Vrije Universiteit Amsterdam

Bachelor Thesis

Development of a front-end for an inventory of models for real-time collaborative modeling

Author: Yu Chen (2668777)

1st supervisor: dr. Ivano Malavolta
2nd reader: Kousar Aslam

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 15, 2022

Contents

Abstract

This project is to design and develop a web-based inventory viewer for the BUMBLE cross-platform real-time collaboration engine. The engine's model inventory holds DSML definitions, models, and their respective relationships. The web-based inventory viewer will offer a graphical user interface for people working on different platforms to examine and modify atomic units contained in the model inventory with convenience. This includes standard CRUD operations and real-time display of updated contents.

1 Introduction

Model-Driven Engineering (MDE) [6] is now one of the most common approaches for designing software, as modeling is a useful tool for improving software development's efficiency and effectiveness. To enable model designers to interact remotely and flexibly, it is highly demanded to combine the MDE with real-time collaboration (RTC) [17] technology. Although more research is being conducted in the field of collaborative MDE, most academic topics do not fulfil the requirements of industries according to the research conducted by BUMBLE [12]. BUMBLE (Blended Modelling for Enhanced Software and Systems Engineering) is a European project aiming for building a creative system and software development framework that is based on blended modelling notations/languages. To achieve cross-platform real-time collaboration on modeling, the BUMBLE collaboration engine is designed and will be deployed on a server, enabling multiple users to collaborate on the same models in different modeling environments.

Figure 1 depicts the detailed architecture of the BUMBLE cross-platform real-time collaboration engine. Model Inventory is a repository used by the collaborative engine to hold models, domain-specific modeling language (DSML) definitions, their relationships, and relevant metadata. A Model Inventory Controller is linked with the Model Inventory and offers CRUD[1] operations as well as authentication and authorization. This paper focuses on the development of the BUMBLE Web-based Inventory Viewer (red box in Figure 1) for the collaborative engine that communicates with the Model Inventory Controller through the REST API Gateway. The Web-based Inventory Viewer provides users working in different development environments with a graphical interface for inspecting model artifacts and interacting with Model Inventory units through REST API. The other parts depicted in the architecture of the engine are used for supporting cross-platform real-time collaboration and is out of the scope for this paper. This project's implementation is provided to the public as an open-source application[2].

[1]https://nl.wikipedia.org/wiki/CRUD
[2]https://github.com/Yunabell-VU/BUMBLE-Inventory-Viewer-Bsc-Project

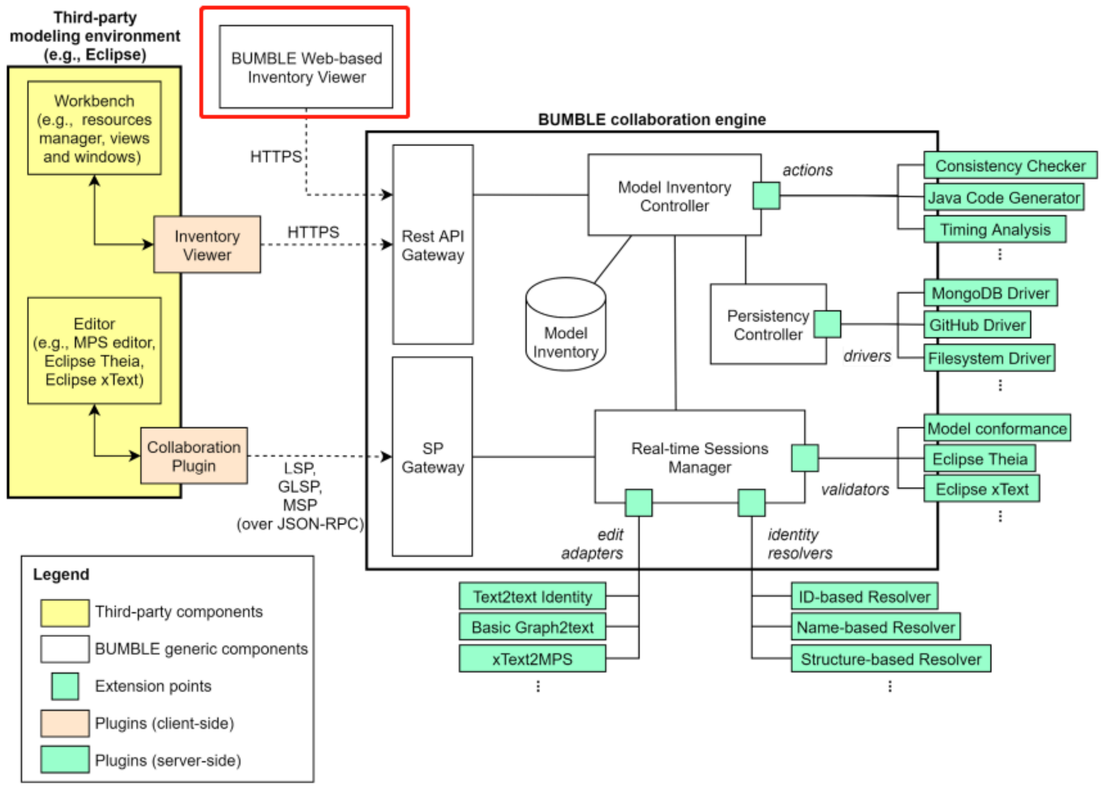


Figure 1: Architecture of the BUMBLE cross-platform real-time collaboration engine [12]

2 Background

The Model Inventory Controller server is build on the existing EMF.cloud model server [10], and the web-based inventory viewer is developed using the Vue.js [4] (Vue) web user interface framework. The server and the front-end client is able to communicate through the REST API Gateway. In addition, WebSocket protocol is used in this project to fulfill the requirement of displaying the most recent inventory data in real-time to users.

2.1 EMF.cloud model server

Eclipse is one of the most popular development environments that provide tools for MDSE, and the Eclipse Modeling Framework (EMF) is one of the popular technologies used for model-driven engineering in Eclipse [6]. Users are able to define metamodels in EMF using the metamodeling language Ecore. The Ecore file specifies the schema of model attributes and relations between model components, such as composition and many-to-one relations. EMF models are programmatically manipulable through a Java-based API, and EMF offers a converter for serializing and deserializing models to/from XMI.

EMF.cloud [2] is the umbrella project for components and technologies that make the EMF and its features accessible on the web and in the cloud. The current EMF implementation natively supports loading, model manipulation, and serialization. To connect web clients, the EMF.cloud model server builds a foundation on top of current technologies. It can control the state of loaded models in a shared editing domain during run-time. It permits applying modifications using a command pattern and registering for modifications. It offers a Java and JavaScript REST API that supports various formats (JSON or XMI)[10].

2.2 Front-end framework

2.2.1 Vue.js

Vue.js is a user interface framework that builds on top of HTML, CSS, and JavaScript standards. It offers declarative and component-based programming methods that improve the productivity of user interface development. Vue is selected as the framework for this project because to its great flexibility and fastest performance[18] among popular frameworks currently available. It is reasonably simple to learn and use, and better suited for smaller projects[19].

2.2.2 Vuex

State management is important for the web-based inventory viewer in order to maintain track of the current status of inventory items and user information. Since they are application-level states and are used across components. Vuex, the official state management library for Vue.js, is used to fulfill this need. Vuex adheres to the same Flux principles[9], a set of guiding principles that define statement management patterns:

- Principle1: Single source of truth
- Principle2: Data is read-only

- Principle3: Mutations are synchronous

According to the three principles, shared data across components shall be maintained in a single location, the **store**. In each component, it shall be simple to retrieve data from the store but impossible to directly modify the data. Updates to the store are done synchronously to ensure that all components read and render using the same data. Figure 2 illustrates how data state flow works using Vuex [5].

