



Vending Machine System

Full-Stack Coding Assessment Guide

React 18 + TypeScript + .NET 8 + Entity Framework Core

Assessment Overview

Expected Duration: 90 Minutes

- **Backend (.NET 8):** 45 minutes
- **Frontend (React + TypeScript):** 45 minutes

This is an estimate based on a mid-level developer completing all core tasks. Strong candidates may finish earlier; struggling with basic concepts indicates a mismatch for the role.

What You're Building

A simplified vending machine interface where users can:

- View available products with stock levels
- Start the machine to get a random balance (\$1-\$10)
- Purchase products with quantity selection
- View purchase history with filtering capabilities

Setup Instructions

Prerequisites

- **Windows:** Visual Studio 2022 with .NET 8 SDK
- **Mac/Linux:** .NET 8 SDK + VS Code (or any editor)
- **Node.js:** Version 18+ (check with `node --version`)
- **npm:** Comes with Node.js (check with `npm --version`)

Option 1: Visual Studio (Recommended for Windows)

1. Navigate to backend folder
2. Double-click VendingMachine.sln
3. Press F5 to run
 - Backend starts at `http://localhost:5000`
 - Swagger opens automatically at `http://localhost:5000/swagger`
 - Database auto-creates with sample data
4. Open separate terminal:

```
cd frontend
npm install
npm run dev
```

 - Frontend opens at `http://localhost:5173`
5. Visit `http://localhost:5173` in browser

Option 2: Command Line (Mac/Linux/Windows)

```
# Terminal 1 - Backend
cd backend
dotnet restore
dotnet run
# Backend API: http://localhost:5000

# Terminal 2 - Frontend
cd frontend
npm install
npm run dev
# Frontend: http://localhost:5173
```

Tasks & TODO List

Backend Tasks (ProductsController.cs)

[TODO 1]: Get All Products

Return all products from the database.

```
// GET: api/products
[HttpGet]
public async Task

Expected: Use Entity Framework to return all products. Should be 1-2 lines of code.


```

[TODO 2]: Implement Purchase Logic

Implement complete purchase workflow with validation:

1. Validate request payload (product ID exists)
2. Calculate updated stock accordingly
3. Create Purchase record and save to database
4. Return success response

```
// POST: api/products/purchase
[HttpPost("purchase")]
public async Task

API Expected to throw errors when:



- a. Missing request payload (400 Bad Request)
- b. Product not found (404 Not Found with proper message)
- c. Out of stock (400 Bad Request with proper message)



Note: You can add helper functions/variables inside the controller class if needed.


```

[BONUS TODO 3]: Prevent Duplicate Purchase Within 5 Seconds **BONUS**

Prevent calling the purchase API again within 5 seconds of the last purchase (to simulate real machine behavior - preventing button mashing).

Hint: Track last purchase timestamp per user/session. You can add instance variables to the controller.

[TODO 4]: Get Purchase History

Return all purchases from the database.

```
// GET: api/products/purchases
[HttpGet("purchases")]
public async Task
```

[BONUS TODO 5]: Server-Side Filtering & Sorting **BONUS**

Support server-side filtering using the `PurchaseFilterParams` class:

- `searchTerm` : Filter by product name (case-insensitive)
- `machineId` : Filter by specific machine
- `hours` : Filter by time range (e.g., 24 = last 24 hours)
- `sortField` : Sort by "date", "amount", or "product"
- `sortOrder` : "asc" or "desc"

```
// Example query:
// GET /api/products/purchases?searchTerm=coke&hours=24&sortField=amount&sortOrder=desc
```

Hint: Use LINQ's `.Where()`, `.OrderBy()`, `.OrderByDescending()`.

Frontend Tasks

[TODO 1]: Complete buyProduct and updateQuantity Functions

Located in `ProductsPage.tsx`

Part A: updateQuantity Function

```
function updateQuantity(productId: string, newQty: number) {
    // [TODO 1]: Find correct product and update quantity in state
}
```

Update the `quantities` state with the new quantity for the given product.

