Designing a Web Application with Complex REST-generated GUIs Using JHipster and Angular.

1. Introduction.

This document describes the process of designing and developing a web application for the discovery, creation, removal, and modification of resources. By the term 'resource' we mean a basic entity of a complex Information System (which hereafter we will refer to as the IS), described in terms of metadata. Metadata for each resource, as well as hierarchies and relationships between resources, are returned in JSON format by Java APIs of the IS. A peculiarity of the web app we will describe lies in the fact that the resources' structures can vary dynamically over time, so it was necessary to build graphical interfaces based on descriptive metadata gathered in real time.

2. Working Environment.

The web app was developed using the JHipster platform (https://www.jhipster.tech/), which provides an integrated full-stack development environment web applications and microservices, based on Java and Node.js. For the backend side, JHipster leverages the Java reference frameworks Spring Boot and Spring Security (for the security and authentication layers), while for the frontend developers can choose among three of the most used frameworks based on Typescript, i.e., Angular, React and Vue. We decided to adopt Angular.

Since the IS APIs required Java 11, we could not use the latest version of JHipster, which required at least Java 17 and did not guarantee full retro-compatibility. The final configuration of the working environment resulted the following:

JAVA	openjdk version 11.0.20
JHIPSTER	v. 7.9.3
Angular CLI	14.2.1 (using rxjs 7.5.6 and typescript 4.8.2)
Node.js	14.20.1
Package Manager	npm 9.8.1
IDE	SpringToolSuite(STS) 4 Mozilla Firefox Developer Tools

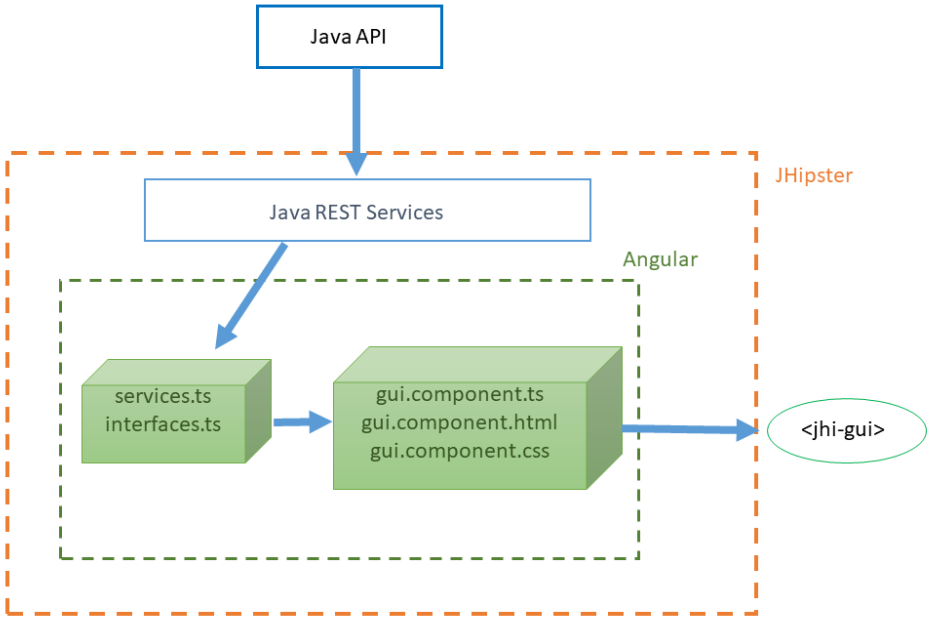
Angular natively supports Bootstrap for styling GUIs, and, in its most recent versions, also Material. We had decided to use Material, but the version of Angular shipped within Jhipster 7.9.3 still did not support it, so it was necessary to manually import and configure in Angular the Material widgets we wanted to use (e.g., Navigation Trees, Tabbed panes, Tables, etc.). The GUIs were tested and debugged on Mozilla Firefox and Google Chrome.

3. Archictecture.

The backend of a web app developed with JHipster is based on REST services. In our case, this meant that first of all it was necessary to develop a layer of Java code that exploited the APIs of the external environment (i.e., the Information System) to produce web services to be exposed by JHipster to precise endpoints. These endpoints are used by typescript services that expose data and functionalities to the

GUIs of the frontend, the exchange format between services being JSON. For both the backend and frontend, it is necessary to provide entities to hold the data to be exchanged. In the backend this is done through so-called DTOs (Data Transfer Objects). A DTO is essentially a java bean with no internal logic that is used to transfer data towards the frontend. Similarly, a Typescript interface is used in Angular to define the fields that will be returned by a service connected to a REST endpoint exposed by JHipster's backend. Figures 1 and 2 schematically show what we have just described.

Figure 1 A high-level description of the JHipster architecture.



Once the web app is deployed, the complete list of the backend web services can be found on a specific web page provided by JHipster. This is very useful for debugging purposes, as it is possible to query the services created by passing any required parameters, and to check the JSON response. Figure 3 shows what this page looks like in the case of our web app; we can distinguish between POST requests, which are related to CRUD methods, and GET requests, related to discovery operations performed on the Information System (i.e., operations to get all the resources, the JSON specification of each resource, etc.).

Figure 2: A high-level description of the architecture of our web application.

