Student Management Decentralized Application Part I

1. Setting up the development environment.
2. Creating a node project using an express generator.
3. Writing the smart contract.
4. Running the geth client.
5. Compiling and deploying the smart contract.
6. Creating a user interface to interact with the smart contract.

Background

The student management system is a decentralized application for managing the details of students. The application stores a Unique ID, Name, Age and disability status of the student to the blockchain and uses the unique id to retrieve the data from the blockchain.

Setting up the development environment

Before we start we need some tools and dependencies. Please install the following:

- Node.js and npm (comes with Node)
- Git
- Geth

Once we have those installed, we need to install **express-generator** to create an express application template. Install using the following command.

```
npm install express-generator -g
```

Creating a node project using the express generator

The express generator initializes the current folder with a node express project. so first create a directory in your development folder of choice and then moving inside it.

```
$ mkdir smd
```

```
$ cd smd
```

Use the express command to initialize the node project:

```
$ express --view=ejs
```

Note: ejs (same as html) is the template engine we use for this project.

Now use npm to install all the necessary modules

```
$ npm install
```

We also need to install web3, a library that is a bridge between the front end and the blockchain, use the following command.

```
$ npm install web3
```

Directory structure

The project directory contains the following files and folders.

★ **app.js**: routing specification.
★ **bin/**: contains server configuration file.
★ **node_modules/**: node packages needed for the project are stored here.
★ **package.json**: lists the packages used for the projects and any associated scripts.
★ **package-lock.json**: lists out the exact version and source of the packages used for the project.
★ **public/**: js script, picture, css (can be accessed without server).
★ **routes/**: routes js files are stored here.
★ **views/**: contains UI files

SMD Application Part I

Writing the smart contract.

We'll start our dapp by writing the smart contract that acts as the back-end logic and storage. Create directory named **contracts**

```
$ mkdir contracts
```

Create a new file named **SMD.sol** in the contracts/ directory. Add the following content to the file:

```
pragma solidity ^0.5.0;

contract Person {

}
```

Note: The minimum version of Solidity required is noted at the top of the contract: **pragma solidity ^0.5. 0**. The pragma command means "additional information that only the compiler cares about", while the caret symbol (^) means "the version indicated or higher".

Solidity is a statically-typed language, meaning data types like strings, integers, and arrays must be defined. Solidity supports the use of structures. A struct in solidity is just a custom type that you can define. You define the struct with a name and associated properties inside of it.

Add the following variable on the next line after **contract Person {**.

```
struct details{
      string name;
      uint age;
      bool disabled;
}
```

We created a strut called details for storing the details of the student.

Next add the following lines, which creates a mapping to the struct details. Mappings are hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type’s default value.

```
mapping(uint => details) profile;
```

Here we create mapping an **uint** to the **struct**, that is we can use role number to uniquely identify each entry in the mapping.

SMD Application Part I

Let’s create our first function: storing student data. Add the following function to the smart contract after the variable declaration we set up above.

```
function setProfile(uint _studendId,string memory _name, uint _age, bool  _disabled) public {
        profile[_studendId] = details(_name, _age, _disabled);
}
```

The **setProfile** function adds the student’s data to the mapping, using the role number as the index for each entry. The memory keyword is used indicate the use of dynamic variable.

As for our second function, the above function enters value into the mapping. The **getProfile** function retrieves the data from mapping.

```
function getProfile(uint _studentId) public view returns(string memory _name, uint _age, bool _disabled) {
        return (profile[_studentId].name, profile[_studentId].age, profile[_studentId].disabled);
}
```

Note: In Solidity the types of both the function parameters and output must be specified. The view keyword indicates that the function is only reading from the blockchain and no state change is made, in other words no gas cost.