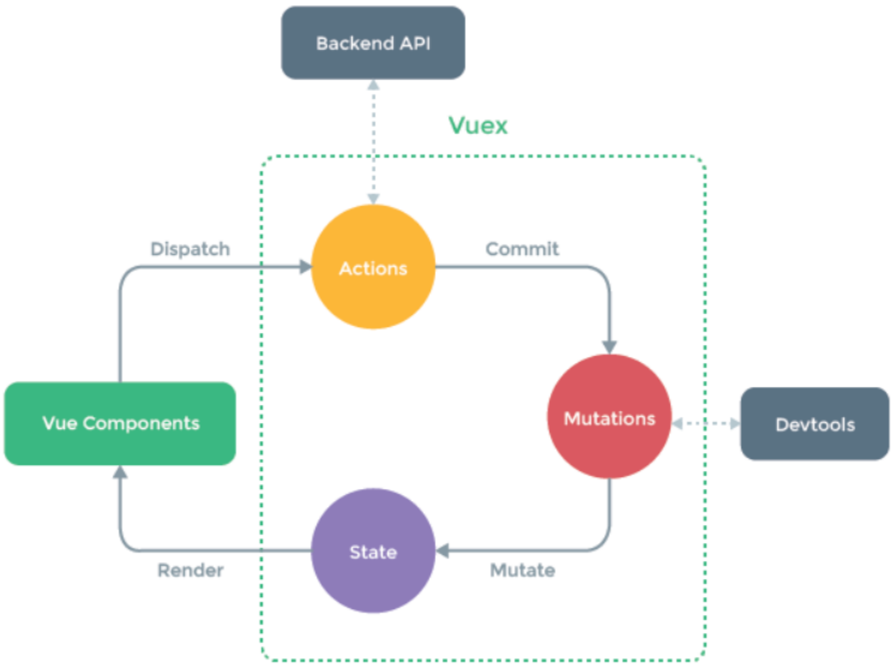


Figure 2: Vuex state flow [5]

Initial application level states are defined in the store State and supplied to Vue components using `getters`. If a component requires to modify State data, it `dispatches` an Action, which in turn sends a `Commit` event to the associated Mutation. The Mutation then updates the State, which is rendered across all components.

State management Patterns provide the advantage of managing the state systematically, which improves the maintainability and scalability of the project[9].

2.3 REST API

A Rest API Gateway is used in the BUMBLE project to connect the model inventory controller with the web-based model inventory viewer. It enables cooperating parties to interface with the controller on non-homogeneous platforms. This is made possible by the portability feature of REST API.

REST (abbreviated form for REpresentational State Transfer) is a protocol-independent architectural style developed to facilitate the development and organization of distributed systems[7]. RESTful API is a collection of APIs created with the REST architecture. It offers a simple interface and great performance. This indicates that the communication is effective and it takes less effort to deploy and adding new components or features on both the server and client sides. Portability is a significant feature of the REST style. The REST architecture is technology and language independent. This means that a REST-style system can be accessible by a variety of platforms and languages using a uniform interface.The client using REST API is only required to know the protocol endpoints and not the underlying mechanism of API services. Any modifications to the back-end or server-side do not effect the front-end implementation. This also preserves the security of the back-end, as client-side self-discovery is impossible.

The BUMBLE Model Inventory Controller uses the HTTP protocol and RESTful API. The requests delivered from the client to the server shall include the following information[3]:

- HTTP Method
- Path
- Query Parameter
- Request Body

HTTP method such as `GET` and `POST` specifies the type of request action. The path specifies specific tasks such as requesting a model or validating a model. Certain requests have optional query parameters. The majority of query parameters are used to identify the specific model and the server's return data type. HTTP methods such as `POST` and `PUT` require a request body send to the server for further process.

2.4 WebSocket

WebSocket is an IETF[4]-standardized computer communication protocol. Through a single TCP connection, it enables full-duplex communication between a client and remote server[8]. WebSocket sends real-time data more quickly and with reduced latency compared to other continuous polling protocols[16]. The current API specification that allows web applications to use this protocol is known as WebSockets. It is a working draft and maintained by W3C[3].

The EMF.cloud model server supports model subscriptions through a WebSocket endpoint. By subscribing to a particular model, any server-posted updates to that model will be broadcasted to all subscribers. This implies the web-based model inventory viewer can be notified of Model Inventory updates in real time as a subscriber.

[3]https://github.com/eclipse-emfcloud/emfcloud-modelserver#readme
[4]https://www.ietf.org/

3 Design

The project's goal is to create a web-based inventory viewer that will make it convenient for model inventory users to examine and interact with the megamodel that is saved in the inventory[12].

3.1 Model Inventory

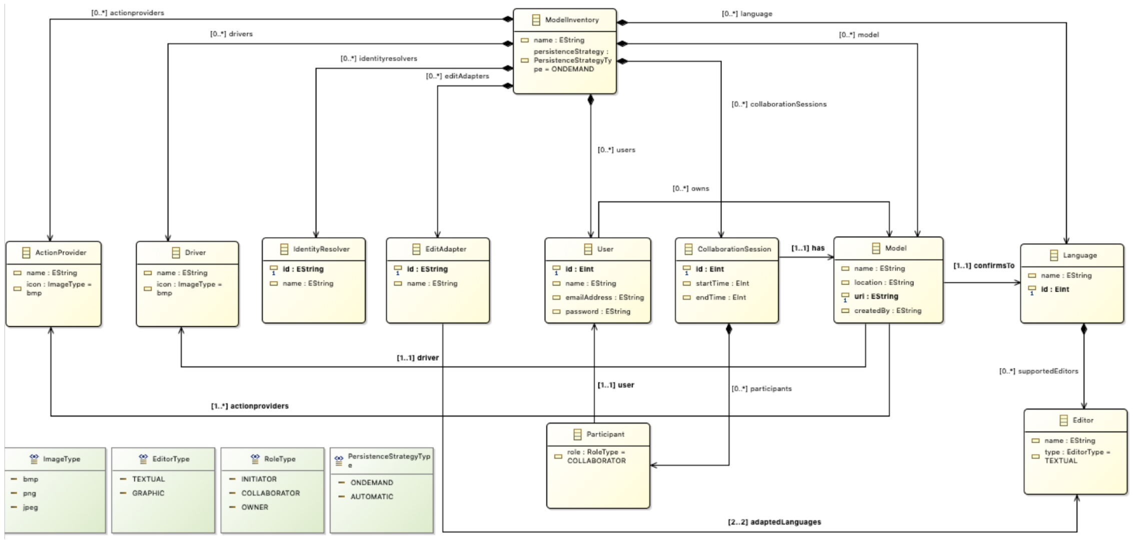


Figure 3: The metamodel of Model Inventory megamodel

Figure 3 is the class diagram that illustrates all the classes and enumerators that belongs to the Model Inventory megamodel. A version of the figure with a high resolution is available in the open-source repository [5].

3.1.1 Classes

1. **Languages**: Domain-Specific Modeling Language used to define models. This class contains the name of the language and one or many supported editors.
2. **Model**: Each model possesses the following attributes: its name, location, URI, and creator. It is evident from the references which language the model conforms to, as well as its action provider and driver.
3. **User**: Name, password, and email address are included in the User attributes. In addition, it contains references to models owned by the user.
4. **CollaborationSession**: Each collaboration session is related to one model and it is the session that opened for users to collaboratively modify a model in real-time. It has start and end time as its attributes. *Participants* class is contained in this class which indicate the users that participate in the session.

[5]https://github.com/Yunabell-VU/BUMBLE-Inventory-Viewer-Bsc-Project/blob/master/demo/megamodel$_{i}$*nventory.png*

5. **Plugins**: *Driver*, *ActionProvider*, *EditAdapter* and *IdentiryResolver* are plugins used on the server side to support real-time collaboration session by establishing and maintaining a mutual connection channel with the collaboration engine.

