



SILAH

صلة
نربط
الأعمال

Connecting Businesses

An Intermediary Between Businesses and Suppliers

Student #1 Shahad Aljohani 443007510

Student #2 Mayar Al-Shafi 442004436

Student #3 Hend Alotaibi 443007591

Student #4 Amal Alzahrani 441001577

Student #5 Fayrouz Sagga 442005394

Supervised By

Dr. Abeer Alarfaj

A Graduation Project Report Submitted to

College of Computer Sciences and Information at PNU in Partial Fulfillment of the
Requirements for the Degree of

Bachelor of Science

In Computer Science

CCIS, PNU | Riyadh, KSA | 1446 – 1447 H

Table of Contents

List of Tables	iv
List of Figures	vi
Acknowledgments.....	xxiii
Abstract.....	xxiv
Chapter 1 Introduction.....	1
1.1 Problem Statement & Significance	2
1.2 Proposed Solution (System).....	2
1.3 Project Domain & Limitations.....	4
1.4 The Methodology	5
1.5 Definitions of New Terms	6
Chapter 2 Background Information & Related Work	9
2.1 Background Information	10
2.1.1 Business Background.....	10
2.1.2 Technical Background	12
2.2 Related Work Survey	21
2.3 Proposed & Similar Systems Comparison	28
Chapter 3 System Analysis.....	33
3.1 Requirements Specification	34
3.1.1 Requirements Gathering	34
3.1.2 Algorithm Definition	37
3.1.3 System Requirements.....	41
3.2 Requirements Analysis	48
3.2.1 Use Case Diagrams & Descriptions.....	48
3.2.2 Sequence Diagrams.....	66
3.2.3 Class Diagrams	87

Chapter 4	System Design	106
4.1	System Architecture.....	107
4.1.1	High-Level System Overview.....	107
4.1.2	Backend Modules.....	108
4.1.3	Database Schema & Datasets.....	112
4.2	User Interface Design	127
4.2.1	Guest User Interface	130
4.2.2	Supplier User Interface	133
4.2.3	Buyer User Interface	145
Chapter 5	Implementation	158
5.1	Implementation Requirements	159
5.1.1	Software Requirements	159
5.1.2	Hardware Requirements.....	160
5.2	Implementation Details	160
5.2.1	Artificial Intelligence Implementation.....	163
5.2.2	Backend Implementation	182
5.2.3	Frontend Implementation.....	230
5.3	Results: Interface and User Interaction Screens	268
5.3.1	Guest Pages	268
5.3.2	Supplier Pages.....	273
5.3.3	Buyer Pages	295
5.3.4	Language Support & Empty States.....	329
Chapter 6	Testing.....	336
6.1	Test Plan.....	337
6.2	Test Cases	339
6.2.1	Unit Testing	339
6.2.2	Integration Testing.....	341

6.2.3	Functional Testing	343
6.2.4	Non-Functional Testing	354
6.3	Summary of Test Results	369
Chapter 7	Conclusion	370
7.1	Evaluation	371
7.2	Future Work	375
References.....		380
Appendix.....		387
Appendix A: Survey Questions		387
Appendix B: Detailed System Requirements Tables		390
Appendix C: Development Workflow and GitHub Repositories		398
Appendix D: Full Prisma Schema Code		407
Appendix E: Detailed Individual Usability Testing Results.....		416

List of Tables

Table 1-1: Terms and Definitions	6
Table 2-1: Summarizes the key differences between B2C and B2B business models [7], [8]	11
Table 2-2: Comparison of Architectural Styles [15].....	15
Table 2-3: Research Papers Comparison	24
Table 2-4: Related Websites Comparison.....	32
Table 3-1: Actor Descriptions.....	48
Table 3-2: Sing-up Use Case Description.....	49
Table 3-3: Login Use Case Description.....	50
Table 3-4: Logout Use Case Description.....	50
Table 3-5: Change Role Use Case Description.....	51
Table 3-6: View Notifications Use Case Description.....	51
Table 3-7: Send Message Use Case Description	51
Table 3-8: View Storefront Listings Use Case Description.....	53
Table 3-9: View Business Insights Use Case Description.....	54
Table 3-10: View Orders Use Case Description	55
Table 3-11: Create an Invoice Use Case Description	56
Table 3-12: View Bids Use Case Description	56
Table 3-13: Manage Subscription Use Case Description	57
Table 3-14: Pay Premium Plan Use Case Description.....	58
Table 3-15: Search for an Item Use Case Description.....	59
Table 3-16: Search for Similar Products Use Case Description	60
Table 3-17: View Invoices Use Case Description	61
Table 3-18: Add an Item to Wishlist Use Case Description	62
Table 3-19: Add Product to Cart Use Case Description	62
Table 3-20: View Orders Use Case Description	63
Table 3-21: Join a Group Purchase Use Case Description	64
Table 3-22: Open a Bid Use Case Description	65
Table 5-1: Defined RESTful routes for Products Resource.....	185
Table 6-1: Unit Testing Results	339
Table 6-2: Integration Testing Results.....	341
Table 6-: Functional Testing Results	343

Table 6-: Functional Test Results Summary.....	353
Table 6-: Reliability Test Results	358
Table 6-: Portability Test Results	365
Table 6-: Aggregated Usability Testing Results (10 participants & 20 tasks)	365
Table 6-: Overall Summary of Test Results	369
Table 7-1: Achievement of Project Aims and Goals	373

List of Figures

Figure 1-1: The Waterfall Model.....	6
Figure 2-1: HTTP Request-Response Process in a B2B Platform [12]	13
Figure 2-2: Establishing a WebSocket Connection in a Real-Time Messaging System [13].....	14
Figure 2-3: WebSocket-Based Real-Time Messaging Between Buyers and Suppliers	14
Figure 2-4: Architectural Styles Comparison, based on the degree of deployment separation and modularity [14]	15
Figure 2-5: Common Time Series Patterns [17]	17
Figure 2-6: The BERT model architecture [18].....	19
Figure 2-7: Dual encoder model with BERT based encoding modules [19]	20
Figure 2-8: Cosine Similarity Representation [20].....	21
Figure 3-1: Distribution of Industries Represented in the Survey	35
Figure 3-2: Respondent Roles in the B2B Marketplace	35
Figure 3-3: Frequency of Transactions in the B2B Marketplace.....	36
Figure 3-4: Current Procurement Methods Used by Businesses	36
Figure 3-5: Key Procurement Challenges in the B2B Marketplace	37
Figure 3-6: Key Challenges Faced by Suppliers When Selling to Businesses	37
Figure 3-7: Key Challenges Faced by Buyers When Sourcing Suppliers	37
Figure 3-8: AI Model Development and Deployment Workflow	40
Figure 3-9: Guest Use Case Diagram	49
Figure 3-10: User Use Case Diagram	50
Figure 3-11: Supplier Use Case Diagram	53
Figure 3-12: Buyer Use Case Diagram	59
Figure 3-13: Sign-Up Process Sequence Diagram.....	67
Figure 3-14: Login Process Sequence Diagram.....	68
Figure 3-15: Password Reset Sequence Diagram	69
Figure 3-16: Switch User Role Sequence Diagram	70
Figure 3-17: Reactivate Supplier Account Sequence Diagram	70
Figure 3-18: Logout Sequence Diagram.....	71
Figure 3-19: View Notifications Sequence Diagram	71
Figure 3-20: Chat Flow Sequence Diagram.....	72

Figure 3-21: Search Flow Sequence Diagram	72
Figure 3-22: View Storefront Listings & Stock Demand Sequence Diagram	73
Figure 3-23: Create Product Sequence Diagram	73
Figure 3-24: Create Service Sequence Diagram	74
Figure 3-25: Create Invoice Sequence Diagram	75
Figure 3-26: Subscription Management Sequence Diagram	76
Figure 3-27: Supplier View Analytics & Insights Sequence Diagram	77
Figure 3-28: Supplier Order Management Sequence Diagram.....	78
Figure 3-29: Supplier Bid Participation Sequence Diagram.....	79
Figure 3-30: Buyer Bid Creation & Offer Handling Sequence Diagram	80
Figure 3-31: Find Similar Products Sequence Diagram	80
Figure 3-32: Wishlist Interaction Sequence Diagram.....	81
Figure 3-33: Add to Cart and Stock Validation Sequence Diagram.....	82
Figure 3-34: Join or Start Group Purchase Sequence Diagram	83
Figure 3-35: Payment Flow Using Tap Gateway Sequence Diagram	84
Figure 3-36: Buyer Order Completion and Review Submission Sequence Diagram..	85
Figure 3-37: View Invoices Sequence Diagram	86
Figure 3-38: Module usage relationships in Silah's backend	87
Figure 3-39: Authentication Module Class Diagram	87
Figure 3-40: User Module Class Diagram	88
Figure 3-41: Supplier Module Class Diagram	89
Figure 3-42: Buyer Module Class Diagram	89
Figure 3-43: Category Module Class Diagram	90
Figure 3-44: Settings Module Class Diagram.....	91
Figure 3-45: Notification Module Class Diagram	92
Figure 3-46: Chat Module Class Diagram	93
Figure 3-47: Bid Module Class Diagram.....	94
Figure 3-48: Offer Module Class Diagram	95
Figure 3-49: Product Module Class Diagram	96
Figure 3-50: Service Module Class Diagram.....	97
Figure 3-51: Group Purchase Module Class Diagram	98
Figure 3-52: Demand Prediction Module Class Diagram.....	99
Figure 3-53: Similar Products Module Class Diagram.....	100

Figure 3-54: Order Module Class Diagram	101
Figure 3-55: Invoice Module Class Diagram.....	102
Figure 3-56: Review Module Class Diagram	103
Figure 3-57: Payments Module Class Diagram	104
Figure 3-58: Search Module Class Diagram.....	105
Figure 4-1: Full System Architecture	107
Figure 4-2: Modular Structure of the Backend System	109
Figure 4-3: The Components that make up the Modules.....	111
Figure 4-4: Key Tables and Relationships in Silah's Database	114
Figure 4-5: User, Supplier, Buyer, and Category Management Tables	115
Figure 4-6: Product, Service, Wishlist, and Embedding Tables	116
Figure 4-7: Cart, Order, and Order Fulfillment Tables.....	117
Figure 4-8: Invoice, PreInvoice, and Billing Tables	118
Figure 4-9: Group Purchase Tables	119
Figure 4-10: Bid and Offer Tables	120
Figure 4-11: Review and ItemReview Tables	120
Figure 4-12: Chat, Messaging, and Notification Tables	121
Figure 4-13: Overview of 'Order Date' Column	123
Figure 4-14: Overview of 'Product ID' Column.....	123
Figure 4-15: Overview of 'Sales' Column.....	123
Figure 4-16: Overview of 'product_id' Column	125
Figure 4-17: Overview of 'product_name' Column	126
Figure 4-18: Overview of 'category' Column	126
Figure 4-19: Overview of 'about_product' Column	126
Figure 4-20: Footer Navigation Structure.....	127
Figure 4-21: Guest Header Navigation	128
Figure 4-22: Buyer Header Navigation.....	128
Figure 4-23: Supplier Sidebar Navigation	129
Figure 4-24: User Journey from Buyer to Supplier Role and Navigation Flow	129
Figure 4-25: The Landing Page	131
Figure 4-26: Sing-up Page (Step 1)	132
Figure 4-27: Sing-up Page (Step 2)	132
Figure 4-28: Sing-up Page (Step 3)	132

Figure 4-29: The Login Page	133
Figure 4-30: The Overview Page	134
Figure 4-31: Products & Services Page	135
Figure 4-32: The Product Details Page	136
Figure 4-33: The Service Details Page	137
Figure 4-34: Stock Predication Page.....	138
Figure 4-35: Stock Predication Page (Unauthorized Supplier).....	138
Figure 4-36: Biddings Page	138
Figure 4-37: The Bid Details Page.....	139
Figure 4-38: Write an Offer Page	139
Figure 4-39: Supplier Settings Page	140
Figure 4-40: Subscriptions Page	141
Figure 4-41: Supplier Notifications Page	141
Figure 4-42: Orders Page	142
Figure 4-43: The Order Details Page	142
Figure 4-44: Chats Page.....	142
Figure 4-45: The Chat Page	142
Figure 4-46: Create an Invoice Page.....	143
Figure 4-47: Link a Listing Page	143
Figure 4-48: Invoices Page	144
Figure 4-49: The Invoice Details Page	144
Figure 4-50: Store Analytics & Insights Page	145
Figure 4-51: The homepage	146
Figure 4-52: Alternative Products Page	147
Figure 4-53: The Product Details Page	148
Figure 4-54: The Cart Page.....	149
Figure 4-55: The Service Details Page	149
Figure 4-56: Supplier Storefront Page	150
Figure 4-57: Inactive Supplier Storefront Page	150
Figure 4-58: Searched for a Listing Page	151
Figure 4-59: Searched for a Supplier Page	151
Figure 4-60: Browsing by Category Page.....	151
Figure 4-61: The Wishlist Page	152

Figure 4-62: Bids Page.....	152
Figure 4-63: Create a Bid Page.....	153
Figure 4-64: Offers Received Page.....	153
Figure 4-65: The Offer Details Page.....	154
Figure 4-66: buyer Settings Page (No Card Stored)	155
Figure 4-67: Buyer Settings Page	155
Figure 4-68: Buyer Notifications Page	155
Figure 4-69: Orders Page	156
Figure 4-70: The Order Details Page (Shipped)	156
Figure 4-71: The Order Details Page (Completed).....	156
Figure 4-72: Write a Review Page	156
Figure 4-73: Chats Page.....	157
Figure 4-74: The Search for a Chat Page.....	157
Figure 4-75: Invoices Page	157
Figure 4-76: The Invoice Details Page	157
Figure 5-1: Frontend cloc output	162
Figure 5-2: Backend cloc output.....	162
Figure 5-3: Code snippet showing dataset loading, column renaming, and library imports used during Prophet model preparation	164
Figure 5-4: Implementation of Development Mode and Dataset Splitting for Prophet Model Training	165
Figure 5-5: Identification of Most Sold Product Used for Model Training.....	165
Figure 5-6: Model Evaluation Metrics (MAE and RMSE) for the Most Sold Product	166
Figure 5-7: Correction of Negative Forecast Values Generated by Prophet	166
Figure 5-8: Predicted Sales for the Next Three Months (January–March 2018).....	166
Figure 5-9: Actual vs. Forecasted Sales Over Time for the Selected Product.....	166
Figure 5-10: Dataset loading and preprocessing for LaBSE fine-tuning.....	168
Figure 5-11: Library imports and LaBSE model initialization with fixed random seeds and CPU configuration	168
Figure 5-12: Implementation of the build_positive_pairs() function for generating anchor-positive text pairs from the dataset	169

Figure 5-13: DataLoader and loss function setup using MultipleNegativesRankingLoss for contrastive fine-tuning.....	169
Figure 5-14: LaBSE fine-tuning loop executed for one epoch with progress tracking and warm-up scheduling	170
Figure 5-15: Model saving and similarity validation between products from the same and different categories.....	170
Figure 5-16: Evaluation code for computing Top-1 and Top-5 accuracy across the test set	171
Figure 5-17: FastAPI Project Structure.....	172
Figure 5-18: FastAPI data models for sales forecasting requests (Prophet)	174
Figure 5-19: Request and Response Body Examples (/demand).....	174
Figure 5-20: FastAPI /demand endpoint defined in main.py	175
Figure 5-21: Implementation of run_forecast() function inside prophetAI.py, showing input arguments and preprocessing logic.....	176
Figure 5-22: Forecast generation and aggregation steps, converting Prophet's daily predictions into monthly demand values	176
Figure 5-23: FastAPI data models for product similarity requests (LaBSE).....	177
Figure 5-24: Request and Response Body Examples (/similar-search).....	177
Figure 5-25: FastAPI /similar-search endpoint defined in main.py.....	178
Figure 5-26: Implementation of find_similar_items() inside labse_ai.py (1/3).....	180
Figure 5-27: Implementation of find_similar_items() inside labse_ai.py (2/3).....	180
Figure 5-28: Implementation of find_similar_items() inside labse_ai.py (3/3).....	181
Figure 5-29: silah-site-server Droplet on DigitalOcean.....	182
Figure 5-30: Swagger UI — General API description and structure.....	187
Figure 5-31: Auth module endpoints displayed in Swagger UI	188
Figure 5-32: Signup endpoint description and request body example	188
Figure 5-33: Signup endpoint request body schema	188
Figure 5-34: Signup endpoint response body and error response examples.....	189
Figure 5-35: Signup endpoint swagger documentation code (1/5)	190
Figure 5-36: Signup endpoint swagger documentation code (2/5)	191
Figure 5-37: Signup endpoint swagger documentation code (3/5)	191
Figure 5-38: Signup endpoint swagger documentation code (4/5)	191
Figure 5-39: Signup endpoint swagger documentation code (5/5)	191

Figure 5-40: Postman workspace showing the main project collections	192
Figure 5-41: Successful signup request example with a 201 Created response	193
Figure 5-42: Failed signup request example showing 400 (Bad Request) error response.....	193
Figure 5-43: NestJS Project root structure	194
Figure 5-44: Inside the NestJS src/ directory.....	195
Figure 5-45: Example module (auth/)... .	196
Figure 5-46: BidModule	197
Figure 5-47: BidController (1/2).....	197
Figure 5-48: BidController (2/2).....	197
Figure 5-49: BidService (1/3)	198
Figure 5-50: BidService (2/3)	198
Figure 5-51: BidService (3/3)	198
Figure 5-52: CreateBidDto	199
Figure 5-53: BidResponseDto.....	199
Figure 5-54: JwtAuthGuard	200
Figure 5-55: RolesGuard.....	200
Figure 5-56: VerifiedGuard	201
Figure 5-57: ParseCrnPipe	201
Figure 5-58: ParseEmailPipe	202
Figure 5-59: AllExceptionsFilter	203
Figure 5-60: LoggerMiddleware.....	204
Figure 5-61: RolesDecorator.....	204
Figure 5-62: User Model on shema.prisma.....	206
Figure 5-63: Example of how to use Prisma Client	206
Figure 5-64: categorySeed.ts (1/3).....	207
Figure 5-65: categorySeed.ts (2/3).....	207
Figure 5-66: categorySeed.ts (3/3).....	207
Figure 5-67: Fuzzy Search Example (1/2)	208
Figure 5-68: Fuzzy Search Example (2/2)	208
Figure 5-69: ChatGateway (1/2)	209
Figure 5-70: ChatGateway (2/2)	209
Figure 5-71: SSE Endpoint in NotificationController	210

Figure 5-72: Global NotificationModule	210
Figure 5-73: createNotification method on NotificationService (1/2)	211
Figure 5-74: createNotification method on NotificationService (2/2)	211
Figure 5-75: Usage of NotificationService on the InvoiceService	211
Figure 5-76: Send Image Endpoint on ChatController	212
Figure 5-77: Send Image Logic on ChatService	212
Figure 5-78: DeepL Integration on TranslationService (1/2)	213
Figure 5-79: DeepL Integration on TranslationService (2/2)	213
Figure 5-80: createCustomer on TapPaymentService	214
Figure 5-81: createCharge on TapPaymentService	215
Figure 5-82: validateCharge on TapPaymentService	215
Figure 5-83: getCharge on TapPaymentService	215
Figure 5-84: WathqService	216
Figure 5-85: FileService (1/2).....	217
Figure 5-86: FileService (2/2).....	217
Figure 5-87: DemandPredictionController	218
Figure 5-88: DemandPredictionService (1/3)	218
Figure 5-89: DemandPredictionService (2/3)	219
Figure 5-90: DemandPredictionService (3/3)	219
Figure 5-91: DemandPredicationResponseDto.....	220
Figure 5-92: SmartSearchController	220
Figure 5-93: SmartSearchRequestDto	220
Figure 5-94: SmartSearchService (1/8)	221
Figure 5-95: SmartSearchService (2/8)	221
Figure 5-96: SmartSearchService (3/8)	222
Figure 5-97: SmartSearchService (4/8)	222
Figure 5-98: SmartSearchService (5/8)	223
Figure 5-99: SmartSearchService (6/8)	223
Figure 5-100: SmartSearchService (7/8)	223
Figure 5-101: SmartSearchService (8/8)	223
Figure 5-102: SmartSearchResponseDto	224
Figure 5-103: GitHub Actions Job Script	225
Figure 5-104: PM2 Process.....	225

Figure 5-105: NGIX Configuration for silah-backend	226
Figure 5-106: React Project Structure.....	233
Figure 5-107: main.jsx file.....	235
Figure 5-108: Simple App.jsx Example.....	235
Figure 5-109: App.jsx (1/3)	237
Figure 5-110: App.jsx (2/3)	237
Figure 5-111: App.jsx (3/3)	237
Figure 5-112: Public Layout	239
Figure 5-113: Guest Layout.....	239
Figure 5-114: Buyer Layout.....	239
Figure 5-115: Supplier Layout.....	239
Figure 5-116: Shared Layout	239
Figure 5-117: ProtectedRoute.jsx (1/2).....	240
Figure 5-118: ProtectedRoute.jsx (2/2).....	240
Figure 5-119: Example of a simple i18n setup	240
Figure 5-120: i18n.js file.....	241
Figure 5-121: ar/chats.json file	241
Figure 5-122: en/chats.json file.....	241
Figure 5-123: Setting page title dynamically	241
Figure 5-124: Settting page title dynamically with variables example.....	241
Figure 5-125: AuthContext.jsx (1/3).....	243
Figure 5-126: AuthContext.jsx (2/3).....	243
Figure 5-127: AuthContext.jsx (3/3).....	243
Figure 5-128: Example of Axios integration in the login page (Login.jsx).....	244
Figure 5-129: Login.jsx (1/3).....	245
Figure 5-130: Login.jsx (2/3).....	245
Figure 5-131: Login.jsx (3/3).....	245
Figure 5-132: Page structure showing Login.jsx and Login.css under pages/Login/	246
Figure 5-133: useNotification.js hook (1/3).....	248
Figure 5-134: useNotification.js hook (2/3).....	248
Figure 5-135: useNotification.js hook (3/3).....	248
Figure 5-136: NotificationListener.jsx.....	249
Figure 5-137: NotificationContext.jsx	249

Figure 5-138: NotificationItemContext.jsx	250
Figure 5-139: Notifications.jsx (1/5)	251
Figure 5-140: Notifications.jsx (2/5)	251
Figure 5-141: Notifications.jsx (3/5)	252
Figure 5-142: Notifications.jsx (4/5)	252
Figure 5-143: Notifications.jsx (5/5)	252
Figure 5-144: socket.js file	253
Figure 5-145: Receiving new messages logic on Chats.jsx	254
Figure 5-146: Receiving new messages logic on Chats/[id].jsx	254
Figure 5-147: Send a message logic on Chats/[id].jsx	254
Figure 5-148: Send an image logic on Chats/[id].jsx	254
Figure 5-149: Alternatives.jsx (1/3).....	256
Figure 5-150: Alternatives.jsx (2/3).....	256
Figure 5-151: Alternatives.jsx (3/3).....	256
Figure 5-152: Demand/[id].jsx (1/5).....	257
Figure 5-153: Demand/[id].jsx (2/5).....	257
Figure 5-154: Demand/[id].jsx (3/5).....	258
Figure 5-155: Demand/[id].jsx (4/5).....	258
Figure 5-156: Demand/[id].jsx (5/5).....	258
Figure 5-157: TapCardForm.jsx (1/3).....	260
Figure 5-158: TapCardForm.jsx (2/3).....	260
Figure 5-159: TapCardForm.jsx (3/3).....	260
Figure 5-160: TapCardFrom usage in Buyer/Settings.jsx	260
Figure 5-161: handleTokenGenerated method in Buyer/Settings.jsx	260
Figure 5-162: Payment/Callback.jsx (1/3)	261
Figure 5-163: Payment/Callback.jsx (2/3)	261
Figure 5-164: Payment/Callback.jsx (3/3)	261
Figure 5-165: NGIX Configuration for frontend-silah	263
Figure 5-166: GitHub Actions Job Script for Frontend Deployment	264
Figure 5-167: Overview of the Methodology and Tools Used in Silah’s Lifecycle..	267
Figure 5-168: Landing Page (1/3).....	269
Figure 5-169: Landing Page (2/3).....	269
Figure 5-170: Landing Page (3/3).....	270

Figure 5-171: Sign-up Page (1/3)	271
Figure 5-172: Sign-up Page (2/3)	271
Figure 5-173: Sign-up Page (3/3)	271
Figure 5-174: Login Page	272
Figure 5-175: Initial Verify Email Dialog	273
Figure 5-176: Email Successfully Verified Dialog.....	273
Figure 5-177: Request Resend Verification Email	273
Figure 5-178: Resent Verification Email Successfully.....	273
Figure 5-179: Sample verification email received by the user	273
Figure 5-180: Overview Page (Opened Store).....	274
Figure 5-181: Overview Page (Closed Store)	274
Figure 5-182: Listings Page (Premium Plan).....	275
Figure 5-183: Listings Page (Basic Plan)	275
Figure 5-184: Listings Page (Search Results).....	276
Figure 5-185: Supplier's Product Details Page (1/2)	277
Figure 5-186: Supplier's Product Details Page (2/2)	277
Figure 5-187: Update Stock Modal.....	278
Figure 5-188: Supplier's Service Details Page (1/2)	279
Figure 5-189: Supplier's Service Details Page (2/2)	279
Figure 5-190: Demand Predications Page (Needs Restocking)	280
Figure 5-191: Demand Predications Page (No Restocking Needed).....	280
Figure 5-192: Demand Predications Page (Basic Plan).....	281
Figure 5-193: Demand Predications Page (No Sales).....	281
Figure 5-194: Demand Predications Page (Not Enough Sales)	281
Figure 5-195: Supplier's Biddings Page (All)	282
Figure 5-196: Supplier's Biddings Page (Joined Only)	282
Figure 5-197: Supplier's Bid Details Page	283
Figure 5-198: Write an Offer Page	283
Figure 5-199: Supplier's Settings Page (General).....	284
Figure 5-200: Supplier's Settings Page (Account)	284
Figure 5-201: Supplier's Settings Page (Notifications).....	285
Figure 5-202: Supplier's Settings Page (Store)	285
Figure 5-203: Supplier's Settings Page (Support).....	286

Figure 5-204: Choose Plan Page.....	286
Figure 5-205: Supplier's Notifications Page	287
Figure 5-206: Supplier's Notification Toast.....	287
Figure 5-207: Supplier's Orders Page	288
Figure 5-208: Supplier's Order Details Page	288
Figure 5-209: Supplier's Chats Page	289
Figure 5-210: Supplier's Chats Page (Search Results).....	289
Figure 5-211: Supplier's Chat Page	290
Figure 5-212: Create an Invoice Page (1/2).....	291
Figure 5-213: Create an Invoice Page (2/2).....	291
Figure 5-214: Link an Item Modal.....	292
Figure 5-215: Supplier's Invoices Page	293
Figure 5-216: Supplier's Invoice Details Page.....	293
Figure 5-217: Supplier's Pre-Invoice Details Page (Group Purchase).....	294
Figure 5-218: Supplier's Pre-Invoice Details Page (Offer).....	294
Figure 5-219: Analytics Page (Premium Plan)	295
Figure 5-220: Analytics Page (Basic Plan)	295
Figure 5-221: Buyer Header	297
Figure 5-222: Buyer Header (Opened Category Megamenu).....	297
Figure 5-223: Buyer Header (Opened Category Megamenu + Hovered on Main Menu List Item).....	297
Figure 5-224: Buyer Header (Opened Notifications Menu)	297
Figure 5-225: Buyer Header (Opened Notifications Menu + No Unread Notification)	297
Figure 5-226: Buyer Header (Opened Profile Menu)	297
Figure 5-227: Homepage (1/2).....	298
Figure 5-228: Homepage (2/2).....	298
Figure 5-229: Buyer's Product Details Page (Out of Stock).....	299
Figure 5-230: Alternatives Page	300
Figure 5-231: Buyer's Product Details Page (1/2) + Join Group Purchase Section ...	301
Figure 5-232: Buyer's Product Details Page (2/2)	301
Figure 5-233: Joined Group Purchase Dialog.....	302
Figure 5-234: Start Group Purchase Section	302

Figure 5-235: Started Group Purchase Dialog	303
Figure 5-236: Buyer's Product Details Page (No Group Purchase)	303
Figure 5-237: Added Item to Cart Dialog	304
Figure 5-238: Cart Page	304
Figure 5-239: Tap ACS Emulator	305
Figure 5-240: Payment Callback (Cart Paid + Order Made)	305
Figure 5-241: Payment Callback (Invoice Paid)	305
Figure 5-242: Payment Callback (Card Saved)	305
Figure 5-243: Buyer's Service Details Page (1/2).....	306
Figure 5-244: Buyer's Service Details Page (2/2).....	306
Figure 5-245: Storefront Page (1/2).....	307
Figure 5-246: Storefront Page (2/2).....	307
Figure 5-247: Storefront Page (Store Closed)	307
Figure 5-248: Storefront Page (Inactive Supplier).....	307
Figure 5-249: Search Results Page (Products).....	309
Figure 5-250: Search Results Page (Services)	309
Figure 5-251: Search Results Page (Suppliers)	310
Figure 5-252: Browse by Category Page (Main Category)	311
Figure 5-253: Browse by Category Page (Subcategory)	311
Figure 5-254: Wishlist Page.....	312
Figure 5-255: Buyer's Biddings Page	313
Figure 5-256: Create a New Bid Page	313
Figure 5-257: Buyer's Bid Details Page.....	314
Figure 5-258: Received Offers Page.....	315
Figure 5-259: Received Offers Page (Declined Offer)	315
Figure 5-260: Received Offers Page (Accepted Offer)	315
Figure 5-261: Offer Details Page.....	315
Figure 5-262: Buyer's Settings Page (General).....	317
Figure 5-263: Buyer's Settings Page (Account).....	317
Figure 5-264: Buyer's Settings Page (Notifications)	318
Figure 5-265: Buyer's Settings Page (Payment Method + Stored Card)	318
Figure 5-266: Buyer's Settings Page (Payment Method + No Stored Card)	319
Figure 5-267: Buyer's Settings Page (Support).....	319

Figure 5-268: Buyer's Notifications Page	320
Figure 5-269: Buyer's Notification Toast	320
Figure 5-270: Buyer's Orders Page.....	321
Figure 5-271: Buyer's Order Details Page	321
Figure 5-272: Buyer's Order Details Page (Shipped)	322
Figure 5-273: Buyer's Chats Page.....	323
Figure 5-274: Buyer's Chats Page (Search Results)	323
Figure 5-275: Buyer's Chat Page	324
Figure 5-276: Buyer's Invoices Page	325
Figure 5-277: Buyer's Invoice Details Page (Pending).....	325
Figure 5-278: Buyer's Invoice Details Page (Accepted).....	326
Figure 5-279: Buyer's Pre-Invoice Details Page (Group Purchase).....	326
Figure 5-280: Buyer's Pre-Invoice Details Page (Offer).....	327
Figure 5-281: Buyer's Order Details Page (Completed)	328
Figure 5-282: Buyer's Invoice Details Page (Fully Paid)	328
Figure 5-283: Write a Review Page.....	329
Figure 5-284: Overview in RTL	330
Figure 5-285: Analytics in RTL.....	330
Figure 5-286: Homepage in RTL.....	331
Figure 5-287: Storefront in RTL.....	331
Figure 5-288: Listings (No Search Result)	332
Figure 5-289: Supplier's Orders Page (No Orders).....	332
Figure 5-290: Analytics (No Data)	333
Figure 5-291: Cart (Empty)	333
Figure 5-292: Search Results Page (Products + No Results).....	334
Figure 5-293: Received Offers Page (No Offers)	334
Figure 5-294: Buyer's Chat Page (New Chat).....	335
Figure 61: CRN exists.....	343
Figure 62: NID exits	344
Figure 63: CRN must be 10 digits	344
Figure 64: NID must be 10 digits	344
Figure 65: Enter valid email	344
Figure 66: Weak password error.....	344

Figure 67: Passwords does not match	344
Figure 68: City name must be letters only	345
Figure 69: CRN is not real error	345
Figure 610: CRN is not active error.....	345
Figure 611: Verify Email Page (Missing Email Dialog)	345
Figure 612: Email Verification Required Modal Dialog	345
Figure 613: Session Expired Dialog	346
Figure 614: Inactive Supplier Dialog.....	346
Figure 615: Inactive Supplier Can't Access Toast	346
Figure 616: Unauthorized Access Dialog	347
Figure 617: Unauthorized Access Dialog	347
Figure 618: Business Activity is Required (Settings Page Error).....	347
Figure 619: Supplier Favoriting Categories	347
Figure 620: Price is required.....	348
Figure 621: At least one image is required	348
Figure 622: Can't delete last image.....	348
Figure 623: Minimum order requirement must be of the case quantity	348
Figure 624: Maximum order requirement must be of the case quantity	348
Figure 625: Group purchase price must be less than standard price.....	349
Figure 626: Add at least one item to the invoice	349
Figure 627: Can't delete last item Toast.....	349
Figure 628: Must link the item.....	349
Figure 629: Invoice total price validation	350
Figure 630: Handle not found cases.....	350
Figure 631: Login Required Dialog	350
Figure 632: Wishlist Updated Dialog	351
Figure 633: Quantity must be in multiple of the case quantity	351
Figure 634: Validate quantity is not less than minimum order quantity.....	351
Figure 635: Validate quantity is not greater than stock	352
Figure 636: Auto-mark cart items out of stock (if applicable)	352
Figure 637: Can't checkout without saving a card first.....	352
Figure 638: Payment Callback Page (Not Successful)	353
Figure 639: Can't join the same group purchase twice	353

Figure 6-40: DigitalOcean Incident History	355
Figure 6-41: Droplet Activity History	355
Figure 6-42: Monitoring Graphs (CPU)	355
Figure 6-43: Monitoring Graphs (Load, Memory, Disk I/O)	355
Figure 6-44: Monitoring Graphs (Disk Usage, Bandwidth)	355
Figure 6-45: Corresponding draft stored in localStorage (keys: signUpForm and signUpStep)	356
Figure 6-46: Draft in localStorage (key format: invoice_draft_{invoiceId}).....	357
Figure 6-47: Unsent message stored in localStorage (key format: chat_draft_{conversationId})	357
Figure 6-48: Draft in localStorage (key: createBidForm).....	357
Figure 6-49: Corresponding entry in localStorage (key: bidOfferDraft).....	357
Figure 6-50: Draft in localStorage (key: newProductForm).....	358
Figure 6-51: Draft in localStorage (key: newServiceForm)	358
Figure 6-52: Draft in localStorage (key: review_draft_{orderId}).....	358
Figure 6-53: Landing page (1/2) On Chrome	360
Figure 6-54: Landing page (1/2) On Safari.....	360
Figure 6-55: Landing page (2/2) On Chrome	360
Figure 6-56: Landing page (2/2) On Safari.....	360
Figure 6-57: Homepage on Chrome	361
Figure 6-58: Homepage on Safari.....	361
Figure 6-59: Storefront (1/2) On Chrome	361
Figure 6-60: Storefront (1/2) On Safari	361
Figure 6-61: Storefront (2/2) On Chrome	361
Figure 6-62: Storefront (2/2) On Safari	361
Figure 6-63: Buyer Product Details on Chrome	362
Figure 6-64: Buyer Product Details on Safari.....	362
Figure 6-65: Cart on Chrome	362
Figure 6-66: Cart on Safari	362
Figure 6-67: Chats on Chrome.....	362
Figure 6-68: Chats on Safari	362
Figure 6-69: Invoices on Chrome	363
Figure 6-70: Invoices on Safari.....	363

Figure 6-71: Overview on Chrome	363
Figure 6-72: Overview on Safari	363
Figure 6-73: Listings on Chrome	363
Figure 6-74: Listings on Safari	363
Figure 6-75: Create a Service on Chrome.....	364
Figure 6-76: Create a Service on Safari.....	364
Figure 6-77: Create an Invoice on Chrome.....	364
Figure 6-78: Create an Invoice on Safari.....	364
Figure 6-79: Orders on Chrome.....	364
Figure 6-80: Orders on Safari	364

Acknowledgments

We, the Silah team, would like to express our heartfelt gratitude to everyone who has supported and guided us throughout this project.

A special thank you to our professors, Dr. Abeer Alarfaj and L. Fawziah Alqahtani, for their unwavering kindness, patience, and invaluable guidance. Their support, encouragement, and expertise have played a crucial role in the success of this project, and we are deeply grateful for their dedication to helping us achieve our goals.

We also want to thank Ayesha Rasheed, the talented designer behind our logo. Her creative vision and skill played a key role in shaping the visual identity of our platform.

Finally, we'd like to express our appreciation to our families and friends for their constant love, patience, and encouragement. Their belief in us kept us motivated, and this project wouldn't have been possible without their support.

Abstract

Silah is an AI-augmented B2B platform that addresses inefficiencies in supplier discovery, procurement, and market competitiveness across Saudi Arabia. The platform enables verified companies to engage in e-bidding, group purchasing, and multilingual product search with alternative recommendations. By leveraging Facebook Prophet and Language-agnostic BERT Sentence Embedding models, Silah enhances demand forecasting accuracy and product discovery in both Arabic and English. Built with a modular monolith architecture, the system ensures efficiency, maintainability, and role-based user experiences. Silah aims to support digital transformation efforts aligned with Saudi Vision 2030 through a secure and transparent procurement ecosystem.

Keywords: B2B platform, e-bidding, group purchasing, demand forecasting, LaBSE, FB Prophet, digital procurement.

Chapter 1 Introduction

1.1 Problem Statement & Significance

B2B e-commerce adoption in Saudi Arabia remains limited, with only 9% of commercial organizations utilizing digital platforms for business transactions [1]. Existing solutions such as Forsah and Etimad focus primarily on e-bidding and procurement, but they lack essential features like inventory management, real-time communication, and bulk purchasing. These functional gaps contribute to inefficient procurement, delayed supplier-buyer interactions, and limited scalability, ultimately slowing the Kingdom's digital transformation efforts.

Underlying these issues are several persistent challenges. Businesses often struggle to connect with verified suppliers [1], rely on outdated and manual procurement processes [2], and face forecasting inefficiencies that result in 20% higher stockout rates and 30% lower inventory turnover compared to AI-driven systems [3]. These barriers increase operational costs and reduce agility across supply chains.

At the same time, Saudi Arabia's e-commerce market is projected to grow significantly, reaching USD 49.49 billion by 2030 with a compound annual growth rate of 12.1% between 2025 and 2030 [4]. This anticipated growth highlights the urgent need for a more advanced B2B ecosystem, one that supports innovation, operational efficiency, and global competitiveness, in alignment with the goals of Saudi Vision 2030.

To address these gaps, our platform introduces AI-powered demand forecasting, integrated group purchasing, and real-time communication between buyers and suppliers. These features streamline procurement, enhance supplier relationships, and ease the transition to digital commerce. By targeting the root causes of inefficiency, our solution contributes to a more flexible, transparent, and digitally enabled B2B environment in line with national development goals.

1.2 Proposed Solution (System)

The proposed project aims to develop a B2B intermediary platform that connects businesses (buyers) with suppliers. The platform streamlines sourcing and purchasing by improving supplier discovery and offering tools to manage products and services more efficiently. With user-friendly interfaces, real-time communication, and advanced features such as bulk purchasing and e-bidding, the platform simplifies the

procurement process and creates a more transparent, collaborative experience for both buyers and suppliers.

As Saudi Arabia's B2B sector advances toward digital transformation, building an efficient digital platform requires a solid technical foundation. Our system uses modern web technologies to support seamless buyer-supplier interactions. HTTP enables reliable client-server communication, while WebSockets provide persistent, bidirectional connections essential for real-time features such as direct messaging.

The platform also integrates AI models for demand forecasting and product recommendation. These models analyze real-time and historical procurement data to generate insights, which are dynamically delivered to users through the interface. To handle time-sensitive workflows, such as group purchasing and bidding, the system uses automated background processes to monitor deadlines and trigger appropriate actions. For instance, a group purchase that fails to meet its participant requirement by the set deadline is automatically cancelled, while bidding deadlines ensure suppliers receive timely responses. These mechanisms support features like bulk purchasing collaboration, AI-powered suggestions of alternative products, and demand forecasting, ensuring efficient procurement and better business decisions.

By combining intelligent automation with real-time communication and smart decision support, the platform enhances market competitiveness, empowers small and medium enterprises (SMEs), and contributes to a more efficient and inclusive procurement ecosystem.

Aims (What We Intend to Learn & Apply):

- Understanding and applying full-stack web development, covering both front-end and back-end technologies, as well as core web development languages.
- Exploring B2B business models and market structures to understand how to build an effective intermediary platform.
- Applying architecture patterns and design patterns to improve performance and maintenance.
- Implementing an AI-powered alternative recommendation system to suggest alternative products, while learning how to prepare data and train models using fine-tuning techniques.
- Managing databases efficiently using SQL to ensure effective data storage and analysis.
- Exploring cloud deployment and web hosting techniques to ensure the platform is operational and secure.
- Enhancing knowledge of procurement systems and e-Bidding management to optimize digital purchasing processes.

Goals (What Will Be Delivered & Achieved):

- Developing a fully functional web platform that acts as an intermediary between businesses.
- Building an e-Bidding system that enables companies to submit RFPs and receive bids in an organized and transparent manner.

- Implementing a group purchasing feature to allow buyers to negotiate better deals collectively.
- Developing an “Alternative-Product” recommendation system to help businesses discover competitive alternatives to well-known products.

This approach ensures that we both learn key technologies and deliver a practical, high-impact solution in the B2B market.

1.3 Project Domain & Limitations

System Boundaries (Limitations):

To ensure focus and feasibility, the system will have the following limitations:

1. Accessibility: The system will not include specialized features for visually or hearing-impaired users.
2. Theme Mode: The system will support only light mode to ensure a more visually appealing experience.
3. Time Constraints: The system will be developed according to the deadlines set by PNU University.
4. Product & Service Availability: The range of available products and services will depend on the suppliers who join the platform.
5. Geographical Scope: The system is exclusively designed for businesses within Saudi Arabia and will not support international suppliers or shipping.

System Context:

The platform is a B2B digital system designed to connect businesses with suppliers within Saudi Arabia, enhancing transparency and competitiveness.

- Users:

- Buyers: Businesses seeking suppliers for products or services.
- Suppliers: Businesses offering products and services.

- Excluded Features:

- International suppliers or shipping.
- Integration with non-Saudi verification services.

System Environment:

The system will operate as a web-based platform, accessible through standard web browsers.

- Hosting & Deployment: The system is expected to be hosted on cloud for both frontend and backend.

- User Characteristics:

- Age Group: Between 25-60 years old (business owners, procurement officers).
- Education Level: Users are expected to have at least a basic understanding of procurement processes.
- Technical Skills: Since some users may lack advanced technical expertise, the interface will be designed to be user-friendly and intuitive.

1.4 The Methodology

This project followed the Waterfall Model, a linear development methodology with clearly defined phases: requirements, design, implementation, testing, and deployment. It was chosen because it aligns closely with the university's academic structure and the GP report chapters (GP1 and GP2), making it easier to meet fixed milestones and maintain clear documentation. While less flexible than Agile, Waterfall offers predictability and minimizes scope creep, making it well-suited for projects with a defined scope and timeline.

The key phases of the model are as follows:

1. Requirements Analysis: Identifying system needs through research.
2. System Design: Defining architecture, database structure, and user interfaces.
3. Implementation: Developing core platform features.
4. Testing: Validating functionality, usability, and reliability.
5. Deployment: Preparing the platform for cloud-based hosting.

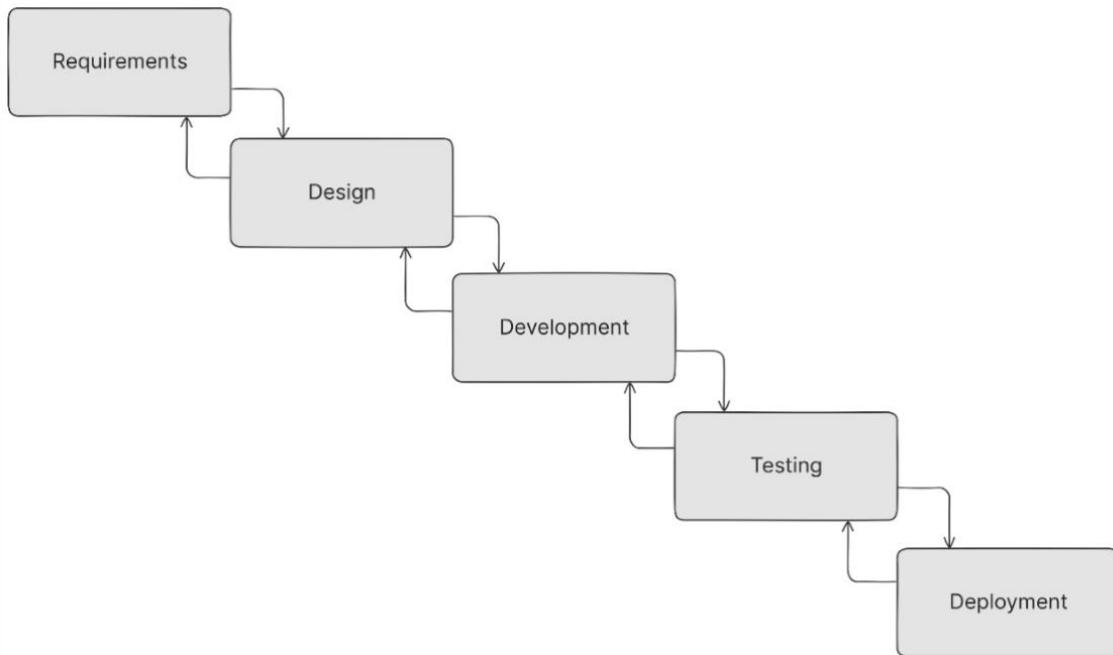


Figure 1-1: The Waterfall Model

1.5 Definitions of New Terms

Table 1-1: Terms and Definitions

Term	Definition
Supplier	An individual or business that provides wholesale products to retailers, distributors, or other companies. Suppliers act as intermediaries between manufacturers and businesses, offering goods in bulk for resale.
Buyer	A business or individual that purchases wholesale products from suppliers for resale or operational use. Buyers can include retailers, distributors, or companies seeking bulk procurement to meet business needs.
Business	Any operational entity involved in commerce – includes microbusinesses, online stores, and service providers, not just large corporations. In the context of our platform, Businesses can act as buyers (procuring goods for resale or operations)

	or suppliers (offering wholesale products to other businesses).
Product	any item listed for sale by a supplier, excluding services. It represents goods intended for wholesale transactions between businesses.
Service	Non-physical offering listed by suppliers (e.g., design, consulting). Treated as separate from 'products' in the system design.
Procurement	The process of acquiring goods or services from external suppliers, often involves negotiation, bidding, and supplier evaluation.
Bulk Discounts	Price reductions offered by suppliers when buyers purchase large quantities of products or services.
Group Purchasing	In Silah, group purchasing refers to a collaborative buying feature where multiple business buyers can collectively place orders for the same product to meet the supplier's minimum quantity requirement for bulk discounts.
Bid	In Silah, a bid refers to a supplier's offer submitted in response to a buyer's posted Request for Proposal (RFP), as part of the e-Bidding system.
Request for Proposal (RFP)	A document posted by buyers inviting suppliers to submit bids for a specific product or service.

API (Application Programming Interface)	A set of protocols and tools that allow different software systems to communicate. In our platform, APIs enable seamless data exchange between buyers, suppliers, and system components, facilitating real-time transactions, product listings, and automated supplier-buyer interactions.
Modular Monolith	A software architecture that organizes code into independent modules within a single deployable unit, combining the simplicity of monoliths with the modularity of microservices.
Semantic Similarity	The process of determining how closely two pieces of text (e.g., product descriptions) are related in meaning, used in Silah to recommend alternative products across languages.
Time-Series Forecasting	A method of predicting future data points by analyzing previously observed data trends over time. Silah applies time-series forecasting through FB Prophet to predict product demand.
AI (Artificial Intelligent)	The simulation of human intelligence in machines that are programmed to think, learn, and make decisions. In Silah, AI is used for tasks like demand forecasting and alternative product recommendations.
ML (Machine Learning)	A subset of AI that enables systems to learn from data and improve performance over time without being explicitly programmed. Silah uses ML algorithms to analyze procurement data and predict stock demand.
NLP (Natural Language Processing)	A field of AI focused on enabling machines to understand, interpret, and respond to human language. Silah uses NLP techniques to recommend semantically similar products across Arabic and English descriptions.

Chapter 2 Background Information & Related Work

This chapter provides a comprehensive review of existing literature and related systems relevant to our study. In Section 2.1, we established the necessary background information, defining key concepts and foundational technologies that underpin our work. Section 2.2 presents a survey of related research papers, summarizing their methodologies, findings, and contributions to the field. In Section 2.3, we investigate similar systems, examining their features and functionalities to compare them with our proposed solution. This comparative analysis allows us to highlight the strengths and shortcomings of current platforms and justify our approach in addressing unmet needs in the domain.

2.1 Background Information

Building a successful B2B platform requires more than just connecting buyers and suppliers, it needs a solid business strategy and a robust technical foundation. This section provides an overview of the business context, including the role of B2B platforms, supply chain dynamics, and gaps in the Saudi market. Additionally, it covers the technical foundation of our platform, highlighting the key technologies, architecture, and AI-driven features that enhance its functionality.

2.1.1 Business Background

1 Introduction to Business and Business Models

A business is an organization engaged in commercial, industrial, or professional activities, aiming to produce or provide goods and services to consumers for profit [5]. Businesses vary in size and structure, ranging from sole proprietorships to multinational corporations, and they operate across diverse industries.

Within this realm, business-to-business (B2B) refers to transactions conducted between companies, such as a manufacturer selling to a wholesaler or a wholesaler supplying a retailer. These transactions typically involve larger order volumes and structured procurement processes, distinguishing them from business-to-consumer (B2C) models [6].

While both B2B and B2C involve commercial transactions, they operate under different principles, sales strategies, and purchasing behaviors. B2B transactions often require multiple decision-makers and long-term agreements, whereas B2C sales are frequently impulsive and emotion-driven. Understanding these distinctions is crucial,

as they highlight why B2B platforms require specialized features that differ from consumer-focused marketplaces.

Table 2-1: Summarizes the key differences between B2C and B2B business models [7], [8]

Criteria	Business-to-Consumer (B2C)	Business-to-Business (B2B)
Target Audience	Individual consumers (general public)	Enterprises, businesses, and organizations
Market Size	Large, serves millions of consumers	Smaller, fewer but larger clients
Sales Volume	Low, individual transactions with small orders	High, bulk orders and recurring contracts
Decision Making	Fast, made by a single person or household	Longer, made by multiple stakeholders
Purchasing Process	Short and impulsive; driven by emotions	Complex, requires negotiation, approval, and contracts
Payment	Instant, paid via credit cards, cash, or online payments	May involve credit terms, invoicing, and delayed payments
Transaction Type	Simple, one-time purchases	Requires structured procurement and supply chain systems
Customer Relationship	Short-term, focused on one-time sales	Long-term, built on trust, service, and partnership
Branding & Marketing	Heavily relies on mass media and digital ads	Relies on direct sales, networking, and relationship-building

2 Supply Chain Management

Supply Chain Management (SCM) is a critical component of modern business operations, ensuring that products are delivered in the right quantity, at the right time, and at the right cost to maintain operational efficiency and customer satisfaction [9]. In a B2B environment, supply chains tend to be highly complex, involving multiple entities such as manufacturers, wholesalers, logistics providers, and retailers, which necessitate well-coordinated procurement and distribution strategies [10].

One of the primary challenges in SCM is managing supply chain complexity, as disruptions and inefficiencies can lead to inventory shortages, delayed deliveries, and increased costs [10]. Research suggests that poorly managed supply chains can significantly impact business profitability and lead to inefficiencies that reduce competitiveness in the market [11].

To maintain an efficient supply chain, businesses must accurately predict stock demand. While traditional forecasting methods rely on historical sales data and manual estimation, they often result in inaccuracies that lead to supply inefficiencies. According to Verma [3], AI-powered demand forecasting improves accuracy by integrating customer purchase histories, local events, and even social media sentiment analysis. This approach has led to a 20% reduction in stockouts and a 30% increase in

inventory turnover, helping companies better align inventory levels with actual demand. Additionally, companies utilizing AI have reported a 35% reduction in overall inventory levels, as real-time data minimizes the need for excess stock [3].

Our platform enhances supply chain efficiency by integrating AI-driven demand forecasting, allowing suppliers to anticipate stock needs, prevent shortages, and improve inventory management through data-driven insights and predictive analytics. By leveraging these technologies, companies can reduce waste, improve decision-making, and enhance overall supply chain resilience.

2.1.2 Technical Background

1 Web Communications

HTTP (Hyper Text Transfer Protocol)

HTTP is a stateless communication protocol, meaning that each request sent by a client is processed independently, without the server retaining any memory of previous requests. This characteristic simplifies communication, making HTTP lightweight and scalable, but it also requires additional mechanisms, such as cookies, sessions, or tokens, to maintain user-specific data.

HTTP follows a request-response model, where a client (such as a web browser) sends a request to a server, which processes the request and returns a response. For example, when a user enters a website URL in their browser, the browser sends an HTTP request to the server. The server retrieves the requested webpage, sends it back in an HTTP response, and the browser then renders the webpage for the user.

Similarly, when a user submits a form -such as signing up on a website- an HTTP request is made to store the user's information on the server. However, since HTTP is stateless, the server does not automatically remember the user's session, requiring external storage solutions to maintain user authentication or shopping cart data.

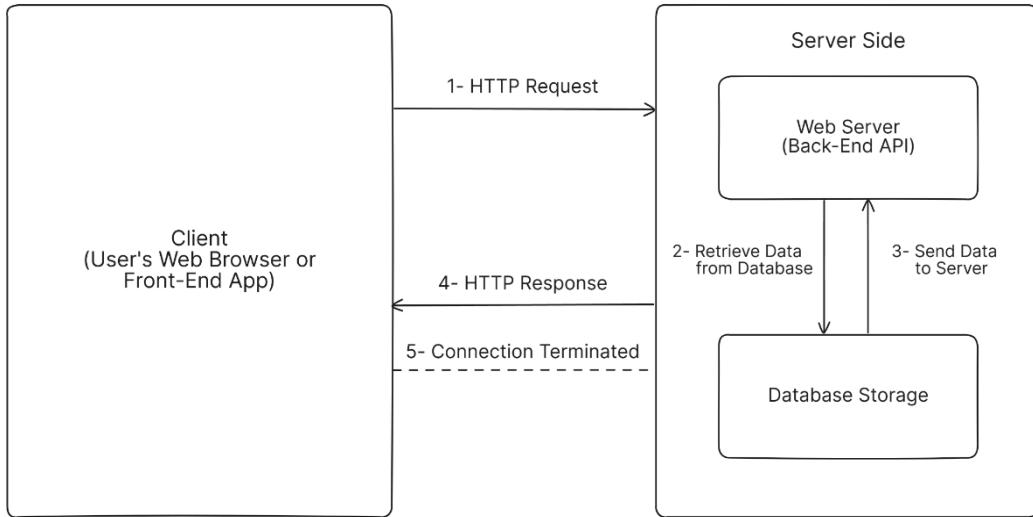


Figure 2-1: HTTP Request-Response Process in a B2B Platform [12]

In our project, HTTP serves as the foundation for data exchange and platform accessibility. The backend is designed to handle API requests efficiently, ensuring a structured approach to client-server communication. HTTP methods facilitate key platform functionalities, with the GET method allowing buyers and suppliers to retrieve product listings and supplier details. Methods like POST and PUT enable data submission and updates, allowing users to send inquiries, update product details, and manage orders effectively.

WebSockets

WebSockets provide a persistent, full-duplex communication channel between the client and server, enabling real-time data exchange. Unlike HTTP, which follows a request-response model, WebSockets establish a continuous connection, allowing both the client and server to send and receive messages at any time without reopening a connection. This eliminates the need for repeated requests, making WebSockets ideal for real-time applications.

To initiate a WebSocket connection, the client first sends an HTTP request to the server, known as the WebSocket handshake. If the server accepts, it upgrades the connection from HTTP to WebSockets, enabling a continuous, bidirectional communication channel. Once established, the client and server can exchange messages without requiring new requests, unlike HTTP, where each interaction requires a separate connection.

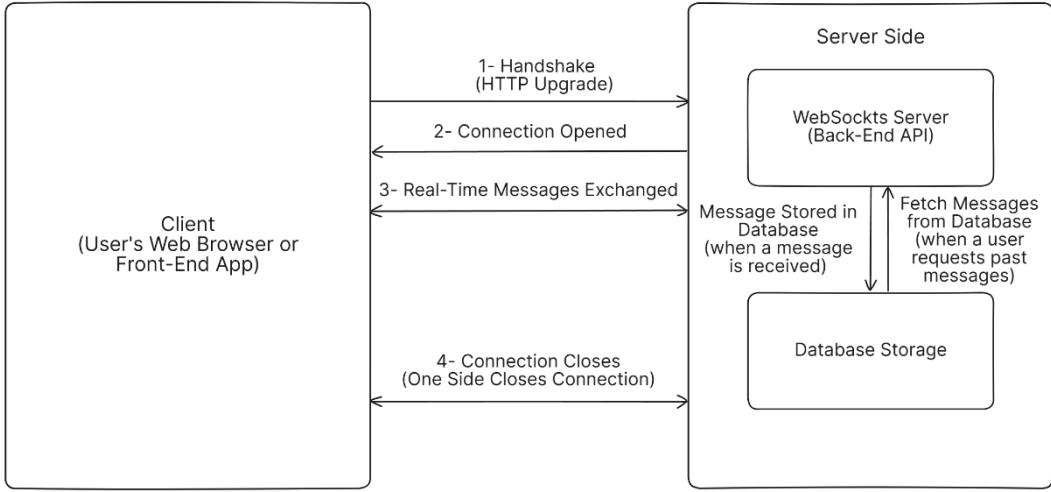


Figure 2-2: Establishing a WebSocket Connection in a Real-Time Messaging System [13]

In our B2B platform, WebSockets play a crucial role in Direct Messaging (DM) between buyers and suppliers. Since business negotiations often require instant communication, WebSockets allow users to send and receive messages in real-time without refreshing the page. This ensures a seamless chat experience without the inefficiencies of traditional server polling, where clients must repeatedly send requests to check for new messages. By using WebSockets, our platform optimizes performance, reduces unnecessary network traffic, and enhances real-time collaboration between businesses.

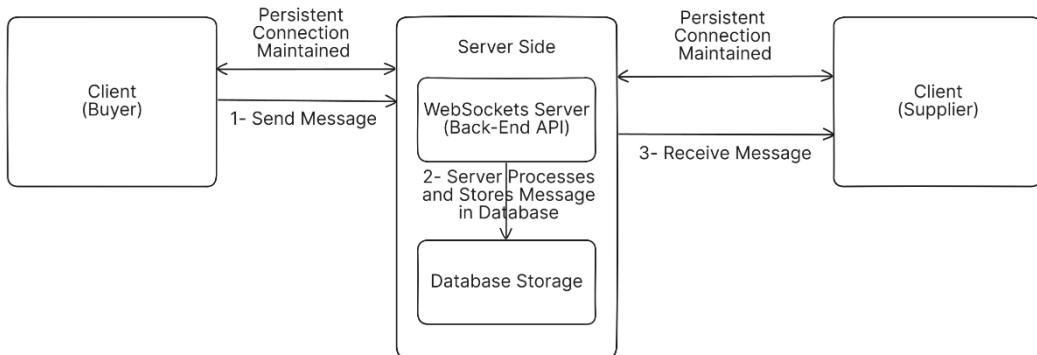


Figure 2-3: WebSocket-Based Real-Time Messaging Between Buyers and Suppliers

2 System Architecture

Selecting the right architecture for our website is a critical decision, as it directly impacts development efficiency, scalability, system performance, and cost-effectiveness. To build a robust and maintainable platform, we opted for the Modular Monolith architecture, as it strikes a balance between simplicity, modularity, and future scalability.

The following diagram (Figure 2-4) illustrates the key differences between Monolithic, Modular Monolith, and Microservices architectures in terms of

deployment and modularity. This visual representation helps contextualize our decision before diving into the detailed comparison in Table 2-2.

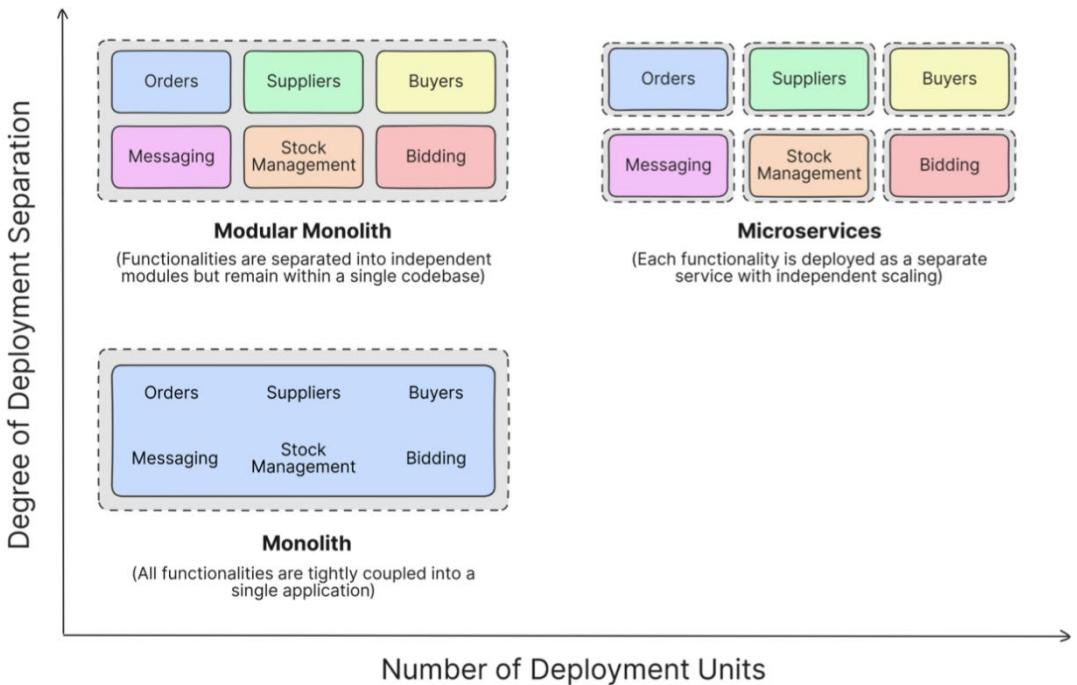


Figure 2-4: Architectural Styles Comparison, based on the degree of deployment separation and modularity [14]

Building on this visual representation, Table 2-2 provides a more detailed comparison of these architectural styles, highlighting differences in scalability, maintainability, and complexity based on Skalicky's analysis [15].

Table 2-2: Comparison of Architectural Styles [15]

Criteria	Monolithic	Modular Monolith	Microservices
Structure	Single, tightly coupled application.	Multiple loosely coupled modules within a single application.	Fully independent microservices.
Communication	Direct function API calls.	Direct function API calls between modules.	Network-based communication (API calls).
Scalability	Add instance of whole application.	Add instance of individual modules.	Scale specific microservices independently.
Architecture Overhead	Minimal.	Moderate.	Higher due to distributed system complexity.
Maintainability	Modifications affect the entire application.	Modifications are isolated to specific modules.	Modifications impact only one microservice.

Complexity	Higher due to tightly coupled components.	Lower complexity but still requires modular structure.	High complexity due to interdependencies.
Collaboration	Everyone works on one codebase.	Teams collaborate within their assigned modules.	Independent teams manage separate microservices.
Deployment	Entire application must be deployed as a single unit.	Modules can be deployed independently but remain in one application.	Independent deployment of microservices.

The Modular Monolith architecture allows us to develop independent modules within a single codebase, making it easier to distribute tasks among team members, which is a crucial advantage given our tight schedule. Additionally, this architecture ensures cost-efficient deployment, as the platform can initially run on a single server, reducing infrastructure expenses. Unlike Microservices, which require multiple services to be deployed and managed separately, a Modular Monolith enables us to minimize operational overhead while keeping future scalability options open.

One of the key benefits of this approach is its seamless transition to Microservices when needed. If the platform encounters increased traffic, we can deploy specific modules independently and integrate them using a load balancer, allowing for incremental scalability. This method aligns with industry best practices, where systems are initially designed as Modular Monoliths and gradually evolve by extracting critical modules into Microservices as necessary.

To implement this architecture, we are using a modular backend framework that supports scalability and long-term growth. This framework enforces modularity and allows us to structure the platform in a way that supports future development needs. Its built-in tooling and strict coding conventions help maintain a well-structured, maintainable codebase, ensuring that we follow the best development practices throughout the project.

3 AI-Driven Enhancements for B2B

3.1 Stock Demand Forecasting

To help suppliers make informed inventory decisions, our platform integrates a time-series forecasting tool that predicts product demand over the next three months. For this purpose, we selected Facebook Prophet [16], a widely recognized open-source forecasting model developed by Meta. Prophet is designed to handle time-series data with minimal configuration, making it ideal for businesses without deep expertise in data science or machine learning.

Prophet models time-series data using an additive approach, breaking it into components such as trend, seasonality, and an error term. The general formula used

by Prophet is:

$$y(t) = g(t) + s(t) + e(t)$$

In the formula, $g(t)$ represents the overall trend over time (such as increasing or decreasing sales), $s(t)$ captures recurring seasonal patterns (like monthly demand spikes), and $e(t)$ accounts for irregular variations or noise in the data.

This structure allows the model to provide interpretable forecasts and adapt to various types of demand behavior. Since our platform focuses on historical sales data only, we use Prophet in its most basic form, without incorporating external factors such as holidays or promotional events, keeping the forecasting process simple and efficient.

Prophet was chosen for several reasons: its ease of use, which reduces development effort and complexity; its ability to automatically handle missing data and outliers, ensuring accuracy even with imperfect data; and its interpretable visualizations, which allow for easy understanding of the results.

In our platform, each supplier's past sales records are used as input to generate demand forecasts tailored to their product offerings. The model processes this data to identify historical patterns and predict future demand levels. These forecasts are then displayed on the demand forecast page, helping businesses adjust inventory in advance, minimizing the risks of overstocking or understocking.

As illustrated in Figure 2-5, stock demand may follow various time-series patterns, such as linear trends, seasonal fluctuations, or random noise. Prophet is capable of modeling these patterns and producing accurate forecasts, even in the presence of irregularities or shifting trends.

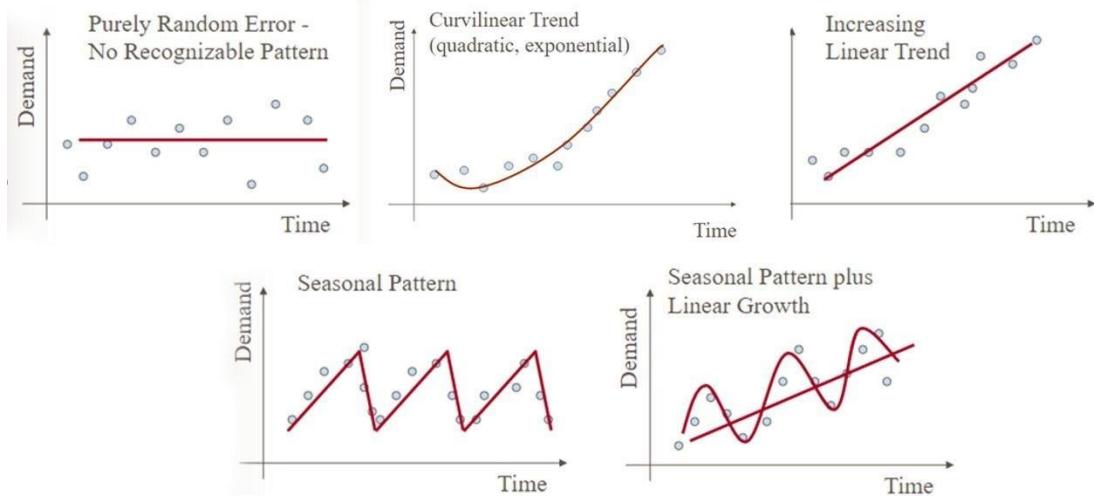


Figure 2-5: Common Time Series Patterns [17]

By leveraging Prophet's forecasting capabilities, our platform improves supply chain efficiency and helps suppliers make proactive, data-driven stocking decisions. This contributes to a more resilient and responsive procurement ecosystem, ensuring product availability while minimizing waste and uncertainty.

3.2 Alternative Product Recommendation System

In modern B2B e-commerce platforms, helping buyers discover suitable product alternatives is essential for improving procurement efficiency, encouraging supplier diversity, and supporting informed decision-making. Traditional recommendation systems typically rely on user behavior and purchase history, which may not suit the precise and objective nature of B2B procurement. In contrast, our platform introduces a targeted, transparent approach based on semantic similarity between product descriptions.

Our system features an AI-powered alternative product recommendation tool, accessible on the "Alternative Products" page. Buyers can enter the name of a product they are interested in, and the system returns a list of relevant alternatives offered by different suppliers. This functionality relies on semantic text similarity rather than behavioral data, ensuring that recommendations are based on product relevance and not previous user activity.

By adopting a semantic-based approach, our platform enhances product visibility, promotes supplier diversity, and helps buyers make more informed procurement decisions. This method allows buyers to identify alternatives based on meaning and functional similarity rather than on usage patterns, ultimately supporting a fair and competitive marketplace.

We chose semantic similarity over behavior-based recommendations to better reflect the nature of B2B transactions. Unlike B2C platforms, where user preferences are often inferred through interaction history, B2B buyers are more focused on finding specific items and discovering alternative suppliers. A semantic approach ensures precise and meaningful results that align with buyer intent, especially in domains with standardized product descriptions.

When a buyer searches for a product, the system retrieves its description and applies Natural Language Processing (NLP) techniques to understand its semantic meaning. This is achieved using LaBSE (Language-Agnostic BERT Sentence Embedding), a multilingual, pre-trained sentence embedding model that supports both Arabic and English.

LaBSE is built on top of BERT (Bidirectional Encoder Representations from Transformers), a transformer-based deep learning architecture developed by Google. BERT is capable of understanding language by processing input in both directions (left-to-right and right-to-left), which allows it to grasp the context and meaning of words more effectively than earlier models [18].

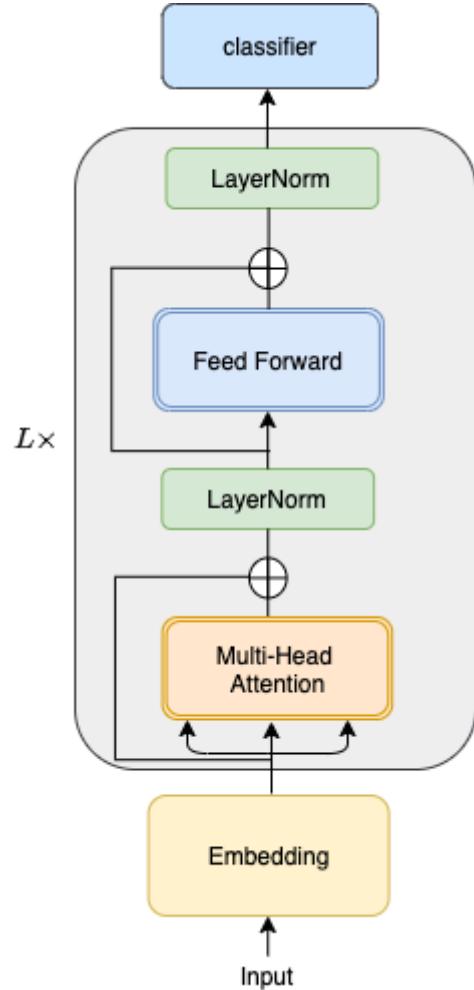


Figure 2-6: The BERT model architecture [18]

LaBSE extends BERT by incorporating a dual-encoder structure that processes sentence pairs in different languages. It uses shared parameters and is fine-tuned using a translation ranking loss with an additive margin, enabling it to produce aligned embeddings across languages. This allows our system to match product descriptions between Arabic and English without needing separate models [19].

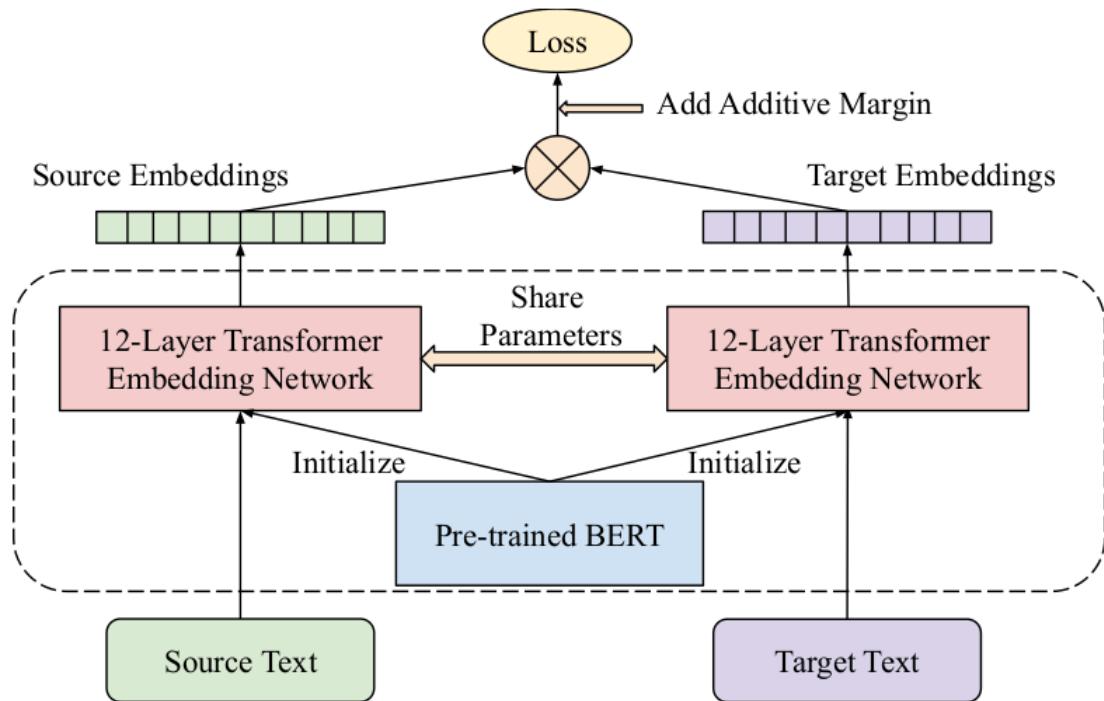


Figure 2-7: Dual encoder model with BERT based encoding modules [19]

Compared to traditional methods like TF-IDF or bag-of-words, which rely on exact word matches, LaBSE captures deeper semantic meaning. For instance, a search for "آلة صنع الإسبريسو" (coffee machine) can still return results like "ماكينة قهوة" (espresso maker), even though the words differ.

To rank the most relevant alternatives, we apply Cosine Similarity, a mathematical method that quantifies how close two text vectors are by measuring the angle between them. Product descriptions are converted into dense numerical vectors using LaBSE, and then compared using this formula:

$$\text{sim}(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

Where A and B represent the LaBSE-generated vectors, and the angle between them determines how similar the products are. A value near 1 indicates high similarity, while values closer to -1 suggest opposing meanings.

As shown in Figure 2-6, when the angle θ between two vectors is close to 0° , the Cosine Similarity value is close to 1, indicating a highly similar product. Conversely, when θ is close to 180° , the Cosine Similarity value approaches -1, indicating that the products have opposite meanings.

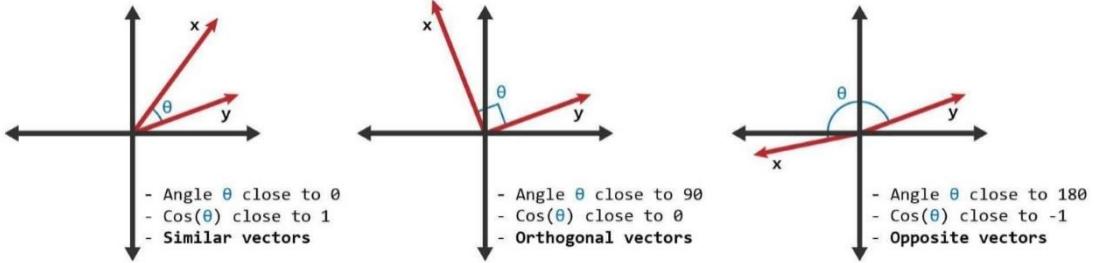


Figure 2-8: Cosine Similarity Representation [20]

By combining LaBSE embeddings with cosine similarity scoring, our system provides accurate, scalable, and language-aware product recommendations. This empowers buyers to efficiently explore meaningful alternatives across a diverse supplier base and contributes to a more inclusive and competitive B2B ecosystem.

2.2 Related Work Survey

To build a well-informed foundation for our project, this section surveys relevant research studies that explore various aspects of our domain. By analyzing previous work, we aim to understand the methodologies, technologies, and architectural choices that have been proposed and implemented in similar contexts. This review allows us to identify recurring patterns, best practices, and common challenges faced in the field. Additionally, by evaluating the limitations and gaps in existing research, we can better position our project and refine our approach to address these shortcomings effectively. The insights gathered from this survey will contribute to the justification and refinement of our proposed solution.

1 System Architecture in B2B E-Commerce Platforms

One significant study in this area is *Development of a Marketing Automation Platform to Integrate Online E-Commerce Services* (2022) [21], which focuses on designing a scalable B2B e-commerce platform using microservices architecture. The study highlights how modularity and scalability improve business transactions and supplier management, leveraging technologies such as Nest.js for the backend and Vue.js for the frontend. It also incorporates MongoDB and Redis for data management [21]. While this approach enhances efficiency and modularity, the study acknowledges security risks and the absence of AI-driven optimizations as key limitations [21]. These insights influenced our architectural choices, leading us to adopt Nest.js for our backend while integrating AI-driven features to enhance automation and decision-making processes.

Another relevant study, *Analysis and Comparison of Application Architecture: Monolith, Microservices, and Modular Approach* (2024) [15], provides a comparative analysis of monolithic, microservices, and modular monolith (Modulith) architectures. The study critiques the excessive complexity of microservices and presents Modulith as a balanced alternative that retains modularity while minimizing deployment and management overhead [15]. The findings reinforce our decision to adopt modular monolith architecture, ensuring scalability while maintaining efficiency and ease of management.

2 AI & Machine Learning in Demand Forecasting

Several studies explore the role of artificial intelligence (AI) and machine learning (ML) in demand forecasting.

Review and Analysis of Artificial Intelligence Methods for Demand Forecasting in Supply Chain Management (2022) [22] evaluates AI-based forecasting models, demonstrating that deep learning techniques such as Long Short-Term Memory (LSTM) and Bidirectional LSTM (BLSTM) achieve the highest accuracy rates (92-97%), outperforming traditional statistical approaches. Other models like Artificial Neural Networks (ANN), Multilayer Perceptron (MLP), and ensemble techniques such as XGBoost and Random Forest (RF) also exhibit strong predictive capabilities [22]. However, challenges such as high computational costs and model interpretability remain key concerns.

Similarly, *Machine Learning Demand Forecasting and Supply Chain Performance* (2020) [23] investigates ML-based forecasting techniques, comparing ARIMAX models with Neural Networks. The results indicate that ML significantly improves accuracy over traditional time-series forecasting methods, with Neural Networks achieving the highest accuracy (89.4%) [23]. The study underscores the need for high-quality data and computational resources to effectively implement these techniques.

Predictive Analytics in Supply Chain Management: The Role of Artificial Intelligence and Machine Learning in Demand Forecasting (2024) [24] builds on these insights, demonstrating that AI-driven forecasting can reduce inventory holding costs by 30% and improve forecasting accuracy by 20% compared to traditional methods. However, the study notes that high implementation costs may limit adoption, particularly for small businesses [24].

A complementary study, *Reinforcement Learning for Adaptive Supply Chain Forecasting* (2023) [25], investigates reinforcement learning (RL) techniques to improve adaptability in supply chain forecasting. Using Deep Q-Networks (DQN) and event-based simulations, RL-based models successfully reduced supply chain disruptions by 25% and improved supplier selection by 40%. Despite its effectiveness, RL requires continuous model training and high computational resources, making it more applicable to large enterprises [25].

3 Tree-Based Models vs. Deep Learning in Demand Forecasting

Two studies compare tree-based ensemble models and deep learning techniques for demand forecasting in retail.

Applying Machine Learning in Retail Demand Prediction—A Comparison of Tree-Based Ensembles and Long Short-Term Memory-Based Deep Learning (2023) [26] investigates tree-based models (Extra Trees Regressor, XGBoost, Random Forest) and LSTM networks for demand forecasting using six years of sales data from an Austrian retailer. The findings indicate that Extra Trees Regressor (ETR) outperforms LSTM, achieving the lowest MAPE, MAE, and RMSE, particularly for fresh meat products [26]. The study recommends tree-based models over LSTM, as they offer higher accuracy with less preprocessing.

A related study, *Demand Prediction Using Sequential Deep Learning Model* (2023) [27], introduces a hybrid deep learning architecture combining 1D CNN and Bidirectional LSTM to capture short- and long-term sales patterns. The model demonstrates high forecasting accuracy but is not tested in real-time applications, raising concerns about its adaptability in dynamic retail environments [27].

4 Meta-Learning for Demand Prediction in E-Commerce

Relation-aware Meta-learning for E-commerce Market Segment Demand Prediction with Limited Records (2021) [28] presents a meta-learning framework (RMLDP) designed to transfer knowledge from data-rich to data-poor segments. The model integrates a Multi-Pattern Fusion Network (MPFN) and a Knowledge Graph, improving demand forecasting in low-data e-commerce segments. Experiments on Juhuasuan and Tiantantemai datasets confirm that RMLDP outperforms traditional models like XGBoost and LSTM, leading to better order rates when deployed on Taobao [28]. Despite its success, performance depends on segment relationships in the knowledge graph, and computational overhead is a challenge.

