INVENTORY REQUEST MOVEMENT DOCUMENTATION

Supervision under:
En Arif

Prepared by:

Ahmad Ameerul Rasyid bin Hamidun (Rasyidies IT)
Muhammad Arif Hilmi bin Rezuan (Hilmies IT)

This software documentation consists of only the newest patch Item Movement and Item Request module.

This documentation is a technical documentation in the aim to help other programmers to understand the interface and coding structure of the newest patch in the system development.

This documentation is prepared by individuals that work directly on the newest patch, turning this documentation is almost very technical.

Graphical representation in this documentation aims to support a quick understanding of the topic.

Many of the coding representations are attached as images, and are advised to only refer as reference.

This documentation should not be shared to other organizations or other foreign individuals. Please keep this document confidential.

Table Of Content

1. System diagram

1.1 Process flow Diagram

Each process flow diagram will be attached specifically for its process in the upcoming topics.

1.2 Data flow diagram

The system consists of Item Movement and Item Request Module. Hence these modules, there is no direct relation between these modules. The data flow diagram of the system is shown in Diagram 1.1.

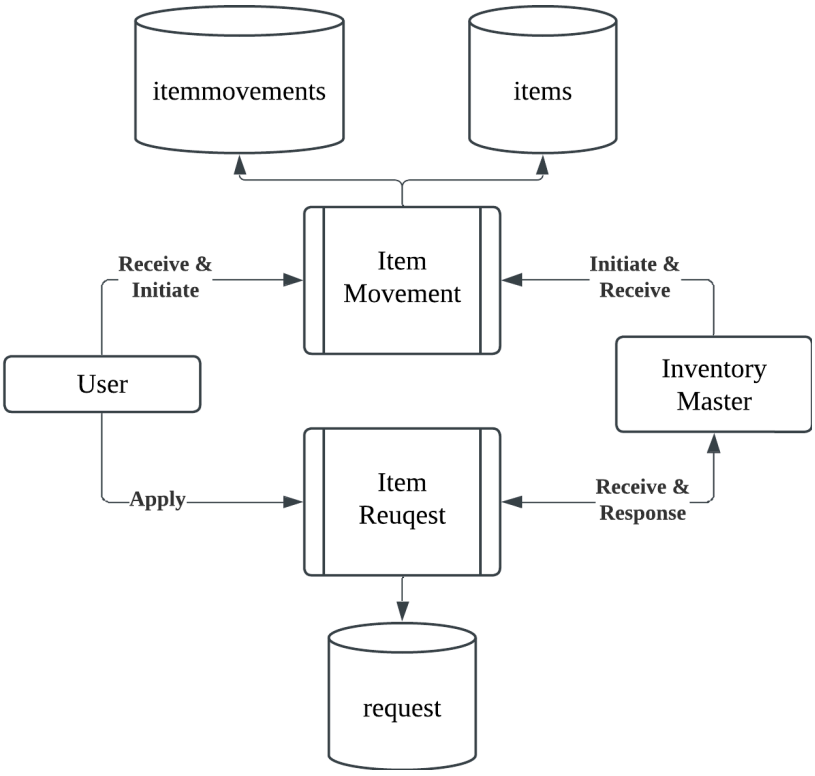


Diagram 1.1 Data Flow Diagram

2. User Stories

User stories aim to provide a context of the user's expectations of the system. The system should be able to address users' needs reasonably.

2.1 Inventory Master

Inventory Master refers to inventory masters. The person with this role has their own assigned store to be referred throughout their interaction in the system.

2.1.1 Item Request

User Story	As an Inventory Master, I want to approve and reject item requests by users.
Acceptance Criteria	1. The Item request page must be easily accessible 2. The approve/reject process should be easily carried 3. The interface should reflects to each approve and reject process accordingly

2.1.2 Item Movement

User Story	As an Inventory Master, I want to send, cancel, item requests by users. I also want to view all historical data of item movement of items in my possession. In some cases, I should be able to receive items.
Acceptance Criteria	1. Items can be sent to others by scanning the item's qr code. 2. Items can be received by scanning the item's qr code. 3. The Item movement page must be usable and easily accessible 4. The Item movements can be viewed as per all, items and stations.

2.2 Admin

Admin is the admin of the system. The person with this role should be able to navigate freely in the controlled system to provide power to carry tasks as system admin.

2.2.1 Item Request

User Story	As an Admin, I want to approve and reject item requests by users.
Acceptance Criteria	1. The Item request page must be easily accessible 2. The approve/reject process should be easily carried 3. The interface should reflects to each approve and reject process accordingly

2.2.2 Item Movement

User Story	As an Inventory Master, I want to send, cancel, item requests by users. I also want to view all historical data of item movement of items in my possession. In some cases, I should be able to receive items.
Acceptance Criteria	1. Items can be sent to others by scanning the item's qr code. 2. Items can be received by scanning the item's qr code. 3. The Item movement page must be usable and easily accessible 4. The Item movements can be viewed as per all, items and stations and logs.

2.3 Technician

Technician is a user for the system. Technicians may have been assigned to a station. This information will be referred throughout their interaction in the system.

2.3.1 Item Request

User Story	As a technician, I want to send item requests and wait for approval from the Inventory Master..
Acceptance Criteria	1. The Item request page must be easily accessible 2. The request process should be easily carried 3. The interface should reflects to each incomplete and complete process accordingly

2.3.1 Item Movement

User Story	As a technician, I want to receive & return items from the inventory master. Also assign items to stations.
Acceptance Criteria	1. Items can be received by scanning the item's qr code. 2. Items can be returned by scanning the item's qr code. 3. The Item movement page must be usable and easily accessible 4. The Item movements can be viewed as per all, items and stations and logs. 5. The interface should reflects to each receive item, return item and assign station process accordingly

3. Database

This section will elaborate on both Item Request and Item Movement databases. Each process will have its own database design and database flow. Each process may also have different scenarios in their database flow. This document will go through Item Request including its database design and database flows of different scenarios then continued with Item movement.

3.1 Item Request

Technicians can initiate product requests to Inventory Masters and assign it to an assigned station or self assigned. All Inventory Masters can approve or reject the request product made by Technicians. Technicians have to wait for their request to be accepted or rejected. The completed request will be displayed in the interface.

Meanwhile, Inventory masters too can initiate product requests from another Inventory Masters based on their Store. Only the requested Inventory Master can approve or reject the product request made by other Inventory Master. Inventory Masters will not be able to approve or reject their own requests. Inventory Masters have to wait for their request to be accepted or rejected. The completed request will be displayed in the interface separately.

3.1.1 Database Design

Table 3.1 shows the attributes and its description in the *request* table.

Table 3.1 Attributes and Descriptions in *request* Table

requestID	Request Unique ID. This will be the Primary key of the data
ProductId	Product Id. This is the foreign key **ProductId** in the table *products* table of the requested product.
StationId	Station Id. This is the foreign key **StationId** in the table *stations* of the assigned station of the Technician. This can be NULL if not applicable.
UserId	User Id. This is the foreign key **Id** in the table *aspnetusers* of the request initiator. This cannot be NULL.
ProductCategory	String. This will autofill the value based on the selected **ProductId** table *products*. Data will be either :- 1. ‘Asset’ 2. ‘Part’ 3. ‘Disposable’
RequestQuantity	Integer. This column stores the quantity the user requests for the product.

Document	String. This column stores the path of the picture. The actual uploaded file is stored in the server's folder.
remarkUser	String. This column stores general notes of requested User
remarkMasterInv	String. This column stores general notes of approval or disapproval Inventory Master
status	String. This column stores the status of the request process. Data is initially be Requested. Data may be either :- 1. 'Requested' (When User send request to Inventory Master) 2. 'Approved' (When Inventory Master approve the request) 3. 'Rejected'(When Inventory Master reject the request)
requestDate	DateTime. This column stores the request date of the product.
approvalDate	DateTime. This column stores the approval date of the request.
fromStoreItem	StoreId. This is the foreign key **Id** in the table *stores* of the requested store which is in the possession of the product. Only requests by Inventory Masters will make use of this column. This can be NULL if not applicable.
assignStoreItem	This is the foreign key **Id** in the table *stores* of the requesting Inventory Master's assigned store. Only requests by Inventory Masters will make use of this column. This can be NULL if not applicable.

3.1.2 Database Flow

Inventory Masters or Technicians can request product, the differences between these process are :-

1. Technicians request items from the Inventory Master as self assigned or station assigned.
2. Inventory Masters request products from **other** Inventory Master's stores to store the items in **their** store.

3.1.2.1 Request Technicians

Technicians initiate product requests to Inventory Masters. Diagram 3.1 shows the overall process flow and will be elaborated further in the following sections.

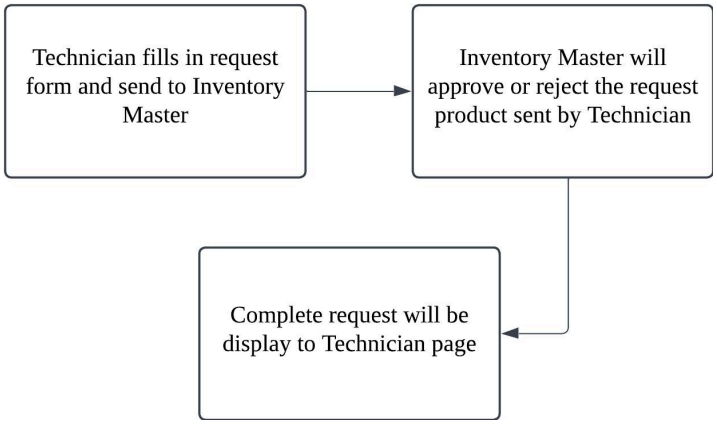


Diagram 3.1 Request Product Technicians to Inventory Master Process Flow

Technicians will fill in the request form and send it to the Inventory Master. Then, Technicians have to wait until the request gets approval from the Inventory Master. This request will create a new data row in the *request* table. Diagram 3.2 shows the initial data saved into the request table.

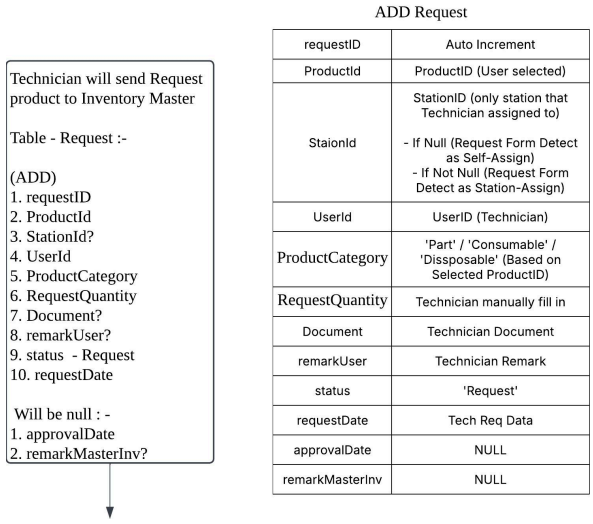


Diagram 3.2 Technician send request product form to Inventory Master dataflow

Upon the request being sent, the Inventory Master will give approval to the request, either Approved or Rejected. This will update the *request* data row. Diagram 3.3 shows this behaviour.

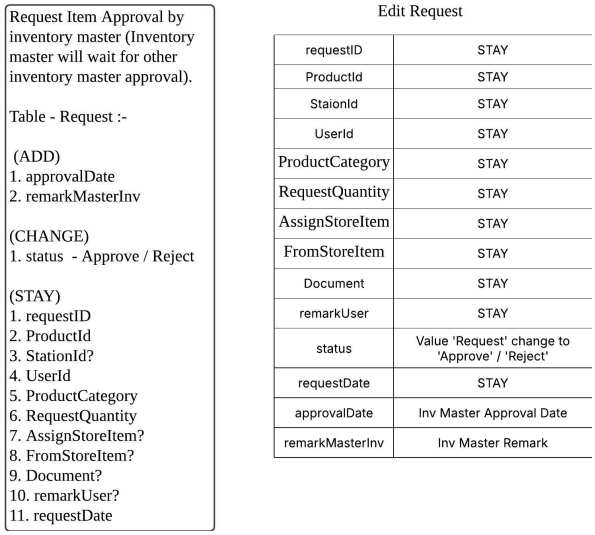


Diagram 3.3 Inventory Master request approval dataflow

Once the Inventory Master approved or rejected the product request sent by Technician, it will be reflected in the Technician page for their reference.

3.1.2.2 Request Inventory Master

Inventory Master requests products to the other Inventory Master. Diagram 3.4 shows the overall process flow and will be elaborated further in the following sections.

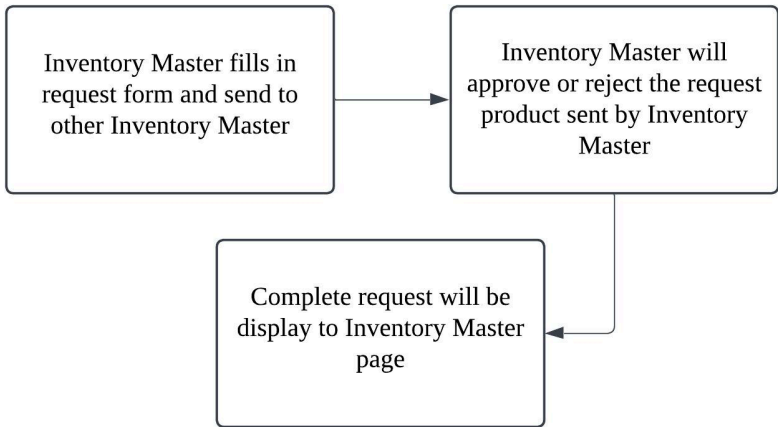


Diagram 3.4 Request Product Inventory Master to another Inventory Master Process Flow

The Inventory Master will fill in the request form and send it to another Inventory Master. Then, the Inventory Master has to wait until the request gets approval by the other Inventory Master. This request will create a new data row in table *request*. Diagram 3.5 shows the initial data saved into the *request* table.

ADD Request

requestID	Auto Increment
ProductId	ProductID (User selected)
StaionId	NULL
UserId	UserID (Technician)
ProductCategory	'Part' / 'Consumable' / 'Dissposable' (Based on Selected ProductID)
RequestQuantity	Technician manually fill in
assignStoreItem	Assign StoreId
fromStoreItem	From StoreId
Document	Technician Document
remarkUser	Technician Remark
status	'Request'
requestDate	Tech Req Data
approvalDate	NULL
remarkMasterInv	NULL

Diagram 3.5 Inventory Master send request product form to another Inventory Master dataflow

Upon the request being sent, the other Inventory Master will give approval either Approved or Rejected. This will update the *request* data row. Diagram 3.6 shows this behaviour.

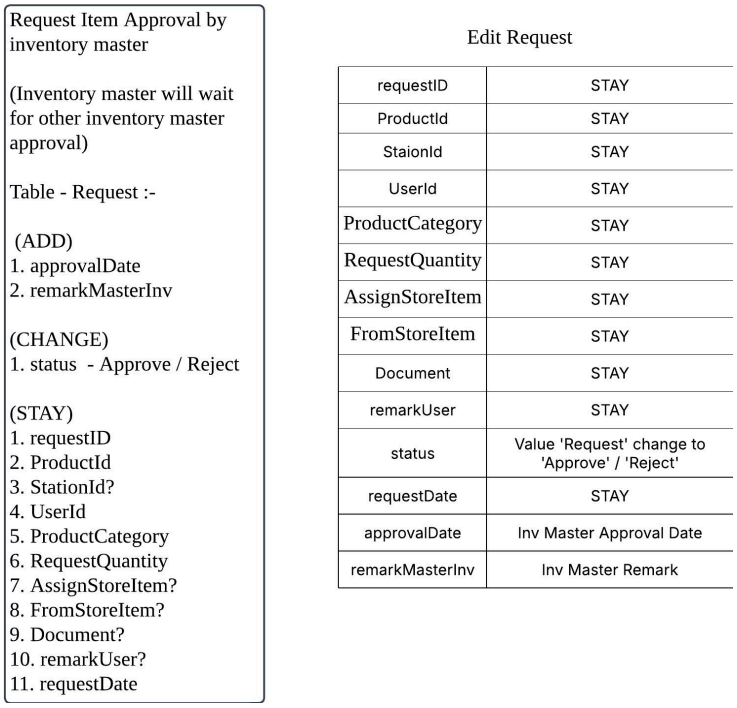


Diagram 3.6 Inventory Master will give approval dataflow

Once the other Inventory Master approved or rejected the product request, It will reflect to the requesting Inventory Master’s request page.

3.2 Item Movement

Inventory Master will send items to another user. Inventory Master will have to scan the item and fill in where the item is delivering to. Inventory Master can use the tab configuration to send to either user, station, store, supplier or mark as faulty. Other entities like Technicians or other Inventory Masters will receive the item which happens when the receiver scans the item to mark it as delivered in the system. Technicians can next return an item by scanning the item again. Returned item will be marked as Faulty and can no longer be sent to other users. Items will first need to be sent to suppliers for repair or calibration before being able to be sent again to other users.

3.2.1 Database Design

Table 3.2 shows the attributes and its description in the *itemmovements* table.

Table 3.2 *itemmovements* Table

Id	Movement Unique ID. This will be the Primary key of the data
ItemId	Item Id. This is the ItemId of the movement
ToStation	Station Id. This is the foreign key **StationId** in the table *stations* of the item sender or movement initiator. This can be NULL if not applicable.
ToStore	Store Id. This is the foreign key **Id** in the table *stores* of the item sender or movement initiator. This can be NULL if not applicable.
ToUser	User Id. This is the foreign key **Id** in the table *aspnetusers* of the sender or movement initiator. This cannot be NULL.
ToOther	String. This column stores short notes of the movement. Data will be null upon 'Register' in column *Action*. Data may be either :- 1. 'Repair' (When item to be sent to supplier for repair) 2. 'Calibration' (When item to be sent to supplier for calibration) 3. 'Faulty'(When item to be mark as faulty) 4. 'Ready To Deploy' (When item is ready to be deployed to others) 5. 'On Delivery' (When item to be sent to user) 6. 'Return' (When item to be sent back to Inv Master)
SendDate	DateTime. This column stores the assigned date upon initiating movement. Users are allowed to backdate as per requirement of the project.
Action	String. This column stores general notes of the movement. Data may be either :- 1. 'Register' (When item is registered for the first time) 2. 'Stock In' (When item is in possession of Inv Master) 3. 'Stock Out' (When item to be sent out from Inv Master possession) 4. 'Assign' (When item is assigned to different station by Technicians)

Quantity	Number. This column stores the quantity of the item which is referred to the *products* table of the item.
Remark	String. This column stores remarks of the sender or movement initiator.
ConsignmentNote	String. This column stores the path to the document attached by the sender or movement initiator.
Date	DateTime. This column stores the registration date of the item.
LastUser	User Id. This is the foreign key **Id** in the table *aspnetusers* of the item receiver. This cannot be NULL.
LastStore	Store Id. This is the foreign key **Id** in the table *stores* of the item receiver. This can be NULL if not applicable.
LastStation	Station Id. This is the foreign key **StationId** in the table *stations* of the receiver. This can be NULL if not applicable.
LatestStatus	String. This column stores short notes of the movement. Data will be NULL upon submitting. Data may be either :- 1. 'Faulty' (When item returned by Technicians) 2. 'Ready To Deploy' (When item received by Inv Master after sent to supplier for repair or calibration) 2. 'Delivered' (When item is received by User)
receiveDate	DateTimeNow. This column stores the exact date upon receiving the item.
MovementComplete	Boolean. This column stores the status of the movement. Data will be updated upon receiving. Data may be either :- 1. '1' (When item movement is completed) 2. '0' (When item movement is incompleted)

3.2.2 Database Flow

Inventory Master may initiate item movement to Technicians by user, Technicians by station, Stores, Supplier or mark as Faulty.

3.2.2.1 Item to Users

Inventory Master may initiate item movement to user, which can be either Technicians by user, Technician by station and other Stores. Diagram 3.7 shows the overall process flow and will be elaborated further in the following sections.

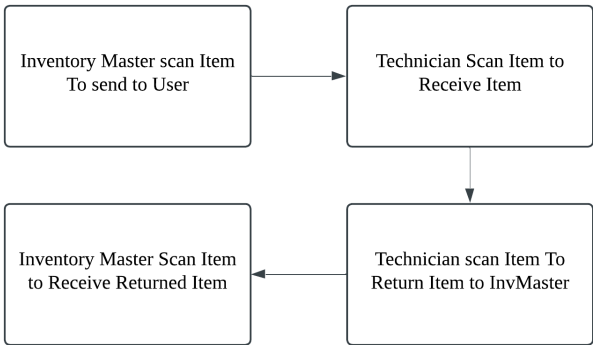


Diagram 3.7 Item Movement to Users Process Flow

Inventory Master will first scan the item to be sent to the user. This will create a new data row in table *itemmovements* and updates the item's *MovementId* in the *items* table. Diagram 3.8 shows an example of the initial data saved into the items table when an item is being sent to Technician by user.