3.1.2 Enums

The enumerators that used as data type for its corresponding classes. For example, *RoleType* is used to define the role of participant that joined in collaboration sessions. It has three elements:

- INITIATOR: The user that starts a collaboration session.
- OWNER: The user who owns the model associated with the session.
- COLLABORATOR: The user participates in the session.

Other Enums such as *EditorType* and *PersistenceStrategyType* will not be covered in depth since they are outside the scope of this project.

3.2 Use Case

The web-based inventory viewer has multiple use cases. One example of these use cases is a user (collaborator) working together with another user on a particular model. The model on which the collaborator is now participating could be found on the homepage of the inventory viewer. The collaborator has access to the model's details and the collaboration session in which the model has been edited. Additionally, the collaborator may see who else is working on the same model.

Maintaining the plugin values and adding or removing users from the inventory would be the use case for an administrator of the inventory.

3.3 Requirements

Given below is a list of the functionalities that the inventory viewer shall provide.

R1 The inventory viewer shall have login and logout functions and shall verify the user's input in terms of username and password.

R2 The model inventory viewer will show the detailed structure of the model inventory including:

- **Classes**: Each class type shall have information about all its attributes and references.
- **Enums**: Each enumeration type shall have information about all its enumerators' names and values.

R3 The inventory viewer shall list all the class instances stored in the inventory, this includes:

- models

- collaboration sessions
- languages
- users
- plugins

R4 Instances listed in Requirements 3 shall be able to be edited, the edit operations include:

- create a new instance
- update an existing instance
- delete an existing instance

R5 The inventory view shall update the changes from the server in real-time.

3.4 User Interface

The user interface was designed with respect to the usability principles [13] and accessibility guidelines [1] to meet the functionalities proposed in the requirements with usability and accessibility.

The inventory viewer has two distinguish layouts in page style, namely the **Login page** layout and the **Home page** layout. The page structure for the entire web-based inventory viewer is depicted in Figure 4. All the structures listed behind the Home page are different sub-pages that are displayed on the information board. This will be explained in details in the Home page section below.

3.4.1 Login page

The process for logging into and out of the inventory viewer is shown in Figure 5.

If a user exits the inventory viewer without logging out, the inventory viewer shall record the login status. This indicates that if the login status is `isLogin`, the user is not required to log in using the username and password. If the user is logged in, the Login page shall not appear or be able to be opened. The inventory viewer will only be able to route users to the Home page while this status is in effect.

The user is routed to the Login page where their input will be collected and confirmed if `isLogin` is not recorded or if its value is false. A user who successfully logs in is taken to the Home page. Users can reset their login status and exit the Home page by clicking the logout button on the main page. The user shall see the Login page after activating the logout button.

3.4.2 Home page

The Home page of the inventory is where users may access information stored in the model inventory. Within the Home page layout, sections such as collaboration sessions and languages are located. The default view of the Home page is the Inventory-Instances view. This means that this is the sub-page to which the user will be directed after logging in. Except for the Inventory section, all sections of the Home page are classes that belong

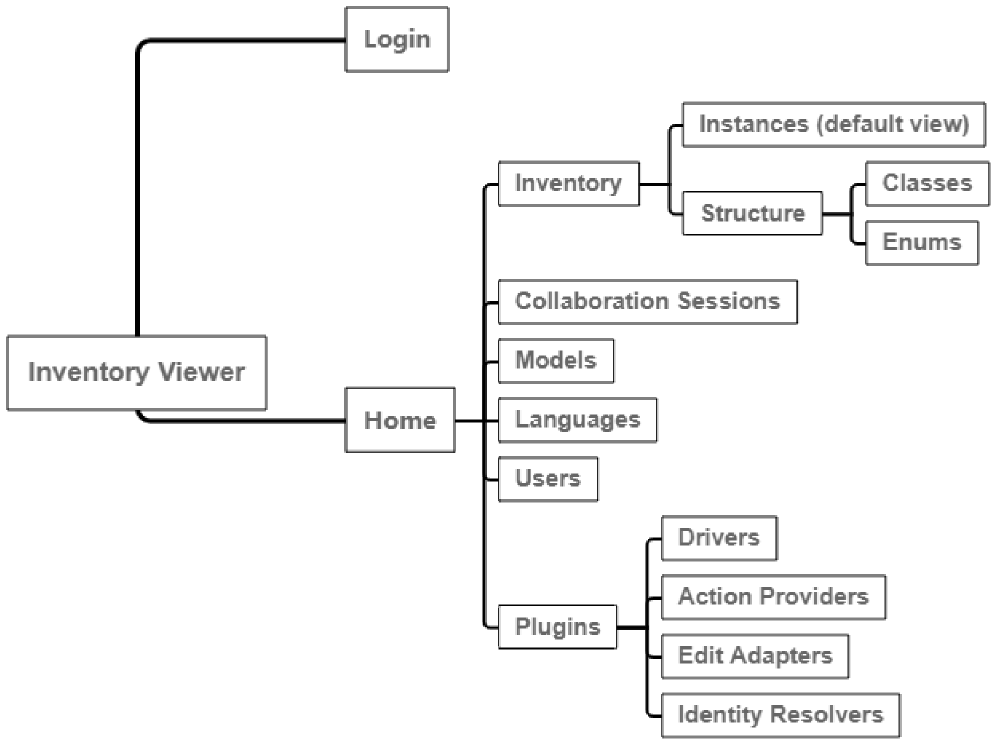


Figure 4: Overview of the layout of the inventory viewer pages

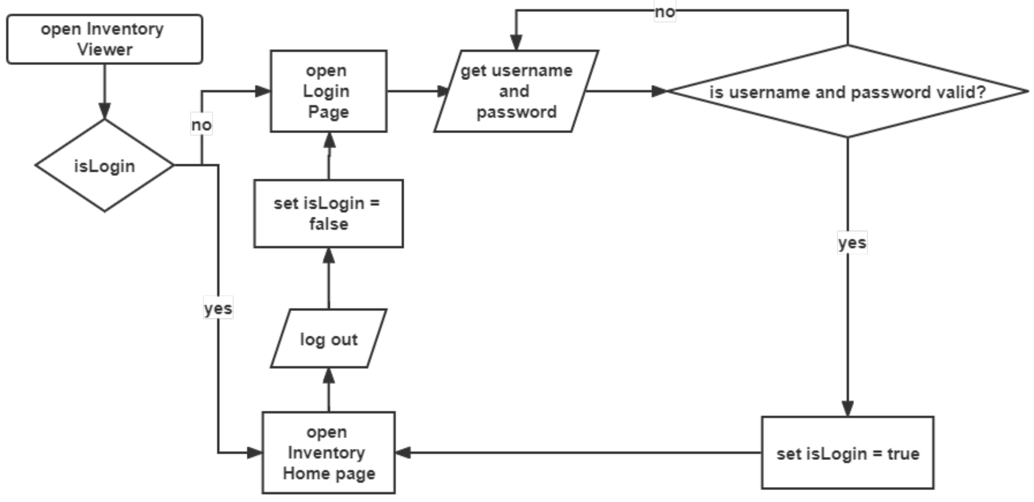
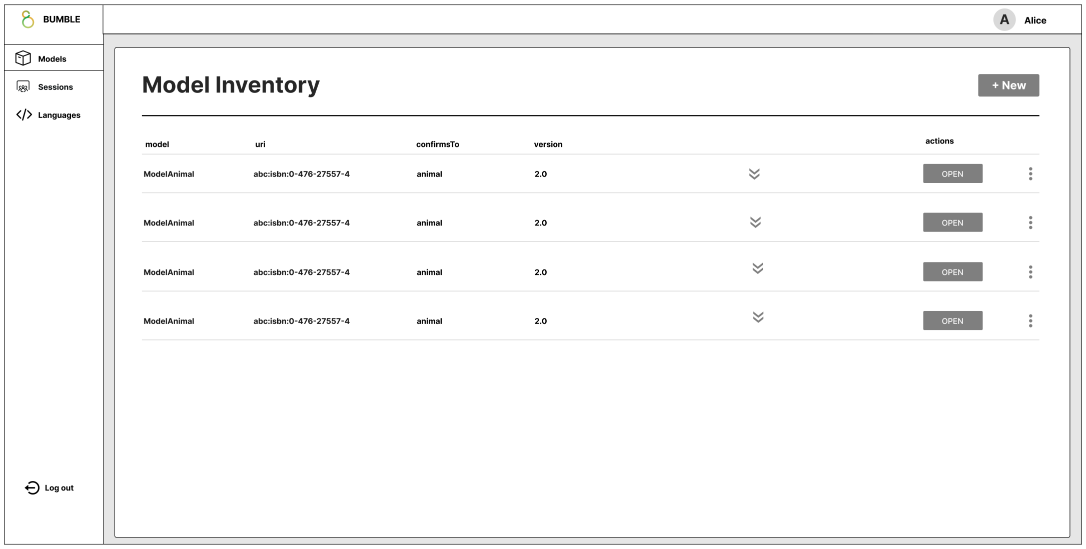