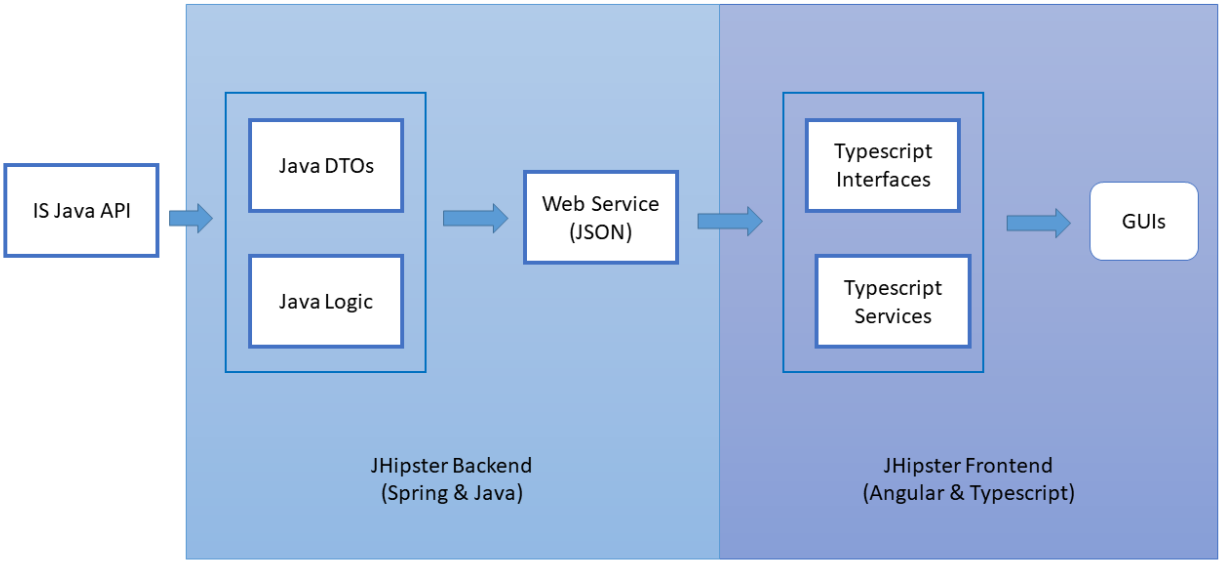
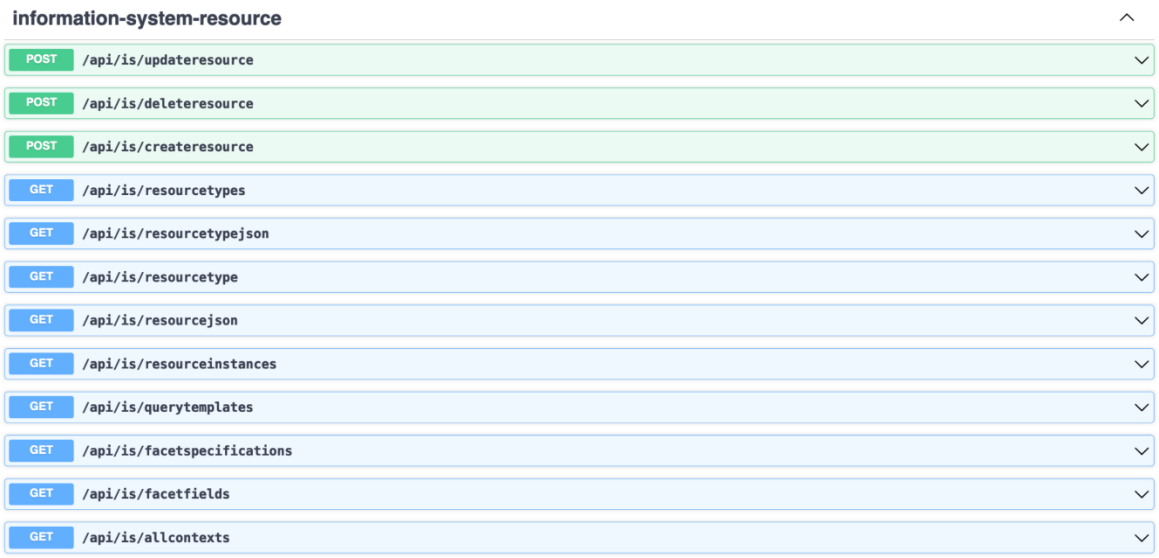


Figure 3: JHipster's web page to access to backend webservices.