The following table compares the AI research papers discussed, highlighting their datasets, techniques, accuracy, key findings, and limitations. This comparison helps identify the most effective approaches for demand forecasting in supply chain management and informs the selection of AI methodologies for our platform.

Table 2-3: Research Papers Comparison

Study (Year and Ref.)	Dataset Used	Accuracy	Techniques Applied	Key Findings	Limitations
AI Methods for Demand Forecasting in SCM (2022) [22]	Analyzed multiple datasets from referenced studies [6,22–25,29,33,34], but no specific dataset collected.	92-95%	Deep learning methods, including LSTM, MLP, and ANN, combined with statistical approaches.	LSTM is highly effective for demand forecasting, especially with multivariate datasets. Hybrid models (AI + statistical) further enhance accuracy.	No primary dataset; relies on literature review. Challenges in AI-driven forecasting include data quality, computational cost, and lack of collaborative forecasting methods.
Machine Learning Demand Forecasting and Supply Chain Performance (2020) [23]	Observational data from a steel manufacturing company (2012-2017), including monthly sales	89.4%	Hybrid forecasting models: - ARIMAX (1,1,0) - ARIMAX (3,0,0) - Neural Network (best model)	NN outperformed ARIMAX and traditional models, achieving the highest accuracy (89.4%).	Only one company's dataset is used (steel industry). Not tested in fast-changing industries like fashion or tech.

	volume, macro-economic indicators (30 variables), and financial performance data (inventory levels, revenue, cost of goods sold, etc.).			Hybrid models (time series + macroeconomic factors) improved forecast accuracy. A 5% accuracy boost led to better inventory management and supply chain efficiency.	Results may not generalize to heterogeneous datasets.
AI & ML in Demand Forecasting for SCM (2024) [24]	The study does not use a specific dataset but conducts a systematic literature review analyzing various datasets from previous research in supply chain demand forecasting.	85-95%	Predictive analytics, deep learning, reinforcement learning, IoT integration, and big data analytics enhance demand forecasting by processing real-time and historical data, capturing complex patterns, and adapting to supply chain fluctuations.	AI-driven forecasting improves accuracy, enables real-time insights, automates supply chain operations, enhances sustainability by reducing waste, and provides a competitive edge through better market resilience.	Challenges include data quality issues, model interpretability (black-box nature), high implementation costs, privacy and ethical concerns, and limited generalization in unpredictable disruptions like pandemics.

Reinforcement Learning for Adaptive Supply Chain Forecasting (2023) [25]	Various datasets, including historical sales data, real-time market trends, and industry-specific data (e.g., C3 AI, Amazon, Quantzig, SolveIT Software).	The study is a literature review and does not conduct experiments to measure accuracy.	RL-based event simulations, OpenAI Gym.	Reduces disruptions by 25%, improves supplier selection by 40%.	Continuous training & high computing power required.
Machine Learning in Retail Demand Prediction (2023) [26]	6 years of Austrian retailer sales data (5.2M entries)	55-60%	Tree-based models vs. Deep Learning (LSTM)	ETR outperforms LSTM for demand forecasting.	Single retailer dataset, external demand factors not considered.
Meta-Learning for E-commerce Demand Prediction (2021) [27]	Two large-scale real-world datasets from e-commerce platforms, Juhuasuan (China-based group-buying platform,	For Juhuasuan: 73-77% (Estimated R ²)	MPFN (GRUs) + MAML Meta-learning	RMLDP outperforms traditional models, deployed on Taobao.	Model depends on knowledge graph relationships, high computational cost.

	4000+ segments), Tiantiantemai (low-cost product marketplace, 6000+ segments)	For Tiantiantemai: 71-72% (Estimated R ²)			
Sequential Deep Learning for Demand Prediction (2023) [28]	“Store Item Demand Forecasting Challenge” dataset from Kaggle, which consists of historical sales data from a chain of stores, comprising 913,000 rows with features including “date,” “store,” “item,” and “sales”	94% (Estimated R ²)	Deep learning for capturing short/long-term trends	High accuracy in sales forecasting.	No real-time validation, dataset from a single company.

2.3 Proposed & Similar Systems Comparison

To better position our platform within the existing landscape, this section examines similar systems and compares their features and functionalities with our proposed solution. By identifying key differentiators, we highlight how our approach addresses gaps and limitations in existing solutions. This comparative analysis helps establish a clear rationale for our proposed system and demonstrates its potential advantages over current alternatives.

1. Faire

F A I R E

Faire is a wholesale B2B marketplace connecting small retailers with suppliers, offering access to a diverse range of quality products at discounted prices. This platform simplifies bulk purchasing, enabling retailers to stock unique items while helping brands expand their reach in the market.

Faire offers a range of features for both buyers and sellers, enhancing communication, transactions, and inventory management. Shared functionalities include in-app messaging and secure transactions with fraud detection. Sellers benefit from customizable store pages, easy product listing updates, batch editing, and sales performance tracking. Buyers enjoy AI-powered recommendations, advanced filtering, and the "buy now, pay later" option for greater purchasing flexibility.

Advantages of Faire:

- Faire offers retailers access to a diverse range of products.
- Low minimum order requirements make it easier for small businesses to start wholesale.
- The platform is user-friendly, providing smooth navigation and ordering experience.
- Faire has a large buyer base, connecting sellers with thousands of retailers worldwide.
- It helps businesses save costs by eliminating the need for expensive trade shows and sales agents.
- The platform increases efficiency by reducing manual outreach, allowing businesses to scale faster.
- Sellers can expand their market reach, accessing international and underrepresented regions.
- Faire integrates with Shopify, making inventory management

Disadvantages of Faire:

- Faire's pricing is higher compared to traditional wholesale markets, which may impact affordability for retailers.
- Sellers depend on supplier stock availability and shipping times, which can cause shipping and delivery delays.
- Building brand loyalty is difficult since buyers often associate their purchases with Faire rather than the individual brand.
- Faire operates with a retailer-focused model, prioritizing buyers over sellers, which can sometimes affect profitability for sellers.
- Repeat business is uncertain, as many first-time buyers do not place additional orders, making long-term revenue generation challenging.
- Competition is intense, with many brands offering similar products,

- across multiple sales channels seamless.
 - There is no upfront cost to join, allowing businesses to try wholesale with minimal risk.
- making it harder for sellers to stand out.

2. Wondda



Wondda is a platform for private label and contract manufacturing, connecting brands with verified suppliers. It streamlines product sourcing, supply chain management, and procurement, helping businesses efficiently launch and expand their product lines.

The platform offers a product discovery tool for browsing thousands of private-label product ideas and supply chain management features to simplify procurement and supplier interactions. It also serves as a collaborative hub for seamless team and partner communication. Additionally, its smart quoting system allows users to request and compare quotes from multiple suppliers, ensuring competitive pricing and informed decision-making.

Advantages of Wondda:

- Access to a network of over 20,000 verified suppliers.
- Ability to receive multiple quotes from different manufacturers.
- Saves time and effort with centralized supply chain management.
- It helps improve profit margins by securing competitive pricing.

Disadvantages of Wondda:

- Primarily focused on the European market, which may limit access for businesses in other regions.
- The platform does not integrate with external business systems, making it difficult for suppliers to keep their product information synchronized across multiple websites.

3. Etimad



Etimad is a digital platform for managing government tenders and procurement. It automates bid issuance, bid evaluation, and contract awarding, enabling government entities to post tenders efficiently while businesses submit bids seamlessly. With a user-friendly interface, it ensures transparency and compliance with regulations.

The platform offers bid creation, bid submission, and contract awarding with automated tools for efficiency. It includes audit and compliance monitoring, multi-level approvals, and secure access and authentication to enhance accountability and data protection. Notifications and real-time tracking keep stakeholders informed throughout the procurement process.

Advantages of Etimad:

- Automated & Transparent: Reduces manual paperwork and ensures fair competition.
- Timesaving: Speeds up the procurement process for both government and suppliers.
- Compliance-Oriented: Ensures adherence to government procurement policies.
- User-Friendly Interface: Easy navigation and structured workflow.
- Document Management: Secure handling of all procurement-related documents.

Disadvantages of Etimad:

- Access Restrictions: Primarily for government entities and registered suppliers.
- Learning Curve: New users may require training to utilize all features effectively.
- Dependence on System Availability: Any technical downtime could delay processes.

4. Forsah



Forsah is a Saudi-based B2B platform that connects Small Medium Enterprises (SMEs) with buyers in the public and private sectors. It facilitates electronic bidding and procurement, enabling businesses to submit and manage price offers digitally. The platform enhances market transparency and provides tools like feasibility studies, business model support, and commercial consultations to help companies grow.

Forsah offers digital bidding, document management, and secure payment systems to streamline procurement. It improves market transparency, helping businesses increase visibility and competitiveness. The platform also provides business tools and sends notifications and alerts to keep users informed about deadlines and new opportunities.

Advantages of Forsah:

- User-friendly design that simplifies digital procurement and bidding processes.
- Seamless interaction between SMEs and large buyers in both public and private sectors.
- Transparent market access, providing equal opportunities for businesses.
- Access to valuable business development tools (feasibility studies, business model support, etc.).

Disadvantages of Forsah:

- Limited to businesses within Saudi Arabia, restricting international access.
- May require additional training for users unfamiliar with digital bidding processes.
- Some businesses may face challenges adapting to the platform if they are not digitally savvy.

- Enhances efficiency and saves time compared to traditional procurement methods.

5. Alibaba



Alibaba is a global B2B e-commerce marketplace that connects businesses with suppliers across various industries. It facilitates wholesale trade, allowing businesses to source bulk products at competitive prices. The platform ensures secure transactions, supplier verification, and logistics support for smooth operations.

Alibaba offers in-app messaging, multi-currency transactions, bulk order customization, supplier verification, and logistics tracking for international trade. Sellers benefit from customizable storefronts, AI-powered demand forecasting, inventory management, and escrow payment protection. Buyers gain access to AI-driven recommendations, advanced search filters, custom bulk order requests, buyer protection policies, and trade financing for flexible payments.

Advantages of Alibaba:

- Extensive product variety from global suppliers
- Competitive pricing with bulk discounts
- Verified suppliers for secure transactions
- Logistics and shipping support
- AI-powered search and product recommendations

Disadvantages of Alibaba:

- High competition among sellers
- Potential for long shipping times on international orders
- Some suppliers require large minimum order quantities
- Quality variation among different suppliers

The following table compares the previously discussed platforms with ours, highlighting key features, target audiences, and regional focus. This comparison helps identify the unique advantages of our platform and how it differentiates itself from existing solutions in the market.

Table 2-4: Related Websites Comparison

	 Silah	FAIRE Faire	 Wondda	 Etimad	Forsah	 Alibaba
Target Audience	SMEs	Small Retailers	SMEs	Government institutions, Suppliers and companies, SMEs & Large corporations	SMEs and Large Enterprises	Businesses, wholesaler, large-scale retailers
Primary Region	KSA	USA	Europe	KSA	KSA	China
e-Bidding	✓	✗	✓	✓	✓	✓
Group Purchase	✓	✗	✗	✗	✗	✗
Find Products Similar-to	✓	✗	✗	✗	✗	✓
Forecasting Stock Demand	✓	✗	✗	✗	✗	✓

Chapter 3 System Analysis

This chapter provides an analysis of the system requirements. In Section 3.1, we discuss the process of gathering and documenting the system requirements, outlining the key features and specifications that the platform must meet. In Section 3.2, we analyze these requirements through the creation of use cases, sequence diagrams, and class diagrams, which help in visualizing and refining the platform's functionality. This chapter ensures that the system's features are well-defined and aligned with user needs, laying the foundation for the design and architecture discussed in the next chapter.

3.1 Requirements Specification

3.1.1 Requirements Gathering

Gathering requirements was a crucial phase in shaping our platform to meet the needs of both end-users and business stakeholders. We began by analyzing similar platforms and reviewing user feedback to identify common challenges and user expectations, which helped shape our initial system vision.

To validate these findings and better understand our target users, we conducted a bilingual survey (Arabic and English) targeting both suppliers and buyers in the B2B space. The survey explored business roles, procurement behaviors, and feature preferences. Although response numbers were limited, the data provided valuable insights into real-world pain points and expectations. For consistency, responses are presented in English. The full list of survey questions is available in Appendix A.

Survey Analysis

The graphs below illustrate the most relevant patterns observed in the survey. Each chart is followed by a concise analysis explaining how the responses informed our design decisions.

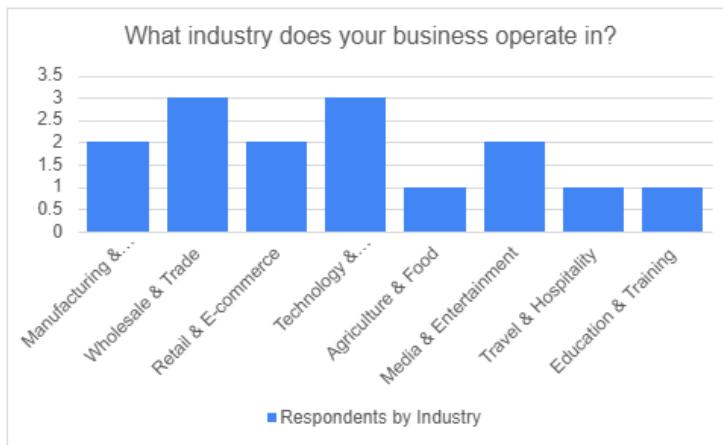


Figure 3-1: Distribution of Industries Represented in the Survey

Most respondents came from the Wholesale, Technology, and Retail sectors, emphasizing the need for product catalogs, order management, and product discovery features.

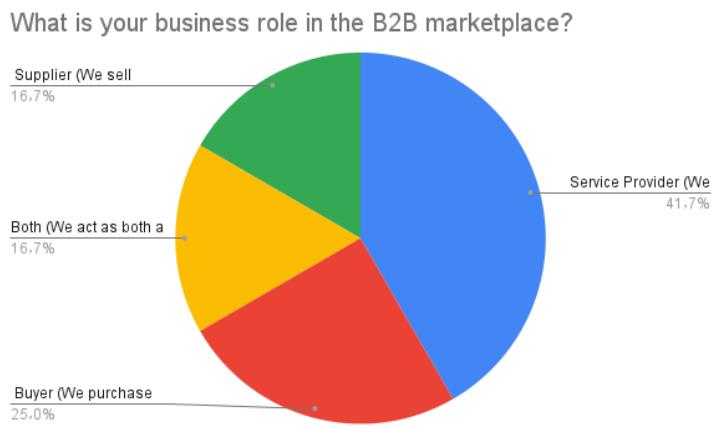


Figure 3-2: Respondent Roles in the B2B Marketplace

A majority were service providers, reinforcing the need for service management tools, while strong buyer and supplier representation supports dual-role platform features.

How often does your business buy or sell products/services?

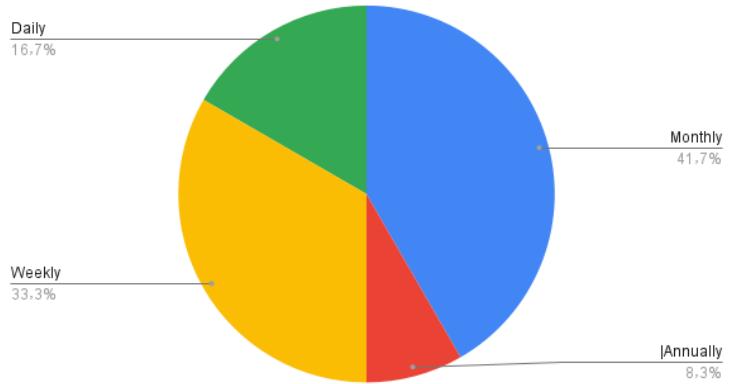


Figure 3-3: Frequency of Transactions in the B2B Marketplace

Monthly and weekly transactions were most common, highlighting the need for bulk order handling, inventory tools, and timely notifications.



Figure 3-4: Current Procurement Methods Used by Businesses

Many still rely on manual methods or existing platforms, supporting the shift toward digital procurement with direct supplier connection features.



Figure 3-5: Key Procurement Challenges in the B2B Marketplace

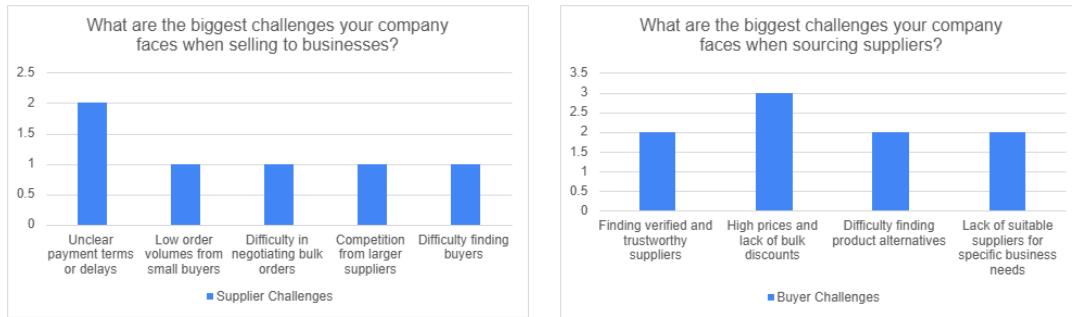


Figure 3-6: Key Challenges Faced by Suppliers When Selling to Businesses

Figure 3-7: Key Challenges Faced by Buyers When Sourcing Suppliers

The most common challenges include supplier discovery, product alternatives, high costs, and slow communication. These findings support features like group purchasing, real-time messaging, and supplier verification.

Summary of Key Insights:

The survey revealed strong interest in features that improve supplier discovery, bulk purchasing, real-time communication, and cost transparency. Both buyers and suppliers highlighted challenges around finding the right partners, negotiating bulk deals, and accessing alternative options, insights that directly influenced the platform's category structure and feature set.

3.1.2 Algorithm Definition

This section outlines the structured algorithms used in the B2B platform, including the Stock Demand Forecasting Algorithm and the Alternative Product

Recommendation System. Each algorithm processes data based on specific inputs and generates relevant outputs to optimize business operations.

1 Stock Demand Forecasting Algorithm Using FB Prophet

This algorithm predicts future stock demand based on historical sales trends, enabling businesses to optimize inventory planning.

Inputs:

- Historical sales records, past purchases, order volumes, and purchase frequency.
- Supplier stock levels, which help determine if the business is at risk of understocking or overstocking.

Outputs:

- Predicted demand trend (e.g., 10% increase next month).
- Recommended restocking quantity.
- Forecast confidence score, indicating how reliable the prediction is.

Algorithm Steps:

1. Collect Historical Data: The system collects historical sales data, including supplier-specific sales records, stock levels, and past demand fluctuations. Missing data points are identified and cleaned to ensure the accuracy of the dataset.
2. Identify Missing Data Points and Apply Automated Data Cleaning: Automated data cleaning is applied to remove any inconsistencies or outliers, and missing data points are addressed using interpolation or forward-filling methods.
3. Decompose Time-Series Patterns Using FB Prophet: FB Prophet is used to model the demand trend based on historical data. The model focuses on the additive forecasting equation to represent demand trends without considering seasonality, holiday effects, or cyclical trends. Only long-term demand growth or decline is factored into the forecast.
4. Automated Time-Series Analysis (Trend Detection): The algorithm identifies long-term linear or curvilinear demand trends, focusing solely on past sales patterns. No adjustments for seasonality or holidays are made.
5. Generate Forecasted Demand & Confidence Score: The model predicts demand for the next three months, computing the forecast confidence score to indicate the reliability of the predictions. Only the trend-based forecast is returned without considering additional factors like seasonality.
6. Display Forecast Insights on the Supplier Demand Prediction Page: The forecasted demand for the next three months is displayed on the supplier demand prediction page. Visualized demand trends assist suppliers in understanding when demand is expected to rise or fall and how much stock should be ordered.

2 Alternative Product Recommendation System Algorithm Using LaBSE

The Alternative Product Recommendation System suggests products similar to the ones searched for by buyers, helping them discover alternatives. This algorithm

leverages LaBSE (Language-Agnostic BERT Sentence Embeddings) to ensure accurate recommendations in both Arabic and English.

Inputs:

- Product name and description entered by the buyer.
- Stored product descriptions in the database.
- The category of the product being searched (e.g., "Electronics", "Clothing"), used to filter recommendations.

Outputs:

- A list of similar products ranked based on their similarity to the product searched.
- A relevance score, a numerical value indicating how closely related each recommended product is.

Algorithm Steps:

1. User Inputs a Product Name: The buyer inputs a product name into the search field, initiating the recommendation process.
2. Retrieve Product Description: The system fetches the corresponding description of the searched product from the database.
3. Vectorization Using LaBSE: The input text (product name and description) is converted into a numerical vector representation using LaBSE, a pre-trained NLP model designed for multilingual sentence embeddings. This method captures the meaning of words more effectively than traditional keyword-based approaches, making it suitable for cross-language recommendations.
4. Compare Against Stored Product Vectors: All stored product descriptions are converted into numerical vectors using LaBSE. The system computes the similarity between the input product and all stored products by comparing their vector representations. If a category is provided, the system can filter the product pool by category before calculating similarity.
5. Compute Cosine Similarity: Cosine similarity is used to measure the similarity between the product vectors. A higher cosine similarity score indicates greater similarity between the products.
6. Sort and Rank Products: The products are ranked based on their similarity scores. Products with higher similarity (closer to a score of 1) are ranked higher, while those with lower scores are ranked lower.
7. Display Recommendations: The system returns the top-ranked similar products to the user, allowing them to explore alternative products from different suppliers.

3 Integration Considerations

The FastAPI framework will be used to expose dedicated endpoints for both the Stock Demand Forecasting Algorithm and the Alternative Product Recommendation System.

1. Stock Demand Forecasting Algorithm:

An endpoint will be created to accept historical sales data from suppliers. The backend will process this data using the FB Prophet model to predict stock demand for the next three months. The forecasted demand, along with a confidence score, will be returned as the output, helping suppliers make informed decisions about inventory restocking.

2. Alternative Product Recommendation System:

An endpoint will accept the product search input and the product category. The backend will filter the available products by that category before proceeding with the recommendation process. The backend will compute the similarity between the searched product and the stored products, returning the top-ranked recommendations based on similarity scores. This ensures that the recommendations are relevant to the user's query, with an emphasis on improving the product discovery experience.

Both algorithms will be seamlessly integrated into the FastAPI framework, ensuring real-time processing and smooth interaction between the front-end and back-end.

4 AI Development Methodology

This section outlines the development methodology followed for both AI models implemented in the system; the Prophet model for demand forecasting and the LaBSE model for semantic similarity in product recommendations. Although the models serve different purposes, they share the same development pipeline, which includes dataset preparation, model training, evaluation, and deployment.

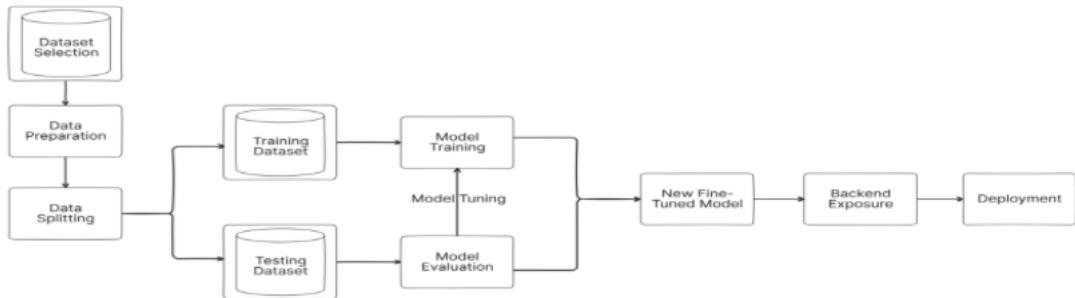


Figure 3-8: AI Model Development and Deployment Workflow

Methodology Overview:

1. Dataset Selection:

Relevant datasets were collected for each AI task. Historical demand data was used for the Prophet model, while textual product data was used for the LaBSE model.

2. Data Preparation and Splitting:

The datasets were cleaned, preprocessed, and divided into training and testing subsets (80% for training and 20% for testing) to ensure unbiased model evaluation.

3. Model Training:

Each model was trained using its corresponding dataset. Prophet learned temporal demand patterns, while LaBSE generated semantic embeddings from multilingual text data.

4. Model Evaluation:

The trained models were evaluated using the test datasets to assess their

accuracy and reliability. Fine-tuning was performed as needed to optimize performance.

5. Backend Integration:

After validation, the models were integrated into a FastAPI backend, which exposed REST endpoints for prediction (Prophet) and similarity calculation (LaBSE using cosine similarity).

6. Deployment:

The FastAPI services were deployed on the server and connected to the main system for real-time inference.

3.1.3 System Requirements

3.1.3.1 Non-Functional Requirements

- Availability
 1. The system shall maintain a minimum uptime of 98% per month.
 2. The system shall not experience downtime exceeding 2 consecutive hours.
- Reliability
 1. In the event of a crash or accidental page closure, the system shall restore the most recently saved user input, allowing users to continue without re-entering lost data.
- Usability
 1. The system shall offer an intuitive and user-friendly interface with clear navigation.
 2. The Usability Test score shall not be below 70%, based on user surveys and feedback.
- Portability
 1. The system shall be compatible with the major web browsers, Google Chrome, Mozilla Firefox, and Safari.

3.1.3.2 Functional Requirements

To maintain clarity and usability, functional requirements are categorized into three sections: Shared (for Both), Suppliers', and Buyers'.

1 Shared Functional Requirements

1.1 User Registration

1.1.1 The system shall allow the users to sign up.

1.1.1.1 The user shall enter his business name, email, password, confirm password, location, industry type, and his Commercial Registration number.

1.1.1.2 The system shall confirm that the Commercial Registration is valid and active.

1.1.1.3 Upon successful registration, the user shall be assigned the Buyer role by default.

1.2 User Authentication

1.2.1 The system shall allow the users to login.

1.2.1.1 The user shall enter their Commercial Registration number or email and password.

1.2.1.2 The system shall verify the user's credentials, assign them the Buyer role by default after signup, and grant them the corresponding permissions.

1.2.2 The system shall enforce password complexity rules for secure authentication.

1.2.2.1 The password shall be between 8 to 28 characters long.

1.2.2.2 The password shall contain at least one uppercase letter, one lowercase letter, and one number.

1.2.2.3 The password may contain special characters (@, #, !, \$) but it is not required.

1.2.3 The system shall allow users to reset their password if they have forgotten it.

1.3 Account Management

1.3.1 The system shall allow users to update their business details, including business name, the name of the person responsible for company account, location, industry type, and password.

1.3.2 The system shall allow users to log out at any time.

1.3.3 The users shall have the option to switch between roles at any time through a toggle button, gaining access to the corresponding functionalities (e.g., switching from Buyer to Supplier and vice versa).

1.4 Notifications

1.4.1 The system shall notify users via in-platform notifications about important updates (e.g., new messages, bid requests, order updates, invoice status changes).

1.4.2 The system shall allow users to manage in-platform notification settings (enable/disable specific alerts).

1.5 Messaging

1.5.1 The system shall allow buyers and suppliers to communicate via an internal direct messaging system.

1.5.2 The system shall allow suppliers and buyers to send images in chat.

2 Suppliers' Functional Requirements

2.1 Listings Management

2.1.1 The system shall allow suppliers to add, edit, duplicate, and delete listings (products or services).

2.1.1.1 The system shall allow suppliers to add a new listing.

2.1.1.2 The system shall allow suppliers to edit an existing listing.

2.1.1.3 The system shall allow suppliers to duplicate an existing listing.

2.1.1.4 The system shall allow suppliers to delete a listing.

2.1.2 The system shall allow suppliers to write the product name, description, pricing, and upload images when adding a new product or service.

2.1.3 The system shall allow suppliers to categorize their listings by industry when adding a new product or service.

2.1.4 For products, the platform shall enforce orders to be placed in fixed multiples of the minimum order quantity (case quantity) as specified by the supplier. Buyers shall only be able to order in multiples of this quantity (e.g., 5, 10, 15, etc.).

2.1.5 Each product shall have a minimum and/or maximum order quantity set by the supplier. The minimum and maximum order quantities shall be specified in multiples of the case quantity. For example, if the case quantity is 4, the minimum order quantity could be 4, and the maximum could be 8, 12, 16, etc. Buyers will not be able to place an order for less or more than these specified quantities.

2.1.7 For products, the system shall allow suppliers to enable or disable group purchasing.

2.1.6.1 If group purchasing is enabled, suppliers must specify the minimum order quantity per group (total units or packages required), the discounted price per unit for group purchases, and an order deadline (either 3, 5, or 7 days).

2.1.8 For services, suppliers shall specify service availability (Available on weekdays, Available on weekends, Available 24/7, By appointment only).

2.2 Product Order Management

2.2.1 The system shall notify suppliers upon order placement by buyers.

2.2.2 The system shall set the default status of a new order to "Pending".

2.2.3 The system shall allow suppliers to view the order details, including the products ordered and buyer information.

2.2.4 The system shall allow suppliers to change the status of an order.

2.2.4.1 The supplier can set the order status to "Processing" when they begin working on the order.

2.2.4.2 The supplier can set the order status to "Shipped" once the order has been dispatched.

2.2.4.3 The system sets the order status to "Completed" once the buyer confirms delivery.

2.3 Bidding Management

2.3.1 The system shall allow suppliers to view bid requests, including relevant bid details such as project title, main activity, and submission deadline.

2.3.2 The system shall allow suppliers to submit offers for bid requests.

2.3.3 The system shall send notifications to suppliers when their offer is accepted or rejected by the buyer.

2.3.4 If the buyer does not respond within the specified expected response time, the system shall send a rejection notification to all suppliers who participated in the bid.

2.4 Invoice Management

2.4.1 The system shall allow suppliers to create invoices for buyers. Invoices shall include the product/service name, agreed details (e.g., customization, service duration), price (per unit or total), delivery date, and payment terms (Full Payment or Partial Payment).

2.4.2 The system shall allow suppliers to receive invoices for successful group purchases.

2.4.3 The system shall allow suppliers to view their invoice history, including statuses: Accepted, Rejected, Partially Paid, Fully Paid.

2.4.4 The system shall notify suppliers of any status changes to their invoices, such as when an invoice is accepted, rejected, or paid.

2.5 Storefront Management

2.5.1 The system shall allow the supplier to set a fixed delivery fee which will be applied to all orders placed.

2.5.2 The system shall allow suppliers to customize their storefront by adding a banner and a business bio.

2.5.3 The system shall allow suppliers to temporarily close their store.

2.5.4 When a supplier's store is closed, the storefront page shall display a customizable "Out of Order" message, defined by the supplier.

2.5.5 The system shall notify the supplier of new reviews.

2.6 Business Insights & Analytics

2.6.1 The system shall display key business insights for suppliers, including total sales, overall supplier rating, most ordered product, and most wish-listed product.

2.6.2 The system shall display AI-powered stock demand forecasts for each product on the products and services page, with predictions for the next three months.

2.6.3 The system shall provide recommendations for restocking based on predicted demand, which shall be visible on the products and services page.

2.7 Subscription Management

2.7.1 The platform shall offer two subscription plans for suppliers: a Basic Free Plan with limited features and a Premium Plan (50 SAR/month) that provides additional benefits.

2.7.2 The Basic Free Plan shall include the following features: creation of a storefront, receiving orders, listing up to 10 products and 3 services, and the ability to view and respond to buyer messages.

2.7.3 The Premium Plan (50 SAR/month) shall include all features of the Basic Free Plan, with the addition of unlimited product and service listings, and stock demand forecasting powered by AI-driven insights.

2.7.4 All new suppliers shall receive a one-month free trial of the Premium Plan, with no credit card required for activation.

2.7.5 If a supplier fails to renew their subscription, their storefront and products shall be temporarily hidden from buyers until payment is made.

3 Buyers' Functional Requirements

3.1 Product Discovery

3.1.1 The system shall allow buyers to search for products, services, and suppliers by entering keywords or selecting relevant categories.

3.1.2 The system shall allow buyers to filter search results based on criteria such as minimum order price, which will exclude products with a total order value exceeding the specified amount (e.g., if the minimum order requirement for a product is 25, and the buyer enters 15, the system will not display this product).

3.1.3 When displaying search results, the system shall present each product with its image, name, price, and the supplier's name. For products, the system shall also display the minimum order quantity requirement.

3.1.4 When a buyer selects a product, the system shall display all details from 3.1.3, including the full product description, product rating, and group purchasing details (if applicable). These details will include information such as the minimum required units per group, the discount offered, and the order deadline for group purchases, which is applicable when a group purchase exists in the buyer's city and has not yet met the deadline or reached the minimum required number of buyers (5 buyers).

3.1.5 The system shall allow buyers to save products to a wish list for future reference.

3.1.6 The system shall offer a "Find Alternatives" option next to the wish listed products, allowing buyers to discover similar products for them.

3.1.7 The system shall allow users to enter a product name that they want to find alternatives for at the bottom of the homepage. The users will be redirected to a "Product Similar-To" page after hitting enter.

3.2 Order Placement

3.2.1 The system shall allow buyers to add items to their cart, either from the same supplier or multiple suppliers.

3.2.2 The system shall allow buyers to proceed to checkout and complete the purchase using a Mada card.

3.2.3 The system shall prevent buyers from ordering products that are out of stock.

3.2.3.1 Out-of-stock products shall be automatically marked as unavailable, and the order cannot be completed until they are removed from the cart.

3.2.3.2 The system shall provide a 'Want to search similar products?' link below each product that went out of stock in the cart, enabling buyers to explore alternatives.

3.3 Order Management

3.3.1 The system shall track and display the status of orders (Pending, Processing, Shipped, Completed).

3.3.1.1 The system shall send notifications to buyers when the order status changes (e.g., when an order is processed, shipped, or completed).

3.3.2 The system shall allow buyers to mark an order as "Completed" once they confirm delivery.

3.3.3 After confirming the delivery of the order, the system shall allow buyers to submit reviews and rate products and suppliers for completed orders.

3.4 Bidding Management

3.4.1 The system shall allow buyers to create and submit Requests for Proposal (RFPs), where they can specify details such as the Bidding Project Title, Main Activity, Submission Deadline, and Response Deadline for Offers, with the option to select a response deadline of 2, 4, or 6 weeks after the submission deadline.

3.4.1.1 If the buyer does not respond to supplier offers within the selected timeframe, the system shall automatically reject all pending offers, and the buyer will no longer be able to view them.

3.4.2 The system shall send notifications to buyers when a supplier submits an offer in response to their RFP.

3.4.3 The system shall allow buyers to review the offers received from suppliers before accepting one.

3.4.4 Once an offer is accepted, the system shall automatically generate an invoice based on the agreed price and the details provided in the RFP and the offer.

3.5 Invoice Management

3.5.1 During negotiation, the system shall allow buyers to choose from two payment options: full payment (paying the full price after accepting the invoice) or partial payment (paying half the price upfront and the remaining amount upon delivery).

3.5.2 The system shall notify buyers when a new invoice is created.

3.5.3 The system shall display a list of the buyer's invoices along with their status (Accepted, Rejected, Partially Paid, Fully Paid).

3.5.4 The system shall allow buyers to review and accept invoices issued by suppliers.

3.5.5 Once an invoice is accepted, the buyer can proceed with the chosen payment option (full or partial payment).

3.5.6 When a buyer joins a group purchase, a Pre-Invoice shall be created to allow the buyer to track the group purchase status.

3.5.7 The system shall assign a status to the Pre-Invoice, which can be Pending if the group is still gathering buyers, Failed if the group fails to meet the supplier's order requirement, or Waiting for Payment if the group purchase is successful and the buyer

can proceed with payment. At this stage, the system shall convert the Pre-Invoice into a standard Invoice.

3.5.8 The system shall send notifications to buyers regarding updates on their group purchase status.

3.5.9 The system shall allow the buyer to write reviews for the invoice items and supplier after the invoice has been paid.

For more detailed system requirements, including prioritization and dependencies, please refer to Appendix B.

3.2 Requirements Analysis

3.2.1 Use Case Diagrams & Descriptions

The table below presents the different user roles in the platform, outlining their key responsibilities and access levels. These roles range from Guests, who can browse products, to Buyers and Suppliers with specific functionalities like placing orders, managing listings, and communicating with other users. It also includes Inactive Suppliers, whose listings are hidden due to unpaid subscriptions.

Table 3-1: Actor Descriptions

Actor	Description
Guest	Can browse products but cannot interact (unregistered users).
User	A logged in user who can log out, view notifications, and message other users (inherited by Supplier and Buyer).
Supplier	Can create a storefront, list products and services, and receive payments.
Inactive Supplier	A supplier who hasn't paid for a subscription, their storefront and listings are hidden.
Buyer	Can search for products, place orders, submit bids, and pay invoices.

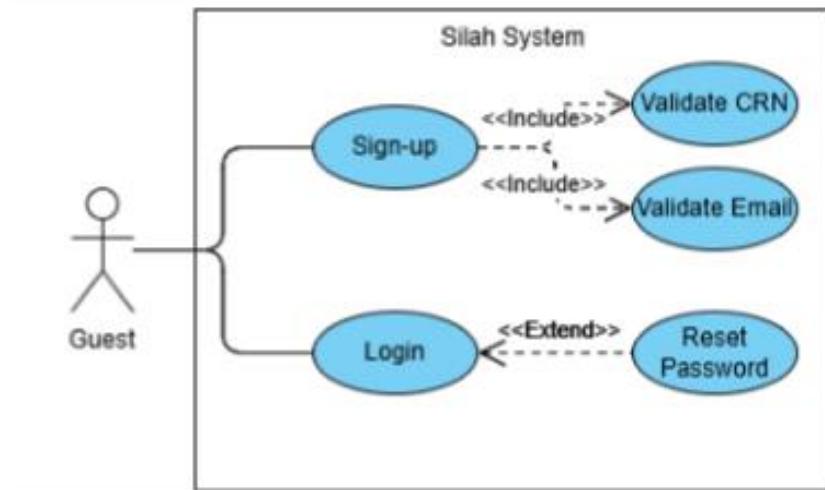


Figure 3-9: Guest Use Case Diagram

Table 3-2: Sing-up Use Case Description

Use Case Name	Sign-up
Actor	Guest
Description	Allows users to create an account by providing their business and personal details.
Pre-Condition	The user must not have a registered account.
Event Flow	<ol style="list-style-type: none"> 1. The user accesses the sign-up page. 2. The user enters business details: <ul style="list-style-type: none"> • Business name • Commercial registration number • Business activity 3. The user clicks “Next” to proceed. 4. The user fills in personal details: <ul style="list-style-type: none"> • Full name • National ID • City 5. The user clicks “Next” to proceed. 6. The user enters account credentials: <ul style="list-style-type: none"> • Email • Password • Confirm password 7. The user agrees to the terms and conditions. 8. The user submits the registration form. 9. The system verifies the provided details. 10. If valid, the account is successfully created, and the user is redirected to the login page, and an email verification link will be sent. 11. If invalid, an error message is displayed.
Post-Condition	The user has a registered account and can log in to the system.

Table 3-3: Login Use Case Description

Use Case Name	Login
Actor	Guest
Description	Allows users to authenticate and access their accounts.
Pre-Condition	The user must have a registered account.
Event Flow	<ol style="list-style-type: none"> 1. The user accesses the login page. 2. The user enters their credentials: <ul style="list-style-type: none"> • Email or Commercial Registration Number • Password 3. The user clicks the “Login” button. 4. The system verifies the credentials. 5. If valid, the user is logged in and redirected to their homepage. 6. If invalid, an error message is displayed. 7. If the user forgets their password, they can click on “Forgot Password?” to reset it. 8. If the user does not have an account, they can click on “Sign Up” to register.
Post-Condition	The user is successfully logged in, resets their password, or proceeds to account registration.

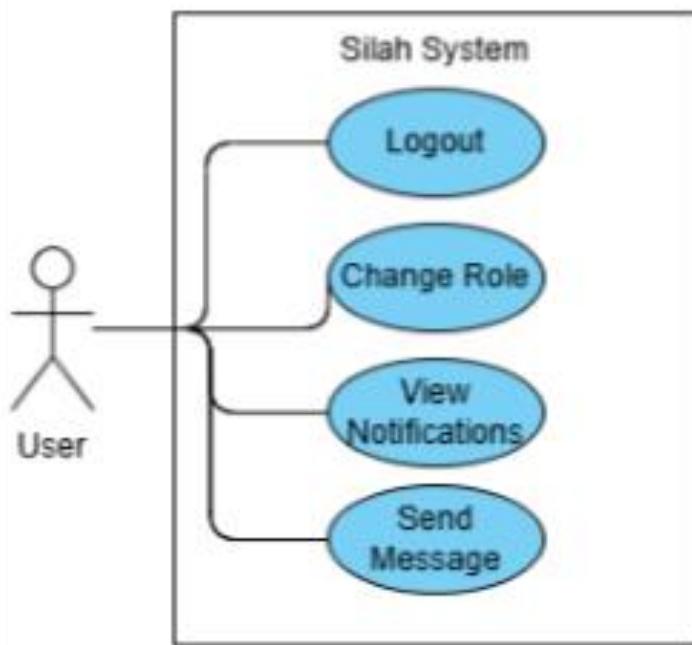


Figure 3-10: User Use Case Diagram

Table 3-4: Logout Use Case Description

Use Case Name	Logout
Actor	User (Inherited by both Supplier and Buyer)

Description	The user logs out of the system.
Pre-Condition	The user must be logged in.
Event Flow	<ol style="list-style-type: none"> 1. The “Logout” button is clicked by the user. 2. The current session is ended by the system. 3. The landing page is displayed to the user.
Post-Condition	The user is successfully logged out.

Table 3-5: Change Role Use Case Description

Use Case Name	Change Role
Actor	User (Inherited by both Supplier and Buyer)
Description	This option allows the user to switch roles between Supplier and Buyer.
Pre-Condition	The user must be logged in.
Event Flow	<ol style="list-style-type: none"> 1. The user clicks the “Change Role” button. 2. If the user was a Buyer, the system switches his role to Supplier and navigates the user to the Overview page. 3. If the user was a Supplier, the system switches his role to Buyer and navigates the user to the Homepage.
Post-Condition	The user's role is successfully changed, and the system is updated accordingly.

Table 3-6: View Notifications Use Case Description

Use Case Name	View Notifications
Actor	User (Inherited by both Supplier and Buyer)
Description	The user can view notifications in the system.
Pre-Condition	The user must be logged in.
Event Flow	<ol style="list-style-type: none"> 1. The user navigates to the Notifications page. 2. The system displays the recent notifications along with their date and time. 3. New unseen notifications are displayed with a circle next to them. 4. The user can filter the notifications based on type (e.g., New Messages, New Orders, New Invoices). 5. The user can filter the notifications based on time (e.g., Today, This Week).
Post-Condition	All notifications viewed are marked as “seen” by the system.

Table 3-7: Send Message Use Case Description

Use Case Name	Send Message
Actor	User (Inherited by both Supplier and Buyer)
Description	This use case allows the user to send a message to another user within the system.

Pre-Condition	The user must be logged in.
Event Flow	<ol style="list-style-type: none"> 1. The user navigates to the Chats page. 2. The system displays all previous chats, including the recipient's name, last message sent, and its timestamp. 3. The user can filter chats based on read/unread status. 4. The user can filter chats based on time (e.g., Today, This Week). 5. The user can search for another user in the search bar and can start a conversation with them. 6. The user selects a recipient, and the system redirects them to the chat page. 7. The user can write a message and send it. 8. The user can attach an image and send it. 9. The recipient receives the message along with a notification.
Post-Condition	<ul style="list-style-type: none"> • The message is successfully sent and can be viewed by the recipient. • A notification is sent to the recipient.

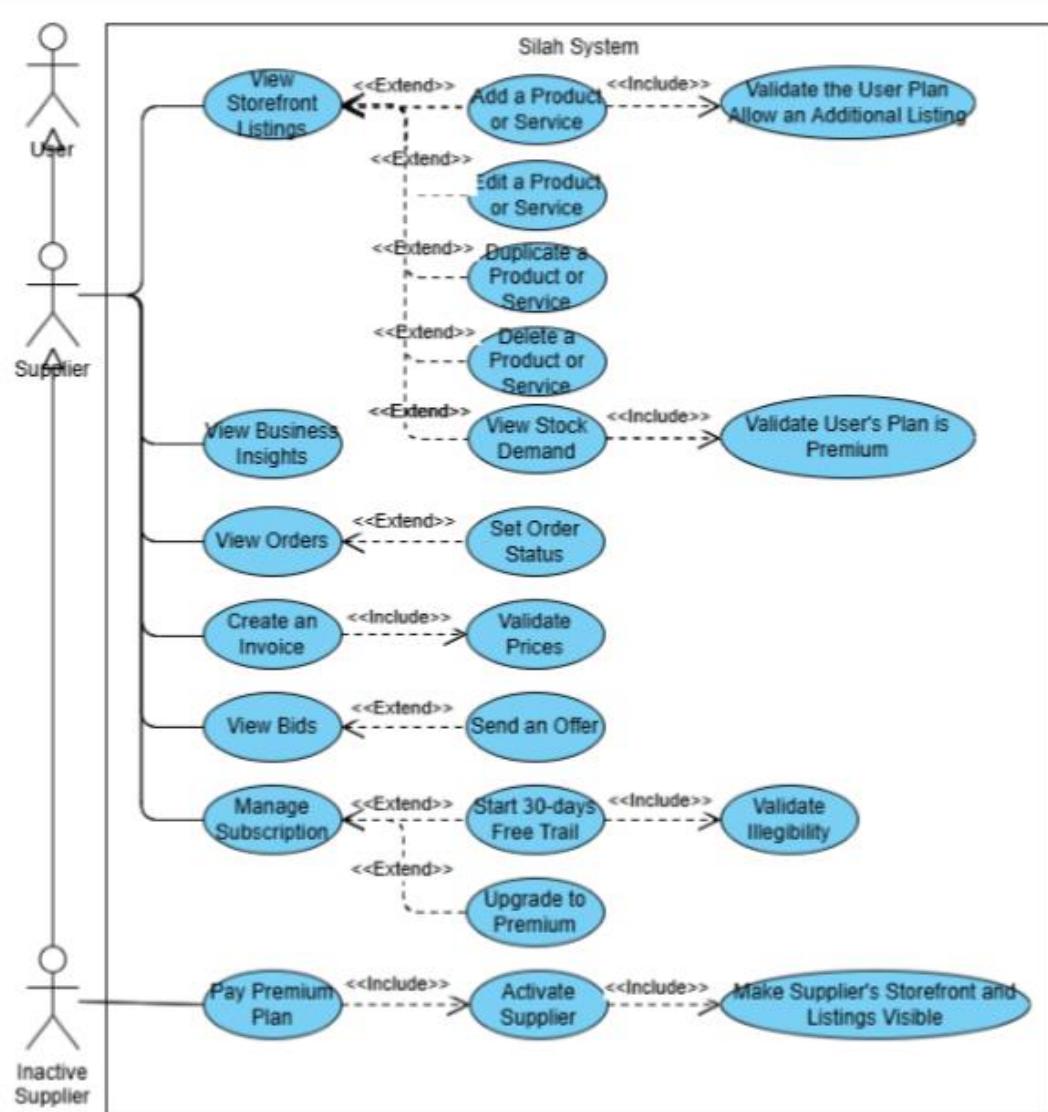


Figure 3-11: Supplier Use Case Diagram

Table 3-8: View Storefront Listings Use Case Description

Use Case Name	View Storefront Listings
Actor	Supplier
Description	This use case allows the supplier to view his listings (products and services) and manage them.
Pre-Condition	The user must be logged in as a Supplier.
Event Flow	<ol style="list-style-type: none"> 1. The supplier navigates to the Products & Services page. 2. The system displays all the supplier's listings, including product or service name, image, number of wish-listers (if the supplier is on the premium plan), unit price, stock (if applicable), and status (Published or Unpublished). 3. A Predict Demand button is displayed next to each product listing. 4. The supplier can search for a listing or filter listings by type (Products or Services).

	<ol style="list-style-type: none"> 5. If the supplier clicks the Predict Demand button, the system navigates to the Demand Prediction page. 6. On the Demand Prediction page, if the supplier has a premium plan, the system displays the product name and image, a prediction graph for the next three months, and a suggested stocking unit based on current stock; otherwise, the page content is blurred. 7. The supplier can perform the following actions on listings: Add a new product or service, duplicate an existing listing, edit an existing listing, and delete a listing. 8. When the supplier chooses to add a new product or service, the system checks the current subscription plan limits. 9. If the limits have not been exceeded (maximum 10 products and 3 services on the basic plan), the system navigates the supplier to the Product Details or Service Details page. 10. If the limits have been exceeded, the system displays a warning message and prevents the supplier from adding a new listing. 11. On the Product Details page, the supplier fills out the product information, including basic details such as the name, description, and category. They can upload up to three images, set the unit price, and define order requirements like case quantity, minimum order quantity, and maximum order quantity. If the supplier enables group purchasing for the product, they must also specify the minimum number of group orders, the order deadline, and the new discounted price. 12. On the Service Details page, the supplier fills out the service information, including basic details such as the name, description, and category. They can upload up to three images, set the service price with the option to mark it as negotiable, and specify service availability, such as whether it is offered 24/7, on weekdays, or on weekends.
Post-Condition	<ul style="list-style-type: none"> • The supplier has successfully viewed their product and service listings. • Any actions taken (add, duplicate, edit, or delete) are processed and saved by the system. • The storefront is updated to reflect the latest state of the supplier's listings.

Table 3-9: View Business Insights Use Case Description

Use Case Name	View Business Insights
Actor	Supplier

Description	This use case allows the supplier to access data and analytics related to the performance of their store.
Pre-Condition	The user must be logged in as a Supplier.
Event Flow	<ol style="list-style-type: none"> 1. The supplier navigates to the Analytics & Insights page. 2. The system displays performance data for the past three months. 3. The displayed data includes total sales, insights on top-performing products, overall store rating, and new reviews written by buyers about the supplier.
Post-Condition	The insights help the supplier make informed business decisions regarding inventory, pricing, and customer engagement.

Table 3-10: View Orders Use Case Description

Use Case Name	View Orders
Actor	Supplier
Description	This use case allows the supplier to view all received orders and their associated details.
Pre-Condition	<p>The user must be logged in as a Supplier. The supplier must have listed products and received orders.</p>
Event Flow	<ol style="list-style-type: none"> 1. The supplier navigates to the Orders page. 2. The system checks whether the supplier has received any orders. 3. If no orders are found, the system displays a message such as: "You haven't received any orders yet." 4. If there are orders, the system displays them in descending order, starting with the most recent. Each order includes the order number, creation date, buyer name, status, and total price. 5. The supplier can update an order's status by clicking on it: <ul style="list-style-type: none"> o If the status is Pending, the available options are Processing and Shipped. o If the status is Processing, the available options are Pending and Shipped. o If the status is Shipped, the available options are Pending and Processing. o The Completed status is not set manually and cannot be changed. 6. The supplier can click on an order to navigate to the Order Details page. 7. The system displays the full order details including the order number, status, buyer details, and ordered items. The supplier can also update the order status on the details page, following the same status change rules mentioned above.
Post-Condition	<p>Any updates to order statuses are stored in the database. The supplier can track order status and process them.</p>

Table 3-11: Create an Invoice Use Case Description

Use Case Name	Create an Invoice
Actor	Supplier
Description	This use case allows the supplier to create an invoice through the chat page after negotiating with a buyer. It is typically used when the buyer requests a custom product or service.
Pre-Condition	<ul style="list-style-type: none"> • The user must be logged in as a Supplier. • The buyer must have sent at least one message within the past three days.
Event Flow	<ol style="list-style-type: none"> 1. The supplier clicks on the Create an Invoice button found on the Chat page. 2. The system navigates the supplier to the Create Invoice page. 3. The system pre-fills some details, such as the issue date, and the supplier and buyer information. 4. The supplier fills out the invoice details, including the terms of payment (either Partially Paid or Fully Paid), the payment amount, and item details such as the item name, description, agreed-upon specifications, quantity, and unit price, along with any additional notes and terms related to the invoice. 5. The system calculates the total price for each item entered and displays it. 6. The system calculates the invoice total based on the entered items and validates it against the entered payment amount. 7. If errors are found, the system displays an appropriate error message to the supplier. 8. If no errors are found, the supplier can click on Create Invoice. 9. The system sends a notification about the new invoice to the buyer.
Post-Condition	<ul style="list-style-type: none"> • The invoice is successfully created and made available to the buyer for review and payment once accepted. • A notification is sent to the buyer.

Table 3-12: View Bids Use Case Description

Use Case Name	View Bids
Actor	Supplier
Description	Allows suppliers to view and participate in open bids opportunities by submitting their offers.
Pre-Condition	The user must be registered and logged into the system as a Supplier.
Event Flow	<ol style="list-style-type: none"> 1. The supplier accesses the Recent Bids page.

	<ol style="list-style-type: none"> 2. The system displays a list of open bids that have not yet reached the submission deadline. Each bid includes: <ul style="list-style-type: none"> • Bid name • Main activity • Reference number • Submission deadline • Date of publication 3. If the supplier is interested in a bid, they click the See Details button. 4. The supplier is redirected to the Bid Details page, which displays: <ul style="list-style-type: none"> • Company name • Time remaining until the bid closes • Expected response time • All details from the Recent Biddings page 5. If the supplier wants to participate, they click the Participate button. 6. If they are not interested, they click the Back button to return to the Recent Bids page. 7. After clicking Participate, the supplier is redirected to the Write & Submit an Offer page. 8. The supplier fills in the offer details: <ul style="list-style-type: none"> • Proposed Amount • Expected Completion Time • Technical Offer Details • Project Execution Duration • Additional Notes (if any) 9. The supplier clicks the Submit Offer button.
Post-Condition	The supplier has successfully submitted an offer, or they have navigated back to the list of bids.

Table 3-13: Manage Subscription Use Case Description

Use Case Name	Manage Subscription
Actor	Supplier
Description	This use case allows the supplier to manage their subscription by either starting a 30-day free trial (if eligible) or upgrading to the Premium Plan.
Pre-Condition	The user must be logged in as a Supplier.
Event Flow	<ol style="list-style-type: none"> 1. The supplier navigates to the Subscription Management page from the settings page or the overview page. 2. The system displays a “Subscribe to Premium” button with the monthly fee (50 SAR). 3. The system displays the current subscription status (Basic, Premium) and the remaining days if on a trial. 4. To upgrade the plan, the supplier clicks the “Subscribe to Premium” button.

	<ol style="list-style-type: none"> 5. The system updates the supplier subscription to Premium. 6. The system unlocks Premium features, such as unlimited listings and stock demand forecasting. 7. A success message is shown 8. If the supplier has not yet activated the free trial, the system shows a “Start Free Trial” button. 9. The supplier clicks “Start Free Trial.” 10. The system checks eligibility and activates the Premium Plan for 30 days with no card required.
Post-Condition	<ul style="list-style-type: none"> • The supplier subscribes to the Premium Plan or has successfully started their free trial. • Premium features become available on the supplier’s account. • The subscription status is updated in the system.

Table 3-14: Pay Premium Plan Use Case Description

Use Case Name	Pay Premium Plan
Actor	Inactive Supplier (Inherits Supplier)
Description	This use case allows the supplier to reactivate their account by paying the monthly fee for the Premium Plan. A supplier account becomes inactive if the free trial period has expired or if the supplier has upgraded to the Premium Plan but failed to complete the payment.
Pre-Condition	<ul style="list-style-type: none"> • The user must be logged in as a Supplier whose account is currently marked as inactive. • The supplier has exceeded the free trial limits; more than 10 products and/or more than 3 services. • The supplier previously attempted to upgrade to the Premium Plan but failed to complete the payment.
Event Flow	<ol style="list-style-type: none"> 1. The supplier logs onto the platform. 2. The system detects that the supplier account is inactive and displays a message indicating the reason for inactivity, along with a “Reactivate Account” button. 3. The supplier clicks the “Reactivate Account” button. 4. The system updates the supplier’s subscription status. 5. The system reactivates the supplier account and unlocks all Premium features.
Post-Condition	<ul style="list-style-type: none"> • The supplier’s account is reactivated and restored to full functionality. • Premium features are enabled. • The supplier is granted access to their storefront and product/service listings.

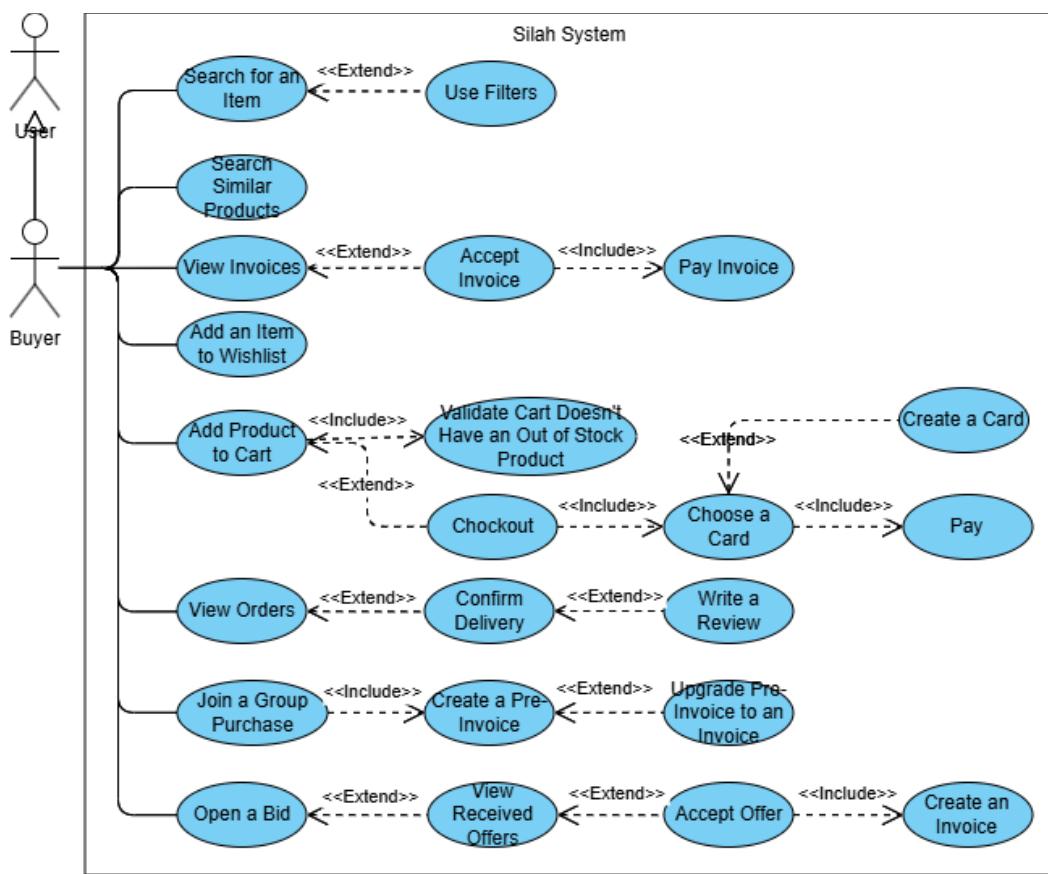


Figure 3-12: Buyer Use Case Diagram

Table 3-15: Search for an Item Use Case Description

Use Case Name	Search for an Item
Actor	Buyer
Description	This use case allows the buyer to search for available products and services listed by suppliers using keywords and filters to refine the results.
Pre-Condition	The user must be logged in as a Buyer.
Event Flow	<ol style="list-style-type: none"> 1. The buyer uses the search bar found in the header. 2. The buyer enters a keyword (e.g., product name, service, or supplier name) into the search bar. 3. The buyer selects the search type (product, service, supplier). 4. The system returns a list of matching results. 5. For each result, the system displays the product/service name, image, unit price, average rating, and supplier name. 6. The system displays filtering options such as minimum order price. 7. The system updates the search results based on the filters. 8. The buyer may click on any product or service to view its detailed information, including full description, average

	<p>rating, group purchase availability (if applicable), and reviews.</p> <p>9. The buyer may also save the item to a Wishlist or proceed to add it to the cart.</p>
Post-Condition	<ul style="list-style-type: none"> The buyer views a filtered list of products and services matching the search criteria. The buyer may proceed to view, save, or purchase any item from the results.

Table 3-16: Search for Similar Products Use Case Description

Use Case Name	Search for Similar Products
Actor	Buyer
Description	This use case allows the buyer to discover alternative products that are similar to a specific product of interest. This can be initiated from a wish list, out-of-stock cart item, or the homepage.
Pre-Condition	The user must be logged in as a Buyer.
Event Flow	<ol style="list-style-type: none"> The buyer navigates to the Wish List, Cart, or Homepage. In the Wish List, each saved product has a “Find Alternatives” button under it. In the Cart, if a product becomes out of stock, the system displays a “Want to search similar products?” link next to the unavailable product. On the Homepage, the buyer enters the name of a product they want to find alternatives for in the “Search Similar Products” section at the bottom of the page. The system redirects the buyer to the “Product Similar-To” page. The system uses semantic similarity AI to display a list of products similar to the selected or entered product ranked based on the cosine similarity score. For each recommended product, the system displays the name, image, supplier, price, and minimum order quantity. The buyer may click on any of the recommended products to view full details. The buyer may choose to add a similar product to their wish list or cart.
Post-Condition	<ul style="list-style-type: none"> The buyer is presented with a list of alternative products. The buyer can take further action such as viewing details, saving, or purchasing a suggested item.

Table 3-17: View Invoices Use Case Description

Use Case Name	View Invoices
Actor	Buyer
Description	This use case allows the buyer to view, filter, and manage invoices, including payment or rejection.
Pre-Condition	<ul style="list-style-type: none"> The user must be registered and logged in as a buyer.
Event Flow	<ol style="list-style-type: none"> The buyer navigates to the Invoices page. The system displays all invoices associated with the buyer. If no invoice exists, the system shall display “Nothing Yet”. The buyer can filter invoices by type: <ul style="list-style-type: none"> All Products Services Bids Group Purchases The buyer can filter invoices by status: <ul style="list-style-type: none"> All Accepted Rejected Partially Paid Fully Paid The system displays the filtered invoices, showing: <ul style="list-style-type: none"> Supplier name Order creation date Invoice number Invoice status Group Purchase Status (if applicable) Total amount When the buyer clicks on an invoice he will be navigated to the Invoice Details page. The system will display invoice details, along with buyer and supplier information. The buyer can then: <ul style="list-style-type: none"> Accept and pay the invoice by clicking “Accept & Pay”. Reject the invoice by clicking “Reject”. If accepted the buyer will be redirected to the Checkout page. If rejected the buyer will be redirected to the Invoices page again.
Post-Condition	<ul style="list-style-type: none"> Upon payment: The invoice status is updated to “Partially Paid” or “Fully Paid” depending on the terms of payment, and the supplier is notified. Upon rejection: The invoice is declined, and the supplier is notified.

Table 3-18: Add an Item to Wishlist Use Case Description

Use Case Name	Add an Item to Wishlist
Actor	Buyer
Description	This use case allows the buyer to save a product or service to their personal Wishlist for future reference. Buyers can also search for similar items from their Wishlist.
Pre-Condition	The user must be logged in as a Buyer.
Event Flow	<ol style="list-style-type: none"> 1. The buyer navigates to any page that lists a product or service, such as the search results page, supplier storefront, or homepage. 2. The buyer clicks the heart outline icon button found on top of the product/service image. 3. The system saves the item to the buyer's wish list and shows the heart icon as solid instead of outline. 4. If the heart icon is clicked again, the system removes the item from the buyer's wish list and shows the heart icon as an outline. 5. The buyer navigates to the "Wishlist" page from their dashboard or profile. 6. The system displays all saved items in a list, each showing the product/service name, image, price, and supplier name. 7. For each item in the Wishlist, the system also displays a "Find Alternatives" button. 8. The buyer may click on "Find Alternatives" to trigger the Search Similar Products use case.
Post-Condition	<ul style="list-style-type: none"> • The selected product or service is saved to the buyer's Wishlist. • The buyer can later review, or take action on saved items (view details, find alternatives, remove, or purchase).

Table 3-19: Add Product to Cart Use Case Description

Use Case Name	Add Product to Cart
Actor	Buyer
Description	This use case allows the buyer to add a product to their cart, in preparation for checkout and purchase.
Pre-Condition	The user must be logged in as a Buyer.
Event Flow	<ol style="list-style-type: none"> 1. The buyer navigates to the product details page from search results, supplier storefront, or homepage. 2. The system displays the product's full information, including name, description, images, unit price, supplier name, stock availability, case quantity, and order limits (minimum and maximum).

	<ol style="list-style-type: none"> 3. The buyer selects the desired quantity, which must be a multiple of the case quantity and within the supplier's allowed order range. 4. The buyer clicks the "Add to Cart" button. 5. The system adds the item to the buyer's cart. 6. The buyer can view their cart by clicking the cart icon on the header. 7. The system executes the "Validate Cart Doesn't Have an Out-of-Stock Product" use case to ensure all products in the cart are in-stock. 8. If a product is out of stock, the system prevents the user from checking out and displays a message with a link "Search Similar Products." 9. The system displays all cart items, including supplier name, product name, quantity, subtotal, and total cart value. 10. The buyer may update quantities or remove products from the cart at any time. 11. If the buyer proceeds to checkout, they must choose an existing card to use for payment. 12. If the buyer does not have a saved card, they must create a new card before proceeding to payment. 13. The buyer completes payment for the cart items using the selected card.
Post-Condition	<ul style="list-style-type: none"> • The selected product is successfully added to the buyer's cart. • The cart reflects the updated contents and total value. • If the product is out of stock, the buyer is alerted and guided to similar products. • If the buyer proceeds to checkout, they must complete payment before the order is confirmed.

Table 3-20: View Orders Use Case Description

Use Case Name	View Orders
Actor	Buyer
Description	Allows the buyer to view past orders, confirm the delivery of shipped orders, and write a review after confirming delivery.
Pre-Condition	The user is logged into the system.
Event Flow	<ol style="list-style-type: none"> 1. The user navigates to the Orders page. 2. The system checks if the user has any previous orders. 3. If the user has no orders, the system displays a message such as "You haven't made orders yet". 4. If the user has previous orders, the system displays them in descending order, starting from the most recent. Each

	<p>order includes the order number, creation date, supplier name, status, and total price.</p> <ol style="list-style-type: none"> 5. The user clicks on an order to navigate to the Order Details page. 6. The system displays the order details, including the order number, status, supplier details, and ordered items. 7. If the order status is "Shipped", the system displays a "Confirm Delivery" button. 8. The user clicks the "Confirm Delivery" button. 9. The system updates the order status to "Completed". 10. If the order status is "Completed", the system presents a "Write a Review" button. 11. The user clicks the "Write a Review" button and is redirected to the Write a Review page. 12. The system displays the supplier's information, a star rating box, and an optional text box for writing a review about the supplier. 13. The system lists each order item, allowing the user to rate and review individual products separately. 14. The user selects ratings, writes reviews if desired, and clicks the "Submit Review" button. 15. The system saves the review and redirects the user back to the Orders page.
Post-Condition	<ul style="list-style-type: none"> • If the user confirmed delivery, the order status is updated to "Completed". • If the user wrote a review, the review is saved in the system and becomes visible to other users.

Table 3-21: Join a Group Purchase Use Case Description

Use Case Name	Join a Group Purchase
Actor	Buyer
Description	Allows buyers to join an existing group purchase to benefit from bulk pricing or start one.
Pre-Condition	<ul style="list-style-type: none"> • The user must be registered in the system. • To join a group purchase, it must be open for participation. This means the user is in the same city as an existing group purchase that has not yet reached the minimum number of buyers (5) and the required minimum order quantity, and the joining deadline has not passed. • If a group purchase doesn't exist, the user can start one if the supplier allows it.
Event Flow	<ol style="list-style-type: none"> 1. The user navigates to the Product Details page. 2. The system displays the product details along with information about any existing open group purchases.

	<ol style="list-style-type: none"> 3. If a group purchase exists, the system shows the number of buyers joined, remaining time, discount, quantity input field, and a "Join Group Purchase" button. 4. If no group purchase exists, the system displays the discount, quantity input field, and a "Start Group Purchase" button, provided the supplier allows it. 5. The user either joins an existing group purchase or starts a new one by clicking the respective button. 6. The system creates a pre-invoice containing the product details, quantity, and discounted price. 7. If the group purchase meets the required conditions (minimum 5 buyers and minimum order quantity before the deadline), it is marked as Successful, and all pre-invoices are upgraded to invoices for payment. 8. If the group purchase does not meet the conditions by the deadline, it is marked as Failed, and all associated pre-invoices are marked as Canceled. 9. The system notifies all participants about the success or failure of the group purchase.
Post-Condition	<ul style="list-style-type: none"> • If successful, the user is now part of an active group purchase, and a valid invoice is issued. • If unsuccessful, the group purchase is marked as Failed, and all pre-invoices are automatically canceled.

Table 3-22: Open a Bid Use Case Description

Use Case Name	Open a Bid
Actor	Buyer
Description	This use case allows the buyer to create bids and review supplier offers after the submission deadline.
Pre-Condition	The user must be registered and logged into the system as a buyer.
Event Flow	<ol style="list-style-type: none"> 1. The buyer navigates to the Biddings You Created page. 2. The system displays a list of bids created by the buyer, including their: Bid name, Main activity, Publication date, Submission deadline, and Reference number. 3. The buyer can click "View Details" to see the bid details or click "View Offers" to browse supplier-submitted offers, this option is only shown after the submission deadline. 4. If the buyer clicks "Create a New Bid" they are redirected to the Create a New Bid page. 5. The buyer enters the bid details, including Bid name, Main activity, Submission deadline, and Response deadline for offers. 6. The buyer clicks "Publish Bid" to complete the process.

	<ol style="list-style-type: none"> 7. If the buyer clicked “View Offers” for a specific bid. 8. The system redirects the buyer to the Bidding Offers page, displaying a summary of submitted offers, including Supplier name, Proposed amount, Project completion time, Supplier logo, and Offer submission time. 9. The buyer can decline the offer by clicking “Decline Offer.” or accept the offer by clicking “Accept Offer.” or view offer details by clicking “View Details.” 10. If the buyer clicks “View Details,” they are redirected to the Supplier Offer Details page, which displays: Technical offer details, Project execution duration, Supplier’s notes (if any), and Proposed amount. 11. If the buyer accepts the offer, the system redirects them to the Invoice Details page.
Post-Condition	<ul style="list-style-type: none"> • If the offer is accepted: An invoice is generated between both parties, and the remaining suppliers are notified that their offers were not accepted. • If all offers are rejected: The buyer will need to create a new bid if they wish to find new offers.

3.2.2 Sequence Diagrams

To ensure better readability and a clearer view of the sequence diagrams included in this report, a high-resolution version of the diagrams can be accessed through the following link:

https://drive.google.com/drive/folders/13NGV2bg7yJ6Qw9nkdPYbq7KfWaed5ZUR?usp=drive_link

The diagrams provided in the link are the same as those in the report but are presented in higher quality, offering enhanced clarity for a deeper understanding of the depicted processes.

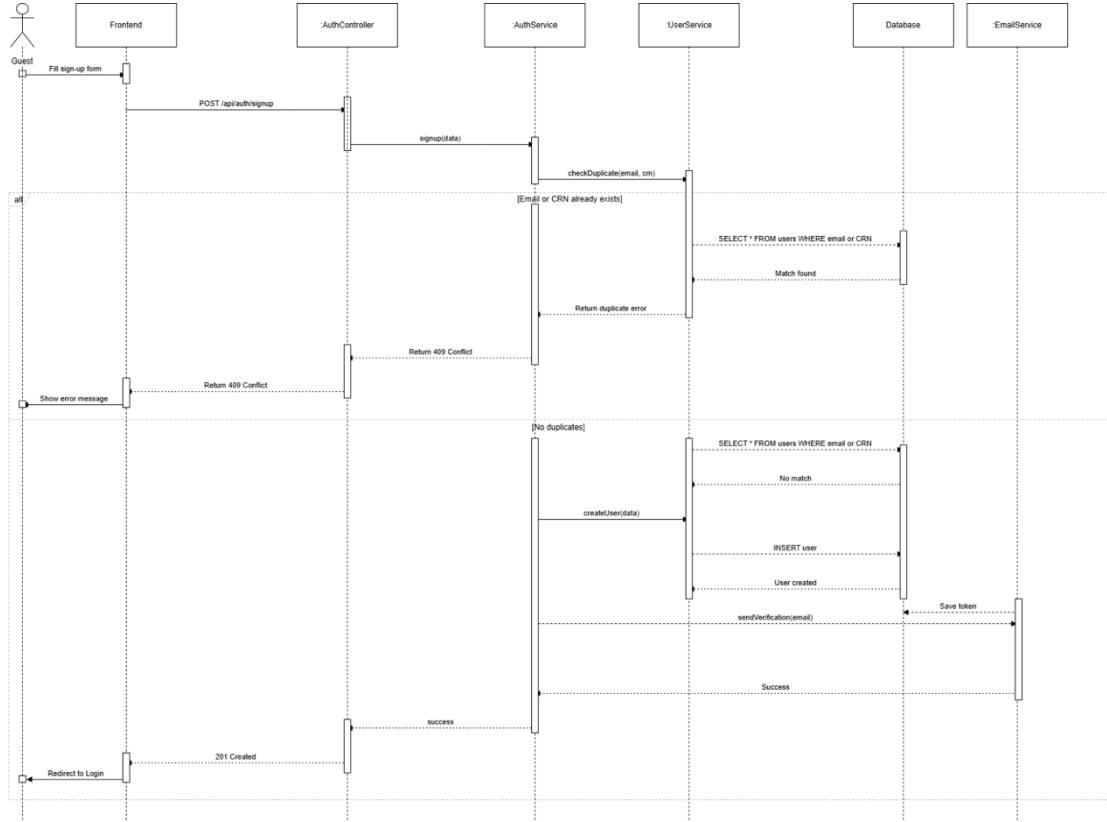


Figure 3-13: Sign-Up Process Sequence Diagram

This diagram illustrates the user registration flow. After the guest fills out the sign-up form, the AuthController sends the data to AuthService, which checks for duplicate emails or CRNs using the UserService. If a duplicate is found, a conflict error is returned. Otherwise, the new user is inserted into the database, a verification token is saved, and a verification email is sent before redirecting the user to the login page.

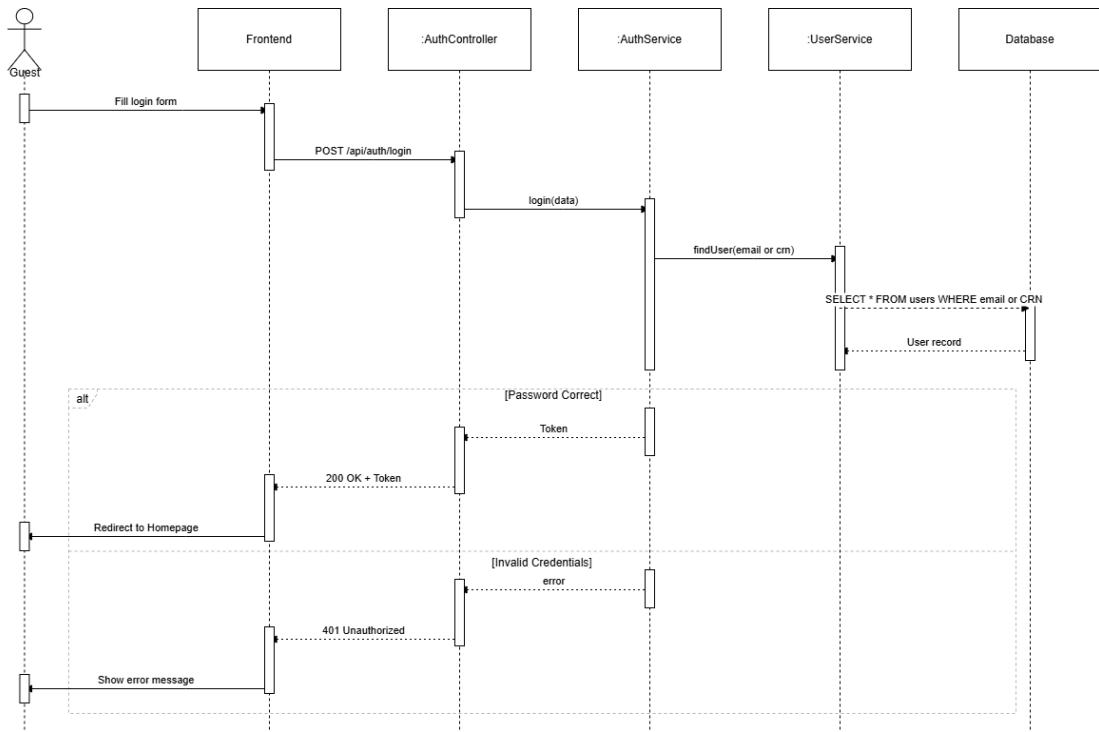


Figure 3-14: Login Process Sequence Diagram

This diagram shows how a guest logs into the system. After submitting the login form, the frontend sends a request to the AuthController, which calls AuthService to validate credentials via UserService. If the credentials are valid, a token is returned, and the user is redirected. Otherwise, an error message is shown.

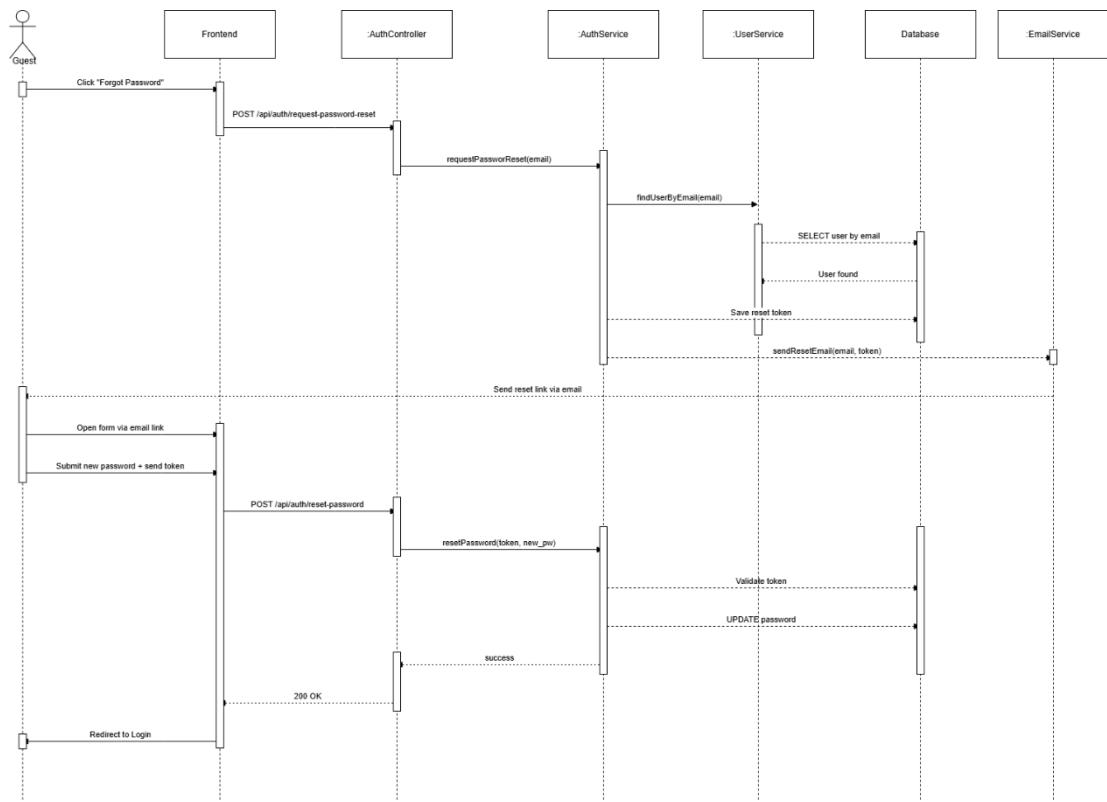


Figure 3-15: Password Reset Sequence Diagram

This diagram demonstrates how a guest resets their password. The flow starts with requesting a reset link, continues with email delivery of a token, and ends with password update upon token verification.