Figure 5: Login and Logout Flow

to the Model Inventory megamodel. Drivers, Action Providers, Edit Adapters and Identity Resolvers are grouped as one Plugins section.

To progress the design of the inventory Home page layout, a low-fidelity prototype was created and, based on the low-fidelity sketch, a medium-fidelity prototype was drawn to illustrate the interface features and interactive components. Figure 6 shows the example of the medium-fidelity prototype for the inventory viewer's Home page



model	uri	confirmsTo	version		actions
ModelAnimal	abc:isbn:0-476-27557-4	animal	2.0		OPEN
ModelAnimal	abc:isbn:0-476-27557-4	animal	2.0		OPEN
ModelAnimal	abc:isbn:0-476-27557-4	animal	2.0		OPEN
ModelAnimal	abc:isbn:0-476-27557-4	animal	2.0		OPEN

Figure 6: A medium-fidelity prototype of inventory viewer Home page

Three sections make up the entire page. The header is at the top of the page, the navigation bar is on the far left, and the information board fills the remaining space. Header can be used for the following functions:

1. Display the current user information, this includes the user's avatar and the username.

2. Display notifications about server updates.

3. Allow users to choose between automated updates and manual page refresh.

The last two functions are not in the scope of this project. Ideas of their usage will be discussed in section 7.2.

The default subsection to display is the Inventory-instances sub-page (default view). Users can log out of the Home page and use the navigation bar to browse between the sub-sections depicted inside the Home page in Figure 4. In order to improve the user experience, the background and font color of the activated section of the navigation bar shall differ from those of other sub-pages. In addition, the entire page is not reloaded when a section is altered using the navigation bar. Only the information board displays the section corresponding to the navigation menu, while the header and navigation remain static.

The information board consists of two distinct vertically-displayed parts: the title part and the content part. The title part displays the title of the current section, such as **Languages**, as well as function buttons such as the create a new instance button and sub-section navigation buttons. The content part lists all the instances in the current section. The Inventory section and Collaboration Session section are distinct from other sections in the way that they are **read only**. For the other sections having operational functionalities, the edition and deletion operations are provided for each instance.

For example, if the user opens the Models section, the user shall be able to see the section name Models on the left-hand side of the title part and a **+New** button on the right end of the title part. By clicking the +New button, the user will be able to see an overlay (Figure 7) on top of the information board containing a list of all the model class's attributes, and will then be able to enter values for these attributes. The edition function is similar to the creation function in that it similarly opens an overlay, but in the edition function's overlay, the user can adjust the instance's information depending on the original information.

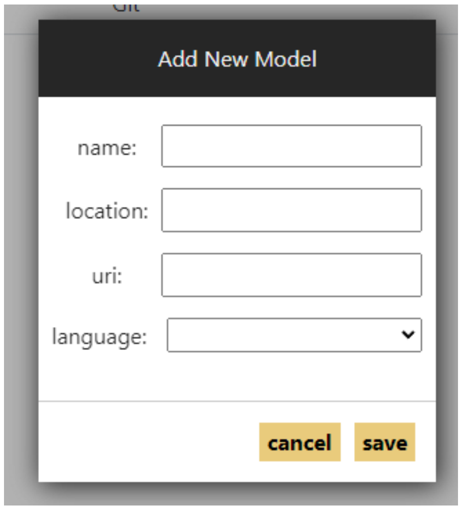


Figure 7: The overlay form for model instance creation

A further distinction between the Inventory section and the other sections is that it not only showing the instances stored in the model inventory server, but also the main structure of the model inventory as a megamodel, including all classes and enumeration data structures.

4 Implementation

The aim of this project's implementation is to provide a web-based graphical user interface for the model inventory. It shall conform to the design specifications and user interface outlined in the Design section. This project's model inventory server relies on the EMF.cloud model server. Its inner workings are outside the scope of the project. From the standpoint of the front-end inventory viewer, the inventory server provides HTTP endpoints for the clients to get and edit the inventory structure recorded in the Ecore file and instances such as models and languages saved in XMI format. It should be noted that only XMI content is changed via the front-end in this project, and the schema saved in the Ecore file is read-only.

This report will focus on the following phases to describe how the front-end of the inventory viewer was developed:

1. Structure of application
2. REST API and WebSocket
3. Store
4. CRUD

4.1 Structure of application

Components[6] are the basis of a Vue project. Customized contents (HTML and CSS) along with their logic (JavaScript) are encapsulated in each component. In this project, the Vue framework consists of various sized nested components. Figure 8 depicts the primary layout structure of this project's components. Notice that there are a number of local components nested inside the components depicted in this picture, however they are not presented here.

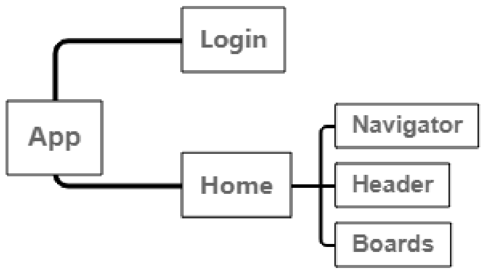


Figure 8: The main structure of components for inventory viewer

Component App.vue is the entrance component of the inventory viewer and it is mounted on main.js which serves as the application launcher. The HTML part of App.vue comprises simply a router-view component. It is a component provided by the official

[6]https://vuejs.org/guide/essentials/component-basics.html

Router[7] library and intended for usage with Router logic. In src/router/index.js file, the rules of routers are defined. In Section 3.4.1, the design requires a login status check and, depending on the login state, directs the user to either the login page or the homepage. This is achieved by the condition defined within the router.beforeEach function. If the login status saved in localStorage is false, the router will route to the /login URL and App will mount the Login component. In contrast, if the login status is true, the router and thus App will lead users to the Home component and prevent them from accessing the login page.

As described in Section 3.4.2, when the user clicks on the navigation menus to switch between boards, the page as a whole shall not refresh but only updates the board view. As shown in Listing 1, the Home component consists of three components and the Boards part listed in Figure 8 is replaced with a router-view component.

```
<template>
  <div class="layout-wrapper">
    <div class="layout-left">
      <Navigator :close-web-socket="closeWebSocket" />
    </div>
    <div class="layout-right">
      <Header />
      <router-view></router-view>
    </div>
  </div>
</template>
```

Listing 1: HTML part of Home component

The router-view component serves as a placeholder, and the router determines which component will be mounted in its place. With the aid of the Router library, it is possible to render a portion of the page when the URL path changes via child routers. Once the user clicks on an inactive navigation menu on the Navigator component, the router will redirect to another child path of Home, therefore replacing the current board component with one that matches to the path name. For example, if the current sub-page is Users, then the Home component renders Navigator, Header and User components. And when clicking the Languages menu, the User component is substituted with the Language component, allowing the user to enter the Language view.

