Bonus Task: Blockchain-native Applications

November 2018

Overview

The master's thesis *Blockchain-native Applications* deals with the characteristics of applications facilitating blockchains and smart contracts. As an output of this thesis, we have developed a framework which enables developers to make use of the properties and features provided by blockchains while hiding their underlying complexities. The goal of this bonus task is to evaluate the usefulness of the framework.

The assignment consists of three parts, an introductory task explaining the basics of the framework and two tasks showing its usage in the scenario of a smart refrigerator. These tasks use two different blockchain networks as backends, Ethereum[1] and Hyperledger Fabric[2].

Along with each task, a feedback form in TUWEL `https://tuwel.tuwien.ac.at/mod/feedback/view.php?id=571540` is to be filled out which includes the time spent on each task. Please be honest about how long it took you to complete a task or how difficult it was to you to do the task. Importantly, neither the time you have spent on doing the tasks nor the stated difficulty for you will have any influence on the bonus points, so there is really no reason to be not completely honest.

[1] `https://www.ethereum.org`
[2] `https://www.hyperledger.org/projects/fabric`

General Remarks

- **Don't be put off by the length of this document, each task's description is longer and more detailed than their solution.**
- A Git repository containing the required dependencies and setup for this task is available at `https://hyde.infosys.tuwien.ac.at`. You may have already seen that such a personalized repository has been set up for you and is available in your project list at the server.
- Hand in your solution until Monday, 26.11.2018 0:01 by pushing it to the Git repository.
- Use the TUWEL forum for questions. A specific forum has been set up at `https://tuwel.tuwien.ac.at/mod/forum/view.php?id=5715430`.
- Check the common issues at the end of the document before posting your question in TUWEL.
- The framework's documentation can be found in the folder `documentation`.
- Check out each tasks' hints as well as the javadoc and comments of the provided projects, they offer additional insights.
- You need to stop the time you have used to do the tasks. Please exclude any larger breaks from this and try to be precise down to the level of minutes.

Requirements

Please make sure to have the following software installed before starting the assignment.

- Java JDK 10+[3]
- docker[4]
- docker-compose[5]

[3] `https://www.oracle.com/technetwork/java/javase/downloads/index.html`
[4] `https://docs.docker.com/install/`
[5] `https://docs.docker.com/compose/install/`

Executing the tasks

The tasks in this assignment require a blockchain to be executed. Tasks 1.1 and 1.2 use an Ethereum blockchain, while task 1.3 applies a Hyperledger Fabric network. The required blockchain setup for these tasks is provided via `docker` images in the folders `network/ethereum` and `network/fabric`, respectively.

Both networks are started with `docker-compose up` in the corresponding folder and terminated via `docker-compose down`. For further remarks check the `README` of both folders.

The provided Java projects use Gradle as build system. In order to execute their main class, use `./gradlew build` and `./gradlew run`. When first executing one of these commands, Gradle will automatically be downloaded if not yet installed.
You might need to set executable flag via `chmod +x gradlew` in order to run the commands on MacOs/Linux.

Blockchain-native Applications

The framework *blockchain-native applications* aims to provide an abstraction of common features found in blockchain-based, distributed systems and enables application developers to use them without requiring substantiated knowledge about individual systems.

In this assignment, the usage of *smart contracts* is demonstrated. Smart contracts are programs that operate on data contained in a blockchain. They are deployed to the network and executed by its peers.

The framework provides the ability to invoke smart contracts of various blockchain-based systems as if they were simple Java classes. This is done by declaring a Java interface whose methods match the ones declared by the smart contract.
In the following tasks, this functionality is to be explored.

1 Tasks

1.1 Task 1: Warm Up

In this task, we will walk you through the process of declaring and using a Java wrapper interface for a simple Ethereum smart contract.

Ethereum smart contracts are programs that allow its users to perform Turing-complete computations with the blockchain as data storage. Once deployed, these contracts can be accessed through any node participating in the network, since their program code is stored in the blockchain as well. Ethereum provides multiple languages which can be used to write smart contracts, in this example we use Solidity. While not required for solving this task, an in-depth description of Solidity can be found at [6].

Your solution is to be implemented in the prepared project `1_warmup`.

This project provides the Ethereum smart contract `HelloContract` in its source and binary form in the resource directory.

1.1.1 Declaring the Interface

In order to create a wrapper class for calling the Ethereum smart contract, a Java interface needs to be created which declares the desired methods.