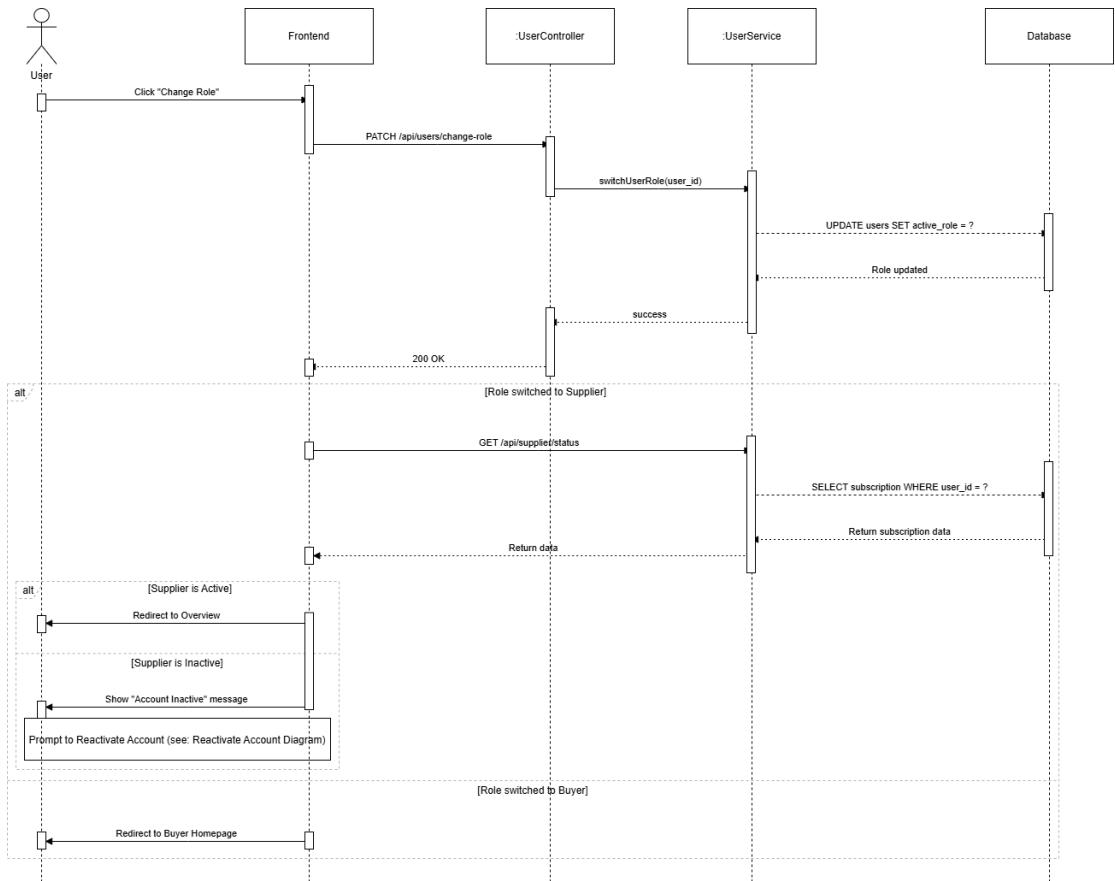


Figure 3-16: Switch User Role Sequence Diagram

This flow shows how a logged-in user switches their active role (between buyer and supplier). The system updates the role and fetches supplier subscription data to determine the correct redirection.

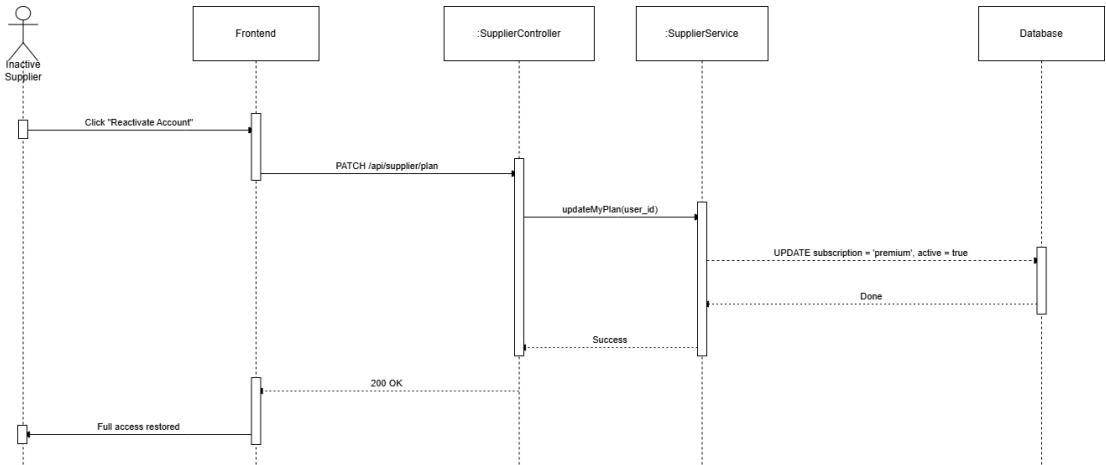


Figure 3-17: Reactivate Supplier Account Sequence Diagram

This diagram illustrates how an inactive supplier reactivates their account. When the user clicks "Reactivate Account," a request is sent to update the subscription plan. The backend updates the database to activate the subscription and grant full access.

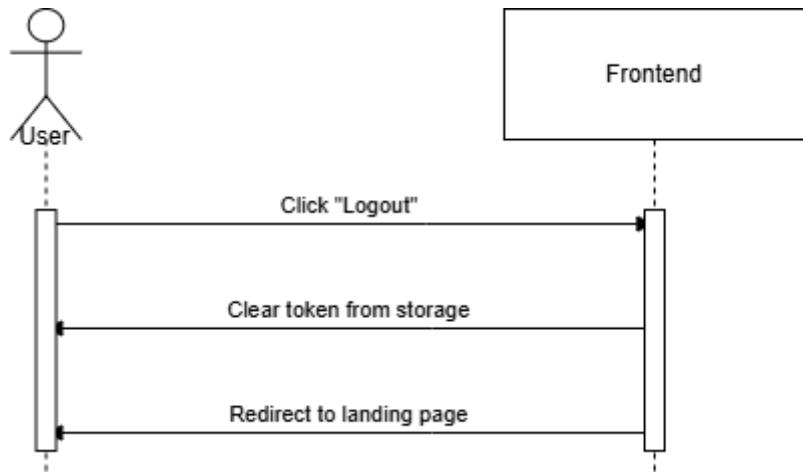


Figure 3-18: Logout Sequence Diagram

The logout sequence is handled entirely on the frontend by clearing the stored token and redirecting the user to the landing page.

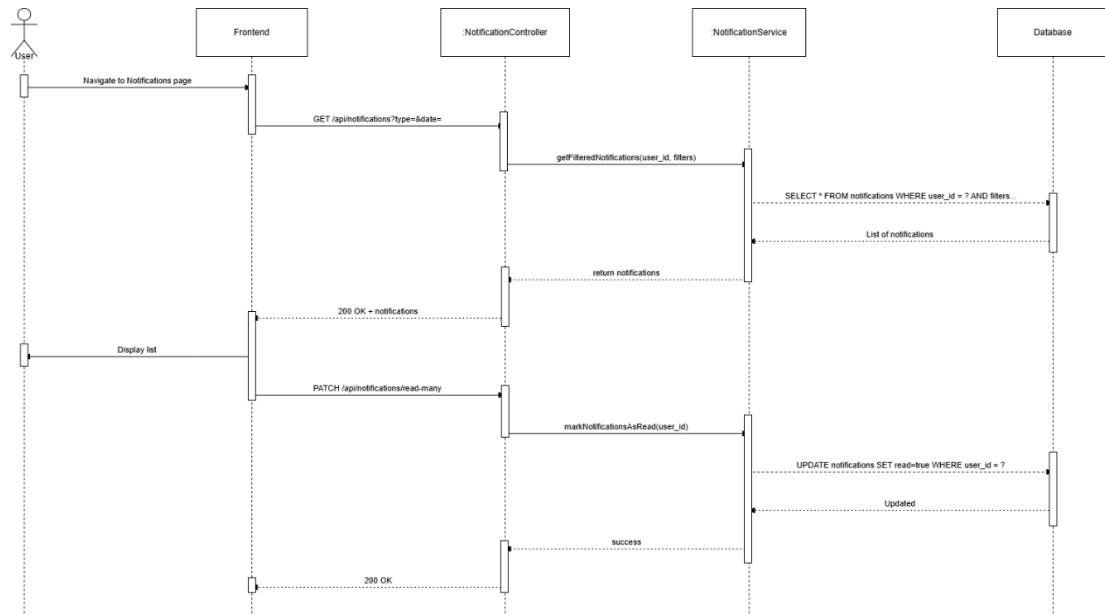


Figure 3-19: View Notifications Sequence Diagram

This diagram shows how the user retrieves their notifications and marks them as read using a bulk update action.

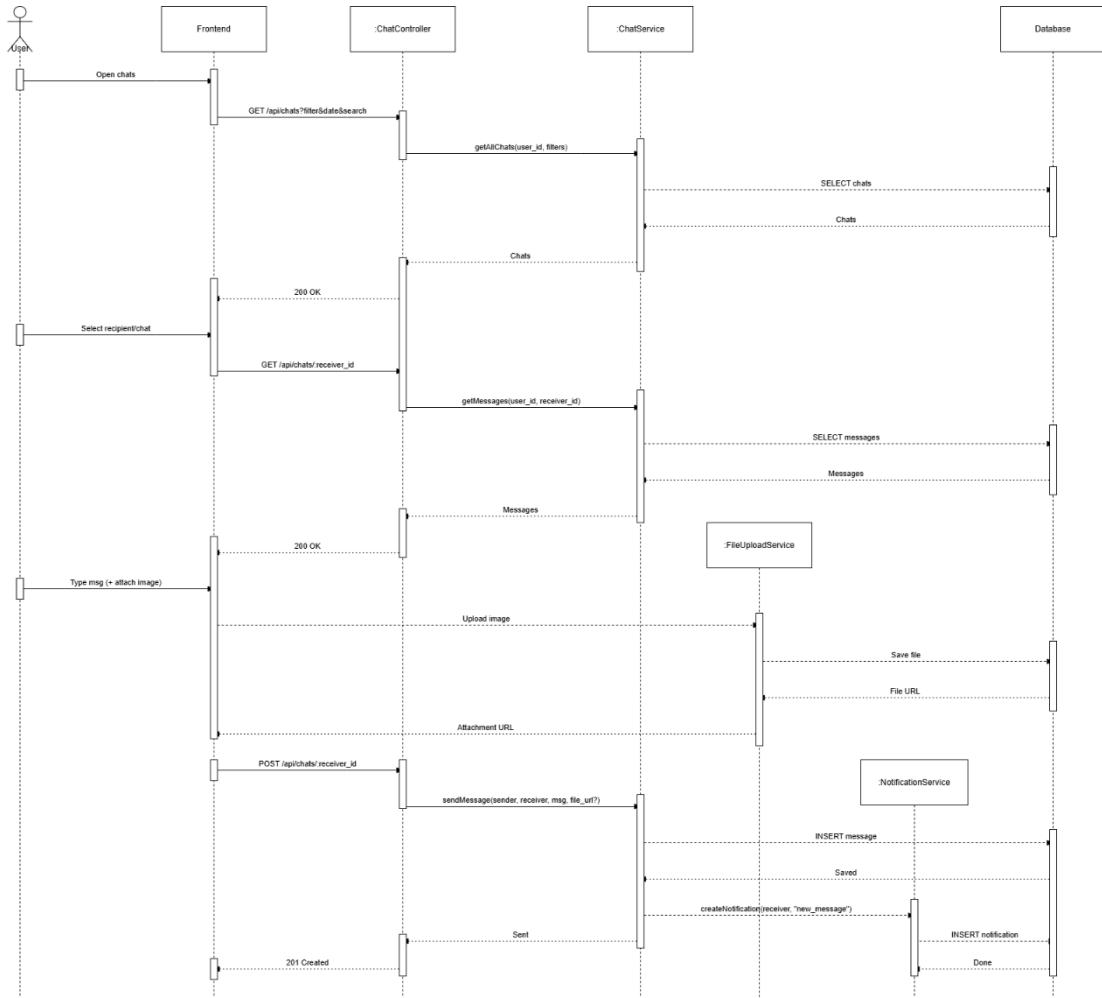


Figure 3-20: Chat Flow Sequence Diagram

This diagram illustrates the user's interaction with the chat system. It starts by fetching the list of chats with optional filters, retrieving message history for a selected receiver, and optionally uploading a file if attached. The message is then sent and stored in the database, and a notification is created for the receiver.

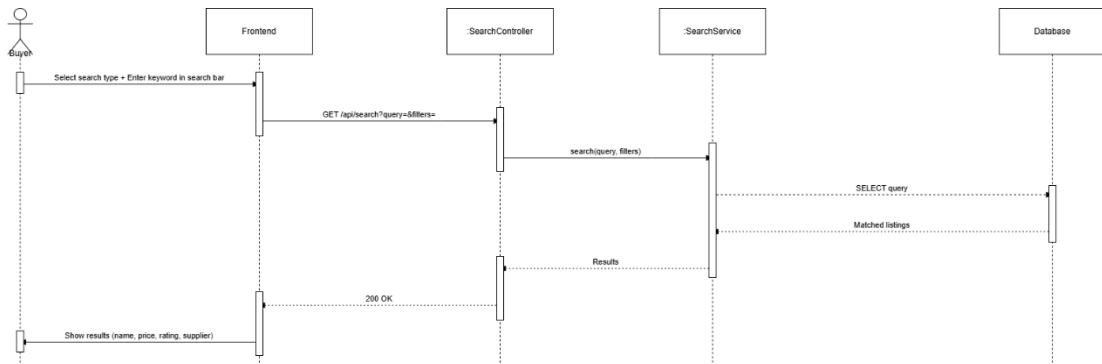


Figure 3-21: Search Flow Sequence Diagram

This diagram shows how a buyer interacts with the search feature. The process begins when the buyer selects the search type and inputs a query. The system applies

optional filters, queries the database for matching listings, and returns the results to be displayed on the frontend.

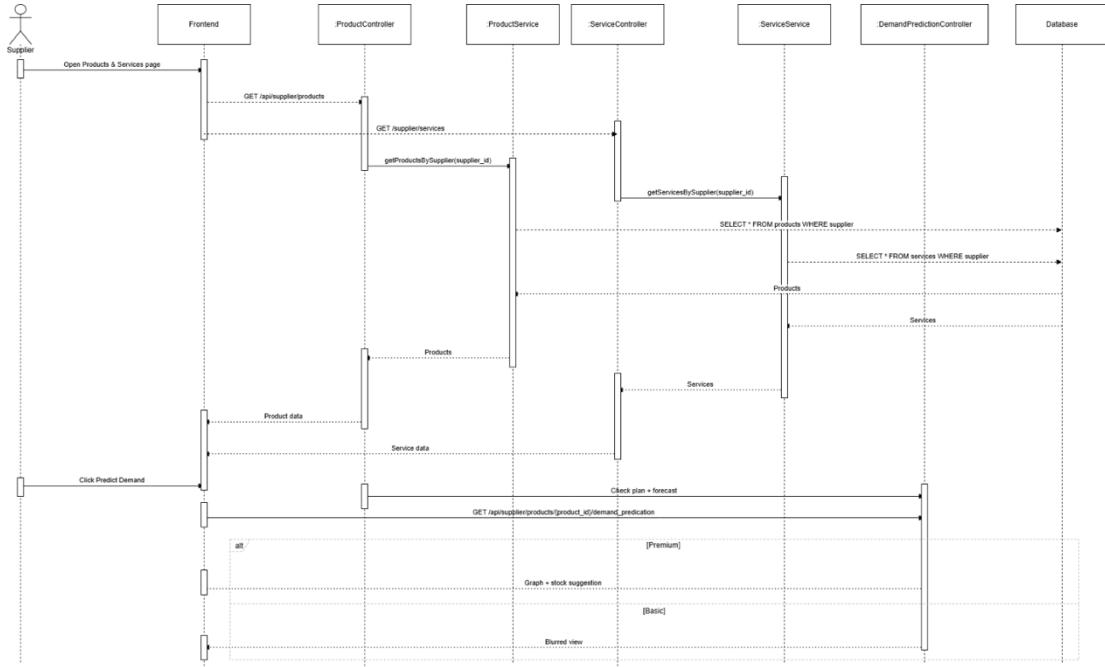


Figure 3-22: View Storefront Listings & Stock Demand Sequence Diagram

This diagram shows how a supplier views their listed products and services, and checks stock demand. Upon visiting the page, two requests are triggered to retrieve the supplier's products and services. When the supplier clicks on "Predict Demand," the system checks their subscription plan. If premium, it displays a demand forecast graph with suggestions; otherwise, a blurred version is shown.

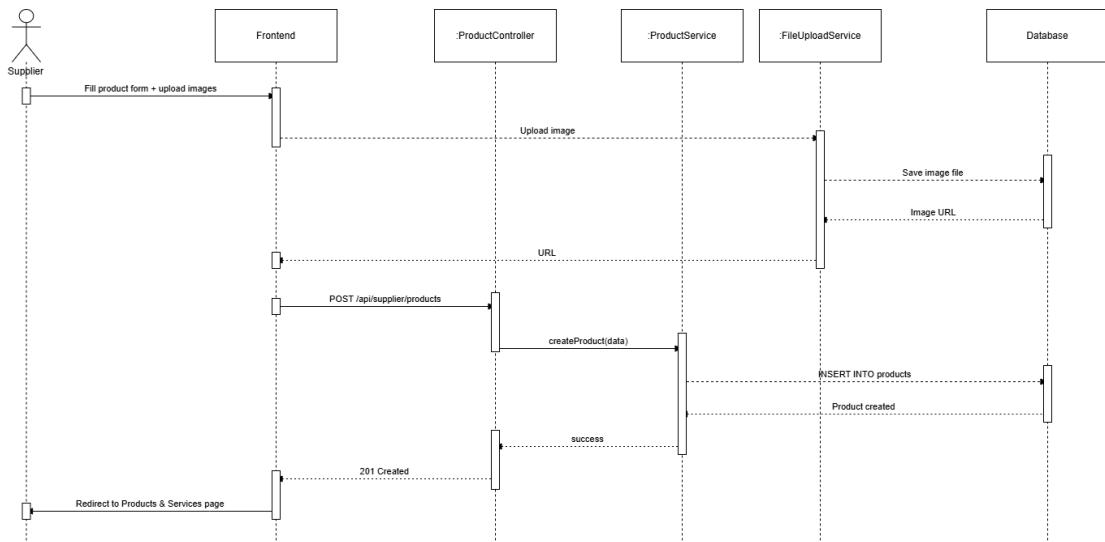


Figure 3-23: Create Product Sequence Diagram

This diagram explains the process of how a supplier adds a new product. The supplier fills out the product form and uploads images. The image is stored through the FileUploadService, returning a URL. The product details, including the image URL,

are then submitted to the backend, which saves them in the database. Once successful, the user is redirected to the Products & Services page.

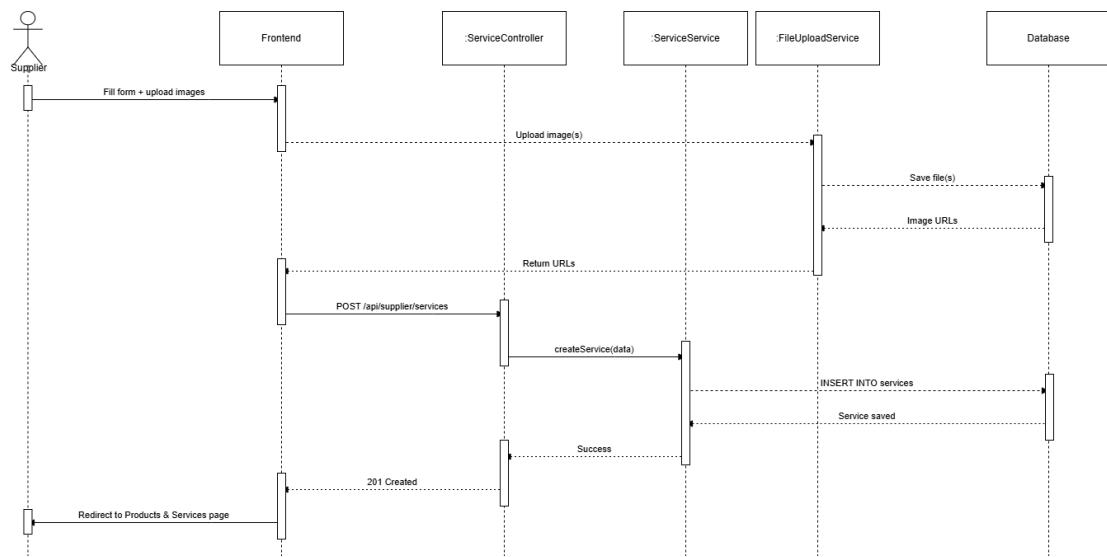


Figure 3-24: Create Service Sequence Diagram

This diagram illustrates the process for a supplier to create a new service listing. After filling out the service form and uploading images, the frontend uploads the images through the FileUploadService, which saves them and returns URLs. These URLs are included in the service creation request. The backend processes and stores the service details, then redirects the supplier to the Products & Services page upon successful creation.

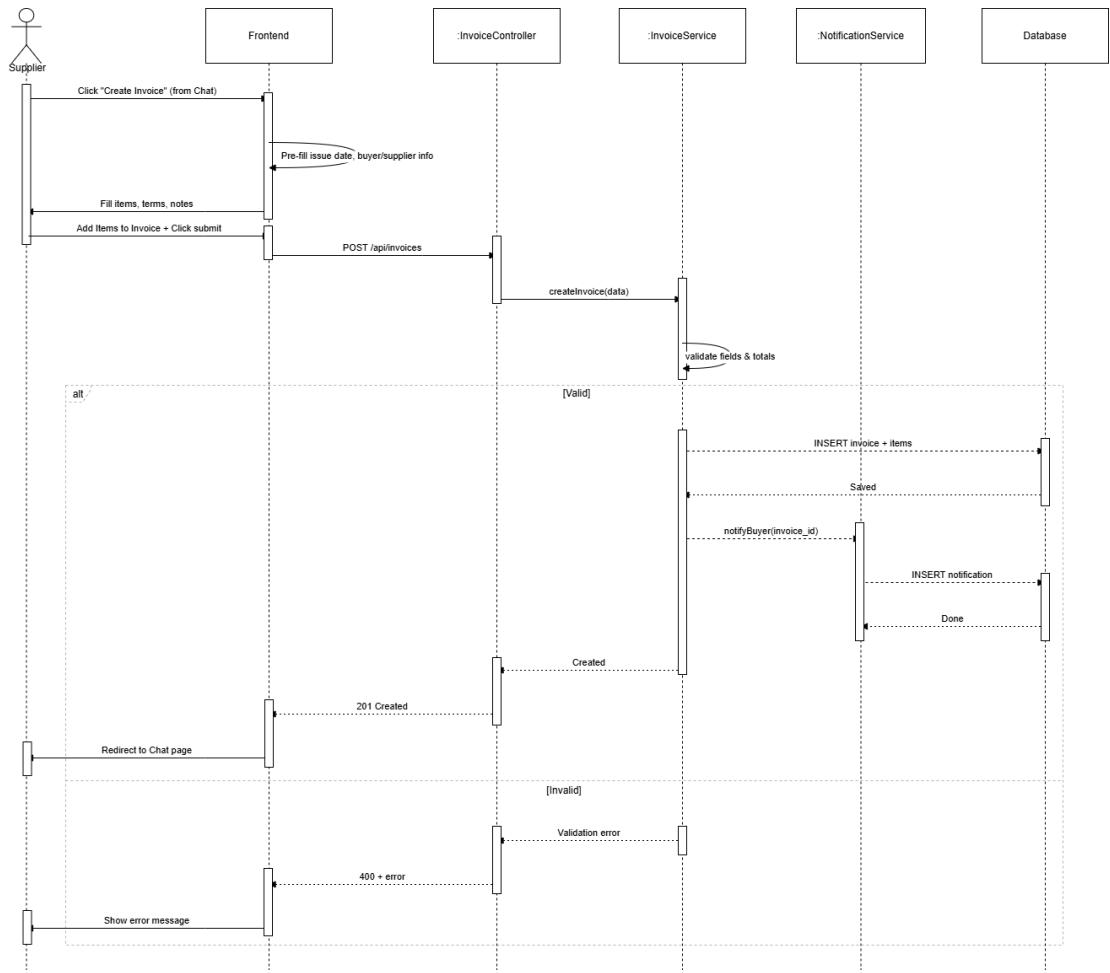


Figure 3-25: Create Invoice Sequence Diagram

This diagram shows how a supplier creates an invoice, typically after a conversation in chat. The frontend pre-fills known data, then the supplier adds invoice items and submits the form. The backend validates the data and, if valid, stores the invoice and its items in the database, then sends a notification to the buyer. If the data is invalid, the system returns an error, prompting the user to correct the form.

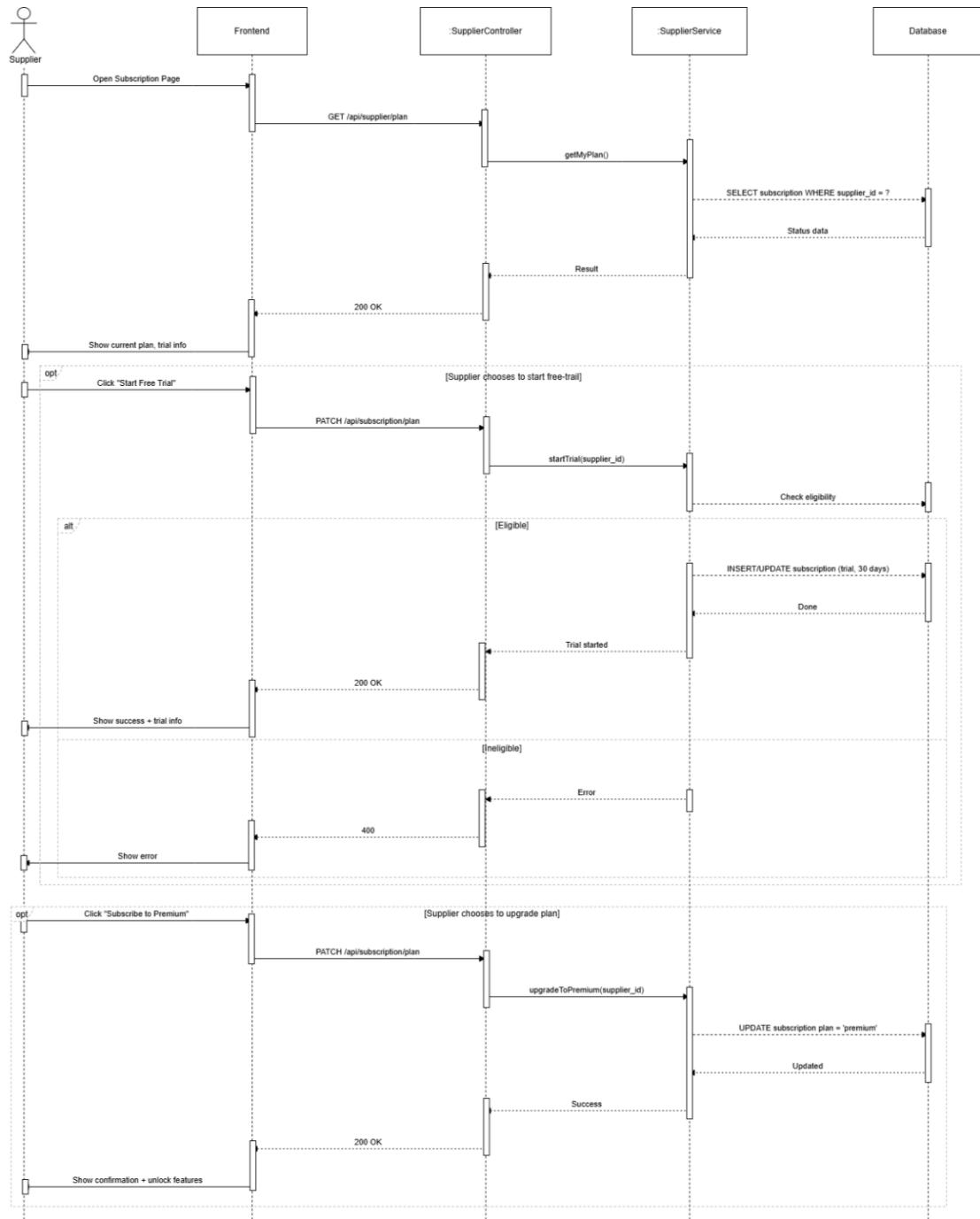


Figure 3-26: Subscription Management Sequence Diagram

This diagram illustrates how a supplier interacts with the subscription system. First, the current subscription plan is retrieved and displayed. If the supplier opts for a free trial, the system checks eligibility and either starts a trial or returns an error if ineligible. If the supplier decides to upgrade, the plan is updated to "premium," and access to premium features is granted.

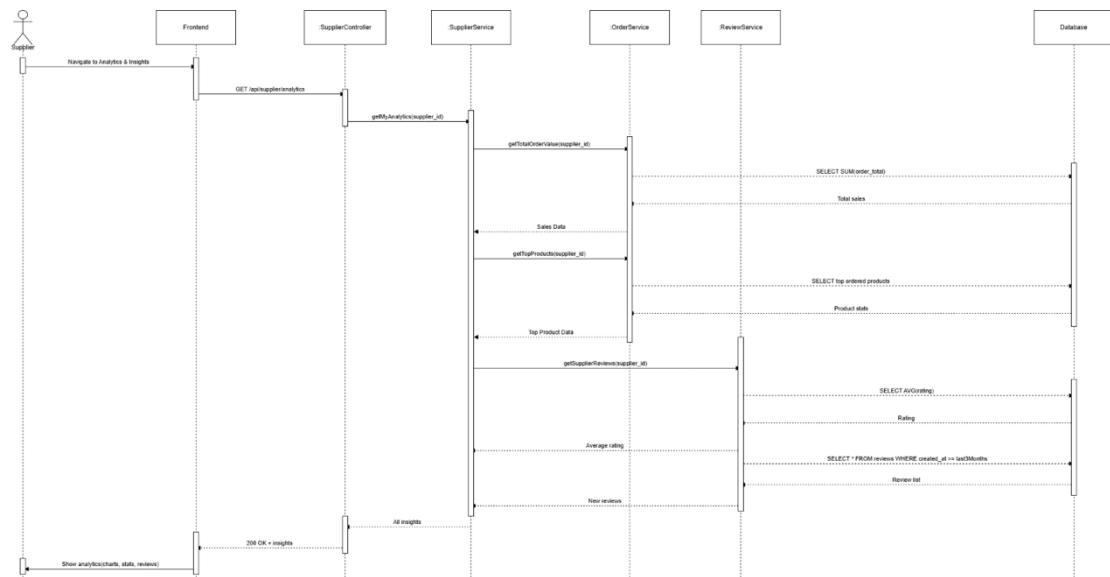


Figure 3-27: Supplier View Analytics & Insights Sequence Diagram

This diagram details the process followed when a supplier navigates to their analytics dashboard. Upon request, the system aggregates total order value, identifies top-performing products, and fetches recent reviews and average ratings. The insights are then returned to the frontend to be visualized as charts, statistics, and review summaries.

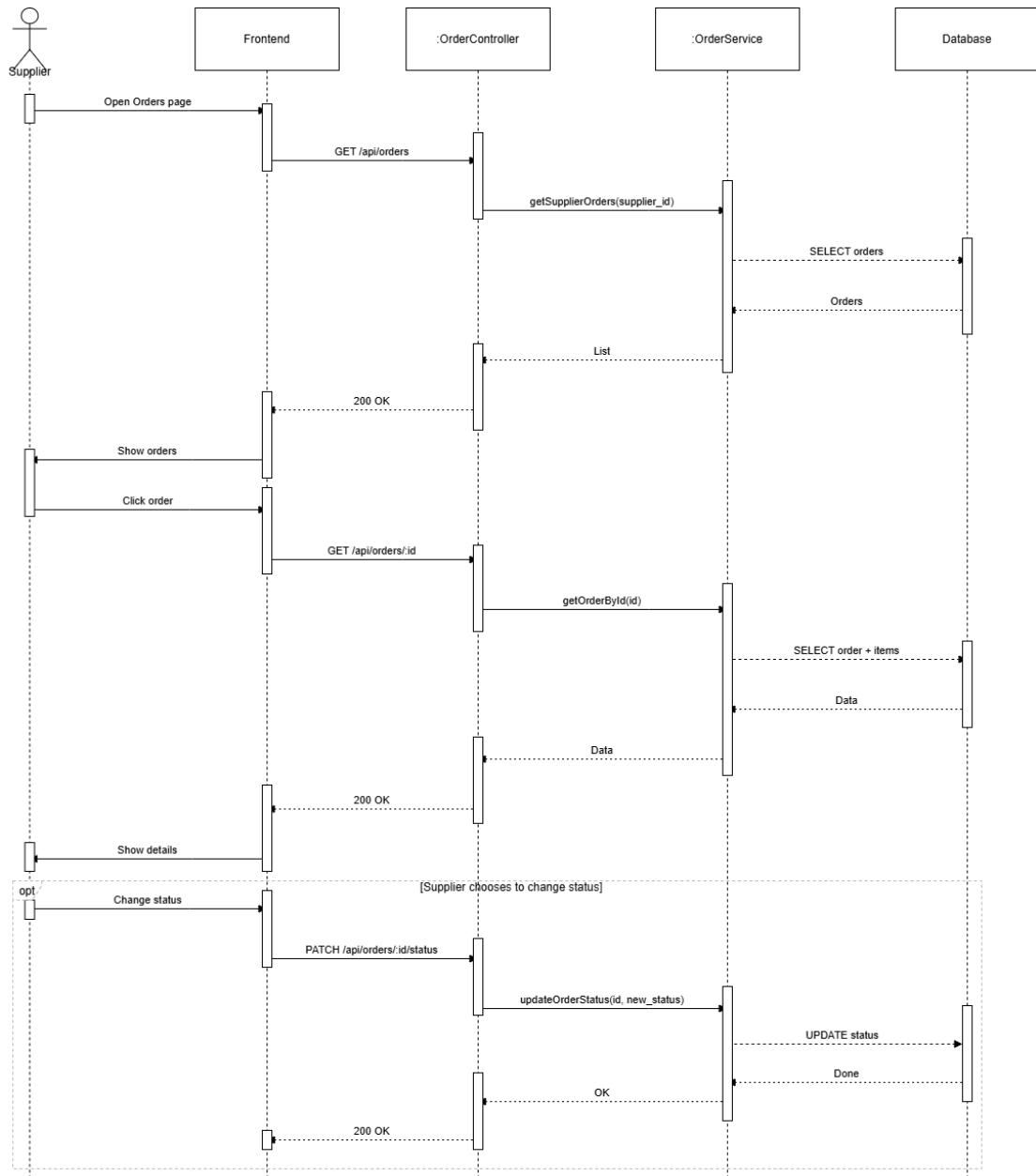


Figure 3-28: Supplier Order Management Sequence Diagram

This diagram shows how a supplier views and manages their orders. The process begins when the supplier opens the Orders page, triggering a request to fetch all their orders. Upon selecting an order, the system returns its details. If the supplier decides to update the order status (e.g., from Processing to Shipped), a PATCH request is sent, and the change is saved in the database.

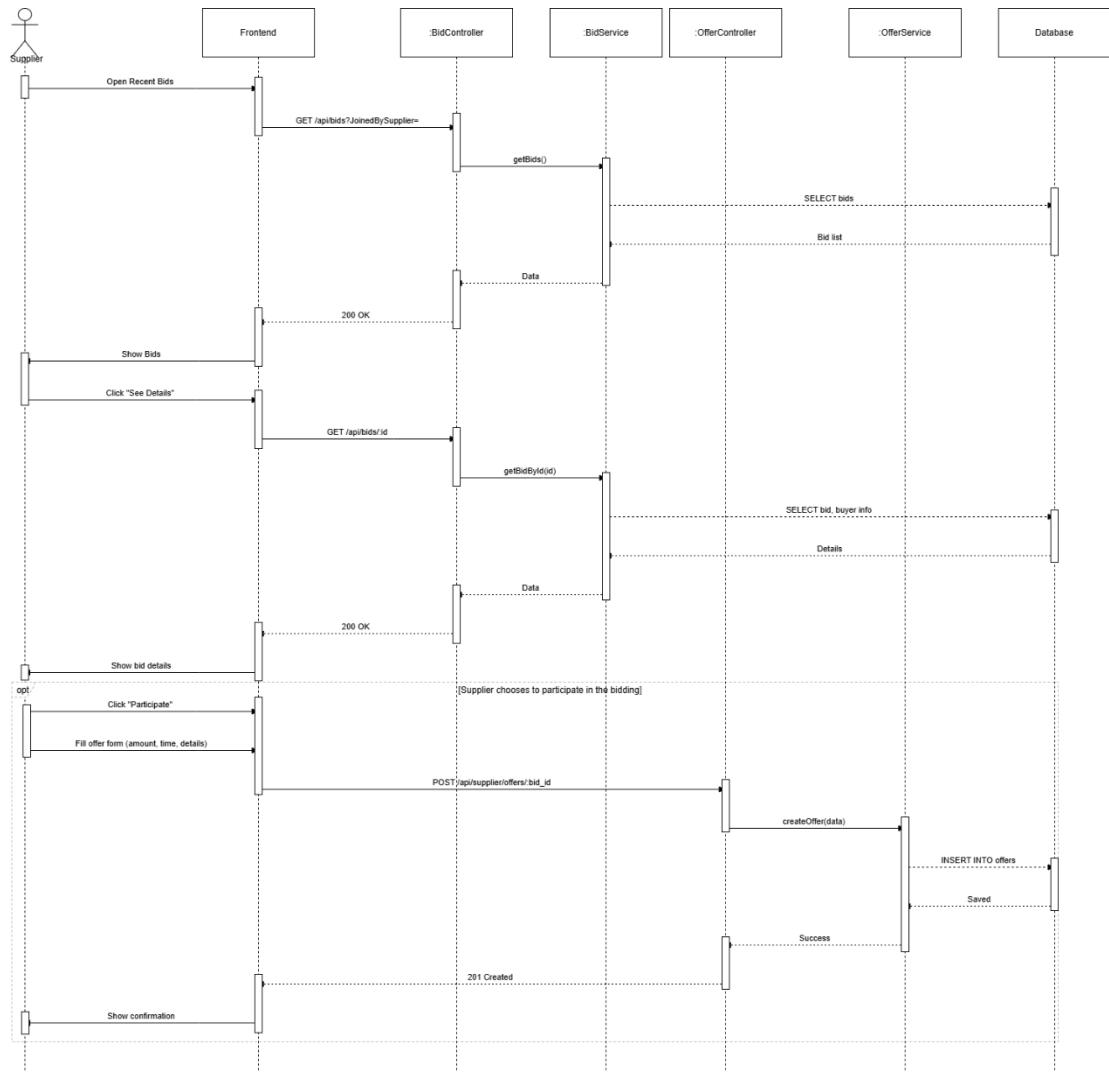


Figure 3-29: Supplier Bid Participation Sequence Diagram

This diagram illustrates how a supplier views recent bids and participates by submitting an offer. The sequence starts with fetching joined bids, followed by viewing specific bid details. When the supplier decides to participate, they fill out the offer form (proposed amount, timeline, execution details), which is then submitted and saved in the database through the OfferService. Upon success, a confirmation is shown.

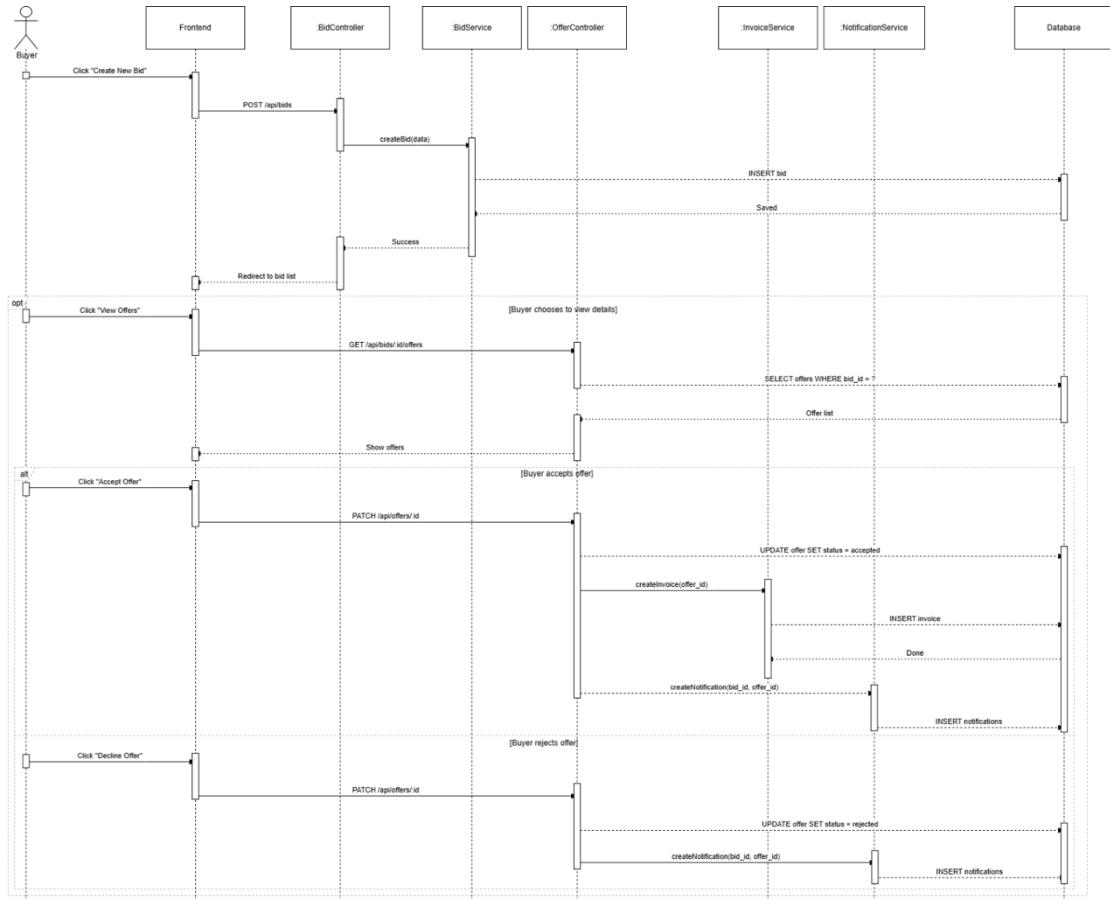


Figure 3-30: Buyer Bid Creation & Offer Handling Sequence Diagram

This diagram describes the buyer's journey in the bidding process. It starts with creating a new bid, which is stored in the database. The buyer then views all offers submitted for the bid. Based on their evaluation, they can either accept or decline an offer. Accepting an offer triggers invoice creation and notifies the supplier, while declining sends a rejection notification. This ensures structured and transparent communication throughout the bidding lifecycle.

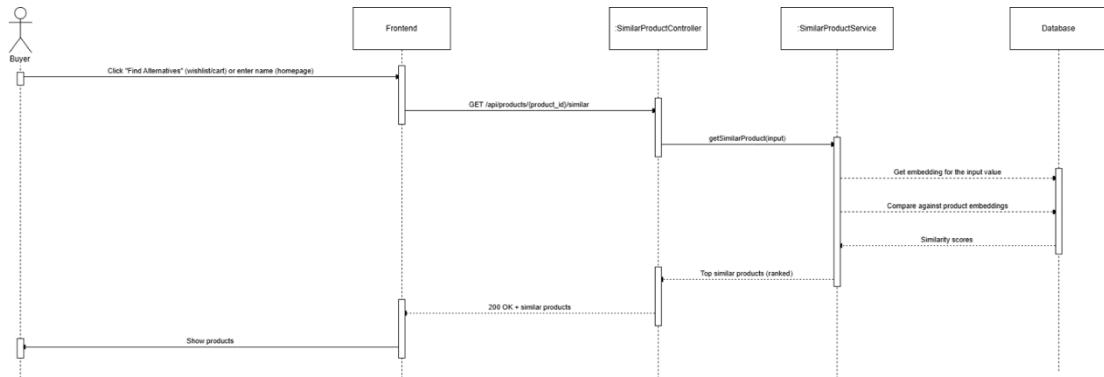


Figure 3-31: Find Similar Products Sequence Diagram

This diagram illustrates the process of retrieving similar products based on a selected item from the wishlist, cart, or homepage. When the user clicks "Find Alternatives,"

the system sends a request to fetch top similar products using product embeddings. The SimilarProductService queries the database for the input's embedding, compares it with other product vectors, and returns the most relevant alternatives. These alternatives are then ranked and displayed to the user as substitute options.

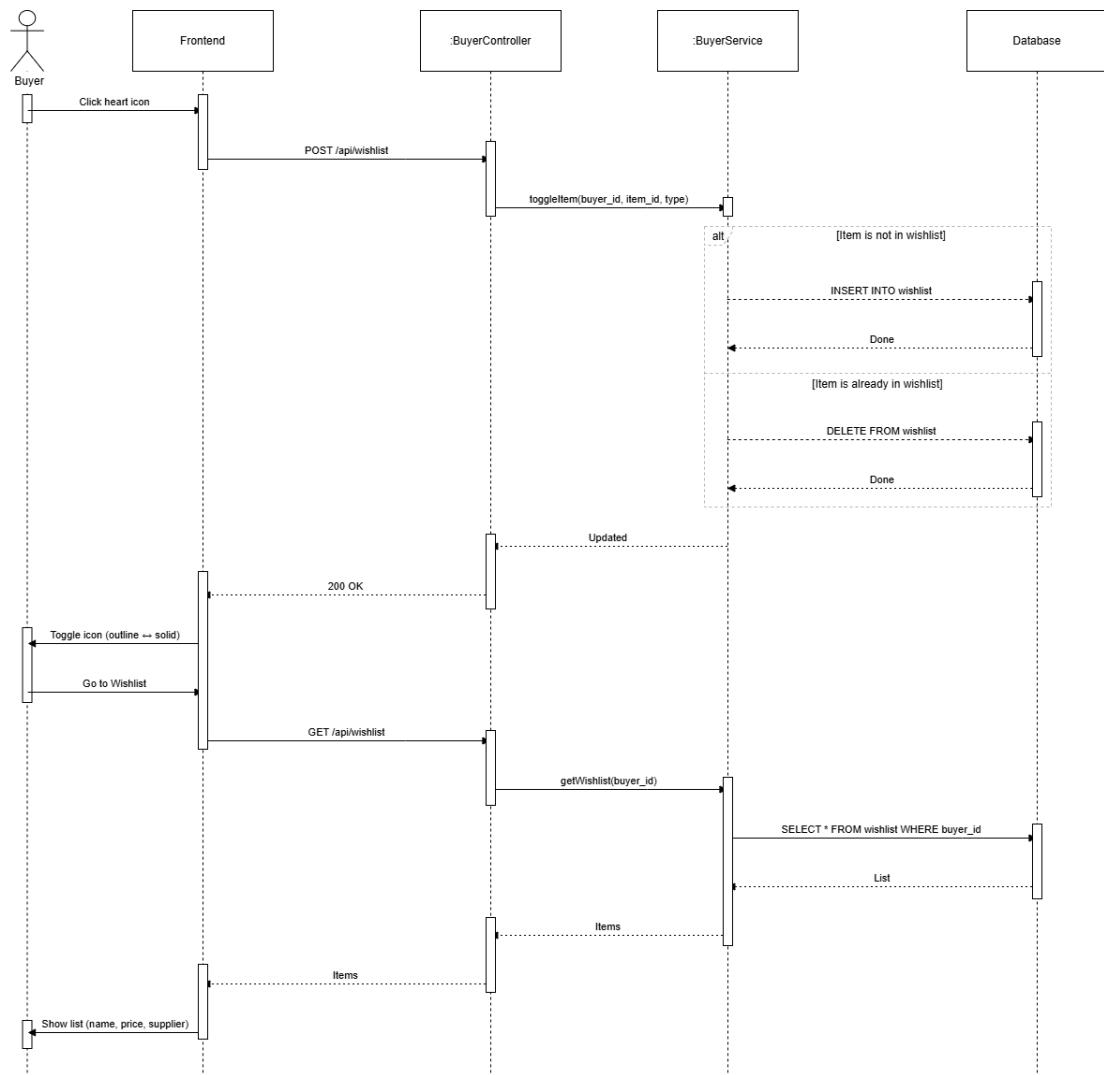


Figure 3-32: Wishlist Interaction Sequence Diagram

This diagram shows how a buyer can interact with the wishlist feature. When the user clicks the heart icon on a product or service, the system toggles the item's status in the wishlist by adding it if it's not present or removing it if it already exists. A visual update is triggered in the UI (icon changes). When the buyer visits the wishlist page, the system retrieves and displays a list of all saved items, including key details like name, price, and supplier.

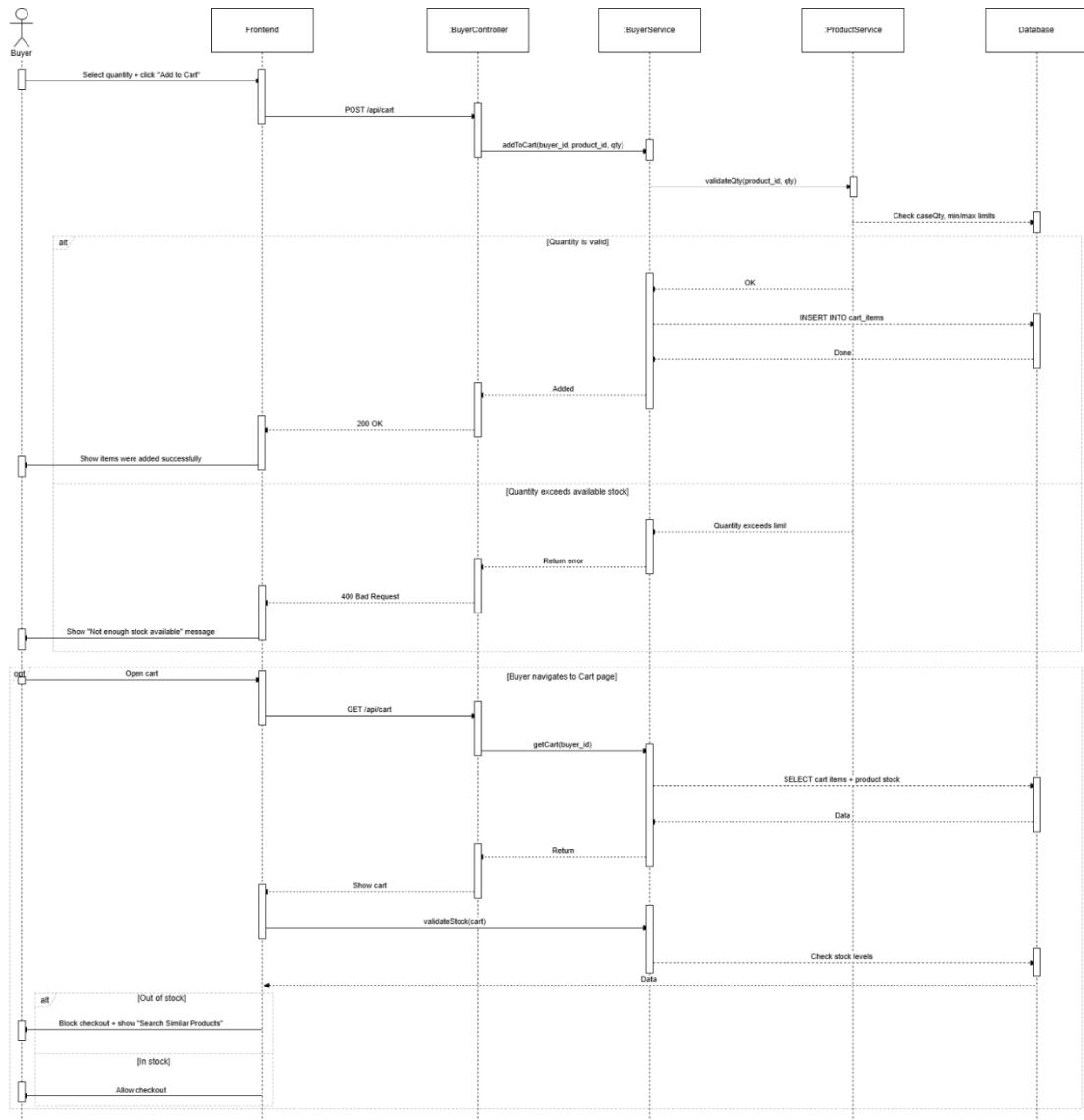


Figure 3-33: Add to Cart and Stock Validation Sequence Diagram

This diagram illustrates how a buyer adds a product to their cart. The system checks if the requested quantity is valid and within stock limits. If valid, the item is added to the cart and confirmed; otherwise, an error is returned. When the buyer views their cart, the system retrieves all cart items along with current stock levels. Before allowing checkout, the cart is validated to ensure all items are still in stock. If any item is unavailable, the system blocks checkout and suggests similar alternatives.

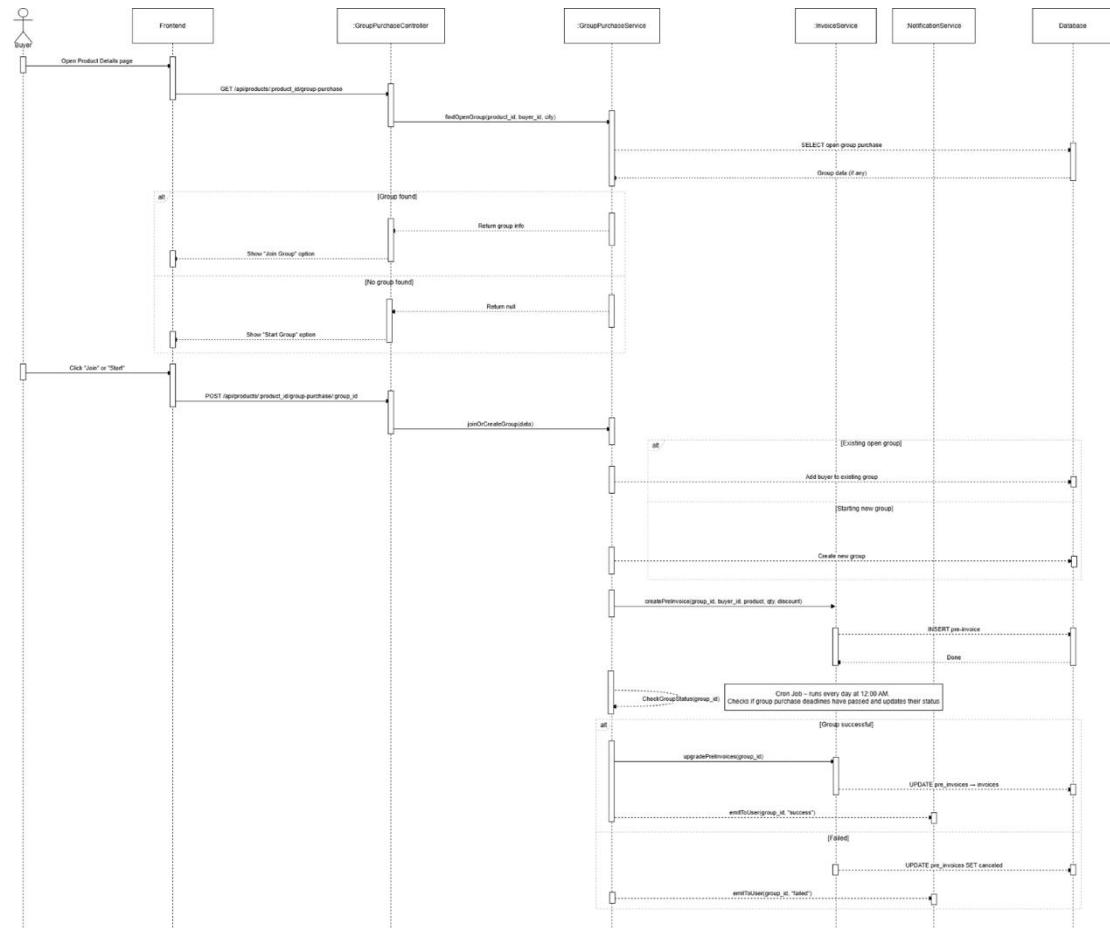


Figure 3-34: Join or Start Group Purchase Sequence Diagram

This diagram demonstrates how a buyer can either join an existing open group or start a new group purchase. When a buyer visits the product detail page, the system checks for open groups in their city. If one is found, the "Join Group" option is shown; otherwise, the "Start Group" option is offered. Upon confirming participation, the system creates a pre-invoice entry. A scheduled cron job runs daily to check if the group deadline has passed. If the minimum quantity is met, the group is marked as successful, and the pre-invoice is upgraded to a full invoice. Otherwise, the group is marked as failed, and the pre-invoice is canceled.

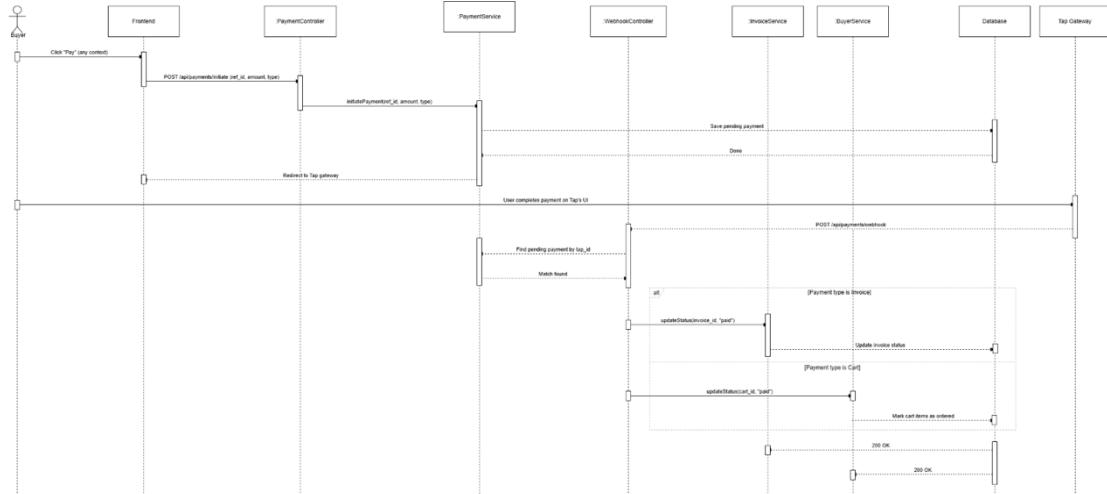


Figure 3-35: Payment Flow Using Tap Gateway Sequence Diagram

This diagram illustrates the complete flow of a buyer making a payment via the Tap payment gateway. When the buyer initiates payment, the system stores the pending transaction and redirects the user to the Tap UI. After the user completes payment, Tap sends a webhook back to the server. The system looks up the matching pending payment using the Tap transaction ID. Based on the payment context (Cart or Invoice), the appropriate module updates its status: either the invoice is marked as paid or cart items are confirmed as ordered.

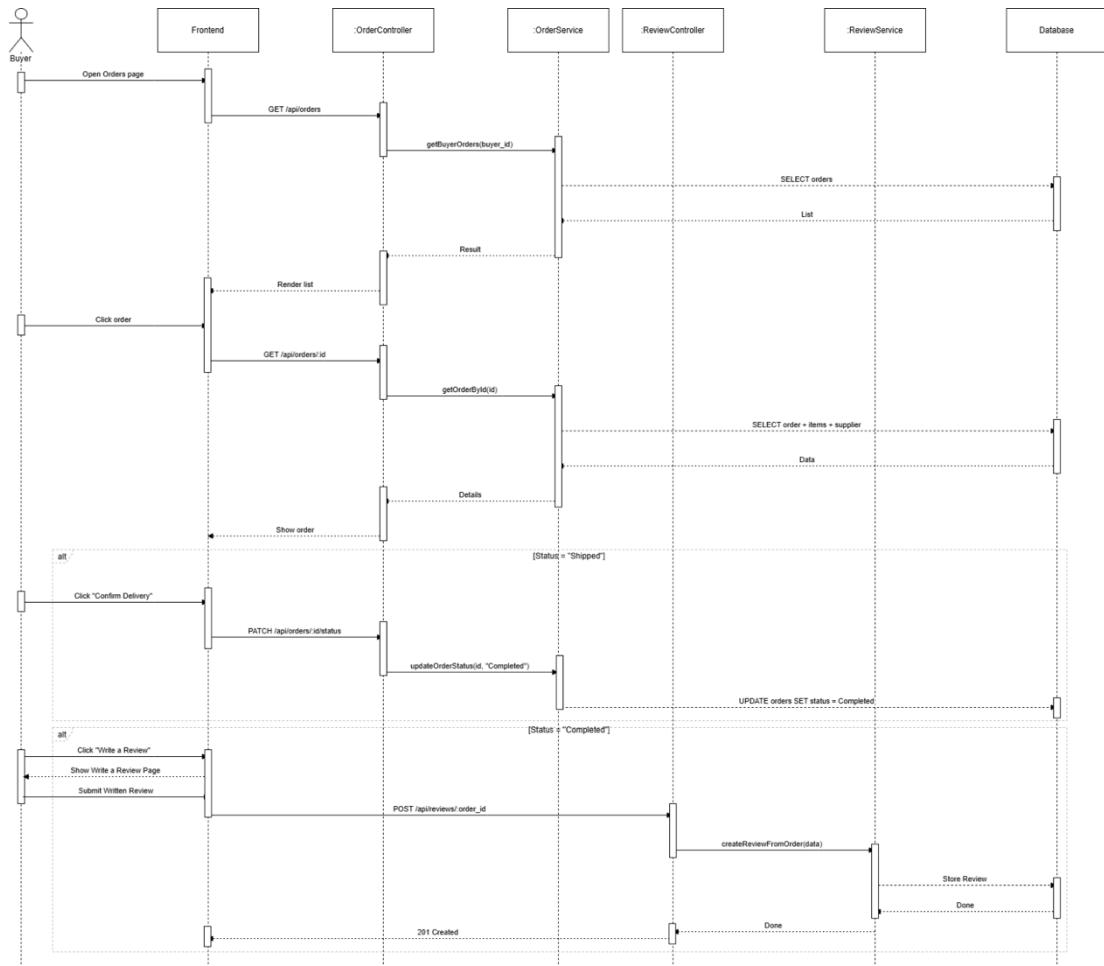


Figure 3-36: Buyer Order Completion and Review Submission Sequence Diagram

This diagram outlines how a buyer interacts with their order. After opening the orders page and viewing order details, the buyer can confirm delivery if the order status is "Shipped", which updates the status to "Completed". Once completed, the buyer may optionally submit a written review. The review is created and stored via the ReviewController and ReviewService.

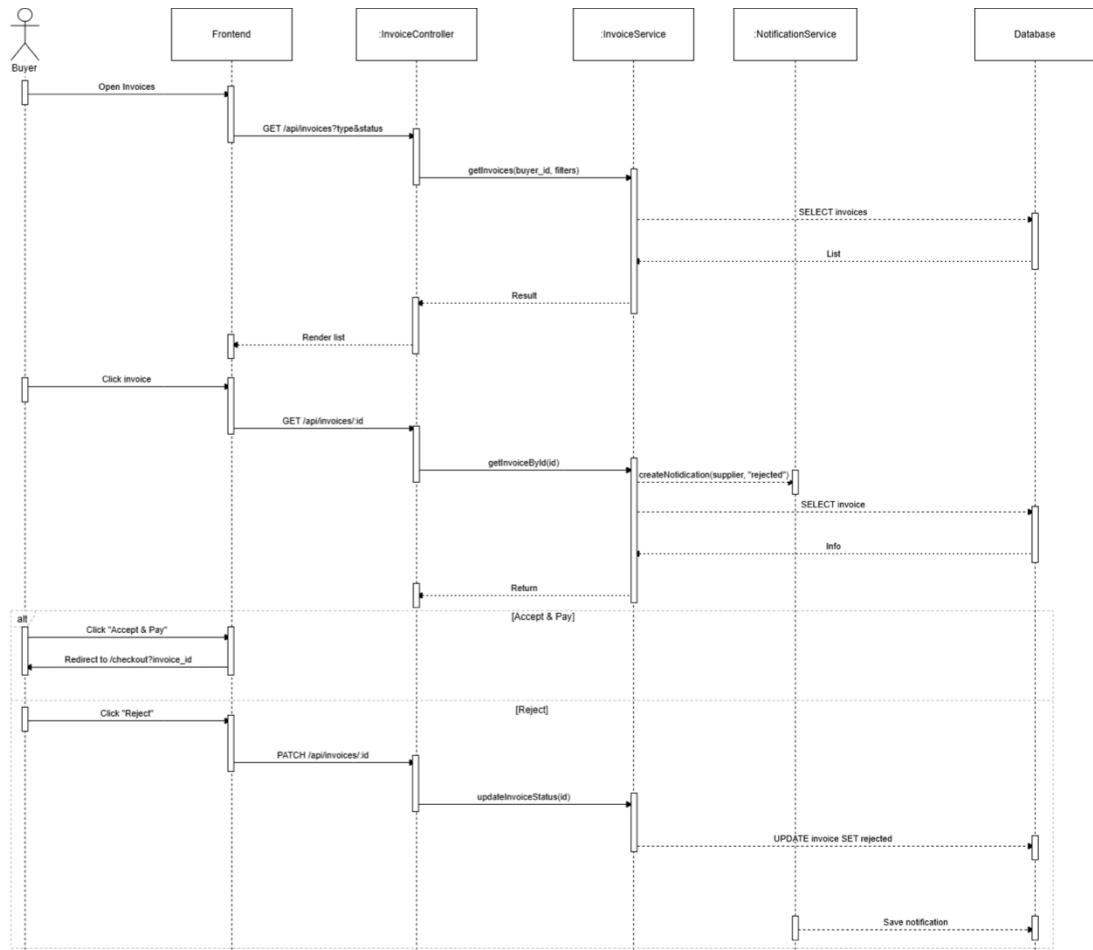


Figure 3-37: View Invoices Sequence Diagram

This diagram shows the buyer workflow for interacting with an invoice. The buyer first fetches and views a list of invoices, selects one, and sees the detailed information. Based on the content, the buyer can choose to either "Accept & Pay" (which redirects to the payment flow) or "Reject" the invoice. Upon rejection, the system updates the invoice status to be rejected and sends a notification to the supplier.

The sequence diagrams presented above illustrate the dynamic interactions between users, frontend interfaces, backend controllers, services, and external systems across various use cases in Silah. They serve to highlight the flow of data, validate system logic, and ensure proper coordination between components. These diagrams not only aid in understanding user behavior and system responsibilities but also provide a valuable reference for future development and debugging. Together with the class diagrams, they form a complete picture of the platform's architecture and design.

3.2.3 Class Diagrams

The figure below illustrates the high-level usage relationships between the backend modules of the Silah system. Each arrow indicates a dependency or interaction, where each module calls service or uses logic defined in another. Modules are grouped and color-coded based on their responsibility domains.

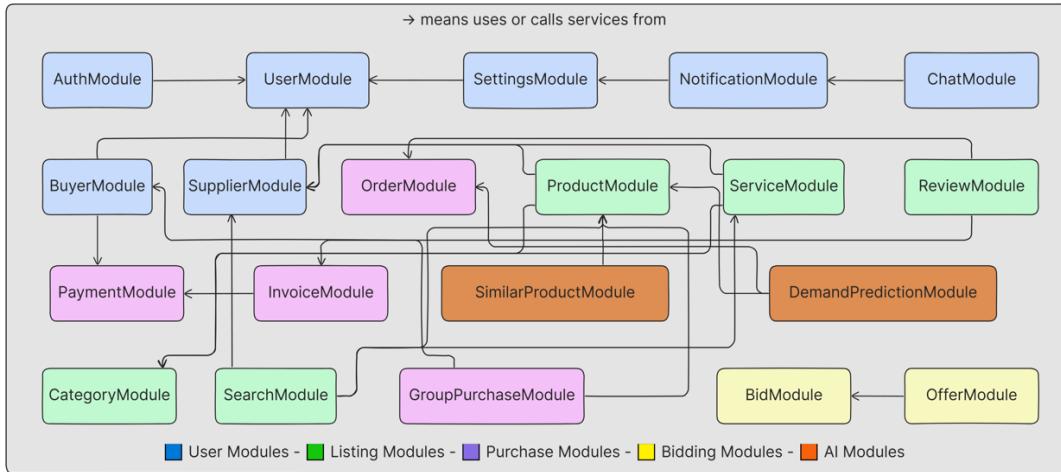


Figure 3-38: Module usage relationships in Silah's backend

This overview was positioned before the class diagrams to give readers a broader architectural context, making it easier to follow the object-level structure and interactions.

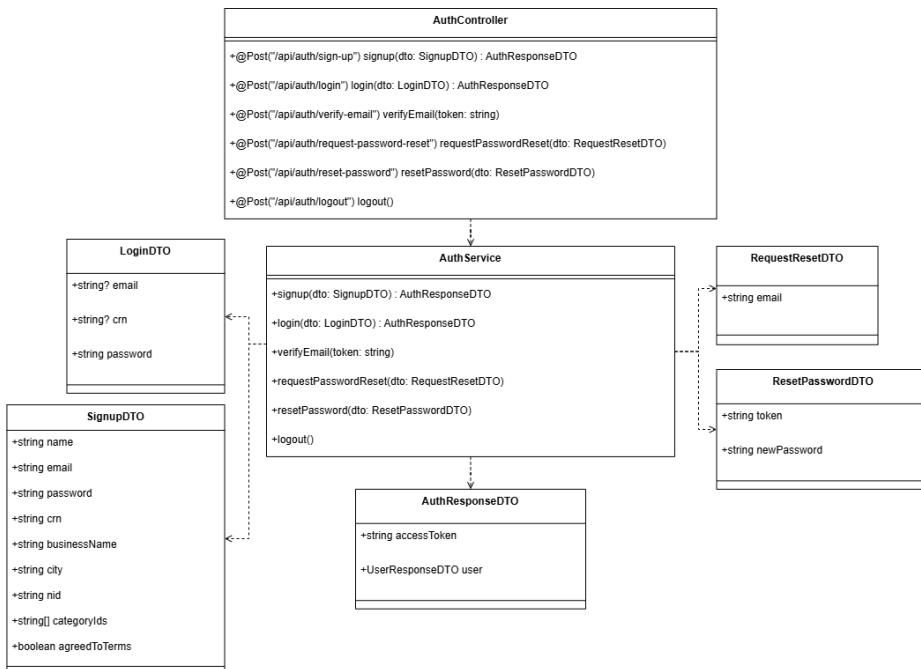


Figure 3-39: Authentication Module Class Diagram

This diagram shows the structure of the AuthModule, which handles sign-up, login, email verification, and password reset. The AuthController manages API endpoints and delegates logic to AuthService. Different DTOs are used for input (SignupDTO,

LoginDTO, etc.) and output (AuthResponseDTO). The design separates concerns clearly, making the module easy to maintain.

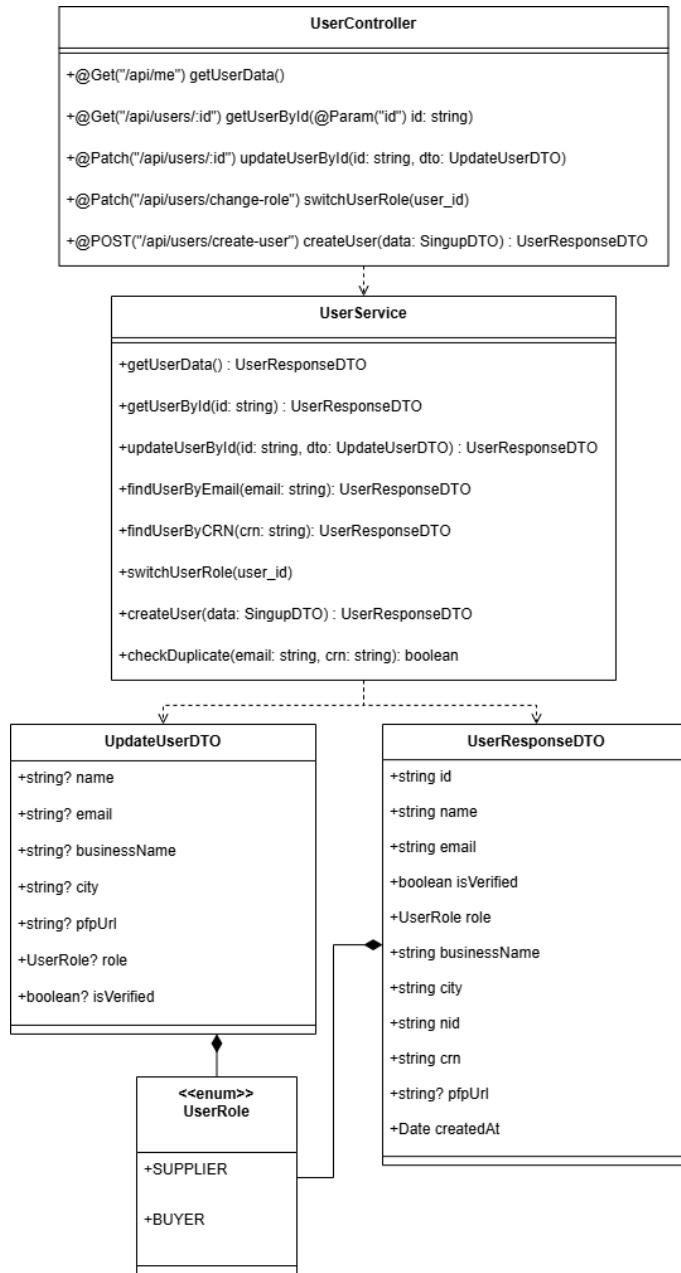


Figure 3-40: User Module Class Diagram

This diagram shows how the UserModule handles user-related operations like fetching user data, updating profiles, and switching roles. The UserController exposes the routes, while UserService handles the logic. It uses UpdateUserDTO and UserResponseDTO for request and response data, and the UserRole Enum defines whether the user is a buyer or a supplier.

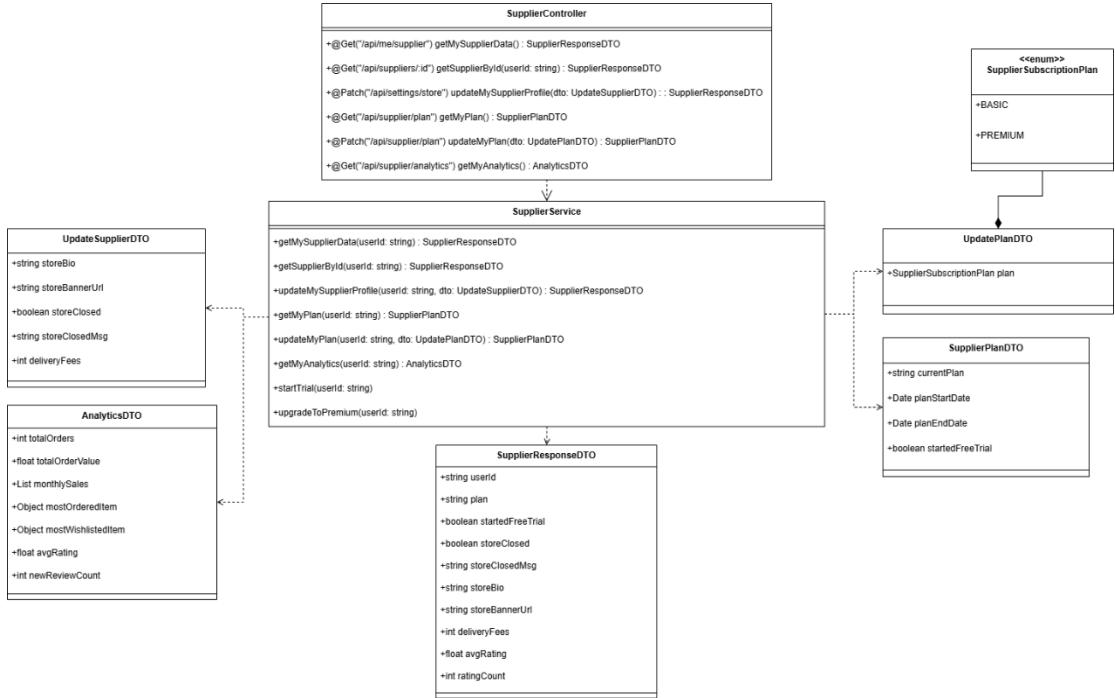


Figure 3-41: Supplier Module Class Diagram

This diagram outlines how the **SupplierModule** manages supplier profiles, subscription plans, and analytics, using various DTOs to update store settings, track performance, and upgrade plans.

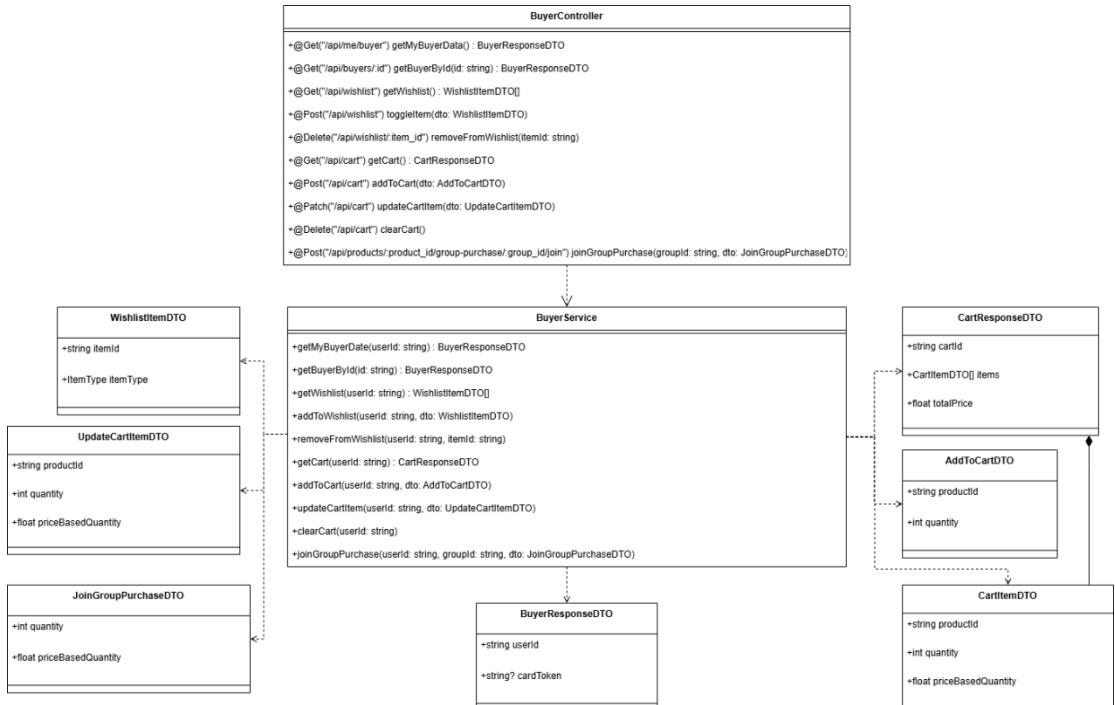


Figure 3-42: Buyer Module Class Diagram

This diagram shows how the **BuyerModule** manages buyer profiles, wishlists, carts, and group purchase participation using well-structured DTOs for each feature.

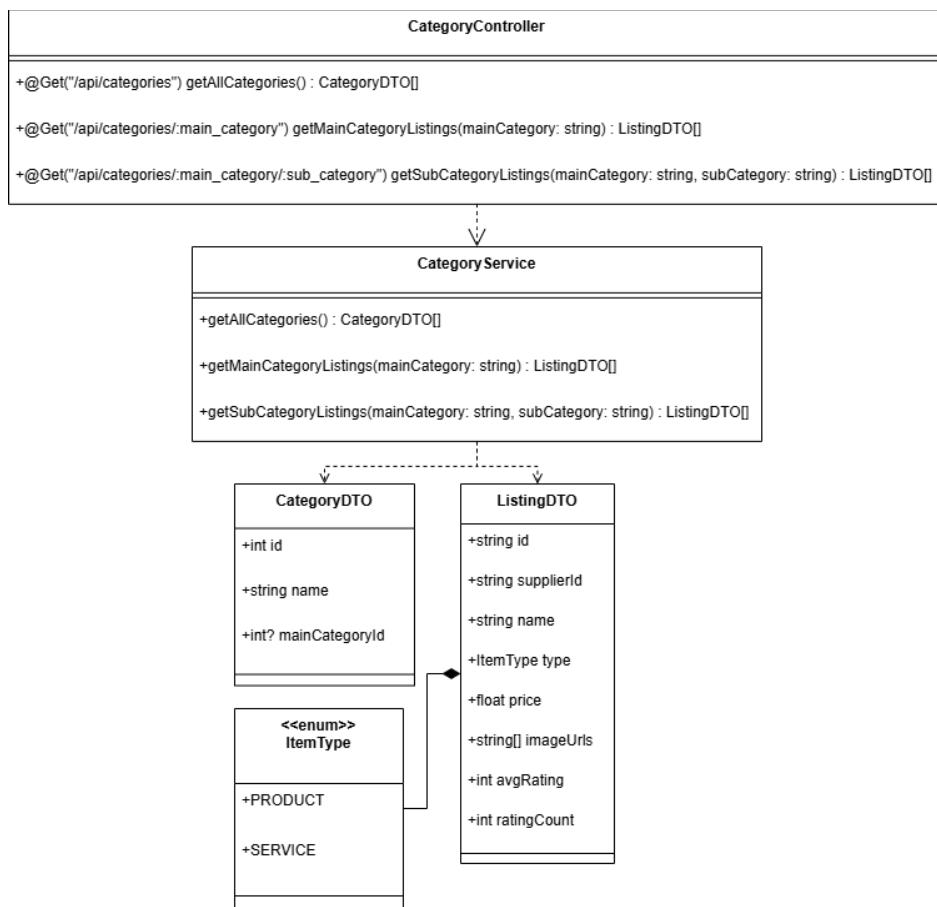


Figure 3-43: Category Module Class Diagram

This diagram shows how the CategoryModule fetches all categories and returns related listings using CategoryDTO and ListingDTO, with listings classified as either products or services.