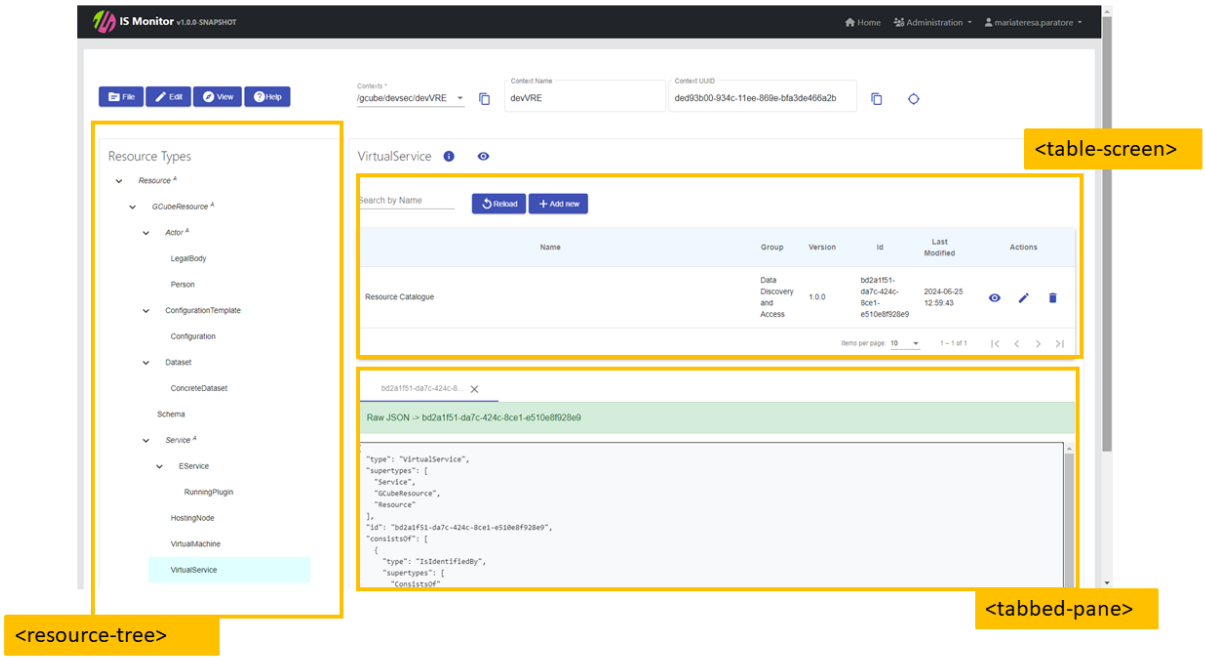
4. Dynamic GUI Creation.

Angular allows GUIs to be created according to a modular mechanism, i.e. it is possible to compose a GUI using independent, self-consistent base elements, called 'components'. An Angular component is a graphical interface defined by the developer, and can have its own parameters, can be connected to

Angular services, has its own style, etc. A component can interact via parameters with other components, parents or children, and above all it can be used several times in the web app with different configurations and data. Each component is uniquely identified by a tag, by which it must be referred to within the html code (https://angular.io/guide/component-overview).

Figure 5 shows a web page of our application, in which some components are highlighted.

Figure 4: Angular components used to compose a GUI.



As an example, consider the creation of the resource-tree component for resource navigation. In the backend layer, we have provided a method to get the hyerarchy of the resources from the IS, and exposed it through a web service at the endpoint '/api/is/resourceinstances'. On the Angular frontend side, we created the following pieces of code:

- resource-tree.component.ts: the Typescript logic for the component
- resource-tree.component.html: the component 's HTML structure
- resource-tree.component.scss: for the component 's styling
- resource-tree.service.ts: the backend communication service injected into the component
- i-resource.ts: the Typescript interface which allows to translate the JSON objects retrieved by the service into proper Typescript objects.

In resource-tree.component.ts an Angular tree is instantiated. The component also contains the code to manipulate the resource implementations, so as to create objects suitable to be hierarchically displayed in the tree. Moreover, it defines the interactive behavior of the interface (e.g., how to respond to 'onClick' and 'onHover' events).

5. Forms for Creating Resources.

The structure of each IS resource is described in terms of elementary sub-resources, called “facets”. The IS APIs provide methods to return the JSON description of a resource in terms of facets and, for each facet, the associated data structure. Combining this methods allowed us to discover in real time the composition of a resource and consequently set up its creation GUI. In Angular, it is possible to build forms dynamically through the “reactive forms” feature (https://angular.dev/guide/forms/reactive-forms). As the official Angular states, “reactive forms provide a model-driven approach to handling form inputs whose values change over time”. We defined a number of form templates in JSON format, to account for data-entries required for the different facets; these templates could be dynamically composed in accordance with each resource specification. Using the proper methods of the IS API, we then created a method the backend REST, to retrieve the form specification for any kind of resource. Figure 5 shows part of the JSON response of the REST for a specific resource type. The JSON description of the data entry fields required for the creation of a facet is highlighted.

Figure 5: JSON specification of the fields required for the creation of a MemoryFacet, as part of the form for creating a HostingNode resource instance.


```
  ],
  "relation": "ConsistsOf",
  "min": "1",
  "max": "many"
},
{
  "name": "MemoryFacet",
  "description": "MemoryFacet captures information on computer memory equipping the resource and its usage. Any resource describing a
  "properties": null,
  "guiProps": [
    {
      "type": "number",
      "label": "Size",
      "name": "size",
      "value": "",
      "validations": [],
      "pattern": null,
      "propDescription": ""
    },
    {
      "type": "number",
      "label": "Used",
      "name": "used",
      "value": "",
      "validations": [],
      "pattern": null,
      "propDescription": ""
    },
    {
      "type": "text",
      "label": "Unit",
      "name": "unit",
      "value": "",
      "validations": [
        {
          "name": "pattern",
          "validator": "pattern",
          "message": "Wrong field format, required is: ^(Byte|kB|MB|GB|TB|PB|EB|ZB|YB)$"
        }
      ],
      "pattern": "^(Byte|kB|MB|GB|TB|PB|EB|ZB|YB)$",
      "propDescription": ""
    }
  ],
  "relationOptions": [
    "HasPersistentMemory"
  ],
  "relation": "HasPersistentMemory",
  "min": "1",
  "max": "many"
},
{
  "name": "MemoryFacet",
```


Since a resource may contain a variable number of facets of the same type, it was necessary to give users the possibility to dynamically add and remove the form potions related to facets. The overall GUI for the creation of a resource, therefore, is composed by the creation forms of the various facets, which can be added or removed depending on what is specified in the IS. The appendix to this document shows various screenshots related to the creation of a resource, namely an ‘Eservice’.

