
Armors Labs
BALA
Smart Contract Audit

BALA Audit Summary

Project name : BALA Contract

Project address: None

Code URL : https://github.com/bala-finance

Commit : 57d0b5bdf681697df97eb4cf69f62019a8ca61a3

Project target : BALA Contract Audit

Blockchain : Hoo Smart Chain (HSC)

Test result : PASSED

Audit Info

Audit NO : 0X202107010006

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

BALA Audit

The BALA team asked us to review and audit their BALA contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
BALA Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-07-01

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the BALA contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

(Statement: Armors Labs reports only on facts that have occurred or existed before this report is issued and assumes corresponding responsibilities. Armors Labs is not able to determine the security of its smart contracts and is not responsible for any subsequent or existing facts after this report is issued. The security audit analysis and other content of this report are only based on the documents and information provided by the information provider to Armors Labs at the time of issuance of this report (" information provided " for short). Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered,

deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused.)

Audited target file

file	md5
./mocks/oracle/GenericOracleI.sol	5a53cc0b4d73f6587b7151710be72fc9
./mocks/oracle/LendingRateOracle.sol	d43c7781f902fb9aff858d7a1d5cfde3
./mocks/oracle/IExtendedPriceAggregator.sol	03959a5a1fa940eacb23f234b5706a92
./mocks/oracle/ChainlinkUSDETHOracleI.sol	2c81bceee227fa7898f696f3cffd0abf
./mocks/oracle/PriceOracle.sol	2593d848f2b4fd8909c77f41eccc907f
./mocks/oracle/CLAggregators/MockAggregator.sol	14719f5b7dc1aa0718c6d8b1fa8c66ec
./mocks/swap/MockUniswapV2Router02.sol	99a2dca6d1d755059e31da5e751b0ab7
./mocks/flashloan/MockFlashLoanReceiver.sol	6078d580dd7157b6831bc0215b87978e
./mocks/dependencies/weth/WETH9.sol	5175625bc241ea512297dba1b5711d06
./mocks/tokens/MintableERC20.sol	79dd5fd193bd7e8ee1dfb805489712fd
./mocks/tokens/WETH9Mocked.sol	11ba6b97c698a623974d2b9b4c420629
./mocks/tokens/MintableDelegationERC20.sol	3ac76949d15be4c32fb63f4341fc5fa8
./mocks/attacks/SefldestructTransfer.sol	ed16f0edec9773b8bc25fc80e145c3c0
./mocks/upgradeability/MockAToken.sol	6295cd444714e8621a695d0d8a6c5d3b
./mocks/upgradeability/MockStableDebtToken.sol	4ab5a951ea6ba42e9aab3f3a33bb04fd
./mocks/upgradeability/MockVariableDebtToken.sol	9d0b10d9a3cae8bc3ace26265574aa55
./misc/WalletBalanceProvider.sol	3f8f7c5fe582fc84f541ca363efb5a3e
./misc/WETHGateway.sol	cb2cfb673f837bf5a8a3c74bf73365fa
./misc/AaveOracle.sol	b3426d7969d83741b47664f79f9b919b
./misc/AaveProtocolDataProvider.sol	8b8de07327a400bba8b27b197762f8fb
./misc/UiPoolDataProvider.sol	ffa70b16406144cff4ac72fb7b038493
./misc/interfaces/IWETHGateway.sol	d56a0efb3776bfcbb0e887dcf33d5016
./misc/interfaces/IUiPoolDataProvider.sol	a8366845fd50b1a4bd870fb5b3fbc98d
./misc/interfaces/IWETH.sol	ab2af8b1175a4e9dfdf3f7754dbee9a5
./misc/interfaces/IUniswapV2Router01.sol	4e9a370698f8a3b06afe7e5f89d85186
./misc/interfaces/IUniswapV2Router02.sol	d75fbd3aaa9e4f38730e4faf2b8fbe56
./misc/interfaces/IERC20DetailedBytes.sol	ca7eceb3285ab76efb59fd93d468ad41
./flashloan/base/FlashLoanReceiverBase.sol	342099eb47e5ca9af28b333264a5a149

file	md5
./flashloan/interfaces/IFlashLoanReceiver.sol	5e569987e7faa6a858ede0e47939966b
.../contracts/SafeERC20.sol	7274eaf096edf0d7dde9b3282e4fcb42
.../contracts/Context.sol	35c596d58cea32ae629aeb27ad2ead6c
.../contracts/SafeMath.sol	cf6dbbc25d7163c47e5260b755361955
.../contracts/Ownable.sol	42b1116808efc7ee5bd735beefb0c3a0
.../contracts/Address.sol	bdb6e9218c297368a6ccd434a6ceb0ff
.../contracts/ERC20.sol	1c9ef76d716ea1e56f8e93d99a49bb10
.../contracts/IERC20.sol	91c0afdd4eb122474c0d0c040dab794a
.../contracts/IERC20Detailed.sol	0448c76ea7803fca57646a5beb17e8a2
.../upgradeability/UpgradeabilityProxy.sol	0f693cbb0e05cb04aa30f79b105385b1
.../upgradeability/AdminUpgradeabilityProxy.sol	1895075e0d13e7bfddc4deee8a19a368
.../upgradeability/Initializable.sol	13ee7092ae365253018384630bc40f37
.../upgradeability/InitializableUpgradeabilityProxy.sol	4578675ddb0759973506da311540ec82
.../upgradeability/Proxy.sol	ebf21fde26904e77797d7ada834c29e5
.../upgradeability/BaseUpgradeabilityProxy.sol	843dbecc88a254c6e02012b1efb48bcc
.../upgradeability/InitializableAdminUpgradeabilityProxy.sol	02d15f5e79d44c710cb65d24f96e8a61
.../upgradeability/BaseAdminUpgradeabilityProxy.sol	2bd0cc4dc7392dc2c414ca903c3e2ada
.../configuration/LendingPoolAddressesProviderRegistry.sol	15a1e88b6cd45ae8e6d6bac72f9e87be
.../configuration/LendingPoolAddressesProvider.sol	c7a933bb7c54957268e6889c75ba6158
.../configuration/ReserveConfiguration.sol	c0f46e3d5df412e9da76cd291344f026
.../configuration/UserConfiguration.sol	14cd911a7c407bdc1b0d23cb7a258a54
.../types/DataTypes.sol	2127b8d36674faba96bc703f72164a1a
.../logic/ReserveLogic.sol	d15ddc108407134c8293813a2bf3d2bb
.../logic/ValidationLogic.sol	956cfd1dde7c76c40951271fc7ab70aa
.../logic/GenericLogic.sol	63c0b6b808431eddcdd4f72f15116e1c
.../math/MathUtils.sol	11581a64630b7297869b359ac31927e7
.../math/WadRayMath.sol	bf062018f7a0f55c2c54909629b3028d
.../math/PercentageMath.sol	ac3f1f6dfed160907baca58211bb6d37
.../.../BaseImmutableAdminUpgradeabilityProxy.sol	c2f279451bae93b05a05b4dfe73965cc
.../.../InitializableImmutableAdminUpgradeabilityProxy.sol	b6341ef33dcb8301a510795e68fdfb47
.../.../VersionedInitializable.sol	e46d1675e7ec7446ef0536ab7c3db963
.../helpers/Helpers.sol	2c6f19fbad822bd9fc72c2012188eedc

file	md5
.../helpers/Errors.sol	f9f147c0810b2cf78866930180648be5
.../tokenization/VariableDebtToken.sol	46ae51546a44c7d66e49bb017eeb4985
.../tokenization/AToken.sol	618e6f62742873950a54f2d6cbe4950c
.../tokenization/StableDebtToken.sol	e4103247d379ff0070b87ce4bf9fa4d4
.../tokenization/IncentivizedERC20.sol	2eaf57d54f36c3017a3596f77be8532d
.../tokenization/DelegationAwareAToken.sol	04f040cdfc28acce298999828702fa64
.../tokenization/base/DebtTokenBase.sol	c7a46dce60aa28172ccb12e1eee3143d
.../lendingpool/LendingPoolCollateralManager.sol	bd0641b34d2bdf3a679abf9e1f34acea
.../lendingpool/LendingPoolConfigurator.sol	a450e1be0037ab48cc05717afb1e6889
.../lendingpool/DefaultReserveInterestRateStrategy.sol	18d3168edac10e76ae1214192ac4483d
.../lendingpool/LendingPoolStorage.sol	71ed7efdbf03ef30cf22a32e94aee84e
.../lendingpool/LendingPool.sol	b9d1d1e3f8c43d0a15e7d4474b1e411b
./adapters/BaseUniswapAdapter.sol	865cf347d6d906fb50a90f811e6a42fc
./adapters/FlashLiquidationAdapter.sol	32701d733ab5ba66408cb8939382aa2f
./adapters/UniswapRepayAdapter.sol	6fd39f9dbf395553165724c6e0654eea
./adapters/UniswapLiquiditySwapAdapter.sol	11dbb42a3bae79b9f0ca43fcca69625c
./adapters/interfaces/IBaseUniswapAdapter.sol	a62c64a0a9901e9f9806fb32b0366a01
./interfaces/IERC20WithPermit.sol	cd97ed0a4faa085377a953815750d216
./interfaces/IScaledBalanceToken.sol	05f5c83b51e4b04e17785f92ca7840de
./interfaces/IChainlinkAggregator.sol	7a25050278be7e9f3ac7424c0d69a049
./interfaces/IInitializableDebtToken.sol	ae4d2dc34379e54d05226248c41da496
./interfaces/IVariableDebtToken.sol	297a4e07a70791964b5e5b96fee1572a
./interfaces/ILendingRateOracle.sol	dc30a43ed82b7bfe230593226324c891
./interfaces/ILendingPoolCollateralManager.sol	7b3bff7810d10b985d9e2fca06efb374
./interfaces/IUniswapExchange.sol	02272f4b5f947c3607168fc3eaaaa29d
./interfaces/IReserveInterestRateStrategy.sol	b11d9c14920ab9302b232bd3f7226e36
./interfaces/IDelegationToken.sol	082e2de4fca3cfa794b548eb64959089
./interfaces/IAaveIncentivesController.sol	1e9878616988e4923ea5563cc7d7e921
./interfaces/ILendingPoolAddressesProvider.sol	a487bfd067ce6e71ddc7e9cc91302bae
./interfaces/IUniswapV2Router02.sol	e6a3fa068b9ccc4f9d95180fbc8536f4
./interfaces/ILendingPool.sol	c9d32b9924ef08162c2453bcb64c569a
./interfaces/IPriceOracle.sol	2f743812a3f5003b3a8aad85caa6e335

file	md5
./interfaces/ICreditDelegationToken.sol	9d4c60aae9eb8093c9475ab517bdcd99
./interfaces/IInitializableAToken.sol	b408cb493b8651eca053fc831bea1cd8
./interfaces/IPriceOracleGetter.sol	56181b85c44591f931b25fa0b92ed581
./interfaces/IExchangeAdapter.sol	0b4971bce9c94dba30de0d4f92394ebe
./interfaces/ILendingPoolAddressesProviderRegistry.sol	249239470e9a5ff454bf3c1aa2f5bfd3
./interfaces/ILendingPoolConfigurator.sol	26bc9369a4456e17626fe2c3a8a6d6db
./interfaces/IStableDebtToken.sol	ce31ee6194c17b0b68078bcade919040
./interfaces/IAToken.sol	325b239ba3bede374aaa1931b90550d5
./deployments/ATokensAndRatesHelper.sol	cd58ce56df80332597f200f2c512d9a1
./deployments/StringLib.sol	062b80046c48b7a0945936544e868b62
./deployments/StableAndVariableTokensHelper.sol	54833bad78c142b27ac0ec19df58faf0

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe

Vulnerability	status
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {StableDebtToken} from '../protocol/tokenization/StableDebtToken.sol';
import {VariableDebtToken} from '../protocol/tokenization/VariableDebtToken.sol';
import {LendingRateOracle} from '../mocks/oracle/LendingRateOracle.sol';
import {Ownable} from '../dependencies/openzeppelin/contracts/Ownable.sol';
import {StringLib} from './StringLib.sol';

contract StableAndVariableTokensHelper is Ownable {
  address payable private pool;
  address private addressesProvider;
  event deployedContracts(address stableToken, address variableToken);

  constructor(address payable _pool, address _addressesProvider) public {
    pool = _pool;
    addressesProvider = _addressesProvider;
  }

  function initDeployment(address[] calldata tokens, string[] calldata symbols) external onlyOwner {
    require(tokens.length == symbols.length, 'Arrays not same length');
    require(pool != address(0), 'Pool can not be zero address');
    for (uint256 i = 0; i < tokens.length; i++) {
      emit deployedContracts(address(new StableDebtToken()), address(new VariableDebtToken()));
    }
  }

  function setOracleBorrowRates(
    address[] calldata assets,
    uint256[] calldata rates,
    address oracle
  ) external onlyOwner {
    require(assets.length == rates.length, 'Arrays not same length');

    for (uint256 i = 0; i < assets.length; i++) {
      // LendingRateOracle owner must be this contract
      LendingRateOracle(oracle).setMarketBorrowRate(assets[i], rates[i]);
    }
  }

  function setOracleOwnership(address oracle, address admin) external onlyOwner {
```

```
    require(admin != address(0), 'owner can not be zero');
    require(LendingRateOracle(oracle).owner() == address(this), 'helper is not owner');
    LendingRateOracle(oracle).transferOwnership(admin);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

library StringLib {
  function concat(string memory a, string memory b) internal pure returns (string memory) {
    return string(abi.encodePacked(a, b));
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {LendingPool} from '../protocol/lendingpool/LendingPool.sol';
import {
  LendingPoolAddressesProvider
} from '../protocol/configuration/LendingPoolAddressesProvider.sol';
import {LendingPoolConfigurator} from '../protocol/lendingpool/LendingPoolConfigurator.sol';
import {AToken} from '../protocol/tokenization/AToken.sol';
import {
  DefaultReserveInterestRateStrategy
} from '../protocol/lendingpool/DefaultReserveInterestRateStrategy.sol';
import {Ownable} from '../dependencies/openzeppelin/contracts/Ownable.sol';
import {StringLib} from './StringLib.sol';

contract ATokensAndRatesHelper is Ownable {
  address payable private pool;
  address private addressesProvider;
  address private poolConfigurator;
  event deployedContracts(address aToken, address strategy);

  struct InitDeploymentInput {
    address asset;
    uint256[6] rates;
  }

  struct ConfigureReserveInput {
    address asset;
    uint256 baseLTV;
    uint256 liquidationThreshold;
    uint256 liquidationBonus;
    uint256 reserveFactor;
    bool stableBorrowingEnabled;
  }

  constructor(
    address payable _pool,
    address _addressesProvider,
    address _poolConfigurator
  ) public {
    pool = _pool;
    addressesProvider = _addressesProvider;
    poolConfigurator = _poolConfigurator;
  }

  function initDeployment(InitDeploymentInput[] calldata inputParams) external onlyOwner {
    for (uint256 i = 0; i < inputParams.length; i++) {
      emit deployedContracts(
        address(new AToken()),
        address(
          new DefaultReserveInterestRateStrategy(
            LendingPoolAddressesProvider(addressesProvider),
```

```
            inputParams[i].rates[0],
            inputParams[i].rates[1],
            inputParams[i].rates[2],
            inputParams[i].rates[3],
            inputParams[i].rates[4],
            inputParams[i].rates[5]
          )
        )
      );
    }
  }

  function configureReserves(ConfigureReserveInput[] calldata inputParams) external onlyOwner {
    LendingPoolConfigurator configurator = LendingPoolConfigurator(poolConfigurator);
    for (uint256 i = 0; i < inputParams.length; i++) {
      configurator.configureReserveAsCollateral(
        inputParams[i].asset,
        inputParams[i].baseLTV,
        inputParams[i].liquidationThreshold,
        inputParams[i].liquidationBonus
      );

      configurator.enableBorrowingOnReserve(
        inputParams[i].asset,
        inputParams[i].stableBorrowingEnabled
      );
      configurator.setReserveFactor(inputParams[i].asset, inputParams[i].reserveFactor);
    }
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';
import {IScaledBalanceToken} from './IScaledBalanceToken.sol';
import {IInitializableAToken} from './IInitializableAToken.sol';
import {IAaveIncentivesController} from './IAaveIncentivesController.sol';

interface IAToken is IERC20, IScaledBalanceToken, IInitializableAToken {
  /**
   * @dev Emitted after the mint action
   * @param from The address performing the mint
   * @param value The amount being
   * @param index The new liquidity index of the reserve
   **/
  event Mint(address indexed from, uint256 value, uint256 index);

  /**
   * @dev Mints `amount` aTokens to `user`
   * @param user The address receiving the minted tokens
   * @param amount The amount of tokens getting minted
   * @param index The new liquidity index of the reserve
   * @return `true` if the the previous balance of the user was 0
   */
  function mint(
    address user,
    uint256 amount,
    uint256 index
  ) external returns (bool);

  /**
   * @dev Emitted after aTokens are burned
   * @param from The owner of the aTokens, getting them burned
   * @param target The address that will receive the underlying
   * @param value The amount being burned
   * @param index The new liquidity index of the reserve
```

```
  **/
event Burn(address indexed from, address indexed target, uint256 value, uint256 index);

/**
 * @dev Emitted during the transfer action
 * @param from The user whose tokens are being transferred
 * @param to The recipient
 * @param value The amount being transferred
 * @param index The new liquidity index of the reserve
 **/
event BalanceTransfer(address indexed from, address indexed to, uint256 value, uint256 index);

/**
 * @dev Burns aTokens from `user` and sends the equivalent amount of underlying to `receiverOfUnder
 * @param user The owner of the aTokens, getting them burned
 * @param receiverOfUnderlying The address that will receive the underlying
 * @param amount The amount being burned
 * @param index The new liquidity index of the reserve
 **/
function burn(
  address user,
  address receiverOfUnderlying,
  uint256 amount,
  uint256 index
) external;

/**
 * @dev Mints aTokens to the reserve treasury
 * @param amount The amount of tokens getting minted
 * @param index The new liquidity index of the reserve
 */
function mintToTreasury(uint256 amount, uint256 index) external;

/**
 * @dev Transfers aTokens in the event of a borrow being liquidated, in case the liquidators reclai
 * @param from The address getting liquidated, current owner of the aTokens
 * @param to The recipient
 * @param value The amount of tokens getting transferred
 **/
function transferOnLiquidation(
  address from,
  address to,
  uint256 value
) external;

/**
 * @dev Transfers the underlying asset to `target`. Used by the LendingPool to transfer
 * assets in borrow(), withdraw() and flashLoan()
 * @param user The recipient of the underlying
 * @param amount The amount getting transferred
 * @return The amount transferred
 **/
function transferUnderlyingTo(address user, uint256 amount) external returns (uint256);

/**
 * @dev Invoked to execute actions on the aToken side after a repayment.
 * @param user The user executing the repayment
 * @param amount The amount getting repaid
 **/
function handleRepayment(address user, uint256 amount) external;

/**
 * @dev Returns the address of the incentives controller contract
 **/
function getIncentivesController() external view returns (IAaveIncentivesController);
```

```
  /**
   * @dev Returns the address of the underlying asset of this aToken (E.g. WETH for aWETH)
   **/
  function UNDERLYING_ASSET_ADDRESS() external view returns (address);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IScaledBalanceToken} from './IScaledBalanceToken.sol';
import {IInitializableDebtToken} from './IInitializableDebtToken.sol';
import {IAaveIncentivesController} from './IAaveIncentivesController.sol';

/**
 * @title IStableDebtToken
 * @notice Defines the interface for the stable debt token
 * @dev It does not inherit from IERC20 to save in code size
 * @author Aave
 **/

interface IStableDebtToken is IScaledBalanceToken,IInitializableDebtToken {
  /**
   * @dev Emitted when new stable debt is minted
   * @param user The address of the user who triggered the minting
   * @param onBehalfOf The recipient of stable debt tokens
   * @param amount The amount minted
   * @param currentBalance The current balance of the user
   * @param balanceIncrease The increase in balance since the last action of the user
   * @param newRate The rate of the debt after the minting
   * @param avgStableRate The new average stable rate after the minting
   * @param newTotalSupply The new total supply of the stable debt token after the action
   **/
  event Mint(
    address indexed user,
    address indexed onBehalfOf,
    uint256 amount,
    uint256 currentBalance,
    uint256 balanceIncrease,
    uint256 newRate,
    uint256 avgStableRate,
    uint256 newTotalSupply
  );

  /**
   * @dev Emitted when new stable debt is burned
   * @param user The address of the user
   * @param amount The amount being burned
   * @param currentBalance The current balance of the user
   * @param balanceIncrease The the increase in balance since the last action of the user
   * @param avgStableRate The new average stable rate after the burning
   * @param newTotalSupply The new total supply of the stable debt token after the action
   **/
  event Burn(
    address indexed user,
    uint256 amount,
    uint256 currentBalance,
    uint256 balanceIncrease,
    uint256 avgStableRate,
    uint256 newTotalSupply
  );

  /**
   * @dev Mints debt token to the `onBehalfOf` address.
   * - The resulting rate is the weighted average between the rate of the new debt
   * and the rate of the previous debt
   * @param user The address receiving the borrowed underlying, being the delegatee in case
   * of credit delegate, or same as `onBehalfOf` otherwise
```

```
   * @param onBehalfOf The address receiving the debt tokens
   * @param amount The amount of debt tokens to mint
   * @param rate The rate of the debt being minted
   **/
  function mint(
    address user,
    address onBehalfOf,
    uint256 amount,
    uint256 rate
  ) external returns (bool);

  /**
   * @dev Burns debt of `user`
   * - The resulting rate is the weighted average between the rate of the new debt
   * and the rate of the previous debt
   * @param user The address of the user getting his debt burned
   * @param amount The amount of debt tokens getting burned
   **/
  function burn(address user, uint256 amount) external;

  /**
   * @dev Returns the average rate of all the stable rate loans.
   * @return The average stable rate
   **/
  function getAverageStableRate() external view returns (uint256);

  /**
   * @dev Returns the stable rate of the user debt
   * @return The stable rate of the user
   **/
  function getUserStableRate(address user) external view returns (uint256);

  /**
   * @dev Returns the timestamp of the last update of the user
   * @return The timestamp
   **/
  function getUserLastUpdated(address user) external view returns (uint40);

  /**
   * @dev Returns the principal, the total supply and the average stable rate
   **/
  function getSupplyData()
    external
    view
    returns (
      uint256,
      uint256,
      uint256,
      uint40
    );

  /**
   * @dev Returns the timestamp of the last update of the total supply
   * @return The timestamp
   **/
  function getTotalSupplyLastUpdated() external view returns (uint40);

  /**
   * @dev Returns the total supply and the average stable rate
   **/
  function getTotalSupplyAndAvgRate() external view returns (uint256, uint256);

  /**
   * @dev Returns the principal debt balance of the user
   * @return The debt balance of the user since the last burn/mint action
   **/
```

```
  function principalBalanceOf(address user) external view returns (uint256);

  /**
   * @dev Returns the address of the incentives controller contract
   **/
  function getIncentivesController() external view returns (IAaveIncentivesController);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

interface ILendingPoolConfigurator {
  struct InitReserveInput {
    address aTokenImpl;
    address stableDebtTokenImpl;
    address variableDebtTokenImpl;
    uint8 underlyingAssetDecimals;
    address interestRateStrategyAddress;
    address underlyingAsset;
    address treasury;
    address incentivesController;
    string underlyingAssetName;
    string aTokenName;
    string aTokenSymbol;
    string variableDebtTokenName;
    string variableDebtTokenSymbol;
    string stableDebtTokenName;
    string stableDebtTokenSymbol;
    bytes params;
  }

  struct UpdateATokenInput {
    address asset;
    address treasury;
    address incentivesController;
    string name;
    string symbol;
    address implementation;
    bytes params;
  }

  struct UpdateDebtTokenInput {
    address asset;
    address incentivesController;
    string name;
    string symbol;
    address implementation;
    bytes params;
  }

  /**
   * @dev Emitted when a reserve is initialized.
   * @param asset The address of the underlying asset of the reserve
   * @param aToken The address of the associated aToken contract
   * @param stableDebtToken The address of the associated stable rate debt token
   * @param variableDebtToken The address of the associated variable rate debt token
   * @param interestRateStrategyAddress The address of the interest rate strategy for the reserve
   **/
  event ReserveInitialized(
    address indexed asset,
    address indexed aToken,
    address stableDebtToken,
    address variableDebtToken,
    address interestRateStrategyAddress
  );
```

```
  /**
   * @dev Emitted when borrowing is enabled on a reserve
   * @param asset The address of the underlying asset of the reserve
   * @param stableRateEnabled True if stable rate borrowing is enabled, false otherwise
   **/
  event BorrowingEnabledOnReserve(address indexed asset, bool stableRateEnabled);

  /**
   * @dev Emitted when borrowing is disabled on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  event BorrowingDisabledOnReserve(address indexed asset);

  /**
   * @dev Emitted when the collateralization risk parameters for the specified asset are updated.
   * @param asset The address of the underlying asset of the reserve
   * @param ltv The loan to value of the asset when used as collateral
   * @param liquidationThreshold The threshold at which loans using this asset as collateral will be
   * @param liquidationBonus The bonus liquidators receive to liquidate this asset
   **/
  event CollateralConfigurationChanged(
    address indexed asset,
    uint256 ltv,
    uint256 liquidationThreshold,
    uint256 liquidationBonus
  );

  /**
   * @dev Emitted when stable rate borrowing is enabled on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  event StableRateEnabledOnReserve(address indexed asset);

  /**
   * @dev Emitted when stable rate borrowing is disabled on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  event StableRateDisabledOnReserve(address indexed asset);

  /**
   * @dev Emitted when a reserve is activated
   * @param asset The address of the underlying asset of the reserve
   **/
  event ReserveActivated(address indexed asset);

  /**
   * @dev Emitted when a reserve is deactivated
   * @param asset The address of the underlying asset of the reserve
   **/
  event ReserveDeactivated(address indexed asset);

  /**
   * @dev Emitted when a reserve is frozen
   * @param asset The address of the underlying asset of the reserve
   **/
  event ReserveFrozen(address indexed asset);

  /**
   * @dev Emitted when a reserve is unfrozen
   * @param asset The address of the underlying asset of the reserve
   **/
  event ReserveUnfrozen(address indexed asset);

  /**
   * @dev Emitted when a reserve factor is updated
   * @param asset The address of the underlying asset of the reserve
```

```
    * @param factor The new reserve factor
    **/
  event ReserveFactorChanged(address indexed asset, uint256 factor);

  /**
    * @dev Emitted when the reserve decimals are updated
    * @param asset The address of the underlying asset of the reserve
    * @param decimals The new decimals
    **/
  event ReserveDecimalsChanged(address indexed asset, uint256 decimals);

  /**
    * @dev Emitted when a reserve interest strategy contract is updated
    * @param asset The address of the underlying asset of the reserve
    * @param strategy The new address of the interest strategy contract
    **/
  event ReserveInterestRateStrategyChanged(address indexed asset, address strategy);

  /**
    * @dev Emitted when an aToken implementation is upgraded
    * @param asset The address of the underlying asset of the reserve
    * @param proxy The aToken proxy address
    * @param implementation The new aToken implementation
    **/
  event ATokenUpgraded(
    address indexed asset,
    address indexed proxy,
    address indexed implementation
  );

  /**
    * @dev Emitted when the implementation of a stable debt token is upgraded
    * @param asset The address of the underlying asset of the reserve
    * @param proxy The stable debt token proxy address
    * @param implementation The new aToken implementation
    **/
  event StableDebtTokenUpgraded(
    address indexed asset,
    address indexed proxy,
    address indexed implementation
  );

  /**
    * @dev Emitted when the implementation of a variable debt token is upgraded
    * @param asset The address of the underlying asset of the reserve
    * @param proxy The variable debt token proxy address
    * @param implementation The new aToken implementation
    **/
  event VariableDebtTokenUpgraded(
    address indexed asset,
    address indexed proxy,
    address indexed implementation
  );
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title LendingPoolAddressesProviderRegistry contract
 * @dev Main registry of LendingPoolAddressesProvider of multiple Aave protocol's markets
 * - Used for indexing purposes of Aave protocol's markets
 * - The id assigned to a LendingPoolAddressesProvider refers to the market it is connected with,
 *   for example with `0` for the Aave main market and `1` for the next created
 * @author Aave
 **/
interface ILendingPoolAddressesProviderRegistry {
```

```
  event AddressesProviderRegistered(address indexed newAddress);
  event AddressesProviderUnregistered(address indexed newAddress);

  function getAddressesProvidersList() external view returns (address[] memory);

  function getAddressesProviderIdByAddress(address addressesProvider)
    external
    view
    returns (uint256);

  function registerAddressesProvider(address provider, uint256 id) external;

  function unregisterAddressesProvider(address provider) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';

interface IExchangeAdapter {
  event Exchange(
    address indexed from,
    address indexed to,
    address indexed platform,
    uint256 fromAmount,
    uint256 toAmount
  );

  function approveExchange(IERC20[] calldata tokens) external;

  function exchange(
    address from,
    address to,
    uint256 amount,
    uint256 maxSlippage
  ) external returns (uint256);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title IPriceOracleGetter interface
 * @notice Interface for the Aave price oracle.
 **/

interface IPriceOracleGetter {
  /**
   * @dev returns the asset price in ETH
   * @param asset the address of the asset
   * @return the ETH price of the asset
   **/
  function getAssetPrice(address asset) external view returns (uint256);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ILendingPool} from './ILendingPool.sol';
import {IAaveIncentivesController} from './IAaveIncentivesController.sol';

/**
 * @title IInitializableAToken
 * @notice Interface for the initialize function on AToken
 * @author Aave
 **/
interface IInitializableAToken {
  /**
```

```
    * @dev Emitted when an aToken is initialized
    * @param underlyingAsset The address of the underlying asset
    * @param pool The address of the associated lending pool
    * @param treasury The address of the treasury
    * @param incentivesController The address of the incentives controller for this aToken
    * @param aTokenDecimals the decimals of the underlying
    * @param aTokenName the name of the aToken
    * @param aTokenSymbol the symbol of the aToken
    * @param params A set of encoded parameters for additional initialization
    **/
  event Initialized(
    address indexed underlyingAsset,
    address indexed pool,
    address treasury,
    address incentivesController,
    uint8 aTokenDecimals,
    string aTokenName,
    string aTokenSymbol,
    bytes params
  );

  /**
   * @dev Initializes the aToken
   * @param pool The address of the lending pool where this aToken will be used
   * @param treasury The address of the Aave treasury, receiving the fees on this aToken
   * @param underlyingAsset The address of the underlying asset of this aToken (E.g. WETH for aWETH)
   * @param incentivesController The smart contract managing potential incentives distribution
   * @param aTokenDecimals The decimals of the aToken, same as the underlying asset's
   * @param aTokenName The name of the aToken
   * @param aTokenSymbol The symbol of the aToken
   */
  function initialize(
    ILendingPool pool,
    address treasury,
    address underlyingAsset,
    IAaveIncentivesController incentivesController,
    uint8 aTokenDecimals,
    string calldata aTokenName,
    string calldata aTokenSymbol,
    bytes calldata params
  ) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface ICreditDelegationToken {
  event BorrowAllowanceDelegated(
    address indexed fromUser,
    address indexed toUser,
    address asset,
    uint256 amount
  );

  /**
   * @dev delegates borrowing power to a user on the specific debt token
   * @param delegatee the address receiving the delegated borrowing power
   * @param amount the maximum amount being delegated. Delegation will still
   * respect the liquidation constraints (even if delegated, a delegatee cannot
   * force a delegator HF to go below 1)
   **/
  function approveDelegation(address delegatee, uint256 amount) external;

  /**
   * @dev returns the borrow allowance of the user
   * @param fromUser The user to giving allowance
   * @param toUser The user to give allowance to
```

```
   * @return the current allowance of toUser
   **/
  function borrowAllowance(address fromUser, address toUser) external view returns (uint256);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/************
@title IPriceOracle interface
@notice Interface for the Aave price oracle.*/
interface IPriceOracle {
  /***********
    @dev returns the asset price in ETH
     */
  function getAssetPrice(address asset) external view returns (uint256);

  /***********
    @dev sets the asset price, in wei
     */
  function setAssetPrice(address asset, uint256 price) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {ILendingPoolAddressesProvider} from './ILendingPoolAddressesProvider.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';

interface ILendingPool {
  /**
   * @dev Emitted on deposit()
   * @param reserve The address of the underlying asset of the reserve
   * @param user The address initiating the deposit
   * @param onBehalfOf The beneficiary of the deposit, receiving the aTokens
   * @param amount The amount deposited
   * @param referral The referral code used
   **/
  event Deposit(
    address indexed reserve,
    address user,
    address indexed onBehalfOf,
    uint256 amount,
    uint16 indexed referral
  );

  /**
   * @dev Emitted on withdraw()
   * @param reserve The address of the underlyng asset being withdrawn
   * @param user The address initiating the withdrawal, owner of aTokens
   * @param to Address that will receive the underlying
   * @param amount The amount to be withdrawn
   **/
  event Withdraw(address indexed reserve, address indexed user, address indexed to, uint256 amount);

  /**
   * @dev Emitted on borrow() and flashLoan() when debt needs to be opened
   * @param reserve The address of the underlying asset being borrowed
   * @param user The address of the user initiating the borrow(), receiving the funds on borrow() or
   * initiator of the transaction on flashLoan()
   * @param onBehalfOf The address that will be getting the debt
   * @param amount The amount borrowed out
   * @param borrowRateMode The rate mode: 1 for Stable, 2 for Variable
   * @param borrowRate The numeric rate at which the user has borrowed
   * @param referral The referral code used
   **/
  event Borrow(
```

```
    address indexed reserve,
    address user,
    address indexed onBehalfOf,
    uint256 amount,
    uint256 borrowRateMode,
    uint256 borrowRate,
    uint16 indexed referral
  );

  /**
   * @dev Emitted on repay()
   * @param reserve The address of the underlying asset of the reserve
   * @param user The beneficiary of the repayment, getting his debt reduced
   * @param repayer The address of the user initiating the repay(), providing the funds
   * @param amount The amount repaid
   **/
  event Repay(
    address indexed reserve,
    address indexed user,
    address indexed repayer,
    uint256 amount
  );

  /**
   * @dev Emitted on swapBorrowRateMode()
   * @param reserve The address of the underlying asset of the reserve
   * @param user The address of the user swapping his rate mode
   * @param rateMode The rate mode that the user wants to swap to
   **/
  event Swap(address indexed reserve, address indexed user, uint256 rateMode);

  /**
   * @dev Emitted on setUserUseReserveAsCollateral()
   * @param reserve The address of the underlying asset of the reserve
   * @param user The address of the user enabling the usage as collateral
   **/
  event ReserveUsedAsCollateralEnabled(address indexed reserve, address indexed user);

  /**
   * @dev Emitted on setUserUseReserveAsCollateral()
   * @param reserve The address of the underlying asset of the reserve
   * @param user The address of the user enabling the usage as collateral
   **/
  event ReserveUsedAsCollateralDisabled(address indexed reserve, address indexed user);

  /**
   * @dev Emitted on rebalanceStableBorrowRate()
   * @param reserve The address of the underlying asset of the reserve
   * @param user The address of the user for which the rebalance has been executed
   **/
  event RebalanceStableBorrowRate(address indexed reserve, address indexed user);

  /**
   * @dev Emitted on flashLoan()
   * @param target The address of the flash loan receiver contract
   * @param initiator The address initiating the flash loan
   * @param asset The address of the asset being flash borrowed
   * @param amount The amount flash borrowed
   * @param premium The fee flash borrowed
   * @param referralCode The referral code used
   **/
  event FlashLoan(
    address indexed target,
    address indexed initiator,
    address indexed asset,
    uint256 amount,
```

```
    uint256 premium,
    uint16 referralCode
  );

  /**
   * @dev Emitted when the pause is triggered.
   */
  event Paused();

  /**
   * @dev Emitted when the pause is lifted.
   */
  event Unpaused();

  /**
   * @dev Emitted when a borrower is liquidated. This event is emitted by the LendingPool via
   * LendingPoolCollateral manager using a DELEGATECALL
   * This allows to have the events in the generated ABI for LendingPool.
   * @param collateralAsset The address of the underlying asset used as collateral, to receive as res
   * @param debtAsset The address of the underlying borrowed asset to be repaid with the liquidation
   * @param user The address of the borrower getting liquidated
   * @param debtToCover The debt amount of borrowed `asset` the liquidator wants to cover
   * @param liquidatedCollateralAmount The amount of collateral received by the liiquidator
   * @param liquidator The address of the liquidator
   * @param receiveAToken `true` if the liquidators wants to receive the collateral aTokens, `false`
   * to receive the underlying collateral asset directly
   **/
  event LiquidationCall(
    address indexed collateralAsset,
    address indexed debtAsset,
    address indexed user,
    uint256 debtToCover,
    uint256 liquidatedCollateralAmount,
    address liquidator,
    bool receiveAToken
  );

  /**
   * @dev Emitted when the state of a reserve is updated. NOTE: This event is actually declared
   * in the ReserveLogic library and emitted in the updateInterestRates() function. Since the functio
   * the event will actually be fired by the LendingPool contract. The event is therefore replicated
   * gets added to the LendingPool ABI
   * @param reserve The address of the underlying asset of the reserve
   * @param liquidityRate The new liquidity rate
   * @param stableBorrowRate The new stable borrow rate
   * @param variableBorrowRate The new variable borrow rate
   * @param liquidityIndex The new liquidity index
   * @param variableBorrowIndex The new variable borrow index
   **/
  event ReserveDataUpdated(
    address indexed reserve,
    uint256 liquidityRate,
    uint256 stableBorrowRate,
    uint256 variableBorrowRate,
    uint256 liquidityIndex,
    uint256 variableBorrowIndex
  );

  /**
   * @dev Deposits an `amount` of underlying asset into the reserve, receiving in return overlying aT
   * - E.g. User deposits 100 USDC and gets in return 100 aUSDC
   * @param asset The address of the underlying asset to deposit
   * @param amount The amount to be deposited
   * @param onBehalfOf The address that will receive the aTokens, same as msg.sender if the user
   *   wants to receive them on his own wallet, or a different address if the beneficiary of aTokens
   *   is a different wallet
```

```
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   **/
  function deposit(
    address asset,
    uint256 amount,
    address onBehalfOf,
    uint16 referralCode
  ) external;

  /**
   * @dev Withdraws an `amount` of underlying asset from the reserve, burning the equivalent aTokens
   * E.g. User has 100 aUSDC, calls withdraw() and receives 100 USDC, burning the 100 aUSDC
   * @param asset The address of the underlying asset to withdraw
   * @param amount The underlying amount to be withdrawn
   *   - Send the value type(uint256).max in order to withdraw the whole aToken balance
   * @param to Address that will receive the underlying, same as msg.sender if the user
   *   wants to receive it on his own wallet, or a different address if the beneficiary is a
   *   different wallet
   * @return The final amount withdrawn
   **/
  function withdraw(
    address asset,
    uint256 amount,
    address to
  ) external returns (uint256);

  /**
   * @dev Allows users to borrow a specific `amount` of the reserve underlying asset, provided that t
   * already deposited enough collateral, or he was given enough allowance by a credit delegator on t
   * corresponding debt token (StableDebtToken or VariableDebtToken)
   * - E.g. User borrows 100 USDC passing as `onBehalfOf` his own address, receiving the 100 USDC in
   *   and 100 stable/variable debt tokens, depending on the `interestRateMode`
   * @param asset The address of the underlying asset to borrow
   * @param amount The amount to be borrowed
   * @param interestRateMode The interest rate mode at which the user wants to borrow: 1 for Stable,
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   * @param onBehalfOf Address of the user who will receive the debt. Should be the address of the bo
   * calling the function if he wants to borrow against his own collateral, or the address of the cre
   * if he has been given credit delegation allowance
   **/
  function borrow(
    address asset,
    uint256 amount,
    uint256 interestRateMode,
    uint16 referralCode,
    address onBehalfOf
  ) external;

  /**
   * @notice Repays a borrowed `amount` on a specific reserve, burning the equivalent debt tokens own
   * - E.g. User repays 100 USDC, burning 100 variable/stable debt tokens of the `onBehalfOf` address
   * @param asset The address of the borrowed underlying asset previously borrowed
   * @param amount The amount to repay
   * - Send the value type(uint256).max in order to repay the whole debt for `asset` on the specific
   * @param rateMode The interest rate mode at of the debt the user wants to repay: 1 for Stable, 2 f
   * @param onBehalfOf Address of the user who will get his debt reduced/removed. Should be the addre
   * user calling the function if he wants to reduce/remove his own debt, or the address of any other
   * other borrower whose debt should be removed
   * @return The final amount repaid
   **/
  function repay(
    address asset,
    uint256 amount,
    uint256 rateMode,
```

```
    address onBehalfOf
  ) external returns (uint256);

  /**
   * @dev Allows a borrower to swap his debt between stable and variable mode, or viceversa
   * @param asset The address of the underlying asset borrowed
   * @param rateMode The rate mode that the user wants to swap to
   **/
  function swapBorrowRateMode(address asset, uint256 rateMode) external;

  /**
   * @dev Rebalances the stable interest rate of a user to the current stable rate defined on the res
   * - Users can be rebalanced if the following conditions are satisfied:
   *     1. Usage ratio is above 95%
   *     2. the current deposit APY is below REBALANCE_UP_THRESHOLD * maxVariableBorrowRate, which me
   *        borrowed at a stable rate and depositors are not earning enough
   * @param asset The address of the underlying asset borrowed
   * @param user The address of the user to be rebalanced
   **/
  function rebalanceStableBorrowRate(address asset, address user) external;

  /**
   * @dev Allows depositors to enable/disable a specific deposited asset as collateral
   * @param asset The address of the underlying asset deposited
   * @param useAsCollateral `true` if the user wants to use the deposit as collateral, `false` otherw
   **/
  function setUserUseReserveAsCollateral(address asset, bool useAsCollateral) external;

  /**
   * @dev Function to liquidate a non-healthy position collateral-wise, with Health Factor below 1
   * - The caller (liquidator) covers `debtToCover` amount of debt of the user getting liquidated, an
   *   a proportionally amount of the `collateralAsset` plus a bonus to cover market risk
   * @param collateralAsset The address of the underlying asset used as collateral, to receive as res
   * @param debtAsset The address of the underlying borrowed asset to be repaid with the liquidation
   * @param user The address of the borrower getting liquidated
   * @param debtToCover The debt amount of borrowed `asset` the liquidator wants to cover
   * @param receiveAToken `true` if the liquidators wants to receive the collateral aTokens, `false`
   * to receive the underlying collateral asset directly
   **/
  function liquidationCall(
    address collateralAsset,
    address debtAsset,
    address user,
    uint256 debtToCover,
    bool receiveAToken
  ) external;

  /**
   * @dev Allows smartcontracts to access the liquidity of the pool within one transaction,
   * as long as the amount taken plus a fee is returned.
   * IMPORTANT There are security concerns for developers of flashloan receiver contracts that must b
   * For further details please visit https://developers.aave.com
   * @param receiverAddress The address of the contract receiving the funds, implementing the IFlashL
   * @param assets The addresses of the assets being flash-borrowed
   * @param amounts The amounts amounts being flash-borrowed
   * @param modes Types of the debt to open if the flash loan is not returned:
   *   0 -> Don't open any debt, just revert if funds can't be transferred from the receiver
   *   1 -> Open debt at stable rate for the value of the amount flash-borrowed to the `onBehalfOf` a
   *   2 -> Open debt at variable rate for the value of the amount flash-borrowed to the `onBehalfOf`
   * @param onBehalfOf The address  that will receive the debt in the case of using on `modes` 1 or 2
   * @param params Variadic packed params to pass to the receiver as extra information
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   **/
  function flashLoan(
    address receiverAddress,
```

```
    address[] calldata assets,
    uint256[] calldata amounts,
    uint256[] calldata modes,
    address onBehalfOf,
    bytes calldata params,
    uint16 referralCode
  ) external;

  /**
   * @dev Returns the user account data across all the reserves
   * @param user The address of the user
   * @return totalCollateralETH the total collateral in ETH of the user
   * @return totalDebtETH the total debt in ETH of the user
   * @return availableBorrowsETH the borrowing power left of the user
   * @return currentLiquidationThreshold the liquidation threshold of the user
   * @return ltv the loan to value of the user
   * @return healthFactor the current health factor of the user
   **/
  function getUserAccountData(address user)
    external
    view
    returns (
      uint256 totalCollateralETH,
      uint256 totalDebtETH,
      uint256 availableBorrowsETH,
      uint256 currentLiquidationThreshold,
      uint256 ltv,
      uint256 healthFactor
    );

  function initReserve(
    address reserve,
    address aTokenAddress,
    address stableDebtAddress,
    address variableDebtAddress,
    address interestRateStrategyAddress
  ) external;

  function setReserveInterestRateStrategyAddress(address reserve, address rateStrategyAddress)
    external;

  function setConfiguration(address reserve, uint256 configuration) external;

  /**
   * @dev Returns the configuration of the reserve
   * @param asset The address of the underlying asset of the reserve
   * @return The configuration of the reserve
   **/
  function getConfiguration(address asset)
    external
    view
    returns (DataTypes.ReserveConfigurationMap memory);

  /**
   * @dev Returns the configuration of the user across all the reserves
   * @param user The user address
   * @return The configuration of the user
   **/
  function getUserConfiguration(address user)
    external
    view
    returns (DataTypes.UserConfigurationMap memory);

  /**
   * @dev Returns the normalized income normalized income of the reserve
   * @param asset The address of the underlying asset of the reserve
```

```
    * @return The reserve's normalized income
    */
  function getReserveNormalizedIncome(address asset) external view returns (uint256);

  /**
    * @dev Returns the normalized variable debt per unit of asset
    * @param asset The address of the underlying asset of the reserve
    * @return The reserve normalized variable debt
    */
  function getReserveNormalizedVariableDebt(address asset) external view returns (uint256);

  /**
    * @dev Returns the state and configuration of the reserve
    * @param asset The address of the underlying asset of the reserve
    * @return The state of the reserve
    **/
  function getReserveData(address asset) external view returns (DataTypes.ReserveData memory);

  function finalizeTransfer(
    address asset,
    address from,
    address to,
    uint256 amount,
    uint256 balanceFromAfter,
    uint256 balanceToBefore
  ) external;

  function getReservesList() external view returns (address[] memory);

  function getAddressesProvider() external view returns (ILendingPoolAddressesProvider);

  function setPause(bool val) external;

  function paused() external view returns (bool);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IUniswapV2Router02 {
  function swapExactTokensForTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external returns (uint256[] memory amounts);

  function swapTokensForExactTokens(
    uint256 amountOut,
    uint256 amountInMax,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external returns (uint256[] memory amounts);

  function getAmountsOut(uint256 amountIn, address[] calldata path)
    external
    view
    returns (uint256[] memory amounts);

  function getAmountsIn(uint256 amountOut, address[] calldata path)
    external
    view
    returns (uint256[] memory amounts);
}
// SPDX-License-Identifier: agpl-3.0
```

```
pragma solidity 0.6.12;

