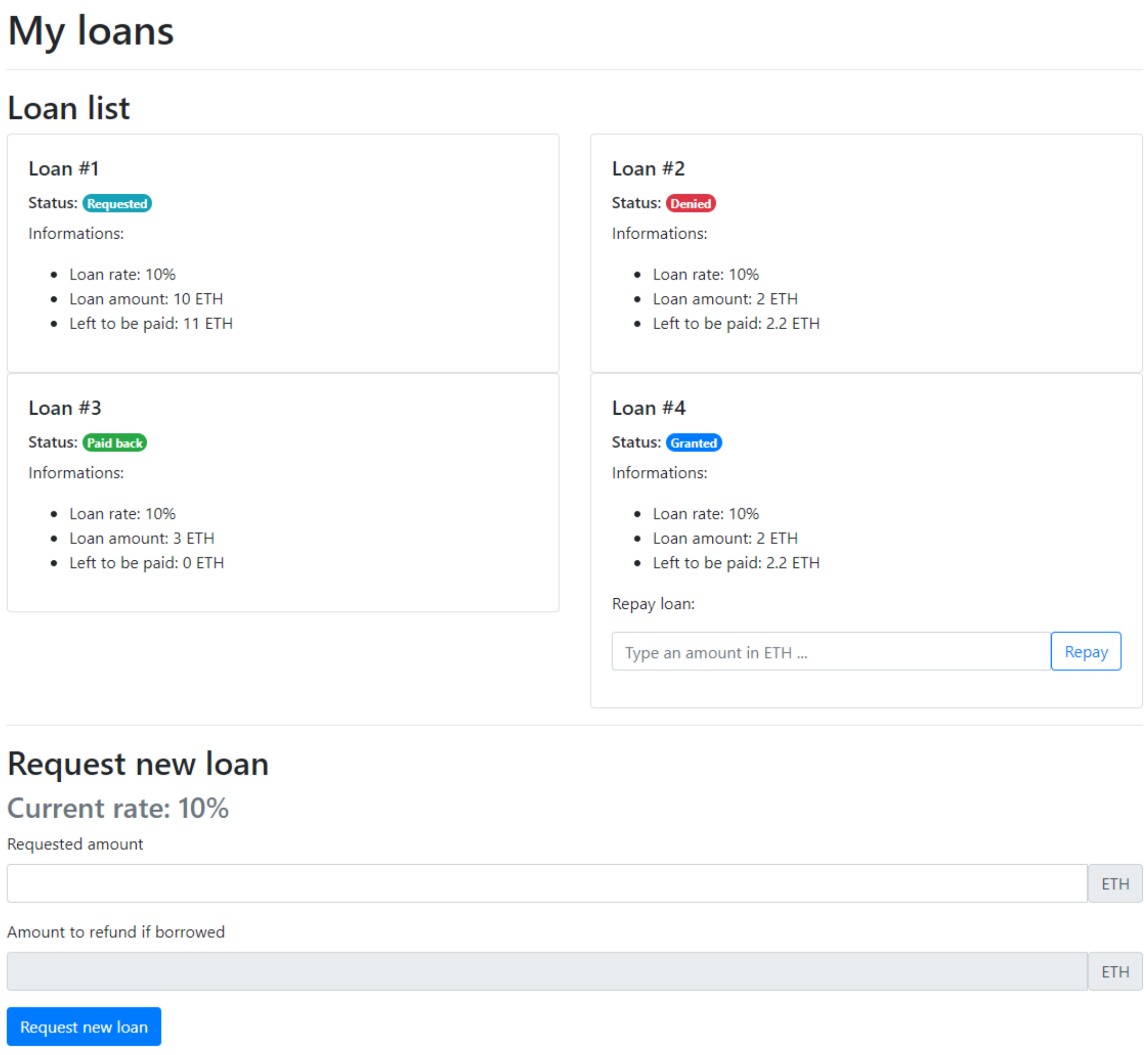
Course #2 - Exercise

Create a frontend application to interact with your smart-contract

Context

In the last course, you have learned about how to develop, compile and deploy a smart-contract in a fake blockchain using Remix. Following that, you learned how to leverage Metamask, Ganache, Truffle and you briefly saw the possibilities of Web3.

This exercise allows you to put everything you learned in practice to build a frontend application that is able to access your smart-contract instance, and leverage it. Final result will look like the following:



My loans

Loan list

Loan #1
Status: Requested
Informations:
- Loan rate: 10%
- Loan amount: 10 ETH
- Left to be paid: 11 ETH

Loan #2
Status: Denied
Informations:
- Loan rate: 10%
- Loan amount: 2 ETH
- Left to be paid: 2.2 ETH

Loan #3
Status: Paid back
Informations:
- Loan rate: 10%
- Loan amount: 3 ETH
- Left to be paid: 0 ETH

Loan #4
Status: Granted
Informations:
- Loan rate: 10%
- Loan amount: 2 ETH
- Left to be paid: 2.2 ETH

Repay loan:

