Capstone:
Ethereum Smart Contracts for Blockchain-Based Infrastructure Funding

Project Members: Derek Dang, Samuel Knox, Altynbek Yermurat, Michael Gadda, Shuwen Xu, Abhimanyu Bais

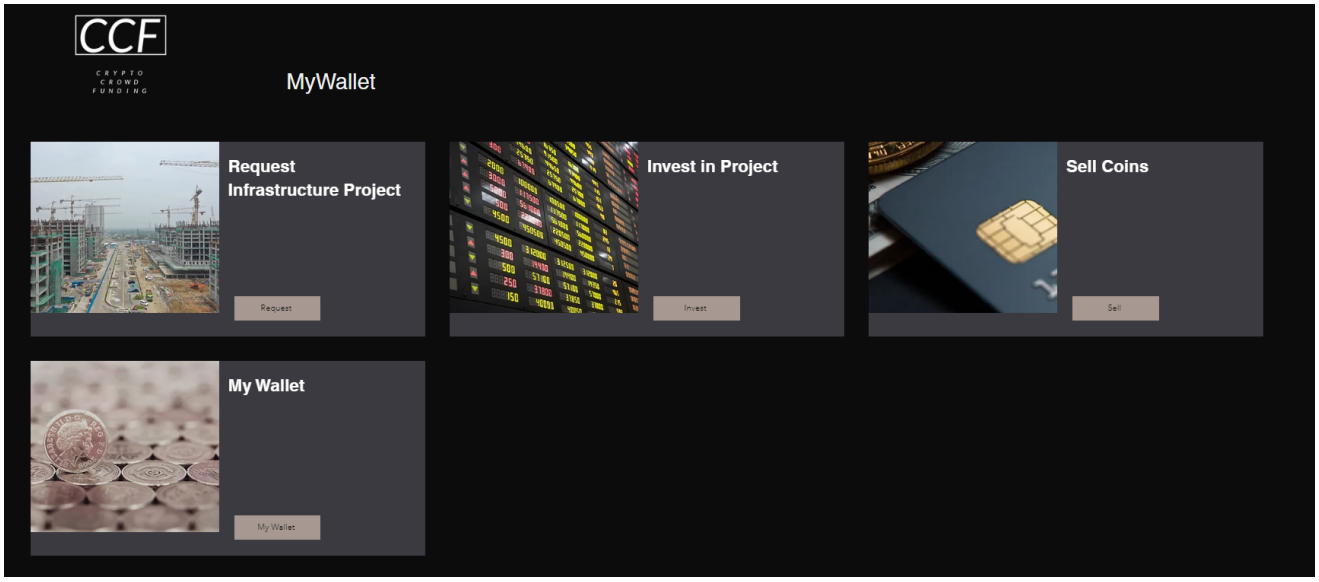


Table of Contents

Introduction

Our primary objective is to eliminate delays and minimize cost overruns in the development of infrastructure projects by utilizing blockchain-based smart contracts to streamline the management of funding and ownership of these assets. By harnessing the power of blockchain technology, we aim to significantly enhance the transparency and accountability of transactions, thereby fostering trust among all stakeholders, including investors, developers, and end-users.

To achieve this, we are developing a cutting-edge prototype of an Ethereum-based token that enables the creation of smart contracts to automate various operations associated with infrastructure projects. These asset-tokens will serve as digital representations of ownership stakes in infrastructure projects, and they will be tied to the revenue generated by the projects, such as tolls collected from commuters.

One practical application of our innovative solution would be the allocation of ownership interests in a bridge project, wherein the value of the asset-tokens would be directly correlated with the toll revenues generated by the bridge. This approach allows investors to actively participate in the growth of the infrastructure projects they have funded while ensuring a seamless, automated distribution of revenue through the use of smart contracts.

In conclusion, our blockchain-based smart contracts platform has the potential to revolutionize the infrastructure development landscape by minimizing delays, reducing cost overruns, and improving overall transparency. This innovative approach will not only attract more investors to the sector but also contribute to the growth of sustainable, efficient infrastructure that benefits society as a whole.

The deliverables involve creating a prototype ethereum-based token for which smart-contracts can be created to automate operations. An example would be dividing ownership of a bridge using these asset-tokens and tying its value to the toll collected from commuters.

Traditional Infrastructure Funding

Private-Public Partnership Summary

A public private partnership is an agreement between a government or public agency and a private entity to deliver a service for the public benefit. PPPs are typically used for infrastructure projects such as building highways, airports, or bridges. PPPs have certain fundamentals that define the process. Both public and private parties bear the risk and the reward of the projects success. Each party also has clearly defined roles and responsibilities, governments of public agencies typically set project goals and objectives, while the private sector would be responsible for designing, building, and financing the project.

Cryptocurrencies can improve the current PPP process by increasing transparency, accountability, and efficiency through blockchain technology. The blockchain is useful for this process as it can provide a record of all transactions and activities within a project. This allows all parties to have immediate access to all information of a project in real time. Applying governance tokens allow investors to have the ability to vote and change infrastructure if needed. The use of smart contracts in PPPs can automate most if not all of the contractual obligations and payments, reducing the need for intermediaries. Cryptocurrencies can also be used to facilitate payments between parties which can lower transaction costs. Overall, the use of cryptocurrencies in PPPs can drastically reduce costs, increase transparency and efficiency.

Decentralized Finance

Decentralized finance (DeFi) is a financial system built on blockchain technology that aims to provide more accessible, transparent, and secure financial services to users without the need for intermediaries such as banks or financial institutions. DeFi allows anyone with an internet connection to access various financial services, such as lending, borrowing, trading, and investing, by interacting with smart contracts on decentralized platforms.

DeFi has the potential to improve the process of public-private partnerships (PPP) by enabling a more transparent and secure way to manage and distribute funds. PPPs involve collaboration between public and private entities to finance and deliver public services, such as infrastructure projects. However, the traditional PPP model often involves complex and opaque financing arrangements, which can lead to inefficiencies, corruption, and lack of accountability.

With DeFi, PPPs could be managed through smart contracts that automatically execute the terms of the agreement, including the distribution of funds to various parties involved in the project. Smart contracts can also enable more transparent and auditable financial reporting, as all transactions are recorded on a public blockchain ledger. Additionally, DeFi can provide more accessible and flexible financing options, such as peer-to-peer lending or crowdfunding, which

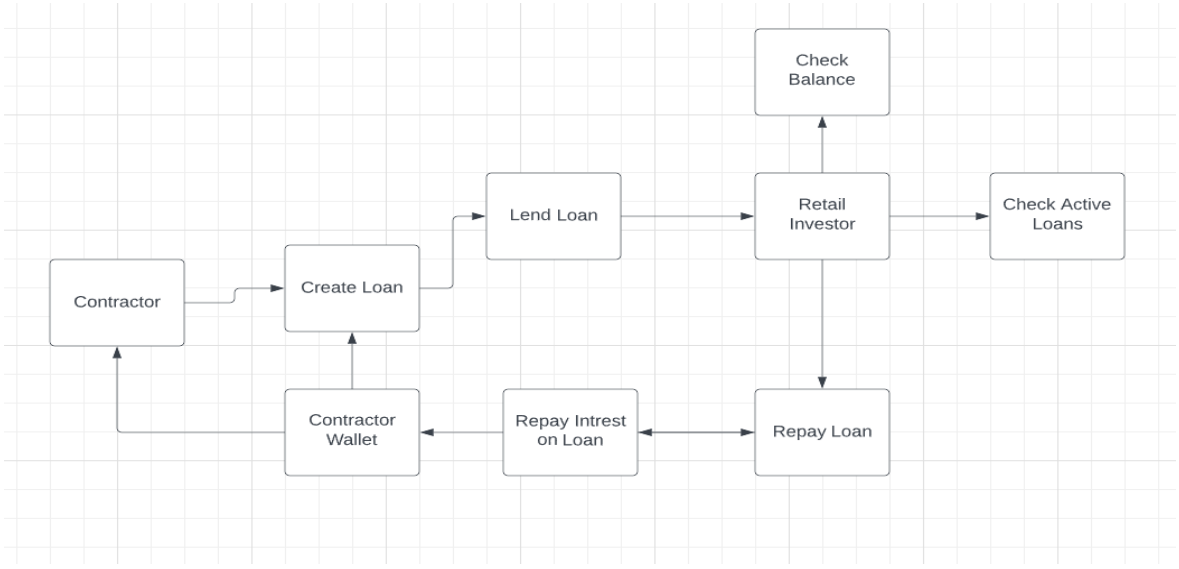
can help to democratize the process of PPPs and reduce reliance on traditional financing sources.