SMD Application Part I

Our final contract will look like this:

```
pragma solidity ^0.5.0;

contract Person {

    struct details{
        string name;
        uint age;
        bool disabled;
    }

    mapping(uint => details) profile;

    function setProfile(uint _studendId,string memory _name, uint
_age, bool  _disabled) public {
        profile[_studendId] = details(_name, _age, _disabled);
    }

    function getProfile(uint _studentId) public view returns(string
memory _name, uint _age, bool _disabled) {
        return (profile[_studentId].name, profile[_studentId].age,
profile[_studentId].disabled);
    }
}
```

Running the geth client

Now that we have written our contract, the next step is to compile and deploy. For that we will run the geth client in dev mode, by using the following command:

```
$ geth --dev --rpc --rpcport 8545 --rpccorsdomain "*" --rpcapi " personal,
db, eth, net, web3"
```

This will start our geth client on dev mode, with one prefunded account.

Compiling and deploying the smart contract.

Now open the Remix IDE

Go to: http://remix.ethereum.org

1. Compile the contract
2. Connect to the geth client using the Web3 Provider environment in remix and deploy the contract.
3. Store the contract address and ABI somewhere, for later use.

Note: Details on how to use remix can be found in the “Introduction to Remix IDE” manual.

Creating a user interface to interact with the smart contract

Now that we have our contract compiled and deployed our contract to the blockchain network, let's create an interface to interact with the smart contract.

Open the app.js file and add the following line below other require statements, to import and create a web3 object to be used throughout the project.

```
Web3 = require("web3");
```

Next, to establish connection to the blockchain, add the following code in the next line.

```
const web3 = new Web3('http://localhost:8545');
```

The chain is running in localhost on port 8545.

Now let's declare some global variables, add the following code in the next line:

```
accountAddress = "<address>";
const contractAddress = "<address>";
const contractAbi = <smart contract abi>
```

Remember I told you to store the ABI and address of the contract we compiled and deployed to the network paste them here, also paste an account address in the network, from which transaction cost for all the write operation to the blockchain can be given. Usually the Coinbase is used.

Let's create an object using the above values to interact with contract, add the following code in the next line:

```
MyContract = new web3.eth.Contract(contractAbi, contractAddress);
```

SMD Application Part I

Now let's create some routes add the following code:

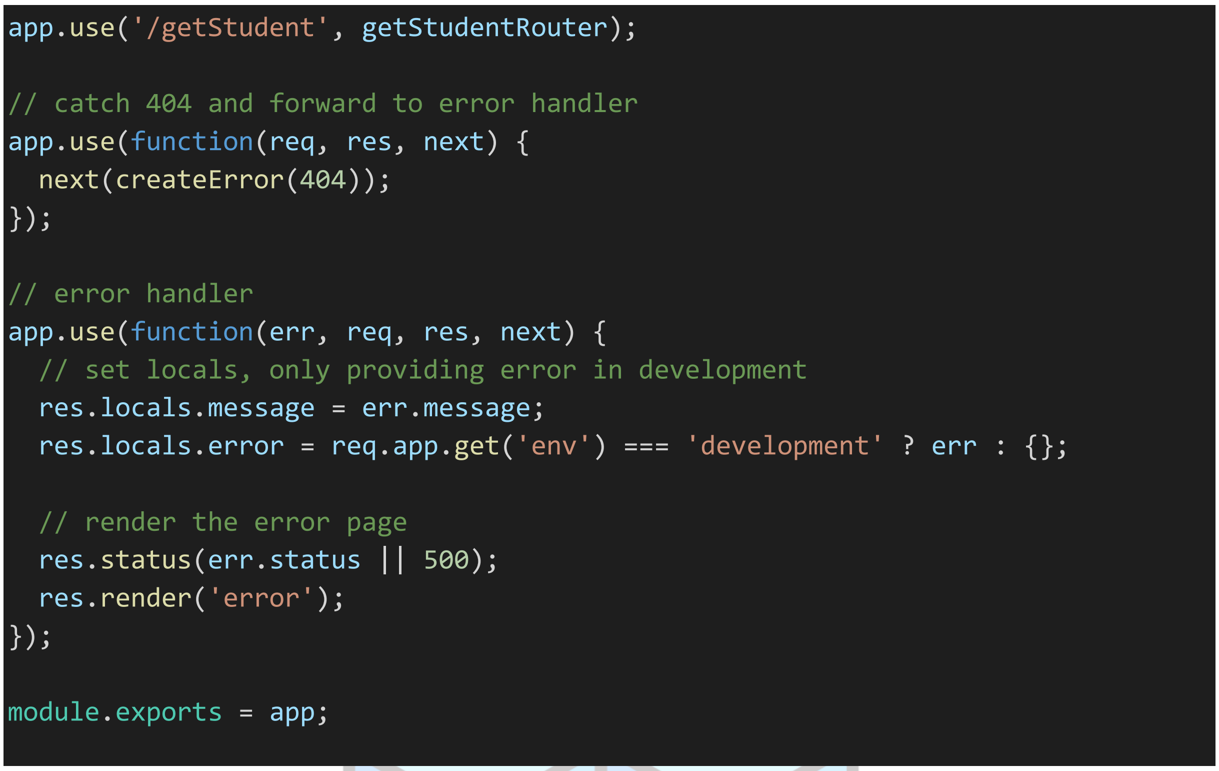
```
var setStudentRouter = require("./routes/setStudent");
var getStudentRouter = require("./routes/getStudent");
```

```
app.use('/setStudent', setStudentRouter);
app.use('/getStudent', getStudentRouter);
```

