**Step 2**

**Project intro**

Our project describes a road system that will be represented by 2 overarching smart contracts:

* Regulator
* TollBoothOperator

These other elements of the system are represented by externally owned accounts:

* owner of Regulator
* owner of TollBoothOperator
* individual vehicles
* individual toll booths

**Route price**

The price of a route will be determined by 3 variables:

* the entry toll booth
* the exit toll booth
* the vehicle type

These variables will be used thus:

* the TollBoothOperator defines a base route price from the entry booth to the exit booth. The base route price from the exit booth to the entry booth may be non-existent, equal or different.
* the TollBoothOperator defines a multiplier for each vehicle type. This is the number by which the base route price is multiplied with to get the route price or deposit applicable to a specific vehicle.

**External accounts**

There are a certain number of off-chain actions, such as proof of identity or secret exchange, that we will address only with a wave of the hand in this project. Smart contracts will make these actions possible via public functions, though.

**Vehicles**

We only care about vehicles, not about drivers. It means it can be driven by the vehicle's rightful owner, another driver, or no human driver at all.

* Only vehicles registered with the regulator are allowed to enter the road system.
* Before entering the road system, registered vehicles must make a deposit of at least the amount required by the operator for their vehicle type.
* Upon exit, the exit toll booth will trigger the payment off the deposit and refund the difference to the vehicle.

Additionally,

* This deposit must be accompanied by the address of the toll booth by which they will enter.
* When exiting the road system, the vehicle gives, off-chain, a secret to the exit toll booth.
* The exit toll booth sends this secret to the toll booth operator contract, which is used to unlock the deposit, then pay the toll booth operator the proper fee for the route taken, then refunds the difference to the vehicle.

The off-chain exchange of the secret is handled by a wave of the hand in this project. You are not asked to work on this massive "detail".

**Toll booths**

* When faced with a vehicle that wants to enter the road system, a toll booth will confirm, off-chain, the address of the incoming vehicle.
* If the vehicle has made the proper deposit for its type, the booth opens the gate. This part will be simulated in tests.

Here too, you need not work on the off-chain exchange of address of the incoming vehicles.

**Owner of Regulator**

This account will:

* update the vehicle types in the smart contract.
* ask the smart contract to deploy new TollBoothOperators.
* inform whether an address is a registered TollBoothOperator.
* unregister ToolBoothOperators when needed.

**Owner of TollBoothOperator**

This account will:

* update the base deposit it sees fit.
* update the base price of routes for all toll booth pairs it sees fit.
* update the multiplier of all vehicle types it sees fit.

In particular:

* if a vehicle just completed a route for which there is no base price, the owner should put a price to it before it can unlock the deposit.
* if a vehicle type has no multiplier, then the road system is closed to this vehicle type.

We will simplify the system by overlooking the fact that the owner may disallow a vehicle type *after* such a vehicle has entered the system.

**Smart contracts**

**Regulator**

This contract does 2 main things (studiously avoiding the word *functions* here):

* keep track of the vehicle type for each vehicle address
* deploy new TollBoothOperator

This ensures that:

* because the regulator deploys the toll booth operator, road users are confident they are exchanging with a vetted smart contract.
* an address that has no vehicle type is not registered and should not be allowed onto the road system.

Additionally:

* the regulator collects no fees.
* a type of 0 denotes an unregistered vehicle.
* for mnemonics only, you can assign type 1 for motorbikes, 2 for cars and 3 for lorries.
* the latest type is the valid type, even if the regulator changed the type after the vehicle entered a road system.

**TollBoothOperator**

This contract has many functions which have been parcelled out to its inheritance tree, see its interfaces. The things it does:

* can be paused / resumed to pause the vehicle-facing operations.
* keeps track of the regulator.
* keeps track of the base deposit.
* keeps track of the multipliers of vehicle types.
* keeps track of the toll booths under its purview.
* keeps track of the route base prices.
* accepts deposits from vehicles.
* accepts exit calls from toll booths.
* accepts messages to clear the exit backlog.

For simplicity's sake:

* there is only 1 (or 0) toll booth at a given kilometre on the road. So one booth may have as many gates as there are lanes on the road.
* a base route price of 0 denotes an absence of information.
* a multiplier of 0 denotes an unauthorised vehicle on the road system.
* the entry and exit booths of a route cannot be the same address.
* the TollBoothOperator has total control over the deposit, the base route prices, and the multipliers.
* there is no congestion or pollution charges.
* the latest route base price is the valid route base price, even if the toll booth operator changed the price after the vehicle entered the road system.
* the latest multiplier is the valid multiplier, even if the toll booth operator changed the multiplier after the vehicle entered the road system.

