2025 - 3팀 블록체인 최종 보고서

AutoRent

	학번	이름	역할
팀장	2020810010	김수현	보고서 작성, 웹, 자동차 대여
팀원	2019245042	김민성	웹, 렌터카 정보 관리
	2020810026	남규민	웹, 자동차 대여
	2020810034	박준기	보증금 반환
	2020810072	진영호	회원 계정 관리 및 인증

제출일 : **2025.06.04**

목　　차

1. 개발시스템 명세

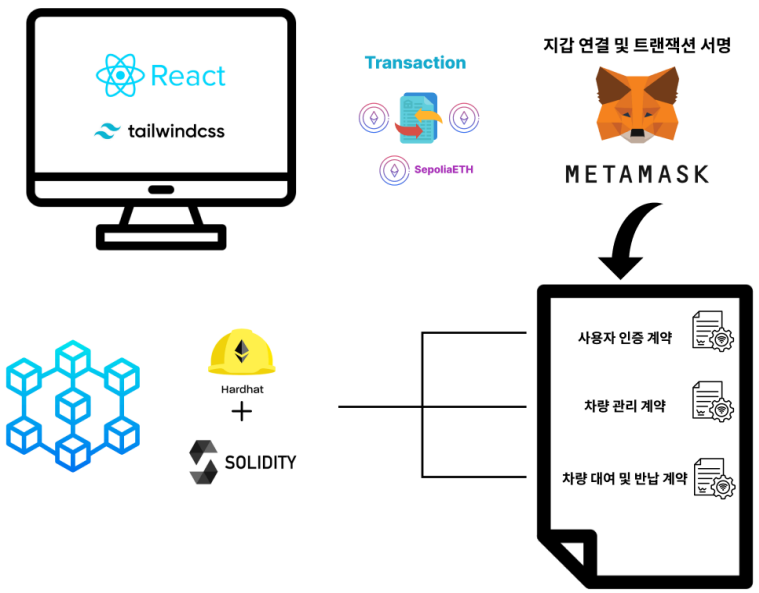
1.1 개발시스템의 목적

스마트 컨트랙트를 통해 렌트카 대여 과정을 자동화하고, 중개자 없이 신뢰할 수 있는 계약 체결과 실행이 가능하며, 대여 조건, 결제, 반납 등의 절차를 블록체인 상에서 자동으로 처리한다. 이를 통해 거래의 투명성을 확보한다. 최종종적으로 효율적이고 안전한 렌트카 대여 서비스를 제공하는 것이 개발의 목적이다.

1.2 개발시스템의 예상 사용자 및 사용자가 느끼는 예상 효용

예상 사용자	예상 효용
(차량)공급자	1. 스마트 계약 기반으로 대여 조건 자동 이행 2. 계약 위조 혹은 미납 방지 3. 보증금 자동 반환으로 분쟁 최소화 4. 차량 위치, 주행 거리 상태 등 실시간 추적, 블록체인에 저장
수요자	1. 중계 수수료 최소화로 비용 절감 2. 스마트 계약으로 계약의 신뢰성 확보 3. 보증금 자동 반환으로 분쟁 최소화

1.3 개발시스템 개요도

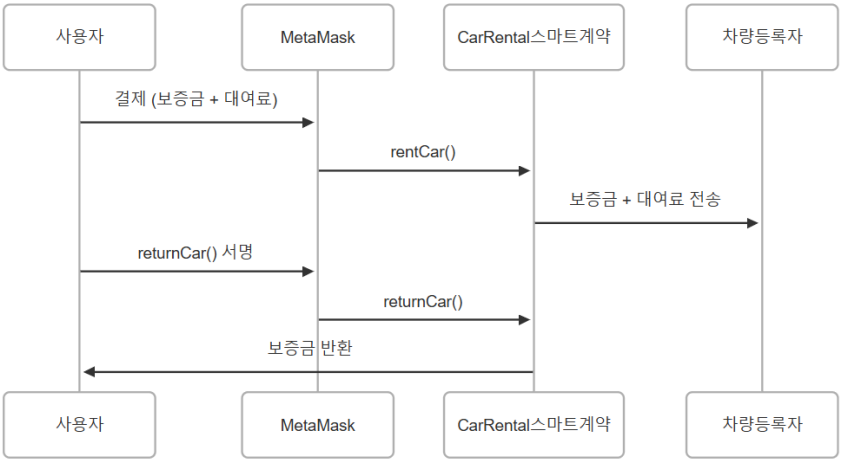


본 시스템은 React기반으로 만든 사용자 UI에서 MetaMask를 사용하여 SepoliaEth 테스트넷을 사용한다. MetaMask를 통해 지갑 연결 및 트랜잭션 서명을 수행하며 hardhat플랫폼을 사용하여 사용자 인증, 차량 관리, 차량 대여 및 반납 등에 대한 sol계약을 SepiliaEth를 사용할 수 있는 테스트넷에 배포하였다. MetaMask를 통해 테스트넷에 배포된 sol계약에 접근할 수 있으며 사용자 인터페이스에서 기능을 수행하면 MetaMask에서 트랜잭션 요청 확인을 요구하는데 확인을 하면 Sepolia 테스트넷에 배포된 계약들에 접근하여 기능을 수행 할 수 있다.