The final **app.js** file will look like this:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');

//-------------------WEB3 Integration starts----------------------

Web3 = require("web3");

const web3 = new Web3('http://localhost:8545');

accountAddress = "<address>";
const contractAddress = "<address>";
const contractAbi = <smart contract abi>

MyContract = new web3.eth.Contract(contractAbi, contractAddress);

//-------------------WEB3 Integration Stops-----------------------

var indexRouter = require('./routes/index');
var setStudentRouter = require("./routes/setStudent");
var getStudentRouter = require("./routes/getStudent");

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/setStudent', setStudentRouter);
```

SMD Application Part I



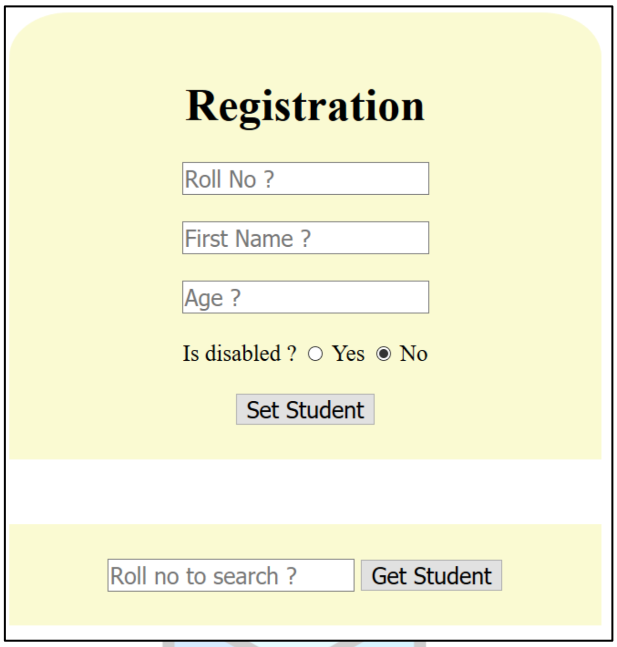
```
app.use('/getStudent', getStudentRouter);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

