Katie Bramlett, Claire Furtick, Ada Kilic, Genevieve Flynn
12/10/21

Architecture & Design Assignment 2

System Overview

Tenants and landlords normally use a traditional lease agreement to protect their interests during the lease and use third-party payment instruments for the payment transaction process. The issues with this approach are that it relies on the tenant to remember to make timely and correct payments to the landlord and it is not able to completely store the transaction records generated during the lease period. Therefore, it is easy for both parties to generate inevitable disputes that must be solved through a third party. Blockchain technology can solve the problem of centralized access control and not being able to systematically manage the leasing process. Blockchain uses peer-to-peer technology, which makes it possible to implement decentralized applications. This is used to ensure that the stored block information cannot be tampered with.

Our solution is an Ethereum smart contract for rental contracts with a decentralized application (dApp) front-end that the user will interact with. By utilizing blockchain technology, specifically an Ethereum-based application, we are alleviating the tedious and complicated process of paying rent to a landlord. Traditionally, landlords must rely on the tenant to make the monthly payment, whereas a smart contract is binded by the code that is written. Additionally, there are numerous benefits of smart contracts including autonomy, security, not needing to trust third parties, cost-effectiveness by removing intermediaries, fast performance, accuracy, fairness, and visibility. It also offers more payment flexibility for those who want to use cryptocurrency. Our users will be tenants and landlords who are interested in a rental contract, especially those who are interested in alternative cryptocurrency payment methods as they become more widely popular. Their use case would be to utilize the smart contract for the rental contract payment. Through our dApp, they can navigate through starting a rental contract, checking existing contracts, or terminating a contract.