1.4 개발시스템의 주요기능 및 개발된 내용 설명

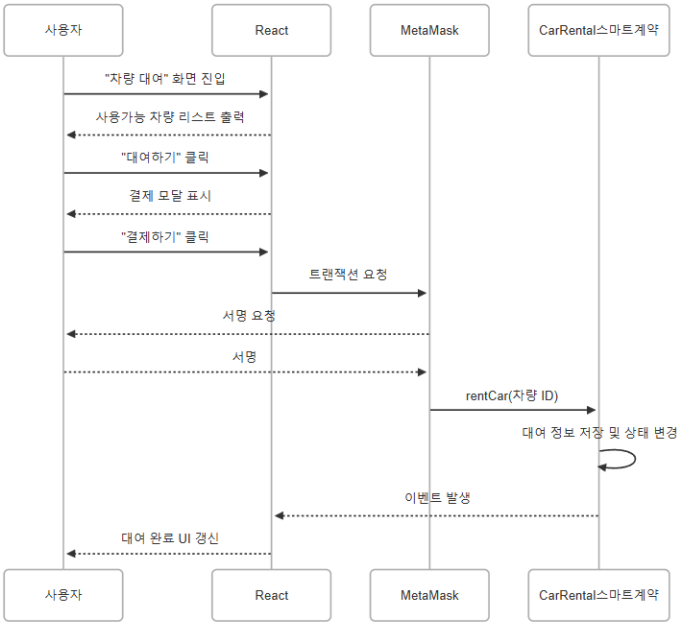
❖ 차량 대여 및 반납

➢ 보증금 + 대여료 지급 및 보증금 반환



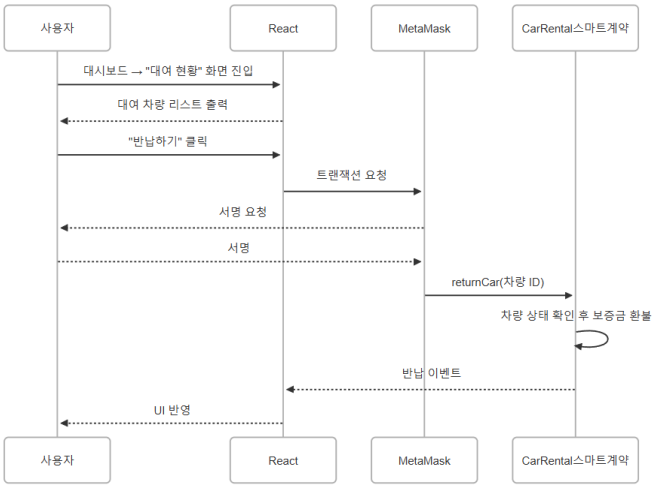
차량 대여 과정에서 사용자는 대여료와 보증금을 동시에 지불하게 되며, 이 트랜잭션은 MetaMask를 통해 실행된다. 스마트 계약 내부에서는 `rentCar` 함수 호출 시 `msg.value`를 이용해 지불된 금액을 확인하고, 해당 차량의 등록자 주소로 자동 송금한다. 보증금과 대여료는 구분 저장되며, 구조체 내 별도 필드로 기록된다. 반납 시에는 `returnCar` 함수가 호출되며, 해당 함수에서는 대여 상태와 시간을 기준으로 유효성을 검증한 후, 보증금을 대여자 주소로 반환하는 로직이 포함되어 있다. 반환 처리 역시 MetaMask를 통한 트랜잭션 서명 후 자동 전송된다. 스마트 계약에서는 이를 위해 `mapping(uint => Rental)` 구조를 사용하며, 차량 ID를 기준으로 각 대여 내역을 저장하고 추적한다. 현재 구현된 상태는 사용자의 지불 및 반환 흐름을 완전하게 처리할 수 있는 프로토타입 수준의 스마트 계약 로직이며, UI 연동 및 MetaMask 트랜잭션 처리까지 연계되어 안정적으로 작동한다.