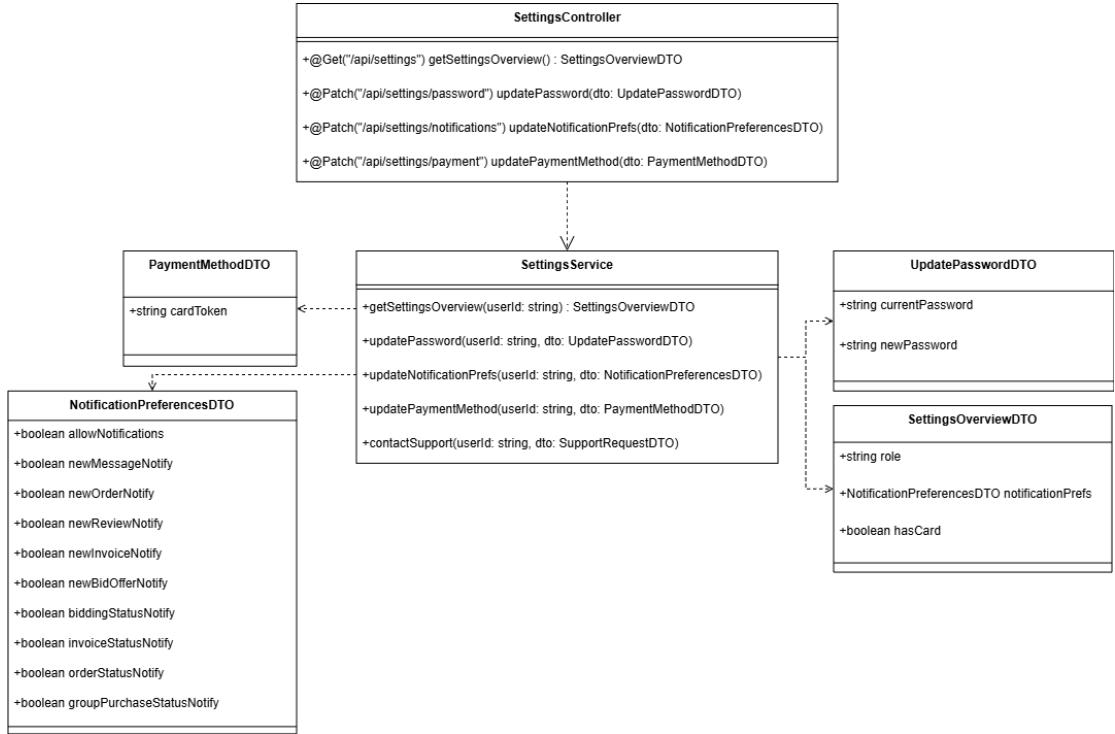


Figure 3-44: Settings Module Class Diagram

This diagram shows how the SettingsModule handles user settings such as password, notification preferences, and payment methods using dedicated DTOs for each type of update.

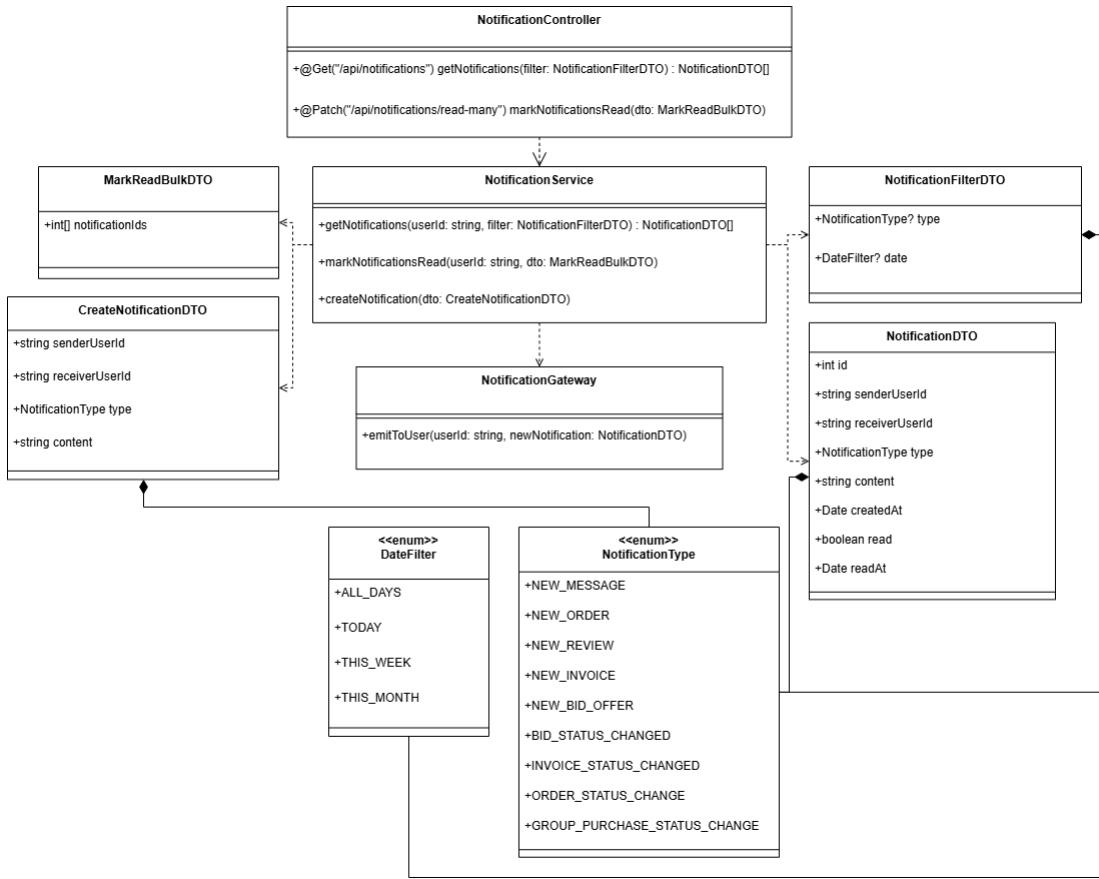


Figure 3-45: Notification Module Class Diagram

This diagram presents the NotificationModule, which manages filtering, bulk marking, and real-time delivery of notifications using various DTOs and Enums for type and date filtering.

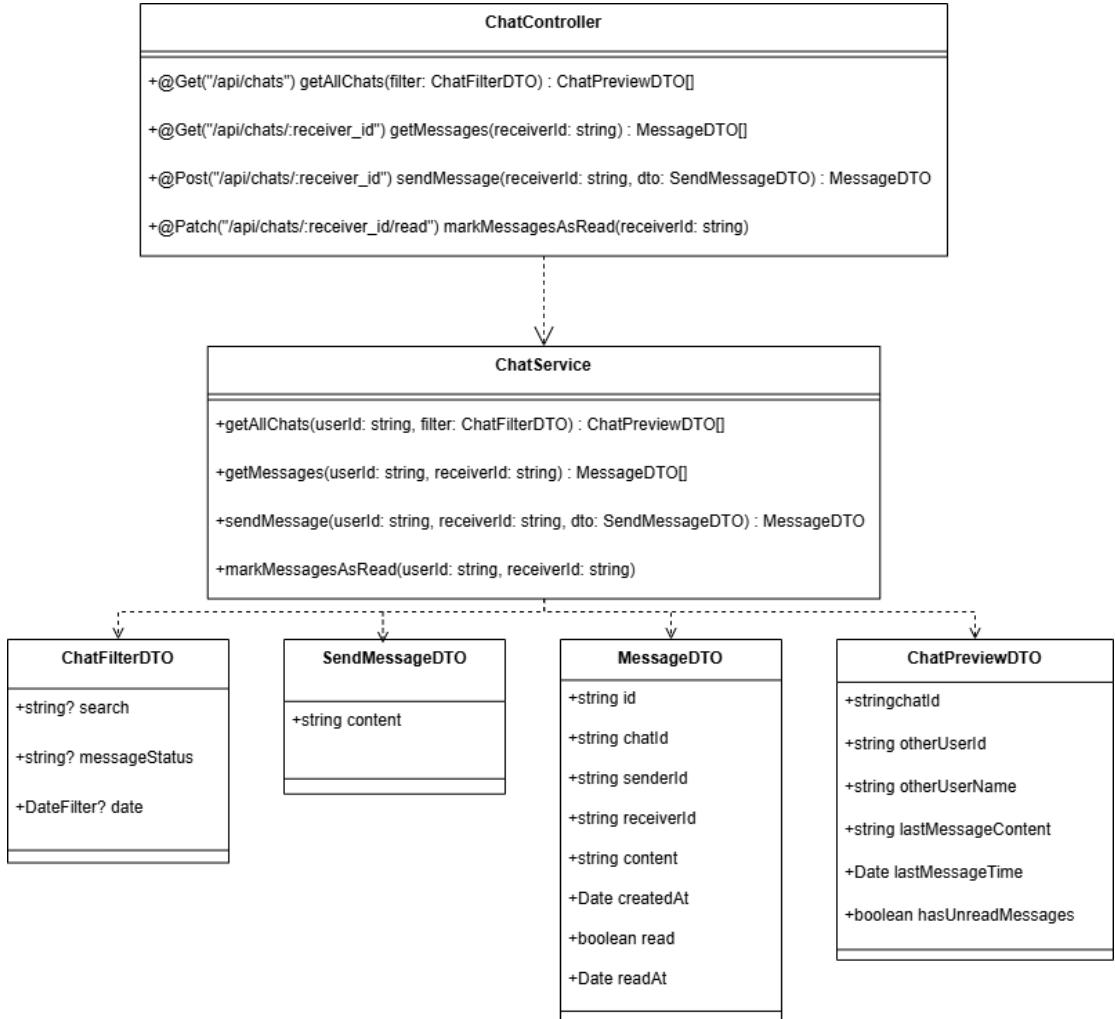


Figure 3-46: Chat Module Class Diagram

This diagram describes the ChatModule, which handles message sending, retrieval, and status updates. It supports filtering via ChatFilterDTO, real-time message tracking through MessageDTO, and shows conversation summaries with ChatPreviewDTO.

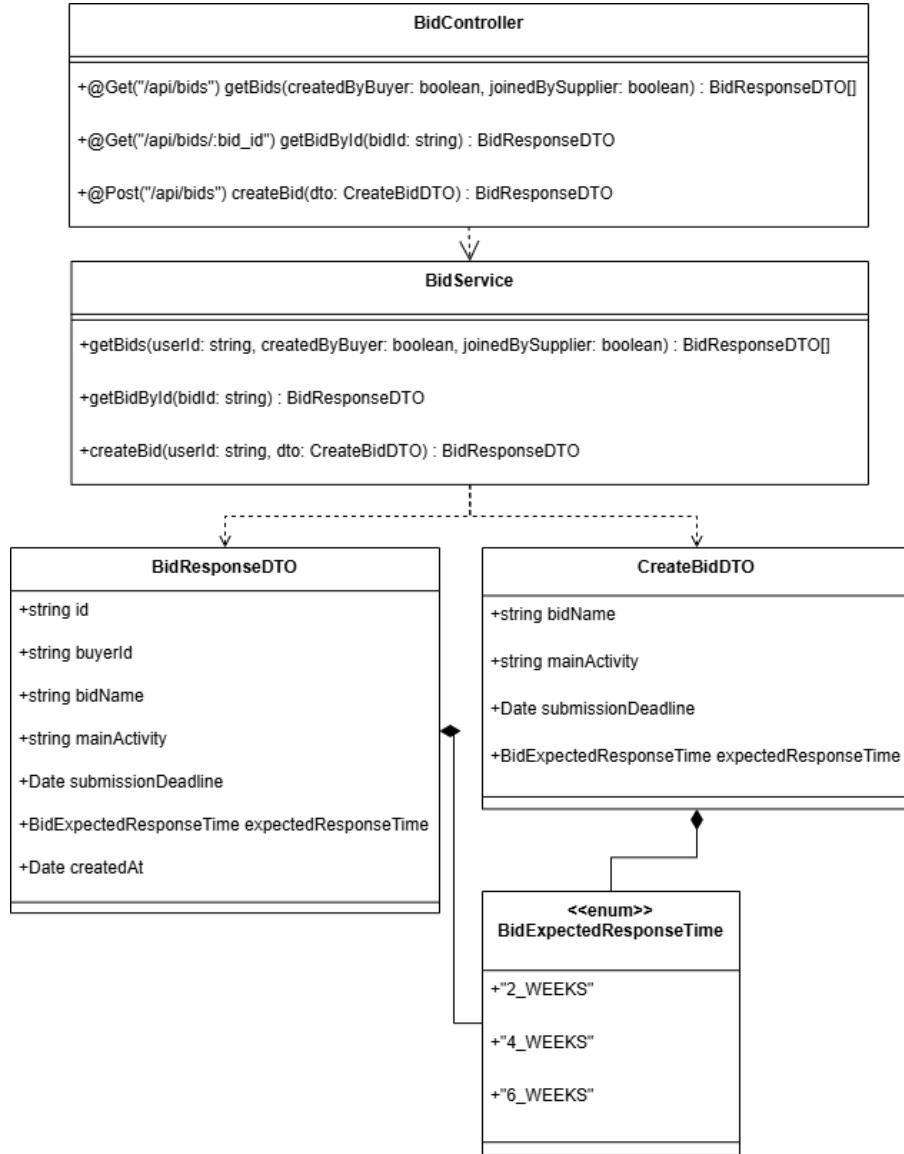


Figure 3-47: Bid Module Class Diagram

This diagram illustrates the structure of the BidModule, where buyers can create bids and suppliers can browse them. The module uses Enums like `BidExpectedResponseTime` to define delivery expectations and returns structured responses using `BidResponseDTO`.

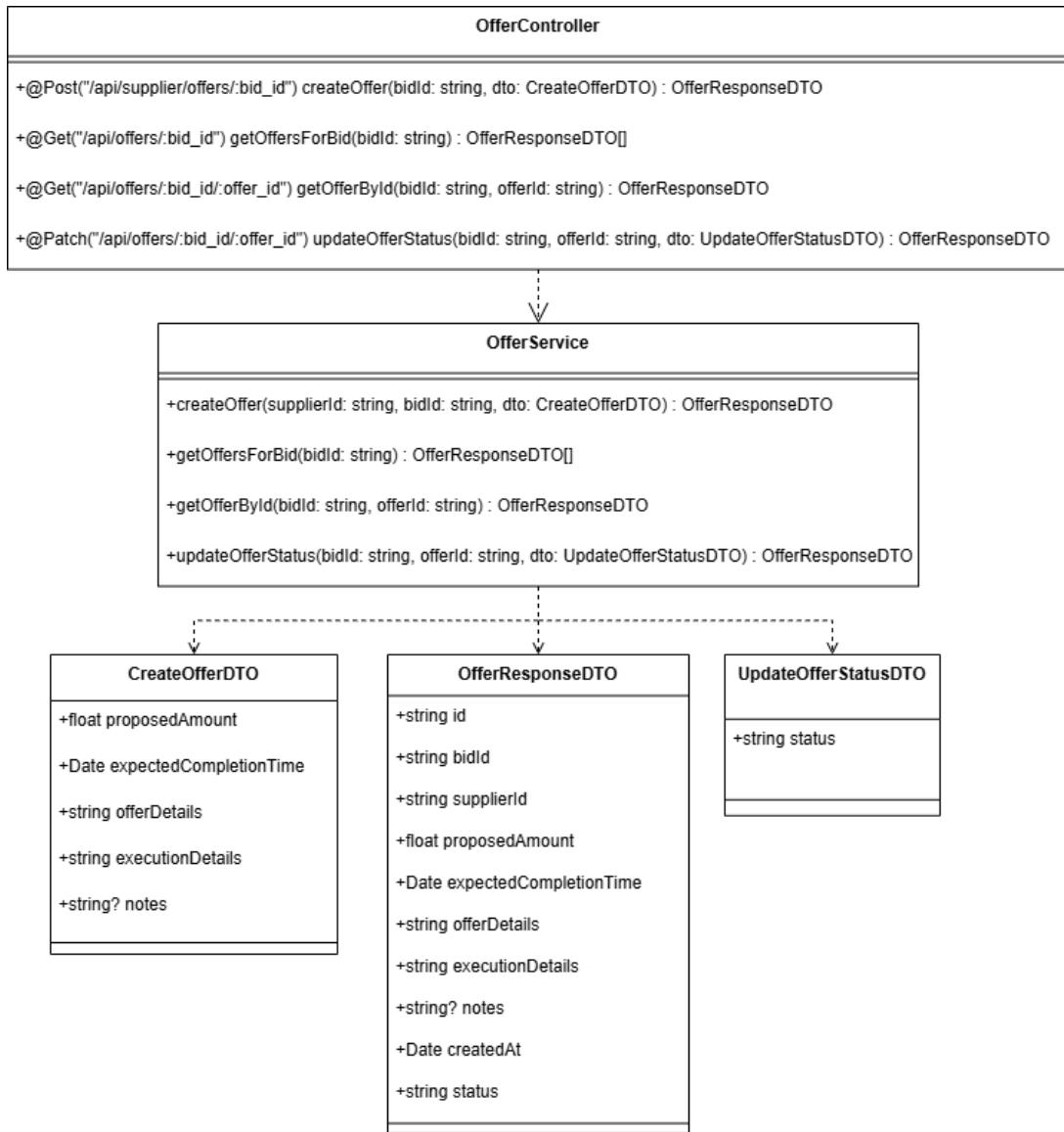


Figure 3-48: Offer Module Class Diagram

This diagram presents how suppliers create and manage offers for specific bids. The **OfferService** handles offer creation, retrieval, and status updates, while **OfferResponseDTO** standardizes the output.

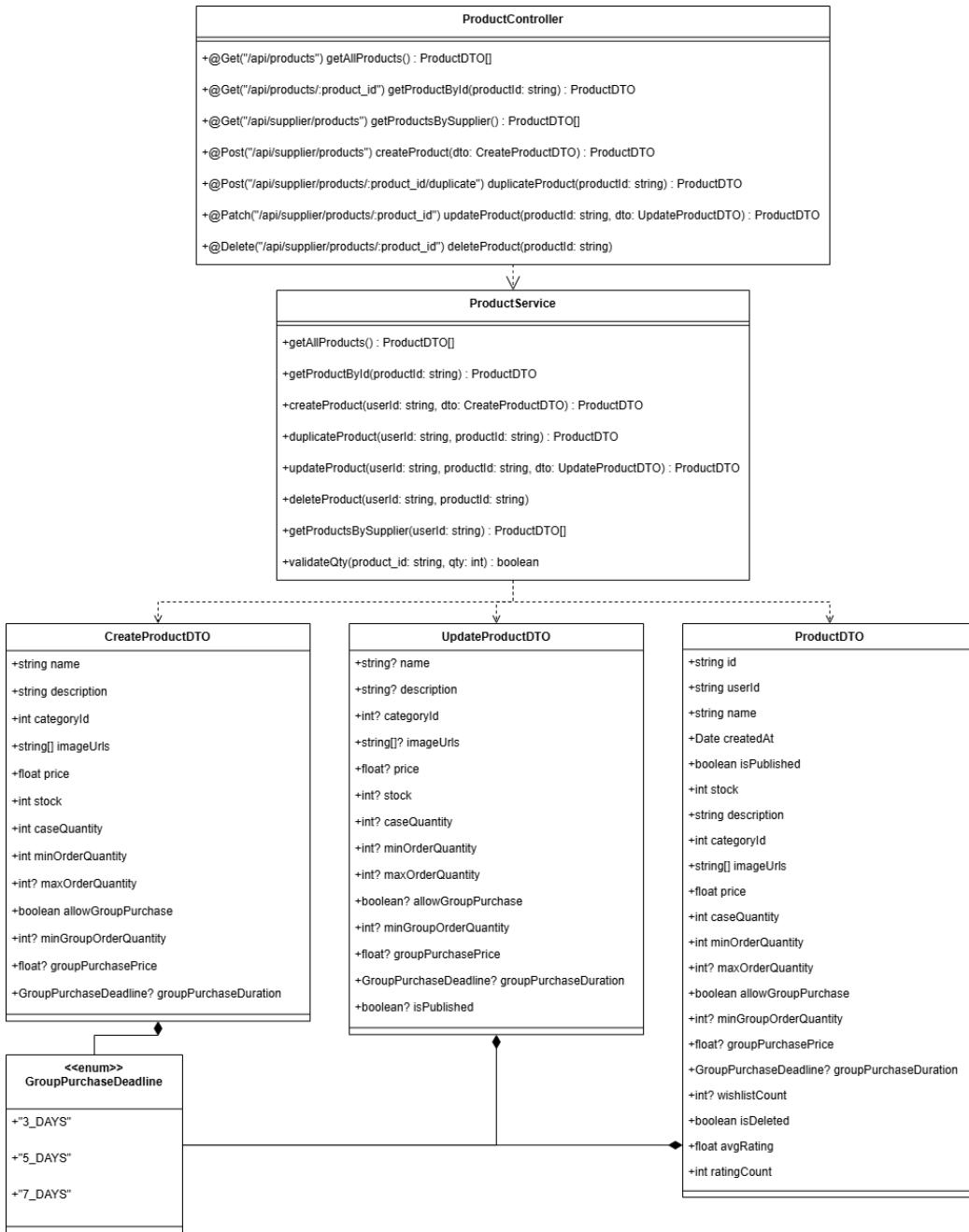


Figure 3-49: Product Module Class Diagram

This diagram defines how suppliers manage their product listings. It includes creation, duplication, validation, and publishing products, with support for group purchases and quantity limits.

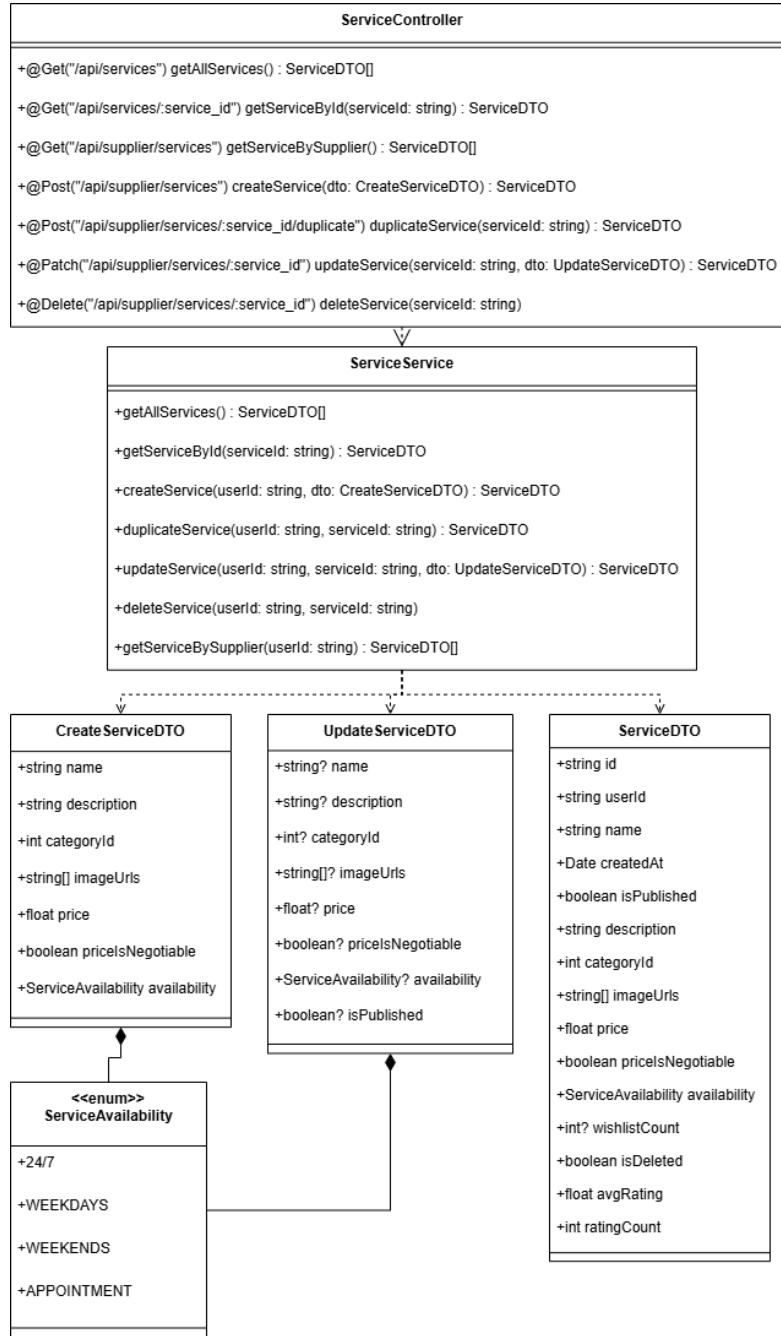


Figure 3-50: Service Module Class Diagram

This diagram shows how suppliers create, update, duplicate, and delete services. It supports price negotiation, category assignment, availability types (e.g., 24/7 or appointments), and service publication.

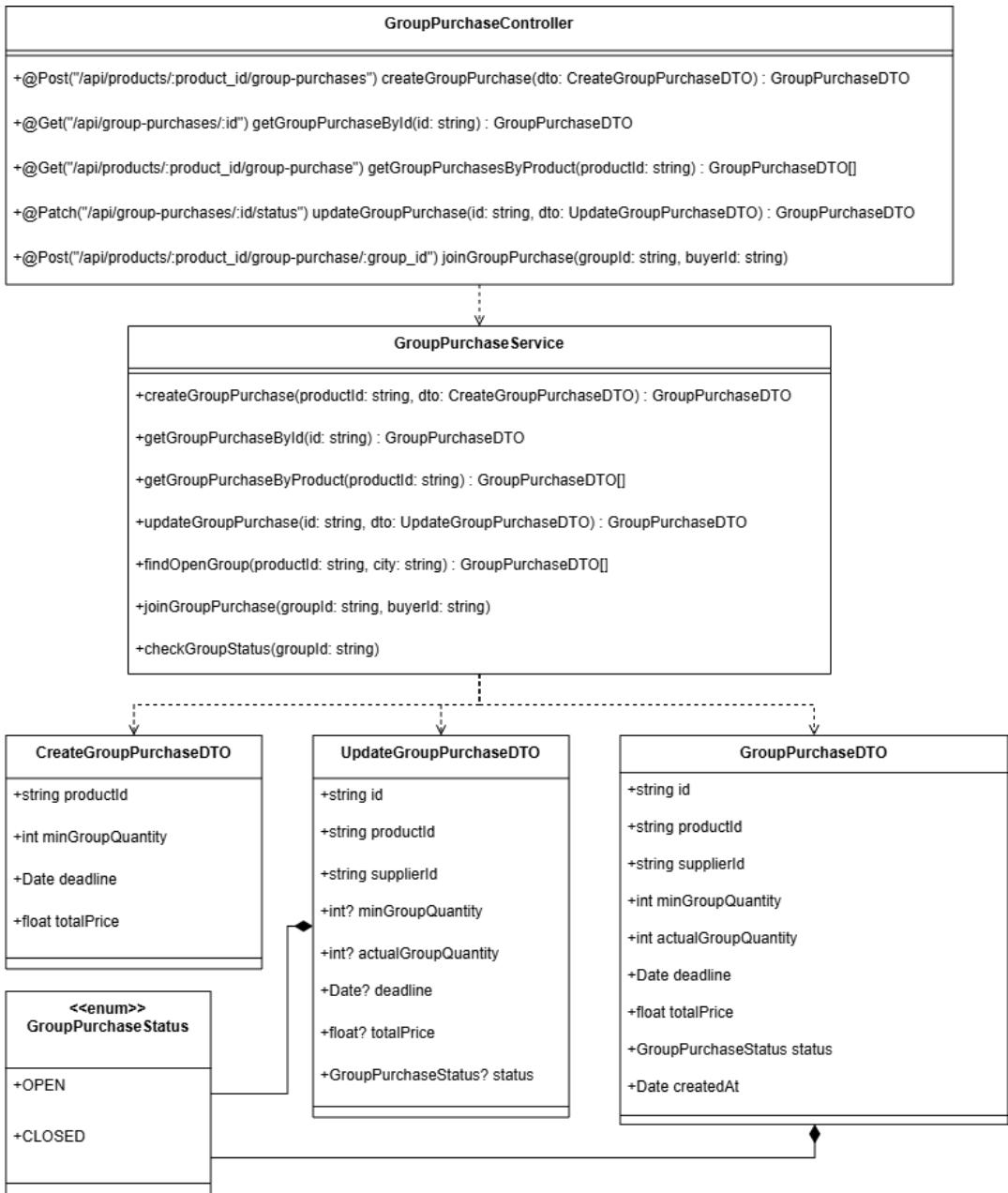


Figure 3-51: Group Purchase Module Class Diagram

This diagram represents how buyers can start or join group purchases. It tracks product, quantity, deadline, and purchase status, and includes functionality to find open groups or update group progress.

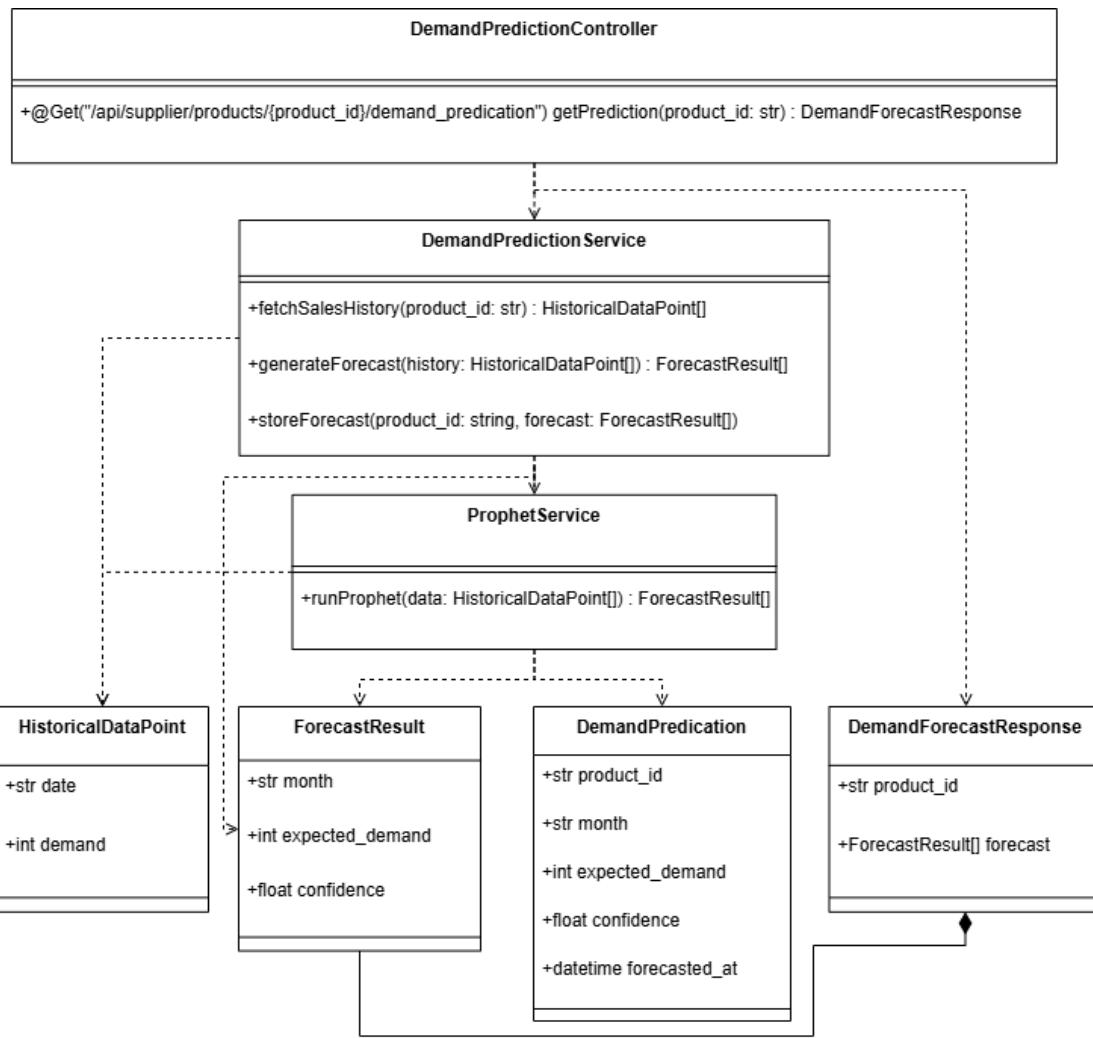


Figure 3-52: Demand Prediction Module Class Diagram

This diagram illustrates how demand forecasting is handled using historical sales data and Facebook Prophet. It shows the process of retrieving historical data, generating a forecast, storing results, and returning expected demand with confidence scores.

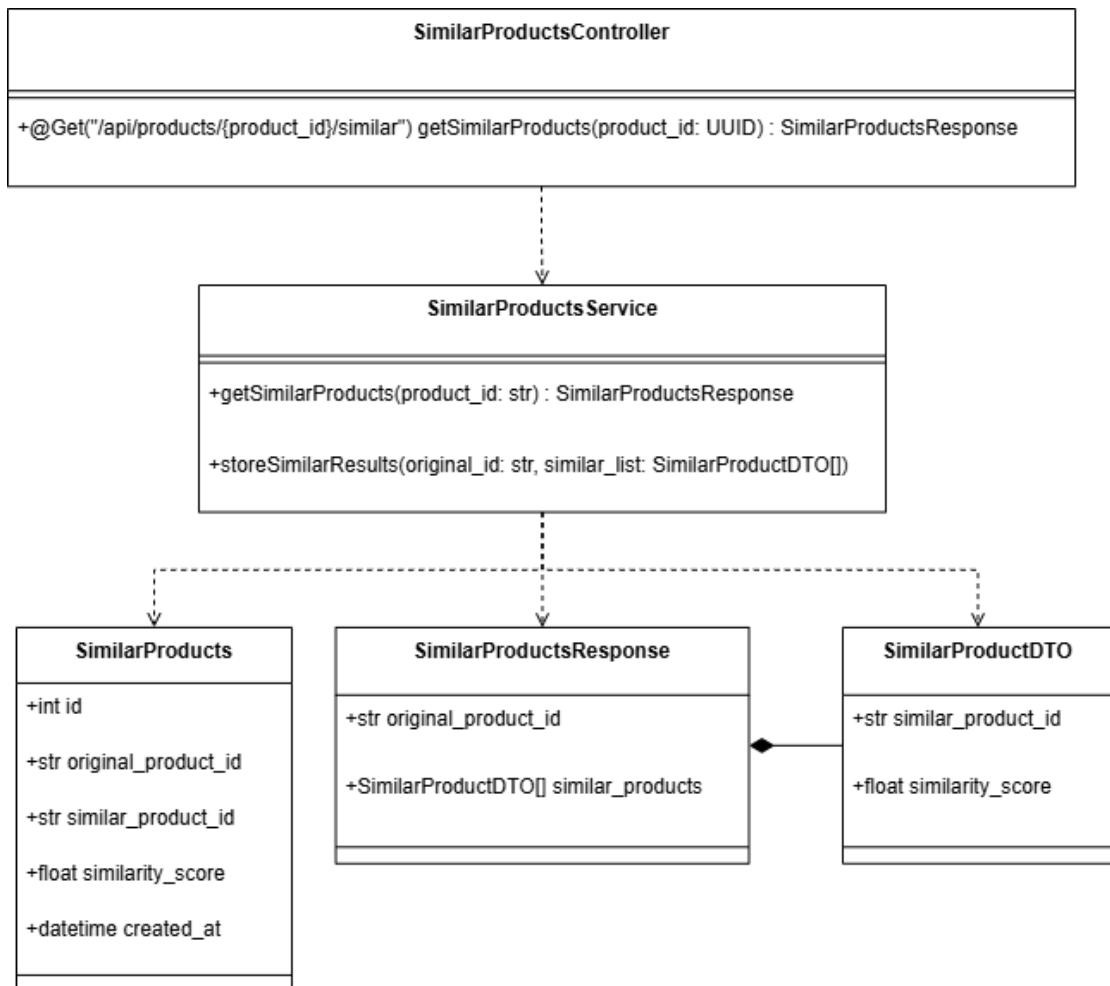


Figure 3-53: Similar Products Module Class Diagram

This diagram shows how the system identifies and returns products similar to a given product based on a similarity score. It includes classes for storing, retrieving, and formatting the results of similarity computations.

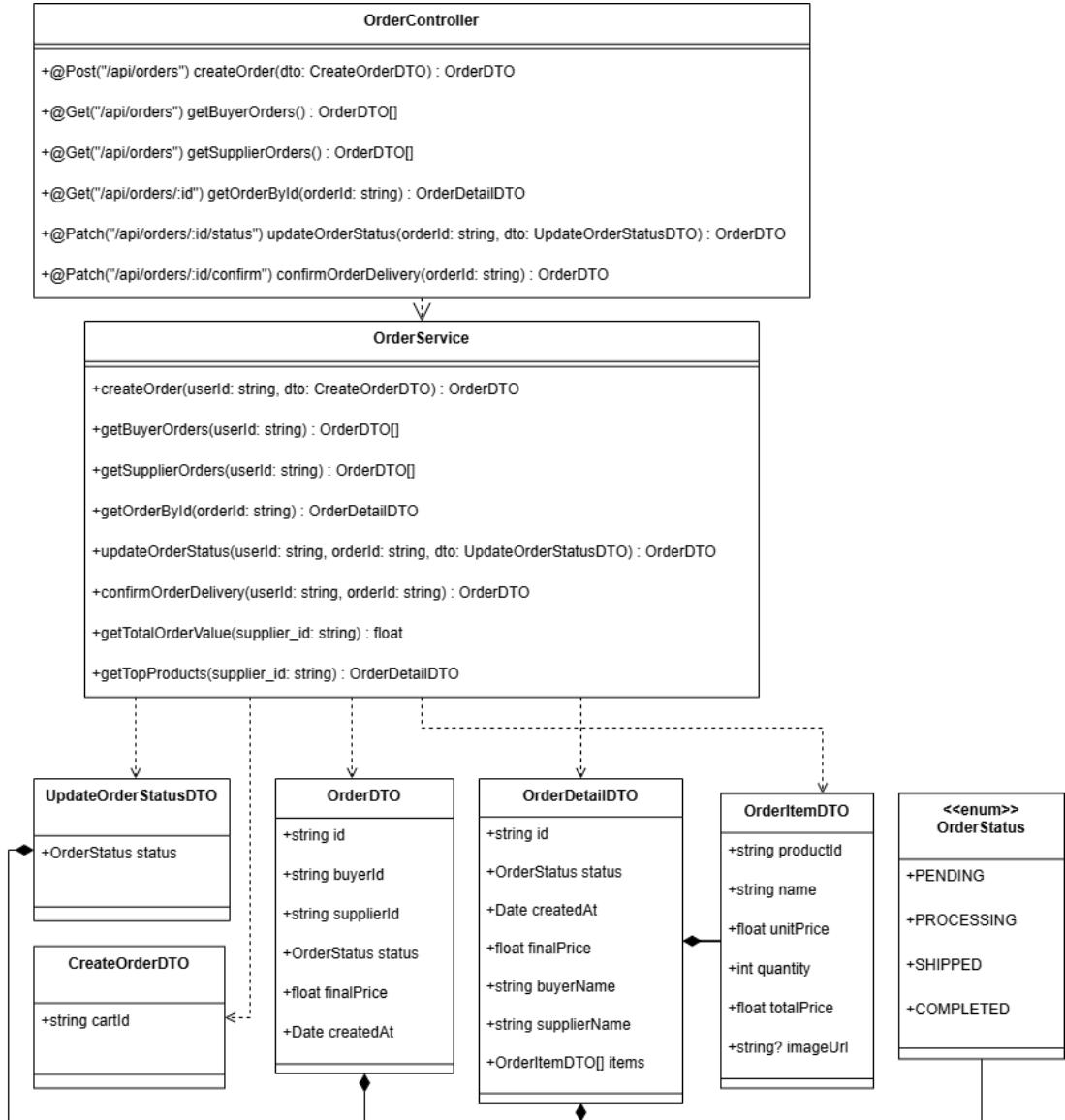


Figure 3-54: Order Module Class Diagram

This diagram describes the structure and responsibilities of the Order module, which handles order creation, status updates, supplier/buyer order views, and top product analytics.

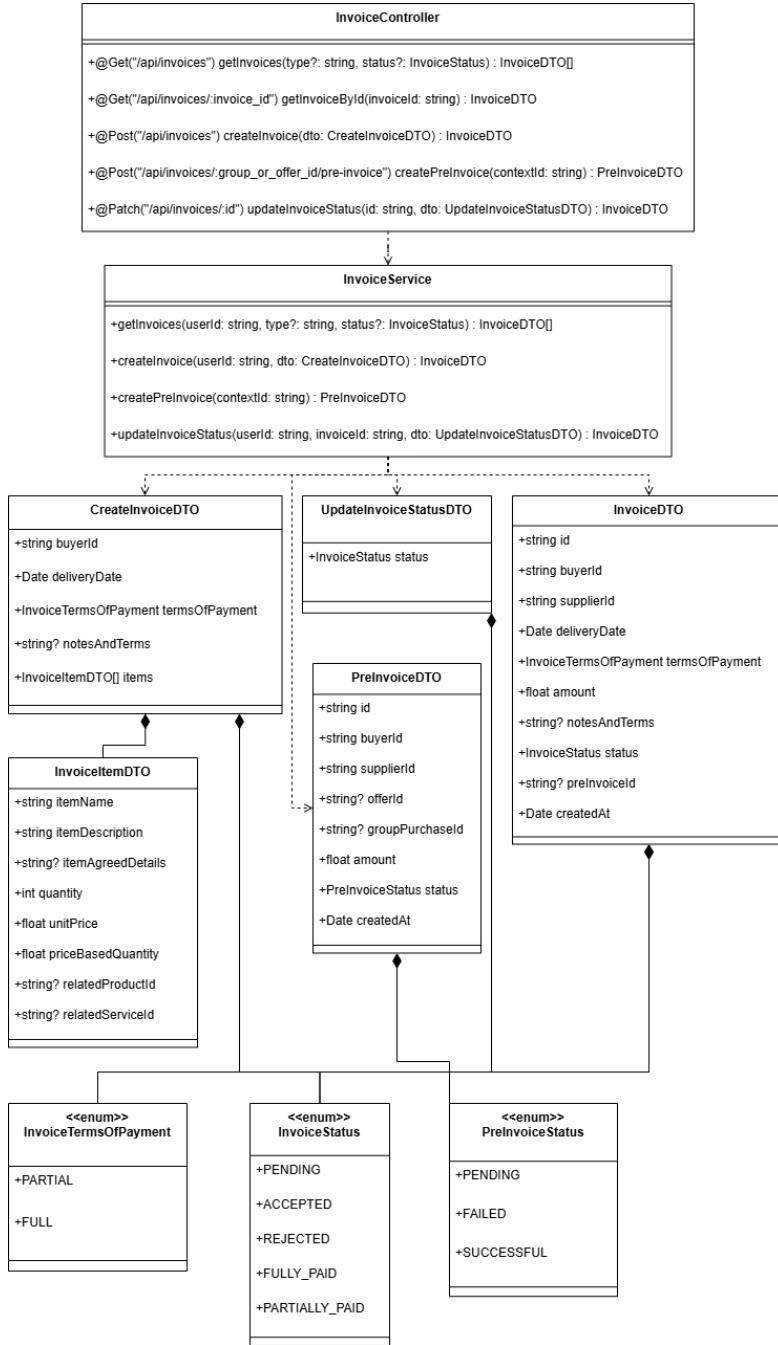


Figure 3-55: Invoice Module Class Diagram

This diagram illustrates the structure of the Invoice module, including how standard invoices and pre-invoices are created, updated, and linked with order or group purchase contexts.

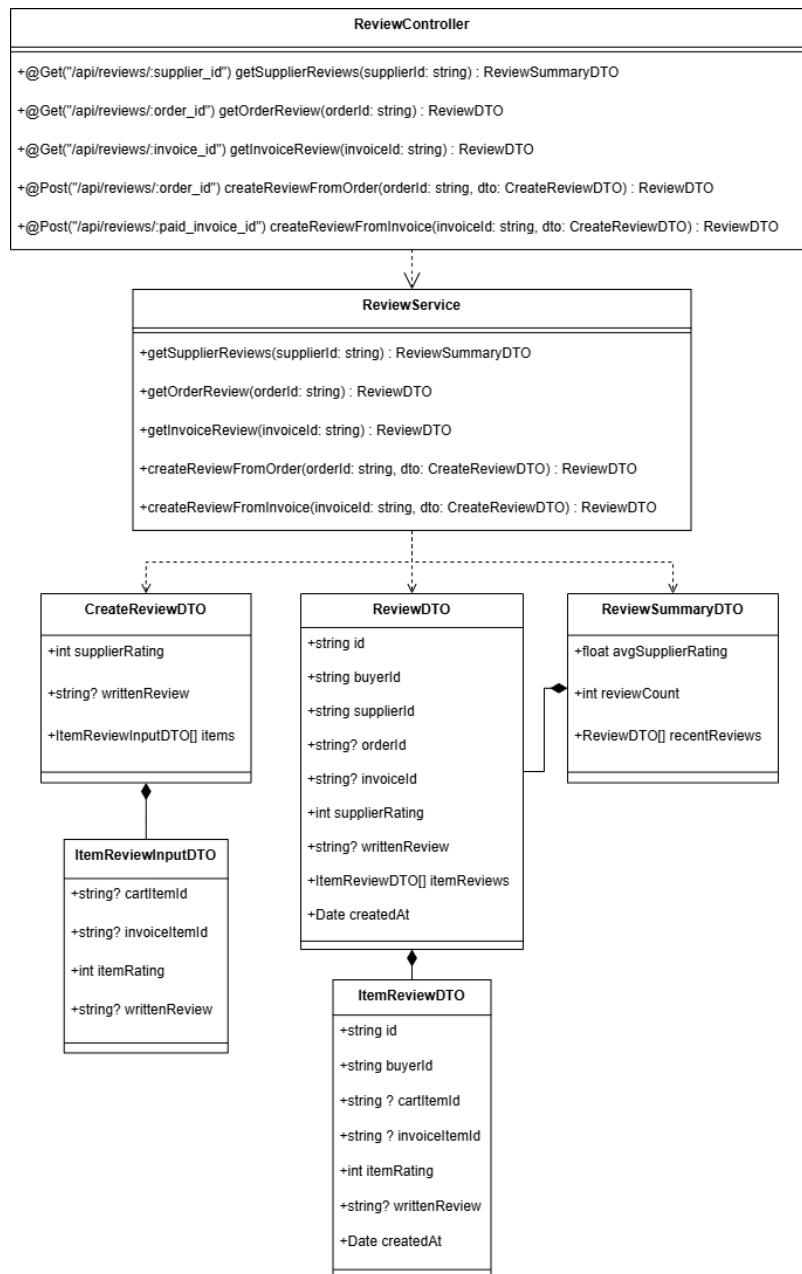


Figure 3-56: Review Module Class Diagram

This diagram illustrates the structure of the Review module, showing how buyers can review suppliers and individual items based on orders or invoices.

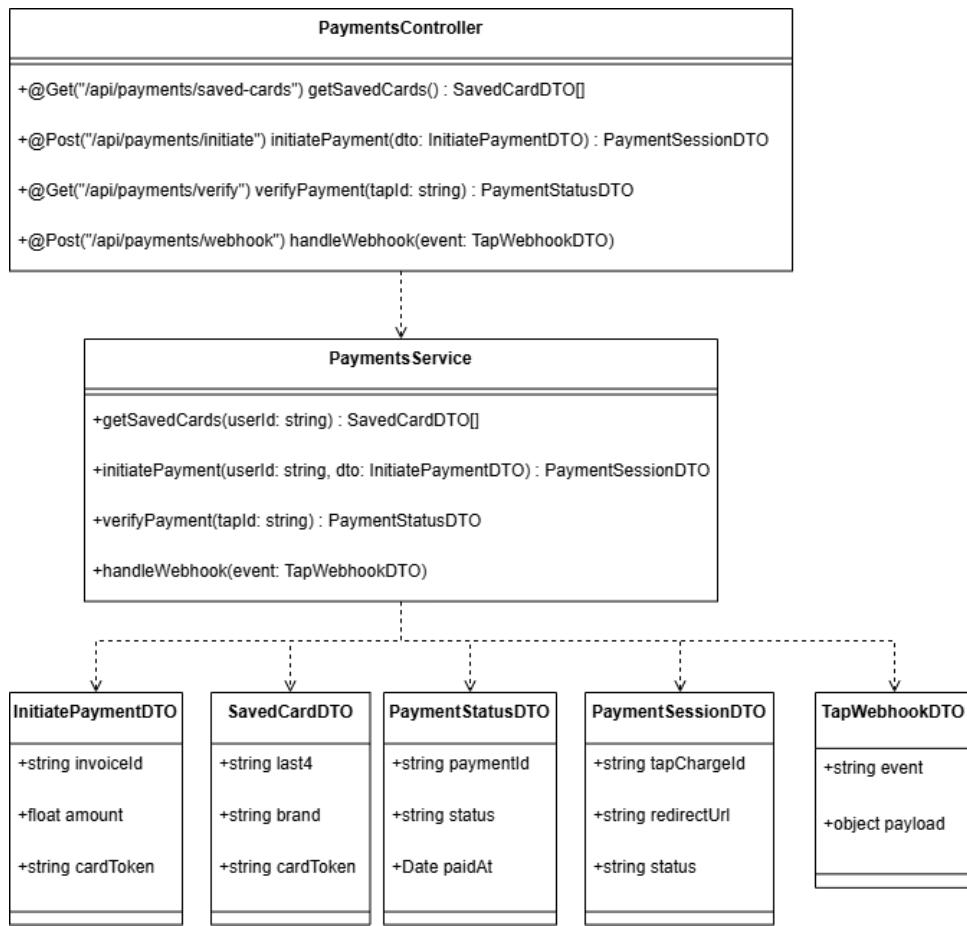


Figure 3-57: Payments Module Class Diagram

This diagram represents the Payments module structure, covering saved cards, payment initiation, status verification, and webhook handling with Tap.

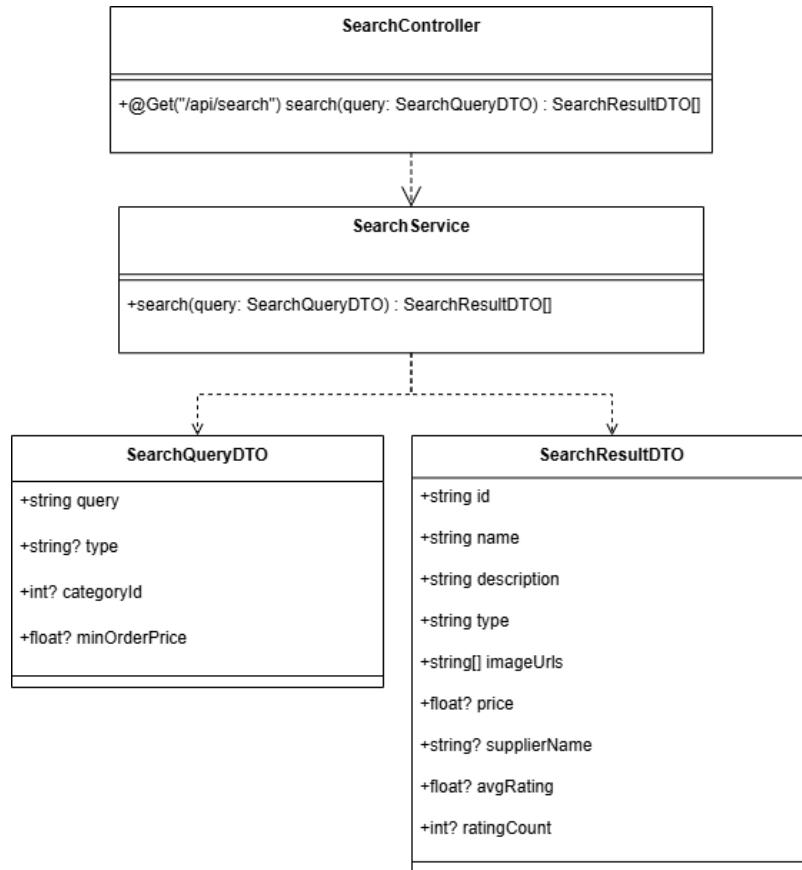


Figure 3-58: Search Module Class Diagram

This diagram illustrates how the system processes search queries and returns matching products or services through the **SearchController** and **SearchService**.

The class diagrams presented in this section provide a clear overview of the internal structure and behavior of each module in the system. They illustrate how data is transferred using DTOs, how services handle core business logic, and how controllers expose functionality to the client. This modular and well-organized structure facilitates maintainability and ease of collaboration across different features of the Silah platform.

Chapter 4 System Design

In this chapter, we will present the Silah System Design based on a 3-tier architecture. We will first discuss the business layer, which handles the backend logic, followed by the presentation layer, the user interface. This structure ensures a clear separation of concerns within the system.

4.1 System Architecture

In this section, we present the system architecture of Silah, highlighting the flow between its core components: the users, frontend, backend, database, and AI services. The architecture is designed to ensure modularity, seamless user experience, and efficient developer experience.

4.1.1 High-Level System Overview

The following diagram illustrates a high-level overview of the system's architecture. It presents the end-to-end flow, starting from the users interacting with the frontend, continuing through the backend where business logic, database access, and AI integrations occur.

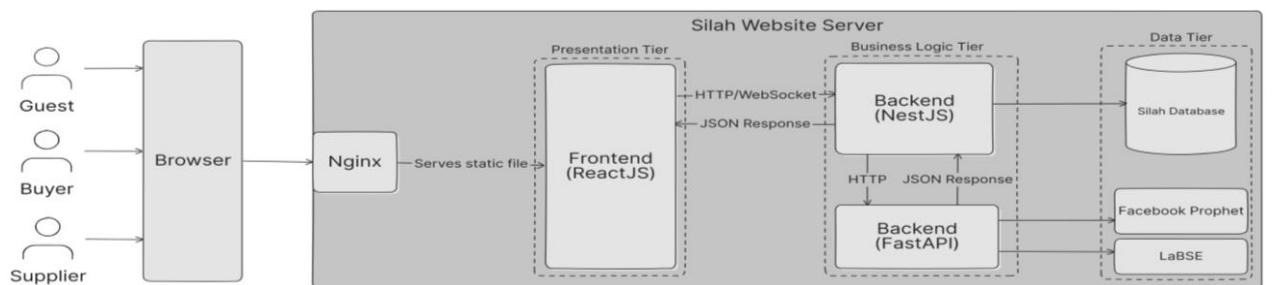


Figure 4-1: Full System Architecture

The system follows a Three-Tier architecture, which separates concerns into three distinct layers: the presentation layer (ReactJS frontend), the application layer (NestJS and FastAPI backends), and the data layer (database and AI models). This structure ensures a clear division between the user interface, business logic, and data management, promoting modularity and ease of maintenance.

In the architecture, Nginx is responsible for serving the static frontend files and routing all client traffic into the system. The frontend communicates exclusively with the NestJS backend using Axios for HTTP API requests and socket.io-client for

WebSocket communication. This setup enables both traditional interactions (e.g., browsing or submitting forms) and real-time experiences (e.g., direct messaging).

To further separate responsibilities, the FastAPI backend is a dedicated service that handles all AI-related operations. It interacts only with external AI models, while the NestJS backend remains the only backend that interfaces with the Silah database. NestJS and FastAPI communicate internally via HTTP, allowing the application layer to remain cleanly organized between business logic, AI processing, and data management.

4.1.2 Backend Modules

To implement a clean and maintainable structure, we adopted the Modular Monolith architecture pattern using the NestJS framework. This approach allows us to break down the system into multiple independent modules while still maintaining the simplicity of a monolithic deployment.

Each module encapsulates a specific domain or feature set of the system, containing its own controllers, services, entities, and other related logic. This modular approach enhances scalability, testability, and team collaboration, as each team member can work on a specific module without affecting others.

The following figure provides an overview of the system modules that form the core structure of our website:

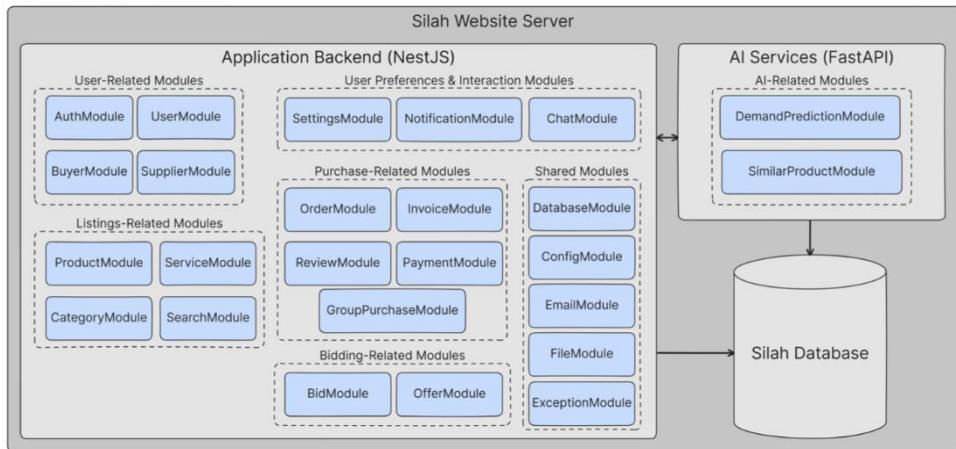


Figure 4-2: Modular Structure of the Backend System

Each module is responsible for a specific domain of the system, encapsulating related business logic, database access, and API endpoints. Below is a brief description of each module and its role in the system:

- **AuthModule:** Handles user authentication, login, registration, and JWT token issuance.
- **UserModule:** Manages user data and roles.
- **SupplierModule:** Contains all supplier-specific logic, including store details and subscription plan.
- **BuyerModule:** Manages buyer activities like adding products to cart and viewing the wishlist.
- **ProductModule:** Handles product creation, editing, listing, and categorization.
- **ServiceModule:** Similar to ProductModule but for non-physical offerings such as design services.
- **CategoryModule:** Manages product and service categories, including the creation of main and subcategories to help organize listings efficiently.
- **SearchModule:** Enables users to search for products, services, and suppliers using keywords and filters.
- **SettingsModule:** Manages user account settings, such as notifications preferences, stored bank cards, and password changes.
- **NotificationModule:** Sends in-app notification for events like new messages.
- **ChatModule:** Supports real-time messaging between buyers and suppliers.

- **OrderModule:** Manages order creation and status updates.
- **InvoiceModule:** Manages invoice and pre-invoice creation and status updates.
- **ReviewModule:** Allows buyers to rate and review suppliers and items after order completion or paying an invoice.
- **PaymentModule:** Handles payment-related logic and integration with Tap payment gateway.
- **GroupPurchaseModule:** Manages group purchase creation and tracks their progress until completion.
- **BidModule:** Manages bids and allows buyers to create bids.
- **OfferModule:** Allows suppliers to create offers in response to bids.
- **DemandPredictionModule:** Predicts stock demand based on historical data using a pre-trained model.
- **SimilarProductModule:** Recommends similar products to buyers based on cosine similarity after finding semantic similarities between products in both Arabic and English.
- **Shared Modules** (e.g., DatabaseModule, EmailModule, ConfigModule): Provide reusable utilities across the system like DB connection, email service, environment config, file upload, and error handling.

The backend modules are built using common components that ensure effective interaction between modules and maintain consistency across the platform. These components are designed to handle critical tasks such as processing HTTP requests, managing data flow, and securing access to resources.

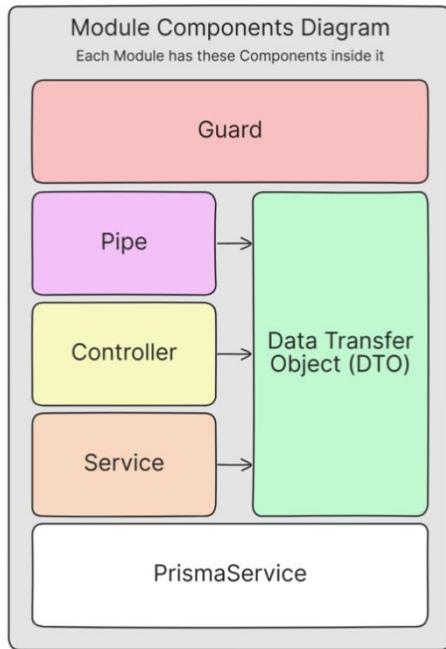


Figure 4-3: The Components that make up the Modules

Brief explanation of the components:

- **Guard**: Secures routes by verifying that the user has the necessary permissions to access specific resources or endpoints. For example, we validate that the user is a "Buyer" before allowing them to enter "www.silah.site/wishlist", ensuring only buyers can view and interact with their wish lists.
- **Pipe**: Validates and sanitizes input data before it reaches the controller, ensuring data integrity and correctness. For instance, when a user submits their email during registration, the EmailPipe would validate the format, making sure it follows the correct email pattern (e.g., "example@gmail.com") before processing it further. Pipes are primarily responsible for transforming data and enforcing validation rules.
- **DTO (Data Transfer Object)**: Defines the structure of data exchanged between the client and the API, providing a clear contract for data formatting. DTOs help ensure consistency in the format of data being transferred. For example, in our system, when a user submits their email, the `@IsEmail` decorator ensures the input is a valid email address (like user@gmail.com), preventing invalid or improperly formatted emails from being accepted. While

DTOs define the structure of data, Pipes are responsible for validating and transforming that data according to the rules specified.

- **Controller:** Handles HTTP requests from the client, invoking the appropriate services to process business logic and return responses. For instance, when a user visits "www.silah.site/", the HomeController checks if the user is logged in and then either redirects them to the homepage (for logged-in users) or to the landing page (for guests).
- **Service:** Contains core business logic, processes data, and communicates with the database to handle operations. For example, the OrderService handles order creation, payment processing, and updates to order status after successful payment.
- **PrismaService:** Provides an abstraction layer over Prisma ORM to manage database interactions, facilitating smooth and efficient communication with the database. It allows modules like OrderService or ProductService to perform operations like querying, inserting, updating, and deleting database records with ease.

These components work together across all modules to enforce structure and data consistency within the backend.

4.1.3 Database Schema & Datasets

A critical aspect of the system's design lies in how data is organized and managed. The database schema plays a pivotal role in ensuring that data is stored, retrieved, and processed efficiently across all system modules. It defines the structure and relationships of the data, helping to maintain consistency and optimize performance.

In conjunction with the database schema, the datasets provide the necessary data inputs to power key features such as demand forecasting and product recommendations, ensuring the system can make data-driven decisions to enhance user experience.

Database Schema

The database schema in our system is designed as a single unified schema to support modular monolithic architecture. Even though the system is divided into multiple

modules, the database schema is not separated per module. Instead, all modules share a common database, with entities and relationships mapped to represent the system's key features and data flows.

This approach ensures that all modules can interact with a single data source, enabling consistency and simplifying data management across the platform. By using one database, we avoid the complexities of multiple databases while retaining the modularity needed for scalability and maintainability.

The schema is structured to handle all system data and ensure smooth communication between modules. For example, entities like User, Order, Product, and Category are centrally managed, enabling all modules such as the OrderModule, ProductModule, and UserModule to access and update data in a consistent and coordinated manner.

While the following diagrams provide detailed views of individual tables including all columns and fields, we have also included a simplified diagram that highlights only the primary keys, foreign keys, and the core relationships between the major entities in the system. This high-level view helps clarify how key modules are linked together through the database without overwhelming the reader with internal column details.

To explore the complete and interactive version of the full database schema, including all table fields, constraints, and relationships, please visit the following link:

[https://dbdiagram.io/d/\[GP\]-Silah-Database-Schema-REAL-68c717ab841b2935a6855db8](https://dbdiagram.io/d/[GP]-Silah-Database-Schema-REAL-68c717ab841b2935a6855db8)

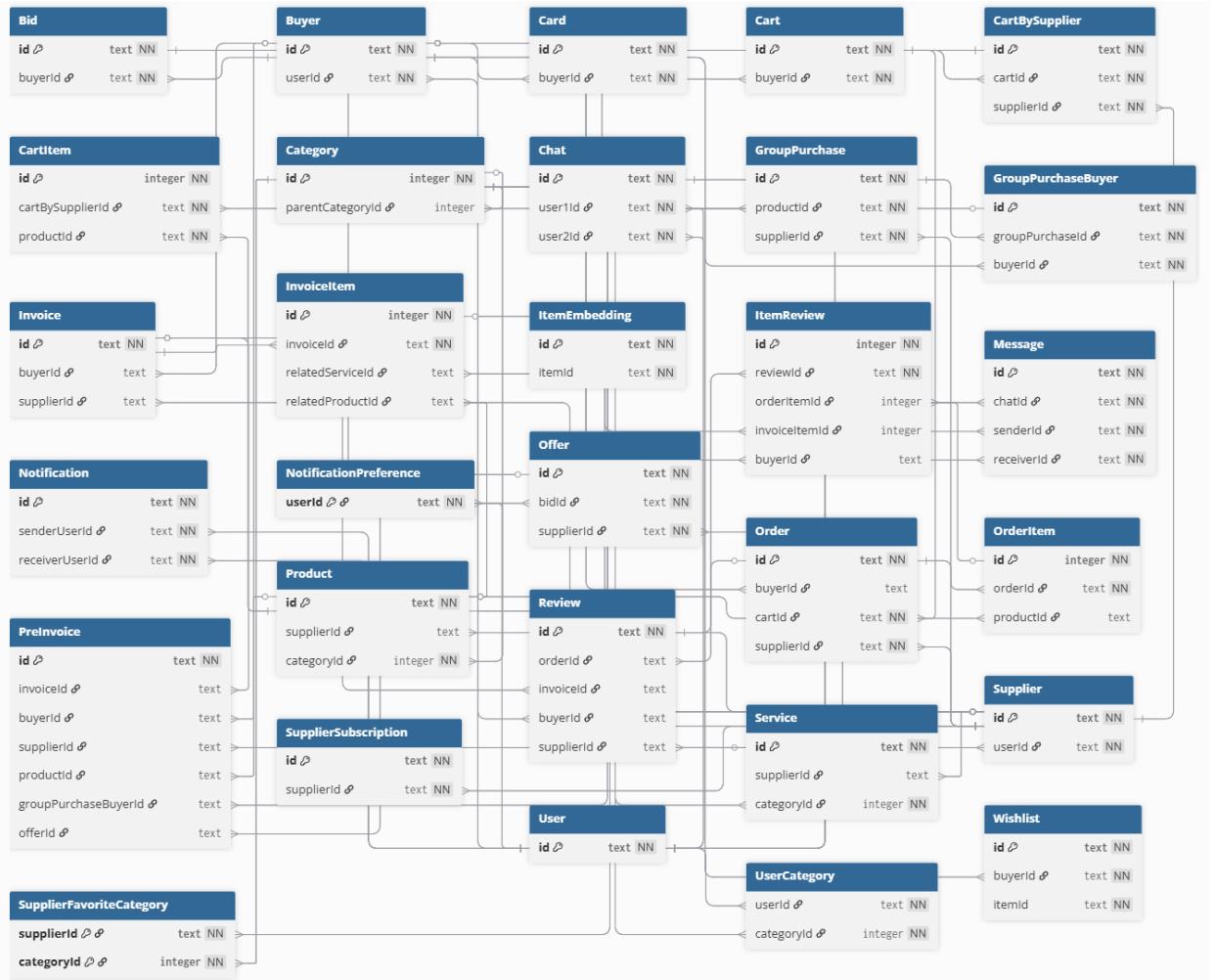


Figure 4-4: Key Tables and Relationships in Silah's Database

The following diagrams present detailed portions of the database schema. Each one includes full table columns, data types, and relationships to provide deeper insight into how data is structured and connected throughout the system.

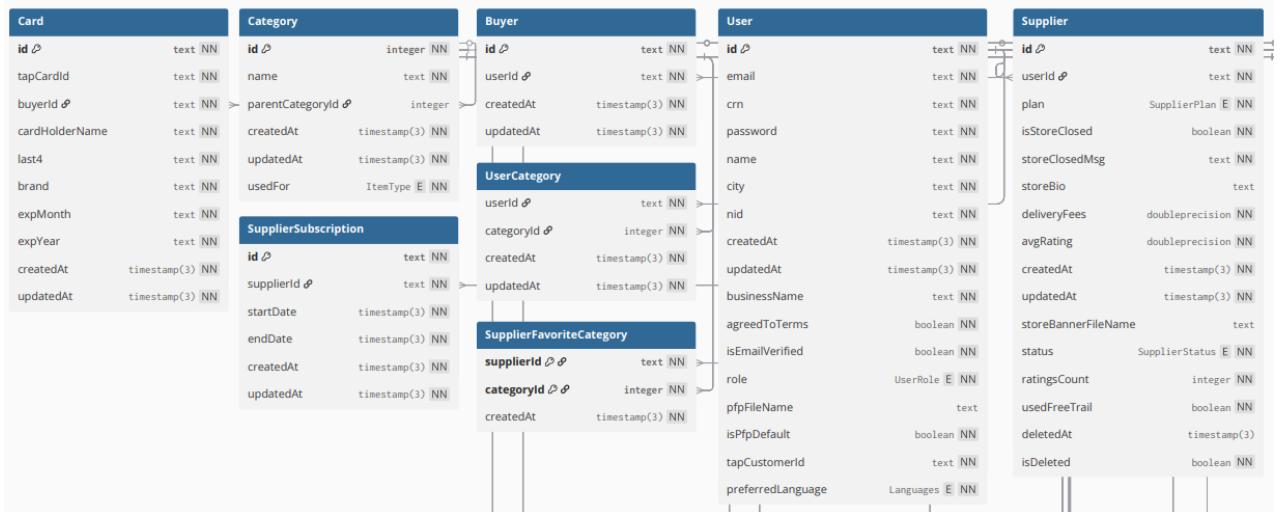


Figure 4-5: User, Supplier, Buyer, and Category Management Tables

The first diagram represents the core entities responsible for managing users, suppliers, buyers, categories, and subscription data. At the center of this structure is the User table, which stores essential personal and authentication information such as email, password, contact details, business data, and language settings. The Buyer and Supplier tables extend the User entity to support role-specific needs: buyers include their purchasing-related attributes, while suppliers maintain store-related information such as delivery fees, ratings, store banners, operational status, and subscription plan details. Categories are managed through a hierarchical Category table that allows flexible product and service classification. This categorization system is further enhanced through linking tables such as UserCategory, which tracks category preferences for different users, and SupplierFavoriteCategory, which identifies which categories suppliers frequently operate in or prioritize. Supplier subscriptions are maintained through the SupplierSubscription table, which records active and historical subscription periods. Payment cards saved by buyers are stored in the Card table, capturing card metadata such as masked numbers, brand, and expiration details. Altogether, this part of the schema forms the structural core of identity, role management, category organization, subscriptions, and user payment information across the system. The next set of tables handles key aspects such as product listings, bidding, and demand predictions, supporting essential platform operations like the wish list and similar products.

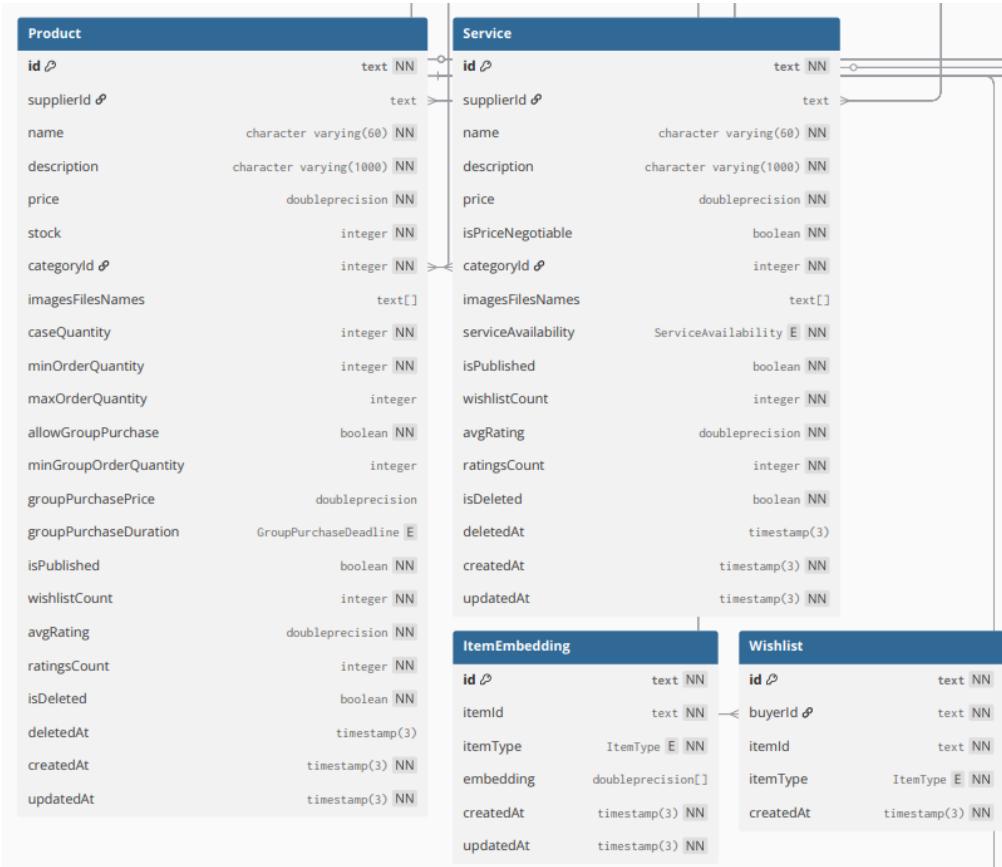


Figure 4-6: Product, Service, Wishlist, and Embedding Tables

The second diagram focuses on marketplace listings and the surrounding functionality that supports product management, service offerings, personalization, and buyer engagement. The Product table stores all product-related data including names, descriptions, prices, stock levels, images, category assignments, and various ordering and group-purchase rules. It also includes publication status and rating statistics that help manage product visibility and performance. The Service table mirrors this structure but adapts it for service-based offerings by including negotiable pricing options and availability details, while still supporting images, ratings, and publication status. The system's AI-driven recommendation feature is supported through the ItemEmbedding table, which stores vector embeddings for both products and services, allowing the platform to compute similarity, improve search relevance, and personalize item suggestions. Buyer engagement is captured through the Wishlist table, which records the items that buyers save for later, using a unified model that supports both products and services through a shared item ID and item type reference. This portion of the schema forms the operational backbone of product and service listings, recommendation functionality, and wishlist management.

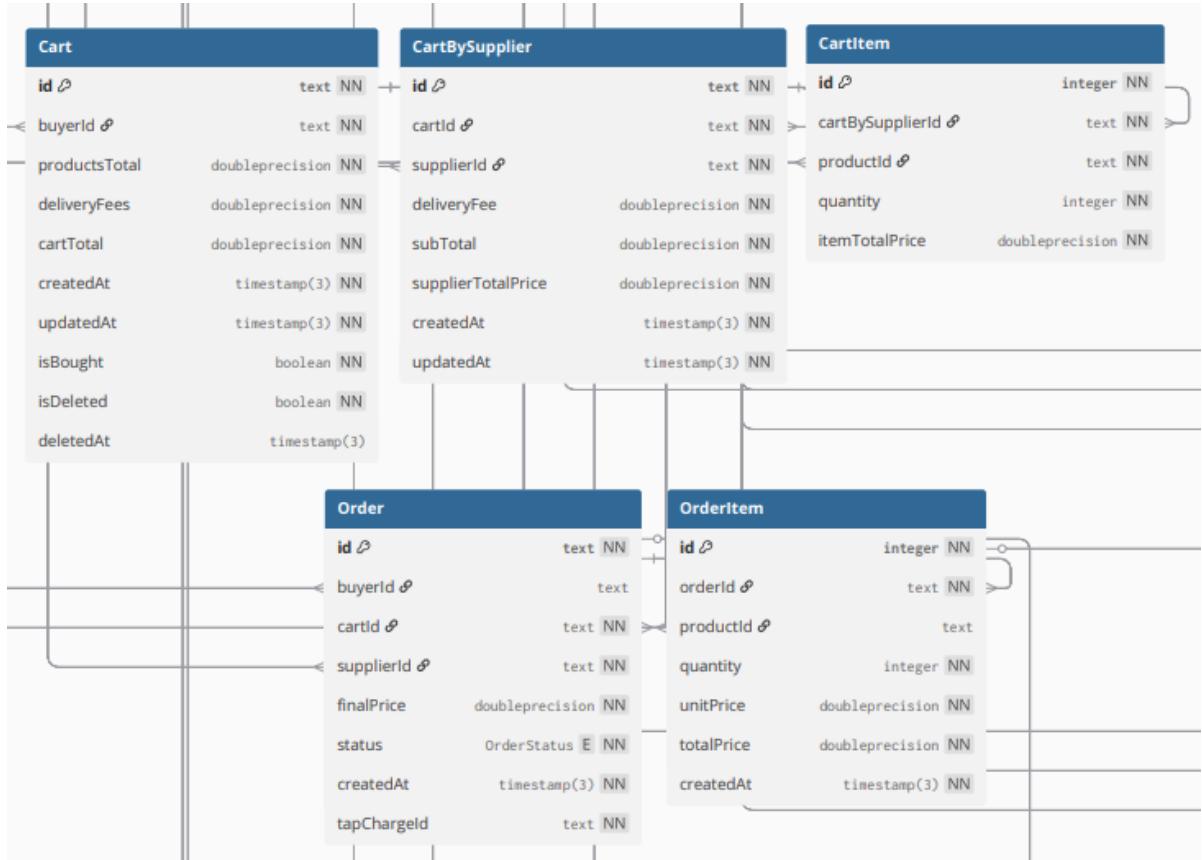


Figure 4-7: Cart, Order, and Order Fulfillment Tables

The third diagram presents the complete shopping cart and order management subsystem, encompassing both buyer-driven and supplier-managed purchasing flows in a multi-vendor marketplace. The **Cart** table functions as the central pre-checkout container, aggregating `productsTotal`, `deliveryFees`, and `cartTotal` while supporting soft deletion and conversion tracking via `isBought` and `isDeleted` flags. It connects to **CartItem** for standard buyer-added products and to **CartBySupplier** for supplier-specific sub-carts that carry their own `deliveryFee`, `subTotal`, and `supplierTotalPrice` calculations. Upon successful checkout, the **Order** table is created, linking back to the original cart and supplier, and capturing `finalPrice`, `status`, `orderStatus` enum, and Tap payment metadata. Individual purchased items are stored in **OrderItem** with `quantity`, `unitPrice`, and `totalPrice` snapshots taken at the moment of purchase. This cohesive structure provides full traceability from cart composition through multi-supplier checkout to finalized orders, forming the transactional core of the platform's purchasing lifecycle.

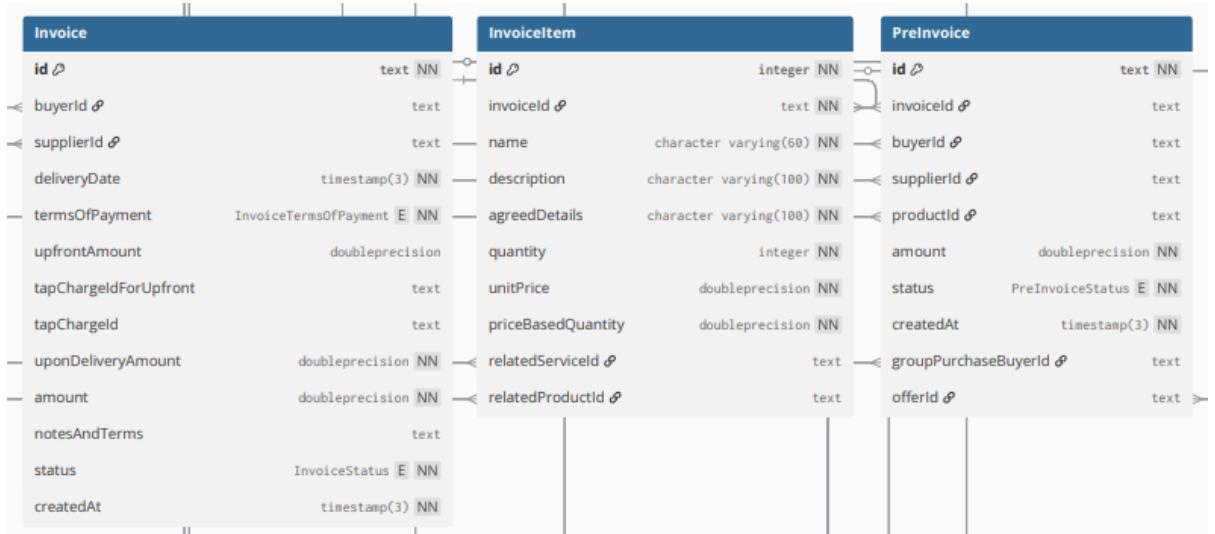


Figure 4-8: Invoice, PreInvoice, and Billing Tables

The fourth diagram details the comprehensive invoicing and financial settlement layer that supports flexible payment workflows, partial upfront charges, and formal billing.

The Invoice table records official invoices issued to buyers, containing deliveryDate, termsOfPayment, upfrontAmount, amount, tapChargeId details, notesAndTerms, and invoice status, while relating to the buyer and supplier. Detailed per-item breakdown is provided by InvoiceItem, which stores description, quantity, unitPrice, priceBasedQuantity, and an optional link to the original product or service via relatedProductId or relatedServiceId. The PreInvoice table serves as a provisional billing mechanism used primarily for group purchases, negotiated deals, or scenarios requiring upfront collection before final fulfillment, tracking provisional amount, status, and flexible references to either a group purchase or an offer. Together, these tables enable split payments, accurate revenue allocation, escrow-style upfront charges, and complete audit trails from order completion to final invoicing and settlement.

