Lesson 11 Documentation

APIconsumer.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

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

/**
 * @title The APIConsumer contract
 * @notice An API Consumer contract that makes GET requests to obtain 24h trading volume of ETH in USD
 */
contract APIConsumer is ChainlinkClient {
  using Chainlink for Chainlink.Request;

  uint256 public volume;
  address private immutable oracle;
  bytes32 private immutable jobId;
  uint256 private immutable fee;

  event DataFullfilled(uint256 volume);

  /**
   * @notice Executes once when a contract is created to initialize state variables
   *
   * @param _oracle - address of the specific Chainlink node that a contract makes an API call from
   * @param _jobId - specific job for :_oracle: to run; each job is unique and returns different types of data
   * @param _fee - node operator price per API call / data request
   * @param _link - LINK token address on the corresponding network
   *
   * Network: Rinkeby
   * Oracle: 0xc57b33452b4f7bb189bb5afae9cc4aba1f7a4fd8
   * Job ID: 6b88e0402e5d415eb946e528b8e0c7ba
   * Fee: 0.1 LINK
   */
```

```
  constructor(
    address _oracle,
    bytes32 _jobId,
    uint256 _fee,
    address _link
  ) {
    if (_link == address(0)) {
      setPublicChainlinkToken();
    } else {
      setChainlinkToken(_link);
    }
    oracle = _oracle;
    jobId = _jobId;
    fee = _fee;
  }

  /**
   * @notice Creates a Chainlink request to retrieve API response, find the target
   * data, then multiply by 1000000000000000000 (to remove decimal places from data).
   *
   * @return requestId - id of the request
   */
  function requestVolumeData() public returns (bytes32 requestId) {
    Chainlink.Request memory request = buildChainlinkRequest(
      jobId,
      address(this),
      this.fulfill.selector
    );

    // Set the URL to perform the GET request on
    request.add("get", "https://min-api.cryptocompare.com/data/pricemultifull?fsyms=ETH&tsyms=USD");
```

```
  // Set the path to find the desired data in the API response, where the response format is:
  // {"RAW":
  //   {"ETH":
  //    {"USD":
  //     {
  //      "VOLUME24HOUR": xxx.xxx,
  //     }
  //    }
  //   }
  //  }
  // request.add("path", "RAW.ETH.USD.VOLUME24HOUR"); // Chainlink nodes prior to 1.0.0 support this format
  request.add("path", "RAW,ETH,USD,VOLUME24HOUR"); // Chainlink nodes 1.0.0 and later support this format

  // Multiply the result by 1000000000000000000 to remove decimals
  int256 timesAmount = 10**18;
  request.addInt("times", timesAmount);

  // Sends the request
  return sendChainlinkRequestTo(oracle, request, fee);
}

/**
 * @notice Receives the response in the form of uint256
 *
 * @param _requestId - id of the request
 * @param _volume - response; requested 24h trading volume of ETH in USD
 */
function fulfill(bytes32 _requestId, uint256 _volume)
  public
  recordChainlinkFulfillment(_requestId)
{
  volume = _volume;
  emit DataFullfilled(volume);
```

```
    volume = _volume;
    emit DataFullfilled(volume);
  }

  /**
   * @notice Witdraws LINK from the contract
   * @dev Implement a withdraw function to avoid locking your LINK in the contract
   */
  function withdrawLink() external {}
}
```

Keeperscounter.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

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

/**
 * @title The Counter contract
 * @notice  A keeper-compatible contract that increments counter variable at fixed time intervals
 */
contract KeepersCounter is KeeperCompatibleInterface {
  /**
   * Public counter variable
   */
  uint256 public counter;

  /**
   * Use an interval in seconds and a timestamp to slow execution of Upkeep
   */
  uint256 public immutable interval;
  uint256 public lastTimeStamp;

  /**
   * @notice Executes once when a contract is created to initialize state variables
   *
   * @param updateInterval - Period of time between two counter increments expressed as UNIX timestamp value
   */
  constructor(uint256 updateInterval) {
    interval = updateInterval;
    lastTimeStamp = block.timestamp;

    counter = 0;
  }
```

```
  /**
   * @notice Checks if the contract requires work to be done
   */
  function checkUpkeep(
    bytes memory /* checkData */
  )
    public
    override
    returns (
      bool upkeepNeeded,
      bytes memory /* performData */
    )
  {
    upkeepNeeded = (block.timestamp - lastTimeStamp) > interval;
    // We don't use the checkData in this example. The checkData is defined when the Upkeep was registered.
  }

  /**
   * @notice Performs the work on the contract, if instructed by :checkUpkeep():
   */
  function performUpkeep(
    bytes calldata /* performData */
  ) external override {
    // add some verification
    (bool upkeepNeeded, ) = checkUpkeep("");
    require(upkeepNeeded, "Time interval not met");

    lastTimeStamp = block.timestamp;
    counter = counter + 1;
    // We don't use the performData in this example. The performData is generated by the Keeper's call to your checkUpkeep fun
  }
}
```

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import "../../KeepersCounter.sol";

contract KeepersCounterEchidnaTest is KeepersCounter {
  constructor() KeepersCounter(8 days) {}

  function echidna_test_perform_upkeep_gate() public view returns (bool) {
    return counter == 0;
  }
}
```