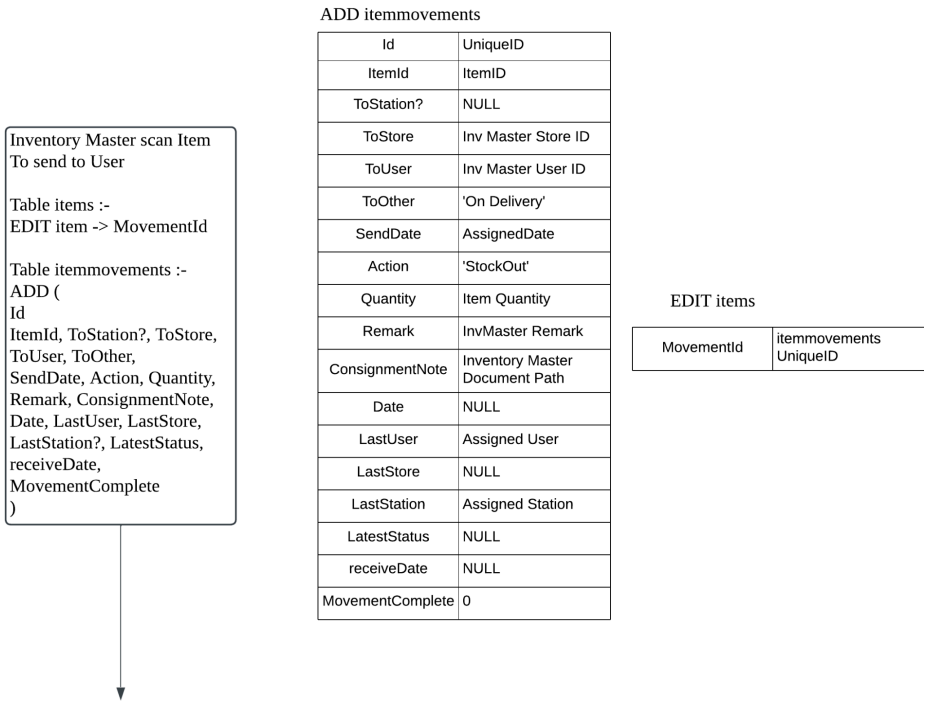


Diagram 3.8 Inventory Master scan item to send to user dataflow

Upon receiving, the user will scan the item to mark it as delivered. This will update the *itemmovements* data row. Diagram 3.9 shows this behaviour.

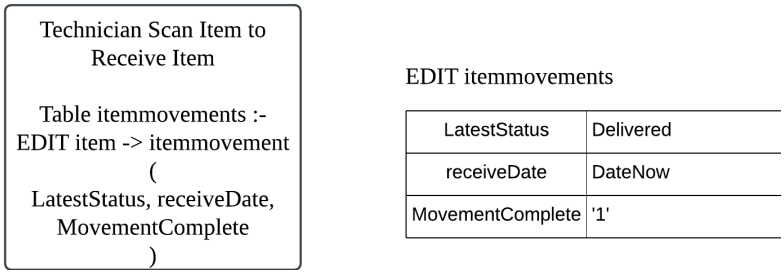


Diagram 3.9 Technician scan item to receive item dataflow

When a Technician decides to return the item to Inventory Master, they can initiate item return by scanning the item using the user's qr scanner. Diagram 3.10 shows the dataflow of this process.

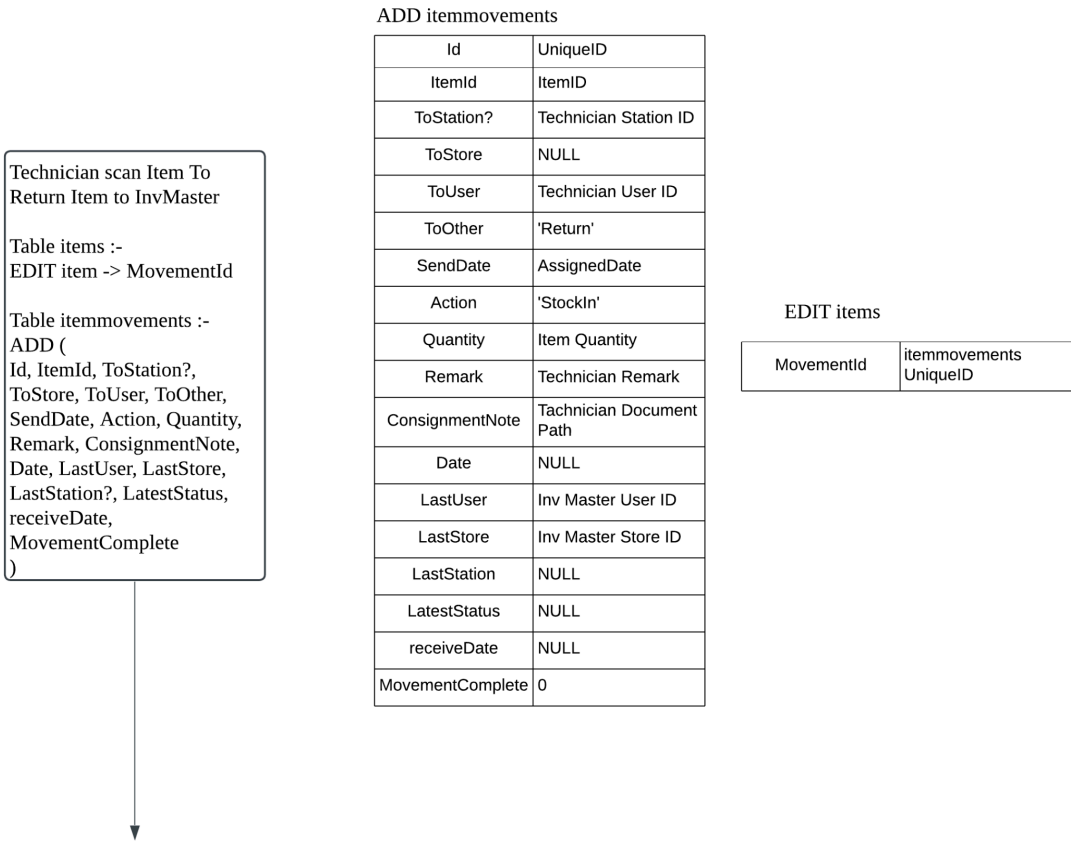


Diagram 3.10 Technician scan item to return item to Inventory Master dataflow

The Inventory Master will have to scan the item again to mark it as received. Diagram 3.11 shows the dataflow of this process.

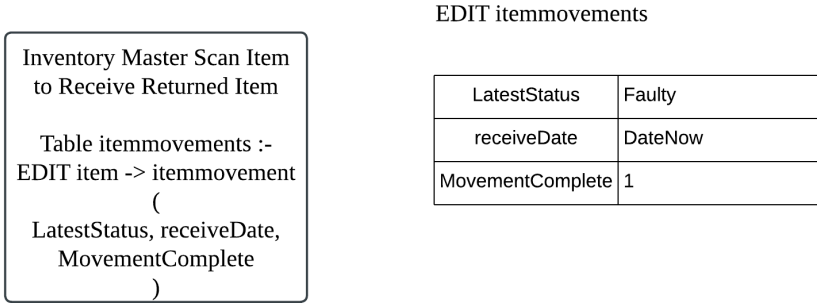


Diagram 3.11 Inventory Master scan item to receive dataflow

3.2.2.2 Item to Supplier

Inventory master may initiate item movement to a Supplier for Repair or Calibration purposes. Diagram 3.12 shows the overall process flow and will be elaborated further in the following sections.

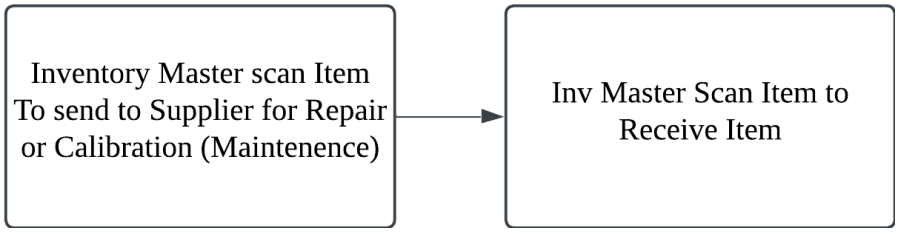


Diagram 3.12 Item Movement to Supplier Process Flow

Inventory Master will first scan the item to be sent to a Supplier. This will create a new data row table *itemmovements* and update item's *MovementId* in *items* table. Diagram 3.13 shows an example item sent to a supplier for Repair, hence it is the initial data saved into *items* table.

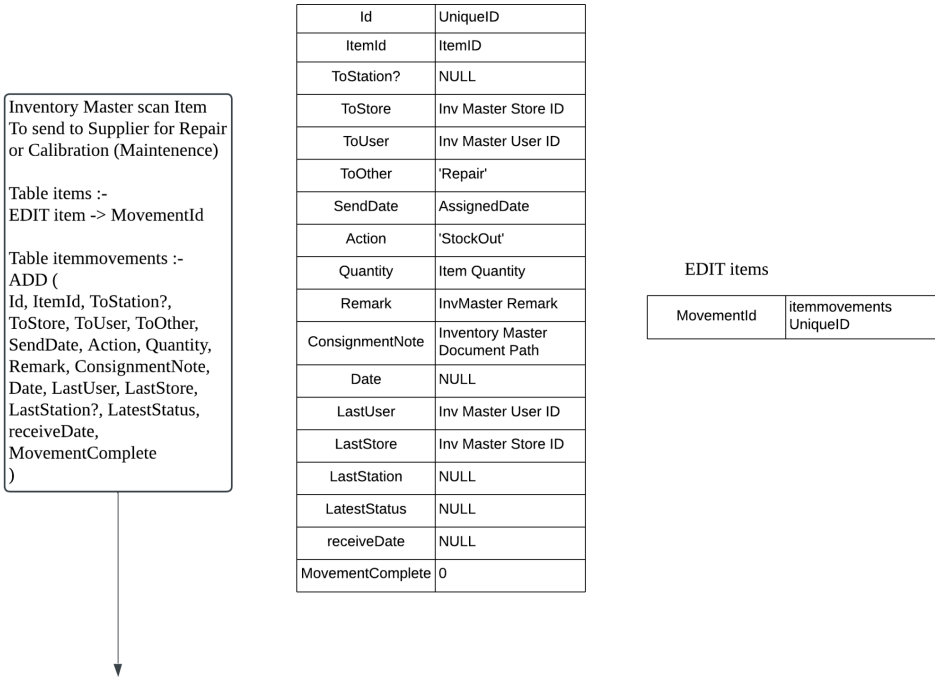


Diagram 3.13 Inventory Master scan item to send to Supplier for Repair dataflow

Upon receiving back from the Supplier, Inventory Master will scan the item again to mark it as Ready To Deploy and the item is ready to be deployed again. Diagram 3.14 shows this behaviour.

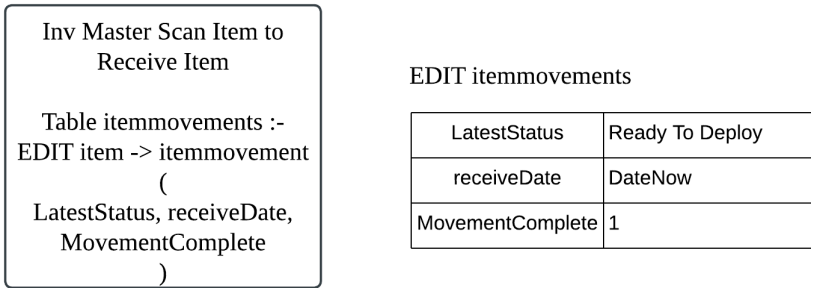


Diagram 3.14 Inventory Master scan item to receive item from supplier dataflow

3.2.2.3 Item mark Faulty

Inventory Master may mark an item as Faulty. By doing this, the item will be marked as faulty and can only be sent to Suppliers before it can start to be sent to Technicians again. This process only requires a process, hence no diagram needed to explain the process. *MovementComplete* will automatically set as **TRUE**.Diagram 3.15 shows the dataflow of this process.

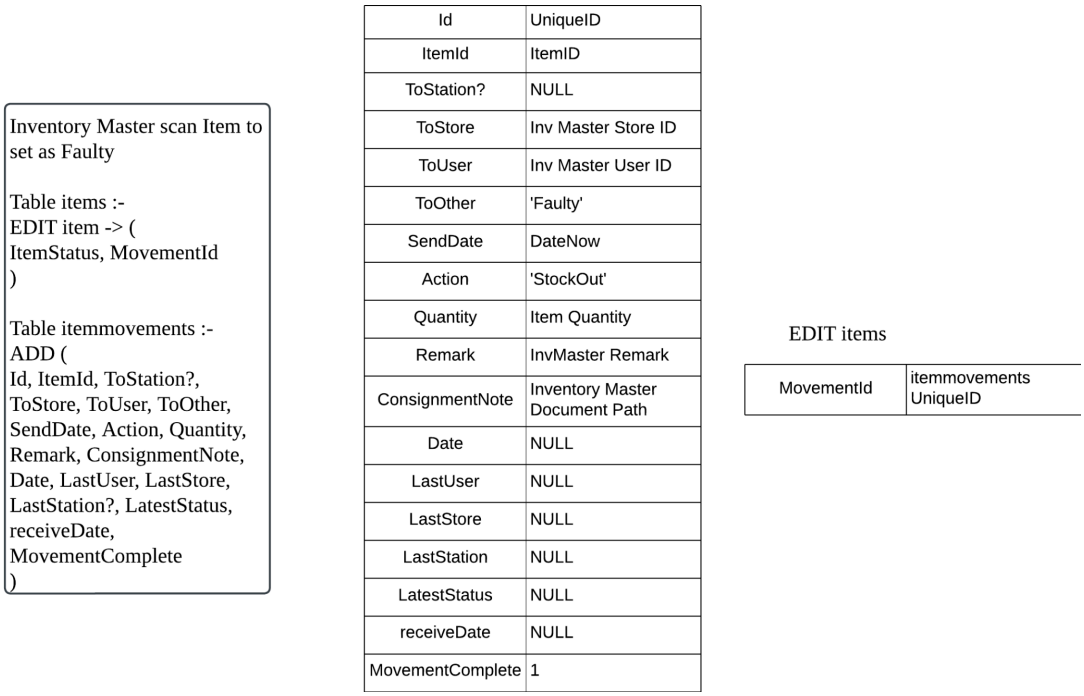


Diagram 3.15 Inventory Master mark item as Faulty

4. Admin and Inventory Master

Admin and Inventory use the same features and coding file. This document will use the term Inventory Master to include both roles. The use of the term Admin will define that only Admin is mentioned.

Admin and inventory Master will use the following coding files:
1. ItemMovement.cshtml
2. QrMaster.cshtml
3. InvMainAPI.cs

Inventory Master can navigate to different features in the system using the Inventory Admin Dashboard. Figure 4.1 shows the Admin Dashboard interface.

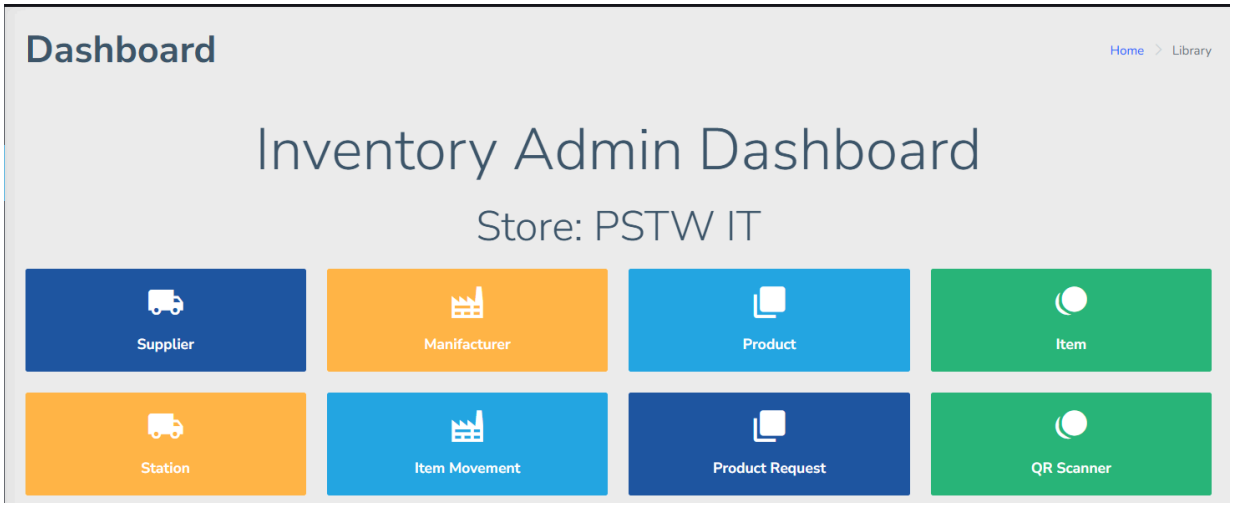


Figure 4.1 Admin Dashboard

4.1 Qr Scanner

Inventory Masters are to send items by scanning the item's qr code. Item is available to be sent to others while latest status is 'Ready To Deploy' and only can be sent to suppliers or marked as faulty when latest status is 'Faulty'. This means a 'Faulty' item are not able to be sent to Technicians

Latest status defined by *ToOther* and *LatestStatus*, whichever comes last.

4.1.1 User Interface

Inventory Master can start scanning items in Qr Scanner. Figure 4.2 shows the behaviour of the Qr Scanner.

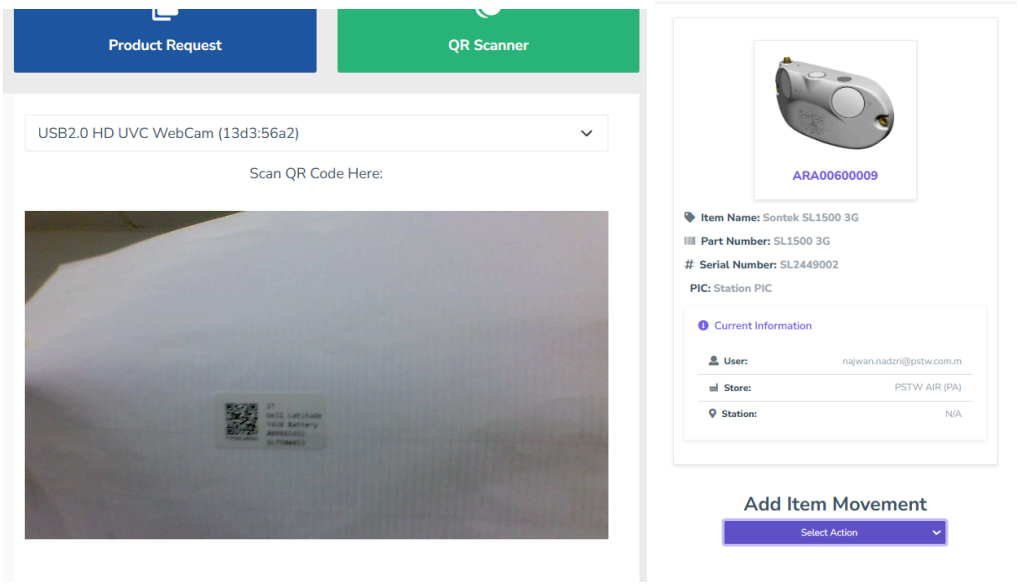


Figure 4.2 Qr Scanner Interface

After scanning, the user fills a form to start sending items to others. Figure 4.3 shows an example of the form displayed to initiate item movement to Technician by user.