/**
 * @title LendingPoolAddressesProvider contract
 * @dev Main registry of addresses part of or connected to the protocol, including permissioned roles
 * - Acting also as factory of proxies and admin of those, so with right to change its implementation
 * - Owned by the Aave Governance
 * @author Aave
 **/
interface ILendingPoolAddressesProvider {
  event MarketIdSet(string newMarketId);
  event LendingPoolUpdated(address indexed newAddress);
  event ConfigurationAdminUpdated(address indexed newAddress);
  event EmergencyAdminUpdated(address indexed newAddress);
  event LendingPoolConfiguratorUpdated(address indexed newAddress);
  event LendingPoolCollateralManagerUpdated(address indexed newAddress);
  event PriceOracleUpdated(address indexed newAddress);
  event LendingRateOracleUpdated(address indexed newAddress);
  event ProxyCreated(bytes32 id, address indexed newAddress);
  event AddressSet(bytes32 id, address indexed newAddress, bool hasProxy);

  function getMarketId() external view returns (string memory);

  function setMarketId(string calldata marketId) external;

  function setAddress(bytes32 id, address newAddress) external;

  function setAddressAsProxy(bytes32 id, address impl) external;

  function getAddress(bytes32 id) external view returns (address);

  function getLendingPool() external view returns (address);

  function setLendingPoolImpl(address pool) external;

  function getLendingPoolConfigurator() external view returns (address);

  function setLendingPoolConfiguratorImpl(address configurator) external;

  function getLendingPoolCollateralManager() external view returns (address);

  function setLendingPoolCollateralManager(address manager) external;

  function getPoolAdmin() external view returns (address);

  function setPoolAdmin(address admin) external;

  function getEmergencyAdmin() external view returns (address);

  function setEmergencyAdmin(address admin) external;

  function getPriceOracle() external view returns (address);

  function setPriceOracle(address priceOracle) external;

  function getLendingRateOracle() external view returns (address);

  function setLendingRateOracle(address lendingRateOracle) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

interface IAaveIncentivesController {
  function handleAction(
    address user,
```

```
    uint256 userBalance,
    uint256 totalSupply
  ) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title IDelegationToken
 * @dev Implements an interface for tokens with delegation COMP/UNI compatible
 * @author Aave
 **/
interface IDelegationToken {
  function delegate(address delegatee) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title IReserveInterestRateStrategyInterface interface
 * @dev Interface for the calculation of the interest rates
 * @author Aave
 */
interface IReserveInterestRateStrategy {
  function baseVariableBorrowRate() external view returns (uint256);

  function getMaxVariableBorrowRate() external view returns (uint256);

  function calculateInterestRates(
    address reserve,
    uint256 availableLiquidity,
    uint256 totalStableDebt,
    uint256 totalVariableDebt,
    uint256 averageStableBorrowRate,
    uint256 reserveFactor
  )
    external
    view
    returns (
      uint256,
      uint256,
      uint256
    );

  function calculateInterestRates(
    address reserve,
    address aToken,
    uint256 liquidityAdded,
    uint256 liquidityTaken,
    uint256 totalStableDebt,
    uint256 totalVariableDebt,
    uint256 averageStableBorrowRate,
    uint256 reserveFactor
  )
    external
    view
    returns (
      uint256 liquidityRate,
      uint256 stableBorrowRate,
      uint256 variableBorrowRate
    );
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IUniswapExchange {
```

```
  event TokenPurchase(
    address indexed buyer,
    uint256 indexed eth_sold,
    uint256 indexed tokens_bought
  );
  event EthPurchase(address indexed buyer, uint256 indexed tokens_sold, uint256 indexed eth_bought);
  event AddLiquidity(
    address indexed provider,
    uint256 indexed eth_amount,
    uint256 indexed token_amount
  );
  event RemoveLiquidity(
    address indexed provider,
    uint256 indexed eth_amount,
    uint256 indexed token_amount
  );
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title ILendingPoolCollateralManager
 * @author Aave
 * @notice Defines the actions involving management of collateral in the protocol.
 **/
interface ILendingPoolCollateralManager {
  /**
   * @dev Emitted when a borrower is liquidated
   * @param collateral The address of the collateral being liquidated
   * @param principal The address of the reserve
   * @param user The address of the user being liquidated
   * @param debtToCover The total amount liquidated
   * @param liquidatedCollateralAmount The amount of collateral being liquidated
   * @param liquidator The address of the liquidator
   * @param receiveAToken true if the liquidator wants to receive aTokens, false otherwise
   **/
  event LiquidationCall(
    address indexed collateral,
    address indexed principal,
    address indexed user,
    uint256 debtToCover,
    uint256 liquidatedCollateralAmount,
    address liquidator,
    bool receiveAToken
  );

  /**
   * @dev Emitted when a reserve is disabled as collateral for an user
   * @param reserve The address of the reserve
   * @param user The address of the user
   **/
  event ReserveUsedAsCollateralDisabled(address indexed reserve, address indexed user);

  /**
   * @dev Emitted when a reserve is enabled as collateral for an user
   * @param reserve The address of the reserve
   * @param user The address of the user
   **/
  event ReserveUsedAsCollateralEnabled(address indexed reserve, address indexed user);

  /**
   * @dev Users can invoke this function to liquidate an undercollateralized position.
   * @param collateral The address of the collateral to liquidated
   * @param principal The address of the principal reserve
   * @param user The address of the borrower
   * @param debtToCover The amount of principal that the liquidator wants to repay
```

```
    * @param receiveAToken true if the liquidators wants to receive the aTokens, false if
    * he wants to receive the underlying asset directly
    **/
  function liquidationCall(
    address collateral,
    address principal,
    address user,
    uint256 debtToCover,
    bool receiveAToken
  ) external returns (uint256, string memory);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title ILendingRateOracle interface
 * @notice Interface for the Aave borrow rate oracle. Provides the average market borrow rate to be u
 **/

interface ILendingRateOracle {
  /**
    @dev returns the market borrow rate in ray
    **/
  function getMarketBorrowRate(address asset) external view returns (uint256);

  /**
    @dev sets the market borrow rate. Rate value must be in ray
    **/
  function setMarketBorrowRate(address asset, uint256 rate) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IScaledBalanceToken} from './IScaledBalanceToken.sol';
import {IInitializableDebtToken} from './IInitializableDebtToken.sol';
import {IAaveIncentivesController} from './IAaveIncentivesController.sol';

/**
 * @title IVariableDebtToken
 * @author Aave
 * @notice Defines the basic interface for a variable debt token.
 **/
interface IVariableDebtToken is IScaledBalanceToken, IInitializableDebtToken {
  /**
    * @dev Emitted after the mint action
    * @param from The address performing the mint
    * @param onBehalfOf The address of the user on which behalf minting has been performed
    * @param value The amount to be minted
    * @param index The last index of the reserve
    **/
  event Mint(address indexed from, address indexed onBehalfOf, uint256 value, uint256 index);

  /**
    * @dev Mints debt token to the `onBehalfOf` address
    * @param user The address receiving the borrowed underlying, being the delegatee in case
    * of credit delegate, or same as `onBehalfOf` otherwise
    * @param onBehalfOf The address receiving the debt tokens
    * @param amount The amount of debt being minted
    * @param index The variable debt index of the reserve
    * @return `true` if the the previous balance of the user is 0
    **/
  function mint(
    address user,
    address onBehalfOf,
    uint256 amount,
    uint256 index
```

```
  ) external returns (bool);

  /**
   * @dev Emitted when variable debt is burnt
   * @param user The user which debt has been burned
   * @param amount The amount of debt being burned
   * @param index The index of the user
   **/
  event Burn(address indexed user, uint256 amount, uint256 index);

  /**
   * @dev Burns user variable debt
   * @param user The user which debt is burnt
   * @param index The variable debt index of the reserve
   **/
  function burn(
    address user,
    uint256 amount,
    uint256 index
  ) external;

  /**
   * @dev Returns the address of the incentives controller contract
   **/
  function getIncentivesController() external view returns (IAaveIncentivesController);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ILendingPool} from './ILendingPool.sol';
import {IAaveIncentivesController} from './IAaveIncentivesController.sol';

/**
 * @title IInitializableDebtToken
 * @notice Interface for the initialize function common between debt tokens
 * @author Aave
 **/
interface IInitializableDebtToken {
  /**
   * @dev Emitted when a debt token is initialized
   * @param underlyingAsset The address of the underlying asset
   * @param pool The address of the associated lending pool
   * @param incentivesController The address of the incentives controller for this aToken
   * @param debtTokenDecimals the decimals of the debt token
   * @param debtTokenName the name of the debt token
   * @param debtTokenSymbol the symbol of the debt token
   * @param params A set of encoded parameters for additional initialization
   **/
  event Initialized(
    address indexed underlyingAsset,
    address indexed pool,
    address incentivesController,
    uint8 debtTokenDecimals,
    string debtTokenName,
    string debtTokenSymbol,
    bytes params
  );

  /**
   * @dev Initializes the debt token.
   * @param pool The address of the lending pool where this aToken will be used
   * @param underlyingAsset The address of the underlying asset of this aToken (E.g. WETH for aWETH)
   * @param incentivesController The smart contract managing potential incentives distribution
   * @param debtTokenDecimals The decimals of the debtToken, same as the underlying asset's
   * @param debtTokenName The name of the token
   * @param debtTokenSymbol The symbol of the token
```

```
   */
  function initialize(
    ILendingPool pool,
    address underlyingAsset,
    IAaveIncentivesController incentivesController,
    uint8 debtTokenDecimals,
    string memory debtTokenName,
    string memory debtTokenSymbol,
    bytes calldata params
  ) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IChainlinkAggregator {
  function latestAnswer() external view returns (int256);

  function latestTimestamp() external view returns (uint256);

  function latestRound() external view returns (uint256);

  function getAnswer(uint256 roundId) external view returns (int256);

  function getTimestamp(uint256 roundId) external view returns (uint256);

  event AnswerUpdated(int256 indexed current, uint256 indexed roundId, uint256 timestamp);
  event NewRound(uint256 indexed roundId, address indexed startedBy);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IScaledBalanceToken {
  /**
   * @dev Returns the scaled balance of the user. The scaled balance is the sum of all the
   * updated stored balance divided by the reserve's liquidity index at the moment of the update
   * @param user The user whose balance is calculated
   * @return The scaled balance of the user
   **/
  function scaledBalanceOf(address user) external view returns (uint256);

  /**
   * @dev Returns the scaled balance of the user and the scaled total supply.
   * @param user The address of the user
   * @return The scaled balance of the user
   * @return The scaled balance and the scaled total supply
   **/
  function getScaledUserBalanceAndSupply(address user) external view returns (uint256, uint256);

  /**
   * @dev Returns the scaled total supply of the variable debt token. Represents sum(debt/index)
   * @return The scaled total supply
   **/
  function scaledTotalSupply() external view returns (uint256);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';

interface IERC20WithPermit is IERC20 {
  function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    uint8 v,
```

```
    bytes32 r,
    bytes32 s
  ) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {IPriceOracleGetter} from '../../interfaces/IPriceOracleGetter.sol';
import {IUniswapV2Router02} from '../../interfaces/IUniswapV2Router02.sol';

interface IBaseUniswapAdapter {
  event Swapped(address fromAsset, address toAsset, uint256 fromAmount, uint256 receivedAmount);

  struct PermitSignature {
    uint256 amount;
    uint256 deadline;
    uint8 v;
    bytes32 r;
    bytes32 s;
  }

  struct AmountCalc {
    uint256 calculatedAmount;
    uint256 relativePrice;
    uint256 amountInUsd;
    uint256 amountOutUsd;
    address[] path;
  }

  function WETH_ADDRESS() external returns (address);

  function MAX_SLIPPAGE_PERCENT() external returns (uint256);

  function FLASHLOAN_PREMIUM_TOTAL() external returns (uint256);

  function USD_ADDRESS() external returns (address);

  function ORACLE() external returns (IPriceOracleGetter);

  function UNISWAP_ROUTER() external returns (IUniswapV2Router02);

  /**
   * @dev Given an input asset amount, returns the maximum output amount of the other asset and the p
   * @param amountIn Amount of reserveIn
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @return uint256 Amount out of the reserveOut
   * @return uint256 The price of out amount denominated in the reserveIn currency (18 decimals)
   * @return uint256 In amount of reserveIn value denominated in USD (8 decimals)
   * @return uint256 Out amount of reserveOut value denominated in USD (8 decimals)
   * @return address[] The exchange path
   */
  function getAmountsOut(
    uint256 amountIn,
    address reserveIn,
    address reserveOut
  )
    external
    view
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      address[] memory
```

```
    );

  /**
   * @dev Returns the minimum input asset amount required to buy the given output asset amount and th
   * @param amountOut Amount of reserveOut
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @return uint256 Amount in of the reserveIn
   * @return uint256 The price of in amount denominated in the reserveOut currency (18 decimals)
   * @return uint256 In amount of reserveIn value denominated in USD (8 decimals)
   * @return uint256 Out amount of reserveOut value denominated in USD (8 decimals)
   * @return address[] The exchange path
   */
  function getAmountsIn(
    uint256 amountOut,
    address reserveIn,
    address reserveOut
  )
    external
    view
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      address[] memory
    );
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {BaseUniswapAdapter} from './BaseUniswapAdapter.sol';
import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {IUniswapV2Router02} from '../interfaces/IUniswapV2Router02.sol';
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';

/**
 * @title UniswapLiquiditySwapAdapter
 * @notice Uniswap V2 Adapter to swap liquidity.
 * @author Aave
 **/
contract UniswapLiquiditySwapAdapter is BaseUniswapAdapter {
  struct PermitParams {
    uint256[] amount;
    uint256[] deadline;
    uint8[] v;
    bytes32[] r;
    bytes32[] s;
  }

  struct SwapParams {
    address[] assetToSwapToList;
    uint256[] minAmountsToReceive;
    bool[] swapAllBalance;
    PermitParams permitParams;
    bool[] useEthPath;
  }

  constructor(
    ILendingPoolAddressesProvider addressesProvider,
    IUniswapV2Router02 uniswapRouter,
    address wethAddress
  ) public BaseUniswapAdapter(addressesProvider, uniswapRouter, wethAddress) {}

  /**
```

```
  * @dev Swaps the received reserve amount from the flash loan into the asset specified in the param
  * The received funds from the swap are then deposited into the protocol on behalf of the user.
  * The user should give this contract allowance to pull the ATokens in order to withdraw the underl
  * repay the flash loan.
  * @param assets Address of asset to be swapped
  * @param amounts Amount of the asset to be swapped
  * @param premiums Fee of the flash loan
  * @param initiator Address of the user
  * @param params Additional variadic field to include extra params. Expected parameters:
  *   address[] assetToSwapToList List of the addresses of the reserve to be swapped to and deposite
  *   uint256[] minAmountsToReceive List of min amounts to be received from the swap
  *   bool[] swapAllBalance Flag indicating if all the user balance should be swapped
  *   uint256[] permitAmount List of amounts for the permit signature
  *   uint256[] deadline List of deadlines for the permit signature
  *   uint8[] v List of v param for the permit signature
  *   bytes32[] r List of r param for the permit signature
  *   bytes32[] s List of s param for the permit signature
  */
function executeOperation(
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata premiums,
  address initiator,
  bytes calldata params
) external override returns (bool) {
  require(msg.sender == address(LENDING_POOL), 'CALLER_MUST_BE_LENDING_POOL');

  SwapParams memory decodedParams = _decodeParams(params);

  require(
    assets.length == decodedParams.assetToSwapToList.length &&
      assets.length == decodedParams.minAmountsToReceive.length &&
      assets.length == decodedParams.swapAllBalance.length &&
      assets.length == decodedParams.permitParams.amount.length &&
      assets.length == decodedParams.permitParams.deadline.length &&
      assets.length == decodedParams.permitParams.v.length &&
      assets.length == decodedParams.permitParams.r.length &&
      assets.length == decodedParams.permitParams.s.length &&
      assets.length == decodedParams.useEthPath.length,
    'INCONSISTENT_PARAMS'
  );

  for (uint256 i = 0; i < assets.length; i++) {
    _swapLiquidity(
      assets[i],
      decodedParams.assetToSwapToList[i],
      amounts[i],
      premiums[i],
      initiator,
      decodedParams.minAmountsToReceive[i],
      decodedParams.swapAllBalance[i],
      PermitSignature(
        decodedParams.permitParams.amount[i],
        decodedParams.permitParams.deadline[i],
        decodedParams.permitParams.v[i],
        decodedParams.permitParams.r[i],
        decodedParams.permitParams.s[i]
      ),
      decodedParams.useEthPath[i]
    );
  }

  return true;
}

struct SwapAndDepositLocalVars {
```

```
    uint256 i;
    uint256 aTokenInitiatorBalance;
    uint256 amountToSwap;
    uint256 receivedAmount;
    address aToken;
  }

  /**
   * @dev Swaps an amount of an asset to another and deposits the new asset amount on behalf of the u
   * a flash loan. This method can be used when the temporary transfer of the collateral asset to thi
   * does not affect the user position.
   * The user should give this contract allowance to pull the ATokens in order to withdraw the underl
   * perform the swap.
   * @param assetToSwapFromList List of addresses of the underlying asset to be swap from
   * @param assetToSwapToList List of addresses of the underlying asset to be swap to and deposited
   * @param amountToSwapList List of amounts to be swapped. If the amount exceeds the balance, the to
   * @param minAmountsToReceive List of min amounts to be received from the swap
   * @param permitParams List of struct containing the permit signatures
   *   uint256 permitAmount Amount for the permit signature
   *   uint256 deadline Deadline for the permit signature
   *   uint8 v param for the permit signature
   *   bytes32 r param for the permit signature
   *   bytes32 s param for the permit signature
   * @param useEthPath true if the swap needs to occur using ETH in the routing, false otherwise
   */
  function swapAndDeposit(
    address[] calldata assetToSwapFromList,
    address[] calldata assetToSwapToList,
    uint256[] calldata amountToSwapList,
    uint256[] calldata minAmountsToReceive,
    PermitSignature[] calldata permitParams,
    bool[] calldata useEthPath
  ) external {
    require(
      assetToSwapFromList.length == assetToSwapToList.length &&
        assetToSwapFromList.length == amountToSwapList.length &&
        assetToSwapFromList.length == minAmountsToReceive.length &&
        assetToSwapFromList.length == permitParams.length,
      'INCONSISTENT_PARAMS'
    );

    SwapAndDepositLocalVars memory vars;

    for (vars.i = 0; vars.i < assetToSwapFromList.length; vars.i++) {
      vars.aToken = _getReserveData(assetToSwapFromList[vars.i]).aTokenAddress;

      vars.aTokenInitiatorBalance = IERC20(vars.aToken).balanceOf(msg.sender);
      vars.amountToSwap = amountToSwapList[vars.i] > vars.aTokenInitiatorBalance
        ? vars.aTokenInitiatorBalance
        : amountToSwapList[vars.i];

      _pullAToken(
        assetToSwapFromList[vars.i],
        vars.aToken,
        msg.sender,
        vars.amountToSwap,
        permitParams[vars.i]
      );

      vars.receivedAmount = _swapExactTokensForTokens(
        assetToSwapFromList[vars.i],
        assetToSwapToList[vars.i],
        vars.amountToSwap,
        minAmountsToReceive[vars.i],
        useEthPath[vars.i]
      );
```

```
    // Deposit new reserve
    IERC20(assetToSwapToList[vars.i]).safeApprove(address(LENDING_POOL), 0);
    IERC20(assetToSwapToList[vars.i]).safeApprove(address(LENDING_POOL), vars.receivedAmount);
    LENDING_POOL.deposit(assetToSwapToList[vars.i], vars.receivedAmount, msg.sender, 0);
  }
}

/**
 * @dev Swaps an `amountToSwap` of an asset to another and deposits the funds on behalf of the init
 * @param assetFrom Address of the underlying asset to be swap from
 * @param assetTo Address of the underlying asset to be swap to and deposited
 * @param amount Amount from flash loan
 * @param premium Premium of the flash loan
 * @param minAmountToReceive Min amount to be received from the swap
 * @param swapAllBalance Flag indicating if all the user balance should be swapped
 * @param permitSignature List of struct containing the permit signature
 * @param useEthPath true if the swap needs to occur using ETH in the routing, false otherwise
 */

struct SwapLiquidityLocalVars {
  address aToken;
  uint256 aTokenInitiatorBalance;
  uint256 amountToSwap;
  uint256 receivedAmount;
  uint256 flashLoanDebt;
  uint256 amountToPull;
}

function _swapLiquidity(
  address assetFrom,
  address assetTo,
  uint256 amount,
  uint256 premium,
  address initiator,
  uint256 minAmountToReceive,
  bool swapAllBalance,
  PermitSignature memory permitSignature,
  bool useEthPath
) internal {
  SwapLiquidityLocalVars memory vars;

  vars.aToken = _getReserveData(assetFrom).aTokenAddress;

  vars.aTokenInitiatorBalance = IERC20(vars.aToken).balanceOf(initiator);
  vars.amountToSwap = swapAllBalance && vars.aTokenInitiatorBalance.sub(premium) <= amount
    ? vars.aTokenInitiatorBalance.sub(premium)
    : amount;

  vars.receivedAmount = _swapExactTokensForTokens(
    assetFrom,
    assetTo,
    vars.amountToSwap,
    minAmountToReceive,
    useEthPath
  );

  // Deposit new reserve
  IERC20(assetTo).safeApprove(address(LENDING_POOL), 0);
  IERC20(assetTo).safeApprove(address(LENDING_POOL), vars.receivedAmount);
  LENDING_POOL.deposit(assetTo, vars.receivedAmount, initiator, 0);

  vars.flashLoanDebt = amount.add(premium);
  vars.amountToPull = vars.amountToSwap.add(premium);

  _pullAToken(assetFrom, vars.aToken, initiator, vars.amountToPull, permitSignature);
```

```
    // Repay flash loan
    IERC20(assetFrom).safeApprove(address(LENDING_POOL), 0);
    IERC20(assetFrom).safeApprove(address(LENDING_POOL), vars.flashLoanDebt);
  }

  /**
   * @dev Decodes the information encoded in the flash loan params
   * @param params Additional variadic field to include extra params. Expected parameters:
   *   address[] assetToSwapToList List of the addresses of the reserve to be swapped to and deposite
   *   uint256[] minAmountsToReceive List of min amounts to be received from the swap
   *   bool[] swapAllBalance Flag indicating if all the user balance should be swapped
   *   uint256[] permitAmount List of amounts for the permit signature
   *   uint256[] deadline List of deadlines for the permit signature
   *   uint8[] v List of v param for the permit signature
   *   bytes32[] r List of r param for the permit signature
   *   bytes32[] s List of s param for the permit signature
   *   bool[] useEthPath true if the swap needs to occur using ETH in the routing, false otherwise
   * @return SwapParams struct containing decoded params
   */
  function _decodeParams(bytes memory params) internal pure returns (SwapParams memory) {
    (
      address[] memory assetToSwapToList,
      uint256[] memory minAmountsToReceive,
      bool[] memory swapAllBalance,
      uint256[] memory permitAmount,
      uint256[] memory deadline,
      uint8[] memory v,
      bytes32[] memory r,
      bytes32[] memory s,
      bool[] memory useEthPath
    ) =
      abi.decode(
        params,
        (address[], uint256[], bool[], uint256[], uint256[], uint8[], bytes32[], bytes32[], bool[])
      );

    return
      SwapParams(
        assetToSwapToList,
        minAmountsToReceive,
        swapAllBalance,
        PermitParams(permitAmount, deadline, v, r, s),
        useEthPath
      );
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {BaseUniswapAdapter} from './BaseUniswapAdapter.sol';
import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {IUniswapV2Router02} from '../interfaces/IUniswapV2Router02.sol';
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';

/**
 * @title UniswapRepayAdapter
 * @notice Uniswap V2 Adapter to perform a repay of a debt with collateral.
 * @author Aave
 **/
contract UniswapRepayAdapter is BaseUniswapAdapter {
  struct RepayParams {
    address collateralAsset;
    uint256 collateralAmount;
```

```
    uint256 rateMode;
    PermitSignature permitSignature;
    bool useEthPath;
  }

  constructor(
    ILendingPoolAddressesProvider addressesProvider,
    IUniswapV2Router02 uniswapRouter,
    address wethAddress
  ) public BaseUniswapAdapter(addressesProvider, uniswapRouter, wethAddress) {}

  /**
   * @dev Uses the received funds from the flash loan to repay a debt on the protocol on behalf of th
   * the collateral from the user and swaps it to the debt asset to repay the flash loan.
   * The user should give this contract allowance to pull the ATokens in order to withdraw the underl
   * and repay the flash loan.
   * Supports only one asset on the flash loan.
   * @param assets Address of debt asset
   * @param amounts Amount of the debt to be repaid
   * @param premiums Fee of the flash loan
   * @param initiator Address of the user
   * @param params Additional variadic field to include extra params. Expected parameters:
   *   address collateralAsset Address of the reserve to be swapped
   *   uint256 collateralAmount Amount of reserve to be swapped
   *   uint256 rateMode Rate modes of the debt to be repaid
   *   uint256 permitAmount Amount for the permit signature
   *   uint256 deadline Deadline for the permit signature
   *   uint8 v V param for the permit signature
   *   bytes32 r R param for the permit signature
   *   bytes32 s S param for the permit signature
   */
  function executeOperation(
    address[] calldata assets,
    uint256[] calldata amounts,
    uint256[] calldata premiums,
    address initiator,
    bytes calldata params
  ) external override returns (bool) {
    require(msg.sender == address(LENDING_POOL), 'CALLER_MUST_BE_LENDING_POOL');

    RepayParams memory decodedParams = _decodeParams(params);

    _swapAndRepay(
      decodedParams.collateralAsset,
      assets[0],
      amounts[0],
      decodedParams.collateralAmount,
      decodedParams.rateMode,
      initiator,
      premiums[0],
      decodedParams.permitSignature,
      decodedParams.useEthPath
    );

    return true;
  }

  /**
   * @dev Swaps the user collateral for the debt asset and then repay the debt on the protocol on beh
   * without using flash loans. This method can be used when the temporary transfer of the collateral
   * contract does not affect the user position.
   * The user should give this contract allowance to pull the ATokens in order to withdraw the underl
   * @param collateralAsset Address of asset to be swapped
   * @param debtAsset Address of debt asset
   * @param collateralAmount Amount of the collateral to be swapped
   * @param debtRepayAmount Amount of the debt to be repaid
```

```
   * @param debtRateMode Rate mode of the debt to be repaid
   * @param permitSignature struct containing the permit signature
   * @param useEthPath struct containing the permit signature

   */
  function swapAndRepay(
    address collateralAsset,
    address debtAsset,
    uint256 collateralAmount,
    uint256 debtRepayAmount,
    uint256 debtRateMode,
    PermitSignature calldata permitSignature,
    bool useEthPath
  ) external {
    DataTypes.ReserveData memory collateralReserveData = _getReserveData(collateralAsset);
    DataTypes.ReserveData memory debtReserveData = _getReserveData(debtAsset);

    address debtToken =
      DataTypes.InterestRateMode(debtRateMode) == DataTypes.InterestRateMode.STABLE
        ? debtReserveData.stableDebtTokenAddress
        : debtReserveData.variableDebtTokenAddress;

    uint256 currentDebt = IERC20(debtToken).balanceOf(msg.sender);
    uint256 amountToRepay = debtRepayAmount <= currentDebt ? debtRepayAmount : currentDebt;

    if (collateralAsset != debtAsset) {
      uint256 maxCollateralToSwap = collateralAmount;
      if (amountToRepay < debtRepayAmount) {
        maxCollateralToSwap = maxCollateralToSwap.mul(amountToRepay).div(debtRepayAmount);
      }

      // Get exact collateral needed for the swap to avoid leftovers
      uint256[] memory amounts =
        _getAmountsIn(collateralAsset, debtAsset, amountToRepay, useEthPath);
      require(amounts[0] <= maxCollateralToSwap, 'slippage too high');

      // Pull aTokens from user
      _pullAToken(
        collateralAsset,
        collateralReserveData.aTokenAddress,
        msg.sender,
        amounts[0],
        permitSignature
      );

      // Swap collateral for debt asset
      _swapTokensForExactTokens(collateralAsset, debtAsset, amounts[0], amountToRepay, useEthPath);
    } else {
      // Pull aTokens from user
      _pullAToken(
        collateralAsset,
        collateralReserveData.aTokenAddress,
        msg.sender,
        amountToRepay,
        permitSignature
      );
    }

    // Repay debt. Approves 0 first to comply with tokens that implement the anti frontrunning approv
    IERC20(debtAsset).safeApprove(address(LENDING_POOL), 0);
    IERC20(debtAsset).safeApprove(address(LENDING_POOL), amountToRepay);
    LENDING_POOL.repay(debtAsset, amountToRepay, debtRateMode, msg.sender);
  }

  /**
   * @dev Perform the repay of the debt, pulls the initiator collateral and swaps to repay the flash
```

```
   *
   * @param collateralAsset Address of token to be swapped
   * @param debtAsset Address of debt token to be received from the swap
   * @param amount Amount of the debt to be repaid
   * @param collateralAmount Amount of the reserve to be swapped
   * @param rateMode Rate mode of the debt to be repaid
   * @param initiator Address of the user
   * @param premium Fee of the flash loan
   * @param permitSignature struct containing the permit signature
   */
  function _swapAndRepay(
    address collateralAsset,
    address debtAsset,
    uint256 amount,
    uint256 collateralAmount,
    uint256 rateMode,
    address initiator,
    uint256 premium,
    PermitSignature memory permitSignature,
    bool useEthPath
  ) internal {
    DataTypes.ReserveData memory collateralReserveData = _getReserveData(collateralAsset);

    // Repay debt. Approves for 0 first to comply with tokens that implement the anti frontrunning ap
    IERC20(debtAsset).safeApprove(address(LENDING_POOL), 0);
    IERC20(debtAsset).safeApprove(address(LENDING_POOL), amount);
    uint256 repaidAmount = IERC20(debtAsset).balanceOf(address(this));
    LENDING_POOL.repay(debtAsset, amount, rateMode, initiator);
    repaidAmount = repaidAmount.sub(IERC20(debtAsset).balanceOf(address(this)));

    if (collateralAsset != debtAsset) {
      uint256 maxCollateralToSwap = collateralAmount;
      if (repaidAmount < amount) {
        maxCollateralToSwap = maxCollateralToSwap.mul(repaidAmount).div(amount);
      }

      uint256 neededForFlashLoanDebt = repaidAmount.add(premium);
      uint256[] memory amounts =
        _getAmountsIn(collateralAsset, debtAsset, neededForFlashLoanDebt, useEthPath);
      require(amounts[0] <= maxCollateralToSwap, 'slippage too high');

      // Pull aTokens from user
      _pullAToken(
        collateralAsset,
        collateralReserveData.aTokenAddress,
        initiator,
        amounts[0],
        permitSignature
      );

      // Swap collateral asset to the debt asset
      _swapTokensForExactTokens(
        collateralAsset,
        debtAsset,
        amounts[0],
        neededForFlashLoanDebt,
        useEthPath
      );
    } else {
      // Pull aTokens from user
      _pullAToken(
        collateralAsset,
        collateralReserveData.aTokenAddress,
        initiator,
        repaidAmount.add(premium),
        permitSignature
```

```
      );
    }

    // Repay flashloan. Approves for 0 first to comply with tokens that implement the anti frontrunni
    IERC20(debtAsset).safeApprove(address(LENDING_POOL), 0);
    IERC20(debtAsset).safeApprove(address(LENDING_POOL), amount.add(premium));
  }

  /**
   * @dev Decodes debt information encoded in the flash loan params
   * @param params Additional variadic field to include extra params. Expected parameters:
   *   address collateralAsset Address of the reserve to be swapped
   *   uint256 collateralAmount Amount of reserve to be swapped
   *   uint256 rateMode Rate modes of the debt to be repaid
   *   uint256 permitAmount Amount for the permit signature
   *   uint256 deadline Deadline for the permit signature
   *   uint8 v V param for the permit signature
   *   bytes32 r R param for the permit signature
   *   bytes32 s S param for the permit signature
   *   bool useEthPath use WETH path route
   * @return RepayParams struct containing decoded params
   */
  function _decodeParams(bytes memory params) internal pure returns (RepayParams memory) {
    (
      address collateralAsset,
      uint256 collateralAmount,
      uint256 rateMode,
      uint256 permitAmount,
      uint256 deadline,
      uint8 v,
      bytes32 r,
      bytes32 s,
      bool useEthPath
    ) =
      abi.decode(
        params,
        (address, uint256, uint256, uint256, uint256, uint8, bytes32, bytes32, bool)
      );

    return
      RepayParams(
        collateralAsset,
        collateralAmount,
        rateMode,
        PermitSignature(permitAmount, deadline, v, r, s),
        useEthPath
      );
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {BaseUniswapAdapter} from './BaseUniswapAdapter.sol';
import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {IUniswapV2Router02} from '../interfaces/IUniswapV2Router02.sol';
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';
import {Helpers} from '../protocol/libraries/helpers/Helpers.sol';
import {IPriceOracleGetter} from '../interfaces/IPriceOracleGetter.sol';
import {IAToken} from '../interfaces/IAToken.sol';
import {ReserveConfiguration} from '../protocol/libraries/configuration/ReserveConfiguration.sol';

/**
 * @title UniswapLiquiditySwapAdapter
 * @notice Uniswap V2 Adapter to swap liquidity.
```

```
 * @author Aave
 **/
contract FlashLiquidationAdapter is BaseUniswapAdapter {
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  uint256 internal constant LIQUIDATION_CLOSE_FACTOR_PERCENT = 5000;

  struct LiquidationParams {
    address collateralAsset;
    address borrowedAsset;
    address user;
    uint256 debtToCover;
    bool useEthPath;
  }

  struct LiquidationCallLocalVars {
    uint256 initFlashBorrowedBalance;
    uint256 diffFlashBorrowedBalance;
    uint256 initCollateralBalance;
    uint256 diffCollateralBalance;
    uint256 flashLoanDebt;
    uint256 soldAmount;
    uint256 remainingTokens;
    uint256 borrowedAssetLeftovers;
  }

  constructor(
    ILendingPoolAddressesProvider addressesProvider,
    IUniswapV2Router02 uniswapRouter,
    address wethAddress
  ) public BaseUniswapAdapter(addressesProvider, uniswapRouter, wethAddress) {}

  /**
   * @dev Liquidate a non-healthy position collateral-wise, with a Health Factor below 1, using Flash
   * - The caller (liquidator) with a flash loan covers `debtToCover` amount of debt of the user gett
   *   a proportionally amount of the `collateralAsset` plus a bonus to cover market risk minus the f
   * @param assets Address of asset to be swapped
   * @param amounts Amount of the asset to be swapped
   * @param premiums Fee of the flash loan
   * @param initiator Address of the caller
   * @param params Additional variadic field to include extra params. Expected parameters:
   *   address collateralAsset The collateral asset to release and will be exchanged to pay the flash
   *   address borrowedAsset The asset that must be covered
   *   address user The user address with a Health Factor below 1
   *   uint256 debtToCover The amount of debt to cover
   *   bool useEthPath Use WETH as connector path between the collateralAsset and borrowedAsset at Un
   */
  function executeOperation(
    address[] calldata assets,
    uint256[] calldata amounts,
    uint256[] calldata premiums,
    address initiator,
    bytes calldata params
  ) external override returns (bool) {
    require(msg.sender == address(LENDING_POOL), 'CALLER_MUST_BE_LENDING_POOL');

    LiquidationParams memory decodedParams = _decodeParams(params);

    require(assets.length == 1 && assets[0] == decodedParams.borrowedAsset, 'INCONSISTENT_PARAMS');

    _liquidateAndSwap(
      decodedParams.collateralAsset,
      decodedParams.borrowedAsset,
      decodedParams.user,
      decodedParams.debtToCover,
      decodedParams.useEthPath,
      amounts[0],
```

```
      premiums[0],
      initiator
    );

    return true;
  }

  /**
   * @dev
   * @param collateralAsset The collateral asset to release and will be exchanged to pay the flash lo
   * @param borrowedAsset The asset that must be covered
   * @param user The user address with a Health Factor below 1
   * @param debtToCover The amount of debt to coverage, can be max(-1) to liquidate all possible debt
   * @param useEthPath true if the swap needs to occur using ETH in the routing, false otherwise
   * @param flashBorrowedAmount Amount of asset requested at the flash loan to liquidate the user pos
   * @param premium Fee of the requested flash loan
   * @param initiator Address of the caller
   */
  function _liquidateAndSwap(
    address collateralAsset,
    address borrowedAsset,
    address user,
    uint256 debtToCover,
    bool useEthPath,
    uint256 flashBorrowedAmount,
    uint256 premium,
    address initiator
  ) internal {
    LiquidationCallLocalVars memory vars;
    vars.initCollateralBalance = IERC20(collateralAsset).balanceOf(address(this));
    if (collateralAsset != borrowedAsset) {
      vars.initFlashBorrowedBalance = IERC20(borrowedAsset).balanceOf(address(this));

      // Track leftover balance to rescue funds in case of external transfers into this contract
      vars.borrowedAssetLeftovers = vars.initFlashBorrowedBalance.sub(flashBorrowedAmount);
    }
    vars.flashLoanDebt = flashBorrowedAmount.add(premium);

    // Approve LendingPool to use debt token for liquidation
    IERC20(borrowedAsset).approve(address(LENDING_POOL), debtToCover);

    // Liquidate the user position and release the underlying collateral
    LENDING_POOL.liquidationCall(collateralAsset, borrowedAsset, user, debtToCover, false);

    // Discover the liquidated tokens
    uint256 collateralBalanceAfter = IERC20(collateralAsset).balanceOf(address(this));

    // Track only collateral released, not current asset balance of the contract
    vars.diffCollateralBalance = collateralBalanceAfter.sub(vars.initCollateralBalance);

    if (collateralAsset != borrowedAsset) {
      // Discover flash loan balance after the liquidation
      uint256 flashBorrowedAssetAfter = IERC20(borrowedAsset).balanceOf(address(this));

      // Use only flash loan borrowed assets, not current asset balance of the contract
      vars.diffFlashBorrowedBalance = flashBorrowedAssetAfter.sub(vars.borrowedAssetLeftovers);

      // Swap released collateral into the debt asset, to repay the flash loan
      vars.soldAmount = _swapTokensForExactTokens(
        collateralAsset,
        borrowedAsset,
        vars.diffCollateralBalance,
        vars.flashLoanDebt.sub(vars.diffFlashBorrowedBalance),
        useEthPath
      );
      vars.remainingTokens = vars.diffCollateralBalance.sub(vars.soldAmount);
```

```
    } else {
      vars.remainingTokens = vars.diffCollateralBalance.sub(premium);
    }

    // Allow repay of flash loan
    IERC20(borrowedAsset).approve(address(LENDING_POOL), vars.flashLoanDebt);

    // Transfer remaining tokens to initiator
    if (vars.remainingTokens > 0) {
      IERC20(collateralAsset).transfer(initiator, vars.remainingTokens);
    }
  }

  /**
   * @dev Decodes the information encoded in the flash loan params
   * @param params Additional variadic field to include extra params. Expected parameters:
   *   address collateralAsset The collateral asset to claim
   *   address borrowedAsset The asset that must be covered and will be exchanged to pay the flash lo
   *   address user The user address with a Health Factor below 1
   *   uint256 debtToCover The amount of debt to cover
   *   bool useEthPath Use WETH as connector path between the collateralAsset and borrowedAsset at Un
   * @return LiquidationParams struct containing decoded params
   */
  function _decodeParams(bytes memory params) internal pure returns (LiquidationParams memory) {
    (
      address collateralAsset,
      address borrowedAsset,
      address user,
      uint256 debtToCover,
      bool useEthPath
    ) = abi.decode(params, (address, address, address, uint256, bool));

    return LiquidationParams(collateralAsset, borrowedAsset, user, debtToCover, useEthPath);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {PercentageMath} from '../protocol/libraries/math/PercentageMath.sol';
import {SafeMath} from '../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';
import {IERC20Detailed} from '../dependencies/openzeppelin/contracts/IERC20Detailed.sol';
import {SafeERC20} from '../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {Ownable} from '../dependencies/openzeppelin/contracts/Ownable.sol';
import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';
import {IUniswapV2Router02} from '../interfaces/IUniswapV2Router02.sol';
import {IPriceOracleGetter} from '../interfaces/IPriceOracleGetter.sol';
import {IERC20WithPermit} from '../interfaces/IERC20WithPermit.sol';
import {FlashLoanReceiverBase} from '../flashloan/base/FlashLoanReceiverBase.sol';
import {IBaseUniswapAdapter} from './interfaces/IBaseUniswapAdapter.sol';

/**
 * @title BaseUniswapAdapter
 * @notice Implements the logic for performing assets swaps in Uniswap V2
 * @author Aave
 **/
abstract contract BaseUniswapAdapter is FlashLoanReceiverBase, IBaseUniswapAdapter, Ownable {
  using SafeMath for uint256;
  using PercentageMath for uint256;
  using SafeERC20 for IERC20;

  // Max slippage percent allowed
  uint256 public constant override MAX_SLIPPAGE_PERCENT = 3000; // 30%
  // FLash Loan fee set in lending pool
```

```
  uint256 public constant override FLASHLOAN_PREMIUM_TOTAL = 9;
  // USD oracle asset address
  address public constant override USD_ADDRESS = 0x10F7Fc1F91Ba351f9C629c5947AD69bD03C05b96;

  address public immutable override WETH_ADDRESS;
  IPriceOracleGetter public immutable override ORACLE;
  IUniswapV2Router02 public immutable override UNISWAP_ROUTER;

  constructor(
    ILendingPoolAddressesProvider addressesProvider,
    IUniswapV2Router02 uniswapRouter,
    address wethAddress
  ) public FlashLoanReceiverBase(addressesProvider) {
    ORACLE = IPriceOracleGetter(addressesProvider.getPriceOracle());
    UNISWAP_ROUTER = uniswapRouter;
    WETH_ADDRESS = wethAddress;
  }

  /**
   * @dev Given an input asset amount, returns the maximum output amount of the other asset and the p
   * @param amountIn Amount of reserveIn
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @return uint256 Amount out of the reserveOut
   * @return uint256 The price of out amount denominated in the reserveIn currency (18 decimals)
   * @return uint256 In amount of reserveIn value denominated in USD (8 decimals)
   * @return uint256 Out amount of reserveOut value denominated in USD (8 decimals)
   */
  function getAmountsOut(
    uint256 amountIn,
    address reserveIn,
    address reserveOut
  )
    external
    view
    override
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      address[] memory
    )
  {
    AmountCalc memory results = _getAmountsOutData(reserveIn, reserveOut, amountIn);

    return (
      results.calculatedAmount,
      results.relativePrice,
      results.amountInUsd,
      results.amountOutUsd,
      results.path
    );
  }

  /**
   * @dev Returns the minimum input asset amount required to buy the given output asset amount and th
   * @param amountOut Amount of reserveOut
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @return uint256 Amount in of the reserveIn
   * @return uint256 The price of in amount denominated in the reserveOut currency (18 decimals)
   * @return uint256 In amount of reserveIn value denominated in USD (8 decimals)
   * @return uint256 Out amount of reserveOut value denominated in USD (8 decimals)
   */
  function getAmountsIn(
```

```
    uint256 amountOut,
    address reserveIn,
    address reserveOut
  )
    external
    view
    override
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      address[] memory
    )
  {
    AmountCalc memory results = _getAmountsInData(reserveIn, reserveOut, amountOut);

    return (
      results.calculatedAmount,
      results.relativePrice,
      results.amountInUsd,
      results.amountOutUsd,
      results.path
    );
  }

  /**
   * @dev Swaps an exact `amountToSwap` of an asset to another
   * @param assetToSwapFrom Origin asset
   * @param assetToSwapTo Destination asset
   * @param amountToSwap Exact amount of `assetToSwapFrom` to be swapped
   * @param minAmountOut the min amount of `assetToSwapTo` to be received from the swap
   * @return the amount received from the swap
   */
  function _swapExactTokensForTokens(
    address assetToSwapFrom,
    address assetToSwapTo,
    uint256 amountToSwap,
    uint256 minAmountOut,
    bool useEthPath
  ) internal returns (uint256) {
    uint256 fromAssetDecimals = _getDecimals(assetToSwapFrom);
    uint256 toAssetDecimals = _getDecimals(assetToSwapTo);

    uint256 fromAssetPrice = _getPrice(assetToSwapFrom);
    uint256 toAssetPrice = _getPrice(assetToSwapTo);

    uint256 expectedMinAmountOut =
      amountToSwap
        .mul(fromAssetPrice.mul(10**toAssetDecimals))
        .div(toAssetPrice.mul(10**fromAssetDecimals))
        .percentMul(PercentageMath.PERCENTAGE_FACTOR.sub(MAX_SLIPPAGE_PERCENT));

    require(expectedMinAmountOut < minAmountOut, 'minAmountOut exceed max slippage');

    // Approves the transfer for the swap. Approves for 0 first to comply with tokens that implement
    IERC20(assetToSwapFrom).safeApprove(address(UNISWAP_ROUTER), 0);
    IERC20(assetToSwapFrom).safeApprove(address(UNISWAP_ROUTER), amountToSwap);

    address[] memory path;
    if (useEthPath) {
      path = new address[](3);
      path[0] = assetToSwapFrom;
      path[1] = WETH_ADDRESS;
      path[2] = assetToSwapTo;
    } else {
```

```
      path = new address[](2);
      path[0] = assetToSwapFrom;
      path[1] = assetToSwapTo;
    }
    uint256[] memory amounts =
      UNISWAP_ROUTER.swapExactTokensForTokens(
        amountToSwap,
        minAmountOut,
        path,
        address(this),
        block.timestamp
      );

    emit Swapped(assetToSwapFrom, assetToSwapTo, amounts[0], amounts[amounts.length - 1]);

    return amounts[amounts.length - 1];
  }

  /**
   * @dev Receive an exact amount `amountToReceive` of `assetToSwapTo` tokens for as few `assetToSwap
   * possible.
   * @param assetToSwapFrom Origin asset
   * @param assetToSwapTo Destination asset
   * @param maxAmountToSwap Max amount of `assetToSwapFrom` allowed to be swapped
   * @param amountToReceive Exact amount of `assetToSwapTo` to receive
   * @return the amount swapped
   */
  function _swapTokensForExactTokens(
    address assetToSwapFrom,
    address assetToSwapTo,
    uint256 maxAmountToSwap,
    uint256 amountToReceive,
    bool useEthPath
  ) internal returns (uint256) {
    uint256 fromAssetDecimals = _getDecimals(assetToSwapFrom);
    uint256 toAssetDecimals = _getDecimals(assetToSwapTo);

    uint256 fromAssetPrice = _getPrice(assetToSwapFrom);
    uint256 toAssetPrice = _getPrice(assetToSwapTo);

    uint256 expectedMaxAmountToSwap =
      amountToReceive
        .mul(toAssetPrice.mul(10**fromAssetDecimals))
        .div(fromAssetPrice.mul(10**toAssetDecimals))
        .percentMul(PercentageMath.PERCENTAGE_FACTOR.add(MAX_SLIPPAGE_PERCENT));

    require(maxAmountToSwap < expectedMaxAmountToSwap, 'maxAmountToSwap exceed max slippage');

    // Approves the transfer for the swap. Approves for 0 first to comply with tokens that implement
    IERC20(assetToSwapFrom).safeApprove(address(UNISWAP_ROUTER), 0);
    IERC20(assetToSwapFrom).safeApprove(address(UNISWAP_ROUTER), maxAmountToSwap);

    address[] memory path;
    if (useEthPath) {
      path = new address[](3);
      path[0] = assetToSwapFrom;
      path[1] = WETH_ADDRESS;
      path[2] = assetToSwapTo;
    } else {
      path = new address[](2);
      path[0] = assetToSwapFrom;
      path[1] = assetToSwapTo;
    }

    uint256[] memory amounts =
      UNISWAP_ROUTER.swapTokensForExactTokens(
```

```
      amountToReceive,
      maxAmountToSwap,
      path,
      address(this),
      block.timestamp
    );

  emit Swapped(assetToSwapFrom, assetToSwapTo, amounts[0], amounts[amounts.length - 1]);

  return amounts[0];
}

/**
 * @dev Get the price of the asset from the oracle denominated in eth
 * @param asset address
 * @return eth price for the asset
 */
function _getPrice(address asset) internal view returns (uint256) {
  return ORACLE.getAssetPrice(asset);
}