DeFi has the potential to improve the efficiency, transparency, and accessibility of financial services, including those related to public-private partnerships, by leveraging the benefits of blockchain technology and removing the need for intermediaries.

Smart Contract Breakdown

Lenders Contract-

The lenders contract contains multiple public and private functions within the contract. The code contains several functions that are used to manage, lenders, lendees, and loa



ns stored in a database. Within the contract, one is able to check the remaining loan balance, create a loan, check the remaining time on the loan, and pay debt. There are also multiple functions that check whether a user has requested a loan, if a user is in debt, and getting a list of all investors.

createLoan-

```
function createLoan(address payable _lendee, address payable _lender, uint256 _loanAmount, uint256 _interestRate, uint256 _loanPeriod, uint256 _loanInstallmentPeriod) public {
    require(msg.sender == _lender);
    require(_loanAmount > 0);
    require(_interestRate > 0 && _interestRate <= 100);
    require(_loanPeriod > 0);
    loan memory newLoan;
    newLoan.loanId = nextLoanId;
    nextLoanId += 1;
    newLoan.lender = _lendee;
    newLoan.lender = _lender;
    newLoan.loanAmount = _loanAmount;
    newLoan.interestRate = _interestRate;
    newLoan.loanPeriod = _loanPeriod;
    newLoan.loanInstallmentPeriod = _loanInstallmentPeriod;
    newLoan.daysBetweenInstallments = _loanInstallmentPeriod;
    newLoan.installmentAmount = _loanAmount/_loanInstallmentPeriod;
    newLoan.loanStart = block.timestamp;
    newLoan.loanEnd = block.timestamp + _loanPeriod;
    newLoan.totalInterest = _loanAmount*_interestRate/100;
    newLoan.interestLeft = newLoan.totalInterest;
    newLoan.loanAmountLeft = _loanAmount;
    newLoan.totalReceivedAmount = 0;
    newLoan.principleLoanPayed = 0;
    newLoan.interestPayed = 0;
    newLoan.previousLoanInstallmentDate = block.timestamp;
    loans.push(newLoan);
    lenderDeposit = true;
    uint lenderArrayIndex = getLenderArrayIndex(_lender);
    if (lenderArrayIndex != 999999999) {
        lenders[lenderArrayIndex].loanIds.push(newLoan.loanId);
    } else {
        addNewLenderToDataBase(_lender);
        lenders[getLenderArrayIndex(_lender)].loanIds.push(newLoan.loanId);
    }
    uint lendeeArrayIndex = getLendeeArrayIndex(_lendee);
    if (lendeeArrayIndex != 999999999) {
        lendees[lendeeArrayIndex].loanIds.push(newLoan.loanId);
    } else {
        addNewLendeeToDataBase(_lendee);
        lendees[getLendeeArrayIndex(_lendee)].loanIds.push(newLoan.loanId);
    }
    emit LoanCreated("Please write down your loanId so you can access it at a later date.", newLoan.loanId, newLoan.lender, newLoan.lendee, newLoan.loanAmount, newLoan.interestRate, newLoan.loanPeriod, newLoan.loanInstallmentPer
```

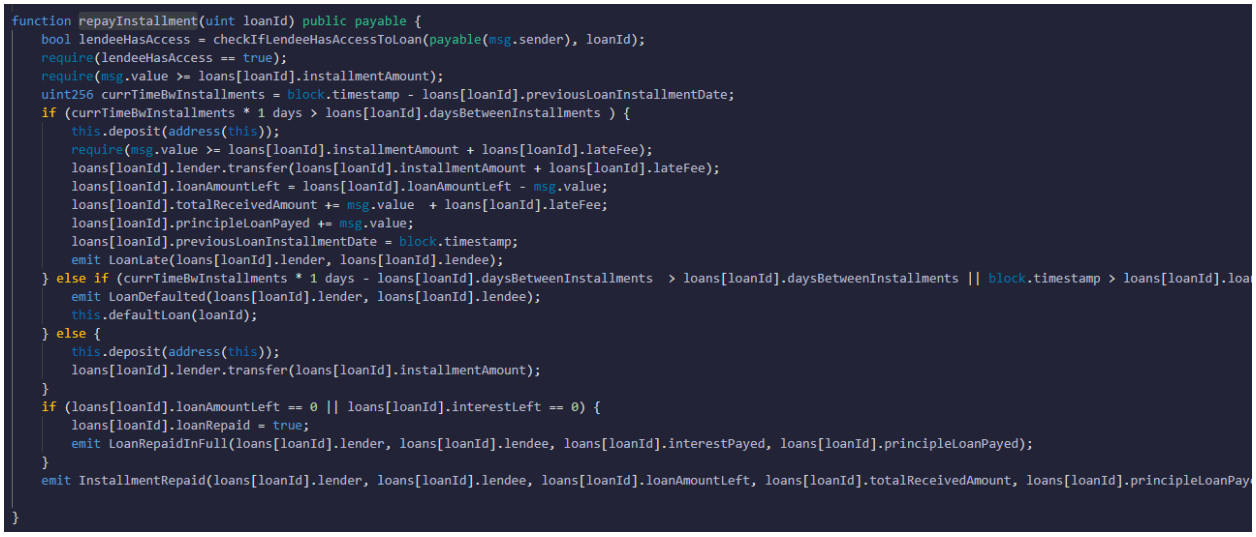
The "createLoan" function is a Solidity function that allows a lender to create a new loan. It takes several arguments representing the details of the loan, such as the lendee's address, the loan amount, the interest rate, the loan period, and the loan installment period. The "createLoan" function allows a lender to create a new loan by creating a new "loan" struct with the provided information and adding it to the end of the "loans" list. The function also updates the lender and lendee information and emits an event with the loan details.

lendLoan-

```
function lendLoan(uint256 loanId) public payable {
    require(msg.sender == loans[loanId].lender);
    require(msg.value >= loans[loanId].loanAmount);
    require(lenderDeposit == true);
    this.deposit(address(this));
    //require(msg.value == loans[loanId].loanAmount);
    loans[loanId].lendee.transfer(loans[loanId].loanAmount);
    emit Lended(loans[loanId].lendee, loans[loanId].lender, loans[loanId].loanAmount);
    }
```

The "lendLoan" function allows a lender to lend an existing loan by transferring the loan amount to the lendee's address. The function ensures that the caller is the lender, the value sent is sufficient, and the lender deposit flag is true before transferring the loan amount. It also emits an event with the loan details.

repayInstallment-



```
function repayInstallment(uint loanId) public payable {
    bool lendeeHasAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender), loanId);
    require(lendeeHasAccess == true);
    require(msg.value >= loans[loanId].installmentAmount);
    uint256 currTimeBwInstallments = block.timestamp - loans[loanId].previousLoanInstallmentDate;
    if (currTimeBwInstallments * 1 days > loans[loanId].daysBetweenInstallments ) {
        this.deposit(address(this));
        require(msg.value >= loans[loanId].installmentAmount + loans[loanId].lateFee);
        loans[loanId].lender.transfer(loans[loanId].installmentAmount + loans[loanId].lateFee);
        loans[loanId].loanAmountLeft = loans[loanId].loanAmountLeft - msg.value;
        loans[loanId].totalReceivedAmount += msg.value  + loans[loanId].lateFee;
        loans[loanId].principleLoanPayed += msg.value;
        loans[loanId].previousLoanInstallmentDate = block.timestamp;
        emit LoanLate(loans[loanId].lender, loans[loanId].lendee);
    } else if (currTimeBwInstallments * 1 days - loans[loanId].daysBetweenInstallments  > loans[loanId].daysBetweenInstallments || block.timestamp > loans[loanId].loa
        emit LoanDefaulted(loans[loanId].lender, loans[loanId].lendee);
        this.defaultLoan(loanId);
    } else {
        this.deposit(address(this));
        loans[loanId].lender.transfer(loans[loanId].installmentAmount);
    }
    if (loans[loanId].loanAmountLeft == 0 || loans[loanId].interestLeft == 0) {
        loans[loanId].loanRepaid = true;
        emit LoanRepaidInFull(loans[loanId].lender, loans[loanId].lendee, loans[loanId].interestPayed, loans[loanId].principleLoanPayed);
    }
    emit InstallmentRepaid(loans[loanId].lender, loans[loanId].lendee, loans[loanId].loanAmountLeft, loans[loanId].totalReceivedAmount, loans[loanId].principleLoanPay

}
```

The function first checks whether the lendee has access to the loan by calling the "checkIfLendeeHasAccessToLoan" function. If the lendee has access, the function checks whether the value sent is greater than or equal to the installment amount.
If the current time between installments is greater than the days between installments or the current time exceeds the loan end date, the function checks whether the value sent is greater than or equal to the installment amount plus the late fee. If this condition is satisfied, the function transfers the installment amount plus the late fee to the lender's address using the "transfer" function.If the current time between installments is less than or equal to the days between installments and the current time does not exceed the loan end date, the function transfers only the installment amount to the lender's address.