6. Main Functionalities.

In this section, the main functionalities of the application are described. Screenshots of the application which display them are shown in the Appendix.

6.1 Switching Context.

The IS accounts for different 'contexts', i.e. different working environments, each containing different resources. The main GUI of the application provides a drop-down menu to switch between the available contexts and consequently retrieve the list of the available resource types and instances.

6.2 Navigating Through Resource Types.

Resource types are organized hierarchically, therefore they are shown by means of a tree view. Some resource types are 'abstract', i.e., they do not allow for instances; these resource types are highlighted using italics and labelled with the letter 'A'; no event is triggered when clicking on an abstract resource type.

6.3 Browsing and Searching Resources.

'onClick' events are captured on each non-abstract node of the resource types' tree and trigger the research of all the correspondent resource instances, which are shown in table form. Resource instances of a given type are shown in a table which displays the resources' main features. The number of rows in the table depends on the pagination parameter set by the user. Alphabetical and numerical (ascending/descending) ordering for each column of the table is supported; searching on the basis of the resource name is also supported.

6.4 Visualizing Descriptive Information and Metadata.

For each resource type, it is possible to access a high-level textual description and its complete JSON specification. The JSON metadata file of each resource implementation is also available for display.

6.5 CRUD Operations.

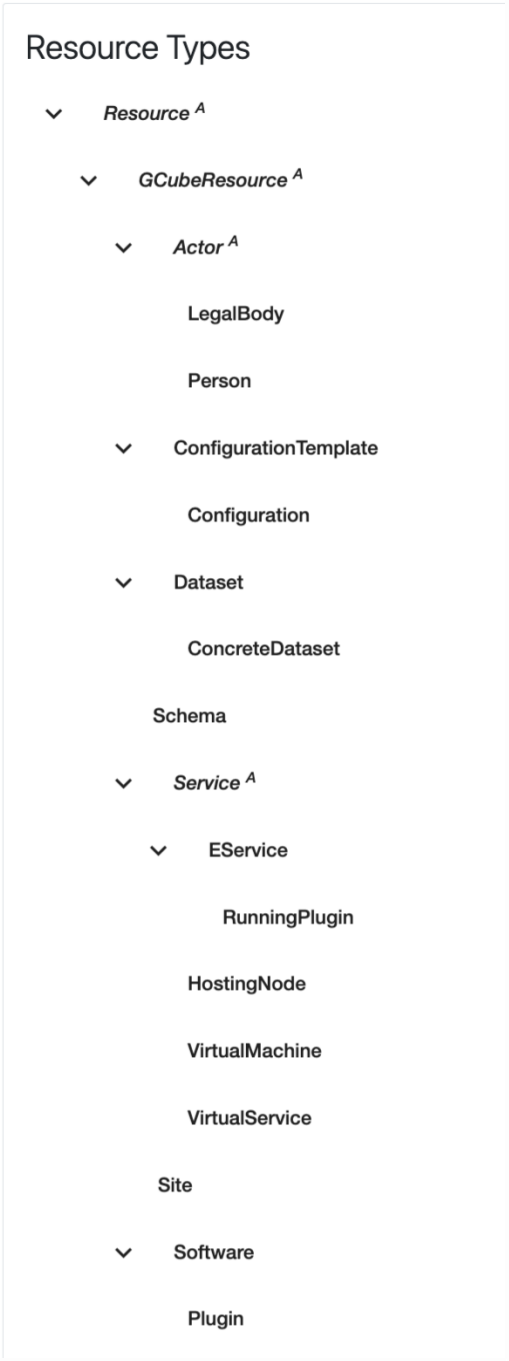
In each row of the table containing the resources two buttons are available, one for editing the resource and one for removing it. A button placed before the table allows to create a new resource instance for the current type.

Appendix A – Screenshots of the Main Functionalities.

This appendix gathers screenshots of the most important GUIs of the web application. All of them have been developed in Angular using Material design components (https://material.angular.io/). This was challenging, since the JHipster version we used did not support Material for Angular natively, so we had to manually import and set up each Material component we wanted to use (tree view, table, tabbed view, etc.).