/**
 * @dev Get the decimals of an asset
 * @return number of decimals of the asset
 */
function _getDecimals(address asset) internal view returns (uint256) {
  return IERC20Detailed(asset).decimals();
}

/**
 * @dev Get the aToken associated to the asset
 * @return address of the aToken
 */
function _getReserveData(address asset) internal view returns (DataTypes.ReserveData memory) {
  return LENDING_POOL.getReserveData(asset);
}

/**
 * @dev Pull the ATokens from the user
 * @param reserve address of the asset
 * @param reserveAToken address of the aToken of the reserve
 * @param user address
 * @param amount of tokens to be transferred to the contract
 * @param permitSignature struct containing the permit signature
 */
function _pullAToken(
  address reserve,
  address reserveAToken,
  address user,
  uint256 amount,
  PermitSignature memory permitSignature
) internal {
  if (_usePermit(permitSignature)) {
    IERC20WithPermit(reserveAToken).permit(
      user,
      address(this),
      permitSignature.amount,
      permitSignature.deadline,
      permitSignature.v,
      permitSignature.r,
      permitSignature.s
    );
  }

  // transfer from user to adapter
  IERC20(reserveAToken).safeTransferFrom(user, address(this), amount);
```

```
  // withdraw reserve
  LENDING_POOL.withdraw(reserve, amount, address(this));
}

/**
 * @dev Tells if the permit method should be called by inspecting if there is a valid signature.
 * If signature params are set to 0, then permit won't be called.
 * @param signature struct containing the permit signature
 * @return whether or not permit should be called
 */
function _usePermit(PermitSignature memory signature) internal pure returns (bool) {
  return
    !(uint256(signature.deadline) == uint256(signature.v) && uint256(signature.deadline) == 0);
}

/**
 * @dev Calculates the value denominated in USD
 * @param reserve Address of the reserve
 * @param amount Amount of the reserve
 * @param decimals Decimals of the reserve
 * @return whether or not permit should be called
 */
function _calcUsdValue(
  address reserve,
  uint256 amount,
  uint256 decimals
) internal view returns (uint256) {
  uint256 ethUsdPrice = _getPrice(USD_ADDRESS);
  uint256 reservePrice = _getPrice(reserve);

  return amount.mul(reservePrice).div(10**decimals).mul(ethUsdPrice).div(10**18);
}

/**
 * @dev Given an input asset amount, returns the maximum output amount of the other asset
 * @param reserveIn Address of the asset to be swap from
 * @param reserveOut Address of the asset to be swap to
 * @param amountIn Amount of reserveIn
 * @return Struct containing the following information:
 *   uint256 Amount out of the reserveOut
 *   uint256 The price of out amount denominated in the reserveIn currency (18 decimals)
 *   uint256 In amount of reserveIn value denominated in USD (8 decimals)
 *   uint256 Out amount of reserveOut value denominated in USD (8 decimals)
 */
function _getAmountsOutData(
  address reserveIn,
  address reserveOut,
  uint256 amountIn
) internal view returns (AmountCalc memory) {
  // Subtract flash loan fee
  uint256 finalAmountIn = amountIn.sub(amountIn.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000));

  if (reserveIn == reserveOut) {
    uint256 reserveDecimals = _getDecimals(reserveIn);
    address[] memory path = new address[](1);
    path[0] = reserveIn;

    return
      AmountCalc(
        finalAmountIn,
        finalAmountIn.mul(10**18).div(amountIn),
        _calcUsdValue(reserveIn, amountIn, reserveDecimals),
        _calcUsdValue(reserveIn, finalAmountIn, reserveDecimals),
        path
      );
```

```
    }

    address[] memory simplePath = new address[](2);
    simplePath[0] = reserveIn;
    simplePath[1] = reserveOut;

    uint256[] memory amountsWithoutWeth;
    uint256[] memory amountsWithWeth;

    address[] memory pathWithWeth = new address[](3);
    if (reserveIn != WETH_ADDRESS && reserveOut != WETH_ADDRESS) {
      pathWithWeth[0] = reserveIn;
      pathWithWeth[1] = WETH_ADDRESS;
      pathWithWeth[2] = reserveOut;

      try UNISWAP_ROUTER.getAmountsOut(finalAmountIn, pathWithWeth) returns (
        uint256[] memory resultsWithWeth
      ) {
        amountsWithWeth = resultsWithWeth;
      } catch {
        amountsWithWeth = new uint256[](3);
      }
    } else {
      amountsWithWeth = new uint256[](3);
    }

    uint256 bestAmountOut;
    try UNISWAP_ROUTER.getAmountsOut(finalAmountIn, simplePath) returns (
      uint256[] memory resultAmounts
    ) {
      amountsWithoutWeth = resultAmounts;

      bestAmountOut = (amountsWithWeth[2] > amountsWithoutWeth[1])
        ? amountsWithWeth[2]
        : amountsWithoutWeth[1];
    } catch {
      amountsWithoutWeth = new uint256[](2);
      bestAmountOut = amountsWithWeth[2];
    }

    uint256 reserveInDecimals = _getDecimals(reserveIn);
    uint256 reserveOutDecimals = _getDecimals(reserveOut);

    uint256 outPerInPrice =
      finalAmountIn.mul(10**18).mul(10**reserveOutDecimals).div(
        bestAmountOut.mul(10**reserveInDecimals)
      );

    return
      AmountCalc(
        bestAmountOut,
        outPerInPrice,
        _calcUsdValue(reserveIn, amountIn, reserveInDecimals),
        _calcUsdValue(reserveOut, bestAmountOut, reserveOutDecimals),
        (bestAmountOut == 0) ? new address[](2) : (bestAmountOut == amountsWithoutWeth[1])
          ? simplePath
          : pathWithWeth
      );
  }

  /**
   * @dev Returns the minimum input asset amount required to buy the given output asset amount
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @param amountOut Amount of reserveOut
   * @return Struct containing the following information:
```

```
   *   uint256 Amount in of the reserveIn
   *   uint256 The price of in amount denominated in the reserveOut currency (18 decimals)
   *   uint256 In amount of reserveIn value denominated in USD (8 decimals)
   *   uint256 Out amount of reserveOut value denominated in USD (8 decimals)
   */
  function _getAmountsInData(
    address reserveIn,
    address reserveOut,
    uint256 amountOut
  ) internal view returns (AmountCalc memory) {
    if (reserveIn == reserveOut) {
      // Add flash loan fee
      uint256 amountIn = amountOut.add(amountOut.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000));
      uint256 reserveDecimals = _getDecimals(reserveIn);
      address[] memory path = new address[](1);
      path[0] = reserveIn;

      return
        AmountCalc(
          amountIn,
          amountOut.mul(10**18).div(amountIn),
          _calcUsdValue(reserveIn, amountIn, reserveDecimals),
          _calcUsdValue(reserveIn, amountOut, reserveDecimals),
          path
        );
    }

    (uint256[] memory amounts, address[] memory path) =
      _getAmountsInAndPath(reserveIn, reserveOut, amountOut);

    // Add flash loan fee
    uint256 finalAmountIn = amounts[0].add(amounts[0].mul(FLASHLOAN_PREMIUM_TOTAL).div(10000));

    uint256 reserveInDecimals = _getDecimals(reserveIn);
    uint256 reserveOutDecimals = _getDecimals(reserveOut);

    uint256 inPerOutPrice =
      amountOut.mul(10**18).mul(10**reserveInDecimals).div(
        finalAmountIn.mul(10**reserveOutDecimals)
      );

    return
      AmountCalc(
        finalAmountIn,
        inPerOutPrice,
        _calcUsdValue(reserveIn, finalAmountIn, reserveInDecimals),
        _calcUsdValue(reserveOut, amountOut, reserveOutDecimals),
        path
      );
  }

  /**
   * @dev Calculates the input asset amount required to buy the given output asset amount
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @param amountOut Amount of reserveOut
   * @return uint256[] amounts Array containing the amountIn and amountOut for a swap
   */
  function _getAmountsInAndPath(
    address reserveIn,
    address reserveOut,
    uint256 amountOut
  ) internal view returns (uint256[] memory, address[] memory) {
    address[] memory simplePath = new address[](2);
    simplePath[0] = reserveIn;
    simplePath[1] = reserveOut;
```

```
    uint256[] memory amountsWithoutWeth;
    uint256[] memory amountsWithWeth;
    address[] memory pathWithWeth = new address[](3);

    if (reserveIn != WETH_ADDRESS && reserveOut != WETH_ADDRESS) {
      pathWithWeth[0] = reserveIn;
      pathWithWeth[1] = WETH_ADDRESS;
      pathWithWeth[2] = reserveOut;

      try UNISWAP_ROUTER.getAmountsIn(amountOut, pathWithWeth) returns (
        uint256[] memory resultsWithWeth
      ) {
        amountsWithWeth = resultsWithWeth;
      } catch {
        amountsWithWeth = new uint256[](3);
      }
    } else {
      amountsWithWeth = new uint256[](3);
    }

    try UNISWAP_ROUTER.getAmountsIn(amountOut, simplePath) returns (
      uint256[] memory resultAmounts
    ) {
      amountsWithoutWeth = resultAmounts;

      return
        (amountsWithWeth[0] < amountsWithoutWeth[0] && amountsWithWeth[0] != 0)
          ? (amountsWithWeth, pathWithWeth)
          : (amountsWithoutWeth, simplePath);
    } catch {
      return (amountsWithWeth, pathWithWeth);
    }
  }

  /**
   * @dev Calculates the input asset amount required to buy the given output asset amount
   * @param reserveIn Address of the asset to be swap from
   * @param reserveOut Address of the asset to be swap to
   * @param amountOut Amount of reserveOut
   * @return uint256[] amounts Array containing the amountIn and amountOut for a swap
   */
  function _getAmountsIn(
    address reserveIn,
    address reserveOut,
    uint256 amountOut,
    bool useEthPath
  ) internal view returns (uint256[] memory) {
    address[] memory path;

    if (useEthPath) {
      path = new address[](3);
      path[0] = reserveIn;
      path[1] = WETH_ADDRESS;
      path[2] = reserveOut;
    } else {
      path = new address[](2);
      path[0] = reserveIn;
      path[1] = reserveOut;
    }

    return UNISWAP_ROUTER.getAmountsIn(amountOut, path);
  }

  /**
   * @dev Emergency rescue for token stucked on this contract, as failsafe mechanism
```

```
   * - Funds should never remain in this contract more time than during transactions
   * - Only callable by the owner
   **/
  function rescueTokens(IERC20 token) external onlyOwner {
    token.transfer(owner(), token.balanceOf(address(this)));
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';
import {SafeERC20} from '../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {Address} from '../../dependencies/openzeppelin/contracts/Address.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {IAToken} from '../../interfaces/IAToken.sol';
import {IVariableDebtToken} from '../../interfaces/IVariableDebtToken.sol';
import {IFlashLoanReceiver} from '../../flashloan/interfaces/IFlashLoanReceiver.sol';
import {IPriceOracleGetter} from '../../interfaces/IPriceOracleGetter.sol';
import {IStableDebtToken} from '../../interfaces/IStableDebtToken.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {Helpers} from '../libraries/helpers/Helpers.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {ReserveLogic} from '../libraries/logic/ReserveLogic.sol';
import {GenericLogic} from '../libraries/logic/GenericLogic.sol';
import {ValidationLogic} from '../libraries/logic/ValidationLogic.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../libraries/configuration/UserConfiguration.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';
import {LendingPoolStorage} from './LendingPoolStorage.sol';

/**
 * @title LendingPool contract
 * @dev Main point of interaction with an Aave protocol's market
 * - Users can:
 *   # Deposit
 *   # Withdraw
 *   # Borrow
 *   # Repay
 *   # Swap their loans between variable and stable rate
 *   # Enable/disable their deposits as collateral rebalance stable rate borrow positions
 *   # Liquidate positions
 *   # Execute Flash Loans
 * - To be covered by a proxy contract, owned by the LendingPoolAddressesProvider of the specific mar
 * - All admin functions are callable by the LendingPoolConfigurator contract defined also in the
 *   LendingPoolAddressesProvider
 * @author Aave
 **/
contract LendingPool is VersionedInitializable, ILendingPool, LendingPoolStorage {
  using SafeMath for uint256;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using SafeERC20 for IERC20;

  uint256 public constant LENDINGPOOL_REVISION = 0x2;

  modifier whenNotPaused() {
    _whenNotPaused();
    _;
  }

  modifier onlyLendingPoolConfigurator() {
```

```
    _onlyLendingPoolConfigurator();
    _;
  }

  function _whenNotPaused() internal view {
    require(!_paused, Errors.LP_IS_PAUSED);
  }

  function _onlyLendingPoolConfigurator() internal view {
    require(
      _addressesProvider.getLendingPoolConfigurator() == msg.sender,
      Errors.LP_CALLER_NOT_LENDING_POOL_CONFIGURATOR
    );
  }

  function getRevision() internal pure override returns (uint256) {
    return LENDINGPOOL_REVISION;
  }

  /**
   * @dev Function is invoked by the proxy contract when the LendingPool contract is added to the
   * LendingPoolAddressesProvider of the market.
   * - Caching the address of the LendingPoolAddressesProvider in order to reduce gas consumption
   *   on subsequent operations
   * @param provider The address of the LendingPoolAddressesProvider
   **/
  function initialize(ILendingPoolAddressesProvider provider) public initializer {
    _addressesProvider = provider;
    _maxStableRateBorrowSizePercent = 2500;
    _flashLoanPremiumTotal = 9;
    _maxNumberOfReserves = 128;
  }

  /**
   * @dev Deposits an `amount` of underlying asset into the reserve, receiving in return overlying aT
   * - E.g. User deposits 100 USDC and gets in return 100 aUSDC
   * @param asset The address of the underlying asset to deposit
   * @param amount The amount to be deposited
   * @param onBehalfOf The address that will receive the aTokens, same as msg.sender if the user
   *   wants to receive them on his own wallet, or a different address if the beneficiary of aTokens
   *   is a different wallet
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   **/
  function deposit(
    address asset,
    uint256 amount,
    address onBehalfOf,
    uint16 referralCode
  ) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    ValidationLogic.validateDeposit(reserve, amount);

    address aToken = reserve.aTokenAddress;

    reserve.updateState();
    reserve.updateInterestRates(asset, aToken, amount, 0);

    IERC20(asset).safeTransferFrom(msg.sender, aToken, amount);

    bool isFirstDeposit = IAToken(aToken).mint(onBehalfOf, amount, reserve.liquidityIndex);

    if (isFirstDeposit) {
      _usersConfig[onBehalfOf].setUsingAsCollateral(reserve.id, true);
      emit ReserveUsedAsCollateralEnabled(asset, onBehalfOf);
```

```
    }

    emit Deposit(asset, msg.sender, onBehalfOf, amount, referralCode);
  }

  /**
   * @dev Withdraws an `amount` of underlying asset from the reserve, burning the equivalent aTokens
   * E.g. User has 100 aUSDC, calls withdraw() and receives 100 USDC, burning the 100 aUSDC
   * @param asset The address of the underlying asset to withdraw
   * @param amount The underlying amount to be withdrawn
   *   - Send the value type(uint256).max in order to withdraw the whole aToken balance
   * @param to Address that will receive the underlying, same as msg.sender if the user
   *   wants to receive it on his own wallet, or a different address if the beneficiary is a
   *   different wallet
   * @return The final amount withdrawn
   **/
  function withdraw(
    address asset,
    uint256 amount,
    address to
  ) external override whenNotPaused returns (uint256) {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    address aToken = reserve.aTokenAddress;

    uint256 userBalance = IAToken(aToken).balanceOf(msg.sender);

    uint256 amountToWithdraw = amount;

    if (amount == type(uint256).max) {
      amountToWithdraw = userBalance;
    }

    ValidationLogic.validateWithdraw(
      asset,
      amountToWithdraw,
      userBalance,
      _reserves,
      _usersConfig[msg.sender],
      _reservesList,
      _reservesCount,
      _addressesProvider.getPriceOracle()
    );

    reserve.updateState();

    reserve.updateInterestRates(asset, aToken, 0, amountToWithdraw);

    if (amountToWithdraw == userBalance) {
      _usersConfig[msg.sender].setUsingAsCollateral(reserve.id, false);
      emit ReserveUsedAsCollateralDisabled(asset, msg.sender);
    }

    IAToken(aToken).burn(msg.sender, to, amountToWithdraw, reserve.liquidityIndex);

    emit Withdraw(asset, msg.sender, to, amountToWithdraw);

    return amountToWithdraw;
  }

  /**
   * @dev Allows users to borrow a specific `amount` of the reserve underlying asset, provided that t
   * already deposited enough collateral, or he was given enough allowance by a credit delegator on t
   * corresponding debt token (StableDebtToken or VariableDebtToken)
   * - E.g. User borrows 100 USDC passing as `onBehalfOf` his own address, receiving the 100 USDC in
   *   and 100 stable/variable debt tokens, depending on the `interestRateMode`
```

```
   * @param asset The address of the underlying asset to borrow
   * @param amount The amount to be borrowed
   * @param interestRateMode The interest rate mode at which the user wants to borrow: 1 for Stable,
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   * @param onBehalfOf Address of the user who will receive the debt. Should be the address of the bo
   * calling the function if he wants to borrow against his own collateral, or the address of the cre
   * if he has been given credit delegation allowance
   **/
  function borrow(
    address asset,
    uint256 amount,
    uint256 interestRateMode,
    uint16 referralCode,
    address onBehalfOf
  ) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    _executeBorrow(
      ExecuteBorrowParams(
        asset,
        msg.sender,
        onBehalfOf,
        amount,
        interestRateMode,
        reserve.aTokenAddress,
        referralCode,
        true
      )
    );
  }

  /**
   * @notice Repays a borrowed `amount` on a specific reserve, burning the equivalent debt tokens own
   * - E.g. User repays 100 USDC, burning 100 variable/stable debt tokens of the `onBehalfOf` address
   * @param asset The address of the borrowed underlying asset previously borrowed
   * @param amount The amount to repay
   * - Send the value type(uint256).max in order to repay the whole debt for `asset` on the specific
   * @param rateMode The interest rate mode at of the debt the user wants to repay: 1 for Stable, 2 f
   * @param onBehalfOf Address of the user who will get his debt reduced/removed. Should be the addre
   * user calling the function if he wants to reduce/remove his own debt, or the address of any other
   * other borrower whose debt should be removed
   * @return The final amount repaid
   **/
  function repay(
    address asset,
    uint256 amount,
    uint256 rateMode,
    address onBehalfOf
  ) external override whenNotPaused returns (uint256) {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(onBehalfOf, reserve);

    DataTypes.InterestRateMode interestRateMode = DataTypes.InterestRateMode(rateMode);

    ValidationLogic.validateRepay(
      reserve,
      amount,
      interestRateMode,
      onBehalfOf,
      stableDebt,
      variableDebt
    );

    uint256 paybackAmount =
```

```
      interestRateMode == DataTypes.InterestRateMode.STABLE ? stableDebt : variableDebt;

    if (amount < paybackAmount) {
      paybackAmount = amount;
    }

    reserve.updateState();

    if (interestRateMode == DataTypes.InterestRateMode.STABLE) {
      IStableDebtToken(reserve.stableDebtTokenAddress).burn(onBehalfOf, paybackAmount);
    } else {
      IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
        onBehalfOf,
        paybackAmount,
        reserve.variableBorrowIndex
      );
    }

    address aToken = reserve.aTokenAddress;
    reserve.updateInterestRates(asset, aToken, paybackAmount, 0);

    if (stableDebt.add(variableDebt).sub(paybackAmount) == 0) {
      _usersConfig[onBehalfOf].setBorrowing(reserve.id, false);
    }

    IERC20(asset).safeTransferFrom(msg.sender, aToken, paybackAmount);

    IAToken(aToken).handleRepayment(msg.sender, paybackAmount);

    emit Repay(asset, onBehalfOf, msg.sender, paybackAmount);

    return paybackAmount;
  }

  /**
   * @dev Allows a borrower to swap his debt between stable and variable mode, or viceversa
   * @param asset The address of the underlying asset borrowed
   * @param rateMode The rate mode that the user wants to swap to
   **/
  function swapBorrowRateMode(address asset, uint256 rateMode) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(msg.sender, reserve);

    DataTypes.InterestRateMode interestRateMode = DataTypes.InterestRateMode(rateMode);

    ValidationLogic.validateSwapRateMode(
      reserve,
      _usersConfig[msg.sender],
      stableDebt,
      variableDebt,
      interestRateMode
    );

    reserve.updateState();

    if (interestRateMode == DataTypes.InterestRateMode.STABLE) {
      IStableDebtToken(reserve.stableDebtTokenAddress).burn(msg.sender, stableDebt);
      IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
        msg.sender,
        msg.sender,
        stableDebt,
        reserve.variableBorrowIndex
      );
    } else {
      IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
```

```
        msg.sender,
        variableDebt,
        reserve.variableBorrowIndex
      );
      IStableDebtToken(reserve.stableDebtTokenAddress).mint(
        msg.sender,
        msg.sender,
        variableDebt,
        reserve.currentStableBorrowRate
      );
    }

    reserve.updateInterestRates(asset, reserve.aTokenAddress, 0, 0);

    emit Swap(asset, msg.sender, rateMode);
  }

  /**
   * @dev Rebalances the stable interest rate of a user to the current stable rate defined on the res
   * - Users can be rebalanced if the following conditions are satisfied:
   *     1. Usage ratio is above 95%
   *     2. the current deposit APY is below REBALANCE_UP_THRESHOLD * maxVariableBorrowRate, which me
   *        borrowed at a stable rate and depositors are not earning enough
   * @param asset The address of the underlying asset borrowed
   * @param user The address of the user to be rebalanced
   **/
  function rebalanceStableBorrowRate(address asset, address user) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    IERC20 stableDebtToken = IERC20(reserve.stableDebtTokenAddress);
    IERC20 variableDebtToken = IERC20(reserve.variableDebtTokenAddress);
    address aTokenAddress = reserve.aTokenAddress;

    uint256 stableDebt = IERC20(stableDebtToken).balanceOf(user);

    ValidationLogic.validateRebalanceStableBorrowRate(
      reserve,
      asset,
      stableDebtToken,
      variableDebtToken,
      aTokenAddress
    );

    reserve.updateState();

    IStableDebtToken(address(stableDebtToken)).burn(user, stableDebt);
    IStableDebtToken(address(stableDebtToken)).mint(
      user,
      user,
      stableDebt,
      reserve.currentStableBorrowRate
    );

    reserve.updateInterestRates(asset, aTokenAddress, 0, 0);

    emit RebalanceStableBorrowRate(asset, user);
  }

  /**
   * @dev Allows depositors to enable/disable a specific deposited asset as collateral
   * @param asset The address of the underlying asset deposited
   * @param useAsCollateral `true` if the user wants to use the deposit as collateral, `false` otherw
   **/
  function setUserUseReserveAsCollateral(address asset, bool useAsCollateral)
    external
    override
```

```
    whenNotPaused
  {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    ValidationLogic.validateSetUseReserveAsCollateral(
      reserve,
      asset,
      useAsCollateral,
      _reserves,
      _usersConfig[msg.sender],
      _reservesList,
      _reservesCount,
      _addressesProvider.getPriceOracle()
    );

    _usersConfig[msg.sender].setUsingAsCollateral(reserve.id, useAsCollateral);

    if (useAsCollateral) {
      emit ReserveUsedAsCollateralEnabled(asset, msg.sender);
    } else {
      emit ReserveUsedAsCollateralDisabled(asset, msg.sender);
    }
  }

  /**
   * @dev Function to liquidate a non-healthy position collateral-wise, with Health Factor below 1
   * - The caller (liquidator) covers `debtToCover` amount of debt of the user getting liquidated, an
   *   a proportionally amount of the `collateralAsset` plus a bonus to cover market risk
   * @param collateralAsset The address of the underlying asset used as collateral, to receive as res
   * @param debtAsset The address of the underlying borrowed asset to be repaid with the liquidation
   * @param user The address of the borrower getting liquidated
   * @param debtToCover The debt amount of borrowed `asset` the liquidator wants to cover
   * @param receiveAToken `true` if the liquidators wants to receive the collateral aTokens, `false`
   * to receive the underlying collateral asset directly
   **/
  function liquidationCall(
    address collateralAsset,
    address debtAsset,
    address user,
    uint256 debtToCover,
    bool receiveAToken
  ) external override whenNotPaused {
    address collateralManager = _addressesProvider.getLendingPoolCollateralManager();

    //solium-disable-next-line
    (bool success, bytes memory result) =
      collateralManager.delegatecall(
        abi.encodeWithSignature(
          'liquidationCall(address,address,address,uint256,bool)',
          collateralAsset,
          debtAsset,
          user,
          debtToCover,
          receiveAToken
        )
      );

    require(success, Errors.LP_LIQUIDATION_CALL_FAILED);

    (uint256 returnCode, string memory returnMessage) = abi.decode(result, (uint256, string));

    require(returnCode == 0, string(abi.encodePacked(returnMessage)));
  }

  struct FlashLoanLocalVars {
    IFlashLoanReceiver receiver;
```

```
    address oracle;
    uint256 i;
    address currentAsset;
    address currentATokenAddress;
    uint256 currentAmount;
    uint256 currentPremium;
    uint256 currentAmountPlusPremium;
    address debtToken;
  }

  /**
   * @dev Allows smartcontracts to access the liquidity of the pool within one transaction,
   * as long as the amount taken plus a fee is returned.
   * IMPORTANT There are security concerns for developers of flashloan receiver contracts that must b
   * For further details please visit https://developers.aave.com
   * @param receiverAddress The address of the contract receiving the funds, implementing the IFlashL
   * @param assets The addresses of the assets being flash-borrowed
   * @param amounts The amounts amounts being flash-borrowed
   * @param modes Types of the debt to open if the flash loan is not returned:
   *   0 -> Don't open any debt, just revert if funds can't be transferred from the receiver
   *   1 -> Open debt at stable rate for the value of the amount flash-borrowed to the `onBehalfOf` a
   *   2 -> Open debt at variable rate for the value of the amount flash-borrowed to the `onBehalfOf`
   * @param onBehalfOf The address  that will receive the debt in the case of using on `modes` 1 or 2
   * @param params Variadic packed params to pass to the receiver as extra information
   * @param referralCode Code used to register the integrator originating the operation, for potentia
   *   0 if the action is executed directly by the user, without any middle-man
   **/
  function flashLoan(
    address receiverAddress,
    address[] calldata assets,
    uint256[] calldata amounts,
    uint256[] calldata modes,
    address onBehalfOf,
    bytes calldata params,
    uint16 referralCode
  ) external override whenNotPaused {
    FlashLoanLocalVars memory vars;

    ValidationLogic.validateFlashloan(assets, amounts);

    address[] memory aTokenAddresses = new address[](assets.length);
    uint256[] memory premiums = new uint256[](assets.length);

    vars.receiver = IFlashLoanReceiver(receiverAddress);

    for (vars.i = 0; vars.i < assets.length; vars.i++) {
      aTokenAddresses[vars.i] = _reserves[assets[vars.i]].aTokenAddress;

      premiums[vars.i] = amounts[vars.i].mul(_flashLoanPremiumTotal).div(10000);

      IAToken(aTokenAddresses[vars.i]).transferUnderlyingTo(receiverAddress, amounts[vars.i]);
    }

    require(
      vars.receiver.executeOperation(assets, amounts, premiums, msg.sender, params),
      Errors.LP_INVALID_FLASH_LOAN_EXECUTOR_RETURN
    );

    for (vars.i = 0; vars.i < assets.length; vars.i++) {
      vars.currentAsset = assets[vars.i];
      vars.currentAmount = amounts[vars.i];
      vars.currentPremium = premiums[vars.i];
      vars.currentATokenAddress = aTokenAddresses[vars.i];
      vars.currentAmountPlusPremium = vars.currentAmount.add(vars.currentPremium);

      if (DataTypes.InterestRateMode(modes[vars.i]) == DataTypes.InterestRateMode.NONE) {
```

```
        _reserves[vars.currentAsset].updateState();
        _reserves[vars.currentAsset].cumulateToLiquidityIndex(
          IERC20(vars.currentATokenAddress).totalSupply(),
          vars.currentPremium
        );
        _reserves[vars.currentAsset].updateInterestRates(
          vars.currentAsset,
          vars.currentATokenAddress,
          vars.currentAmountPlusPremium,
          0
        );

        IERC20(vars.currentAsset).safeTransferFrom(
          receiverAddress,
          vars.currentATokenAddress,
          vars.currentAmountPlusPremium
        );
      } else {
        // If the user chose to not return the funds, the system checks if there is enough collateral
        // eventually opens a debt position
        _executeBorrow(
          ExecuteBorrowParams(
            vars.currentAsset,
            msg.sender,
            onBehalfOf,
            vars.currentAmount,
            modes[vars.i],
            vars.currentATokenAddress,
            referralCode,
            false
          )
        );
      }
      emit FlashLoan(
        receiverAddress,
        msg.sender,
        vars.currentAsset,
        vars.currentAmount,
        vars.currentPremium,
        referralCode
      );
    }
  }

  /**
   * @dev Returns the state and configuration of the reserve
   * @param asset The address of the underlying asset of the reserve
   * @return The state of the reserve
   **/
  function getReserveData(address asset)
    external
    view
    override
    returns (DataTypes.ReserveData memory)
  {
    return _reserves[asset];
  }

  /**
   * @dev Returns the user account data across all the reserves
   * @param user The address of the user
   * @return totalCollateralETH the total collateral in ETH of the user
   * @return totalDebtETH the total debt in ETH of the user
   * @return availableBorrowsETH the borrowing power left of the user
   * @return currentLiquidationThreshold the liquidation threshold of the user
   * @return ltv the loan to value of the user
```

```
   * @return healthFactor the current health factor of the user
   **/
  function getUserAccountData(address user)
    external
    view
    override
    returns (
      uint256 totalCollateralETH,
      uint256 totalDebtETH,
      uint256 availableBorrowsETH,
      uint256 currentLiquidationThreshold,
      uint256 ltv,
      uint256 healthFactor
    )
  {
    (
      totalCollateralETH,
      totalDebtETH,
      ltv,
      currentLiquidationThreshold,
      healthFactor
    ) = GenericLogic.calculateUserAccountData(
      user,
      _reserves,
      _usersConfig[user],
      _reservesList,
      _reservesCount,
      _addressesProvider.getPriceOracle()
    );

    availableBorrowsETH = GenericLogic.calculateAvailableBorrowsETH(
      totalCollateralETH,
      totalDebtETH,
      ltv
    );
  }

  /**
   * @dev Returns the configuration of the reserve
   * @param asset The address of the underlying asset of the reserve
   * @return The configuration of the reserve
   **/
  function getConfiguration(address asset)
    external
    view
    override
    returns (DataTypes.ReserveConfigurationMap memory)
  {
    return _reserves[asset].configuration;
  }

  /**
   * @dev Returns the configuration of the user across all the reserves
   * @param user The user address
   * @return The configuration of the user
   **/
  function getUserConfiguration(address user)
    external
    view
    override
    returns (DataTypes.UserConfigurationMap memory)
  {
    return _usersConfig[user];
  }

  /**
```

```
  * @dev Returns the normalized income per unit of asset
  * @param asset The address of the underlying asset of the reserve
  * @return The reserve's normalized income
  */
function getReserveNormalizedIncome(address asset)
  external
  view
  virtual
  override
  returns (uint256)
{
  return _reserves[asset].getNormalizedIncome();
}

/**
  * @dev Returns the normalized variable debt per unit of asset
  * @param asset The address of the underlying asset of the reserve
  * @return The reserve normalized variable debt
  */
function getReserveNormalizedVariableDebt(address asset)
  external
  view
  override
  returns (uint256)
{
  return _reserves[asset].getNormalizedDebt();
}

/**
  * @dev Returns if the LendingPool is paused
  */
function paused() external view override returns (bool) {
  return _paused;
}

/**
  * @dev Returns the list of the initialized reserves
  **/
function getReservesList() external view override returns (address[] memory) {
  address[] memory _activeReserves = new address[](_reservesCount);

  for (uint256 i = 0; i < _reservesCount; i++) {
    _activeReserves[i] = _reservesList[i];
  }
  return _activeReserves;
}

/**
  * @dev Returns the cached LendingPoolAddressesProvider connected to this contract
  **/
function getAddressesProvider() external view override returns (ILendingPoolAddressesProvider) {
  return _addressesProvider;
}

/**
  * @dev Returns the percentage of available liquidity that can be borrowed at once at stable rate
  */
function MAX_STABLE_RATE_BORROW_SIZE_PERCENT() public view returns (uint256) {
  return _maxStableRateBorrowSizePercent;
}

/**
  * @dev Returns the fee on flash loans
  */
function FLASHLOAN_PREMIUM_TOTAL() public view returns (uint256) {
  return _flashLoanPremiumTotal;
```

```
}

/**
 * @dev Returns the maximum number of reserves supported to be listed in this LendingPool
 */
function MAX_NUMBER_RESERVES() public view returns (uint256) {
  return _maxNumberOfReserves;
}

/**
 * @dev Validates and finalizes an aToken transfer
 * - Only callable by the overlying aToken of the `asset`
 * @param asset The address of the underlying asset of the aToken
 * @param from The user from which the aTokens are transferred
 * @param to The user receiving the aTokens
 * @param amount The amount being transferred/withdrawn
 * @param balanceFromBefore The aToken balance of the `from` user before the transfer
 * @param balanceToBefore The aToken balance of the `to` user before the transfer
 */
function finalizeTransfer(
  address asset,
  address from,
  address to,
  uint256 amount,
  uint256 balanceFromBefore,
  uint256 balanceToBefore
) external override whenNotPaused {
  require(msg.sender == _reserves[asset].aTokenAddress, Errors.LP_CALLER_MUST_BE_AN_ATOKEN);

  ValidationLogic.validateTransfer(
    from,
    _reserves,
    _usersConfig[from],
    _reservesList,
    _reservesCount,
    _addressesProvider.getPriceOracle()
  );

  uint256 reserveId = _reserves[asset].id;

  if (from != to) {
    if (balanceFromBefore.sub(amount) == 0) {
      DataTypes.UserConfigurationMap storage fromConfig = _usersConfig[from];
      fromConfig.setUsingAsCollateral(reserveId, false);
      emit ReserveUsedAsCollateralDisabled(asset, from);
    }

    if (balanceToBefore == 0 && amount != 0) {
      DataTypes.UserConfigurationMap storage toConfig = _usersConfig[to];
      toConfig.setUsingAsCollateral(reserveId, true);
      emit ReserveUsedAsCollateralEnabled(asset, to);
    }
  }
}

/**
 * @dev Initializes a reserve, activating it, assigning an aToken and debt tokens and an
 * interest rate strategy
 * - Only callable by the LendingPoolConfigurator contract
 * @param asset The address of the underlying asset of the reserve
 * @param aTokenAddress The address of the aToken that will be assigned to the reserve
 * @param stableDebtAddress The address of the StableDebtToken that will be assigned to the reserve
 * @param aTokenAddress The address of the VariableDebtToken that will be assigned to the reserve
 * @param interestRateStrategyAddress The address of the interest rate strategy contract
 **/
function initReserve(
```

```
    address asset,
    address aTokenAddress,
    address stableDebtAddress,
    address variableDebtAddress,
    address interestRateStrategyAddress
  ) external override onlyLendingPoolConfigurator {
    require(Address.isContract(asset), Errors.LP_NOT_CONTRACT);
    _reserves[asset].init(
      aTokenAddress,
      stableDebtAddress,
      variableDebtAddress,
      interestRateStrategyAddress
    );
    _addReserveToList(asset);
  }

  /**
   * @dev Updates the address of the interest rate strategy contract
   * - Only callable by the LendingPoolConfigurator contract
   * @param asset The address of the underlying asset of the reserve
   * @param rateStrategyAddress The address of the interest rate strategy contract
   **/
  function setReserveInterestRateStrategyAddress(address asset, address rateStrategyAddress)
    external
    override
    onlyLendingPoolConfigurator
  {
    _reserves[asset].interestRateStrategyAddress = rateStrategyAddress;
  }

  /**
   * @dev Sets the configuration bitmap of the reserve as a whole
   * - Only callable by the LendingPoolConfigurator contract
   * @param asset The address of the underlying asset of the reserve
   * @param configuration The new configuration bitmap
   **/
  function setConfiguration(address asset, uint256 configuration)
    external
    override
    onlyLendingPoolConfigurator
  {
    _reserves[asset].configuration.data = configuration;
  }

  /**
   * @dev Set the _pause state of a reserve
   * - Only callable by the LendingPoolConfigurator contract
   * @param val `true` to pause the reserve, `false` to un-pause it
   */
  function setPause(bool val) external override onlyLendingPoolConfigurator {
    _paused = val;
    if (_paused) {
      emit Paused();
    } else {
      emit Unpaused();
    }
  }

  struct ExecuteBorrowParams {
    address asset;
    address user;
    address onBehalfOf;
    uint256 amount;
    uint256 interestRateMode;
    address aTokenAddress;
    uint16 referralCode;
```

```
    bool releaseUnderlying;
  }

  function _executeBorrow(ExecuteBorrowParams memory vars) internal {
    DataTypes.ReserveData storage reserve = _reserves[vars.asset];
    DataTypes.UserConfigurationMap storage userConfig = _usersConfig[vars.onBehalfOf];

    address oracle = _addressesProvider.getPriceOracle();

    uint256 amountInETH =
      IPriceOracleGetter(oracle).getAssetPrice(vars.asset).mul(vars.amount).div(
        10**reserve.configuration.getDecimals()
      );

    ValidationLogic.validateBorrow(
      vars.asset,
      reserve,
      vars.onBehalfOf,
      vars.amount,
      amountInETH,
      vars.interestRateMode,
      _maxStableRateBorrowSizePercent,
      _reserves,
      userConfig,
      _reservesList,
      _reservesCount,
      oracle
    );

    reserve.updateState();

    uint256 currentStableRate = 0;

    bool isFirstBorrowing = false;
    if (DataTypes.InterestRateMode(vars.interestRateMode) == DataTypes.InterestRateMode.STABLE) {
      currentStableRate = reserve.currentStableBorrowRate;

      isFirstBorrowing = IStableDebtToken(reserve.stableDebtTokenAddress).mint(
        vars.user,
        vars.onBehalfOf,
        vars.amount,
        currentStableRate
      );
    } else {
      isFirstBorrowing = IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
        vars.user,
        vars.onBehalfOf,
        vars.amount,
        reserve.variableBorrowIndex
      );
    }

    if (isFirstBorrowing) {
      userConfig.setBorrowing(reserve.id, true);
    }

    reserve.updateInterestRates(
      vars.asset,
      vars.aTokenAddress,
      0,
      vars.releaseUnderlying ? vars.amount : 0
    );

    if (vars.releaseUnderlying) {
      IAToken(vars.aTokenAddress).transferUnderlyingTo(vars.user, vars.amount);
    }
```

```
    emit Borrow(
      vars.asset,
      vars.user,
      vars.onBehalfOf,
      vars.amount,
      vars.interestRateMode,
      DataTypes.InterestRateMode(vars.interestRateMode) == DataTypes.InterestRateMode.STABLE
        ? currentStableRate
        : reserve.currentVariableBorrowRate,
      vars.referralCode
    );
  }

  function _addReserveToList(address asset) internal {
    uint256 reservesCount = _reservesCount;

    require(reservesCount < _maxNumberOfReserves, Errors.LP_NO_MORE_RESERVES_ALLOWED);

    bool reserveAlreadyAdded = _reserves[asset].id != 0 || _reservesList[0] == asset;

    if (!reserveAlreadyAdded) {
      _reserves[asset].id = uint8(reservesCount);
      _reservesList[reservesCount] = asset;

      _reservesCount = reservesCount + 1;
    }
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {UserConfiguration} from '../libraries/configuration/UserConfiguration.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {ReserveLogic} from '../libraries/logic/ReserveLogic.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';

contract LendingPoolStorage {
  using ReserveLogic for DataTypes.ReserveData;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  ILendingPoolAddressesProvider internal _addressesProvider;

  mapping(address => DataTypes.ReserveData) internal _reserves;
  mapping(address => DataTypes.UserConfigurationMap) internal _usersConfig;

  // the list of the available reserves, structured as a mapping for gas savings reasons
  mapping(uint256 => address) internal _reservesList;

  uint256 internal _reservesCount;

  bool internal _paused;

  uint256 internal _maxStableRateBorrowSizePercent;

  uint256 internal _flashLoanPremiumTotal;

  uint256 internal _maxNumberOfReserves;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IReserveInterestRateStrategy} from '../../interfaces/IReserveInterestRateStrategy.sol';
```

```
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingRateOracle} from '../../interfaces/ILendingRateOracle.sol';
import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';
import 'hardhat/console.sol';

/**
 * @title DefaultReserveInterestRateStrategy contract
 * @notice Implements the calculation of the interest rates depending on the reserve state
 * @dev The model of interest rate is based on 2 slopes, one before the `OPTIMAL_UTILIZATION_RATE`
 * point of utilization and another from that one to 100%
 * - An instance of this same contract, can't be used across different Aave markets, due to the cachi
 *   of the LendingPoolAddressesProvider
 * @author Aave
 **/
contract DefaultReserveInterestRateStrategy is IReserveInterestRateStrategy {
  using WadRayMath for uint256;
  using SafeMath for uint256;
  using PercentageMath for uint256;

  /**
   * @dev this constant represents the utilization rate at which the pool aims to obtain most competi
   * Expressed in ray
   **/
  uint256 public immutable OPTIMAL_UTILIZATION_RATE;

  /**
   * @dev This constant represents the excess utilization rate above the optimal. It's always equal t
   * 1-optimal utilization rate. Added as a constant here for gas optimizations.
   * Expressed in ray
   **/

  uint256 public immutable EXCESS_UTILIZATION_RATE;

  ILendingPoolAddressesProvider public immutable addressesProvider;

  // Base variable borrow rate when Utilization rate = 0. Expressed in ray
  uint256 internal immutable _baseVariableBorrowRate;

  // Slope of the variable interest curve when utilization rate > 0 and <= OPTIMAL_UTILIZATION_RATE.
  uint256 internal immutable _variableRateSlope1;

  // Slope of the variable interest curve when utilization rate > OPTIMAL_UTILIZATION_RATE. Expressed
  uint256 internal immutable _variableRateSlope2;

  // Slope of the stable interest curve when utilization rate > 0 and <= OPTIMAL_UTILIZATION_RATE. Ex
  uint256 internal immutable _stableRateSlope1;

  // Slope of the stable interest curve when utilization rate > OPTIMAL_UTILIZATION_RATE. Expressed i
  uint256 internal immutable _stableRateSlope2;

  constructor(
    ILendingPoolAddressesProvider provider,
    uint256 optimalUtilizationRate,
    uint256 baseVariableBorrowRate,
    uint256 variableRateSlope1,
    uint256 variableRateSlope2,
    uint256 stableRateSlope1,
    uint256 stableRateSlope2
  ) public {
    OPTIMAL_UTILIZATION_RATE = optimalUtilizationRate;
    EXCESS_UTILIZATION_RATE = WadRayMath.ray().sub(optimalUtilizationRate);
    addressesProvider = provider;
    _baseVariableBorrowRate = baseVariableBorrowRate;
    _variableRateSlope1 = variableRateSlope1;
```

```
    _variableRateSlope2 = variableRateSlope2;
    _stableRateSlope1 = stableRateSlope1;
    _stableRateSlope2 = stableRateSlope2;
  }

  function variableRateSlope1() external view returns (uint256) {
    return _variableRateSlope1;
  }

  function variableRateSlope2() external view returns (uint256) {
    return _variableRateSlope2;
  }

  function stableRateSlope1() external view returns (uint256) {
    return _stableRateSlope1;
  }

  function stableRateSlope2() external view returns (uint256) {
    return _stableRateSlope2;
  }

  function baseVariableBorrowRate() external view override returns (uint256) {
    return _baseVariableBorrowRate;
  }

  function getMaxVariableBorrowRate() external view override returns (uint256) {
    return _baseVariableBorrowRate.add(_variableRateSlope1).add(_variableRateSlope2);
  }

  /**
   * @dev Calculates the interest rates depending on the reserve's state and configurations
   * @param reserve The address of the reserve
   * @param liquidityAdded The liquidity added during the operation
   * @param liquidityTaken The liquidity taken during the operation
   * @param totalStableDebt The total borrowed from the reserve a stable rate
   * @param totalVariableDebt The total borrowed from the reserve at a variable rate
   * @param averageStableBorrowRate The weighted average of all the stable rate loans
   * @param reserveFactor The reserve portion of the interest that goes to the treasury of the market
   * @return The liquidity rate, the stable borrow rate and the variable borrow rate
   **/
  function calculateInterestRates(
    address reserve,
    address aToken,
    uint256 liquidityAdded,
    uint256 liquidityTaken,
    uint256 totalStableDebt,
    uint256 totalVariableDebt,
    uint256 averageStableBorrowRate,
    uint256 reserveFactor
  )
    external
    view
    override
    returns (
      uint256,
      uint256,
      uint256
    )
  {
    uint256 availableLiquidity = IERC20(reserve).balanceOf(aToken);
    //avoid stack too deep
    availableLiquidity = availableLiquidity.add(liquidityAdded).sub(liquidityTaken);

    return
      calculateInterestRates(
        reserve,
```

```
      availableLiquidity,
      totalStableDebt,
      totalVariableDebt,
      averageStableBorrowRate,
      reserveFactor
    );
}

struct CalcInterestRatesLocalVars {
  uint256 totalDebt;
  uint256 currentVariableBorrowRate;
  uint256 currentStableBorrowRate;
  uint256 currentLiquidityRate;
  uint256 utilizationRate;
}

/**
 * @dev Calculates the interest rates depending on the reserve's state and configurations.
 * NOTE This function is kept for compatibility with the previous DefaultInterestRateStrategy inter
 * New protocol implementation uses the new calculateInterestRates() interface
 * @param reserve The address of the reserve
 * @param availableLiquidity The liquidity available in the corresponding aToken
 * @param totalStableDebt The total borrowed from the reserve a stable rate
 * @param totalVariableDebt The total borrowed from the reserve at a variable rate
 * @param averageStableBorrowRate The weighted average of all the stable rate loans
 * @param reserveFactor The reserve portion of the interest that goes to the treasury of the market
 * @return The liquidity rate, the stable borrow rate and the variable borrow rate
 **/
function calculateInterestRates(
  address reserve,
  uint256 availableLiquidity,
  uint256 totalStableDebt,
  uint256 totalVariableDebt,
  uint256 averageStableBorrowRate,
  uint256 reserveFactor
)
  public
  view
  override
  returns (
    uint256,
    uint256,
    uint256
  )
{
  CalcInterestRatesLocalVars memory vars;

  vars.totalDebt = totalStableDebt.add(totalVariableDebt);
  vars.currentVariableBorrowRate = 0;
  vars.currentStableBorrowRate = 0;
  vars.currentLiquidityRate = 0;

  vars.utilizationRate = vars.totalDebt == 0
    ? 0
    : vars.totalDebt.rayDiv(availableLiquidity.add(vars.totalDebt));

  vars.currentStableBorrowRate = ILendingRateOracle(addressesProvider.getLendingRateOracle())
    .getMarketBorrowRate(reserve);

  if (vars.utilizationRate > OPTIMAL_UTILIZATION_RATE) {
    uint256 excessUtilizationRateRatio =
      vars.utilizationRate.sub(OPTIMAL_UTILIZATION_RATE).rayDiv(EXCESS_UTILIZATION_RATE);

    vars.currentStableBorrowRate = vars.currentStableBorrowRate.add(_stableRateSlope1).add(
      _stableRateSlope2.rayMul(excessUtilizationRateRatio)
    );
```