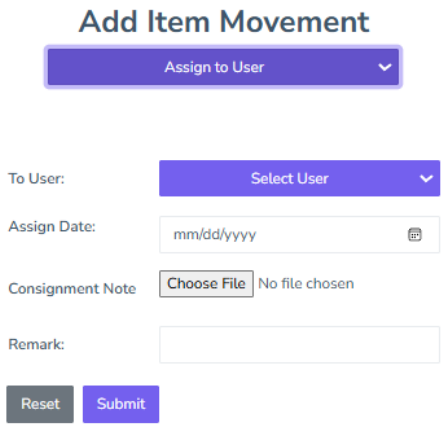


Figure 4.3 Assign to User Form

4.1.2 Coding Structure

Qr scanner is defined as in Figure 4.4. Qr scanner in this system is using a vue library[1].

```
<qrcode-stream :constraints="selectedConstraints" :formats="['qr_code']" :track="trackFunctionSelected.value"
               v-on:camera-on="onCameraReady" v-on:detect="onDecode"v-on:error="onError">
</qrcode-stream>
```

Figure 4.4 Definement of Qr scanner

Input select allows users to use different camera input devices. Figure 4.5 shows this element.

```
<select class="form-select" v-if="!thisItem"  v-model="selectedCameraId" v-on:change="updateCamera">
    <option v-for="device in videoInputDevices" :key="device.deviceId" :value="device.deviceId">
        {{ device.label || `Camera ${videoInputDevices.indexOf(device) + 1}` }}
    </option>
</select>
```

Figure 4.5 Definement of Select Input Camera Device

Qr scanner depends on these methods

1. OnCameraReady. This method handles camera settings such as tweak sharpness and resolution to pull out the best of the camera settings. Figure 4.6 shows this method.

```
//Setting Camera
async onCameraReady(videoElement) {

    const video = document.querySelector("video");
    const track = video.srcObject.getVideoTracks()[0];

    if (track && track.getCapabilities) {
        const capabilities = track.getCapabilities(); // Get camera capabilities

        if (capabilities.sharpness) {

            track.applyConstraints({ advanced: [{ width: 1280, height: 720 }]
            }).then(() => { return track.applyConstraints({ advanced: [{ sharpness: 100 }] });
            }).then(() => { return track.applyConstraints({ advanced: [{ exposureMode: 'continuous' }] });
            }).then(() => {})
            .then(() => { console.log("📷 Applied Constraintsss:", track.getSettings());
            }).catch(err => console.error("❌ Failed to apply constraints:", err)
            );

        } else {
            console.warn("⚠️ Some settings are not supported on this camera");
        }
    }
```

Figure 4.6 onCameraReady method

The following (Figure 4.7) gives options to select camera input devices in the same method

```
    try {
        const devices = await navigator.mediaDevices.enumerateDevices();
        this.videoInputDevices = devices.filter(device => device.kind === 'videoinput');

        if (this.videoInputDevices.length > 0) {
            // Keep the selected camera if already chosen
            if (!this.selectedCameraId) {
                this.selectedCameraId = this.videoInputDevices[0].deviceId;
            }

            this.selectedConstraints = { deviceId: { exact: this.selectedCameraId } };
        } else {
            this.error = "No camera detected.";
        }
    } catch (err) {
        this.error = "Error accessing camera: " + err.message;
    }
},
```

Figure 4.7 Continuation of method onCameraReady

2. On Error. This method handles errors of the camera. Figure 4.8 shows this method.

```
//Showing Qr Error
onError(err) {
    let message = `[${err.name}]: `;
    if (err.name === "NotAllowedError") {
        message += "You have to allow camera acceccs.";
    } else if (err.name === "NotFoundError") {
        message += "There's no camera detect.";
    } else if (err.name === "NotReadableError") {
        message += "You are using camera on the other application.";
    } else {
        message += err.message;
    }
    this.error = message;
},
```

Figure 4.8 onError method

3. On Decode. This handles the QR code decoding. This will initiate the item fetching using `this.fetchItem()`. Figure 4.9 shows this method.

```
onDecode(detectedCodes) {
    // const endTime = performance.now();
    // this.scanTime = Math.round(endTime - this.scanStartTime); Calculate scan time

    if (detectedCodes.length > 0) {
        this.qrCodeResult = detectedCodes[0].rawValue; // Ambil URL dari rawValue
        this.UniqueID = this.qrCodeResult.split('/').pop(); // Ambil UniqueID dari URL
        this.fetchItem(this.UniqueID);
    }
},
```

Figure 4.9 OnDecode method

Next, Calling `fetchItem()` will determine the availability of the item to be sent. The interface will be reflected by parameters *itemlateststatus* and *itemassignedtouser*. Figure 4.10 shows the method.

```
async fetchItem(itemid) {
    try {
        const response = await fetch('/InvMainAPI/GetItem/' + itemid, {
            method: 'POST',}
        );
        if (response.ok) {

            this.thisItem = await response.json();
            this.itemlateststatus = this.thisItem.latestStatus ? this.thisItem.latestStatus : this.thisItem.toOther;
            this.itemassignedtouser = (this.thisItem.toUser === this.currentUser.id || this.thisItem.lastUser  === this.currentUser.id )
                                    && this.thisItem.lastStore === this.currentUser.store ? true : false;

        } else {
            console.error('Failed to fetch item information');
            this.responseMessage = await response.text();
        }
    } catch (error) {
        console.error('Error fetching item information:', error);
    }
},
```

Figure 4.10 `fetchItem()` method

There will be different form displayed based on the item's latest status such as Add Item Movement, Receive Item and etc.. Form will be displayed based on parameter *itemlateststatus* and *itemassignedtouser*. Figure 4.11 shows select options to send items to others.

```
<div v-if="itemlateststatus == 'Ready To Deploy' && this.itemassignedtouser">
        <h2 class="register-heading">Add Item Movement</h2>
        <div class="col-sm-3"></div>
        <div class="col-sm-6 offset-sm-3">
            <div class="dropdown">
                <select class="btn btn-primary dropdown-toggle col-md-10 " v-model="selectedAction" required>
                    <option class="btn-light" value="" disabled selected>Select Action</option>
                    <option class="btn-light" value="user">Assign to User</option>
                    <option class="btn-light" value="station">Assign to Station</option>
                    <option class="btn-light" value="store">Assign to Store</option>
                    <option class="btn-light" value="supplier">Assign to Supplier</option>
                    <option class="btn-light" value="faulty">Faulty</option>
                </select>
            </div>
        </div>
    </div>
</div>
```

Figure 4.11 Definement of Select input to send to other user element

While `addItemMovement` (Figure 4.12) form binding different sets of input display, parameters will later be checked to determine the following values

```
<form v-on:submit.prevent="addItemMovement" data-aos="fade-right">

    <div v-if="selectedAction === 'user'">...

    <div v-if="selectedAction === 'station'">...

    <div v-if="selectedAction === 'store'">...

    <div v-if="selectedAction === 'supplier'">...

    <div v-if="selectedAction === 'faulty'">...

</form>
```

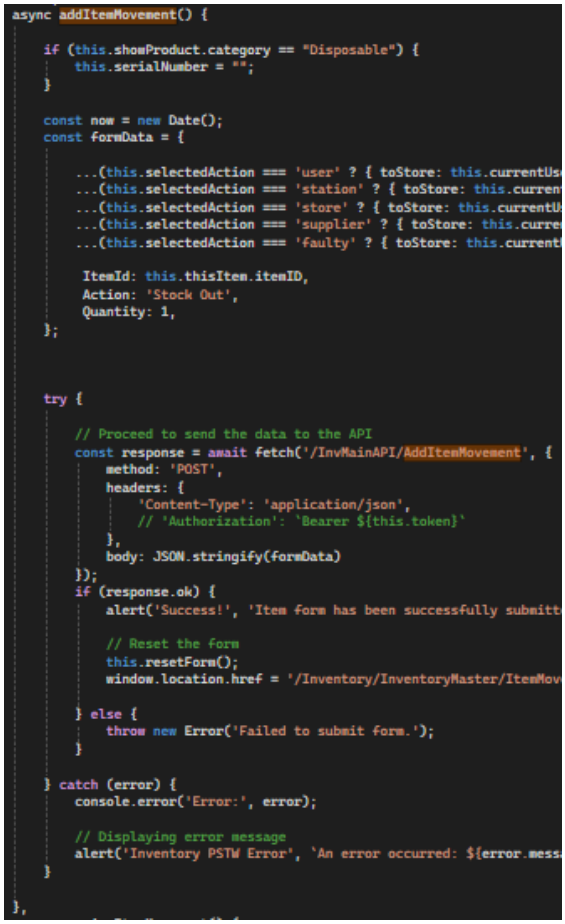
Figure 4.12 `addItemMovement` Form

Each set of input forms will use the same form with custom parameters to determine other functions. Figure 4.13 shows an example from one of the sets, which is Faulty.

```
<div v-if="selectedAction === 'faulty'">

    <div class="row register-form">
        <div class="col-md-3"></div>
        <div class="col-md-6">

            <div class="form-group row">
                <label class="col-sm-4 col-form-label">Consignment Note</label>
                <div class="col-sm-8">
                    <input type="file" id="document" name="document" class="form-control-file"
                           v-on:change="handleFileUpload" accept="image/png, image/jpeg, application/pdf" />
                </div>
            </div>

            <div class="form-group row">
                <label class="col-sm-4 col-form-label">Remark:</label>
                <div class="col-sm-8">
                    <input type="text" id="remark" name="remark" v-model="remark" class="form-control" required /
                </div>
            </div>


        </div>

        @* Submit and Reset Buttons *@
        <div class="form-group row">
            <div class="col-sm-8 offset-sm-3">
                <button type="button" v-on:click="resetForm" class="btn btn-secondary m-1">Reset</button>
                <button type="submit" class="btn btn-primary m-1">Submit</button>
            </div>
        </div>
    </div>
</div>
```

Figure 4.13 Faulty set of inputs

Form `addItemMovement` is as defined in Figure 4.14. FormData will be prepared based on parameter *selectedAction* and execute api accordingly.



```
async addItemMovement() {

    if (this.showProduct.category == "Disposable") {
        this.serialNumber = "";
    }

    const now = new Date();
    const formData = {

        ...(this.selectedAction === 'user' ? { toStore: this.currentUs
        ...(this.selectedAction === 'station' ? { toStore: this.curren
        ...(this.selectedAction === 'store' ? { toStore: this.currentU
        ...(this.selectedAction === 'supplier' ? { toStore: this.curre
        ...(this.selectedAction === 'faulty' ? { toStore: this.current

         ItemId: this.thisItem.itemID,
         Action: 'Stock Out',
         Quantity: 1,
    };



    try {

        // Proceed to send the data to the API
        const response = await fetch('/InvMainAPI/AddItemMovement', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                // 'Authorization': `Bearer ${this.token}`
            },
            body: JSON.stringify(formData)
        });
        if (response.ok) {
            alert('Success!', 'Item form has been successfully submitt

            // Reset the form
            this.resetForm();
            window.location.href = '/Inventory/InventoryMaster/ItemMov

        } else {
            throw new Error('Failed to submit form.');
        }

    } catch (error) {
        console.error('Error:', error);

        // Displaying error message
        alert('Inventory PSTW Error', `An error occurred: ${error.mess
    }

},
```

Figure 4.14 `addItemMovement` method

AddItemMovement API is defined in *InvMainAPI.cs*. The following will explain the operation of the API. Figure 4.15 shows the definement of *AddItemMovement* in the file.

```
[HttpPost("AddItemMovement")]
0 references
public async Task<IActionResult> AddItemMovement([FromBody] ItemMovementModel itemmovement)
{
```

Figure 4.15 Definement of *AddItemMovement*

The continuation of the api is defined in Figure 4.16.

```
var inventoryMaster = await _centralDbContext.InventoryMasters.Include("User").FirstOrDefaultAsync(i => i.UserId == itemmovement.ToUser);
if (inventoryMaster != null)
{
    itemmovement.ToStore = inventoryMaster.StoreId;
}


if (!string.IsNullOrEmpty(itemmovement.ConsignmentNote))
{
    var bytes = Convert.FromBase64String(itemmovement.ConsignmentNote);
    string filePath = "";

    string uniqueName = $"{itemmovement.ItemId}_{Guid.NewGuid()}";

    if (IsImage(bytes))
    {
        filePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot/media/inventory/itemmovement", uniqueName + itemmovement.ItemId + "_Request.jpg");
        itemmovement.ConsignmentNote = "/media/inventory/itemmovement/" + uniqueName + itemmovement.ItemId + "_Request.jpg";
    }
    else if (IsPdf(bytes))
    {
        filePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot/media/inventory/itemmovement", uniqueName + itemmovement.ItemId + "_Request.pdf");
        itemmovement.ConsignmentNote = "/media/inventory/itemmovement/" + uniqueName + itemmovement.ItemId + "_Request.pdf";
    }
    else
    {
        return BadRequest("Unsupported file format.");
    }

    await System.IO.File.WriteAllBytesAsync(filePath, bytes);
}

_centralDbContext.ItemMovements.Add(itemmovement);

await _centralDbContext.SaveChangesAsync(); // This generates the auto-incremented ItemID
```

Figure 4.16 *AddItemMovement* API operation

While checking if data *toUser* is an Inventory Master and defines the *StoreId*, the operation continues with storing the document submitted as *ConsignmentNote* in */media/inventory/itemmovement/* path. Then proceed to save changes and update the database. Figure 4.17 shows the second phase of the operation.

```
var updateItem = await _centralDbContext.Items.FindAsync(itemmovement.ItemId);

if (updateItem != null)
{
    if (itemmovement.ToOther == "On Delivery")
    {
        updateItem.ItemStatus = 2;
    }
    else if (itemmovement.ToOther == "Repair" || itemmovement.ToOther == "Cali
    {
        updateItem.ItemStatus = 4;
    }
    else if (itemmovement.ToOther == "Faulty")
    {
        updateItem.ItemStatus = 8;
    }

    //Console.WriteLine("updateItem.MovementId" + updateItem.MovementId);
    //Console.WriteLine("itemmovement.Id" + itemmovement.Id);

    updateItem.MovementId = itemmovement.Id;
    _centralDbContext.Items.Update(updateItem);
    await _centralDbContext.SaveChangesAsync(); // save changes for table item


}
```

Figure 4.17 Second phase of *AddItemMovement* API

Based on the API, after the operation in the *itemmovements* table, the operation continues to update item's status based on *ToOther* defined in passed FormData and the latest movement id in *items* table to the latest movement. These phases are designed such ways to acquire the

latest movement id first before updating the item's latest movement id in the items table based on the acquired movement id.

4.2 Item request

Inventory Masters can request products to other Inventory Masters and give approval for request products sent by Technicians or other Inventory Masters.

4.2.1 User Interface

This view can be accessed from the admin dashboard. Figure 4.18 shows different sort listings in this feature. The interfaces will be explained further in the following sections.

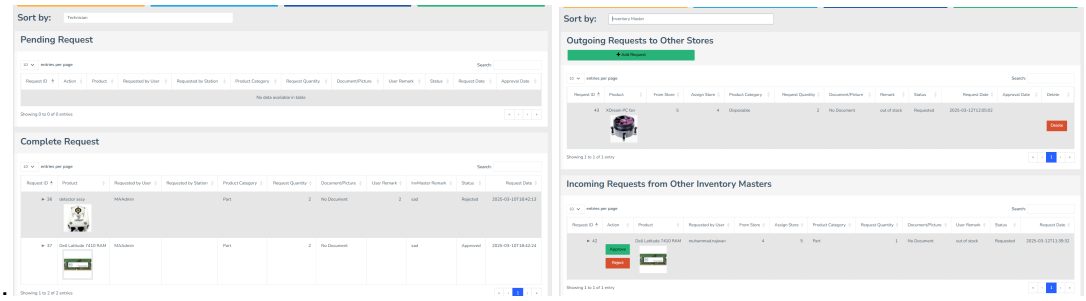


Figure 4.18 Multiple views of request sorting Interface

4.2.1.1 Sort by Technician

Product request sorted by 'Technician' shows the details of the Technician requests. Figure 4.19 shows this behaviour.

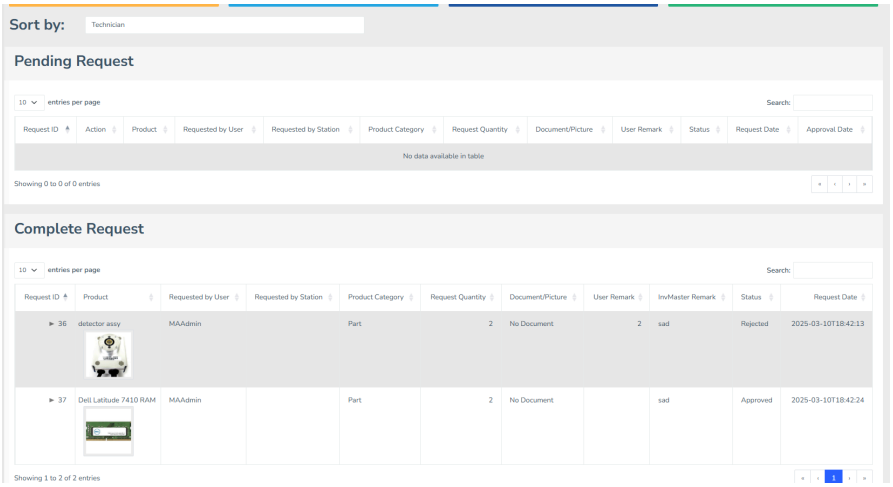


Figure 4.19 Sort by Technician Interface

4.2.1.2 Sort by Inventory Master

Product Request sorted by Inventory Master to show detailed Inventory Master requests. This helps to differentiate outgoing requests and incoming requests from other Inventory Masters. The interface followed with table completed requests. Figure 4.20 shows this behavior.

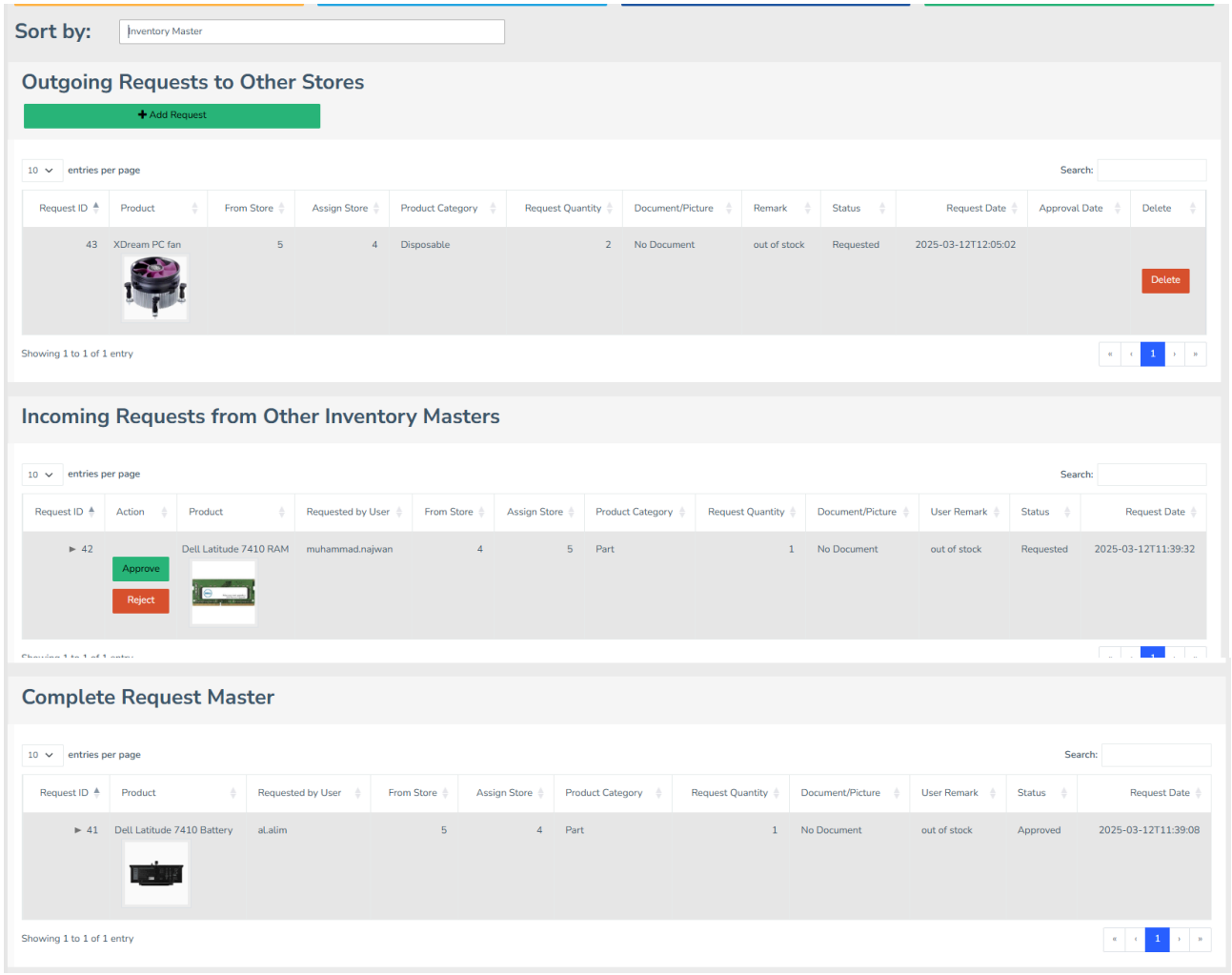


Figure 4.20 Sort by Inventory Master Interface

4.2.2 Coding Structure

This section shows the coding structure by each sorting type. Figure 4.21 shows the element to select sorting type.

```
<div class="row mb-3">
    <h2 for="sortSelect" class="col-sm-1 col-form-h2" style="min-width:150px;">Sort by:</h2>
    <div class="col-sm-4">
        <select id="sortSelect" class="form-control" v-model="sortBy" v-on:change="handleSorting">
            <option value="tech">Technician</option>
            <option value="master">Inventory Master</option>
        </select>
    </div>
</div>
```

Figure 4.21 Definement of Select Sort by Type input

Entering the request master page, Vue js will call `fetchRequest()` and fetch API to get the data of all item requests list using */ItemRequestList* API.
All request data will be stored within variable *items*. `fetchRequest()` methods allows Admin and Inventory Masters to retrieve all the data. Figure 4.22 shows this behaviour.

```
async fetchRequest() {
    try {
        const response = await fetch('/InvMainAPI/ItemRequestList', {
        method: 'GET', // Specify the HTTP method
        headers: {
            'Content-Type': 'application/json', // Set content type
        }
    });

    if (!response.ok) {
        throw new Error('Failed to fetch item');
    }
    this.items = await response.json();

    if (this.requestDatatable) {
        this.requestDatatable.clear().destroy();
    }
    if (this.settledrequestDatatable) {
        this.settledrequestDatatable.clear().destroy();
    }
    if (this.requestMasterDatatable) {
        this.requestMasterDatatable.clear().destroy();
    }
    if (this.requestOtherMasterDatatable) {
        this.requestOtherMasterDatatable.clear().destroy();
    }
    if (this.settledrequestMasterDatatable) {
        this.settledrequestMasterDatatable.clear().destroy();
    }
    this.initiateTable();
    }
    catch (error) {
        console.error('Error fetching item:', error);
    }
},
```

Figure 4.22 Definement of `fetchRequest()` method

4.2.2.1 Sort by Technician

Tab contents are displayed based on the variable *sortBy*. Figure 4.23 shows the usage of the `sortBy == tech` conditional rule and two tables defined for the tab content.

```
<div v-show="sortBy === 'tech'">
    <div class="row card">
        <div class="card-header">
            <h2>Pending Request</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="requestDatatable" style=" width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>

    <div class="row card">
        <div class="card-header">
            <h2>Complete Request</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="settledrequestDatatable" style=" width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>
</div>
```

Figure 4.23 Definement of Sort by Technician tab content

These tables sort the requested data by Technicians and its completeness. Figure 4.24 shows this behaviour using the *item.status* and *item.assignStoreItem* variable to display data in both tables. *item.status* is used to differentiate between responded and unresponded requests. `item.assignStoreItem != null` refers to Inventory Master requests and `item.assignStoreItem == null` refers to Technician requests.

```
this.requestDatatable = $('#requestDatatable').DataTable({
    "data": this.items.filter(item => item.status === "Requested" && item.assignStoreItem == null),
    "columns": [
        { "title": "Request ID", "data": "requestID" },
        { "title": "Action", "data": "requestID", "render": renderActionButtons, "className": "align-middle" },
        { "title": "Product", "data": "productName", "render": renderDocument },
        { "title": "Requested by User", "data": "userName" },
        { "title": "Requested by Station", "data": "stationName" },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Document/Picture", "data": "document", "render": renderDocument },
        { "title": "User Remark", "data": "remarkUser" },
        { "title": "Status", "data": "status" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Approval Date", "data": "approvalDate" }
    ],
    responsive: true,
    drawCallback: function (settings) { }
});

this.settledrequestDatatable = $('#settledrequestDatatable').DataTable({
    "data": this.items.filter(item => item.status !== "Requested" && item.assignStoreItem == null),
    "columns": [
        { "title": "Request ID", "data": "requestID" },
        { "title": "Product", "data": "productName", "render": renderDocument },
        { "title": "Requested by User", "data": "userName" },
        { "title": "Requested by Station", "data": "stationName" },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Document/Picture", "data": "document", "render": renderDocument },
        { "title": "User Remark", "data": "remarkUser" },
        { "title": "InvMaster Remark", "data": "remarkMasterInv" },
        { "title": "Status", "data": "status" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Approval Date", "data": "approvalDate" }
    ],
    responsive: true
});
```

Figure 4.24 Filter request table using *item.status and item.assignStoreItem* variable.