After the installment amount is transferred, the function updates the loan details such as the loan amount left, the total received amount, the principle loan paid, and the previous loan installment date. If the loan amount left or the interest left is zero, the function sets the "loanRepaid" flag to true and emits an event called "LoanRepaidInFull". If not, the function emits an event called "InstallmentRepaid" with the updated loan details. If the current time between installments exceeds the days between installments or the current time exceeds the loan end date, the function emits an event called "LoanDefaulted" and calls the "defaultLoan" function to handle the defaulted loan.

The "repayInstallment" function allows a lendee to repay an installment of an existing loan. The function checks whether the lendee has access to the loan, whether the value sent is sufficient, and whether the loan is defaulted or repaid in full. The function also updates the loan details and emits events with the updated loan details.

repayInterest-

```
function repayInterest(uint loanId) public payable {
    bool lendeeHasAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender), loanId);
    require(lendeeHasAccess == true);
    require(msg.value >= 0);
    if (block.timestamp > loans[loanId].loanEnd) {
        this.defaultLoan(loanId);
        emit LoanDefaulted(loans[loanId].lender, loans[loanId].lendee);
    } else if (msg.value <= loans[loanId].interestLeft) {
        this.deposit(address(this));
        loans[loanId].lender.transfer(msg.value);
        loans[loanId].interestPayed += msg.value;
        loans[loanId].interestLeft -= msg.value;
        loans[loanId].totalReceivedAmount += msg.value;
    }
    if (loans[loanId].loanAmountLeft == 0 || loans[loanId].interestLeft == 0) {
        loans[loanId].loanRepaid = true;
        emit LoanRepaidInFull(loans[loanId].lender, loans[loanId].lendee, loans[loanId].interestPayed, loans[loanId].principleLoanPayed);
    }
    emit InterestRepaid(loans[loanId].lender, loans[loanId].lendee, loans[loanId].interestLeft, loans[loanId].interestPayed, loans[loanId].totalReceivedAmount);

}
```

The "repayInterest" function first checks whether the lendee has access to the loan by calling the "checkIfLendeeHasAccessToLoan" function. If the lendee has access, the function checks whether the value sent is greater than or equal to 0. If the value sent is less than or equal to the interest left on the loan, the function transfers the value sent to the lender's address using the "transfer" function. The function updates the loan details such as the interest paid, the interest left, and the total received amount. The "repayInterest" function allows a lendee to repay the interest on an existing loan. The function checks whether the lendee has access to the loan, whether the value sent is sufficient, and whether the loan is defaulted or repaid in full. The function also updates the loan details and emits events with the updated loan details.

balanceOf-

```
function balanceOf() public view returns(uint) {
    require(msg.sender == owner);
    return address(this).balance;
}
```

All this function does is get the balance of a specific user when requested.

remainingLoanBalance-

```
function remainingLoanBalance(uint loanId) public {
    bool lendeeAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender),  loanId);
    bool lenderAccess =  checkifLenderHasAccessToLoan(payable(msg.sender),  loanId);
    require(lendeeAccess == true || lenderAccess == true, "You do not have access to this loan.");
    emit amountLeft(loanId, loans[loanId].loanAmountLeft);
    //return loans[loanId].loanAmountLeft;
}
```

The function first checks whether the lendee or the lender has access to the loan by calling the "checkIfLendeeHasAccessToLoan" and "checkIfLenderHasAccessToLoan" functions, respectively. If the caller has access, the function emits an event called "amountLeft" with the

remaining loan balance. The "remainingLoanBalance" function allows the lender or the lendee to check the remaining balance on an existing loan. The function checks whether the caller has access to the loan and emits an event with the remaining loan balance.

remainingInterestBalance-

```
function remainingInterestBalance(uint loanId) public {
    bool lendeeAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender),  loanId);
    bool lenderAccess =  checkifLenderHasAccessToLoan(payable(msg.sender),  loanId);
    require(lendeeAccess == true || lenderAccess == true, "You do not have access to this loan.");
    emit amountLeft(loanId, loans[loanId].interestLeft);
    //return loans[loanId].interestLeft;
}
```

The "remainingInterestBalance" function allows the lender or the lendee to check the remaining interest balance on an existing loan. The function checks whether the caller has access to the loan and emits an event with the remaining interest balance. The function first checks whether the lendee or the lender has access to the loan by calling the "checkIfLendeeHasAccessToLoan" and "checkIfLenderHasAccessToLoan" functions, respectively. If the caller has access, the function emits an event called "amountLeft" with the remaining interest balance.

remainingTimeForLoan-

```
function remainingTimeForLoan(uint loanId) public {
    bool lendeeAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender),  loanId);
    bool lenderAccess =  checkifLenderHasAccessToLoan(payable(msg.sender),  loanId);
    require(lendeeAccess == true || lenderAccess == true, "You do not have access to this loan.");
    emit amountLeft(loanId, (loans[loanId].loanEnd - block.timestamp));
    //return (loans[loanId].loanEnd - block.timestamp) * 1 days;
}
```

This function calculates and returns the remaining time for a loan to reach its end date. It checks if the caller is either the lendee or the lender associated with the loan. If the caller has access to the loan, it emits an event with the remaining time in seconds until the loan end date.

checkPaidBalance-

```
function checkPaidBalance(uint loanId) public returns(uint) {
    bool lendeeAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender),  loanId);
    bool lenderAccess =  checkifLenderHasAccessToLoan(payable(msg.sender),  loanId);
    require(lendeeAccess == true || lenderAccess == true, "You do not have access to this loan.");
    return loans[loanId].totalReceivedAmount;
}
```

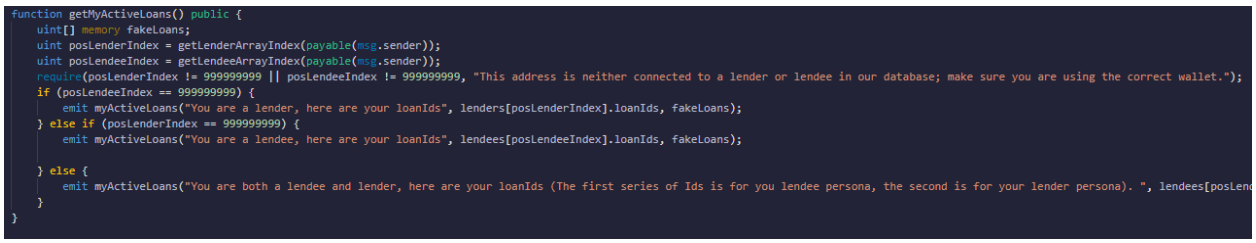
This function checks whether the caller has access to the loan by verifying whether they are either the lendee or the lender. If they have access, it returns the total amount of money that has been paid back towards the loan.

repayCustAmountLoan-

```
function repayCustAmountLoan(uint loanId) public payable {
    bool lendeeAccess = checkIfLendeeHasAccessToLoan(payable(msg.sender),  loanId);
    require(lendeeAccess == true, "You do not have access to this loan.");
    require(msg.value >= 0);
    loans[loanId].loanAmountLeft = loans[loanId].loanAmountLeft - msg.value;
    loans[loanId].totalReceivedAmount += msg.value  + loans[loanId].lateFee;
    loans[loanId].principleLoanPayed += msg.value;
    this.deposit(address(this));
    loans[loanId].lender.transfer(msg.value);
    }
```

