EXAMINATION: Web Systems and Technologies June 2024

Nest: Name:

Permutation A

☞ GJeans

After uploading the code to moixero.uab.cat fill in the following box.

Electronic Delivery

The SHA1 checksum of the received file is:

.....

Time stamp is:

.....

Keep a copy of the file you deliver.

Fix Guide:

The grade will be...	When...	Scoring Guide
0 to 5 points	When the whole exercise doesn't work and there are *serious* concept errors in any of the following key elements: callbacks, closures, classes, inheritance, promises, and the various elements of vue (reactivity, directives, events , props, components, ...).	Start from 5 and subtract 1 point for each concept error.
5 to 8 marks	When the exercise does not work and there are no serious concept errors.	Start from 8 and subtract 0.25 points for each error.
From 8 to 10 points	The exam passes all the tests.	You got a 10. You deserve it.

Instructions

Follow the instructions below to boot the lab machine and import the project skeleton.

- Start the machine if you haven't started it already and select the Linux partition (user:exam and passwd:exam).
- Open a console: Applications → Terminal.
- Download the project skeleton by running the command:

  `wget https://moixero.uab.cat/ExamenSTW.7z`

- Unzip the file.

  `7z x ExamSTW.7z`

- You have the skeleton in the directory **stw**. Take the exam (you can use the same terminal you already have opened, and open the **stw** directory with Visual Studio Code).
- To run the application:
  1. the client: **`npm run dev`**
  2. the server: Go to the **`src`** directory and run **`node exam.js`** or **`nodemon exam.js`**.
- Note: If you are using Chrome, you will see that it opens in incognito mode and the Vue addon is not available. Open a new Chrome (Ctrl+N). The new window is no longer incognito and the Vue addon can be used.
- Once you have finished developing the project, you must submit your code electronically. **If the entire application works for you, please let us know before submitting electronically.**
- To deliver electronically, create a zip as follows:
  Move to the **`src`** directory

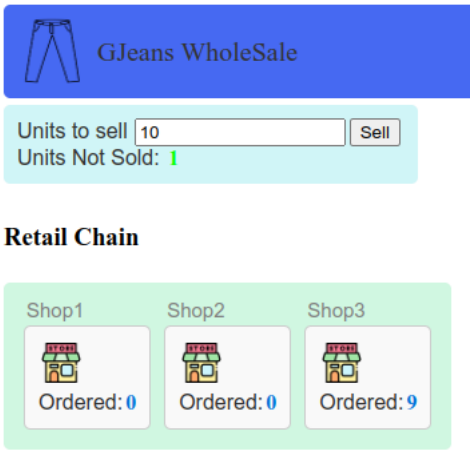
  ```
  7z a sol.zip exam.js App.vue components/SellForm.vue components/Shop.vue
  ```

- Check the contents of the file you will deliver (open the file and look at the contents of the files inside).
- Once you know for sure that you want to deliver this file, upload it to: `https://moixero.uab.cat/`.
- Write down the two values provided by the server (the checksum and the timestamp) on the exam and give it to the supervisor.
- **When you're done, don't log out and don't stop the machine!!!!!!!**

Context

You need to develop the front and back-end of a portal to wholesale jeans. Through this portal it will be possible to offer a stock of jeans (Figure 1 (a)). The back-end will receive the stock of jeans and offer them to different stores who will buy random units. When all the stores have decided the number of jeans they will buy, the final purchase distribution between the shops will be displayed in the front-end (Figure 1 (b)).

(a) WholeSale Portal Main Page



(b) Result of bidding ten jeans. Shop1 and Shop2 have not bought any jeans. Shop3 bought 9. We have a stock of one unsold jean left.

Figura 1: Wholesale portal.

You can start this portal by running:

1. the client: **`npm run dev`**
2. the server: Go to the **`src`** directory and run **`node exam.js`** or **`nodemon exam.js`**.

You will see the view in figure 2.

Figura 2: Skeleton view.

Frontend Exercise

You need to complete the **`RootComponent`** (**`App.vue`**) and the **`SellForm`** and **`Shop`** components.
Below are the component specifications.

RootComponent (App.vue)

This component includes the **`SellForm`** component from whom you get the shops purchase distribution. From this distribution use the **`Shop`** component to display the units sold in each shop.