Approve and Reject button will only pop up in incomplete requests tables for each request. Figure 4.25 shows the corresponding modal up when either button was clicked.

```
function renderActionButtons(data, type, row) {
    var actiontButtons = `<div class="row" style="padding: 5px;"> <button type="button" class="btn btn-success approve-btn" data-id="${data}">Approve</button></div> <div class="row" style="padding: 5px;">
    <button type="button" class="btn btn-danger reject-btn" data-id="${data}">Reject</button></div>`;
    return actiontButtons;
}
```

```
// Attach event listeners
$('#requestDatatable tbody').on('click', '.reject-btn', function () {
    const requestID = $(this).data('id');
    self.rejectRequestModal(requestID);
});

$('#requestDatatable tbody').on('click', '.approve-btn', function () {
    const requestID = $(this).data('id');
    self.approveRequestModal(requestID);
});
```

```
<div class="modal fade" id="rejectModal" tabindex="-1" role="dialog" aria-labelledby="rejectRequestModalLabel" aria-hidden="true">...

<div class="modal fade" id="approveModal" tabindex="-1" role="dialog" aria-labelledby="approveRequestModalLabel" aria-hidden="true">...
```

Figure 4.25 Approve and Reject message in incomplete requests table

4.2.2.2 Sort by Inventory Master

Tab contents are displayed based on the variable *sortBy*. Figure 4.26 shows the usage of the *sortBy* variable and three tables defined in the tab content.

```
<div v-show="sortBy === 'master'">
    <div class="row card">
        <div class="card-header">
            <h2>Outgoing Requests to Other Stores</h2>
            <button id="addRequestBtn" class="btn btn-success col-md-3 col-lg-3 m-1 col-12"
                    v-on:click="showRequestModal = true">
                <i class="fa fa-plus"></i> Add Request
            </button>

        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="requestMasterDatatable" style=" width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>

    <div class="row card">
        <div class="card-header">
            <h2>Incoming Requests from Other Inventory Masters</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="requestOtherMasterDatatable" style=" width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>

    <div class="row card">
        <div class="card-header">
            <h2>Complete Request Master</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="settledrequestMasterDatatable" style=" width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>
</div>
```

Figure 4.26 Definement of Sort by Inventory Master tab content

Three variables are used in this element. *item.assignStoreItem, item.status* and *item.userId* controls the view of tab details content in each associated data. Figure 4.27 shows this behaviour.

For Inventory Master's part, there are three tables :-
1. *#requestMasterDatatable* - Display their own unresponded request (Only view and delete)
2. *#requestOtherMasterDatatable* - Display other Inv Master request (Only view & approval)
3. *#settledrequestMasterDatatable* - Display complete Inv Master request (Only view)

```
this.requestMasterDatatable = $('#requestMasterDatatable').DataTable({
    "data": this.items.filter(item => item.assignStoreItem != null && item.status == "Requested" && item.userId == this.currentUserId),
    "columns": [
        { "title": "Request ID", "data": "requestID" },
        { "title": "Product", "data": "productName", "render": renderDocument },
        { "title": "From Store", "data": "fromStoreItem" },
        { "title": "Assign Store", "data": "assignStoreItem" },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Document/Picture", "data": "document", "render": renderDocument },
        { "title": "Remark", "data": "remarkUser" },
        { "title": "Status", "data": "status" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Approval Date", "data": "approvalDate" },
        {
            "title": "Delete", "data": "requestID",
            "render": function (data) {
                var deleteButton = `<button type="button" class="btn btn-danger delete-btn" data-id="${data}">Delete</button>`;
                return deleteButton;
            },
            "className": "align-middle",
        }
    ],
    responsive: true,
    drawCallback: function (settings) { }
});
this.requestOtherMasterDatatable = $('#requestOtherMasterDatatable').DataTable({
    "data": this.items.filter(item => item.status == "Requested" && this.storeUser.some(store => store.id === item.fromStoreItem)),
    "columns": [
        { "title": "Request ID", "data": "requestID" },
        { "title": "Action", "data": "requestID", "render": renderActionButtons, "className": "align-middle" },
        { "title": "Product", "data": "productName", "render": renderDocument },
        { "title": "Requested by User", "data": "userName" },
        { "title": "From Store", "data": "fromStoreItem" },
        { "title": "Assign Store", "data": "assignStoreItem" },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Document/Picture", "data": "document", "render": renderDocument },
        { "title": "User Remark", "data": "remarkUser" },
        { "title": "Status", "data": "status" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Approval Date", "data": "approvalDate" }
    ],
    responsive: true
});
this.settledrequestMasterDatatable = $('#settledrequestMasterDatatable').DataTable({
    "data": this.items.filter(item => item.status != "Requested" && item.assignStoreItem != null),
    "columns": [
        { "title": "Request ID", "data": "requestID" },
        { "title": "Product", "data": "productName", "render": renderDocument },
        { "title": "Requested by User", "data": "userName" },
        { "title": "From Store", "data": "fromStoreItem" },
        { "title": "Assign Store", "data": "assignStoreItem" },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Document/Picture", "data": "document", "render": renderDocument },
        { "title": "User Remark", "data": "remarkUser" },
        { "title": "Status", "data": "status" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Approval Date", "data": "approvalDate" }
    ],
    responsive: true
});
```

Figure 4.27 Definement of table in Inventory Master tab content.

The request form modal will only pop up after clicking ‘Add Request’ button on top of *#requestMasterDatatable* table. `watch:` and `computed:` Vue Object used in request form modal to handle frontend output properly.

Figure 4.28 shows `watch:` that using `showRequestModal()` to handle modal visibility, `computed:` that `showProduct()` to fetch category and the image of selected product and `filteredProducts()` to filter by search, then the line code of the request message.

```
<div class="modal fade" id="requestModal" tabindex="-1" role="dialog" aria-labelledby="addRequestModalLabel" aria-hidden="true">...
```

```
watch: {
    showRequestModal(newVal) {
        if (newVal) {
            $('#requestModal').modal('show');
        } else {
            $('#requestModal').modal('hide');
        }
    }
},
```

```
computed: {
    showProduct() {
        if (!this.productId) {
            return []; // No company selected, return empty list
        }
        const product = this.products.find(c => c.productId === this.productId);

        this.productCategory = product.category;

        return product ? product : {};
    },
    filteredProducts() {
        return this.products.filter(item =>
            item.productName.toLowerCase().includes(this.searchQuery.toLowerCase()) ||
            item.modelNo.toLowerCase().includes(this.searchQuery.toLowerCase())
        );
    }
},
```

Figure 4.28 Request message in *#requestMasterDatatable* table

Delete button will be available in *#requestMasterDatatable* table, which is designed to cancel the product request. Only unresponded requests can be deleted, the responded requests can’t be deleted. This action will delete the actual row in the database. Figure 4.29 shows delete function behaviour.

```
$('#requestMasterDatatable tbody').off('click', '.delete-btn');

$('#requestMasterDatatable tbody').on('click', '.delete-btn', function () {
    const requestID = $(this).data('id');
    self.deleteRequestItem(requestID);
});
```

Figure 4.29 Delete function in *#requestMasterDatatable* table

Figure 4.30 and Figure 4.31 shows the definement of Delete handler and its correspond API.

```
async deleteRequestItem(requestID) {
    if (!confirm("Are you sure you want to delete this request?")) {
        return false;
    }
    try {
        const response = await fetch(`/InvMainAPI/DeleteRequest/${requestID}`, {
            method: 'DELETE',
            headers: {
                'Content-Type': 'application/json',
            },
        });
        const result = await response.json();

        if (result.success) {
            alert(result.message);

                if ($.fn.dataTable.isDataTable('#requestDatatable')) {
                    const table = $('#requestDatatable').DataTable();
                    table.row($(`.delete-btn[data-id="${requestID}"]`).closest('tr')).remove().draw();
                }

                this.request = this.request.filter(req => req.requestID !== requestID);
        } else {
            alert(result.message);
        }
    }
    catch (error) {
        console.error("Error deleting item:", error);
            alert("An error occurred while deleting the item.");
    }
    finally {
        this.loading = false;
    }
},
```

Figure 4.30 Definement of Delete Request Handler

```
[HttpDelete("DeleteRequest/{requestId}")]
0 references
public async Task<IActionResult> DeleteRequest(int requestId)
{
    var requestApprove = _centralDbContext.Requests.FirstOrDefault(r => r.requestID == requestId && r.approvalDate != null);
    var requestDelete = _centralDbContext.Requests.FirstOrDefault(r => r.requestID == requestId);

    if (requestApprove != null)
    {
        return NotFound(new { success = false, message = "Request not found Or Admin Already Approve or Reject" });
    }

    if (requestDelete == null)
    {
        return NotFound(new { success = false, message = "Request not found" });
    }

    _centralDbContext.Requests.Remove(requestDelete);
    await _centralDbContext.SaveChangesAsync();

    return Ok(new { success = true, message = "Request deleted successfully" });
}
```

Figure 4.31 Definement of */DeleteRequest* API

4.3 Item Movement

Inventory Master can view pending item movement and completed item movement by All, Items, Stations. This helps keeping track of all items.

4.3.1 User Interface

This view can be accessed from the admin dashboard. Figure 4.32 shows different sort listings in this feature. Each sorting type will be elaborated further in the following sections.

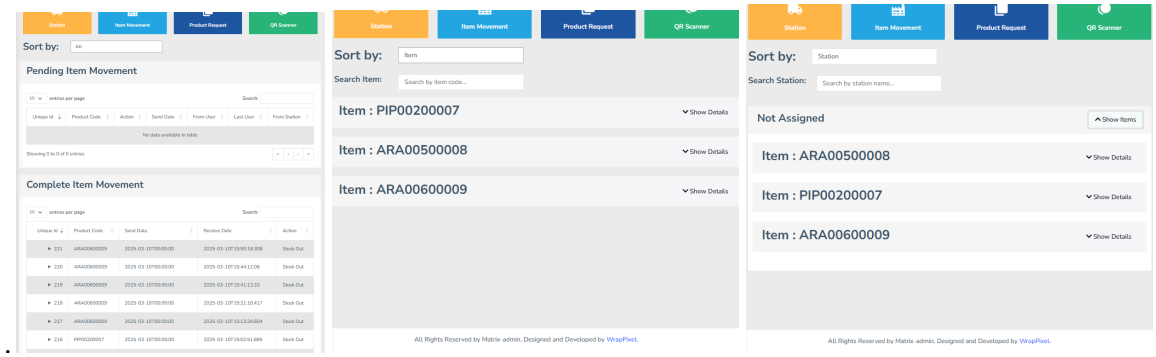


Figure 4.32 Multiple views of item movement sorting Interface

4.3.1.1 Sort by All

This interface will show pending and completed item movements in different tables. Item movements record sorted by All shows details of each movement. Figure 4.33 shows this behaviour.

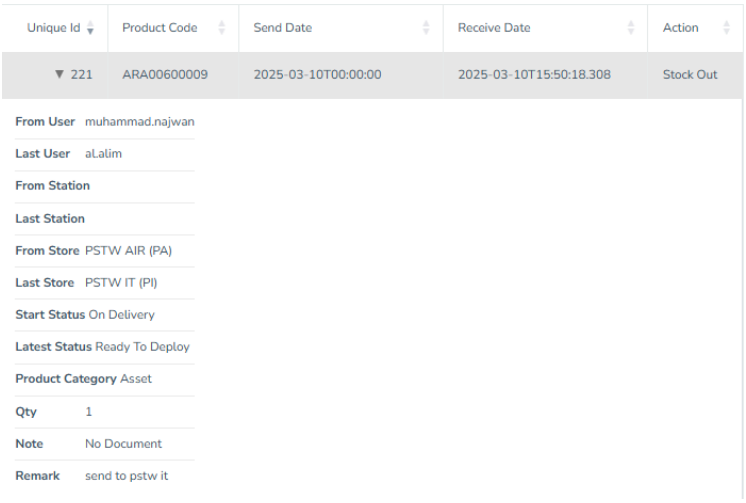


Figure 4.33 Sort by All Interface Item Movement Details

4.3.1.2 Sort by Items

Item movement records sorted by Items show the list by item. Inventory Master can use the search bar to look for specific records. This helps in tracking an item's movement. Each detail is provided with a graphical interface to enhance understanding. Figure 4.34 shows this behavior.

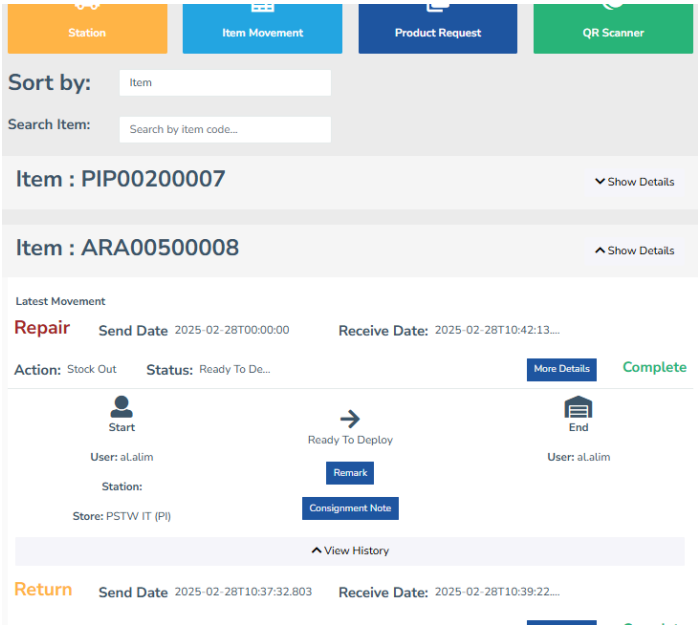


Figure 4.34 Item details in Sort by Item Interface

4.3.1.3 Sort by Stations

Item movement records sorted by Stations show the list by station. If an item is unassigned to any station, It will display under the 'Not Assigned' tab. Inventory Master can use the search bar to look for specific stations. Each detail is provided with a graphical interface to enhance understanding. Figure 4.35 shows this behaviour.

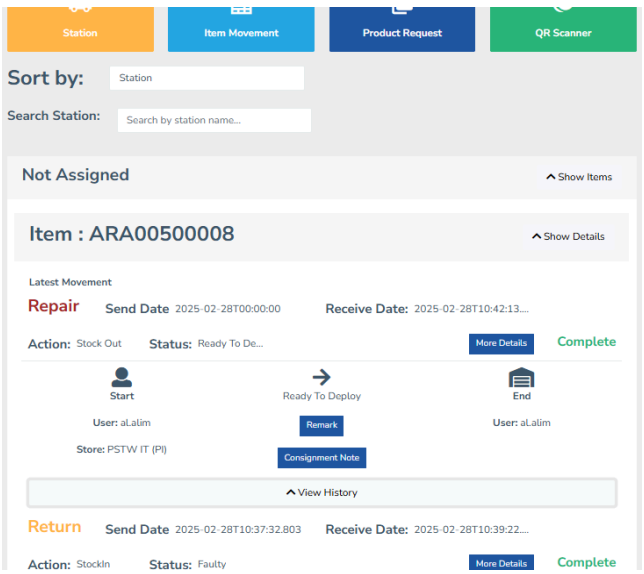


Figure 4.35 Sort by Stations

4.3.1.4 Logs (Admin Only)

Item movement records in logs show the list of item movement records. All item movement records will be displayed here and sorted by the latest item movement record. Admin can use the search bar to look for specific records. Figure 4.36 shows this behaviour.

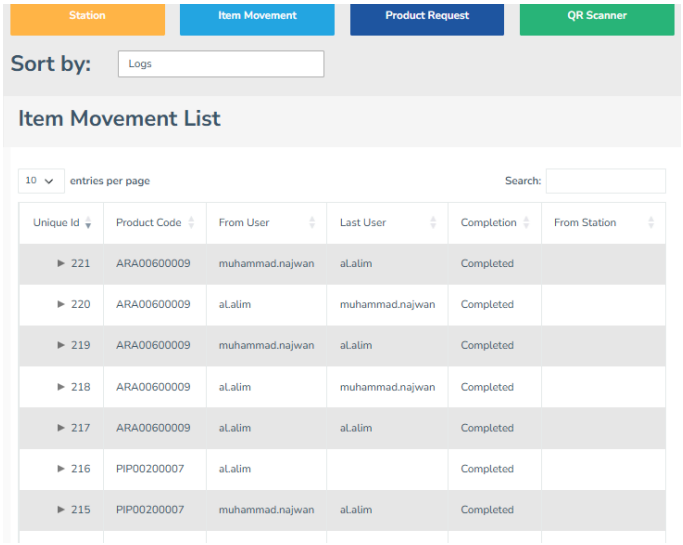


Figure 4.36 Sort by Stations

4.3.2 Coding Structure

This section shows the coding structure by each sorting type. Figure 4.37 shows the element to select sorting type.

```
<div class="row mb-3" >
    <h2 for="sortSelect" class="col-sm-1 col-form-h2" style="min-width:140px;">Sort by:</h2>
    <div class="col-sm-4">
        <select id="sortSelect" class="form-control" v-model="sortBy" v-on:change="handleSorting">
            <option value="all" >All</option>
            <option value="item">Item</option>
            <option value="station">Station</option>
            <option value="logs" v-if="currentRole == 'Super Admin'">Logs</option>
        </select>
    </div>
</div>
```

Figure 4.37 Select Sort by Type input

Item movements are defined as *items* as per this file. `fetchItem()` methods allows admin to retrieve all the data while Inventory Masters will only retrieve item movement records by their user Id and store Id. Figure 4.38 shows this behaviour.

```
async fetchItem() {
    this.loading = true;
    await this.fetchUser();
    try {
        // const token = localStorage.getItem('token'); // Get the token
        const response = await fetch('/InvMainAPI/ItemMovementList', {
            method: 'POST', // Specify the HTTP method
            headers: {
                'Content-Type': 'application/json', // Set content type
                // 'Authorization': `Bearer ${token}` // Include the tok
            }
        });

        if (!response.ok) {
            throw new Error('Failed to fetch item');
        }
        if(this.currentRole == "Super Admin"){
            this.items = await response.json();
            this.initAllTables();

        } else {
            const data = await response.json();
            this.items = data.filter(item =>
            item.toUser === this.currentUser.id ||
            item.lastUser === this.currentUser.id ||
            item.toStore === this.currentUser.store ||
            item.lastStore === this.currentUser.store
            );

            this.initAllTables();

        }
```

Figure 4.38 Definement of `fetchItems()` method

`handleSorting()` method called to initiate all tables using `renderTables()`. This behaviour shown in Figure 4.39.

```
handleSorting() {
    this.renderTables();
},
renderTables() {

    this.initAllTables();
    this.initiateTable();

},
```

Figure 4.39 Definement of `handleSorting()` and `renderTables()` method

4.3.2.1 Sort by All

Tab contents are displayed based on the variable *sortBy*. Figure 4.40 shows the usage of the sortBy variable and two tables defined for the tab content.

```
<div v-if="sortBy === 'all'">
    <div class="row card">
        <div class="card-header">
            <h2>Pending Item Movement</h2>
        </div>
        <div class="card-body">
            <div v-if="loading">
                <div class="spinner-border text-info" role="status">
                    <span class="visually-hidden">Loading...</span>
                </div>
            </div>
            <table class="table table-bordered table-hover table-striped no-wrap" id="itemMovementNotCompleteDatatable" st
        </div>
    </div>

    <div class="row card">
        <div class="card-header">
            <h2>Complete Item Movement</h2>
        </div>
        <div class="card-body">
            <div v-if="loading">
                <div class="spinner-border text-info" role="status">
                    <span class="visually-hidden">Loading...</span>
                </div>
            </div>
            <table class="table table-bordered table-hover table-striped no-wrap" id="itemMovementCompleteDatatable" style
        </div>
    </div>
</div>
```

Figure 4.40 Definement of Sort by All tab content

These tables sort the item movements data by its completeness. Figure 4.41 shows this behaviour using the *item.movementmentComplete* variable.

```
this.itemMovementNotCompleteDatatable = $("#itemMovementNotCompleteDatatable").DataTable({
        data: this.items.filter((m) => m.movementComplete == 0),
    columns: [...],
    order: [[0, "desc"]],
    responsive: true,
});

this.itemMovementCompleteDatatable = $("#itemMovementCompleteDatatable").DataTable({
    data: this.items.filter((m) => m.movementComplete == 1),
    columns: [...],
    order: [[0, "desc"]],
    responsive: true,
});
```

Figure 3.41 Filter items in table using *item.movementmentComplete* variable.