➢ 차량 대여



사용자는 차량 대여 화면에서 리스트 중 원하는 차량을 선택하고 "대여하기" 버튼을 클릭함으로써 대여 요청을 시작한다. 이후 표시되는 결제 모달을 통해 금액을 확인하고 "결제하기" 버튼을 누르면 MetaMask를 통해 트랜잭션 요청이 이루어진다. 스마트 계약에서는 `rentCar(uint carId)` 함수가 호출되며, 입력된 `carId`를 기준으로 해당 차량의 상태를 확인하고, `Rental` 구조체에 대여자 주소, 시작 시간, 보증금, 대여료 등의 정보를 저장한다. 이때 차량의 상태는 `enum`을 통해 '대여 가능'에서 '대여 중'으로 변경되며, 이벤트가 발생하여 UI에 반영된다. 차량 정보 및 대여 상태 관리를 위해 `mapping(uint => Car)`와 `mapping(uint => Rental)` 두 가지 주요 맵핑이 사용된다. 현재 구현된 로직은 실제 사용자의 입력과 스마트 계약 간 상호작용이 완전히 연동되어 있으며, 정상적인 대여 흐름을 시뮬레이션할 수 있는 프로토타입 수준이다.

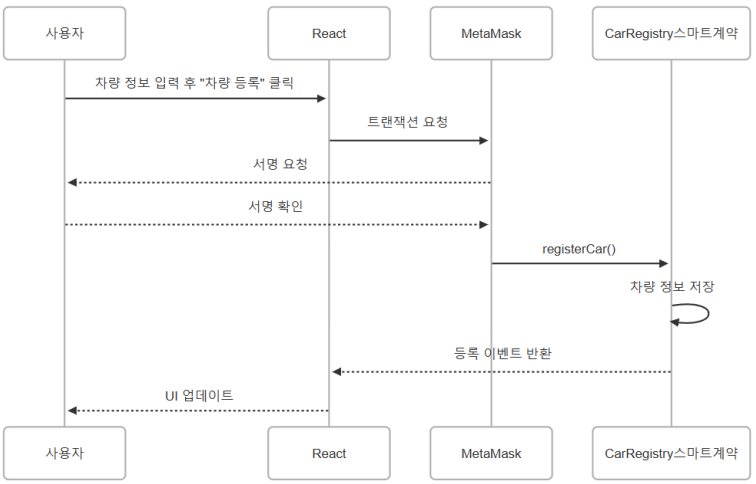
➢ 차량 반납



차량 반납은 대시보드의 "대여 현황" 화면에서 사용자가 직접 수행할 수 있다. "반납하기" 버튼 클릭 시 MetaMask를 통해 트랜잭션 요청이 발생하며, 스마트 계약의 `returnCar(uint carId)` 함수가 실행된다. 이 함수는 차량 ID를 기준으로 해당 대여 기록을 불러오고, 차량 상태가 '대여 중'인지 여부를 확인한 뒤 반납 처리를 진행한다. 반납 처리 시 차량 상태는 '반납 완료'로 변경되고, 보증금은 대여자 지갑 주소로 환불된다. 이를 위해 스마트 계약에서는 `require` 조건문을 통해 유효성 검증을 수행하고, 성공적인 반환 처리를 위해 이벤트를 발생시켜 프론트엔드에 상태 변화를 알린다. 현재 구현은 보증금 반환 로직까지 완성되어 있으며, 대여 정보 구조(`struct Rental`)와 트랜잭션 검증 절차가 모두 포함되어 있는 상태로, 전체 반납 프로세스는 프로토타입 단계에서 실제 운용 가능한 수준으로 구현되어 있다.

