



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
```

💡 What's Already Set Up For You

- SQLite database with Entity Framework Core
- 6 pre-Seeded products (Coke, Pepsi, Water, Chips, Chocolate, Cookies)
- CORS configured for localhost:5173
- Swagger UI for API testing
- Axios client configured with proxy
- Tailwind CSS with custom utility classes
- TypeScript types for all models
- React Router setup with navigation
- Beautiful, responsive UI with gradient designs

You only need to implement the business logic!

🎯 Tasks & TODO List

Backend Tasks (`ProductsController.cs`)

1 [TODO 1]: Get All Products

Return all products from the database.

```
// GET: api/products
[HttpGet]
public async Task<ActionResult<IEnumerable<Product>>> GetProducts() {
    // [TODO 1]: here
    throw new NotImplementedException();
}
```

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

2 [TODO 2]: Implement Purchase Logic

Implement complete purchase workflow with validation:

```
// POST: api/products/purchase
[HttpPost("purchase")]
public async Task<ActionResult<PurchaseResponse>> Purchase([FromBody] PurchaseRequest? request) {
    Thread.Sleep(5000); // simulates machine processing time
    // [TODO 2]: here
    throw new NotImplementedException();
}
```

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.

3 [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.

4 [TODO 4]: Get Purchase History

Return all purchases from the database.

```
// GET: api/products/purchases
[HttpGet("purchases")]
public async Task<ActionResult<IEnumerable<Purchase>>> GetPurchases([FromQuery] PurchaseFilterParams? filter) {
    // [TODO 4]: here
    throw new NotImplementedException();
}
```

5 [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

1 [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 current balance 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 "Please start the machine first"
    // 2. balance < cost => return and show error "Insufficient funds! You have $Y.YY but need $Y.YY"
    // 3. product not found => return (already handled above)
    // 4. out of stock => return (already handled above)
    // 5. quantity is 0 => 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 });
```

2 [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: "Out of Stock"
- Already purchasing → Show text: "Processing..."

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

3 [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.

4 [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]);
```

5 [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?'+params.toString())
```

Requires: Completing Backend [BONUS TODO 5] first.

6 [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

- User selects quantity using +/- buttons (1 to stock limit)
- Frontend checks if balance is sufficient
- API call sent to backend with product ID and quantity
- Backend validates, decrements stock, creates purchase record
- Frontend receives response and updates UI
- 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

5. 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...
git add .
```

```
git commit -m "Complete vending machine assessment - John Doe"
```

```
# 4. Push your fork/branch
git push origin candidate/john-doe
```

```
# 5. Create Pull Request on GitHub
```

Alternative: Email/ZIP Submission

If GitHub is not suitable, candidates can:

- Complete the assessment locally
- Exclude `node_modules`, `bin`, `obj` folders
- ZIP the project folder
- 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:

- Open Pull Request tab on GitHub

- Click candidate's PR

- Go to "Files changed" tab

- Review only the files they were supposed to modify:

```
o backend/controllers/ProductsController.cs
```

```
o frontend/src/pages/ProductsPage.tsx
```

```
o frontend/src/pages/PurchasePage.tsx
```

```
o frontend/src/components/ProductCard.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.