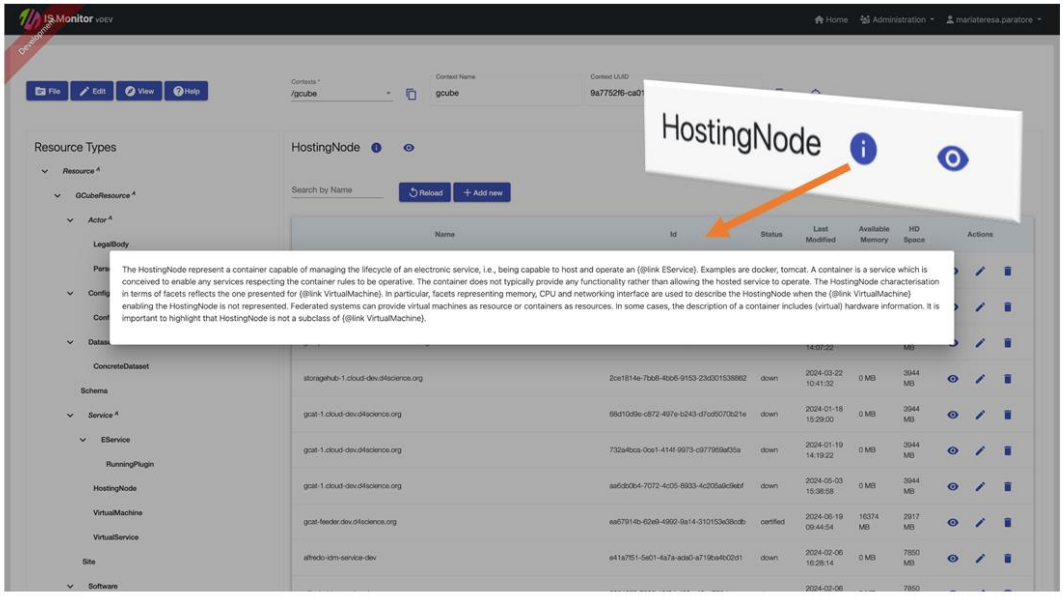
A.1 Browsing and Discovery

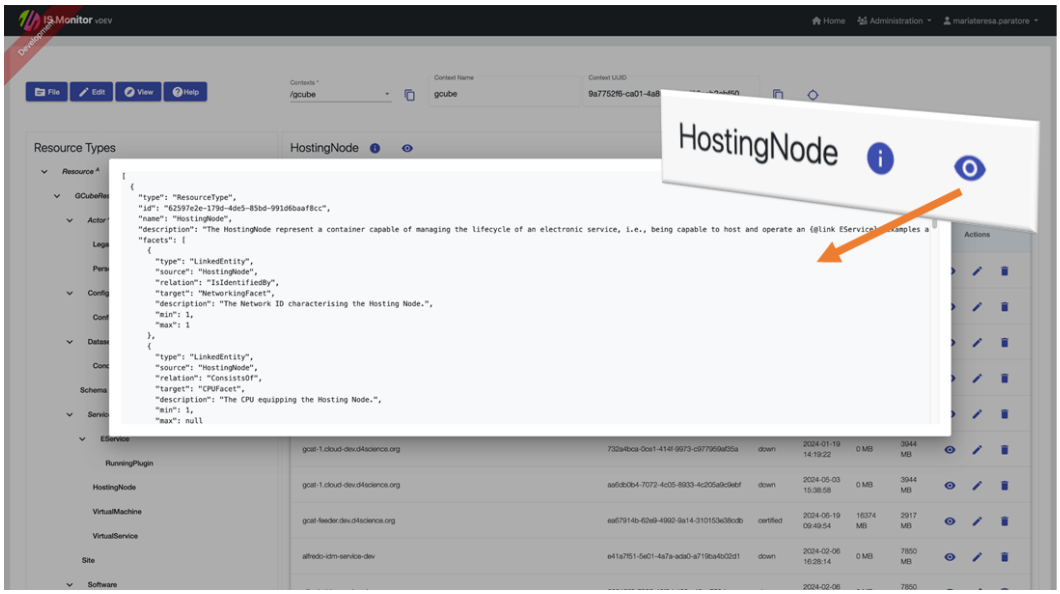
An expandable tree is used to show the relations between resource types. The component leverages the Material look and feel.



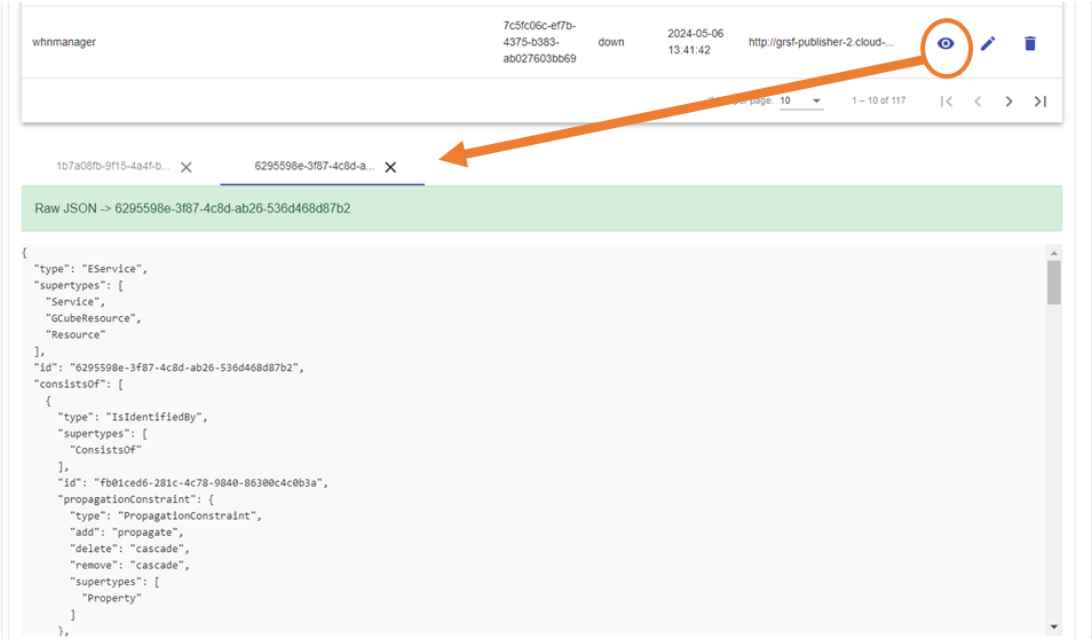
A.2 Providing Descriptions and Metadata.