This function allows the lendee to repay a specific amount of the loan by sending ether to the contract. It checks if the lendee has access to the loan, the value sent is greater than or equal to zero, and then subtracts the amount sent from the remaining loan balance. It also updates the total amount received from the lendee, the amount of principal loan paid, and transfers the ether to the lender.

getMyActiveLoans-

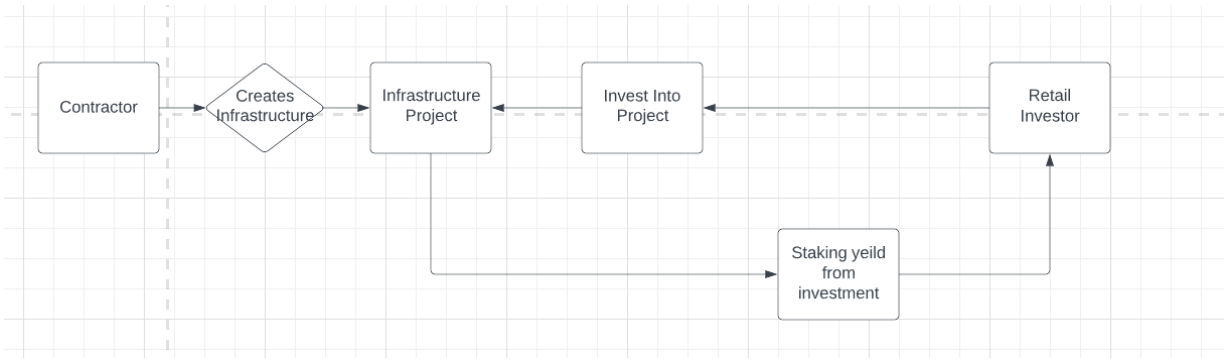


```
function getMyActiveLoans() public {
    uint[] memory fakeLoans;
    uint posLenderIndex = getLenderArrayIndex(payable(msg.sender));
    uint posLendeeIndex = getLendeeArrayIndex(payable(msg.sender));
    require(posLenderIndex != 999999999 || posLendeeIndex != 999999999, "This address is neither connected to a lender or lendee in our database; make sure you are using the correct wallet.");
    if (posLendeeIndex == 999999999) {
        emit myActiveLoans("You are a lender, here are your loanIds", lenders[posLenderIndex].loanIds, fakeLoans);
    } else if (posLenderIndex == 999999999) {
        emit myActiveLoans("You are a lendee, here are your loanIds", lendees[posLendeeIndex].loanIds, fakeLoans);

    } else {
        emit myActiveLoans("You are both a lendee and lender, here are your loanIds (The first series of Ids is for you lendee persona, the second is for your lender persona). ", lendees[posLend
    }
}
```

This function retrieves the active loans associated with the caller's address, by checking if the address is registered as a lender or lendee in the database. If the address is registered as a lender, it returns the loan IDs associated with that lender. If the address is registered as a lendee, it returns the loan IDs associated with that lendee. If the address is registered as both a lendee and a lender, it returns two sets of loan IDs, one for the lendee and one for the lender. The function emits an event with the loan IDs as output.

Staking Contract-



The staking contract contains a couple public functions that pay investors that have funded a company. An investor is payed 1/1000 of the invested amount. The payout function is used to distribute the payout to every investor. It transfers the payout amount to each address and updates that last payout time.

invest-

```
function invest(address payable _investor, uint256 _amount) public {
    investments[_investor].investor = _investor;
    investments[_investor].amount = _amount;
    investments[_investor].lastPayout = block.timestamp;
    investors.push(_investor);
}
```

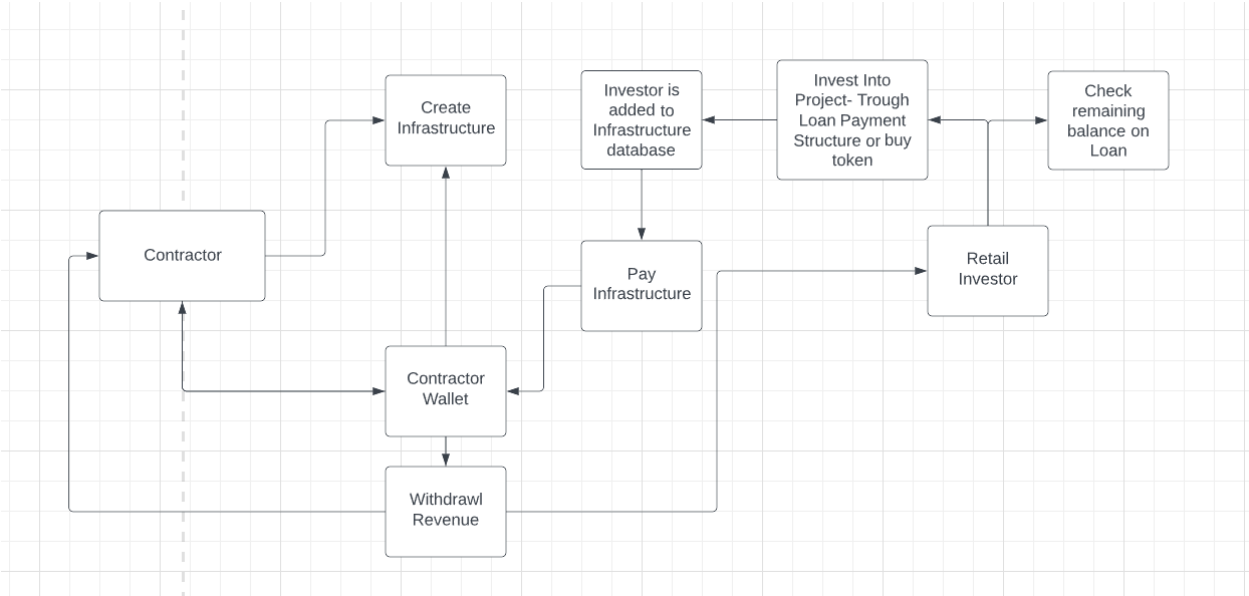
This function allows an investor to invest a certain amount of funds by adding a new entry to the investments mapping with the investor's address as the key. The amount invested is stored in the amount field of the Investment struct. The lastPayout field is set to the current block timestamp, indicating that the investor has not yet received any payouts. Finally, the investor's address is added to the investors array to keep track of all investors.

payout-

```
function payout() public {
    for (uint256 i = 0; i < investors.length; i++) {
        address payable investor = investors[i];
        Investment storage investment = investments[investor];
        //uint256 duration = block.timestamp - investment.lastPayout;
        uint256 payoutAmount = calculatePayout(investment.amount);
        investor.transfer(payoutAmount);
        investment.lastPayout = block.timestamp;
        investment.payout = payoutAmount;
    }
}
```

This function performs a payout to all investors in a smart contract. It loops through all investors and calculates the payout amount for each one based on their investment amount and the current time. Then it transfers the payout amount to the investor's address and updates the investment's last payout and payout amount in the contract storage.

Toll-based Contract-



The toll-based contract is used to create a new project and list it. We are able to add users that invested into the project and we plan to communicate with the lending contract in future iterations. As of now we are able to deposit a test token, add infrastructures, and add users to a database. When the contract is complete, users will be able to see what infrastructure they are invested in, and how much of a stake they have in the project.

addInfStructure-

```
function addInfStructure(address payable infWallet, string memory infName, uint member_type1_cost, uint member_type2_cost, uint member_type3_cost,
    require(msg.sender == owner, "You must be the owner of the contract to add a new infrastructure, please contact the owner!");
    require(checkInfNameIsNotAlreadyInArray(infName) == false, "This Name is already in use, please choose another name and try again.");
    infrastructure memory newInf;
    newInf.infWallet = infWallet;
    newInf.infName = infName;
    newInf.member_type1_cost = member_type1_cost;
    newInf.member_type2_cost = member_type2_cost;
    newInf.member_type3_cost = member_type3_cost;
    newInf.member_type1_threshold = member_type1_threshold;
    newInf.member_type2_threshold = member_type2_threshold;
    newInf.member_type3_threshold = member_type3_threshold;
    infStructArray.push(newInf);
}
```

The "addInfStructure" function adds a new infrastructure wallet to the database by creating an "infrastructure" struct with the provided information and adding it to the end of the "infStructArray" list. The function requires the caller to be the owner of the contract and checks whether the infrastructure name is already in use before adding the new infrastructure wallet.