4.3.2.2 Sort by Item

Tab contents are displayed based on the variable *sortBy*. Figure 4.42 shows the usage of the *sortBy* variable.

```
<div v-if="sortBy === 'item'">
    <div v-if="loading">
        <div class="spinner-border text-info" role="status">...
    </div>
    <div v-for="(group, itemId) in filteredItems" :key="itemId" class="row card">
        <div class="card-header d-flex justify-content-between align-items-center">...

        <!-- Hide all details unless button is clicked -->
        <div v-show="categoryVisible[itemId]" class="card-body">
            <div v-for="(movement, index) in group.movements.sort((a, b) => a.id - b.id).reverse()" :key="movement.id" class="movement-row">
                <!-- 📌 Show Only Latest Movement -->
                <div v-if="index === 0" class="row">
                    <strong>Latest Movement</strong>
                    <div class="col-md-12 d-flex flex-wrap align-items-center gap-3 p-2 border-bottom">...

                    <div v-show="detailsVisible[movement.id]" class="col-md-12 mt-2">...

                </div>
            </div>

            <!-- 📌 Single View History Button -->
            <button class="btn btn-light w-100 text-left" v-on:click="toggleHistory(itemId)">
                <i :class="historyVisible[itemId] ? 'fas fa-chevron-up' : 'fas fa-chevron-down'"></i> View History
            </button>

            <div v-show="historyVisible[itemId]" class="history-row">
                <div v-for="(movement, i) in group.movements.slice(1)" :key="i" class="row mt-2">
                    <div class="col-md-12 d-flex flex-wrap align-items-center gap-3 p-2 border-bottom">...


                    <!-- 📌 Details Section (Hidden by Default) -->
                    <div v-show="detailsVisible[movement.id]" class="col-md-12 mt-2">...

                </div>
            </div>

        </div>
```

Figure 4.42 Definement of Sort by item tab content

Two variables are used in this element. `categoryVisible[]` and `historyVisible[]` controls the view of tab details content in each associated data. Meanwhile, `categoryVisible[]` and `historyVisible[]` perform the same behaviour in displaying details of each movement. Figure 4.43 shows this behaviour.

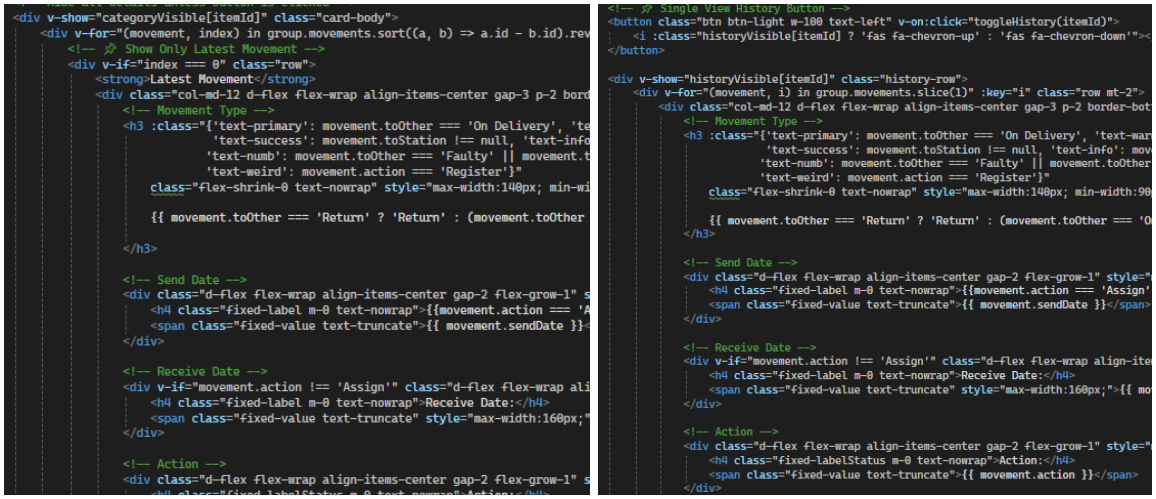


Figure 4.43 Behaviour of `categoryVisible[]` and `historyVisible[]`

Column records are customly defined to show contents based on fetched data. Figure 4.44 shows this behaviour.

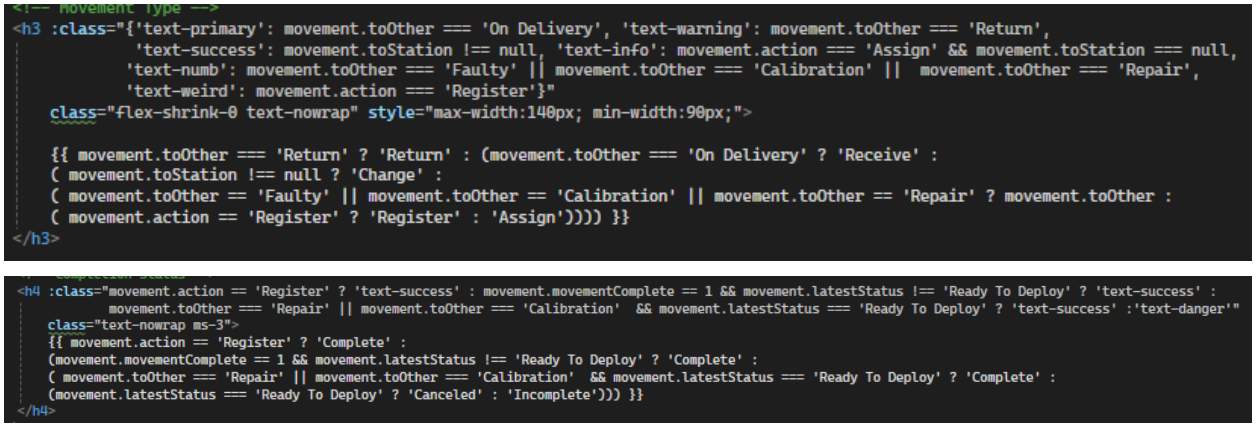


Figure 4.44 Example of custom content in element

`groupedItems()` and `filteredItems()` defined in Vue computed objects are responsible for grouping the records and filtering based on the search bar. Figure 4.45 and Figure 4.46 show the definement of these methods.

```
computed: {
    groupedItems() {
        let grouped = this.items.reduce((acc, movement) => {
            if (!acc[movement.itemId]) {
                acc[movement.itemId] = {
                    uniqueID: movement.uniqueID,
                    movements: [],
                };
            }
            acc[movement.itemId].movements.push(movement);
            return acc;
        }, {});

        // Sort items from newest to oldest & filter them
            for (let itemId in grouped) {
                let movements = grouped[itemId].movements
                    .sort((a, b) => b.id - a.id); // Newest to oldest

                // Ensure at least 3 movements before stopping
                let stopIndex = movements.slice(3).findIndex(m => m.toOther === 'Return' && m.movementComplete == 1);

                if (stopIndex !== -1) {
                    stopIndex += 3; // Adjust index since we sliced after the first 3
                    movements = movements.slice(0, stopIndex + 1); // Keep at least 3 + the "Return" row
                }

                grouped[itemId].movements = movements;
            }
            return grouped;
    },
```

Figure 4.45 Definement of `groupedItems()` method

Variable *stopIndex* stores the stop index of the record. The records of each item will return records of the first 3 item movements until the following 'Return' record with completed item movement. This will debloat the interface with only relevant records.

Figure 4.44 shows the definition of `filteredItems()` which will use the data computed by the groupItems `()` method then apply filtering from the search bar.

```
filteredItems() {
        if (!this.searchQuery.trim()) {
            return this.groupedItems;
        }
        const searchLower = this.searchQuery.toLowerCase();
        return Object.fromEntries(
            Object.entries(this.groupedItems).filter(([_, group]) =>
                group.uniqueID.toLowerCase().includes(searchLower)
            )
        );

},
```

Figure 4.46 Definement of `filteredItems()` method

4.3.2.3 Sort by Station

Tab contents are displayed based on the variable *sortBy*. Figure 4.47 shows the usage of the *sortBy* variable.

```
<div v-if="sortBy === 'station'">
    <div v-for="(items, station) in filteredStation" :key="stationName"
    :class="{'bg-light-gray': station === 'Unassign Station', 'bg-white': station !== 'Unassign Station'}" class="station-category card mt-3">
        <!-- Station Header -->
        <div class="card-header d-flex justify-content-between align-items-center">
            <h3>{{ station }}</h3>
            <button class="btn btn-light" v-on:click="toggleCategory(station)">
                <i :class="categoryVisible[station] ? 'fas fa-chevron-up' : 'fas fa-chevron-down'"></i> Show Items
            </button>
        </div>

        <!-- Show Items Under Each Station -->
        <div v-show="categoryVisible[station]" class="card-body">
            <div v-for="(group, itemId) in items" :key="itemId" class="row card">
                <!-- Item Header -->
                <div class="card-header d-flex justify-content-between align-items-center">...

                <!-- Show Movements for Each Item -->
                <div v-show="categoryVisible[itemId]" class="card-body">
                    <div v-for="(movement, index) in group.movements.sort((a, b) => a.id - b.id).reverse()" :key="movement.id" class="movement-row">...

                    <!-- Single View History Button -->
                    <button class="btn btn-light w-100 text-left" v-on:click="toggleHistory(itemId)">
                        <i :class="historyVisible[itemId] ? 'fas fa-chevron-up' : 'fas fa-chevron-down'"></i> View History
                    </button>

                    <div v-show="historyVisible[itemId]" class="history-row">...

                </div>
            </div>
        </div>
    </div>
</div>
```

Figure 4.47 Definement of Sort by Station Tab Content

While this definement behaves just like as defined in sort by items, this time, items are again grouped by station. This means that each item in the station behaves the same as in Sort by items.

Figure 4.48 and Figure 4.49 show the methods that are responsible to group the records into a group by stations and the display element will use what `filteredStation()` returns. `filteredStation()` will filter data grouped by stations from the `groupByStation()` method by the search bar before passing to the display element.

```
groupedByStation() {

        let groupedByItem = this.items.reduce((acc, movement) => {
            if (!acc[movement.uniqueID]) {
                acc[movement.uniqueID] = {
                    uniqueID: movement.uniqueID,
                    movements: [],
                };
            }
            acc[movement.uniqueID].movements.push(movement);
            return acc;
        }, {});

        let groupedByStation = {};

        Object.keys(groupedByItem).forEach(itemId => {
            let movements = groupedByItem[itemId].movements
                .sort((a, b) => b.id - a.id); // Newest → Oldest

            // Ensure at least 3 movements before stopping
            let stopIndex = movements.slice(3).findIndex(m => m.toOther === 'Return' && m.movementComplete == 1);

            // Remove older movements
            if (stopIndex !== -1) {
                stopIndex += 3; // Adjust index since we sliced after the first 3
                movements = movements.slice(0, stopIndex);
            }

            if (movements.length > 0) {
                let latestMovement = movements[0];
                let station = latestMovement.lastStationName || latestMovement.toStationName || "Not Assigned";

                if (!groupedByStation[station]) { groupedByStation[station] = {}; }

                groupedByStation[station][itemId] = { uniqueID: itemId, movements };
            }
        });

        // 🔹 **Sort stations & move 'Unassign Station' to last**
        let sortedKeys = Object.keys(groupedByStation).sort((a, b) => {
            if (a === "Unassign Station") return 1;
            if (b === "Unassign Station") return -1;
            return a.localeCompare(b);
        });

        let sortedGrouped = {};
        sortedKeys.forEach(key => {
            sortedGrouped[key] = groupedByStation[key];
        });

        return sortedGrouped;

},
```

Figure 4.48 Definement of *groupByStation()* method

```
filteredStation() {
        if (!this.searchStation) {
            return this.groupedByStation;
        }

        let searchQuery = this.searchStation.toLowerCase();
        let grouped = this.groupedByStation;
        let filtered = {};

        Object.keys(grouped).forEach(station => {
            if (station.toLowerCase().includes(searchQuery)) {
                filtered[station] = grouped[station];
            }
        });
        return filtered;
},
```

Figure 4.49 Definement of *filteredStation()* method

4.3.2.4 Logs (Admin Only)

Tab contents are displayed based on the variable *sortBy*. Figure 4.50 shows the usage of the sortBy variable.

```
<div v-if="sortBy === 'logs'">
    <div v-if="loading">
        <div class="spinner-border text-info" role="status">
            <span class="visually-hidden">Loading...</span>
        </div>
    </div>
    <div class="row card">
        <div class="card-header">
            <h2>Item Movement List</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="itemDatatable"
                   style=" width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>
</div>
```

Figure 4.50 Definement of Logs Tab Content

Table with id *itemDatatable* returns the logs. This is defined in `initiateTable()` method. Figure 4.51 shows its behaviour.

```
initiateTable() {
    this.loading = true;

    self = this;
    this.itemDatatable = $('#itemDatatable').DataTable({
        "data": this.items,
        "columns": [...],
        order: [[0, "desc"]],  // Sorting by "Product Code"
        responsive: true,
        drawCallback: function (settings) {...},
    })

    this.loading = false;
},
```

Figure 4.51 Definement of *initiateTable()* method

5. Technician

Technician is a role that has been assigned to a station. Technicians may have been assigned to multiple stations.

Technicians will use the following coding files:
1. ItemMovementUser.cshtml
2. QrUser.cshtml
3. InvMainAPI.cs

5.1 Product Request

Technicians can send request products to Inventory Masters, wait for Inventory master approval and can delete incomplete only before approval.

5.1.2 User Interface

This view can be accessed from the User dashboard. Figure 5.1 shows Technician request table and button add request and Figure 5.2 shows the form modal to initiate product request.

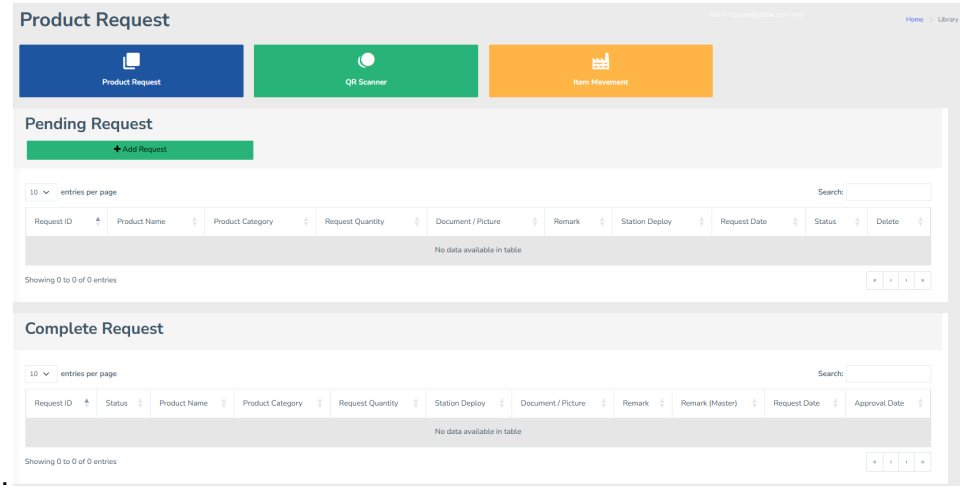


Figure 5.1 Add Request button on top of the Datatable

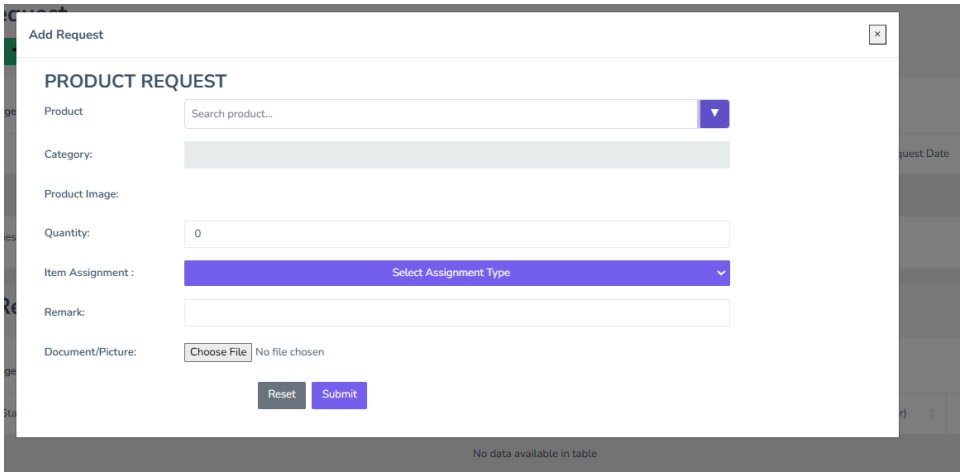


Figure 5.2 Product request form modal

5.1.3 Coding Structure

Entering the request Technician page, Vue js will call `fetchRequest()` and fetch *ItemRequestListEachUser* API to retrieve the data. All request data will be stored as variable *this.request*. `fetchRequest()` method allows Technicians to retrieve only the data that is related to them by referring to their *UserId*. Figure 5.3 shows this behaviour. Figure 5.4 shows the definement of *ItemRequestListEachUser*.

```
async fetchRequest() {
    try
    {
        const response = await fetch(`/InvMainAPI/ItemRequestListEachUser/${this.userId}`,
        {
            method: 'GET', // Specify the HTTP method
            headers: {
                'Content-Type': 'application/json', // Set content type
            }
        });

        if (!response.ok) {
            throw new Error('Failed to fetch request List');
        }
        this.request = await response.json();

        if ($.fn.dataTable.isDataTable('#requestDatatable')) {
            $('#requestDatatable').DataTable().clear().destroy();
        }
        if ($.fn.dataTable.isDataTable('#settledrequestDatatable')) {
            $('#settledrequestDatatable').DataTable().clear().destroy();
        }

        this.initiateTable();
    }
    catch (error) {
        console.error('Error fetching request List:', error);
    }
},
```

Figure 5.3 Definement of `fetchRequest()` method

```
[HttpGet("ItemRequestListEachUser/{userId}")]
0 references
public async Task<IActionResult> ItemRequestListEachUser(int userId)
{
    var requests = await _centralDbContext.Requests
        .Include(i => i.Product).Include(i => i.Station).Where(r => r.UserId == userId).ToListAsync();

    return Json(requests.Select(i => new
    {
        i.requestID,
        i.ProductId,
        productName = i.Product?.ProductName,
        productPicture = i.Product?.ImageProduct,
        i.UserId,
        i.status,
        i.StationId,
        stationName = i.Station?.StationName,
        i.RequestQuantity,
        i.requestDate,
        i.ProductCategory,
        i.Document,
        i.approvalDate,
        i.remarkMasterInv,
        i.remarkUser,
    }));
}
```

Figure 5.4 Definement of *ItemRequestListEachUser* API

There are two tables on the Technician page and these tables sort the request data by pending and completed. Figure 5.5 shows this behaviour using the *req.status* variable.

```
this.pendingRequestDatatable = $('#requestDatatable').DataTable({
    "data": this.request.filter(req => req.status === "Requested"),
    "columns": [
        { "title": "Request ID", "data": "requestID", "createdCell": (td, cellData) => $(td).attr('id', `qr${cellData}`) },
        { "title": "Product Name", "data": "productName", "render": (data, type, full) => renderDocument(full.productPicture) },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Document / Picture", "data": "document", "render": (data, type, full) => renderDocument(data) },
        { "title": "Remark", "data": "remarkUser" },
        { "title": "Station Deploy", "data": "stationName", "render": (data) => data || "Self Assign" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Status", "data": "status" },
        { "title": "Delete", "data": "requestID", "render": renderDeleteButton, "className": "align-middle" }
    ],
    responsive: true,
});

this.settledRequestDatatable = $('#settledrequestDatatable').DataTable({
    "data": this.request.filter(req => req.status !== "Requested"),
    "columns": [
        { "title": "Request ID", "data": "requestID", "createdCell": (td, cellData) => $(td).attr('id', `qr${cellData}`) },
        { "title": "Status", "data": "status" },
        { "title": "Product Name", "data": "productName", "render": (data, type, full) => renderDocument(full.productPicture) },
        { "title": "Product Category", "data": "productCategory" },
        { "title": "Request Quantity", "data": "requestQuantity" },
        { "title": "Station Deploy", "data": "stationName", "render": (data) => data || "Self Assign" },
        { "title": "Document / Picture", "data": "document", "render": (data, type, full) => renderDocument(data) },
        { "title": "Remark", "data": "remarkUser" },
        { "title": "Remark (Master)", "data": "remarkMasterInv" },
        { "title": "Request Date", "data": "requestDate" },
        { "title": "Approval Date", "data": "approvalDate" }
    ],
    responsive: true,
});
```

Figure 5.5 Filter request Technician in table by *req.status* variable.