4.2 REST API and WebSocket

To test how the REST API works and what type of data can be fetched from the server, an API platform Postman[8] is used before the real front-end development to aid in exploring the mechanisms of the server API. Separately kept on the Model Inventory server are an Ecore file and an XMI file that specify the rules of the megamodel Model Inventory and the instances of classes belonging to the megamodel.

Listing 2 shows the result converted from megamodel Model Inventory by EMF. It is saved in an Ecore file with XML format. The EMF.cloud model server will further convert the content to JSON type. Listing 3 gives part of the response fetched from the server

[7]https://router.vuejs.org/introduction.html
[8]https://www.postman.com/

using GET request required from HTTP endpoint: `/api/v2/models/?modeluri=modelInventory.ecore`

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage ...>
  <eClassifiers xsi:type="ecore:EClass" name="ModelInventory">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" eType="
        ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="language"
        upperBound="-1"
         eType="#//Language" containment="true"/>
    ...
  </eClassifiers>
    ...
</ecore:EPackage>
```

Listing 2: Example of modelInventory.ecore

The server-retrieved JSON data provides a clearer picture of the sort of information the model inventory megamodel may provide. The `"type": "success"` tells the status of the response. If the `modelInventory.ecore` file is not stored in the server's workplace, the server shall response with an `error` status type.

```
{
    "type": "success",
    "data": {
        "$type": "http://www.eclipse.org/emf/2002/Ecore#//EPackage",
        "name": "modelInventory",
        ...
        "eClassifiers": [
            {
                "$type": "http://www.eclipse.org/emf/2002/Ecore#//EClass",
                "$id": "//Model",
                "name": "Model",
                "eStructuralFeatures": [
                    {
                        "$type": "http://www.eclipse.org/emf/2002/Ecore#//
                            EAttribute",
                        "$id": "//Model/name",
                        "name": "name",
                        "eType": {
                            "$type": "http://www.eclipse.org/emf/2002/Ecore
                                #//EDataType",
                            "$ref": "http://www.eclipse.org/emf/2002/Ecore
                                #//EString"
                        }
                    },
            ...
}
```

Listing 3: Example of JSON data get from model inventory ecore

In `data.eClassifiers`, all the classes and enumeration types are listed. From the JSON content shown in Listing 3, information about ”Model” such as its type and its attributes include the attributes datatype can be retrieved to understand this classifier.

As is the case with the Ecore information on the server, data about the instances stored in the inventory is also obtained via GET requests from HTTP endpoint: `/api/v2/models/?modeluri=ModelInventory.xmi`

Listing 4 gives the example of model instances fetched from the server. In contrast to the Ecore file, in which just the schema is defined, each class such as Languages and Users in the Xmi file is instantiated with many instances, and all of its properties are assigned with precise values.

```
{
    "model": [
            {
                "$id": "ModelShape.xmi",
                "name": "ModelShape",
                "location": "Git",
                "uri": "ModelShape.xmi",
                "confirmsTo": {
                    "$type": "file:/D:/emfcloud-modelserver-bumble/examples
                        /org.eclipse.emfcloud.modelserver.example/target/.
                        temp/workspace/modelInventory.ecore#//Language",
                    "$ref": "//@language.0"
                }
            },
            {
                "$id": "ModelAnimal.xmi",
                "name": "ModelAnimal",
                ...
                },
            }
        ]
}
```

Listing 4: Example of JSON data get from model inventory xmi

The aforementioned results were achieved with the Postman API interface. However, for the web-based inventory viewer to operate, the API must be properly implemented in the front-end frame. In this project, the Axios [9] library is used to aid the web client in sending HTTP requests to the server. Various sorts of Axios requests are encapsulated in arrow functions for better reusability and maintainability.

```
export const get = (url) => {
    return new Promise((resolve, reject) => {
        axios.get(url, {
            baseURL:"http://localhost:8081/api/v2",
        }).then((response) => {
            resolve(response.data)
        }, err => {
            reject(err)
        })
    })
}
```

Listing 5: Encapsulated get method for GET requests

[9]https://axios-http.com/docs/intro

The example of the get method created in this project is shown in Listing 5. It produces a Promise[10] object, which in JavaScript is a proxy that enables asynchronous functions, such as axios.get in this case, to return a promise to deliver the value in the future. A successful response from the server will then deliver the response.data through this get function. To update the instances in the model inventory, a similar `put` function is created using axios.put with new JSON data as parameter.

Since REST APIs are stateless, communication between the client and server ends when the server sends the response to the HTTP request from the client. However, in order to subscribe to server updates in real-time, we require a continuous connection between the inventory and inventory viewer. The WebSocket **subsribe** endpoint provided by the EMF.cloud model server makes the real-time communication possible.

