya Sharma", 19, 90, [timestamp])

  - For getStudent(3) (Saanvi Patel):

    ("Saanvi Patel", 17, 88, [timestamp])

  - Note: The [timestamp] will show the block timestamp when the student was added.

Sample Test Case 3: Update Student Information

Next, let's update **Saanvi Patel's** age and grade using the updateStudent function:

```
updateStudent(3, 18, 92);
```

- **Expected Output**:
  - After the update, the following StudentUpdated event will be emitted:

    StudentUpdated(3, 18, 92)

- **Retrieve the updated student information** using getStudent(3):

  ```
  getStudent(3);  // Saanvi Patel
  ```

  - **Expected Output**:

    ("Saanvi Patel", 18, 92, [timestamp])

    - The age will now be 18 and grade will be 92.

Sample Test Case 4: Check if Student Exists

Now, we'll use the studentExists function to check if students with IDs 1, 2, and 3 exist in the system.

```
studentExists(1);  // True
studentExists(2);  // True
studentExists(3);  // True
```

- **Expected Output:**
  - For studentExists(1):

    true

  - For studentExists(2):

    true

  - For studentExists(3):

    true

If you check for a non-existent student, like studentExists(4), the expected output should be false.