Challenge: Onchain riddle website.

Key Points:

- Here is the contract address on Sepolia (verified): https://sepolia.etherscan.io/address/0x06BC0F3cF252735E8C0AcD7A94577417790A50Fe#readContract
- I chose to generate an address to deploy it and to use it in my "script" (an API endpoint) to set a new riddle.
- I use Wagmi and Viem to generate the whole typing and hooks for the project.
- I had to modify the smart contract to prevent users from submitting an answer that has already been submitted for the riddle.
- I used hooks to listen to the events.
- It took me about 10 hours to do it (in several sessions).

I took the time to read through the exercise. It simply involves creating a frontend and retrieving riddles from a contract, then being able to propose another riddle. The difficulty may come from proposing a new riddle. Looking at the contract, only the "bot", which is actually the account that deploys the contract (the owner), can propose a new riddle.

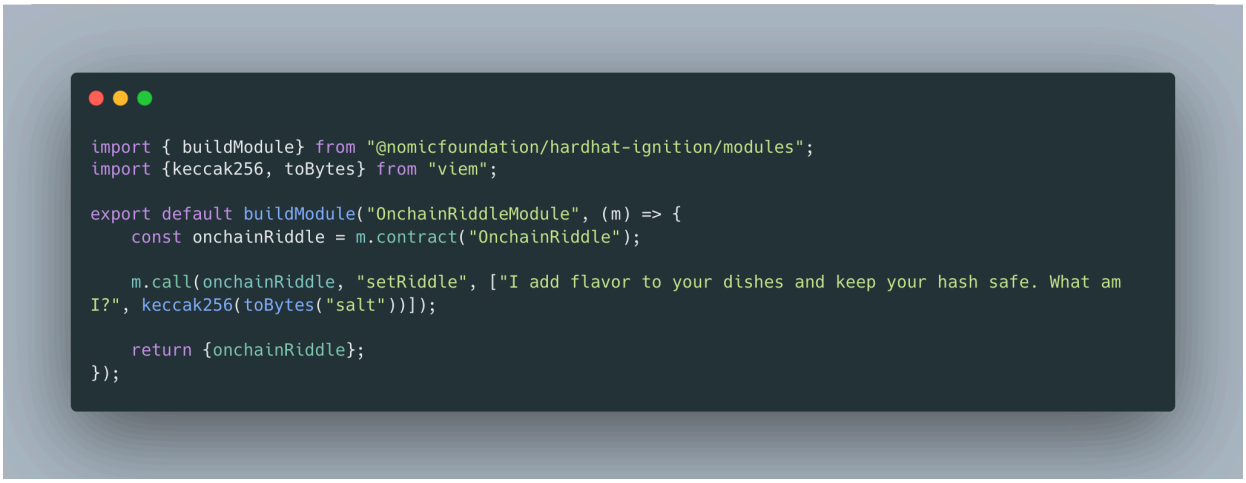
A simple approach is to generate an address beforehand that will be responsible for deploying the contract and can then add new riddles. On local, I will use the default first address and on Sepolia I will generate a new address.

PUBLIC_KEY= 0xF1F136Fa96C6087eAB1f7f91Cb5F9B073e5E00ca
PRIVATE_KEY=5fbfa298677053aec347ff2aa2d3c8895e9b3ce586b80fd87ba161f56a233e1b

At this stage of my thinking, I deduce that I need a "backend" to execute a signed transaction with the "bot/admin" address to perform this type of transaction without exposing the private key to the client. I therefore choose to go with a Next.js project for the frontend part and a classic Hardhat project for the contract.

I created a Hardhat project in which I copy and paste the onChainRiddle contract. I slightly modified the contract, but I'll come back to this later.

I create the deployment script in which I also define the first riddle that was provided to me as well as the hash of the answer.



```
import { buildModule} from "@nomicfoundation/hardhat-ignition/modules";
import {keccak256, toBytes} from "viem";

export default buildModule("OnchainRiddleModule", (m) => {
    const onchainRiddle = m.contract("OnchainRiddle");

    m.call(onchainRiddle, "setRiddle", ["I add flavor to your dishes and keep your hash safe. What am
I?", keccak256(toBytes("salt"))]);

    return {onchainRiddle};
});
```

I modified the Hardhat configuration file to be able to deploy on Sepolia.

```
import { type HardhatUserConfig, vars } from "hardhat/config";
import "@nomicfoundation/hardhat-toolbox-viem";

const PRIVATE_KEY = vars.get("PRIVATE_KEY");
const ALCHEMY_URL = vars.get("ALCHEMY_URL");
const ETHERSCAN_API_KEY = vars.get("ETHERSCAN_API_KEY");

const config: HardhatUserConfig = {
  solidity: "0.8.28",
  etherscan: {
    apiKey: ETHERSCAN_API_KEY,
  },
  networks: {
    sepolia: {
      url: `https://eth-
sepoliacgoatchemy.com/v2/${ALCHEMY_URL}`,
    },
  },
};

export default config;
```