addUserToDataBase-

```
function addUserToDataBase(string memory firstname, string memory lastname, string memory infName) public {
    require(keccak256(bytes(firstname)) != keccak256(bytes("NULL")));
    individualUser memory newUser;
    newUser.userFirstName = firstname;
    newUser.userLastName = lastname;
    newUser.userWallet = payable(msg.sender);
    newUser.userType = 1;
    newUser.infName = infName;
    userStructArray.push(newUser);
}
```

The "addUserToDataBase" function adds a new user to the database by creating an "individualUser" struct with the provided information and adding it to the end of the "userStructArray" list.

withdrawRevenue-

```
function withdrawRevenue(uint amountToWithdraw) public {
    require(amountToWithdraw > 0, "Please enter a number greater than 0 to withdraw!");
    infrastructure memory infStruct = findInfWalletByAddressForWithdrawal(payable(msg.sender));
    require(keccak256(bytes(infStruct.infName)) != keccak256(bytes("NULL")));
    if (amountToWithdraw > infStruct.withdrawableRevenue){
        emit insufficientFunds(amountToWithdraw, infStruct.withdrawableRevenue, infStruct.infName, infStruct.infWallet);
        require(amountToWithdraw <= infStruct.withdrawableRevenue);
    } else if(amountToWithdraw <= infStruct.withdrawableRevenue) {
        infStruct.infWallet.transfer(amountToWithdraw);
        updateInfWithdrawableAmount(infStruct, amountToWithdraw);
        emit successfulWithdrawal(infStruct.infName, infStruct.infWallet, "The funds were successfully withdrawn into the infrastuctures' wallet!");
    }
}
```

The "withdrawRevenue" function allows infrastructure wallets to withdraw their revenue from the contract, subject to the available withdrawable amount. If the requested amount is greater than the available withdrawable revenue, an event is emitted and the withdrawal fails. Otherwise, the requested amount is transferred to the infrastructure wallet, and the available withdrawable amount is updated.

checkHowMuchIOwe-

```
function checkHowMuchIOwe(string memory infName) public {
    individualUser memory currUser = findUserInDataBase(payable(msg.sender), infName);
    infrastructure memory currInf = findInfWallet(infName);
    require(keccak256(bytes(currUser.userFirstName)) != keccak256(bytes("NULL")), "We could not find the user!");
    require(keccak256(bytes(currInf.infName)) != keccak256(bytes("NULL")), "We could not find the current infrastructure..");
    uint userCost = getUserCost(currUser, currInf);
    emit sendCost("The amount you owe is presented below: ",  userCost);


}
```

The "checkHowMuchIOwe" purpose is to allow individual users to check how much they owe for the infrastructure services provided by a particular infrastructure wallet.

To achieve this, the function first finds the individual user making the payment and the infrastructure wallet by calling the "findUserInDataBase" and "findInfWallet" functions, respectively. It then checks whether the user and infrastructure exist in the database.

Next, the function calculates the cost for the user by calling the "getUserCost" function with the user and infrastructure as arguments. It emits an event called "sendCost" with the string "The amount you owe is presented below: " and the user's cost as arguments.

It is important to note that this function does not transfer any funds or update any information in the database. It only retrieves information and displays it to the user.

IndUsrpayInfrastructureWallet-

```
function IndUsrpayInfrustuctureWallet(string memory infName) public payable {
    //require(msg.sender == X );
    individualUser memory currUser = findUserInDataBase(payable(msg.sender), infName);
    infrastructure memory currInf = findInfWallet(infName);
    require(keccak256(bytes(currUser.userFirstName)) != keccak256(bytes("NULL")), "We could not find the user!");
    require(keccak256(bytes(currInf.infName)) != keccak256(bytes("NULL")), "We could not find the current infrastructure..");
    uint userCost = getUserCost(currUser, currInf);
    require(msg.value == userCost);
    updateCurrInfRevenue(msg.value, currInf);
    this.deposit(address(this));
    updateUserType(currUser, currInf);
    updateUserTotalPayed(currUser, userCost);
    emit userPaymentSuccess(userCost, currUser.userFirstName, currUser.userLastName,currInf.infName);
    }
```

The "IndUsrpayInfrustuctureWallet" function is designed to enable individual users to pay for infrastructure services provided by a particular infrastructure wallet.To achieve this, the function first finds the individual user making the payment and the infrastructure wallet by calling the "findUserInDataBase" and "findInfWallet" functions, respectively. It then checks whether the user and infrastructure exist in the database. Next, the function calculates the cost for the user by calling the "getUserCost" function with the user and infrastructure as arguments. It checks whether the amount of ether sent by the user matches the calculated cost.

If the payment amount is correct, the function updates the revenue of the current infrastructure by calling the "updateCurrInfRevenue" function with the amount of ether sent and the infrastructure as arguments. It then transfers the ether to the contract address by calling the "deposit" function with the address of the contract as the argument. The function then updates the user's type and the total amount paid by the user by calling the "updateUserType" and "updateUserTotalPayed" functions, respectively, with the user and infrastructure as arguments. Finally, the function emits an event called "userPaymentSuccess" with the cost, user's first and last name, and infrastructure name as arguments.

Roadmap-

- ***Done so Far:***
  - Design project protocol + token requirements & develop our dApp.
  - Implement protocols for the lending process between lenders & public contractors.
  - Model how retail investors will receive incentives from providing liquidity to a loan pool.
  - Integrate loan terms (repayment schedule, interest, & more)
  - Refine our very simple dApp user interface

- ***Future Milestones:***
  - Tokens can be transacted through our dApp network to access certain tolled infrastructure (i.e. road tolls, bridges, ferries, etc).
  - Integrate additional DeFi loan protocols
  - Explore tokenomic models & bonding curves
  - Implement cross-chain investment bridge protocols

Document and report all project details

Interaction between Front End and Back end would be achieved via REST API. By having both these sides running on different machines, project would have this positive sides:

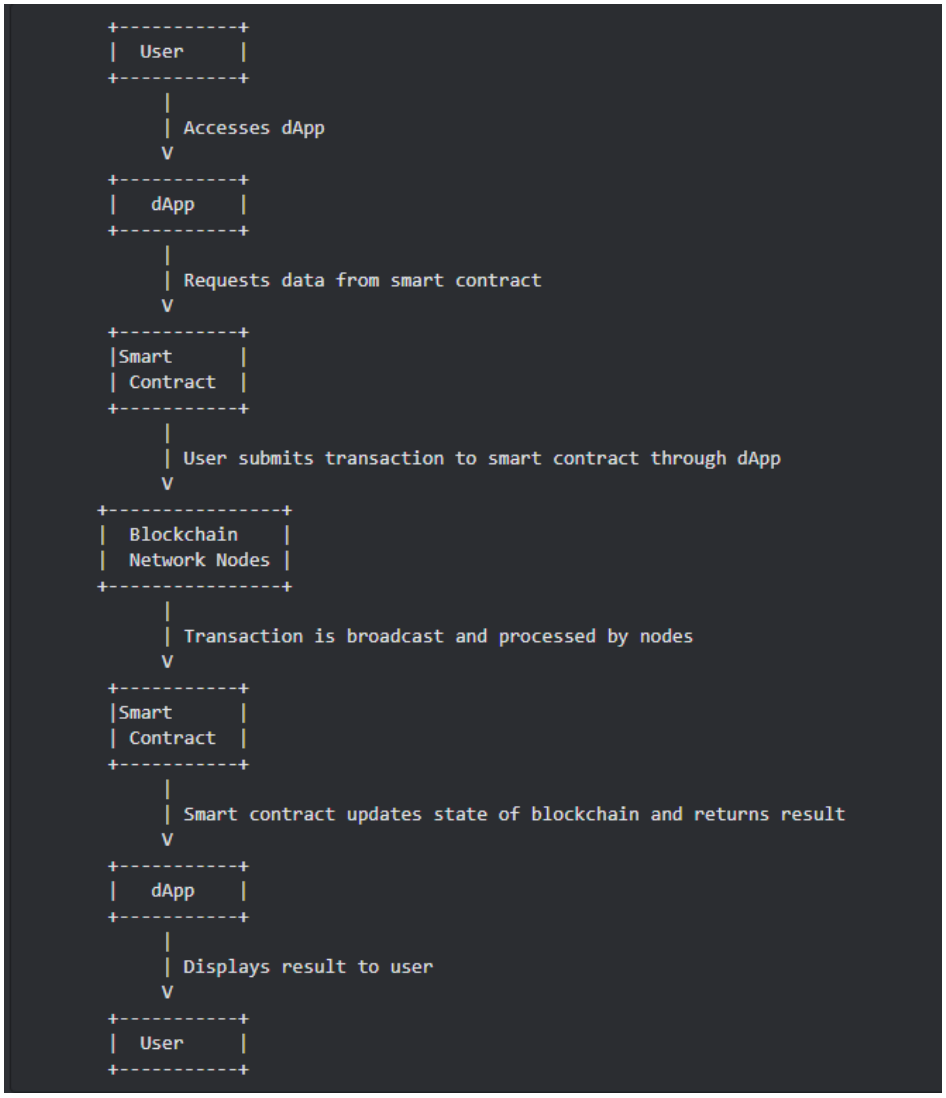
Better organization: Separating the frontend and backend code into different files and directories makes it easier to find and modify code as the project grows.

