

# Test: IDF Procurement API (NestJS)

## Story

צה"ל רוצה מערכת לניהול רכש, שמאפשרת לספקים לשלוח בקשות מכירה לצה"ל או קנייה מצה"ל.

צה"ל מבקש מחיילי 8200 לבנות מערכת כזו.

במערכת יש DB לבחירתכם שמאחסן את כל הפריטים שיש לצה"ל באותה טבלה/COLLECTION.

לצה"ל יש תקציב **שנשמר בקובץ**, ומולו בודקים האם יש לצה"ל כסף לקנות מוצר מסוים, ואילו מוסיפים כסף אחרי שפריטים נמכרים.

אתם צריכים ליצור שרת שמאפשר לספק למכור פריטים לצה"ל (בתנאי שיש לצה"ל מספיק כסף בתקציב) ולקנות פריטים מצה"ל (ולהוסיף את הכסף החדש לתקציב).

אם צה"ל קונה פריטים - צריך לבדוק בDB אם יש כבר פריטים כאלה. אם יש - מעדכנים את הכמות. אם אין - יוצרים רשומה חדשה בטבלת/אוסף הפריטים.

אם מוכרים (**בונוס**): יש לוודא שכמות הפריטים שהספק מבקש לקנות לא מורידה את כמות הפריטים השמורה אל מתחת ל0, והפריט המבוקש קיים. אם הכל בסדר - מעדכנים את התקציב ואת כמות הפריטים החדשה אחרי שנמכרו X פריטים בDB.

---

## Mandatory Technical Rules

1. Use **NestJS**.
  2. Create txt File and load the **budget from the file** when the server starts - you need to load it and write to it using **fs**
  3. How can You run something when the app loads? Read [here](#)
  4. Store **items in a database** (any DB is allowed - including fs based).
  5. Do not create endpoints for listing or reading items.
-

# Data

## Budget (from file)

- `currentBudget: number`
- The budget must **never go below 0**.

---

## Item (in DB) - create a table/collection

Each item has:

Field	Type	Notes
<code>id</code>	string or number	unique
<code>name</code>	string	required
<code>type</code>	string	required
<code>quantity</code>	number	whole number, $\geq 0$
<code>pricePerUnit</code>	number	$> 0$
<code>hasImage</code>	boolean	starts as <code>false</code>

---

# Business Rules

## Buying items

- One request can buy **many items**
  - For each item:
    - $\text{cost} = \text{quantity} \times \text{pricePerUnit}$
    - Reduce the budget by **cost**
    - check item record in DB:
      - if does not exists - create it with the current quantity
      - if exists: updated the quantity
  - If the budget would become **negative** as a result of buying this quantity of items at this pricePerUnit value:
    - Reject the whole request
    - Do not change anything
    - Log Error
-

# Required Endpoints (mandatory)

---

## 1) Purchase items

**POST** /transactions/purchase

### What it does

Buy equipment and reduce the budget.

### Request body

```
{
  "purchases": [
    {
      "id": "A1",
      "name": "Night Vision",
      "type": "optics",
      "quantity": 3,
      "pricePerUnit": 2000
    },
    {}...
  ]
}
```

### Rules

- All fields are required
- If the item exists in DB → increase quantity
- If it does not exist → create it
- **hasImage** must start as **false**
- Reject and change nothing if budget goes below 0 because of this purchase
- This request can include more than one object to buy

### Response

```
{
  "results": [
    { "id": "A1", "newQuantity": 5, "spent": 6000 }
  ]
}
```

```
}
```

### 3) Check image

**POST** `/images/check/:itemId`

`multipart/form-data`

Field name: `image`

#### What it does

Checks if an image is safe.

#### Rules

- Max size: **250 MB**  
Type: **PNG only**

In order to learn how to upload a file and how to validate its size and type, you must read and understand [the documentation](#)

- **Response**

```
{  
  "itemId": "A1",  
  "isValid": true,  
  "reason": null  
}
```

If not valid:

```
{  
  "itemId": "A1",  
  "isValid": false,  
  "reason": "File is too large"  
}
```

Area	Points
Project setup + Budget file (fs: TXT or JSON)	15
DB / Storage for Items	25
POST /transactions/purchase	40
POST /images/check/:itemId	10
Validation (request DTOs) + Business constraints	5
Clean Code	5

## Bonus Features

### Bonus 1) Set image to item

POST `/images/set/:itemId`

#### What it does

- Upload an image
- Validate it
- If valid:
  - Set `hasImage = true`

---

### Bonus 2) Store image in DB

When setting the image:

- Save the image data in the database
  - Link it to the item
- 

## Selling items

- One request can sell **many items**
- You can only sell quantity that is equal or smaller then the stored amount in the db for this item.
- Refund calculation:
  - `money = quantity × pricePerUnit`
  - `reduce 10% of the money - because it's used.`
- Add “money” to the budget - after the 10% reduction
- Reduce item quantity from the one stored in the DB
- If any item is invalid:
  - Reject the whole request
  - Log error



## 2) Sell items

**POST** /transactions/sell

### What it does

Sell equipment and return money to the budget.

### Request body

```
{
  "sales": [
    { "id": "A1", "quantity": 2 },
    { "id": "BC12", "quantity": 15 }
  ]
}
```

### Rules

- Item must exist
- Cannot sell more than stock in DB
- Refund uses 10% reduction of price before adding it to the budget

### Response

```
{  
  "results": [  
    { "id": "A1", "newQuantity": 3 }  
  ]  
}
```

---