❖ 차량 관리

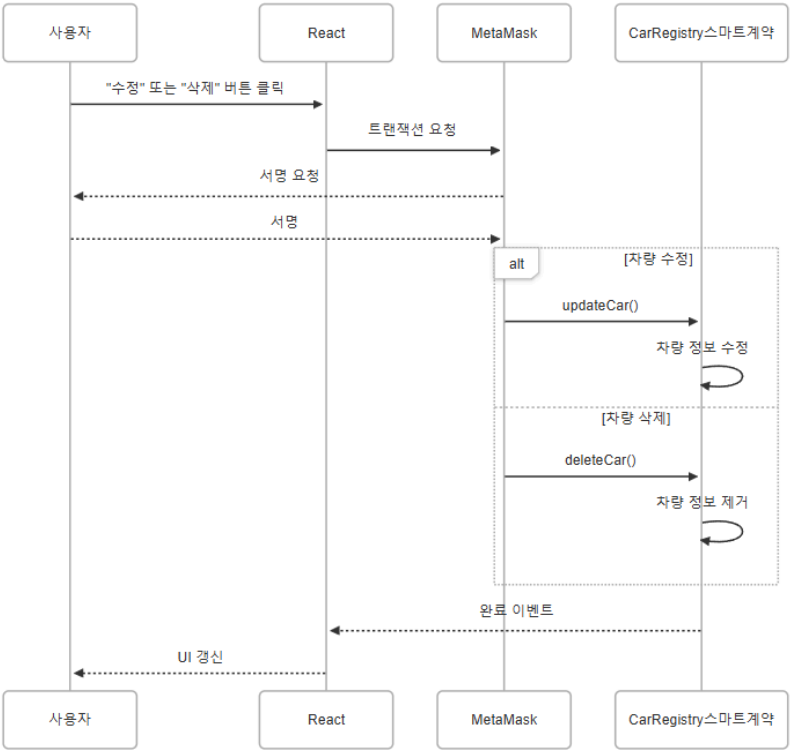
➢ 차량 등록



사용자는 대시보드 화면 내 "차량 등록하기" 메뉴에서 차량 정보를 입력하고 등록할 수 있다. 등록 화면은 번호판을 직접 입력하고 차량 모델, 대여 가능 지역, 일일 대여료를

선택하는 방식으로 구성되어 있으며, 모든 항목 입력 후 "차량 등록하기" 버튼을 클릭하면 MetaMask를 통해 트랜잭션 서명이 요청된다. 해당 트랜잭션을 승인하면 차량 정보가 스마트 계약에 기록되며, 이 과정은 사용자의 지갑 주소를 기준으로 온체인 상에서 처리된다. 스마트 계약에서는 차량 정보를 저장하기 위해 `struct Vehicle`을 정의하고 있으며, 각 사용자의 차량 리스트는 `mapping(address => Vehicle[])` 형태로 저장된다. 트랜잭션 성공 시, 차량 정보는 해당 사용자 지갑 주소에 종속된 배열에 추가되며, 별도의 이벤트를 통해 등록 완료 여부를 프론트엔드로 전달한다. 현재 구현은 Solidity 기반 스마트 계약과 Web3 연동으로 프로토타입 수준에서 완성되었으며, 실시간 등록 확인 및 UI 연동까지 구축이 완료된 상태다.