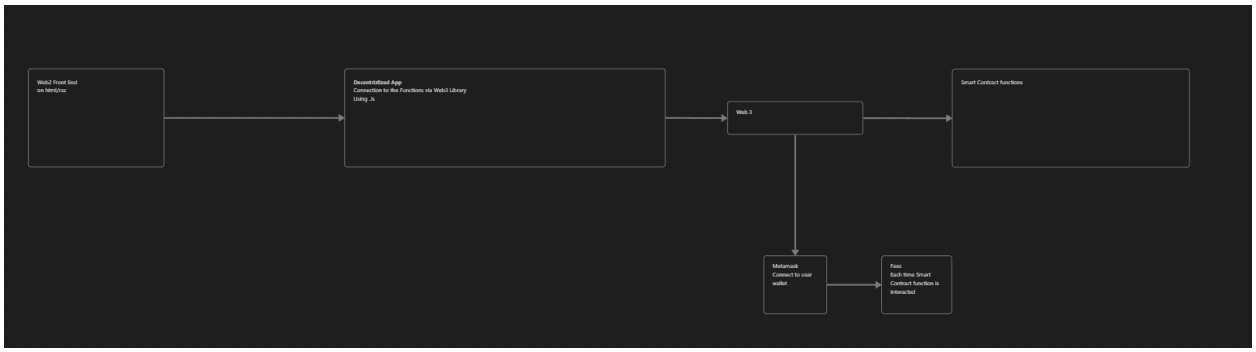
Code reusability: Separating the code into distinct modules and files allows for better code reusability, which can save time and effort in the long run.

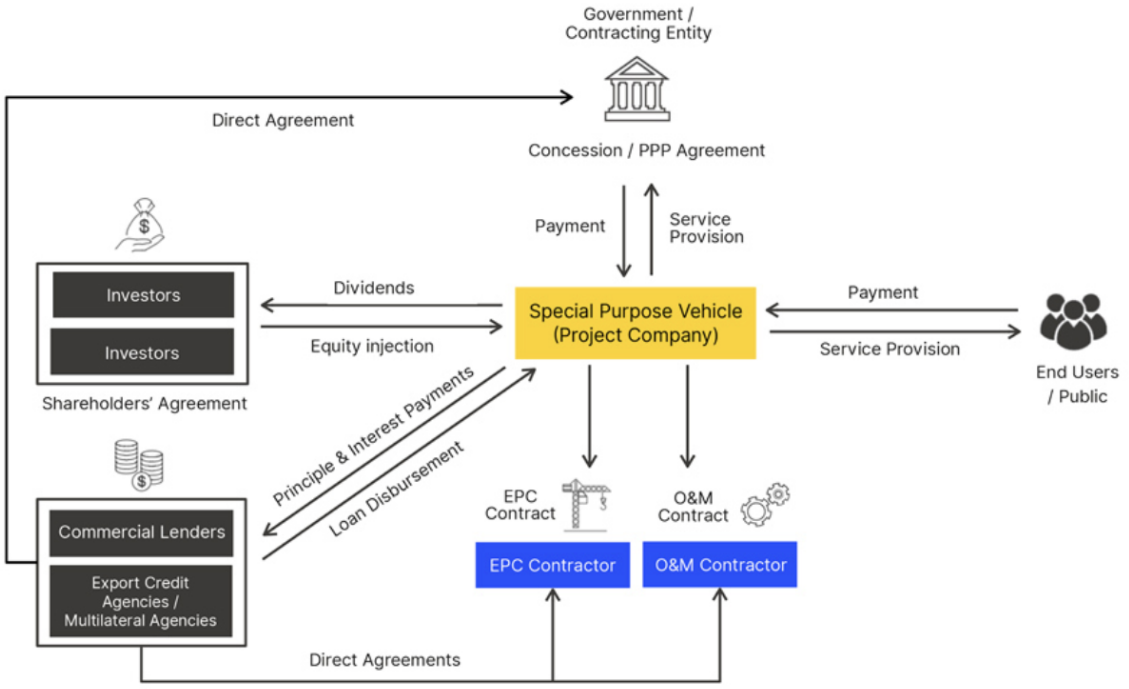
Scalability: Separating the frontend and backend components allows for better scalability, as it makes it easier to add new features and functionality to the dApp over time.

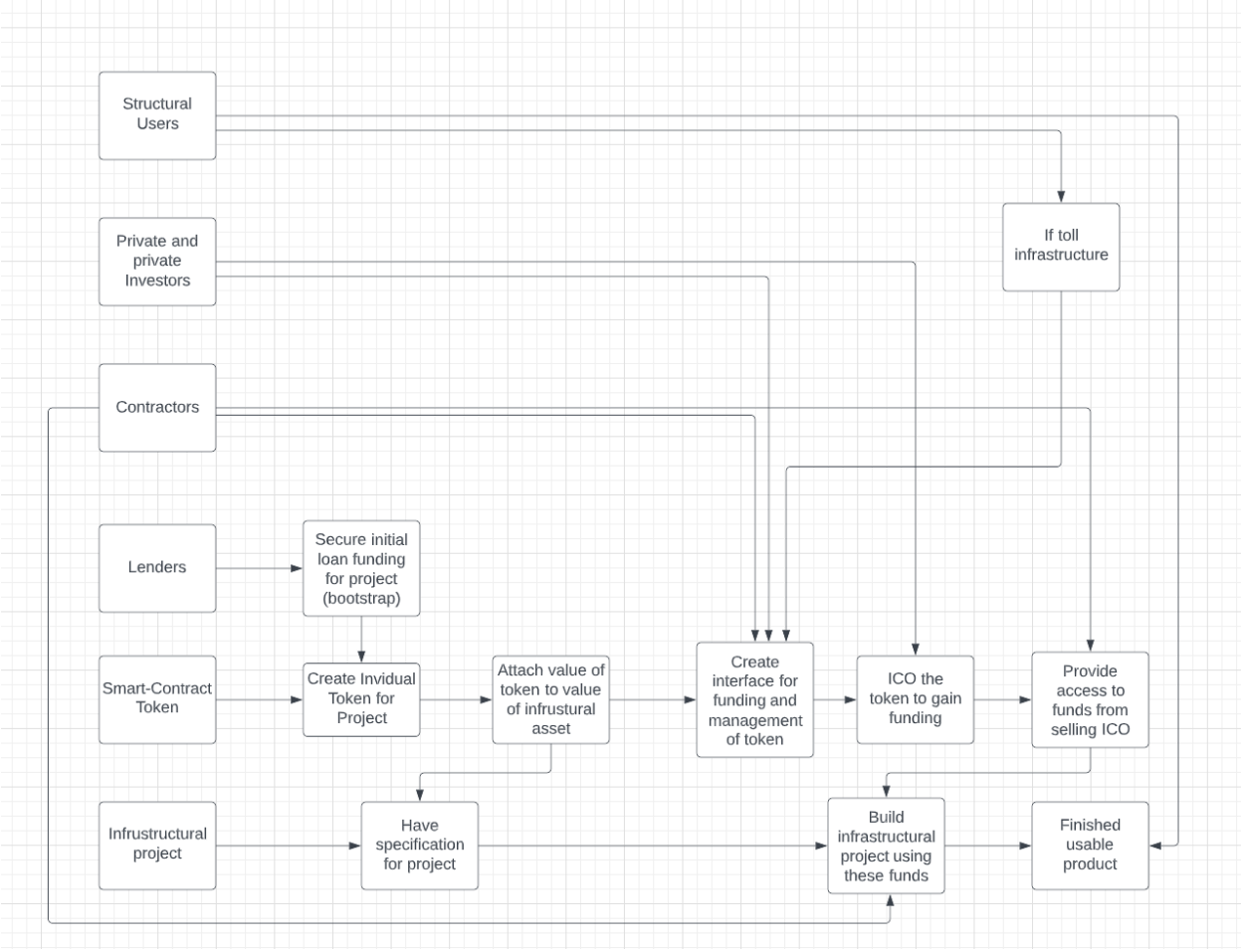
Security: Separating the backend code from the frontend code can help improve the security of the dApp, as it can prevent unauthorized access to sensitive data and code.

