BlockVerse

Emanuele Andaloro

Email: emanuele.andaloro@mail.polimi.it

***Abstract*—KATENA is a framework for the deployment and management of Blockchain applications. In particular, it focuses on applications that are compatible with Ethereum, a popular general-purpose Blockchain technology. The aim of this paper is to explain the dashboard functionalities added to KATENA, specifically focusing on topology printing, creation, and editing.**

KEYWORDS

software engineering, blockchain, TOSCA,deployment, topology management.

I. Introduction

KATENA is a framework for the deployment and management of Blockchain applications. It is particularly focused on applications that are compatible with Ethereum, a popular general-purpose Blockchain technology. The aim of this paper is to provide a dashboard to KATENA, specifically focusing on topology printing, creation, and editing.It's particularly important to have a graphical interpretation of the components that are provided by KATENA in particular the different features provided by it as a framework could be enriched having a GUI which make the user able to either to print a topology like simple nodes and relationships up to more difficult ones like ens,dydx or dark forest.The Ethereum Name Service is widely used, with over 465K ENS names registered and 180K of them active. It is composed of four contracts that are used to register the DNS domain along with its owner, and to provide additional features such as subdomains. Its architecture is quite simple, it provides three Contract-Contract relationships between the core registry, named ENSRegistry, and the three others. Figure 5 shows a comparison between Hardhat and KATENA on the implementation of one of these relationships (between contract PublicResolver and ENSRegistry). DYDX is one of the largest exchanges for cryptocurrencies and it employs 28 smart contracts with the following dependencies:2 Contract-Library, 21 Contract-Contract, and 10 Lazy-Contract-Contract dependencies. Compared to ENS, DYDX uses Truffle for its operation and the code is, generally, much more complex given the higher amount of contracts and dependencies. Notably, KATENA showed a significant improvement compared to Truffle in our quantitative evaluation. Our solution led to a reduction of 39.5% of NoT with 559 tokens against 923 of the original script. This highlights that as the complexity of operations increases, KATENA appears to achieve a larger reduction in the efforts required to write the scripts. Once again, this can be intuitively explained by the more abstract and orchestration-driven approach of KATENA that significantly simplifies the deployment and management process. Lastly, Dark Forest uses Hardhat for its operation, and it is composed of 5 libraries and 11 smart contracts. Two significant characteristics of this application are the usage of the Diamond pattern and Library-Library dependencies. Users of KATENA can create diamonds with ad-hoc types (katena.nodes.diamond) and requirements (useCut and useFacet) which, behind the scenes, automate the wiring of the dependencies and instruct the orchestrator on the instructions to execute. On the other hand, Hardhat users must deal with complex operations "manually", and the resulting code does not allow them to simply understand the structure of the application.Then the user will be also able to personalize it's own topology and also to create it. After the graphical orchestration the user can return to a more useful interpretation which is a TOSCA one necessary for a future deployment.

II. Goals

- Provide a dashboard for the given deployment and management framework
- Possibility of importing topologies;
- Possibility of editing a topology

III. Domain Assumption

[D1] KATENA is an asset for the platform;
[D2] KATENA deployment is invoked inside the application;
[D3] The user can see the topology as graph;
[D4] The user can export the YAML file of the topology

IV. System Architecture and Design

For what concerns the architecture, the backend is hosted on Firebase, the authentication part is ensured by both checking the user inputs on client side and for what concerns the server side APIs provided by Firebase Authentication are used. User Data are stored inside Firebase Firestore this permits a graceful management of Data because you can exploit the features of NO-SQL databases. Here you can find an overview of the Distributed architecture.

A. 3-tier Architecture

The architectural style which has been used is the 3-tier architecture style because it provides many benefits the main one is the modularization in three independent layers:

- The web server:is the presentation tier and provides the user interface.This is usually a web page or a web site.The

content can be static or dynamic, and is implemented using Flutter.

- The application server:it is the middle tier, implementing the management logic.
- The database server:it is the backend tier of a web application.It runs on Firebase Platform.

B. the Singleton pattern

Singletons are classes which can be instantiated once, and can be accessed globally. This single instance can be shared throughout our application, which makes Singletons great for managing global state in an application.

V. The GUI

Here in this section, is presented the graphic user interface of the application accordingly to the goals there are three Screens which are most important not all the screens and functionalities are presented for simplicity.