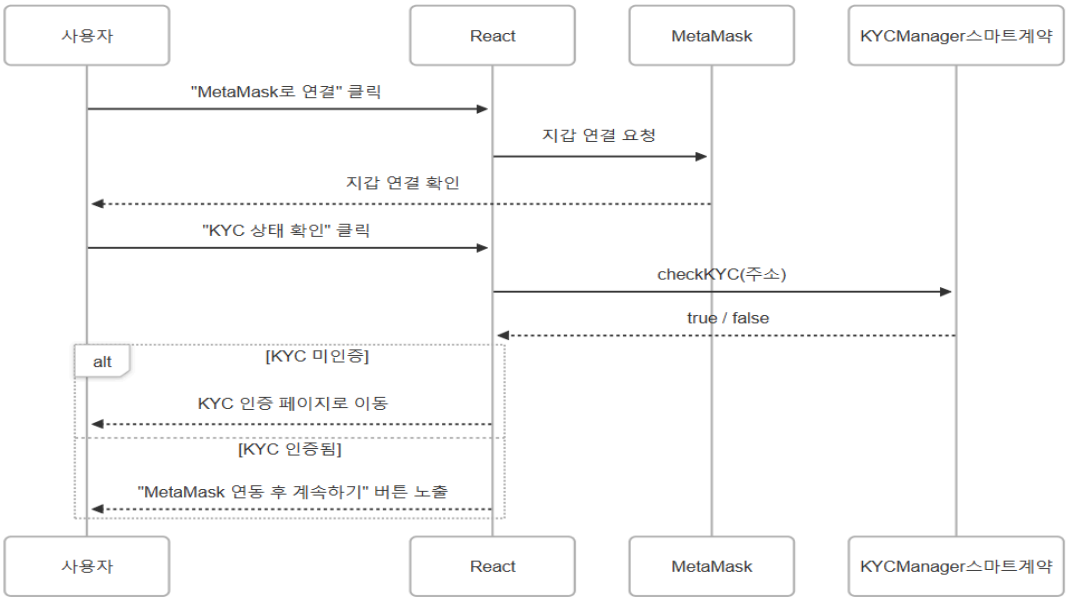
➢ 차량 수정 및 삭제



"등록차량" 메뉴에서는 사용자가 등록한 차량 목록을 확인하고, 각 차량의 정보를 수정하거나 삭제할 수 있다. 수정 기능은 차량 모델, 대여 지역, 일일 대여료 항목에 대해 변경이 필요한 값을 선택한 후 "수정" 버튼을 클릭하면 트랜잭션 요청이 MetaMask를 통해 발생하고, 승인 시 스마트 계약의 해당 차량 정보가 업데이트된다. 이를 위해 스마트 계약에는 특정 차량의 인덱스를 기준으로 정보를 갱신할 수 있도록 인자로 index 값을 받는 `updateVehicle` 함수가 구현되어 있으며, 내부에서는 사용자 지갑 주소에 종속된 배열 내의 데이터를 수정한다. 삭제 기능 또한 동일한 흐름으로 진행되며, 차량의 인덱스를 기반으로 해당 데이터를 배열에서 제거하는 `deleteVehicle` 함수가 정의되어 있다. 트랜잭션 승인 후에는 스마트 계약 내부의 데이터가 즉시 갱신되며, 등록/수정/삭제 동작 모두에서 이벤트가 발생되어 프론트엔드에서 UI 갱신이 가능하도록

처리되어 있다. 현재 구현 수준은 기능 단위별 트랜잭션 흐름, 스마트 계약 로직, UI 인터페이스가 연동된 형태로 안정적으로 동작하는 상태다.

❖ 사용자 인증



스마트 계약 기반의 P2P 렌터카 자동 대여 시스템은 사용자 인증을 위해 KYC 기능을 도입하였으며, 이 흐름은 프론트엔드에서 MetaMask 지갑 연결을 시작으로 진행된다. 사용자는 지갑 주소를 입력한 후 KYC 상태를 확인할 수 있으며, 인증 여부에 따라 KYC 인증 화면 또는 서비스 선택 화면으로 자동 전환된다. 이때 인증이 완료되지 않은 사용자는 KYC 페이지에서 신원 인증을 수행하게 되고, 인증이 완료되면 자동으로 서비스 이용이 가능하다. 인증이 완료된 지갑의 경우에는 MetaMask와의 연동을 통해 로그인과 동시에 서비스에 접근할 수 있도록 구성되어 있다. KYC 구현은 현재 외부 인증기관 API 연동 없이, 내부에서 지갑 주소의 인증 상태를 판단하는 수준의 프로토타입으로 설계되었으며, 실제 인증 정보는 오프체인에서 처리되고 스마트 계약에서는 단순히 인증 여부(boolean)만 상태값으로 관리한다. 이로 인해 온체인에는 민감한 정보가 저장되지 않으며, 인증 상태 변경 시 스마트 계약의 이벤트를 통해 상태가 반영된다. 전체 인증 및 연결 흐름은 스마트 계약과의 연동을 기반으로 하여, 지갑 주소 단위로 사용자의 접근 권한을 제어하고 있다.

1.5 개발시스템의 범위

❖ 개발 부분

1. 스마트 컨트랙트(자동차 대여 및 반납, 보증금 반환)
2. 렌트카 관리 시스템
3. 웹 프론트엔드

❖ 외부 프로그램

1. 메타마스크
2. 하드햇

1.6 유사시스템의 존재 여부와 기존 유사시스템과의 장단점 비교

항목	쏘카	스마트 컨트랙트 기반 자동 렌트카 서비스
계약 방식	오프라인 또는 앱 수동 계약	블록체인 스마트 컨트랙트 자동 계약
보증금 반환	수작업 확인 후 반환	자동 상태 확인 후 즉시 반환
데이터 관리	중앙 서버	블록체인 분산 저장
신뢰성	사업자 신뢰 기반	코드 기반 신뢰
투명성	비공개 계약 조건	공개 스마트 컨트랙트

1.7 상호평가 (각 팀원이 다른 팀원을 100점 만점 기준평가한 평가표 (팀원별 평균 포함)

	학번	이름	김수현	김민성	남규민	박준기	진영호
팀장	2020810010	김수현		100	100	100	100
팀원	2019245042	김민성	100		98	98	96
	2020810026	남규민	100	100		100	100
	2020810034	박준기	100	100	100		100
	2020810072	진영호	100	100	100	100	
평균	/	/	**100**	**100**	**99.5**	**99.5**	**99**

2. 부록

❖ Github

https://github.com/jjangsh01/RentCarEth

❖ 프로그램소스 (스마트계약 sol코드)