Models and Design & Documents






Government / Contracting Entity
Direct Agreement
Concession / PPP Agreement
Payment
Service Provision
Investors
Investors
Shareholders' Agreement
Dividends
Equity injection
Special Purpose Vehicle (Project Company)
Payment
Service Provision
End Users / Public
Principle & Interest Payments
Loan Disbursement
Commercial Lenders
Export Credit Agencies / Multilateral Agencies
EPC Contract
O&M Contract
EPC Contractor
O&M Contractor
Direct Agreements


Structural Users
Private and private Investors
Contractors
Lenders
Secure initial loan funding for project (bootstrap)
Smart-Contract Token
Create Invidual Token for Project
Attach value of token to value of infrustural asset
Create interface for funding and management of token
ICO the token to gain funding
Provide access to funds from selling ICO
If toll infrastructure
Infrustructural project
Have specification for project
Build infrastructural project using these funds
Finished usable product

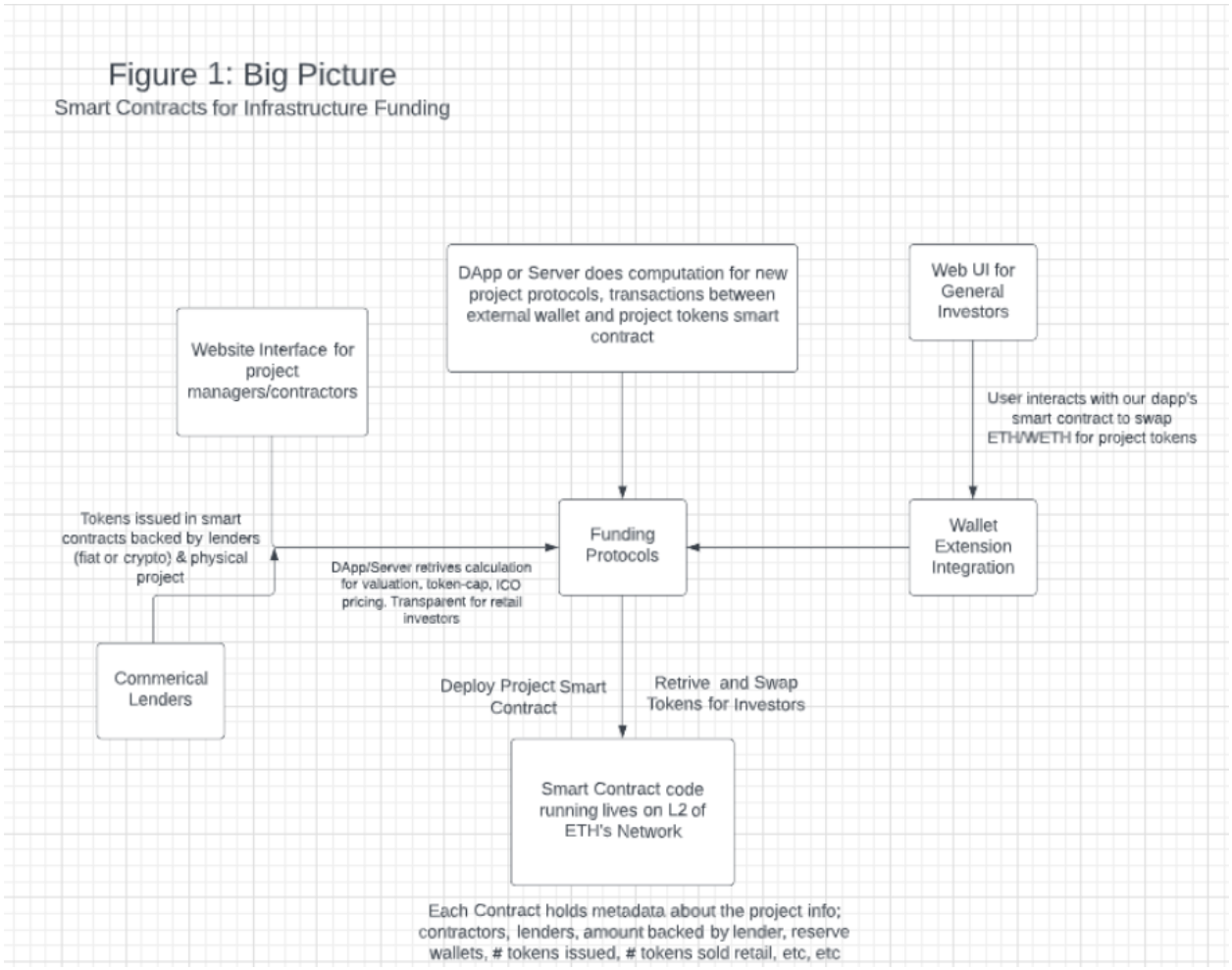
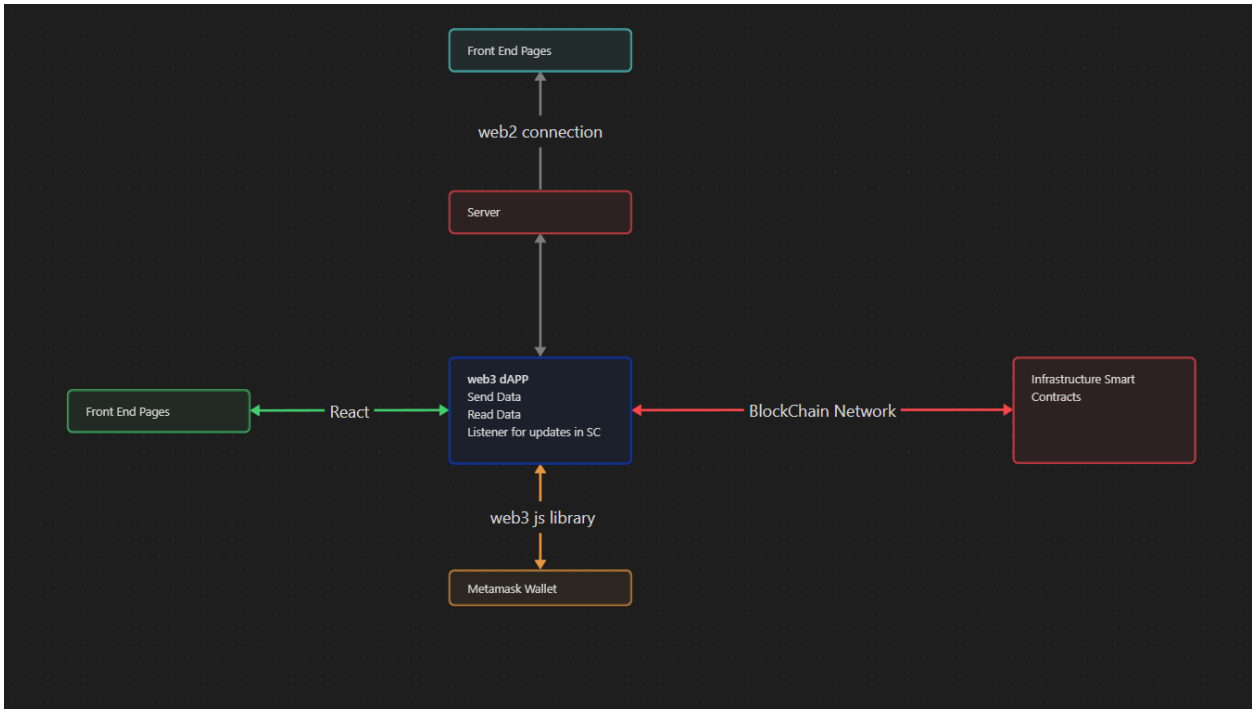
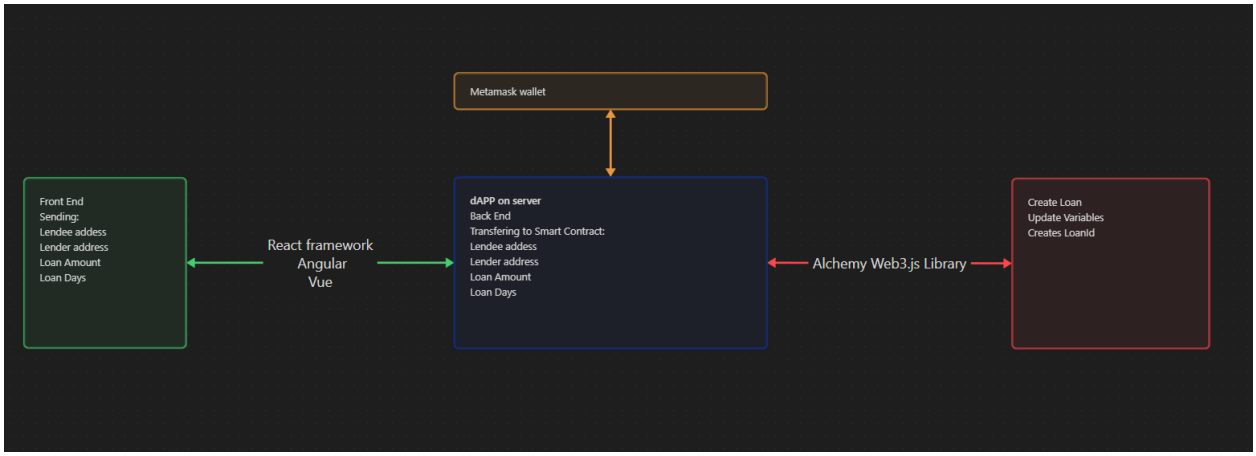
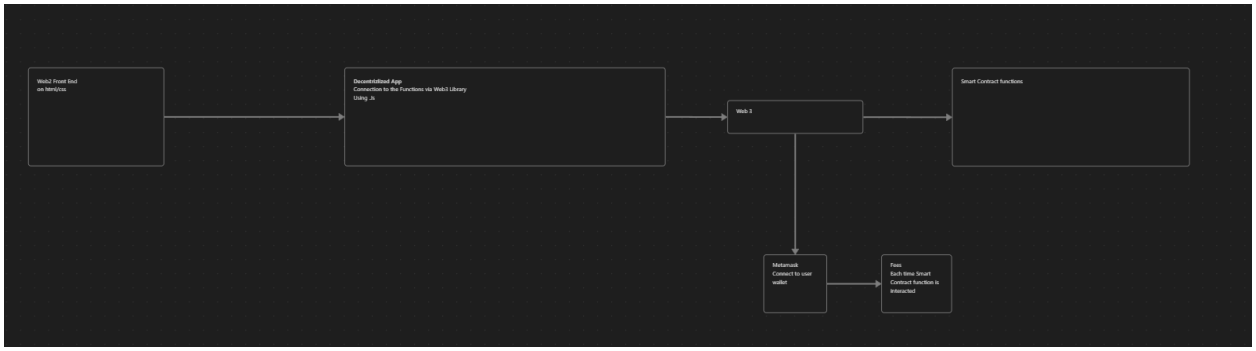
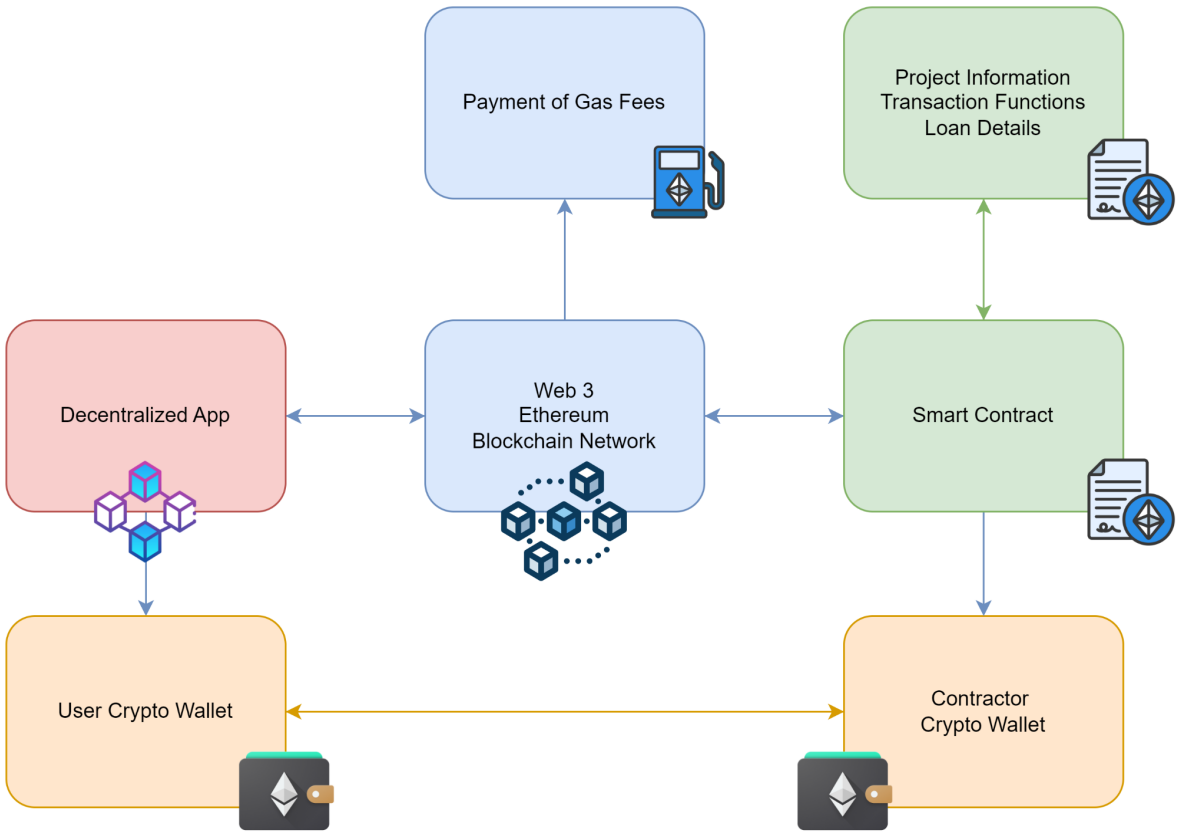


