EVM-Side Lending Platform

🧠 EVM-Side Lending Platform

🌐 Hosted UI (Vercel)

I've created a fully functional front-end for this lending platform and deployed it on Vercel for easier testing and interaction. The deployed UI provides an intuitive way to interact with the smart contract without the need to deploy it manually.

Vercel Link: https://lending-pool.vercel.app/

Deploy all these step by step :-

1. **ERC20 (Collateral Token)**
2. **ERC20Stable (Lending Token)**
3. **CollateralPriceContract**
4. **LendingPoolSimple**

Private key import & token funding enabled Check the .env

🛠 Setup & Token Deployment Steps:

1. **Install dependencies**
   Run the following command to install packages:

   ```
   npm install --legacy-peer-deps
   ```

2. **Configure Environment Variables**
   Ensure the `.env` file is correctly set up with all necessary variables.
3. **Import the Private Key**
   Load the private address (owner/deployer) into your wallet or script environment.
4. **Import Tokens**
   Add the deployed token contracts to your wallet or project for interaction.
5. **Mint Tokens (if needed)**
   Mint tokens as required for testing or initial distribution.
6. **Preloaded Tokens for Owner**
   Note: The owner account has already been credited with minted tokens.
7. **Start Depositing Roy-Token**
   Begin the deposit process using the Roy-Token.

📌 What’s Covered in the Smart Contract

This project is a simplified DeFi lending and borrowing protocol that allows users to:

1. ✅ Deposit Assets to Earn Interest

- Users can deposit a predefined ERC20 token (`ROY-TOKEN`) into the lending pool.
- The system tracks each user’s deposited balance along with accrued interest.
- Interest is calculated **per block** and accumulated internally via the `_accrueInterest()` function.

2. ✅ Withdraw Funds with Interest

- Users can withdraw their deposited assets along with the interest earned.
- Withdrawals are restricted if the user has a borrowing position that would become undercollateralized post-withdrawal.
- The contract uses real-time health checks to ensure safety before withdrawals.

3. ✅ Collateralized Borrowing

- Users can deposit a **second token (`COL-TOKEN`)** as collateral.
- Borrowing is allowed only if the **collateral value** satisfies the **150% collateralization ratio**.
- Borrowing amount is restricted based on live price fetched from an external oracle (`IPriceFeed`).

4. ✅ Repayment of Debt

- Users can repay part or all of their debt using the `repay()` function.
- Overpayments are automatically capped to avoid reverting.

5. ✅ Collateral Management

- Users can add or remove collateral.
- Removing collateral is only permitted if the health factor remains above the required threshold.

6. ✅ Liquidation Logic

- If a user’s **health factor falls below 125%**, anyone can liquidate their position.
- Liquidators repay a portion of the debt and receive discounted collateral (via a **5% liquidation bonus**).
- Price feed ensures the liquidation happens based on real-time asset value.

💻 Frontend Flow (UI Walkthrough)

The front-end interface was designed with clarity and usability in mind. It reflects the logical flows of the protocol:

🧾 Dashboard Overview

- **Balances Panel**: Shows the user’s deposit, borrowed amount, and current collateral.

- **Health Factor & Interest Gained**: Real-time metrics based on contract state.
- **Action Panel**: Toggle between deposit, withdraw, borrow, and repay actions with real-time interest rate info.

⚙️ Collateral Management

- A dedicated section to add or remove collateral securely.
- Uses real-time checks to prevent accidental undercollateralization.

📊 Market Info

- Displays the protocol's configuration:
  - Collateralization Ratio
  - Liquidation Threshold
  - Liquidation Bonus
  - Collateral Token Price (fetched from oracle)
  - Current Block

🔐 Security Considerations

- **Reentrancy Protection**: Achieved via OpenZeppelin's `ReentrancyGuard`.
- **Safe Transfers**: All ERC20 interactions use `SafeERC20`.
- **Proper Access Controls**: Price feed and sensitive operations are restricted to `owner`.

Feature	Description
✅ **Oracle Integration**	Mocked `IPriceFeed` to simulate Chainlink-style oracles.
✅ **Health Factor Calculations**	Implemented to safeguard collateralized positions.
✅ **Dynamic UI**	Live updates of deposited tokens, borrowed value, health factor.
✅ **Liquidation Bonus**	Built-in incentive for third-party liquidators.
✅ **Per-block Interest Model**	Achieves precision while being efficient.

🧪 How to Try It

1. Open the Vercel-hosted frontend.
2. Import your wallet (compatible with Sepolia).
3. Add test tokens (`ROY-TOKEN`, `COL-TOKEN`).
4. Interact with the protocol directly:
   - Deposit → Earn → Borrow → Repay → Liquidate → Withdraw.

✅ Summary

This was an end-to-end DeFi lending/borrowing platform simulation built as part of your task. I focused on implementing core features **securely and clearly**, with **clean logic and a responsive frontend**.

The liquidation mechanism is implemented at the smart contract level and is fully functional. However, it's not currently exposed via the frontend UI. You can test the liquidation by calling the liquidate() function directly via script or contract interaction using remix or hardhat. I’m planning to add a test case in Hardhat to demonstrate how and when liquidation should be triggeredwhen a borrower's health factor falls below the threshold.