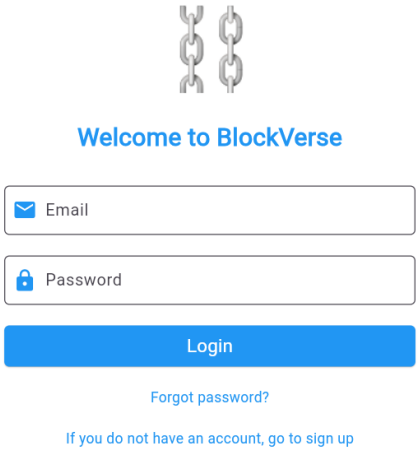

Figure 1. User login screen

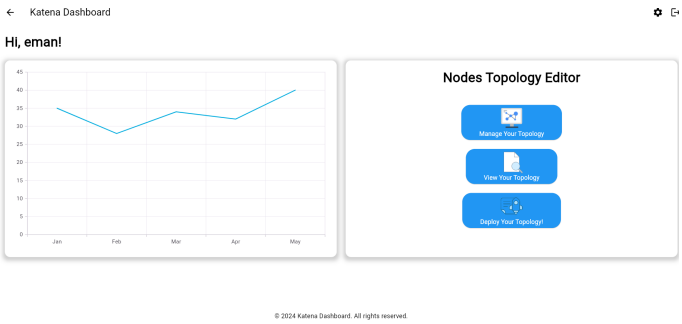

Figure 2. Dashboard screen

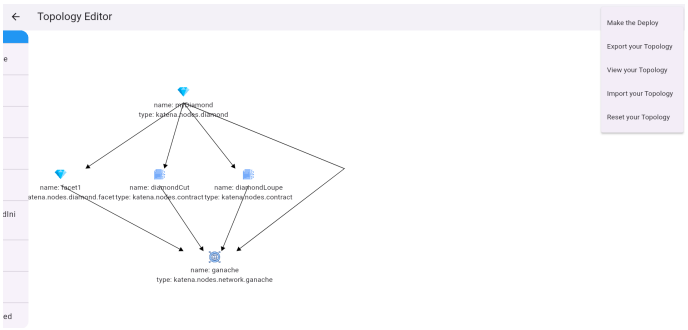

Figure 3. Topology management screen

VI. Implementation

Here, are presented the three fundamental snippet which are the core of the existing application important to achieve the given goals.

[M1] Topology printing

```
1
2
3   Future<Graph?> TopologyGraphFromYaml
        () async {
4     var yamlFile = await
          ServiceProvider.ImportYaml();
5     var nodeProperties = yamlFile?['
          topology_template']['
          node_templates'];
6
7     if (nodeProperties == null) {
8       print("No node templates found in
            YAML.");
9       return null;
10    }
11    print(yamlFile);
12    Graph graph = Graph()
13      ..isTree = false;
14    List<String> imports = [];
15
16    // Load imports
17    if (yamlFile?["imports"] != null &&
          yamlFile!["imports"].
          isNotEmpty) {
18      for (var importPath in yamlFile["
            imports"]) {
19        try {
20          YamlMap? yamlMap = await
                loadYamlFromAssets(
21                "katena-main/$importPath"
                      );
22          var nodeTypes = yamlMap?['
                node_types'];
23          if (nodeTypes != null) {
24            imports.addAll(nodeTypes.
                  keys.cast<String>());
25          }
26        } catch (e) {
27          print("Error loading import:
                $importPath - $e");
28        }
29      }
30    } else {
```

```
    print("No imports found in YAML.");
  }

  // Create nodes and add to graph
  Map<String, Node> nodes = {};
  for (var key in nodeProperties.keys) {
    if (imports.contains(nodeProperties[key]["type"])) {
      Node node = Node.Id("name:$key\ntype:${nodeProperties[key]["type"]}");
      graph.addNode(node);
      nodes[key] = node;
      print("Node added: ${node.key?.value}"); // Debug print
    } else {
      print("Node type not in imports: ${nodeProperties[key]["type"]}"); // Debug print
    }
  }

  // Add edges based on requirements
  for (var key in nodeProperties.keys) {
    var requirements = nodeProperties[key]["requirements"];
    if (requirements != null) {
      Node? node = nodes[key];
      if (node != null) {
        for (var requirement in requirements) {
          var targetNodeName = requirement.values.first;
          print("Node $key requires: $targetNodeName"); // Debug print
          Node? targetNode;
          if (targetNodeName is YamlMap) {
            print("Target node name (YamlMap): ${targetNodeName.values.first}");
            targetNode = nodes[targetNodeName.values.first.toString()];
          } else {
            print("Target node name: $targetNodeName");
            targetNode = nodes[targetNodeName];
          }

          if (targetNode != null) {
            if(node!=targetNode) {
              graph.addEdge(node, targetNode);
            }else{
              print("the node is connected to itself");
            }
          } else {
            print("Target node not found: $targetNodeName"); // Debug print
          }
        }
      }
    }

    print("Graph nodes: ${graph.nodes.length}, ${graph.edges.length} edges created."); // Debug print
    return graph;
  }
```

Listing 1. Topology printer

This method prints the Topology by the means of a GraphView Library. Firstly, the TOSCA is parsed an loaded into the dashboard then the TOSCA grammar is used in order to adapt the graph components to the TOSCA grammar which is a non trivial issue but by ease of abrastraction the nodes of the graph becomes Katena nodes and the requirements of each node becomes arcs of the graph.