The product request modal pops up by a click of ‘Add Request’ button. Figure 5.6 shows the definement of the Add Request button. Figure 5.7 shows how the clicking event is handled. Figure 5.8 shows the definement of the modal and *addRequest* form.

```
<div class="row card">
    <div class="card-header">
        <h2>Pending Request</h2>
        <button id="addRequestBtn" class="btn btn-success col-md-3 col-lg-3 m-1 col-12"><i class="fa fa-plus"></i> Add Request</button>
    </div>
    <div class="card-body">
    @*  <div v-if="loading">
            <div class="spinner-border text-info" role="status">
                <span class="visually-hidden">Loading...</span>
            </div>
        </div> *@
        <table class="table table-bordered table-hover table-striped no-wrap" id="requestDatatable" style=" width:100%;border-style: solid; border-width: 1px"></table>
    </div>

</div>
```

Figure 5.6 Definement of Add Request button

```
$(function () {
        app.mount('#requestProduct');

        // Attach a click event listener to elements with the class 'btn-success'.
        $('#addRequestBtn').on('click', function () {
            $('#requestModal').modal('show');
        });
        $('.closeModal').on('click', function () {
            $('.modal').modal('hide');
        });
});
```

Figure 5.7 Add Button Click Handler

```
<div class="modal fade" id="requestModal" tabindex="-1" role="dialog" aria-labelledby="addRequestModalLabel" aria-hidden="true">
    <div class="modal-dialog modal-dialog-centered modal-xl" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="addRequestModalLabel">Add Request</h5>
                <button type="button" class="closeModal" data-dismiss="modal" aria-label="Close" v-on:click="showRequestModal=false">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <div class="container-fluid">
                    <form v-on:submit.prevent="addRequest" data-aos="fade-right">
                        <div class=" register" data-aos="fade-right">
                            <div class="row" data-aos="fade-right">

                                <div class="col-md-12">
                                    <div class="tab-content" id="myTabContent">
                                        <div class="tab-pane fade show active" id="home" role="tabpanel" aria-labelledby="home-tab">
                                            <h3 class="register-heading">PRODUCT REQUEST</h3>
                                            <div class="row register-form">
                                                <div class="col-md-13">

                                                    <!-- Product Name -->
                                                    <div class="form-group row">
                                                        <label class="col-sm-2 col-form-label">Product</label>
                                                        <div class="col-sm-8">
                                                            <div class="dropdown" v-click-outside="closeDropdown">
                                                                <!-- Button + Input dalam satu box -->
                                                                <div class="dropdown-toggle-box" v-on:click="dropdownOpen = !dropdownOpen">
                                                                    <input type="text" class="form-control" v-model="searchQuery"
                                                                           placeholder="Search product..." v-on:focus="dropdownOpen = true" v-on:click.stop />
                                                                    <button type="button" class="btn btn-primary dropdown-btn" v-on:click.stop="dropdownOpen = !dropdownOpen">
                                                                        ▼
```

Figure 5.8 Definement of *requestModal* modal including *addRequest* form

addRequest form will make use of */AddRequest* API to handle submit product request processes. Figure 5.9 shows the API usage. Figure 5.10 Definement of API used.

```
// Send the data to the API
const response = await fetch('/InvMainAPI/AddRequest', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(requestData)
});
```

Figure 5.9 `addRequest()` method sent request data to */AddRequest* API

```
#region ItemRequestUser
[HttpPost("AddRequest")]
0 references
public async Task<IActionResult> AddRequest([FromBody] RequestModel request)
{
    if (!ModelState.IsValid)...
    try...
}
```

Figure 5.10 */AddRequest* API inside *ItemRequest region* save request data to database

Each pending product request will be able to be cancelled by the requester. Technicians can make use of the delete button to cancel requests. Figure 5.11 shows the definement of the Delete button. Figure 5.12 shows the definement of delete button handler. Figure 5.13 shows the definement of deleting method of *deleteRequestItem*. Figure 5.14 deleting API of */DeleteRequest*.

```
this.requestDatatable = $('#requestDatatable').DataTable({
    "data": this.request.filter(request => request.status == "Requested"),
    "columns": [
        {...,
            {...,
        {...,
        {...,

        {...,
        {...,
        {...,
        {...,
        {...,
        {
            "title": "Delete",
            "data": "requestID",
            "render": function (data) {
                var deleteButton = `<button type="button" class="btn btn-danger delete-btn" data-id="${data}">Delete</button>`;
                return deleteButton;
            },
            "className": "align-middle",
        }

    ],
    responsive: true,
});
```

Figure 5.11 Definement of Delete button in datatable

```
$('#requestDatatable tbody').off('click', '.delete-btn');

$('#requestDatatable tbody').on('click', '.delete-btn', function () {
    const requestID = $(this).data('id');
    self.deleteRequestItem(requestID);
});
```

Figure 5.12 Definement of Delete button handler

```
async deleteRequestItem(requestID) {
    if (!confirm("Are you sure you want to delete this request?")) {
        return false;
    }
    try {
        const response = await fetch(`/InvMainAPI/DeleteRequest/${requestID}`, {
            method: 'DELETE',
            headers: {
                'Content-Type': 'application/json',
            },
        });
        const result = await response.json();

        if (result.success) {
            alert(result.message);

                if ($.fn.dataTable.isDataTable('#requestDatatable')) {
                    const table = $('#requestDatatable').DataTable();
                    table.row($(`.delete-btn[data-id="${requestID}"]`).closest('tr')).remove().draw();
                }

                this.request = this.request.filter(req => req.requestID !== requestID);
        } else {
            alert(result.message);
        }
    }
    catch (error) {
        console.error("Error deleting item:", error);
            alert("An error occurred while deleting the item.");
    }
    finally {
        this.loading = false;
    }
},
```

Figure 5.13 Definement of `deleteRequestItem()` method

```
[HttpDelete("DeleteRequest/{requestId}")]
0 references
public async Task<IActionResult> DeleteRequest(int requestId)
{
    var requestApprove = _centralDbContext.Requests.FirstOrDefault(r => r.requestID == requestId && r.approvalDate != null);
    var requestDelete = _centralDbContext.Requests.FirstOrDefault(r => r.requestID == requestId);

    if (requestApprove != null)
    {
        return NotFound(new { success = false, message = "Request not found Or Admin Already Approve or Reject" });
    }

    if (requestDelete == null)
    {
        return NotFound(new { success = false, message = "Request not found" });
    }

    _centralDbContext.Requests.Remove(requestDelete);
    await _centralDbContext.SaveChangesAsync();

    return Ok(new { success = true, message = "Request deleted successfully" });
}
```

Figure 5.14 AddRequest API inside *ItemRequest region* save request data to database

5.2 Item Movement

Technicians can view pending item movement and completed item movement by All, Items, and Stations. This helps keeping track of all items.

5.2.1 User Interface

This view can be accessed from the User dashboard. Figure 5.15 shows different sort listings in this feature.

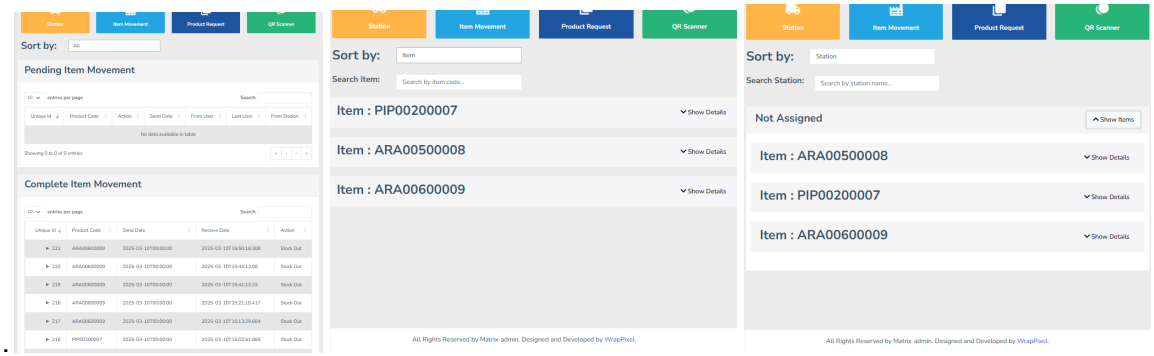


Figure 5.15 Multiple views of item movement sorting Interface

5.2.1.1 Sort by All

Item movements record sorted by All shows the details of the movement. Pending and Completed item movement are separated into 2 different tables. Figure 5.16 shows the details of each movement.

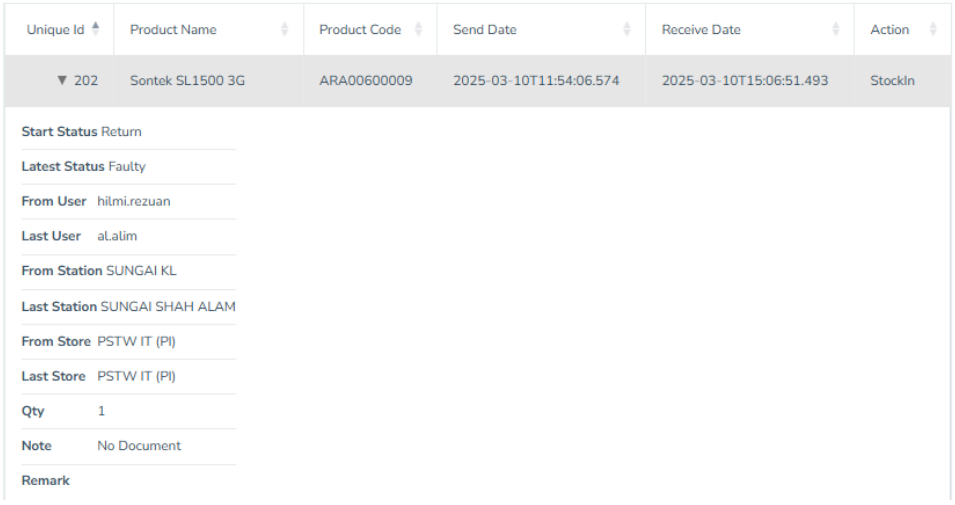


Figure 5.16 Item Movement details in Sort by All Interface

5.2.1.2 Sort by Items

Item movements record sorted by Items shows the list by item. Technicians can use the search bar to look for specific records. This helps in tracking an item's movement. A graphical interface is provided to enhance understanding of the data. Figure 5.18 shows this behavior.

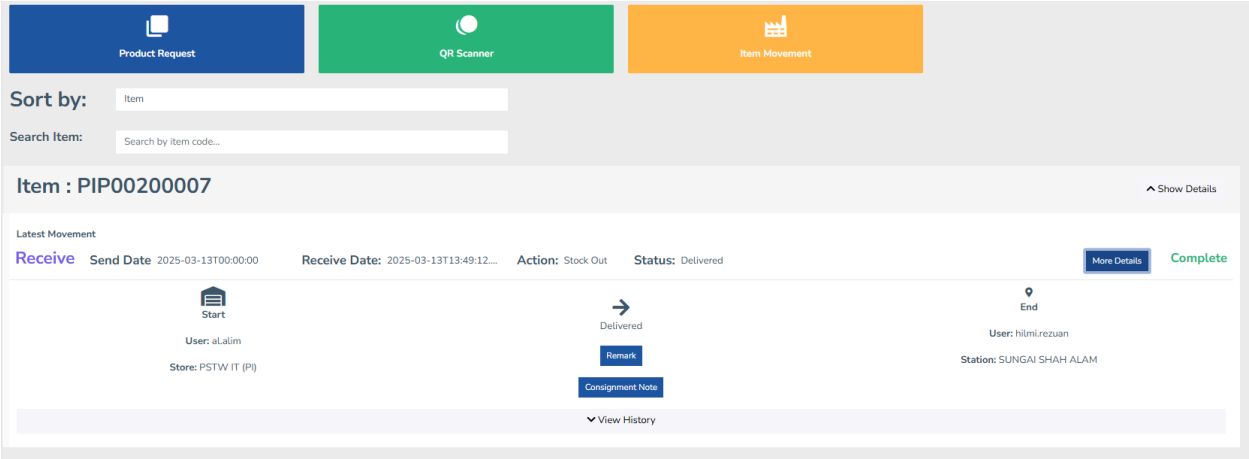


Figure 5.18 Sort by Item Interface

5.2.1.3 Sort by Stations

Item movements record sorted by Stations shows the list by station. If an item is unassigned to any station, It will display under the Self Assigned tab. Technicians can use the search bar to look for specific records. A graphical interface is provided to enhance understanding of the data. Figure 5.19 shows an example of the interface.

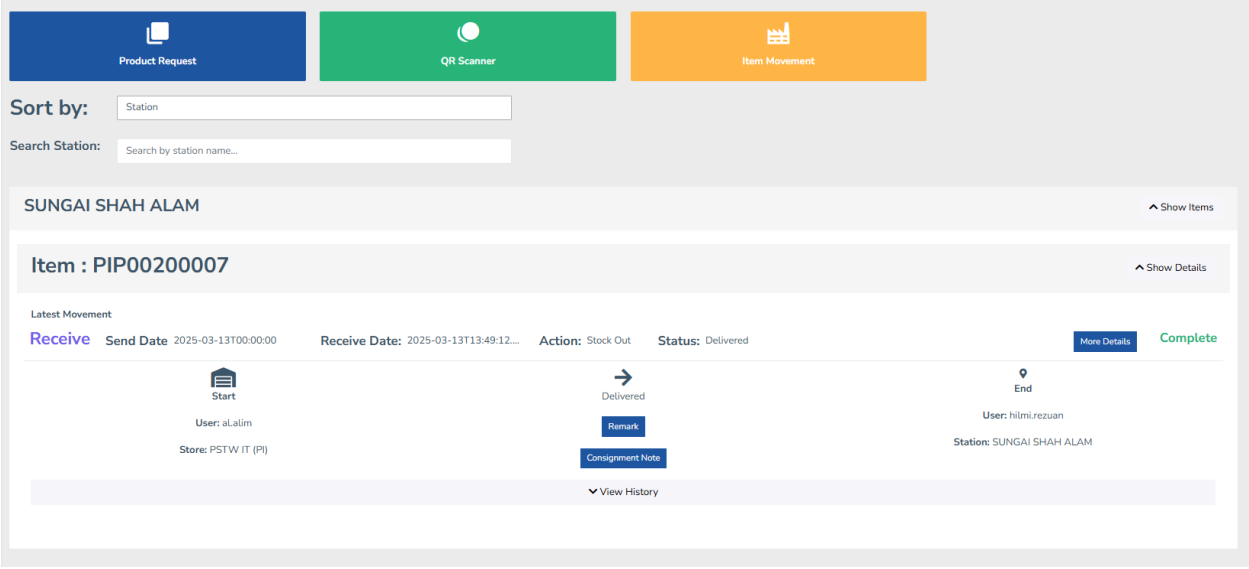


Figure 5.19 Sort by Stations Interface

5.2.2 Coding Structure

This section shows the coding structure by each sorting type. Figure 5.20 shows the element to select sorting type.

```
<div class="row mb-3">
    <h2 for="sortSelect" class="col-sm-1 col-form-h2" style="min-width:150px;">Sort by:</h2>
    <div class="col-sm-4">
        <select id="sortSelect" class="form-control" v-model="sortBy" v-on:change="handleSorting">
            <option value="all">All</option>
            <option value="item">Item</option>
            <option value="station">Station</option>
        </select>
    </div>
</div>
```

Figure 5.20 Select Sort by Type input

`fetchItemMovement()` method allows Technicians to retrieve the data that is assigned to them by their user Id and station Id. Item movements are defined as *this.itemMovements* as per this file. Filtering per *userId* and *stationId* happens in the API. Figure 5.21 shows `fetchItemMovement()` method. Figure 5.22 shows the definement of the used API.

```
async fetchItemMovement() {
    try {
        const response = await fetch("/InvMainAPI/ItemMovementUser", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
        });

        if (!response.ok) throw new Error("Failed to fetch item movement");

        const data = await response.json();
        this.itemMovements = data.map((movement) => ({
            ...movement,
            showDetails: false,
        }));

        this.renderTables();
    } catch (error) {
        console.error("Error fetching item:", error);
    }
},
```

Figure 5.21 Definement of `fetchItemMovement()` method

```
[HttpPost("ItemMovementUser")]
0 references
public async Task<IActionResult> ItemMovementUser()
{
    try
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound("Item movement record not found.");
        }

        var itemMovementUser = await _centralDbContext.ItemMovements
                                .Include(i => i.Item)
                                .ThenInclude(i => i.Product)
                                .Include(i => i.FromStore)
                                .Include(i => i.FromStation)
                                .Include(i => i.FromUser)
                                .Include(i => i.NextStore)
                                .Include(i => i.NextStation)
                                .Include(i => i.NextUser)
                                .Where(i => i.ToUser == user.Id || i.LastUser == user.Id)
                                .ToListAsync();

        return Json(itemMovementUser.Select(i => new...));
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
```

Figure 5.22 Definement of */ItemMovementUser* API

`handleSorting()` method called to initiate all tables using `renderTables()`. This behaviour is shown in Figure 5.23.

```
handleSorting() {
    this.renderTables();
},

renderTables() {
    if (this.sortBy === "all") {
        this.initAllTables();
    }
},
```

Figure 5.23 Definement of `handleSorting()` and `renderTables()` method

Each sorting type will be explained further in the following sections.

5.2.2.1 Sort by All

Tab contents are displayed based on the variable *sortBy*. Figure 5.24 shows the usage of the *sortBy* variable and two tables defined for the tab content.

```
<div v-if="sortBy === 'all'">
    <div class="row card">
        <div class="card-header">
            <h2>Pending Item Movement</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="itemMovementNotCompleteDatatable"
            style="width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>

    <div class="row card">
        <div class="card-header">
            <h2>Complete Item Movement</h2>
        </div>
        <div class="card-body">
            <table class="table table-bordered table-hover table-striped no-wrap" id="itemMovementCompleteDatatable"
            style="width:100%;border-style: solid; border-width: 1px"></table>
        </div>
    </div>
</div>
```

Figure 5.24 Definement of Sort by All tab content

These tables sort the item movements data by its completeness. Figure 5.25 shows this behaviour using the *item.movementmentComplete* variable.

```
// Table 1: Not Complete Movements
this.itemMovementNotCompleteDatatable = $("#itemMovementNotCompleteDatatable").DataTable({
    data: notCompleteData,
    columns: [...],
    responsive: true,
});

// Table 2: Completed Movements
this.itemMovementCompleteDatatable = $("#itemMovementCompleteDatatable").DataTable({
    data: completeData,
    columns: [...],
    responsive: true,
});
```

Figure 5.25 Filter items in table using *item.movementmentComplete* variable.

The Item Movement completeness filtering filters data by its *movementComplete* variable. Figure 5.26 shows this behaviour.

```
latestMovements.forEach(movement => {
    if (movement.movementComplete == 0) {
        notCompleteData.push(movement);
    } else if (movement.movementComplete == 1) {
        completeData.push(movement);
    }
});
```

Figure 5.26 Definement of Item Movement Completeness filtering

5.2.2.2 Sort by Item

Tab contents are displayed based on the variable *sortBy*. Figure 5.27 shows the usage of the *sortBy* variable.

```
<!-------------------------------------------ITEM CATEGORY------------------------------------------------------------------------------------------------>
<div v-if="sortBy === 'item'">
    <div v-for="(group, itemId) in filteredItems" :key="itemId" class="row card">
        <div class="card-header d-flex justify-content-between align-items-center">...

        <!-- Hide all details unless button is clicked -->
        <div v-show="categoryVisible[itemId]" class="card-body">
            <div v-for="(movement, index) in group.movements.sort((a, b) => a.id - b.id).reverse()" :key="movement.id" class="movement-row">

                <div v-if="index === 0" class="row">
                    <strong>Latest Movement</strong>
                    <div class="col-md-12 d-flex flex-wrap align-items-center gap-3 p-2 border-bottom">...

                    <div v-show="detailsVisible[movement.id]" class="col-md-12 mt-2">...

                </div>
            </div>

            <button class="btn btn-light w-100 text-left" v-on:click="toggleHistory(itemId)">
                <i :class="historyVisible[itemId] ? 'fas fa-chevron-up' : 'fas fa-chevron-down'"></i> View History
            </button>

            <div v-show="historyVisible[itemId]" class="history-row">
                <div v-for="(movement, i) in group.movements.slice(1)" :key="i" class="row mt-2">
                    <div class="col-md-12 d-flex flex-wrap align-items-center gap-3 p-2 border-bottom">...

                    <div v-show="detailsVisible[movement.id]" class="col-md-12 mt-2">...

                </div>
            </div>

        </div>
    </div>
</div>
```

Figure 5.27 Definement of Sort by item tab content

Two variables are used in this element. `categoryVisible[]` and `historyVisible[]` controls the view of tab details content in each associated data. Meanwhile, `categoryVisible[]` and `historyVisible[]` perform the same behaviour in displaying details of each movement. Figure 5.28 shows this behaviour.