➢ CarRegistry.sol

```
pragma solidity ^0.8.19;
/// @title 차량 등록 및 관리 스마트 컨트랙트
contract CarRegistry {

    // 차량 상태를 나타내는 열거형: 사용 가능, 대여 중, 정비 중
    enum CarStatus { Available, Rented, Maintenance }

    // 차량 정보 구조체
    struct Car {
        string plateNumber;     // 차량 번호판
        string model;           // 차량 모델명
        string location;        // 차량 위치
        uint256 pricePerDay;    // 일일 대여료
        uint8 status;           // 차량 상태 (enum을 uint8로 저장)
        address renter;         // 현재 대여자 주소
        address owner;          // 차량 등록자 주소
    }
    // 차량 번호판 => 차량 정보 매핑
    mapping(string => Car) public cars;

    // 전체 차량 번호판 목록 (조회용)
    string[] public carPlates;

    // 차량이 존재하는지 확인하는 modifier
    modifier carExists(string memory plateNumber) {
        require(bytes(cars[plateNumber].plateNumber).length != 0, "Car does not exist");
        _;
    }

    // 차량 소유자인지 확인하는 modifier
    modifier onlyOwner(string memory plateNumber) {
        require(cars[plateNumber].owner == msg.sender, "Only car owner can modify");
        _;
    }
    //  차량 등록 함수
    function addCar(
        string memory plateNumber,
        string memory model,
```

```
        string memory location,
        uint256 pricePerDay
    ) external {
        // 중복 등록 방지
        require(bytes(cars[plateNumber].plateNumber).length == 0, "Car already exists");

        // 차량 정보 저장
        cars[plateNumber] = Car({
            plateNumber: plateNumber,
            model: model,
            location: location,
            pricePerDay: pricePerDay,
            status: 0,  // 기본 상태: Available
            renter: address(0), // 초기 대여자 없음
            owner: msg.sender   // 현재 호출자가 소유자
        });

        // 차량 번호판 목록에 추가
        carPlates.push(plateNumber);
    }
    // 🛠️ 차량 정보 수정 (등록자만 가능)
    function updateCar(
        string memory plateNumber,
        string memory model,
        string memory location,
        uint256 pricePerDay
    ) external carExists(plateNumber) onlyOwner(plateNumber) {
        Car storage car = cars[plateNumber];
        // 빈 값이 아닌 경우만 수정
        if (bytes(model).length > 0) car.model = model;
        if (bytes(location).length > 0) car.location = location;
        if (pricePerDay > 0) car.pricePerDay = pricePerDay;
    }

    //  차량 삭제 (등록자만 가능)
    function deleteCar(string memory plateNumber) external carExists(plateNumber)
onlyOwner(plateNumber) {
        // 차량 정보 삭제
        delete cars[plateNumber];

        // 차량 번호판 배열에서 제거 (swap & pop 방식)
        for (uint i = 0; i < carPlates.length; i++) {
            if (keccak256(bytes(carPlates[i])) == keccak256(bytes(plateNumber))) {
                carPlates[i] = carPlates[carPlates.length - 1];
                carPlates.pop();
                break;
            }
        }
    }

    // 전체 차량 번호판 목록 조회
```

```
    function getCarPlates() external view returns (string[] memory) {
        return carPlates;
    }

    // 특정 차량 정보 조회
    function getCar(string memory plateNumber) external view carExists(plateNumber)
returns (
        string memory, string memory, string memory, uint256, uint8, address, address
    ) {
        Car memory car = cars[plateNumber];
        return (
            car.plateNumber,
            car.model,
            car.location,
            car.pricePerDay,
            car.status,
            car.renter,
            car.owner
        );
    }

    // 차량 상태를 "대여 중"으로 설정하고 대여자 등록
    function setCarRented(string memory plateNumber, address renter) external
carExists(plateNumber) {
        cars[plateNumber].status = 1; // Rented
        cars[plateNumber].renter = renter;
    }

    // 차량 상태를 "사용 가능"으로 설정하고 대여자 초기화
    function setCarAvailable(string memory plateNumber) external
carExists(plateNumber) {
        cars[plateNumber].status = 0; // Available
        cars[plateNumber].renter = address(0);
    }
}
```

➢ CarRental.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "./interfaces/ICarRegistry.sol";

// KYC 검증 인터페이스
interface IKYCManager {
    function checkKYC(address user) external view returns (bool);
}
// 보증금 입금 및 환불을 위한 Vault 인터페이스
interface IRentalVault {
```

```
    function deposit() external payable;
    function refund(address to, uint256 amount) external;
}

