VVS02 - WebApp Integration Tests Report

This report details the comprehensive integration testing strategy implemented to verify the correct interaction between different components of a web application. Three complementary testing approaches were used:

- HtmlUnit for end-to-end testing of complete user workflows through the UI layer
- DBSetup for validating database operations and data integrity
- Mockito for testing service layer interactions in isolation.

Together, these testing methods provide thorough coverage of integration points while allowing targeted testing of specific components.

End-to-end Testing

Five key test narratives were implemented using HtmlUnit inside the test class `HtmlUnitTests.java` of the package `vvs_webapp` to end-to-end test the SUT.

Test Architecture

All tests share:

- a WebClient configuration
- a reference to the `index.html` page
- helper methods for common operations (adding/removing customers, creating sales, etc.)

Each test follows the same pattern:

- set up the necessary preconditions
- perform the actions being tested
- verify the intermediary steps and results with assertions
- clean up by removing any test data to leave the database in its original state

To leave the database in its original state, I created a temporary customer before each test, performed the actions to it and removed it in the end, which deletes on cascade its information (Sales, Addresses and Deliveries).

1. Adding Addresses to Existing Customer

Method: `addTwoAddressesToCustomerTest()`

- Retrieves an existing customer's VAT number
- Gets initial count of customer addresses
- Adds two new addresses with predefined data
- Verifies the address table now includes both new addresses
- Confirms the total row count increased by exactly two

2. Customer Insertion

Method: `insertTwoCustomersTest()`

- Adds two new customers with VAT, designation, and phone details
- Navigates to the "List All Customers" page
- Verifies all customer information appears correctly in the list
- Cleans up by removing the test customers

3. Sale Creation

Method: `insertSaleTest()`

- Creates a temporary customer
- Adds a new sale for this customer
- Verifies the sale appears with status "O" (Open)
- Confirms the sale is properly associated with the customer's VAT number
- Cleans up by removing the temporary customer

4. Sale Closure

Method: `closeSaleTest()`

- Creates a temporary customer
- Adds a new sale to the customer
- Retrieves the sale ID
- Closes the sale
- Verifies the sale status changes to "C" (Closed)
- Cleans up by removing the temporary customer

5. Delivery Creation

Method: `insertDeliveryTest()`

- Creates a new customer with complete details
- Adds an address to the customer
- Creates a new sale for the customer
- Navigates to the delivery creation page
- Retrieves the previously inserted sale ID and address ID
- Creates a delivery connecting the sale and address
- Verifies the delivery appears correctly in the delivery table

Database Testing

Three test classes were implemented using DbSetup to test the database operations: `CustomersDBTest`, `SalesDBTest`, and `SaleDeliveriesDBTest`. The tests are supported by a utility class `DBSetupUtils` that handles database setup and provides common operations.

Database Setup

The `DBSetupUtils` class provides:

- Constants for database connection

- Operations for cleaning the database (`DELETE_ALL`)
- Predefined test data including customers, sales, addresses, and deliveries
- Combined operations like `INSERT_CUSTOMER_SALE_DATA`, `INSERT_CUSTOMER_ADDRESS_DATA` and `INSERT_CUSTOMER_ADDRESS_SALE_DATA`

Each test class uses a similar setup strategy:

- `@BeforeAll`: Connects to the test database
- `@BeforeEach`: Resets the database and loads appropriate test data

Customer Tests

Method: `addCustomerWithExistingVATTest()`

Tests that the SUT prevents adding a customer with an existing VAT number

1. Retrieves all existing customers
2. Attempts to add each customer again with the same data
3. Verifies that an `ApplicationException` is thrown for each attempt
4. Verifies that the number of customers matches the initial count

Method: `updateCustomerContactTest()`

Tests that customer contact information is properly updated

1. Retrieves all existing customers
2. Updates the phone number for all customers to a new value
3. Verifies that all customers now have the new phone number

Method: `deleteAllCustomersTest()`

Tests that deleting all customers results in an empty customer list

1. Retrieves all existing customers
2. Deletes all customers one by one
3. Verifies that the customer list is empty after deletion

Method: `deleteCustomerTest()`

Tests that a deleted customer can be added back without exceptions

1. Saves the initial list of customers
2. Deletes all customers and verifies the list is empty
3. Adds all customers back with their original information
4. Verifies that the number of customers matches the initial count

---

Sales Tests

Method: `deleteCustomerSalesAreDeletedTest()`

Tests that deleting a customer also removes its associated sales

1. Verifies that a specific customer exists and has sales
2. Deletes the customer
3. Confirms the customer no longer exists
4. Verifies that no sales remain for that customer's VAT number