C4 Model Software and Component Diagrams

Figure 1 - Context

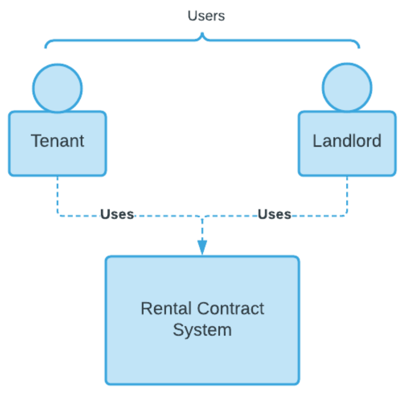


Figure 2 - Containers

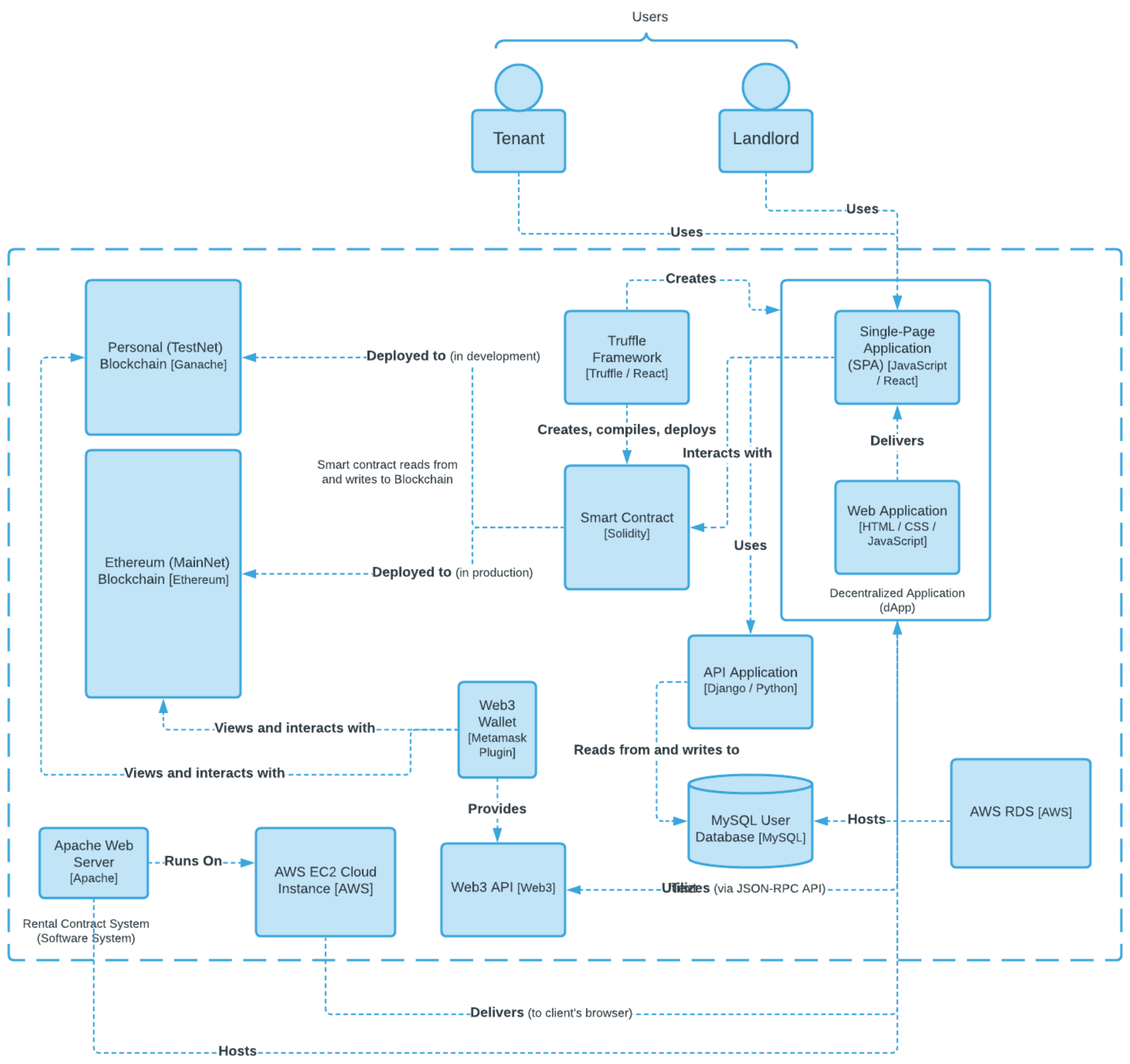
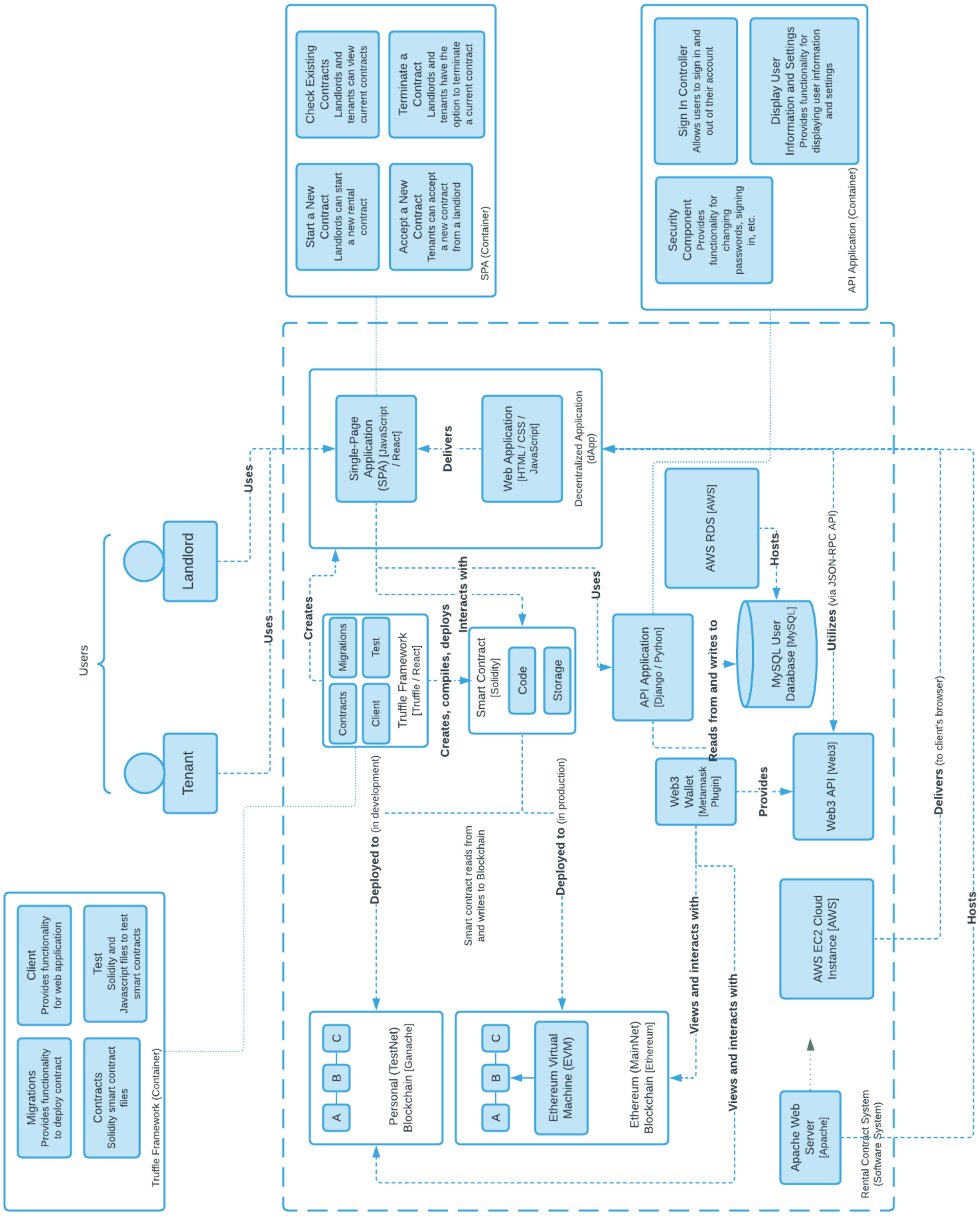