For each resource type it is possible to view a general descriptive information and its JSON specification. The following screenshots have been taken for the ‘Hosting Node’ type.



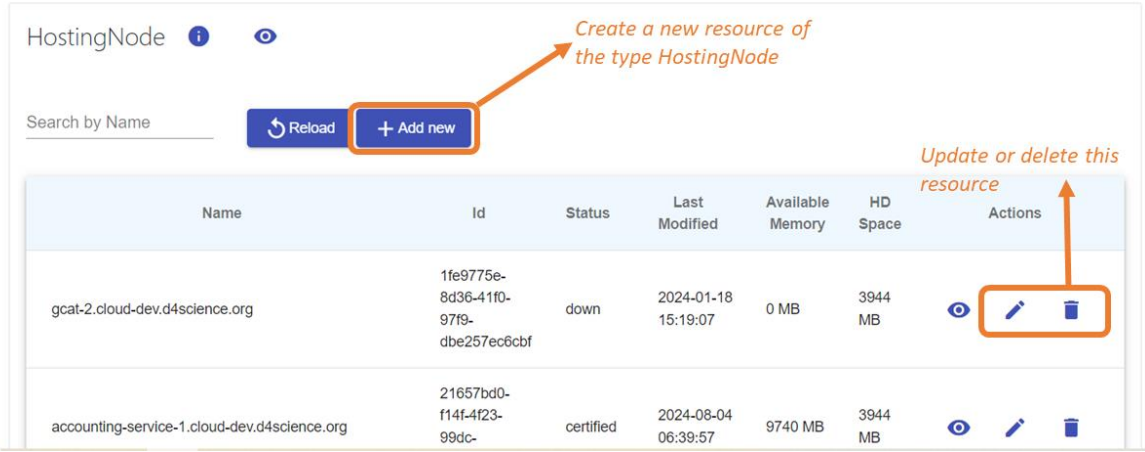


For each resource implementation, it is also possible to view its JSON metadata. All the resources of a certain type are shown in a table. Each row of the table contains the resource's main properties and three buttons, namely, 'view', edit, and 'delete'. When the 'view' button is clicked, a new tab is shown in a tabbed pane placed below the table, containing the JSON description retrieved from the IS.



A.3 CRUD Operations

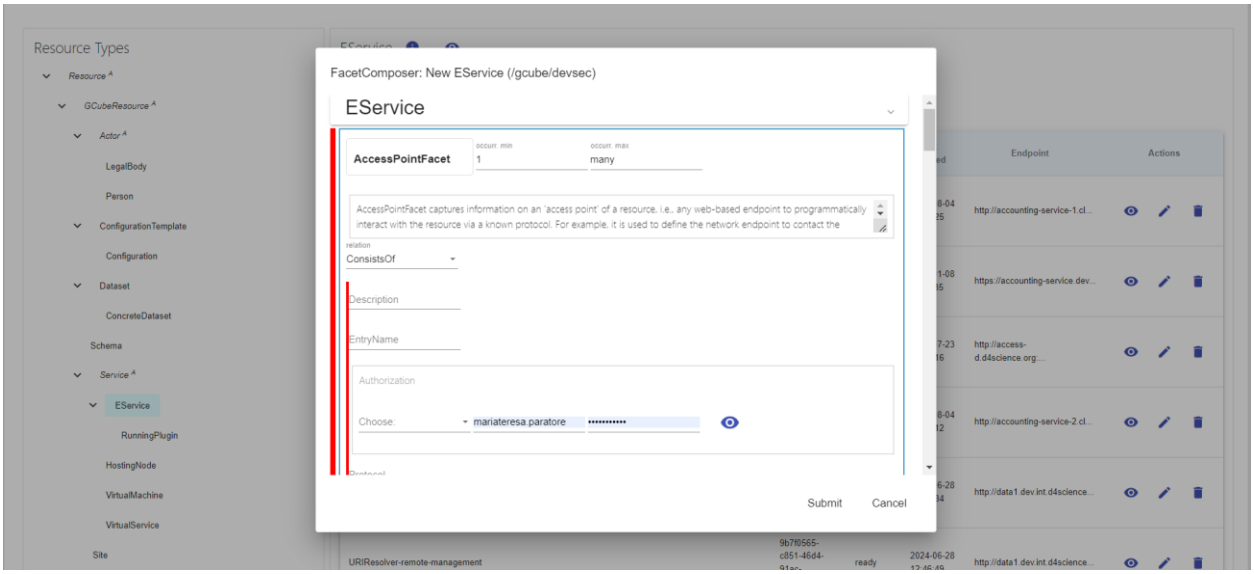
CRUD operations on the resources are allowed by means of the buttons highligted in the picture below. The 'Add new' button opens a form to create a new instance for a given resource type, while the 'edit' and 'remove' buttons placed on the row of each resource allow its modification or removal.