I can run a Hardhat node using npx hardhat node. I wrote a script to deploy it on my two environments (Hardhat and Sepolia) that I can use by running pnpm run deploy or deploy-sepolia.

```
{
  "name": "hardhat-project",
  "scripts": {
    "deploy": "npx hardhat ignition deploy ignition/modules/Onchainriddle.ts --network localhost",
    "deploy-sepolia": "npx hardhat ignition deploy ignition/modules/Onchainriddle.ts --network sepolia
--deployment-id sepolia-deployment",
    "test": "npx hardhat test"
  },
  "devDependencies": {
    "@nomicfoundation/hardhat-toolbox-viem": "^3.0.0",
    "hardhat": "^2.24.1"
  },
  "dependencies": {
  }
}
```

Now I can move on to the frontend.

I go with a classic Next.js installation (TypeScript and Tailwind). I add Viem and Wagmi to be able to simply use blockchain call functions (viem) and reactivity (wagmi).

I write a wagmi.config.ts and use the wagmi CLI (npx wagmi generate) to be able to type my project from the Hardhat project.

I took care beforehand to add Hardhat and React plugins as dependencies to have my smart contract hooks directly generated. The output is found in the src/generated.ts file.

```
import { defineConfig } from '@wagmi/cli'
import { hardhat, react } from '@wagmi/cli/plugins'

export default defineConfig({
    out: 'src/generated.ts',
    plugins: [hardhat({
        project: '../zama-contract/',
        deployments: {
            OnchainRiddle: {
                31337: '0x5FbDB2315678afecb367f032d93F642f64180aa3', // Hardhat deterministic first
contract address
                11155111: '0x06BC0F3cF252735E8C0AcD7A94577417790A50Fe' // Sepolia deployment address
            }
        }
    }), react()]
});
```

I then group my Wagmi and TanStack Query providers which will allow me to make my blockchain calls

```
"use client";
import { WagmiProvider } from 'wagmi'
import { config } from '@/config';
import { QueryClient, QueryClientProvider } from '@tanstack/react-
query';
const queryClient = new QueryClient()


const Providers = ({
    children,
}: Readonly<{
    children: React.ReactNode;
}>) => {
    return <><WagmiProvider config={config}>
        <QueryClientProvider client={queryClient}>
            {children}
        </QueryClientProvider>
    </WagmiProvider></>
}

export default Providers;
```

I use AI to create a basic frontend in terms of design.

I can then add the basic functions in a header.tsx component (useConnect, useSwitchChain, useAccount, useBalance).

The main code for the riddle game is located in riddle-form.tsx.

I use different hooks to retrieve the properties that interest me and that allow displaying the UI according to the conditions.

```
const [attempts, setAttempts] = useState<string[]>([]);

    const { isLoading: isRiddleWinnerLoading, data: riddleWinner, refetch: refetchWinner } =
useReadOnchainRiddleWinner();
    const { isLoading, data: riddle, refetch: refetchRiddle } = useReadOnchainRiddle(
        {
            functionName: 'riddle',
            query: {
                enabled: isConnected,
            }
        }
    )
    const { writeContractAsync, data: hash } = useWriteOnchainRiddleSubmitAnswer();
    const { isLoading: isTransactionPending } =
        useWaitForTransactionReceipt({
            hash,
        })

    useWatchOnchainRiddleRiddleSetEvent({
        onLogs: async () => {
            await refetchRiddle()
            await refetchWinner();
        }
    });

    useWatchOnchainRiddleAnswerAttemptEvent({
        fromBlock: BigInt(0),
        args: {
            riddleHash: keccak256(toBytes(riddle!))
        },
        onLogs: async (logs) => {
            const previousAnwser = logs.map(x => x.args.anwser).filter(x => x !== null && x !==
undefined);
            const uniqueAnswer = [...new Set(previousAnwser)];
            if (uniqueAnswer && uniqueAnswer.length !== attempts.length) {
                setAttempts(uniqueAnswer);
            }
            await refetchWinner();
        },
        enabled: riddle !== null && riddle !== undefined && riddleWinner === zeroAddress
    });
```

In the exercise description it was indicated that **'Avoid several players to submit the same answer to a riddle.**'
For this I had to modify the smart contract so that each time there is an AnswerAttempt event I keep a trace in the logs of the question and the answer. This way I only have to listen to the logs and I know which answers have been proposed for which question.

```
emit AnswerAttempt(msg.sender, keccak256(abi.encodePacked(riddle)), winner == msg.sender,
_answer);
```