[M2] Topology management

```


  Future<Graph?> TopologyCreatorNodes(String name, String type, Graph graph,
      Node? root) async {
    Provider serviceProvider = Provider.instance;


    if (graph.hasNodes()) {
      print("graph is not empty");
      graph.removeNode(root);
    }
    else {
      print("graph is empty a new one is created");
      graph = Graph()
        ..isTree = false;
    }
    Node node;
    final yamlMap = await serviceProvider.GetDescriptionByType(type) as YamlMap;


    if (yamlMap != null) {
      node = Node.Id("name:$name\ntype:$type");
      print(node);
      graph.addNode(node);
    }
```

```
27
28
29      print(yamlMap);
30      return graph;
31    }
32
33    Future<Graph?> TopologyCreatorEdges(
          String type, Graph graph, Node
          sourceNode,
34        Node destinationNode) async {
35      Provider serviceProvider = Provider
            .instance;
36
37
38      if (graph.nodes.length == 1 && type
            == "Add edge") {
39        Node node1 = Node.Id("Root Node")
              ;
40        graph.addNode(node1);
41
42        return graph;
43      } else if (type == "Add edge" &&
            graph.nodes.length > 1) {
44        String? key1;
45        String? value;
46        String nodeId = sourceNode.key?.
              value;
47        List<String> lines = nodeId.split
              ('\n');
48        Map<String, String> typeMap = {};
49        for (var line in lines) {
50          List<String> parts = line.split
                (':');
51          if (parts.length == 2) {
52            key1 = parts[0].trim();
53            value = parts[1].trim();
54            typeMap[key1] = value;
55          }
56        }
57        print(value! + 'c');
58        String? key2;
59        String? value2;
60        String nodeId2 = destinationNode.
              key?.value;
61        List<String> lines2 = nodeId2.
              split('\n');
62        Map<String, String> typeMap2 =
              {};
63        for (var line2 in lines2) {
64          List<String> parts2 = line2.
                split(':');
65          if (parts2.length == 2) {
66            key2 = parts2[0].trim();
67            value2 = parts2[1].trim();
68            typeMap2[key2] = value2;
69          }
70        }
71
72  YamlMap? source = await serviceProvider
        .GetDescriptionByTypeforManagement(
73            value!);
74  YamlMap? destination = await
        serviceProvider
75     .GetDescriptionByTypeforManagement(
         value2!);
76
77        if (source?["requirements"] !=
              null) {
78          var sourceReq = source?["
                requirements"];
79
80          for (var req in sourceReq) {
81            var sreq = req.values.first;
82            print(sreq["capability"]);
83
84  if (sreq["capability"] != null) {
85
86              String? capacity_fatality =
                    await serviceProvider
87                  .GetCapabilitiesByType(
                        value2);
88              print(value2+"\n"+
                    capacity_fatality!);
89
90  if (sreq["capability"] ==
        capacity_fatality) {
91            graph.addEdge(sourceNode,
                destinationNode);
92              } else {
93   if (sreq["node"] != null) {
94    YamlMap? inheritedRequirements =
          await serviceProvider
95            .
                GetDescriptionByTypeforManagement
                (sreq["node"]);
96                  // print(
                        inheritedRequirements
                        );
97
98                  if (
                        inheritedRequirements
                         != null) {
99                    var inReqs =
                          inheritedRequirements
                          ["requirements"];
100
101                   for (var inreq in
                          inReqs) {
102                     var inreqsource =
                            inreq.values.
                            first;
103                     print(inreqsource);
104                     String?
                            capacity_fatality2
                             = await
                            serviceProvider
105                         .
                            GetCapabilitiesByType
                            (value2);
106
107                     if (inreqsource["
                            capability"] ==
108                         capacity_fatality2
                                ||
                            inreqsource
                            ["node"] ==
                             value2) {
109                       graph.addEdge(
                              sourceNode,
                              destinationNode
                              );
110                     }
111                     else {
112                       print("the two
```

```
                                nodes are not
                                 compatible")
                                ;
                        }
                    }
                }
            }
        }
      } else {
        print("if requirement is
            null the node is not
            connectable");
      }
    }
  } else {
    print("up to now do nothing");
  }
  print(source);
  var some_der_req= await
      serviceProvider.
      getInheritedRequirements(
      value,source!);
  print(some_der_req);
  var sourceReq2 = some_der_req?["
      requirements"];
  for(var i_req in sourceReq2)
  {

    var inreqsource2 = i_req.values
        .first;
    print(inreqsource2);
    String? capacity_fatality3=
        await serviceProvider
        .GetCapabilitiesByType(
            value2);

    if (inreqsource2["capability"]
        ==
        capacity_fatality3||
            inreqsource2["node"] ==
             value2) {
      graph.addEdge(sourceNode,
          destinationNode);
    }
    else {
      print("the two nodes are not
          compatible");
    }
  }


  return graph;
 }
 return null;
}
```