Request new loan
Current rate: 10%
Requested amount

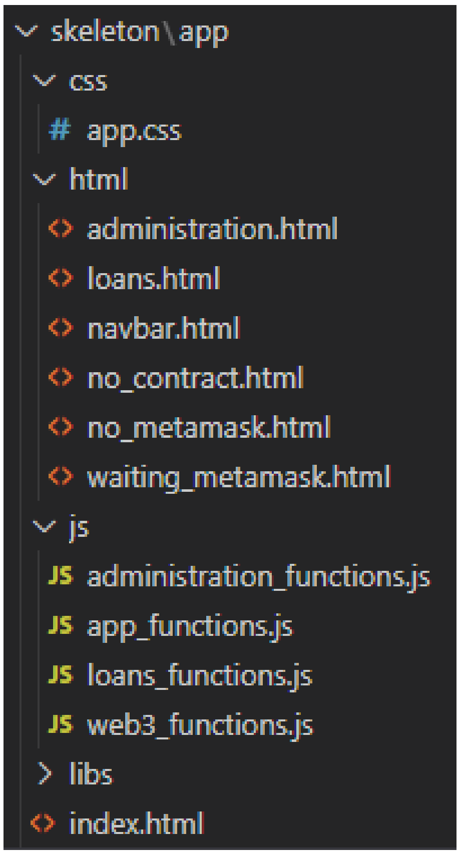
Amount to refund if borrowed

The frontend will be a simplified app, built in JQuery & Bootstrap. You will not need extensive knowledge in JavaScript or JQuery to complete the exercise.

To construct the app, you will rely on a skeleton app that simplifies your task. This skeleton already imports JQuery and Bootstrap for you, and setups Metamask as your provider. Thus, you will not have to recode everything from scratch : just fill function skeletons and add JS logic to skeleton pages if needed. You are free to modify the skeleton at will, as long as all contract functionalities are present in your app.

Note that the provider is already set up for you : just use web3 global variable wherever you have the need to.

The skeleton app is splitted as the following:



```
skeleton\app
  css
    app.css
  html
    administration.html
    loans.html
    navbar.html
    no_contract.html
    no_metamask.html
    waiting_metamask.html
  js
    administration_functions.js
    app_functions.js
    loans_functions.js
    web3_functions.js
  libs
  index.html
```

- *css*: contains a single app.css file, to style your application.
- *html*: contains views of your application (views import js files to function, already done in the skeleton).
- *js:* contains the logic of your application. Each page has its own js file, and a web3 js file is dedicated to make blockchain requests.
- *libs*: contains JQuery and Bootstrap libraries (already imported).
- *index.html*: the entry point, set up Metamask as our provider to interact with the blockchain.

First step : deploy the contract on Ganache

At the moment, you should have a Ganache instance running and Truffle installed. Let's deploy the contract on-chain !

1. Take a bit of time to explore the skeleton folder, notably truffle-config.js. You'll see what files are used by Truffle to deploy your contract on Ganache, as presented during the course.

2. Open a terminal and use the command *truffle migrate* to deploy the contract on Ganache. If it worked, a *build* folder should appear inside the *skeleton* folder, containing ABIs.

Second step : code user functions to request loans

Let's begin slowly by implementing user functions. You are free to code your own functions to perform expected features, but placeholders are here to help you. Basically, you have to use the views to call your logic functions, and use web3 functions to interact with the blockchain for your logic functions.

1. Complete the web3 function *getCurrentAccount* to retrieve the current Metamask account using the web3 global variable.

2. Use this function and the existing function named *getLenderContractObj* to code the *requestLoan* function that sends a loan request using the current account in Metamask.

3. Use the form inside *loans.html* to call this function with the right parameters. *You can set onClick events for form buttons and access form values using JQuery and form elements IDs.*

4. (Optional) Code and use the web3 *getCurrentRate* function to display the rate on the front page, and use the input with the id *input-torefund* to display the amount of Ether the user will have to repay, depending on the amount typed in the request form.

5. Display all your loans by implementing the logic function *displayLoans* that calls the web3 function *getUserLoans* to retrieve all your loans (whatever the state they are in), and then display them as html *divs* (or even better, Bootstrap cards).

Third step : code administration functions

In this third step, we will develop functions for the contract owner to manage requests to come and fund its contract. But before all of that, we must modify our skeleton to not display the administration page if you are not the contract owner. It requires to find out, when loading the app, if the current Metamask account is the contract owner or not.

1. Complete the web3 function *checkIfContractAccountIsOwner* that returns true if yes, false if not. Use this function to protect the administration page, in the navbar view *(This is not the best practice to protect a page, but for the sake of this exercise we're focusing on web3 rather than complex JS mechanisms)*.

2. Display the current contract balance by implementing a web3 function that gets the current balance from the contract, then a logic function that modifies the html file to display it.

3. Use the form inside administration.html to fund contract or retrieve funds from contract. One button will be used to send the amount of Ether typed in the input from the contract owner to the contract, the other will do the opposite.

4. Complete the web3 function *getLoanRequests* to retrieve open requests (you should have some if you tried your functions in the last step), and the *displayLoanRequests* function inside *administration_functions.js* to retrieve the loans from the contract, then display them using JQuery.

5. For each displayed request, add two buttons *Accept* and *Deny* to manage loans. Code the according logic (buttons handlers in the logic js file, web3 request in the web3 js file).

Fourth step : manage reimbursements

Now that you can request loans and accept or deny it, it is time to code the reimbursement features.

1. Modify your *displayLoans* function to also display an input and a button, only for loans in *Granted* state.

2. Implement a logic function to handle the click on the button, and a web3 function to send money from the user to the contract to repay the loan. This function will be called by the handler using the amount specified in the input.