```
export default {
  name: "Home",
  data() {
    return {
      webSocket: null,
    };
  },
  methods: {
    closeWebSocket() {
      this.webSocket.close();
    },
  },
  mounted() {
    this.webSocket = new WebSocket(
      ‘ws://localhost:8081/api/v2/subscribe?modeluri=ModelInventory.xmi‘
    );
    this.webSocket.onmessage = (event) => {
      const data = JSON.parse(event.data);
      if (data.type === "fullUpdate") {
        this.$store.dispatch("updateModelInventory");
      }
    };
  },
};
```

Listing 6: WebSocket usage in VUE

As described in Background 2.4, W3C provides a standard WebSocket API that enables clients to utilize the WebSocket protocol. After a WebSocket object has been constructed in JavaScript, the onmessage function allows developers to specify how to handle server-sent events. Listing 6 displays the WebSocket-related Vue component snippet from this project. The home page's mounted stage initiates a web socket to subscribe to ModelInventory.xmi stored on the model inventory server. Through this subscription, the client will be notified of any ModelInventory.xmi server-side modifications. Once the web socket gets the updating event and verifies that the event type is "fullUpdate", an action will be dispatched to update the model inventory states contained in the inventory viewer's store. This will result in a re-rendering of the web page's presented data.

A click on the Logout button will trigger closeWebSocket function and then close the connection.

[10]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global$_{O}bjects/Promise$

4.3 Store

The Store placed in the inlinestore/index.js file serves as the project's internal inventory. Once the App component has been mounted, dispatches are sent to the store Actions in order to retrieve the model inventory Ecore and Xmi contents from the server using the get method. After receiving the data, the action emits a commit to Mutations in changing the States associated with model inventory data. Components that require inventory data to render their pages will then retrieve the data needed via mapGetters[11]. As stated in Background 2.2.2, Vuex store enables the monitoring of state changes and ensures that all components access the data synchronously.

The verification of login relies on data retrieved from the model inventory server about the user. This user data is an array of objects containing all instances of users with name and password information. On the login page, the handleLogin method verifies if the user's inputs match any recorded in the store. If a match is detected, this user information will be saved as the currentUser state in the store. The information about the user presented in the Header is obtained from the store's currentUser. This state is also used when creating a new model. The current user's name will be added automatically to the createdBy property of the model.

4.4 CRUD

On the web-based inventory viewer, users can perform CRUD actions on model inventory's class instances with the exception of collaboration sessions. This is because the model inventory controls the sessions, and users of inventory viewer are not permitted to edit them in this project.

The REST API provided from the server and the fact that model inventory exists as a megamodel makes the CRUD requests different from common deployment. Normally, create, delete and update requests are separated using different HTTP methods such as POST, DELETE and PUT. The API offered by the server to DELETE and POST a model, however, only supports independent models. Model inventory is a single megamodel, and all class instances are kept in the ModelInventory.xmi file. This transforms every CRUD operation on a model inventory instance into a UPDATE operation on the ModelInventory.xmi file. Therefore, all of create, delete and update actions applied to the instances are achieved through PUT request to update the full model inventory xmi.

[11]https://vuex.vuejs.org/guide/getters.html

5 Model Inventory Viewer

This section illustrates the web-based model inventory viewer in its complete functionality. If a user's login status is false or null, they will initially be directed to the login page. Figure 9 depicts how users can input their username and password before clicking the Login button. In the inventory viewer, there is no new account registration function.

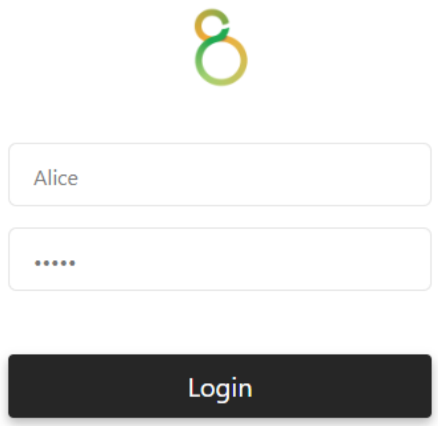


Figure 9: Login page with user inputs

If there is no matching username in the Store, a warning will appear stating that the username does not exist (Figure 10). If the username is valid but the password is incorrect, the alert dialogue will state "wrong password."

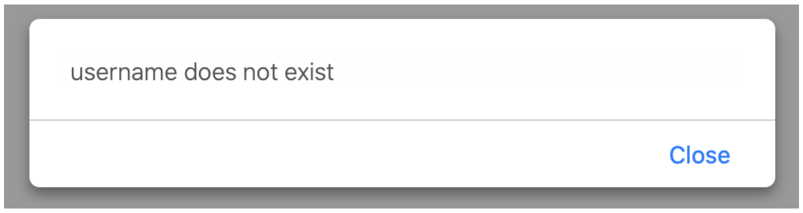
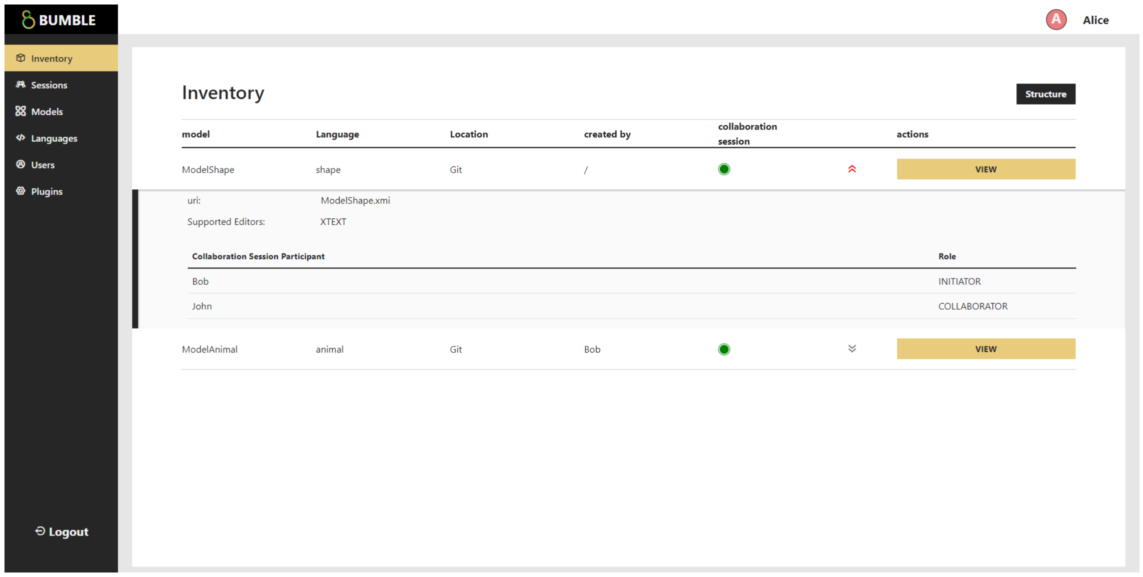

username does not exist

Figure 10: Alerts on wrong username

After successfully logging in, the user is sent to the Inventory - instances sub-page (Figure 11) by default. At this view, a table of models is displayed, with each row representing an inventory model. On the collapse mode of rows, users may view the languages used for the model, as well as its location, creator, and whether or not a collaborative session exists. If there is a collaboration session for the model, the circle indicating the session's state will be green. Otherwise, it should be red. By pressing the down arrow symbol, the row can be extended to reveal further information, such as supported editors and session participants. The VIEW button at the right end of each row leads to the real location of the model, such as a GitHub repository. This project simply offers the button, not the function for redirecting.



Inventory

model	Language	Location	created by	collaboration session		actions
ModelShape	shape	Git	/	●		VIEW
ModelAnimal	animal	Git	Bob	●		VIEW

uri: ModelShape.xmi

Supported Editors: XTEXT

Collaboration Session Participant	Role
Bob	INITIATOR
John	COLLABORATOR

Figure 11: Inventory Instances view