Now open index.ejs file and paste the following code:

```
<html>
  <head>
    <title>Web3 Example</title>
    <style>
      input { font-size: 20px; }
      form {
        background-color:lightgoldenrodyellow;
        margin-left: 500px;
        margin-right: 500px;
        padding: 30px;
        text-align: center;
        font-size: 20px;
      }
      #form1{
        border-top-right-radius: 10%;
        border-top-left-radius: 10%;
      }
      #form2{
        border-bottom-left-radius: 50%;
        border-bottom-right-radius: 50%;
      }
    </style>
  </head>
  <body>
    <form action="setStudent" method="POST" id="form1">
      <h1>Registration</h1>
      <input type="text" name="uid" placeholder=" Roll No ?" required/>
      </br></br>
      <input type="text" name="firstname" placeholder="First Name ?"
required/>
      </br></br>
      <input type="text" name="age" placeholder="Age ?" required/>
      </br></br>
      Is disabled ?
      <input type="radio" id="rbtrue" name="disabled" value="true" />
      <label for="rbtrue">Yes</label>
      <input type="radio" id="rbfalse" name="disabled" value="false" checked/>
      <label for="rbfalse">No</label>
      </br></br>
      <input type="submit" value="Set Student"/>
    </form>

    <br/>
    <br/>

    <form action="getStudent" method="GET" id="form2">
```

```
      <input type="text" name="uid" placeholder=" Roll No to search ?"
required/>
      <input type="submit" value="Get Student"/>
    </form>
  </body>
</html>
```

Which will result in a web page as follows:



Registration

Roll No ?

First Name ?

Age ?

Is disabled ? ○ Yes [✓ No]

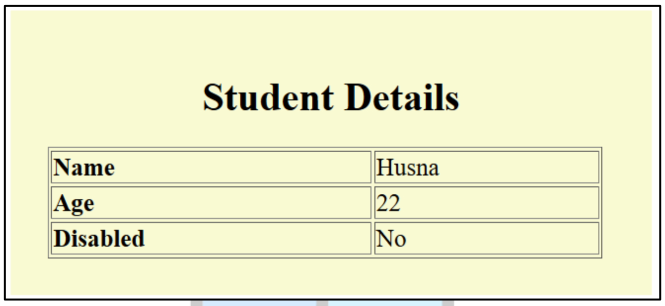
Set Student

Roll no to search ? | Get Student

Create a new file **studentView.ejs** and paste the following code:

```
<html>
<head>
    <title>Web3 Example</title>
    <style>
        input {
            font-size: 20px;
        }

        div {
            background-color:lightgoldenrodyellow;
            margin-left: 500px;
            margin-right: 500px;
            padding: 30px;
        }

        div>table {
            font-size: 20px;
            text-align: left;
        }

        h1{text-align: center;}

    </style>
</head>

<body>
    <div id="div1">
        <h1>Student Details</h1>

      <table border="1" width="450px;">
            <tr>
                <th>Name</th>
                <td><%= result._name %></td>
            </tr>
            <tr>
                <th>Age</th>
                <td><%= result._age %></td>
            </tr>
            <tr>
                <th>Disabled</th>
                <td>
                    <% if(result._disabled){ %>
                        Yes
                    <% } else { %>
                        No
                    <% } %>
                </td>
            </tr>
        </table>
    </div>
</body>
</html>
```

Which will result in a web page as follows:



Student Details

Name	Husna
Age	22
Disabled	No

SMD Application Part I

Now in the routes folder create a file setStudent.js and paste the following code:

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.post('/', function(req, res, next) {
  data = req.body;
  console.log(data);
  MyContract.methods.setProfile(data.uid, data.firstname, data.age,
JSON.parse(data.disabled))
    .send({from:accountAddress})
    .then((txn) => {
        res.send(txn);
    })
});

module.exports = router;
```

Here using **MyContract** object we call the **setProfile** function by passing roll no, firsname, age and disabled or not and will get the transaction receipt as output.

Next in the routes folder create a file getStudent.js and paste the following code:

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.get('/', function(req, res, next) {
  MyContract.methods.getProfile(req.query.uid)
    .call({from:accountAddress})
    .then((result) => {
        console.log(result);
        res.render("studentView", {result : result});
    })
});

module.exports = router;
```

The getProfile function takes in roll number as input and returns the corresponding details and is displayed on the webpage.

Now the moment of truth, run the project using:

```
$ npm start
```

Go to: http://localhost:3000