```
// SPDX-License-Identifier: MIT


pragma solidity ^0.8.7;
import "@chainlink/contracts/src/v0.8/ChainlinkClient.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

contract APIConsumer is ChainlinkClient, Initializable{
 using Chainlink for Chainlink.Request;
 using SafeMath for uint256;
 uint256 public price;
 bytes32 private jobId;
 uint256 private fee;
 uint256 public _linkFee;

 event RequestPrice(bytes32 indexed requestId, uint256 price);

 function initialize() public virtual onlyInitializing {
   setChainlinkToken(0x326C977E6efc84E512bB9C30f76E30c160eD06FB);
   setChainlinkOracle(0xCC79157eb46F5624204f47AB42b3906cAA40eaB7);
   jobId = "ca98366cc7314957b8c012c72f05aeeb";
   fee = (_linkFee * LINK_DIVISIBILITY) / 10; // 0,1 * 10**18 (Varies by network and
job)
 }

modifier onlyRequestPriceData() {
  require(msg.sig == bytes4(keccak256("requestPriceData()")));
  _;
  }
 /**
  * Create a Chainlink request to retrieve API response, find the target
  * data, then multiply by 1000000000000000000 (to remove decimal places from data).
  */
 function requestPriceData() public returns (bytes32 requestId) {
     Chainlink.Request memory req = buildChainlinkRequest(jobId, address(this),
this.fulfill.selector);
     // Set the URL to perform the GET request on
     req.add("get","https://api.metals.live/v1/spot");
      req.add("path", "2,platinum"); // Chainlink nodes 1.0.0 and later support this
format
     // Multiply the result by 1000000000000000000 to remove decimals
     int256 timesAmount = 10 ** 6;
     req.addInt("times", timesAmount);
     // Sends the request
     return sendChainlinkRequest(req, fee);
 }
 /**
  * Receive the response in the form of uint256
  */
 function fulfill(bytes32 _requestId,uint256 _price) public
recordChainlinkFulfillment(_requestId) {
     emit RequestPrice(_requestId, _price);
     price = _price;
  }
}



pragma solidity ^0.8.17;
import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
import
"@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC20BurnableUpgradeable.s
ol";
import "@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";


contract builder is OwnableUpgradeable, PausableUpgradeable, APIConsumer {

mapping(address => bool) private whitelists; //maps the whitelist

//checks if wallet is on the whitelist
modifier onlyWhitelist() {
 require(whitelists[msg.sender], "Caller is not white list member");
 _;
 }
// adds and removes wallets from the white list
// can only be called by owner
function setWhiteList(address to, bool value) public onlyOwner {
 whitelists[to] = value;
}
```

```
// chekcs to see if wallet is whitelisted
function isWhitelist(address to) public view returns (bool) {
 return whitelists[to];
}

//transfer any ERC20 held by this wallet to an entered address
//can only be called by owner
function rescueERC20(address tokenContract, address to, uint256 amount) external
whenNotPaused onlyOwner {
      ERC20(tokenContract).transfer(to, amount);
  }
// owner can manually set the mint price
function _setPrice(uint256 setPrice) public onlyOwner{
    price = setPrice;
}

}




contract xTST is ERC20Upgradeable,  UUPSUpgradeable, ERC20BurnableUpgradeable,
builder{


//custom:oz-upgrades-unsafe-allow constructor
   constructor() {
       _disableInitializers();
   }

//This line states that we are using the SafeMath library for all uint256 variables in
the contract
using SafeMath for uint256;


mapping(address => bool) private  blacklists; // mapping for addresses that are
blacklisted
mapping(address => bool) private feeLess_address;  //mapping for addresses that are
there are no transfer fees if tokens are sent there
uint256 public feePercent; // transfer fee for sending tokens
```

```
uint256 private restPercent; // percent leftover after sending the fee amount in a
transfer
address public feeAddress; // address that transfer fees are sent to
uint256 public _mintableTokens; // amount of tokens that are currently available to be
minted
uint256 public _buff; //fee to mint tokens
address [] public TokenInfo; // stores the ERC20 contract addresses that can be used
to mint

// intializes the contract and can only be called once
function initialize() initializer public override {
  APIConsumer.initialize();
  __ERC20_init("Test_TST", "TST");
  __ERC20Burnable_init();
  __Pausable_init();
  __Ownable_init();
  feePercent = 100000000000000000; /* 0.1% percent */