```
      vars.currentVariableBorrowRate = _baseVariableBorrowRate.add(_variableRateSlope1).add(
        _variableRateSlope2.rayMul(excessUtilizationRateRatio)
      );
    } else {
      vars.currentStableBorrowRate = vars.currentStableBorrowRate.add(
        _stableRateSlope1.rayMul(vars.utilizationRate.rayDiv(OPTIMAL_UTILIZATION_RATE))
      );
      vars.currentVariableBorrowRate = _baseVariableBorrowRate.add(
        vars.utilizationRate.rayMul(_variableRateSlope1).rayDiv(OPTIMAL_UTILIZATION_RATE)
      );
    }

    vars.currentLiquidityRate = _getOverallBorrowRate(
      totalStableDebt,
      totalVariableDebt,
      vars
        .currentVariableBorrowRate,
      averageStableBorrowRate
    )
      .rayMul(vars.utilizationRate)
      .percentMul(PercentageMath.PERCENTAGE_FACTOR.sub(reserveFactor));

    return (
      vars.currentLiquidityRate,
      vars.currentStableBorrowRate,
      vars.currentVariableBorrowRate
    );
  }

  /**
   * @dev Calculates the overall borrow rate as the weighted average between the total variable debt
   * @param totalStableDebt The total borrowed from the reserve a stable rate
   * @param totalVariableDebt The total borrowed from the reserve at a variable rate
   * @param currentVariableBorrowRate The current variable borrow rate of the reserve
   * @param currentAverageStableBorrowRate The current weighted average of all the stable rate loans
   * @return The weighted averaged borrow rate
   **/
  function _getOverallBorrowRate(
    uint256 totalStableDebt,
    uint256 totalVariableDebt,
    uint256 currentVariableBorrowRate,
    uint256 currentAverageStableBorrowRate
  ) internal pure returns (uint256) {
    uint256 totalDebt = totalStableDebt.add(totalVariableDebt);

    if (totalDebt == 0) return 0;

    uint256 weightedVariableRate = totalVariableDebt.wadToRay().rayMul(currentVariableBorrowRate);

    uint256 weightedStableRate = totalStableDebt.wadToRay().rayMul(currentAverageStableBorrowRate);

    uint256 overallBorrowRate =
      weightedVariableRate.add(weightedStableRate).rayDiv(totalDebt.wadToRay());

    return overallBorrowRate;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {
  InitializableImmutableAdminUpgradeabilityProxy
```

```
} from '../libraries/aave-upgradeability/InitializableImmutableAdminUpgradeabilityProxy.sol';
import {ReserveConfiguration} from '../libraries/configuration/ReserveConfiguration.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IERC20Detailed} from '../../dependencies/openzeppelin/contracts/IERC20Detailed.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';
import {IInitializableDebtToken} from '../../interfaces/IInitializableDebtToken.sol';
import {IInitializableAToken} from '../../interfaces/IInitializableAToken.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';
import {ILendingPoolConfigurator} from '../../interfaces/ILendingPoolConfigurator.sol';

/**
 * @title LendingPoolConfigurator contract
 * @author Aave
 * @dev Implements the configuration methods for the Aave protocol
 **/

contract LendingPoolConfigurator is VersionedInitializable, ILendingPoolConfigurator {
  using SafeMath for uint256;
  using PercentageMath for uint256;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;

  ILendingPoolAddressesProvider internal addressesProvider;
  ILendingPool internal pool;

  modifier onlyPoolAdmin {
    require(addressesProvider.getPoolAdmin() == msg.sender, Errors.CALLER_NOT_POOL_ADMIN);
    _;
  }

  modifier onlyEmergencyAdmin {
    require(
      addressesProvider.getEmergencyAdmin() == msg.sender,
      Errors.LPC_CALLER_NOT_EMERGENCY_ADMIN
    );
    _;
  }

  uint256 internal constant CONFIGURATOR_REVISION = 0x1;

  function getRevision() internal pure override returns (uint256) {
    return CONFIGURATOR_REVISION;
  }

  function initialize(ILendingPoolAddressesProvider provider) public initializer {
    addressesProvider = provider;
    pool = ILendingPool(addressesProvider.getLendingPool());
  }

  /**
   * @dev Initializes reserves in batch
   **/
  function batchInitReserve(InitReserveInput[] calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;
    for (uint256 i = 0; i < input.length; i++) {
      _initReserve(cachedPool, input[i]);
    }
  }

  function _initReserve(ILendingPool pool, InitReserveInput calldata input) internal {
    address aTokenProxyAddress =
      _initTokenWithProxy(
        input.aTokenImpl,
        abi.encodeWithSelector(
```

```
        IInitializableAToken.initialize.selector,
        pool,
        input.treasury,
        input.underlyingAsset,
        IAaveIncentivesController(input.incentivesController),
        input.underlyingAssetDecimals,
        input.aTokenName,
        input.aTokenSymbol,
        input.params
      )
    );

  address stableDebtTokenProxyAddress =
    _initTokenWithProxy(
      input.stableDebtTokenImpl,
      abi.encodeWithSelector(
        IInitializableDebtToken.initialize.selector,
        pool,
        input.underlyingAsset,
        IAaveIncentivesController(input.incentivesController),
        input.underlyingAssetDecimals,
        input.stableDebtTokenName,
        input.stableDebtTokenSymbol,
        input.params
      )
    );

  address variableDebtTokenProxyAddress =
    _initTokenWithProxy(
      input.variableDebtTokenImpl,
      abi.encodeWithSelector(
        IInitializableDebtToken.initialize.selector,
        pool,
        input.underlyingAsset,
        IAaveIncentivesController(input.incentivesController),
        input.underlyingAssetDecimals,
        input.variableDebtTokenName,
        input.variableDebtTokenSymbol,
        input.params
      )
    );

  pool.initReserve(
    input.underlyingAsset,
    aTokenProxyAddress,
    stableDebtTokenProxyAddress,
    variableDebtTokenProxyAddress,
    input.interestRateStrategyAddress
  );

  DataTypes.ReserveConfigurationMap memory currentConfig =
    pool.getConfiguration(input.underlyingAsset);

  currentConfig.setDecimals(input.underlyingAssetDecimals);

  currentConfig.setActive(true);
  currentConfig.setFrozen(false);

  pool.setConfiguration(input.underlyingAsset, currentConfig.data);

  emit ReserveInitialized(
    input.underlyingAsset,
    aTokenProxyAddress,
    stableDebtTokenProxyAddress,
    variableDebtTokenProxyAddress,
    input.interestRateStrategyAddress
```

```
    );
  }

  /**
   * @dev Updates the aToken implementation for the reserve
   **/
  function updateAToken(UpdateATokenInput calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;

    DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(input.asset);

    (, , , uint256 decimals, ) = cachedPool.getConfiguration(input.asset).getParamsMemory();

    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableAToken.initialize.selector,
        cachedPool,
        input.treasury,
        input.asset,
        input.incentivesController,
        decimals,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      reserveData.aTokenAddress,
      input.implementation,
      encodedCall
    );

    emit ATokenUpgraded(input.asset, reserveData.aTokenAddress, input.implementation);
  }

  /**
   * @dev Updates the stable debt token implementation for the reserve
   **/
  function updateStableDebtToken(UpdateDebtTokenInput calldata input) external onlyPoolAdmin {
    ILendingPool cachedPool = pool;

    DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(input.asset);

    (, , , uint256 decimals, ) = cachedPool.getConfiguration(input.asset).getParamsMemory();

    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableDebtToken.initialize.selector,
        cachedPool,
        input.asset,
        input.incentivesController,
        decimals,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      reserveData.stableDebtTokenAddress,
      input.implementation,
      encodedCall
    );

    emit StableDebtTokenUpgraded(
      input.asset,
      reserveData.stableDebtTokenAddress,
      input.implementation
    );
```

```
  }

  /**
   * @dev Updates the variable debt token implementation for the asset
   **/
  function updateVariableDebtToken(UpdateDebtTokenInput calldata input)
    external
    onlyPoolAdmin
  {
    ILendingPool cachedPool = pool;

    DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(input.asset);

    (, , , uint256 decimals, ) = cachedPool.getConfiguration(input.asset).getParamsMemory();

    bytes memory encodedCall = abi.encodeWithSelector(
        IInitializableDebtToken.initialize.selector,
        cachedPool,
        input.asset,
        input.incentivesController,
        decimals,
        input.name,
        input.symbol,
        input.params
      );

    _upgradeTokenImplementation(
      reserveData.variableDebtTokenAddress,
      input.implementation,
      encodedCall
    );

    emit VariableDebtTokenUpgraded(
      input.asset,
      reserveData.variableDebtTokenAddress,
      input.implementation
    );
  }

  /**
   * @dev Enables borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   * @param stableBorrowRateEnabled True if stable borrow rate needs to be enabled by default on this
   **/
  function enableBorrowingOnReserve(address asset, bool stableBorrowRateEnabled)
    external
    onlyPoolAdmin
  {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setBorrowingEnabled(true);
    currentConfig.setStableRateBorrowingEnabled(stableBorrowRateEnabled);

    pool.setConfiguration(asset, currentConfig.data);

    emit BorrowingEnabledOnReserve(asset, stableBorrowRateEnabled);
  }

  /**
   * @dev Disables borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function disableBorrowingOnReserve(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setBorrowingEnabled(false);
```

```
    pool.setConfiguration(asset, currentConfig.data);
    emit BorrowingDisabledOnReserve(asset);
  }

  /**
   * @dev Configures the reserve collateralization parameters
   * all the values are expressed in percentages with two decimals of precision. A valid value is 100
   * @param asset The address of the underlying asset of the reserve
   * @param ltv The loan to value of the asset when used as collateral
   * @param liquidationThreshold The threshold at which loans using this asset as collateral will be
   * @param liquidationBonus The bonus liquidators receive to liquidate this asset. The values is alw
   * means the liquidator will receive a 5% bonus
   **/
  function configureReserveAsCollateral(
    address asset,
    uint256 ltv,
    uint256 liquidationThreshold,
    uint256 liquidationBonus
  ) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    //validation of the parameters: the LTV can
    //only be lower or equal than the liquidation threshold
    //(otherwise a loan against the asset would cause instantaneous liquidation)
    require(ltv <= liquidationThreshold, Errors.LPC_INVALID_CONFIGURATION);

    if (liquidationThreshold != 0) {
      //liquidation bonus must be bigger than 100.00%, otherwise the liquidator would receive less
      //collateral than needed to cover the debt
      require(
        liquidationBonus > PercentageMath.PERCENTAGE_FACTOR,
        Errors.LPC_INVALID_CONFIGURATION
      );

      //if threshold * bonus is less than PERCENTAGE_FACTOR, it's guaranteed that at the moment
      //a loan is taken there is enough collateral available to cover the liquidation bonus
      require(
        liquidationThreshold.percentMul(liquidationBonus) <= PercentageMath.PERCENTAGE_FACTOR,
        Errors.LPC_INVALID_CONFIGURATION
      );
    } else {
      require(liquidationBonus == 0, Errors.LPC_INVALID_CONFIGURATION);
      //if the liquidation threshold is being set to 0,
      // the reserve is being disabled as collateral. To do so,
      //we need to ensure no liquidity is deposited
      _checkNoLiquidity(asset);
    }

    currentConfig.setLtv(ltv);
    currentConfig.setLiquidationThreshold(liquidationThreshold);
    currentConfig.setLiquidationBonus(liquidationBonus);

    pool.setConfiguration(asset, currentConfig.data);

    emit CollateralConfigurationChanged(asset, ltv, liquidationThreshold, liquidationBonus);
  }

  /**
   * @dev Enable stable rate borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function enableReserveStableRate(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setStableRateBorrowingEnabled(true);
```

```
    pool.setConfiguration(asset, currentConfig.data);

    emit StableRateEnabledOnReserve(asset);
  }

  /**
   * @dev Disable stable rate borrowing on a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function disableReserveStableRate(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setStableRateBorrowingEnabled(false);

    pool.setConfiguration(asset, currentConfig.data);

    emit StableRateDisabledOnReserve(asset);
  }

  /**
   * @dev Activates a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function activateReserve(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setActive(true);

    pool.setConfiguration(asset, currentConfig.data);

    emit ReserveActivated(asset);
  }

  /**
   * @dev Deactivates a reserve
   * @param asset The address of the underlying asset of the reserve
   **/
  function deactivateReserve(address asset) external onlyPoolAdmin {
    _checkNoLiquidity(asset);

    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setActive(false);

    pool.setConfiguration(asset, currentConfig.data);

    emit ReserveDeactivated(asset);
  }

  /**
   * @dev Freezes a reserve. A frozen reserve doesn't allow any new deposit, borrow or rate swap
   *  but allows repayments, liquidations, rate rebalances and withdrawals
   * @param asset The address of the underlying asset of the reserve
   **/
  function freezeReserve(address asset) external onlyPoolAdmin {
    DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

    currentConfig.setFrozen(true);

    pool.setConfiguration(asset, currentConfig.data);

    emit ReserveFrozen(asset);
  }

  /**
```

```
  * @dev Unfreezes a reserve
  * @param asset The address of the underlying asset of the reserve
  **/
function unfreezeReserve(address asset) external onlyPoolAdmin {
  DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

  currentConfig.setFrozen(false);

  pool.setConfiguration(asset, currentConfig.data);

  emit ReserveUnfrozen(asset);
}

/**
  * @dev Updates the reserve factor of a reserve
  * @param asset The address of the underlying asset of the reserve
  * @param reserveFactor The new reserve factor of the reserve
  **/
function setReserveFactor(address asset, uint256 reserveFactor) external onlyPoolAdmin {
  DataTypes.ReserveConfigurationMap memory currentConfig = pool.getConfiguration(asset);

  currentConfig.setReserveFactor(reserveFactor);

  pool.setConfiguration(asset, currentConfig.data);

  emit ReserveFactorChanged(asset, reserveFactor);
}

/**
  * @dev Sets the interest rate strategy of a reserve
  * @param asset The address of the underlying asset of the reserve
  * @param rateStrategyAddress The new address of the interest strategy contract
  **/
function setReserveInterestRateStrategyAddress(address asset, address rateStrategyAddress)
  external
  onlyPoolAdmin
{
  pool.setReserveInterestRateStrategyAddress(asset, rateStrategyAddress);
  emit ReserveInterestRateStrategyChanged(asset, rateStrategyAddress);
}

/**
  * @dev pauses or unpauses all the actions of the protocol, including aToken transfers
  * @param val true if protocol needs to be paused, false otherwise
  **/
function setPoolPause(bool val) external onlyEmergencyAdmin {
  pool.setPause(val);
}

function _initTokenWithProxy(address implementation, bytes memory initParams)
  internal
  returns (address)
{
  InitializableImmutableAdminUpgradeabilityProxy proxy =
    new InitializableImmutableAdminUpgradeabilityProxy(address(this));

  proxy.initialize(implementation, initParams);

  return address(proxy);
}

function _upgradeTokenImplementation(
  address proxyAddress,
  address implementation,
  bytes memory initParams
) internal {
```

```
    InitializableImmutableAdminUpgradeabilityProxy proxy =
      InitializableImmutableAdminUpgradeabilityProxy(payable(proxyAddress));

    proxy.upgradeToAndCall(implementation, initParams);
  }

  function _checkNoLiquidity(address asset) internal view {
    DataTypes.ReserveData memory reserveData = pool.getReserveData(asset);

    uint256 availableLiquidity = IERC20Detailed(asset).balanceOf(reserveData.aTokenAddress);

    require(
      availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
      Errors.LPC_RESERVE_LIQUIDITY_NOT_0
    );
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {SafeMath} from '../../dependencies/openzeppelin/contracts//SafeMath.sol';
import {IERC20} from '../../dependencies/openzeppelin/contracts//IERC20.sol';
import {IAToken} from '../../interfaces/IAToken.sol';
import {IStableDebtToken} from '../../interfaces/IStableDebtToken.sol';
import {IVariableDebtToken} from '../../interfaces/IVariableDebtToken.sol';
import {IPriceOracleGetter} from '../../interfaces/IPriceOracleGetter.sol';
import {ILendingPoolCollateralManager} from '../../interfaces/ILendingPoolCollateralManager.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {GenericLogic} from '../libraries/logic/GenericLogic.sol';
import {Helpers} from '../libraries/helpers/Helpers.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {PercentageMath} from '../libraries/math/PercentageMath.sol';
import {SafeERC20} from '../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {ValidationLogic} from '../libraries/logic/ValidationLogic.sol';
import {DataTypes} from '../libraries/types/DataTypes.sol';
import {LendingPoolStorage} from './LendingPoolStorage.sol';

/**
 * @title LendingPoolCollateralManager contract
 * @author Aave
 * @dev Implements actions involving management of collateral in the protocol, the main one being the
 * IMPORTANT This contract will run always via DELEGATECALL, through the LendingPool, so the chain of
 * is the same as the LendingPool, to have compatible storage layouts
 **/
contract LendingPoolCollateralManager is
  ILendingPoolCollateralManager,
  VersionedInitializable,
  LendingPoolStorage
{
  using SafeERC20 for IERC20;
  using SafeMath for uint256;
  using WadRayMath for uint256;
  using PercentageMath for uint256;

  uint256 internal constant LIQUIDATION_CLOSE_FACTOR_PERCENT = 5000;

  struct LiquidationCallLocalVars {
    uint256 userCollateralBalance;
    uint256 userStableDebt;
    uint256 userVariableDebt;
    uint256 maxLiquidatableDebt;
    uint256 actualDebtToLiquidate;
    uint256 liquidationRatio;
    uint256 maxAmountCollateralToLiquidate;
    uint256 userStableRate;
```

```
    uint256 maxCollateralToLiquidate;
    uint256 debtAmountNeeded;
    uint256 healthFactor;
    uint256 liquidatorPreviousATokenBalance;
    IAToken collateralAtoken;
    bool isCollateralEnabled;
    DataTypes.InterestRateMode borrowRateMode;
    uint256 errorCode;
    string errorMsg;
  }

  /**
   * @dev As thIS contract extends the VersionedInitializable contract to match the state
   * of the LendingPool contract, the getRevision() function is needed, but the value is not
   * important, as the initialize() function will never be called here
   */
  function getRevision() internal pure override returns (uint256) {
    return 0;
  }

  /**
   * @dev Function to liquidate a position if its Health Factor drops below 1
   * - The caller (liquidator) covers `debtToCover` amount of debt of the user getting liquidated, an
   *   a proportionally amount of the `collateralAsset` plus a bonus to cover market risk
   * @param collateralAsset The address of the underlying asset used as collateral, to receive as res
   * @param debtAsset The address of the underlying borrowed asset to be repaid with the liquidation
   * @param user The address of the borrower getting liquidated
   * @param debtToCover The debt amount of borrowed `asset` the liquidator wants to cover
   * @param receiveAToken `true` if the liquidators wants to receive the collateral aTokens, `false`
   * to receive the underlying collateral asset directly
   **/
  function liquidationCall(
    address collateralAsset,
    address debtAsset,
    address user,
    uint256 debtToCover,
    bool receiveAToken
  ) external override returns (uint256, string memory) {
    DataTypes.ReserveData storage collateralReserve = _reserves[collateralAsset];
    DataTypes.ReserveData storage debtReserve = _reserves[debtAsset];
    DataTypes.UserConfigurationMap storage userConfig = _usersConfig[user];

    LiquidationCallLocalVars memory vars;

    (, , , , vars.healthFactor) = GenericLogic.calculateUserAccountData(
      user,
      _reserves,
      userConfig,
      _reservesList,
      _reservesCount,
      _addressesProvider.getPriceOracle()
    );

    (vars.userStableDebt, vars.userVariableDebt) = Helpers.getUserCurrentDebt(user, debtReserve);

    (vars.errorCode, vars.errorMsg) = ValidationLogic.validateLiquidationCall(
      collateralReserve,
      debtReserve,
      userConfig,
      vars.healthFactor,
      vars.userStableDebt,
      vars.userVariableDebt
    );

    if (Errors.CollateralManagerErrors(vars.errorCode) != Errors.CollateralManagerErrors.NO_ERROR) {
      return (vars.errorCode, vars.errorMsg);
```

```
    }

    vars.collateralAtoken = IAToken(collateralReserve.aTokenAddress);

    vars.userCollateralBalance = vars.collateralAtoken.balanceOf(user);

    vars.maxLiquidatableDebt = vars.userStableDebt.add(vars.userVariableDebt).percentMul(
      LIQUIDATION_CLOSE_FACTOR_PERCENT
    );

    vars.actualDebtToLiquidate = debtToCover > vars.maxLiquidatableDebt
      ? vars.maxLiquidatableDebt
      : debtToCover;

    (
      vars.maxCollateralToLiquidate,
      vars.debtAmountNeeded
    ) = _calculateAvailableCollateralToLiquidate(
      collateralReserve,
      debtReserve,
      collateralAsset,
      debtAsset,
      vars.actualDebtToLiquidate,
      vars.userCollateralBalance
    );

    // If debtAmountNeeded < actualDebtToLiquidate, there isn't enough
    // collateral to cover the actual amount that is being liquidated, hence we liquidate
    // a smaller amount

    if (vars.debtAmountNeeded < vars.actualDebtToLiquidate) {
      vars.actualDebtToLiquidate = vars.debtAmountNeeded;
    }

    // If the liquidator reclaims the underlying asset, we make sure there is enough available liquid
    // collateral reserve
    if (!receiveAToken) {
      uint256 currentAvailableCollateral =
        IERC20(collateralAsset).balanceOf(address(vars.collateralAtoken));
      if (currentAvailableCollateral < vars.maxCollateralToLiquidate) {
        return (
          uint256(Errors.CollateralManagerErrors.NOT_ENOUGH_LIQUIDITY),
          Errors.LPCM_NOT_ENOUGH_LIQUIDITY_TO_LIQUIDATE
        );
      }
    }

    debtReserve.updateState();

    if (vars.userVariableDebt >= vars.actualDebtToLiquidate) {
      IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
        user,
        vars.actualDebtToLiquidate,
        debtReserve.variableBorrowIndex
      );
    } else {
      // If the user doesn't have variable debt, no need to try to burn variable debt tokens
      if (vars.userVariableDebt > 0) {
        IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
          user,
          vars.userVariableDebt,
          debtReserve.variableBorrowIndex
        );
      }
      IStableDebtToken(debtReserve.stableDebtTokenAddress).burn(
        user,
```

```
        vars.actualDebtToLiquidate.sub(vars.userVariableDebt)
      );
    }

    debtReserve.updateInterestRates(
      debtAsset,
      debtReserve.aTokenAddress,
      vars.actualDebtToLiquidate,
      0
    );

    if (receiveAToken) {
      vars.liquidatorPreviousATokenBalance = IERC20(vars.collateralAtoken).balanceOf(msg.sender);
      vars.collateralAtoken.transferOnLiquidation(user, msg.sender, vars.maxCollateralToLiquidate);

      if (vars.liquidatorPreviousATokenBalance == 0) {
        DataTypes.UserConfigurationMap storage liquidatorConfig = _usersConfig[msg.sender];
        liquidatorConfig.setUsingAsCollateral(collateralReserve.id, true);
        emit ReserveUsedAsCollateralEnabled(collateralAsset, msg.sender);
      }
    } else {
      collateralReserve.updateState();
      collateralReserve.updateInterestRates(
        collateralAsset,
        address(vars.collateralAtoken),
        0,
        vars.maxCollateralToLiquidate
      );

      // Burn the equivalent amount of aToken, sending the underlying to the liquidator
      vars.collateralAtoken.burn(
        user,
        msg.sender,
        vars.maxCollateralToLiquidate,
        collateralReserve.liquidityIndex
      );
    }

    // If the collateral being liquidated is equal to the user balance,
    // we set the currency as not being used as collateral anymore
    if (vars.maxCollateralToLiquidate == vars.userCollateralBalance) {
      userConfig.setUsingAsCollateral(collateralReserve.id, false);
      emit ReserveUsedAsCollateralDisabled(collateralAsset, user);
    }

    // Transfers the debt asset being repaid to the aToken, where the liquidity is kept
    IERC20(debtAsset).safeTransferFrom(
      msg.sender,
      debtReserve.aTokenAddress,
      vars.actualDebtToLiquidate
    );

    emit LiquidationCall(
      collateralAsset,
      debtAsset,
      user,
      vars.actualDebtToLiquidate,
      vars.maxCollateralToLiquidate,
      msg.sender,
      receiveAToken
    );

    return (uint256(Errors.CollateralManagerErrors.NO_ERROR), Errors.LPCM_NO_ERRORS);
  }

  struct AvailableCollateralToLiquidateLocalVars {
```

```
    uint256 userCompoundedBorrowBalance;
    uint256 liquidationBonus;
    uint256 collateralPrice;
    uint256 debtAssetPrice;
    uint256 maxAmountCollateralToLiquidate;
    uint256 debtAssetDecimals;
    uint256 collateralDecimals;
  }

  /**
   * @dev Calculates how much of a specific collateral can be liquidated, given
   * a certain amount of debt asset.
   * - This function needs to be called after all the checks to validate the liquidation have been pe
   *   otherwise it might fail.
   * @param collateralReserve The data of the collateral reserve
   * @param debtReserve The data of the debt reserve
   * @param collateralAsset The address of the underlying asset used as collateral, to receive as res
   * @param debtAsset The address of the underlying borrowed asset to be repaid with the liquidation
   * @param debtToCover The debt amount of borrowed `asset` the liquidator wants to cover
   * @param userCollateralBalance The collateral balance for the specific `collateralAsset` of the us
   * @return collateralAmount: The maximum amount that is possible to liquidate given all the liquida
   *                           (user balance, close factor)
   *         debtAmountNeeded: The amount to repay with the liquidation
   **/
  function _calculateAvailableCollateralToLiquidate(
    DataTypes.ReserveData storage collateralReserve,
    DataTypes.ReserveData storage debtReserve,
    address collateralAsset,
    address debtAsset,
    uint256 debtToCover,
    uint256 userCollateralBalance
  ) internal view returns (uint256, uint256) {
    uint256 collateralAmount = 0;
    uint256 debtAmountNeeded = 0;
    IPriceOracleGetter oracle = IPriceOracleGetter(_addressesProvider.getPriceOracle());

    AvailableCollateralToLiquidateLocalVars memory vars;

    vars.collateralPrice = oracle.getAssetPrice(collateralAsset);
    vars.debtAssetPrice = oracle.getAssetPrice(debtAsset);

    (, , vars.liquidationBonus, vars.collateralDecimals, ) = collateralReserve
      .configuration
      .getParams();
    vars.debtAssetDecimals = debtReserve.configuration.getDecimals();

    // This is the maximum possible amount of the selected collateral that can be liquidated, given t
    // max amount of liquidatable debt
    vars.maxAmountCollateralToLiquidate = vars
      .debtAssetPrice
      .mul(debtToCover)
      .mul(10**vars.collateralDecimals)
      .percentMul(vars.liquidationBonus)
      .div(vars.collateralPrice.mul(10**vars.debtAssetDecimals));

    if (vars.maxAmountCollateralToLiquidate > userCollateralBalance) {
      collateralAmount = userCollateralBalance;
      debtAmountNeeded = vars
        .collateralPrice
        .mul(collateralAmount)
        .mul(10**vars.debtAssetDecimals)
        .div(vars.debtAssetPrice.mul(10**vars.collateralDecimals))
        .percentDiv(vars.liquidationBonus);
    } else {
      collateralAmount = vars.maxAmountCollateralToLiquidate;
      debtAmountNeeded = debtToCover;
```

```
    }
    return (collateralAmount, debtAmountNeeded);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ILendingPool} from '../../../interfaces/ILendingPool.sol';
import {ICreditDelegationToken} from '../../../interfaces/ICreditDelegationToken.sol';
import {
  VersionedInitializable
} from '../../../libraries/aave-upgradeability/VersionedInitializable.sol';
import {IncentivizedERC20} from '../IncentivizedERC20.sol';
import {Errors} from '../../../libraries/helpers/Errors.sol';

/**
 * @title DebtTokenBase
 * @notice Base contract for different types of debt tokens, like StableDebtToken or VariableDebtToke
 * @author Aave
 */

abstract contract DebtTokenBase is
  IncentivizedERC20('DEBTTOKEN_IMPL', 'DEBTTOKEN_IMPL', 0),
  VersionedInitializable,
  ICreditDelegationToken
{
  mapping(address => mapping(address => uint256)) internal _borrowAllowances;

  /**
   * @dev Only lending pool can call functions marked by this modifier
   **/
  modifier onlyLendingPool {
    require(_msgSender() == address(_getLendingPool()), Errors.CT_CALLER_MUST_BE_LENDING_POOL);
    _;
  }

  /**
   * @dev delegates borrowing power to a user on the specific debt token
   * @param delegatee the address receiving the delegated borrowing power
   * @param amount the maximum amount being delegated. Delegation will still
   * respect the liquidation constraints (even if delegated, a delegatee cannot
   * force a delegator HF to go below 1)
   **/
  function approveDelegation(address delegatee, uint256 amount) external override {
    _borrowAllowances[_msgSender()][delegatee] = amount;
    emit BorrowAllowanceDelegated(_msgSender(), delegatee, _getUnderlyingAssetAddress(), amount);
  }

  /**
   * @dev returns the borrow allowance of the user
   * @param fromUser The user to giving allowance
   * @param toUser The user to give allowance to
   * @return the current allowance of toUser
   **/
  function borrowAllowance(address fromUser, address toUser)
    external
    view
    override
    returns (uint256)
  {
    return _borrowAllowances[fromUser][toUser];
  }

  /**
   * @dev Being non transferrable, the debt token does not implement any of the
   * standard ERC20 functions for transfer and allowance.
```

```
 **/
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
  recipient;
  amount;
  revert('TRANSFER_NOT_SUPPORTED');
}

function allowance(address owner, address spender)
  public
  view
  virtual
  override
  returns (uint256)
{
  owner;
  spender;
  revert('ALLOWANCE_NOT_SUPPORTED');
}

function approve(address spender, uint256 amount) public virtual override returns (bool) {
  spender;
  amount;
  revert('APPROVAL_NOT_SUPPORTED');
}

function transferFrom(
  address sender,
  address recipient,
  uint256 amount
) public virtual override returns (bool) {
  sender;
  recipient;
  amount;
  revert('TRANSFER_NOT_SUPPORTED');
}

function increaseAllowance(address spender, uint256 addedValue)
  public
  virtual
  override
  returns (bool)
{
  spender;
  addedValue;
  revert('ALLOWANCE_NOT_SUPPORTED');
}

function decreaseAllowance(address spender, uint256 subtractedValue)
  public
  virtual
  override
  returns (bool)
{
  spender;
  subtractedValue;
  revert('ALLOWANCE_NOT_SUPPORTED');
}

function _decreaseBorrowAllowance(
  address delegator,
  address delegatee,
  uint256 amount
) internal {
  uint256 newAllowance =
    _borrowAllowances[delegator][delegatee].sub(amount, Errors.BORROW_ALLOWANCE_NOT_ENOUGH);
```

```
    _borrowAllowances[delegator][delegatee] = newAllowance;

    emit BorrowAllowanceDelegated(delegator, delegatee, _getUnderlyingAssetAddress(), newAllowance);
  }

  function _getUnderlyingAssetAddress() internal view virtual returns (address);

  function _getLendingPool() internal view virtual returns (ILendingPool);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IDelegationToken} from '../../interfaces/IDelegationToken.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {AToken} from './AToken.sol';

/**
 * @title Aave AToken enabled to delegate voting power of the underlying asset to a different address
 * @dev The underlying asset needs to be compatible with the COMP delegation interface
 * @author Aave
 */
contract DelegationAwareAToken is AToken {
  modifier onlyPoolAdmin {
    require(
      _msgSender() == ILendingPool(_pool).getAddressesProvider().getPoolAdmin(),
      Errors.CALLER_NOT_POOL_ADMIN
    );
    _;
  }

  /**
   * @dev Delegates voting power of the underlying asset to a `delegatee` address
   * @param delegatee The address that will receive the delegation
   **/
  function delegateUnderlyingTo(address delegatee) external onlyPoolAdmin {
    IDelegationToken(_underlyingAsset).delegate(delegatee);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {Context} from '../../dependencies/openzeppelin/contracts/Context.sol';
import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';
import {IERC20Detailed} from '../../dependencies/openzeppelin/contracts/IERC20Detailed.sol';
import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';

/**
 * @title ERC20
 * @notice Basic ERC20 implementation
 * @author Aave, inspired by the Openzeppelin ERC20 implementation
 **/
abstract contract IncentivizedERC20 is Context, IERC20, IERC20Detailed {
  using SafeMath for uint256;

  mapping(address => uint256) internal _balances;

  mapping(address => mapping(address => uint256)) private _allowances;
  uint256 internal _totalSupply;
  string private _name;
  string private _symbol;
  uint8 private _decimals;

  constructor(
    string memory name,
```

```
    string memory symbol,
    uint8 decimals
  ) public {
    _name = name;
    _symbol = symbol;
    _decimals = decimals;
  }

  /**
   * @return The name of the token
   **/
  function name() public view override returns (string memory) {
    return _name;
  }

  /**
   * @return The symbol of the token
   **/
  function symbol() public view override returns (string memory) {
    return _symbol;
  }

  /**
   * @return The decimals of the token
   **/
  function decimals() public view override returns (uint8) {
    return _decimals;
  }

  /**
   * @return The total supply of the token
   **/
  function totalSupply() public view virtual override returns (uint256) {
    return _totalSupply;
  }

  /**
   * @return The balance of the token
   **/
  function balanceOf(address account) public view virtual override returns (uint256) {
    return _balances[account];
  }

  /**
   * @return Abstract function implemented by the child aToken/debtToken.
   * Done this way in order to not break compatibility with previous versions of aTokens/debtTokens
   **/
  function _getIncentivesController() internal view virtual returns(IAaveIncentivesController);

  /**
   * @dev Executes a transfer of tokens from _msgSender() to recipient
   * @param recipient The recipient of the tokens
   * @param amount The amount of tokens being transferred
   * @return `true` if the transfer succeeds, `false` otherwise
   **/
  function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    emit Transfer(_msgSender(), recipient, amount);
    return true;
  }

  /**
   * @dev Returns the allowance of spender on the tokens owned by owner
   * @param owner The owner of the tokens
   * @param spender The user allowed to spend the owner's tokens
   * @return The amount of owner's tokens spender is allowed to spend
```

```
 **/
function allowance(address owner, address spender)
  public
  view
  virtual
  override
  returns (uint256)
{
  return _allowances[owner][spender];
}

/**
 * @dev Allows `spender` to spend the tokens owned by _msgSender()
 * @param spender The user allowed to spend _msgSender() tokens
 * @return `true`
 **/
function approve(address spender, uint256 amount) public virtual override returns (bool) {
  _approve(_msgSender(), spender, amount);
  return true;
}

/**
 * @dev Executes a transfer of token from sender to recipient, if _msgSender() is allowed to do so
 * @param sender The owner of the tokens
 * @param recipient The recipient of the tokens
 * @param amount The amount of tokens being transferred
 * @return `true` if the transfer succeeds, `false` otherwise
 **/
function transferFrom(
  address sender,
  address recipient,
  uint256 amount
) public virtual override returns (bool) {
  _transfer(sender, recipient, amount);
  _approve(
    sender,
    _msgSender(),
    _allowances[sender][_msgSender()].sub(amount, 'ERC20: transfer amount exceeds allowance')
  );
  emit Transfer(sender, recipient, amount);
  return true;
}

/**
 * @dev Increases the allowance of spender to spend _msgSender() tokens
 * @param spender The user allowed to spend on behalf of _msgSender()
 * @param addedValue The amount being added to the allowance
 * @return `true`
 **/
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
  _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
  return true;
}

/**
 * @dev Decreases the allowance of spender to spend _msgSender() tokens
 * @param spender The user allowed to spend on behalf of _msgSender()
 * @param subtractedValue The amount being subtracted to the allowance
 * @return `true`
 **/
function decreaseAllowance(address spender, uint256 subtractedValue)
  public
  virtual
  returns (bool)
{
  _approve(
```

```
      _msgSender(),
      spender,
      _allowances[_msgSender()][spender].sub(
        subtractedValue,
        'ERC20: decreased allowance below zero'
      )
    );
    return true;
  }

  function _transfer(
    address sender,
    address recipient,
    uint256 amount
  ) internal virtual {
    require(sender != address(0), 'ERC20: transfer from the zero address');
    require(recipient != address(0), 'ERC20: transfer to the zero address');

    _beforeTokenTransfer(sender, recipient, amount);

    uint256 oldSenderBalance = _balances[sender];
    _balances[sender] = oldSenderBalance.sub(amount, 'ERC20: transfer amount exceeds balance');
    uint256 oldRecipientBalance = _balances[recipient];
    _balances[recipient] = _balances[recipient].add(amount);

    if (address(_getIncentivesController()) != address(0)) {
      uint256 currentTotalSupply = _totalSupply;
      _getIncentivesController().handleAction(sender, currentTotalSupply, oldSenderBalance);
      if (sender != recipient) {
        _getIncentivesController().handleAction(recipient, currentTotalSupply, oldRecipientBalance);
      }
    }
  }

  function _mint(address account, uint256 amount) internal virtual {
    require(account != address(0), 'ERC20: mint to the zero address');

    _beforeTokenTransfer(address(0), account, amount);

    uint256 oldTotalSupply = _totalSupply;
    _totalSupply = oldTotalSupply.add(amount);

    uint256 oldAccountBalance = _balances[account];
    _balances[account] = oldAccountBalance.add(amount);

    if (address(_getIncentivesController()) != address(0)) {
      _getIncentivesController().handleAction(account, oldTotalSupply, oldAccountBalance);
    }
  }

  function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), 'ERC20: burn from the zero address');

    _beforeTokenTransfer(account, address(0), amount);

    uint256 oldTotalSupply = _totalSupply;
    _totalSupply = oldTotalSupply.sub(amount);

    uint256 oldAccountBalance = _balances[account];
    _balances[account] = oldAccountBalance.sub(amount, 'ERC20: burn amount exceeds balance');

    if (address(_getIncentivesController()) != address(0)) {
      _getIncentivesController().handleAction(account, oldTotalSupply, oldAccountBalance);
    }
  }
```

```
  function _approve(
    address owner,
    address spender,
    uint256 amount
  ) internal virtual {
    require(owner != address(0), 'ERC20: approve from the zero address');
    require(spender != address(0), 'ERC20: approve to the zero address');

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
  }

  function _setName(string memory newName) internal {
    _name = newName;
  }

  function _setSymbol(string memory newSymbol) internal {
    _symbol = newSymbol;
  }

  function _setDecimals(uint8 newDecimals) internal {
    _decimals = newDecimals;
  }

  function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
  ) internal virtual {}
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {DebtTokenBase} from './base/DebtTokenBase.sol';
import {MathUtils} from '../libraries/math/MathUtils.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {IStableDebtToken} from '../../interfaces/IStableDebtToken.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';
import {Errors} from '../libraries/helpers/Errors.sol';

/**
 * @title StableDebtToken
 * @notice Implements a stable debt token to track the borrowing positions of users
 * at stable rate mode
 * @author Aave
 **/
contract StableDebtToken is IStableDebtToken, DebtTokenBase {
  using WadRayMath for uint256;

  uint256 public constant DEBT_TOKEN_REVISION = 0x1;

  uint256 internal _avgStableRate;
  mapping(address => uint40) internal _timestamps;
  mapping(address => uint256) internal _usersStableRate;
  uint40 internal _totalSupplyTimestamp;

  ILendingPool internal _pool;
  address internal _underlyingAsset;
  IAaveIncentivesController internal _incentivesController;

  /**
   * @dev Initializes the debt token.
   * @param pool The address of the lending pool where this aToken will be used
   * @param underlyingAsset The address of the underlying asset of this aToken (E.g. WETH for aWETH)
   * @param incentivesController The smart contract managing potential incentives distribution
```

```
   * @param debtTokenDecimals The decimals of the debtToken, same as the underlying asset's
   * @param debtTokenName The name of the token
   * @param debtTokenSymbol The symbol of the token
   */
  function initialize(
    ILendingPool pool,
    address underlyingAsset,
    IAaveIncentivesController incentivesController,
    uint8 debtTokenDecimals,
    string memory debtTokenName,
    string memory debtTokenSymbol,
    bytes calldata params
  ) public override initializer {
    _setName(debtTokenName);
    _setSymbol(debtTokenSymbol);
    _setDecimals(debtTokenDecimals);

    _pool = pool;
    _underlyingAsset = underlyingAsset;
    _incentivesController = incentivesController;

    emit Initialized(
      underlyingAsset,
      address(pool),
      address(incentivesController),
      debtTokenDecimals,
      debtTokenName,
      debtTokenSymbol,
      params
    );
  }

  /**
   * @dev Gets the revision of the stable debt token implementation
   * @return The debt token implementation revision
   **/
  function getRevision() internal pure virtual override returns (uint256) {
    return DEBT_TOKEN_REVISION;
  }

  /**
   * @dev Returns the average stable rate across all the stable rate debt
   * @return the average stable rate
   **/
  function getAverageStableRate() external view virtual override returns (uint256) {
    return _avgStableRate;
  }

  /**
   * @dev Returns the timestamp of the last user action
   * @return The last update timestamp
   **/
  function getUserLastUpdated(address user) external view virtual override returns (uint40) {
    return _timestamps[user];
  }

  /**
   * @dev Returns the stable rate of the user
   * @param user The address of the user
   * @return The stable rate of user
   **/
  function getUserStableRate(address user) external view virtual override returns (uint256) {
    return _usersStableRate[user];
  }

  /**
```

```
   * @dev Calculates the current user debt balance
   * @return The accumulated debt of the user
   **/
  function balanceOf(address account) public view virtual override returns (uint256) {
    uint256 accountBalance = super.balanceOf(account);
    uint256 stableRate = _usersStableRate[account];
    if (accountBalance == 0) {
      return 0;
    }
    uint256 cumulatedInterest =
      MathUtils.calculateCompoundedInterest(stableRate, _timestamps[account]);
    return accountBalance.rayMul(cumulatedInterest);
  }

  struct MintLocalVars {
    uint256 previousSupply;
    uint256 nextSupply;
    uint256 amountInRay;
    uint256 newStableRate;
    uint256 currentAvgStableRate;
  }

  /**
   * @dev Mints debt token to the `onBehalfOf` address.
   * -  Only callable by the LendingPool
   * - The resulting rate is the weighted average between the rate of the new debt
   * and the rate of the previous debt
   * @param user The address receiving the borrowed underlying, being the delegatee in case
   * of credit delegate, or same as `onBehalfOf` otherwise
   * @param onBehalfOf The address receiving the debt tokens
   * @param amount The amount of debt tokens to mint
   * @param rate The rate of the debt being minted
   **/
  function mint(
    address user,
    address onBehalfOf,
    uint256 amount,
    uint256 rate
  ) external override onlyLendingPool returns (bool) {
    MintLocalVars memory vars;

    if (user != onBehalfOf) {
      _decreaseBorrowAllowance(onBehalfOf, user, amount);
    }

    (, uint256 currentBalance, uint256 balanceIncrease) = _calculateBalanceIncrease(onBehalfOf);

    vars.previousSupply = totalSupply();
    vars.currentAvgStableRate = _avgStableRate;
    vars.nextSupply = _totalSupply = vars.previousSupply.add(amount);

    vars.amountInRay = amount.wadToRay();

    vars.newStableRate = _usersStableRate[onBehalfOf]
      .rayMul(currentBalance.wadToRay())
      .add(vars.amountInRay.rayMul(rate))
      .rayDiv(currentBalance.add(amount).wadToRay());

    require(vars.newStableRate <= type(uint128).max, Errors.SDT_STABLE_DEBT_OVERFLOW);
    _usersStableRate[onBehalfOf] = vars.newStableRate;

    //solium-disable-next-line
    _totalSupplyTimestamp = _timestamps[onBehalfOf] = uint40(block.timestamp);

    // Calculates the updated average stable rate
    vars.currentAvgStableRate = _avgStableRate = vars
```

```
      .currentAvgStableRate
      .rayMul(vars.previousSupply.wadToRay())
      .add(rate.rayMul(vars.amountInRay))
      .rayDiv(vars.nextSupply.wadToRay());

    _mint(onBehalfOf, amount.add(balanceIncrease), vars.previousSupply);

    emit Transfer(address(0), onBehalfOf, amount);

    emit Mint(
      user,
      onBehalfOf,
      amount,
      currentBalance,
      balanceIncrease,
      vars.newStableRate,
      vars.currentAvgStableRate,
      vars.nextSupply
    );

    return currentBalance == 0;
  }

  /**
   * @dev Burns debt of `user`
   * @param user The address of the user getting his debt burned
   * @param amount The amount of debt tokens getting burned
   **/
  function burn(address user, uint256 amount) external override onlyLendingPool {
    (, uint256 currentBalance, uint256 balanceIncrease) = _calculateBalanceIncrease(user);

    uint256 previousSupply = totalSupply();
    uint256 newAvgStableRate = 0;
    uint256 nextSupply = 0;
    uint256 userStableRate = _usersStableRate[user];

    // Since the total supply and each single user debt accrue separately,
    // there might be accumulation errors so that the last borrower repaying
    // mght actually try to repay more than the available debt supply.
    // In this case we simply set the total supply and the avg stable rate to 0
    if (previousSupply <= amount) {
      _avgStableRate = 0;
      _totalSupply = 0;
    } else {
      nextSupply = _totalSupply = previousSupply.sub(amount);
      uint256 firstTerm = _avgStableRate.rayMul(previousSupply.wadToRay());
      uint256 secondTerm = userStableRate.rayMul(amount.wadToRay());

      // For the same reason described above, when the last user is repaying it might
      // happen that user rate * user balance > avg rate * total supply. In that case,
      // we simply set the avg rate to 0
      if (secondTerm >= firstTerm) {
        newAvgStableRate = _avgStableRate = _totalSupply = 0;
      } else {
        newAvgStableRate = _avgStableRate = firstTerm.sub(secondTerm).rayDiv(nextSupply.wadToRay());
      }
    }

    if (amount == currentBalance) {
      _usersStableRate[user] = 0;
      _timestamps[user] = 0;
    } else {
      //solium-disable-next-line
      _timestamps[user] = uint40(block.timestamp);
    }
    //solium-disable-next-line
```

```
    _totalSupplyTimestamp = uint40(block.timestamp);

    if (balanceIncrease > amount) {
      uint256 amountToMint = balanceIncrease.sub(amount);
      _mint(user, amountToMint, previousSupply);
      emit Mint(
        user,
        user,
        amountToMint,
        currentBalance,
        balanceIncrease,
        userStableRate,
        newAvgStableRate,
        nextSupply
      );
    } else {
      uint256 amountToBurn = amount.sub(balanceIncrease);
      _burn(user, amountToBurn, previousSupply);
      emit Burn(user, amountToBurn, currentBalance, balanceIncrease, newAvgStableRate, nextSupply);
    }

    emit Transfer(user, address(0), amount);
  }

  /**
   * @dev Calculates the increase in balance since the last user interaction
   * @param user The address of the user for which the interest is being accumulated
   * @return The previous principal balance, the new principal balance and the balance increase
   **/
  function _calculateBalanceIncrease(address user)
    internal
    view
    returns (
      uint256,
      uint256,
      uint256
    )
  {
    uint256 previousPrincipalBalance = super.balanceOf(user);

    if (previousPrincipalBalance == 0) {
      return (0, 0, 0);
    }

    // Calculation of the accrued interest since the last accumulation
    uint256 balanceIncrease = balanceOf(user).sub(previousPrincipalBalance);

    return (
      previousPrincipalBalance,
      previousPrincipalBalance.add(balanceIncrease),
      balanceIncrease
    );
  }