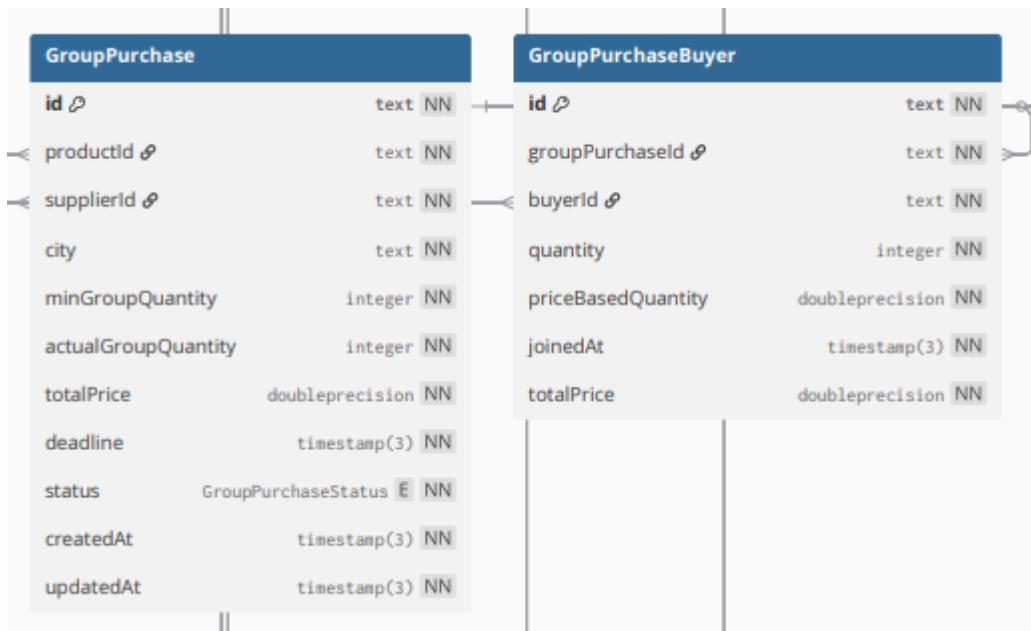


Figure 4-9: Group Purchase Tables

The fifth diagram covers the platform's group-buying functionality, enabling multiple buyers to collectively purchase a product at a lower price once a minimum participation threshold is met. The **GroupPurchase** table defines each group-buy event, linking to a specific product and supplier, specifying the city of fulfillment, the required **minGroupQuantity**, the evolving **actualGroupQuantity**, **totalPrice**, **deadline** for joining, and current status via the **GroupPurchaseStatus** enum. The **GroupPurchaseBuyer** table serves as the junction between individual buyers and active group purchases, recording each participant's quantity, any **priceBasedQuantity** adjustments, the exact **totalPrice** they committed to, and the timestamp when they joined. This structure supports dynamic group-buy campaigns with clear participation tracking, automatic success/failure determination upon deadline, and accurate per-buyer financial obligations.



Figure 4-10: Bid and Offer Tables

The sixth diagram details the RFP and bidding system that facilitates negotiated purchases, custom orders, and service contracts. The Bid table represents a buyer-initiated procurement request, containing the buyerId, a descriptive bidName and mainActivity, submissionDeadline, expectedResponseTime, and current BidStatus. Once a supplier responds, an Offer record is created that references the original bid, specifies the supplier, proposedAmount, expectedCompletionTime, detailed offerDetails and executionDetails, optional notes, and OfferStatus. This bidirectional flow enables transparent negotiation, price discovery for non-standard items or bulk orders, and full auditability of the entire RFP lifecycle from posting through acceptance or rejection.

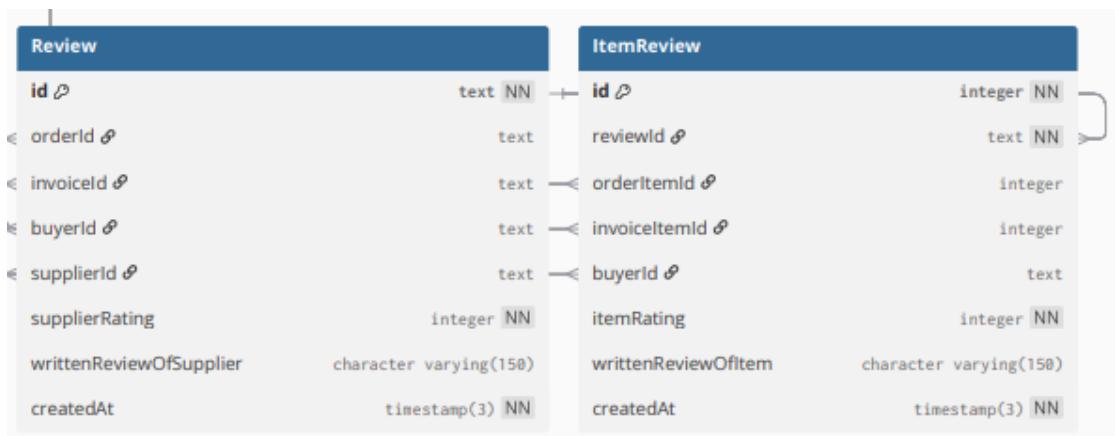


Figure 4-11: Review and ItemReview Tables

The seventh diagram captures the platform's post-purchase feedback and reputation system. The Review table records a buyer's overall experience with a specific order

and supplier, linking directly to orderId, invoiceId, buyerId, and supplierId, and storing a numeric supplierRating along with an optional writtenReviewOfSupplier (up to 150 characters). For greater detail at the product or service level, the ItemReview table provides individual item feedback, connecting to a specific reviewId, orderItemId or invoiceItemId, and containing its own itemRating and writtenReviewOfItem. This dual-layer structure enables both supplier reputation tracking and product/service quality insights, supporting accurate ratings aggregation and buyer decision-making across the marketplace.



Figure 4-12: Chat, Messaging, and Notification Tables

The eighth diagram encompasses real-time communication and user notification features essential for marketplace interactions. The Chat table represents a private conversation thread between exactly two participants (user1Id and user2Id), with

timestamps for creation and last update. Individual Message records belong to a chat and store senderId, receiverId, text content, optional imageFileName, read status, and precise readAt timestamp. User notification preferences are managed in the NotificationPreference table, allowing each user to enable or disable specific event types. Actual in-app notifications are stored in the Notification table, which includes sender/receiver, type, title, content, read status, entity references (entityId and entityType), and soft-deletion support. Together, these tables deliver reliable private messaging, fine-grained notification control, and a complete audit trail of all system-to-user and user-to-user communications.

Datasets

Prophet Model

General Information about the Dataset

The dataset used for the Prophet model was obtained from an open-source platform and is known as the Superstore Dataset (Final) [29]. It is a comprehensive dataset widely used in data analytics and forecasting projects, containing real-world retail transactions across various product categories, regions, and customer segments. It includes details such as order information, product data, customer details, and sales values, making it highly suitable for time-series forecasting and machine learning applications.

- Number of rows: 9,994 (each record represents one order transaction)
- Original number of columns: 21
- Why we chose it: It provides a large number of records per product, ensuring enough time-series data for reliable forecasting. The dataset is also clean, structured, and realistic — commonly used in business analytics and AI projects.

Description of Selected Columns

Although the dataset contains many fields, only the most relevant columns were selected for the Prophet model:

- Order Date: The date when each purchase occurred. It serves as the ds column in Prophet, representing the time component of the forecast.

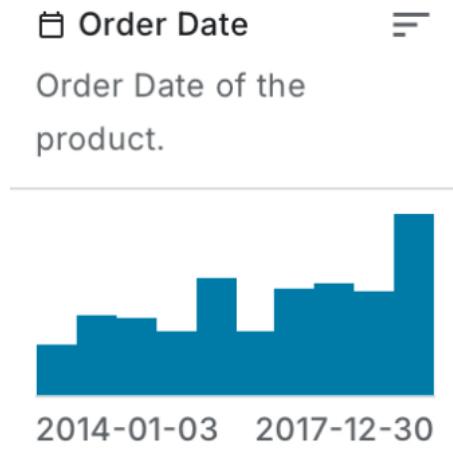


Figure 4-13: Overview of 'Order Date' Column

- Product ID: A unique identifier for each product. Used to group data per product and generate product-level forecasts.



Figure 4-14: Overview of 'Product ID' Column

- Sales: The total sales amount per transaction. This column is used as the y variable — the main target Prophet predicts.

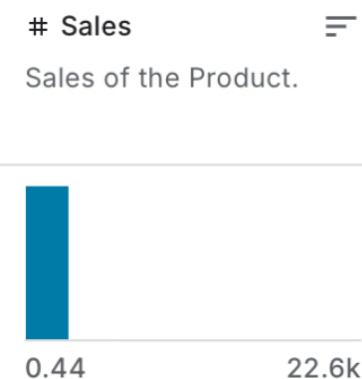


Figure 4-15: Overview of 'Sales' Column

Note: Non-essential columns such as customer names, postal codes, or descriptive text fields were excluded because they do not contribute to the forecasting process. Only the columns directly influencing the time-based prediction were retained to optimize model performance and reduce noise.

How the Dataset Will Be Used

The dataset was employed to train a Facebook Prophet model that predicts future sales trends for each product. Prophet is particularly effective in modeling seasonal and temporal patterns in sales data.

Steps to Apply the Model:

- Data Cleaning: Removed unnecessary fields and ensured the dataset contained no missing or invalid values.
- Formatting for Prophet: Prophet requires two columns:
 - ds: Date column → (Order Date)
 - y: Target column → (Sales)
- Modeling Strategy: Each product was processed individually to create product-specific forecasts based on its sales history.

Data Splitting Strategy:

To evaluate the accuracy of our model, the dataset was divided into training and testing subsets. An 80/20 split was implemented, with 80% of the data allocated for training the Prophet model and the remaining 20% reserved for testing. This configuration allows the model to learn underlying temporal patterns from historical observations and subsequently generate forecasts for unseen data, aligning with established practices in time-series analysis.

For instance, in the study titled “*Time-series Forecasting of Web Traffic Using Prophet Machine Learning Model*” [30], the authors employed the same 80/20 split when applying Prophet to comparable forecasting tasks. Their justification for this ratio highlights its balanced nature proving ample data for effective model training while preserving a sufficient portion of unseen samples to robustly evaluate generalization performance.

LaBSE Model

For our project, we used Amazon Sales Dataset from Kaggle [31]. It includes 1,465 products with useful information like product name, category, and a detailed description. The data is clean (all the main columns are 100% complete), and it covers a wide range of product types, which makes it a good fit for our use case.

We're using this dataset to fine-tune the LaBSE model. The goal is to help the system understand the meaning of product descriptions and find similar or alternative products based on that. We combine the product name and description to create text input for the model. To train and test the model, we split the dataset into 80% training and 20% testing — this split is based on what was used in a research paper by Salunkhe & Patil [32], where they fine-tuned LaBSE on text classification tasks. The 80/20 split gives a good balance and is commonly used in NLP projects.

Dataset Columns:

- product_id

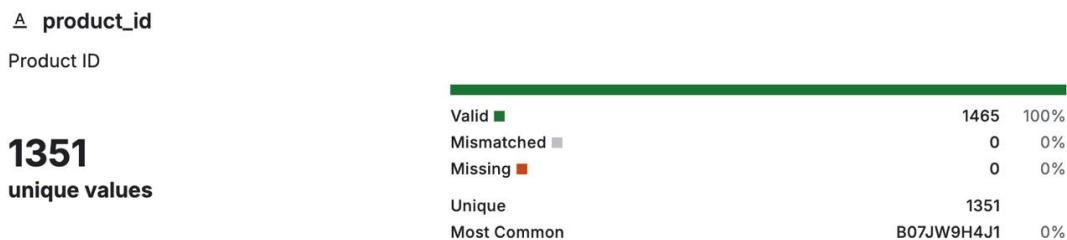


Figure 4-16: Overview of 'product_id' Column

This is just a unique number or code for each product. It helps us keep things organized but we don't use it directly in training the model.

- product_name



Figure 4-17: Overview of 'product_name' Column

In this dataset, the Product_name field is more than just a short title — it often contains detailed keywords and product attributes, making it quite similar to a summarized description. Therefore, we treat it as an important part of the input during training, alongside the full product description

- category

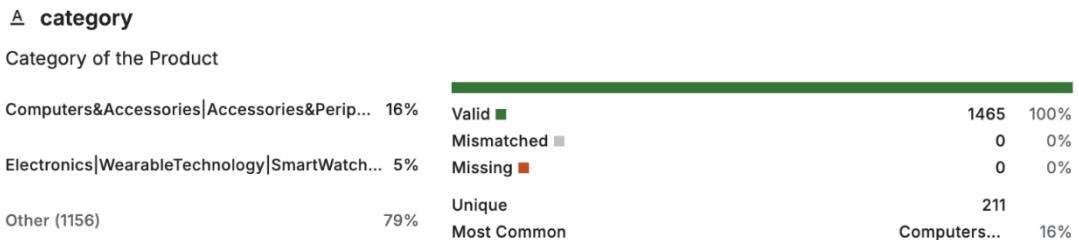


Figure 4-18: Overview of 'category' Column

The product's category on Amazon (like electronics, accessories, etc.). This helps us match similar products when we're building training examples.

- about_product



Figure 4-19: Overview of 'about_product' Column

This is the full description of the product — it has all the details. It's the most important part for our model because that's the text we want it to understand and compare.

4.2 User Interface Design

For the design of the user interface (UI), we utilized Figma to create detailed wireframes and high-fidelity prototypes. The design process was organized by segmenting the pages into distinct categories based on user roles: Guests, Buyers, and Suppliers. This allowed us to tailor the UI to the specific needs of each user group while maintaining a consistent and intuitive design across the platform.

SiteMaps

The sitemaps below provide a general overview of how the platform is structured, offering a visual representation of the website's navigation. These diagrams help readers understand the flow of the site and how users interact with different sections. It includes key pages and navigation elements available to each user type, such as Guests, Buyers, and Suppliers. Please note that some diagrams may be clearer by viewing them on Figma through this link:

<https://www.figma.com/board/b9MJ3Oqk5wJIDgEYmSsXDv/FigJam-Basics?node-id=1249-3390&t=EzPOR1SsTw4qu7kR-1>

The first diagram shows the footer layout, which is consistent across all user types and ensures easy access to essential sections of the platform from any page.

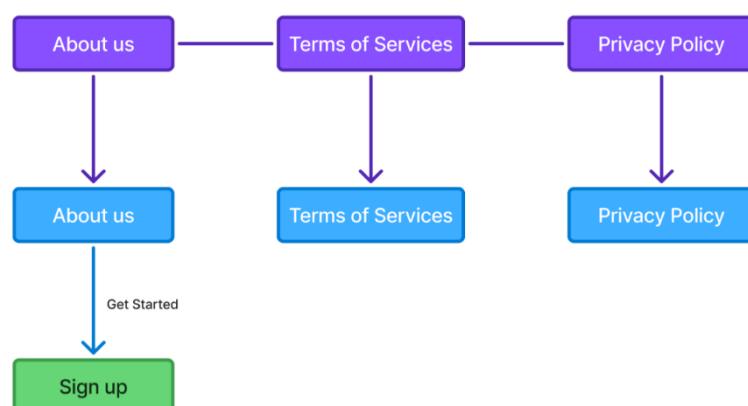


Figure 4-20: Footer Navigation Structure

The next diagram presents the Guest Header, which is displayed for users who have not yet logged in or registered. The guest header includes essential navigation elements such as the Login and Sign-up buttons, along with options to explore the categories and access core features of the platform. This header guides new users towards registration while introducing them to the basic offerings of the platform.



Figure 4-21: Guest Header Navigation

Following the Guest Header, the Buyer Header appears for users who are logged in as buyers. This header provides easy access to buyer-specific pages such as shopping cart, invoices, order management, and notifications. The Buyer Header streamlines the navigation experience, allowing buyers to efficiently manage their shopping experience, orders, and accounts.

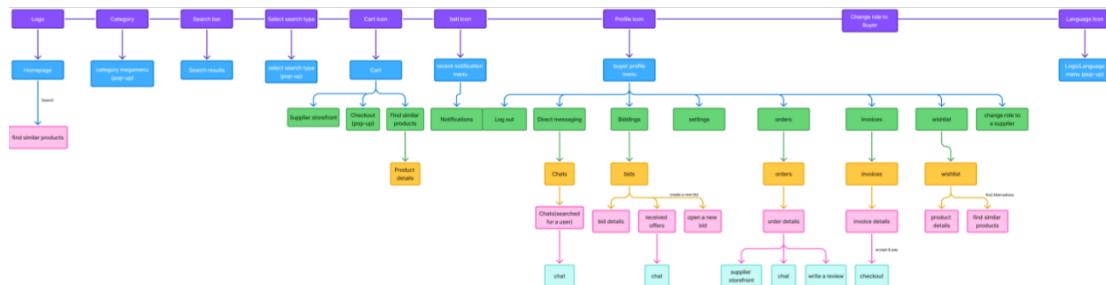


Figure 4-22: Buyer Header Navigation

Following the Buyer Header, the Supplier Sidebar appears for users who are logged in as suppliers. This sidebar includes important pages like Orders, Products & Services, Bidding, Invoices, Analytics & Insights, and Direct Messages. These elements give suppliers quick access to essential features, including order management, invoicing, bidding participation, and communication with buyers. The Supplier Sidebar ensures that suppliers can efficiently manage their store at any time.



Figure 4-23: Supplier Sidebar Navigation

The next diagram illustrates the journey of a user who first signs up via the Landing Page, leading them to the Buyer Homepage. From there, the user can switch to the Supplier role using the "Switch Role" button, which takes them to the Supplier Overview Page. This diagram shows the navigation flow for the supplier role, outlining the key pages and features available as the user transitions between the buyer and supplier roles, ensuring a smooth and intuitive experience on the platform.

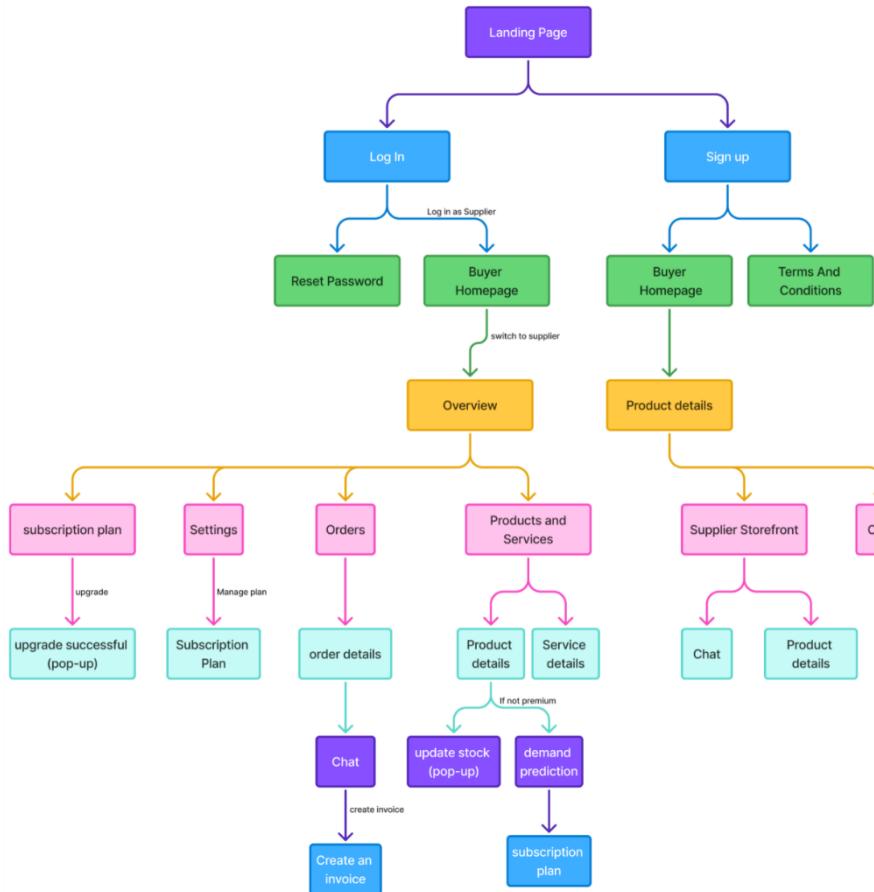


Figure 4-24: User Journey from Buyer to Supplier Role and Navigation Flow

4.2.1 Guest User Interface

The Guest user interface is designed for users who are not logged in. It offers simple and welcoming experience with easy access to key features such as browsing products and services, exploring categories, and viewing informational pages like About Us.

Guest UI encourages users to sign-up or login to access more features, such as making purchases or interacting with suppliers.

The Landing Page

The landing page serves as the public-facing entry point to the Silah platform, designed to introduce the platform's value proposition and encourage new users to register. The header contains a search bar, category selector, language switcher, and two main calls to action: "Login" and "Sign Up", allowing guest users to easily access or create an account.

The page emphasizes simplicity and trust by showcasing the platform's benefits through a clean design and clear messaging, such as "Find Trusted Suppliers & Grow Your Business." Additionally, a visual "How It Works" section walks users through the platform's basic steps: signing up, switching roles, exploring suppliers, and closing deals. The page also features an interactive category navigation system, allowing guests to browse both products and services before registration.

The footer provides essential legal and contact information, including links to Terms of Service, Privacy Policy, and a contact email. This layout ensures that visitors can quickly understand the platform's offerings and are encouraged to join with minimal friction.



Figure 4-25: The Landing Page

The Sing-up Process

The sign-up process is split into three simple steps. First, users enter their business information, including the business name, commercial registration number, and business activity. Next, they provide personal details such as name, national ID, and city. Finally, they set up their account credentials by entering an email and password, then confirming the terms and conditions. The process is straightforward and tailored to ensure all necessary business data.

The screenshot shows the first step of a three-step sign-up process. At the top right, there are two icons: a gear and a globe. Below them is the heading "Join us!". On the left, there are three input fields: "Business Name" (The Electric Hour Co.), "Commercial Register" (1010123456), and "Business Activity" (a dropdown menu with "Select a Category"). To the right of these fields is a vertical navigation bar with three items: "Business Info" (highlighted with a purple circle and number 1), "User Info" (with a grey circle and number 2), and "Account Info" (with a grey circle and number 3). At the bottom center is a purple "Next" button. Below it, a small note says "Already have an account? [Log in](#)". At the very bottom, there is a footer with links: "About Us | Terms of Service | Privacy Policy | Contact Us: infogsilah.site" and "© 2025 Silah. All Rights Reserved."

Figure 4-26: Sing-up Page (Step 1)

This screenshot shows the second step of the sign-up process. The layout is identical to Figure 4-26, with the "User Info" section highlighted in the vertical navigation bar. The "Name" field contains "Mohammed", the "National Id" field contains "1234567890", and the "City" field contains "Riyadh". The "Back" and "Next" buttons are at the bottom, and the footer links are at the bottom.

Figure 4-27: Sing-up Page (Step 2)

This screenshot shows the third step of the sign-up process. The "Account Info" section is highlighted in the vertical navigation bar. It includes fields for "Email" (Example@gmail.com), "Password" (12345678), "Confirm Password" (12345678), and a checkbox for "I agree on the terms and conditions". The "Back" and "Done" buttons are at the bottom, and the footer links are at the bottom.

Figure 4-28: Sing-up Page (Step 3)

The Login Page

The login screen allows registered users to access their accounts by entering their email or commercial registration number along with their password. Users can also reset their password if needed or navigate to the sign-up page if they don't have an account yet. The process is designed to be simple and quick, ensuring easy access for both buyers and suppliers.

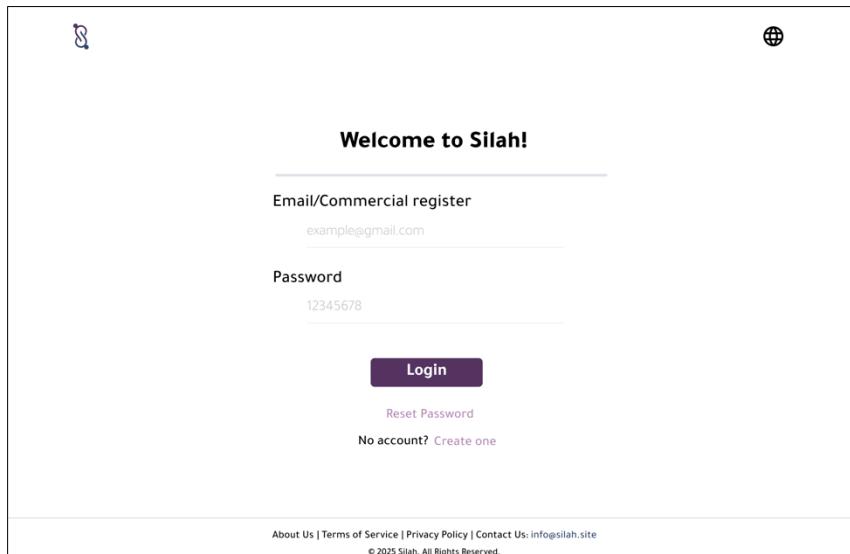


Figure 4-29: The Login Page

4.2.2 Supplier User Interface

The Supplier user interface is tailored to meet the needs of suppliers who manage and sell products or services on the platform. It provides comprehensive pages where suppliers can easily manage their store details, update product listings, participate in bids, and track orders and invoices statuses.

The Supplier UI is designed to be efficient and intuitive, enabling suppliers to quickly upload new products or services, and manage their inventory. Additionally, suppliers can communicate directly with buyers through the messaging system and review their sales performance. The design aims to streamline their experience, ensuring they have all the necessary tools at their fingertips to run their business effectively.

The Overview Page

After logging in as a Supplier, suppliers are greeted with a personalized dashboard that provides a quick summary of their business status. It displays the store status, the number of new orders, and current stock levels with quick action buttons. The page also shows the supplier's subscription plan and offers an upgrade option to unlock premium features like AI insights. The navigation bar gives access to all key modules, including listings, biddings, invoices, analytics, and settings. A button is also available to switch roles to Buyer as needed.

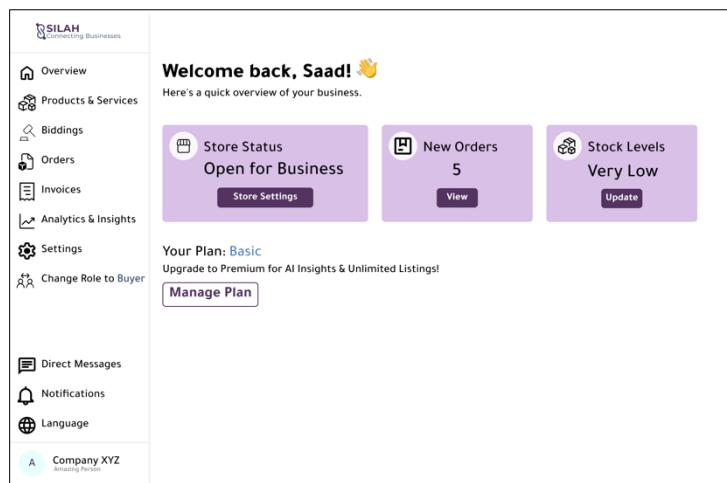


Figure 4-30: The Overview Page

Manage Listings Process

The listings page shows a table of all the supplier's products and services. Each row displays the item name, price, stock level, and publish status. Suppliers can search, filter by item type (products or services), and take actions like edit, publish, unpublish, duplicate, or delete. There's also a "Predict Demand" button for eligible product listings, and quick-access buttons to add new products or services.

Select item(s) to take an action						
	Image	Item Name	Unit Price	Stock	Status	
<input type="checkbox"/>		Amber - 45ml Soy Candle	42	10	26	Published Predict Demand
<input type="checkbox"/>		Amber - 250ml Soy Candle	42	10	26	Published Predict Demand
<input type="checkbox"/>		Amber - 45ml Soy Candle	42	10	0	Unpublished Predict Demand
<input type="checkbox"/>		Logo Design	23	50	-	Published

Figure 4-31: Products & Services Page

On the product editing page, suppliers can update all the essential details of a product. This includes the name, description, images, price, stock, and order requirements like case quantity and minimum order quantity. Group purchasing settings are also configurable, allowing suppliers to enable discounts when buyers order in bulk. The product must be assigned a category for buyers to find it through search and filters.

Basic Information

Product Details
Build buyers confidence with a clear, detailed product listing.

Name
Amber - 250ml Soy Candle 24/60

Description
Experience the warm and inviting glow of the Amber 250ml Soy Candle. Infused with rich amber notes, soft vanilla undertones, and a hint of earthy musk, this candle delivers a sophisticated and soothing scent.

Crafted with 100% natural soy wax, this hand-poured candle offers a clean burn and a long-lasting fragrance. Perfect for relaxation, aromatherapy, and pheromones, it is a safe and eco-friendly choice for any home. The minimalist design adds a touch of elegance to any room. A perfect decorative piece for bedrooms, living rooms, or workspaces.

Product Category
This information will help us categorize your product.

Category
 Select a Category ▼
This Product will appear to buyers in:
Home & Living > Furniture

Images

Image Tips

- Use at least 1050px by 1050px.
- Use a white or natural background.
- Ensure bright lighting with minimal shadows.
- Include front, side, and close-up views.

Price

Price Per Unit
10

Order Requirements

Case Quantity
4 Case quantity refers to the number of units that come in a single case or package. Buyers are required to purchase products in this case quantity and cannot be ordered in increments of the case quantity. For example, if the case quantity is 10, buyers must order in increments of 10 units (e.g., 10, 20, 30 units, etc.).

Minimum Order Quantity
8 The minimum order quantity is the smallest number of units that a buyer must order from you. This ensures that you can sell products in bulk, optimizing your production, inventory, and shipping processes.

Maximum Order Quantity
Unlimited The maximum order quantity is the largest number of units or items a buyer can order from you at one time. This limit is set to ensure that products are distributed fairly, prevent market saturation, manage inventory effectively, and avoid stock shortages.

Group Purchasing Settings

Would you like to enable group purchasing?
 Yes No Group purchasing allows five buyers from the same city to collectively place a bulk order for a product, enabling them to access discounted prices. This strategy helps you increase sales volume, attract more buyers, and move inventory more efficiently while offering competitive pricing.

Minimum Group Orders Quantity
8 The minimum group orders quantity is the smallest number of units or packages that must be collectively ordered by all buyers in a group purchase. This ensures that you can meet your production and inventory requirements efficiently while offering a discount to buyers.

Order Deadline
After 3 days The order deadline is the specified time by which the minimum number of buyers (which is 5) and the minimum group orders quantity must be met for the group purchase to be completed. If the requirements are not fulfilled by the deadline, the group purchase will be canceled.

Group Purchase Price Per Unit
7.5

Save

Figure 4-32: The Product Details Page

The next page lets suppliers manage service listings. They can provide a detailed description, upload images, set the base price, and mark the price as negotiable if needed. Availability options allow suppliers to indicate when the service can be provided (e.g., 24/7). Like product listings, services must be categorized to ensure visibility to buyers.

The screenshot shows the service details page for a logo design service. The left sidebar includes links for Overview, Products & Services, Orders, Invoices, Analytics & Insights, Settings, Direct Messages, Notifications, and Language, along with a Company XYZ banner. The main content area has tabs for Basic Information, Images, Price, and Service Details. Under Basic Information, there are sections for Service Details (Name: Logo Design, Description: A detailed service listing), Service Category (Category dropdown), and a large text area for the service's mission statement. Under Images, there are tips for image quality and two upload buttons. Under Price, there is a price input field set to 50 and a checkbox for price negotiability. Under Service Details, there is a section for Service Availability (Available 24/7) and a summary with a 'Save' button.

Figure 4-33: The Service Details Page

View Stock Demand

The Predict Demand page shows AI-generated forecasts for a selected product over the next three months. It displays a line chart comparing predicted demand and current stock levels to help suppliers plan ahead. A summary below the chart suggests the recommended restocking quantity based on forecasted demand. This feature helps suppliers avoid shortages and make smarter inventory decisions.

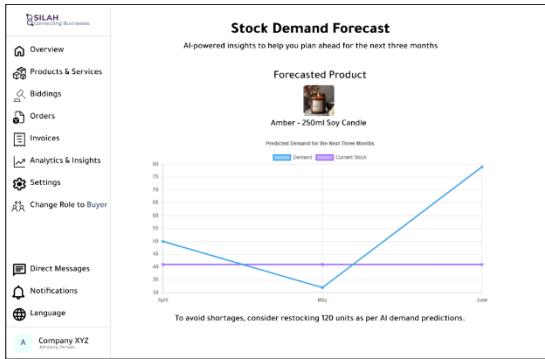


Figure 4-34: Stock Prediction Page

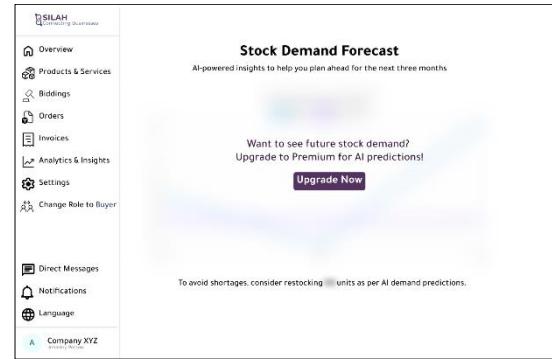


Figure 4-35: Stock Prediction Page (Unauthorized Supplier)

Participating in Biddings

The Biddings page shows a list of available opportunities for suppliers to participate in open bids. Each card displays the bid title, main activity category, reference number, and submission deadline. Suppliers can easily browse recent bids and click “View Details” to learn more or submit an offer. A checkbox at the top also allows users to filter only the bids they have already joined.

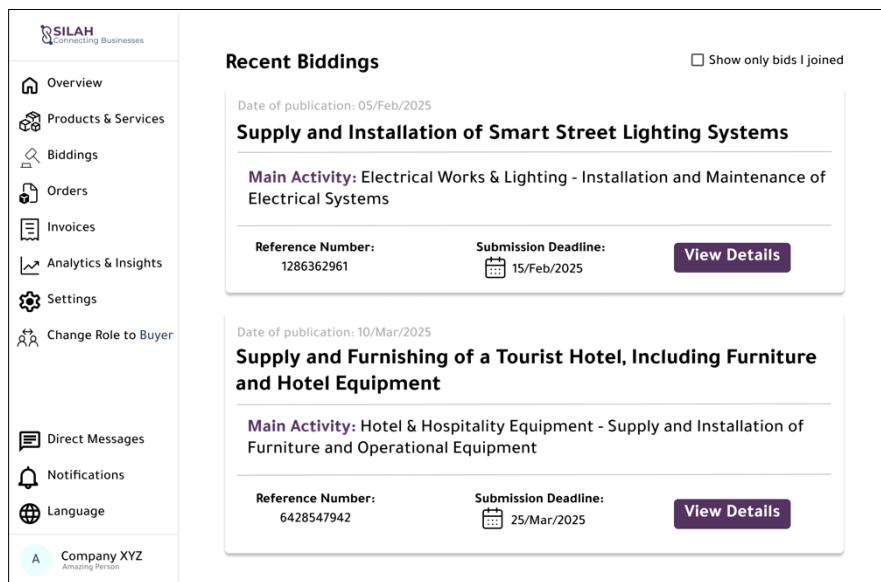


Figure 4-36: Biddings Page

The next page provides all the essential information about a specific bid. It includes the bid name, activity category, organization name, reference number, publication date, submission deadline, and time remaining. Suppliers can also see the expected response time and decide whether to participate by clicking the “Participate” button.

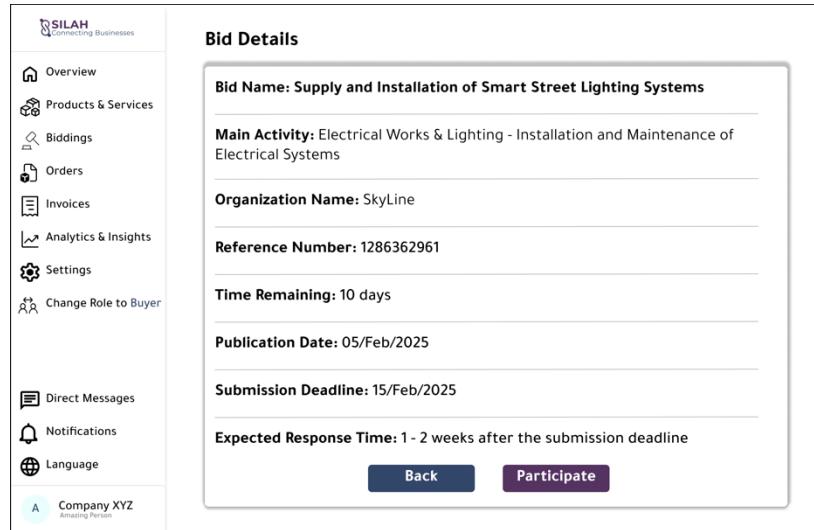


Figure 4-37: The Bid Details Page

After choosing to participate, the supplier can submit an offer by filling out key details. These include the proposed amount, expected completion time, technical offer description, and project execution duration. There's also space to add additional notes before submitting. Once ready, the supplier clicks "Submit Offer" to officially enter the bid.

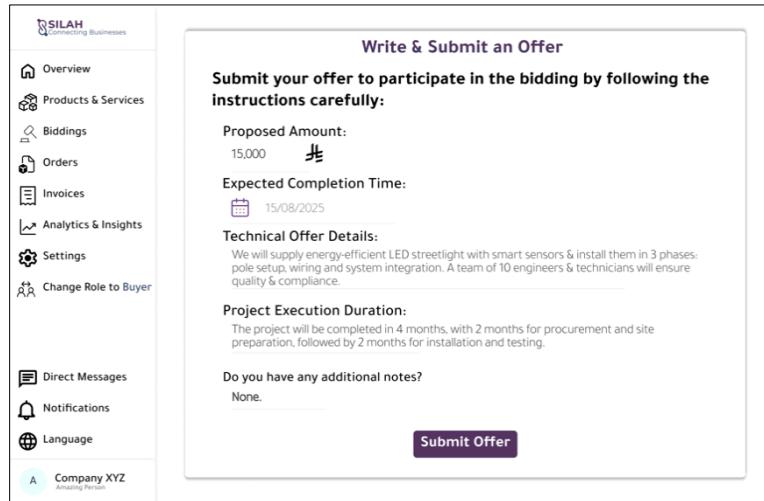


Figure 4-38: Write an Offer Page

Supplier Settings

The Settings page allows suppliers to manage their account, business, notifications, and store preferences. Suppliers can update their personal and business information, change their password, and manage their subscription plan. The Notifications section lets users customize alerts for messages, orders, and invoices. The Store section

allows suppliers to set temporary closures, display custom messages, manage delivery fees, and upload storefront images. A support section is also available for assistance, with an option to contact customer service directly.

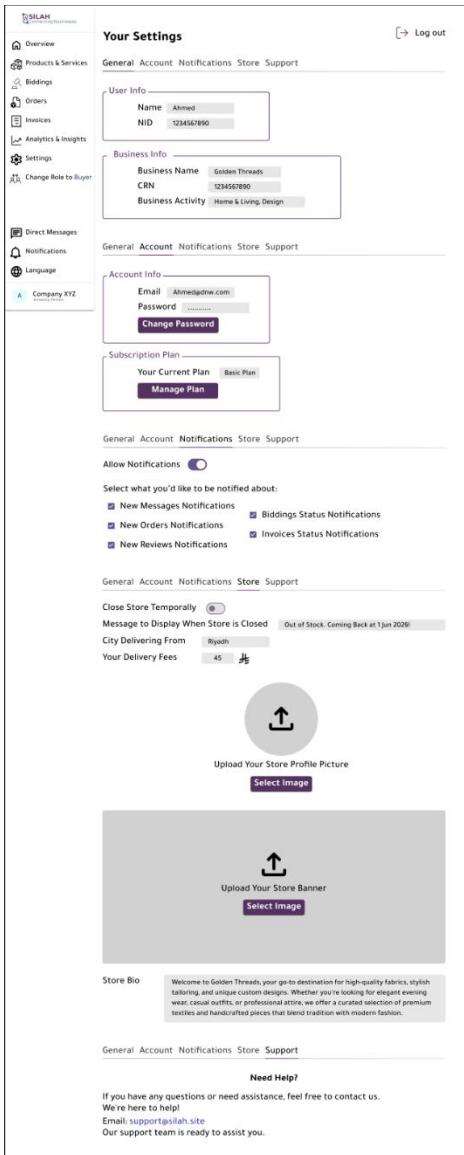


Figure 4-39: Supplier Settings Page

Subscription Plan Management

The Pricing Plans page offers suppliers two options: Basic and Premium. The Basic plan is free and provides essential features like managing a storefront, listing up to 10 products, and responding to buyer messages. The Premium plan costs 50 SAR/month and includes all the features of the Basic plan, along with additional perks such as

unlimited products and services, access to product wish-list counts, and AI-powered demand forecasting. Suppliers can start a 30-day trial of the Premium plan to explore the additional features before committing to the subscription.

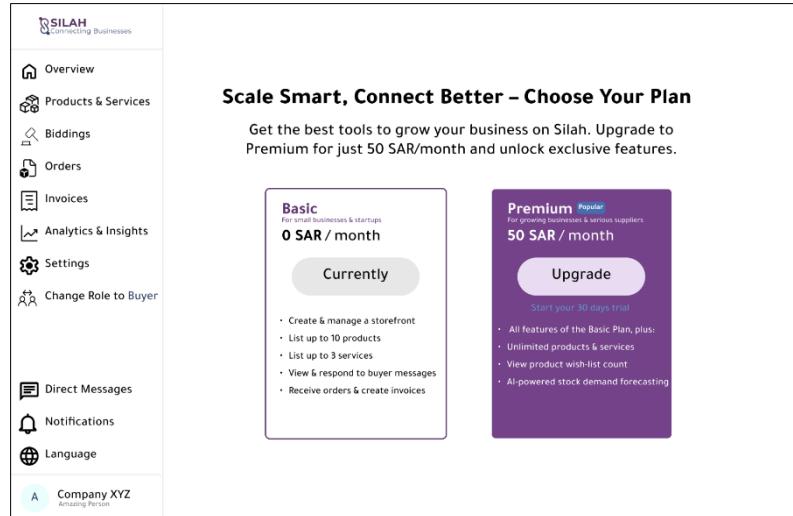


Figure 4-40: Subscriptions Page

Supplier Notifications

The Notifications page shows important updates for suppliers, including messages, reviews, new orders, and offer acceptances. Notifications are listed with the sender's name, event type, and date. Suppliers can filter by type or date to easily manage updates.

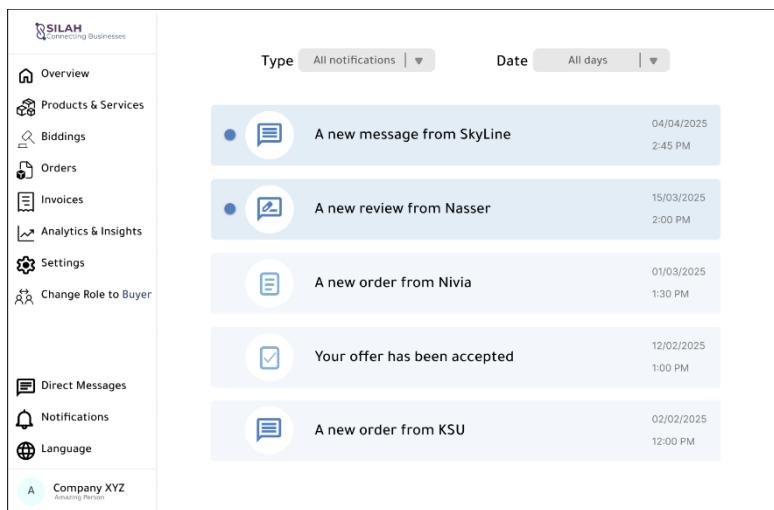


Figure 4-41: Supplier Notifications Page

Order Management

The Orders page provides an overview of all orders with their status, including Pending, Processing, Shipped, and Completed. Suppliers can quickly see details such as the buyer's name, order date, and total price. Clicking on a specific order takes you to the Order Details page, where more information about the items is displayed, including images, unit price, quantity, and total price for each item in the order. This seamless navigation helps suppliers track and manage their orders effectively.

Figure 4-42: Orders Page

Figure 4-43: The Order Details Page

Direct Messaging

The Direct Messages page allows suppliers to communicate with buyers, addressing inquiries and discussing product details. Once a conversation is concluded, the supplier can easily create an invoice based on the discussion, ensuring a smooth transition from messaging to transaction. After chatting with the buyer, the supplier can click the "Create an Invoice" button to generate an invoice for the agreed products or services. This seamless integration between messaging and invoicing helps suppliers manage their business efficiently.

Figure 4-44: Chats Page

Figure 4-45: The Chat Page

Invoice Management

The "Create an Invoice" page allows suppliers to generate invoices for orders and optionally link items to existing product or service listings. By selecting a listing, the system ensures that the item is associated with the appropriate listing, which enables the review to be automatically rendered later for that product or service. Suppliers can specify payment terms, including upfront and upon delivery amounts, and select items from their published catalog, making the invoice creation process seamless and efficient.

Figure 4-46: Create an Invoice Page

Figure 4-47: Link a Listing Page

The Invoices page displays a list of all invoices with filters for different statuses: Accepted, Rejected, Partially Paid, and Fully Paid. Each invoice entry shows essential details, such as the buyer, invoice status, and total amount. Clicking on an invoice leads to the Invoice Details page, where the supplier can view specific details, including terms of payment, item descriptions, agreed details, and the payment breakdown. This comprehensive view allows suppliers to easily manage and track their invoices.

Your Invoices

All	Accepted	Rejected	Partially Paid	Fully Paid
#556903778	22 Mar	StyLine Logistics	Accepted	₼ 10,000
#556903778	19 Mar	King Saud University	Accepted	₼ 30,000
#490903778	18 Mar	Almuneer	Rejected	₼ 59,580
#446372778	17 Mar	King Khalid University	Accepted	₼ 30,000
#556903778	22 Mar	Skyline Logistics	Fully Paid	₼ 10,000
#556903778	19 Mar	King Saud University	Accepted	₼ 30,000
#490903778	18 Mar	Almuneer	Rejected	₼ 59,580
#446372778	17 Mar	King Khalid University	Partially Paid	₼ 30,000
#556903778	19 Mar	King Saud University	Fully Paid	₼ 30,000
#490903778	18 Mar	Almuneer	Fully Paid	₼ 59,580
#446372778	17 Mar	King Khalid University	Rejected	₼ 30,000

Invoice Details

Invoice No: 12345678	Terms of Payment: Partially Paid
Issue Date: 23/Mar/2025	Upfront Payment Amount: 4000 ₩
Delivery Date: 06/Apr/2025	Upon Delivery Amount: 8000 ₩

Sender

Golden Threads
Employee: Amazing Person
City: Riyadh
Email: info@goldenthreads.com

Receiver

Nivia Co.
Employee: Ahmed Ahmed
City: Makkah
Email: deal@nivia.com

Item

Item	Item Description	Agreed Details	Qty.	Unit Price	Total Price
Custom Fabric Merchandise	High-quality custom fabric merchandise	Delivery within 10 business days from the date of the order confirmation	1	₼ 800	₼ 800

Final Price

10000 ₩

Notes & Terms

The customer will receive 10% discount on all purchases made within the next 30 days. Delivery is free for orders above 5000 ₩. Returns are accepted within 7 days of delivery if the product is not as described.

Figure 4-48: Invoices Page

Figure 4-49: The Invoice Details Page

Store Analytics & Insights

The Store Analytics & Insights page provides a detailed overview of the supplier's store performance over the past three months. It includes key metrics like total sales value and total orders, along with a bar chart showing sales trends for each month. The page also highlights insights into the most ordered and most wish-listed products. Suppliers can track their store's ratings, including the overall rating and the number of new reviews. This page helps suppliers evaluate their store's success and understand customer feedback to improve their offerings.

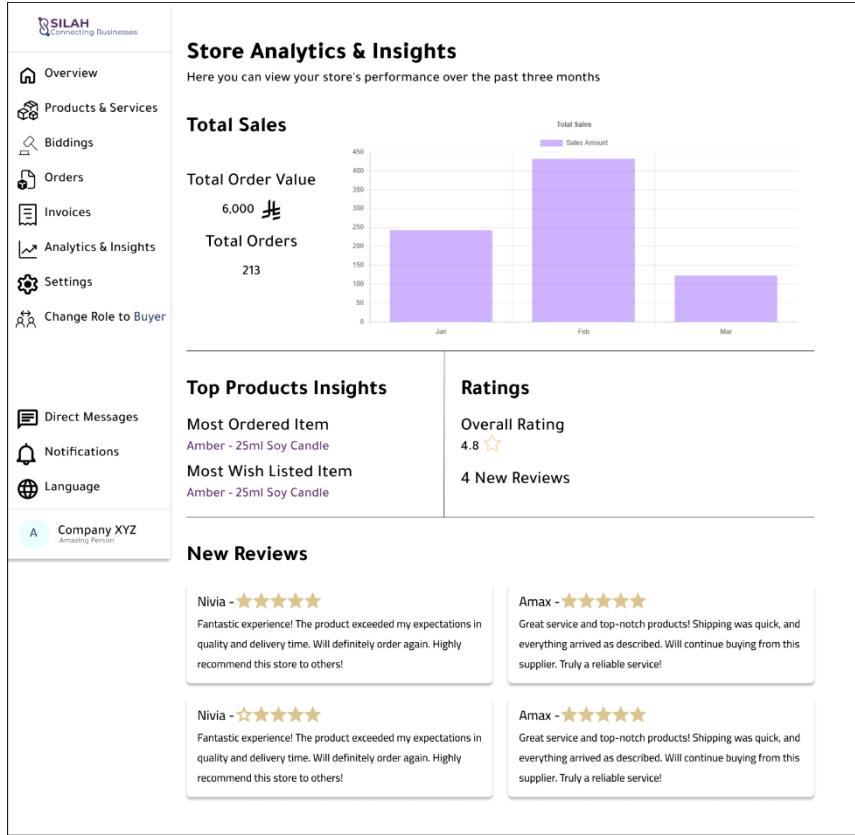


Figure 4-50: Store Analytics & Insights Page

4.2.3 Buyer User Interface

The Buyer user interface is designed to provide a seamless shopping experience, allowing buyers to browse, purchase, and manage their orders with ease. Buyers can explore products and services through an intuitive search and filtering system, add items to their cart or wish list, and place orders. The UI also offers easy access to order history, invoice details, and payment management. Buyers can leave reviews and ratings for products/services and suppliers, enabling them to share their experiences with others.

The Buyer UI is focused on delivering a smooth and engaging user experience, providing buyers with all the necessary information and tools to make informed purchasing decisions and manage their transactions.

The Homepage

The buyer homepage showcases recent products and services. Buyers are greeted with two horizontal sections “Products you might like” and “Services you might like” each showing listings with ratings, prices, and supplier names. A heart icon allows buyers to favorite listings for future reference. At the bottom, there’s a search input where users can enter a product name to discover similar products, powered by the platform’s AI-based alternative recommendation system.

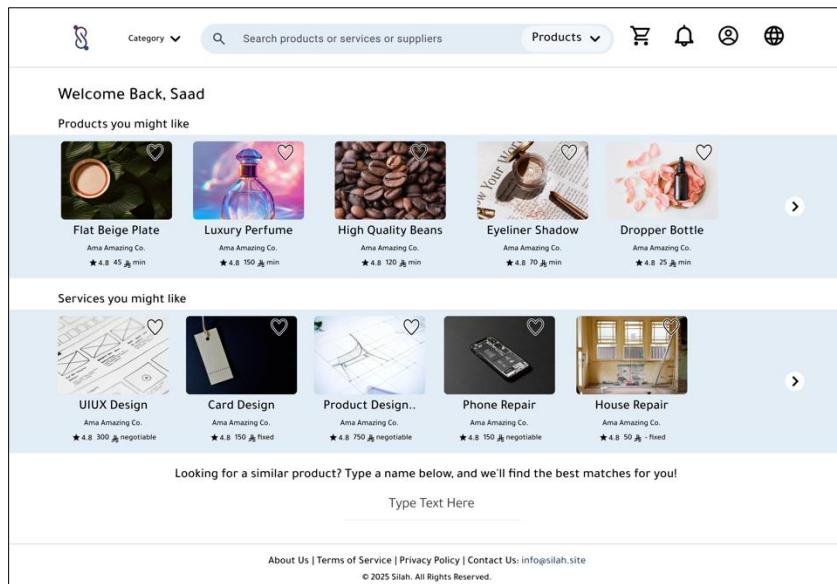


Figure 4-51: The homepage

Alternative Products Page

This page displays a list of similar products based on the buyer’s input or selection. For example, when searching for “Dropper Bottle”, the system finds and ranks semantically similar alternatives using AI-powered matching. Each result shows the product name, supplier, price, rating, and a match percentage to indicate relevance. Buyers can quickly explore related options and favorite any product for future reference. This feature supports smarter procurement by helping buyers discover comparable items from different suppliers.

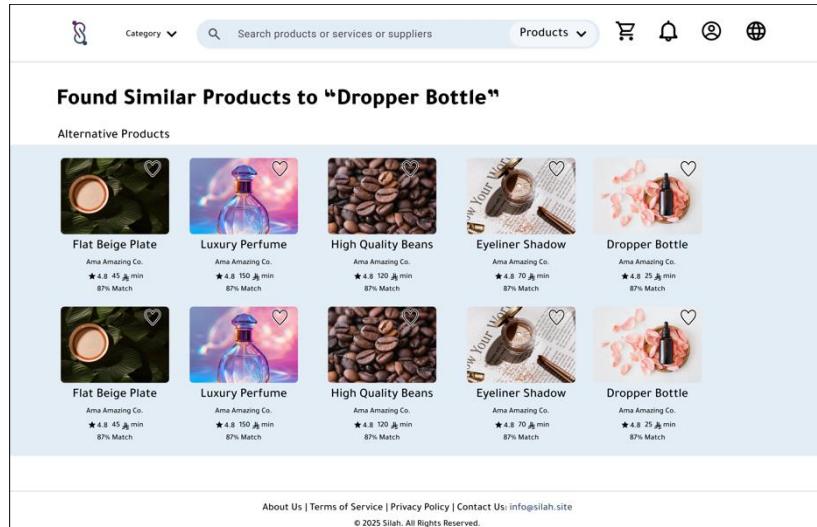


Figure 4-52: Alternative Products Page

Product Details & Join a Group Purchase

On the product details page, buyers can see if a group purchase is currently active. If available, they can choose to join other buyers and receive a discount by purchasing together. The interface shows how many more buyers are needed and how much time is left to complete the group. Buyers can also enter a custom quantity and join with one click. This feature encourages bulk ordering and helps buyers save money while improving supplier sales volume.

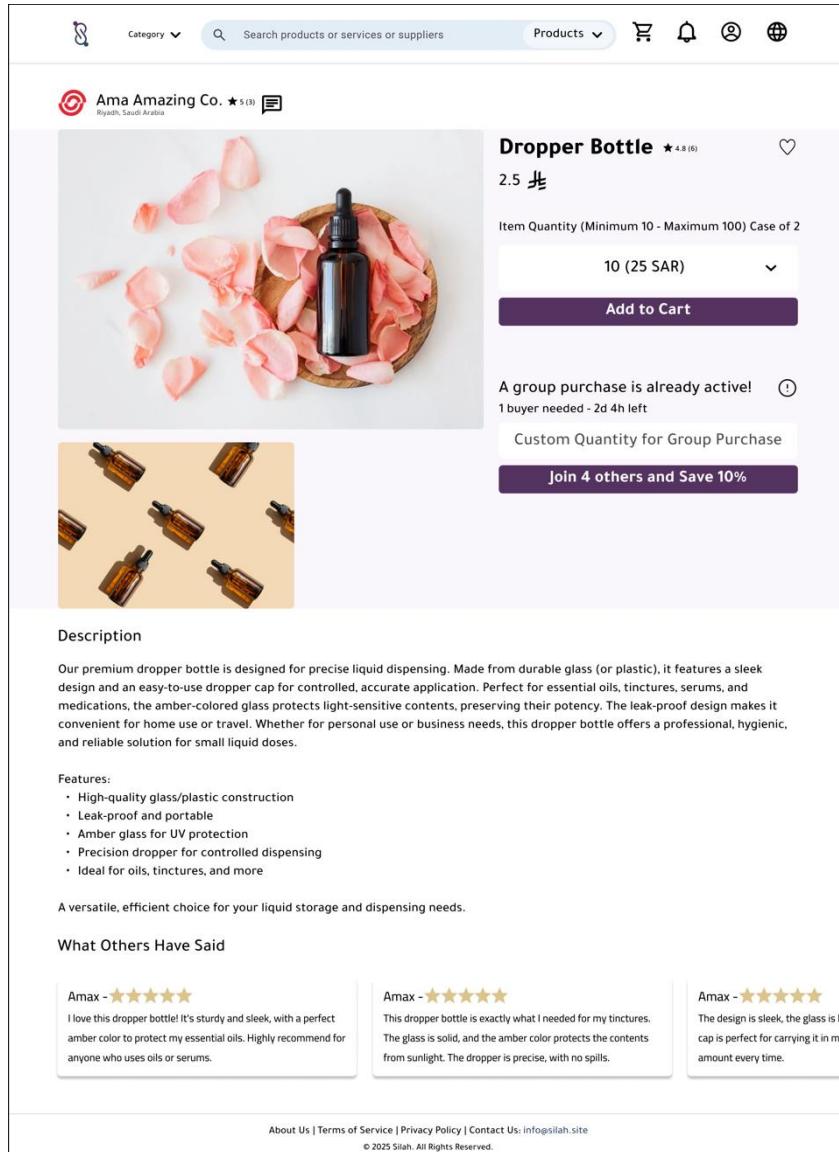


Figure 4-53: The Product Details Page

Cart

The Cart Page displays the items the buyer has added to their cart, grouped by supplier. Each item shows the quantity and price. If an item is out of stock, it is marked as unavailable, with an option to search for similar products. The cart shows the total for products, delivery fees, and the overall cart total. Buyers can remove items or clear the entire cart. To proceed, buyers must remove out-of-stock items to complete the checkout process.

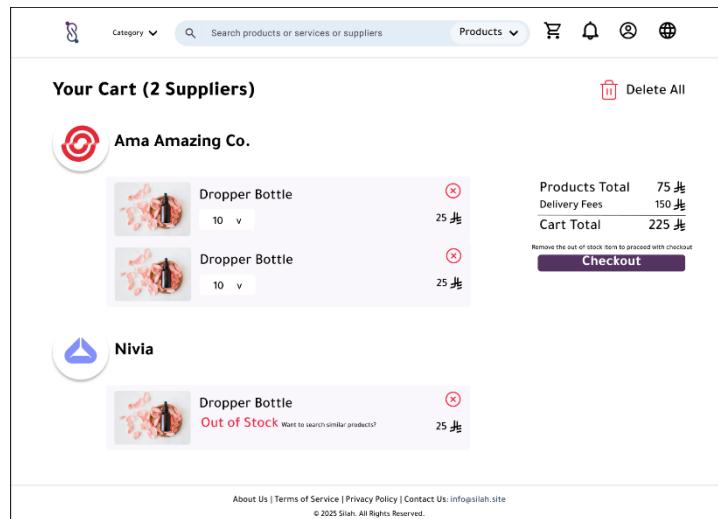


Figure 4-54: The Cart Page

Service Details

The Service Details Page provides buyers with key information about a service, including price, availability, description, images, and customer reviews. Buyers can request the service or negotiate the price directly with the supplier using the "Request Service" button.

UIUX Design • 300 ₢ Contact Ama to negotiate price

• Service Availability: Available 24/7

Request Service from Ama

Description

I am a creative and expert UI/UX Mobile App Designer with experience in designing intuitive and beautiful interfaces for mobile (iOS, Android), web apps, and tablets. I specialize in modern, responsive, and custom designs for various industries, including E-commerce, Education, Real Estate, Health & Fitness, Entertainment, and more.

What can I offer?

- Elegant & Attractive Designs
- User-friendly Interfaces
- Fully Responsive Layouts
- Custom Designs tailored to your needs
- Figma, Adobe XD, Photoshop designs
- Day-to-day UI/UX Work
- Professional Figma
- Editable Source Files with all Assets
- Guaranteed Satisfaction & Lifetime Support

You will receive a stunning and functional website or app that is easy to navigate and fully optimized.

UIUX Design | Mobile App Design | Digital Product Design | Figma | UI/UX Designer

Delivery Time: 15 days
Revisions: 1
Pages: 15

What Others Have Said

Amax - ★★★★★ I love this dropper bottle! It's sturdy and sleek, with a perfect amber color to protect my essential oils. Highly recommend for anyone who uses oils or serums.	Amax - ★★★★★ This dropper bottle is exactly what I needed for my tinctures. The glass is solid, and the amber color protects the contents from sunlight. The dropper is precise, with no spills.	Amax - ★★★★★ The design is sleek, the glass in the cap is perfect for carrying it in my bag every time.
---	--	---

About Us | Terms of Service | Privacy Policy | Contact Us: infospilah.site
© 2025 Spilah. All Rights Reserved.

Figure 4-55: The Service Details Page

Supplier Storefront

The Supplier Storefront Page displays the supplier's profile, including their name, location, and a brief description of their offerings. It shows the supplier's products and services, along with ratings, prices, and the option for buyers to add items to their favorites. If the supplier's account is inactive, the page is blurred. Buyers can also read customer reviews to help make purchasing decisions. This page serves as the main hub for buyers to explore and engage with the supplier's listings.

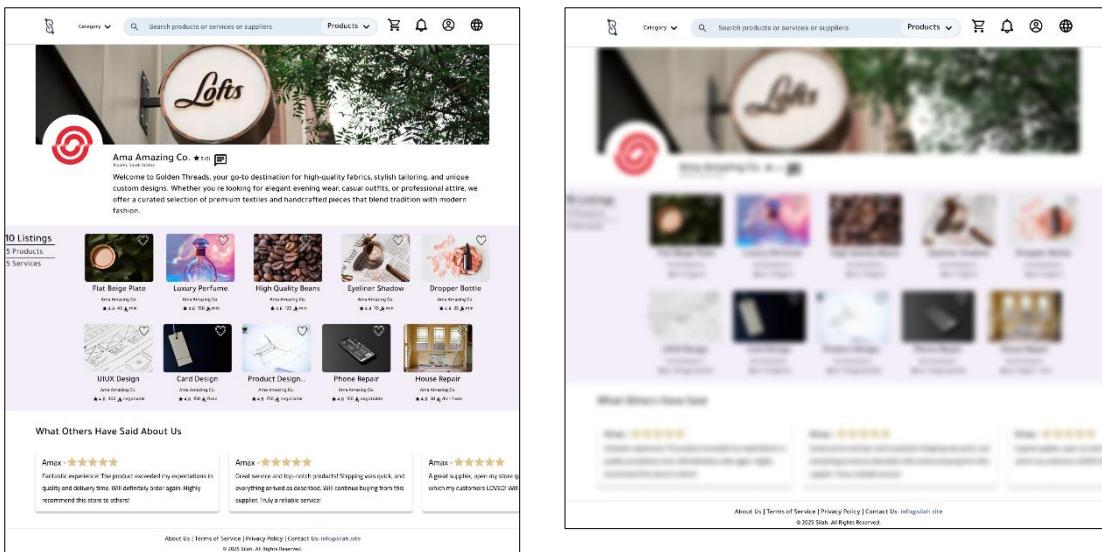


Figure 4-56: Supplier Storefront Page

Figure 4-57: Inactive Supplier Storefront Page

Search Results & Browsing by Category

The Search Results Page allows buyers to search for products or suppliers based on keywords. When searching for a product, buyers can filter results by the minimum order price. The page shows a grid of matching products, each with a name, rating, and time duration. If a buyer searches for a supplier, the page displays the supplier's profile with their description, along with some of their products and services. The search results make it easy for buyers to discover relevant items or suppliers and explore their offerings.

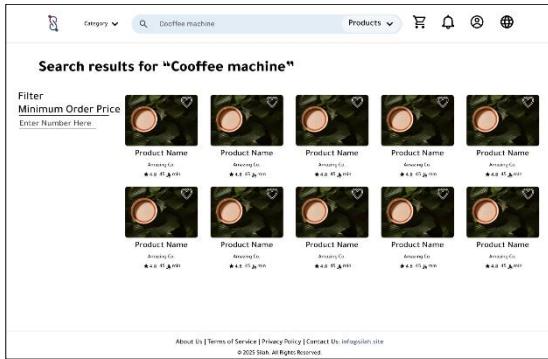


Figure 4-58: Searched for a Listing Page

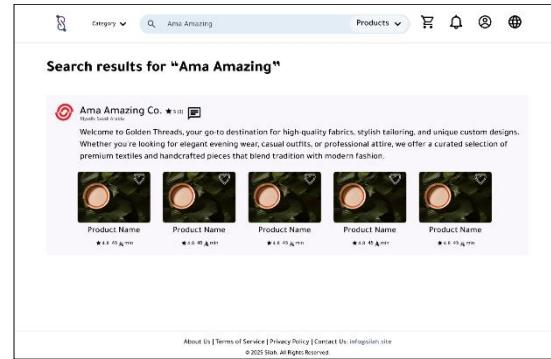


Figure 4-59: Searched for a Supplier Page

The Browsing by Category Page allows buyers to explore different categories of services or products. In this case, the category "Design Services" is displayed, with its sub-categories Product Design, Logo & Branding Design, and UI/UX & Graphic Design. Buyers can browse through various services within the selected category, see ratings, and filter results based on the minimum order price. Each service is displayed with its name, supplier information, and price. The page also allows buyers to add services to their favorites for easy access later.

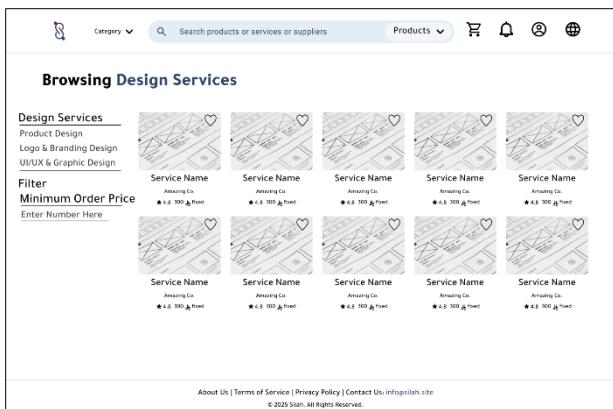


Figure 4-60: Browsing by Category Page

Wishlist

In the Wishlist Page the buyers can view the items they've saved for future reference. The page displays both products and services the buyer has shown interest in, each with the option to Find Alternatives. This feature helps the buyer easily search for similar products or services that might meet their needs, allowing them to explore options.

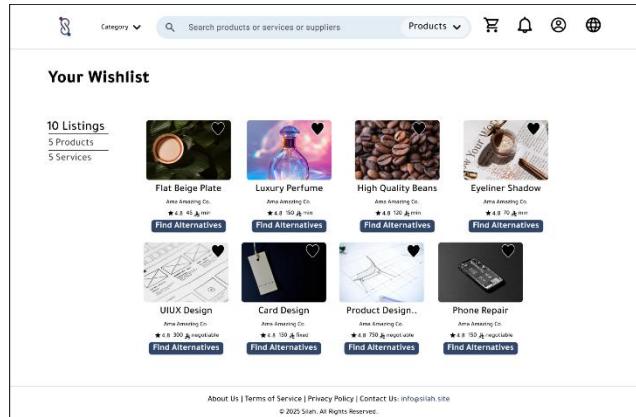


Figure 4-61: The Wishlist Page

Creating a New Bid

This page displays all the bids that the buyer has created. Each bid shows key information such as the publication date, reference number, and submission deadline. Buyers can click “View Details” to see the bid information or “View Offers” to browse submitted proposals, but only after the submission deadline has passed. A button at the top allows buyers to quickly create a new bid.

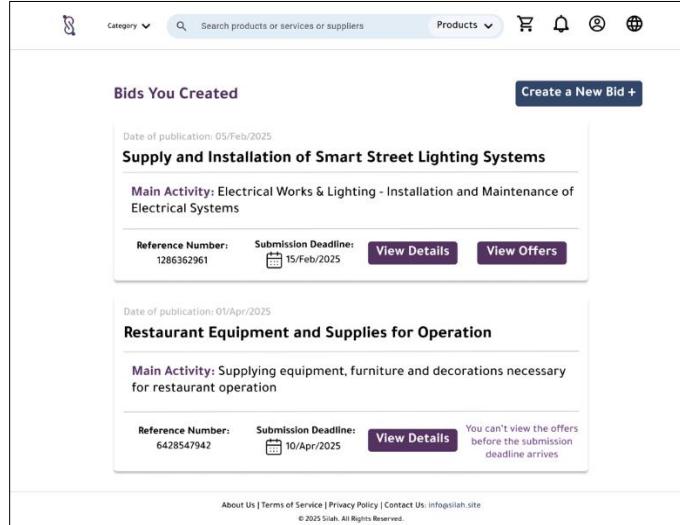


Figure 4-62: Bids Page

Buyers can create new bids by entering the bid name, selecting the main activity, setting the submission deadline, and choosing the expected response time for offers. Once filled out, they can publish the bid with a single click. This allows businesses to easily announce procurement needs and receive offers from suppliers directly through the platform.

Create a New Bid

Bid Name
Supply and Installation of Smart Street Lighting Systems

Main Activity
Electrical Works & Lighting - Installation and Maintenance of Electrical Systems

Submission Deadline
15/02/2025

Response Deadline for Offers
2 weeks after the submission deadline

Publish Bid

About Us | Terms of Service | Privacy Policy | Contact Us: infogsilah.site
© 2025 Silah. All Rights Reserved.

Figure 4-63: Create a Bid Page

Accepting an Offer

This page displays all the offers submitted by suppliers for a specific bid. This interface helps buyers compare proposals efficiently and make faster decisions.

Supplier	Offer Date	Proposed Amount	Project Completion Time
Vertex Suppliers	06/02/2025	15,000 ₦	4 months
Logen Industries	07/02/2025	25,000 ₦	6 months

About Us | Terms of Service | Privacy Policy | Contact Us: infogsilah.site
© 2025 Silah. All Rights Reserved.

Figure 4-64: Offers Received Page

On the next page, buyers can view the details of an individual supplier's offer. The page includes the proposed amount, expected completion date, technical offer description, project execution timeline, and any additional notes. Buyers can then accept or decline the offer directly from the page, allowing for clear and informed decision-making.

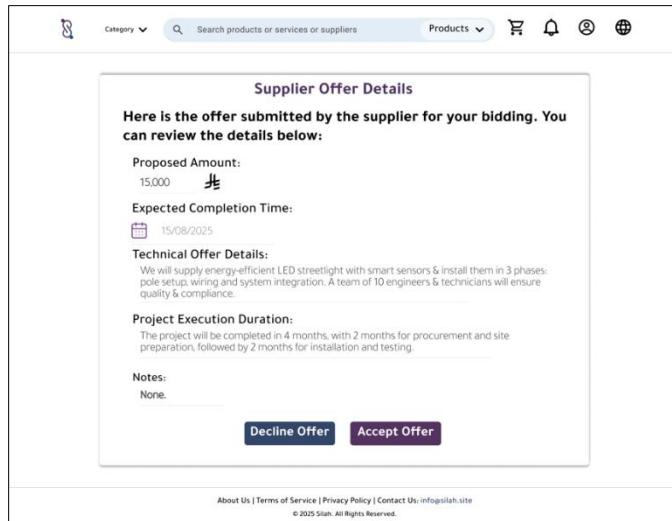


Figure 4-65: The Offer Details Page

Buyer Settings

The Buyer Settings Page allows users to manage their account settings, notifications, and payment methods. In the first image, where no card is stored, the user is prompted to add a card to make payments easier. Users can also update their profile picture, change their password, and manage the notifications they wish to receive, such as order status or new messages. The page also includes a support section for contacting customer service.

Figure 4-66: buyer Settings Page (No Card Stored)
Buyer Notifications

The Buyer Notifications Page displays important updates, including new messages, invoices, and order statuses. Notifications are categorized by type and date, helping buyers stay informed about their interactions and transactions.

Figure 4-67: Buyer Settings Page

Type	Date	Details
A new message from SkyLine	04/03/2025 2:45 PM	
A new invoice from Nasser - KSU	10/02/2025 1:20 PM	
The group purchase order has been canceled due to not meeting the required minimum of 5 buyers	06/02/2025 12:30 PM	
Order #2345 is now Shipped	23/01/2025 12:00 PM	

Figure 4-68: Buyer Notifications Page

Order History

The Orders Page allows buyers to view a list of their orders. Clicking on an order opens the Order Details Page, where buyers can review the specifics of the order. If the order status is marked as Shipped, the buyer has the option to Confirm Delivery. Once the order is marked as completed, the buyer can leave a Review.

All Orders	Pending	Processing	Shipped	Completed
#556903778	22 Mar	Skyline	Pending	₼ 10,000
#556903778	19 Mar	King Saud University	Shipped	₼ 30,000
#490903778	18 Mar	Almuneef	Pending	₼ 59,580
#446372778	17 Mar	King Khalid University	Completed	₼ 30,000
#556903778	22 Mar	Skyline	Completed	₼ 10,000
#556903778	19 Mar	King Saud University	Pending	₼ 30,000
#490903778	18 Mar	Almuneef	Shipped	₼ 59,580
#446372778	17 Mar	King Khalid University	Processing	₼ 30,000
#556903778	19 Mar	King Saud University	Completed	₼ 30,000
#490903778	18 Mar	Almuneef	Completed	₼ 59,580
#446372778	17 Mar	King Khalid University	Processing	₼ 30,000

Figure 4-69: Orders Page

Order Items				
Image	Item Name	Unit Price	Quantity	Total Price
	Amber - 45ml Soy Candle	₼ 10	16	₼ 160
	Amber - 45ml Soy Candle	₼ 10	16	₼ 160

Figure 4-70: The Order Details Page (Shipped)

Order Items			
✓ You Confirmed The Delivery of This Order			
	Amber - 45ml Soy Candle	₼ 10	16
	Amber - 45ml Soy Candle	₼ 10	16

Figure 4-71: The Order Details Page (Completed)

Ama Amazing Co.

Leave a Review for Ama A. Co.

★★★★★
Fantastic experience! The product exceeded my expectations in quality and delivery time. Will definitely order again. Highly recommend this store to others!

Leave a Review for the 2 ordered items

	Amber - 45ml Soy Candle
	Amber - 45ml Soy Candle

★★★★★
optional

★★★★★
optional

Figure 4-72: Write a Review Page

Direct Messaging & Search for Chats

The chat page, accessible from the buyer's header, allows buyers to communicate directly with suppliers. Since the design of the chat page for buyers and suppliers is quite similar, we've excluded the chat page here to avoid repetition of the supplier section shown earlier. Instead, we're showcasing the "Search for Chats" feature. This search bar, located on the chat page, allows buyers or suppliers to type a username or store name into the search field. The results, as shown in the second image, help users

easily find and jump to specific conversations without having to manually scroll through their messages.

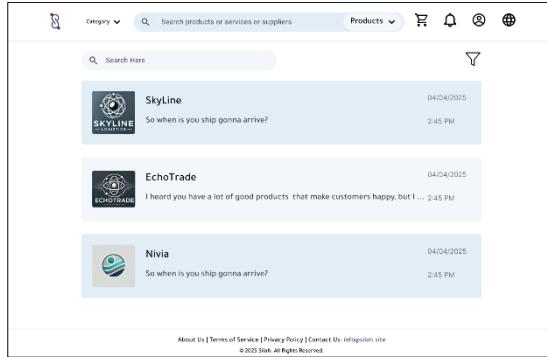


Figure 4-73: Chats Page

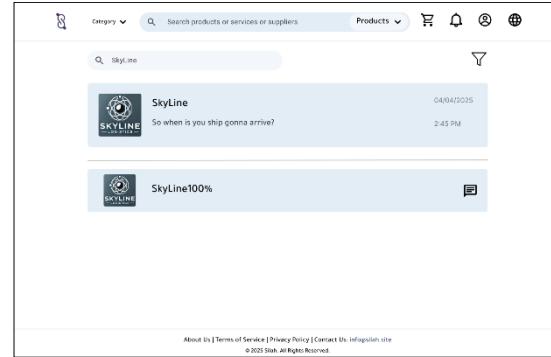


Figure 4-74: The Search for a Chat Page

Invoice Management

The Pre-Invoice status is displayed on the invoice list page, where buyers can see its progress in relation to the group purchase. The status may be "Pending" if the group is still gathering buyers, "Failed" if the minimum buyer requirement isn't met, or "Successful" when the group purchase reaches the required number of buyers.

Show invoices for:					
All	Accepted	Rejected	Partially Paid	Fully Paid	
#556903778	New	22 Mar	Skyline		₼ 10,000
#556903778		19 Mar	King Saudi University	Accepted	₼ 30,000
#490903778	Canceled	18 Mar	Almuneef	Failed	₼ 59,580
#446372778		17 Mar	King Khalid University	Accepted	₼ 30,000
#556903778		22 Mar	Skyline	Fully Paid	₼ 10,000
#556903778	Pending	19 Mar	King Saudi University	Pending	₼ 30,000
#490903778		18 Mar	Almuneef	Rejected	₼ 59,580
#446372778		17 Mar	King Khalid University	Partially Paid	₼ 30,000
#556903778		19 Mar	King Saudi University	Fully Paid	₼ 30,000
#490903778		18 Mar	Almuneef	Fully Paid	₼ 59,580
#446372778		17 Mar	King Khalid University	Accepted	₼ 30,000

About Us | Terms of Service | Privacy Policy | Contact Us | [infogroup.site](#)
© 2023 Stah. All Rights Reserved.

Figure 4-75: Invoices Page

Invoice Details																			
Invoice No: 12345678	Terms of Payment: Partially Paid																		
Issue Date: 23/Mar/2025	Upfront Payment Amount: 4000 ₩																		
Delivery Date: 06/Apr/2025	Upon Delivery Amount: 8000 ₩																		
Sender																			
Golden Threads Employee: Amazing Person City: Riyadh Email: info@goldenthreads.com	Receiver Nivia Co. Employee: Ahmed Ahmed City: Makkah Email: deals@nivia.com																		
<table border="1"> <thead> <tr> <th>Item</th> <th>Item Description</th> <th>Agreed Details</th> <th>Qty.</th> <th>Unit Price</th> <th>Total Price</th> </tr> </thead> <tbody> <tr> <td>Custom Tshirts</td> <td>Medium Size</td> <td>Delivery in 10 working days</td> <td>1</td> <td>1000</td> <td>1000</td> </tr> <tr> <td colspan="5">Final Price</td> <td>11000</td> </tr> </tbody> </table>		Item	Item Description	Agreed Details	Qty.	Unit Price	Total Price	Custom Tshirts	Medium Size	Delivery in 10 working days	1	1000	1000	Final Price					11000
Item	Item Description	Agreed Details	Qty.	Unit Price	Total Price														
Custom Tshirts	Medium Size	Delivery in 10 working days	1	1000	1000														
Final Price					11000														
Notes & Terms <small>The customer will receive 1 month of warranty for the same broken color and size, consistency with the order description and delivery location. Minimum days for delivery are 10 days. If there is any delay in delivery, the supplier must inform the customer and provide an explanation. The supplier is responsible for delivery, through any changes after production may affect the final product, the supplier becomes liable and cannot complain.</small>																			
<input type="button" value="Reject"/> <input type="button" value="Accept & Pay"/>																			

About Us | Terms of Service | Privacy Policy | Contact Us | [infogroup.site](#)
© 2023 Stah. All Rights Reserved.

Figure 4-76: The Invoice Details Page

Throughout the design of the platform, we focused on creating a seamless and intuitive experience for both buyers and suppliers. By maintaining consistency across similar UIs, we made it easier for users to navigate and interact with the platform, ensuring a smooth journey from one feature to the next. This approach not only enhances the overall usability but also ensures that the platform is easy to build, making it a more efficient and enjoyable experience for everyone involved.

Chapter 5 Implementation

In this chapter, we detail the implementation of Silah, translating the previously defined requirements and design choices into a working solution. We describe the practical steps taken to bring the system architecture to life, outline the challenges encountered during development, and explain the decisions made to overcome them. By examining both the technical obstacles and the reasoning behind our chosen approaches, this chapter provides a clear view of how the conceptual design evolved into a fully implemented system.

5.1 Implementation Requirements

This section outlines the implementation requirements necessary to develop and deploy the proposed system, including both software and hardware specifications.

5.1.1 Software Requirements

- **Frontend:** React.js (JavaScript) developed using Vite as a fast build and development tool for an interactive user interface.
- **Backend:** NestJS (Node.js) for API development, business logic, and authentication.
- **Database:** PostgreSQL for structured data storage, integrated with Prisma ORM for efficient database access and schema management.
- **Authentication Service:** Wathq API for Commercial Registration verification.
- **AI Models:**
 - FB Prophet for demand forecasting.
 - LaBSE for semantic similarity in multilingual product recommendations.
- **Hosting & Deployment:** DigitalOcean, a cloud provider.
- **Version Control:** Git and GitHub for code management and collaboration.
- **File Storage:** Cloudflare R2 for storing product images and documents.
- **Payment Gateway:** Tap Payments for secure transaction processing between buyers and suppliers.
- **Translation API:** DeepL Translation API to automatically translate product and services descriptions and names in the database based on the user's language selection.

5.1.2 Hardware Requirements

Server Requirements (For hosting the backend, database, and AI models):

- **Processor:** 2 vCPU.
- **RAM:** 4 GB for request handling and AI model execution.
- **Storage:** 80 GB SSD.
- **Bandwidth:** 4 TB per month.
- **Operating System:** Ubuntu 24.04.
- **Web Server:** Nginx is used as a reverse proxy to handle client requests and route them to the NestJS backend, while also serving the React frontend as static files.

Client Device Compatibility (Requirements for users accessing the system):

- **Operating Systems:** Windows, macOS, Linux.
- **Web Browsers:** Google Chrome, Mozilla Firefox, Safari (latest versions).
- **Internet Connection:** An active internet connection is required to access and use the website, as all interactions are processed online.

5.2 Implementation Details

Our project development was organized around three main areas: Artificial Intelligence (AI), Backend Development, and Frontend Development. To ensure efficiency and specialization, the team was divided into smaller sub-teams, each responsible for one of these areas. This structure reflected both the technical requirements of the system and the individual strengths of the team members.

