**SupplyChain.sol :**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract SupplyChain {

    enum Status { Created, InTransit, Delivered }

    struct Product {

        uint id;

        string name;

        string description;

        uint price;

        address manufacturer;

        Status status;

        uint timestamp;

    }

    struct Order {

        uint orderId;

        uint productId;

        uint quantity;

        uint totalPrice;

        string shippingAddress;

        Status status;

        uint timestamp;

        address customer;

        address retailer;

    }

    struct OrderTimeline {

        uint timestamp;

        string status;

        string location;

        address updatedBy;

        bool isManufacturerUpdate;

    }

    mapping(uint => Product) public products;

    mapping(address => Order[]) public customerOrders;

    mapping(address => Order[]) public retailerOrders;

    mapping(address => mapping(uint => uint)) public retailerInventory;

    mapping(address => bool) public isRetailer;

    mapping(uint => OrderTimeline[]) public manufacturerToRetailerTimeline;

    mapping(uint => OrderTimeline[]) public retailerToCustomerTimeline;

    mapping(uint => address) public orderToRetailer;

    mapping(uint => uint[]) public productOrders;

    mapping(uint => Order) public ordersById;

    mapping(uint => address) public orderToRetailerMapping;

    address public immutable manufacturer;

    uint public productCount;

    uint public orderCount;

    address[] public customers;

    address[] public retailers;

    event ProductCreated(uint id, string name, uint price);

    event RetailerOrderPlaced(uint orderId, uint productId, address retailer, uint quantity);

    event CustomerOrderPlaced(uint orderId, uint productId, address customer, address retailer);

    event OrderStatusUpdated(uint indexed orderId, Status status, address indexed updatedBy);

    event RetailerAdded(address retailer);

    event InventoryUpdated(address retailer, uint productId, uint quantity);

    constructor() {

        manufacturer = msg.sender;

    }

    modifier onlyManufacturer() {

        require(msg.sender == manufacturer, "Only manufacturer");

        \_;

    }

    modifier onlyRetailer() {

        require(isRetailer[msg.sender], "Only retailer");

        \_;

    }

    function addRetailer(address \_retailer) public onlyManufacturer {

        require(!isRetailer[\_retailer], "Already retailer");

        retailers.push(\_retailer);

        isRetailer[\_retailer] = true;

        emit RetailerAdded(\_retailer);

    }

    function createProduct(string memory \_name, string memory \_description, uint \_price) public onlyManufacturer {

        productCount++;

        products[productCount] = Product({

            id: productCount,

            name: \_name,

            description: \_description,

            price: \_price,

            manufacturer: msg.sender,

            status: Status.Created,

            timestamp: block.timestamp

        });

        emit ProductCreated(productCount, \_name, \_price);

    }

    function placeRetailerOrder(uint \_productId, uint \_quantity) public payable onlyRetailer {

        Product storage product = products[\_productId];

        uint totalPrice = product.price \* \_quantity;

        orderCount++;

        uint orderId = orderCount;

        Order memory newOrder = Order({

            orderId: orderId,

            productId: \_productId,

            quantity: \_quantity,

            totalPrice: totalPrice,

            shippingAddress: "",

            status: Status.Created,

            timestamp: block.timestamp,

            customer: msg.sender,

            retailer: msg.sender

        });

        retailerOrders[msg.sender].push(newOrder);

        ordersById[orderId] = newOrder;

        orderToRetailerMapping[orderId] = msg.sender;

        productOrders[\_productId].push(orderId);

        payable(manufacturer).transfer(totalPrice);

        emit RetailerOrderPlaced(orderId, \_productId, msg.sender, \_quantity);

        manufacturerToRetailerTimeline[orderId].push(OrderTimeline(

            block.timestamp,

            "ORDER\_PLACED",

            "Manufacturer warehouse",

            msg.sender,

            false

        ));

    }

    function placeCustomerOrder(uint \_productId, uint \_quantity, string memory \_shippingAddress, address \_retailer) public payable {

        require(isRetailer[\_retailer], "Invalid retailer");

        uint retailPrice = products[\_productId].price \* 120 / 100;

        uint totalPrice = retailPrice \* \_quantity;

        if (customerOrders[msg.sender].length == 0) {

            customers.push(msg.sender);

        }

        orderCount++;

        uint orderId = orderCount;

        Order memory newOrder = Order({

            orderId: orderId,

            productId: \_productId,

            quantity: \_quantity,

            totalPrice: totalPrice,

            shippingAddress: \_shippingAddress,

            status: Status.Created,

            timestamp: block.timestamp,

            customer: msg.sender,

            retailer: \_retailer

        });

        customerOrders[msg.sender].push(newOrder);

        ordersById[orderId] = newOrder;

        orderToRetailer[orderId] = \_retailer;

        payable(\_retailer).transfer(totalPrice \* 20 / 100);

        payable(manufacturer).transfer(totalPrice \* 80 / 100);

        retailerInventory[\_retailer][\_productId] -= \_quantity;

        emit CustomerOrderPlaced(orderId, \_productId, msg.sender, \_retailer);

        retailerToCustomerTimeline[orderId].push(OrderTimeline(

            block.timestamp,

            "ORDER\_RECEIVED",

            \_shippingAddress,

            \_retailer,

            false

        ));

