بسم تعالی
گزارش پروژه درس رمز ارز

سید علی حسینی نسب ـ۴۰۱۲۴۳۱۳۳

محمد حسین کریمی ـ ۹۹۲۴۳۰۵۹

در ابتدا فایل StockMarket را توضیح میدهیم؛

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";
import "contracts/StockToken.sol";

/// @title Stock Market contract acting as a factory and controller
contract StockMarket is Ownable {
    struct Stock {
        string name;
        address token;
        uint256 lastPrice; // scaled price, e.g., with 8 decimals
        uint256 lastUpdated;
    }
```

برای تعیین نسخه Solidity مورد استفاده در این کد اولین خط نوشته شده است. این خط یعنی این قرارداد با نسخه 0.8.0 یا بالاتر (تا قبل از نسخه ناسازگار بعدی) کامپایل می‌شود.
در ادامه نیز قرارداد Ownable از OpenZeppelin برای کنترل مالکیت استفاده شده است. با این کار، می‌توان فقط به صاحب قرارداد اجازه دسترسی به توابع خاص را داد.
در آخرین خط importنیز فایل StockToken.sol که باید شامل توکنی باشد که نماینده یک سهم (stock) است وارد شده است.
پس از این تنظیمات اولیه حالا به سراغ خود کد میرویم؛ در اولین خط بعد از فراخوانی ها تعریف قرارداد اصلی به نام StockMarket که از Ownable ارث بری می‌کند را داریم. یعنی تابع onlyOwner برای کنترل دسترسی دارد. در ادامه نیز تعریف یک struct به نام Stock داریم که شامل:
name: نام کامل سهم.
token: آدرس قرارداد توکن ERC20 مربوط به این سهم.
lastPrice: آخرین قیمت سهم (مقیاس یافته با ۸ رقم اعشار).
lastUpdated: زمان آخرین به روزرسانی قیمت (به ثانیه از ابتدای epoch).

```
mapping(string => Stock) public stocks; // symbol => Stock
mapping(string => bool) public registered;

event StockAdded(string symbol, address tokenAddress);
event PriceUpdated(string symbol, uint256 price, uint256 timestamp);

constructor() Ownable(msg.sender) {}
```

درابتدای تصویر ۲ مپ داریم؛
stocks: نگهداری اطلاعات هر سهم با کلید symbol مثل نماد apple در بازار سهام .
registered: بررسی اینکه آیا یک سهم با نماد خاص قبلاً ثبت شده یا نه.

۲ ایونت نیز ایجاد شده اند که هنگام ثبت سهم جدید یا به‌روزرسانی قیمت منتشر می‌شوند. این ایونت ها برای تعامل با رابط‌های کاربری و ابزارها مثل Etherscan کاربرد دارند.

در خط پایانی تصویر نیز سازنده قرارداد که کنترل آن را به آدرسی که قرارداد را مستقر می‌کند می‌دهد.

```
/// @notice Add a new stock and deploy its ERC20 token
function addStock(string calldata symbol, string calldata name) external onlyOwner {
    require(!registered[symbol], "Stock already registered");

    StockToken token = new StockToken(name, symbol, address(this));
    stocks[symbol] = Stock({
        name: name,
        token: address(token),
        lastPrice: 0,
        lastUpdated: 0
    });

    registered[symbol] = true;
    emit StockAdded(symbol, address(token));
}
```

این تابعی است که فقط مالک می‌تواند آن را فراخوانی کند. بررسی می‌کند که سهم با نماد مشخص قبلاً ثبت نشده باشد.
در ادامه نیز یک قرارداد جدید از نوع StockToken برای این سهم خاص ایجاد شده است. آدرس قرارداد بازار را به توکن می‌دهد تا تنها این قرارداد بتواند mint و burn کند. در خطوط پایین تر آن نیز اطلاعات سهم جدید در ساختار stocks ذخیره می شود.
در نهایت نیز سهم به عنوان ثبت شده علامت گذاری میشود و ایونت به بیرون ارسال میشود.