Unlike the backend and frontend, which required continuous development throughout the project lifecycle, the AI-related tasks were more limited and temporary. Therefore, we did not assign a permanent AI Engineer role. Instead, some members from the frontend team temporarily shifted their focus to handle AI-specific responsibilities such as model fine-tuning, integration, and deployment.

This approach ensured multiple benefits:

1. **Workload Balance:** no single team member was overwhelmed with additional responsibilities.
2. **Skill Growth:** members gained exposure to multiple aspects of the project.

3. **Smoother Progress:** tasks were completed in parallel without delaying core backend and frontend development.
4. **Experience Utilization:** members were encouraged to work in areas where they already had prior experience, allowing the team to progress more efficiently instead of spending time learning technologies from scratch.
5. **Interest and Comfort:** all members selected their roles voluntarily based on what they enjoyed most and where they felt more confident. For example, some members chose frontend over backend because React was easier for them to learn and apply than NestJS, which requires deeper design decisions.
6. **Workload Distribution:** the frontend required significantly more code (three to four times larger than the backend), making it natural to assign more members to frontend tasks.

As a result of these considerations, the backend was assigned to a single member who was the most experienced in the group. Since the backend workload was concentrated within a relatively short period (about one and a half months), this distribution was both efficient and practical. The remaining four members were assigned to the frontend, which demanded continuous and larger-scale development. Within the frontend team, three of the members were also temporarily assigned to AI tasks, ensuring that those responsibilities were covered without delaying frontend progress.

It is also worth noting that all team members were balancing this project alongside university requirements, including other courses and co-op training. These external commitments naturally limited the time each member could dedicate, which made it even more important to distribute the workload in a practical and flexible way. Assigning four members to the frontend ensured that the large and ongoing tasks of that component were consistently covered, despite variations in availability.

To further support this distribution, the total lines of code in each area were measured after completing the implementation using the cloc tool. The analysis showed that the frontend codebase contains 39,739 lines, while the backend contains 23,763 lines. This result reflects a common trend in modern software development, where frontend implementation often requires greater effort and larger teams. This is

primarily due to the interactive and iterative nature of user interfaces, which demand continuous adjustments for usability, responsiveness, and visual consistency across devices.

github.com/AlDanial/cloc v 1.98 T=0.72 s (429.9 files/s, 61910.1 lines/s)					github.com/AlDanial/cloc v 1.98 T=0.24 s (1545.0 files/s, 111283.7 lines/s)				
Language	files	blank	comment	code	Language	files	blank	comment	code
JSX	95	1852	687	18451	TypeScript	373	1956	1148	23763
CSS	114	1556	689	12164	SUM:	373	1956	1148	23763
JSON	99	62	0	9867					
HTML	2	14	48	57					
SUM:	310	3484	1416	39739					

Figure 5-2: Backend cloc output

Figure 5-1: Frontend cloc output

As our team observed throughout the project, “UI work is deceptively difficult — the closer you get to the user, the harder programming becomes.” While this phrase emerged from our own experience, it echoes sentiments shared by many developers in the industry, including discussions on the *Coding Horror* blog [33] [34]. Similar patterns have been noted in surveys such as the *Stack Overflow Developer Survey (2024)* [35], where frontend and full-stack roles together constitute the majority of active development positions, indicating the growing emphasis on user-facing development in the industry.

In addition to dividing development tasks by technical area, the project itself was structured into three independent subprojects, each representing a complete software component with its own dependencies, folder structure, and lifecycle. These subprojects were:

- **silah-ai:** the AI backend (responsible for all machine learning models and FastAPI endpoints).
- **silah-backend:** the main backend (NestJS server managing business logic, database access, and integration with silah-ai and other service providers).
- **silah-frontend:** the frontend (React-based web application that interacts with the NestJS backend).

Each subproject was maintained in a separate GitHub repository under a shared organization account (<https://github.com/GP-Silah>), enabling clear separation of responsibilities and simplified version control. This structure also allowed the team to work in parallel without dependency conflicts and to manage issues, milestones, and pull requests more effectively.

Beyond workflow organization, this setup also proved beneficial for performance and practicality. Since each subproject could be run and tested independently, it placed a lighter load on our laptops; an important consideration given our limited hardware resources as students. Although we still encountered occasional performance issues during development, this modular setup helped minimize them and made the overall workflow much smoother.

For additional details about our GitHub setup, branching strategy, and project management practices (including the use of GitHub Projects and Issues), refer to Appendix C.

The following subsections provide the implementation details for each subproject: AI, Backend, and Frontend.

5.2.1 Artificial Intelligence Implementation

This section describes the work related to the two AI models (Facebook Prophet for demand forecasting and LaBSE for semantic search), as well as their integration into the system through FastAPI and deployment. The work is presented based on the lifecycle of AI development: training, integration, and deployment.

5.2.1.1 Fine-tuning / Training

Facebook Prophet

Facebook Prophet was used to forecast sales for each product individually. Its purpose is to enable suppliers to view predicted future sales for their products, helping them optimize stock management and supply decisions.

Model Setup and Data Preparation

Before training the forecasting model, the dataset and development environment were prepared to ensure compatibility with Prophet's input requirements. The dataset used was the *Sample Superstore* dataset, which originally contained around 18 features. From these, three key columns were selected and cleaned:

- **Order Date** → renamed to **ds** (required datetime field in Prophet)
- **Product ID** → used for grouping products
- **Sales** → renamed to **y** (target variable Prophet predicts)

All missing values were removed, and date formats were standardized.

The implementation was carried out in **Python 3.10**, using the following main libraries and tools:

- **Facebook Prophet** — main forecasting algorithm
- **pandas, numpy** — data preprocessing

- **scikit-learn** — accuracy metrics (MAE, RMSE)
- **matplotlib** — visualization
- **FastAPI** — integration and deployment of the trained model with the backend

```

import pandas as pd
from prophet import Prophet
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np
import matplotlib.pyplot as plt

# ----- Colors for printing -----
RED = "\033[91m"
GREEN = "\033[92m"
YELLOW = "\033[93m"
RESET = "\033[0m"

# 1. Load the dataset (اسم الملف الجديد)
df = pd.read_csv("data/Sample-Superstore.csv", sep=";", encoding="latin1")
df = df[['Order Date', 'Product ID', 'Sales']].copy()
df.rename(columns={'Order Date': 'ds', 'Sales': 'y'}, inplace=True)
df['ds'] = pd.to_datetime(df['ds'], errors='coerce')
df = df.dropna(subset=['ds', 'y'])

print(df['Product ID'].value_counts().head(10))

```

Figure 5-3: Code snippet showing dataset loading, column renaming, and library imports used during Prophet model preparation

Training Process

In a typical machine learning workflow, a model is trained using the entire dataset to learn general patterns and make predictions. However, during this project, we implemented a practical mechanism to simplify testing during development. Instead of training on all products at once, the system was configured to automatically identify and use the most frequently sold product in the dataset when running in development mode. This allowed the model to train faster and produce reliable results without the need to process the entire dataset. In user mode, by contrast, a specific product ID can be manually selected for forecasting, enabling flexibility during actual system use.

After selecting the target product, its sales records are divided into two subsets, 80% for training and 20% for testing, to allow balanced model evaluation. The Prophet model is then trained on the training subset to learn historical sales patterns and seasonal trends. Once training is complete, the model generates forecasts for the next three months (approximately 90 days). The forecast accuracy is evaluated using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) metrics, which measure how close the predicted sales values are to the actual data. Finally, the trained model is saved as `prophet_model_new.pkl` for later deployment.

During the training phase of the Prophet model, the dataset was split into 80% for training and 20% for testing. This ratio was not chosen arbitrarily; it was adopted

based on prior academic research that applied the Prophet model in similar time-series forecasting contexts. According to the study titled “Time-series Forecasting of Web Traffic Using Prophet Machine Learning Model” [5], the authors used the same 80/20 split when evaluating Prophet’s performance. They justified this choice as a balanced approach that provides sufficient data for learning while reserving enough unseen samples to assess generalization accuracy. Accordingly, the same methodology was applied in this project to maintain alignment with widely accepted research practices and to achieve an effective balance between training quality and evaluation reliability.

Next figure shows the code logic for development mode, filtering data for the selected product, and splitting it into training/testing subsets.

```
# ----- Mode -----
mode = "dev" # dev = development mode, user = seller mode

if mode == "dev":
    product_id = df['Product ID'].value_counts().index[0] # product with most rows
    print(f'{GREEN}🔍 Development Mode + Selected product {product_id}(RESET)')
else:
    product_id = "FUR-CH-1000454" # Example: seller manually selects
    print(f'{GREEN}🔍 User Mode + Selected product {product_id}(RESET)")

# 2. Filter data for the chosen product
df_product = df[df['Product ID'] == product_id][['ds', 'y']].copy()

if len(df_product) < 50:
    print(f'{YELLOW}⚠️ Warning: Product {product_id} has only {len(df_product)} rows → forecast may be inaccurate(RESET)")

# 3. Train/Test split (80/20)
split_idx = int(len(df_product) * 0.8)
train = df_product.iloc[:split_idx]
test = df_product.iloc[split_idx:]

model = Prophet()
model.fit(train)

future = pd.DataFrame(test['ds'])
forecast = model.predict(future)
```

Figure 5-4: Implementation of Development Mode and Dataset Splitting for Prophet Model Training

Next figure visualizes how the dataset distribution looks before training; showing which product has the most sales and is thus chosen for development mode training.

Product ID	Count
OFF-PA-10001970	19
TEC-AC-10003832	18
FUR-FU-10004270	16
FUR-CH-10002647	15
TEC-AC-10003628	15
TEC-AC-10002049	15
FUR-CH-10001146	15
OFF-BI-10001524	14
FUR-CH-10003774	14
OFF-PA-10002377	14

Name: count, dtype: int64
 🔎 Development Mode → Selected product OFF-PA-10001970
 ⚠️ Warning: Product OFF-PA-10001970 has only 19 rows → forecast may be inaccurate

Figure 5-5: Identification of Most Sold Product Used for Model Training

Model Evaluation

The Prophet model achieved an MAE of 166.27 and an RMSE of 181.21 when forecasting sales for the selected product.

These results indicate that, on average, the model’s predictions deviated moderately

from actual sales values, showing reasonable forecasting performance given the dataset's limited size and variability.

- Accuracy check for product OFF-PA-10001970
- MAE: 166.27
- RMSE: 181.21

Figure 5-6: Model Evaluation Metrics (MAE and RMSE) for the Most Sold Product

Forecast Results

Prophet predicted sales for the next three months as follows: 6119.33 units in January 2018, 2573.65 in February 2018, and 2213.72 in March 2018. However, since some products had very limited historical data, the model initially produced a few negative forecasts, which were automatically corrected to zero, as sales cannot be negative (Figure X). The final forecasted sales values for the next three months are presented in Figure Y.

- ⚠ Note: 24 forecast values were negative and were adjusted to 0.
- 💡 Explanation: Sales cannot be negative. They were set to 0.

Figure 5-7: Correction of Negative Forecast Values Generated by Prophet

- 📊 Sales forecast for the next 3 months:
- 2018-01: 6119.33 sales
- 2018-02: 2573.65 sales
- 2018-03: 2213.72 sales

Figure 5-8: Predicted Sales for the Next Three Months (January–March 2018)

The figure below shows the actual sales (blue) versus forecasted values (red) over time. It illustrates Prophet's learned pattern and the forecasted sales extension.

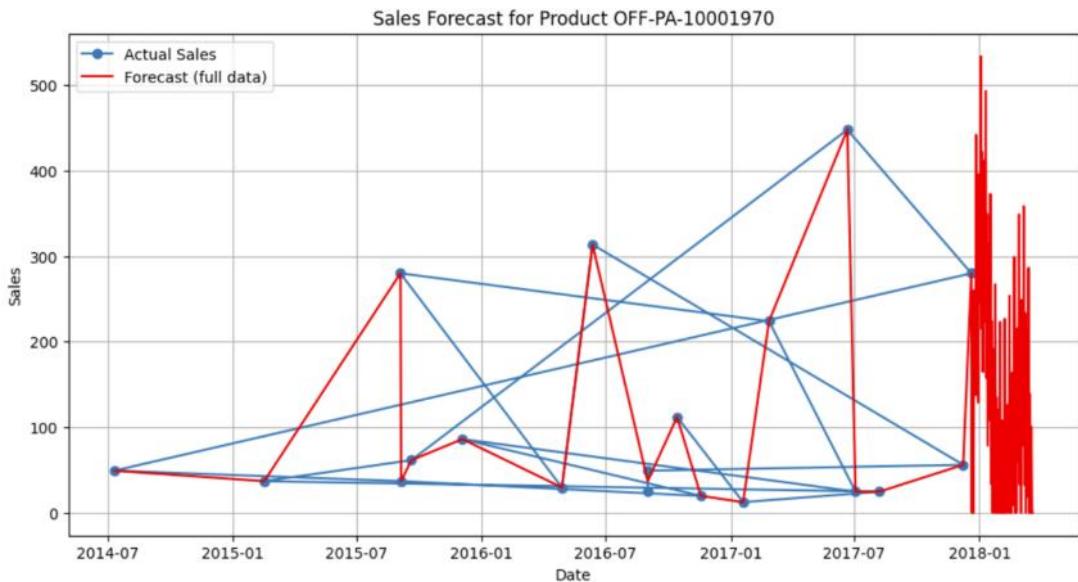


Figure 5-9: Actual vs. Forecasted Sales Over Time for the Selected Product

Challenges

During model development and testing, several issues were encountered. One of the main limitations was the small size of the dataset for certain products. In some cases, only 19 records were available, which reduced the reliability of the forecasts. Prophet also occasionally overfitted or produced negative predictions, particularly for products with irregular or sparse sales data. Because Prophet's accuracy depends heavily on both the volume and consistency of historical records, results varied between products. These limitations suggest that applying additional smoothing techniques or feature engineering could further enhance forecast stability and performance.

LaBSE

The LaBSE (Language-Agnostic BERT Sentence Embedding) model was employed for semantic product search and recommendation. Its purpose is to enable buyers to find products that are conceptually similar, even when different words or languages are used. By converting product titles and descriptions into high-dimensional vectors, LaBSE allows the system to compare the semantic meaning of different items, supporting multilingual search and recommendation functionalities.

Model Setup and Data Preparation

The LaBSE model was fine-tuned using the Amazon Sales Dataset from Kaggle, which contains 1,465 product entries. Each entry includes a product name, category, and detailed description. This dataset was chosen because it provides rich textual information and diverse product types, making it suitable for learning semantic similarities between product descriptions.

Before training, the data was carefully prepared within a Jupyter Notebook. The columns `product_id`, `product_name`, `about_product`, and `category` were first selected from the dataset. To create a single input text for the model, the product name, category, and full description were concatenated into a new column called `text_for_labse`. This combination allowed the model to learn from both concise and detailed product information.

To ensure cleaner data, rare categories (those appearing less than twice) were grouped under an “Other” label. Additionally, duplicate or incomplete entries were removed, and text fields were standardized by converting to lowercase, removing extra spaces, and stripping special characters.

Finally, the dataset was split into 80% for training and 20% for testing to evaluate the model performance.

```

# 1) بجانب النتبوت CSV ضعيف (أقرأ ملف)
df = pd.read_csv("amazon.csv")

# 2) خفي الأصناف المهمة
df_subset = df[["product_id", "product_name", "about_product", "category"]].copy()

# 3) نص التدريب = اسم المنتج + الوصف + الفئة (مع تعيين القيم المفارة)
df_subset["text_for_labse"] = (
    df_subset["product_name"].fillna("") + " " +
    df_subset["about_product"].fillna("") + " " +
    df_subset["category"].fillna("")
)

# 4) لغاري أخطاء التصنيف "Other" تجمع الفئات النادرة تحته: توسيع صور
counts = df_subset["category"].value_counts()
rare_mask = df_subset["category"].map(counts) < 2 # في نادرة أقل من عصرين
df_subset["category_for_split"] = df_subset["category"].where(~rare_mask, "Other")

print("Rows:", len(df_subset))
print("Tail of category_for_split counts:")
print(df_subset["category_for_split"].value_counts().tail(10))
df_subset[["text_for_labse"]].head(3)

```

Figure 5-10: Dataset loading and preprocessing for LaBSE fine-tuning

The fine-tuning process was implemented in Python 3.10 using several essential libraries:

- **PyTorch** — for model training and optimization.
- **Sentence-Transformers** — for handling LaBSE embeddings and fine-tuning utilities.
- **Pandas** and **NumPy** — for data preprocessing and manipulation.
- **Scikit-learn** — for model evaluation metrics.
- **FastAPI** — for integrating the trained model into the backend system.

To maintain reproducibility, a fixed random seed (42) was applied for both Python and PyTorch. The LaBSE model was loaded on the CPU (due to hardware limitations) and configured with a maximum sequence length of 128 tokens, balancing memory usage and input coverage.

```

import sys, os, math, random
import pandas as pd
import torch
from sentence_transformers import SentenceTransformer, InputExample, losses

# ثبت العشوائية لإعادة النتائج
random.seed(42)
torch.manual_seed(42)

# تحذيف مشارك الذاكرة على أجهزة CPU 8 GB
model = SentenceTransformer("sentence-transformers/LaBSE", device="cpu")
model.max_seq_length = 128 # حد أقصى لطول التسلسل (بميد الذاكرة)

print("LaBSE loaded ✅ | max_seq_length:", model.max_seq_length)
print("Python:", sys.version.split()[0], "| Torch:", torch.__version__, "| Device forced:", next(model.parameters()).device)

```

Figure 5-11: Library imports and LaBSE model initialization with fixed random seeds and CPU configuration

Training Process

To train the LaBSE model for product similarity detection, we designed a fine-tuning pipeline using Sentence-Transformers. Instead of training from scratch, the pre-trained LaBSE model was adapted to learn product-level relationships. That is, items within the same category should have similar embeddings, while unrelated products should remain distinct.

First, a custom function was implemented to automatically build positive text pairs. Each product served as an *anchor*, and another item from the same category was selected as its *positive example*. This ensured that the model learned meaningful category-based relationships. Products belonging to categories with only one item were skipped, since no valid positive pairs could be formed.

```
from sentence_transformers import InputExample
import pandas as pd
import random

def build_positive_pairs(texts, labels, per_anchor=1, seed=42):
    """
    تختار له منفذ آخر من نفس الفئة ليكون (positive). لكل منصر
    إذا كانت الفئة تحتوي على منفذ واحد فقط تلبيه منها.
    """
    rng = random.Random(seed)
    df_train = pd.DataFrame({"text": texts.values, "label": labels.values})

    pairs = []
    for _, grp in df_train.groupby("label"):
        items = grp["text"].tolist()
        if len(items) < 2:
            continue
        for i, anchor in enumerate(items):
            for _ in range(per_anchor):
                j = rng.randrange(len(items) - 1) # 0..len-2
                if j >= i:
                    j += 1
                positive = items[j]
                pairs.append(InputExample(texts=[anchor, positive]))
    return pairs

train_examples = build_positive_pairs(X_train, y_train, per_anchor=1, seed=42)
print("train_examples =", len(train_examples))
print("sample anchor:", train_examples[0].texts[0][:120], "\npositive:", train_examples[0].texts[1][:120])

train_examples = 1172
sample anchor: rts [2 Pack] Mini USB C Type C Adapter Plug, Type C Female to USB A Male Charger Charging Cable Adapter compat
positive: Kanget [2 Pack] Type C Female to USB A Male Charger | Charging Cable Adapter Converter compatible for iPhone 14, 13, 12,
```

Figure 5-12: Implementation of the `build_positive_pairs()` function for generating anchor-positive text pairs from the dataset

Next, the generated text pairs (1,172 examples in total) were loaded into a PyTorch DataLoader with a small batch size of four, chosen to reduce memory usage during CPU-based training. The training objective used was MultipleNegativesRankingLoss, which encourages semantically similar products to have higher cosine similarity scores.

```
from torch.utils.data import DataLoader
from sentence_transformers import losses

batch_size = 4 # مسح لتخفيض الذاكرة، ويمكن رفعه لاحقاً إذا أردت
train_dataloader = DataLoader(
    train_examples,
    shuffle=True,
    batch_size=batch_size,
    drop_last=True # أخير مسح قد يترك بعض الـ batch
)
losses.MultipleNegativesRankingLoss(model)

print(f"Train examples: {len(train_examples)} | Batches per epoch: {len(train_dataloader)}")
Train examples: 1172 | Batches per epoch: 293
```

Figure 5-13: DataLoader and loss function setup using `MultipleNegativesRankingLoss` for contrastive fine-tuning

The model was fine-tuned for one epoch, using a warm-up phase equal to 10% of the total steps. Training progress was tracked in Jupyter Notebook, and once completed, the model weights were saved to `/output/labse_finetuned_cpu/`.

```

num_epochs = 1
warmup_steps = math.ceil(len(train_dataloader) * num_epochs * 0.1)

model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    epochs=num_epochs,
    warmup_steps=warmup_steps,
    show_progress_bar=True,
    use_amps=False # على CPU نقل اللغة المختلقة أفضل لا
)
print("✅ Training finished.")

Error displaying widget: model not found
/Users/ro8oz/Desktop/ai-env/lib/python3.9/site-packages/torch/utils/data/dataloader.py:684: UserWarning: 'pin_memory' argument is set as true but not supported on MPS now, then device pinned memory won't be used.
warnings.warn(warn_msg)
[293/293 5:11:54, Epoch 1/1]

Step Training Loss
✅ Training finished.

```

Figure 5-14: LaBSE fine-tuning loop executed for one epoch with progress tracking and warm-up scheduling

This fine-tuning process successfully aligned the embedding space with product semantics, improving search and recommendation accuracy, especially for bilingual (Arabic/English) product names.

Model Evaluation

The model's performance was evaluated using Cosine Similarity and Mean Average Precision (MAP) between predicted and actual related products. High similarity scores (close to 1.0) indicated that the fine-tuned LaBSE effectively captured semantic meaning between related items.

Two products from the same category were compared with one from a different category. The model produced a cosine similarity of 0.96 for the same class and 0.32 for different classes, confirming that it could successfully distinguish between related and unrelated items.

```

save_dir = "outputs/labse_finetuned_cpu"
os.makedirs(save_dir, exist_ok=True)
model.save(save_dir)
print("Model saved to:", save_dir)

# اختبار سريع: تشابه بين جملتين من نفس اللغة مقابل فئة أخرى
from sentence_transformers import util

s1 = X_train.iloc[0]
s2 = X_train[y_train == y_train.iloc[0]].iloc[1] # من نفس اللغة
s3 = X_train[y_train != y_train.iloc[0]].iloc[0] # من فئة مختلفة

emb = model.encode([s1, s2, s3], convert_to_tensor=True, show_progress_bar=False)
sim12 = util.cos_sim(emb[0], emb[1]).item()
sim13 = util.cos_sim(emb[0], emb[2]).item()
print(f"cosine(same class) = {sim12:.3f} | cosine(diff class) = {sim13:.3f}")

Model saved to: outputs/labse_finetuned_cpu
cosine(same class) = 0.960 | cosine(diff class) = 0.324

```

Figure 5-15: Model saving and similarity validation between products from the same and different categories

The model was further tested using Top-1 and Top-5 accuracy metrics. For each product in the test set, its five nearest neighbors were retrieved based on embedding similarity. If the top prediction matched the correct category, it was counted as a Top-1 success; if the correct category appeared within the top five, it counted as a Top-5 success.

```

import numpy as np

# تجريب أقرب الجيران لكل مثال في test
D, I = nn.kneighbors(emb_test, n_neighbors=5) # D=مسافات I=جيران داخل train

# نظائر المثلثات
y_train_arr = np.array(y_train) # تأكيد alignment مع emb_train
y_test_arr = np.array(y_test)

top1_correct = 0
top5_correct = 0

for idx_test, neigh_idx in enumerate(I):
    true_label = y_test_arr[idx_test]
    neigh_labels = y_train_arr[neigh_idx]

    if neigh_labels[0] == true_label:
        top1_correct += 1
    if (neigh_labels == true_label).any():
        top5_correct += 1

top1 = top1_correct / len(y_test_arr)
top5 = top5_correct / len(y_test_arr)

print(f"Top-1 accuracy: {top1:.3f}")
print(f"Top-5 accuracy: {top5:.3f}")

Top-1 accuracy: 0.850
Top-5 accuracy: 0.962

```

Figure 5-16: Evaluation code for computing Top-1 and Top-5 accuracy across the test set

The results showed:

- **Top-1 Accuracy:** 0.85 → The most relevant product was correctly identified 85% of the time.
- **Top-5 Accuracy:** 0.96 → The correct product appeared within the top five recommendations 96% of the time.

Challenges

Several challenges were encountered during the fine-tuning stage:

1. Limited Arabic Product Data

The dataset contained relatively few Arabic product descriptions compared to English ones, which required additional preprocessing and class balancing to prevent the model from becoming biased toward English text patterns.

2. Hardware Constraints

Due to the lack of GPU acceleration, the fine-tuning process was executed on CPU. This imposed strict memory limits, especially when experimenting with larger batch sizes, leading to slower training iterations.

3. Mixed-Language Product Descriptions

Many product names and details combined Arabic and English terms (e.g., “محول Type-C”), which occasionally confused the pre-trained model. Extra tokenization and normalization steps were needed to ensure consistent embedding representations.

4. Hyperparameter Optimization

Determining the best learning rate, batch size, and number of epochs required several experimental runs. Subtle changes in these parameters had a noticeable impact on semantic similarity accuracy, particularly for bilingual samples.

5.2.1.2 FastAPI endpoints & integration

Both AI models (Prophet for sales forecasting and LaBSE for product similarity) were integrated into the system using the FastAPI framework.

This approach was necessary because the AI models were developed in Python, while the main backend (NestJS) uses TypeScript and therefore cannot directly invoke Python-based logic. FastAPI serves as a lightweight bridge between the AI models and the main application. Each model exposes its own endpoint through a shared `main.py` entry point, while the core logic resides in separate modules (`prophetAI.py` and `labse_ai.py`).

As illustrated in Figure 5-17, the FastAPI project (named `silah-ai`) includes these main Python modules along with essential configuration and documentation files such as `.gitignore`, `requirements.txt`, `README.md`, and a `CUSTOM-LICENSE`. These files collectively define the project environment, manage dependencies, and ensure proper documentation and licensing practices.

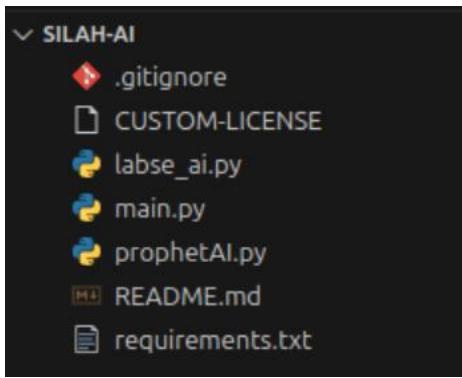


Figure 5-17: FastAPI Project Structure

The backend (NestJS) communicates with these endpoints internally, ensuring that no AI service is directly exposed to the client interface.

This design choice is essential because the data exchanged with the AI models differs from what the frontend provides or expects. For example, the frontend only sends a product ID to request a forecast or find similar items, whereas Prophet requires the full sales history for that product, and LaBSE requires metadata for all products to compute similarity scores.

NestJS therefore acts as a translator between layers: it retrieves and prepares the necessary input data from the database, calls the corresponding FastAPI endpoint, and then enriches the AI model's output before returning it to the frontend.

Once the AI models return their raw predictions, NestJS enriches the results before sending them to the frontend. In the case of Prophet, the system combines the forecasted values with additional business logic to estimate the optimal stock quantities for each product, helping maintain a balance between understocking and overstocking. Similarly, for LaBSE, the similarity results are post-processed by NestJS to include full product information retrieved from the database and to organize the output into a ranked list of the Top-10 most relevant items.

NestJS also maintains exclusive control over the PostgreSQL database. The AI backend (FastAPI) does not communicate with the database directly. This design decision was made for two key reasons.

First, consistency and simplicity: NestJS already manages all database operations through Prisma ORM, including schema definition, migrations, and queries. Allowing Python-based services to access the same database would require either implementing raw SQL queries or introducing a separate ORM in Python, which would complicate maintenance and risk synchronization issues.

Second, practical efficiency: as a student project with limited time, computational resources, and experience, this separation made development faster and safer. Instead of connecting multiple technologies to the same data source, NestJS simply sends the required data to FastAPI as structured JSON. This approach reduced setup complexity, avoided permission and ORM conflicts, and allowed the team to focus on delivering working AI functionality within project constraints.

Additionally, the frontend does not call the FastAPI endpoints directly, primarily due to authentication and access control requirements. All authentication logic is handled within NestJS, which ensures that only authorized users can interact with the AI services. For example, in the case of Prophet, only suppliers are permitted to request forecasts, and they can only generate predictions for products they own; buyers or guest users are not allowed to access these endpoints. By centralizing authentication and user validation in NestJS, the system avoids redundant or potentially inconsistent logic in FastAPI, simplifies security management, and maintains a single source of truth for user permissions. This design also aligns with the overall principle that FastAPI serves purely as an inference layer, while NestJS controls access, data preparation, and result enrichment.

Facebook Prophet

After training and saving the Prophet model, it was integrated into FastAPI through a dedicated endpoint (/demand). This allows the backend (NestJS) to request forecasts programmatically and return results to the frontend without direct exposure of the model logic. The forecasting workflow is encapsulated within prophetAI.py, while the main.py file defines the API routes and request models.

To ensure consistent and structured communication, two Pydantic models were defined:

- **SaleRecord:** represents a single daily sales record, including the purchase date and quantity sold.
- **ForecastRequest:** defines the overall forecasting request, containing the product_id, a list of sales records, and the number of months to forecast (default = 3).

```

class SaleRecord(BaseModel):
    date: date
    quantity: int

class ForecastRequest(BaseModel):
    product_id: str
    sales: List[SaleRecord]
    months: int = 3 # default forecast horizon

```

Figure 5-18: FastAPI data models for sales forecasting requests (Prophet)

These data models ensure a consistent, JSON-based exchange format between NestJS and FastAPI, implementing the communication structure introduced in Chapter 4.

To illustrate how these Pydantic models are used in practice, Figure Y shows sample request and response bodies for the /demand endpoint. The request example demonstrates how the NestJS backend sends a product ID along with its historical sales records and the desired forecast horizon. The response example highlights the structured JSON returned by FastAPI, containing the product ID and a list of forecasted monthly demand values.

Example Request Body:

```
{
  "product_id": "239284-4324-efsd",
  "months": 3,
  "sales": [
    {"date": "2025-09-01", "quantity": 5},
    {"date": "2025-09-02", "quantity": 2}
  ]
}
```

Example Response Body:

```
{
  "product_id": "239284-4324-efsd",
  "forecast": [
    {"month": "2025-10", "demand": 30},
    {"month": "2025-11", "demand": 25},
    {"month": "2025-12", "demand": 28}
  ]
}
```

Figure 5-19: Request and Response Body Examples (/demand)

After defining the DTOs, a dedicated FastAPI endpoint (/demand) was created to handle POST requests containing the formatted sales data. When triggered, this endpoint calls the run_forecast(req, model) function from the prophetAI.py module. This function loads the pre-trained Prophet model, performs the sales forecast for the next n months (default = 3), and returns the results in a structured JSON response.

```
@app.post("/demand")
def forecast_post(req: ForecastRequest):
    return run_forecast(req, model)
```

Figure 5-20: FastAPI /demand endpoint defined in main.py

The forecasting process is handled by the `run_forecast()` function inside the `prophetAI.py` module. This function is responsible for generating a monthly demand forecast for a given product. It accepts two main arguments: the forecast request DTO, which includes the product's sales history, and a Boolean flag `use_pretrained`, which determines whether to load an existing Prophet model or to fit a new one.

When invoked, the function first converts the incoming sales records into a structured Pandas DataFrame, where the transaction dates are mapped to Prophet's expected time-series field `ds`, and the corresponding quantities are mapped to `y`. This format is required by Prophet for model fitting.

If a pre-trained model is available and the flag `use_pretrained` is set to True, the function loads that model directly to save time. Otherwise, it fits a new Prophet instance using the provided sales data. Once the model is ready, it generates daily forecasts for the specified future horizon and then aggregates the results into monthly totals to better align with the business reporting period.

Finally, the function compiles the output into a structured dictionary (dict in Python, meaning a key-value data structure similar to JSON) containing the product `_id` and a list of forecasted values, each representing the expected demand per month. The result is then returned to the FastAPI endpoint as a JSON response.

```

def run_forecast(req, use_pretrained=True):
    """
    Generate a monthly demand forecast for a product.

    Args:
        req: ForecastRequest object from FastAPI
        use_pretrained: bool, whether to try using pre-trained model first

    Returns:
        dict with product_id and monthly forecast
    """

    # Convert sales records to DataFrame
    df = pd.DataFrame([{"ds": pd.to_datetime(r.date), "y": r.quantity} for r in req.sales])

```

Figure 5-21: Implementation of run_forecast() function inside prophetAI.py, showing input arguments and preprocessing logic

```

# Aggregate daily predictions into monthly sums
monthly = fc_future.set_index("ds")["yhat"].resample("M").sum().head(req.months)
forecast_list = [
    {"month": d.strftime("%Y-%m"), "demand": int(v)}
    for d, v in monthly.items()
]

return {
    "product_id": req.product_id,
    "forecast": forecast_list
}

```

Figure 5-22: Forecast generation and aggregation steps, converting Prophet's daily predictions into monthly demand values

LaBSE

To ensure structured communication between the backend (NestJS) and FastAPI, the /similar-search endpoint relies on a dedicated DTO, SimilarSearchRequest, implemented as a Pydantic model in Python. This DTO defines all necessary input fields and supports two main search scenarios:

1. Text-based search: when a user types a query in the search bar.
2. Product-based search: when a user selects an existing product to find similar items.

The **SimilarSearchRequest** includes the following fields:

- **text** (string, required): the search query entered by the user. If the search is triggered by selecting a product, NestJS automatically fills this field with the product name.
- **item_id** (string, optional): the ID of the selected product. Optional because it is only needed when searching for similar products.
- **embedding** (list of floats, optional): a precomputed embedding vector for the product. Optional because it is only used when searching for similar products, it also can be null if the AI service couldn't generate embeddings earlier, in

which case LaBSE computes it dynamically now. This makes the result faster in case embeddings were found for all candidates.

- **candidates** (list of product objects): each candidate includes the following fields:

- **id** (string)
- **name** (string)
- **description** (string, optional)
- **category_name** (string, optional)
- **embedding** (list of floats, optional)

These data models guarantee a consistent JSON-based interface, simplifying validation and error handling while ensuring NestJS can reliably forward requests to FastAPI.

```
class Candidate(BaseModel):
    id: str
    name: str
    description: Optional[str] = None
    category_name: Optional[str] = None
    embedding: Optional[List[float]] = None

class SimilarSearchRequest(BaseModel):
    text: str                      # required: the text used in the request -> written by the user to search for similar items
    item_id: Optional[str] = None   # optional: the item being searched id (if any)
    embedding: Optional[List[float]] = None # optional: the item being searched embedding (if any)
    candidates: List[Candidate]
```

Figure 5-23: FastAPI data models for product similarity requests (LaBSE)

Example Request Body:

```
{
    "text": "Dog food for puppies",
    "item_id": null,
    "embedding": null,
    "candidates": [
        {
            "id": "prod-123",
            "name": "Premium Puppy Food",
            "description": "High protein puppy food",
            "category_name": "Pet Supplies",
            "embedding": null
        }
    ]
}
```

Example Response Body:

```
[
    {"id": "prod-123", "rank": 1, "score": 0.87}
]
```

Figure 5-24: Request and Response Body Examples (/similar-search)

The /similar-search endpoint is defined in main.py and handles POST requests containing a SimilarSearchRequest object. Upon receiving a request, it calls find_similar_items() in labse_ai.py, which:

1. Generates or loads embeddings for the query.
2. Computes cosine similarity scores between the query and all candidate products.
3. Ranks the results by similarity and returns the top-K matches.

The response is a JSON array where each element contains:

- **id**: product identifier
- **rank**: similarity rank
- **score**: cosine similarity value

This design ensures that the AI service remains internal to the backend, while NestJS enriches results with additional product information from the database before sending them to the frontend, just like Prophet.

```
# ----- LaBSE endpoints -----
@app.post("/similar-search")
def similar_search_post(req: SimilarSearchRequest):
    try:
        # req.dict() is not required - pass fields explicitly
        results = find_similar_items(
            text=req.text,
            item_id=req.item_id,
            embedding=req.embedding,
            candidates=[c.dict() for c in req.candidates],
            top_k=10,
        )
    return results
```

Figure 5-25: FastAPI /similar-search endpoint defined in main.py

The core functionality of the /similar-search endpoint is implemented in the `find_similar_items()` function inside `labse_ai.py`. This function performs the following steps:

1. **Validate input:** Ensures that candidate products are provided. If missing, a 400 Bad Request error is returned.
2. **Prepare query embedding:** If the request includes an embedding, it is converted to a tensor. Otherwise, LaBSE computes the embedding from the query text dynamically.
3. **Prepare candidate embeddings:** Converts existing embeddings to tensors and computes missing embeddings in batch. Any failures result in a Bad Gateway error.
4. **Compute similarity:** Cosine similarity is calculated between the query embedding and all candidate embeddings. A fallback implementation ensures reliability if the primary method fails.

5. **Compile and rank results:** Creates a list containing the candidate ID and similarity score, sorts by similarity, selects the Top-K matches (default 10), and assigns ranks.

```

def find_similar_items(
    text: str,
    item_id: Optional[str],
    embedding: Optional[List[float]],
    candidates: List[Dict],
    top_k: int = 10,
) -> List[Dict]:
    """
    Main similarity function.
    - text: the search text (string) – always present.
    - item_id: optional
    - embedding: optional embedding for the query (list of floats)
    - candidates: list of dicts with keys:
        - id (str), name (str), description (opt), category_name (opt), embedding (opt list)
    Returns top_k results sorted by similarity as:
    [{"id": ..., "rank": 1, "score": 0.87}, ...]
    """

    if not isinstance(candidates, list) or len(candidates) == 0:
        raise HTTPException(status_code=400, detail="Candidates list required")

    model = get_model()

    # 1) Build query embedding
    try:
        if embedding:
            # Convert provided embedding to tensor
            query_emb = _to_tensor(embedding).float()
            if query_emb.dim() == 1:
                query_emb = query_emb.unsqueeze(0) # shape (1, dim)
        else:
            # compute from text
            try:
                query_emb = model.encode(text, convert_to_tensor=True)
            except TypeError:
                # model doesn't accept convert_to_tensor param
                q_np = model.encode(text, convert_to_numpy=True)
                query_emb = torch.tensor(q_np)
            if query_emb.dim() == 1:
                query_emb = query_emb.unsqueeze(0)
    except Exception as e:
        raise HTTPException(status_code=502, detail=f"Failed to compute query embedding: {e}")

```

Figure 5-26: Implementation of `find_similar_items()` inside `labse_ai.py` (1/3)

```

# 2) Prepare candidate embeddings: use provided ones where available; compute missing ones in batch
candidate_emb_tensors = [None] * len(candidates)
missing_texts = []
missing_indices = []

for idx, c in enumerate(candidates):
    if c.get("embedding"):
        try:
            tensor = _to_tensor(c["embedding"]).float()
            if tensor.dim() == 1:
                tensor = tensor.unsqueeze(0)
            candidate_emb_tensors[idx] = tensor
        except Exception:
            # if conversion fails, treat as missing
            missing_indices.append(idx)
            missing_texts.append(" ".join(filter(None, [c.get("name","", c.get("description") or "", c.get("category_name") or "")].strip())))
    else:
        missing_indices.append(idx)
        missing_texts.append(" ".join(filter(None, [c.get("name","", c.get("description") or "", c.get("category_name") or "")].strip())))

# compute missing embeddings in batch (if any)
if missing_indices:
    try:
        missing_embs = model.encode(missing_texts, convert_to_tensor=True)
    except TypeError:
        missing_embs_np = model.encode(missing_texts, convert_to_numpy=True)
        missing_embs = torch.tensor(missing_embs_np)
    # missing_embs should be shape (n, dim)
    for i, idx in enumerate(missing_indices):
        vec = missing_embs[i]
        if vec.dim() == 1:
            vec = vec.unsqueeze(0)
        candidate_emb_tensors[idx] = vec
    except Exception as e:
        raise HTTPException(status_code=502, detail=f"Failed to compute candidate embeddings: {e}")

```

Figure 5-27: Implementation of `find_similar_items()` inside `labse_ai.py` (2/3)

```

# 3) Stack candidate_tensors into a single tensor (n, dim)
try:
    # ensure all candidate_emb_tensors are set
    candidate_embs = [c.squeeze(0) if c.dim()==2 and c.size(0)==1 else c for c in candidate_emb_tensors]
    candidate_matrix = torch.stack(candidate_embs) # shape (n, dim)
except Exception as e:
    raise HTTPException(status_code=500, detail=f"Failed to create candidate embedding matrix: {e}")

# 4) Compute cosine similarity
try:
    # util.cos_sim returns (1, n) for query_emb (1, dim) vs candidate_matrix (n, dim)
    sims = util.cos_sim(query_emb, candidate_matrix)[0] # tensor length n
    sims_np = sims.detach().cpu().numpy().tolist()
except Exception as e:
    # fallback to manual numpy cosine if util fails
    try:
        import numpy as np
        q = query_emb.detach().cpu().numpy().reshape(-1)
        cm = candidate_matrix.detach().cpu().numpy()
        dot = (cm @ q)
        qnorm = (q ** 2).sum() ** 0.5
        cmnorm = (cm ** 2).sum(axis=1) ** 0.5
        sims_np = (dot / (cmnorm * qnorm + 1e-10)).tolist()
    except Exception as e2:
        raise HTTPException(status_code=502, detail=f"Failed to compute similarity: {e} / {e2}")

# 5) Build result list and sort
results = []
for idx, c in enumerate(candidates):
    score = float(sims_np[idx]) if idx < len(sims_np) else 0.0
    results.append({"id": c.get("id"), "score": score})

results = [r for r in results if r["id"] is not None]
results.sort(key=lambda x: x["score"], reverse=True)

# 6) Take top_k and assign ranks (1 = most similar)
top = results[:top_k]
out = []
for i, r in enumerate(top):
    out.append({"id": r["id"], "rank": i + 1, "score": r["score"]})

return out

```

Figure 5-28: Implementation of find_similar_items() inside labse_ai.py (3/3)

This process ensures that the AI service does not access the database directly, while NestJS enriches the results with additional product information before sending them to the frontend.

5.2.1.3 Deployment

The AI backend for Silah, which includes the Prophet-based demand forecasting and LaBSE semantic similarity models, was deployed on our DigitalOcean droplet named silah-site-server. This deployment centralizes all backend services, allowing the frontend team to access APIs directly from their local machines without the need to run or manage the AI or NestJS servers locally. This setup improves developer experience, reduces resource usage on student laptops, and ensures a consistent environment for testing and development.

Initially, the droplet was provisioned with modest resources to minimize costs, offering 1 vCPU, 2 GB of RAM, 50 GB of SSD storage, 2 TB of monthly bandwidth, and Ubuntu 24.04 as the operating system. Secure SSH key-based access was configured to manage the server remotely. The AI backend repository was cloned from GitHub, and a virtual environment was created to isolate Python dependencies. During the installation of transformer-based packages required for LaBSE, the process repeatedly failed due to insufficient RAM, triggering the Linux kernel's Out-of-Memory (OOM) killer. To resolve this, the droplet was upgraded to 2 vCPUs, 4 GB of RAM, 80 GB of SSD storage, and 4 TB of monthly bandwidth. After the upgrade, all dependencies, including PyTorch and sentence-transformers, were installed successfully.

Once dependencies were in place, the FastAPI server was started using uvicorn main:app. To ensure the server continued running independently of the SSH session, the session was mirrored and detached using Ctrl+A + D in screen. This allows the AI backend to remain active and accessible even after logging out from the server.

With this deployment, frontend developers can make real API calls without managing local servers, and computationally intensive tasks, such as generating embeddings and running forecasts, are offloaded from student laptops. Overall, this strategy provided a stable, production-ready environment that is easy to maintain, reproducible, and accessible at all times.

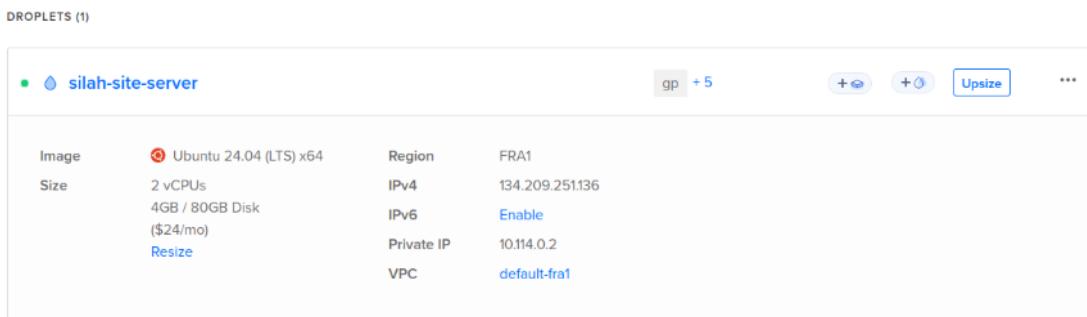


Figure 5-29: silah-site-server Droplet on DigitalOcean

5.2.2 Backend Implementation

The backend serves as the core engine of the system, responsible for processing business logic, managing data persistence, and enabling communication between the user interface, the database, and the AI services. It is the component that ensures smooth request-response handling, transforming raw inputs into structured data that the front end can interpret and present to users. All communication between the client and the server follows the JSON format, allowing consistent, lightweight, and language-independent data exchange. This design facilitates seamless integration between the frontend and backend, enabling clear communication pathways across all system layers.

The backend was developed using NestJS with TypeScript, running in a Node.js environment. NestJS was selected because of its opinionated and modular architecture, which provides a well-defined structure that enforces good practices by default. The framework supports controllers, services, guards, interceptors, and decorators, all of which promote maintainability, scalability, and testability. TypeScript, being a statically typed superset of JavaScript, enhances code quality by introducing type safety and compile-time error detection. Together, they offer a

modern, strongly typed, and developer-friendly ecosystem that aligns with professional backend development standards.

At the early planning stage of the project, Java Spring Boot was initially considered for the backend implementation. Given that we had been extensively taught Java throughout our computer science curriculum, it appeared to be a natural choice. However, that same familiarity became the main reason to explore a different path. The project was seen not only as a system to build, but as an opportunity to learn and to expand beyond the confines of console-based Java and experience real-world development in a modern ecosystem. Moving towards JavaScript and TypeScript allowed us to acquire new technical skills, gain exposure to web technologies, and build a more diverse portfolio. In essence, the decision to use NestJS was both a technical and educational one: it challenged the team to grow and helped them appreciate that programming extends far beyond a single language or paradigm.

When compared to Express.js, another common Node.js framework, NestJS offered several crucial advantages that suited our goals and experience levels. Express is minimalistic and flexible, but that same freedom can easily lead to inconsistencies and technical debt when used by teams still developing their architectural understanding. Without enforced conventions, beginners often end up with large, unstructured files and tightly coupled logic. NestJS, on the other hand, is intentionally opinionated; it enforces structure through modules, controllers, and providers, and promotes dependency injection by design. This opinionated nature proved beneficial for collaboration, ensuring that even as the project grew, the codebase remained organized, readable, and aligned with best practices.

From a design perspective, the backend was built upon fundamental software engineering principles such as the SOLID principles and Clean Code guidelines. Each module and service was designed to have a single responsibility, with clear separation of concerns across the architecture. The use of dependency injection in NestJS made it easier to decouple components, increasing flexibility and testability. Throughout development, particular emphasis was placed on defensive programming; writing code that anticipates potential failures and handles them gracefully, rather than assuming ideal inputs. This approach reduced unexpected behavior, improved

reliability, and ensured that edge cases were properly managed. In addition, the backend was implemented following the philosophy of negative space programming, which emphasizes handling not only successful or “happy path” scenarios but also the valid edge cases that may naturally occur in real-world usage. Instead of assuming that every request will yield data, the system was designed to gracefully respond to situations where no matching records are found or a process simply has no content to return. For instance, when the frontend requests a resource that exists but currently has no associated data, the backend returns an empty array or a meaningful message rather than an error. This approach ensures that the system behaves predictably under all logical conditions, making the application more robust and user-friendly.

Altogether, these design philosophies created a backend that is modular, stable, and easy to maintain. NestJS’s structure ensured that every component from the smallest service to the largest module follows a consistent and intuitive pattern. This not only accelerated development but also improved collaboration and integration with the frontend and AI services. The next part of this section discusses how the backend exposes its functionality through a RESTful API design, and how tools like Swagger and Postman were used to ensure reliability and clarity in communication across the system.

5.2.2.1 RESTful API Design, Documentation, and Testing

The backend of the system was implemented following the RESTful API architectural style, which stands for Representational State Transfer. REST is not a framework or a library, but rather a design philosophy that defines how web services should be structured and communicate over HTTP. Its goal is to make systems simple, predictable, and scalable by following well-defined conventions.

In RESTful design, each piece of data or functionality in the system is represented as a resource, and each resource is accessible through a unique endpoint (URL).

Clients interact with these resources using standard HTTP methods such as:

- GET to retrieve data
- POST to create a new resource

- PUT/PATCH to update existing resources
- DELETE to remove resources

For example, the Products Module was designed as follows:

Table 5-1: Defined RESTful routes for Products Resource

Action	HTTP Method	Endpoint	Description
Retrieve all products	GET	/products	Returns a list of all products
Retrieve a single product	GET	/products/:id	Returns a specific product by its ID
Create a new product	POST	/products	Adds a new product
Update an existing product	PATCH	/products/:id	Updates product details
Delete a product	DELETE	/products/:id	Removes a product from the database

In this structure:

:id represents a path parameter, which identifies a specific resource within the collection. For instance, /products/5 retrieves the product whose ID is 5. Path parameters are part of the endpoint itself and are essential for locating individual records.

REST endpoints also support query parameters, which are used to filter, sort, or search through collections without creating additional routes. They are appended to the endpoint after a “?” symbol. For example:

GET /products?category=tools&lang=ar

This query retrieves the products in the “tools” category in the Arabic language (this involves using an external translation service). Query parameters make the API more flexible, enabling customized client requests without complicating the URL structure.

When sending data to the backend, request bodies are used, typically with the POST and PATCH methods. The backend supports two formats:

- **JSON body**, for sending structured data such as user credentials, product details, or cart information.
- **Form data**, for uploading files (such as service images) along with text fields, enabling multimedia content handling.

Additionally, standard HTTP status codes were consistently used to communicate request outcomes clearly:

- **200 OK** — The request was successful.
- **201 Created** — A new resource was successfully added.
- **400 Bad Request** — The request was invalid, often due to validation errors.
- **404 Not Found** — The requested resource does not exist.
- **500 Internal Server Error** — An unexpected error occurred on the server side.

This consistent structure and clear response handling not only improved maintainability but also made testing, debugging, and documentation (later performed through Swagger and Postman) much easier and more reliable.

While other API paradigms such as GraphQL or gRPC exist, REST was chosen for this project due to its simplicity, readability, and predictable structure. REST follows standard HTTP conventions, making it easy for team members to learn and debug, and it integrates seamlessly with modern JavaScript frontends through standard HTTP requests. Unlike GraphQL, which introduces additional query complexity, or gRPC, which requires binary communication and is more suitable for microservices, REST provides straightforward, reliable data flows for typical web applications. Furthermore, REST remains widely adopted in industry, benefiting from extensive community support and robust documentation [36] [37].

To ensure clear communication between the backend and frontend teams, the entire RESTful API was documented using Swagger UI. Swagger is an industry-standard tool that provides an interactive and automatically generated interface for API documentation. It is based on the OpenAPI Specification (OAS 3.0), which defines a standardized format for describing REST APIs.

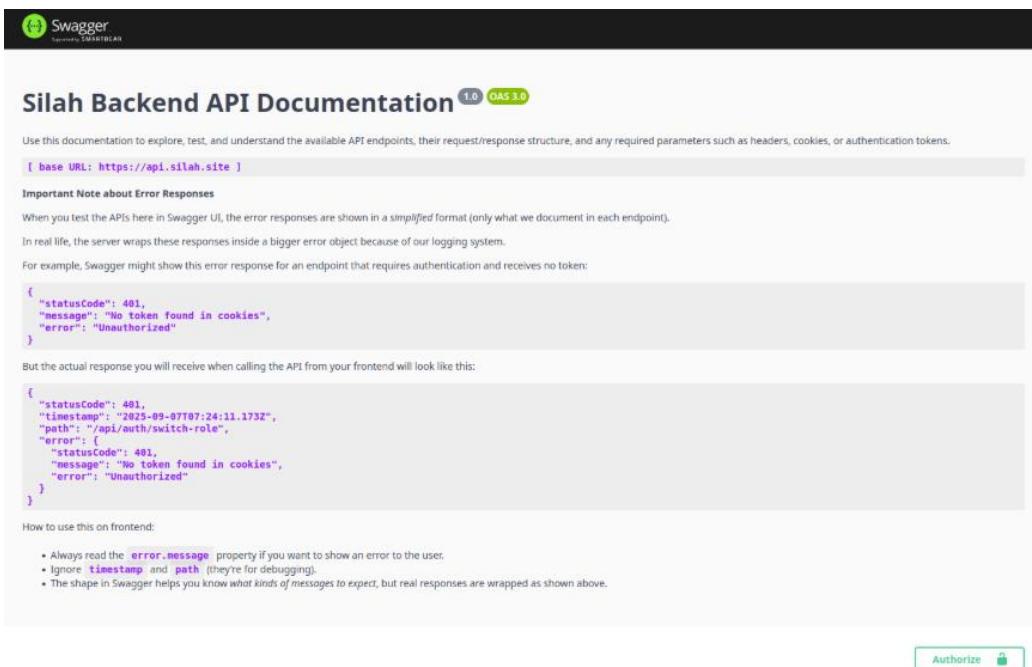
The Swagger documentation for our system is available online at:

<https://api.silah.site/api/docs>

This interactive interface allows developers to view, test, and understand each endpoint's purpose, required parameters, and expected responses without the need for external tools. Every time a new controller, route, or DTO (Data Transfer Object) is added, the corresponding documentation is automatically updated through custom decorators that describe each operation's summary, request body, and response format.

Although Swagger is described as “automated,” the automation comes from the code annotations and decorators that developers explicitly define. In this system, each module includes a separate .docs.ts file where the documentation details are written manually to ensure accuracy and readability.

The following figures show an example from the auth.docs.ts file, which defines the documentation for all authentication-related endpoints. This file specifies endpoint descriptions, request and response examples, and error messages using custom Swagger decorators. Together, these definitions generate the corresponding Swagger UI components automatically.



The screenshot shows the Swagger UI interface for the Silah Backend API. At the top, there is a navigation bar with the Silah logo and a search bar. Below the navigation bar, the title "Silah Backend API Documentation" is displayed, along with a "1.0" version indicator and a "GAS 3.0" badge. A sub-header "Use this documentation to explore, test, and understand the available API endpoints, their request/response structure, and any required parameters such as headers, cookies, or authentication tokens." follows. A note "[base URL: https://api.silah.site]" is present. A section titled "Important Note about Error Responses" explains that error responses are simplified in the UI but wrapped in a bigger object in reality. It provides an example of a simplified error response (just statusCode, message, and error) and a detailed real-life response (including timestamp, path, and nested error objects). A note at the bottom explains how to use this on frontend, listing three bullet points: Always read the `error.message` property if you want to show an error to the user, Ignore `timestamp` and `path` (they're for debugging), and The shape in Swagger helps you know what kinds of messages to expect, but real responses are wrapped as shown above. At the bottom right, there is a "Authorize" button with a lock icon.

Figure 5-30: Swagger UI — General API description and structure

The screenshot shows the Auth module endpoints in the Swagger UI. There are 12 entries listed:

- POST /api/auth/login**: Login user and send back JWT token as a cookie.
- POST /api/auth/logout**: Logout user by clearing token cookie.
- PATCH /api/auth/me/change-password**: Change user password (requires authentication). (Locked)
- POST /api/auth/request-password-reset**: Request password reset link.
- POST /api/auth/resend-verification-email**: Resend verification email to user.
- POST /api/auth/reset-password**: Reset user password using token.
- POST /api/auth/signup**: Registers a new user and returns a JWT token in a cookie.
- PATCH /api/auth/switch-role**: Switch user role (BUYER => SUPPLIER). (Locked)
- POST /api/auth/verify-email**: Verify user email using token.

Figure 5-31: Auth module endpoints displayed in Swagger UI

This screenshot shows the details for the **POST /api/auth/signup** endpoint. The description states: "Registers a new user and returns a JWT token. This endpoint validates the following:"

- Category IDs must exist and be main categories (no parentCategoryId).
- NID, CRN, and Email must be unique in the system.
- CRN must exist and be active in the official Wathq registry.
- On success, a JWT is returned (and stored as a cookie in the controller).

Parameters: No parameters

Request body (**required**): **application/json**

Example Value: Schema

```
{
  "email": "user@example.com",
  "password": "StrongPass123",
  "name": "John Doe",
  "crm": "1234567890",
  "businessName": "Acme Corp",
  "city": "Riyadh",
  "nid": "0987654321",
  "categories": [
    1,
    2,
    3
  ],
  "agreedToTerms": true,
  "preferredLanguage": "ar"
}
```

Figure 5-32: Signup endpoint description and request body example

Request body (**required**): **application/json**

Example Value: Schema

```
SignupDto {
  email*           string
  password*        string
  name*            string
  crm*             string
  businessName*   string
  city*            string
  nid*             string
  categories*     [string]
  agreedToTerms*  boolean
  preferredLanguage* string
}
```

SignupDto (Required):

- email***: string (example: user@example.com)
- password***: string (example: StrongPass123, minLength: 8, maxLength: 28)
Password must contain at least one uppercase, one lowercase, and one number. Special characters (@, #, !, \$) are optional.
- name***: string (example: John Doe)
- crm***: string (example: 1234567890)
- businessName***: string (example: Acme Corp)
- city***: string (example: Riyadh)
- nid***: string (example: 0987654321)
- categories***: [string]
List of main category IDs (must be a main category).
- agreedToTerms***: boolean (example: true)
- preferredLanguage***: string (example: ar)
Set user preferred language.

Enum:
[AR, EN]

Figure 5-33: Signup endpoint request body schema

Responses		
Code	Description	Links
201	User signed up successfully. JWT sent in cookie. Media type <input checked="" type="button"/> application/json Controls Accept header. Example Value Schema <pre>{"message": "Signup successful", "token": "<jwt_token>"}</pre>	No links
400	Bad Request - validation, duplication, or Wathq inactive CRN Media type Examples <input checked="" type="button"/> application/json <input type="button"/> Invalid Categories Example Value Schema <pre>{"statusCode": 400, "message": "These categories are invalid or not main categories: 33, 44", "error": "Bad Request"}</pre>	No links
404	Commercial Registration not found (from Wathq API) Media type Examples <input checked="" type="button"/> application/json <input type="button"/> Wathq returned no results Example Value Schema <pre>{"code": "404.2.1", "message": "No Results Found"}</pre>	No links
500	Unexpected internal error during signup or Wathq request Media type <input checked="" type="button"/> application/json Example Value Schema <pre>{"statusCode": 500, "message": "Failed to contact Wathq service", "error": "Internal Server Error"}</pre>	No links
504	Temporary issue with Wathq provider or connection failure Media type <input checked="" type="button"/> application/json Example Value Schema <pre>{"statusCode": 502, "message": "Temporary issue with provider", "error": "Bad Gateway"}</pre>	No links

Figure 5-34: Signup endpoint response body and error response examples

Below is the corresponding code snippet that defines this documentation:

```
ts auth.docs.ts x
src > auth > ts auth.docs.ts > ApiDocsSignUp
    You, 3 days ago | 1 author (You)
1   import { applyDecorators } from '@nestjs/common';
2   import {
3     ApiBadRequest,
4     ApiBody,
5     ApiGatewayTimeoutResponse,
6     ApiInternalServerErrorResponse,
7     ApiNotFoundResponse,
8     ApiOkResponse,
9     ApiOperation,
10    ApiQuery,
11    ApiResponse,
12  } from '@nestjs/swagger';
13  import { SignupDto } from './dtos/signup.dto';
14  import { LoginDto } from './dtos/login.dto';
15  import { ResetPasswordDto } from './dtos/resetPassword.dto';
16  import { ChangePasswordDto } from './dtos/changePassword.dto';
17  import { EmailDto } from './dtos/email.dto';
18  import { UserRole } from '@prisma/client';
19
20 export function ApiDocsSignUp() {
21   return applyDecorators(
22    ApiOperation({
23       summary: 'Registers a new user and returns a JWT token in a cookie',
24       description: `
25 Registers a new user and returns a JWT token.
26 This endpoint validates the following:
27 - Category IDs must exist and be main categories (no parentCategoryId).
28 - NID, CRN, and Email must be unique in the system.
29 - CRN must exist and be **active** in the official Wathq registry.
30 - On success, a JWT is returned (and stored as a cookie in the controller).`,
31     }),
32
33   ApiBody({ type: SignupDto }),
34
35   ApiResponse({
36     status: 201,
37     description: 'User signed up successfully. JWT sent in cookie.',
38     schema: {
39       example: {
40         message: 'Signup successful',
41         token: '<jwt_token>',
42       },
43     },
44   }),
45 }
```

Figure 5-35: Signup endpoint swagger documentation code (1/5)

```

19 // 400 - Validation / Conflict errors
20
21 ApiBadRequestResponse: {
22   description: 'Bad Request - validation, duplication, or Mathq inactive CRN',
23   content: {
24     'application/json': {
25       schema: {
26         type: 'object',
27         properties: {
28           errors: {
29             type: 'array',
30             items: {
31               example: {
32                 statusCode: 400,
33                 message: 'These categories are invalid or not main categories: 33, 44',
34                 error: 'Bad Request',
35               }
36             },
37           },
38         },
39       },
40     },
41   },
42   examples: {
43     'application/json': {
44       example: {
45         statusCode: 400,
46         message: 'CRN already exists',
47         error: 'Bad Request',
48       },
49     },
50   },
51   examples: {
52     'application/json': {
53       example: {
54         statusCode: 400,
55         message: 'Email already exists',
56         error: 'Bad Request',
57       },
58     },
59   },
60   examples: {
61     'application/json': {
62       example: {
63         statusCode: 400,
64         message: 'The provided CRN does not exist in Mathq records',
65         error: 'Bad Request',
66       },
67     },
68   },
69 }

```

Figure 5-36: Signup endpoint swagger documentation code (2/5)

```

10
11   examples: {
12     'application/json': {
13       example: {
14         statusCode: 400,
15         message: 'The CRN exists but is not active (status: suspended)',
16         error: 'Bad Request',
17       },
18     },
19   },
20   examples: {
21     'application/json': {
22       example: {
23         summary: 'Invalid categories',
24         value: {
25           statusCode: 400,
26           message: 'These categories are invalid or not main categories: 33, 44',
27           error: 'Bad Request',
28         },
29       },
30     },
31   },
32   examples: {
33     'application/json': {
34       example: {
35         summary: 'CRN already exists',
36         value: {
37           statusCode: 400,
38           message: 'CRN already exists',
39           error: 'Bad Request',
40         },
41       },
42     },
43   },
44   examples: {
45     'application/json': {
46       example: {
47         summary: 'Email already exists',
48         value: {
49           statusCode: 400,
50           message: 'Email already exists',
51           error: 'Bad Request',
52         },
53       },
54     },
55   },
56 }

```

Figure 5-37: Signup endpoint swagger documentation code (3/5)

```

125   crnNotFound: {
126     summary: 'CRN does not exist in Mathq',
127     value: {
128       statusCode: 400,
129       message: 'The provided CRN does not exist in Mathq records',
130       error: 'Bad Request',
131     },
132   },
133   crnInactive: {
134     summary: 'CRN exists but not active',
135     value: {
136       statusCode: 400,
137       message: 'The CRN exists but is not active (status: suspended)',
138       error: 'Bad Request',
139     },
140   },
141 },
142 },
143 },
144 },
145 },
146 },
147 },
148 },
149 },
150 },
151 },
152 },
153 },
154 },
155 },
156 },
157 },
158 },
159 },
160 },
161 },
162 },
163 },
164 },
165 },
166 },
167 },
168 },
169 },
170 },
171 },
172 },
173 },
174 },
175 },
176 },
177 },
178 },
179 },
180 },
181 },
182 },
183 },
184 },
185 },
186 },
187 },
188 },
189 },
190 },
191 },
192 },
193 },
194 },
195 },
196 },
197 },
198 },
199 },
200 },
201 },
202 },
203 },
204 },
205 },
206 },
207 },
208 },
209 },
210 },
211 },
212 },
213 },
214 },
215 },
216 }

```

Figure 5-38: Signup endpoint swagger documentation code (4/5)

```

182   // 502 - Provider down or Mathq internal errors
183   ApiGatewayTimeoutResponse: {
184     description: 'Temporary issue with Mathq provider or connection failure',
185     content: {
186       'application/json': {
187         schema: {
188           example: {
189             statusCode: 502,
190             message: 'Temporary issue with provider',
191             error: 'Bad Gateway',
192           }
193         },
194       },
195     },
196   },
197 },
198 },
199 },
200   // 500 - Unexpected backend failure
201   ApiInternalServerError: {
202     description: 'Unexpected internal error during signup or Mathq request',
203     content: {
204       'application/json': {
205         schema: {
206           example: {
207             statusCode: 500,
208             message: 'Failed to contact Mathq service',
209             error: 'Internal Server Error',
210           }
211         },
212       },
213     },
214   },
215 },
216 }

```

Figure 5-39: Signup endpoint swagger documentation code (5/5)

This modular documentation approach makes it easy to keep the API documentation synchronized with the source code, although it still requires manual updates and verification whenever the implementation changes.

During development, Swagger UI was primarily used for understanding and verifying endpoint structure rather than for direct testing. It allowed developers to quickly review each endpoint's request format, required parameters, and expected responses without having to open the source code. This made it easier to prepare accurate test cases and validate logic using external tools such as Postman.

To complement Swagger documentation, all backend endpoints were rigorously tested using Postman, a dedicated API testing platform. Postman provides a graphical

interface that allows developers to simulate real frontend requests and observe detailed responses from the backend. It supports handling headers, cookies, JWT authentication tokens, request bodies, and query parameters, making it ideal for validating business logic before frontend integration.

Postman was essential in ensuring that endpoints behaved as expected under different conditions; successful, invalid, and edge cases.

The team used the shared workspace available at <https://gp-silah.postman.co/>

Each collection in Postman was organized to mirror the NestJS module structure, where every module corresponds to a REST resource (for example, Auth, Products, Orders, Users, and so on). This one-to-one mapping made the testing process intuitive and consistent with the backend's architecture. Within each collection, predefined environments (such as base URLs) were configured to ensure that all team members could test endpoints under the same conditions and share results efficiently.

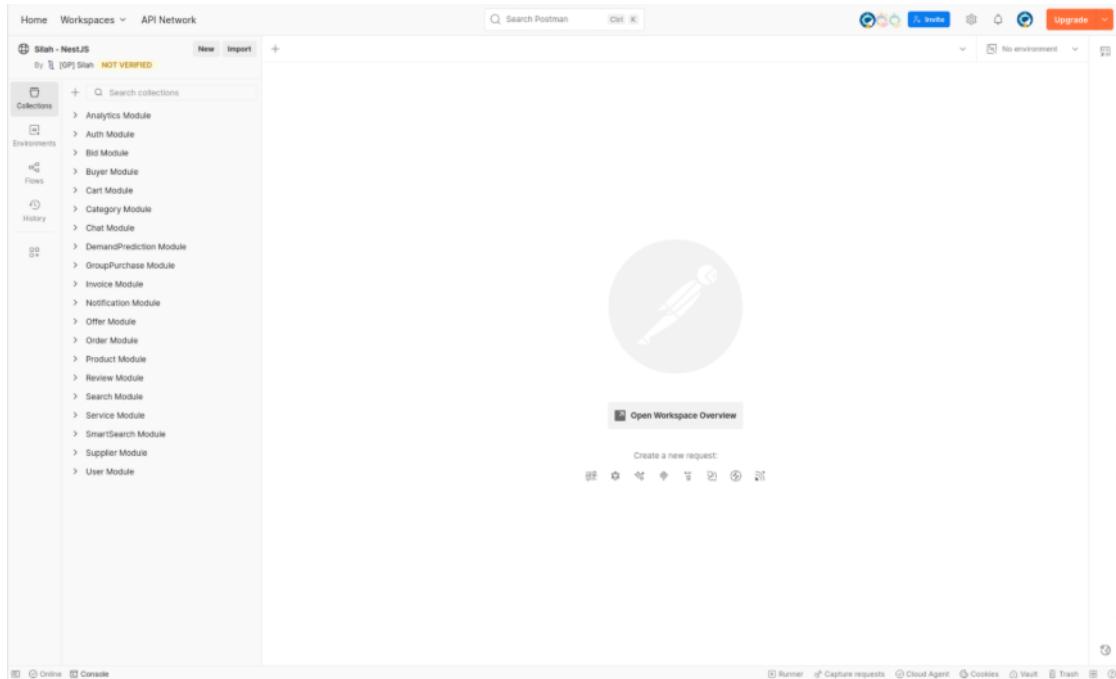


Figure 5-40: Postman workspace showing the main project collections

The Auth collection served as one of the most frequently tested modules. The following figures illustrate the testing workflow for the /api/auth/signup endpoint, which registers new users and returns a JWT cookie upon success.

The screenshot shows a POST request to `((baseURL))/api/auth/signup`. The request body is a JSON object containing user registration details. The response status is 201 Created, with a response time of 2.12 s and a size of 823 B. The response body is a JSON object with a single key "message": "Signup successful".

```

1 {
2   "email": "testing5@gmail.com",
3   "password": "Tt123456",
4   "name": "Tester Mr",
5   "crn": "10108000000",
6   "businessName": "TEST",
7   "city": "Riyadh",
8   "nid": "1634049222",
9   "categories": [],
10  "agreedToTerms": true,
11  "preferredLanguage": "EN"
12 }

```

```

{
  "message": "Signup successful"
}

```

Figure 5-41: Successful signup request example with a 201 Created response

The screenshot shows a POST request to `((baseURL))/api/auth/signup`. The request body is identical to the successful request in Figure 5-41. The response status is 400 Bad Request, with a response time of 573 ms and a size of 563 B. The response body is a JSON object containing error details, including a timestamp, path, error message, and status code.

```

1 {
2   "email": "testing6@gmail.com",
3   "password": "Tt123456",
4   "name": "Tester Mc",
5   "crn": "7001234567",
6   "businessName": "TEST",
7   "city": "Riyadh",
8   "nid": "1634049262",
9   "categories": [],
10  "agreedToTerms": true,
11  "preferredLanguage": "EN"
12 }

```

```

{
  "statusCode": 400,
  "timestamp": "2025-10-31T20:11:32.048Z",
  "path": "/api/auth/signup",
  "error": {
    "message": "The provided CRN does not exist in Wathq records",
    "error": "Bad Request",
    "statusCode": 400
  }
}

```

Figure 5-42: Failed signup request example showing 400 (Bad Request) error response

These tests verified not only that the endpoint logic worked as expected, but also that error handling, status codes, and response structures aligned with the Swagger documentation.

Together, Swagger and Postman formed a reliable workflow. Swagger for documentation and communication, Postman for validation and debugging; ensuring the backend was both transparent and dependable.

5.2.2.2 Project Structure and Application Logic

Our backend follows a modular NestJS architecture, where each feature (or “module”) is self-contained and organized under the `src/` directory. To help readers navigate the project, the following screenshots provide a quick visual overview of our folder structure and key files.

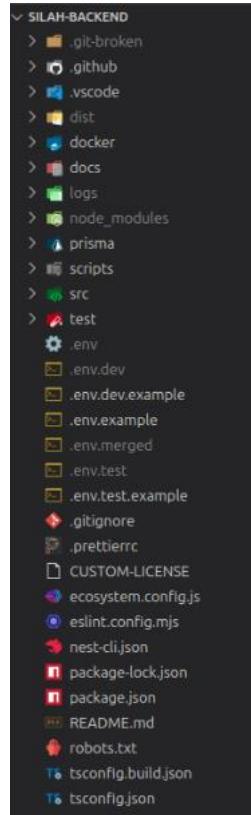


Figure 5-43: NestJS Project root structure

At the root level, we keep configuration, build, and infrastructure files:

- **package.json** — defines dependencies, scripts, and metadata.
- **tsconfig.json** — configures TypeScript compilation.
- **docker/** — contains Docker setup and scripts.
- **prisma/** — manages database schema, migrations, and seeding.
- **src/** — the main application source code (where all business logic lives).
- **test/** — organized test files mirroring the structure of `src/`.

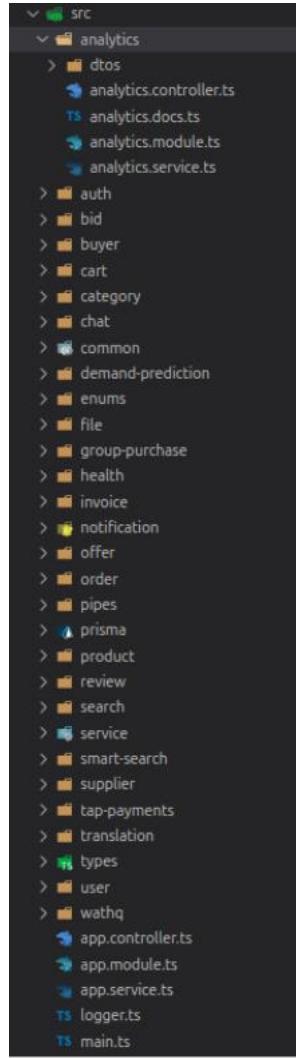


Figure 5-44: Inside the NestJS src/ directory

Each folder under src/ represents a NestJS module, which maps directly to a REST resource. This organization ensures that related files (controllers, services, and documentation) are grouped together for cleaner development and faster navigation. For example, the analytics/ module contains:

- `analytics.controller.ts` — handles incoming requests.
- `analytics.service.ts` — contains analytics logic.
- `analytics.docs.ts` — defines Swagger documentation for the Analytics API.
- `analytics.module.ts` — registers all the above within NestJS.

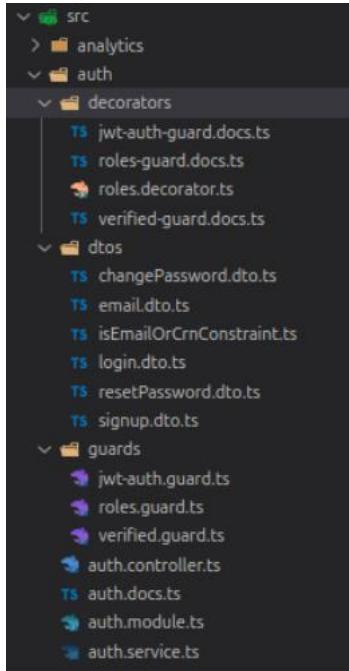


Figure 5-45: Example module (auth/)

The auth module is a representative example of this modular design. In addition to the standard controller and service files, it also includes:

- **dtos/** — defines data transfer objects for input validation.
- **guards/** — implements authentication and authorization logic.

This consistent structure allows every feature to be developed, tested, and maintained independently, reducing coupling and improving scalability as the system grows.

The Bid Module is a clear example of how each feature in our backend follows NestJS's modular architecture pattern. It demonstrates the complete flow of a RESTful resource, from HTTP requests handled by the controller to business logic in the service layer, and data validation via DTOs.

Each module encapsulates its own functionality and dependencies. The BidModule imports the BuyerModule (since bids are created by buyers), declares its controller and services, and exports the BidService for use in other modules (the OfferModule specifically).

```

src > bid > bid.module.ts
You, 2 weeks ago | 1 author (You)
1 import { Module } from '@nestjs/common';
2 import { BidService } from './bid.service';
3 import { BidController } from './bid.controller';
4 import { BuyerModule } from 'src/buyer/buyer.module';
5 import { BidCronService } from './bid-cron.service';
6
7 @Module({
8   imports: [BuyerModule],
9   controllers: [BidController],
10  providers: [BidService, BidCronService],
11  exports: [BidService],
12 })
13 export class BidModule {}

```

Figure 5-46: BidModule

The controller defines all the RESTful endpoints related to bids. Each route is clearly mapped to an HTTP method and path, and uses decorators to handle authentication, role restrictions, and Swagger documentation.

This layer handles routing (e.g., GET /bids, POST /bids), applies authentication and authorization through guards, validates request bodies using DTOs, and delegates the core business logic to the service layer.

```

src > bid > bid.controllers.ts
You, 2 weeks ago | 1 author (You)
1 import {
2   Body,
3   Controller,
4   Get,
5   Param,
6   Post,
7   Req,
8   UseGuards,
9 } from '@nestjs/common';
10 import { BidService } from './bid.service';
11 import { ApiTags } from '@nestjs/swagger';
12 import { ApiDocsJwtAuthGuard } from 'src/auth/decorators/jwt-auth.guard.docs';
13 import { ApiDocsRolesGuard } from 'src/auth/decorators/roles.guard.docs';
14 import { ApiDocsVerifiedGuard } from 'src/auth/decorators/verified.guard.docs';
15 import { Roles } from 'src/auth/decorators/roles.decorator';
16 import { UserRole } from 'src/enums/userRole.enum';
17 import { JwtAuthGuard } from 'src/auth/guards/jwt-auth.guard';
18 import { RolesGuard } from 'src/auth/guards/roles.guard';
19 import { VerifiedGuard } from 'src/auth/guards/verified.guard';
20 import { Request } from 'express';
21 import { CreateBidDto } from './dtos/createBid.dto';
22 import {
23   ApiDocsCreateBid,
24   ApiDocsGetAllBids,
25   ApiDocsGetBidById,
26   ApiDocsGetBidsJoined,
27   ApiDocsGetMyBids,
28 } from './bid.docs';
29
30 You, 2 weeks ago | 1 author (You)
31 @ApiTags('Bids')
32 @Controller('bids')
33 export class BidController {
34   constructor(private readonly bidService: BidService) {}
35
36   @Get()
37   @ApiDocsGetAllBids()
38   async getAllBids() {
39     return this.bidService.getAllBids();
40   }
41
42   @Get(':id')
43   @ApiDocsGetBidById()
44   async getBidById(@Param('id') bidId: string) {
45     return this.bidService.getBidById(bidId);
46   }
47
48   @ApiDocsJwtAuthGuard()
49   @ApiDocsRolesGuard()
50   @ApiDocsVerifiedGuard()
51   @Roles(UserRole.BUYER)
52   @UseGuards(JwtAuthGuard, RolesGuard, VerifiedGuard)
53   @Get('created/me')
54   @ApiDocsGetMyBids()
55   async getMyBids(@Req() req: Request) {
56     const userId = req.tokenData.sub;
57     return this.bidService.getMyBids(userId);
58   }
59
60   @ApiDocsJwtAuthGuard()
61   @ApiDocsRolesGuard()
62   @ApiDocsVerifiedGuard()
63   @Roles(UserRole.BUYER)
64   @UseGuards(JwtAuthGuard, RolesGuard, VerifiedGuard)
65   @Post()
66   @ApiDocsCreateBid()
67   async createBid(@Req() req: Request, @Body() dto: CreateBidDto) {
68     const userId = req.tokenData.sub;
69     return this.bidService.createBid(userId, dto);
70   }
71
72   @ApiDocsJwtAuthGuard()
73   @ApiDocsRolesGuard()
74   @ApiDocsVerifiedGuard()
75   @Roles(UserRole.SUPPLIER)
76   @UseGuards(JwtAuthGuard, RolesGuard, VerifiedGuard)
77   @Get('joined/me')
78   @ApiDocsGetBidsJoined()
79   async getBidsJoined(@Req() req: Request) {
80     const userId = req.tokenData.sub;
81     return this.bidService.getBidsJoined(userId);
82   }
83

```

Figure 5-47: BidController (1/2)

Figure 5-48: BidController (2/2)

Each endpoint is decorated with custom Swagger decorators (@ApiDocs GetAllBids) to automatically generate documentation in Swagger UI.