Instructions: Complete the interface `org.blockchainnative.warmup.HelloContract` by declaring matching methods for the method '*hello*' and the event '*greeted*' defined by '*HelloContract.sol*'.

Hints:

- In the folder `documentation`, take a look at the javadoc of the class `org.blockchainnative.ethereum.EthereumContractWrapperGenerator` to see how Solidity types are mapped to Java types.
- Explore the documentation of the annotations defined in the package `org.blockchainnative.annotations` and choose the appropriate ones.
- Make use of the inner class `HelloEvent` for event '*greeted*'.
- In general it is possible to correctly define a smart contract wrapper interface without the use of any annotations and solely rely on the methods provided by contract info builder API introduced in 1.1.2. Nevertheless we will use them for this example as they are more straight forward to use.

[6] `https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html#a-simple-smart-contract`

1.1.2 Creating the Contract Info

The framework requires a Java interface (as defined in the previous sub task), together with blockchain specific information about the smart contract to create a wrapper class. This information is declared through `ContractInfo` objects and its subtypes. In this case, as we use Ethereum, the class `EthereumContractInfo` is used. For interacting with Ethereum contracts, three properties are of particular importance: The application binary interface (ABI), i.e., a JSON file describing the methods and events of the smart contract, the contract address and the compiled binary of the contract. While the later is only required if we want to deploy the smart contract through the framework mechanisms, the ABI is mandatory in any case. The contract address is needed is to locate the contract on the blockchain. It is required to interact with its methods and events, however, since we did not deploy our contract yet, we will leave out this property for now but will come back to it in 1.1.5.

Instructions: Create a contract info object for the smart contract wrapper interface `HelloContract` by using the fluent API provided by the class `EthereumContractInfoBuilder` in `org.blockchainnative.warmup.WarmUp`.

Hints:

- Specify an identifier for the contract info.
- Make sure to include the smart contract's ABI and binary in the contract info. Both files are provided in the resources folder of the project.
- The contract info's properties can either be declared through the appropriate annotations on the contract interface or through calling the corresponding methods on the contract info builder.
- Take a look at the documentation of `org.blockchainnative.ethereum.builder.EthereumContractInfoBuilder` to see which annotations or method calls are required to register the method and event.

1.1.3 Create the Contract Registry

The contract info object structure also holds state information about smart contracts, e.g., in the case of Ethereum, the address at which a contract is found. This state information can be stored and loaded through the use of implementations of `ContractRegistry`. By using the methods supplied by this class, developers are still required to declare the initial contract info, however, state information changes during program execution can be persisted and reloaded on subsequent executions. The framework provides the class `org.blockchainnative.registry.FileSystemContractRegistry` as an implementation of `ContractRegistry` that allows storing contract info objects as JSON files.

Instructions: Continue the implemenation in the class `WarmUp`. Create an instance of `FileSystemContractRegistry`, call its `load` method and add the previously created contract info to the registry.

Hints:

- `ContractRegistry` provides multiple ways of adding contract info objects. Make sure to only add the contract info if no object with the same identifier is already present in the registry.
- When first executing the example there is obviously no contract info object to be loaded from the file system. However, in subsequent executions we will use the stored information about the contract instead of deploying it each time.
- Make sure to register an instance of `org.blockchainnative.ethereum.serialization.EthereumMetadataModule` with the contract registry in order to make sure that the (de-)serialization works correctly.

1.1.4 Generate the Contract Wrapper

When all required information about the smart contract is registered, a wrapper class can be built by using implementations of `ContractWrapperGenerator`. In this case, `EthereumContractWrapperGenerator` is used. The generated wrapper class allows you to interact with the smart contract without having to worry about the details of how those methods are actually invoked. Moreover, the Java interface representing the smart contract stays mostly the same as you switch to a different blockchain.

Instructions: Create a wrapper class for the smart contract `HelloContract` using the class `EthereumContractWrapperGenerator`.

Hints:

- Take a look at the methods provided in `org.blockchainnative.warmup.WarmUp` and use them to create an instance of `EthereumContractWrapperGenerator`.
- Instead of using the previously created contract info object when generating the wrapper, use the `ContractRegistry` to retrieve it by its identifier. This makes sure that the wrapper class uses the latest contract info in subsequent executions of the program.

1.1.5 Deploy the Contract

Before a smart contract is ready to be used, it needs to be deployed to the blockchain network. The generated wrapper class provides this functionality through special methods. In the context of the framework, special methods denote methods declared on smart contract wrapper interfaces which are not directly mapped to a function or event but rather have a special meaning which needs to be interpreted by the wrapper generator.