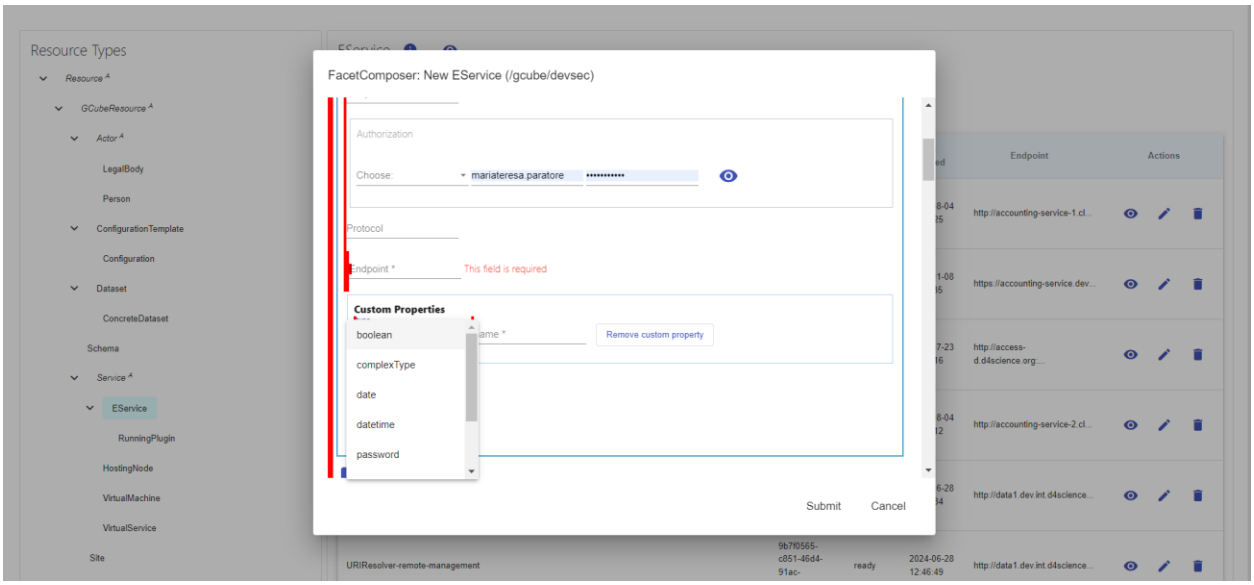


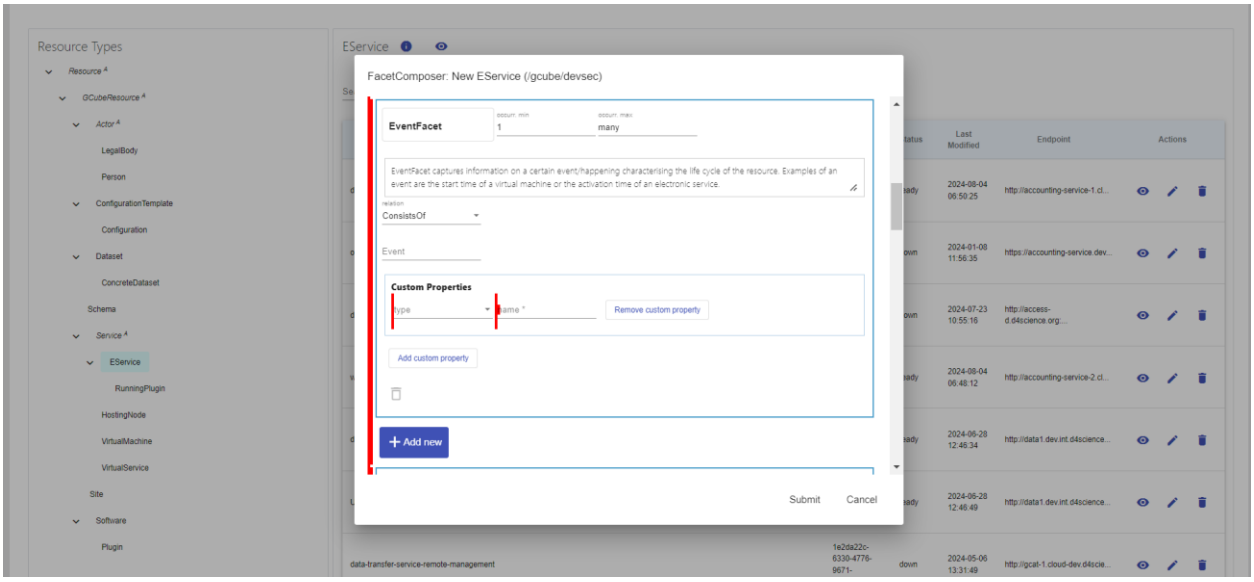
A.4 Creation Forms

The following screenshots show parts of the form for the creation of a new 'EService' instance. The form is composed of a number of facet-related forms, one for each facet that makes up the resource. Since the number of facets of the same type may be variable, facet-related forms can be added or removed

dynamically by the user. The screenshots here shown represent the portions of the GUI for the creation of a new EService, related to two of its facets, namely, an AccessPointFacet and an EventFacet.







Appendix B – Code Samples for the Creation of the "Resource Tree" Component.

This appendix shows the code used to create the resource-tree navigation component.