The service layer contains the main business logic of the feature. It interacts with the Prisma ORM for database access and with other services (e.g., BuyerService) when needed. This layer is responsible for fetching and manipulating data in the

database, handling domain-specific logic (such as ensuring only buyers can create bids), and returning consistent, typed responses using BidResponseDto.

```

1 import { Injectable, NotFoundException } from '@nestjs/common';
2 import { BuyerService } from 'src/buyer/buyer.service';
3 import { PrismaService } from 'src/prisma/prisma.service';
4 import { CreateBidDto } from './dtos/createBid.dto';
5 import {
6   Bid,
7   BidExpectedResponseTime,
8   BidStatus,
9   Buyer,
10  Card,
11  User,
12 } from '@prisma/client';
13 import { BidResponseDto } from './dtos/bidResponse.dto';
14 import { BuyerResponseDto } from 'src/buyer/dtos/buyerResponse.dto';
15
16 You, 16 seconds ago | author (You)
17 @Injectable()
18 export class BidService {
19   constructor(
20     private readonly prisma: PrismaService,
21     private readonly buyerService: BuyerService,
22   ) {}
23
24   /**
25    * Converts a Prisma Bid entity into a BidResponseDto.
26    * Includes nested buyer data (converted to BuyerResponseDto).
27    */
28   async toBidResponseDto(
29     bid: any & { buyer: Buyer & { user: User; card: Card } },
30   ): Promise<BidResponseDto> {
31     const buyerDto: BuyerResponseDto = await this.buyerService.toBuyerResponseDto(
32       bid.buyer.user,
33       bid.buyer,
34     );
35
36     return {
37       bidId: bid.id,
38       bidName: bid.bidName,
39       mainActivity: bid.mainActivity,
40       submissionDeadline: bid.submissionDeadline,
41       expectedResponseTime:
42         bid.expectedResponseTime as BidExpectedResponseTime,
43       status: bid.status as BidStatus,
44       buyer: buyerDto,
45       createdAt: bid.createdAt,
46     };
47   }
48

```

Figure 5-49: BidService (1/3)

```

49   async getAllBids(): Promise<BidResponseDto[]> {
50     const bids = await this.prisma.bid.findMany({
51       where: {
52         submissionDeadline: {
53           gt: new Date(), // greater than now - deadline hasn't passed
54         },
55       },
56       include: { buyer: { include: { user: true, card: true } } },
57     });
58     return Promise.all(bids.map((b) => this.toBidResponseDto(b)));
59   }
60
61   async getBidById(bidId: string): Promise<BidResponseDto> {
62     const bid = await this.prisma.bid.findFirst({
63       where: { id: bidId },
64       include: { buyer: { include: { user: true, card: true } } },
65     });
66     if (!bid) {
67       throw new NotFoundException(`Bid with ID ${bidId} not found`);
68     }
69     return this.toBidResponseDto(bid);
70   }
71
72   async getMyBids(userId: string): Promise<BidResponseDto[]> {
73     const buyer = await this.prisma.buyer.findFirst({
74       where: { userId },
75     });
76     if (!buyer) {
77       throw new Error('Buyer not found');
78     }
79     const bids = await this.prisma.bid.findMany({
80       where: { buyerId: buyer.id },
81       include: { buyer: { include: { user: true, card: true } } },
82       orderBy: { createdAt: 'desc' },
83     });
84     return Promise.all(bids.map((bid) => this.toBidResponseDto(bid)));
85   }
86

```

Figure 5-50: BidService (2/3)

```

87   async createBid(
88     userId: string,
89     dto: CreateBidDto,
90   ): Promise<BidResponseDto> {
91     const buyer = await this.prisma.buyer.findFirst({
92       where: { userId },
93     });
94     if (!buyer) {
95       throw new Error('Buyer not found');
96     }
97     const bid = await this.prisma.bid.create({
98       data: [
99         {
100           buyerId: buyer.id,
101           bidName: dto.bidName,
102           mainActivity: dto.mainActivity,
103           submissionDeadline: new Date(dto.submissionDeadline),
104           expectedResponseTime: dto.expectedResponseTime,
105         },
106       ],
107       include: { buyer: { include: { user: true, card: true } } },
108     });
109     return this.toBidResponseDto(bid);
110   }
111
112   async getBidsJoined(userId: string): Promise<BidResponseDto[]> {
113     const supplier = await this.prisma.supplier.findFirst({
114       where: { userId },
115     });
116     if (!supplier) throw new NotFoundException('Supplier not found');
117     const bids = await this.prisma.bid.findMany({
118       where: {
119         offers: { some: { supplierId: supplier.id } },
120       },
121       include: { buyer: { include: { user: true, card: true } } },
122       orderBy: { createdAt: 'desc' },
123     });
124     return Promise.all(bids.map((b) => this.toBidResponseDto(b)));
125   }
126

```

Figure 5-51: BidService (3/3)

DTOs define how data is validated (for incoming requests) and shaped (for outgoing responses).

- CreateBidDto ensures that every bid creation request includes a valid name, activity, and future submission deadline.
- BidResponseDto defines the structure returned to clients when fetching bids, ensuring consistency in the API output.

```

src > bid > dtos > createBid.dto.ts
  You 2 weeks ago | author (You)
  1 import { ApiProperty } from '@nestjs/swagger';
  2 import { BidExpectedResponseTime } from '@prisma/client';
  3 import {
  4   IsString,
  5   IsNotEmpty,
  6   IsDateString,
  7   IsEnum,
  8   MaxLength,
  9 } from 'class-validator';
 10 import { IsFutureDate } from 'src/invoice/dtos/createInvoice.dto';
 11
 12 You 2 weeks ago | author (You)
 13 export class CreateBidDto {
 14   @ApiProperty({
 15     description: 'Name of the bid',
 16     example: 'IT Hardware Supply',
 17     maxLength: 100,
 18   })
 19   @IsString()
 20   @IsNotEmpty()
 21   @MaxLength(100, { message: 'Bid name must not exceed 100 characters' })
 22   bidName: string;
 23
 24   @ApiProperty({
 25     description: 'Main activity or purpose of the bid',
 26     example: 'Supplying laptops and accessories for company use',
 27     maxLength: 500,
 28   })
 29   @IsString()
 30   @IsNotEmpty()
 31   @MaxLength(500, { message: 'Main activity must not exceed 500 characters' })
 32   mainActivity: string;
 33
 34   @ApiProperty({
 35     description: 'Deadline for suppliers to submit offers (must be a future date)',
 36     example: '2025-11-30',
 37   })
 38   @IsDateString()
 39   @IsFutureDate({ message: 'Submission deadline must be in the future' })
 40   submissionDeadline: Date;
 41
 42   @ApiProperty({
 43     description: 'Expected response time',
 44     enum: BidExpectedResponseTime,
 45     example: BidExpectedResponseTime.ONE_WEEK,
 46   })
 47   @IsEnum(BidExpectedResponseTime)
 48   expectedResponseTime: BidExpectedResponseTime;
 49 }

```

Figure 5-52: CreateBidDto

```

src > bid > dtos > bidResponse.dto.ts
  You 2 weeks ago | author (You)
  1 import { ApiProperty } from '@nestjs/swagger';
  2 import { BidExpectedResponseTime, BidStatus } from '@prisma/client';
  3 import { BuyerResponseDto } from 'src/buyer/dtos/buyerResponse.dto';
  4
  5 You 2 weeks ago | author (You)
  6 export class BidResponseDto {
  7   @ApiProperty({
  8     description: 'Unique identifier of the bid',
  9     example: 'b5fa7c48-9c12-4a5f-a421-57dcf621d932',
 10   })
 11   bidId: string;
 12
 13   @ApiProperty({
 14     description: 'Name of the bid',
 15     example: 'Construction Equipment Supply',
 16   })
 17   bidName: string;
 18
 19   @ApiProperty({
 20     description: 'The main activity or subject of the bid',
 21     example: 'Supplying construction materials and heavy machinery',
 22   })
 23   mainActivity: string;
 24
 25   @ApiProperty({
 26     description: 'Deadline for submitting offers for this bid',
 27     example: '2025-12-31T23:59:59Z',
 28     type: String,
 29     format: 'date-time',
 30   })
 31   submissionDeadline: Date;
 32
 33   @ApiProperty({
 34     description: 'Expected response time from suppliers',
 35     enum: BidExpectedResponseTime,
 36     example: BidExpectedResponseTime.ONE_WEEK,
 37   })
 38   expectedResponseTime: BidExpectedResponseTime;
 39
 40   @ApiProperty({
 41     description: 'Current status of the bid',
 42     enum: BidStatus,
 43     example: BidStatus.OPEN,
 44   })
 45   status: BidStatus;
 46
 47   @ApiProperty({
 48     description: 'Information about the buyer who created this bid',
 49     type: () => BuyerResponseDto,
 50   })
 51   buyer: BuyerResponseDto;
 52
 53   @ApiProperty({
 54     description: 'Date and time when this bid was created',
 55     example: '2025-10-17T10:00:00Z',
 56     type: String,
 57     format: 'date-time',
 58   })
 59   createdAt: Date;

```

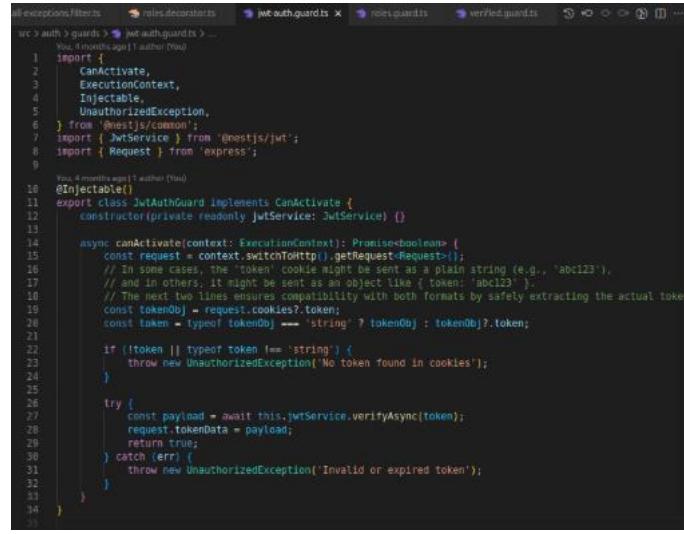
Figure 5-53: BidResponseDto

Beyond individual modules and their business logic, the application also implements several global mechanisms that enforce access control and ensure security at the request level. These mechanisms operate before the controller or service logic is executed, determining whether a request should even reach the underlying business code. In Silah, this is primarily achieved through a structured use of NestJS guards.

Guards in NestJS determine whether a given request can proceed to the route handler. They are heavily used in Silah for authentication, authorization, and account verification.

a. JwtAuthGuard

Ensures that incoming requests contain a valid JWT stored in cookies. It extracts and verifies the token using the JwtService, attaches the decoded payload to the request, and blocks access if the token is missing or invalid. Its purpose is to protect all routes that require user authentication.

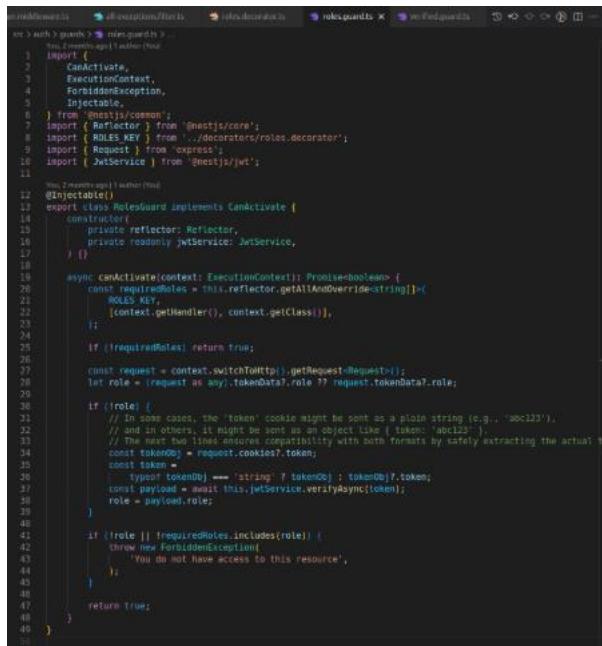


```
src > auth > guards > JwtAuthGuard.ts | roles.decorators | jwtauth.guard.ts | rolesguards | verifiedguard.ts ...  
src > auth > guards > JwtAuthGuard.ts  
  <-- AuthorGuard (Mu)  
  1 import {  
  2   CanActivate,  
  3   ExecutionContext,  
  4   Injectable,  
  5   UnauthorizedException,  
  6 } from '@nestjs/common';  
  7 import { JwtService } from '@nestjs/jwt';  
  8 import { Request } from 'express';  
  9  
  You, 4 months ago | Author (Mu)  
10 @Injectable()  
11 export class JwtAuthGuard implements CanActivate {  
12   constructor(private readonly jwtService: JwtService) {}  
13  
14   async canActivate(context: ExecutionContext): Promise<boolean> {  
15     const request = context.switchToHttp().getRequest<Request>();  
16     // In some cases, the 'token' cookie might be sent as a plain string (e.g., 'abc123'),  
17     // and in others, it might be sent as an object like { token: 'abc123' }.  
18     // The next two lines ensures compatibility with both formats by safely extracting the actual token.  
19     const tokenObj = request.cookies?.token;  
20     const token = typeof tokenObj === 'string' ? tokenObj : tokenObj?.token;  
21  
22     if (!token || typeof token !== 'string') {  
23       throw new UnauthorizedException('No token found in cookies');  
24     }  
25  
26     try {  
27       const payload = await this.jwtService.verifyAsync(token);  
28       request.userData = payload;  
29       return true;  
30     } catch (err) {  
31       throw new UnauthorizedException('Invalid or expired token');  
32     }  
33   }  
34 }  
35
```

Figure 5-54: JwtAuthGuard

b. RolesGuard

Works with the custom `@Roles()` decorator to enforce role-based access control (RBAC). It reads the required roles from route metadata (set by `SetMetadata`) and compares them with the user's role in the token payload. Its purpose is to restrict sensitive endpoints (e.g., supplier-only or buyer-only routes).

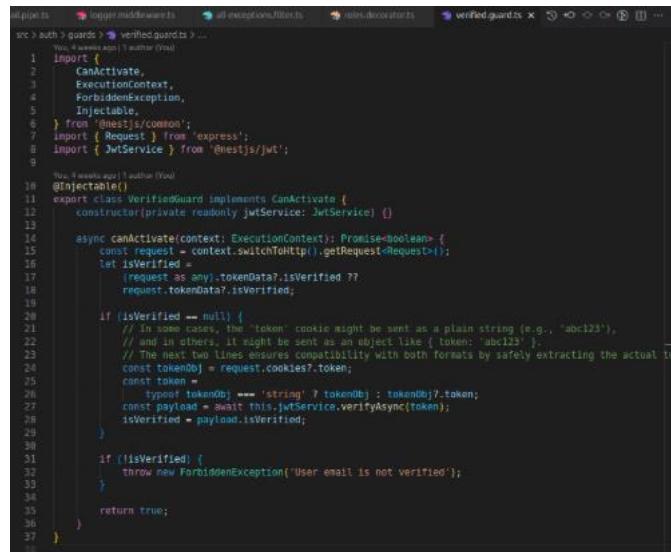


```
src > auth > guards > RolesGuard.ts | roles.decorators | jwtauth.guard.ts | rolesguards | verifiedguard.ts ...  
src > auth > guards > RolesGuard.ts  
  <-- Author (Mu)  
  1 import {  
  2   CanActivate,  
  3   ExecutionContext,  
  4   ForbiddenException,  
  5   Injectable,  
  6 } from '@nestjs/common';  
  7 import { Reflector } from '@nestjs/core';  
  8 import { ROLES_KEY } from '../decorators/roles.decorator';  
  9 import { Request } from 'express';  
10 import { JwtService } from '@nestjs/jwt';  
11  
  You, 4 months ago | Author (Mu)  
12 @Injectable()  
13 export class RolesGuard implements CanActivate {  
14   constructor(  
15     private reflector: Reflector,  
16     private readonly jwtService: JwtService,  
17   ) {}  
18  
19   async canActivate(context: ExecutionContext): Promise<boolean> {  
20     const requiredRoles = this.reflector.getAllAndOverride<string>([  
21       ROLES_KEY,  
22       [context.getHandler(), context.getClass()],  
23     ]);  
24  
25     if (!requiredRoles) return true;  
26  
27     const request = context.switchToHttp().getRequest<Request>();  
28     let role = (request as any).userData?.role ?? request.userData?.role;  
29  
30     if (!role) {  
31       // In some cases, the 'token' cookie might be sent as a plain string (e.g., 'abc123'),  
32       // and in others, it might be sent as an object like { token: 'abc123' }.  
33       // The next two lines ensures compatibility with both formats by safely extracting the actual token.  
34       const tokenObj = request.cookies?.token;  
35       const token = typeof tokenObj === 'string' ? tokenObj : tokenObj?.token;  
36       const payload = await this.jwtService.verifyAsync(token);  
37       role = payload.role;  
38     }  
39  
40     if (role) {  
41       if (requiredRoles.includes(role)) {  
42         throw new ForbiddenException(  
43           `You do not have access to this resource`  
44         );  
45       }  
46     }  
47     return true;  
48   }  
49 }  
50
```

Figure 5-55: RolesGuard

c. VerifiedGuard

Ensures that a user has verified their email before performing restricted actions (e.g., posting listings or orders). It reads the isVerified flag from the JWT payload and denies access otherwise. Its purpose is to enforce email verification for better platform integrity.



```

src > auth > guards > verified.guard.ts ...
You, 4 months ago | author (You)
1 import { Injectable, CanActivate, ExecutionContext, ForbiddenException } from '@nestjs/common';
2 import { Request } from 'express';
3 import { JwtService } from '@nestjs/jwt';
4
5 @Injectable()
6 export class VerifiedGuard implements CanActivate {
7   constructor(private readonly jwtService: JwtService) {}
8
9   async canActivate(context: ExecutionContext): Promise<boolean> {
10     const request = context.switchToHttp().getRequest<Request>();
11     let isVerified =
12       (request as any).tokenData?.isVerified ??
13       request.tokenData?.isVerified;
14
15     if (isVerified === null) {
16       // In some cases, the "token" cookie might be sent as a plain string (e.g., "abc123"),
17       // and in others, it might be sent as an object like { token: "abc123" };
18       // The next two lines ensures compatibility with both formats by safely extracting the actual token
19       const tokenObj = request.cookies?.token;
20       const token = typeof tokenObj === 'string' ? tokenObj : tokenObj?.token;
21       const payload = await this.jwtService.verifyAsync(token);
22       isVerified = payload.isVerified;
23     }
24
25     if (!isVerified) {
26       throw new ForbiddenException('User email is not verified');
27     }
28
29     return true;
30   }
31 }
32
33
34
35
36
37

```

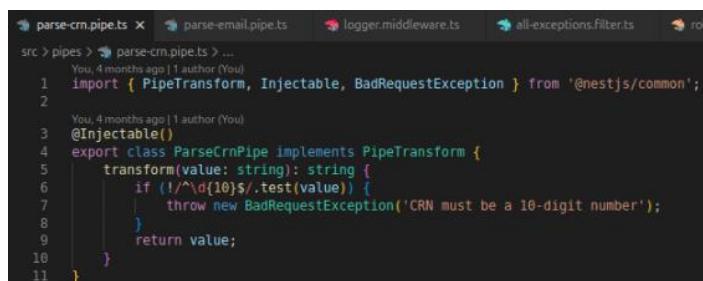
Figure 5-56: VerifiedGuard

While guards focus on controlling access and protecting routes, another important layer of the application focuses on data integrity; ensuring that any information entering the system is valid and properly formatted. This is handled by pipes, which validate and transform incoming data before it reaches the controller, providing early and reliable protection against invalid or malformed inputs.

a. ParseCrnPipe

It validates that a given CRN (Commercial Registration Number) is a 10-digit number.

If validation fails, it throws a BadRequestException.



```

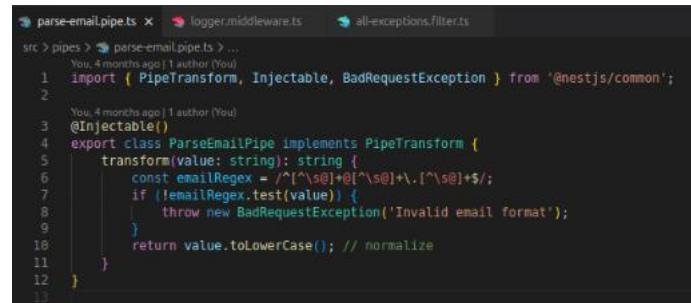
src > pipes > parse-crn.pipe.ts ...
You, 4 months ago | author (You)
1 import { PipeTransform, Injectable, BadRequestException } from '@nestjs/common';
2
3 @Injectable()
4 export class ParseCrnPipe implements PipeTransform {
5   transform(value: string): string {
6     if (!/^{\d{10}}/.test(value)) {
7       throw new BadRequestException('CRN must be a 10-digit number');
8     }
9     return value;
10   }
11 }

```

Figure 5-57: ParseCrnPipe

b. ParseEmailPipe

Ensures that an email parameter is in a valid format and automatically converts it to lowercase for consistency. If the format is invalid, it throws a BadRequestException.



```
parse-email.pipe.ts  logger.middleware.ts  all-exceptions.filter.ts
src > pipes > parse-email.pipe.ts > ...
You, 4 months ago | author (You)
1 import { PipeTransform, Injectable, BadRequestException } from '@nestjs/common';
2
3 @Injectable()
4 export class ParseEmailPipe implements PipeTransform {
5   transform(value: string): string {
6     const emailRegex = /^[^\s]+@[^\s]+\.[^\s]+$/;
7     if (!emailRegex.test(value)) {
8       throw new BadRequestException('Invalid email format');
9     }
10    return value.toLowerCase(); // normalize
11  }
12}
13
```

Figure 5-58: ParseEmailPipe

Even with validation and access control in place, errors are inevitable in any backend system. To handle them gracefully and consistently, the application uses exception filters, which act as a centralized error-handling layer. These filters capture unexpected errors, log them, and ensure the client receives clear, standardized responses regardless of where the error originated.

AllExceptionsFilter

A global exception filter applied across the entire application. It captures all thrown errors (both `HttpException` and unhandled exceptions), logs them using the internal logger, and returns a standardized JSON response to the client. Its purpose is to maintain a consistent error format across all endpoints.

```

src > common > filters > all-exceptions.filter.ts > AllExceptionsFilter > catch
You, 2 months ago | author (You)
1 import (
2   ExceptionFilter,
3   Catch,
4   ArgumentsHost,
5   HttpException,
6   HttpStatus,
7 ) from '@nestjs/common';
8 import { Request, Response } from 'express';
9 import logger from '../../../../../logger';
10
11 @Catch()
12 export class AllExceptionsFilter implements ExceptionFilter {
13   catch(exception: unknown, host: ArgumentsHost) {
14     const ctx = host.switchToHttp();
15     const response = ctx.getResponse<Response>();
16     const request = ctx.getRequest<Request>();
17
18     const isHttp = exception instanceof HttpException;
19     const status = isHttp
20       ? exception.getStatus()
21       : HttpStatus.INTERNAL_SERVER_ERROR;
22
23     // Normalize client-facing payload
24     const responseError = isHttp
25       ? () => {
26         const res = exception.getResponse();
27         // If response is string, wrap it in { message }
28         return typeof res === 'string' ? { message: res } : res;
29       }()
30       : { message: 'Internal server error' };
31
32     // Log full details internally
33     if (!isHttp && exception instanceof Error) {
34       logger.error(
35         `${request.method} ${request.url} ${status} - ${exception.message}\n${exception.stack}`,
36       );
37     } else {
38       logger.error(
39         `${request.method} ${request.url} ${status} - ${JSON.stringify(responseError)}`,
40       );
41     }
42
43     // Send normalized error to client
44     response.status(status).json({
45       statusCode: status,
46       timestamp: new Date().toISOString(),
47       path: request.url,
48       error: responseError,
49     });
50   }
51 }

```

Figure 5-59: AllExceptionsFilter

This helps the frontend parse and display error messages uniformly.

Before any of these request-level mechanisms take effect, middleware plays a role earlier in the pipeline. Middleware functions execute as soon as a request enters the server, making them ideal for logging, request preprocessing, or metrics collection. In Silah, a lightweight logging middleware captures essential request data for debugging and monitoring purposes.

LoggerMiddleware

It executes on every incoming request, logging the HTTP method and endpoint path using the centralized logger utility. Its purpose is to provide transparent request-level logging for debugging and performance tracking.

```

logger.middleware.ts ×
src > common > middleware > logger.middleware.ts > ...
You, 2 months ago | 1 author (You)
1 import { Injectable, NestMiddleware } from '@nestjs/common';
2 import { Request, Response, NextFunction } from 'express';
3 import logger from '../../logger';
4
5 You, 2 months ago | 1 author (You)
6 @Injectable()
7 export class LoggerMiddleware implements NestMiddleware {
8   use(req: Request, res: Response, next: NextFunction) {
9     logger.debug(`[${req.method}] ${req.originalUrl}`);
10    next();
11  }

```

Figure 5-60: LoggerMiddleware

Finally, to make the access control logic more expressive and developer-friendly, the system defines a custom decorator. This decorator simplifies how role-based permissions are declared within controllers, providing a clean and readable way to link route handlers with the guards that protect them.

Custom Decorator: `@Roles()`

A reusable decorator that sets role-based metadata on route handlers. Used together with RolesGuard, it enables expressive and declarative access control in controllers:

```

roles.decorator.ts ×
src > auth > decorators > roles.decorator.ts > ...
You, 2 months ago | 1 author (You)
1 import { SetMetadata } from '@nestjs/common';
2 import { UserRole } from '../../../../../enums/userRole.enum';
3
4 export const ROLES_KEY = 'roles';
5 export const Roles = (...roles: UserRole[]) => SetMetadata(ROLES_KEY, roles);
6

```

Figure 5-61: RolesDecorator

5.2.2.3 Database Layer with Prisma ORM

The database layer serves as the foundation for data persistence, providing structured storage, relational consistency, and efficient querying across all entities of the system. It is where every transaction, user profile, and business process ultimately converges and is recorded. In this project, the database was implemented using PostgreSQL and accessed through the Prisma ORM within the NestJS backend.

PostgreSQL was chosen as the primary database management system for multiple reasons, both technical and educational. From a learning perspective, the team sought to move beyond the more commonly used MySQL environment taught in university and gain experience with an enterprise-grade relational database.

PostgreSQL is widely adopted in industry for its robustness, open-source reliability, and advanced feature set, making it ideal for projects that may evolve in complexity.

Its support for complex data types, JSON fields, and extension ecosystem further set it apart. In particular, the project leveraged PostgreSQL's fuzzy search capabilities through the pg_trgm extension, which provides approximate string matching. This allowed for more natural and tolerant search results within the system, enabling users to find items or suppliers even when queries contained typos or incomplete names; an improvement over exact string matching.

From an architectural standpoint, PostgreSQL offered strong ACID compliance, ensuring that every transaction (such as order creation, invoice generation, or user registration) remained consistent and reliable even under concurrent load. This reliability was essential for maintaining data integrity across interconnected modules like Orders, Suppliers, Products, and Reviews.

The connection between the backend logic and PostgreSQL was facilitated through Prisma, a modern ORM (Object Relational Mapper) that integrates seamlessly with TypeScript. Prisma was chosen because of its type-safety, auto-generated client API, and developer productivity benefits. It abstracts SQL complexity while still allowing low-level control when necessary, providing the best of both worlds; safety and flexibility.

Earlier, the decision to use Prisma was motivated by the same principle that guided the choice of NestJS: balancing learning with maintainability. The Prisma schema offers a declarative, human-readable model of the database, where all tables, fields, relationships, and enums are defined in a single place. This schema acts as both documentation and the source of truth for the entire data model.

Below is an excerpt from the Prisma schema showing the User model, which illustrates how Prisma maps models to database tables:

```

17 // ===== USER =====
18 model User {
19   id      String    @id @default(uuid())
20   tapCustomerId String  @unique
21   email   String   @unique
22   crn     String   @unique
23   password String
24   name    String
25   role    UserRole @default(BUYER)
26   businessName String
27   city    String
28   nid     String   @unique
29   agreedToTerms Boolean @default(true)
30   isEmailVerified Boolean @default(false)
31   pfpFileName String?
32   isPfpDefault Boolean @default(true)
33   preferredLanguage Languages @default(EN)
34   createdAt DateTime @default(now())
35   updatedAt  DateTime @updatedAt
36
37   categories  UserCategory[]
38   supplier    Supplier?
39   buyer       Buyer?
40   notificationPreference NotificationPreference?
41   sentNotifications Notification[] @relation("SentNotifications")
42   receivedNotifications Notification[] @relation("ReceivedNotifications")
43   firstChatUser Chat[] @relation("FirstUser")
44   secondChatUser Chat[] @relation("SecondUser")
45   sentMessages  Message[] @relation("SentMessages")
46   receivedMessages Message[] @relation("ReceivedMessages")
47 }

```

Figure 5-62: User Model on shema.prisma

The complete Prisma schema can be found in Appendix D.

Each Prisma model maps directly to a table in PostgreSQL. The schema syntax supports features such as `@relation` (for foreign keys), `@default` (for default values), and `@enum` (for controlled attributes like roles or statuses). From this schema, Prisma automatically generates a type-safe client that can be imported anywhere in the backend, allowing database operations like this:

```

const user = await this.prisma.user.findUnique({
  where: { id },
});

```

Figure 5-63: Example of how to use Prisma Client

This strongly typed approach eliminates entire categories of runtime errors, providing autocompletion and validation during development.

Another major strength of Prisma is its migrations system. Each time the schema is updated, Prisma generates a corresponding SQL migration file that records the structural change. This feature proved invaluable for team collaboration:

- Every developer could synchronize database changes consistently using “`npx prisma migrate deploy`” command.
- Version control ensured that schema evolution remained traceable and reversible.

- Deployment to staging and production environments became more predictable, as the same migration files could be executed identically on all servers.

Moreover, Prisma migrations integrated seamlessly with the CI/CD pipeline (discussed later), enabling automatic schema updates during deployment without manual intervention. This greatly reduced the chance of schema mismatches between local and remote databases.

To initialize static data (particularly the hierarchical category structure for products and services) we implemented a custom seeding script named `categorySeed.ts`. This script programmatically inserts all parent and child categories using Prisma's client API, ensuring that every deployment starts with the same consistent dataset. It can be executed using the “`npm run prisma:seed:category`” command.

```
prismaClient.$transaction(async (tx) => {
  const productCategories = tx.$queryRaw`SELECT * FROM product_categories WHERE parent_id IS NULL`;
  const serviceCategories = tx.$queryRaw`SELECT * FROM service_categories WHERE parent_id IS NULL`;

  const parentInserts = productCategories.map((category) => {
    const categoryWithId = { ...category, id: generateId() };
    const parent = tx.$queryRaw`SELECT * FROM product_categories WHERE name = ${categoryWithId.name}`;
    if (!parent) {
      return categoryWithId;
    }
    categoryWithId.parentId = parent.id;
    return categoryWithId;
  });

  const categoryUpdates = parentInserts.map((category) => {
    const updatedCategory = { ...category, parent_id: category.parentId };
    return updatedCategory;
  });

  await tx.$batchUpdate(`UPDATE product_categories SET ${categoryUpdates.map((category) => `parent_id = ${category.parentId}`).join('')}`);
  await tx.$batchInsert(`INSERT INTO product_categories ${parentInserts.map((category) => `(${category.id}, ${category.name}, ${category.parent_id})`).join(',')}`);
  await tx.$batchUpdate(`UPDATE service_categories SET ${parentInserts.map((category) => `parent_id = ${category.parentId}`).join('')}`);
  await tx.$batchInsert(`INSERT INTO service_categories ${parentInserts.map((category) => `(${category.id}, ${category.name}, ${category.parentId})`).join(',')}`);
});
```

Figure 5-64: `categorySeed.ts` (1/3)

```
for (const category of productCategories) {
  const parent = await tx.$queryRaw`SELECT * FROM product_categories WHERE name = ${category.name}`;
  if (!parent) {
    category.parentId = generateId();
    await tx.$insert(`INSERT INTO product_categories ${category}`);
  } else {
    category.parentId = parent.id;
    await tx.$update(`UPDATE product_categories SET parent_id = ${parent.id} WHERE name = ${category.name}`);
  }
}

for (const category of serviceCategories) {
  const parent = await tx.$queryRaw`SELECT * FROM service_categories WHERE name = ${category.name}`;
  if (!parent) {
    category.parentId = generateId();
    await tx.$insert(`INSERT INTO service_categories ${category}`);
  } else {
    category.parentId = parent.id;
    await tx.$update(`UPDATE service_categories SET parent_id = ${parent.id} WHERE name = ${category.name}`);
  }
}
```

Figure 5-65: `categorySeed.ts`

```
for (const category of productCategories) {
  const parent = await tx.$queryRaw`SELECT * FROM product_categories WHERE name = ${category.name}`;
  if (!parent) {
    category.parentId = generateId();
    await tx.$insert(`INSERT INTO product_categories ${category}`);
  } else {
    category.parentId = parent.id;
    await tx.$update(`UPDATE product_categories SET parent_id = ${parent.id} WHERE name = ${category.name}`);
  }
}

for (const category of serviceCategories) {
  const parent = await tx.$queryRaw`SELECT * FROM service_categories WHERE name = ${category.name}`;
  if (!parent) {
    category.parentId = generateId();
    await tx.$insert(`INSERT INTO service_categories ${category}`);
  } else {
    category.parentId = parent.id;
    await tx.$update(`UPDATE service_categories SET parent_id = ${parent.id} WHERE name = ${category.name}`);
  }
}

try {
  await tx.$batchUpdate(`UPDATE product_categories SET ${parentInserts.map((category) => `parent_id = ${category.parentId}`).join('')}`);
  await tx.$batchInsert(`INSERT INTO product_categories ${parentInserts.map((category) => `(${category.id}, ${category.name}, ${category.parentId})`).join(',')}`);
  await tx.$batchUpdate(`UPDATE service_categories SET ${parentInserts.map((category) => `parent_id = ${category.parentId}`).join('')}`);
  await tx.$batchInsert(`INSERT INTO service_categories ${parentInserts.map((category) => `(${category.id}, ${category.name}, ${category.parentId})`).join(',')}`);
} catch (err) {
  console.error(err);
  process.exit(1);
}
```

Figure 5-66: `categorySeed.ts` (3/3)
(2/3)

This approach was cleaner and more maintainable than raw SQL inserts, as it reused Prisma's models, validations, and type definitions.

PostgreSQL's extensibility was also leveraged to enhance search functionality within the application. The pg_trgm extension enabled fuzzy string matching, allowing searches to account for typos, partial matches, and similar patterns. For example, when users searched for suppliers or products, the system ranked results based on similarity scores rather than exact equality. This created a smoother and more natural search experience.

```

211     async searchProducts(
212       name?: string,
213       mainCategoryId?: string,
214       subCategoryId?: string,
215       minPrice?: string = '1',
216       maxPrice?: string,
217       targetLang?: 'ar' | 'en',
218     ) {
219       // Convert query param safely
220       const mainId = mainCategoryId ? Number(mainCategoryId) : undefined;
221       const subId = subCategoryId ? Number(subCategoryId) : undefined;
222       const minPriceAsNumber =
223         minPrice && Number(minPrice) > 0 ? Number(minPrice) : undefined;
224       const maxPriceAsNumber =
225         maxPrice && Number(maxPrice) > 0 ? Number(maxPrice) : undefined;
226
227       // Sanitize name
228       let q = name?.trim() ? replace(/\//g, '') : '';
229       if (q.length > 100) q = q.substring(0, 100);
230
231       // 1. Build base where
232       const whereClause: any = {
233         isDeleted: false,
234         isPublished: true,
235         supplier: [
236           { isStoreClosed: false, status: SupplierStatus.ACTIVE },
237         ],
238       };
239
240       // 2. Handle category filters
241       if (mainId && !subId) {
242         const subs = await this.prisma.category.findMany({
243           where: { parentCategoryId: mainId },
244           select: { id: true },
245         });
246         if (!subs.length) return [];
247         whereClause.categoryId = { in: subs.map(s => s.id) };
248       } else if (subId) {
249         const exists = await this.prisma.category.findUnique({
250           where: { id: subId },
251           select: { id: true },
252         });
253         if (!exists) return [];
254         whereClause.categoryId = subId;
255       }
256     }
257   }

```

Figure 5-67: Fuzzy Search Example (1/2)

By combining Prisma's query syntax with PostgreSQL's fuzzy search extension, the system provided both precision and tolerance in search results; a balance that was crucial for usability.

```

258   // 3. Fuzzy search
259   let productIds: string[] = [];
260   if (q) {
261     const query = `
262       SELECT p."id", similarity(p."name", '$(q)') AS sim
263       FROM "Product" p
264       JOIN "Supplier"s ON p."supplierId" = s."id"
265       WHERE p."isDeleted" = false
266       AND p."isPublished" = true
267       AND s."isStoreClosed" = false
268       AND similarity(p."name", '$(q)') > 0.1
269       ORDER BY sim DESC
270       LIMIT 50
271     `;
272
273     const fuzzyRows = await this.prisma.$queryRawUnsafe(` ${query}`);
274     productIds = fuzzyRows.map(r => r.id);
275     if (!productIds.length) return [];
276     whereClause.id = { in: productIds };
277   }
278
279   // 4. Apply price filters
280   whereClause.price = {
281     ...(minPriceAsNumber !== undefined
282       ? { gte: minPriceAsNumber }
283       : {}),
284     ...(maxPriceAsNumber !== undefined
285       ? { lte: maxPriceAsNumber }
286       : {}),
287   };
288
289   // 5. Fetch products
290   let products = await this.prisma.product.findMany({
291     where: whereClause,
292     include: { supplier: { include: { user: true } }, category: true },
293   });
294
295   if (!products.length) return [];
296
297   // 6. Map to DTOs
298   return Promise.all(
299     products.map(p =>
300       this.productService.toProductResponseDto(p, targetLang),
301     ),
302   );
303 }
304

```

Figure 5-68: Fuzzy Search Example (2/2)

5.2.2.4 Real-Time and Streaming Features

Real-time communication is a cornerstone of modern web applications, enabling instant feedback, live updates, and interactive experiences. In Silah, this capability is essential for delivering timely chat messages and push notifications to users without requiring constant polling. Two complementary technologies were implemented: WebSocket for bidirectional, low-latency chat, and Server-Sent Events (SSE) for unidirectional, lightweight notification streaming. This hybrid approach balances performance, scalability, and development simplicity.

WebSocket was selected for the chat service due to its full-duplex, bidirectional communication model, which allows both the client and server to push data at any time without the overhead of repeated HTTP handshakes. This capability ensures low-latency message delivery, making it ideal for rapid, conversational exchanges where timing is critical. Unlike stateless HTTP, WebSocket maintains a stateful, persistent connection, enabling users to join chat-specific rooms for targeted message broadcasting and efficient scaling across multiple concurrent conversations.

Furthermore, its event-driven nature naturally supports interactive features such as typing indicators, read receipts, and message acknowledgments, enriching the real-time user experience with immediate feedback and dynamic interaction.

NestJS provides a powerful abstraction over Socket.IO through the `@nestjs/websockets` package. The ChatGateway serves as the entry point for all WebSocket connections.

```

src > chat > ChatGateway.ts | ChatGateway > constructor
  You, 3 weeks ago | author | You!
  1 import {
  2   WebSocketGateway,
  3   WebSocketServer,
  4   SubscribeMessage,
  5   MessageBody,
  6   ConnectedSocket,
  7 } from '@nestjs/websockets';
  8 import * as cookie from 'cookie';
  9 import { JwtService } from '@nestjs/jwt';
 10 import { Server, Socket } from 'socket.io';
 11 import { ChatService } from './chat.service';
 12 import { MessageResponseDto } from './dtos/messageResponse.dto';
 13 import { SendMessageDto } from './dtos/sendMessage.dto';
 14 import [
 15   Inject,
 16   UnauthorizedException,
 17   Injectable,
 18   forwardRef,
 19 ] from '@nestjs/common';
 20
 21 @Injectable()
 22 @WebSocketGateway({ cors: { origin: true, credentials: true } })
 23 export class ChatGateway {
 24   constructor(
 25     @Inject(forwardRef(() => ChatService))
 26     private readonly chatService: ChatService,
 27     private readonly jwtService: JwtService,
 28   ) {}
 29
 30   async handleConnection(client: Socket) {
 31     try {
 32       const rawCookies = client.handshake.headers.cookie || '';
 33       const parsedCookies = cookie.parse(rawCookies);
 34       let token = parsedCookies['token'];
 35       if (!token)
 36         throw new UnauthorizedException('No token found in cookies');
 37
 38       // Strip the 'j:' prefix if present
 39       if (token.startsWith('j:')) token = token.slice(2);
 40
 41       // Now parse JSON if it looks like an object
 42       if (token.startsWith('{') && token.endsWith('}')) {
 43         const obj = JSON.parse(token);
 44         token = obj.token;
 45       }
 46
 47       const payload = await this.jwtService.verifyAsync(token);
 48       ({client as any}.userId = payload.sub);
 49     } catch (err) {
 50       console.log(`Invalid socket connection:: ${err.message}`);
 51       client.disconnect(true);
 52     }
 53   }
 54
 55   @WebSocketServer() server: Server;

```

Figure 5-69: ChatGateway (1/2)

```

  57   @SubscribeMessage('join_chat')
  58   handleJoinChat(
  59     @ConnectedSocket() client: Socket,
  60     @MessageBody() chatId: string,
  61   ) {
  62     client.join(`chat_${chatId}`);
  63     client.emit('joined_chat', { chatId });
  64   }
  65
  66   @SubscribeMessage('send message')
  67   handleSendMessage(
  68     @ConnectedSocket() client: Socket,
  69     @MessageBody() data: SendMessageDto,
  70   ) {
  71     const senderId = (client as any).userId;
  72     const { receiverId, chatId, text } = data;
  73
  74     // 1. If no chatId - create chat first
  75     let finalChatId = chatId;
  76     if (!finalChatId) {
  77       const chat = await this.chatService.createChatIfNotExists(
  78         senderId,
  79         receiverId,
  80       );
  81       finalChatId = chat.id;
  82       client.join(`chat_${finalChatId}`);
  83     }
  84
  85     // 2. Save message + Send notification
  86     const message = await this.chatService.createMessage(
  87       senderId,
  88       receiverId,
  89       finalChatId,
  90       text,
  91     );
  92
  93     // 3. Emit to receiver specifically (in case they aren't in the chat)
  94     this.server.to(receiverId).emit('message_notification', {
  95       chatId: finalChatId,
  96       preview: text,
  97     });
  98
  99     // 4. Acknowledge sender
 100     client.emit('message_sent', { chatId: finalChatId, message });
 101   }
 102
 103   @NewMessage(chatId: string, message: MessageResponseDto) {
 104     this.server.to(`chat_${chatId}`).emit('new_message', message);
 105   }
 106
 107   @SubscribeMessage('join_user')
 108   handleJoinUser(@ConnectedSocket() client: Socket) {
 109     const userId = (client as any).userId;
 110     client.join(userId);
 111     client.emit('joined_user_room', userId);
 112   }

```

Figure 5-70: ChatGateway (2/2)

While WebSocket excels in interactive scenarios, Server-Sent Events (SSE) was chosen for notification delivery due to its unidirectional, server-to-client streaming model, which perfectly aligns with the one-way nature of push notifications where the server initiates updates and the client only receives them. Built directly on standard HTTP, SSE introduces minimal protocol overhead compared to WebSocket, eliminating the need for additional message framing or handshake complexity, while still supporting automatic reconnection through the browser's native EventSource API; ensuring reliable delivery even during temporary network interruptions [38]. This lightweight approach enables efficient, scalable broadcasting to thousands of connected clients using a single long-lived connection per user, with the server filtering events by receiverUserId before transmission. Additionally, SSE integrates seamlessly with existing NestJS infrastructure, leveraging RxJS observables and standard HTTP endpoints, making it simpler to implement, debug, and secure than a full bidirectional protocol for a use case that requires no client-to-server event flow.

The NotificationController exposes an SSE endpoint at GET /notifications/stream.

```

@ApiDocsJwtAuthGuard()
@ApiDocsVerifiedGuard()
@UseGuards(JwtAuthGuard, VerifiedGuard)
@ApiDocsNotificationStream()
@Sse('stream')
notificationStream(
  @Req() req: Request,
  @Res({ passthrough: true }) res: Response,
): Observable<MessageEvent> {
  const userId = req.userData!.sub;

  const disconnect$ = new Subject<void>(); // This will signal when to stop
  // When the browser or client disconnects:
  req.on('close', () => {
    disconnect$.next(); // trigger the stop
    disconnect$.complete();
  });

  return this.notificationService.getNotificationStream().pipe(
    // tap((event) =>
    //   console.log(`Stream event: ${event.receiverUserId}`),
    // ), // add tap from rxjs/operators
    filter((event) => event.receiverUserId === userId),
    takeUntil(disconnect$), // Stop streaming when disconnects emits
    map((event) => {
      const payload = JSON.stringify(event.notification);
      return { data: payload }; // must be stringified
    }),
  );
}

```

Figure 5-71: SSE Endpoint in NotificationController

A critical design decision was making the `NotificationModule` global using the `@Global()` decorator. This ensures the `NotificationService` is a singleton across the entire application.

```

src > notification > notification.module.ts > ...
  You, 3 weeks ago | 1 author (You)
1 import { Global, Module } from '@nestjs/common';
2 import { NotificationService } from './notification.service';
3 import { NotificationController } from './notification.controller';
4 import { UserModule } from 'src/user/user.module';
5 import { TranslationModule } from 'src/translation/translation.module';
6
  You, 3 weeks ago | 1 author (You)
7 @Global()
8 @Module({
9   imports: [UserModule, TranslationModule],
10  controllers: [NotificationController],
11  providers: [NotificationService],
12  exports: [NotificationService],
13 })
14 export class NotificationModule {}

```

Figure 5-72: Global NotificationModule

This ensures a unified source of truth for real-time notifications.

By maintaining a single `notifications$` RxJS Subject, all modules (such as Chat, Invoice, and Order) emit events to the same observable stream, eliminating the risk of fragmented or duplicated event flows that could arise from multiple independent instances. This centralized stream guarantees that every notification, regardless of its origin, is processed consistently and delivered exactly once to the appropriate SSE clients. Moreover, this approach is memory-efficient, as only one observer pipeline exists in the application's lifecycle, avoiding the overhead of creating separate streams per module or request while enabling seamless, scalable broadcasting to all connected users through a single, globally accessible service.

```

448     async createNotification(
449       data: CreateNotification,
450     ): Promise<NotificationResponseDto> {
451       // Step 1: Create the notification in the database
452       const notification = await this.prisma.notification.create({
453         data: {
454           senderUserId: data.senderUserId,
455           receiverUserId: data.receiverUserId,
456           type: data.type,
457           title: data.title,
458           content: data.content,
459           entityId: data.entityId,
460           entityType: data.entityType,
461         },
462         include: { sender: true, receiver: true },
463       });
464
465       // Step 2: Convert to DTO
466       const dto = await this.toNotificationResponseDto(notification);
467
468       // Step 2.5: Load notification preferences
469       const preference = await this.prisma.notificationPreference.findUnique({
470         where: { userId: data.receiverUserId },
471       });
472
473       const userLang = await this.getUserLanguage(data.receiverUserId);
474
475       // Translate before emitting
476       const translatedDto =
477         userLang && userLang !== 'en'
478           ? await this.translateNotification(dto, userLang)
479           : dto;

```

Figure 5-73: createNotification method on NotificationService (1/2)

```

481   // Step 3: Emit based on preference
482   if (!preference.followNotifications) {
483     shouldSend = false;
484   }
485
486   switch (data.type) {
487     case 'NEW_MESSAGE':
488       shouldSend = preference.newMessageNotify;
489       break;
490     case 'NEW_ORDER':
491       shouldSend = preference.newOrderNotify;
492       break;
493     case 'NEW_REVIEWS':
494       shouldSend = preference.newReviewNotify;
495       break;
496     case 'NEW_INVOICE':
497       shouldSend = preference.newInvoiceNotify;
498       break;
499     case 'NEW_OFFER':
500       shouldSend = preference.newOfferNotify;
501       break;
502     case 'BID_STATUS_CHANGED':
503       shouldSend = preference.biddingStatusNotify;
504       break;
505     case 'INVOICE_STATUS_CHANGED':
506       shouldSend = preference.invoiceStatusNotify;
507       break;
508     case 'ORDER_STATUS_CHANGED':
509       shouldSend = preference.orderStatusNotify;
510       break;
511     case 'GROUP_PURCHASE_STATUS_CHANGED':
512       shouldSend = preference.groupPurchaseStatusNotify;
513       break;
514   }
515
516   if (shouldSend) {
517     this.notifications$.next({
518       receiverUserId: data.receiverUserId,
519       notification: translatedDto,
520     });
521   }
522
523   return translatedDto;
524 }
525
526 )
527
528 )
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576

```

Figure 5-74: createNotification method on NotificationService (2/2)

Any service can inject NotificationService and call createNotification(), the event is automatically pushed to all connected SSE clients of the receiver.

```

// Send a notification for the buyer
await this.notificationService.createNotification({
  senderUserId: supplier.userId,
  receiverUserId: buyer.userId,
  type: NotificationType.NEW_INVOICE,
  title: 'New Invoice!',
  content: `You have received a new invoice from ${invoice.supplier!.user.businessName}`,
  entityId: invoice.id,
  entityType: NotificationEntityType.INVOICE,
});

```

Figure 5-75: Usage of NotificationService on the InvoiceService

While WebSocket handles text messages efficiently, file uploads (images in chat) are managed via REST using POST /chats/me/:id/upload. REST was chosen because it natively supports multipart/form-data for efficient binary transfer, provides built-in validation through NestJS pipes (e.g., file size and type checks), and leverages browser-native progress tracking and automatic retry mechanisms inherent to HTTP. In contrast, sending files over WebSocket would require base64 encoding, increasing payload size by ~33%, complicating parsing, and risking full retransmission on connection loss. By handling uploads via REST and then emitting the resulting message via WebSocket, the system combines the reliability and simplicity of HTTP for file transfer with the real-time responsiveness of WebSocket for message delivery, ensuring robust, maintainable, and user-friendly media sharing without compromising performance or developer experience.

```

99      @ApiDocsJwtAuthGuard()
100     @ApiDocsVerifiedGuard()
101     @UseGuards(JwtAuthGuard, VerifiedGuard)
102     @UseInterceptors(FileInterceptor('file')) // "file" = form field name
103     @Post('me/:id/upload')
104     @ApiDocsSendImageMessage()
105     async sendImage(
106       @UploadedFile()
107       new ParseFilePipe({
108         validators: [
109           new MaxFileSizeValidator({ maxSize: 5 * 1024 * 1024 }),
110           new FileTypeValidator({
111             fileType: /image\/(png|jpe?g|webp)$/,
112             skipMagicNumbersValidation: true,
113           }),
114         ],
115       }),
116     )
117     file: Express.Multer.File,
118     @Req() req: Request,
119     @Param('id') chatId: string,
120   ) {
121     const userId = req.tokenData!.sub;
122     return this.chatService.sendImageMessage(file, userId, chatId);
123   }

```

Figure 5-76: Send Image Endpoint on ChatController

```

194     async sendImageMessage(
195       file: Express.Multer.File,
196       userId: string,
197       chatId: string,
198     ) {
199       const chat = await this.prima.chat.findUnique({
200         where: { id: chatId },
201         select: { ownerId: true, userId: true },
202       });
203       if (!chat || (chat.ownerId !== userId && chat.userId !== userId)) {
204         throw new ForbiddenException('You are not part of this chat.');
205       }
206
207       const fileName = await this.fileservice.uploadFile(file);
208       const receiverId = this.getOtherUserId(chat, userId);
209
210       const message = await this.prima.message.create({
211         data: {
212           chatId,
213           ownerId: userId,
214           receiverId,
215           text: null,
216           imageUrlName: fileName,
217         },
218         include: [
219           { sender: { include: { firstChatUser: true } } },
220           { receiver: { include: { secondChatUser: true } } },
221         ],
222       });
223
224       await this.prima.chat.update({
225         where: { id: chatId },
226         data: { updatedAt: new Date() },
227       });
228
229       const messageInfo = await this.toMessageResponseDto(message);
230
231       this.chatGateway.emitNewMessage(chatId, messageInfo);
232
233       await this.notificationService.createNotification({
234         senderUserId: userId,
235         receiverUserId: receiverId,
236         type: NotificationType.NEW_MESSAGE,
237         title: `New message from ${message.sender.businessName}`,
238         content: message.text ? message.text : 'Sent you an image',
239         entityId: message.id,
240         entityType: NotificationEntityType.CHAT,
241       });
242
243       return {
244         message: 'Image sent successfully',
245         data: messageInfo,
246       };
247     }
248   }

```

Figure 5-77: Send Image Logic on ChatService

5.2.2.5 Integrations with External Services

The Silah platform relies on a suite of third-party services to extend its core functionality, ensuring reliability, compliance, scalability, and user trust. These integrations are abstracted into dedicated NestJS services, enabling clean separation of concerns and secure configuration via environment variables. Each service addresses a specific operational need (translation, payment processing, business verification, file storage, and AI inference) while maintaining consistency in error handling, logging, and API contract design.

To support bilingual user experiences in Arabic and English, the system integrates with the DeepL API, a high-accuracy machine translation service. DeepL was selected for its superior natural language quality over alternatives like Google Translate, particularly in handling Arabic technical and business terminology [39]. The TranslationService encapsulates all translation logic, supporting both single-string and batch translation to minimize API rate limits and improve performance during bulk operations such as product listings or notifications.

```

src > translation > ts > translationService > TranslationService.js
  You, yesterday | Author (You)
  import { Injectable, Logger } from '@nestjs/common';
  import axios from 'axios';
  import * as qs from 'qs';

  You, yesterday | Author (You)
  @Injectable()
  export class TranslationService {
    private readonly apiUrl = process.env.DEEPL_API_KEY;
    private readonly baseUrl = 'https://api-free.deepl.com/v2/translate';
    private readonly logger = new Logger(TranslationService.name);

    /**
     * Translate a single text string
     */
    async translateText({
      text: string,
      targetLang: 'en' | 'en',
      sourceLang: string,
    }: Promise<string>): Promise<string> {
      if (!text || !targetLang) return text;

      const params: any = {
        auth_key: this.apiUrl,
        text,
        target_lang: targetLang.toUpperCase(),
      };
      if (sourceLang) params.source_lang = sourceLang.toUpperCase();

      try {
        const response = await axios.post(this.baseUrl, null, { params });
        return response.data.translations[0].text;
      } catch (err) {
        this.logger.error(`DeepL translation error (single): ${err.response?.data?.message || err.message}`);
      }
      return text;
    }

    /**
     * Translate multiple texts in batches (avoids 429 Too Many Requests)
     */
    async translateBatch({
      texts: string[],
      targetLang: 'en' | 'en',
      sourceLang: string,
    }: Promise<string>): Promise<string> {
      if (!texts) return texts;

      this.logger.debug(`Translating batch of ${texts.length} texts`);

      const invalid = texts.filter((t) => typeof t !== 'string' || !t.trim().length);
      if (invalid.length > 0) {
        this.logger.warn(`⚠️ Found ${invalid.length} invalid texts`);
        console.log(invalid);
      }

      const validTexts = texts.filter((t) => typeof t === 'string' && t.trim().length > 0);
    }
  }

```

Figure 5-78: DeepL Integration on TranslationService (1/2)

```

66   if (validTexts.length === 0) {
67     this.logger.warn(`⚠️ No valid texts to translate.`);
68     return texts;
69   }

70   const CHUNK_SIZE = 40;
71   const results: string[] = [];
72
73   for (let i = 0; i < validTexts.length; i += CHUNK_SIZE) {
74     const chunk = validTexts.slice(i, i + CHUNK_SIZE);
75
76     // ✅ Use application/x-www-form-urlencoded body, not query params
77     const body = qs.stringify({
78       auth_key: this.apiUrl,
79       target_lang: targetLang.toUpperCase(),
80       ...(sourceLang && {
81         source_lang: sourceLang.toUpperCase(),
82       }),
83       text: chunk, // DeepL supports multiple "text" fields
84     }, { arrayFormat: 'repeat' });
85
86     try {
87       const response = await axios.post(this.baseUrl, body, {
88         headers: {
89           'Content-Type': 'application/x-www-form-urlencoded',
90         },
91       });
92
93       const translatedChunk = response.data.translations.map(
94         (t) => t.text,
95       );
96       results.push(...translatedChunk);
97
98       if (i + CHUNK_SIZE < validTexts.length)
99         await new Promise((r) => setTimeout(r, 1000));
100     } catch (err) {
101       this.logger.error(`DeepL translation error (batch): ${err.response?.data?.message || err.message}`);
102     }
103   }
104
105   const translatedMap = new Map();
106   validTexts.map((t, i) => [t, results[i] || t]);
107
108   return texts.map(t => translatedMap.get(t) || t);
109 }
110
111
112
113
114

```

Figure 5-79: DeepL Integration on TranslationService (2/2)

The service sends a POST request to <https://api-free.deepl.com/v2/translate> with the API key in the auth_key parameter and text encoded via application/x-www-form-urlencoded. For batch operations, texts are chunked into groups of 40 to avoid the 429 Too Many Requests error, with a 1-second delay between chunks. Errors are gracefully handled by returning the original text, ensuring the application remains functional even during transient API failures. This fallback mechanism preserves user experience while logging issues for monitoring.

Payment processing is handled through Tap Payments, a PCI-DSS-compliant gateway widely used in the MENA region. Tap was chosen for its support for saved cards, 3D Secure authentication, and reliable redirect-based flow, which enables secure, reusable payment methods without requiring server-to-server webhooks.

The TapPaymentsService manages the full payment lifecycle: customer creation, card tokenization, and charge initiation.

```

src > tap-payments > tap-payments.services > TapPaymentsService
  You, 4 weeks ago | 1 author (You)
  1 import (
  2   BadRequestException,
  3   Injectables,
  4   InternalServerErrorException,
  5 ) from '@nestjs/common';
  6 import axios from 'axios';
  7
  8 @Injectable()
  9 export class TapPaymentsService {
 10   /**
 11    * Creates a new customer in Tap Payments.
 12    *
 13    * Sends a POST request to the Tap Payments API with the provided customer information.
 14    *
 15    * @param {Object} data - Customer information.
 16    * @param {String} data.first_name - Customer's first name.
 17    * @param {String} [data.middle_name] - Customer's middle name (optional).
 18    * @param {String} [data.last_name] - Customer's last name (optional).
 19    * @param {String} data.email - Customer's email address.
 20    * @param {Object} [data.phone] - Customer's phone information (optional).
 21    * @param {String} [data.phone.country_code] - Country code of the phone number (e.g., '966').
 22    * @param {String} [data.phone.number] - Phone number without country code.
 23    *
 24    * @returns {Promise<String>} The ID of the newly created Tap customer.
 25    *
 26    * @throws {Error} Throws an error if the API request fails. The error message will contain
 27    * the response from Tap Payments if available, or the generic error message.
 28    *
 29    * @example
 30    * const customerId = await tapService.createCustomer({
 31    *   first_name: 'John',
 32    *   last_name: 'Doe',
 33    *   email: 'john.doe@example.com',
 34    *   phone: { country_code: '966', number: '566600000' },
 35    * });
 36    * console.log(customerId); // e.g., 'cus_123456789'
 37    */
 38   async createCustomer(data: {
 39     first_name: string;
 40     middle_name?: string;
 41     last_name?: string;
 42     email: string;
 43     phone?: { country_code: string; number: string };
 44   }) {
 45     try {
 46       const response = await axios.post(
 47         'https://api.tap.company/v2/customers',
 48         data,
 49         {
 50           headers: {
 51             Authorization: `Bearer ${process.env.TAP_SECRET_KEY}`,
 52             'Content-Type': 'application/json',
 53           },
 54         },
 55       );
 56       return response.data.id;
 57     } catch (err: any) {
 58       console.error(err.response?.data || err.message);
 59       throw err;
 60     }
 61   }
 62 }

```

Figure 5-80: createCustomer on TapPaymentService

The payment flow begins when a buyer adds a card or completes checkout using Tap's JavaScript SDK, which generates a secure token. This token is sent to the backend, where createCharge() initiates a minimal 1 SAR authorization (for card saving) or full order amount, with save_card: true and a redirect.url pointing to a frontend callback page (e.g., /payment/callback). After 3DS verification, Tap redirects the user to this frontend URL with query parameters including tap_id (the charge ID).

```

async createCharge({
  tokenId: string,
  tapCustomerId: string,
  redirectUrl: string,
  amount: number = 1,
  currency: string = 'SAR',
}) {
  try {
    const response = await axios.post(
      'https://api.tap.company/v2/charges/',
      {
        amount,
        currency,
        customer_initiated: true,
        threeDSecure: true,
        save_card: true, // mark true to save the card
        description: 'Card save charge',
        metadata: { udfl: 'Save card test' },
        receipt: { email: false, sms: false },
        // reference: {
        //   transaction: 'txns_00000000000000000000000000000000',
        //   order: 'ord_00000000000000000000000000000000',
        // },
        customer: { id: tapCustomerId },
        merchant: { id: process.env.TAP_MERCHANT_ID },
        source: { id: tokenId },
        // post: { url: 'http://your website.com/post_url' },
        redirect: { url: redirectUrl }, // redirect URL required by Tap to redirect after 3DS (OTP step)
      },
      {
        headers: {
          Authorization: `Bearer ${process.env.TAP_SECRET_KEY}`,
          'Content-Type': 'application/json',
          Accept: 'application/json',
        },
      },
    );
    return response.data;
  } catch (err: any) {
    console.error(err.response?.data || err.message);
    throw new Error('Failed to create charge in Tap');
  }
}

```

Figure 5-81: createCharge on TapPaymentService

The frontend PaymentCallback component extracts tap_id and makes a direct request to a backend confirmation endpoint (e.g., /api/buyers/me/card/confirm or /api/cart/checkout). This backend endpoint:

- Validates the tap_id
- Calls tapService.getCharge(tap_id) to verify status (CAPTURED, AUTHORIZED, etc.)
- Updates the database (saves card, marks order as paid, etc.)
- Returns success/failure

```

async validateCharge(charge: any) {
  const createdMs =
    charge.transaction?.created || charge.transaction?.date?.created;
  if (!createdMs) {
    throw new InternalServerErrorException(
      'Cannot determine charge creation time.',
    );
  }

  const chargeCreationTime = new Date(Number(createdMs));
  const now = new Date();

  const timeDifference = now.getTime() - chargeCreationTime.getTime();
  // 1 hour = 3600000 ms
  if (timeDifference > 3600000) {
    throw new BadRequestException(
      'Charge ID is too old. Please initiate a new payment.',
    );
  }
}

async getCard(customerId: string, cardId: string) {
  try {
    const response = await axios.get(
      'https://api.tap.company/v2/card/${customerId}/${cardId}',
      {
        headers: {
          accept: 'application/json',
          Authorization: `Bearer ${process.env.TAP_SECRET_KEY}`,
        },
      },
    );
    return response.data;
  } catch (err: any) {
    console.error(err.response?.data || err.message);
    throw new Error('Failed to fetch card from Tap');
  }
}

```

Figure 5-83: getCharge on TapPaymentService

Figure 5-82: validateCharge on TapPaymentService

This frontend-driven confirmation approach simplifies the architecture by eliminating the need for webhook handling, signature verification, or public

endpoints. It leverages the user's authenticated session (via cookies) and ensures atomic updates within a single request context.

To ensure platform integrity and compliance with Saudi regulations, supplier registration requires a valid Commercial Registration Number (CRN).

The Wathq API, provided by the Ministry of Commerce, is used to verify CRN authenticity in real time. The WathqService queries the sandbox or production endpoint with the CRN and retrieves structured business data including company name, status, and expiry date.

```
src > wathq > wathq.service.ts > ...
You, 5 days ago | 1 author (You)
1 import { Injectable, HttpException, HttpStatus } from '@nestjs/common';
2 import axios, { AxiosInstance } from 'axios';
3
4 You, 5 days ago | 1 author (You)
5 @Injectable()
6 export class WathqService {
7     private client: AxiosInstance;
8
9     constructor() {
10         // Base URL for sandbox or production
11         this.client = axios.create({
12             baseURL: 'https://api.wathq.sa/sandbox',
13             timeout: 10000,
14         });
15
16         /**
17          * Get basic commercial registration info by CR or entity number
18          */
19         async getBasicInfo(id: string, language: 'ar' | 'en' = 'en') {
20             try {
21                 const response = await this.client.get(
22                     `/commercial-registration/info/${id}`,
23                     {
24                         headers: {
25                             apiKey: process.env.WATHQ_CONSUMER_KEY,
26                             accept: 'application/json',
27                         },
28                         params: { language },
29                     },
30                 );
31
32                 return response.data;
33             } catch (err: any) {
34                 const res = err.response;
35
36                 if (res) {
37                     const code = res.data?.code;
38
39                     // Handle "No Results Found" gracefully
40                     if (code === '404.2.1') {
41                         return null; // No such CRN
42                     }
43
44                     // Optional: handle known business error codes more nicely
45                     if (code.startsWith('400')) {
46                         throw new HttpException(
47                             res.data.message || 'Invalid request to Wathq',
48                             HttpStatus.BAD_REQUEST,
49                         );
50                     }
51
52                     if (String(code).startsWith('500')) {
53                         console.warn(`[Wathq] Provider internal error: ${code}`);
54                         throw new HttpException(
55                             'Temporary issue with provider',
56                             HttpStatus.BAD_GATEWAY,
57                         );
58                     }
59                 }
60
61                 console.log(err);
62                 throw new HttpException(
63                     'Failed to contact Wathq service',
64                     HttpStatus.INTERNAL_SERVER_ERROR,
65                 );
66             }
67         }
68     }
}
```

Figure 5-84: WathqService

The service gracefully handles the 404.2.1 error code (indicating no record found) by returning null, allowing the registration flow to reject invalid entries without crashing. Known error codes are mapped to appropriate HTTP exceptions, while

unexpected failures trigger a 502 Bad Gateway to indicate upstream issues. This integration enforces regulatory compliance and prevents fraudulent supplier onboarding.

All user-uploaded images (product photos, chat media, and profile pictures) are stored in Cloudflare R2, an S3-compatible object storage service. R2 was selected for its low egress costs, global CDN integration, and zero vendor lock-in due to S3 API compatibility. The FileService handles upload validation, storage, and secure retrieval via signed URLs.

```
git > cd file-service & ls
You are in file-service
1  import {
2    BadRequestException,
3    Injectable,
4    InternalServerErrorException,
5  } from '@nestjs/common';
6  import {
7    Client,
8    PutObjectCommand,
9    GetObjectCommand,
10 } from 'aws-sdk/client-s3';
11 import { v4 as uuid } from 'uuid';
12 import { getSignedUrl } from 'aws-sdk/s3-request-presigner';
13 import { fileTypeFrom } from 'file-type';
14 /**
15 * Service for handling file storage and retrieval using Cloudflare R2 (S3-compatible API).
16 * Provides methods for uploading images and generating signed URLs for file access.
17 */
18 You have 2 unused imports (useful)
19 @Injectable()
20 export class FileService {
21   private s3 = new S3Client();
22   region: 'auto';
23   endpoint: process.env.R2_ENDPOINT;
24   credentials: {
25     accessKeyId: process.env.R2_ACCESS_KEY_ID,
26     secretAccessKey: process.env.R2_SECRET_ACCESS_KEY,
27   };
28   private bucket: string = process.env.R2_BUCKET_NAME!;
29
30   /**
31    * Generates a signed URL for accessing a file stored in R2.
32    * @param {string} key The unique key (filename) of the file in the bucket.
33    * @returns {Promise<string>} A signed URL valid for 1 hour.
34    * @throws {Error} If the signing process fails.
35    */
36   async getFileUrl(key: string): Promise<string> {
37     const command = new GetObjectCommand({
38       Bucket: this.bucket,
39       Key: key,
40     });
41     return getSignedUrl(this.s3, command, { expiresIn: 3600 }); // 1 hour
42   }
43 }
```

Figure 5-85: FileService (1/2)

```
54  async uploadData(file: Express.Multer.File): Promise<string> {
55    if (!file) {
56      throw new BadRequestException('No file provided');
57    }
58
59    // Size validation
60    const maxBytes = parseInt(
61      process.env.MAX_UPLOAD_BYTES || `${5 * 1024 * 1024}`,
62      10
63    );
64    if (file.size > maxBytes) {
65      throw new BadRequestException(
66        `File size exceeds ${Math.floor(maxBytes / 1024)} MB limit`;
67      );
68    }
69
70    // Detect file type using magic numbers
71    const detectedType = await filetype.fromBuffer(file.buffer);
72    if (!detectedType) {
73      throw new BadRequestException('Unable to determine file type');
74    }
75
76    const allowedMime = ['image/jpeg', 'image/png', 'image/webp'];
77    if (!allowedMime.includes(detectedType.mime)) {
78      throw new BadRequestException();
79      'Only JPEG, PNG, and WebP files are allowed';
80    }
81
82    // Sanitize filename
83    let basename = file.originalname.replace(/[^a-zA-Z0-9_]/g, '');
84    if (basename.startsWith('/')) {
85      basename = 'file' + basename.slice(1);
86    }
87
88    const ext = detectedType.ext;
89    const truncated = basename.slice(0, 100).replace(/\.\w+$/, '');
90    const finalName = `${truncated}-${uuid()}.${ext}`;
91
92    try {
93      await this.s3.putObject({
94        Bucket: process.env.R2_BUCKET_NAME,
95        Key: finalName,
96        Body: file.buffer,
97        ContentType: detectedType.mime,
98      });
99    } catch (e) {
100      throw new InternalServerErrorException(
101        'Failed to upload file to storage provider',
102      );
103    }
104  }
105
106  return finalName;
107}
108
```

Figure 5-86: FileService (2/2)

Uploads are validated for size ($\leq 5\text{MB}$) and type (JPEG, PNG, WebP) using magic number detection via the file-type library, preventing spoofed extensions. Filenames are sanitized and appended with a UUID to avoid collisions. Files are stored with appropriate Content-Type, and access is controlled through time-limited signed URLs (1-hour expiry), ensuring security without public bucket exposure.

The platform integrates two machine learning models (Facebook Prophet for demand forecasting and fine-tuned LaBSE for semantic search) through a dedicated AI backend built with FastAPI and hosted in a separate repository (silah-ai). This backend serves as an inference-only service, executing pre-trained models without direct access to the PostgreSQL database. The NestJS backend (silah-backend) acts as the orchestrator, handling authentication, data preparation, model invocation, and result enrichment. This decoupled, service-oriented architecture ensures security,

scalability, and maintainability while leveraging the strengths of both Python (for ML) and TypeScript (for business logic).

Suppliers access demand predictions via the GET /demand-predictions/:productId endpoint, protected by JwtAuthGuard, RolesGuard (Supplier only), and VerifiedGuard. The DemandPredictionService first validates ownership and plan eligibility (Basic suppliers are blocked). It then retrieves daily sales history using a raw SQL query to aggregate CartItem quantities by order date, filling missing days with zero to satisfy Prophet's continuous time-series requirement.

```
src > demand-prediction > demand-prediction.controller.ts > ...
1 import { Controller, Get, Param, Req, UseGuards } from '@nestjs/common';
2 import { DemandPredictionService } from './demand-prediction.service';
3 import { Request } from 'express';
4 import { ApiDocsJwtAuthGuard } from 'src/auth/decorators/jwt-auth.guard.docs';
5 import { ApiDocsRolesGuard } from 'src/auth/decorators/roles.guard.docs';
6 import { UserRole } from 'src/enums/userRole.enum';
7 import { Roles } from 'src/auth/decorators/roles.decorator';
8 import { JwtAuthGuard } from 'src/auth/guards/jwt-auth.guard';
9 import { RolesGuard } from 'src/auth/guards/roles.guard';
10 import { ApiTags } from '@nestjs/swagger';
11 import { DemandPredictionResponseDto } from './dtos/demandPredictionResponse.dto';
12 import { ApiDocsGetDemandPrediction } from './demand-prediction.docs';
13 import { ApiDocsVerifiedGuard } from 'src/auth/decorators/verified-guard.docs';
14 import { VerifiedGuard } from 'src/auth/guards/verified.guard';
15
16 @UseGuards(jwtAuthGuard)
17 @ApiTags('Demand Predictions')
18 @Controller('demand-predictions')
19 export class DemandPredictionController {
20   constructor(
21     private readonly demandPredictionService: DemandPredictionService,
22   ) {}
23
24   @ApiDocsJwtAuthGuard()
25   @ApiDocsRolesGuard()
26   @ApiDocsVerifiedGuard()
27   @Get(':userId')
28   @Get('/:productId')
29   @ApiDocsGetDemandPrediction()
30   async getPredictionForProduct(
31     @Param('productId') productId: string,
32     @Req() req: Request,
33   ): Promise<DemandPredictionResponseDto> {
34     const userId = req.tokenData.sub;
35     return this.demandPredictionService.getPredictionForProduct(
36       productId,
37       userId,
38     );
39   }
40 }
```

Figure 5-87: DemandPredictionController

A minimum of 10 sales days is enforced to ensure forecast reliability. The service constructs a ForecastRequest payload containing the product_id, sales array (with date and quantity), and months: 3. This is sent to the FastAPI endpoint at \${AI_BACKEND_URL}/demand.

```
src > demand-prediction > demand-prediction.service.ts > ...
1 import { Injectable } from '@nestjs/common';
2 import { NotFoundException, UnauthorizedException, BadRequestException, InternalServerErrorException, NotAcceptableException } from '@nestjs/core';
3 import { PrismaClient } from 'src/prisma/prisma-client';
4 import { Sales } from 'src/entities/sales.entity';
5 import { DemandPredictionResponse } from './dtos/demandPredictionResponse.dto';
6 import { ForecastRequest } from './dtos/forecastRequest.dto';
7 import { DemandPredictionResponse } from './dtos/demandPredictionResponse.dto';
8 import { Product } from 'src/entities/product.entity';
9 import { User } from 'src/entities/user.entity';
10 import { CartItem } from 'src/entities/cartItem.entity';
11 import { Order } from 'src/entities/order.entity';
12 import { Sales } from 'src/entities/sales.entity';
13 import { DemandPredictionResponse } from './dtos/demandPredictionResponse.dto';
14 import { ForecastRequest } from './dtos/forecastRequest.dto';
15
16 @Injectable()
17 export class DemandPredictionService {
18   constructor() {}
19
20   private readonly prisma: PrismaClient;
21
22   private readonly logger = new Logger(DemandPredictionService.name);
23
24   async getDemandPredictionForProduct(productId: string, userId: string): Promise<DemandPredictionResponse> {
25     const supplier = await this.prisma.supplier.findUnique({
26       where: { id: userId },
27       include: { user: true },
28     });
29
30     if (!supplier || !supplier.supplierId) {
31       throw new NotFoundException(`Supplier not found`);
32     }
33
34     const userRole = supplier.user.role;
35
36     if (userRole === 'SUPPLIER') {
37       throw new UnauthorizedException(`User is not a supplier`);
38     }
39
40     const product = await this.prisma.product.findFirst({
41       where: { id: productId, supplierId: supplier.id, isDeleted: false },
42     });
43
44     if (!product) {
45       throw new NotFoundException(`Product not found`);
46     }
47
48     const initialSales = await this.getInitialSalesForProduct(productId);
49
50     // If there are less than 10 sales days
51     const salesDays = await this.getSalesDaysForProduct(productId);
52
53     if (salesDays < 10) {
54       throw new BadRequestException(`Not enough sales data to forecast. Minimum ${MIN_SALES} sales days required, Found ${salesDays}`);
55     }
56   }
57
58   private async getInitialSalesForProduct(productId: string): Promise<Sales[]> {
59     const sales = await this.prisma.sales.findMany({
60       where: { productId },
61       select: { date: true, quantity: true },
62     });
63
64     // Check for min sales days threshold
65     const salesDays = sales.length;
66
67     if (salesDays < MIN_SALES) {
68       throw new BadRequestException(`Not enough sales data to forecast. Minimum ${MIN_SALES} sales days required, Found ${salesDays}`);
69     }
70   }
71
72   private async getSalesDaysForProduct(productId: string): Promise<number> {
73     const sales = await this.prisma.sales.findMany({
74       where: { productId },
75       select: { date: true, quantity: true },
76     });
77
78     const dates = sales.map((s) => s.date);
79
80     const uniqueDates = new Set(dates);
81
82     const salesDays = uniqueDates.size;
83
84     return salesDays;
85   }
86 }
```

Figure 5-88: DemandPredictionService (1/3)

```

56     // Call FastAPI
57     try {
58       const response = await axios.post(
59         `${process.env.AI_BACKEND_URL}/demand`,
60         {
61           product_id: product.id,
62           sales: allPastSales,
63           months: 3, // Predict for next 3-months
64         },
65       );
66       const forecast = response.data;
67
68       // Add a lowAccuracy flag if sales history is small
69       const lowAccuracy = allPastSales.length < 50;
70
71       // Stocking recommendation (meaning how much more supplier should stock)
72       const currentStock = product.stock ?? 0;
73       const totalForecast = forecast.forecast.reduce(
74         (sum: number, t: { demand: number }) => sum + t.demand,
75         0,
76       );
77       const recommendedStock = Math.max(totalForecast - currentStock, 0);
78
79       return {
80         productName: product.name,
81         productFirstImageFileUrl: this.fileService.getFileUrl(
82           product.imagesFileNames[0],
83         ),
84         product_id: product.id,
85         ...forecast,
86         lowAccuracy,
87         salesCount: allPastSales.length,
88         currentStock,
89         totalForecast,
90         recommendedStock,
91       } as DemandPredictionResponseDto;
92     } catch (err: any) {
93       console.error('FastAPI request failed:', err.message);
94       throw new HttpException(
95         'Failed to get prediction from AI backend',
96         HttpStatus.BAD_GATEWAY,
97       );
98     }
99   }

```

Figure 5-89: DemandPredictionService (2/3)

```

101    async getDailySalesForProduct(productId: string) {
102      // Raw SQL query: group sales by order date (all time)
103      const dailySales = await this.prisma.$queryRaw` 
104        > SELECT
105          DATE(o."createdAt") AS date,
106          SUM(c1."quantity") AS quantity
107        FROM "Order" o
108        INNER JOIN "Cart" c ON o."cartId" = c."id"
109        INNER JOIN "CartBySupplier" cs ON cs."cartId" = c."id"
110        INNER JOIN "CartItem" ci ON ci."cartBySupplierId" = cs."id"
111        WHERE ci."productId" = ${productId}
112        GROUP BY DATE(o."createdAt")
113        ORDER BY date ASC;
114      `;
115
116      if (dailySales.length === 0) {
117        return [];
118      }
119
120      // Fill missing days with 0 (from first sale to today)
121      const result: { date: string; quantity: number }[] = [];
122      const firstDate = new Date(dailySales[0].date); // earliest sale date
123      const today = new Date();
124
125      for (let d = new Date(firstDate);
126        d <= today;
127        d.setDate(d.getDate() + 1)) {
128        const dayStr = d.toISOString().split('T')[0];
129        const entry = dailySales.find(
130          (s) => s.date.toISOString().split('T')[0] === dayStr,
131        );
132        if (!entry) {
133          result.push({
134            date: dayStr,
135            quantity: 0,
136          });
137        } else {
138          result.push({
139            date: dayStr,
140            quantity: entry.quantity,
141          });
142        }
143      }
144    }

```

Figure 5-90: DemandPredictionService (3/3)

FastAPI loads the pre-trained Prophet model, generates daily forecasts, and aggregates them into monthly demand. The NestJS service receives this raw output, computes a total forecast, compares it with currentStock, and calculates a recommendedStock value. It also sets a lowAccuracy flag if sales history is below 50 days. Finally, it enriches the response with productName, productFirstImageFileUrl (via signed R2 URL), and metadata before returning a DemandPredictionResponseDto.

```

src > demand-prediction > dist > ts demandPredictionResponse.dto.ts ...
  You last month | Author (You)
  1 import { ApiProperty } from '@nestjs/swagger';
  2
  3 export class ForecastMonthDto {
  4   @ApiProperty({
  5     example: '2023-01',
  6     description: 'The forecasted month in YYYY-MM format',
  7   })
  8   months: string;
  9
 10   @ApiProperty({
 11     example: 120,
 12     description: 'Total predicted demand for this month',
 13   })
 14   demand: number;
 15 }
 16
  You last month | Author (You)
 17 export class DemandPredictionResponseDto {
 18   @ApiProperty({
 19     example: 'Candle Holder',
 20     description: 'Name of the product being forecasted',
 21   })
 22   productName: string;
 23
 24   @ApiProperty({
 25     example: 'https://cdn.example.com/images/abc.jpg',
 26     description: 'URL of the first image of the product',
 27   })
 28   productFirstImageFileUrl: string;
 29
 30   @ApiProperty({
 31     example: '123',
 32     description: 'ID of the product being forecasted',
 33   })
 34   product_id: string;
 35
 36   @ApiProperty({
 37     type: [ForecastMonthDto],
 38     description:
 39       'List of monthly demand predictions for the requested horizon',
 40   })
 41   forecast: ForecastMonthDto[];
 42
 43   @ApiProperty({
 44     example: false,
 45     description:
 46       'Indicates if the prediction accuracy is low (true when sales history < 50 days)',
 47   })
 48   lowAccuracy: boolean;
 49
 50   @ApiProperty({
 51     example: 42,
 52     description: 'Total number of days of past sales data used in training',
 53   })
 54   salesCount: number;
 55
 56   @ApiProperty({
 57     example: 210,
 58     description:
 59       'How many additional units supplier should stock to meet forecasted demand',
 60   })
 61   recommendedStock: number;
 62 }

```

Figure 5-91: DemandPredictionResponseDto

Buyers and suppliers perform smart search via POST /smart-search, accepting a SmartSearchRequestDto with either itemId (for product-based search) or text (free-text query). The SmartSearchService resolves the target language using query parameters, headers, or user preference, ensuring multilingual relevance.

```

src > smart-search > controllers > ...
  1 import { Controller, Post, Body, Req, Query, Headers } from 'nestjs/common';
  2 import { Controller, Post, Body, Req, Query, Headers } from '@nestjs/common';
  3 import { ApisTags } from '@nestjs/swagger';
  4 import { SmartSearchRequestDto } from './dtos/smartSearchRequest.dto';
  5 import { SmartSearchService } from './smart-search.service';
  6 import { Request } from 'express';
  7
 8 You last month | Author (You)
 9 @ApisTags('Smart Search')
10 @Controller('smart-search')
11 export class SmartSearchController {
12   constructor(private readonly smartSearchService: SmartSearchService) {}
13
14   /** Helper function to determine target language */
15   private async resolveTargetLang(
16     req: Request,
17     lang?: string,
18     langHeader?: 'ar' | 'en',
19   ) {
20     let targetLang = lang || langHeader || 'en';
21
22     // Priority: query param > header > user preference > default
23     if (!lang) {
24       targetLang = lang;
25     } else if (langHeader) {
26       targetLang = langHeader;
27     } else if (req.query.lang) {
28       const user = await this.smartSearchService.getUserLanguage(
29         req.query.lang,
30       );
31       if (user) targetLang = user;
32     }
33
34     return targetLang;
35   }
36
37   @Post()
38   @ApiOperation('Smart Search')
39   @ApiResponse({ status: 200, type: SmartSearchRequestDto },
40   { status: 400, type: Req })
41   @Headers('accept-language': 'language': 'ar' | 'en',
42   { query: 'lang': 'lang': 'ar' | 'en',
43   })
44   const targetingLang = await this.resolveTargetLang(req.lang, langHeader);
45   return this.smartSearchService.getSimilarItemsDto(targetingLang);
46 }