  /**
   * @dev Returns the principal and total supply, the average borrow rate and the last supply update
   **/
  function getSupplyData()
    public
    view
    override
    returns (
      uint256,
      uint256,
      uint256,
      uint40
```

```
  )
{
  uint256 avgRate = _avgStableRate;
  return (super.totalSupply(), _calcTotalSupply(avgRate), avgRate, _totalSupplyTimestamp);
}

/**
 * @dev Returns the the total supply and the average stable rate
 **/
function getTotalSupplyAndAvgRate() public view override returns (uint256, uint256) {
  uint256 avgRate = _avgStableRate;
  return (_calcTotalSupply(avgRate), avgRate);
}

/**
 * @dev Returns the total supply
 **/
function totalSupply() public view override returns (uint256) {
  return _calcTotalSupply(_avgStableRate);
}

/**
 * @dev Returns the timestamp at which the total supply was updated
 **/
function getTotalSupplyLastUpdated() public view override returns (uint40) {
  return _totalSupplyTimestamp;
}

/**
 * @dev Returns the principal debt balance of the user from
 * @param user The user's address
 * @return The debt balance of the user since the last burn/mint action
 **/
function principalBalanceOf(address user) external view virtual override returns (uint256) {
  return super.balanceOf(user);
}

/**
 * @dev Returns the principal debt balance of the user from
 * @return The debt balance of the user since the last burn/mint action
 **/
function scaledBalanceOf(address user) public view virtual override returns (uint256) {
  return super.balanceOf(user);
}

/**
 * @dev Returns the scaled total supply of the variable debt token. Represents sum(debt/index)
 * @return the scaled total supply
 **/
function scaledTotalSupply() public view virtual override returns (uint256) {
  return super.totalSupply();
}

/**
 * @dev Returns the principal balance of the user and principal total supply.
 * @param user The address of the user
 * @return The principal balance of the user
 * @return The principal total supply
 **/
function getScaledUserBalanceAndSupply(address user)
  external
  view
  override
  returns (uint256, uint256)
{
  return (super.balanceOf(user), super.totalSupply());
```

```
}

/**
 * @dev Returns the address of the underlying asset of this aToken (E.g. WETH for aWETH)
 **/
function UNDERLYING_ASSET_ADDRESS() public view returns (address) {
  return _underlyingAsset;
}

/**
 * @dev Returns the address of the lending pool where this aToken is used
 **/
function POOL() public view returns (ILendingPool) {
  return _pool;
}

/**
 * @dev Returns the address of the incentives controller contract
 **/
function getIncentivesController() external view override returns (IAaveIncentivesController) {
  return _getIncentivesController();
}

/**
 * @dev For internal usage in the logic of the parent contracts
 **/
function _getIncentivesController() internal view override returns (IAaveIncentivesController) {
  return _incentivesController;
}

/**
 * @dev For internal usage in the logic of the parent contracts
 **/
function _getUnderlyingAssetAddress() internal view override returns (address) {
  return _underlyingAsset;
}

/**
 * @dev For internal usage in the logic of the parent contracts
 **/
function _getLendingPool() internal view override returns (ILendingPool) {
  return _pool;
}

/**
 * @dev Calculates the total supply
 * @param avgRate The average rate at which the total supply increases
 * @return The debt balance of the user since the last burn/mint action
 **/
function _calcTotalSupply(uint256 avgRate) internal view virtual returns (uint256) {
  uint256 principalSupply = super.totalSupply();

  if (principalSupply == 0) {
    return 0;
  }

  uint256 cumulatedInterest =
    MathUtils.calculateCompoundedInterest(avgRate, _totalSupplyTimestamp);

  return principalSupply.rayMul(cumulatedInterest);
}

/**
 * @dev Mints stable debt tokens to an user
 * @param account The account receiving the debt tokens
 * @param amount The amount being minted
```

```
    * @param oldTotalSupply the total supply before the minting event
    **/
  function _mint(
    address account,
    uint256 amount,
    uint256 oldTotalSupply
  ) internal {
    uint256 oldAccountBalance = _balances[account];
    _balances[account] = oldAccountBalance.add(amount);

    if (address(_incentivesController) != address(0)) {
      _incentivesController.handleAction(account, oldTotalSupply, oldAccountBalance);
    }
  }

  /**
   * @dev Burns stable debt tokens of an user
   * @param account The user getting his debt burned
   * @param amount The amount being burned
   * @param oldTotalSupply The total supply before the burning event
   **/
  function _burn(
    address account,
    uint256 amount,
    uint256 oldTotalSupply
  ) internal {
    uint256 oldAccountBalance = _balances[account];
    _balances[account] = oldAccountBalance.sub(amount, Errors.SDT_BURN_EXCEEDS_BALANCE);

    if (address(_incentivesController) != address(0)) {
      _incentivesController.handleAction(account, oldTotalSupply, oldAccountBalance);
    }
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';
import {SafeERC20} from '../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IAToken} from '../../interfaces/IAToken.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {VersionedInitializable} from '../libraries/aave-upgradeability/VersionedInitializable.sol';
import {IncentivizedERC20} from './IncentivizedERC20.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';

/**
 * @title Aave ERC20 AToken
 * @dev Implementation of the interest bearing token for the Aave protocol
 * @author Aave
 */
contract AToken is
  VersionedInitializable,
  IncentivizedERC20('ATOKEN_IMPL', 'ATOKEN_IMPL', 0),
  IAToken
{
  using WadRayMath for uint256;
  using SafeERC20 for IERC20;

  bytes public constant EIP712_REVISION = bytes('1');
  bytes32 internal constant EIP712_DOMAIN =
    keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)');
  bytes32 public constant PERMIT_TYPEHASH =
    keccak256('Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)');
```

```
  uint256 public constant ATOKEN_REVISION = 0x1;

  /// @dev owner => next valid nonce to submit with permit()
  mapping(address => uint256) public _nonces;

  bytes32 public DOMAIN_SEPARATOR;

  ILendingPool internal _pool;
  address internal _treasury;
  address internal _underlyingAsset;
  IAaveIncentivesController internal _incentivesController;

  modifier onlyLendingPool {
    require(_msgSender() == address(_pool), Errors.CT_CALLER_MUST_BE_LENDING_POOL);
    _;
  }

  function getRevision() internal pure virtual override returns (uint256) {
    return ATOKEN_REVISION;
  }

  /**
   * @dev Initializes the aToken
   * @param pool The address of the lending pool where this aToken will be used
   * @param treasury The address of the Aave treasury, receiving the fees on this aToken
   * @param underlyingAsset The address of the underlying asset of this aToken (E.g. WETH for aWETH)
   * @param incentivesController The smart contract managing potential incentives distribution
   * @param aTokenDecimals The decimals of the aToken, same as the underlying asset's
   * @param aTokenName The name of the aToken
   * @param aTokenSymbol The symbol of the aToken
   */
  function initialize(
    ILendingPool pool,
    address treasury,
    address underlyingAsset,
    IAaveIncentivesController incentivesController,
    uint8 aTokenDecimals,
    string calldata aTokenName,
    string calldata aTokenSymbol,
    bytes calldata params
  ) external override initializer {
    uint256 chainId;

    //solium-disable-next-line
    assembly {
      chainId := chainid()
    }

    DOMAIN_SEPARATOR = keccak256(
      abi.encode(
        EIP712_DOMAIN,
        keccak256(bytes(aTokenName)),
        keccak256(EIP712_REVISION),
        chainId,
        address(this)
      )
    );

    _setName(aTokenName);
    _setSymbol(aTokenSymbol);
    _setDecimals(aTokenDecimals);

    _pool = pool;
    _treasury = treasury;
    _underlyingAsset = underlyingAsset;
    _incentivesController = incentivesController;
```

```
    emit Initialized(
      underlyingAsset,
      address(pool),
      treasury,
      address(incentivesController),
      aTokenDecimals,
      aTokenName,
      aTokenSymbol,
      params
    );
  }

  /**
   * @dev Burns aTokens from `user` and sends the equivalent amount of underlying to `receiverOfUnder
   * - Only callable by the LendingPool, as extra state updates there need to be managed
   * @param user The owner of the aTokens, getting them burned
   * @param receiverOfUnderlying The address that will receive the underlying
   * @param amount The amount being burned
   * @param index The new liquidity index of the reserve
   **/
  function burn(
    address user,
    address receiverOfUnderlying,
    uint256 amount,
    uint256 index
  ) external override onlyLendingPool {
    uint256 amountScaled = amount.rayDiv(index);
    require(amountScaled != 0, Errors.CT_INVALID_BURN_AMOUNT);
    _burn(user, amountScaled);

    IERC20(_underlyingAsset).safeTransfer(receiverOfUnderlying, amount);

    emit Transfer(user, address(0), amount);
    emit Burn(user, receiverOfUnderlying, amount, index);
  }

  /**
   * @dev Mints `amount` aTokens to `user`
   * - Only callable by the LendingPool, as extra state updates there need to be managed
   * @param user The address receiving the minted tokens
   * @param amount The amount of tokens getting minted
   * @param index The new liquidity index of the reserve
   * @return `true` if the the previous balance of the user was 0
   */
  function mint(
    address user,
    uint256 amount,
    uint256 index
  ) external override onlyLendingPool returns (bool) {
    uint256 previousBalance = super.balanceOf(user);

    uint256 amountScaled = amount.rayDiv(index);
    require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);
    _mint(user, amountScaled);

    emit Transfer(address(0), user, amount);
    emit Mint(user, amount, index);

    return previousBalance == 0;
  }

  /**
   * @dev Mints aTokens to the reserve treasury
   * - Only callable by the LendingPool
   * @param amount The amount of tokens getting minted
```

```
   * @param index The new liquidity index of the reserve
   */
  function mintToTreasury(uint256 amount, uint256 index) external override onlyLendingPool {
    if (amount == 0) {
      return;
    }

    address treasury = _treasury;

    // Compared to the normal mint, we don't check for rounding errors.
    // The amount to mint can easily be very small since it is a fraction of the interest ccrued.
    // In that case, the treasury will experience a (very small) loss, but it
    // wont cause potentially valid transactions to fail.
    _mint(treasury, amount.rayDiv(index));

    emit Transfer(address(0), treasury, amount);
    emit Mint(treasury, amount, index);
  }

  /**
   * @dev Transfers aTokens in the event of a borrow being liquidated, in case the liquidators reclai
   * - Only callable by the LendingPool
   * @param from The address getting liquidated, current owner of the aTokens
   * @param to The recipient
   * @param value The amount of tokens getting transferred
   **/
  function transferOnLiquidation(
    address from,
    address to,
    uint256 value
  ) external override onlyLendingPool {
    // Being a normal transfer, the Transfer() and BalanceTransfer() are emitted
    // so no need to emit a specific event here
    _transfer(from, to, value, false);

    emit Transfer(from, to, value);
  }

  /**
   * @dev Calculates the balance of the user: principal balance + interest generated by the principal
   * @param user The user whose balance is calculated
   * @return The balance of the user
   **/
  function balanceOf(address user)
    public
    view
    override(IncentivizedERC20, IERC20)
    returns (uint256)
  {
    return super.balanceOf(user).rayMul(_pool.getReserveNormalizedIncome(_underlyingAsset));
  }

  /**
   * @dev Returns the scaled balance of the user. The scaled balance is the sum of all the
   * updated stored balance divided by the reserve's liquidity index at the moment of the update
   * @param user The user whose balance is calculated
   * @return The scaled balance of the user
   **/
  function scaledBalanceOf(address user) external view override returns (uint256) {
    return super.balanceOf(user);
  }

  /**
   * @dev Returns the scaled balance of the user and the scaled total supply.
   * @param user The address of the user
   * @return The scaled balance of the user
```

```
   * @return The scaled balance and the scaled total supply
   **/
  function getScaledUserBalanceAndSupply(address user)
    external
    view
    override
    returns (uint256, uint256)
  {
    return (super.balanceOf(user), super.totalSupply());
  }

  /**
   * @dev calculates the total supply of the specific aToken
   * since the balance of every single user increases over time, the total supply
   * does that too.
   * @return the current total supply
   **/
  function totalSupply() public view override(IncentivizedERC20, IERC20) returns (uint256) {
    uint256 currentSupplyScaled = super.totalSupply();

    if (currentSupplyScaled == 0) {
      return 0;
    }

    return currentSupplyScaled.rayMul(_pool.getReserveNormalizedIncome(_underlyingAsset));
  }

  /**
   * @dev Returns the scaled total supply of the variable debt token. Represents sum(debt/index)
   * @return the scaled total supply
   **/
  function scaledTotalSupply() public view virtual override returns (uint256) {
    return super.totalSupply();
  }

  /**
   * @dev Returns the address of the Aave treasury, receiving the fees on this aToken
   **/
  function RESERVE_TREASURY_ADDRESS() public view returns (address) {
    return _treasury;
  }

  /**
   * @dev Returns the address of the underlying asset of this aToken (E.g. WETH for aWETH)
   **/
  function UNDERLYING_ASSET_ADDRESS() public override view returns (address) {
    return _underlyingAsset;
  }

  /**
   * @dev Returns the address of the lending pool where this aToken is used
   **/
  function POOL() public view returns (ILendingPool) {
    return _pool;
  }

  /**
   * @dev For internal usage in the logic of the parent contract IncentivizedERC20
   **/
  function _getIncentivesController() internal view override returns (IAaveIncentivesController) {
    return _incentivesController;
  }

  /**
   * @dev Returns the address of the incentives controller contract
   **/
```

```
function getIncentivesController() external view override returns (IAaveIncentivesController) {
  return _getIncentivesController();
}

/**
 * @dev Transfers the underlying asset to `target`. Used by the LendingPool to transfer
 * assets in borrow(), withdraw() and flashLoan()
 * @param target The recipient of the aTokens
 * @param amount The amount getting transferred
 * @return The amount transferred
 **/
function transferUnderlyingTo(address target, uint256 amount)
  external
  override
  onlyLendingPool
  returns (uint256)
{
  IERC20(_underlyingAsset).safeTransfer(target, amount);
  return amount;
}

/**
 * @dev Invoked to execute actions on the aToken side after a repayment.
 * @param user The user executing the repayment
 * @param amount The amount getting repaid
 **/
function handleRepayment(address user, uint256 amount) external override onlyLendingPool {}

/**
 * @dev implements the permit function as for
 * https://github.com/ethereum/EIPs/blob/8a34d644aacf0f9f8f00815307fd7dd5da07655f/EIPS/eip-2612.md
 * @param owner The owner of the funds
 * @param spender The spender
 * @param value The amount
 * @param deadline The deadline timestamp, type(uint256).max for max deadline
 * @param v Signature param
 * @param s Signature param
 * @param r Signature param
 */
function permit(
  address owner,
  address spender,
  uint256 value,
  uint256 deadline,
  uint8 v,
  bytes32 r,
  bytes32 s
) external {
  require(owner != address(0), 'INVALID_OWNER');
  //solium-disable-next-line
  require(block.timestamp <= deadline, 'INVALID_EXPIRATION');
  uint256 currentValidNonce = _nonces[owner];
  bytes32 digest =
    keccak256(
      abi.encodePacked(
        '\x19\x01',
        DOMAIN_SEPARATOR,
        keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, currentValidNonce, deadline))
      )
    );
  require(owner == ecrecover(digest, v, r, s), 'INVALID_SIGNATURE');
  _nonces[owner] = currentValidNonce.add(1);
  _approve(owner, spender, value);
}

/**
```

```
   * @dev Transfers the aTokens between two users. Validates the transfer
   * (ie checks for valid HF after the transfer) if required
   * @param from The source address
   * @param to The destination address
   * @param amount The amount getting transferred
   * @param validate `true` if the transfer needs to be validated
   **/
  function _transfer(
    address from,
    address to,
    uint256 amount,
    bool validate
  ) internal {
    address underlyingAsset = _underlyingAsset;
    ILendingPool pool = _pool;

    uint256 index = pool.getReserveNormalizedIncome(underlyingAsset);

    uint256 fromBalanceBefore = super.balanceOf(from).rayMul(index);
    uint256 toBalanceBefore = super.balanceOf(to).rayMul(index);

    super._transfer(from, to, amount.rayDiv(index));

    if (validate) {
      pool.finalizeTransfer(underlyingAsset, from, to, amount, fromBalanceBefore, toBalanceBefore);
    }

    emit BalanceTransfer(from, to, amount, index);
  }

  /**
   * @dev Overrides the parent _transfer to force validated transfer() and transferFrom()
   * @param from The source address
   * @param to The destination address
   * @param amount The amount getting transferred
   **/
  function _transfer(
    address from,
    address to,
    uint256 amount
  ) internal override {
    _transfer(from, to, amount, true);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IVariableDebtToken} from '../../interfaces/IVariableDebtToken.sol';
import {WadRayMath} from '../libraries/math/WadRayMath.sol';
import {Errors} from '../libraries/helpers/Errors.sol';
import {DebtTokenBase} from './base/DebtTokenBase.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';

/**
 * @title VariableDebtToken
 * @notice Implements a variable debt token to track the borrowing positions of users
 * at variable rate mode
 * @author Aave
 **/
contract VariableDebtToken is DebtTokenBase, IVariableDebtToken {
  using WadRayMath for uint256;

  uint256 public constant DEBT_TOKEN_REVISION = 0x1;

  ILendingPool internal _pool;
```

```
  address internal _underlyingAsset;
  IAaveIncentivesController internal _incentivesController;

  /**
   * @dev Initializes the debt token.
   * @param pool The address of the lending pool where this aToken will be used
   * @param underlyingAsset The address of the underlying asset of this aToken (E.g. WETH for aWETH)
   * @param incentivesController The smart contract managing potential incentives distribution
   * @param debtTokenDecimals The decimals of the debtToken, same as the underlying asset's
   * @param debtTokenName The name of the token
   * @param debtTokenSymbol The symbol of the token
   */
  function initialize(
    ILendingPool pool,
    address underlyingAsset,
    IAaveIncentivesController incentivesController,
    uint8 debtTokenDecimals,
    string memory debtTokenName,
    string memory debtTokenSymbol,
    bytes calldata params
  ) public override initializer {
    _setName(debtTokenName);
    _setSymbol(debtTokenSymbol);
    _setDecimals(debtTokenDecimals);

    _pool = pool;
    _underlyingAsset = underlyingAsset;
    _incentivesController = incentivesController;

    emit Initialized(
      underlyingAsset,
      address(pool),
      address(incentivesController),
      debtTokenDecimals,
      debtTokenName,
      debtTokenSymbol,
      params
    );
  }

  /**
   * @dev Gets the revision of the stable debt token implementation
   * @return The debt token implementation revision
   **/
  function getRevision() internal pure virtual override returns (uint256) {
    return DEBT_TOKEN_REVISION;
  }

  /**
   * @dev Calculates the accumulated debt balance of the user
   * @return The debt balance of the user
   **/
  function balanceOf(address user) public view virtual override returns (uint256) {
    uint256 scaledBalance = super.balanceOf(user);

    if (scaledBalance == 0) {
      return 0;
    }

    return scaledBalance.rayMul(_pool.getReserveNormalizedVariableDebt(_underlyingAsset));
  }

  /**
   * @dev Mints debt token to the `onBehalfOf` address
   * -  Only callable by the LendingPool
   * @param user The address receiving the borrowed underlying, being the delegatee in case
```

```
   * of credit delegate, or same as `onBehalfOf` otherwise
   * @param onBehalfOf The address receiving the debt tokens
   * @param amount The amount of debt being minted
   * @param index The variable debt index of the reserve
   * @return `true` if the the previous balance of the user is 0
   **/
  function mint(
    address user,
    address onBehalfOf,
    uint256 amount,
    uint256 index
  ) external override onlyLendingPool returns (bool) {
    if (user != onBehalfOf) {
      _decreaseBorrowAllowance(onBehalfOf, user, amount);
    }

    uint256 previousBalance = super.balanceOf(onBehalfOf);
    uint256 amountScaled = amount.rayDiv(index);
    require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);

    _mint(onBehalfOf, amountScaled);

    emit Transfer(address(0), onBehalfOf, amount);
    emit Mint(user, onBehalfOf, amount, index);

    return previousBalance == 0;
  }

  /**
   * @dev Burns user variable debt
   * - Only callable by the LendingPool
   * @param user The user whose debt is getting burned
   * @param amount The amount getting burned
   * @param index The variable debt index of the reserve
   **/
  function burn(
    address user,
    uint256 amount,
    uint256 index
  ) external override onlyLendingPool {
    uint256 amountScaled = amount.rayDiv(index);
    require(amountScaled != 0, Errors.CT_INVALID_BURN_AMOUNT);

    _burn(user, amountScaled);

    emit Transfer(user, address(0), amount);
    emit Burn(user, amount, index);
  }

  /**
   * @dev Returns the principal debt balance of the user from
   * @return The debt balance of the user since the last burn/mint action
   **/
  function scaledBalanceOf(address user) public view virtual override returns (uint256) {
    return super.balanceOf(user);
  }

  /**
   * @dev Returns the total supply of the variable debt token. Represents the total debt accrued by t
   * @return The total supply
   **/
  function totalSupply() public view virtual override returns (uint256) {
    return super.totalSupply().rayMul(_pool.getReserveNormalizedVariableDebt(_underlyingAsset));
  }

  /**
```

```
    * @dev Returns the scaled total supply of the variable debt token. Represents sum(debt/index)
    * @return the scaled total supply
    **/
  function scaledTotalSupply() public view virtual override returns (uint256) {
    return super.totalSupply();
  }

  /**
   * @dev Returns the principal balance of the user and principal total supply.
   * @param user The address of the user
   * @return The principal balance of the user
   * @return The principal total supply
   **/
  function getScaledUserBalanceAndSupply(address user)
    external
    view
    override
    returns (uint256, uint256)
  {
    return (super.balanceOf(user), super.totalSupply());
  }

  /**
   * @dev Returns the address of the underlying asset of this aToken (E.g. WETH for aWETH)
   **/
  function UNDERLYING_ASSET_ADDRESS() public view returns (address) {
    return _underlyingAsset;
  }

  /**
   * @dev Returns the address of the incentives controller contract
   **/
  function getIncentivesController() external view override returns (IAaveIncentivesController) {
    return _getIncentivesController();
  }

  /**
   * @dev Returns the address of the lending pool where this aToken is used
   **/
  function POOL() public view returns (ILendingPool) {
    return _pool;
  }

  function _getIncentivesController() internal view override returns (IAaveIncentivesController) {
    return _incentivesController;
  }

  function _getUnderlyingAssetAddress() internal view override returns (address) {
    return _underlyingAsset;
  }

  function _getLendingPool() internal view override returns (ILendingPool) {
    return _pool;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title Errors library
 * @author Aave
 * @notice Defines the error messages emitted by the different contracts of the Aave protocol
 * @dev Error messages prefix glossary:
 *  - VL = ValidationLogic
 *  - MATH = Math libraries
 *  - CT = Common errors between tokens (AToken, VariableDebtToken and StableDebtToken)
```

```
 *   - AT = AToken
 *   - SDT = StableDebtToken
 *   - VDT = VariableDebtToken
 *   - LP = LendingPool
 *   - LPAPR = LendingPoolAddressesProviderRegistry
 *   - LPC = LendingPoolConfiguration
 *   - RL = ReserveLogic
 *   - LPCM = LendingPoolCollateralManager
 *   - P = Pausable
 */
library Errors {
  //common errors
  string public constant CALLER_NOT_POOL_ADMIN = '33'; // 'The caller must be the pool admin'
  string public constant BORROW_ALLOWANCE_NOT_ENOUGH = '59'; // User borrows on behalf, but allowance

  //contract specific errors
  string public constant VL_INVALID_AMOUNT = '1'; // 'Amount must be greater than 0'
  string public constant VL_NO_ACTIVE_RESERVE = '2'; // 'Action requires an active reserve'
  string public constant VL_RESERVE_FROZEN = '3'; // 'Action cannot be performed because the reserve
  string public constant VL_CURRENT_AVAILABLE_LIQUIDITY_NOT_ENOUGH = '4'; // 'The current liquidity i
  string public constant VL_NOT_ENOUGH_AVAILABLE_USER_BALANCE = '5'; // 'User cannot withdraw more th
  string public constant VL_TRANSFER_NOT_ALLOWED = '6'; // 'Transfer cannot be allowed.'
  string public constant VL_BORROWING_NOT_ENABLED = '7'; // 'Borrowing is not enabled'
  string public constant VL_INVALID_INTEREST_RATE_MODE_SELECTED = '8'; // 'Invalid interest rate mode
  string public constant VL_COLLATERAL_BALANCE_IS_0 = '9'; // 'The collateral balance is 0'
  string public constant VL_HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD = '10'; // 'Health factor
  string public constant VL_COLLATERAL_CANNOT_COVER_NEW_BORROW = '11'; // 'There is not enough collat
  string public constant VL_STABLE_BORROWING_NOT_ENABLED = '12'; // stable borrowing not enabled
  string public constant VL_COLLATERAL_SAME_AS_BORROWING_CURRENCY = '13'; // collateral is (mostly) t
  string public constant VL_AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE = '14'; // 'The requested amount
  string public constant VL_NO_DEBT_OF_SELECTED_TYPE = '15'; // 'for repayment of stable debt, the us
  string public constant VL_NO_EXPLICIT_AMOUNT_TO_REPAY_ON_BEHALF = '16'; // 'To repay on behalf of a
  string public constant VL_NO_STABLE_RATE_LOAN_IN_RESERVE = '17'; // 'User does not have a stable ra
  string public constant VL_NO_VARIABLE_RATE_LOAN_IN_RESERVE = '18'; // 'User does not have a variabl
  string public constant VL_UNDERLYING_BALANCE_NOT_GREATER_THAN_0 = '19'; // 'The underlying balance
  string public constant VL_DEPOSIT_ALREADY_IN_USE = '20'; // 'User deposit is already being used as
  string public constant LP_NOT_ENOUGH_STABLE_BORROW_BALANCE = '21'; // 'User does not have any stabl
  string public constant LP_INTEREST_RATE_REBALANCE_CONDITIONS_NOT_MET = '22'; // 'Interest rate reba
  string public constant LP_LIQUIDATION_CALL_FAILED = '23'; // 'Liquidation call failed'
  string public constant LP_NOT_ENOUGH_LIQUIDITY_TO_BORROW = '24'; // 'There is not enough liquidity
  string public constant LP_REQUESTED_AMOUNT_TOO_SMALL = '25'; // 'The requested amount is too small
  string public constant LP_INCONSISTENT_PROTOCOL_ACTUAL_BALANCE = '26'; // 'The actual balance of th
  string public constant LP_CALLER_NOT_LENDING_POOL_CONFIGURATOR = '27'; // 'The caller of the functi
  string public constant LP_INCONSISTENT_FLASHLOAN_PARAMS = '28';
  string public constant CT_CALLER_MUST_BE_LENDING_POOL = '29'; // 'The caller of this function must
  string public constant CT_CANNOT_GIVE_ALLOWANCE_TO_HIMSELF = '30'; // 'User cannot give allowance t
  string public constant CT_TRANSFER_AMOUNT_NOT_GT_0 = '31'; // 'Transferred amount needs to be great
  string public constant RL_RESERVE_ALREADY_INITIALIZED = '32'; // 'Reserve has already been initiali
  string public constant LPC_RESERVE_LIQUIDITY_NOT_0 = '34'; // 'The liquidity of the reserve needs t
  string public constant LPC_INVALID_ATOKEN_POOL_ADDRESS = '35'; // 'The liquidity of the reserve nee
  string public constant LPC_INVALID_STABLE_DEBT_TOKEN_POOL_ADDRESS = '36'; // 'The liquidity of the
  string public constant LPC_INVALID_VARIABLE_DEBT_TOKEN_POOL_ADDRESS = '37'; // 'The liquidity of th
  string public constant LPC_INVALID_STABLE_DEBT_TOKEN_UNDERLYING_ADDRESS = '38'; // 'The liquidity o
  string public constant LPC_INVALID_VARIABLE_DEBT_TOKEN_UNDERLYING_ADDRESS = '39'; // 'The liquidity
  string public constant LPC_INVALID_ADDRESSES_PROVIDER_ID = '40'; // 'The liquidity of the reserve n
  string public constant LPC_INVALID_CONFIGURATION = '75'; // 'Invalid risk parameters for the reserv
  string public constant LPC_CALLER_NOT_EMERGENCY_ADMIN = '76'; // 'The caller must be the emergency
  string public constant LPAPR_PROVIDER_NOT_REGISTERED = '41'; // 'Provider is not registered'
  string public constant LPCM_HEALTH_FACTOR_NOT_BELOW_THRESHOLD = '42'; // 'Health factor is not belo
  string public constant LPCM_COLLATERAL_CANNOT_BE_LIQUIDATED = '43'; // 'The collateral chosen canno
  string public constant LPCM_SPECIFIED_CURRENCY_NOT_BORROWED_BY_USER = '44'; // 'User did not borrow
  string public constant LPCM_NOT_ENOUGH_LIQUIDITY_TO_LIQUIDATE = '45'; // "There isn't enough liquid
  string public constant LPCM_NO_ERRORS = '46'; // 'No errors'
  string public constant LP_INVALID_FLASHLOAN_MODE = '47'; //Invalid flashloan mode selected
  string public constant MATH_MULTIPLICATION_OVERFLOW = '48';
  string public constant MATH_ADDITION_OVERFLOW = '49';
```

```
  string public constant MATH_DIVISION_BY_ZERO = '50';
  string public constant RL_LIQUIDITY_INDEX_OVERFLOW = '51'; //  Liquidity index overflows uint128
  string public constant RL_VARIABLE_BORROW_INDEX_OVERFLOW = '52'; //  Variable borrow index overflow
  string public constant RL_LIQUIDITY_RATE_OVERFLOW = '53'; //  Liquidity rate overflows uint128
  string public constant RL_VARIABLE_BORROW_RATE_OVERFLOW = '54'; //  Variable borrow rate overflows
  string public constant RL_STABLE_BORROW_RATE_OVERFLOW = '55'; //  Stable borrow rate overflows uint
  string public constant CT_INVALID_MINT_AMOUNT = '56'; //invalid amount to mint
  string public constant LP_FAILED_REPAY_WITH_COLLATERAL = '57';
  string public constant CT_INVALID_BURN_AMOUNT = '58'; //invalid amount to burn
  string public constant LP_FAILED_COLLATERAL_SWAP = '60';
  string public constant LP_INVALID_EQUAL_ASSETS_TO_SWAP = '61';
  string public constant LP_REENTRANCY_NOT_ALLOWED = '62';
  string public constant LP_CALLER_MUST_BE_AN_ATOKEN = '63';
  string public constant LP_IS_PAUSED = '64'; // 'Pool is paused'
  string public constant LP_NO_MORE_RESERVES_ALLOWED = '65';
  string public constant LP_INVALID_FLASH_LOAN_EXECUTOR_RETURN = '66';
  string public constant RC_INVALID_LTV = '67';
  string public constant RC_INVALID_LIQ_THRESHOLD = '68';
  string public constant RC_INVALID_LIQ_BONUS = '69';
  string public constant RC_INVALID_DECIMALS = '70';
  string public constant RC_INVALID_RESERVE_FACTOR = '71';
  string public constant LPAPR_INVALID_ADDRESSES_PROVIDER_ID = '72';
  string public constant VL_INCONSISTENT_FLASHLOAN_PARAMS = '73';
  string public constant LP_INCONSISTENT_PARAMS_LENGTH = '74';
  string public constant UL_INVALID_INDEX = '77';
  string public constant LP_NOT_CONTRACT = '78';
  string public constant SDT_STABLE_DEBT_OVERFLOW = '79';
  string public constant SDT_BURN_EXCEEDS_BALANCE = '80';

  enum CollateralManagerErrors {
    NO_ERROR,
    NO_COLLATERAL_AVAILABLE,
    COLLATERAL_CANNOT_BE_LIQUIDATED,
    CURRRENCY_NOT_BORROWED,
    HEALTH_FACTOR_ABOVE_THRESHOLD,
    NOT_ENOUGH_LIQUIDITY,
    NO_ACTIVE_RESERVE,
    HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD,
    INVALID_EQUAL_ASSETS_TO_SWAP,
    FROZEN_RESERVE
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IERC20} from '../../../dependencies/openzeppelin/contracts/IERC20.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title Helpers library
 * @author Aave
 */
library Helpers {
  /**
   * @dev Fetches the user current stable and variable debt balances
   * @param user The user address
   * @param reserve The reserve data object
   * @return The stable and variable debt balance
   **/
  function getUserCurrentDebt(address user, DataTypes.ReserveData storage reserve)
    internal
    view
    returns (uint256, uint256)
  {
    return (
      IERC20(reserve.stableDebtTokenAddress).balanceOf(user),
```

```
      IERC20(reserve.variableDebtTokenAddress).balanceOf(user)
    );
  }

  function getUserCurrentDebtMemory(address user, DataTypes.ReserveData memory reserve)
    internal
    view
    returns (uint256, uint256)
  {
    return (
      IERC20(reserve.stableDebtTokenAddress).balanceOf(user),
      IERC20(reserve.variableDebtTokenAddress).balanceOf(user)
    );
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @title VersionedInitializable
 *
 * @dev Helper contract to implement initializer functions. To use it, replace
 * the constructor with a function that has the `initializer` modifier.
 * WARNING: Unlike constructors, initializer functions must be manually
 * invoked. This applies both to deploying an Initializable contract, as well
 * as extending an Initializable contract via inheritance.
 * WARNING: When used with inheritance, manual care must be taken to not invoke
 * a parent initializer twice, or ensure that all initializers are idempotent,
 * because this is not dealt with automatically as with constructors.
 *
 * @author Aave, inspired by the OpenZeppelin Initializable contract
 */
abstract contract VersionedInitializable {
  /**
   * @dev Indicates that the contract has been initialized.
   */
  uint256 private lastInitializedRevision = 0;

  /**
   * @dev Indicates that the contract is in the process of being initialized.
   */
  bool private initializing;

  /**
   * @dev Modifier to use in the initializer function of a contract.
   */
  modifier initializer() {
    uint256 revision = getRevision();
    require(
      initializing || isConstructor() || revision > lastInitializedRevision,
      'Contract instance has already been initialized'
    );

    bool isTopLevelCall = !initializing;
    if (isTopLevelCall) {
      initializing = true;
      lastInitializedRevision = revision;
    }

    _;

    if (isTopLevelCall) {
      initializing = false;
    }
  }
```

```
  /**
   * @dev returns the revision number of the contract
   * Needs to be defined in the inherited class as a constant.
   **/
  function getRevision() internal pure virtual returns (uint256);

  /**
   * @dev Returns true if and only if the function is running in the constructor
   **/
  function isConstructor() private view returns (bool) {
    // extcodesize checks the size of the code stored in an address, and
    // address returns the current address. Since the code is still not
    // deployed when running a constructor, any checks on its code size will
    // yield zero, making it an effective way to detect if a contract is
    // under construction or not.
    uint256 cs;
    //solium-disable-next-line
    assembly {
      cs := extcodesize(address())
    }
    return cs == 0;
  }

  // Reserved storage space to allow for layout changes in the future.
  uint256[50] private ______gap;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './BaseImmutableAdminUpgradeabilityProxy.sol';
import '../../../dependencies/openzeppelin/upgradeability/InitializableUpgradeabilityProxy.sol';

/**
 * @title InitializableAdminUpgradeabilityProxy
 * @dev Extends BaseAdminUpgradeabilityProxy with an initializer function
 */
contract InitializableImmutableAdminUpgradeabilityProxy is
  BaseImmutableAdminUpgradeabilityProxy,
  InitializableUpgradeabilityProxy
{
  constructor(address admin) public BaseImmutableAdminUpgradeabilityProxy(admin) {}

  /**
   * @dev Only fall back when the sender is not the admin.
   */
  function _willFallback() internal override(BaseImmutableAdminUpgradeabilityProxy, Proxy) {
    BaseImmutableAdminUpgradeabilityProxy._willFallback();
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import '../../../dependencies/openzeppelin/upgradeability/BaseUpgradeabilityProxy.sol';

/**
 * @title BaseImmutableAdminUpgradeabilityProxy
 * @author Aave, inspired by the OpenZeppelin upgradeability proxy pattern
 * @dev This contract combines an upgradeability proxy with an authorization
 * mechanism for administrative tasks. The admin role is stored in an immutable, which
 * helps saving transactions costs
 * All external functions in this contract must be guarded by the
 * `ifAdmin` modifier. See ethereum/solidity#3864 for a Solidity
 * feature proposal that would enable this to be done automatically.
 */
contract BaseImmutableAdminUpgradeabilityProxy is BaseUpgradeabilityProxy {
  address immutable ADMIN;
```

```
  constructor(address admin) public {
    ADMIN = admin;
  }

  modifier ifAdmin() {
    if (msg.sender == ADMIN) {
      _;
    } else {
      _fallback();
    }
  }

  /**
   * @return The address of the proxy admin.
   */
  function admin() external ifAdmin returns (address) {
    return ADMIN;
  }

  /**
   * @return The address of the implementation.
   */
  function implementation() external ifAdmin returns (address) {
    return _implementation();
  }

  /**
   * @dev Upgrade the backing implementation of the proxy.
   * Only the admin can call this function.
   * @param newImplementation Address of the new implementation.
   */
  function upgradeTo(address newImplementation) external ifAdmin {
    _upgradeTo(newImplementation);
  }

  /**
   * @dev Upgrade the backing implementation of the proxy and call a function
   * on the new implementation.
   * This is useful to initialize the proxied contract.
   * @param newImplementation Address of the new implementation.
   * @param data Data to send as msg.data in the low level call.
   * It should include the signature and the parameters of the function to be called, as described in
   * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding
   */
  function upgradeToAndCall(address newImplementation, bytes calldata data)
    external
    payable
    ifAdmin
  {
    _upgradeTo(newImplementation);
    (bool success, ) = newImplementation.delegatecall(data);
    require(success);
  }

  /**
   * @dev Only fall back when the sender is not the admin.
   */
  function _willFallback() internal virtual override {
    require(msg.sender != ADMIN, 'Cannot call fallback function from the proxy admin');
    super._willFallback();
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
```

```
import {Errors} from '../helpers/Errors.sol';

/**
 * @title PercentageMath library
 * @author Aave
 * @notice Provides functions to perform percentage calculations
 * @dev Percentages are defined by default with 2 decimals of precision (100.00). The precision is in
 * @dev Operations are rounded half up
 **/

library PercentageMath {
  uint256 constant PERCENTAGE_FACTOR = 1e4; //percentage plus two decimals
  uint256 constant HALF_PERCENT = PERCENTAGE_FACTOR / 2;

  /**
   * @dev Executes a percentage multiplication
   * @param value The value of which the percentage needs to be calculated
   * @param percentage The percentage of the value to be calculated
   * @return The percentage of value
   **/
  function percentMul(uint256 value, uint256 percentage) internal pure returns (uint256) {
    if (value == 0 || percentage == 0) {
      return 0;
    }

    require(
      value <= (type(uint256).max - HALF_PERCENT) / percentage,
      Errors.MATH_MULTIPLICATION_OVERFLOW
    );

    return (value * percentage + HALF_PERCENT) / PERCENTAGE_FACTOR;
  }

  /**
   * @dev Executes a percentage division
   * @param value The value of which the percentage needs to be calculated
   * @param percentage The percentage of the value to be calculated
   * @return The value divided the percentage
   **/
  function percentDiv(uint256 value, uint256 percentage) internal pure returns (uint256) {
    require(percentage != 0, Errors.MATH_DIVISION_BY_ZERO);
    uint256 halfPercentage = percentage / 2;

    require(
      value <= (type(uint256).max - halfPercentage) / PERCENTAGE_FACTOR,
      Errors.MATH_MULTIPLICATION_OVERFLOW
    );

    return (value * PERCENTAGE_FACTOR + halfPercentage) / percentage;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {Errors} from '../helpers/Errors.sol';

/**
 * @title WadRayMath library
 * @author Aave
 * @dev Provides mul and div function for wads (decimal numbers with 18 digits precision) and rays (d
 **/

library WadRayMath {
  uint256 internal constant WAD = 1e18;
  uint256 internal constant halfWAD = WAD / 2;
```

```
uint256 internal constant RAY = 1e27;
uint256 internal constant halfRAY = RAY / 2;

uint256 internal constant WAD_RAY_RATIO = 1e9;

/**
 * @return One ray, 1e27
 **/
function ray() internal pure returns (uint256) {
  return RAY;
}

/**
 * @return One wad, 1e18
 **/

function wad() internal pure returns (uint256) {
  return WAD;
}

/**
 * @return Half ray, 1e27/2
 **/
function halfRay() internal pure returns (uint256) {
  return halfRAY;
}

/**
 * @return Half ray, 1e18/2
 **/
function halfWad() internal pure returns (uint256) {
  return halfWAD;
}

/**
 * @dev Multiplies two wad, rounding half up to the nearest wad
 * @param a Wad
 * @param b Wad
 * @return The result of a*b, in wad
 **/
function wadMul(uint256 a, uint256 b) internal pure returns (uint256) {
  if (a == 0 || b == 0) {
    return 0;
  }

  require(a <= (type(uint256).max - halfWAD) / b, Errors.MATH_MULTIPLICATION_OVERFLOW);

  return (a * b + halfWAD) / WAD;
}

/**
 * @dev Divides two wad, rounding half up to the nearest wad
 * @param a Wad
 * @param b Wad
 * @return The result of a/b, in wad
 **/
function wadDiv(uint256 a, uint256 b) internal pure returns (uint256) {
  require(b != 0, Errors.MATH_DIVISION_BY_ZERO);
  uint256 halfB = b / 2;

  require(a <= (type(uint256).max - halfB) / WAD, Errors.MATH_MULTIPLICATION_OVERFLOW);

  return (a * WAD + halfB) / b;
}

/**
```

```
   * @dev Multiplies two ray, rounding half up to the nearest ray
   * @param a Ray
   * @param b Ray
   * @return The result of a*b, in ray
   **/
  function rayMul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0 || b == 0) {
      return 0;
    }

    require(a <= (type(uint256).max - halfRAY) / b, Errors.MATH_MULTIPLICATION_OVERFLOW);

    return (a * b + halfRAY) / RAY;
  }

  /**
   * @dev Divides two ray, rounding half up to the nearest ray
   * @param a Ray
   * @param b Ray
   * @return The result of a/b, in ray
   **/
  function rayDiv(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b != 0, Errors.MATH_DIVISION_BY_ZERO);
    uint256 halfB = b / 2;

    require(a <= (type(uint256).max - halfB) / RAY, Errors.MATH_MULTIPLICATION_OVERFLOW);

    return (a * RAY + halfB) / b;
  }

  /**
   * @dev Casts ray down to wad
   * @param a Ray
   * @return a casted to wad, rounded half up to the nearest wad
   **/
  function rayToWad(uint256 a) internal pure returns (uint256) {
    uint256 halfRatio = WAD_RAY_RATIO / 2;
    uint256 result = halfRatio + a;
    require(result >= halfRatio, Errors.MATH_ADDITION_OVERFLOW);

    return result / WAD_RAY_RATIO;
  }

  /**
   * @dev Converts wad up to ray
   * @param a Wad
   * @return a converted in ray
   **/
  function wadToRay(uint256 a) internal pure returns (uint256) {
    uint256 result = a * WAD_RAY_RATIO;
    require(result / WAD_RAY_RATIO == a, Errors.MATH_MULTIPLICATION_OVERFLOW);
    return result;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {SafeMath} from '../../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {WadRayMath} from './WadRayMath.sol';

library MathUtils {
  using SafeMath for uint256;
  using WadRayMath for uint256;

  /// @dev Ignoring leap years
  uint256 internal constant SECONDS_PER_YEAR = 365 days;
```

```
/**
 * @dev Function to calculate the interest accumulated using a linear interest rate formula
 * @param rate The interest rate, in ray
 * @param lastUpdateTimestamp The timestamp of the last update of the interest
 * @return The interest rate linearly accumulated during the timeDelta, in ray
 **/

function calculateLinearInterest(uint256 rate, uint40 lastUpdateTimestamp)
  internal
  view
  returns (uint256)
{
  //solium-disable-next-line
  uint256 timeDifference = block.timestamp.sub(uint256(lastUpdateTimestamp));

  return (rate.mul(timeDifference) / SECONDS_PER_YEAR).add(WadRayMath.ray());
}

/**
 * @dev Function to calculate the interest using a compounded interest rate formula
 * To avoid expensive exponentiation, the calculation is performed using a binomial approximation:
 *
 *  (1+x)^n = 1+n*x+[n/2*(n-1)]*x^2+[n/6*(n-1)*(n-2)*x^3...
 *
 * The approximation slightly underpays liquidity providers and undercharges borrowers, with the ad
 * The whitepaper contains reference to the approximation and a table showing the margin of error p
 *
 * @param rate The interest rate, in ray
 * @param lastUpdateTimestamp The timestamp of the last update of the interest
 * @return The interest rate compounded during the timeDelta, in ray
 **/
function calculateCompoundedInterest(
  uint256 rate,
  uint40 lastUpdateTimestamp,
  uint256 currentTimestamp
) internal pure returns (uint256) {
  //solium-disable-next-line
  uint256 exp = currentTimestamp.sub(uint256(lastUpdateTimestamp));

  if (exp == 0) {
    return WadRayMath.ray();
  }

  uint256 expMinusOne = exp - 1;

  uint256 expMinusTwo = exp > 2 ? exp - 2 : 0;

  uint256 ratePerSecond = rate / SECONDS_PER_YEAR;

  uint256 basePowerTwo = ratePerSecond.rayMul(ratePerSecond);
  uint256 basePowerThree = basePowerTwo.rayMul(ratePerSecond);

  uint256 secondTerm = exp.mul(expMinusOne).mul(basePowerTwo) / 2;
  uint256 thirdTerm = exp.mul(expMinusOne).mul(expMinusTwo).mul(basePowerThree) / 6;

  return WadRayMath.ray().add(ratePerSecond.mul(exp)).add(secondTerm).add(thirdTerm);
}

/**
 * @dev Calculates the compounded interest between the timestamp of the last update and the current
 * @param rate The interest rate (in ray)
 * @param lastUpdateTimestamp The timestamp from which the interest accumulation needs to be calcul
 **/
function calculateCompoundedInterest(uint256 rate, uint40 lastUpdateTimestamp)
  internal
```

```
    view
    returns (uint256)
  {
    return calculateCompoundedInterest(rate, lastUpdateTimestamp, block.timestamp);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {SafeMath} from '../../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../../../dependencies/openzeppelin/contracts/IERC20.sol';
import {ReserveLogic} from './ReserveLogic.sol';
import {ReserveConfiguration} from '../configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../configuration/UserConfiguration.sol';
import {WadRayMath} from '../math/WadRayMath.sol';
import {PercentageMath} from '../math/PercentageMath.sol';
import {IPriceOracleGetter} from '../../../interfaces/IPriceOracleGetter.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title GenericLogic library
 * @author Aave
 * @title Implements protocol-level logic to calculate and validate the state of a user
 */
library GenericLogic {
  using ReserveLogic for DataTypes.ReserveData;
  using SafeMath for uint256;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  uint256 public constant HEALTH_FACTOR_LIQUIDATION_THRESHOLD = 1 ether;

  struct balanceDecreaseAllowedLocalVars {
    uint256 decimals;
    uint256 liquidationThreshold;
    uint256 totalCollateralInETH;
    uint256 totalDebtInETH;
    uint256 avgLiquidationThreshold;
    uint256 amountToDecreaseInETH;
    uint256 collateralBalanceAfterDecrease;
    uint256 liquidationThresholdAfterDecrease;
    uint256 healthFactorAfterDecrease;
    bool reserveUsageAsCollateralEnabled;
  }