My function that allow to submit an answer is the following.

```
const submit = async (e: React.FormEvent<HTMLFormElement>) => {
        e.preventDefault();
        const formData = new FormData(e.target as HTMLFormElement);
        const answer = formData.get('answer') as string;

        if (attempts.includes(answer.toLowerCase())) {
            toast.error("Someone already tried this answer.");
            return;
        }

        await writeContractAsync({
            args: [answer.toLowerCase()],
        }, {
            onSuccess: async () => {
                const { data } = await refetchWinner();
                if (data === address) {
                    toast.success(`Congratulation ! You found the answer :
${answer}`)
                    setAttempts([])
                    try {
                        await fetch('/api/riddle/');
                    } catch (err) {
                        console.error('Unable to generate a new riddle:', err);
                    }
                } else {
                    toast.error(`Wrong answer !`)
                    setAttempts(a => [...a, answer])
                }
            },
            onError: async(error) => {
                console.log(error.message)
            }
        });

    }
```

Once the answer is found, I call the 'bot' which is actually an API endpoint in my case to regenerate a new answer. I create a Viem client that I extend with public functionalities

to be able to make public calls.

Notably to ensure that the current riddle has indeed been found.

If indeed the answer has been found then I can continue and pick a new riddle.

Since I use the same private key as the address that deployed the contract I can go throught the 'onlyBot' modifier and define a new riddle.

```ts
import { createWalletClient, http, keccak256, type Hex, publicActions, toBytes, zeroAddress } from
"viem";
import { privateKeyToAccount } from "viem/accounts";
import { onchainRiddleAbi, onchainRiddleAddress } from "../../../src/generated";
import { hardhat, sepolia } from "viem/chains";

export async function GET() {
    try {
        if(!process.env.NETWORK && !process.env.PRIVATE_KEY) {
            throw new Error("Missing environment variables !");
        }
        const network = process.env.NETWORK as string;
        const botPrivateKey = process.env.PRIVATE_KEY as Hex;
        const chainId = network === "sepolia" ? hardhat.id : sepolia.id
        const riddles = [
            {
                "riddle": "What has roots as nobody sees, Is taller than trees, Up, up it goes, And yet
never grows?",
                "answer": "mountain"
            },
            {
                "riddle": "Voiceless it cries, Wingless flutters, Toothless bites, Mouthless mutters.",
                "answer": "wind"
            },
            {
                "riddle": "It cannot be seen, cannot be felt, Cannot be heard, cannot be smelt. It lies
behind stars and under hills, And empty holes it fills. It comes out first and follows after, Ends
life, kills laughter.",
                "answer": "dark"
            },
            {
                "riddle": "Alive without breath, As cold as death; Never thirsty, ever drinking, All in
mail never clinking.",
                "answer": "fish"
            },
            {
                "riddle": "This thing all things devours; Birds, beasts, trees, flowers; Gnaws iron,
bites steel; Grinds hard stones to meal; Slays king, ruins town, And beats mountain down.",
                "answer": "time"
            }
        ];

        const account = privateKeyToAccount(botPrivateKey);

        const client = createWalletClient({
            chain: hardhat,
            transport: http(),
            account
        }).extend(publicActions);

        const winnerAddress = await client.readContract({
            abi: onchainRiddleAbi,
            functionName: "winner",
            address: onchainRiddleAddress[chainId]
        });

        if (winnerAddress === zeroAddress) {
            return Response.json({
                message: "There is still an active riddle",
            }, { status: 200 });
        }

        const randomRiddle = riddles[Math.floor(Math.random() * riddles.length)];

        const { request } = await client.simulateContract({
            abi: onchainRiddleAbi,
            functionName: 'setRiddle',
            args: [randomRiddle.riddle, keccak256(toBytes(randomRiddle.answer))],
            address: onchainRiddleAddress[hardhat.id]
        });

        const res = await client.writeContract(request);

        return Response.json({
            message: "Riddle set successfully",
            transactionHash: res,
            riddle: randomRiddle.riddle
        }, { status: 200 });
    } catch (error) {
```

What could be improved:

- The design of course. I just used Tailwind to make it a little bit responsive and clear but in terms of styling we could probably do better.
- I did write some basic tests on the frontend and on the smart contract part. I could have probably written more but this is not my strength.
- There is maybe too much logic inside the riddle-form component. But I thought it was probably not the most interesting thing to showcase.

Overall it was an interesting project to do. And I learn new things while doing it.

Let me know if something is unclear or you need more explanation on my side.

Thanks in advance,

Sylvain