Users
Tenant
Landlord
Uses
Uses
Creates
Personal (TestNet) Blockchain [Ganache]
Deployed to (in development)
Truffle Framework [Truffle / React]
Single-Page Application (SPA) [JavaScript / React]
Creates, compiles, deploys
Interacts with
Delivers
Smart contract reads from and writes to Blockchain
Smart Contract [Solidity]
Uses
Web Application [HTML / CSS / JavaScript]
Ethereum (MainNet) Blockchain [Ethereum]
Deployed to (in production)
Decentralized Application (dApp)
API Application [Django / Python]
Web3 Wallet [Metamask Plugin]
Views and interacts with
Views and interacts with
Reads from and writes to
Provides
MySQL User Database [MySQL]
Hosts
AWS RDS [AWS]
Apache Web Server [Apache]
Runs On
AWS EC2 Cloud Instance [AWS]
Web3 API [Web3]
Utilizes (via JSON-RPC API)
Rental Contract System (Software System)
Delivers (to client's browser)
Hosts

Figure 3 - Components



Users
Landlord
Tenant
Uses
Uses
Single-Page Application (SPA) [JavaScript / React]
Delivers
Web Application [HTML / CSS / JavaScript]
Decentralized Application (dApp)
Creates
Migrations
Test
Contracts
Client
Truffle Framework [Truffle / React]
Creates, compiles, deploys
Interacts with
Smart Contract [Solidity]
Code
Storage
Uses
API Application [Django / Python]
AWS RDS [AWS]
Hosts
MySQL User Database [MySQL]
Reads from and writes to
Utilizes (via JSON-RPC API)
Web3 Wallet [Metamask Plugin]
Provides
Web3 API [Web3]
Delivers (to client's browser)
Hosts
Deployed to (in development)
Smart contract reads from and writes to Blockchain
Deployed to (in production)
Views and interacts with
Views and interacts with
AWS EC2 Cloud Instance [AWS]
Apache Web Server [Apache]
Rental Contract System (Software System)
A
B
C
Personal (TestNet) Blockchain [Ganache]
A
B
C
Ethereum Virtual Machine (EVM)
Ethereum (MainNet) Blockchain [Ethereum]
Check Existing Contracts Landlords and tenants can view current contracts
Terminate a Contract Landlords and tenants have the option to terminate a current contract
Start a New Contract Landlords can start a new rental contract
Accept a New Contract Tenants can accept a new contract from a landlord
SPA (Container)
Sign In Controller Allows users to sign in and out of their account
Display User Information and Settings Provides functionality for displaying user information and settings
Security Component Provides functionality for changing passwords, signing in, etc.
API Application (Container)
Client Provides functionality for web application
Test Solidity and Javascript files to test smart contracts
Migrations Provides functionality to deploy contract
Contracts Solidity smart contract files
Truffle Framework (Container)

At the context level diagram (Figure 1), two types of users (landlord and tenant) will interact with the rental contract system.

Digging further under the hood, we have the container level diagram (Figure 2). Here, we have the pipeline from the user to the Ethereum blockchain, utilizing the dApp, smart contract, and necessary API and frameworks.

Finally, at the most detailed level, we have the component level diagram (Figure 3). First, the Truffle Framework is broken down into four main components, which are Migrations, Contracts, Client, and Test -- each of which has a specific role in generating the smart contract and dApp pages. Next, the Smart Contract consists of code and storage, and the Single-Page Application (SPA) consists of four main components for each use case -- starting a new contract, checking current contracts, accepting a new contract, and terminating existing contracts. Lastly, the API Application has three main components, which are the security component, the sign in controller, and the display of user information and settings.