`HelloContract` provides the predefined special method `deploy`, that installs the smart contract on blockchain. After a successful deployment, the wrapper will update the contract info's address property to correctly reflect its state.

Instructions: Verify whether the contract wrapper is ready to be used (i.e. has been deployed before) by checking its contract info and call the predefined `deploy` method if needed. Update the contract info in the registry after the deployment.

Hints:

- Retrieve the contract info from the wrapper by calling `getContractInfo` declared by the interface `org.blockchainnative.ethereum.EthereumSmartContract`.
- Since we did not specify the address on the contract info, the contract is obviously not considered to be deployed when first executing this example.

1.1.6 Subscribe to the Event

In order to be notified about events, the blockchain-native applications framework provides observables[7] defined in the wrapper interfaces. This allows users to react on incoming events. Event observables are only supported by contract wrapper generators which target blockchains that themselves provide events in their implementation of smart contracts.

Instructions: Subscribe to the contract's '*greeted*' event using the predefined observer in `org.blockchainnative.warmup.WarmUp`.

1.1.7 Call the Method

Instructions: After subscribing to the '*greeted*' event, invoke the method '*hello*' and print the result to `System.out`. Moreover, check the output to see if the event has been captured by the observer.

1.1.8 Persist the Contract Registry

Since smart contracts, like any other programs, are typically designed to be deployed once and used multiple times, we do not want to deploy the contract on each execution of this example. Using the contract registry, the contract's state information can be stored and reloaded as done in sub task 1.1.3.

Instructions: Use the contract registry to persist the contract info object.

1.1.9 Run the program

After implementing the previous sub tasks, it is time to execute the program.

Start the dockerized Ethereum network provided in the folder `network/ethereum` by running `docker-compose up` and execute the application using `gradlew run`.

Note that each time the network is restarted, **all previous state is lost**. Therefore, the state information stored in previously persisted contract info objects do not reflect the network state anymore. Make sure to remove them from the file system before you execute the application. This applies to the following tasks as well.

[7] `http://reactivex.io/documentation/observable.html`

1.2 Task 2: Smart Refrigerator – Ethereum

After getting familiar with using a smart contract wrapper generated by the framework, it is time to move to an example that is closer to a real-world application.

In this example we use a *smart refrigerator* that is able to automatically order missing items when its contents drop below a predefined minimum supply.
The code for it is found in the project `2_ethereum`.

In the method `run` of class the `Application`, a scenario is prepared that needs to be completed. The scenario tests the implementation of the refrigerator and its ability to order items. At first, a smart contract which stores hash values of receipts on the blockchain is deployed. After that, items are removed from the prepared refrigerator and it is instructed to restock itself. It is then checked if the hash value of the receipt created in this process can be retrieved through the smart contract. The usage of hash values in this example allows to check whether a particular receipt exists without disclosing information about the order.

As this task intents to be a more realistic example, we also make use of the concept of dependency injection[8] which is used heavily in real world software engineering. By applying this concept, the framework is able to create commonly required objects automatically which allows developers to focus on the actual application logic. More specifically, the developer only needs to register the contract info objects and can directly obtain wrapper classes through dependency injection. The framework uses Spring[9] and more precisely Spring Boot[10] for this task. However, no special knowledge about Spring is required for solving this exercise as the provided project has everything setup for you already.

A new feature introduced in this task is the concept of `TypeConverter`. Each blockchain provider has a set of supported types that can be used for parameters and return values. If an unsupported type is used, a type converter needs to be defined in order to convert the declared type from/to the corresponding type in the smart contract. In this example, we want to store custom `Hash` objects in a smart contract. Their type in the smart contract is declared as byte array. In order to directly use `Hash` objects, we need to specify a type converter that is able to convert byte arrays to `Hash` objects and vice versa.

[8] `https://en.wikipedia.org/wiki/Dependency_injection`
[9] `https://spring.io/`
[10] `https://spring.io/projects/spring-boot`

Instructions:

1. Take a look at the smart contract wrapper interface `ReceiptContract` and create a contract info object in the class `ContractConfiguration`.
2. Finish the implementation of `HashTypeConverter`.
3. Connect the refrigerator to the blockchain by implementing the missing piece in the method `placeOrder()` of `OrderServiceImpl`.
4. Complete the scenario in the method `run` of the class `Application` by performing the following actions:
   (a) Complete the smart contract's deployment procedure
   (b) Take enough items from the refrigerator so that it needs to be restocked
   (c) Check the receipt which is created by calling `restock()` on the refrigerator and make sure that its `Hash` is stored in the `ReceiptContract`.
   (d) Persist the contract registry in order to save the contract info of `ReceiptContract`.