  /**
   * @dev Checks if a specific balance decrease is allowed
   * (i.e. doesn't bring the user borrow position health factor under HEALTH_FACTOR_LIQUIDATION_THRES
   * @param asset The address of the underlying asset of the reserve
   * @param user The address of the user
   * @param amount The amount to decrease
   * @param reservesData The data of all the reserves
   * @param userConfig The user configuration
   * @param reserves The list of all the active reserves
   * @param oracle The address of the oracle contract
   * @return true if the decrease of the balance is allowed
   **/
  function balanceDecreaseAllowed(
    address asset,
    address user,
    uint256 amount,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    DataTypes.UserConfigurationMap calldata userConfig,
```

```
    mapping(uint256 => address) storage reserves,
    uint256 reservesCount,
    address oracle
  ) external view returns (bool) {
    if (!userConfig.isBorrowingAny() || !userConfig.isUsingAsCollateral(reservesData[asset].id)) {
      return true;
    }

    balanceDecreaseAllowedLocalVars memory vars;

    (, vars.liquidationThreshold, , vars.decimals, ) = reservesData[asset]
      .configuration
      .getParams();

    if (vars.liquidationThreshold == 0) {
      return true;
    }

    (
      vars.totalCollateralInETH,
      vars.totalDebtInETH,
      ,
      vars.avgLiquidationThreshold,

    ) = calculateUserAccountData(user, reservesData, userConfig, reserves, reservesCount, oracle);

    if (vars.totalDebtInETH == 0) {
      return true;
    }

    vars.amountToDecreaseInETH = IPriceOracleGetter(oracle).getAssetPrice(asset).mul(amount).div(
      10**vars.decimals
    );

    vars.collateralBalanceAfterDecrease = vars.totalCollateralInETH.sub(vars.amountToDecreaseInETH);

    //if there is a borrow, there can't be 0 collateral
    if (vars.collateralBalanceAfterDecrease == 0) {
      return false;
    }

    vars.liquidationThresholdAfterDecrease = vars
      .totalCollateralInETH
      .mul(vars.avgLiquidationThreshold)
      .sub(vars.amountToDecreaseInETH.mul(vars.liquidationThreshold))
      .div(vars.collateralBalanceAfterDecrease);

    uint256 healthFactorAfterDecrease =
      calculateHealthFactorFromBalances(
        vars.collateralBalanceAfterDecrease,
        vars.totalDebtInETH,
        vars.liquidationThresholdAfterDecrease
      );

    return healthFactorAfterDecrease >= GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD;
  }

  struct CalculateUserAccountDataVars {
    uint256 reserveUnitPrice;
    uint256 tokenUnit;
    uint256 compoundedLiquidityBalance;
    uint256 compoundedBorrowBalance;
    uint256 decimals;
    uint256 ltv;
    uint256 liquidationThreshold;
    uint256 i;
```

```
    uint256 healthFactor;
    uint256 totalCollateralInETH;
    uint256 totalDebtInETH;
    uint256 avgLtv;
    uint256 avgLiquidationThreshold;
    uint256 reservesLength;
    bool healthFactorBelowThreshold;
    address currentReserveAddress;
    bool usageAsCollateralEnabled;
    bool userUsesReserveAsCollateral;
  }

  /**
   * @dev Calculates the user data across the reserves.
   * this includes the total liquidity/collateral/borrow balances in ETH,
   * the average Loan To Value, the average Liquidation Ratio, and the Health factor.
   * @param user The address of the user
   * @param reservesData Data of all the reserves
   * @param userConfig The configuration of the user
   * @param reserves The list of the available reserves
   * @param oracle The price oracle address
   * @return The total collateral and total debt of the user in ETH, the avg ltv, liquidation thresho
   **/
  function calculateUserAccountData(
    address user,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    DataTypes.UserConfigurationMap memory userConfig,
    mapping(uint256 => address) storage reserves,
    uint256 reservesCount,
    address oracle
  )
    internal
    view
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      uint256
    )
  {
    CalculateUserAccountDataVars memory vars;

    if (userConfig.isEmpty()) {
      return (0, 0, 0, 0, uint256(-1));
    }
    for (vars.i = 0; vars.i < reservesCount; vars.i++) {
      if (!userConfig.isUsingAsCollateralOrBorrowing(vars.i)) {
        continue;
      }

      vars.currentReserveAddress = reserves[vars.i];
      DataTypes.ReserveData storage currentReserve = reservesData[vars.currentReserveAddress];

      (vars.ltv, vars.liquidationThreshold, , vars.decimals, ) = currentReserve
        .configuration
        .getParams();

      vars.tokenUnit = 10**vars.decimals;
      vars.reserveUnitPrice = IPriceOracleGetter(oracle).getAssetPrice(vars.currentReserveAddress);

      if (vars.liquidationThreshold != 0 && userConfig.isUsingAsCollateral(vars.i)) {
        vars.compoundedLiquidityBalance = IERC20(currentReserve.aTokenAddress).balanceOf(user);

        uint256 liquidityBalanceETH =
          vars.reserveUnitPrice.mul(vars.compoundedLiquidityBalance).div(vars.tokenUnit);
```

```
      vars.totalCollateralInETH = vars.totalCollateralInETH.add(liquidityBalanceETH);

      vars.avgLtv = vars.avgLtv.add(liquidityBalanceETH.mul(vars.ltv));
      vars.avgLiquidationThreshold = vars.avgLiquidationThreshold.add(
        liquidityBalanceETH.mul(vars.liquidationThreshold)
      );
    }

    if (userConfig.isBorrowing(vars.i)) {
      vars.compoundedBorrowBalance = IERC20(currentReserve.stableDebtTokenAddress).balanceOf(
        user
      );
      vars.compoundedBorrowBalance = vars.compoundedBorrowBalance.add(
        IERC20(currentReserve.variableDebtTokenAddress).balanceOf(user)
      );

      vars.totalDebtInETH = vars.totalDebtInETH.add(
        vars.reserveUnitPrice.mul(vars.compoundedBorrowBalance).div(vars.tokenUnit)
      );
    }
  }

  vars.avgLtv = vars.totalCollateralInETH > 0 ? vars.avgLtv.div(vars.totalCollateralInETH) : 0;
  vars.avgLiquidationThreshold = vars.totalCollateralInETH > 0
    ? vars.avgLiquidationThreshold.div(vars.totalCollateralInETH)
    : 0;

  vars.healthFactor = calculateHealthFactorFromBalances(
    vars.totalCollateralInETH,
    vars.totalDebtInETH,
    vars.avgLiquidationThreshold
  );
  return (
    vars.totalCollateralInETH,
    vars.totalDebtInETH,
    vars.avgLtv,
    vars.avgLiquidationThreshold,
    vars.healthFactor
  );
}

/**
 * @dev Calculates the health factor from the corresponding balances
 * @param totalCollateralInETH The total collateral in ETH
 * @param totalDebtInETH The total debt in ETH
 * @param liquidationThreshold The avg liquidation threshold
 * @return The health factor calculated from the balances provided
 **/
function calculateHealthFactorFromBalances(
  uint256 totalCollateralInETH,
  uint256 totalDebtInETH,
  uint256 liquidationThreshold
) internal pure returns (uint256) {
  if (totalDebtInETH == 0) return uint256(-1);

  return (totalCollateralInETH.percentMul(liquidationThreshold)).wadDiv(totalDebtInETH);
}

/**
 * @dev Calculates the equivalent amount in ETH that an user can borrow, depending on the available
 * average Loan To Value
 * @param totalCollateralInETH The total collateral in ETH
 * @param totalDebtInETH The total borrow balance
 * @param ltv The average loan to value
 * @return the amount available to borrow in ETH for the user
```

```
   **/

  function calculateAvailableBorrowsETH(
    uint256 totalCollateralInETH,
    uint256 totalDebtInETH,
    uint256 ltv
  ) internal pure returns (uint256) {
    uint256 availableBorrowsETH = totalCollateralInETH.percentMul(ltv);

    if (availableBorrowsETH < totalDebtInETH) {
      return 0;
    }

    availableBorrowsETH = availableBorrowsETH.sub(totalDebtInETH);
    return availableBorrowsETH;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {SafeMath} from '../../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../../../dependencies/openzeppelin/contracts/IERC20.sol';
import {ReserveLogic} from './ReserveLogic.sol';
import {GenericLogic} from './GenericLogic.sol';
import {WadRayMath} from '../math/WadRayMath.sol';
import {PercentageMath} from '../math/PercentageMath.sol';
import {SafeERC20} from '../../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {ReserveConfiguration} from '../configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../configuration/UserConfiguration.sol';
import {Errors} from '../helpers/Errors.sol';
import {Helpers} from '../helpers/Helpers.sol';
import {IReserveInterestRateStrategy} from '../../../interfaces/IReserveInterestRateStrategy.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title ReserveLogic library
 * @author Aave
 * @notice Implements functions to validate the different actions of the protocol
 */
library ValidationLogic {
  using ReserveLogic for DataTypes.ReserveData;
  using SafeMath for uint256;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using SafeERC20 for IERC20;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  uint256 public constant REBALANCE_UP_LIQUIDITY_RATE_THRESHOLD = 4000;
  uint256 public constant REBALANCE_UP_USAGE_RATIO_THRESHOLD = 0.95 * 1e27; //usage ratio of 95%

  /**
   * @dev Validates a deposit action
   * @param reserve The reserve object on which the user is depositing
   * @param amount The amount to be deposited
   */
  function validateDeposit(DataTypes.ReserveData storage reserve, uint256 amount) external view {
    (bool isActive, bool isFrozen, , ) = reserve.configuration.getFlags();

    require(amount != 0, Errors.VL_INVALID_AMOUNT);
    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
    require(!isFrozen, Errors.VL_RESERVE_FROZEN);
  }

  /**
```

```
   * @dev Validates a withdraw action
   * @param reserveAddress The address of the reserve
   * @param amount The amount to be withdrawn
   * @param userBalance The balance of the user
   * @param reservesData The reserves state
   * @param userConfig The user configuration
   * @param reserves The addresses of the reserves
   * @param reservesCount The number of reserves
   * @param oracle The price oracle
   */
  function validateWithdraw(
    address reserveAddress,
    uint256 amount,
    uint256 userBalance,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    DataTypes.UserConfigurationMap storage userConfig,
    mapping(uint256 => address) storage reserves,
    uint256 reservesCount,
    address oracle
  ) external view {
    require(amount != 0, Errors.VL_INVALID_AMOUNT);
    require(amount <= userBalance, Errors.VL_NOT_ENOUGH_AVAILABLE_USER_BALANCE);

    (bool isActive, , , ) = reservesData[reserveAddress].configuration.getFlags();
    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);

    require(
      GenericLogic.balanceDecreaseAllowed(
        reserveAddress,
        msg.sender,
        amount,
        reservesData,
        userConfig,
        reserves,
        reservesCount,
        oracle
      ),
      Errors.VL_TRANSFER_NOT_ALLOWED
    );
  }

  struct ValidateBorrowLocalVars {
    uint256 currentLtv;
    uint256 currentLiquidationThreshold;
    uint256 amountOfCollateralNeededETH;
    uint256 userCollateralBalanceETH;
    uint256 userBorrowBalanceETH;
    uint256 availableLiquidity;
    uint256 healthFactor;
    bool isActive;
    bool isFrozen;
    bool borrowingEnabled;
    bool stableRateBorrowingEnabled;
  }

  /**
   * @dev Validates a borrow action
   * @param asset The address of the asset to borrow
   * @param reserve The reserve state from which the user is borrowing
   * @param userAddress The address of the user
   * @param amount The amount to be borrowed
   * @param amountInETH The amount to be borrowed, in ETH
   * @param interestRateMode The interest rate mode at which the user is borrowing
   * @param maxStableLoanPercent The max amount of the liquidity that can be borrowed at stable rate,
   * @param reservesData The state of all the reserves
   * @param userConfig The state of the user for the specific reserve
```

```
   * @param reserves The addresses of all the active reserves
   * @param oracle The price oracle
   */

  function validateBorrow(
    address asset,
    DataTypes.ReserveData storage reserve,
    address userAddress,
    uint256 amount,
    uint256 amountInETH,
    uint256 interestRateMode,
    uint256 maxStableLoanPercent,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    DataTypes.UserConfigurationMap storage userConfig,
    mapping(uint256 => address) storage reserves,
    uint256 reservesCount,
    address oracle
  ) external view {
    ValidateBorrowLocalVars memory vars;

    (vars.isActive, vars.isFrozen, vars.borrowingEnabled, vars.stableRateBorrowingEnabled) = reserve
      .configuration
      .getFlags();

    require(vars.isActive, Errors.VL_NO_ACTIVE_RESERVE);
    require(!vars.isFrozen, Errors.VL_RESERVE_FROZEN);
    require(amount != 0, Errors.VL_INVALID_AMOUNT);

    require(vars.borrowingEnabled, Errors.VL_BORROWING_NOT_ENABLED);

    //validate interest rate mode
    require(
      uint256(DataTypes.InterestRateMode.VARIABLE) == interestRateMode ||
        uint256(DataTypes.InterestRateMode.STABLE) == interestRateMode,
      Errors.VL_INVALID_INTEREST_RATE_MODE_SELECTED
    );

    (
      vars.userCollateralBalanceETH,
      vars.userBorrowBalanceETH,
      vars.currentLtv,
      vars.currentLiquidationThreshold,
      vars.healthFactor
    ) = GenericLogic.calculateUserAccountData(
      userAddress,
      reservesData,
      userConfig,
      reserves,
      reservesCount,
      oracle
    );

    require(vars.userCollateralBalanceETH > 0, Errors.VL_COLLATERAL_BALANCE_IS_0);

    require(
      vars.healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.VL_HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
    );

    //add the current already borrowed amount to the amount requested to calculate the total collater
    vars.amountOfCollateralNeededETH = vars.userBorrowBalanceETH.add(amountInETH).percentDiv(
      vars.currentLtv
    ); //LTV is calculated in percentage

    require(
      vars.amountOfCollateralNeededETH <= vars.userCollateralBalanceETH,
```

```
      Errors.VL_COLLATERAL_CANNOT_COVER_NEW_BORROW
    );

    /**
     * Following conditions need to be met if the user is borrowing at a stable rate:
     * 1. Reserve must be enabled for stable rate borrowing
     * 2. Users cannot borrow from the reserve if their collateral is (mostly) the same currency
     *    they are borrowing, to prevent abuses.
     * 3. Users will be able to borrow only a portion of the total available liquidity
     **/

    if (interestRateMode == uint256(DataTypes.InterestRateMode.STABLE)) {
      //check if the borrow mode is stable and if stable rate borrowing is enabled on this reserve

      require(vars.stableRateBorrowingEnabled, Errors.VL_STABLE_BORROWING_NOT_ENABLED);

      require(
        !userConfig.isUsingAsCollateral(reserve.id) ||
          reserve.configuration.getLtv() == 0 ||
          amount > IERC20(reserve.aTokenAddress).balanceOf(userAddress),
        Errors.VL_COLLATERAL_SAME_AS_BORROWING_CURRENCY
      );

      vars.availableLiquidity = IERC20(asset).balanceOf(reserve.aTokenAddress);

      //calculate the max available loan size in stable rate mode as a percentage of the
      //available liquidity
      uint256 maxLoanSizeStable = vars.availableLiquidity.percentMul(maxStableLoanPercent);

      require(amount <= maxLoanSizeStable, Errors.VL_AMOUNT_BIGGER_THAN_MAX_LOAN_SIZE_STABLE);
    }
  }

  /**
   * @dev Validates a repay action
   * @param reserve The reserve state from which the user is repaying
   * @param amountSent The amount sent for the repayment. Can be an actual value or uint(-1)
   * @param onBehalfOf The address of the user msg.sender is repaying for
   * @param stableDebt The borrow balance of the user
   * @param variableDebt The borrow balance of the user
   */
  function validateRepay(
    DataTypes.ReserveData storage reserve,
    uint256 amountSent,
    DataTypes.InterestRateMode rateMode,
    address onBehalfOf,
    uint256 stableDebt,
    uint256 variableDebt
  ) external view {
    bool isActive = reserve.configuration.getActive();

    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);

    require(amountSent > 0, Errors.VL_INVALID_AMOUNT);

    require(
      (stableDebt > 0 &&
        DataTypes.InterestRateMode(rateMode) == DataTypes.InterestRateMode.STABLE) ||
        (variableDebt > 0 &&
          DataTypes.InterestRateMode(rateMode) == DataTypes.InterestRateMode.VARIABLE),
      Errors.VL_NO_DEBT_OF_SELECTED_TYPE
    );

    require(
      amountSent != uint256(-1) || msg.sender == onBehalfOf,
      Errors.VL_NO_EXPLICIT_AMOUNT_TO_REPAY_ON_BEHALF
```

```
    );
  }

  /**
   * @dev Validates a swap of borrow rate mode.
   * @param reserve The reserve state on which the user is swapping the rate
   * @param userConfig The user reserves configuration
   * @param stableDebt The stable debt of the user
   * @param variableDebt The variable debt of the user
   * @param currentRateMode The rate mode of the borrow
   */
  function validateSwapRateMode(
    DataTypes.ReserveData storage reserve,
    DataTypes.UserConfigurationMap storage userConfig,
    uint256 stableDebt,
    uint256 variableDebt,
    DataTypes.InterestRateMode currentRateMode
  ) external view {
    (bool isActive, bool isFrozen, , bool stableRateEnabled) = reserve.configuration.getFlags();

    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
    require(!isFrozen, Errors.VL_RESERVE_FROZEN);

    if (currentRateMode == DataTypes.InterestRateMode.STABLE) {
      require(stableDebt > 0, Errors.VL_NO_STABLE_RATE_LOAN_IN_RESERVE);
    } else if (currentRateMode == DataTypes.InterestRateMode.VARIABLE) {
      require(variableDebt > 0, Errors.VL_NO_VARIABLE_RATE_LOAN_IN_RESERVE);
      /**
       * user wants to swap to stable, before swapping we need to ensure that
       * 1. stable borrow rate is enabled on the reserve
       * 2. user is not trying to abuse the reserve by depositing
       * more collateral than he is borrowing, artificially lowering
       * the interest rate, borrowing at variable, and switching to stable
       **/
      require(stableRateEnabled, Errors.VL_STABLE_BORROWING_NOT_ENABLED);

      require(
        !userConfig.isUsingAsCollateral(reserve.id) ||
          reserve.configuration.getLtv() == 0 ||
          stableDebt.add(variableDebt) > IERC20(reserve.aTokenAddress).balanceOf(msg.sender),
        Errors.VL_COLLATERAL_SAME_AS_BORROWING_CURRENCY
      );
    } else {
      revert(Errors.VL_INVALID_INTEREST_RATE_MODE_SELECTED);
    }
  }

  /**
   * @dev Validates a stable borrow rate rebalance action
   * @param reserve The reserve state on which the user is getting rebalanced
   * @param reserveAddress The address of the reserve
   * @param stableDebtToken The stable debt token instance
   * @param variableDebtToken The variable debt token instance
   * @param aTokenAddress The address of the aToken contract
   */
  function validateRebalanceStableBorrowRate(
    DataTypes.ReserveData storage reserve,
    address reserveAddress,
    IERC20 stableDebtToken,
    IERC20 variableDebtToken,
    address aTokenAddress
  ) external view {
    (bool isActive, , , ) = reserve.configuration.getFlags();

    require(isActive, Errors.VL_NO_ACTIVE_RESERVE);
```

```
    //if the usage ratio is below 95%, no rebalances are needed
    uint256 totalDebt =
      stableDebtToken.totalSupply().add(variableDebtToken.totalSupply()).wadToRay();
    uint256 availableLiquidity = IERC20(reserveAddress).balanceOf(aTokenAddress).wadToRay();
    uint256 usageRatio = totalDebt == 0 ? 0 : totalDebt.rayDiv(availableLiquidity.add(totalDebt));

    //if the liquidity rate is below REBALANCE_UP_THRESHOLD of the max variable APR at 95% usage,
    //then we allow rebalancing of the stable rate positions.

    uint256 currentLiquidityRate = reserve.currentLiquidityRate;
    uint256 maxVariableBorrowRate =
      IReserveInterestRateStrategy(reserve.interestRateStrategyAddress).getMaxVariableBorrowRate();

    require(
      usageRatio >= REBALANCE_UP_USAGE_RATIO_THRESHOLD &&
        currentLiquidityRate <=
        maxVariableBorrowRate.percentMul(REBALANCE_UP_LIQUIDITY_RATE_THRESHOLD),
      Errors.LP_INTEREST_RATE_REBALANCE_CONDITIONS_NOT_MET
    );
  }

  /**
   * @dev Validates the action of setting an asset as collateral
   * @param reserve The state of the reserve that the user is enabling or disabling as collateral
   * @param reserveAddress The address of the reserve
   * @param reservesData The data of all the reserves
   * @param userConfig The state of the user for the specific reserve
   * @param reserves The addresses of all the active reserves
   * @param oracle The price oracle
   */
  function validateSetUseReserveAsCollateral(
    DataTypes.ReserveData storage reserve,
    address reserveAddress,
    bool useAsCollateral,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    DataTypes.UserConfigurationMap storage userConfig,
    mapping(uint256 => address) storage reserves,
    uint256 reservesCount,
    address oracle
  ) external view {
    uint256 underlyingBalance = IERC20(reserve.aTokenAddress).balanceOf(msg.sender);

    require(underlyingBalance > 0, Errors.VL_UNDERLYING_BALANCE_NOT_GREATER_THAN_0);

    require(
      useAsCollateral ||
        GenericLogic.balanceDecreaseAllowed(
          reserveAddress,
          msg.sender,
          underlyingBalance,
          reservesData,
          userConfig,
          reserves,
          reservesCount,
          oracle
        ),
      Errors.VL_DEPOSIT_ALREADY_IN_USE
    );
  }

  /**
   * @dev Validates a flashloan action
   * @param assets The assets being flashborrowed
   * @param amounts The amounts for each asset being borrowed
   **/
  function validateFlashloan(address[] memory assets, uint256[] memory amounts) internal pure {
```

```
    require(assets.length == amounts.length, Errors.VL_INCONSISTENT_FLASHLOAN_PARAMS);
  }

  /**
   * @dev Validates the liquidation action
   * @param collateralReserve The reserve data of the collateral
   * @param principalReserve The reserve data of the principal
   * @param userConfig The user configuration
   * @param userHealthFactor The user's health factor
   * @param userStableDebt Total stable debt balance of the user
   * @param userVariableDebt Total variable debt balance of the user
   **/
  function validateLiquidationCall(
    DataTypes.ReserveData storage collateralReserve,
    DataTypes.ReserveData storage principalReserve,
    DataTypes.UserConfigurationMap storage userConfig,
    uint256 userHealthFactor,
    uint256 userStableDebt,
    uint256 userVariableDebt
  ) internal view returns (uint256, string memory) {
    if (
      !collateralReserve.configuration.getActive() || !principalReserve.configuration.getActive()
    ) {
      return (
        uint256(Errors.CollateralManagerErrors.NO_ACTIVE_RESERVE),
        Errors.VL_NO_ACTIVE_RESERVE
      );
    }

    if (userHealthFactor >= GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD) {
      return (
        uint256(Errors.CollateralManagerErrors.HEALTH_FACTOR_ABOVE_THRESHOLD),
        Errors.LPCM_HEALTH_FACTOR_NOT_BELOW_THRESHOLD
      );
    }

    bool isCollateralEnabled =
      collateralReserve.configuration.getLiquidationThreshold() > 0 &&
        userConfig.isUsingAsCollateral(collateralReserve.id);

    //if collateral isn't enabled as collateral by user, it cannot be liquidated
    if (!isCollateralEnabled) {
      return (
        uint256(Errors.CollateralManagerErrors.COLLATERAL_CANNOT_BE_LIQUIDATED),
        Errors.LPCM_COLLATERAL_CANNOT_BE_LIQUIDATED
      );
    }

    if (userStableDebt == 0 && userVariableDebt == 0) {
      return (
        uint256(Errors.CollateralManagerErrors.CURRRENCY_NOT_BORROWED),
        Errors.LPCM_SPECIFIED_CURRENCY_NOT_BORROWED_BY_USER
      );
    }

    return (uint256(Errors.CollateralManagerErrors.NO_ERROR), Errors.LPCM_NO_ERRORS);
  }

  /**
   * @dev Validates an aToken transfer
   * @param from The user from which the aTokens are being transferred
   * @param reservesData The state of all the reserves
   * @param userConfig The state of the user for the specific reserve
   * @param reserves The addresses of all the active reserves
   * @param oracle The price oracle
   */
```

```
  function validateTransfer(
    address from,
    mapping(address => DataTypes.ReserveData) storage reservesData,
    DataTypes.UserConfigurationMap storage userConfig,
    mapping(uint256 => address) storage reserves,
    uint256 reservesCount,
    address oracle
  ) internal view {
    (, , , , uint256 healthFactor) =
      GenericLogic.calculateUserAccountData(
        from,
        reservesData,
        userConfig,
        reserves,
        reservesCount,
        oracle
      );

    require(
      healthFactor >= GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.VL_TRANSFER_NOT_ALLOWED
    );
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {SafeMath} from '../../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../../../dependencies/openzeppelin/contracts/IERC20.sol';
import {SafeERC20} from '../../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {IAToken} from '../../../interfaces/IAToken.sol';
import {IStableDebtToken} from '../../../interfaces/IStableDebtToken.sol';
import {IVariableDebtToken} from '../../../interfaces/IVariableDebtToken.sol';
import {IReserveInterestRateStrategy} from '../../../interfaces/IReserveInterestRateStrategy.sol';
import {ReserveConfiguration} from '../configuration/ReserveConfiguration.sol';
import {MathUtils} from '../math/MathUtils.sol';
import {WadRayMath} from '../math/WadRayMath.sol';
import {PercentageMath} from '../math/PercentageMath.sol';
import {Errors} from '../helpers/Errors.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title ReserveLogic library
 * @author Aave
 * @notice Implements the logic to update the reserves state
 */
library ReserveLogic {
  using SafeMath for uint256;
  using WadRayMath for uint256;
  using PercentageMath for uint256;
  using SafeERC20 for IERC20;

  /**
   * @dev Emitted when the state of a reserve is updated
   * @param asset The address of the underlying asset of the reserve
   * @param liquidityRate The new liquidity rate
   * @param stableBorrowRate The new stable borrow rate
   * @param variableBorrowRate The new variable borrow rate
   * @param liquidityIndex The new liquidity index
   * @param variableBorrowIndex The new variable borrow index
   **/
  event ReserveDataUpdated(
    address indexed asset,
    uint256 liquidityRate,
    uint256 stableBorrowRate,
    uint256 variableBorrowRate,
```

```
    uint256 liquidityIndex,
    uint256 variableBorrowIndex
  );

  using ReserveLogic for DataTypes.ReserveData;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;

  /**
   * @dev Returns the ongoing normalized income for the reserve
   * A value of 1e27 means there is no income. As time passes, the income is accrued
   * A value of 2*1e27 means for each unit of asset one unit of income has been accrued
   * @param reserve The reserve object
   * @return the normalized income. expressed in ray
   **/
  function getNormalizedIncome(DataTypes.ReserveData storage reserve)
    internal
    view
    returns (uint256)
  {
    uint40 timestamp = reserve.lastUpdateTimestamp;

    //solium-disable-next-line
    if (timestamp == uint40(block.timestamp)) {
      //if the index was updated in the same block, no need to perform any calculation
      return reserve.liquidityIndex;
    }

    uint256 cumulated =
      MathUtils.calculateLinearInterest(reserve.currentLiquidityRate, timestamp).rayMul(
        reserve.liquidityIndex
      );

    return cumulated;
  }

  /**
   * @dev Returns the ongoing normalized variable debt for the reserve
   * A value of 1e27 means there is no debt. As time passes, the income is accrued
   * A value of 2*1e27 means that for each unit of debt, one unit worth of interest has been accumula
   * @param reserve The reserve object
   * @return The normalized variable debt. expressed in ray
   **/
  function getNormalizedDebt(DataTypes.ReserveData storage reserve)
    internal
    view
    returns (uint256)
  {
    uint40 timestamp = reserve.lastUpdateTimestamp;

    //solium-disable-next-line
    if (timestamp == uint40(block.timestamp)) {
      //if the index was updated in the same block, no need to perform any calculation
      return reserve.variableBorrowIndex;
    }

    uint256 cumulated =
      MathUtils.calculateCompoundedInterest(reserve.currentVariableBorrowRate, timestamp).rayMul(
        reserve.variableBorrowIndex
      );

    return cumulated;
  }

  /**
   * @dev Updates the liquidity cumulative index and the variable borrow index.
   * @param reserve the reserve object
```

```
   **/
  function updateState(DataTypes.ReserveData storage reserve) internal {
    uint256 scaledVariableDebt =
      IVariableDebtToken(reserve.variableDebtTokenAddress).scaledTotalSupply();
    uint256 previousVariableBorrowIndex = reserve.variableBorrowIndex;
    uint256 previousLiquidityIndex = reserve.liquidityIndex;
    uint40 lastUpdatedTimestamp = reserve.lastUpdateTimestamp;

    (uint256 newLiquidityIndex, uint256 newVariableBorrowIndex) =
      _updateIndexes(
        reserve,
        scaledVariableDebt,
        previousLiquidityIndex,
        previousVariableBorrowIndex,
        lastUpdatedTimestamp
      );

    _mintToTreasury(
      reserve,
      scaledVariableDebt,
      previousVariableBorrowIndex,
      newLiquidityIndex,
      newVariableBorrowIndex,
      lastUpdatedTimestamp
    );
  }

  /**
   * @dev Accumulates a predefined amount of asset to the reserve as a fixed, instantaneous income. U
   * the flashloan fee to the reserve, and spread it between all the depositors
   * @param reserve The reserve object
   * @param totalLiquidity The total liquidity available in the reserve
   * @param amount The amount to accomulate
   **/
  function cumulateToLiquidityIndex(
    DataTypes.ReserveData storage reserve,
    uint256 totalLiquidity,
    uint256 amount
  ) internal {
    uint256 amountToLiquidityRatio = amount.wadToRay().rayDiv(totalLiquidity.wadToRay());

    uint256 result = amountToLiquidityRatio.add(WadRayMath.ray());

    result = result.rayMul(reserve.liquidityIndex);
    require(result <= type(uint128).max, Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

    reserve.liquidityIndex = uint128(result);
  }

  /**
   * @dev Initializes a reserve
   * @param reserve The reserve object
   * @param aTokenAddress The address of the overlying atoken contract
   * @param interestRateStrategyAddress The address of the interest rate strategy contract
   **/
  function init(
    DataTypes.ReserveData storage reserve,
    address aTokenAddress,
    address stableDebtTokenAddress,
    address variableDebtTokenAddress,
    address interestRateStrategyAddress
  ) external {
    require(reserve.aTokenAddress == address(0), Errors.RL_RESERVE_ALREADY_INITIALIZED);

    reserve.liquidityIndex = uint128(WadRayMath.ray());
    reserve.variableBorrowIndex = uint128(WadRayMath.ray());
```

```
    reserve.aTokenAddress = aTokenAddress;
    reserve.stableDebtTokenAddress = stableDebtTokenAddress;
    reserve.variableDebtTokenAddress = variableDebtTokenAddress;
    reserve.interestRateStrategyAddress = interestRateStrategyAddress;
  }

  struct UpdateInterestRatesLocalVars {
    address stableDebtTokenAddress;
    uint256 availableLiquidity;
    uint256 totalStableDebt;
    uint256 newLiquidityRate;
    uint256 newStableRate;
    uint256 newVariableRate;
    uint256 avgStableRate;
    uint256 totalVariableDebt;
  }

  /**
   * @dev Updates the reserve current stable borrow rate, the current variable borrow rate and the cu
   * @param reserve The address of the reserve to be updated
   * @param liquidityAdded The amount of liquidity added to the protocol (deposit or repay) in the pr
   * @param liquidityTaken The amount of liquidity taken from the protocol (redeem or borrow)
   **/
  function updateInterestRates(
    DataTypes.ReserveData storage reserve,
    address reserveAddress,
    address aTokenAddress,
    uint256 liquidityAdded,
    uint256 liquidityTaken
  ) internal {
    UpdateInterestRatesLocalVars memory vars;

    vars.stableDebtTokenAddress = reserve.stableDebtTokenAddress;

    (vars.totalStableDebt, vars.avgStableRate) = IStableDebtToken(vars.stableDebtTokenAddress)
      .getTotalSupplyAndAvgRate();

    //calculates the total variable debt locally using the scaled total supply instead
    //of totalSupply(), as it's noticeably cheaper. Also, the index has been
    //updated by the previous updateState() call
    vars.totalVariableDebt = IVariableDebtToken(reserve.variableDebtTokenAddress)
      .scaledTotalSupply()
      .rayMul(reserve.variableBorrowIndex);

    (
      vars.newLiquidityRate,
      vars.newStableRate,
      vars.newVariableRate
    ) = IReserveInterestRateStrategy(reserve.interestRateStrategyAddress).calculateInterestRates(
      reserveAddress,
      aTokenAddress,
      liquidityAdded,
      liquidityTaken,
      vars.totalStableDebt,
      vars.totalVariableDebt,
      vars.avgStableRate,
      reserve.configuration.getReserveFactor()
    );
    require(vars.newLiquidityRate <= type(uint128).max, Errors.RL_LIQUIDITY_RATE_OVERFLOW);
    require(vars.newStableRate <= type(uint128).max, Errors.RL_STABLE_BORROW_RATE_OVERFLOW);
    require(vars.newVariableRate <= type(uint128).max, Errors.RL_VARIABLE_BORROW_RATE_OVERFLOW);

    reserve.currentLiquidityRate = uint128(vars.newLiquidityRate);
    reserve.currentStableBorrowRate = uint128(vars.newStableRate);
    reserve.currentVariableBorrowRate = uint128(vars.newVariableRate);
```

```
    emit ReserveDataUpdated(
      reserveAddress,
      vars.newLiquidityRate,
      vars.newStableRate,
      vars.newVariableRate,
      reserve.liquidityIndex,
      reserve.variableBorrowIndex
    );
  }

  struct MintToTreasuryLocalVars {
    uint256 currentStableDebt;
    uint256 principalStableDebt;
    uint256 previousStableDebt;
    uint256 currentVariableDebt;
    uint256 previousVariableDebt;
    uint256 avgStableRate;
    uint256 cumulatedStableInterest;
    uint256 totalDebtAccrued;
    uint256 amountToMint;
    uint256 reserveFactor;
    uint40 stableSupplyUpdatedTimestamp;
  }

  /**
   * @dev Mints part of the repaid interest to the reserve treasury as a function of the reserveFacto
   * specific asset.
   * @param reserve The reserve reserve to be updated
   * @param scaledVariableDebt The current scaled total variable debt
   * @param previousVariableBorrowIndex The variable borrow index before the last accumulation of the
   * @param newLiquidityIndex The new liquidity index
   * @param newVariableBorrowIndex The variable borrow index after the last accumulation of the inter
   **/
  function _mintToTreasury(
    DataTypes.ReserveData storage reserve,
    uint256 scaledVariableDebt,
    uint256 previousVariableBorrowIndex,
    uint256 newLiquidityIndex,
    uint256 newVariableBorrowIndex,
    uint40 timestamp
  ) internal {
    MintToTreasuryLocalVars memory vars;

    vars.reserveFactor = reserve.configuration.getReserveFactor();

    if (vars.reserveFactor == 0) {
      return;
    }

    //fetching the principal, total stable debt and the avg stable rate
    (
      vars.principalStableDebt,
      vars.currentStableDebt,
      vars.avgStableRate,
      vars.stableSupplyUpdatedTimestamp
    ) = IStableDebtToken(reserve.stableDebtTokenAddress).getSupplyData();

    //calculate the last principal variable debt
    vars.previousVariableDebt = scaledVariableDebt.rayMul(previousVariableBorrowIndex);

    //calculate the new total supply after accumulation of the index
    vars.currentVariableDebt = scaledVariableDebt.rayMul(newVariableBorrowIndex);

    //calculate the stable debt until the last timestamp update
    vars.cumulatedStableInterest = MathUtils.calculateCompoundedInterest(
      vars.avgStableRate,
```

```
      vars.stableSupplyUpdatedTimestamp,
      timestamp
    );

    vars.previousStableDebt = vars.principalStableDebt.rayMul(vars.cumulatedStableInterest);

    //debt accrued is the sum of the current debt minus the sum of the debt at the last update
    vars.totalDebtAccrued = vars
      .currentVariableDebt
      .add(vars.currentStableDebt)
      .sub(vars.previousVariableDebt)
      .sub(vars.previousStableDebt);

    vars.amountToMint = vars.totalDebtAccrued.percentMul(vars.reserveFactor);

    if (vars.amountToMint != 0) {
      IAToken(reserve.aTokenAddress).mintToTreasury(vars.amountToMint, newLiquidityIndex);
    }
  }

  /**
   * @dev Updates the reserve indexes and the timestamp of the update
   * @param reserve The reserve reserve to be updated
   * @param scaledVariableDebt The scaled variable debt
   * @param liquidityIndex The last stored liquidity index
   * @param variableBorrowIndex The last stored variable borrow index
   **/
  function _updateIndexes(
    DataTypes.ReserveData storage reserve,
    uint256 scaledVariableDebt,
    uint256 liquidityIndex,
    uint256 variableBorrowIndex,
    uint40 timestamp
  ) internal returns (uint256, uint256) {
    uint256 currentLiquidityRate = reserve.currentLiquidityRate;

    uint256 newLiquidityIndex = liquidityIndex;
    uint256 newVariableBorrowIndex = variableBorrowIndex;

    //only cumulating if there is any income being produced
    if (currentLiquidityRate > 0) {
      uint256 cumulatedLiquidityInterest =
        MathUtils.calculateLinearInterest(currentLiquidityRate, timestamp);
      newLiquidityIndex = cumulatedLiquidityInterest.rayMul(liquidityIndex);
      require(newLiquidityIndex <= type(uint128).max, Errors.RL_LIQUIDITY_INDEX_OVERFLOW);

      reserve.liquidityIndex = uint128(newLiquidityIndex);

      //as the liquidity rate might come only from stable rate loans, we need to ensure
      //that there is actual variable debt before accumulating
      if (scaledVariableDebt != 0) {
        uint256 cumulatedVariableBorrowInterest =
          MathUtils.calculateCompoundedInterest(reserve.currentVariableBorrowRate, timestamp);
        newVariableBorrowIndex = cumulatedVariableBorrowInterest.rayMul(variableBorrowIndex);
        require(
          newVariableBorrowIndex <= type(uint128).max,
          Errors.RL_VARIABLE_BORROW_INDEX_OVERFLOW
        );
        reserve.variableBorrowIndex = uint128(newVariableBorrowIndex);
      }
    }

    //solium-disable-next-line
    reserve.lastUpdateTimestamp = uint40(block.timestamp);
    return (newLiquidityIndex, newVariableBorrowIndex);
  }
```

```
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

library DataTypes {
  // refer to the whitepaper, section 1.1 basic concepts for a formal description of these properties
  struct ReserveData {
    //stores the reserve configuration
    ReserveConfigurationMap configuration;
    //the liquidity index. Expressed in ray
    uint128 liquidityIndex;
    //variable borrow index. Expressed in ray
    uint128 variableBorrowIndex;
    //the current supply rate. Expressed in ray
    uint128 currentLiquidityRate;
    //the current variable borrow rate. Expressed in ray
    uint128 currentVariableBorrowRate;
    //the current stable borrow rate. Expressed in ray
    uint128 currentStableBorrowRate;
    uint40 lastUpdateTimestamp;
    //tokens addresses
    address aTokenAddress;
    address stableDebtTokenAddress;
    address variableDebtTokenAddress;
    //address of the interest rate strategy
    address interestRateStrategyAddress;
    //the id of the reserve. Represents the position in the list of the active reserves
    uint8 id;
  }

  struct ReserveConfigurationMap {
    //bit 0-15: LTV
    //bit 16-31: Liq. threshold
    //bit 32-47: Liq. bonus
    //bit 48-55: Decimals
    //bit 56: Reserve is active
    //bit 57: reserve is frozen
    //bit 58: borrowing is enabled
    //bit 59: stable rate borrowing enabled
    //bit 60-63: reserved
    //bit 64-79: reserve factor
    uint256 data;
  }

  struct UserConfigurationMap {
    uint256 data;
  }

  enum InterestRateMode {NONE, STABLE, VARIABLE}
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {Errors} from '../helpers/Errors.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title UserConfiguration library
 * @author Aave
 * @notice Implements the bitmap logic to handle the user configuration
 */
library UserConfiguration {
  uint256 internal constant BORROWING_MASK =
    0x5555555555555555555555555555555555555555555555555555555555555555;

  /**
```

```
   * @dev Sets if the user is borrowing the reserve identified by reserveIndex
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @param borrowing True if the user is borrowing the reserve, false otherwise
   **/
  function setBorrowing(
    DataTypes.UserConfigurationMap storage self,
    uint256 reserveIndex,
    bool borrowing
  ) internal {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    self.data =
      (self.data & ~(1 << (reserveIndex * 2))) |
      (uint256(borrowing ? 1 : 0) << (reserveIndex * 2));
  }

  /**
   * @dev Sets if the user is using as collateral the reserve identified by reserveIndex
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @param usingAsCollateral True if the user is usin the reserve as collateral, false otherwise
   **/
  function setUsingAsCollateral(
    DataTypes.UserConfigurationMap storage self,
    uint256 reserveIndex,
    bool usingAsCollateral
  ) internal {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    self.data =
      (self.data & ~(1 << (reserveIndex * 2 + 1))) |
      (uint256(usingAsCollateral ? 1 : 0) << (reserveIndex * 2 + 1));
  }

  /**
   * @dev Used to validate if a user has been using the reserve for borrowing or as collateral
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @return True if the user has been using a reserve for borrowing or as collateral, false otherwis
   **/
  function isUsingAsCollateralOrBorrowing(
    DataTypes.UserConfigurationMap memory self,
    uint256 reserveIndex
  ) internal pure returns (bool) {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    return (self.data >> (reserveIndex * 2)) & 3 != 0;
  }

  /**
   * @dev Used to validate if a user has been using the reserve for borrowing
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
   * @return True if the user has been using a reserve for borrowing, false otherwise
   **/
  function isBorrowing(DataTypes.UserConfigurationMap memory self, uint256 reserveIndex)
    internal
    pure
    returns (bool)
  {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    return (self.data >> (reserveIndex * 2)) & 1 != 0;
  }

  /**
   * @dev Used to validate if a user has been using the reserve as collateral
   * @param self The configuration object
   * @param reserveIndex The index of the reserve in the bitmap
```

```
   * @return True if the user has been using a reserve as collateral, false otherwise
   **/
  function isUsingAsCollateral(DataTypes.UserConfigurationMap memory self, uint256 reserveIndex)
    internal
    pure
    returns (bool)
  {
    require(reserveIndex < 128, Errors.UL_INVALID_INDEX);
    return (self.data >> (reserveIndex * 2 + 1)) & 1 != 0;
  }

  /**
   * @dev Used to validate if a user has been borrowing from any reserve
   * @param self The configuration object
   * @return True if the user has been borrowing any reserve, false otherwise
   **/
  function isBorrowingAny(DataTypes.UserConfigurationMap memory self) internal pure returns (bool) {
    return self.data & BORROWING_MASK != 0;
  }

  /**
   * @dev Used to validate if a user has not been using any reserve
   * @param self The configuration object
   * @return True if the user has been borrowing any reserve, false otherwise
   **/
  function isEmpty(DataTypes.UserConfigurationMap memory self) internal pure returns (bool) {
    return self.data == 0;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {Errors} from '../helpers/Errors.sol';
import {DataTypes} from '../types/DataTypes.sol';

/**
 * @title ReserveConfiguration library
 * @author Aave
 * @notice Implements the bitmap logic to handle the reserve configuration
 */
library ReserveConfiguration {
  uint256 constant LTV_MASK =                   0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000;
  uint256 constant LIQUIDATION_THRESHOLD_MASK = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000FFFFFFFF;
  uint256 constant LIQUIDATION_BONUS_MASK =     0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000FFFFFFFFFF;
  uint256 constant DECIMALS_MASK =              0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0;
  uint256 constant ACTIVE_MASK =                0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEF;
  uint256 constant FROZEN_MASK =                0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFDF;
  uint256 constant BORROWING_MASK =             0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFBF;
  uint256 constant STABLE_BORROWING_MASK =      0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7F;
  uint256 constant RESERVE_FACTOR_MASK =        0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000FFFFFFFF;

  /// @dev For the LTV, the start bit is 0 (up to 15), hence no bitshifting is needed
  uint256 constant LIQUIDATION_THRESHOLD_START_BIT_POSITION = 16;
  uint256 constant LIQUIDATION_BONUS_START_BIT_POSITION = 32;
  uint256 constant RESERVE_DECIMALS_START_BIT_POSITION = 48;
  uint256 constant IS_ACTIVE_START_BIT_POSITION = 56;
  uint256 constant IS_FROZEN_START_BIT_POSITION = 57;
  uint256 constant BORROWING_ENABLED_START_BIT_POSITION = 58;
  uint256 constant STABLE_BORROWING_ENABLED_START_BIT_POSITION = 59;
  uint256 constant RESERVE_FACTOR_START_BIT_POSITION = 64;

  uint256 constant MAX_VALID_LTV = 65535;
  uint256 constant MAX_VALID_LIQUIDATION_THRESHOLD = 65535;
  uint256 constant MAX_VALID_LIQUIDATION_BONUS = 65535;
  uint256 constant MAX_VALID_DECIMALS = 255;
  uint256 constant MAX_VALID_RESERVE_FACTOR = 65535;
```

```
/**
 * @dev Sets the Loan to Value of the reserve
 * @param self The reserve configuration
 * @param ltv the new ltv
 **/
function setLtv(DataTypes.ReserveConfigurationMap memory self, uint256 ltv) internal pure {
  require(ltv <= MAX_VALID_LTV, Errors.RC_INVALID_LTV);

  self.data = (self.data & LTV_MASK) | ltv;
}

/**
 * @dev Gets the Loan to Value of the reserve
 * @param self The reserve configuration
 * @return The loan to value
 **/
function getLtv(DataTypes.ReserveConfigurationMap storage self) internal view returns (uint256) {
  return self.data & ~LTV_MASK;
}

/**
 * @dev Sets the liquidation threshold of the reserve
 * @param self The reserve configuration
 * @param threshold The new liquidation threshold
 **/
function setLiquidationThreshold(DataTypes.ReserveConfigurationMap memory self, uint256 threshold)
  internal
  pure
{
  require(threshold <= MAX_VALID_LIQUIDATION_THRESHOLD, Errors.RC_INVALID_LIQ_THRESHOLD);

  self.data =
    (self.data & LIQUIDATION_THRESHOLD_MASK) |
    (threshold << LIQUIDATION_THRESHOLD_START_BIT_POSITION);
}