Programming Languages

Decentralized Application (dApp): JavaScript / React
Smart Contract: Solidity
User Database: MySQL

Architecture Choices

Our dApp will have a layered software architecture. The following layers will be specified:

- Presentation Layer: pages for each user, the home page, tenant and landlord pages, our UI capabilities (CSS/HTML, etc)
- Business Layer: business logic for the contract will go here, the functionality and rules that are binding, customer objects
- Persistence Layer: the mapping of our database information to user objects
- Database Layer: datastore for our objects

On an individual level, the architecture of the smart contract is event-driven. The contract is created and deployed to the blockchain when a submission (and acceptance) button is clicked. After this, depending on the terms of the rental contract, the smart contract will execute a transaction (payment of x amount of money from the tenant to the landlord) a certain amount of times, where this transaction is triggered by a timer/the date (monthly). The contract will cease executing monthly transactions after either a) the amount of times specified in the contract has occurred, or b) the contract is prematurely terminated by one of the parties.

Like other browser extensions, the Metamask wallet has a microkernel (plug-in) architecture. It runs as a browser extension in Chrome, Firefox, or Edge. The dApp utilizes Metamask as an Ethereum wallet and web3 provider to interact with the Ethereum blockchain.

Software Design Patterns

We are using a state-based software design pattern. The internal state of our object will relate to a contract between two parties status on the blockchain as well as whether it is in use or not. Also, we will be using states to keep track of our Web3 instance used to interact with the blockchain.

Also, we are using React, and have one central JavaScript file for the “main page”, and then the individual components of the web application (i.e. nav bar, login “page”, create new contract “page”, etc.) are all separate JavaScript files that represent an individual React component. Then, in the main JavaScript file, we will use state variables (as described above) to determine which React components should be displayed.

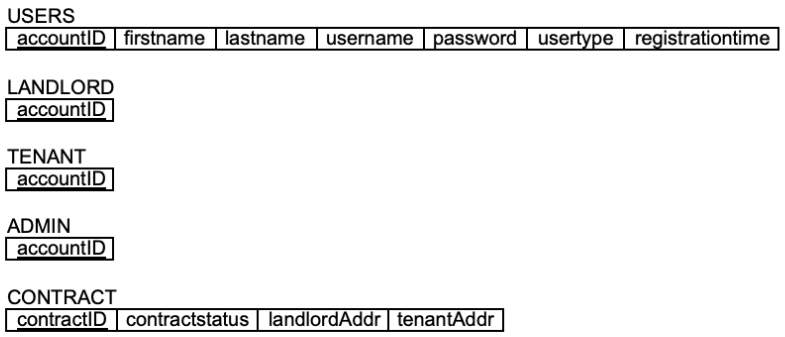
We will also be using scheduling design patterns to deal with the logic of our contract charging on a time-based schedule. This will ensure that the users can set up monthly, recurring payments under one contract.

Inter-Component Protocols, File Formats, and Data Formats/Schemas

Inter-Component Protocols:
The decentralized application interacts with the Web3 API through the JSON-RPC protocol. The user requests the web application through the HTTPS protocol.

File Formats:
The App.js main JavaScript file imports the Solidity smart contract functions through importing the .json file for the smart contract at the top of the file. This allows all back-end smart contract functions to be called in the front-end JavaScript file.

User Database Schema:
The MySQL user database is composed of five tables. The diagram below displays the five tables with their corresponding columns.



USERS

accountID	firstname	lastname	username	password	usertype	registrationtime

LANDLORD

accountID

TENANT

accountID

ADMIN

accountID

CONTRACT

contractID	contractstatus	landlordAddr	tenantAddr

Additionally, the MySQL user database will store all of the values that cannot be stored on the blockchain. An example for the use of the MySQL user database is as follows:

If a user would like to signup and login the following example value will be inserted into the database ->
'0xb704f5ea0ba39494ce839673fffba7429957968','Katie','Love','slove@gmail.com','cryptozzz','tenant', '2008-01-01 00:00:01')
Furthermore, the MySQL database will store information on the rental contract as well. This will include values such as what the status of the contract is. For instance if the contract is initiated etc.
The MySQL user database will also include information for an administrator. The current administrators will only include the four of our team members as admins.

External Services

This application was created with a Truffle framework to integrate with the Ethereum blockchain. This gives us an outline for how to structure our development environment, testing framework, and assets, as well as how to interact with the blockchain. The front end uses React as a framework to break our design into reusable components, the framework was integrated with truffle to make it easier to integrate with our smart contract. We will also be using Django as a framework to interact with our databases. In addition, we will use AWS to host our application on an EC2 instance, as well as RDS for our user database.