Part B: buyProduct Function - Balance Checking

```
async function buyProduct(productId: string) {
    // [TODO 1]: Check insufficient balance before purchase
    // Get the quantity for this product: const quantity = quantities[productId] || 1
    // Calculate cost: const cost = product.price * quantity
    // Add checking if conditions with proper error messages:
    // 1. balance == null -> return and show error "X Please start the machine first"
    // 2. balance < cost -> return and show error "X Insufficient funds! You have $X.XX but need $Y.YY"
    // 3. product not found -> return (already handled above)
    // 4. out of stock -> return (already handled above)
    // 5. quantity is 0 or invalid -> return and show error
}
```

Part C: Post-Purchase Actions

```
if (res.data.success) {
    setMessage(✓ Purchase successful! ${res.data.remaining || 0} items remaining.);
    // [TODO 1]: After successful purchase:
    // 1. Reload products using loadProducts() to get updated stock
    // 2. Update balance: setBalance(balance - res.data.totalCost) to deduct purchase cost
    // 3. Reset quantity for this product: setQuantities([...quantities, {productId: 1}])
}
```

Part D: Send Quantity in API Request

```
// [TODO 1]: Get quantity from state and send it in the request
const quantity = quantities[productId] || 1;
const res = await api.post<PurchaseResponse>('/products/purchase', { productId, quantity });
```

[TODO 2]: Add UI Controls for Out of Stock

Located in `ProductCard.tsx`

Update the "Buy Now" button to be disabled when:

- Balance is not loaded (already done as example)
- Product is out of stock – Show text: "X Out of Stock"
- Already purchasing – Show text: "🔄 Processing..."

Apply appropriate UI styles: grey background, grey text, and remove hover effects when disabled.

[BONUS TODO 3]: Apply UI Styles for Low Stock **BONUS**

Located in `ProductCard.tsx`

When product stock is low (≤ 2 items), show a "Low Stock" badge and apply warning colors to the stock display.

Hint: See commented example in the file around lines 39-45.

[TODO 4]: Complete Purchase History Table

Located in `PurchaseHistoryTable.tsx` and `PurchasePage.tsx`

Part A: Table Headers & Data Mapping

Add table headers and map purchase data to table rows showing:

- Product name (with emoji icon)
- Amount paid
- Purchase time (formatted)

Part B: Search Product Functionality

```
// In PurchasePage.tsx
const filteredAndSortedPurchases = useMemo(() => {
    let filtered = [...purchases];

    if (searchProduct) {
        // TODO 4: client-side filter by product name
    }

    return filtered;
}, [purchases, searchProduct]);
```

[BONUS TODO 5]: Server-Side Filtering & Sorting **BONUS**

Located in `PurchasePage.tsx`

Instead of client-side filtering, send filter parameters to the backend API:

```
// BONUS TODO 5: You can add backend filtering here
// The backend accepts PurchaseFilterParams with these query parameters:
// - hours: number
// - sortField: 'date' | 'amount' | 'product'
// - sortOrder: 'asc' | 'desc'
//
// Example with all filters:
const params = new URLSearchParams({
    hours: dateFilter === '24h' ? '24' : dateFilter === '7d' ? '168' : '',
    sortField: sortField,
    sortOrder: sortOrder
});
// api.get('/products/purchases?s=${params.toString()}')
```

Requires: Completing Backend [BONUS TODO 5] first.

[BONUS TODO 6]: Global UI Checking for Connectivity **BONUS**

Add global UI indicators across all pages that:

- Show connection status to backend (online/offline)
- Automatically retry on connection failure
- Display user-friendly offline message
- Update UI when connection is restored

Hint: Use React Context or create a custom hook to track API health.