Method: `addSaleIncreasesSalesNumberTest()`

Tests that adding a sale increases the total count by one

1. Gets the initial count of all sales
2. Adds a new sale for an existing customer
3. Verifies that the total count has increased by exactly one

Method: `newSaleHasOpenStatusTest()`

Tests that a newly created sale has the "Open" status ('O')

1. Creates a new sale for an existing customer
2. Retrieves the sales for that customer
3. Verifies that the most recent sale has status 'O'

Method: `newSaleHasZeroTotalTest()`

Tests that a newly created sale has a total of 0.0

1. Creates a new sale for an existing customer
2. Retrieves the sales for that customer
3. Verifies that the most recent sale has a total of 0.0

Sale Deliveries Tests

Method: `getSaleDeliveriesForCustomerTest()`

Tests retrieval of sale deliveries for a specific customer

- Verifies that the expected number of deliveries (2) are returned

Method: `addNewSaleDeliveryTest()`

Tests adding a new delivery for a sale

1. Gets an existing sale and address for a customer
2. Records the initial number of deliveries
3. Adds a new sale delivery
4. Verifies that the number of deliveries has increased by one

Service Layer Mocking

The current service layer implementation presents significant challenges for unit testing:

- The services are implemented as Java enums with singleton pattern:

```
public enum SaleService {
    INSTANCE;
    // implementation...
}
```

- This design presents several barriers to mocking:

    1. Java enums cannot be extended or instantiated by Mockito
    2. The singleton pattern with static INSTANCE references creates hard-coded dependencies
    3. No dependency injection is possible with this design pattern

Refactoring for Mockability

To enable proper unit testing with Mockito, the following refactoring would be needed:

1. Create Service interface to define contracts for the Service

```
// Example of Sale Service interface
public interface ISaleService {

    SalesDTO getSaleByCustomerVat(int vat) throws ApplicationException;

    SalesDTO getAllSales() throws ApplicationException;

    // more methods...
}
```

2. Change from the current singleton implementation to a regular class that implements the ISaleService interface:

```
// Example of Sale Service refactoring
public class SaleService implements ISaleService {

    @Override
    public SalesDTO getSaleByCustomerVat (int vat) throws ApplicationException {
        if (!isValidVAT (vat))
            throw new ApplicationException ("Invalid VAT number: " + vat);
    // more implementation...
```

3. Update all code that previously used SaleService.INSTANCE to instead use dependency injection:

```
// Example for GetSalePageController
@WebServlet("/GetSalePageController")
public class GetSalePageController extends PageController{
    private static final long serialVersionUID = 1L;
    private final ISaleService saleService;
    // Dependency injection constructor
    public GetSalePageController(ISaleService saleService) {
        this.saleService = saleService;
    }

    @Override
    protected void process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        SalesHelper sh = new SalesHelper();
        request.setAttribute("salesHelper", sh);

        try{
            String vat = request.getParameter("customerVat");
            if (isInt(sh, vat, "Invalid VAT number")) {
                int vatNumber = intValue(vat);
                SalesDTO sdto = saleService.getSaleByCustomerVat(vatNumber);
                sh.fillWithSales(sdto.sales);
                request.getRequestDispatcher("SalesInfo.jsp").forward(request,
response);
            }
        } catch (ApplicationException e) {
            sh.addMessage("It was not possible to fulfill the request: " +
e.getMessage());
            request.getRequestDispatcher("CustomerError.jsp").forward(request,
response);
        }
    }

}
```

Mockito Test Example

Here's an example of how to test a module that depends on SaleService using Mockito:

```
class GetSalePageControllerTest {

    private GetSalePageController controller;

    ISaleService saleService = mock(ISaleService.class);
    HttpServletRequest request = mock(HttpServletRequest.class);
    HttpServletResponse response = mock(HttpServletResponse.class);
    RequestDispatcher dispatcher = mock(RequestDispatcher.class);

    private static final String VAT_STRING = "274187949";
    private static final int VAT_INT = 274187949;
    private static final SalesDTO SALES = new SalesDTO(List.of(
            new SaleDTO(1, new Date(), 0.0, "O", VAT_INT)
        ));
    @BeforeEach
    public void init() {
        controller = new GetSalePageController(saleService);
    }

    @Test
    public void spyTest() throws Exception {

        // Set up the behavior of our mocks
        when(saleService.getSaleByCustomerVat(VAT_INT)).thenReturn(SALES);
        when(request.getParameter("customerVat")).thenReturn(VAT_STRING);

when(request.getRequestDispatcher("SalesInfo.jsp")).thenReturn(dispatcher);

        // Act
        controller.process(request, response);

        // Verify that the SUT is behaving as expected
        verify(saleService).getSaleByCustomerVat(VAT_INT);
        verify(request).getRequestDispatcher("SalesInfo.jsp");
        verify(dispatcher).forward(request, response);
    }
}
```