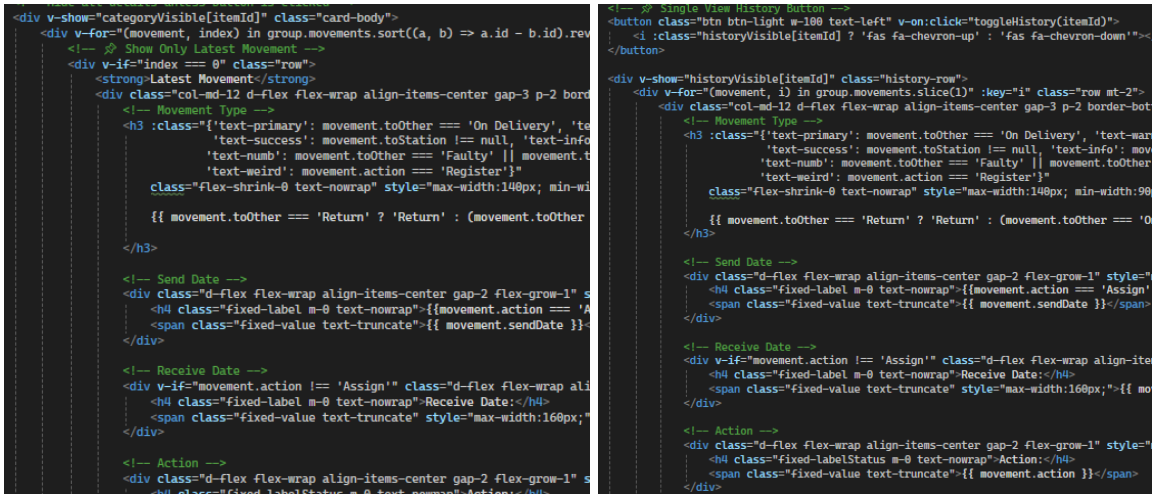


Figure 5.28 Behaviour of `categoryVisible[]` and `historyVisible[]`

Column records are customly defined to show contents based on fetched data. Figure 5.29 shows this behaviour.

```
<h3 :class="{'text-primary': movement.toOther === 'On Delivery', 'text-warning': movement.toOther === 'Return',
            'text-success': movement.toStation !== null, 'text-info': movement.action === 'Assign' && movement.toStation === null}"
    class="flex-shrink-0 text-nowrap" style="max-width:90px; min-width:90px;">

    {{ movement.toOther === 'Return' ? 'Return' : (movement.toOther === 'On Delivery' ? 'Receive' : ( movement.toStation !== null ? 'Change' : 'Assign')) }}

</h3>
```

Figure 5.29 Example of custom content in element

`processedGroupedItems()` and `filteredItems()` defined in Vue computed objects are responsible for grouping the records and filtering based on the search bar. Figure 5.30 and Figure 5.31 show the definement of these methods.

```
processedGroupedItems() {
    let grouped = this.itemMovements.reduce((acc, movement) => {
        if (!acc[movement.itemId]) {
            acc[movement.itemId] = {
                uniqueID: movement.uniqueID,
                movements: [],
            };
        }
        acc[movement.itemId].movements.push(movement);
        return acc;
    }, {});

    let filteredGrouped = {};

    for (let itemId in grouped) {
        let movements = grouped[itemId].movements
            .sort((a, b) => b.id - a.id); // Newest to oldest

        let stopIndex = movements.findIndex(m => m.toOther === 'Return' && m.movementComplete == 1 );

        let nextIndex = movements.findIndex(m => m.latestStatus === 'Ready To Deploy' && m.movementComplete == 1 );

        if (stopIndex !== -1) { movements = movements.slice(0, stopIndex); }

        if (nextIndex !== -1) { movements = movements.slice(0, nextIndex); }

        if (movements.length > 0) {
            filteredGrouped[itemId] = {
                uniqueID: grouped[itemId].uniqueID,
                movements: movements,
            };
        }
    }
    return filteredGrouped;
},
```

Figure 5.30 Definement of processed*GroupedItems()* method

Variable *stopIndex & nextIndex* stores the stop index of the record. The records of each item will return records until the following ‘Return’ or ‘Cancel’ record with completed item movement. This will debloat the interface with only relevant records.

Figure 5.31 shows the definition of *filteredItems()* which will use the data computed by processed*GroupedItems()* method and apply filtering by search bar.

```
filteredItems() {
    if (!this.searchQuery.trim()) {
        return this.processedGroupedItems;
    }
    const searchLower = this.searchQuery.toLowerCase();
    let grouped = this.processedGroupedItems;
    let filtered = {};

    Object.keys(grouped).forEach(item => {
        if (item.toLowerCase().includes(searchLower)) {
            if (grouped[item] > 0) {
                filtered[item] = grouped[item];
            }
        }
    });
    return filtered;
},
```

Figure 5.31 Definement of *filteredItems()* method

5.2.2.3 Sort by Station

Tab contents are displayed based on the variable *sortBy*. Figure 5.31 shows the usage of the sortBy variable.

```
<!-------------------------------STATION CATEGORY---------------------------------------------------------------------------->
<div v-if="sortBy === 'station'">
    <div v-for="(items, station) in filteredStation" :key="stationName"
    :class="{'bg-light-gray': station === 'Unassign Station', 'bg-white': station !== 'Unassign Station'}"   class="station-category card mt-3">
        <!-- Station Header -->
        <div class="card-header d-flex justify-content-between align-items-center">
            <h3>{{ station }}</h3>
            <button class="btn btn-light" v-on:click="toggleCategory(station)">
                <i :class="categoryVisible[station] ? 'fas fa-chevron-up' : 'fas fa-chevron-down'"></i> Show Items
            </button>
        </div>

        <!-- Show Items Under Each Station -->
        <div v-show="categoryVisible[station]" class="card-body">
            <div v-for="(group, itemId) in items" :key="itemId" class="row card">
                <!-- Item Header -->
                <div class="card-header d-flex justify-content-between align-items-center">...

                <!-- Show Movements for Each Item -->
                <div v-show="categoryVisible[itemId]" class="card-body">
                    <div v-for="(movement, index) in group.movements.sort((a, b) => a.id - b.id).reverse()" :key="movement.id" class="movement-row">
                        <div v-if="index === 0" class="row">
                            <strong>Latest Movement</strong>
                            <div class="col-md-12 d-flex flex-wrap align-items-center gap-3 p-2 border-bottom">...

                            <div v-show="detailsVisible[movement.id]" class="col-md-12 mt-2">...
                        </div>
                    </div>

                    <!-- Single View History Button -->
                    <button class="btn btn-light w-100 text-left" v-on:click="toggleHistory(itemId)">
                        <i :class="historyVisible[itemId] ? 'fas fa-chevron-up' : 'fas fa-chevron-down'"></i> View History
                    </button>

                    <div v-show="historyVisible[itemId]" class="history-row">...

                </div>
            </div>
        </div>
    </div>
</div>
```

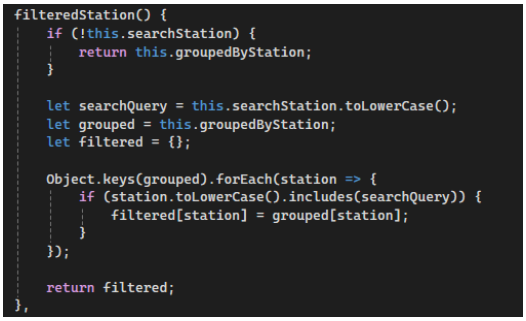
Figure 5.31 Definement of Sort by Station Tab Content

While this definement behaves just like defined in sort by items, this time, items are again grouped by station. This means that each item in the station behaves the same as in sort by item.

Figure 5.32 and Figure 5.33 show the methods that are responsible to group the records into a group by stations and the display element will use what `filteredStation()` returns. `filteredStation()` will filter data grouped by stations from the `groupedByStation()` method.

```
groupedByStation() {
    let groupedByItem = this.itemMovements.reduce((acc, movement) => {
        if (!acc[movement.uniqueID]) {
            acc[movement.uniqueID] = {
                uniqueID: movement.uniqueID,
                movements: [],
            };
        }
        acc[movement.uniqueID].movements.push(movement);
        return acc;
    }, {});

    let groupedByStation = {};

    Object.keys(groupedByItem).forEach(itemId => {
        let movements = groupedByItem[itemId].movements
            .sort((a, b) => b.id - a.id); // Newest → Oldest

        // Find first occurrence of 'Return' complete
        let stopIndex = movements.findIndex(m => m.toOther === 'Return' && m.movementComplete == 1 );

        let nextIndex = movements.findIndex(m => m.latestStatus === 'Ready To Deploy' && m.movementComplete == 1 );

        if (stopIndex !== -1) { movements = movements.slice(0, stopIndex); }

        if (nextIndex !== -1) { movements = movements.slice(0, nextIndex); }

        if (movements.length > 0) {
            let latestMovement = movements[0];
            let station = latestMovement.lastStationName || latestMovement.toStationName || "Self Assigned";

            if (!groupedByStation[station]) { groupedByStation[station] = {}; }

            groupedByStation[station][itemId] = { uniqueID: itemId, movements };
        }
    });

    // 🔹 **Sort stations & move 'Unassign Station' to last**
    let sortedKeys = Object.keys(groupedByStation).sort((a, b) => {
        if (a === "Unassign Station") return 1;
        if (b === "Unassign Station") return -1;
        return a.localeCompare(b);
    });

    let sortedGrouped = {};
    sortedKeys.forEach(key => {
        sortedGrouped[key] = groupedByStation[key];
    });

    return sortedGrouped;
},
```

Figure 5.32 Definement of *groupedByStation()* method



```
filteredStation() {
    if (!this.searchStation) {
        return this.groupedByStation;
    }

    let searchQuery = this.searchStation.toLowerCase();
    let grouped = this.groupedByStation;
    let filtered = {};

    Object.keys(grouped).forEach(station => {
        if (station.toLowerCase().includes(searchQuery)) {
            filtered[station] = grouped[station];
        }
    });

    return filtered;
},
```

Figure 5.33 Definement of *filteredStation()* method

5.3 Qr Scanner

Technicians are to do various actions by scanning the item's qr code. Technicians can receive, return and assign items to the station. All interfaces are displayed based on variables *ToOther, LatestStatus, ToUser, LastUser & Action* all on one page.

5.3.1 User Interface

Technicians can start scanning items in Qr Scanner. Figure 5.33 shows the behaviour of the Qr Scanner.

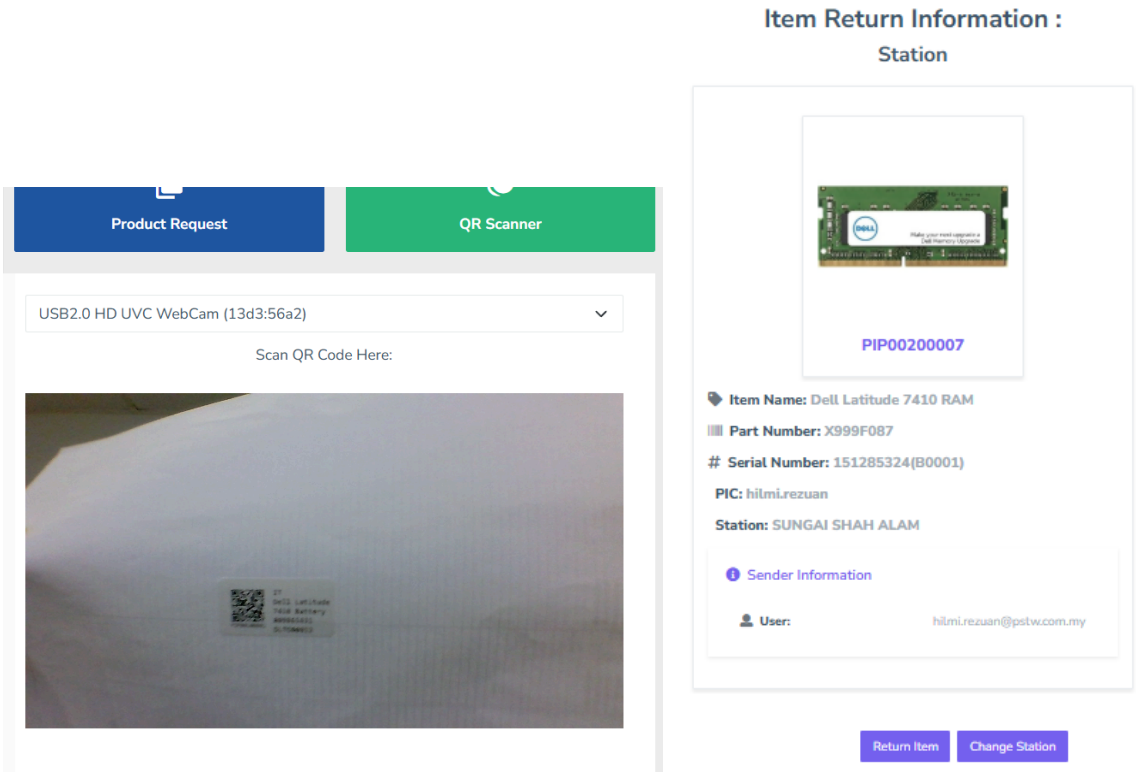


Figure 5.33 Qr Scanner Interface

After scanning, the user can receive an item as station (Station Assign) or (Self Assign). Figure 5.34, Figure 5.35, Figure 5.36, Figure 5.37, Figure 5.38 shows Form and button Technician can click if the item is assigned to them.

Return Item (Button) :- return the item without pressing Receive Item button first
Receive Item (Button) :- Technician receive the item when item arrive

Figure 5.34 Button display before Technician accept Item

Change Station (Button) :- Item already assign to station
Deploy Station (Button) :- Item not assign to any station (Self-Assign)

Figure 5.35 Different button when self-assign and station-assign

Return Item Form :- appear when Technician wants to return the item.

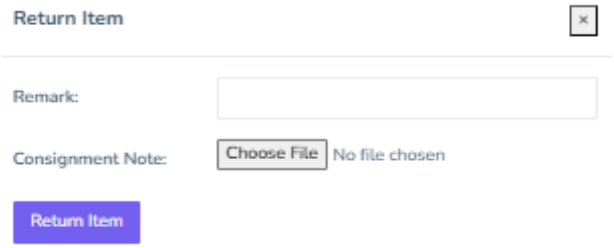


Figure 5.36 Return Form when Technician want to return the Item

Deploy or change Station From :- appear when Technician wants to assign item to station.

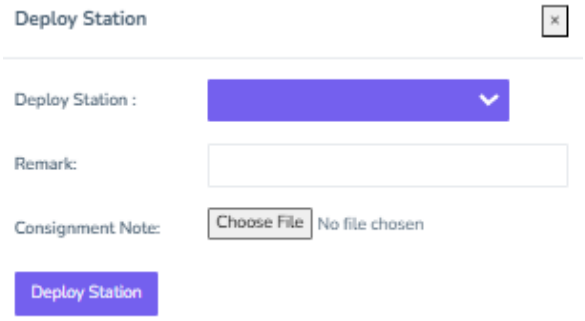


Figure 5.37 Assign or Change Station of received item

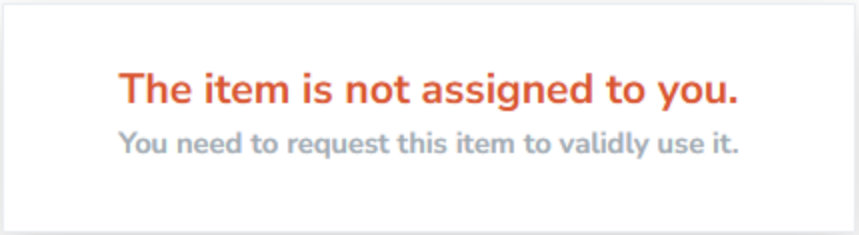


Figure 5.38 Scanning Item that not assign to them

5.3.2 Coding Structure

Qr scanner is defined as in Figure 5.39. Qr scanner in this system is using a vue library[1].

```
<qrcode-stream :constraints="selectedConstraints" :formats="['qr_code']" :track="trackFunctionSelected.value"
               v-on:camera-on="onCameraReady" v-on:detect="onDecode"v-on:error="onError">
</qrcode-stream>
```

Figure 5.39 Definement of Qr scanner

Input select allows users to use different camera input devices. Figure 5.40 shows this element.

```
<select class="form-select" v-if="!thisItem"  v-model="selectedCameraId" v-on:change="updateCamera">
    <option v-for="device in videoInputDevices" :key="device.deviceId" :value="device.deviceId">
        {{ device.label || `Camera ${videoInputDevices.indexOf(device) + 1}` }}
    </option>
</select>
```

Figure 5.40 Definement of Select Input Camera Device

Qr scanner depends on these methods

1. OnCameraReady. This method handles camera settings such as tweak sharpness and resolution to pull out the best of the camera settings. Figure 5.41 shows this method.

```
//Setting Camera
async onCameraReady(videoElement) {

    const video = document.querySelector("video");
    const track = video.srcObject.getVideoTracks()[0];

    if (track && track.getCapabilities) {
        const capabilities = track.getCapabilities(); // Get camera capabilities

        if (capabilities.sharpness) {

            track.applyConstraints({ advanced: [{ width: 1280, height: 720 }]
            }).then(() => { return track.applyConstraints({ advanced: [{ sharpness: 100 }] });
            }).then(() => { return track.applyConstraints({ advanced: [{ exposureMode: 'continuous' }] });
            }).then(() => {})
            .then(() => { console.log("📷 Applied Constraintsss:", track.getSettings());
            }).catch(err => console.error("❌ Failed to apply constraints:", err)
            );

        } else {
            console.warn("⚠️ Some settings are not supported on this camera");
        }
    }
```

Figure 5.41 onCameraReady method

The following (Figure 5.43) gives options for camera input devices in the same method

```
    try {
        const devices = await navigator.mediaDevices.enumerateDevices();
        this.videoInputDevices = devices.filter(device => device.kind === 'videoinput');

        if (this.videoInputDevices.length > 0) {
            // Keep the selected camera if already chosen
            if (!this.selectedCameraId) {
                this.selectedCameraId = this.videoInputDevices[0].deviceId;
            }

            this.selectedConstraints = { deviceId: { exact: this.selectedCameraId } };
        } else {
            this.error = "No camera detected.";
        }
    } catch (err) {
        this.error = "Error accessing camera: " + err.message;
    }
},
```

Figure 5.43 Continuation of method onCameraReady

2. On Error. This method handles errors of the camera. Figure 5.44 shows this method.

```
//Showing Qr Error
onError(err) {
    let message = `[${err.name}]: `;
    if (err.name === "NotAllowedError") {
        message += "You have to allow camera acccess.";
    } else if (err.name === "NotFoundError") {
        message += "There's no camera detect.";
    } else if (err.name === "NotReadableError") {
        message += "You are using camera on the other application.";
    } else {
        message += err.message;
    }
    this.error = message;
},
```

Figure 5.44 onError method

3. On Decode. This handles the QR code decoding. This will initiate the item fetching using `this.fetchItem()`. Figure 5.45 shows this method.

```
onDecode(detectedCodes) {
    // const endTime = performance.now();
    // this.scanTime = Math.round(endTime - this.scanStartTime); Calculate scan time

    if (detectedCodes.length > 0) {
        this.qrCodeResult = detectedCodes[0].rawValue; // Ambil URL dari rawValue
        this.UniqueID = this.qrCodeResult.split('/').pop(); // Ambil UniqueID dari URL
        this.fetchItem(this.UniqueID);
    }
},
```

Figure 5.45 OnDecode method

Next, Calling `fetchItem()` will determine the availability of the item to be sent. The interface will be reflected by parameters `displayStatus`. Figure 5.46 shows the method.

1. this.displayStatus = "arrived" :- Display **Receive Interface** (Return & Receive Button)
2. this.displayStatus = "return" :- Display **Return Interface** (Return & Assign Station Button)
3. this.displayStatus = "requestAgain" : - Display **Not Assign Message**
4. this.displayStatus = "differentUser" :- Display **Different Technician Message**

```
async fetchItem(itemid) {
    try {
        const response = await fetch('/InvMainAPI/GetItem/' + itemid, {
            method: 'POST',
        }
        );

        if (response.ok) {

            this.thisItem = await response.json();
            this.fetchStore(this.thisItem.toStore);

            if (this.thisItem.movementId != null && this.thisItem.toOther === "On Delivery" && this.thisItem.latestStatus == null && this.thisItem.currentUserId == this.currentUserId && this.thisItem.movementComplete == 0) {
                this.displayStatus = "arrived";
            } else if (this.thisItem.movementId != null && this.thisItem.latestStatus != null && this.thisItem.currentUserId == this.currentUserId && this.thisItem.latestStatus != "Ready To Deploy") {
                this.displayStatus = "return";
            } else if (this.thisItem.movementId != null && this.thisItem.toOther === "Return" && this.thisItem.latestStatus == null && this.thisItem.toUser == this.currentUserId) {
                this.displayStatus = "requestAgain";
            } else {
                this.displayStatus = "differentUser";
                this.thisItem = null;
            }
        } else {
            this.error = 'Qr Code Not Register to the system';
        }
    } catch (error) {
        console.error('Error fetching item information:', error);
    }
},
```

Figure 5.46 `fetchItem()` method

There will be different form displayed based on the item's latest status such as Assign Item to station, Receive Item and etc.. Form will be displayed based on parameter *displayStatus*.

