

LAB TEST- 04

Name: D. Sravika Reddy

Roll No: 2403a510d0

Batch: 05

Course: AI Assisted Coding

Q1. An old inventory system is written in PHP and must be migrated to Node.js.

- a) Translate a sample function using AI-assisted prompts.
- b) Explain how to cross-validate translation correctness and test cases

#Prompt:

Translate this PHP getInventoryValue function to Node.js
(CommonJS). Requirements:

- Export getInventoryValue(items).
- Validate input: items is an array of objects with keys sku (string), price (non-negative number), quantity (non-negative integer); throw clear errors for invalid input.
- Include a small CLI/test harness: deterministic sample and randomized cross-check tests; no external dependencies.
- Keep code short, dependency-free, and runnable with node. Return only the Node.js code.

Code:

```

code) > babel.js > ...
1  'use strict';
2  /**
3   * inventory.js
4   * getInventoryValue(items) -> number
5   * Simple, validated translation of the PHP function.
6   */
7
8  function isObject(v) {
9      return v !== null && typeof v === 'object' && !Array.isArray(v);
10 }
11
12 /**
13  * Compute total inventory value.
14  * @param {Array<{sku:string, price:number, quantity:number}>} items
15  * @returns {number}
16  */
17 function getInventoryValue(items) {
18     if (!Array.isArray(items)) throw new TypeError('items must be an array');
19     let total = 0;
20     for (const [i, it] of items.entries()) {
21         if (!isObject(it)) throw new TypeError(`item[${i}] must be an object`);
22         const { sku, price, quantity } = it;
23         if (typeof sku !== 'string') throw new TypeError(`item[${i}].sku must be a string`);
24         if (typeof price !== 'number' || !Number.isFinite(price) || price < 0)
25             throw new TypeError(`item[${i}].price must be a non-negative number`);
26         if (!Number.isInteger(quantity) || quantity < 0)
27             throw new TypeError(`item[${i}].quantity must be a non-negative integer`);
28         total += price * quantity;
29     }
30     return total;
31 }
32
33 module.exports = { getInventoryValue };
34
35 // Self-test when run directly
36 if (require.main === module) [
37     const assert = require('assert');
38
39     const sample = [
40         { sku: 'A1', price: 10.5, quantity: 3 },
41         { sku: 'B2', price: 0.0, quantity: 100 },
42         { sku: 'C3', price: 2.25, quantity: 4 },
43     ];
44     console.log('sample items:', sample);
45     const tv = getInventoryValue(sample);
46     console.log('Total value:', tv.toFixed(2));
47     assert.strictEqual(tv, 10.5*3 + 0*100 + 2.25*4);
48
49     // randomized cross-check against a reference JS implementation
50     function reference(items) {
51         return items.reduce((s, it) => s + it.price * it.quantity, 0);
52     }
53     for (let seed=1; seed<=20; seed++) {
54         const rnd = (n) => Math.floor((seed * 9301 + 49297) % 233280) / 233280 * n;
55         const gen = () => {
56             const n = rnd(6);
57             const arr = [];
58             for (let i=0;i<n;i++) {
59                 arr.push({ sku: `S${i}` , price: parseFloat((rnd(1000)/100).toFixed(2)), quantity: rnd(10) });
60             }
61             return arr;
62         };
63         const items = gen();
64         const a = getInventoryValue(items);
65         const b = reference(items);
66         assert.strictEqual(a, b);
67     }
68     console.log('All tests passed.');
69 ]

```

Output:

```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> cd code
PS C:\Users\sravi\OneDrive\Desktop\AIAC\code> node lab04.js
PS C:\Users\sravi\OneDrive\Desktop\AIAC\code> node lab04.js
Sample items: [
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'B2', price: 0, quantity: 100 },
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'A1', price: 10.5, quantity: 3 },
  { sku: 'B2', price: 0, quantity: 100 },
  { sku: 'C3', price: 2.25, quantity: 4 }
]
Total value: 40.50
All tests passed.
PS C:\Users\sravi\OneDrive\Desktop\AIAC\code>
```

Observations:

- Unit tests: run the Node.js tests above and compare outputs with the original PHP on same fixed inputs.
- Randomized tests: generate identical random inputs in both PHP and Node and compare totals.
- Edge cases: test empty array, zero prices/quantities, very large numbers, and invalid types to ensure both implementations raise expected errors.
- Integration tests: plug Node module into the pipeline and compare aggregated reports against legacy system for a sample dataset.
- Code review & static checks: ensure input validation mirrors PHP assumptions; add type annotations/comments for clarity.

Q2. A company wants AI to convert SQL queries into ORM-based syntax.