  restPercent = 100 * 10 ** 18 - feePercent;
  feeAddress = msg.sender;
  _mintableTokens = 0;
  _buff = 0;
}


/** set the amount of tokens that are available to mint
* can only be called by onlyOwner
*
*/

function setMintableTokens(uint256 new_mintableTokens) public whenNotPaused onlyOwner
{
    _mintableTokens = new_mintableTokens;
}

/** change the minting fee
* can only be called by onlyOwner
*
*/

function _changeBuff(uint256 buff) public whenNotPaused onlyOwner {
    _buff = buff;
}
```

```
/** add address to an ERC20 that can used to
* can only be called by onlyOwner
*
*/

 function _addPaytoken(address _paytokenAddress) public whenNotPaused onlyOwner {
   TokenInfo.push(_paytokenAddress);
   }

/** Whitelisted users can call mint to mint xPT tokens
* enter the amount of tokens desired (only whole tokens can be minted)
* enter the _pid which will choose with ERC20 token will used to pay for the mint
* the price for the token is called from the chainlink API
* a minting fee is added to the price of the token
* user will have to approve the ERC20 to this contract
* the amount of mintable tokens will decrease by the amount that is being minted
* if the owner mints the function will mint the amount of tokens requested
*/


 function mint(uint256 amount, uint256 _pid) public whenNotPaused onlyWhitelist {
  if(msg.sender != owner()) {
     uint256 _spend = amount * (price.mul(_buff).div(10**20) + price);
     IERC20 paytoken = IERC20 (TokenInfo[_pid]);
     require(amount <= _mintableTokens, "Not enough mintable tokens available");
     require(paytoken.balanceOf(msg.sender) >= _spend, "Insufficent balance");
     paytoken.transferFrom(msg.sender, address(this), _spend);
     _mintableTokens = _mintableTokens - amount;
     _mint(msg.sender, amount.mul(10**18));
     }
 else {
     _mint(msg.sender, amount);
}
}

// burns the token and can only be called by owner

function burn(uint256 amount) public whenNotPaused onlyOwner override{
    _burn(msg.sender, amount);
}
```

```
// checks blacklist before transfering tokens

function _beforeTokenTransfer(address from, address to, uint256 amount) internal
whenNotPaused override{
    require(!blacklists[from] && !blacklists[to], 'Blacklisted address');
    super._beforeTokenTransfer(from, to, amount);
}

// sets the address that accepts the transfer fee
function setFeeAddress(address payable _feeAddress) external  whenNotPaused onlyOwner{
    feeAddress = _feeAddress;
}

// sets the addresses that there is no transfer fee if they are the recepient
function setFeeLessAddress(address to, bool value) public whenNotPaused  onlyOwner {
 feeLess_address[to] = value;
}

// shows which addresses are feeless
function is_feeLess_address(address to) public view returns (bool) {
 return feeLess_address[to];
}

// changes the tranfer fee percent
function _changeFeePercent(uint256 _feePercent) public whenNotPaused onlyOwner {
    feePercent = _feePercent;
    restPercent = 100 * 10 ** 18 - feePercent;
}

/* tranfers tokens from one wallet to another
* if the recepient is a feeless address then there is no transfer fee
* else the funtion checks the sender and recipient to make sure they are not on the
blacklist
* the transfer fee is sent to the feeaddress wallet
*
*/

function transfer(address recipient, uint256 amount) public whenNotPaused override
returns (bool) {
  if(feeLess_address[recipient]) {
      super.transfer(recipient, amount);
      return true;
  }
  else {
    require(!blacklists[recipient], 'Blacklisted address');
    super.transfer(feeAddress, amount.mul(feePercent).div(10**20));
    super.transfer(recipient, amount.mul(restPercent).div(10**20));
    return true;
    }
}



/**allows another wallet to pull from the approved wallet
*
* checks the blacklist
*/

function transferFrom(address sender, address recipient, uint256 amount) public
whenNotPaused override returns (bool) {
    require(!blacklists[sender] && !blacklists[recipient], 'Blacklisted address');
    super.transferFrom(sender, feeAddress, amount.mul(feePercent).div(10**20));
    super.transferFrom(sender, recipient, amount.mul(restPercent).div(10**20));
    return true;
}

 /**
 * Allows for an upgrade implementation and can only called by the owner
 */


function _authorizeUpgrade(address newImplementation) internal onlyOwner override {
}



}
```