```
/// @notice Update price manually (placeholder, to be replaced with Chainlink integration)
function updatePrice(string calldata symbol, uint256 newPrice) external onlyOwner {
    require(registered[symbol], "Stock not found");
    stocks[symbol].lastPrice = newPrice;
    stocks[symbol].lastUpdated = block.timestamp;
    emit PriceUpdated(symbol, newPrice, block.timestamp);
}
```

این تابع برای به روزرسانی قیمت سهم به صورت دستی است.
در این تابع برای تغییر قیمت سهم به‌صورت دستی از Oracle استفاده میشود که در فایل مربوط به خود آن را مشاهده میکنیم. در انتهای تابع نیز مقداردهی به قیمت جدید و ثبت زمان و ارسال ایونت را داریم.

```
/// @notice Buy shares using ETH (example logic, no actual pricing calc)
function buyStock(string calldata symbol, uint256 amount) external payable {
    require(registered[symbol], "Invalid stock");

    Stock storage stock = stocks[symbol];
    require(block.timestamp - stock.lastUpdated < 1 hours, "Stale price");

    uint256 cost = (stock.lastPrice * amount) / 1e8;
    require(msg.value >= cost, "Insufficient payment");

    StockToken token = StockToken(stock.token);
    token.mint(msg.sender, amount);
}
```

در این تابع خرید سهم با ETH پیاده سازی شده است.
ابتدا بررسی می شود که سهم موجود است سپس بررسی می کند که قیمت به روزرسانی شده باشد و قدیمی نباشد (حداکثر ۱ ساعت قبل).

محاسبه هزینه خرید به مقیاس 1e8 (برای مثال اگر قیمت 25000 = 0.00025 ETH باشد، و 100 سهم خریداری شود، هزینه برابر 0.025 ETH میشود.)
در انتها نیز صدور توکن به خریدار بر اساس تعداد خواسته شده اطلاع داده میشود.

```
    /// @notice Sell shares back (burn token, refund ETH for example)
    function sellStock(string calldata symbol, uint256 amount) external {
        require(registered[symbol], "Invalid stock");

        Stock storage stock = stocks[symbol];
        require(block.timestamp - stock.lastUpdated < 1 hours, "Stale price");

        uint256 payout = (stock.lastPrice * amount) / 1e8;
        require(address(this).balance >= payout, "Contract lacks funds");

        StockToken token = StockToken(stock.token);
        token.burn(msg.sender, amount);
        payable(msg.sender).transfer(payout);
    }

    /// @notice Fallback to receive ETH
    receive() external payable {}
}
```

در این تابع فروش سهم و دریافت ETH انجام میشود. ابتدا بررسی می شود سهم موجود است یا خیر.
در ادامه اعتبار قیمت بررسی میشود که قدیمی نباشد و محاسبه مبلغ پرداختی و بررسی اینکه قرارداد موجودی کافی دارد انجام میشود.
در نهایت نیز سوزاندن توکن های سهم از حساب فروشنده و انتقال ETH معادل قیمت سهم به آن را داریم.

در خط انتهایی نیز تابع receive() امکان می‌دهد تا قرارداد بدون داده (data ETH) دریافت کند. به عنوان مثال برای شارژ کردن موجودی قرارداد توسط مالک یا کاربران.

حالا به سراغ فایل دوم میرویم. در فایل StockToken قرارداد یک توکن ERC-20 قابل مدیریت است که توسط قرارداد StockMarket (فایل اول) صادر می شود. تنها قرارداد StockMarket مجاز است تا توکن‌ها را ضرب (mint) یا نابود (burn) کند. این کنترل دسترسی با استفاده از modifier تعریف می‌شود.

حالا کد را خط به خط توضیح میدهیم؛

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

/// @title ERC-20 Token representing stock shares
contract StockToken is ERC20 {
    address public market;

    modifier onlyMarket() {
        require(msg.sender == market, "Only StockMarket can call");
        _;
    }
```

مانند فایل اول نسخه تعیین شده است و پس از آن یک import داریم که استاندارد ERC20 از کتابخانه OpenZeppelin که شامل توابع پایه مانند burn، mint، transfer و balanceOf است را به کد میدهد.