- a) Write example transformation: SQL → Django ORM.
- b) State 3 limitations of automated code translation

Prompt:

Translate the SQL in my Django app file (sqltodjango.py) into a concise, runnable Django ORM snippet.

Assume a Product model with fields: id, name, price.

Return: a values()-based QuerySet (or model-based alternative), a one-line comment of the original SQL, and a short note listing any assumptions. Keep output minimal and ready to paste into the app.

Code:

```
⚡ sqltodjango.py > ...
1  from django.db.models import F
2  from .models import Product
3
4  # Django ORM equivalent of:
5  # SELECT id, name, price
6  # FROM products
7  # WHERE price > 100
8  # ORDER BY price DESC;
9
10 # Option 1: Using values() to return dictionaries
11 products = Product.objects.filter(
12     price__gt=100
13 ).values('id', 'name', 'price').order_by('-price')
14
15 # Option 2: Using model instances instead of dictionaries
16 # (remove .values() call if you prefer this approach)
17 # products = Product.objects.filter(
18 #     price__gt=100
19 # ).order_by('-price')
20
21 # Iterating through results
22 for product in products:
23     print(product['id'], product['name'], product['price'])
24
```

```

❷ sqltodjango_demo.py > ...
1 """
2     Django ORM Conversion Demo
3     SQL to Django ORM Translation
4 """
5
6     # This demonstrates the Django ORM syntax equivalent to the SQL query
7     # Original SQL:
8     # SELECT id, name, price
9     # FROM products
10    # WHERE price > 100
11    # ORDER BY price DESC;
12
13    # Django ORM Equivalent:
14    # =====
15
16    print("=" * 60)
17    print("SQL to Django ORM Conversion")
18    print("=" * 60)
19
20    print("\n❸ ORIGINAL SQL QUERY:")
21    print("-" * 60)
22    sql_query = """
23        SELECT id, name, price
24        FROM products
25        WHERE price > 100
26        ORDER BY price DESC;
27    """
28    print(sql_query)
29
30    print("\n❹ DJANGO ORM EQUIVALENT:")
31    print("-" * 60)
32    django_orm = """
33        from django.db.models import F

```

```

35
36    # Option 1: Using values() to return dictionaries
37    products = Product.objects.filter(
38        |   | price__gt=100
39    ).values('id', 'name', 'price').order_by('-price')
40
41    # Option 2: Using model instances
42    products = Product.objects.filter(
43        |   | price__gt=100
44    ).order_by('-price')
45    """
46
47    print(django_orm)
48
49    print("\n❺ BREAKDOWN:")
50    print("-" * 60)
51    print("""
52        1. Product.objects.filter(price__gt=100)
53            |   Equivalent to: WHERE price > 100
54
55        2. .values('id', 'name', 'price')
56            |   Equivalent to: SELECT id, name, price
57
58        3. .order_by('-price')
59            |   Equivalent to: ORDER BY price DESC
60            |   (The '-' prefix indicates DESC order)
61    """)
62
63    print("\n❻ Conversion Complete!")
64    print("=" * 60)

```

Output:

```
=====
SQL to Django ORM Conversion
=====

💡 ORIGINAL SQL QUERY:
-----
SELECT id, name, price
FROM products
WHERE price > 100
ORDER BY price DESC;

💡 DJANGO ORM EQUIVALENT:
-----
from django.db.models import F
from .models import Product

# Option 1: Using values() to return dictionaries
products = Product.objects.filter(
    price__gt=100
).values('id', 'name', 'price').order_by('-price')

# Option 2: Using model instances
products = Product.objects.filter(
    price__gt=100
).order_by('-price')
```

```
💡 BREAKDOWN:
-----
1. Product.objects.filter(price__gt=100)
   └ Equivalent to: WHERE price > 100

2. .values('id', 'name', 'price')
   └ Equivalent to: SELECT id, name, price

3. .order_by('-price')
   └ Equivalent to: ORDER BY price DESC
   └ (The '--' prefix indicates DESC order)

✅ Conversion Complete!
└ (The '--' prefix indicates DESC order)
✿

✅ Conversion Complete!
=====
PS C:\Users\likhi\OneDrive\Desktop\AI_labexam4>
```

Observation:

- Model/schema assumptions: Automated translation must guess model names, field names, and related_name settings – manual adjustment is usually required.
- SQL features mismatch: Advanced SQL (window functions, recursive CTEs, lateral joins, vendor-specific JSON/FULL-TEXT ops) often lack direct ORM equivalents and need raw SQL or custom annotations.
- Performance & semantics: ORM code can change query plans and cause N+1 queries or different locking/transaction behavior – review and add select_related/prefetch_related, indexes, and explicit transactions as needed.