/**
 * @dev Gets the liquidation threshold of the reserve
 * @param self The reserve configuration
 * @return The liquidation threshold
 **/
function getLiquidationThreshold(DataTypes.ReserveConfigurationMap storage self)
  internal
  view
  returns (uint256)
{
  return (self.data & ~LIQUIDATION_THRESHOLD_MASK) >> LIQUIDATION_THRESHOLD_START_BIT_POSITION;
}

/**
 * @dev Sets the liquidation bonus of the reserve
 * @param self The reserve configuration
 * @param bonus The new liquidation bonus
 **/
function setLiquidationBonus(DataTypes.ReserveConfigurationMap memory self, uint256 bonus)
  internal
  pure
{
  require(bonus <= MAX_VALID_LIQUIDATION_BONUS, Errors.RC_INVALID_LIQ_BONUS);

  self.data =
    (self.data & LIQUIDATION_BONUS_MASK) |
    (bonus << LIQUIDATION_BONUS_START_BIT_POSITION);
}
```

```
  /**
   * @dev Gets the liquidation bonus of the reserve
   * @param self The reserve configuration
   * @return The liquidation bonus
   **/
  function getLiquidationBonus(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (uint256)
  {
    return (self.data & ~LIQUIDATION_BONUS_MASK) >> LIQUIDATION_BONUS_START_BIT_POSITION;
  }

  /**
   * @dev Sets the decimals of the underlying asset of the reserve
   * @param self The reserve configuration
   * @param decimals The decimals
   **/
  function setDecimals(DataTypes.ReserveConfigurationMap memory self, uint256 decimals)
    internal
    pure
  {
    require(decimals <= MAX_VALID_DECIMALS, Errors.RC_INVALID_DECIMALS);

    self.data = (self.data & DECIMALS_MASK) | (decimals << RESERVE_DECIMALS_START_BIT_POSITION);
  }

  /**
   * @dev Gets the decimals of the underlying asset of the reserve
   * @param self The reserve configuration
   * @return The decimals of the asset
   **/
  function getDecimals(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (uint256)
  {
    return (self.data & ~DECIMALS_MASK) >> RESERVE_DECIMALS_START_BIT_POSITION;
  }

  /**
   * @dev Sets the active state of the reserve
   * @param self The reserve configuration
   * @param active The active state
   **/
  function setActive(DataTypes.ReserveConfigurationMap memory self, bool active) internal pure {
    self.data =
      (self.data & ACTIVE_MASK) |
      (uint256(active ? 1 : 0) << IS_ACTIVE_START_BIT_POSITION);
  }

  /**
   * @dev Gets the active state of the reserve
   * @param self The reserve configuration
   * @return The active state
   **/
  function getActive(DataTypes.ReserveConfigurationMap storage self) internal view returns (bool) {
    return (self.data & ~ACTIVE_MASK) != 0;
  }

  /**
   * @dev Sets the frozen state of the reserve
   * @param self The reserve configuration
   * @param frozen The frozen state
   **/
  function setFrozen(DataTypes.ReserveConfigurationMap memory self, bool frozen) internal pure {
```

```
    self.data =
      (self.data & FROZEN_MASK) |
      (uint256(frozen ? 1 : 0) << IS_FROZEN_START_BIT_POSITION);
  }

  /**
   * @dev Gets the frozen state of the reserve
   * @param self The reserve configuration
   * @return The frozen state
   **/
  function getFrozen(DataTypes.ReserveConfigurationMap storage self) internal view returns (bool) {
    return (self.data & ~FROZEN_MASK) != 0;
  }

  /**
   * @dev Enables or disables borrowing on the reserve
   * @param self The reserve configuration
   * @param enabled True if the borrowing needs to be enabled, false otherwise
   **/
  function setBorrowingEnabled(DataTypes.ReserveConfigurationMap memory self, bool enabled)
    internal
    pure
  {
    self.data =
      (self.data & BORROWING_MASK) |
      (uint256(enabled ? 1 : 0) << BORROWING_ENABLED_START_BIT_POSITION);
  }

  /**
   * @dev Gets the borrowing state of the reserve
   * @param self The reserve configuration
   * @return The borrowing state
   **/
  function getBorrowingEnabled(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (bool)
  {
    return (self.data & ~BORROWING_MASK) != 0;
  }

  /**
   * @dev Enables or disables stable rate borrowing on the reserve
   * @param self The reserve configuration
   * @param enabled True if the stable rate borrowing needs to be enabled, false otherwise
   **/
  function setStableRateBorrowingEnabled(
    DataTypes.ReserveConfigurationMap memory self,
    bool enabled
  ) internal pure {
    self.data =
      (self.data & STABLE_BORROWING_MASK) |
      (uint256(enabled ? 1 : 0) << STABLE_BORROWING_ENABLED_START_BIT_POSITION);
  }

  /**
   * @dev Gets the stable rate borrowing state of the reserve
   * @param self The reserve configuration
   * @return The stable rate borrowing state
   **/
  function getStableRateBorrowingEnabled(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (bool)
  {
    return (self.data & ~STABLE_BORROWING_MASK) != 0;
```

```
  }

  /**
   * @dev Sets the reserve factor of the reserve
   * @param self The reserve configuration
   * @param reserveFactor The reserve factor
   **/
  function setReserveFactor(DataTypes.ReserveConfigurationMap memory self, uint256 reserveFactor)
    internal
    pure
  {
    require(reserveFactor <= MAX_VALID_RESERVE_FACTOR, Errors.RC_INVALID_RESERVE_FACTOR);

    self.data =
      (self.data & RESERVE_FACTOR_MASK) |
      (reserveFactor << RESERVE_FACTOR_START_BIT_POSITION);
  }

  /**
   * @dev Gets the reserve factor of the reserve
   * @param self The reserve configuration
   * @return The reserve factor
   **/
  function getReserveFactor(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (uint256)
  {
    return (self.data & ~RESERVE_FACTOR_MASK) >> RESERVE_FACTOR_START_BIT_POSITION;
  }

  /**
   * @dev Gets the configuration flags of the reserve
   * @param self The reserve configuration
   * @return The state flags representing active, frozen, borrowing enabled, stableRateBorrowing enab
   **/
  function getFlags(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (
      bool,
      bool,
      bool,
      bool
    )
  {
    uint256 dataLocal = self.data;

    return (
      (dataLocal & ~ACTIVE_MASK) != 0,
      (dataLocal & ~FROZEN_MASK) != 0,
      (dataLocal & ~BORROWING_MASK) != 0,
      (dataLocal & ~STABLE_BORROWING_MASK) != 0
    );
  }

  /**
   * @dev Gets the configuration paramters of the reserve
   * @param self The reserve configuration
   * @return The state params representing ltv, liquidation threshold, liquidation bonus, the reserve
   **/
  function getParams(DataTypes.ReserveConfigurationMap storage self)
    internal
    view
    returns (
      uint256,
```

```
      uint256,
      uint256,
      uint256,
      uint256
    )
  {
    uint256 dataLocal = self.data;

    return (
      dataLocal & ~LTV_MASK,
      (dataLocal & ~LIQUIDATION_THRESHOLD_MASK) >> LIQUIDATION_THRESHOLD_START_BIT_POSITION,
      (dataLocal & ~LIQUIDATION_BONUS_MASK) >> LIQUIDATION_BONUS_START_BIT_POSITION,
      (dataLocal & ~DECIMALS_MASK) >> RESERVE_DECIMALS_START_BIT_POSITION,
      (dataLocal & ~RESERVE_FACTOR_MASK) >> RESERVE_FACTOR_START_BIT_POSITION
    );
  }

  /**
   * @dev Gets the configuration paramters of the reserve from a memory object
   * @param self The reserve configuration
   * @return The state params representing ltv, liquidation threshold, liquidation bonus, the reserve
   **/
  function getParamsMemory(DataTypes.ReserveConfigurationMap memory self)
    internal
    pure
    returns (
      uint256,
      uint256,
      uint256,
      uint256,
      uint256
    )
  {
    return (
      self.data & ~LTV_MASK,
      (self.data & ~LIQUIDATION_THRESHOLD_MASK) >> LIQUIDATION_THRESHOLD_START_BIT_POSITION,
      (self.data & ~LIQUIDATION_BONUS_MASK) >> LIQUIDATION_BONUS_START_BIT_POSITION,
      (self.data & ~DECIMALS_MASK) >> RESERVE_DECIMALS_START_BIT_POSITION,
      (self.data & ~RESERVE_FACTOR_MASK) >> RESERVE_FACTOR_START_BIT_POSITION
    );
  }

  /**
   * @dev Gets the configuration flags of the reserve from a memory object
   * @param self The reserve configuration
   * @return The state flags representing active, frozen, borrowing enabled, stableRateBorrowing enab
   **/
  function getFlagsMemory(DataTypes.ReserveConfigurationMap memory self)
    internal
    pure
    returns (
      bool,
      bool,
      bool,
      bool
    )
  {
    return (
      (self.data & ~ACTIVE_MASK) != 0,
      (self.data & ~FROZEN_MASK) != 0,
      (self.data & ~BORROWING_MASK) != 0,
      (self.data & ~STABLE_BORROWING_MASK) != 0
    );
  }
}
// SPDX-License-Identifier: agpl-3.0
```

```
pragma solidity 0.6.12;

import {Ownable} from '../../dependencies/openzeppelin/contracts/Ownable.sol';

// Prettier ignore to prevent buidler flatter bug
// prettier-ignore
import {InitializableImmutableAdminUpgradeabilityProxy} from '../libraries/aave-upgradeability/Initia

import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';

/**
 * @title LendingPoolAddressesProvider contract
 * @dev Main registry of addresses part of or connected to the protocol, including permissioned roles
 * - Acting also as factory of proxies and admin of those, so with right to change its implementation
 * - Owned by the Aave Governance
 * @author Aave
 **/
contract LendingPoolAddressesProvider is Ownable, ILendingPoolAddressesProvider {
  string private _marketId;
  mapping(bytes32 => address) private _addresses;

  bytes32 private constant LENDING_POOL = 'LENDING_POOL';
  bytes32 private constant LENDING_POOL_CONFIGURATOR = 'LENDING_POOL_CONFIGURATOR';
  bytes32 private constant POOL_ADMIN = 'POOL_ADMIN';
  bytes32 private constant EMERGENCY_ADMIN = 'EMERGENCY_ADMIN';
  bytes32 private constant LENDING_POOL_COLLATERAL_MANAGER = 'COLLATERAL_MANAGER';
  bytes32 private constant PRICE_ORACLE = 'PRICE_ORACLE';
  bytes32 private constant LENDING_RATE_ORACLE = 'LENDING_RATE_ORACLE';

  constructor(string memory marketId) public {
    _setMarketId(marketId);
  }

  /**
   * @dev Returns the id of the Aave market to which this contracts points to
   * @return The market id
   **/
  function getMarketId() external view override returns (string memory) {
    return _marketId;
  }

  /**
   * @dev Allows to set the market which this LendingPoolAddressesProvider represents
   * @param marketId The market id
   */
  function setMarketId(string memory marketId) external override onlyOwner {
    _setMarketId(marketId);
  }

  /**
   * @dev General function to update the implementation of a proxy registered with
   * certain `id`. If there is no proxy registered, it will instantiate one and
   * set as implementation the `implementationAddress`
   * IMPORTANT Use this function carefully, only for ids that don't have an explicit
   * setter function, in order to avoid unexpected consequences
   * @param id The id
   * @param implementationAddress The address of the new implementation
   */
  function setAddressAsProxy(bytes32 id, address implementationAddress)
    external
    override
    onlyOwner
  {
    _updateImpl(id, implementationAddress);
    emit AddressSet(id, implementationAddress, true);
  }
```

```
/**
 * @dev Sets an address for an id replacing the address saved in the addresses map
 * IMPORTANT Use this function carefully, as it will do a hard replacement
 * @param id The id
 * @param newAddress The address to set
 */
function setAddress(bytes32 id, address newAddress) external override onlyOwner {
  _addresses[id] = newAddress;
  emit AddressSet(id, newAddress, false);
}

/**
 * @dev Returns an address by id
 * @return The address
 */
function getAddress(bytes32 id) public view override returns (address) {
  return _addresses[id];
}

/**
 * @dev Returns the address of the LendingPool proxy
 * @return The LendingPool proxy address
 **/
function getLendingPool() external view override returns (address) {
  return getAddress(LENDING_POOL);
}

/**
 * @dev Updates the implementation of the LendingPool, or creates the proxy
 * setting the new `pool` implementation on the first time calling it
 * @param pool The new LendingPool implementation
 **/
function setLendingPoolImpl(address pool) external override onlyOwner {
  _updateImpl(LENDING_POOL, pool);
  emit LendingPoolUpdated(pool);
}

/**
 * @dev Returns the address of the LendingPoolConfigurator proxy
 * @return The LendingPoolConfigurator proxy address
 **/
function getLendingPoolConfigurator() external view override returns (address) {
  return getAddress(LENDING_POOL_CONFIGURATOR);
}

/**
 * @dev Updates the implementation of the LendingPoolConfigurator, or creates the proxy
 * setting the new `configurator` implementation on the first time calling it
 * @param configurator The new LendingPoolConfigurator implementation
 **/
function setLendingPoolConfiguratorImpl(address configurator) external override onlyOwner {
  _updateImpl(LENDING_POOL_CONFIGURATOR, configurator);
  emit LendingPoolConfiguratorUpdated(configurator);
}

/**
 * @dev Returns the address of the LendingPoolCollateralManager. Since the manager is used
 * through delegateCall within the LendingPool contract, the proxy contract pattern does not work p
 * the addresses are changed directly
 * @return The address of the LendingPoolCollateralManager
 **/

function getLendingPoolCollateralManager() external view override returns (address) {
  return getAddress(LENDING_POOL_COLLATERAL_MANAGER);
}
```

```
/**
 * @dev Updates the address of the LendingPoolCollateralManager
 * @param manager The new LendingPoolCollateralManager address
 **/
function setLendingPoolCollateralManager(address manager) external override onlyOwner {
  _addresses[LENDING_POOL_COLLATERAL_MANAGER] = manager;
  emit LendingPoolCollateralManagerUpdated(manager);
}

/**
 * @dev The functions below are getters/setters of addresses that are outside the context
 * of the protocol hence the upgradable proxy pattern is not used
 **/

function getPoolAdmin() external view override returns (address) {
  return getAddress(POOL_ADMIN);
}

function setPoolAdmin(address admin) external override onlyOwner {
  _addresses[POOL_ADMIN] = admin;
  emit ConfigurationAdminUpdated(admin);
}

function getEmergencyAdmin() external view override returns (address) {
  return getAddress(EMERGENCY_ADMIN);
}

function setEmergencyAdmin(address emergencyAdmin) external override onlyOwner {
  _addresses[EMERGENCY_ADMIN] = emergencyAdmin;
  emit EmergencyAdminUpdated(emergencyAdmin);
}

function getPriceOracle() external view override returns (address) {
  return getAddress(PRICE_ORACLE);
}

function setPriceOracle(address priceOracle) external override onlyOwner {
  _addresses[PRICE_ORACLE] = priceOracle;
  emit PriceOracleUpdated(priceOracle);
}

function getLendingRateOracle() external view override returns (address) {
  return getAddress(LENDING_RATE_ORACLE);
}

function setLendingRateOracle(address lendingRateOracle) external override onlyOwner {
  _addresses[LENDING_RATE_ORACLE] = lendingRateOracle;
  emit LendingRateOracleUpdated(lendingRateOracle);
}

/**
 * @dev Internal function to update the implementation of a specific proxied component of the proto
 * - If there is no proxy registered in the given `id`, it creates the proxy setting `newAdress`
 *   as implementation and calls the initialize() function on the proxy
 * - If there is already a proxy registered, it just updates the implementation to `newAddress` and
 *   calls the initialize() function via upgradeToAndCall() in the proxy
 * @param id The id of the proxy to be updated
 * @param newAddress The address of the new implementation
 **/
function _updateImpl(bytes32 id, address newAddress) internal {
  address payable proxyAddress = payable(_addresses[id]);

  InitializableImmutableAdminUpgradeabilityProxy proxy =
    InitializableImmutableAdminUpgradeabilityProxy(proxyAddress);
  bytes memory params = abi.encodeWithSignature('initialize(address)', address(this));
```

```
    if (proxyAddress == address(0)) {
      proxy = new InitializableImmutableAdminUpgradeabilityProxy(address(this));
      proxy.initialize(newAddress, params);
      _addresses[id] = address(proxy);
      emit ProxyCreated(id, address(proxy));
    } else {
      proxy.upgradeToAndCall(newAddress, params);
    }
  }

  function _setMarketId(string memory marketId) internal {
    _marketId = marketId;
    emit MarketIdSet(marketId);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {Ownable} from '../../dependencies/openzeppelin/contracts/Ownable.sol';
import {
  ILendingPoolAddressesProviderRegistry
} from '../../interfaces/ILendingPoolAddressesProviderRegistry.sol';
import {Errors} from '../libraries/helpers/Errors.sol';

/**
 * @title LendingPoolAddressesProviderRegistry contract
 * @dev Main registry of LendingPoolAddressesProvider of multiple Aave protocol's markets
 * - Used for indexing purposes of Aave protocol's markets
 * - The id assigned to a LendingPoolAddressesProvider refers to the market it is connected with,
 *   for example with `0` for the Aave main market and `1` for the next created
 * @author Aave
 **/
contract LendingPoolAddressesProviderRegistry is Ownable, ILendingPoolAddressesProviderRegistry {
  mapping(address => uint256) private _addressesProviders;
  address[] private _addressesProvidersList;

  /**
   * @dev Returns the list of registered addresses provider
   * @return The list of addresses provider, potentially containing address(0) elements
   **/
  function getAddressesProvidersList() external view override returns (address[] memory) {
    address[] memory addressesProvidersList = _addressesProvidersList;

    uint256 maxLength = addressesProvidersList.length;

    address[] memory activeProviders = new address[](maxLength);

    for (uint256 i = 0; i < maxLength; i++) {
      if (_addressesProviders[addressesProvidersList[i]] > 0) {
        activeProviders[i] = addressesProvidersList[i];
      }
    }

    return activeProviders;
  }

  /**
   * @dev Registers an addresses provider
   * @param provider The address of the new LendingPoolAddressesProvider
   * @param id The id for the new LendingPoolAddressesProvider, referring to the market it belongs to
   **/
  function registerAddressesProvider(address provider, uint256 id) external override onlyOwner {
    require(id != 0, Errors.LPAPR_INVALID_ADDRESSES_PROVIDER_ID);

    _addressesProviders[provider] = id;
```

```
    _addToAddressesProvidersList(provider);
    emit AddressesProviderRegistered(provider);
  }

  /**
   * @dev Removes a LendingPoolAddressesProvider from the list of registered addresses provider
   * @param provider The LendingPoolAddressesProvider address
   **/
  function unregisterAddressesProvider(address provider) external override onlyOwner {
    require(_addressesProviders[provider] > 0, Errors.LPAPR_PROVIDER_NOT_REGISTERED);
    _addressesProviders[provider] = 0;
    emit AddressesProviderUnregistered(provider);
  }

  /**
   * @dev Returns the id on a registered LendingPoolAddressesProvider
   * @return The id or 0 if the LendingPoolAddressesProvider is not registered
   */
  function getAddressesProviderIdByAddress(address addressesProvider)
    external
    view
    override
    returns (uint256)
  {
    return _addressesProviders[addressesProvider];
  }

  function _addToAddressesProvidersList(address provider) internal {
    uint256 providersCount = _addressesProvidersList.length;

    for (uint256 i = 0; i < providersCount; i++) {
      if (_addressesProvidersList[i] == provider) {
        return;
      }
    }

    _addressesProvidersList.push(provider);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './UpgradeabilityProxy.sol';

/**
 * @title BaseAdminUpgradeabilityProxy
 * @dev This contract combines an upgradeability proxy with an authorization
 * mechanism for administrative tasks.
 * All external functions in this contract must be guarded by the
 * `ifAdmin` modifier. See ethereum/solidity#3864 for a Solidity
 * feature proposal that would enable this to be done automatically.
 */
contract BaseAdminUpgradeabilityProxy is BaseUpgradeabilityProxy {
  /**
   * @dev Emitted when the administration has been transferred.
   * @param previousAdmin Address of the previous admin.
   * @param newAdmin Address of the new admin.
   */
  event AdminChanged(address previousAdmin, address newAdmin);

  /**
   * @dev Storage slot with the admin of the contract.
   * This is the keccak-256 hash of "eip1967.proxy.admin" subtracted by 1, and is
   * validated in the constructor.
   */
  bytes32 internal constant ADMIN_SLOT =
```

```
    0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103;

  /**
   * @dev Modifier to check whether the `msg.sender` is the admin.
   * If it is, it will run the function. Otherwise, it will delegate the call
   * to the implementation.
   */
  modifier ifAdmin() {
    if (msg.sender == _admin()) {
      _;
    } else {
      _fallback();
    }
  }

  /**
   * @return The address of the proxy admin.
   */
  function admin() external ifAdmin returns (address) {
    return _admin();
  }

  /**
   * @return The address of the implementation.
   */
  function implementation() external ifAdmin returns (address) {
    return _implementation();
  }

  /**
   * @dev Changes the admin of the proxy.
   * Only the current admin can call this function.
   * @param newAdmin Address to transfer proxy administration to.
   */
  function changeAdmin(address newAdmin) external ifAdmin {
    require(newAdmin != address(0), 'Cannot change the admin of a proxy to the zero address');
    emit AdminChanged(_admin(), newAdmin);
    _setAdmin(newAdmin);
  }

  /**
   * @dev Upgrade the backing implementation of the proxy.
   * Only the admin can call this function.
   * @param newImplementation Address of the new implementation.
   */
  function upgradeTo(address newImplementation) external ifAdmin {
    _upgradeTo(newImplementation);
  }

  /**
   * @dev Upgrade the backing implementation of the proxy and call a function
   * on the new implementation.
   * This is useful to initialize the proxied contract.
   * @param newImplementation Address of the new implementation.
   * @param data Data to send as msg.data in the low level call.
   * It should include the signature and the parameters of the function to be called, as described in
   * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding
   */
  function upgradeToAndCall(address newImplementation, bytes calldata data)
    external
    payable
    ifAdmin
  {
    _upgradeTo(newImplementation);
    (bool success, ) = newImplementation.delegatecall(data);
    require(success);
```

```
  }

  /**
   * @return adm The admin slot.
   */
  function _admin() internal view returns (address adm) {
    bytes32 slot = ADMIN_SLOT;
    //solium-disable-next-line
    assembly {
      adm := sload(slot)
    }
  }

  /**
   * @dev Sets the address of the proxy admin.
   * @param newAdmin Address of the new proxy admin.
   */
  function _setAdmin(address newAdmin) internal {
    bytes32 slot = ADMIN_SLOT;
    //solium-disable-next-line
    assembly {
      sstore(slot, newAdmin)
    }
  }

  /**
   * @dev Only fall back when the sender is not the admin.
   */
  function _willFallback() internal virtual override {
    require(msg.sender != _admin(), 'Cannot call fallback function from the proxy admin');
    super._willFallback();
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './BaseAdminUpgradeabilityProxy.sol';
import './InitializableUpgradeabilityProxy.sol';

/**
 * @title InitializableAdminUpgradeabilityProxy
 * @dev Extends from BaseAdminUpgradeabilityProxy with an initializer for
 * initializing the implementation, admin, and init data.
 */
contract InitializableAdminUpgradeabilityProxy is
  BaseAdminUpgradeabilityProxy,
  InitializableUpgradeabilityProxy
{
  /**
   * Contract initializer.
   * @param logic address of the initial implementation.
   * @param admin Address of the proxy administrator.
   * @param data Data to send as msg.data to the implementation to initialize the proxied contract.
   * It should include the signature and the parameters of the function to be called, as described in
   * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding
   * This parameter is optional, if no data is given the initialization call to proxied contract will
   */
  function initialize(
    address logic,
    address admin,
    bytes memory data
  ) public payable {
    require(_implementation() == address(0));
    InitializableUpgradeabilityProxy.initialize(logic, data);
    assert(ADMIN_SLOT == bytes32(uint256(keccak256('eip1967.proxy.admin')) - 1));
    _setAdmin(admin);
```

```
  }

  /**
   * @dev Only fall back when the sender is not the admin.
   */
  function _willFallback() internal override(BaseAdminUpgradeabilityProxy, Proxy) {
    BaseAdminUpgradeabilityProxy._willFallback();
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './Proxy.sol';
import '../contracts/Address.sol';

/**
 * @title BaseUpgradeabilityProxy
 * @dev This contract implements a proxy that allows to change the
 * implementation address to which it will delegate.
 * Such a change is called an implementation upgrade.
 */
contract BaseUpgradeabilityProxy is Proxy {
  /**
   * @dev Emitted when the implementation is upgraded.
   * @param implementation Address of the new implementation.
   */
  event Upgraded(address indexed implementation);

  /**
   * @dev Storage slot with the address of the current implementation.
   * This is the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1, and is
   * validated in the constructor.
   */
  bytes32 internal constant IMPLEMENTATION_SLOT =
    0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;

  /**
   * @dev Returns the current implementation.
   * @return impl Address of the current implementation
   */
  function _implementation() internal view override returns (address impl) {
    bytes32 slot = IMPLEMENTATION_SLOT;
    //solium-disable-next-line
    assembly {
      impl := sload(slot)
    }
  }

  /**
   * @dev Upgrades the proxy to a new implementation.
   * @param newImplementation Address of the new implementation.
   */
  function _upgradeTo(address newImplementation) internal {
    _setImplementation(newImplementation);
    emit Upgraded(newImplementation);
  }

  /**
   * @dev Sets the implementation address of the proxy.
   * @param newImplementation Address of the new implementation.
   */
  function _setImplementation(address newImplementation) internal {
    require(
      Address.isContract(newImplementation),
      'Cannot set a proxy implementation to a non-contract address'
    );
```

```
    bytes32 slot = IMPLEMENTATION_SLOT;

    //solium-disable-next-line
    assembly {
      sstore(slot, newImplementation)
    }
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity ^0.6.0;

/**
 * @title Proxy
 * @dev Implements delegation of calls to other contracts, with proper
 * forwarding of return values and bubbling of failures.
 * It defines a fallback function that delegates all calls to the address
 * returned by the abstract _implementation() internal function.
 */
abstract contract Proxy {
  /**
   * @dev Fallback function.
   * Implemented entirely in `_fallback`.
   */
  fallback() external payable {
    _fallback();
  }

  /**
   * @return The Address of the implementation.
   */
  function _implementation() internal view virtual returns (address);

  /**
   * @dev Delegates execution to an implementation contract.
   * This is a low level function that doesn't return to its internal call site.
   * It will return to the external caller whatever the implementation returns.
   * @param implementation Address to delegate.
   */
  function _delegate(address implementation) internal {
    //solium-disable-next-line
    assembly {
      // Copy msg.data. We take full control of memory in this inline assembly
      // block because it will not return to Solidity code. We overwrite the
      // Solidity scratch pad at memory position 0.
      calldatacopy(0, 0, calldatasize())

      // Call the implementation.
      // out and outsize are 0 because we don't know the size yet.
      let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

      // Copy the returned data.
      returndatacopy(0, 0, returndatasize())

      switch result
        // delegatecall returns 0 on error.
        case 0 {
          revert(0, returndatasize())
        }
        default {
          return(0, returndatasize())
        }
    }
  }

  /**
```

```
   * @dev Function that is run as the first thing in the fallback function.
   * Can be redefined in derived contracts to add functionality.
   * Redefinitions must call super._willFallback().
   */
  function _willFallback() internal virtual {}

  /**
   * @dev fallback implementation.
   * Extracted to enable manual triggering.
   */
  function _fallback() internal {
    _willFallback();
    _delegate(_implementation());
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './BaseUpgradeabilityProxy.sol';

/**
 * @title InitializableUpgradeabilityProxy
 * @dev Extends BaseUpgradeabilityProxy with an initializer for initializing
 * implementation and init data.
 */
contract InitializableUpgradeabilityProxy is BaseUpgradeabilityProxy {
  /**
   * @dev Contract initializer.
   * @param _logic Address of the initial implementation.
   * @param _data Data to send as msg.data to the implementation to initialize the proxied contract.
   * It should include the signature and the parameters of the function to be called, as described in
   * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding
   * This parameter is optional, if no data is given the initialization call to proxied contract will
   */
  function initialize(address _logic, bytes memory _data) public payable {
    require(_implementation() == address(0));
    assert(IMPLEMENTATION_SLOT == bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1));
    _setImplementation(_logic);
    if (_data.length > 0) {
      (bool success, ) = _logic.delegatecall(_data);
      require(success);
    }
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity >=0.4.24 <0.7.0;

/**
 * @title Initializable
 *
 * @dev Helper contract to support initializer functions. To use it, replace
 * the constructor with a function that has the `initializer` modifier.
 * WARNING: Unlike constructors, initializer functions must be manually
 * invoked. This applies both to deploying an Initializable contract, as well
 * as extending an Initializable contract via inheritance.
 * WARNING: When used with inheritance, manual care must be taken to not invoke
 * a parent initializer twice, or ensure that all initializers are idempotent,
 * because this is not dealt with automatically as with constructors.
 */
contract Initializable {
  /**
   * @dev Indicates that the contract has been initialized.
   */
  bool private initialized;

  /**
```

```
    * @dev Indicates that the contract is in the process of being initialized.
    */
  bool private initializing;

  /**
    * @dev Modifier to use in the initializer function of a contract.
    */
  modifier initializer() {
    require(
      initializing || isConstructor() || !initialized,
      'Contract instance has already been initialized'
    );

    bool isTopLevelCall = !initializing;
    if (isTopLevelCall) {
      initializing = true;
      initialized = true;
    }

    _;

    if (isTopLevelCall) {
      initializing = false;
    }
  }

  /// @dev Returns true if and only if the function is running in the constructor
  function isConstructor() private view returns (bool) {
    // extcodesize checks the size of the code stored in an address, and
    // address returns the current address. Since the code is still not
    // deployed when running a constructor, any checks on its code size will
    // yield zero, making it an effective way to detect if a contract is
    // under construction or not.
    uint256 cs;
    //solium-disable-next-line
    assembly {
      cs := extcodesize(address())
    }
    return cs == 0;
  }

  // Reserved storage space to allow for layout changes in the future.
  uint256[50] private ______gap;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './BaseAdminUpgradeabilityProxy.sol';

/**
 * @title AdminUpgradeabilityProxy
 * @dev Extends from BaseAdminUpgradeabilityProxy with a constructor for
 * initializing the implementation, admin, and init data.
 */
contract AdminUpgradeabilityProxy is BaseAdminUpgradeabilityProxy, UpgradeabilityProxy {
  /**
    * Contract constructor.
    * @param _logic address of the initial implementation.
    * @param _admin Address of the proxy administrator.
    * @param _data Data to send as msg.data to the implementation to initialize the proxied contract.
    * It should include the signature and the parameters of the function to be called, as described in
    * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding
    * This parameter is optional, if no data is given the initialization call to proxied contract will
    */
  constructor(
    address _logic,
```

```
    address _admin,
    bytes memory _data
  ) public payable UpgradeabilityProxy(_logic, _data) {
    assert(ADMIN_SLOT == bytes32(uint256(keccak256('eip1967.proxy.admin')) - 1));
    _setAdmin(_admin);
  }

  /**
   * @dev Only fall back when the sender is not the admin.
   */
  function _willFallback() internal override(BaseAdminUpgradeabilityProxy, Proxy) {
    BaseAdminUpgradeabilityProxy._willFallback();
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import './BaseUpgradeabilityProxy.sol';

/**
 * @title UpgradeabilityProxy
 * @dev Extends BaseUpgradeabilityProxy with a constructor for initializing
 * implementation and init data.
 */
contract UpgradeabilityProxy is BaseUpgradeabilityProxy {
  /**
   * @dev Contract constructor.
   * @param _logic Address of the initial implementation.
   * @param _data Data to send as msg.data to the implementation to initialize the proxied contract.
   * It should include the signature and the parameters of the function to be called, as described in
   * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding
   * This parameter is optional, if no data is given the initialization call to proxied contract will
   */
  constructor(address _logic, bytes memory _data) public payable {
    assert(IMPLEMENTATION_SLOT == bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1));
    _setImplementation(_logic);
    if (_data.length > 0) {
      (bool success, ) = _logic.delegatecall(_data);
      require(success);
    }
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IERC20} from './IERC20.sol';

interface IERC20Detailed is IERC20 {
  function name() external view returns (string memory);

  function symbol() external view returns (string memory);

  function decimals() external view returns (uint8);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {
  /**
   * @dev Returns the amount of tokens in existence.
   */
  function totalSupply() external view returns (uint256);
```

```
/**
 * @dev Returns the amount of tokens owned by `account`.
 */
function balanceOf(address account) external view returns (uint256);

/**
 * @dev Moves `amount` tokens from the caller's account to `recipient`.
 *
 * Returns a boolean value indicating whether the operation succeeded.
 *
 * Emits a {Transfer} event.
 */
function transfer(address recipient, uint256 amount) external returns (bool);

/**
 * @dev Returns the remaining number of tokens that `spender` will be
 * allowed to spend on behalf of `owner` through {transferFrom}. This is
 * zero by default.
 *
 * This value changes when {approve} or {transferFrom} are called.
 */
function allowance(address owner, address spender) external view returns (uint256);

/**
 * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
 *
 * Returns a boolean value indicating whether the operation succeeded.
 *
 * IMPORTANT: Beware that changing an allowance with this method brings the risk
 * that someone may use both the old and the new allowance by unfortunate
 * transaction ordering. One possible solution to mitigate this race
 * condition is to first reduce the spender's allowance to 0 and set the
 * desired value afterwards:
 * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
 *
 * Emits an {Approval} event.
 */
function approve(address spender, uint256 amount) external returns (bool);

/**
 * @dev Moves `amount` tokens from `sender` to `recipient` using the
 * allowance mechanism. `amount` is then deducted from the caller's
 * allowance.
 *
 * Returns a boolean value indicating whether the operation succeeded.
 *
 * Emits a {Transfer} event.
 */
function transferFrom(
  address sender,
  address recipient,
  uint256 amount
) external returns (bool);

/**
 * @dev Emitted when `value` tokens are moved from one account (`from`) to
 * another (`to`).
 *
 * Note that `value` may be zero.
 */
event Transfer(address indexed from, address indexed to, uint256 value);

/**
 * @dev Emitted when the allowance of a `spender` for an `owner` is set by
 * a call to {approve}. `value` is the new allowance.
 */
```

```
  event Approval(address indexed owner, address indexed spender, uint256 value);
}
// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import './Context.sol';
import './IERC20.sol';
import './SafeMath.sol';
import './Address.sol';

/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {
  using SafeMath for uint256;
  using Address for address;

  mapping(address => uint256) private _balances;

  mapping(address => mapping(address => uint256)) private _allowances;

  uint256 private _totalSupply;

  string private _name;
  string private _symbol;
  uint8 private _decimals;

  /**
   * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
   * a default value of 18.
   *
   * To select a different value for {decimals}, use {_setupDecimals}.
   *
   * All three of these values are immutable: they can only be set once during
   * construction.
   */
  constructor(string memory name, string memory symbol) public {
    _name = name;
    _symbol = symbol;
    _decimals = 18;
  }

  /**
   * @dev Returns the name of the token.
```

```
 */
function name() public view returns (string memory) {
  return _name;
}

/**
 * @dev Returns the symbol of the token, usually a shorter version of the
 * name.
 */
function symbol() public view returns (string memory) {
  return _symbol;
}

/**
 * @dev Returns the number of decimals used to get its user representation.
 * For example, if `decimals` equals `2`, a balance of `505` tokens should
 * be displayed to a user as `5,05` (`505 / 10 ** 2`).
 *
 * Tokens usually opt for a value of 18, imitating the relationship between
 * Ether and Wei. This is the value {ERC20} uses, unless {_setupDecimals} is
 * called.
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, including
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
function decimals() public view returns (uint8) {
  return _decimals;
}

/**
 * @dev See {IERC20-totalSupply}.
 */
function totalSupply() public view override returns (uint256) {
  return _totalSupply;
}

/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view override returns (uint256) {
  return _balances[account];
}

/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `recipient` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
  _transfer(_msgSender(), recipient, amount);
  return true;
}

/**
 * @dev See {IERC20-allowance}.
 */
function allowance(address owner, address spender)
  public
  view
  virtual
  override
  returns (uint256)
```

```
{
  return _allowances[owner][spender];
}

/**
 * @dev See {IERC20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 amount) public virtual override returns (bool) {
  _approve(_msgSender(), spender, amount);
  return true;
}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20};
 *
 * Requirements:
 * - `sender` and `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `amount`.
 * - the caller must have allowance for ``sender``'s tokens of at least
 * `amount`.
 */
function transferFrom(
  address sender,
  address recipient,
  uint256 amount
) public virtual override returns (bool) {
  _transfer(sender, recipient, amount);
  _approve(
    sender,
    _msgSender(),
    _allowances[sender][_msgSender()].sub(amount, 'ERC20: transfer amount exceeds allowance')
  );
  return true;
}

/**
 * @dev Atomically increases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
  _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
  return true;
}

/**
 * @dev Atomically decreases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
```

```
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue)
      public
      virtual
      returns (bool)
    {
      _approve(
        _msgSender(),
        spender,
        _allowances[_msgSender()][spender].sub(
          subtractedValue,
          'ERC20: decreased allowance below zero'
        )
      );
      return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(
      address sender,
      address recipient,
      uint256 amount
    ) internal virtual {
      require(sender != address(0), 'ERC20: transfer from the zero address');
      require(recipient != address(0), 'ERC20: transfer to the zero address');

      _beforeTokenTransfer(sender, recipient, amount);

      _balances[sender] = _balances[sender].sub(amount, 'ERC20: transfer amount exceeds balance');
      _balances[recipient] = _balances[recipient].add(amount);
      emit Transfer(sender, recipient, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal virtual {
      require(account != address(0), 'ERC20: mint to the zero address');

      _beforeTokenTransfer(address(0), account, amount);
```

```
    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
}

/**
 * @dev Destroys `amount` tokens from `account`, reducing the
 * total supply.
 *
 * Emits a {Transfer} event with `to` set to the zero address.
 *
 * Requirements
 *
 * - `account` cannot be the zero address.
 * - `account` must have at least `amount` tokens.
 */
function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), 'ERC20: burn from the zero address');

    _beforeTokenTransfer(account, address(0), amount);

    _balances[account] = _balances[account].sub(amount, 'ERC20: burn amount exceeds balance');
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
}

/**
 * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
 *
 * This is internal function is equivalent to `approve`, and can be used to
 * e.g. set automatic allowances for certain subsystems, etc.
 *
 * Emits an {Approval} event.
 *
 * Requirements:
 *
 * - `owner` cannot be the zero address.
 * - `spender` cannot be the zero address.
 */
function _approve(
    address owner,
    address spender,
    uint256 amount
) internal virtual {
    require(owner != address(0), 'ERC20: approve from the zero address');
    require(spender != address(0), 'ERC20: approve to the zero address');

    _allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}

/**
 * @dev Sets {decimals} to a value other than the default one of 18.
 *
 * WARNING: This function should only be called from the constructor. Most
 * applications that interact with token contracts will not expect
 * {decimals} to ever change, and may work incorrectly if it does.
 */
function _setupDecimals(uint8 decimals_) internal {
    _decimals = decimals_;
}

/**
 * @dev Hook that is called before any transfer of tokens. This includes
 * minting and burning.
 *
```

```
    * Calling conditions:
    *
    * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
    * will be to transferred to `to`.
    * - when `from` is zero, `amount` tokens will be minted for `to`.
    * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
    * - `from` and `to` are never both zero.
    *
    * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
    */
  function _beforeTokenTransfer(
    address from,
    address to,
    uint256 amount
  ) internal virtual {}
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @dev Collection of functions related to the address type
 */
library Address {
  /**
   * @dev Returns true if `account` is a contract.
   *
   * [IMPORTANT]
   * ====
   * It is unsafe to assume that an address for which this function returns
   * false is an externally-owned account (EOA) and not a contract.
   *
   * Among others, `isContract` will return false for the following
   * types of addresses:
   *
   *  - an externally-owned account
   *  - a contract in construction
   *  - an address where a contract will be created
   *  - an address where a contract lived, but was destroyed
   * ====
   */
  function isContract(address account) internal view returns (bool) {
    // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
    // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
    // for accounts without code, i.e. `keccak256('')`
    bytes32 codehash;
    bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
    // solhint-disable-next-line no-inline-assembly
    assembly {
      codehash := extcodehash(account)
    }
    return (codehash != accountHash && codehash != 0x0);
  }

  /**
   * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
   * `recipient`, forwarding all available gas and reverting on errors.
   *
   * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
   * of certain opcodes, possibly making contracts go over the 2300 gas limit
   * imposed by `transfer`, making them unable to receive funds via
   * `transfer`. {sendValue} removes this limitation.
   *
   * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
   *
   * IMPORTANT: because control is transferred to `recipient`, care must be
   * taken to not create reentrancy vulnerabilities. Consider using
```

```
    * {ReentrancyGuard} or the
    * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects-i
    */
  function sendValue(address payable recipient, uint256 amount) internal {
    require(address(this).balance >= amount, 'Address: insufficient balance');

    // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
    (bool success, ) = recipient.call{value: amount}('');
    require(success, 'Address: unable to send value, recipient may have reverted');
  }
}
// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import './Context.sol';

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {
  address private _owner;

  event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

  /**
   * @dev Initializes the contract setting the deployer as the initial owner.
   */
  constructor() internal {
    address msgSender = _msgSender();
    _owner = msgSender;
    emit OwnershipTransferred(address(0), msgSender);
  }

  /**
   * @dev Returns the address of the current owner.
   */
  function owner() public view returns (address) {
    return _owner;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(_owner == _msgSender(), 'Ownable: caller is not the owner');
    _;
  }

  /**
   * @dev Leaves the contract without owner. It will not be possible to call
   * `onlyOwner` functions anymore. Can only be called by the current owner.
   *
   * NOTE: Renouncing ownership will leave the contract without an owner,
   * thereby removing any functionality that is only available to the owner.
   */
  function renounceOwnership() public virtual onlyOwner {
```

```
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
  }

  /**
   * @dev Transfers ownership of the contract to a new account (`newOwner`).
   * Can only be called by the current owner.
   */
  function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), 'Ownable: new owner is the zero address');
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
  /**
   * @dev Returns the addition of two unsigned integers, reverting on
   * overflow.
   *
   * Counterpart to Solidity's `+` operator.
   *
   * Requirements:
   * - Addition cannot overflow.
   */
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a, 'SafeMath: addition overflow');

    return c;
  }

  /**
   * @dev Returns the subtraction of two unsigned integers, reverting on
   * overflow (when the result is negative).
   *
   * Counterpart to Solidity's `-` operator.
   *
   * Requirements:
   * - Subtraction cannot overflow.
   */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    return sub(a, b, 'SafeMath: subtraction overflow');
  }

  /**
   * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
   * overflow (when the result is negative).
   *
   * Counterpart to Solidity's `-` operator.
   *
```

```
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(
        uint256 a,
        uint256 b,
        string memory errorMessage
    ) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    /**
     * @dev Returns the multiplication of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `*` operator.
     *
     * Requirements:
     * - Multiplication cannot overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, 'SafeMath: multiplication overflow');

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, 'SafeMath: division by zero');
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(
        uint256 a,
        uint256 b,
```

```
    string memory errorMessage
  ) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, errorMessage);
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold

    return c;
  }

  /**
   * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
   * Reverts when dividing by zero.
   *
   * Counterpart to Solidity's `%` operator. This function uses a `revert`
   * opcode (which leaves remaining gas untouched) while Solidity uses an
   * invalid opcode to revert (consuming all remaining gas).
   *
   * Requirements:
   * - The divisor cannot be zero.
   */
  function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    return mod(a, b, 'SafeMath: modulo by zero');
  }