SUT Modifications

Modifications related to BackLog bug fixes

- **Bug [VVS_PROJ2_01-1]**: Lack of VAT verification when creating Sale Delivery

  A security vulnerability was discovered in the Sale Delivery creation process:

  - Fault Location: `SaleService.addSaleDelivery()` method
  - Fault: Missing validation to ensure Sale and Address belong to the same customer
  - Failure: System allowed creating deliveries where a sale from one customer could be delivered to another customer's address
  - Estimated Effort: 45 minutes

  The fault propagates through the following sequence (RIP model):

  1. Reachability: The fault is reached whenever a user attempts to create a delivery using a sale and address from different customers
  2. Infection: The database state becomes infected when a delivery record is created linking a sale from one customer to an address belonging to another customer
  3. Propagation: The infected state propagates to application failures when:
     - Delivery operations attempt to process deliveries with mismatched customer data
     - Reports show incorrect customer-sale-address relationships

  The fault was resolved by:

1. Adding a new method `getAddressById()` to `AddressRowDataGateway` to retrieve address information
2. Modifying `SaleService.addSaleDelivery()` to verify that both the sale and address belong to the same customer by comparing their VAT numbers
3. Throwing an `ApplicationException` if the VAT numbers don't match

This fix ensures proper data integrity by preventing cross-customer deliveries, which can be verified by the `insertDeliveryTest` test case.

- **Bug [VVS_PROJ2_01-3]**: Sale delivery allows multiple addresses for a single Sale

  A data integrity issue was discovered in the Sale Delivery creation process:

  - Fault Location: `SaleService.addSaleDelivery()` method
  - Fault: Missing validation to prevent multiple delivery addresses for the same sale
  - Failure: System allowed a single sale to be associated with multiple delivery addresses, potentially causing delivery confusion
  - Estimated Effort: 45 minutes

  The fault propagates through the following sequence (RIP model):

  1. Reachability: The fault is reached whenever a user attempts to add a delivery address to a sale that already has one
  2. Infection: The database state becomes infected when multiple delivery addresses are incorrectly associated with the same sale
  3. Propagation: The infected state propagates to application failures when:
     - Delivery operations attempt to process multiple addresses for the same sale
     - Reports or queries show inconsistent delivery information
     - Business logic that assumes one delivery per sale produces incorrect results

  The fault was resolved by:

  1. Adding a new method `updateAddressId()` to `SaleDeliveryRowDataGateway` to modify existing delivery addresses
  2. Modifying `SaleService.addSaleDelivery()` to check if a sale already has a delivery
  3. If a delivery exists, updating its address instead of creating a new one
  4. If no delivery exists, creating a new one as before

  This fix ensures that each sale can only have one delivery address at a time, which can be verified by the `insertDeliveryTest` test case.

- **Bug [VVS_PROJ2_01-12]**: Customer deletion does not cascade to related records

  A fault was discovered in the database schema design that manifested as a failure in referential integrity. Specifically:

  - Fault Location: Database schema foreign key constraints between Customer table and its dependent tables (Sales, Addresses, Deliveries)

  - Fault: Missing ON DELETE CASCADE constraints on foreign key relationships
  - Failure: When executing customer deletion operations, orphaned records remained in dependent tables, violating referential integrity
  - Estimated Effort: 30 minutes

  The fault propagates through the following sequence (RIP model):

  1. Reachability: The fault is reached whenever a customer deletion operation is executed
  2. Infection: The database state becomes infected when the deletion occurs but dependent records remain
  3. Propagation: The infected state propagates to application failures when:
     - Queries attempt to reference non-existent customers
     - Data inconsistency causes incorrect business logic execution

  The issue was resolved by:

  1. Adding ON DELETE CASCADE constraints to all foreign key relationships referencing the Customer table in `createDDLHSQLDB.sql`
  2. Modifying table drop order in schema to respect referential integrity in `dropDDLHSQLDB.sql`

  This fix ensures proper cascading deletion behavior, which is verified by the `deleteCustomerSalesAreDeletedTest` test case.

Minor modifications

- To facilitate element selection in HtmlUnit tests, HTML table elements were given unique identifiers across the following JSP pages:
  - `CustomerInfo.jsp` and `addSaleDelivery.jsp`:
    - Added `id="addressesTable`" to address tables
  - `SalesInfo.jsp`, `CloseSale.jsp`, and `addSaleDelivery.jsp`:
    - Added `id="salesTable"` to sales tables
  - `ShowSalesDelivery.jsp` and `SalesDeliveryInfo.jsp`:
    - Added `id="salesDeliveryTable"` to sales delivery tables