در ادامه نیز قرارداد StockToken که از ERC20 ارث بری می‌کند تعریف می شود . با این کار تمام توابع استاندارد توکن را به ارث می‌برد.
در داخل این قرارداد ،آدرس قرارداد بازار سهام (StockMarket) که مجاز به ضرب یا سوزاندن توکن هاست قرار میگیرد .

در انتهای تصویر نیز یک modifier قرار دارد که فقط به قرارداد StockMarket اجازه دسترسی به توابع مشخصی را می‌دهد.

```
constructor(string memory name, string memory symbol, address _market) ERC20(name, symbol) {
    market = _market;
}

function mint(address to, uint256 amount) external onlyMarket {
    _mint(to, amount);
}

function burn(address from, uint256 amount) external onlyMarket {
    _burn(from, amount);
}
```

در ابتدا یک constructor قرار دارد که در زمان ساخت این توکن، نام، نماد(symbol) ، و آدرس بازار را تنظیم می‌کند. تابعی که فقط یک بار هنگام استقرار قرارداد اجرا می‌شود و شامل پارامتر های زیر است؛
name: نام توکن
symbol: نماد توکن
_market: آدرس قرارداد بازار.
فراخوانی سازنده ERC20 و ثبت نام و نماد توکن.

تابع mint توسط StockMarket فراخوانی می شود و برای صدور توکن است .مثلاً هنگام خرید سهام توسط کاربر.
_mint(to, amount) :تابع داخلی استاندارد ERC20 که توکن جدید به آدرس to با مقدار amount اضافه می‌کند.

تابعburn نیز توسط StockMarket فراخوانی می شود و برای سوزاندن توکن‌ها است. مثلاً هنگام فروش سهم.
_burn(from, amount):تابع داخلی ERC20 که توکن‌های آدرس from را به مقدار amount می‌سوزاند .

حالا به سراغ فایل سوم میرویم و خط به خط آن را توضیح میدهیم؛

```
pragma solidity ^0.8.0;

import "@chainlink/contracts/src/v0.8/ChainlinkClient.sol";

contract Oracle {
    address public owner;
    mapping(bytes32 => bool) public pendingRequests;
    event OracleRequest(
        bytes32 indexed specId,
        address requester,
        bytes32 requestId,
        uint256 payment,
        address callbackAddr,
        bytes4 callbackFunctionId,
        uint256 cancelExpiration,
        uint256 dataVersion,
        bytes data
    );
```

مانند فایل قبل نسخه تعیین شده است و پس از آن یک import داریم که کتابخانه ChainlinkClient را فراخوانی میکند.

در ادامه نیز Ownerتعریف شده است که آدرس صاحب این قرارداد است و فقط او می تواند درخواست ها را پاسخ دهد.

در mapping نگه داری وضعیت درخواست هایی که هنوز پاسخ داده نشده‌اند با کلید requestId است.

در ادامه یک ایونت داریم که مشخصات زیر را دارد و زمانی emit می شود که درخواست اوراکل ارسال می‌گردد:

specId: مشخص کننده نوع عملیات یا api در خارج.
Requester: آدرس درخواست دهنده.
requestId: شناسه یکتای درخواست.
payment: مقدار پرداخت شده .
callbackAddr: آدرس قرارداد مقصد برای پاسخ.
callbackFunctionId: تابع مقصد برای بازگرداندن نتیجه.
cancelExpiration: مهلت کنسل شدن درخواست.
dataVersion: نسخه داده.
data: داده‌ی خام ارسال‌شده.

```
event OracleResponse(bytes32 indexed requestId);

modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}

constructor() {
    owner = msg.sender;
}
```