B.1 rsc-tree.component.ts

This is the Typescript code responsible for creating the Angular component <jhi-rsc-tree> in the JHipster environment (the 'jhi-' prefix is required by JHipster). Besides the selector name, the html template and css style filenames are declared. The 'providers' keyword identifies the list of services which will be used to fill the component with data.

```
   /* eslint-disable no-console */
import { Component , OnInit, Output,EventEmitter } from '@angular/core';
import { MatTreeNestedDataSource } from '@angular/material/tree';
import { NestedTreeControl } from '@angular/cdk/tree';
import { RestypesService } from 'app/services/restypes.service';
import { IResource } from 'app/services/i-resource';


@Component({
  selector: 'jhi-rsc-tree',
  templateUrl: './rsc-tree.component.html',
  styleUrls: ['./rsc-tree.component.scss'],
  providers: [RestypesService],
})

export class RscTreeComponent implements OnInit{
  nestedTreeControl = new NestedTreeControl<IResource>(node => node.children);
  nestedDataSource = new MatTreeNestedDataSource<IResource>();
  statusClass = 'not-active';

 @Output() public resourceTypeEm = new EventEmitter<string>();

 constructor(private rtService:RestypesService) {
  }

  ngOnInit(): void {
      this.rtService.fetchAll().subscribe(res => {
```

```
      this.nestedDataSource.data = res;
      this.nestedTreeControl.dataNodes = this.nestedDataSource.data;
      this.nestedTreeControl.expandAll();
       });
        }


hasNestedChild(_: number, node: IResource): boolean {
    if (node.children == null) {
      return false;
    } else {
      return node.children.length > 0;
    }
  }

  //TODO: InformationSystemResourceClient shoud pass a code, not a name!
    onClickNodeTree(node:IResource):void{
        //this.setActiveClass();
        this.resourceTypeEm.emit(node.name);
    }

 setActiveClass():void{
    this.statusClass = 'active';
  }

}
```

B.2 rsc-tree.module.ts

In short, a ‘module’ is a file which declares the dependencies of our component.

```
import { NgModule } from '@angular/core';
import { SharedModule } from 'app/shared/shared.module';
import { MatIconModule } from '@angular/material/icon';
import { MatTreeModule } from '@angular/material/tree';
import { RscTreeComponent } from './rsc-tree.component';


@NgModule({
  imports: [SharedModule, MatIconModule,MatTreeModule],
  declarations: [
    RscTreeComponent
  ],
 entryComponents: [RscTreeComponent],
  exports: [RscTreeComponent]
})
export class RscTreeModule { }
```

B.3 i-resource.ts

This is the simple interface we built to contain data retrieved from the service.

```
export interface IResource {
  name: string;
  id: string;
  children?: IResource[];
}
```

B.4 restypes.service.ts

The Typescript service that connects to the proper JHipster REST endpoint and retrieves data to fill our component. Data are parsed form the JSON format into objects according with the defined interface.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { IResource} from './i-resource';
import { ApplicationConfigService } from 'app/core/config/application-config.service';

@Injectable({
  providedIn: 'root',
})

export class RestypesService {
  httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' }),
  };

  constructor(private http: HttpClient, private applicationConfigService:
ApplicationConfigService){}


  fetchAll(): Observable<IResource[]> {
   const resourceUrl = this.applicationConfigService.getEndpointFor('api/is/resourcetypes');
    return this.http.get<IResource[]>(resourceUrl);
  }



}
```

B.5 rsc-tree.component.scss

This file defines the style to be applied to the component.

```


.example-tree-invisible {
  display: none;
}

.example-tree ul,
.example-tree li {
  margin-top: 0;
  margin-bottom: 0;
  list-style-type: none;
}

.example-tree .mat-tree-node {
    width: 300px;
}


.example-tree .mat-tree-node :hover {
    background-color: bisque;
}

/*
 * This padding sets alignment of the nested nodes.
 */
```

```
.example-tree .mat-nested-tree-node div[role='group'] {
  padding-left: 25px;
}


/*
 * Padding for leaf nodes.
 * Leaf nodes need to have padding so as to align with other non-leaf nodes
 * under the same parent.
 */
.example-tree div[role='group'] > .mat-tree-node {
  padding-left: 25px;
}

/*
.active{
  background-color: bisque;
}
*/
/*
.not-active{
    background-color: transparent;
}
*/
```

B.6 rsc-tree.component.html

This is the code which defines the html template for our component. It leverages Angular Material, in particular the Angular Material Tree (mat-tree) component.

```
 <mat-tree [dataSource]="nestedDataSource" [treeControl]="nestedTreeControl" class="example-
tree">
    <mat-tree-node *matTreeNodeDef="let node" matTreeNodeToggle>
        <button mat-button (click)="onClickNodeTree(node)">
             {{ node.name }}
        </button>
    </mat-tree-node>

    <mat-nested-tree-node *matTreeNodeDef="let node; when: hasNestedChild">
      <div class="mat-tree-node">
        <button mat-icon-button matTreeNodeToggle>
          <mat-icon class="mat-icon-rtl-mirror">
            {{ nestedTreeControl.isExpanded(node) ? 'expand_more' : 'chevron_right' }}
          </mat-icon>
        </button>

        <button mat-button  (click)="onClickNodeTree(node)">
             {{ node.name }}
        </button>
      </div>

      <div [class.example-tree-invisible]="!nestedTreeControl.isExpanded(node)" role="group">
        <ng-container matTreeNodeOutlet></ng-container>
      </div>
    </mat-nested-tree-node>
  </mat-tree>
```

Note: *This report is based on work carried out as part of the FOSSR project (https://www.fossr.eu/) at the InfraScience laboratory of the CNR Institute of Information Science and Technology 'A. Faedo' in Pisa, Italy (https://infrascience.isti.cnr.it/).*