Figure 1: Big Picture
Smart Contracts for Infrastructure Funding
DApp or Server does computation for new project protocols, transactions between external wallet and project tokens smart contract
Web UI for General Investors
Website Interface for project managers/contractors
User interacts with our dapp's smart contract to swap ETH/WETH for project tokens
Tokens issued in smart contracts backed by lenders (fiat or crypto) & physical project
Funding Protocols
Wallet Extension Integration
DApp/Server retrives calculation for valuation, token-cap, ICO pricing. Transparent for retail investors
Commerical Lenders
Deploy Project Smart Contract
Retrive and Swap Tokens for Investors
Smart Contract code running lives on L2 of ETH's Network
Each Contract holds metadata about the project info; contractors, lenders, amount backed by lender, reserve wallets, # tokens issued, # tokens sold retail, etc, etc


Front End Pages
web2 connection
Server
Front End Pages
React
web3 dAPP
Send Data
Read Data
Listener for updates in SC
BlockChain Network
Infrastructure Smart Contracts
web3 js library
Metamask Wallet


Metamask wallet
Front End
Sending:
Lendee addess
Lender address
Loan Amount
Loan Days
React framework
Angular
Vue
dAPP on server
Back End
Transfering to Smart Contract:
Lendee addess
Lender address
Loan Amount
Loan Days
Alchemy Web3.js Library
Create Loan
Update Variables
Creates LoanId




Payment of Gas Fees
Project Information
Transaction Functions
Loan Details
Decentralized App
Web 3
Ethereum
Blockchain Network
Smart Contract
User Crypto Wallet
Contractor
Crypto Wallet

Useful Links-

https://www.web3.university/tracks/create-a-smart-contract/integrate-your-smart-contract-with-a-frontend

Link to project repo-

https://github.com/michaelgadda/CS46X_ETH_SMART_CONTRACTS