    }

    function updateManufacturerShipment(

    uint orderId,

    string memory status,

    string memory location

) public onlyManufacturer {

    Order storage order = ordersById[orderId];

    require(order.orderId == orderId, "Order not found");

    // Convert string status to enum

    if (keccak256(bytes(status)) == keccak256(bytes("IN\_TRANSIT"))) {

        order.status = Status.InTransit;

    } else if (keccak256(bytes(status)) == keccak256(bytes("DELIVERED"))) {

        order.status = Status.Delivered;

        retailerInventory[order.retailer][order.productId] += order.quantity;

    }

    manufacturerToRetailerTimeline[orderId].push(OrderTimeline(

        block.timestamp,

        status,

        location,

        msg.sender,

        true

    ));

    emit OrderStatusUpdated(orderId, order.status, msg.sender);

    emit InventoryUpdated(order.retailer, order.productId, retailerInventory[order.retailer][order.productId]);

}

    function updateRetailerShipment(

    uint orderId,

    string memory status,

    string memory trackingNumber

) public {

    Order storage order = ordersById[orderId];

    require(msg.sender == order.retailer, "Only retailer can update");

    // Convert string status to enum

    if (keccak256(bytes(status)) == keccak256(bytes("SHIPPED"))) {

        order.status = Status.InTransit;

    } else if (keccak256(bytes(status)) == keccak256(bytes("DELIVERED"))) {

        order.status = Status.Delivered;

    }

    retailerToCustomerTimeline[orderId].push(OrderTimeline(

        block.timestamp,

        status,

        trackingNumber,

        msg.sender,

        false

    ));

    emit OrderStatusUpdated(orderId, order.status, msg.sender);

}

    // View functions

    function getManufacturerUpdates(uint orderId) public view returns (OrderTimeline[] memory) {

        return manufacturerToRetailerTimeline[orderId];

    }

    function getRetailerUpdates(uint orderId) public view returns (OrderTimeline[] memory) {

        return retailerToCustomerTimeline[orderId];

    }

    function getOrderRetailer(uint orderId) public view returns (address) {

        return orderToRetailer[orderId];

    }

    function getProduct(uint \_id) public view returns (

        uint, string memory, string memory, uint, address, Status, uint

    ) {

        Product memory p = products[\_id];

        return (p.id, p.name, p.description, p.price, p.manufacturer, p.status, p.timestamp);

    }

    function getCustomerOrders(address \_customer) public view returns (Order[] memory) {

        return customerOrders[\_customer];

    }

    function getRetailerOrders(address \_retailer) public view returns (Order[] memory) {

        return retailerOrders[\_retailer];

    }

    function getRetailerInventory(address \_retailer, uint \_productId) public view returns (uint) {

        return retailerInventory[\_retailer][\_productId];

    }

    function getAllCustomers() public view returns (address[] memory) {

        return customers;

    }

    function getAllRetailers() public view returns (address[] memory) {

        return retailers;

    }

    function getPendingOrders() public view returns (Order[] memory) {

        uint count = 0;

        for (uint i = 0; i < customers.length; i++) {

            Order[] storage orders = customerOrders[customers[i]];

            for (uint j = 0; j < orders.length; j++) {

                if (orders[j].status == Status.Created) {

                    count++;

                }

            }

        }

        Order[] memory result = new Order[](count);

        uint index = 0;

        for (uint i = 0; i < customers.length; i++) {

            Order[] storage orders = customerOrders[customers[i]];

            for (uint j = 0; j < orders.length; j++) {

                if (orders[j].status == Status.Created) {

                    result[index] = orders[j];

                    index++;

                }

            }

        }

        return result;

    }

    function getPendingRetailerOrders() public view returns (Order[] memory) {

        uint count = 0;

        for (uint i = 0; i < retailers.length; i++) {

            Order[] storage orders = retailerOrders[retailers[i]];

            for (uint j = 0; j < orders.length; j++) {

                if (orders[j].status == Status.Created) {

                    count++;

                }

            }

        }

        Order[] memory result = new Order[](count);

        uint index = 0;

        for (uint i = 0; i < retailers.length; i++) {

            Order[] storage orders = retailerOrders[retailers[i]];

            for (uint j = 0; j < orders.length; j++) {

                if (orders[j].status == Status.Created) {

                    result[index] = orders[j];

                    index++;

                }

            }

        }

        return result;

    }

}