**Charge mechanics**

For a vehicle to be accepted on the road system and the operator to be paid at the end of the route, a little dance is engineered:

* before entering the system, the vehicle deposits the required amount into the TollBoothOperator, and passes along the address of the entry booth and a unique hashed secret of its own choice. The vehicle keeps the secret until the end of the trip.
* when faced with the entry toll booth it mentioned when depositing, it proves its identity off-chain (we talk about this identity proof but you need not implement it), after which the booth opens the gate.
* when exiting the road system at a booth, the vehicle gives the exit toll booth its unhashed secret off-chain. Again, we talk about this off-chain exchange but you do not need to implement it.
* the exit toll booth then submits this secret to the TollBoothOperator, which unlocks the deposit for payment and refund of the difference, if applicable.
* if the fee is equal to or higher than the deposit, then the whole deposit is used and no more is asked of the vehicle, now or before any future trip.
* if the fee is smaller than the deposit, then the difference is returned to the vehicle.
* if the fee is not known at the time of exit, i.e. if the fee is 0, the pending payment is recorded, and one "base route price required" event is emitted, and listened to by the operator's oracle.
* when the oracle receives a new base route price request, it submits the base fee, which also clears one pending payment.
* if there are more than 1 pending payments, an additional function is there to progressively clear the backlog a set number of pending payments at a time in a FIFO manner. If both vehicleA and vehicleB entered at the same booths and exited at the same booths, then if vehicleA exited before vehicleB, then vehicleA should pop from the FIFO ahead of vehicleB.
* the vehicle type and multiplier to be used are those at the time of the transaction. So an entry transaction uses values at that time, an exit transaction uses values at that time.

For simplicity's sake, we have not implemented a deadline after which the deposit is returned to the vehicle. Also, the function to clear pending more than 1 payment is an ugly one, because it implies a loop and multiple transfers. We actually use it to see how many pending payments you can cram in a single transaction.

**Contract overview**

All interfaces are found in the contracts/interfaces/ folder and are suffixed with I, a capital i. Here is the list of them:

* OwnedI, which keeps track of the owner of a contract.
* PausableI, which keeps track of the *paused* status of a contract.
* RegulatorI, which describes methods required of the regulator.
* RegulatedI, which keeps track of who the regulator is.
* MultiplierHolderI, which keeps track of the multipliers to attach to vehicle types.
* DepositHolderI, which keeps track of the base deposit required of vehicles.
* TollBoothHolderI, which keeps track of addresses that represent toll booths in the system.
* RoutePriceHolderI, which keeps track of the base route prices between entry and exit toll booths.
* TollBoothOperatorI, which describes methods required of the toll booth operator for interaction with vehicles.

You should not modify these interface .sol files.

The contracts you need to create in their individual files in the contracts/ folder are as follows. Note that we will use your implementations against our battery of Truffle unit tests. You can see a sample of them in the test folder. So it is important that you stick to the naming and the parameters order. Also, if you choose to create or use additional Solidity files, like libraries, make sure they are also in contracts/ and not in a subfolder.

When we say that One inherits from OneI and Two inherits from both OneI and TwoI, we leave it to your best judgement as to how to use One when coding Two. Remember that inheritance is transitive. All the contracts below are not abstract; this means they need to be deployable on their own.

**Owned**

It extends:

* OwnedI

and has:

* a modifier named fromOwner that rolls back the transaction if the transaction sender is not the owner.
* a constructor that takes no parameter.

**Pausable**

It extends:

* OwnedI, remember what we said about transitivity of inheritance
* PausableI

and has:

* a modifier named whenPaused that rolls back the transaction if the contract is in the false paused state.
* a modifier named whenNotPaused that rolls back the transaction if the contract is in the true paused state
* a constructor that takes one bool parameter, the initial paused state.

**Regulator**

It extends:

* OwnedI
* RegulatorI

and has:

* a constructor that takes no parameter.

**Regulated**

It extends:

* RegulatedI

and has:

* a constructor that takes one address parameter, the initial regulator; it should roll back the transaction if the regulator argument is 0.

**MultiplierHolder**

It extends:

* OwnedI
* MultiplierHolderI

and has:

* a constructor that takes no parameter.

**DepositHolder**

It extends:

* OwnedI
* DepositHolderI

and has:

* a constructor that takes one uint parameter, the initial deposit wei value; it should roll back the transaction if the deposit argument is 0.

**TollBoothHolder**

It extends:

* OwnedI
* TollBoothHolderI

and has:

* a constructor that takes no parameter.

**RoutePriceHolder**

It extends:

* OwnedI
* TollBoothHolderI
* RoutePriceHolderI

and has:

* a constructor that takes no parameter.