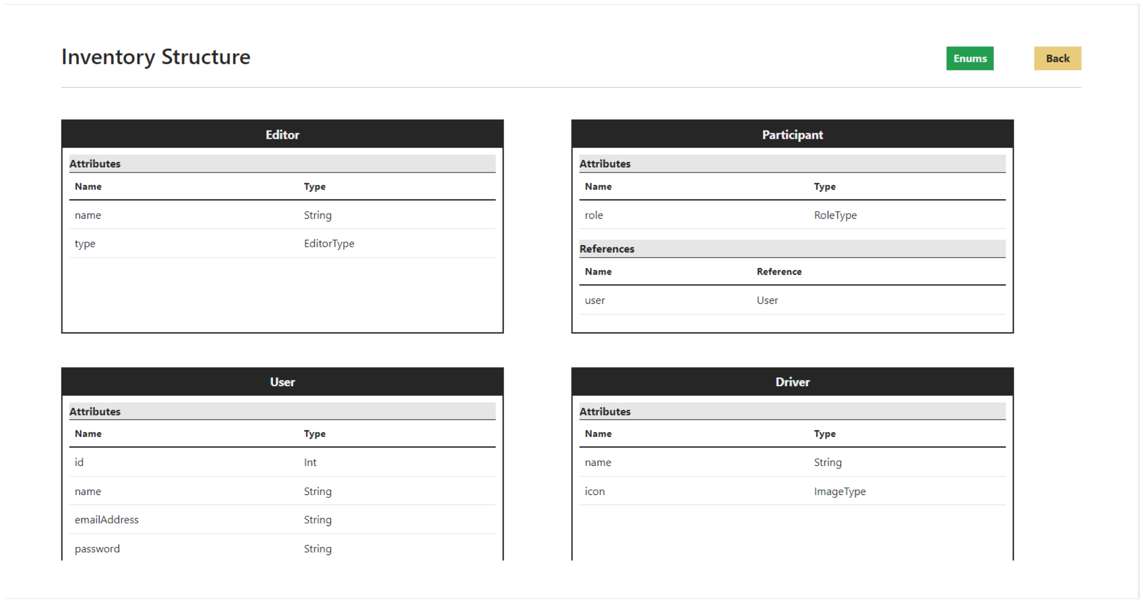
By pressing the Structure button, the Inventory - Structure sub-page can be accessed. Here, all classes and their attributes are displayed in card gallery format. To view enumeration types, users should click the Enums button located in the upper-right corner. Similar to classes, enumerators are also presented in card gallery view.

As seen in Figure 13, the Sessions page lists all collaboration sessions on the board. Each session displays the corresponding model's name, the session id, the start and end times, and a list of participants with their role categories. Sessions and Inventory pages are read-only and none of the CRUD operations are applied.

Models, Languages, and Users pages have a similar appearance and functionality. These classes' instances are shown in a tabular format. There is a + New button in the top right corner that can be used to add a new instance to the associated class. As seen in Figure 14, an overlay with a dark grey translucent background will appear on top of the board. In the middle of the overlay is a form with text or selectable inputs. Users have the option of clicking cancel to close it or save to store the data after filling out the form. Clicking on the overlay's black area will also exit the form. To edit a specific instance, the user can click the edit symbol at the end of each row. An identical overlay will be launched with the initial values already loaded. The user may just press the trash can icon besides the edit to delete an instance.

There are instances of four classes on the Plugins page: Drivers, Action Providers, Edit Adapters, and Identity Resolvers (Figure 15). Therefore, the create button is relocated from the top of the board to the right side of each class's section. CRUD operations perform identically to those on other pages.

Any modification to ModelInventory.xmi on the server, the inventory viewer will update the user with the most recent content without requiring a page refresh. To log out and return to the login page, users can click the Logout button in the lower left corner.



Inventory Structure

Editor

Attributes

Name	Type
name	String
type	EditorType

Participant

Attributes

Name	Type
role	RoleType

References

Name	Reference
user	User

User

Attributes

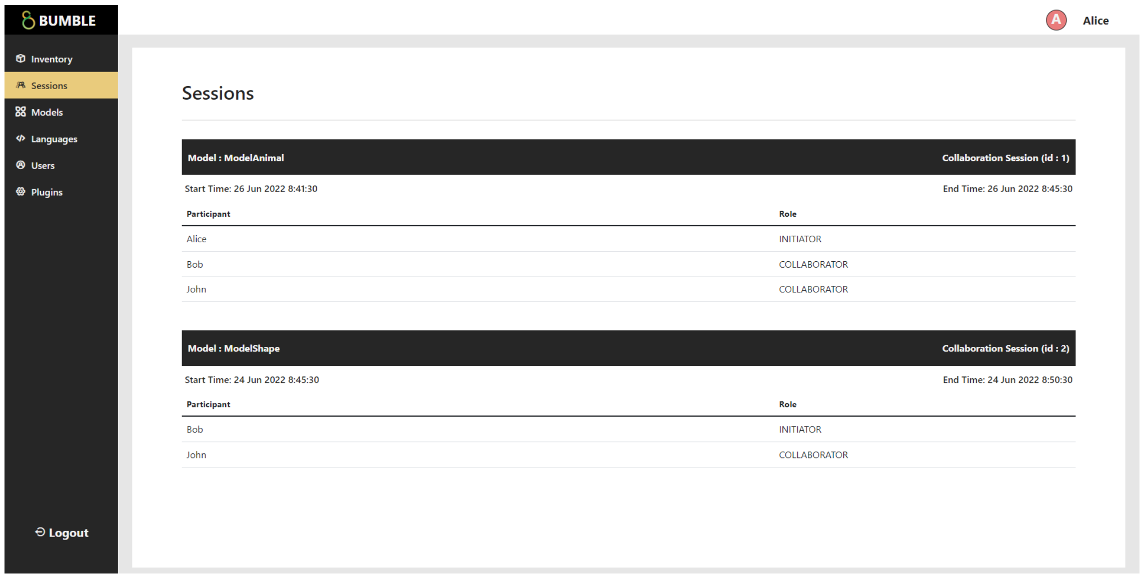
Name	Type
id	Int
name	String
emailAddress	String
password	String

Driver

Attributes

Name	Type
name	String
icon	ImageType

Figure 12: Inventory Structure Classes sub-page excludes Navigator and Header



Sessions

Model : ModelAnimal — **Collaboration Session (id : 1)**

Start Time: 26 Jun 2022 8:41:30 — End Time: 26 Jun 2022 8:45:30

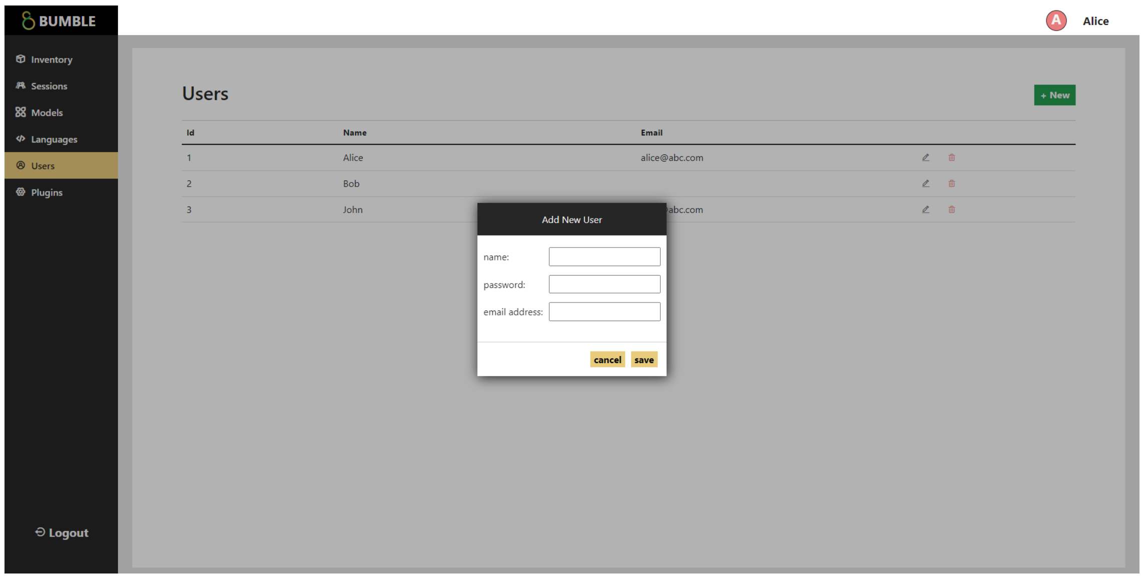
Participant	Role
Alice	INITIATOR
Bob	COLLABORATOR
John	COLLABORATOR

Model : ModelShape — **Collaboration Session (id : 2)**

Start Time: 24 Jun 2022 8:45:30 — End Time: 24 Jun 2022 8:50:30

Participant	Role
Bob	INITIATOR
John	COLLABORATOR

Figure 13: Sessions page



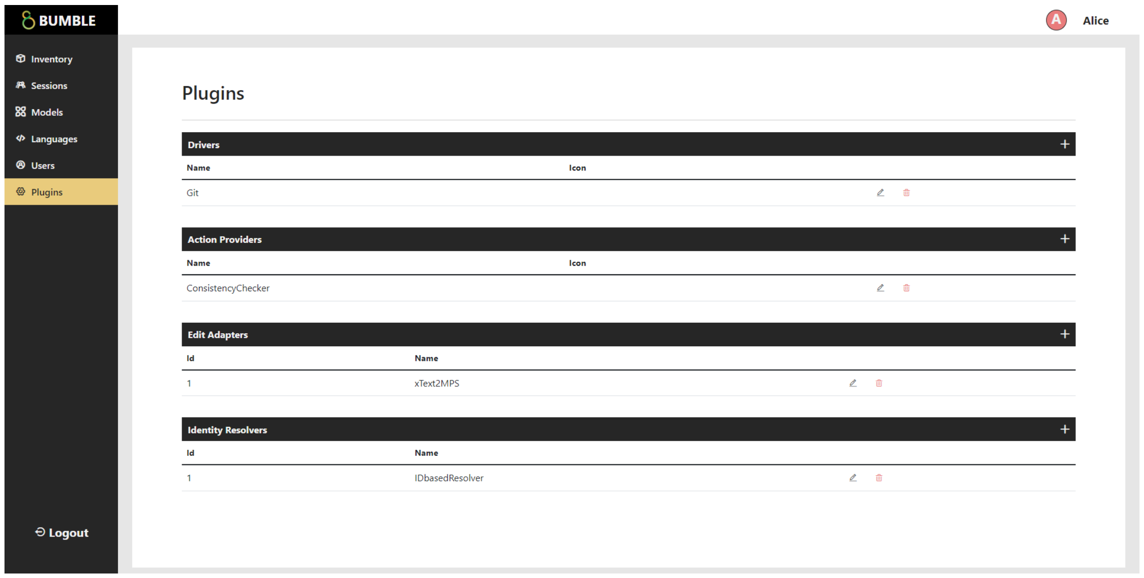
Users

Id	Name	Email
1	Alice	alice@abc.com
2	Bob	
3	John	abc.com

Add New User

name:	
password:	
email address:	

Figure 14: Create a new user



Plugins

Drivers

Name	Icon
Git	

Action Providers

Name	Icon
ConsistencyChecker	

Edit Adapters

Id	Name
1	xText2MPS

Identity Resolvers

Id	Name
1	IDbasedResolver

Figure 15: Plugins page

6 Related Work

Muyumba et al. [14] created a web-based inventory management system based on cloud computing and barcode technologies. The purpose of the project is to switch from a manual and paper-based inventory management system to an automatic one. Unlike the BUMBLE inventory viewer, the inventory system in this work is utilized to store aircraft spare parts. The aircraft spares inventory uses manually scanned barcodes for real-time stock tracking, whereas the BUMBLE inventory viewer uses WebSocket for real-time inventory updates. One thing that can be learned from this article is that they encrypt user passwords using the MD5[12] algorithm, which can also be used to ensure the user and data privacy in the BUMBLE inventory viewer.

Karim et al. [11] built a web-based inventory system for managing lab facilities. The server is developed in PHP, and MySQL is used as the back-end database. The webpage is written in HTML, CSS and JavaScript. The project's front-end technology is rather out of date compare to current popular frameworks. However, the approaches used to assist with design considerations, such as separating the requirements into Scope and Need, can be employed to enhance the BUMBLE inventory viewer's requirements.

The web-based inventory information system developed by J.S Pasaribu [15] is a fairly recent work (2021). The project follows a waterfall life cycle and the style of its design closely resembles that of the BUMBLE inventory viewer. The use of context diagrams for describing the system's scope and data flow diagrams for depicting the system's processes by the authors is an intriguing contribution and may be used to describe the system of BUMBLE inventory viewer.

[12]https://nl.wikipedia.org/wiki/MD5

7 Conclusion

The purpose of the project is to create a web-based front-end inventory viewer for the BUMBLE real-time collaborative modeling engine. It is based on the Vue.js framework and utilizes several official libraries to facilitate the task, including Vuex and Router. There are two ways for the inventory viewer and inventory server to communicate, namely the REST API and the WebSocket. Data retrieval and CRUD activities are performed via REST API. WebSockets are used for subscribing to real-time server updates. To reflect on the effort, we discuss below the project's strengths and limitations.

7.1 Strengths and Limitations