How the Vending Machine Works

1. Starting the Machine

- Click "🎰 Start Machine" button
- Backend generates random balance between \$1.00 and \$10.00
- Balance displays in green badge
- All products become available for purchase

2. Purchasing Products

1. User selects quantity using +/- buttons (1 to stock limit)
2. Frontend checks if balance is sufficient
3. API call sent to backend with product ID and quantity
4. Backend waits 5 seconds (simulates machine processing)
5. Backend validates, decrements stock, creates purchase record
6. Frontend receives response and updates UI
7. Balance is deducted, product list refreshes

3. Purchase History

- All purchases are stored in SQLite database
- Purchase history page loads all transactions
- Users can filter by product name, date range
- Users can sort by date, amount, or product name

4. Machine Behavior Simulation

- Thread.Sleep(5000)** in the Purchase endpoint:
- Simulates real vending machine processing time
 - Tests frontend loading states and user feedback
 - Used for [BONUS TODO 3] to prevent rapid duplicate purchases

Submission Method

Candidate Workflow

- ```
1. Fork or clone the repository
git clone https://github.com/your-org/ivm-assessment.git
cd ivm-assessment

2. Create a new branch with your name
git checkout -b candidate/john-doe

3. Work on the assessment...

4. Commit your changes
git add .
git commit -m "Complete vending machine assessment - John Doe"

5. Push to your fork/branch
git push origin candidate/john-doe

6. Create Pull Request on GitHub
```

### Alternative: Email/ZIP Submission

If GitHub is not suitable, candidates can:

1. Complete the assessment locally
2. Exclude `node_modules`, `bin`, `obj` folders
3. ZIP the project folder
4. Email to designated address with naming: `FirstName.LastName.VendingMachine.zip`

### Viewing & Comparing Submissions

**GitHub Benefits:**

- ✓ View all submissions as Pull Requests
- ✓ See diff/changes for each candidate
- ✓ Add inline code review comments
- ✓ Track completion time via commit timestamps
- ✓ Easy to share with other evaluators
- ✓ Can run automated tests on PRs (optional)

**Reviewing a PR:**

1. Open Pull Request tab on GitHub
2. Click on candidate's PR
3. Go to "Files changed" tab
4. Review only the files they were supposed to modify:
  - `backend/Controllers/ProductsController.cs`
  - `frontend/src/pages/ProductsPage.tsx`
  - `frontend/src/pages/PurchasePage.tsx`
  - `frontend/src/components/ProductCard.tsx`
  - `frontend/src/components/PurchaseHistoryTable.tsx`
5. Add inline comments or approve/request changes

**Important Notes**

**Database Auto-Creation**

- SQLite database ( `vending.db` ) auto-creates on first run
- Pre-seeded with 6 products and 2 categories
- To reset database: Delete `vending.db` and restart backend
- No migrations needed - candidates just write queries

**CORS & Proxy Configuration**

- Backend must run on `http://localhost:5000`
- Frontend runs on `http://localhost:5173`
- Vite proxy configured to forward `/api/*` requests to backend
- CORS is already configured - candidates don't need to modify

## ? FAQ

### Can I use external libraries?

For core tasks, no. Everything needed is already installed. For bonus features, you may use lightweight utilities (like lodash) if it makes sense, but document why.

### What if I can't finish in 90 minutes?

Focus on core TODOs first. Quality partial completion > rushed full completion. We'd rather see 70% done well than 100% done poorly.

### Can I modify the database models?

No. Work with the existing models. If you think something is missing, add a comment explaining what you would add in a real scenario.

### Should I write tests?

Not required for this assessment. Focus on implementation. However, if you finish early and want to showcase testing skills, it's a bonus.

### What if the backend doesn't start?

- Check .NET 8 SDK is installed: `dotnet --version`
  - Try `dotnet restore` in backend folder
  - Check port 5000 isn't already in use
  - Delete `bin` and `obj` folders and try again
- What if frontend can't connect to backend?**
- Verify backend is running at `http://localhost:5000`
  - Check Swagger is accessible at `http://localhost:5000/swagger`
  - Restart frontend dev server
  - Check browser console for CORS errors