Hints:

- Specify an identifier for the contract info.
- Make sure to include the smart contract's ABI and binary in the contract info. Both files are provided in the resources folder of the project.
- An instance of `HashTypeConverter` is registered for dependency injection in the class `ContractConfiguration` and is automatically picked up by the framework.
- In order to execute the example, make sure to have the provided Ethereum network up and running.

1.3 Task 3: Smart Refrigerator – Hyperledger Fabric

The last two tasks used Ethereum as backend, however other platforms support smart contracts as well. In this task, we target smart contracts offered by Hyperledger Fabric, so called chaincode. Whereas in Ethereum all peers have equal rights and possibilities to contribute to the blockchain, Hyperledger Fabric's network structure is divided into multiple organizations and peers with different permissions. Therefore, its deployment procedure greatly varies from Ethereum (see Hints). The Hyperledger Fabric network provided for this example consists of two organizations, each having one peer. Additional knowledge about this blockchain is not required, however interested readers find further resources about Fabric at [11].

Until now, we used annotations to map methods to smart contract functions and events. This time, we will solely use the contract info builder API.

The project `3_fabric` contains the same refrigerator as before, however, a slightly different approach is used to verify that the receipt hashes are stored in the blockchain.

[11] `https://hyperledger-fabric.readthedocs.io/en/latest`

Instructions:

1. Set the remaining required properties on the contract info builder in the class `ContractConfiguration` in order to be able to create a wrapper class for `ReceiptContract`.
2. Finish the implementation of `HashTypeConverter`.
3. Connect the refrigerator to the blockchain by implementing the missing piece in the method `placeOrder()` of `OrderServiceImpl`.
4. Complete the scenario in the method `run` of the class `Application` by performing the following actions:
   (a) Complete the smart contract's deployment procedure.
   (b) Take enough items from the refrigerator so that it needs to be restocked.
   (c) Check the receipt which is created by calling `restock()` on the refrigerator and make sure that its `Hash` is contained in the list of hashes obtained through the predefined event subscriber.
   (d) Persist the contract registry in order to save the contract info of `ReceiptContract`.

Hints:

- The remaining required properties are the chaincode source directory and the chaincode language. Moreover, the `instantiate` method needs to be marked as special method analogue to `install`.
- The framework passes all objects as strings to Hyperledger Fabric chaincode, the class `HashTypeConverter` therefore needs to convert instances of `Hash` to string.
- Smart contract deployment in Hyperledger Fabric is more complex than it is in Ethereum. A contract or chaincode, first needs to be *installed* on all peers which should be able to execute it. This operation needs to be carried out using a user context that is allowed to perform this operation on the specified target peers. Afterwards, the contract needs to be *instantiated* on all peers simultaneously. Since the network structure used in this example features two organizations, `install` needs to be executed twice, once for each organization using the corresponding administrator user.

2 Common Issues

Couldn't connect to Docker daemon In case `docker-compose up` failed with a message similar to:

```
ERROR: Couldn't connect to Docker daemon at http+docker://localhost.
```

Depending on how you installed docker, your current user might not be allowed to run docker containers. If you are on Linux/MacOs, add your current user to the docker group:

```
sudo usermod -a -G docker $USER
```

After that log off and back on again.

Alternatively, you can run `docker-compose up` with administrator privileges.

Stopping docker-compose If you stopped the execution of docker-compose by sending SIGINT (Ctrl + C), make sure to execute `docker-compose down` afterwards to let the program clean up after itself. Otherwise, starting the network again with `docker-compose up` might fail.

Failed to install chaincode 'receiptContract:1.0' If you are facing the following message during execution of task 3, make sure delete any contract info stored in the project folder `contracts` and restart the Hyperledger Fabric docker network.

```
Failed to install chaincode 'receiptContract:1.0' on peer 'peer0.org1.example.com':
error installing chaincode code receiptContract:1.0
(chaincode /var/hyperledger/production/chaincodes/receiptContract.1.0 exists)
```

The cause of this error most probably is that you deployed the smart contract to the peer in a previous execution, but the contract info has not been persisted, e.g., because of an exception. When this happens, the contract info's state information does not represent the real world state anymore.