OracleResponse نیز هنگام پاسخ به یک درخواست منتشر می شود تا ثبت شود که پاسخ ارسال شده است.
Modifier نیز محدود کردن دسترسی فقط به مالک قرارداد را فراهم میکند و بررسی می‌کند که فقط owner مجاز به اجرای تابع است.
در نهایت نیز Constructor تنظیم آدرس ownerبا آدرس کسی که قرارداد را مستقر کرده است.

```
function oracleRequest(
    address sender,
    uint256 payment,
    bytes32 specId,
    address callbackAddr,
    bytes4 callbackFunctionId,
    uint256 nonce,
    uint256 dataVersion,
    bytes calldata data
) external returns (bytes32 requestId) {
    requestId = keccak256(abi.encodePacked(sender, nonce));
    pendingRequests[requestId] = true;

    emit OracleRequest(
        specId,
        sender,
        requestId,
        payment,
        callbackAddr,
        callbackFunctionId,
        block.timestamp + 5 minutes,
        dataVersion,
        data
    );
}
```

تابع oracleRequest توسط کاربر یا قرارداد دیگر صدا زده می‌شود تا یک درخواست اوراکل ارسال کند.
سپس ایونت OracleRequest منتشر می‌شود. ورودی‌های تابع به شرح زیر اند؛

sender: آدرس درخواست‌دهنده.
payment: پرداخت مورد نظر
specId: شناسه‌ای برای نوع داده/عملیات بیرونی.
callbackAddr: آدرس قرارداد دریافت‌کننده پاسخ.
callbackFunctionId: تابعی از آن قرارداد برای دریافت پاسخ.
Nonce: عدد ترتیبی (برای جلوگیری از تکرار درخواست)
dataVersion: نسخه داده.
data: بایت داده‌های مربوط به درخواست

در ادامه تولید یک شناسه یکتا requestId با ترکیب sender و nonce را داریم و ثبت درخواست به عنوان در حال انتظار.

درنهایت ارسال رویداد OracleRequest به بیرون را داریم و cancelExpiration که برابر با ۵ دقیقه بعد از زمان فعلی تنظیم شده است.

```
function fulfillOracleRequest(
    bytes32 requestId,
    // uint256 payment,
    address callbackAddr,
    bytes4 callbackFunctionId,
    // uint256 expiration,
    bytes calldata data
) external onlyOwner returns (bool) {
    require(pendingRequests[requestId], "Request not found");
    delete pendingRequests[requestId];

    (bool success, ) = callbackAddr.call{value: 0}(
        abi.encodeWithSelector(callbackFunctionId, requestId, data)
    );

    require(success, "Callback failed");

    emit OracleResponse(requestId);
    return true;
}
```

تابع fulfillOracleRequest توسط مالک قرارداد برای پاسخ به درخواست صدا زده می شود که شامل ورودی های زیر است؛

requestId: شناسه یکتای درخواست.
callbackAddr: آدرس قرارداد مقصد برای ارسال پاسخ.
callbackFunctionId: تابعی در آن قرارداد که باید صدا زده شود.
Data: داده‌ای که باید به عنوان پاسخ ارسال شود.

Requireنیز بررسی می‌کند که این درخواست قبلاً ثبت شده باشد.

delete pendingRequests نیز حذف درخواست از لیست درخواست های معلق را انجام میدهد.

callbackAddr نیز برای فراخوانی تابع callbackFunctionId از قرارداد مقصد به صورت داینامیک، و ارسال requestIdو data به آن استفاده میشود .

require(success, "Callback failed");نیز بررسی موفقیت آمیز بودنcall را انجام میدهد

و در نهایت انتشار ایونت OracleResponse و بازگرداندنtrue .