Specifications

- Centralize the distribution of purchases in the reactive variable **`orders`**.
- Purchases (**`orders`**) are represented by a JSON object like the following:

```
{remainingStock:1, details:[{id:"Shop1",ordered:9},{id:"Shop2",ordered:0}, {id:"Shop3",ordered:0}]}
```

  Where the options:

  remainingStock: Indicates units that have not been sold.

  details: It is an array of dictionaries. Each dictionary is associated with a store. For each store we show its ID and the number of units it has purchased. Note that a store may not buy any unit.

- Purchases (**`orders`**) are updated by the **`SellForm`** component. We will use this component as follows:

```
<SellForm v-model="orders" />
```

- Iterate over the list of **`orders.details`** and use the **`Shop`** component to display the purchase information for each shop in that list.

Component: SellForm

This component includes an input to specify the units to sell and a button to execute the sale (Figure 3 (a)). When executing the sale, until the server returns the final distribution of the sales between the different stores, we display a message (Figure 3 (b)). When the server returns the distribution of the purchase we show the number of units that we have not sold (Figures 3 (c) and (d)) or no message if we have sold everything.

(a) Units to sell input

(b) Sale transaction in progress

(c) Ten products remain unsold

(d) One product remains unsold

Figura 3: Views of the **`SellForm`** component.

Specifications

- Receive, through the **`v-model`** directive, the dictionary (**`orders`**) to update it with the final purchases. Remember that the following two tags are equivalent:

```
<Component v-model="data" />
<Component :modelValue="data" @update:modelValue="x => data = x" />
```

- Clicking the **`sell`** button makes a request to the server to execute the sale of the units specified in the input:

```
fetch(`http://localhost:3001/wholesale/10`).then(res => res.json()).then(json => {...})
```

  Note that **`/wholesale`** is a server entry point and **`/10`** is the number of units of jeans to sell.

- **With the response we will receive from the server, we will update the `orders` of the `RootComponent`.**
- Once the **`sell`** button is clicked and while waiting for the server's response, it displays the message *"Sale in progress..."* (Figure 3 (b)). When we receive the response from the server we do not render this message.
- If not all units have been sold, we display the message: *"Units Not Sold"* indicating the units that have not been sold with the following color code: **`red`** if nothing has been sold, **`orange`** if less than 50% has been sold, and **`green`** if more than 50% has been sold. You might adapt the following *snipped* of code to determine the color:

```
notSold == unitsToSell ? 'red' : (notsold < (unitsToSell*0.5)) ? 'lime' : 'orange'
```

Component: Shop

This component displays the purchase details of a store specified by the dictionary:

```
{id:"Shop3",ordered:6}
```

where the field **`id`** corresponds to the ID of the store and **`ordered`** corresponds to the number of units that the store has purchased, as shown in the figure 4.

Figura 4: Component view **`Shop.vue`**

Specifications

– The component receives as a parameter the object with the purchase details of a shop: `{id:"Shop3",ordered:6}`.

FrontEnd Test

- We provide you with a dummy server that, after two seconds, always returns the following purchase distribution:

```
{remainingStock:1, details:[{id:"Shop1",ordered:8},{id:"Shop2",ordered:0}, {id:"Shop3",ordered:1}]}
```

Notes

– You cannot use `async/await`.

Backend exercise

The backend consists of a module that acts as a distributor of jeans between an array of stores that is already defined (*hardcoded*). Each shop is an object of type **`Shop`** that we give you already fully programmed. The distributor module (**`wholeSaleModule`**) will offer the units for sale that we receive from the client to the stores. Each store will buy a random number of units or none at all. Finally the module will encapsulate the purchase distribution of the stores in a dictionary that will be returned to the client.

Shop Class Specifications

Attention!! We give you this class already finished.

– The constructor takes as parameters the **`id`** of the store and a **`timeout`**. This timeout is the time the shop has to decide how many units to buy.