Return before and after receive is different. Figure 5.47 shows the usage of *displayStatus* variable. Box Interface section uses modal to allow users to input data upon deployment to station or return items.

Figure 5.47 & Figure 5.48 shows the difference between return form before receive and after receive.

```
<!--RECIEVE INTERFACE -->
<form v-on:submit.prevent="updateItemMovement" v-if="displayStatus === 'arrived'" data-aos="fade-right">...

<!--RETURN INTERFACE -->
<form v-on:submit.prevent="" v-if="displayStatus === 'return'" data-aos="fade-right">...
```

Figure 5.47 Usage of *displayStatus* variable

Click button Return before clicking Receive Item :- send directly to *returnItemMovement()* Method.

```
<!--RETURN MESSAGE BOX INTERFACE-->
<div class="modal fade" id="returnMessage" tabindex="-1" role="dialog" aria-labelledby="returnModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="returnModalLabel">Return Item</h5>
                <button type="button" class="close" data-dismiss="modal" v-on:click="closeMessageModal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <form v-on:submit.prevent="returnItemMovement">

                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Remark:</label>
                        <div class="col-sm-8">
                            <input type="text" class="form-control" v-model="remark" />
                        </div>
                    </div>

                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Consignment Note:</label>
                        <div class="col-sm-8">
                            <input type="file" class="form-control-file" v-on:change="handleFileUpload" accept="image/png, image/jpeg, application/pdf" />
                        </div>
                    </div>
                    <button type="submit" class="btn btn-primary">Return Item</button>
                </form>
            </div>
        </div>
    </div>
</div>
```

Figure 5.47 Return form before Receiving item

Return form after receiving item:- send to *receiveReturnAPI()* first then to *updateItemMovement(),* lastly *returnItemMovement()* Method.

```
<!--RECEIVE THEN RETURN INTERFACE -->
<div class="modal fade" id="returnModal" tabindex="-1" role="dialog" aria-labelledby="returnModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="returnModalLabel">Return Item</h5>
                <button type="button" class="close" data-dismiss="modal" v-on:click="closeModal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <form v-on:submit.prevent="receiveReturnAPI">

                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Remark:</label>
                        <div class="col-sm-8">
                            <input type="text" class="form-control" v-model="remark" />
                        </div>
                    </div>

                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Consignment Note:</label>
                        <div class="col-sm-8">
                            <input type="file" class="form-control-file" v-on:change="handleFileUpload" accept="image/png, image/jpeg, application/pdf" />
                        </div>
                    </div>
                    <button type="submit" class="btn btn-primary">Return Item</button>
                </form>
            </div>
        </div>
    </div>
</div>
```

Figure 5.48 Return form after receive item

Receive button will send the data to `updateItemMovement()`. Figure 5.49 shows the button of receive.

```
@* Submit and Reset Buttons *@
<div class="form-group row">
    <div class="col-sm-8 offset-sm-5">
        <button type="button" v-on:click="receiveReturnMessage" class="btn btn-secondary m-1">Return Item</button>
        <button type="submit" class="btn btn-primary m-1">Receive Item</button>
    </div>
</div>
```

Figure 5.49 Receive Buttons

The Deploy Station allows Technicians to assign items to their assigned station. The form will send data to `updateStationItemMovement()`. Figure 5.50 shows the deploy station form.

```
<!--STATION DEPLOY MESSAGE BOX INTERFACE-->
<div class="modal fade" id="stationMessage" tabindex="-1" role="dialog" aria-labelledby="stationModalLabel" aria-hidden="true">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="stationModalLabel">{{ thisItem?.currentStationId == null ? "Deploy Station" : "Change Station" }}</h5>
                <button type="button" class="close" data-dismiss="modal" v-on:click="closeStationMessageModal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                <form v-on:submit.prevent="updateStationItemMovement">

                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Deploy Station : </label>
                        <div class="col-sm-8">
                            <select class="btn btn-primary dropdown-toggle col-md-10" v-model="selectedStation">
                                <option class="btn-light" value="" disabled selected>Select Station</option>
                                <option v-if="stationList.length === 0" class="btn-light" disabled>No Station Assigned to You</option>
                                <option class="btn-light" v-for="(station, index) in stationList" :key="index" :value="station.stationId">{{ station.stationName}}</option>
                            </select>
                        </div>
                    </div>
                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Remark:</label>
                        <div class="col-sm-8">
                            <input type="text" class="form-control" v-model="remark" />
                        </div>
                    </div>

                    <div class="form-group row">
                        <label class="col-sm-4 col-form-label">Consignment Note:</label>
                        <div class="col-sm-8">
                            <input type="file" class="form-control-file" v-on:change="handleFileUpload" accept="image/png, image/jpeg, application/pdf" />
                        </div>
                    </div>
                    <button type="submit" class="btn btn-primary">Deploy Station</button>
                </form>
            </div>
        </div>
    </div>
</div>
```

Figure 5.50 Deploy Station Form

`updateStationItemMovement()` method will create new rows data. This method will be called from the Deploy to Station form. Figure 5.51 shows the `updateStationItemMovement()`.

```
async updateStationItemMovement() {
    const requiredFields = ['selectedStation'];

    for (let field of requiredFields) {
        if (!this[field]) {
            alert(`Request Error: Please fill in required field ${field}.`, 'warning');
            return;
        }
    }

    try {
        const now = new Date();
        const formData = {
            ItemId: this.thisItem.itemID,
            ToStation: this.thisItem.currentStationId,
            ToStore: this.thisItem.toStore,
            ToUser: this.currentUserId,
            ToOther: "Delivered",
            SendDate: new Date(now.getTime() + 8 * 60 * 60 * 1000).toISOString(),
            Action: "Assign",
            Quantity: this.thisItem.quantity,
            Remark: this.remark,
            ConsignmentNote: this.consignmentNote,
            Date: new Date(now.getTime() + 8 * 60 * 60 * 1000).toISOString(),
            LastUser: this.currentUserId,
            LastStore: this.thisItem.toStore,
            LastStation: this.selectedStation,
            LatestStatus: "Delivered",
            ReceiveDate: new Date(now.getTime() + 8 * 60 * 60 * 1000).toISOString(),
            MovementComplete: true,
        };

        const response = await fetch('/InvMainAPI/StationItemMovementUser', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(formData)
        });

        if (response.ok) {
            this.thisItem = await response.json();
            this.fetchItem(this.thisItem.uniqueID);
            alert('Success! Item assign to the Station.');
            $('#stationMessage').modal('hide');
            this.displayStatus = "return";
        } else {
            throw new Error('Failed to submit form.');
        }
    } catch (error) {
        console.error('Error:', error);
        alert('Inventory PSTW Error: An error occurred.');
    }
},
```

Figure 5.51 `updateStationItemMovement()`

`updateItemMovement()` method will send the received item movement data information to the API, the data received from by clicking the receive button. Figure 5.52 `updateItemMovement()` method. This method only updates the existing row that was used to send to Technician. (click button Return before click button Receive will do `updateItemMovement()` first then `returnItemMovement()`)

```
async updateItemMovement() {

    if (this.receiveReturn == null) {
        if (!confirm("Are you sure you already received this item?")) {
            return false;
        }
    }

    try {
        const now = new Date();
        const formData = {
            Id: this.thisItem.id,
            LastStore: this.thisItem.toStore,
            LatestStatus: "Delivered",
            ReceiveDate: new Date(now.getTime() + 8 * 60 * 60 * 1000).toISOString(),
            MovementComplete: true,
        };

        const response = await fetch('/InvMainAPI/UpdateItemMovementUser', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(formData)
        });

        if (response.ok) {
            if (this.receiveReturn == null) {
                alert('Success! Item has been successfully received.');
                this.thisItem = await response.json();
                this.displayStatus = "return";
                this.fetchItem(this.UniqueID);
                this.resetForm();
            } else {
                this.returnItemMovement();
            }
        } else {
            throw new Error('Failed to submit form.');
        }
    } catch (error) {
        console.error('Error:', error);
        alert('Inventory PSTW Error: An error occurred.');
    }
},
```

Figure 5.52 *updateItemMovement()*

`returnItemMovement()` method will send the return item movement data to the API, the data received from by clicking the return button. Figure 5.53 shows `returnItemMovement()` that will create new rows in *itemmovements* table

```
async returnItemMovement() {

    if (!confirm("Are you sure you want to return this item?")) {
        return false;
    }

    try {
        const now = new Date();
        const formData = {
            ItemId: this.thisItem.itemID,
            ToStation: this.thisItem.currentStationId,
            ToStore: this.thisItem.currentStoreId,
            ToUser: this.currentUserId,
            ToOther: "Return",
            SendDate: new Date(now.getTime() + 8 * 60 * 60 * 1000).toISOString(),
            Action: "StockIn",
            Quantity: this.thisItem.quantity,
            Remark: this.remark,
            ConsignmentNote: this.consignmentNote,
            Date: new Date(now.getTime() + 8 * 60 * 60 * 1000).toISOString(),
            LastUser: this.InventoryMasterId,
            LastStore: this.thisItem.toStore,
            LastStation: this.thisItem.toStation,
            LatestStatus: null,
            ReceiveDate: null,
            MovementComplete: false,
        };

        const response = await fetch('/InvMainAPI/ReturnItemMovementUser', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(formData)
        });

        if (response.ok) {
            alert('Success! Item is on the delivery to return to Inventory Master.');
            this.thisItem = await response.json();
            $('#returnModal').modal('hide');
            $('#returnMessage').modal('hide');
            this.displayStatus = "requestAgain";
            this.resetForm();
        } else {
            throw new Error('Failed to submit form.');
        }
    } catch (error) {
        console.error('Error:', error);
        alert('Inventory PSTW Error: An error occurred.');
    }
},
```

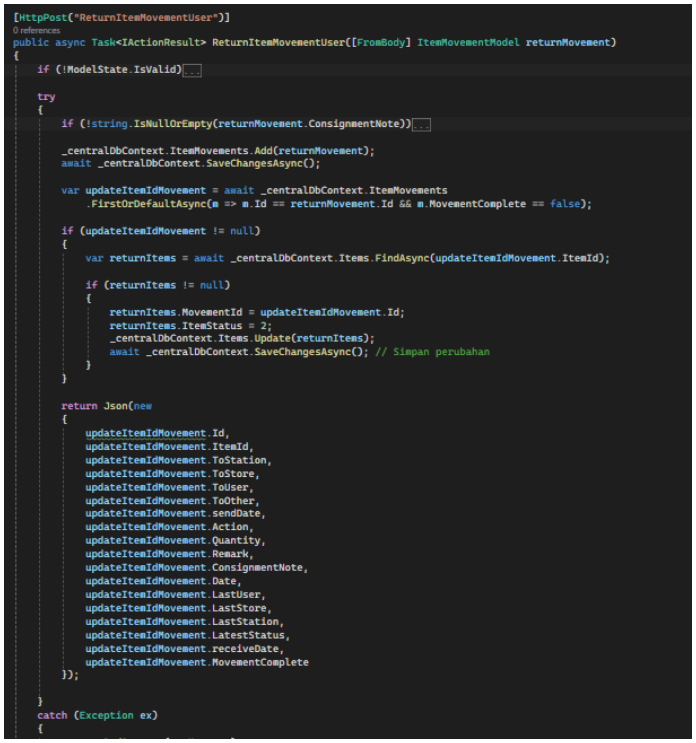
Figure 5.53 *returnItemMovement()*

/UpdateItemMovementUser API that will update the data in *itemmovements* table from Inventory Master to Technician. Figure 5.54 shows */UpdateItemMovementUser* API.

```
#region ScannerUser
[HttpPost("UpdateItemMovementUser")]
0 references
public async Task<IActionResult> UpdateItemMovementUser([FromBody] ItemMovementModel receiveMovement)
{

    try
    {

        var updatedList = await _centralDbContext.ItemMovements.FindAsync(receiveMovement.Id);

        if (updatedList == null)
        {
            return NotFound("Item movement record not found.");
        }
        updatedList.MovementComplete = receiveMovement.MovementComplete;
        updatedList.LatestStatus = receiveMovement.LatestStatus;
        updatedList.receiveDate = receiveMovement.receiveDate;
        updatedList.MovementComplete = receiveMovement.MovementComplete;

        _centralDbContext.ItemMovements.Update(updatedList);
        await _centralDbContext.SaveChangesAsync();

        var receiveItems = await _centralDbContext.Items.FindAsync(receiveMovement.ItemId);

        if (receiveItems != null)
        {
            receiveItems.ItemStatus = 3;
            _centralDbContext.Items.Update(receiveItems);

        }

        await _centralDbContext.SaveChangesAsync();

        return Json(updatedList);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
}
```

Figure 5.54 UpdateItemMovementUser API

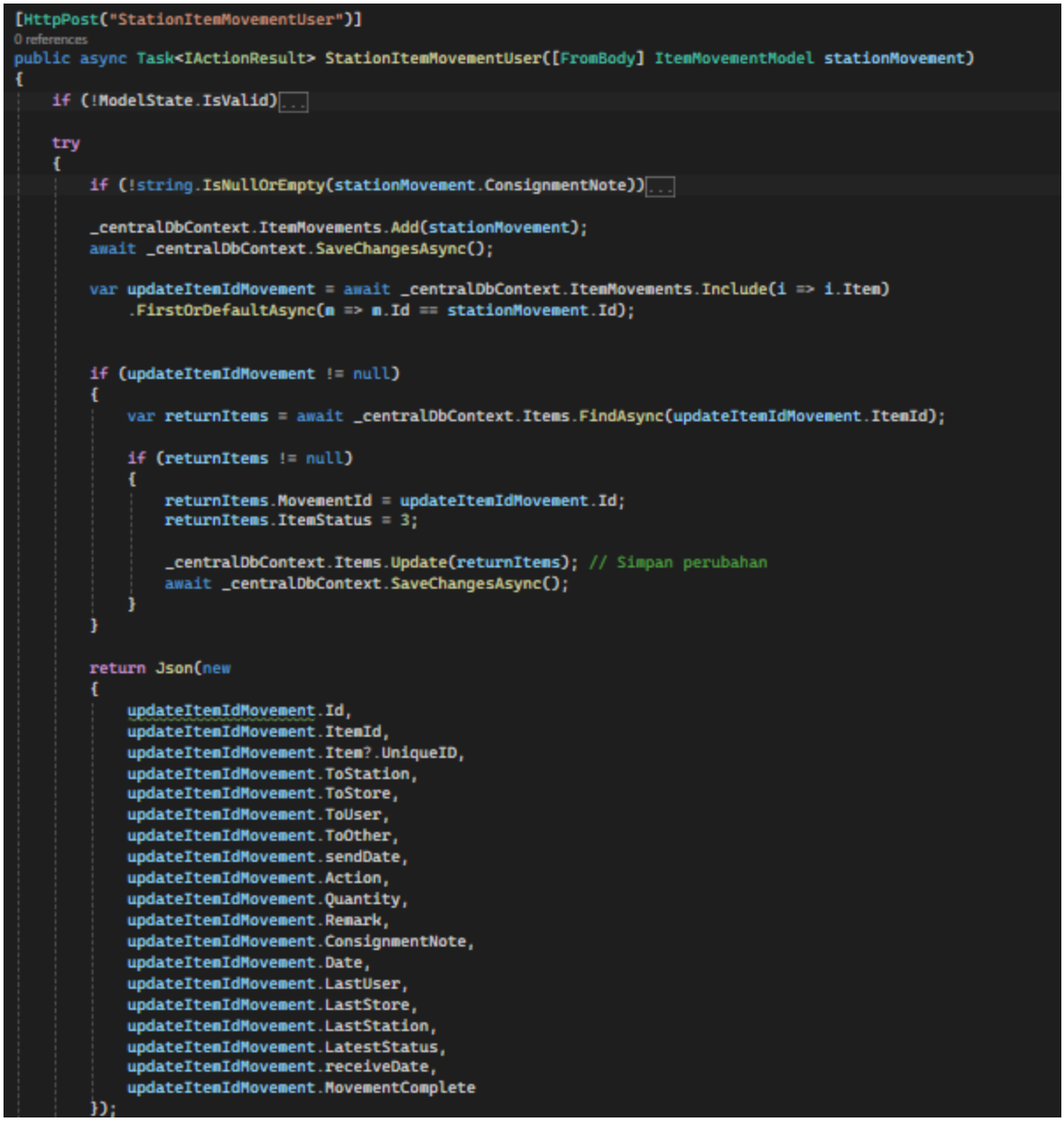
/ReturnItemMovementUser API used for creating new data for return movement from Technician to Inventory Master. Figure 5.55 shows */ReturnItemMovementUser* API.



```
[HttpPost("ReturnItemMovementUser")]
0 references
public async Task<IActionResult> ReturnItemMovementUser([FromBody] ItemMovementModel returnMovement)
{
    if (!ModelState.IsValid)...

    try
    {
        if (!string.IsNullOrEmpty(returnMovement.ConsignmentNote))...

        _centralDbContext.ItemMovements.Add(returnMovement);
        await _centralDbContext.SaveChangesAsync();

        var updateItemIdMovement = await _centralDbContext.ItemMovements
            .FirstOrDefaultAsync(m => m.Id == returnMovement.Id && m.MovementComplete == false);

        if (updateItemIdMovement != null)
        {
            var returnItems = await _centralDbContext.Items.FindAsync(updateItemIdMovement.ItemId);

            if (returnItems != null)
            {
                returnItems.MovementId = updateItemIdMovement.Id;
                returnItems.ItemStatus = 2;
                _centralDbContext.Items.Update(returnItems);
                await _centralDbContext.SaveChangesAsync(); // Simpan perubahan
            }
        }

        return Json(new
        {
            updateItemIdMovement.Id,
            updateItemIdMovement.ItemId,
            updateItemIdMovement.ToStation,
            updateItemIdMovement.ToStore,
            updateItemIdMovement.ToUser,
            updateItemIdMovement.ToOther,
            updateItemIdMovement.sendDate,
            updateItemIdMovement.Action,
            updateItemIdMovement.Quantity,
            updateItemIdMovement.Remark,
            updateItemIdMovement.ConsignmentNote,
            updateItemIdMovement.Date,
            updateItemIdMovement.LastUser,
            updateItemIdMovement.LastStore,
            updateItemIdMovement.LastStation,
            updateItemIdMovement.LatestStatus,
            updateItemIdMovement.receiveDate,
            updateItemIdMovement.MovementComplete
        });

    }
    catch (Exception ex)
    {
```

Figure 5.55 ReturnItemMovementUser API

/StationItemMovementUser API will create new rows for itemMovement for item. This API will purpose either from self-assign to station, station to another station. Figure 5.56 shows the */StationItemMovementUser* API.



```
[HttpPost("StationItemMovementUser")]
0 references
public async Task<IActionResult> StationItemMovementUser([FromBody] ItemMovementModel stationMovement)
{
    if (!ModelState.IsValid)...

    try
    {
        if (!string.IsNullOrEmpty(stationMovement.ConsignmentNote))...

        _centralDbContext.ItemMovements.Add(stationMovement);
        await _centralDbContext.SaveChangesAsync();

        var updateItemIdMovement = await _centralDbContext.ItemMovements.Include(i => i.Item)
            .FirstOrDefaultAsync(m => m.Id == stationMovement.Id);


        if (updateItemIdMovement != null)
        {
            var returnItems = await _centralDbContext.Items.FindAsync(updateItemIdMovement.ItemId);

            if (returnItems != null)
            {
                returnItems.MovementId = updateItemIdMovement.Id;
                returnItems.ItemStatus = 3;

                _centralDbContext.Items.Update(returnItems); // Simpan perubahan
                await _centralDbContext.SaveChangesAsync();
            }
        }

        return Json(new
        {
            updateItemIdMovement.Id,
            updateItemIdMovement.ItemId,
            updateItemIdMovement.Item?.UniqueID,
            updateItemIdMovement.ToStation,
            updateItemIdMovement.ToStore,
            updateItemIdMovement.ToUser,
            updateItemIdMovement.ToOther,
            updateItemIdMovement.sendDate,
            updateItemIdMovement.Action,
            updateItemIdMovement.Quantity,
            updateItemIdMovement.Remark,
            updateItemIdMovement.ConsignmentNote,
            updateItemIdMovement.Date,
            updateItemIdMovement.LastUser,
            updateItemIdMovement.LastStore,
            updateItemIdMovement.LastStation,
            updateItemIdMovement.LatestStatus,
            updateItemIdMovement.receiveDate,
            updateItemIdMovement.MovementComplete
        });
```

Figure 5.56 */StationItemMovementUser* API

References

1. Gruhn. (n.d.). *GitHub - gruhn/vue-qrcode-reader: A set of Vue.js components for detecting and decoding QR codes.* GitHub. https://github.com/gruhn/vue-qrcode-reader