/// @title 차량 대여 계약
contract CarRental {
    // 외부 의존 컨트랙트 (차량 등록소, KYC 관리자, 보증금 금고)
    ICarRegistry public immutable carRegistry;
    IKYCManager public immutable kycManager;
    IRentalVault public immutable vault;

    // 보증금 배수 (ex. 일일 요금 * 2)
    uint256 public constant DEPOSIT_MULTIPLIER = 2;

    // 차량 ID (hash) => 보증금 저장
    mapping(bytes32 => uint256) public deposits;

    // 차량 번호 => 대여 정보
    mapping(string => RentalInfo) public rentalRecords;

    // 차량 소유자 => 누적 수익
    mapping(address => uint256) public ownerRevenue;

    // 대여 정보 구조체
    struct RentalInfo {
        address renter;        // 대여자
        uint256 amountPaid;    // 총 지불 금액
        uint256 rentalFee;     // 대여 요금
        uint256 depositAmount; // 보증금
        uint256 timestamp;     // 대여 시작 시간
        bool returned;         // 반납 여부
    }

    // 생성자: 외부 컨트랙트 주소 설정
    constructor(
        address _carRegistry,
        address _kycManager,
        address _vault
    ) {
        carRegistry = ICarRegistry(_carRegistry);
        kycManager = IKYCManager(_kycManager);
        vault = IRentalVault(_vault);
    }
    // 차량 대여 함수
    function rentCar(string memory plateNumber) external payable {
        // KYC 검증
        require(kycManager.checkKYC(msg.sender), "KYC not approved");

        // 차량 ID 생성 (plateNumber 기반 해시)
        bytes32 carId = keccak256(abi.encodePacked(plateNumber));

        // 차량 정보 조회
        (
            string memory carPlate,
            , // model
            , // location
            uint256 pricePerDay,
            uint8 status,
            address renter,
```

```
            address owner
        ) = carRegistry.getCar(plateNumber);

        require(bytes(carPlate).length != 0, "Car not found");
        require(status == uint8(ICarRegistry.CarStatus.Available), "Car is not available");

        // 요금 계산
        uint256 rentalFee = pricePerDay;
        uint256 depositAmount = rentalFee * DEPOSIT_MULTIPLIER;
        uint256 totalRequired = rentalFee + depositAmount;

        require(msg.value >= totalRequired, "Insufficient payment");

        // 차량 소유자에게 대여료 송금
        (bool sent, ) = payable(owner).call{value: rentalFee}("");
        require(sent, "Transfer to owner failed");

        // 소유자 수익 기록
        ownerRevenue[owner] += rentalFee;

        // 보증금은 별도 Vault로 송금
        vault.deposit{value: depositAmount}();

        // 보증금 저장
        deposits[carId] = depositAmount;

        // 대여 정보 기록
        rentalRecords[plateNumber] = RentalInfo({
            renter: msg.sender,
            amountPaid: msg.value,
            rentalFee: rentalFee,
            depositAmount: depositAmount,
            timestamp: block.timestamp,
            returned: false
        });

        // 차량 상태를 Rented로 변경
        carRegistry.setCarRented(plateNumber, msg.sender);

        emit CarRented(plateNumber, msg.sender, rentalFee, depositAmount);
    }

    // 대여 완료 및 차량 반납 처리
    function completeRental(string memory plateNumber) external {
        bytes32 carId = keccak256(abi.encodePacked(plateNumber));

        // 차량 정보 조회
        (
            string memory carPlate,
            , , , // model, location, pricePerDay
            uint8 status,
            address renter,
            address owner
        ) = carRegistry.getCar(plateNumber);

        require(bytes(carPlate).length != 0, "Car not found");
        require(status == uint8(ICarRegistry.CarStatus.Rented), "Car is not currently
rented");
        require(renter == msg.sender, "You are not the renter");
```

```
    // 차량 상태를 다시 Available로 변경
    carRegistry.setCarAvailable(plateNumber);

    // 보증금 환불
    uint256 depositToRefund = deposits[carId];
    deposits[carId] = 0; // 중복 환불 방지

    if (depositToRefund > 0) {
        vault.refund(renter, depositToRefund);
    }

    // 대여 상태 변경
    rentalRecords[plateNumber].returned = true;

    emit CarReturned(plateNumber, renter, depositToRefund);
}

// 특정 차량의 보증금 확인
function getDeposit(string memory plateNumber) external view returns (uint256) {
    bytes32 carId = keccak256(abi.encodePacked(plateNumber));
    return deposits[carId];
}

// 간단한 대여 정보 확인
function getRentalInfo(string memory plateNumber)
    external
    view
    returns (address renter, uint256 amountPaid, uint256 timestamp, bool returned)
{
    RentalInfo memory info = rentalRecords[plateNumber];
    return (info.renter, info.amountPaid, info.timestamp, info.returned);
}