The developed web-based inventory viewer satisfies the five requirements outlined in section 3.3, as well as the login process and home page layout described in the user interface section. The Home page uses the layout that is currently the most prevalent for web applications: a navigation bar on the left, a header, and the main content board. Because the inventory viewer is not a creative design-driven project, but rather a requirement-driven one, the inventory viewer is more accessible due to its familiar style and straightforward iconography, which decrease the learning curve for users.

Special attention has been paid to improve the code's readability and maintainability. The most of colors used in the style, for instance, are defined as variables in a separate scss file. This makes it simpler to check up the color used in each component and to change the color in the future. In addition, the icons utilized in this project have been substituted with specially designed fonts. This significantly minimizes the amount of pictures and speeds up the loading process.

There are several constraints on the project. First, the system for error prevention is not fully implemented yet. For instance, there is no way to prevent accidental deletion of an instance or inadvertent pressing of the Logout button. Second, the user login verification process is not secure. Users' sensitive information, such as their password, should not be disclosed to the developer or other users. Third, when establishing a new instance or updating an existing one, there are no authentications applied to the user input.

7.2 Future work

The project is work in progress. In the future, we need to work on reducing the above-mentioned limitations. It is also worthwhile to add more user-experience-enhancing functionalities.

- **Error prevention**: To provide an error-prevention mechanism, a confirmation window can be added.
- **Login verification**: The user's password must be encrypted and stored on the server separately from the user's information in *ModelInventory.xmi*, or authentication tokens can be introduced to validate user input. Either login technique necessitates the server to provide a new API.
- **CRUD Authentication**: It is necessary to add input checks for instance creates and editions. Such as data type and format validations.

As stated in section 3.4.2, the Header, which resides at the top of the primary layout, can also serve the following two purposes:

- Display notifications about server updates.
- Allow users to chose between automated updates and manual page refresh.

For the first function, this project's inventory viewer does not indicate to the user if a CRUD operation was successful or not. The user must determine whether the procedure was successful by refreshing the web page to see if the changes were reverted. The notice that translates the server's response might be placed in the Header's center. There are two justifications for placing information in the Header. First, this location is highly visible and the user is able to instantly identify textual changes. Secondly, unlike a pop-up notice window, the information displayed on the Header does not need the user to take additional steps to dismiss it.

A subset of users may benefit from the ability to view the most recent server-side updates in real-time. However, since the real-time updates are achieved by retrieving the whole inventory from the server each time a little change is made. As a result, the data consumed is rather large and wasted. Until a more effective method for updating the inventory is developed, the user should have the option between automatic updating and manual refreshing.

References

[1] Accessibility. `https://www.w3.org/standards/webdesign/accessibility`. Accessed: 2022-6-27.

[2] Emf.cloud - evolve your modeling tools to the web! `https://www.eclipse.org/emfcloud/`. Accessed: 2022-6-27.

[3] The websocket api. `https://www.w3.org/TR/2021/NOTE-websockets-20210128/Overview.html`. Accessed: 2022-6-27.

[4] What is vue? `https://vuejs.org/guide/introduction.html`. Accessed: 2022-6-27.

[5] What is vuex? `https://vuex.vuejs.org/#what-is-a-state-management-pattern`. Accessed: 2022-6-27.

[6] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering*. Morgan and Claypool, 2012.

[7] F. Doglio, Doglio, and Corrigan. *REST API Development with Node. js*, volume 331. Springer, 2018.

[8] I. Fette and A. Melnikov. The websocket protocol. Technical report, 2011.

[9] P. Halliday. *Vue. js 2 Design Patterns and Best Practices: Build enterprise-ready, modular Vue. js applications with Vuex and Nuxt*. Packt Publishing Ltd, 2018.

[10] J. Helming, M. Koegel, and P. Langer. The emf.cloud model server. `https://eclipsesource.com/blogs/2021/02/25/the-emf-cloud-model-server/`. Accessed: 2022-6-27.

[11] A. M. Karim, M. F. Saad, and M. Haque. Development of a prospective web-based inventory system for management of lab facilities. *Journal of Emerging Trends in Engineering and Applied Sciences*, 2(1):36–42, 2011.

[12] I. Malavolta, K. Aslam, L. Berardinelli, W. Bast, and B. Theelen. Bumble deliverable d5.1. ITEA 3, 2021.

[13] R. Molich and J. Nielsen. Improving a human-computer dialogue. *Communications of the ACM*, 33(3):338–348, 1990.

[14] T. Muyumba and J. Phiri. A web based inventory control system using cloud architecture and barcode technology for zambia air force. *International Journal of Advanced Computer Science and Applications*, 8(11), 2017.

[15] J. S. Pasaribu. Development of a web based inventory information system. *International Journal of Engineering, Science and Information Technology*, 1(2):24–31, 2021.

[16] V. Pimentel and B. G. Nickerson. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4):45–53, 2012.

[17] K. Riemer and F. Frößler. Introducing real-time collaboration systems: development of a conceptual scheme and research directions. *Communications of the Association for Information Systems*, 20(1):17, 2007.

[18] E. Saks. Javascript frameworks: Angular vs react vs vue. 2019.

[19] E. Wohlgethan. *Supportingweb development decisions by comparing three major javascript frameworks: Angular, react and vue. js.* PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018.