– Class methods:

toPlainObject returns an object that encapsulates the **`id`** of the store and the number of units the shop has decided to buy: `{id: 'shop3', ordered: 6}`

offer(units):promise Method that receives the units available for sale and returns a promise that, after a timeout, **resolves to the number of units the shop decided to buy** or **rejects to nothing (`reject()`) if the shop has decided not to buy anything**.

wholeSaleModule Module Specifications

– You will develop this module following the *module pattern*. Here is an example.

```
const testModule = (()=>{
  let _a = 1;
  const increase = () => { _a++; },
  const value = () => { return _a; }
  return {
    increase,
    value,
  };
})();
```

– Contains a static list with three stores:

```
let shops = [new Shop("Shop1", 1000), new Shop("Shop2", 700), new Shop("Shop3", 200)]
```

- **Function getDetails**: (**optional**) Private function that might be convenient. This function iterates through the list of stores and returns a list with the purchase distribution they made:

```
[{id:"Shop1",ordered:9},{id:"Shop2",ordered:0}, {id:"Shop3",ordered:0}]
```

  You will need to access the **`toPlainObject`** method of the **`Shop`** class.

  `getDetails` is used by the **`wholeSale`** function, so after the units are distributed to the stores, **`wholeSale`** uses this function to set the returning object's **`details`** field with the result of the purchase distribution of the different stores.

  Note that you must call this function once you know that all stores have already finished their orders.

- Functions exposed (returned) by the module:

  wholeSale(units) : promise Function that receives the number of units specified in the frontend and offers them to the stores. It returns a promise that resolves to the object that contains the number of unsold units and a list with the purchase details of each store:

```
{remainingStock:1, details:[{id:"Shop1",ordered:9},{id:"Shop2",ordered:0}, {id:"Shop3",ordered:0}] }
```

  Attention!!! We must offer to the first shop all the available units, to the next shop, all available units except those bought by the previous shop (remaining stock) and so on. Notice that the function `Shop::offer(units)` returns a promise that resolves to the number of units it buys or rejects if it doesn't buy anything.

  You can program this function in two different ways, which we detail below. Choose the one that suits you best:

  1. **Recursive Option:**
     - With this option we can have a variable number of shops in the list.
  2. **Iterative Option:**
     - You can directly deal with **`shops[0]`**, **`shops[1]`** and **`shops[2]`**.
     - You must use a promise chain. The following code is not a promise chain and will not be accepted:

```
p.then(x1=>{p2.then(x2=>{p3.then(x3=>...).catch()}).catch()}).catch()
```

     - Hint: Check on problem 23: *23. Do a p.then(x => console.log(x))*...
     - If you go for this option you must implement the following function:

       stockUpdater(units): updateStock :
       - It is a private function of the **`wholeSaleModule`** module, that is, we do not export it.
       - This function receives the number of units to sell (stock) from the frontend and returns a function

         `function(unitsSold): remainingStock`

         - You can name this returned function **`updateStock`** and as we have seen, it receives as a parameter the number of units a store buys, updates the stock (stock-=unitsSold) and returns the updated stock.
       - The **`stockUpdater`** function implements a **closure**!

Web Server Specifications

- The server will have an endpoint:

  `/wholesale/:units`, where **`units`** corresponds to the number of units to be distributed between the different stores. Upon receiving this request, the server executes the function **`wholeSale(units)`** and returns the result to the client.

Test BackEnd

- You can test the server directly from a browser window by typing the url:

```
http://localhost:3001/wholesale/10
```

- The browser should load a JSON like the following one:

```
{"remainingStock":1,
"details":[{"id":"Shop1","ordered":9},{"id":"Shop2","ordered":0},{"id":"Shop3","ordered": 0}]}
```

- Ensure that the number of initial units minus units sold is the value of **`remainingStock`**.
- You can refresh the browser to make a new request to the server.

Notes

- You cannot use **`async/await`**.
- **`Promise.resolve(0)`** returns a promise that immediately resolves to 0.
- **`Promise.reject(3)`** returns a promise that immediately rejects to 3.
- **`map`** function usecase:

```
const array1 = [1, 4, 9, 16];

// Pass a function to map
const map1 = array1.map((x) => x * 2);

console.log(map1);
// Expected output: Array [2, 8, 18, 32]
```

- **`forEach`** function usecase:

```
const array1 = ['a', 'b', 'c'];

array1.forEach((item) => console.log(item));

// Expected output: "a"
// Expected output: "b"
// Expected output: "c"
```

This is a draft page. Don't unstaple it!

This is a draft page. Don't unstaple it!