Context and System Environment

The decentralized application (dApp) is designed to run in a web browser, as a larger desktop provides a better and more efficient experience for the user. Currently, Firefox, Chrome, and Edge are all supported web browsers, since they are supported for the Metamask plug-in. Metamask is an Ethereum wallet and web3 provider used to interact with the Ethereum blockchain. Users access their Ethereum wallet through a browser extension. It provides a fast connection to the Ethereum blockchain and enables secure and usable Ethereum-based web sites.

Datastore Choices

To store user information and settings, a MySQL database will be used. The database will store all user information, as well as account settings such as username and password. This will be queried upon login and logout. Essentially, the MySQL database is for storing anything that will not be stored on the blockchain. The decentralized application (dApp) will utilize the smart contract to read from and write to the blockchain. Additionally, we will be writing an API using the Django framework, which the dApp will utilize to read from and write to the user database. Also, the blockchain will store information about the contract transactions that can be read from.

Physical Layer

For the physical layer, we will use an AWS EC2 cloud instance and an AWS RDS cloud instance. On the AWS EC2 instance, we will install an Apache server on a Linux operating system (Ubuntu 18.04). All front-end and back-end source code will be hosted in the AWS EC2 cloud instance. Our single page application will run in the client's browser, being delivered to the browser through EC2. On the AWS Relational Database System (RDS) instance, we will host our user database, which will be written in MySQL.

Core Algorithms

The core algorithm of our application is the smart contract algorithm for executing multiple timely transactions. This will be found in the Solidity file. For this algorithm, we must determine how to instantiate a contract between a landlord and a tenant (the contract must be approved by both parties before being instantiated). After it is created, payments must be executed monthly. The amount of the monthly payment and the number of months the payments occur will be inputted by the user during contract creation. The send method that sends ethereum from one account to another is already built in, but a timer method that triggers a transaction on a timer tick is not, so we must create this ethereum method. Then, after the proper number of transactions have occurred, the contract will be terminated. Once we get this algorithm working, a potential next step would be figuring out how to have the initial contract creation and keeping track of number of transactions that have occurred still execute on the blockchain, but then have the recurring payments execute off chain in order to avoid the transactions fees that come along with executing transactions on the blockchain.

System Data Input and Output Mechanisms

The decentralized application (dApp) will contain a form for the creation of a smart contract. The user will input information into this form, and upon submission, a contract will be generated. The fields of the form will include the smart contract address, the addresses for both the landlord and the tenant, the gas fee, and the amount sent (in Eth and USD equivalent). The gas fee and amount sent and fields will take text as input, while the smart contract address and user addresses will be a drop down menu. We will provide tips and feedback to the user through validation checks on the form. As output, upon submission of the form, the dApp will send the user a confirmation message for the transaction. Additionally, it will store a ledger of the transaction in the blockchain, and can be referenced from the current contracts page.

Runtime Configurations

We will need to build the front-end application and the output files in the public folder within our framework. We will need to compile and migrate the smart contract before we can interact with it through the web application.

Team Member Responsibilities

We have split up responsibilities so that two team members will be focused on the front-end decentralized application (dApp) design and two team members will be focused on the back-end smart contract and user database development.

Claire and Genevieve will be working together on the front-end decentralized application (dApp) design. The front-end dApp will display different pages depending on the user (landlord or tenant). For example, only landlords will be able to start new contracts, and only tenants will be able to accept a new contract. Both users will be able to view existing contracts and terminate an existing contract. Claire and Genevieve will prototype the dApp desktop design using Figma. Claire will be responsible for the How-To, FAQ, and documentation pages, as well as the contract creation form page. Genevieve will be responsible for the contract viewing and terminating page, as well as helping Claire with documentation.

Ada and Katie will be working together on the back-end smart contract and user database development. Ada will be responsible for implementing the user database schema and design, as well as writing the API through Django for querying the user database. Katie will be responsible for writing the Solidity smart contract code that will create a new contract, allow the user to accept the contract, and query the blockchain to display current contracts.

Timeline

By December 27th:

- Set up AWS EC2 instance and AWS RDS user database
- Create user sign up page
- Determine how to have users see different page options depending on user type (with pre-populated dummy data for now)

By January 3rd:

- Implement user sign up and login functionality
- Implement functionality for logged in users to attach wallets to their accounts (determine how the information entered about the wallets stored and then used when initiating payments between the users)
- Implement functionality for a contract to execute x amounts of timely payments

By January 10th:

- Implement tenant acceptance of a contract (front-end page and back-end functionality)
- Complete database API for querying the user database
- Create the “About this dApp” page

By January 23rd:

- Create landlord and tenant “view existing contracts” pages with option to view details, terminate a contract, etc...
- Complete backend solidity functions that allow us to get informations about existing contracts from the blockchain
- Create user settings page