// 상세 대여 정보 확인
function getDetailedRentalInfo(string memory plateNumber)
    external
    view
    returns (
        address renter,
        uint256 totalPaid,
        uint256 rentalFee,
        uint256 depositAmount,
        uint256 timestamp,
        bool returned
    )
{
    RentalInfo memory info = rentalRecords[plateNumber];
    return (
        info.renter,
        info.amountPaid,
        info.rentalFee,
        info.depositAmount,
        info.timestamp,
        info.returned
    );
}

// 보증금 배수 확인
```

```
    function getDepositMultiplier() external pure returns (uint256) {
        return DEPOSIT_MULTIPLIER;
    }

    // 차량 소유자의 누적 수익 확인
    function getOwnerRevenue(address owner) external view returns (uint256) {
        return ownerRevenue[owner];
    }
    // 이벤트: 차량 대여 및 반납 기록
    event CarRented(string plateNumber, address renter, uint256 rentalFee, uint256
deposit);
    event CarReturned(string plateNumber, address renter, uint256 refundedDeposit);
}
```

➢ KYCManager.sol

```
pragma solidity ^0.8.19;
contract KYCManager {
    // 관리자 주소
    address public admin;

    // KYC가 승인된 사용자 목록 (true면 승인된 상태)
    mapping(address => bool) public approvedUsers;

    // 사용자가 KYC 요청했을 때 발생
    event KYCRequested(address indexed user);

    // 관리자가 KYC 승인을 철회했을 때 발생
    event KYCRevoked(address indexed user);

    // 관리자만 호출할 수 있는 제어자
    modifier onlyAdmin() {
        require(msg.sender == admin, "Only admin can perform this action");
        _;
    }

    // 배포할 때 관리자 주소를 설정
    constructor(address _admin) {
        admin = _admin;
    }
    // 사용자가 직접 KYC 신청
    function requestKYC() external {
        require(!approvedUsers[msg.sender], "Already approved");
        approvedUsers[msg.sender] = true;
        emit KYCRequested(msg.sender);
    }
    // 특정 사용자의 KYC 승인 여부 확인
    function checkKYC(address user) external view returns (bool) {
        return approvedUsers[user];
    }

    // 주어진 주소가 관리자인지 확인
    function isAdmin(address user) external view returns (bool) {
        return user == admin;
    }
    // 관리자가 사용자의 KYC 승인을 철회
    function revokeKYC(address user) external onlyAdmin {
        require(approvedUsers[user], "User not approved");
        approvedUsers[user] = false;
        emit KYCRevoked(user);
    }
}
```

➢ RentalVault.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
```

```
contract RentalVault {
    // CarRental 컨트랙트 주소 (이 컨트랙트만 입금/환불 가능)
    address public carRentalContract;

    // CarRental 컨트랙트만 호출 가능하도록 제한
    modifier onlyRentalContract() {
        require(msg.sender == carRentalContract, "Unauthorized");
        _;
    }

    // 배포 시 호출한 주소를 CarRental 컨트랙트로 설정
    constructor() {
        carRentalContract = msg.sender;
    }

    // CarRental 컨트랙트로부터 보증금을 받음
    function deposit() external payable onlyRentalContract {}

    // 사용자의 보증금을 환불
    function refund(address to, uint256 amount) external onlyRentalContract {
        (bool sent, ) = to.call{value: amount}("");
        require(sent, "Refund failed");
    }

    // CarRental 컨트랙트 주소를 변경 (테스트나 재설정용)
    function setRentalContract(address _addr) external {
        carRentalContract = _addr;
    }
}
```

➢ **ICarRegistry.sol(**차량 관리 계약에 대한 인터페이스**)**

```
pragma solidity ^0.8.18;

interface ICarRegistry {
    // 차량 상태: 사용 가능, 대여 중, 점검 중
    enum CarStatus { Available, Rented, Maintenance }

    // 차량 정보 조회
    function getCar(string memory plateNumber) external view returns (
        string memory, // 차량번호
        string memory, // 모델명
        string memory, // 위치
        uint256,       // 일일 요금
        uint8,         // 차량 상태 (enum CarStatus)
        address,       // 대여자 주소
        address        // 소유자 주소
    );

    // 차량을 대여 상태로 설정하고 대여자 주소 기록
    function setCarRented(string memory plateNumber, address renter) external;

    // 차량을 사용 가능 상태로 변경
    function setCarAvailable(string memory plateNumber) external;
}
```