  /**
   * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
   * Reverts with custom message when dividing by zero.
   *
   * Counterpart to Solidity's `%` operator. This function uses a `revert`
   * opcode (which leaves remaining gas untouched) while Solidity uses an
   * invalid opcode to revert (consuming all remaining gas).
   *
   * Requirements:
   * - The divisor cannot be zero.
   */
  function mod(
    uint256 a,
    uint256 b,
    string memory errorMessage
  ) internal pure returns (uint256) {
    require(b != 0, errorMessage);
    return a % b;
  }
}
// SPDX-License-Identifier: MIT
pragma solidity 0.6.12;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
  function _msgSender() internal view virtual returns (address payable) {
    return msg.sender;
  }

  function _msgData() internal view virtual returns (bytes memory) {
    this; // silence state mutability warning without generating bytecode - see https://github.com/et
    return msg.data;
```

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

pragma solidity 0.6.12;

import {IERC20} from './IERC20.sol';
import {SafeMath} from './SafeMath.sol';
import {Address} from './Address.sol';

/**
 * @title SafeERC20
 * @dev Wrappers around ERC20 operations that throw on failure (when the token
 * contract returns false). Tokens that return no value (and instead revert or
 * throw on failure) are also supported, non-reverting calls are assumed to be
 * successful.
 * To use this library you can add a `using SafeERC20 for IERC20;` statement to your contract,
 * which allows you to call the safe operations as `token.safeTransfer(...)`, etc.
 */
library SafeERC20 {
  using SafeMath for uint256;
  using Address for address;

  function safeTransfer(
    IERC20 token,
    address to,
    uint256 value
  ) internal {
    callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
  }

  function safeTransferFrom(
    IERC20 token,
    address from,
    address to,
    uint256 value
  ) internal {
    callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
  }

  function safeApprove(
    IERC20 token,
    address spender,
    uint256 value
  ) internal {
    require(
      (value == 0) || (token.allowance(address(this), spender) == 0),
      'SafeERC20: approve from non-zero to non-zero allowance'
    );
    callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
  }

  function callOptionalReturn(IERC20 token, bytes memory data) private {
    require(address(token).isContract(), 'SafeERC20: call to non-contract');

    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory returndata) = address(token).call(data);
    require(success, 'SafeERC20: low-level call failed');

    if (returndata.length > 0) {
      // Return data is optional
      // solhint-disable-next-line max-line-length
      require(abi.decode(returndata, (bool)), 'SafeERC20: ERC20 operation did not succeed');
    }
  }
}
```

```
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';

/**
 * @title IFlashLoanReceiver interface
 * @notice Interface for the Aave fee IFlashLoanReceiver.
 * @author Aave
 * @dev implement this interface to develop a flashloan-compatible flashLoanReceiver contract
 **/
interface IFlashLoanReceiver {
  function executeOperation(
    address[] calldata assets,
    uint256[] calldata amounts,
    uint256[] calldata premiums,
    address initiator,
    bytes calldata params
  ) external returns (bool);

  function ADDRESSES_PROVIDER() external view returns (ILendingPoolAddressesProvider);

  function LENDING_POOL() external view returns (ILendingPool);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';
import {SafeERC20} from '../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {IFlashLoanReceiver} from '../interfaces/IFlashLoanReceiver.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';

abstract contract FlashLoanReceiverBase is IFlashLoanReceiver {
  using SafeERC20 for IERC20;
  using SafeMath for uint256;

  ILendingPoolAddressesProvider public immutable override ADDRESSES_PROVIDER;
  ILendingPool public immutable override LENDING_POOL;

  constructor(ILendingPoolAddressesProvider provider) public {
    ADDRESSES_PROVIDER = provider;
    LENDING_POOL = ILendingPool(provider.getLendingPool());
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

contract IERC20DetailedBytes {
  bytes32 public name;
  bytes32 public symbol;
  uint256 public decimals;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity >=0.6.2;

import './IUniswapV2Router01.sol';

interface IUniswapV2Router02 is IUniswapV2Router01 {
  function removeLiquidityETHSupportingFeeOnTransferTokens(
    address token,
    uint256 liquidity,
    uint256 amountTokenMin,
    uint256 amountETHMin,
```

```
    address to,
    uint256 deadline
  ) external returns (uint256 amountETH);

  function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
    address token,
    uint256 liquidity,
    uint256 amountTokenMin,
    uint256 amountETHMin,
    address to,
    uint256 deadline,
    bool approveMax,
    uint8 v,
    bytes32 r,
    bytes32 s
  ) external returns (uint256 amountETH);

  function swapExactTokensForTokensSupportingFeeOnTransferTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external;

  function swapExactETHForTokensSupportingFeeOnTransferTokens(
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external payable;

  function swapExactTokensForETHSupportingFeeOnTransferTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity >=0.6.2;

interface IUniswapV2Router01 {
  function factory() external pure returns (address);

  function WETH() external pure returns (address);

  function addLiquidity(
    address tokenA,
    address tokenB,
    uint256 amountADesired,
    uint256 amountBDesired,
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline
  )
    external
    returns (
      uint256 amountA,
      uint256 amountB,
      uint256 liquidity
    );

  function addLiquidityETH(
```

```
    address token,
    uint256 amountTokenDesired,
    uint256 amountTokenMin,
    uint256 amountETHMin,
    address to,
    uint256 deadline
  )
    external
    payable
    returns (
      uint256 amountToken,
      uint256 amountETH,
      uint256 liquidity
    );

  function removeLiquidity(
    address tokenA,
    address tokenB,
    uint256 liquidity,
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline
  ) external returns (uint256 amountA, uint256 amountB);

  function removeLiquidityETH(
    address token,
    uint256 liquidity,
    uint256 amountTokenMin,
    uint256 amountETHMin,
    address to,
    uint256 deadline
  ) external returns (uint256 amountToken, uint256 amountETH);

  function removeLiquidityWithPermit(
    address tokenA,
    address tokenB,
    uint256 liquidity,
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline,
    bool approveMax,
    uint8 v,
    bytes32 r,
    bytes32 s
  ) external returns (uint256 amountA, uint256 amountB);

  function removeLiquidityETHWithPermit(
    address token,
    uint256 liquidity,
    uint256 amountTokenMin,
    uint256 amountETHMin,
    address to,
    uint256 deadline,
    bool approveMax,
    uint8 v,
    bytes32 r,
    bytes32 s
  ) external returns (uint256 amountToken, uint256 amountETH);

  function swapExactTokensForTokens(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
```

```
    uint256 deadline
  ) external returns (uint256[] memory amounts);

  function swapTokensForExactTokens(
    uint256 amountOut,
    uint256 amountInMax,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external returns (uint256[] memory amounts);

  function swapExactETHForTokens(
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external payable returns (uint256[] memory amounts);

  function swapTokensForExactETH(
    uint256 amountOut,
    uint256 amountInMax,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external returns (uint256[] memory amounts);

  function swapExactTokensForETH(
    uint256 amountIn,
    uint256 amountOutMin,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external returns (uint256[] memory amounts);

  function swapETHForExactTokens(
    uint256 amountOut,
    address[] calldata path,
    address to,
    uint256 deadline
  ) external payable returns (uint256[] memory amounts);

  function quote(
    uint256 amountA,
    uint256 reserveA,
    uint256 reserveB
  ) external pure returns (uint256 amountB);

  function getAmountOut(
    uint256 amountIn,
    uint256 reserveIn,
    uint256 reserveOut
  ) external pure returns (uint256 amountOut);

  function getAmountIn(
    uint256 amountOut,
    uint256 reserveIn,
    uint256 reserveOut
  ) external pure returns (uint256 amountIn);

  function getAmountsOut(uint256 amountIn, address[] calldata path)
    external
    view
    returns (uint256[] memory amounts);

  function getAmountsIn(uint256 amountOut, address[] calldata path)
    external
```

```
    view
    returns (uint256[] memory amounts);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IWETH {
  function deposit() external payable;

  function withdraw(uint256) external;

  function approve(address guy, uint256 wad) external returns (bool);

  function transferFrom(
    address src,
    address dst,
    uint256 wad
  ) external returns (bool);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';

interface IUiPoolDataProvider {
  struct AggregatedReserveData {
    address underlyingAsset;
    string name;
    string symbol;
    uint256 decimals;
    uint256 baseLTVasCollateral;
    uint256 reserveLiquidationThreshold;
    uint256 reserveLiquidationBonus;
    uint256 reserveFactor;
    bool usageAsCollateralEnabled;
    bool borrowingEnabled;
    bool stableBorrowRateEnabled;
    bool isActive;
    bool isFrozen;
    // base data
    uint128 liquidityIndex;
    uint128 variableBorrowIndex;
    uint128 liquidityRate;
    uint128 variableBorrowRate;
    uint128 stableBorrowRate;
    uint40 lastUpdateTimestamp;
    address aTokenAddress;
    address stableDebtTokenAddress;
    address variableDebtTokenAddress;
    address interestRateStrategyAddress;
    //
    uint256 availableLiquidity;
    uint256 totalPrincipalStableDebt;
    uint256 averageStableRate;
    uint256 stableDebtLastUpdateTimestamp;
    uint256 totalScaledVariableDebt;
    uint256 priceInEth;
    uint256 variableRateSlope1;
    uint256 variableRateSlope2;
    uint256 stableRateSlope1;
    uint256 stableRateSlope2;
  }
  //
  //  struct ReserveData {
  //    uint256 averageStableBorrowRate;
```

```
  //    uint256 totalLiquidity;
  //  }

  struct UserReserveData {
    address underlyingAsset;
    uint256 scaledATokenBalance;
    bool usageAsCollateralEnabledOnUser;
    uint256 stableBorrowRate;
    uint256 scaledVariableDebt;
    uint256 principalStableDebt;
    uint256 stableBorrowLastUpdateTimestamp;
  }

  //
  //  struct ATokenSupplyData {
  //    string name;
  //    string symbol;
  //    uint8 decimals;
  //    uint256 totalSupply;
  //    address aTokenAddress;
  //  }

  function getReservesData(ILendingPoolAddressesProvider provider, address user)
    external
    view
    returns (
      AggregatedReserveData[] memory,
      UserReserveData[] memory,
      uint256
    );

  //  function getUserReservesData(ILendingPoolAddressesProvider provider, address user)
  //    external
  //    view
  //    returns (UserReserveData[] memory);
  //
  //  function getAllATokenSupply(ILendingPoolAddressesProvider provider)
  //    external
  //    view
  //    returns (ATokenSupplyData[] memory);
  //
  //  function getATokenSupply(address[] calldata aTokens)
  //    external
  //    view
  //    returns (ATokenSupplyData[] memory);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IWETHGateway {
  function depositETH(
    address lendingPool,
    address onBehalfOf,
    uint16 referralCode
  ) external payable;

  function withdrawETH(
    address lendingPool,
    uint256 amount,
    address onBehalfOf
  ) external;

  function repayETH(
    address lendingPool,
    uint256 amount,
    uint256 rateMode,
```

```
    address onBehalfOf
  ) external payable;

  function borrowETH(
    address lendingPool,
    uint256 amount,
    uint256 interesRateMode,
    uint16 referralCode
  ) external;
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {IERC20Detailed} from '../dependencies/openzeppelin/contracts/IERC20Detailed.sol';
import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {IUiPoolDataProvider} from './interfaces/IUiPoolDataProvider.sol';
import {ILendingPool} from '../interfaces/ILendingPool.sol';
import {IPriceOracleGetter} from '../interfaces/IPriceOracleGetter.sol';
import {IAToken} from '../interfaces/IAToken.sol';
import {IVariableDebtToken} from '../interfaces/IVariableDebtToken.sol';
import {IStableDebtToken} from '../interfaces/IStableDebtToken.sol';
import {WadRayMath} from '../protocol/libraries/math/WadRayMath.sol';
import {ReserveConfiguration} from '../protocol/libraries/configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../protocol/libraries/configuration/UserConfiguration.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';
import {
  DefaultReserveInterestRateStrategy
} from '../protocol/lendingpool/DefaultReserveInterestRateStrategy.sol';

contract UiPoolDataProvider is IUiPoolDataProvider {
  using WadRayMath for uint256;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  address public constant MOCK_USD_ADDRESS = 0x10F7Fc1F91Ba351f9C629c5947AD69bD03C05b96;

  function getInterestRateStrategySlopes(DefaultReserveInterestRateStrategy interestRateStrategy)
    internal
    view
    returns (
      uint256,
      uint256,
      uint256,
      uint256
    )
  {
    return (
      interestRateStrategy.variableRateSlope1(),
      interestRateStrategy.variableRateSlope2(),
      interestRateStrategy.stableRateSlope1(),
      interestRateStrategy.stableRateSlope2()
    );
  }

  function getReservesData(ILendingPoolAddressesProvider provider, address user)
    external
    view
    override
    returns (
      AggregatedReserveData[] memory,
      UserReserveData[] memory,
      uint256
    )
  {
    ILendingPool lendingPool = ILendingPool(provider.getLendingPool());
```

```
    IPriceOracleGetter oracle = IPriceOracleGetter(provider.getPriceOracle());
    address[] memory reserves = lendingPool.getReservesList();
    DataTypes.UserConfigurationMap memory userConfig = lendingPool.getUserConfiguration(user);

    AggregatedReserveData[] memory reservesData = new AggregatedReserveData[](reserves.length);
    UserReserveData[] memory userReservesData =
      new UserReserveData[](user != address(0) ? reserves.length : 0);

    for (uint256 i = 0; i < reserves.length; i++) {
      AggregatedReserveData memory reserveData = reservesData[i];
      reserveData.underlyingAsset = reserves[i];

      // reserve current state
      DataTypes.ReserveData memory baseData =
        lendingPool.getReserveData(reserveData.underlyingAsset);
      reserveData.liquidityIndex = baseData.liquidityIndex;
      reserveData.variableBorrowIndex = baseData.variableBorrowIndex;
      reserveData.liquidityRate = baseData.currentLiquidityRate;
      reserveData.variableBorrowRate = baseData.currentVariableBorrowRate;
      reserveData.stableBorrowRate = baseData.currentStableBorrowRate;
      reserveData.lastUpdateTimestamp = baseData.lastUpdateTimestamp;
      reserveData.aTokenAddress = baseData.aTokenAddress;
      reserveData.stableDebtTokenAddress = baseData.stableDebtTokenAddress;
      reserveData.variableDebtTokenAddress = baseData.variableDebtTokenAddress;
      reserveData.interestRateStrategyAddress = baseData.interestRateStrategyAddress;
      reserveData.priceInEth = oracle.getAssetPrice(reserveData.underlyingAsset);

      reserveData.availableLiquidity = IERC20Detailed(reserveData.underlyingAsset).balanceOf(
        reserveData.aTokenAddress
      );
      (
        reserveData.totalPrincipalStableDebt,
        ,
        reserveData.averageStableRate,
        reserveData.stableDebtLastUpdateTimestamp
      ) = IStableDebtToken(reserveData.stableDebtTokenAddress).getSupplyData();
      reserveData.totalScaledVariableDebt = IVariableDebtToken(reserveData.variableDebtTokenAddress)
        .scaledTotalSupply();

      // reserve configuration

      // we're getting this info from the aToken, because some of assets can be not compliant with ET
      reserveData.symbol = IERC20Detailed(reserveData.aTokenAddress).symbol();
      reserveData.name = '';

      (
        reserveData.baseLTVasCollateral,
        reserveData.reserveLiquidationThreshold,
        reserveData.reserveLiquidationBonus,
        reserveData.decimals,
        reserveData.reserveFactor
      ) = baseData.configuration.getParamsMemory();
      (
        reserveData.isActive,
        reserveData.isFrozen,
        reserveData.borrowingEnabled,
        reserveData.stableBorrowRateEnabled
      ) = baseData.configuration.getFlagsMemory();
      reserveData.usageAsCollateralEnabled = reserveData.baseLTVasCollateral != 0;
      (
        reserveData.variableRateSlope1,
        reserveData.variableRateSlope2,
        reserveData.stableRateSlope1,
        reserveData.stableRateSlope2
      ) = getInterestRateStrategySlopes(
        DefaultReserveInterestRateStrategy(reserveData.interestRateStrategyAddress)
```

```
      );

      if (user != address(0)) {
        // user reserve data
        userReservesData[i].underlyingAsset = reserveData.underlyingAsset;
        userReservesData[i].scaledATokenBalance = IAToken(reserveData.aTokenAddress)
          .scaledBalanceOf(user);
        userReservesData[i].usageAsCollateralEnabledOnUser = userConfig.isUsingAsCollateral(i);

        if (userConfig.isBorrowing(i)) {
          userReservesData[i].scaledVariableDebt = IVariableDebtToken(
            reserveData
              .variableDebtTokenAddress
          )
            .scaledBalanceOf(user);
          userReservesData[i].principalStableDebt = IStableDebtToken(
            reserveData
              .stableDebtTokenAddress
          )
            .principalBalanceOf(user);
          if (userReservesData[i].principalStableDebt != 0) {
            userReservesData[i].stableBorrowRate = IStableDebtToken(
              reserveData
                .stableDebtTokenAddress
            )
              .getUserStableRate(user);
            userReservesData[i].stableBorrowLastUpdateTimestamp = IStableDebtToken(
              reserveData
                .stableDebtTokenAddress
            )
              .getUserLastUpdated(user);
          }
        }
      }
    }
    return (reservesData, userReservesData, oracle.getAssetPrice(MOCK_USD_ADDRESS));
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {IERC20Detailed} from '../dependencies/openzeppelin/contracts/IERC20Detailed.sol';
import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingPool} from '../interfaces/ILendingPool.sol';
import {IStableDebtToken} from '../interfaces/IStableDebtToken.sol';
import {IVariableDebtToken} from '../interfaces/IVariableDebtToken.sol';
import {ReserveConfiguration} from '../protocol/libraries/configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../protocol/libraries/configuration/UserConfiguration.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';

contract AaveProtocolDataProvider {
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  address constant MKR = 0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2;
  address constant ETH = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;

  struct TokenData {
    string symbol;
    address tokenAddress;
  }

  ILendingPoolAddressesProvider public immutable ADDRESSES_PROVIDER;

  constructor(ILendingPoolAddressesProvider addressesProvider) public {
```

```
    ADDRESSES_PROVIDER = addressesProvider;
  }

  function getAllReservesTokens() external view returns (TokenData[] memory) {
    ILendingPool pool = ILendingPool(ADDRESSES_PROVIDER.getLendingPool());
    address[] memory reserves = pool.getReservesList();
    TokenData[] memory reservesTokens = new TokenData[](reserves.length);
    for (uint256 i = 0; i < reserves.length; i++) {
      if (reserves[i] == MKR) {
        reservesTokens[i] = TokenData({symbol: 'MKR', tokenAddress: reserves[i]});
        continue;
      }
      if (reserves[i] == ETH) {
        reservesTokens[i] = TokenData({symbol: 'ETH', tokenAddress: reserves[i]});
        continue;
      }
      reservesTokens[i] = TokenData({
        symbol: IERC20Detailed(reserves[i]).symbol(),
        tokenAddress: reserves[i]
      });
    }
    return reservesTokens;
  }

  function getAllATokens() external view returns (TokenData[] memory) {
    ILendingPool pool = ILendingPool(ADDRESSES_PROVIDER.getLendingPool());
    address[] memory reserves = pool.getReservesList();
    TokenData[] memory aTokens = new TokenData[](reserves.length);
    for (uint256 i = 0; i < reserves.length; i++) {
      DataTypes.ReserveData memory reserveData = pool.getReserveData(reserves[i]);
      aTokens[i] = TokenData({
        symbol: IERC20Detailed(reserveData.aTokenAddress).symbol(),
        tokenAddress: reserveData.aTokenAddress
      });
    }
    return aTokens;
  }

  function getReserveConfigurationData(address asset)
    external
    view
    returns (
      uint256 decimals,
      uint256 ltv,
      uint256 liquidationThreshold,
      uint256 liquidationBonus,
      uint256 reserveFactor,
      bool usageAsCollateralEnabled,
      bool borrowingEnabled,
      bool stableBorrowRateEnabled,
      bool isActive,
      bool isFrozen
    )
  {
    DataTypes.ReserveConfigurationMap memory configuration =
      ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).getConfiguration(asset);

    (ltv, liquidationThreshold, liquidationBonus, decimals, reserveFactor) = configuration
      .getParamsMemory();

    (isActive, isFrozen, borrowingEnabled, stableBorrowRateEnabled) = configuration
      .getFlagsMemory();

    usageAsCollateralEnabled = liquidationThreshold > 0;
  }
```

```
  function getReserveData(address asset)
    external
    view
    returns (
      uint256 availableLiquidity,
      uint256 totalStableDebt,
      uint256 totalVariableDebt,
      uint256 liquidityRate,
      uint256 variableBorrowRate,
      uint256 stableBorrowRate,
      uint256 averageStableBorrowRate,
      uint256 liquidityIndex,
      uint256 variableBorrowIndex,
      uint40 lastUpdateTimestamp
    )
  {
    DataTypes.ReserveData memory reserve =
      ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).getReserveData(asset);

    return (
      IERC20Detailed(asset).balanceOf(reserve.aTokenAddress),
      IERC20Detailed(reserve.stableDebtTokenAddress).totalSupply(),
      IERC20Detailed(reserve.variableDebtTokenAddress).totalSupply(),
      reserve.currentLiquidityRate,
      reserve.currentVariableBorrowRate,
      reserve.currentStableBorrowRate,
      IStableDebtToken(reserve.stableDebtTokenAddress).getAverageStableRate(),
      reserve.liquidityIndex,
      reserve.variableBorrowIndex,
      reserve.lastUpdateTimestamp
    );
  }

  function getUserReserveData(address asset, address user)
    external
    view
    returns (
      uint256 currentATokenBalance,
      uint256 currentStableDebt,
      uint256 currentVariableDebt,
      uint256 principalStableDebt,
      uint256 scaledVariableDebt,
      uint256 stableBorrowRate,
      uint256 liquidityRate,
      uint40 stableRateLastUpdated,
      bool usageAsCollateralEnabled
    )
  {
    DataTypes.ReserveData memory reserve =
      ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).getReserveData(asset);

    DataTypes.UserConfigurationMap memory userConfig =
      ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).getUserConfiguration(user);

    currentATokenBalance = IERC20Detailed(reserve.aTokenAddress).balanceOf(user);
    currentVariableDebt = IERC20Detailed(reserve.variableDebtTokenAddress).balanceOf(user);
    currentStableDebt = IERC20Detailed(reserve.stableDebtTokenAddress).balanceOf(user);
    principalStableDebt = IStableDebtToken(reserve.stableDebtTokenAddress).principalBalanceOf(user);
    scaledVariableDebt = IVariableDebtToken(reserve.variableDebtTokenAddress).scaledBalanceOf(user);
    liquidityRate = reserve.currentLiquidityRate;
    stableBorrowRate = IStableDebtToken(reserve.stableDebtTokenAddress).getUserStableRate(user);
    stableRateLastUpdated = IStableDebtToken(reserve.stableDebtTokenAddress).getUserLastUpdated(
      user
    );
    usageAsCollateralEnabled = userConfig.isUsingAsCollateral(reserve.id);
  }
```

```
  function getReserveTokensAddresses(address asset)
    external
    view
    returns (
      address aTokenAddress,
      address stableDebtTokenAddress,
      address variableDebtTokenAddress
    )
  {
    DataTypes.ReserveData memory reserve =
      ILendingPool(ADDRESSES_PROVIDER.getLendingPool()).getReserveData(asset);

    return (
      reserve.aTokenAddress,
      reserve.stableDebtTokenAddress,
      reserve.variableDebtTokenAddress
    );
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {Ownable} from '../dependencies/openzeppelin/contracts/Ownable.sol';
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';

import {IPriceOracleGetter} from '../interfaces/IPriceOracleGetter.sol';
import {IChainlinkAggregator} from '../interfaces/IChainlinkAggregator.sol';
import {SafeERC20} from '../dependencies/openzeppelin/contracts/SafeERC20.sol';

/// @title AaveOracle
/// @author Aave
/// @notice Proxy smart contract to get the price of an asset from a price source, with Chainlink Agg
///         smart contracts as primary option
/// - If the returned price by a Chainlink aggregator is <= 0, the call is forwarded to a fallbackOra
/// - Owned by the Aave governance system, allowed to add sources for assets, replace them
///   and change the fallbackOracle
contract AaveOracle is IPriceOracleGetter, Ownable {
  using SafeERC20 for IERC20;

  event AssetSourceUpdated(address indexed asset, address indexed source);
  event FallbackOracleUpdated(address indexed fallbackOracle);

  mapping(address => IChainlinkAggregator) private assetsSources;
  IPriceOracleGetter private _fallbackOracle;

  /// @notice Constructor
  /// @param assets The addresses of the assets
  /// @param sources The address of the source of each asset
  /// @param fallbackOracle The address of the fallback oracle to use if the data of an
  ///        aggregator is not consistent
  constructor(
    address[] memory assets,
    address[] memory sources,
    address fallbackOracle
  ) public {
    _setFallbackOracle(fallbackOracle);
    _setAssetsSources(assets, sources);
  }

  /// @notice External function called by the Aave governance to set or replace sources of assets
  /// @param assets The addresses of the assets
  /// @param sources The address of the source of each asset
  function setAssetSources(address[] calldata assets, address[] calldata sources)
    external
    onlyOwner
```

```
{
  _setAssetsSources(assets, sources);
}

/// @notice Sets the fallbackOracle
/// - Callable only by the Aave governance
/// @param fallbackOracle The address of the fallbackOracle
function setFallbackOracle(address fallbackOracle) external onlyOwner {
  _setFallbackOracle(fallbackOracle);
}

/// @notice Internal function to set the sources for each asset
/// @param assets The addresses of the assets
/// @param sources The address of the source of each asset
function _setAssetsSources(address[] memory assets, address[] memory sources) internal {
  require(assets.length == sources.length, 'INCONSISTENT_PARAMS_LENGTH');
  for (uint256 i = 0; i < assets.length; i++) {
    assetsSources[assets[i]] = IChainlinkAggregator(sources[i]);
    emit AssetSourceUpdated(assets[i], sources[i]);
  }
}

/// @notice Internal function to set the fallbackOracle
/// @param fallbackOracle The address of the fallbackOracle
function _setFallbackOracle(address fallbackOracle) internal {
  _fallbackOracle = IPriceOracleGetter(fallbackOracle);
  emit FallbackOracleUpdated(fallbackOracle);
}

/// @notice Gets an asset price by address
/// @param asset The asset address
function getAssetPrice(address asset) public view override returns (uint256) {
  IChainlinkAggregator source = assetsSources[asset];

  if (address(source) == address(0)) {
    return _fallbackOracle.getAssetPrice(asset);
  } else {
    int256 price = IChainlinkAggregator(source).latestAnswer();
    if (price > 0) {
      return uint256(price);
    } else {
      return _fallbackOracle.getAssetPrice(asset);
    }
  }
}

/// @notice Gets a list of prices from a list of assets addresses
/// @param assets The list of assets addresses
function getAssetsPrices(address[] calldata assets) external view returns (uint256[] memory) {
  uint256[] memory prices = new uint256[](assets.length);
  for (uint256 i = 0; i < assets.length; i++) {
    prices[i] = getAssetPrice(assets[i]);
  }
  return prices;
}

/// @notice Gets the address of the source for an asset address
/// @param asset The address of the asset
/// @return address The address of the source
function getSourceOfAsset(address asset) external view returns (address) {
  return address(assetsSources[asset]);
}

/// @notice Gets the address of the fallback oracle
/// @return address The addres of the fallback oracle
function getFallbackOracle() external view returns (address) {
```

```
    return address(_fallbackOracle);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

import {Ownable} from '../dependencies/openzeppelin/contracts/Ownable.sol';
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';
import {IWETH} from './interfaces/IWETH.sol';
import {IWETHGateway} from './interfaces/IWETHGateway.sol';
import {ILendingPool} from '../interfaces/ILendingPool.sol';
import {IAToken} from '../interfaces/IAToken.sol';
import {ReserveConfiguration} from '../protocol/libraries/configuration/ReserveConfiguration.sol';
import {UserConfiguration} from '../protocol/libraries/configuration/UserConfiguration.sol';
import {Helpers} from '../protocol/libraries/helpers/Helpers.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';

contract WETHGateway is IWETHGateway, Ownable {
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
  using UserConfiguration for DataTypes.UserConfigurationMap;

  IWETH internal immutable WETH;

  /**
   * @dev Sets the WETH address and the LendingPoolAddressesProvider address. Infinite approves lendi
   * @param weth Address of the Wrapped Ether contract
   **/
  constructor(address weth) public {
    WETH = IWETH(weth);
  }

  function authorizeLendingPool(address lendingPool) external onlyOwner {
    WETH.approve(lendingPool, uint256(-1));
  }

  /**
   * @dev deposits WETH into the reserve, using native ETH. A corresponding amount of the overlying a
   * is minted.
   * @param lendingPool address of the targeted underlying lending pool
   * @param onBehalfOf address of the user who will receive the aTokens representing the deposit
   * @param referralCode integrators are assigned a referral code and can potentially receive rewards
   **/
  function depositETH(
    address lendingPool,
    address onBehalfOf,
    uint16 referralCode
  ) external payable override {
    WETH.deposit{value: msg.value}();
    ILendingPool(lendingPool).deposit(address(WETH), msg.value, onBehalfOf, referralCode);
  }

  /**
   * @dev withdraws the WETH _reserves of msg.sender.
   * @param lendingPool address of the targeted underlying lending pool
   * @param amount amount of aWETH to withdraw and receive native ETH
   * @param to address of the user who will receive native ETH
   */
  function withdrawETH(
    address lendingPool,
    uint256 amount,
    address to
  ) external override {
    IAToken aWETH = IAToken(ILendingPool(lendingPool).getReserveData(address(WETH)).aTokenAddress);
    uint256 userBalance = aWETH.balanceOf(msg.sender);
    uint256 amountToWithdraw = amount;
```

```
    // if amount is equal to uint(-1), the user wants to redeem everything
    if (amount == type(uint256).max) {
      amountToWithdraw = userBalance;
    }
    aWETH.transferFrom(msg.sender, address(this), amountToWithdraw);
    ILendingPool(lendingPool).withdraw(address(WETH), amountToWithdraw, address(this));
    WETH.withdraw(amountToWithdraw);
    _safeTransferETH(to, amountToWithdraw);
  }

  /**
   * @dev repays a borrow on the WETH reserve, for the specified amount (or for the whole amount, if
   * @param lendingPool address of the targeted underlying lending pool
   * @param amount the amount to repay, or uint256(-1) if the user wants to repay everything
   * @param rateMode the rate mode to repay
   * @param onBehalfOf the address for which msg.sender is repaying
   */
  function repayETH(
    address lendingPool,
    uint256 amount,
    uint256 rateMode,
    address onBehalfOf
  ) external payable override {
    (uint256 stableDebt, uint256 variableDebt) =
      Helpers.getUserCurrentDebtMemory(
        onBehalfOf,
        ILendingPool(lendingPool).getReserveData(address(WETH))
      );

    uint256 paybackAmount =
      DataTypes.InterestRateMode(rateMode) == DataTypes.InterestRateMode.STABLE
        ? stableDebt
        : variableDebt;

    if (amount < paybackAmount) {
      paybackAmount = amount;
    }
    require(msg.value >= paybackAmount, 'msg.value is less than repayment amount');
    WETH.deposit{value: paybackAmount}();
    ILendingPool(lendingPool).repay(address(WETH), msg.value, rateMode, onBehalfOf);

    // refund remaining dust eth
    if (msg.value > paybackAmount) _safeTransferETH(msg.sender, msg.value - paybackAmount);
  }

  /**
   * @dev borrow WETH, unwraps to ETH and send both the ETH and DebtTokens to msg.sender, via `approv
   * @param lendingPool address of the targeted underlying lending pool
   * @param amount the amount of ETH to borrow
   * @param interesRateMode the interest rate mode
   * @param referralCode integrators are assigned a referral code and can potentially receive rewards
   */
  function borrowETH(
    address lendingPool,
    uint256 amount,
    uint256 interesRateMode,
    uint16 referralCode
  ) external override {
    ILendingPool(lendingPool).borrow(
      address(WETH),
      amount,
      interesRateMode,
      referralCode,
      msg.sender
    );
```

```
    WETH.withdraw(amount);
    _safeTransferETH(msg.sender, amount);
  }

  /**
   * @dev transfer ETH to an address, revert if it fails.
   * @param to recipient of the transfer
   * @param value the amount to send
   */
  function _safeTransferETH(address to, uint256 value) internal {
    (bool success, ) = to.call{value: value}(new bytes(0));
    require(success, 'ETH_TRANSFER_FAILED');
  }

  /**
   * @dev transfer ERC20 from the utility contract, for ERC20 recovery in case of stuck tokens due
   * direct transfers to the contract address.
   * @param token token to transfer
   * @param to recipient of the transfer
   * @param amount amount to send
   */
  function emergencyTokenTransfer(
    address token,
    address to,
    uint256 amount
  ) external onlyOwner {
    IERC20(token).transfer(to, amount);
  }

  /**
   * @dev transfer native Ether from the utility contract, for native Ether recovery in case of stuck
   * due selfdestructs or transfer ether to pre-computated contract address before deployment.
   * @param to recipient of the transfer
   * @param amount amount to send
   */
  function emergencyEtherTransfer(address to, uint256 amount) external onlyOwner {
    _safeTransferETH(to, amount);
  }

  /**
   * @dev Get WETH address used by WETHGateway
   */
  function getWETHAddress() external view returns (address) {
    return address(WETH);
  }

  /**
   * @dev Only WETH contract is allowed to transfer ETH here. Prevent other addresses to send Ether t
   */
  receive() external payable {
    require(msg.sender == address(WETH), 'Receive not allowed');
  }

  /**
   * @dev Revert fallback calls
   */
  fallback() external payable {
    revert('Fallback not allowed');
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

pragma experimental ABIEncoderV2;

import {Address} from '../dependencies/openzeppelin/contracts/Address.sol';
```

```
import {IERC20} from '../dependencies/openzeppelin/contracts/IERC20.sol';

import {ILendingPoolAddressesProvider} from '../interfaces/ILendingPoolAddressesProvider.sol';
import {ILendingPool} from '../interfaces/ILendingPool.sol';
import {SafeERC20} from '../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {ReserveConfiguration} from '../protocol/libraries/configuration/ReserveConfiguration.sol';
import {DataTypes} from '../protocol/libraries/types/DataTypes.sol';

/**
 * @title WalletBalanceProvider contract
 * @author Aave, influenced by https://github.com/wbobeirne/eth-balance-checker/blob/master/contracts
 * @notice Implements a logic of getting multiple tokens balance for one user address
 * @dev NOTE: THIS CONTRACT IS NOT USED WITHIN THE AAVE PROTOCOL. It's an accessory contract used to
 * towards the blockchain from the Aave backend.
 **/
contract WalletBalanceProvider {
  using Address for address payable;
  using Address for address;
  using SafeERC20 for IERC20;
  using ReserveConfiguration for DataTypes.ReserveConfigurationMap;

  address constant MOCK_ETH_ADDRESS = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;

  /**
    @dev Fallback function, don't accept any ETH
    **/
  receive() external payable {
    //only contracts can send ETH to the core
    require(msg.sender.isContract(), '22');
  }

  /**
    @dev Check the token balance of a wallet in a token contract

    Returns the balance of the token for user. Avoids possible errors:
      - return 0 on non-contract address
    **/
  function balanceOf(address user, address token) public view returns (uint256) {
    if (token == MOCK_ETH_ADDRESS) {
      return user.balance; // ETH balance
      // check if token is actually a contract
    } else if (token.isContract()) {
      return IERC20(token).balanceOf(user);
    }
    revert('INVALID_TOKEN');
  }

  /**
   * @notice Fetches, for a list of _users and _tokens (ETH included with mock address), the balances
   * @param users The list of users
   * @param tokens The list of tokens
   * @return And array with the concatenation of, for each user, his/her balances
   **/
  function batchBalanceOf(address[] calldata users, address[] calldata tokens)
    external
    view
    returns (uint256[] memory)
  {
    uint256[] memory balances = new uint256[](users.length * tokens.length);

    for (uint256 i = 0; i < users.length; i++) {
      for (uint256 j = 0; j < tokens.length; j++) {
        balances[i * tokens.length + j] = balanceOf(users[i], tokens[j]);
      }
    }
```

```
    return balances;
  }

  /**
    @dev provides balances of user wallet for all reserves available on the pool
    */
  function getUserWalletBalances(address provider, address user)
    external
    view
    returns (address[] memory, uint256[] memory)
  {
    ILendingPool pool = ILendingPool(ILendingPoolAddressesProvider(provider).getLendingPool());

    address[] memory reserves = pool.getReservesList();
    address[] memory reservesWithEth = new address[](reserves.length + 1);
    for (uint256 i = 0; i < reserves.length; i++) {
      reservesWithEth[i] = reserves[i];
    }
    reservesWithEth[reserves.length] = MOCK_ETH_ADDRESS;

    uint256[] memory balances = new uint256[](reservesWithEth.length);

    for (uint256 j = 0; j < reserves.length; j++) {
      DataTypes.ReserveConfigurationMap memory configuration =
        pool.getConfiguration(reservesWithEth[j]);

      (bool isActive, , , ) = configuration.getFlagsMemory();

      if (!isActive) {
        balances[j] = 0;
        continue;
      }
      balances[j] = balanceOf(user, reservesWithEth[j]);
    }
    balances[reserves.length] = balanceOf(user, MOCK_ETH_ADDRESS);

    return (reservesWithEth, balances);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {VariableDebtToken} from '../../protocol/tokenization/VariableDebtToken.sol';

contract MockVariableDebtToken is VariableDebtToken {
  function getRevision() internal pure override returns (uint256) {
    return 0x2;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {StableDebtToken} from '../../protocol/tokenization/StableDebtToken.sol';

contract MockStableDebtToken is StableDebtToken {
  function getRevision() internal pure override returns (uint256) {
    return 0x2;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {AToken} from '../../protocol/tokenization/AToken.sol';
import {ILendingPool} from '../../interfaces/ILendingPool.sol';
import {IAaveIncentivesController} from '../../interfaces/IAaveIncentivesController.sol';
```

```
contract MockAToken is AToken {
  function getRevision() internal pure override returns (uint256) {
    return 0x2;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

contract SelfdestructTransfer {
  function destroyAndTransfer(address payable to) external payable {
    selfdestruct(to);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ERC20} from '../../dependencies/openzeppelin/contracts/ERC20.sol';

/**
 * @title ERC20Mintable
 * @dev ERC20 minting logic
 */
contract MintableDelegationERC20 is ERC20 {
  address public delegatee;

  constructor(
    string memory name,
    string memory symbol,
    uint8 decimals
  ) public ERC20(name, symbol) {
    _setupDecimals(decimals);
  }

  /**
   * @dev Function to mint tokensp
   * @param value The amount of tokens to mint.
   * @return A boolean that indicates if the operation was successful.
   */
  function mint(uint256 value) public returns (bool) {
    _mint(msg.sender, value);
    return true;
  }

  function delegate(address delegateeAddress) external {
    delegatee = delegateeAddress;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity >=0.4.22 <=0.6.12;

import {WETH9} from '../dependencies/weth/WETH9.sol';

contract WETH9Mocked is WETH9 {
  // Mint not backed by Ether: only for testing purposes
  function mint(uint256 value) public returns (bool) {
    balanceOf[msg.sender] += value;
    emit Transfer(address(0), msg.sender, value);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ERC20} from '../../dependencies/openzeppelin/contracts/ERC20.sol';

/**
 * @title ERC20Mintable
```

```
 * @dev ERC20 minting logic
 */
contract MintableERC20 is ERC20 {
  constructor(
    string memory name,
    string memory symbol,
    uint8 decimals
  ) public ERC20(name, symbol) {
    _setupDecimals(decimals);
  }

  /**
   * @dev Function to mint tokens
   * @param value The amount of tokens to mint.
   * @return A boolean that indicates if the operation was successful.
   */
  function mint(uint256 value) public returns (bool) {
    _mint(_msgSender(), value);
    return true;
  }
}
// Copyright (C) 2015, 2016, 2017 Dapphub

// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.

// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

pragma solidity >=0.4.22 <=0.6.12;

contract WETH9 {
  string public name = 'Wrapped Ether';
  string public symbol = 'WETH';
  uint8 public decimals = 18;

  event Approval(address indexed src, address indexed guy, uint256 wad);
  event Transfer(address indexed src, address indexed dst, uint256 wad);
  event Deposit(address indexed dst, uint256 wad);
  event Withdrawal(address indexed src, uint256 wad);

  mapping(address => uint256) public balanceOf;
  mapping(address => mapping(address => uint256)) public allowance;

  receive() external payable {
    deposit();
  }

  function deposit() public payable {
    balanceOf[msg.sender] += msg.value;
    emit Deposit(msg.sender, msg.value);
  }

  function withdraw(uint256 wad) public {
    require(balanceOf[msg.sender] >= wad);
    balanceOf[msg.sender] -= wad;
    msg.sender.transfer(wad);
    emit Withdrawal(msg.sender, wad);
  }
```

```
  function totalSupply() public view returns (uint256) {
    return address(this).balance;
  }

  function approve(address guy, uint256 wad) public returns (bool) {
    allowance[msg.sender][guy] = wad;
    emit Approval(msg.sender, guy, wad);
    return true;
  }

  function transfer(address dst, uint256 wad) public returns (bool) {
    return transferFrom(msg.sender, dst, wad);
  }

  function transferFrom(
    address src,
    address dst,
    uint256 wad
  ) public returns (bool) {
    require(balanceOf[src] >= wad);

    if (src != msg.sender && allowance[src][msg.sender] != uint256(-1)) {
      require(allowance[src][msg.sender] >= wad);
      allowance[src][msg.sender] -= wad;
    }

    balanceOf[src] -= wad;
    balanceOf[dst] += wad;

    emit Transfer(src, dst, wad);

    return true;
  }
}

/*
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
```

these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying,

distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your

rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not

invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded

from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that

copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this

License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number. If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation. If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions. However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

```
  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.

*/
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {SafeMath} from '../../dependencies/openzeppelin/contracts/SafeMath.sol';
import {IERC20} from '../../dependencies/openzeppelin/contracts/IERC20.sol';

import {FlashLoanReceiverBase} from '../../flashloan/base/FlashLoanReceiverBase.sol';
import {MintableERC20} from '../tokens/MintableERC20.sol';
import {SafeERC20} from '../../dependencies/openzeppelin/contracts/SafeERC20.sol';
import {ILendingPoolAddressesProvider} from '../../interfaces/ILendingPoolAddressesProvider.sol';

contract MockFlashLoanReceiver is FlashLoanReceiverBase {
  using SafeERC20 for IERC20;
```

```
ILendingPoolAddressesProvider internal _provider;

event ExecutedWithFail(address[] _assets, uint256[] _amounts, uint256[] _premiums);
event ExecutedWithSuccess(address[] _assets, uint256[] _amounts, uint256[] _premiums);

bool _failExecution;
uint256 _amountToApprove;
bool _simulateEOA;

constructor(ILendingPoolAddressesProvider provider) public FlashLoanReceiverBase(provider) {}

function setFailExecutionTransfer(bool fail) public {
  _failExecution = fail;
}

function setAmountToApprove(uint256 amountToApprove) public {
  _amountToApprove = amountToApprove;
}

function setSimulateEOA(bool flag) public {
  _simulateEOA = flag;
}

function amountToApprove() public view returns (uint256) {
  return _amountToApprove;
}

function simulateEOA() public view returns (bool) {
  return _simulateEOA;
}

function executeOperation(
  address[] memory assets,
  uint256[] memory amounts,
  uint256[] memory premiums,
  address initiator,
  bytes memory params
) public override returns (bool) {
  params;
  initiator;

  if (_failExecution) {
    emit ExecutedWithFail(assets, amounts, premiums);
    return !_simulateEOA;
  }

  for (uint256 i = 0; i < assets.length; i++) {
    //mint to this contract the specific amount
    MintableERC20 token = MintableERC20(assets[i]);

    //check the contract has the specified balance
    require(
      amounts[i] <= IERC20(assets[i]).balanceOf(address(this)),
      'Invalid balance for the contract'
    );

    uint256 amountToReturn =
      (_amountToApprove != 0) ? _amountToApprove : amounts[i].add(premiums[i]);
    //execution does not fail - mint tokens and return them to the _destination

    token.mint(premiums[i]);

    IERC20(assets[i]).approve(address(LENDING_POOL), amountToReturn);
  }
```

```
    emit ExecutedWithSuccess(assets, amounts, premiums);

    return true;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IUniswapV2Router02} from '../../interfaces/IUniswapV2Router02.sol';
import {IERC20} from '@openzeppelin/contracts/token/ERC20/IERC20.sol';
import {MintableERC20} from '../tokens/MintableERC20.sol';

contract MockUniswapV2Router02 is IUniswapV2Router02 {
  mapping(address => uint256) internal _amountToReturn;
  mapping(address => uint256) internal _amountToSwap;
  mapping(address => mapping(address => mapping(uint256 => uint256))) internal _amountsIn;
  mapping(address => mapping(address => mapping(uint256 => uint256))) internal _amountsOut;
  uint256 internal defaultMockValue;

  function setAmountToReturn(address reserve, uint256 amount) public {
    _amountToReturn[reserve] = amount;
  }

  function setAmountToSwap(address reserve, uint256 amount) public {
    _amountToSwap[reserve] = amount;
  }

  function swapExactTokensForTokens(
    uint256 amountIn,
    uint256, /* amountOutMin */
    address[] calldata path,
    address to,
    uint256 /* deadline */
  ) external override returns (uint256[] memory amounts) {
    IERC20(path[0]).transferFrom(msg.sender, address(this), amountIn);

    MintableERC20(path[1]).mint(_amountToReturn[path[0]]);
    IERC20(path[1]).transfer(to, _amountToReturn[path[0]]);

    amounts = new uint256[](path.length);
    amounts[0] = amountIn;
    amounts[1] = _amountToReturn[path[0]];
  }

  function swapTokensForExactTokens(
    uint256 amountOut,
    uint256, /* amountInMax */
    address[] calldata path,
    address to,
    uint256 /* deadline */
  ) external override returns (uint256[] memory amounts) {
    IERC20(path[0]).transferFrom(msg.sender, address(this), _amountToSwap[path[0]]);

    MintableERC20(path[1]).mint(amountOut);
    IERC20(path[1]).transfer(to, amountOut);

    amounts = new uint256[](path.length);
    amounts[0] = _amountToSwap[path[0]];
    amounts[1] = amountOut;
  }

  function setAmountOut(
    uint256 amountIn,
    address reserveIn,
    address reserveOut,
    uint256 amountOut
```

```
  ) public {
    _amountsOut[reserveIn][reserveOut][amountIn] = amountOut;
  }

  function setAmountIn(
    uint256 amountOut,
    address reserveIn,
    address reserveOut,
    uint256 amountIn
  ) public {
    _amountsIn[reserveIn][reserveOut][amountOut] = amountIn;
  }

  function setDefaultMockValue(uint256 value) public {
    defaultMockValue = value;
  }

  function getAmountsOut(uint256 amountIn, address[] calldata path)
    external
    view
    override
    returns (uint256[] memory)
  {
    uint256[] memory amounts = new uint256[](path.length);
    amounts[0] = amountIn;
    amounts[1] = _amountsOut[path[0]][path[1]][amountIn] > 0
      ? _amountsOut[path[0]][path[1]][amountIn]
      : defaultMockValue;
    return amounts;
  }

  function getAmountsIn(uint256 amountOut, address[] calldata path)
    external
    view
    override
    returns (uint256[] memory)
  {
    uint256[] memory amounts = new uint256[](path.length);
    amounts[0] = _amountsIn[path[0]][path[1]][amountOut] > 0
      ? _amountsIn[path[0]][path[1]][amountOut]
      : defaultMockValue;
    amounts[1] = amountOut;
    return amounts;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

contract MockAggregator {
  int256 private _latestAnswer;

  event AnswerUpdated(int256 indexed current, uint256 indexed roundId, uint256 timestamp);

  constructor(int256 _initialAnswer) public {
    _latestAnswer = _initialAnswer;
    emit AnswerUpdated(_initialAnswer, 0, now);
  }

  function latestAnswer() external view returns (int256) {
    return _latestAnswer;
  }

  function getTokenType() external view returns (uint256) {
    return 1;
  }
```

```
  // function getSubTokens() external view returns (address[] memory) {
  // TODO: implement mock for when multiple subtokens. Maybe we need to create diff mock contract
  // to call it from the migration for this case??
  // }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {IPriceOracle} from '../../interfaces/IPriceOracle.sol';

contract PriceOracle is IPriceOracle {
  mapping(address => uint256) prices;
  uint256 ethPriceUsd;

  event AssetPriceUpdated(address _asset, uint256 _price, uint256 timestamp);
  event EthPriceUpdated(uint256 _price, uint256 timestamp);

  function getAssetPrice(address _asset) external view override returns (uint256) {
    return prices[_asset];
  }

  function setAssetPrice(address _asset, uint256 _price) external override {
    prices[_asset] = _price;
    emit AssetPriceUpdated(_asset, _price, block.timestamp);
  }

  function getEthUsdPrice() external view returns (uint256) {
    return ethPriceUsd;
  }

  function setEthUsdPrice(uint256 _price) external {
    ethPriceUsd = _price;
    emit EthPriceUpdated(_price, block.timestamp);
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface ChainlinkUSDETHOracleI {
  event AnswerUpdated(int256 indexed current, uint256 indexed answerId);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface IExtendedPriceAggregator {
  event AnswerUpdated(int256 indexed current, uint256 indexed roundId, uint256 timestamp);

  function getToken() external view returns (address);

  function getTokenType() external view returns (uint256);

  function getPlatformId() external view returns (uint256);

  function getSubTokens() external view returns (address[] memory);

  function latestAnswer() external view returns (int256);
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

import {ILendingRateOracle} from '../../interfaces/ILendingRateOracle.sol';
import {Ownable} from '../../dependencies/openzeppelin/contracts/Ownable.sol';

contract LendingRateOracle is ILendingRateOracle, Ownable {
  mapping(address => uint256) borrowRates;
  mapping(address => uint256) liquidityRates;
```

```
  function getMarketBorrowRate(address _asset) external view override returns (uint256) {
    return borrowRates[_asset];
  }

  function setMarketBorrowRate(address _asset, uint256 _rate) external override onlyOwner {
    borrowRates[_asset] = _rate;
  }

  function getMarketLiquidityRate(address _asset) external view returns (uint256) {
    return liquidityRates[_asset];
  }

  function setMarketLiquidityRate(address _asset, uint256 _rate) external onlyOwner {
    liquidityRates[_asset] = _rate;
  }
}
// SPDX-License-Identifier: agpl-3.0
pragma solidity 0.6.12;

interface GenericOracleI {
  // ganache
  event AssetPriceUpdated(address _asset, uint256 _price, uint256 timestamp);
  event EthPriceUpdated(uint256 _price, uint256 timestamp);

  // kovan
  event ProphecySubmitted(
    address indexed _sybil,
    address indexed _asset,
    uint96 _sybilProphecy,
    uint96 _oracleProphecy
  );

  function getAssetPrice(address _asset) external view returns (uint256);

  function getEthUsdPrice() external view returns (uint256);
}
```


Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.
- **Detection results:**

  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io