**TollBoothOperator**

It extends:

* OwnedI
* PausableI
* RegulatedI
* MultiplierHolderI
* DepositHolderI
* TollBoothHolderI
* RoutePriceHolderI
* TollBoothOperatorI

and has:

* a constructor that takes in this order:
  + one bool parameter, the initial paused state.
  + one uint parameter, the initial deposit wei value; it should roll back the transaction if the deposit argument is 0.
  + one address parameter, the initial regulator; it should roll back the transaction if the regulator argument is 0.
* a fallback function that rejects all incoming calls.

**Migrations**

You need to create one migration script 2\_...js that will:

* deploy a regulator,
* then call createNewOperator on it.
* then resumes the newly created operator, which should be paused before the resume step.

**Tests**

A quick note on wording:

* "a / the deposit" refers to the value that the contract requires from entering vehicles.
* "what is deposited" means what was actually sent by the vehicle upon entering.

You may create as many test files as you want, as long as they are not named test/scenarios.js, they will not be part of the grading. Also, do not modify or rename the existing tests that were in the repository at the beginning.

Please create a test file named test/scenarios.js, not in a subfolder. In this test file, please write exactly 6 tests, one for each of the following 6 scenarios. You can have as many describe, before and beforeEach as you want, however you need to have exactly 6 it in this test/scenarios.js file.

* Scenario 1:
  + vehicle1 enters at booth1 and deposits required amount (say 10).
  + vehicle1 exits at booth2, which route price happens to equal the deposit amount (so 10).
  + vehicle1 gets no refund.
* Scenario 2:
  + vehicle1 enters at booth1 and deposits required amount (say 10).
  + vehicle1 exits at booth2, which route price happens to be more than the deposit amount (say 15).
  + vehicle1 gets no refund.
* Scenario 3:
  + vehicle1 enters at booth1 and deposits required amount (say 10).
  + vehicle1 exits at booth2, which route price happens to be less than the deposit amount (say 6).
  + vehicle1 gets refunded the difference (so 4).
* Scenario 4:
  + vehicle1 enters at booth1 and deposits (say 14) more than the required amount (say 10).
  + vehicle1 exits at booth2, which route price happens to equal the deposit amount (so 10).
  + vehicle1 gets refunded the difference (so 4).
* Scenario 5:
  + vehicle1 enters at booth1 and deposits (say 14) more than the required amount (say 10).
  + vehicle1 exits at booth2, which route price happens to be unknown.
  + the operator's owner updates the route price, which happens to be less than the deposited amount (say 11).
  + vehicle1 gets refunded the difference (so 3).
* Scenario 6:
  + vehicle1 enters at booth1 and deposits more (say 14) than the required amount (say 10).
  + vehicle1 exits at booth2, which route price happens to be unknown.
  + vehicle2 enters at booth1 and deposits the exact required amount (so 10).
  + vehicle2 exits at booth2, which route price happens to be unknown.
  + the operator's owner updates the route price, which happens to be less than the required deposit (so 6).
  + vehicle1 gets refunded the difference (so 8).
  + someone (anyone) calls to clear one pending payment.
  + vehicle2 gets refunded the difference (so 4).

We will run your tests against our hopefully-correct implementations, for which we expect your tests to pass. We will also run your tests against our purposefully-incorrect implementations, for which we expect some of your tests to fail and the others to pass.

If you want to use async / await, you need to be able to make it work in the VM with only a modified package.json.

**GUI**

We want you to create a simple UI with 4 pages, or 1 page with 4 tabs if you prefer. You do not need to make it look fancy or add pagination. But it has to be functional. It is ok if it requires copy / paste of addresses and secrets from the human user.

You can use the framework of your choice, and it is good manner to add some instructions on how to make it run within the VM. If there is any NodeJs package that you would like to use make sure they appear in package.json and you committed it too.

So:

* a page for the deployed Regulator's owner, which allows it to:
  + set vehicle types.
  + create a new TollBoothOperator.
  + no need to make it possible to change the regulator on the individual TollBoothOperators.
* a page for the TollBoothOperator's owner, which allows it to:
  + add toll booths.
  + add base route prices.
  + set multipliers.
  + no need to make it possible to change owners of the contract.
  + no need to make it possible to pause the contract.
  + no need to make it possible to remove toll booths.
  + no need to make it possible to change the required deposit.
  + no need to make it possible to clear one pending payment.
* a page for individual vehicles, which allows it to:
  + see its basic Ether balance.
  + make an entry deposit.
  + see its history of entry / exit.
  + no need to see its pending payments.
* a page for individual toll booths, which allows it to:
  + report a vehicle exit.
  + be informed on the status of the refund or of the pending payment of the vehicle reported above.
  + no need to see its history of entry / exit.

**You code!**

This is not a gotcha exercise where a minute error makes you lose huge points. We want to see you at your best so let us know of any access issues or misunderstanding.