Listing 2. Topology Management

This method is necessary for creating a new Topology or for modifying an existing one basically as the method describe before you map katena.nodes as graph nodes then this nodes in the TOSCA grammar have some requirements this, if the nodes in the requiment have the same capability or the same capability and the same type the two nodes are connectable.Some examples of connections will be provided later.

[M3] **Topology exporting** This method after the Creation of a topology exports the TOSCA of the concocted topology graph.So, as the name of the Topology suggest a graph to YAML parsing.

VII. Testing

The testing methodologies that were used for the development of the application are several combined basically:

- Thread strategy: A thread is a portion of several modules that, together,provide a user-visible program feature;
- Bottom-up strategy:It may create several working subsystems;

A. Features Identification

The features to be implemented are described starting from the requirements. Some requirements need the implementation of new components while others doesn't require big changes .

[F1] **Sign-up and login** This feature permits to the user to sign-up and login securely using Firebase

[F2] **Topology printing** This feature allows the user to print a topology on the screen inserting a TOSCA file on the Dashboard.

[F3] **Topology management** This feature allows the user either to import a Topology and print it but also to modify it and also to create it's own topology.

[F4] **Topology exporting** After the Topology creation the user can also export it's own topology by downloading a TOSCA file in his computer.

B. Component Integration and Testing

Here is explained how each component was integrated in each part of the develpoment and how the different components were implemented and tested.

[F1] **Sign-up and login** The only thing to be tested here is the interaction of the dashboard with the Authentication API provided by Firebase which is considered reliable by definition.

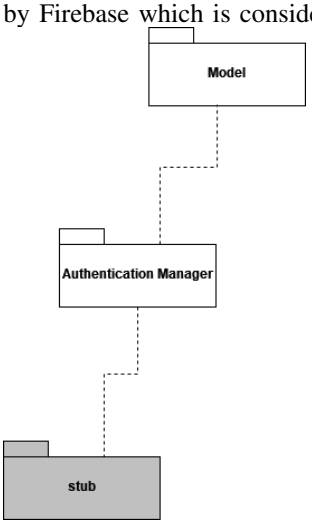

Figure 4. sign-up and login testing

[F2] **Topology printing** Here the new part to be tested is the topology manager which is the component in charge to manage everything regarding the Topology from the printing up to the changing and the exporting.

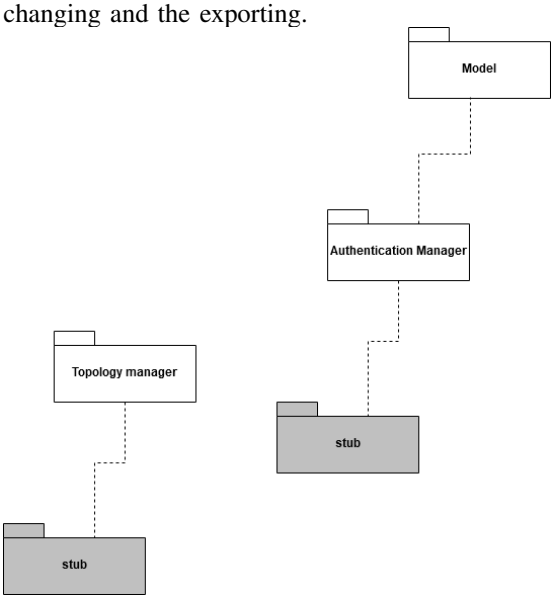

Figure 5. topology printing testing

VIII. Future work

In Blockchain application it's becoming a crucial issue to know where and when a node could fail a further improvement could be to use generative AI to implement a sort of topology assessment which could be useful to understand when a failure could arrive. So the idea is to use a large dataset of topology to train a model for having an assessment provided the TOSCA of the topology.

References

[1] A Declarative Modelling Framework for the Deployment and Management of Blockchain Applications; url:https://arxiv.org/abs/2209.05092