```

Figure 5-92: SmartSearchController

If itemId is provided, the service validates the item (product or service) exists, is active, and belongs to an open supplier. It then fetches all active items of the same

```

src > smart-search > dist > ts smartSearchRequest.dto.ts ...
  You last month | Author (You)
  1 import { IsOptional, IsString, ValidateIf } from 'class-validator';
  2 import { ApiPropertyOptional } from '@nestjs/swagger';
  3
 4 You last month | Author (You)
 5 export class SmartSearchRequestDto {
 6   @ApiPropertyOptional({
 7     description:
 8       'ID of an existing item (product or service) to find similar alternatives',
 9   })
10   @IsOptional()
11   @IsString()
12   itemId: string;
13
14   @ApiPropertyOptional({
15     description:
16       'Name of the item to search by (required if no itemId is provided),
17       example: "Wooden Brush"',
18   })
19   @ValidateIf((o) => !o.itemId)
20   @IsString()
21   text: string;
22 }

```

Figure 5-93: SmartSearchRequestDto

type (products or services) and ensures each has a precomputed LaBSE embedding stored in the ItemEmbedding table. Missing embeddings are generated on-demand via a call to \${AI_BACKEND_URL}/embed, which returns a 768-dimensional vector.

```

1  You, 5 days ago | author (You)
2  import {
3      BadRequestException,
4      HttpException,
5      HttpStatus,
6      Injectable,
7  } from '@nestjs/common';
8  import { PrismaService } from 'src/prisma/prisma.service';
9  import { SmartSearchRequestDto } from './dto/smartSearchRequest.dto';
10 import { SmartSearchResponseDto } from './dto/smartSearchResponse.dto';
11 import { ItemType, SupplierStatus } from '@prisma/client';
12 import axios from 'axios';
13
14 export class SmartSearchService {
15     constructor(private readonly prisma: PrismaService) {}
16
17     /** Helper: fetch user's preferred language */
18     async getUserLanguage(userId: string): Promise<'ar' | 'en' | null> {
19         const user = await this.prisma.user.findUnique({
20             where: { id: userId },
21             select: { preferredLanguage: true },
22         });
23         const lang = user?.preferredLanguage?.toLowerCase();
24         return lang === 'ar' || lang === 'en' ? lang : null;
25     }
26
27     async getSimilarItems() {
28         const { dto, targetLang?: 'ar' | 'en', ...args } = args;
29         const [Promise<SmartSearchResponseDto>] = [
30             const { itemId, text } = dto;
31
32             // 1. Validate at least itemId or itemName must exist
33             if (!itemId && !text) {
34                 throw new BadRequestException(
35                     'Either itemId or text must be provided',
36                 );
37             }
38         ];
39
40         let item;
41         let itemType;
42         let embedding;
43         let productPayloads: any[] = [];
44         let servicePayloads: any[] = [];
45
46         // 2. If itemId is given, validate it exists
47         if (itemId) {
48             // try both tables
49             const product = await this.prisma.productfindFirst({
50                 where: {
51                     id: itemId,
52                     isDeleted: false,
53                     supplier: {
54                         isStoreClosed: false,
55                         status: SupplierStatus.ACTIVE,
56                     },
57                 },
58             });
59             const service = await this.prisma.servicefindFirst({
60                 where: {
61                     id: itemId,
62                     isDeleted: false,
63                     supplier: {
64                         isStoreClosed: false,
65                         status: SupplierStatus.ACTIVE,
66                     },
67                 },
68             });
69             const include: {
70                 supplier: { include: { user: true } },
71                 category: true,
72             };
73             include.supplier!.include!.user = true;
74             include.category = true;
75             const servicePayload = {
76                 item: service,
77                 itemType: ItemType.SERVICE,
78                 ...include,
79             };
80             productPayloads.push(servicePayload);
81         }
82         else {
83             const products = await this.prisma.product.findMany({
84                 where: {
85                     isDeleted: false,
86                     supplier: {
87                         isStoreClosed: false,
88                         status: SupplierStatus.ACTIVE,
89                     },
90                 },
91             });
92             const include: { category: true };
93             include.category = true;
94             const productPayloads = await Promise.all(
95                 products.map(async (p) => ({
96                     id: p.id,
97                     name: p.name,
98                     description: p.description ?? null,
99                     category_name: p.category!.name,
100                    embedding: await this.ensureEmbedding({
101                        id: p.id,
102                        name: p.name,
103                        description: p.description,
104                        category: p.category,
105                        type: ItemType.PRODUCT,
106                    }),
107                })),
108            );
109        }
110    }
111}

```

Figure 5-94: SmartSearchService (1/8)

Figure 5-95: SmartSearchService (2/8)

```

199     } else if (service) {
200       item = service;
201       itemType = ItemType.SERVICE;
202
203       // Fetch all services
204       const allServices = await this.prisma.service.findMany({
205         where: {
206           isDeleted: false,
207           supplier: {
208             isStoreClosed: false,
209             status: SupplierStatus.ACTIVE,
210           },
211         },
212         include: { category: true },
213       });
214
215       servicePayloads = await Promise.all(
216         allServices.map(async (s) => ({
217           id: s.id,
218           name: s.name,
219           description: s.description ?? null,
220           category_name: s.category.name,
221           embedding: await this.ensureEmbedding({
222             id: s.id,
223             name: s.name,
224             description: s.description,
225             category: s.category,
226             type: itemType.SERVICE,
227           })),
228         )),
229     } else {
230       throw new BadRequestException(
231         'No item found with the given ID',
232       );
233     }
234
235     embedding = await this.ensureEmbedding({
236       id: itemId,
237       name: item.name,
238       description: item.description,
239       category: item.category,
240       type: itemType,
241     });
242
243     if (!embedding) {
244       throw new BadRequestException(
245         `Embedding not found for item ${itemId} (${itemType})`,
246       );
247     }
248   }

```

Figure 5-96: SmartSearchService (3/8)

```

301   // Fill missing request fields from DB
302   if (!text) dto.text = item.name;
303
304   if (!itemId) {
305     const [allProducts, allServices] = await Promise.all([
306       this.prisma.product.findMany(),
307       this.prisma.service.findMany()
308     ]);
309
310     where: {
311       isDeleted: false,
312       supplier: {
313         isStoreClosed: false,
314         status: SupplierStatus.ACTIVE,
315       },
316     },
317     include: { category: true },
318   };
319
320   this.prisma.service.findMany({
321     where: {
322       isDeleted: false,
323       supplier: {
324         isStoreClosed: false,
325         status: SupplierStatus.ACTIVE,
326       },
327     },
328     include: { category: true },
329   });
330
331   productPayloads = await Promise.all(
332     allProducts.map(async (p) => ({
333       id: p.id,
334       name: p.name,
335       description: p.description ?? null,
336       category_name: p.category.name ?? null,
337       embedding: await this.ensureEmbedding({
338         id: p.id,
339         name: p.name,
340         description: p.description,
341         category: p.category,
342         type: itemType.PRODUCT,
343       })),
344     ));
345
346   servicePayloads = await Promise.all(
347     allServices.map(async (s) => ({
348       id: s.id,
349       name: s.name,
350       description: s.description ?? null,
351       category_name: s.category.name ?? null,
352       embedding: await this.ensureEmbedding({
353         id: s.id,
354         name: s.name,
355         description: s.description,
356         category: s.category,
357         type: itemType.SERVICE,
358       })),
359     ));
360   }

```

Figure 5-97: SmartSearchService (4/8)

The service builds a SimilarSearchRequest payload including:

- text: query string (item name if itemId given)
- item_id: optional
- embedding: precomputed query vector
- candidates: array of all items with id, name, description, category_name, and embedding

This payload is sent to \${AI_BACKEND_URL}/similar-search. FastAPI computes cosine similarity between the query and all candidates, returning a ranked list of id and score.

```

221 // Build payload for FastAPI
222 let payload;
223 if (itemType === ItemType.PRODUCT) {
224   payload = {
225     text: dto.text,
226     item_id: item.id,
227     embedding,
228     candidates: productPayloads,
229   };
230 } else if (itemType === ItemType.SERVICE) {
231   payload = {
232     text: dto.text,
233     item_id: item.id,
234     embedding,
235     candidates: servicePayloads,
236   };
237 } else {
238   // If itemType is unknown (free-text search with no item), send both products and services
239   payload = {
240     text: dto.text,
241     item_id: null,
242     embedding: null,
243     candidates: [...productPayloads, ...servicePayloads],
244   };
245 }
246
247 // Call FastAPI
248 let aiResults: any[] = [];
249 try {
250   const response = await axios.post(
251     `${process.env.AI_BACKEND_URL}/similar-search`,
252     payload,
253   );
254   aiResults = response.data;
255 } catch (err: any) {
256   console.error(`FastAPI request failed! ${err.message}`);
257   throw new HttpException(
258     'AI Backend is unavailable',
259     HttpStatus.BAD_GATEWAY,
260   );
261 }
262
263 // Validate AI response
264 if (!aiResults || !Array.isArray(aiResults)) {
265   aiResults.every((r) => r.id >> 0 && r.rank === 'medium')
266 } else {
267   throw new HttpException(
268     'Invalid AI response format',
269     HttpStatus.BAD_GATEWAY,
270   );
271 }
272
273

```

Figure 5-98: SmartSearchService (5/8)

```

275 // Fetch details from DB based on resolvedType
276 let items: any[] = [];
277 if (itemType === ItemType.PRODUCT) {
278   items = await this.prisma.product.findMany({
279     where: { id: { in: aiResults.map(r: any) => r.id } },
280     include: {
281       supplier: { include: { user: true } },
282       category: true,
283     },
284   });
285 } else if (itemType === ItemType.SERVICE) {
286   items = await this.prisma.service.findMany({
287     where: { id: { in: aiResults.map(r: any) => r.id } },
288     include: {
289       supplier: { include: { user: true } },
290       category: true,
291     },
292   });
293 } else {
294   // If type unknown, fetch both products and services concurrently
295   const [products, services] = await Promise.all([
296     this.prisma.product.findMany({
297       where: { id: { in: aiResults.map(r: any) => r.id } },
298       include: {
299         supplier: { include: { user: true } },
300         category: true,
301       },
302     }),
303     this.prisma.service.findMany({
304       where: { id: { in: aiResults.map(r: any) => r.id } },
305       include: {
306         supplier: { include: { user: true } },
307         category: true,
308       },
309     }),
310   ]);
311   items = [...products, ...services];
312 }
313
314 // Merge ranking score from FastAPI with DB items
315 return items
316   .map(item => {
317     const match = aiResults.find((r: any) => r.id === item.id);
318     if (!match) return null; // Filter out invalid IDs
319     return { text: dto.text, item, rank: match.rank };
320   })
321   .filter(Boolean) as SmartSearchResponseDto[];
322

```

Figure 5-99: SmartSearchService (6/8)

```

324 // --- Embedding helpers
325 private async ensureEmbedding(item: {
326   id: string;
327   name: string;
328   description?: string | null;
329   category?: { name: string | null };
330   type: itemType;
331 }) {
332   // Try to fetch existing embedding
333   let embeddingRecord = await this.prisma.itemEmbedding.findUnique({
334     where: {
335       itemId_itemType: {
336         itemId: item.id,
337         itemType: item.type,
338       },
339     },
340   });
341
342   if (!embeddingRecord) {
343     // If not found, generate & store it
344     await this.generateAndStoreEmbedding({
345       itemId: item.id,
346       itemType: item.type,
347       name: item.name,
348       description: item.description ?? null,
349       categoryName: item.category?.name ?? '',
350     });
351
352     // Fetch again
353     embeddingRecord = await this.prisma.itemEmbedding.findUnique({
354       where: {
355         itemId_itemType: {
356           itemId: item.id,
357           itemType: item.type,
358         },
359       },
360     });
361   }
362
363   return embeddingRecord?.embedding ?? null;
364 }

```

Figure 5-100: SmartSearchService (7/8)

```

366 async generateAndStoreEmbedding(params: {
367   itemId: string;
368   itemType: itemType;
369   name: string;
370   categoryName: string;
371   description: string | null;
372 }): Promise<void> {
373   const { itemId, itemType, name, description, categoryName } = params;
374
375   try {
376     // 1. Call FastAPI to generate embedding
377     const response = await axios.post(
378       `${process.env.AI_BACKEND_URL}/embed`,
379       {
380         name,
381         description: description ?? null,
382         category_name: categoryName,
383       },
384     );
385
386     const embedding = response.data?.embedding;
387     if (!embedding || !Array.isArray(embedding)) {
388       throw new Error('Invalid embedding response from AI backend');
389     }
390
391     // 2. Upsert into DB
392     await this.prisma.itemEmbedding.upsert({
393       where: {
394         itemId_itemType: {
395           itemId,
396           itemType,
397         },
398       },
399       update: {
400         embedding,
401       },
402       create: {
403         itemId,
404         itemType,
405         embedding,
406       },
407     });
408   } catch (err: any) {
409     console.error(`Embedding generation failed! ${err.message}`);
410     throw new HttpException(
411       'Failed to generate Embedding for item',
412       HttpStatus.BAD_GATEWAY,
413     );
414   }
415 }

```

Figure 5-101: SmartSearchService (8/8)

NestJS receives this list, fetches full item details from the database (including supplier, category, images), and merges them with AI rankings. The final response is an array of SmartSearchResponseDto, each containing the original text, enriched item (product/service), and rank.

```

src > smart-search > dtos > ts smartSearchResponse.dto.ts > ...
  You last month | author (You)
1 import { ApiProperty, getSchemaPath } from '@nestjs/swagger';
2 import { ProductResponseDto } from 'src/product/dtos/productResponse.dto';
3 import { ServiceResponseDto } from 'src/service/dtos/serviceResponse.dto';
4
5 You, last month | author (You)
6 export class SmartSearchResponseDto {
7   @ApiProperty({
8     description: 'The text used by the backend for this smart search',
9     example: 'Organic Honey 500g',
10    })
11   text: string;
12
13   @ApiProperty({
14     description: 'The matched item (product or service)',
15     oneOf: [
16       { $ref: getSchemaPath(ProductResponseDto) },
17       { $ref: getSchemaPath(ServiceResponseDto) },
18     ],
19   })
20   item: ProductResponseDto | ServiceResponseDto;
21
22   @ApiProperty({
23     description:
24       'Ranking position among the top results (1 = most similar)',
25     example: 1,
26   })
27   rank: number;
}

```

Figure 5-102: SmartSearchResponseDto

5.2.2.6 Deployment and DevOps

The backend of the Silah platform was deployed early in the development lifecycle to enable seamless frontend integration and eliminate local environment friction. While the frontend team could have run the NestJS server locally, doing so would have required each developer to install PostgreSQL, manage environment variables, apply database migrations, install extensions such as pg_trgm, and maintain consistent API keys; tasks that are error-prone and time-consuming. By deploying the backend to a centralized production-like server, we ensure consistency, reliability, and accelerated collaboration, allowing frontend developers to focus on UI/UX while making real API calls from day one.

The backend (silah-backend) and AI service (silah-ai) were both deployed on the same DigitalOcean droplet (silah-site-server), following the same initial SSH-based provisioning process described in the AI Deployment section. The droplet runs Ubuntu 24.04 with 2 vCPUs, 4 GB RAM, and 80 GB SSD. After cloning the repository via git clone, the project dependencies were installed using “npm ci”, and the application was built with “npm run build”.

Unlike the AI backend, which required a Python virtual environment and screen for detached execution, the NestJS backend was managed using PM2, a production process manager for Node.js. A .env file containing all secrets (database URL, API keys, etc.) was securely uploaded to the server and loaded at runtime. The PostgreSQL database was created manually, and the pg_trgm extension was enabled to support fuzzy search on product names, supplier names, and categories with the

command “CREATE EXTENSION IF NOT EXISTS pg_trgm;”. This extension powers typo-tolerant search queries using ILIKE with trigram similarity, significantly improving user experience during product discovery.

To ensure zero-downtime deployments and automated consistency, a GitHub Actions CI/CD pipeline was implemented. The workflow triggers on every push to the main branch and executes the following script:

```

github > workflows > %_deploy.yml > (1) jobs > (1) deploy > (1) steps > (1) > (1) with > script
  name: Deploy to Server
  on:
    push:
      branches:
        - main
  jobs:
    deploy:
      runs-on: ubuntu-latest
      steps:
        - name: Checkout code
          uses: actions/checkout@v4
        - name: Deploy via SSH
          uses: appleboy/ssh-action@v1.0.3
          with:
            host: ${{ secrets.SERVER_HOST }}
            username: ${{ secrets.SERVER_USER }}
            key: ${{ secrets.SERVER_SSH_KEY }}
            script: |
              set -eu pipefail
              cd /root/silah-backend
              git fetch origin main --prune
              git reset --hard origin/main
              npm ci --prefer-offline
              npm install
              npx prisma migrate deploy
              pm2 reload silah-backend --update-env || pm2 start ecosystem.config.js

```

Figure 5-103: GitHub Actions Job Script

The ecosystem.config.js file defines the PM2 process:

```

ecosystem.config.js > ...
  module.exports = {
    apps: [
      {
        name: 'silah-backend', // The name that shows in pm2 list
        script: 'dist/src/main.js', // The entry point (after build)
        instances: 1, // Or "max" for all CPU cores
        autorestart: true, // Restart if it crashes
        watch: false, // Set true if you want auto-restart on code changes
        max_memory_restart: '500M', // Restart if memory exceeds this
        env: {
          NODE_ENV: 'development',
        },
        env_production: {
          NODE_ENV: 'production',
        },
      ],
    ];
  };

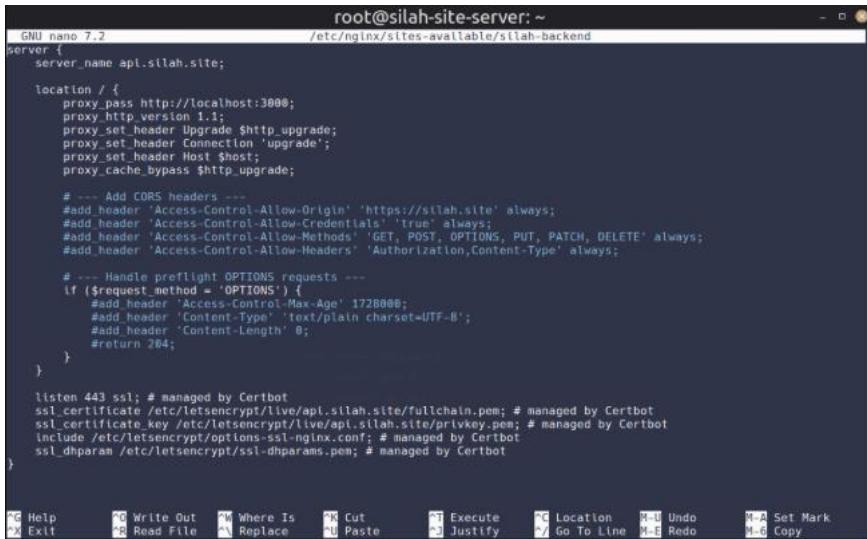
```

Figure 5-104: PM2 Process

This pipeline ensures that every code change is automatically built, migrated, and deployed with minimal manual intervention, reducing human error and enabling rapid iteration.

NGINX operates as a reverse proxy and SSL terminator, exposing only the NestJS backend to the public internet while keeping the FastAPI AI service completely internal and inaccessible from outside the server. This configuration routes all

incoming traffic from <https://api.silah.site> directly to the NestJS application running on localhost:3000, ensuring a clean and secure public API surface.



```
root@silah-site-server: ~
GNU nano 7.2
server {
    server_name api.silah.site;

    location / {
        proxy_pass http://localhost:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;

        # --- Add CORS headers ---
        #add_header 'Access-Control-Allow-Origin' 'https://silah.site' always;
        #add_header 'Access-Control-Allow-Credentials' 'true' always;
        #add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, PUT, PATCH, DELETE' always;
        #add_header 'Access-Control-Allow-Headers' 'Authorization,Content-Type' always;

        # --- Handle preflight OPTIONS requests ---
        if ($request_method = 'OPTIONS') {
            #add_header 'Access-Control-Max-Age' 1728000;
            #add_header 'Content-Type' 'text/plain charset=UTF-8';
            #add_header 'Content-Length' 0;
            #return 204;
        }
    }

    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/api.silah.site/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/api.silah.site/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}

FG Help   CG Write Out   CK Where Is   CK Cut   CK Execute   FC Location   M-U Undo   M-A Set Mark
EX Exit   CR Read File   CR Replace   CU Paste   CJ Justify   CG Go To Line   M-F Redo   M-G Copy
```

Figure 5-105: NGIX Configuration for silah-backend

HTTPS termination is handled entirely by NGINX using Let's Encrypt certificates automatically managed and renewed by Certbot, guaranteeing that all client communication is encrypted in transit. WebSocket support is preserved through proper forwarding of Upgrade and Connection headers, enabling real-time features such as chat messaging and Server-Sent Events (SSE) for live notifications without interruption.

The AI backend remains fully shielded from external access, as it runs on localhost:8000 and is never exposed through NGINX. Instead, the NestJS application communicates with it internally using the AI_BACKEND_URL environment variable (set to <http://localhost:8000>), bypassing the reverse proxy entirely. This design enforces zero exposure of the inference layer, reduces attack surface, and simplifies firewall rules; only port 443 is open to the public.

By centralizing all client-facing traffic through a single, secure entry point, the architecture supports future scalability (such as adding load balancing, rate limiting, or caching) without requiring changes to frontend code or client applications. This layered, defense-in-depth approach aligns with production best practices while maintaining simplicity and performance.

5.2.2.7 Backend Design: Strengths, Trade-offs, and Path Forward

The Silah backend represents a robust, full-featured, and production-ready NestJS application that successfully powers a bilingual B2B marketplace with real-time communication, AI integration, payment processing, and regulatory compliance. Its modular architecture, built on clean separation of concerns, Prisma ORM, and decoupled AI inference, enabled rapid development, seamless frontend integration, and early deployment; critical for a time-constrained graduation project. The system's scalability is evident in its use of WebSocket rooms, SSE streams, and signed R2 URLs, while maintainability is supported by comprehensive Swagger documentation, automated CI/CD, and centralized environment management. The decision to deploy the backend early on a DigitalOcean droplet with NGINX, PM2, and GitHub Actions transformed a potential integration bottleneck into a shared, reliable foundation, allowing the frontend team to build against real APIs from day one.

Despite these strengths, the implementation reflects pragmatic engineering trade-offs made under real-world constraints: limited time and evolving requirements. The most significant technical debt lies in the lack of abstraction between Product and Service entities. Although both share nearly identical lifecycle patterns (creation, update, soft deletion, media handling, and search logic) the system duplicates logic across two parallel modules instead of introducing a shared Item base layer with abstract services, DTOs, and database triggers. This duplication violates the DRY (Don't Repeat Yourself) principle and increases maintenance overhead. However, this decision was deliberate and justified: at the project's outset, the full scope of shared behavior was not yet clear. Implementing abstraction prematurely would have risked over-engineering. Once the pattern emerged, refactoring was deprioritized in favor of delivering working features on time; a rational choice for a graduation project with fixed deadlines and no post-submission maintenance commitment.

Code duplication extends beyond Product and Service. Several utility methods (such as file validation, translation fallbacks, and pagination logic) were reimplemented across services rather than extracted into shared helpers. Some service files exceed 400 lines, with methods spanning over 100 lines, reducing readability and testability. A more scalable approach would involve sub-services (e.g., `ProductCreationService`, `ProductResponseMapper`) or strategy patterns, but

these were deferred to prioritize functional completeness.

Similarly, Swagger documentation, while comprehensive in schema, lacks detailed error response descriptions for many endpoints. A 400 Bad Request is documented, but the specific invalid fields (e.g., “price must be positive”) are not enumerated, forcing clients to infer from DTO validation. The auth module sets a high documentation standard, but others fall short; a reflection of documentation fatigue toward the project’s end.

Deletion semantics also reveal intentional simplifications. Users cannot delete their accounts, and bids, offers, and invoices are immutable post-creation. These constraints were introduced to avoid complex cascading logic and ensure auditability, but they limit real-world flexibility. In a production system, soft-deletion with revision history, user-initiated account removal (with GDPR compliance), and editable bids (with change tracking) would be expected. These omissions were not oversights but conscious scope reductions to meet project deadlines.

A subtle but critical technical debt lies in timestamp inconsistency during multi-entity transactions. When a single business action (such as an invoice payment) affects multiple tables (e.g., Payment, Invoice, Notification), the createdAt timestamps are generated independently by Prisma, resulting in microsecond-level drift (e.g., 2025-04-05T10:00:00.123Z, ...124Z, ...125Z). This makes transactional debugging, audit trails, and replay analysis significantly harder. A best practice would be to capture a single transactionTimestamp at the service layer and apply it uniformly across all records. This oversight was not intentional. At the time of implementation, we were unaware of the long-term implications of Prisma’s default behavior. The issue was only recognized after backend development was complete, upon reading a post discussing transactional consistency in ORMs. Under tight deadlines, no prior experience highlighted the need for manual timestamp synchronization, and the system relied on Prisma’s automatic createdAt population for simplicity. This is a valuable lesson learned; not a justification, but a growth point for future projects.

These trade-offs do not diminish the project’s success. They reflect a mature understanding of engineering reality: perfection is the enemy of progress. The team correctly prioritized working software, early integration, and submission requirements over architectural purity. The backend delivered all core functionalities (real-time

chat, AI-powered search, demand forecasting, payment flows, and multilingual support) within a single semester. For beginners, this is a remarkable achievement.

From a best practices perspective, several opportunities for improvement emerged during code review.

The DemandPredictionService and SmartSearchService perform heavy orchestration (data fetching, validation, AI calls, enrichment) in single methods, making them hard to unit test. Extracting pure functions (e.g., aggregateMonthlyDemand) would improve testability. The ensureEmbedding logic in SmartSearchService is idempotent but not cached aggressively; a background job to pre-generate embeddings for all items on deployment would reduce latency. Additionally, error handling in AI calls uses `HttpException` with `BAD_GATEWAY`, but retry logic or circuit breakers could enhance resilience. Logging is minimal, limited to error-level events for post-mortem review. Structured logs with correlation IDs (e.g., `requestId`, `transactionId`) would dramatically improve debugging in production, especially for distributed flows involving AI, payments, or notifications.

Performance was deprioritized by design. In software engineering, it is well understood that code must be written three times: first to make it work, second to make it readable, and third to make it fast. In a graduation project, the team barely had time for the first pass. Revisiting every line for optimization, let alone conducting stress testing or profiling, was infeasible. No performance benchmarks were run, and no caching strategies were implemented. This was not negligence; it was scope discipline. Performance was not a requirement, and optimizing prematurely would have delayed delivery.

Looking forward, several enhancements could elevate the system to enterprise grade. Redis caching for frequent queries (e.g., product listings, category trees) would reduce database load. Prometheus + Grafana monitoring would provide visibility into API latency, error rates, and AI inference times. Elasticsearch or Meilisearch could replace `pg_trgm` for faster, more flexible full-text search. Background jobs via BullMQ (a Redis-backed job queue) could handle asynchronous tasks such as embedding generation, email digests, invoice reminders, and analytics aggregation. Unlike simple cron jobs, BullMQ supports retries, priority queues, rate limiting, and distributed workers, making it ideal

for microservices or high-throughput environments. Finally, event sourcing for orders and RFPs would replace direct state updates with an immutable log of business events (e.g., OrderCreated, RFPAccepted, PaymentReceived). This approach enables perfect audit trails, flexible analytics (e.g., average time-to-acceptance), and eventual consistency across distributed systems. The current state of any entity can be rebuilt by replaying its event stream, supporting time-travel debugging and schema evolution. While ideal for enterprise systems, event sourcing was rightly excluded from this project due to its complexity and our focus on delivering core functionality on time.

In reflection, this project taught that software is a series of trade-offs, not a checklist of best practices. SOLID principles are aspirational, but YAGNI (You Aren't Gonna Need It) and KISS (Keep It Simple, Stupid) are survival tools under pressure. The decision to deploy early was the single most impactful choice, proving that shared infrastructure accelerates collaboration. The technical debt is not a failure; it is documentation of learning. Every duplicated line, every long method, every missing error message, every microsecond drift in timestamps is a lesson in what to refactor next time. For a graduation project, the backend is not just functional; it is instructive, resilient, and a foundation for future growth.

5.2.3 Frontend Implementation

The frontend is the user-facing layer of Silah; the part of the system that receives human input, renders data coming from the backend, and orchestrates the interactions that form the user experience. It was implemented using React and built with Vite in order to achieve a responsive, modular, and maintainable interface.

React was chosen because its component-oriented model maps directly to the needs of a medium-sized, multi-page application: instead of duplicating markup and event logic across many pages, user interface elements are expressed as small, reusable components that encapsulate structure, presentation, and behavior. This approach reduces repetition, improves predictability when components are composed, and simplifies maintenance: a button, input field, or card implemented once can be used in many places with different data, decreasing the likelihood of subtle inconsistencies and saving development time. For readers unfamiliar with web internals, the

Document Object Model (DOM) is the browser’s tree representation of a page’s HTML elements; manipulating the DOM directly with raw JavaScript can be costly because each change may trigger layout recalculation and repainting. React mitigates these costs with a virtual DOM, an in-memory representation of the UI that React updates first, computes the minimal set of real-DOM changes (the “diff”), and applies only those changes. This diffing strategy typically produces faster and more predictable rendering for dynamic interfaces, particularly when many small updates or frequent state changes occur (for example, live notifications or chat messages).

Beyond rendering performance and reuse, React’s mature ecosystem and widespread industry adoption were decisive factors. The abundance of well-maintained libraries for routing, internationalization, testing, accessibility, UI primitives, and developer tooling significantly accelerated development and reduced the need to reinvent foundational features. Choosing React also served an educational objective; because React is widely used in production environments, learning it during the graduation project increased our readiness for real-world software roles. Alternative technologies were considered but found less aligned with the project’s constraints and goals. Plain HTML, CSS, and vanilla JavaScript were quickly dismissed for a project of this scale; implementing routing, component reuse, state synchronization, and real-time updates by hand across forty-plus pages or so would have been error-prone and time consuming.

When evaluating alternative frameworks, we also considered Angular and Next.js, as both are well-established technologies within the JavaScript ecosystem. Angular was recognized for its completeness and structure, as it provides a full-featured framework that includes routing, dependency injection, and built-in state management. However, despite its strong architecture, Angular is less commonly adopted in the industry compared to React, especially in the local and regional job markets. We also observed through online discussions and developer surveys that Angular is often described as having a steeper learning curve due to its extensive use of TypeScript decorators and its strict architectural conventions. Although we did not attempt to use Angular ourselves, the combination of these reports and its lower popularity made it a less practical choice for our educational goals. Choosing React allowed us to focus on a more widely used framework while reducing the learning overhead and maximizing future career relevance.

Next.js was also evaluated, as it is built on top of React and extends its capabilities to support full-stack web development. Specifically, Next.js combines both the frontend and backend within a single project, providing features like server-side rendering (SSR) and static site generation (SSG). These techniques improve performance and SEO for certain web applications but were unnecessary for our use case, as Silah already had a dedicated backend implemented in NestJS. Incorporating Next.js would have introduced an additional layer of complexity, merging frontend and backend concerns in a way that contradicted our project's modular design. As explained in the beginning of this Section 5.2, our team intentionally divided the system into three separate projects (silah-frontend, silah-backend, and silah-ai) to improve collaboration, maintain clarity of responsibility, and optimize hardware usage during development. Using Next.js would have undermined this separation by forcing both frontend and backend logic into a single codebase. Moreover, Next.js requires prior understanding of React, which would have further extended the learning curve. For these reasons, and to stay aligned with our initial architecture decisions, we opted to build the frontend using plain React instead of Next.js.

In summary, React provided the necessary balance between flexibility, scalability, and maintainability for the frontend of Silah. Its component-based model, efficient rendering, and mature ecosystem aligned well with our goals and the project's modular architecture. Having explained the rationale behind the technology selection and its advantages over alternative frameworks, the following section will illustrate how these concepts were applied in practice by presenting the structure of the frontend application and discussing its key implementation aspects in detail.

5.2.3.1 Project Structure and Application Logic

The frontend codebase is structured in a modular and maintainable way that reflects React's component-based design philosophy and ensures a clear separation of concerns. This organization simplifies development, testing, and future scalability by grouping related functionality together. The root directory contains configuration and build files, while the src/ folder houses all source code responsible for the

application's functionality, appearance, and user interaction. Figure 1 illustrates the overall structure of the frontend project.

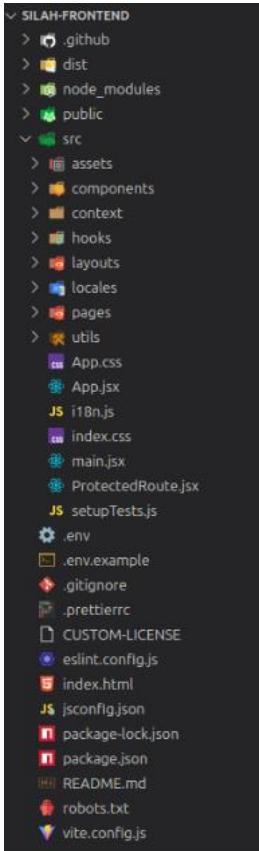


Figure 5-106: React Project Structure

At the root level, configuration and environment setup files define the project's dependencies, build behavior, and development standards. The package.json file lists all libraries used in the project and defines scripts for building, serving, and linting the code. vite.config.js specifies Vite's build configuration, which optimizes both the development and production environments. The jsconfig.json file is used to define custom path aliases, improving code readability and simplifying imports, while eslint.config.js ensures consistent code quality and formatting across contributors. Static files such as images and icons are stored in the public/ directory, which is copied directly to the production build without processing.

The main source code resides inside the src/ folder, where the logical structure of the frontend is defined. The pages/ directory contains page-level components that correspond to the system's main routes, including login, signup, and home pages for different user roles. Each page component encapsulates its own logic and styling, allowing it to be developed and maintained independently.

Reusable elements that appear across multiple pages are stored in the components/ directory. This includes elements such as buttons, input fields, banners, and notification cards. Following React's modularity principles, each component is responsible for a specific piece of functionality and presentation, enabling consistency and reducing duplication throughout the project.

The layouts/ directory defines layout templates for the different user roles (Buyer, Supplier, Guest, and Public). Each layout manages navigation bars, headers, and general structure for its respective role, ensuring a unified and coherent interface across related pages.

Global state management is handled in the context/ directory. This includes the AuthContext.jsx, which maintains the authentication state, user roles, and token management, as well as the notification context, which handles real-time pop-ups of newly received notifications. This approach centralizes shared logic and prevents unnecessary state duplication across components.

The hooks/ directory contains custom React hooks that encapsulate reusable logic. For example, useNotifications manages notification behavior across the system and interacts with both the context layer and the backend.

Localization and bilingual support are managed through i18n.js and the locales/ directory. The latter includes subdirectories for Arabic (ar/) and English (en/), each containing JSON files that define the language resources for different pages. This structure allows runtime switching between Right-to-Left (RTL) and Left-to-Right (LTR) modes based on the selected language.

Styling is applied through modular CSS files (e.g., Component.module.css) scoped to their corresponding components, which ensures maintainable and encapsulated styles. Shared or global styles (such as typography and headers) live in App.css and index.css to maintain a clean separation between component-specific and application-wide styles.

The entry point of the frontend is defined in the main.jsx file, which initializes React and renders the root component of the application. It also applies global providers such as the authentication and routing contexts, ensuring that authentication state, navigation, and translation capabilities are available across all pages.

```

src > main.jsx
You, 3 weeks ago | 3 authors (You and others)
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import { BrowserRouter } from 'react-router-dom';
4
5 import './i18n';
6
7 import App from './App';
8 import './index.css';
9 import { AuthProvider } from './context/AuthContext';
10
11 ReactDOM.createRoot(document.getElementById('root')).render(
12   <React.StrictMode>
13     <BrowserRouter>
14       <AuthProvider>
15         <App />
16       </AuthProvider>
17     </BrowserRouter>
18   </React.StrictMode>,
19 );

```

Figure 5-107: main.jsx file

The App.jsx file defines the main behavior of the application; including how pages are routed, how access is protected based on user roles, and how global wrappers such as the authentication and notification contexts are applied. In Silah, this file plays a critical role because routing, language direction, and layout composition all depend on it.

In React projects, routing is typically implemented manually by explicitly importing each page and defining its path, as shown in the following example:

```

import { Routes, Route } from 'react-router-dom';
import Home from './pages/Home';
import Login from './pages/Login';
import NotFound from './pages/NotFound';

export default function App() {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/login" element={<Login />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  );
}

```

Figure 5-108: Simple App.jsx Example

This traditional approach works well for small projects, but it becomes inefficient for larger applications developed collaboratively. Every time a developer adds or renames a page, they must manually import it and register its route inside App.jsx. In a multi-developer environment using GitHub, this creates a recurring source of merge conflicts.

A GitHub conflict occurs when two or more people modify the same file on different branches before merging. Even if the changes occur in different parts of the file (such as adding a space, indentation, or new import line), GitHub cannot automatically decide which version is correct. The conflict must then be resolved manually, which is time-consuming and error-prone.

To eliminate this issue and streamline development, we implemented dynamic routing. Instead of importing pages manually, the application automatically scans the pages/ directory and generates routes based on the file paths. The logic uses `import.meta.glob()` to detect every `.jsx` file under `pages/`, convert its folder structure into a route path, and dynamically import it when the user navigates to that route.

This technique ensures that adding a new page (for example, `pages/About/About.jsx`) automatically creates the route `/about` without modifying any central file. As a result, developers can work in parallel without touching shared files like `App.jsx`, greatly reducing the likelihood of GitHub conflicts and improving workflow efficiency.

```

git: App.jsx > App.jsx
You, yesterday | 4 authors | You and others
1 // Import React
2 import { Routes, Route, Navigate } from 'react-router-dom';
3 import { useTranslation } from 'react-i18next';
4 import { useAuth } from './context/AuthContext';
5 import ProtectedRoute from './ProtectedRoute';
6 import { ToastProvider } from '@context/NotificationPopupToast/NotificationContext';
7 import NotificationListener from '@components/NotificationListener';
8
9 // Layouts
10 import PublicLayout from './layouts/PublicLayout';
11 import SharedLayout from './layouts/SharedLayout';
12 import BuyerLayout from './layouts/BuyerLayout';
13 import SupplierLayout from './layouts/SupplierLayout';
14
15 // Pages
16 import Landing from './pages/Landing/Landing';
17 import NotFound from './pages/NotFound/NotFound';
18
19 const pages = import.meta.glob('./pages/**/*.{js,jsx}');
20
21 export default function App() {
22   const { i18n } = useTranslation();
23   const { role, loading } = useAuth();
24
25   if (loading) return null;
26
27   // Determine theme (buyer = blue, supplier = purple)
28   const isBuyer = role === 'buyer';
29
30   const redirectByRole = () => {
31     if (role === 'buyer') return Navigate to "/buyer/homepage" replace '/';
32     if (role === 'supplier')
33       return Navigate to "/supplier/overview" replace '/';
34     return Navigate to "/landing" replace '/';
35   };
36
37   const layoutRoutes = { public: [], shared: [], buyer: [], supplier: [] };
38
39   Object.entries(pages).forEach(([filePath, resolve]) => {
40     let routePath = filePath.replace('./pages', '').replace('.{js,jsx}', '');
41     const parts = routePath.split('/');
42
43     // Remove duplicate folder/file name
44     if (
45       parts.length > 1 &&
46       parts.at(-1).toLowerCase() === parts.at(-2).toLowerCase()
47     ) {

```

Figure 5-109: App.jsx (1/3)

Figure 5-110: App.jsx (2/3)

```

94   return (
95     <div className={i18n.language === 'ar' ? 'lang-ar' : 'lang-en'}>
96       <ToastProvider isBuyer={isBuyer}>
97         <NotificationListener />
98         <React.Suspense fallback={null}>
99           <Routes>
100             /* 1. PUBLIC PAGES (Login, signup, 404) */
101             <Route element={<PublicLayout />}>
102               {layoutRoutes.public.map(({ path, Component }) => (
103                 <Route key={path} path={path} element={<Component />} />
104               ))
105             </Route>
106
107             /* 2. SHARED PAGES (search, product, storefront, about) */
108             <Route element={<SharedLayout />}>
109               <Route index element={redirectByRole()} />
110               <Route path="landing" element={<Landing />} />
111               {layoutRoutes.shared.map(({ path, Component }) => (
112                 <Route
113                   key={path}
114                   path={path.replace(/^\//, '')}
115                   element={<Component />}
116                 >
117               ))
118             </Route>
119
120             /* 3. BUYER PRIVATE PAGES */
121             <Route
122               element={[
123                 <ProtectedRoute allowedRoles={['buyer']} redirectTo="/" />
124               ]>
125               <Route path="/buyer/*" element={<BuyerLayout />}>
126                 {layoutRoutes.buyer.map(({ path, Component }) => (
127                   <Route
128                     key={path}
129                     path={path.replace(/\^\/buyer\//, '')}
130                     element={<Component />}
131                   >
132                 ))
133               </Route>
134             </Route>
135
136             /* 4. SUPPLIER PRIVATE PAGES */
137             <Route
138               element={[
139                 <ProtectedRoute allowedRoles={['supplier']} redirectTo="/" />
140               ]>
141               <Route path="/supplier/*" element={<SupplierLayout />}>
142                 {layoutRoutes.supplier.map(({ path, Component }) => (
143                   <Route
144                     key={path}
145                     path={path.replace(/\^\/supplier\//, '')}
146                     element={<Component />}
147                   >
148                 ))
149               </Route>
150             </Route>
151
152             /* 5. ROOT & 404 */
153             <Route path="/" element={redirectByRole()} />
154             <Route path="*" element={<NotFound />} />
155           </Routes>
156         </React.Suspense>
157       <ToastProvider>
158     </div>
159   );

```

Figure 5-111: App.jsx (3/3)

In App.jsx, routes are grouped by layout type (PublicLayout, SharedLayout, BuyerLayout, SupplierLayout), and each section is wrapped by a ProtectedRoute component. This ensures that only authorized roles can access certain parts of the system. For example, a supplier cannot access buyer pages, and users without verified email addresses are redirected to the verification page.

In React, a layout is a high-level structural component that defines the persistent elements of a page (such as headers, sidebars, footers, and banners) while delegating the page-specific content to nested routes through the Outlet component. This mechanism allows for consistent design and behavior across related pages while keeping the code modular and maintainable. The Silah frontend defines several layouts that reflect the roles and navigation patterns of its users. The PublicLayout is used for general, publicly accessible pages such as the login, signup, and not-found screens; it provides a minimal header that includes only the logo and language selector to avoid unnecessary navigation for unauthenticated users.

The GuestLayout serves unregistered visitors browsing the platform, while the BuyerLayout and SupplierLayout define distinct navigational structures for buyers and suppliers respectively. Additionally, a SharedLayout exists for pages accessible to both buyers and guests, such as the search interface. This shared layout dynamically renders either the BuyerHeader or GuestHeader depending on the user's role, ensuring that both audiences receive an appropriate interface without code duplication.

Each layout integrates a global notification context that acts as a single source of truth for notification state across the application. This follows the singleton pattern principle: instead of maintaining multiple, inconsistent notification states in different components, the system maintains a single instance that provides synchronized updates to all subscribed components. For example, when a user marks all notifications as read on one page, the badge counters in the header and sidebar update automatically across the entire application. This design not only improves the user experience but also reduces the risk of synchronization bugs between pages.

The following figures illustrate these layout components and the role-based protection mechanism that governs access to them.

```

src > layouts > PublicLayout.jsx ...
You, 3 weeks ago | author (You)
1 import React from 'react';
2 import SimpleHeader from '@components/SimpleHeader/SimpleHeader';
3 import Footer from '@components/Footer/Footer';
4 import { Outlet } from 'react-router-dom';
5 import DemoBanner from '@components/DemoBanner/DemoBanner';
6
7 export default function PublicLayout() {
8   return (
9     <>
10       <DemoBanner />
11       <SimpleHeader />
12       <main>
13         | <Outlet />
14       </main>
15       <Footer />
16     </>
17   );
18 }

```

Figure 5-112: Public Layout

```

src > layouts > GuestLayout.jsx ...
You, 3 weeks ago | author (You)
1 import React from 'react';
2 import GuestHeader from '@components/GuestHeader/GuestHeader';
3 import Footer from '@components/Footer/Footer';
4 import DemoBanner from '@components/DemoBanner/DemoBanner';
5 import { Outlet } from 'react-router-dom';
6
7 export default function GuestLayout() {
8   return (
9     <>
10       <DemoBanner />
11       <GuestHeader />
12       <main>
13         | <Outlet />
14       </main>
15       <Footer />
16     </>
17   );
18 }

```

Figure 5-113: Guest Layout

```

src > layouts > BuyerLayout.jsx > BuyerLayout
You, 3 weeks ago via PR #57 + add: layouts and fix minor things
6 import { useNotifications } from '../hooks/useNotifications';
7 import { useTranslation } from 'react-i18next';
8
9 export default function BuyerLayout() {
10   const { i18n } = useTranslation();
11   const { notifications, profilePics, markSingleAsRead, markAllAsRead } =
12     useNotifications(i18n.language);
13
14   const unreadCount = notifications.filter((n) => !n.isRead).length;
15
16   const contextValue = {
17     notifications,
18     markAllAsRead,
19     markSingleAsRead,
20     unreadCount,
21   };
22
23   return (
24     <>
25       <DemoBanner />
26       <BuyerHeader
27         unreadCount={unreadCount}
28         notifications={notifications}
29         profilePics={profilePics}
30         markSingleAsRead={markSingleAsRead}
31         markAllAsRead={markAllAsRead}
32       />
33       <main>
34         | <Outlet context={contextValue} />
35       </main>
36       <Footer />
37     </>
38   );
39 }

```

Figure 5-114: Buyer Layout

```

src > layouts > SupplierLayout.jsx > SupplierLayout
You, yesterday | author (You)
1 import React from 'react';
2 import SupplierSidebar from '@components/SupplierSidebar/SupplierSidebar';
3 import Footer from '@components/Footer/Footer';
4 import { Outlet } from 'react-router-dom';
5 import DemoBanner from '@components/DemoBanner/DemoBanner';
6 import { useNotifications } from '../hooks/useNotifications';
7 import { useTranslation } from 'react-i18next';
8
9 export default function SupplierLayout() {
10   const { i18n } = useTranslation();
11   const { notifications, profilePic, markSingleAsRead, markAllAsRead } =
12     useNotifications(i18n.language);
13
14   const unreadCount = notifications.filter((n) => !n.isRead).length;
15
16   const contextValue = {
17     notifications,
18     markAllAsRead,
19     markSingleAsRead,
20     unreadCount,
21   };
22
23   return (
24     <>
25       <DemoBanner />
26       <div className="supplier-layout-container">
27         <SupplierSidebar unreadCount={unreadCount} />
28         <main className="supplier-main">
29           | <Outlet context={contextValue} />
30         </main>
31       </div>
32     </>
33   );
34 }

```

Figure 5-115: Supplier Layout

```

src > layouts > SharedLayout.jsx > SharedLayout
You, yesterday | author (You)
1 import React from 'react';
2 import { Outlet, useLocation } from 'react-router-dom';
3 import { useAuth } from '@context/AuthContext';
4 import { useNotifications } from '../hooks/useNotifications';
5 import { useTranslation } from 'react-i18next';
6 import GuestHeader from '@components/GuestHeader/GuestHeader';
7 import BuyerHeader from '@components/BuyerHeader/BuyerHeader';
8 import DemoBanner from '@components/DemoBanner/DemoBanner';
9 import Footer from '@components/Footer/Footer';
10
11 export default function SharedLayout() {
12   const { role } = useAuth();
13   const { i18n } = useTranslation();
14   const location = useLocation();
15
16   // If I was buyer - i18n.language -->
17   const notificationsData =
18     role === 'buyer' ? useNotifications(i18n.language) : null;
19
20   const unreadCount = notificationsData
21     ? notificationsData.notifications.filter((n) => !n.isRead).length
22     : 0;
23
24   const Header = role === 'buyer' ? BuyerHeader : GuestHeader;
25
26   return (
27     <>
28       <DemoBanner />
29       {role === 'buyer' ? (
30         <BuyerHeader
31           unreadCount={unreadCount}
32           notifications={notificationsData.notifications}
33           profilePics={notificationsData.profilePics}
34           markSingleAsRead={notificationsData.markSingleAsRead}
35           markAllAsRead={notificationsData.markAllAsRead}
36         />
37       ) : (
38         <Header />
39       )}
40       <main>
41         | <Outlet context={[ notificationsData, unreadCount ]} />
42       </main>
43       <Footer />
44     </>
45   );
46 }

```

Figure 5-116: Shared Layout

```

src > ProtectedRoute.js > ProtectedRoute > useEffect() callback
User 2 owns App.js author (You)
1 import React, { useState } from 'react';
2 import { Navigate, Outlet, useLocation, useNavigate } from 'react-router-dom';
3 import { ClipLoader } from 'react-spinners';
4 import { useTranslation } from 'react-i18next';
5 import Swal from 'sweetalert2';
6 import { useAuth } from './context/AuthContext';
7
8 export default function ProtectedRoute({ allowedRoles, redirectTo = '/' }) {
9   const { role, user, location, switching } = useAuth();
10  const location = useLocation();
11  const navigate = useNavigate();
12  const { t } = useTranslation('auto');
13  const [checked, setChecked] = useState(false); // role and email verification checked
14  const [alertShown, setAlertShown] = useState(false);
15
16  // Reset alert and checked when route changes
17  useEffect(() => {
18    setAlertShown(false);
19    setChecked(false);
20  }, [location.pathname]);
21
22  useEffect(() => {
23    // Wait until loading and switching finish
24    if (loading || switching) return;
25
26    if (!checked) {
27      setChecked(true); // mark that we verified role and email for this route
28
29      // Check role first
30      if (allowedRoles.includes(role)) {
31        if (!alertShown) {
32          setAlertShown(true);
33          Swal.fire({
34            icon: 'error',
35            title: 'UnauthorizedTitle',
36            text: 'UnauthorizedText',
37            confirmButtonColor: '#F44336',
38            confirmButtonText: 'OK',
39            allowOutsideClick: false,
40          }).then(() => {
41            navigate(redirectTo, { replace: true });
42          });
43        }
44      }
45    }
46  });

```

Figure 5-117: ProtectedRoute.jsx (1/2)

```

47   // Check email verification for authenticated users
48   if (role === 'guest' && user && !user.isEmailVerified && !alertShown) {
49     setAlertShown(true);
50     Swal.fire({
51       icon: 'warning',
52       title: t('emailNotVerifiedTitle') || 'Email Verification Required',
53       text: t('emailNotVerifiedText') || 'Please verify your email to access most features of this platform.',
54       confirmButtonColor: '#F44336',
55       confirmButtonText: 'OK',
56       allowOutsideClick: false,
57       }).then(() => {
58         navigate('/verify-email', {
59           replace: true,
60           state: { email: user.email },
61         });
62       });
63     }
64   }
65 }
66
67 </>
68 <Role
69   user={user}
70   allowedRoles={allowedRoles}
71   location={location}
72   switching={switching}
73   alertShown={alertShown}
74   navigate={navigate}
75   redirectTo={redirectTo}
76   t={t}
77   checked={checked}
78   >;
79 </>
80 if (loading || switching) {
81   return (
82     <div className='loader-center'>
83       <ClipLoader color='#5B5B5B' size={60} />
84     </div>
85   );
86 }
87 // Only render children if role is allowed and email is verified (or not applicable)
88 return allowedRoles.includes(role) &&
89 (role === 'guest' || user.isEmailVerified) ? (
90   <Outlet />
91 ) : null;
92 }
93 </>

```

Figure 5-118: ProtectedRoute.jsx (2/2)

The App.jsx also wraps the entire routing structure with global providers (discussed later):

- AuthProvider, which manages authentication, token persistence, and role-based access.
- ToastProvider, which handles real-time notifications across the interface.

Language management in Silah was also designed dynamically to complement the routing system. Instead of manually importing every translation file, all language JSON files under locales/ are automatically discovered and loaded at runtime.

In a typical project, translations are imported manually as shown below:

```

import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import en from './locales/en.json';
import ar from './locales/ar.json';

i18n.use(initReactI18next).init({
  resources: { en: { translation: en }, ar: { translation: ar } },
  lng: 'en',
  interpolation: { escapeValue: false },
});

```

Figure 5-119: Example of a simple i18n setup

In contrast, our project uses a dynamic resource loader that automatically imports every JSON file found under the locales/ directory, detects its language code, and registers it in the translation system. This approach eliminates the need for developers to manually update a central registry when adding new translation files; making the process faster, safer, and less prone to conflicts.

```

src > JS i18n.js > ...
  You, 3 weeks ago | 4 authors (You and others)
1 import i18n from 'i18nnext';
2 import { initReactI18nnext } from 'react-i18next';
3 import LanguageDetector from 'i18nnext-browser-languagedetector';
4
5 const resources = {};
6
7 const modules = import.meta.glob('./locales/*/*.json', {
8   eager: true,
9   query: 'raw',
10  import: 'default',
11 });
12
13 for (const path in modules) {
14   // path example: ./locales/en/landing.json
15   const [, lang, ns] = path.match(/^.\/locales\/(.+)\.(.+)\.json/);
16   if (!resources[lang]) resources[lang] = {};
17   resources[lang][ns] = JSON.parse(modules[path]);
18 }
19
20 i18n
21 .use(LanguageDetector)
22 .use(initReactI18nnext)
23 .init({
24   resources,
25   fallbackLng: 'en',
26   defaultNS: 'common',
27   interpolation: { escapeValue: false },
28   detection: {
29     order: ['queryString', 'localStorage', 'navigator'],
30     caches: ['localStorage']
31   },
32 });
33
34 export default i18n;

```

Figure 5-120: i18n.js file

Language keys are then accessed through the t() function provided by the useTranslation() hook. For instance, in the chat module, page titles and interface strings are defined in chats.json and automatically translated at runtime:

```

src > locales > ar > {} chats.json > draft
You, yesterday | 1 author (You)
1
2   "pageTitle": "المحادثات",
3   "allMessages": "جميع الرسائل",
4   "unreadMessages": "غير مفروضة",
5   "readMessages": "مفروضة",
6   "allDays": "جميع الأيام",
7   "today": "اليوم",
8   "thisWeek": "هذا الأسبوع",
9   "thisMonth": "هذا الشهر",
10  "type": "الرسالة",
11  "date": "التاريخ",
12  "searchPlaceholder": "ابحث في المحادثات أو المستخدمين",
13  "noSearchResults": "لم يتم العثور على نتائج",
14  "noChats": "لا يوجد محادثات بعد",
15  "loading": "... جاري التحميل",
16  "errorLoadingChats": "فشل تحميل المحادثات. حاول مرة أخرى",
17  "startNewChat": "ابدأ محادثة جديدة",
18  "ImageMessage": "[صورة]",
19  "noMessagesYet": "لا يوجد رسائل بعد. ابدأ المحادثة!",
20  "typeMessage": "رسالة...",
21  "chatWith": "محادثة مع {{otherUser}}",
22  "draft": "مسودة"
23

```

Figure 5-121: ar/chats.json file

```

src > locales > en > {} chats.json > ...
You, yesterday | 1 author (You)
1
2   "pageTitle": "Chats",
3   "allMessages": "All Messages",
4   "unreadMessages": "Unread",
5   "readMessages": "Read",
6   "allDays": "All Days",
7   "today": "Today",
8   "thisWeek": "This Week",
9   "thisMonth": "This Month",
10  "type": "Type",
11  "date": "Date",
12  "searchPlaceholder": "Search chats or users...",
13  "noSearchResults": "No results found",
14  "noChats": "No chats yet",
15  "loading": "Loading ...",
16  "errorLoadingChats": "Failed to load chats. Please try again.",
17  "startNewChat": "Start new chat",
18  "ImageMessage": "[Image]",
19  "noMessagesYet": "No messages yet. Start the conversation!",
20  "typeMessage": "Type a message ...",
21  "chatWith": "Chat With {{otherUser}}",
22  "draft": "Draft"
23

```

Figure 5-122: en/chats.json file

```

import React, { useEffect } from 'react';
import { useTranslation } from 'react-i18next';

function Landing() {
  const { t, i18n } = useTranslation();

  useEffect(() => {
    document.title = t('pageTitle');
  }, [t, i18n.language]);

  return <div>Landing Page</div>;
}

export default Landing;

```

Figure 5-123: Setting page title dynamically

This dynamic setup proved to be highly beneficial for teamwork, as it minimized human error, simplified translation management, and kept the codebase consistent and conflict-free.

In summary, both the dynamic routing and dynamic internationalization strategies reinforced one of Silah's core architectural principles; automation for maintainability. By allowing the system to automatically discover new pages and language files, we reduced manual coordination overhead, enabling faster, more parallelized development while maintaining a clean and scalable structure.

The authentication context (AuthContext) is responsible for managing the user's authentication state, role, and related access logic across the entire frontend application. It provides global state values such as the current user, role (Guest, Buyer, or Supplier), and supplier status (e.g., INACTIVE). This context ensures that all pages in the system are aware of the user's status without requiring repetitive API calls or prop drilling.

The AuthProvider fetches the authenticated user's information from the backend (/api/users/me) upon application load and automatically determines the appropriate role-based access. If a supplier account is marked as INACTIVE, a localized warning is displayed using SweetAlert2, notifying them that their storefront and products will remain hidden until payment is completed. The context also handles logout, token expiry, and role switching, ensuring smooth session management.

Protected routes are defined using the ProtectedRoute component (explained previously), which consumes the authentication context to restrict access based on the user's role. Unauthorized users are redirected to the appropriate page; for example, guests attempting to access buyer routes are redirected to the /landing page, and users with unverified emails are prompted to verify their accounts before continuing and redirected to the /verify-email page.

This approach allows for a clear separation of concerns; authentication logic is centralized in the AuthContext, while access control is handled by ProtectedRoute. Together, they form the backbone of Silah's role-based navigation system.

```

src > context > AuthContext.jsx > @AuthProvider
  ...
  1 import React, { createContext, useContext, useEffect, useState } from 'react';
  2 import { useTranslation } from 'react-i18next';
  3 import axios from 'axios';
  4 import Swal from 'sweetalert2';
  5 ...
  6 const AuthContext = createContext();
  7 ...
  8 export function AuthProvider({ children }) {
  9   ...
 10   const [user, setUser] = useState(null);
 11   const [role, setRole] = useState('guest');
 12   const [loading, setLoading] = useState(true);
 13   const [switching, setSwitching] = useState(false);
 14   const [supplierStatus, setSupplierStatus] = useState(null);
 15 ...
 16   const INACTIVE_NOTICE_KEY = 'inactiveSupplierNoticeClosed';
 17 ...
 18   const showInactiveSupplierNotice = () => {
 19     const isClosed = sessionStorage.getItem(INACTIVE_NOTICE_KEY) === '1';
 20     if (isClosed) return;
 21 ...
 22     const isArabic = i18n.language === 'ar';
 23 ...
 24     Swal.fire({
 25       icon: 'warning',
 26       title: isArabic
 27         ? `لقد تجاوزت حدود الماء الأساسية !` // You exceeded basic plan limits!
 28         : `You exceeded basic plan limits!` // You exceeded basic plan limits!
 29       , text: isArabic
 30         ? `سيتم إخفاء متجرك ومنتجاته من المشترين حتى يتم دفع المبلغ المتأخر.` // Your storefront and products shall be temporarily hidden from buyers until payment is made.
 31         : `Your storefront and products shall be temporarily hidden from buyers until payment is made.` // Your storefront and products shall be temporarily hidden from buyers until payment is made.
 32       , confirmButtonText: isArabic ? 'نعم' : 'Got it',
 33       confirmButtonColor: '#4472C4',
 34       allowOutsideClick: false,
 35       allowEscapeKey: false,
 36       customClass: {
 37         ...,
 38         popper: 'swal rtl',
 39       },
 40     }).then(() => {
 41       sessionStorage.setItem(INACTIVE_NOTICE_KEY, '1');
 42     });
 43   }

```

Figure 5-125: AuthContext.jsx (1/3)

```

44 const handleLogout = async () => {
45   try {
46     await axios.post(
47       `${import.meta.env.VITE_BACKEND_URL}/api/auth/logout`,
48       {},
49       { withCredentials: true },
50     );
51   } catch (err) {
52     console.warn('Logout failed:', err.response?.data || err.message);
53   } finally {
54     setUser(null);
55     setRole('guest');
56     setSupplierStatus(null);
57     sessionStorage.removeItem(INACTIVE_NOTICE_KEY);
58   }
59 }
60 ...
61 const fetchUser = async () => {
62   try {
63     const res = await axios.get(
64       `${import.meta.env.VITE_BACKEND_URL}/api/users/me`,
65       { withCredentials: true },
66     );
67     const userData = res.data;
68     setUser(userData);
69     const userRole = userData.role.toLowerCase() || 'guest';
70     setRole(userRole);
71 ...
72     // --- (1) عميل - (2) مورد - (3) مورد مزود - (4) مورد مزود مزود ---
73     if (userRole === 'supplier') {
74       try {
75         const supplierRes = await axios.get(
76           `${import.meta.env.VITE_BACKEND_URL}/api/suppliers/me`,
77           { withCredentials: true },
78         );
79         const status = supplierRes.data.supplierStatus;
80         setSupplierStatus(status);
81       }
82       // --- (1) مورد - (2) مورد مزود - (3) مورد مزود مزود ---
83       if (status === 'INACTIVE') {
84         showInactiveSupplierNotice();
85       }
86     } catch (supplierErr) {
87       console.error('Failed to fetch supplier data:', supplierErr);
88       setSupplierStatus(null);
89     }
90   } else {
91     setSupplierStatus(null);
92   }
93 } catch (err) {
94   if (err.response?.status === 401 || err.response?.status === 403) {
95     const message = err.response?.data?.error?.message;
96 ...
97     if (message === 'Invalid or expired token') {
98       await Swal.fire({
99         icon: 'warning',
100        title: t('title'),
101        text: t('text'),
102        confirmButtonColor: '#4472C4',
103        confirmButtonText: 'OK',
104      });
105    }
106  }
107 ...
108  setUser(null);
109  setRole('guest');
110  setSupplierStatus(null);
111 } else {
112   console.error('Fetch user failed:', err);
113   setUser(null);
114   setRole('guest');
115   setSupplierStatus(null);
116 } finally {
117   setLoading(false);
118 }
119 };
120 ...
121 useEffect(() => {
122   fetchUser();
123 }, []);
124 ...
125 const switchRole = async () => {
126   if (switching) return role;
127   setSwitching(true);
128   try {
129     const res = await axios.patch(
130       `${import.meta.env.VITE_BACKEND_URL}/api/auth/switch-role`,
131       {},
132       { withCredentials: true },
133     );
134     await fetchUser();
135     return res.data?.newRole?.toLowerCase() || role;
136   } catch (err) {
137     const msg =
138       err.response?.data?.error?.message || err.response?.data?.message || err.message;
139     Swal.fire({
140       icon: 'error',
141       title: t('switchRoleErrorTitle') || 'Switch role failed',
142       text: msg,
143       confirmButtonColor: '#4472C4',
144     });
145   }
146   return role;
147 } finally {
148   setSwitching(false);
149 }
150 ...
151 };
152 ...
153 return (
154   <AuthContext.Provider
155     value={{
156       user,
157       role,
158       loading,
159       switching,
160       supplierStatus,
161       handleLogout,
162       fetchUser,
163       switchRole,
164     }}
165   >
166     {children}
167   </AuthContext.Provider>
168 );
169 }
170 ...
171 export const useAuth = () => useContext(AuthContext);

```

Figure 5-126: AuthContext.jsx (2/3)

Figure 5-127: AuthContext.jsx (3/3)

The frontend communicates with the NestJS backend using the Axios HTTP client. All requests share a base URL defined in the environment file (.env), allowing flexible switching between production and local development environments.

The base URL is defined as: VITE_BACKEND_URL=<https://api.silah.backend>

During local testing, developers can simply switch this to:

VITE_BACKEND_URL=<http://localhost:3000>

This setup was crucial for team collaboration, as the deployed backend made integration testing faster and avoided the need for every frontend developer to install the backend codebase, its dependencies, and the database locally. Authentication and session persistence rely on HTTP-only cookies managed by the backend, while Axios requests include { withCredentials: true } to allow cross-domain cookie transmission securely.

```
const res = await axios.post(
  `${import.meta.env.VITE_BACKEND_URL}/api/auth/login`,
  payload,
  { withCredentials: true },
);
```

Figure 5-128: Example of Axios integration in the login page (Login.jsx)

A practical example of how authentication and Axios communication work together can be seen in the Login.jsx component. The page allows users to sign in using either their email or commercial registration number (CRN). Upon successful authentication, the system retrieves the user's profile through the AuthContext and navigates them to their appropriate dashboard based on their role.

Validation is performed both locally and via backend responses to ensure secure login behavior. The page also demonstrates how localization is applied dynamically using react-i18next, setting the document title and interface language according to the user's preference.

```

src > pages > Login > Login.jsx > Login.css
You, 2 weeks ago | 2 authors (You and one other)
1 import React, { useState, useEffect } from 'react';
2 import './Login.css';
3 import { useNavigate } from 'react-router-dom';
4 import { useTranslation } from 'react-i18next';
5 import axios from 'axios';
6 import { useAuth } from '@/context/AuthContext';
7
8 function Login() {
9   const { t, i18n } = useTranslation('login');
10  const navigate = useNavigate();
11  const [identifier, setIdentifier] = useState('');
12  const [password, setPassword] = useState('');
13  const [error, setError] = useState('');
14  const [loading, setLoading] = useState(false);
15  const { fetchUser } = useAuth();
16
17  useEffect(() => {
18    document.title = t('pageTitle.login', { ns: 'common' });
19  }, [t, i18n.language]);
20
21  const handleLogin = async () => {
22    setError('');
23    if (!identifier || !password) {
24      setError(t('errors.emptyFields'));
25      return;
26    }
27
28    const isEmail = /^[^\s@]+@[^\s@]+\.\w+\s/.test(identifier);
29    const isCRN = /^\d{10}\$/.test(identifier);
30
31    if (!isEmail && !isCRN) {
32      setError(t('errors.invalidEmailOrCRN'));
33      return;
34    }
35
36    const payload = {
37      password,
38      ...(isEmail ? { email: identifier } : { crn: identifier }),
39    };
40
41    try {
42      setLoading(true);
43      const res = await axios.post(
44        `${import.meta.env.VITE_BACKEND_URL}/api/auth/login`,
45        payload,
46        { withCredentials: true },
47      );
48
49      try {
50        setLoading(true);
51        const res = await axios.post(
52          `${import.meta.env.VITE_BACKEND_URL}/api/auth/login`,
53          payload,
54          { withCredentials: true },
55        );
56
57        // Example response: { message: "Login successful", role: "OWNER" }
58        const role = res.data.role.toLowerCase() || (await refreshUser()).role;
59
60        // Update the user context immediately
61        await fetchUser();
62
63        navigate('/');
64      } catch (err) {
65        console.log(Login.error, err);
66
67        // Handle real + swagger formats, and handle arrays safely
68        let backendMessage;
69        err.response?.data?.error?.message || err.response?.data?.message;
70
71        // If the backend sent an array (e.g. ["Password must be at least 8 characters"])
72        if (Array.isArray(backendMessage)) {
73          backendMessage = backendMessage.join(', ');
74        }
75
76        if (backendMessage === 'User not found') {
77          setError(t('errors.userNotFound'));
78        } else if (backendMessage === 'Invalid credentials') {
79          setError(t('errors.invalidCredentials'));
80        } else if (
81          typeof backendMessage === 'string' ||
82          backendMessage.toLowerCase().includes('token')
83        ) {
84          setError(t('errors.sessionExpired'));
85        } else if (backendMessage) {
86          setError(backendMessage); // show backend message directly
87        } else {
88          setError(t('errors.network'));
89        }
90      } finally {
91        setLoading(false);
92      }
93    } catch (err) {
94      console.error(Login.error, err);
95    }
96  }
97}

```

Figure 5-129: Login.jsx (1/3)

```

87  return (
88    <div className="login-page">
89      <div className="login-container">
90          <h2>{t('title')}</h2>
91
92          <input
93              type="text"
94              placeholder={t('emailOrCRN')}
95              value={identifier}
96              onChange={(e) => setIdentifier(e.target.value)}
97              onKeyDown={(e) => {
98                if (e.key === 'Enter') handleLogin();
99              }}
100             />
101
102          <input
103              type="password"
104              placeholder={t('password')}
105              value={password}
106              onChange={(e) => setPassword(e.target.value)}
107              onKeyDown={(e) => {
108                if (e.key === 'Enter') handleLogin();
109              }}
110             />
111
112          {error && <p className="error-message">{error}</p>}
113
114          <button className="enter-btn" onClick={handleLogin} disabled={loading}>
115            {loading ? t('loading') : t('submit')}
116          </button>
117
118          <div className="login-options">
119              <span
120                  className="text-link reset-link"
121                  onClick={() => navigate('/request-password-reset')}
122              >
123                  {t('resetPassword')}
124              </span>
125
126              <p className="create-account">
127                  {t('noAccount')}{' '}
128                  <span className="text-link" onClick={() => navigate('/signup')}>
129                      {t('createOne')}
130                  </span>
131              </p>
132          </div>
133      </div>
134  </div>;
135  ];
136  ]
137
138  export default Login;

```

Figure 5-131: Login.jsx (3/3)

Each page in the project has its own CSS file, such as pages/Login/Login.css, keeping the styling modular and page-specific. This practice improves maintainability and

aligns with our modular design approach. A visual representation of the folder structure is:

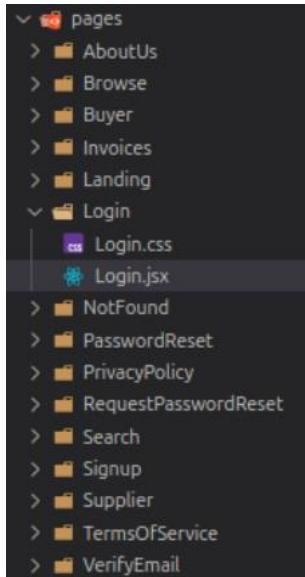


Figure 5-132: Page structure showing Login.jsx and Login.css under pages/Login/

5.2.3.2 Real-Time and Streaming Features

This section describes how the frontend integrates with the backend's real-time services to enable continuous, live interaction. Two streaming technologies are used: Server-Sent Events (SSE) for notifications and WebSocket (Socket.IO) for chat.

Real-time notifications in Silah are delivered through a combination of Server-Sent Events (SSE), a custom React hook (`useNotifications`), and a global notification context (`ToastProvider`) responsible for rendering transient on-screen pop-ups (toasts). Together, these components ensure that users receive live updates about important activities (such as new chat messages, order updates, or group purchase changes) without the need for manual page refreshes or polling.

Unlike WebSockets, which support bidirectional communication, SSE provides a one-way channel from the server to the client. This was ideal for Silah's notification model, where the backend pushes updates to users but the frontend rarely needs to respond immediately. The frontend maintains a single persistent connection to the backend's `/api/notifications/stream` endpoint using the browser's built-in `EventSource` API. This connection automatically reestablishes itself on network interruptions, offering resilience without additional code complexity.

To integrate SSE seamlessly with the React ecosystem, the frontend defines a custom hook named `useNotifications`, which handles three main responsibilities:

1. Fetching initial notifications upon component mount via Axios (`/api/notifications/me`).
2. Listening for live notifications from the SSE stream and appending them to state while avoiding duplicates.
3. Managing auxiliary data, such as fetching sender profile pictures and updating read/unread status through the backend.

```

use > hooks > JS useNotifications.js ...
  You yesterday | 1 author (You)
  1 import { useEffect, useState, useRef } from 'react';           You last week
  2 import axios from 'axios';
  3
  4 export const useNotifications = (lang, onNewNotification) => {
  5   const [notifications, setNotifications] = useState([]);
  6   const [profilePics, setProfilePics] = useState({});
  7   const eventSourceRef = useRef(null);
  8   const seenIdsRef = useRef(new Set());
  9
 10  useEffect(() => {
 11    const fetchInitial = async () => {
 12      try {
 13        const [data] = await axios.get(
 14          `${import.meta.env.VITE_BACKEND_URL}/api/notifications/me`,
 15          { params: { lang }, withCredentials: true },
 16        );
 17        setNotifications(data);
 18        data.forEach((n) => seenIdsRef.current.add(n.notificationId));
 19      } catch (err) {
 20        console.error('Failed to load notifications', err);
 21      }
 22    };
 23
 24    fetchInitial();
 25
 26    if (eventSourceRef.current) {
 27      eventSourceRef.current.close();
 28    }
 29  });

```

Figure 5-133: useNotification.js hook (1/3)

```

30  const es = new EventSource(
31    `${import.meta.env.VITE_BACKEND_URL}/api/notifications/stream`,
32    { withCredentials: true },
33  );
34  eventSourceRef.current = es;
35
36  es.onmessage = async (e) => {
37    try {
38      const parsed = JSON.parse(e.data);
39      const notif = parsed.data ? JSON.parse(parsed.data) : parsed;
40
41      if (seenIdsRef.current.has(notif.notificationId)) return;
42      seenIdsRef.current.add(notif.notificationId);
43
44      setNotifications([prev] => [notif, ...prev]);
45      onNewNotification?.(notif);
46
47      const senderId = notif.sender?.userId;
48      if (senderId && !profilePics[senderId]) {
49        try {
50          const [data] = await axios.get(
51            `${import.meta.env.VITE_BACKEND_URL}
52              /api/users/${senderId}/profile-picture`,
53            { withCredentials: true },
54          );
55          if (data.pfpUrl) {
56            setProfilePics(prev => ({ ...prev, [senderId]: data.pfpUrl }));
57          }
58        } catch (err) {
59          console.error('Failed to fetch pfp', err);
60        }
61      }
62    } catch (err) {
63      console.error('SSE parse error', err);
64    }
65  };
66
67  es.onerror = () => {
68    console.error('SSE error ⚡ browser will retry');
69  };
70
71  return () => {
72    es.close();
73    eventSourceRef.current = null;
74    seenIdsRef.current.clear();
75  };
76}, [lang, onNewNotification]);
77

```

Figure 5-134: useNotification.js hook (2/3)

```

79  const markAllAsRead = async () => {
80    const unreadIds = notifications
81      .filter((n) => !n.isRead)
82      .map((n) => n.notificationId);
83
84    if (unreadIds.length === 0) return;
85
86    try {
87      await axios.patch(
88        `${import.meta.env.VITE_BACKEND_URL}/api/notifications/read-many`,
89        { notificationIds: unreadIds },
90        { withCredentials: true },
91      );
92
93      setNotifications([prev] =>
94        prev.map((n) =>
95          !unreadIds.includes(n.notificationId) ? { ...n, isRead: true } : n,
96        ),
97      );
98    } catch (err) {
99      console.error('Failed to mark all as read', err);
100    }
101  };
102
103  const markSingleAsRead = async (notificationId) => {
104    try {
105      const [data] = await axios.patch(
106        `${import.meta.env.VITE_BACKEND_URL}
107          /api/notifications/${notificationId}/read`,
108        {},
109        { withCredentials: true },
110      );
111      setNotifications([prev] =>
112        prev.map((n) => (n.notificationId === notificationId ? data : n)),
113      );
114    } catch (err) {
115      console.error('Failed to mark single as read', err);
116    }
117  };
118
119  return { notifications, profilePics, markAllAsRead, markSingleAsRead };
120};

```

Figure 5-135: useNotification.js hook (3/3)

The hook is used by a lightweight listener component, `NotificationListener.jsx`, which bridges the SSE stream with the Toast Context. It registers a callback (`handleNewNotification`) that triggers a new toast whenever a notification event arrives. This decoupling allows the toast logic to remain isolated and reusable across different layouts.

```

src > components > NotificationListener.jsx > ...
  You, yesterday | Author: You
  1 import { useEffect, useCallback } from 'react';
  2 import { useTranslation } from 'react-i18next';
  3 import { useNotifications } from '@hooks/useNotifications';
  4 import { useToast } from '@context/NotificationPopupToast/NotificationContext';
  5
  6 export default function NotificationListener() {
  7   const { i18n } = useTranslation();
  8   const { addToast } = useToast();
  9
 10  // Memoize the callback so it's stable across renders
 11  const handleNewNotification = useCallback(
 12    (newNotif) => {
 13      addToast(newNotif);
 14    },
 15    [addToast]
 16  ); // Only recreate if addToast changes (which it doesn't)
 17
 18  // Pass stable callback + language
 19  useNotifications(i18n.language, handleNewNotification);
 20
 21  return null;
 22 }

```

Figure 5-136: NotificationListener.jsx

At the top of the component hierarchy, every layout in Silah wraps its content with a singleton instance of `ToastProvider`, ensuring that all pages share the same notification state and container. This design avoids multiple concurrent toast managers and preserves consistency in both Buyer and Supplier interfaces. The context also supports RTL/LTR adaptation: when the application is in Arabic mode, toast notifications slide in from the right side of the screen, whereas in English mode, they appear on the left, maintaining a natural flow for each language direction.

```

src > context > NotificationContext.jsx > NotificationContext.jsx > ...
  You, yesterday | Author: You
  1 import React, { createContext, useContext, useState } from 'react';
  2 import Timeline from './Timeline';
  3 import './NotificationContext.css';
  4
  5 const NotificationContext = createContext();
  6
  7 export const useToast = () => useContext(NotificationContext);
  8
  9 export const ToastProvider = ({ children, isBuyer = true }) => {
 10   const [toasts, setToasts] = useState([]);
 11
 12   // Play sound
 13   const playSound = (x) => {
 14     const audio = new Audio();
 15     const url = `data:audio/wav;base64,${btoa('d4QwDgADGwYTFwTBIBZAMRAABuADYwAAkABAAZcFyOAAA=')}${x}`;
 16     audio.volume = 0.4;
 17     audio.play();
 18     audio.onerror = () => {};
 19   };
 20
 21   const addToast = (notification) => {
 22     const id = notification.notificationId + Date.now();
 23     const timeout = isBuyer ? 3000 : 5000;
 24     const toast = { ...notification, id, timeout };
 25
 26     // play sound
 27     setToasts([toast, ...toasts]);
 28
 29     const removeToast = () => {
 30       setToasts(toasts.filter((t) => t.id !== id));
 31       const toast = prevFind((t) => t.id === id);
 32       if (toast) clearTimeout(toast.timeout);
 33       if (prevFind((t) => t.id === id)) {
 34         playSound(id);
 35       }
 36     };
 37
 38     return {
 39       <ReactContext.Provider value={{ addToast }}>
 40         {children}
 41       </ReactContext.Provider>
 42     };
 43   };
 44
 45   const cleanName = toastContainer => isBuyer ? 'buyer' : 'supplier';
 46   const find = (name) => toasts.find((t) => t.notification.type === name);
 47   const remove = (id) => toasts.filter((t) => t.id !== id);
 48
 49   return {
 50     addToast,
 51     remove,
 52     find,
 53     cleanName,
 54     removeToast,
 55     Timeline,
 56   };
 57 }

```

Figure 5-137: NotificationContext.jsx

The Toast Context (`NotificationContext.jsx`) encapsulates logic for queueing, displaying, and dismissing notifications. Each toast is automatically removed after three seconds, with the option for the user to dismiss it manually. Toasts are visually represented using the `ToastItem` component, which dynamically selects an icon according to the notification type (such as messages, orders, or offers) using the `react-icons` library.

```

src > context > NotificationPopupToast > ● NotificationToastItem.jsx > ...
  You need to add a loader (You)
  1 import React from 'react';
  2 import FaTimes, {
  3   FaEnvelope,
  4   FaFileInvoice,
  5   FaTag,
  6   FaTruck,
  7   FaUsers,
  8   FaStar,
  9 } from 'react-icons/fa';
 10
 11 const TYPE_ICONS = {
 12   CHAT: FaEnvelope,
 13   INVOICE: FaFileInvoice,
 14   OFFER: FaTag,
 15   ORDER: FaTruck,
 16   GROUP_PURCHASE: FaUsers,
 17   REVIEW: FaStar,
 18   BID: FaTag,
 19 };
 20
 21 const DefaultIcon = FaEnvelope;
 22
 23 const ToastItem = ({ notification, onClose, isBuyer }) => {
 24   const IconComponent =
 25     TYPE_ICONS[notification.relatedEntityType] || DefaultIcon;
 26
 27   return (
 28     <div
 29       className={ `toast-item ${isBuyer ? 'buyer' : 'supplier'} ${
 30         notification.isRead ? 'unread' : ''
 31       }` }
 32     >
 33       <div className="toast-icon-circle">
 34         <IconComponent />
 35       </div>
 36       <div className="toast-content">
 37         <div className="toast-title">{notification.title}</div>
 38         <div className="toast-message">{notification.content}</div>
 39       </div>
 40       <button className="toast-close" onClick={onClose}>
 41         <FaTimes />
 42       </button>
 43     </div>
 44   );
 45 };
 46
 47 export default ToastItem;

```

Figure 5-138: NotificationItemContext.jsx

Finally, each notification type shares a consistent visual style defined in `NotificationToast.css`. This CSS file handles positioning, animations, and theme adjustments for Buyer and Supplier roles. The result is a unified, real-time notification system that operates transparently across the entire application.

The dedicated Notifications page (`Notifications.jsx`) displays all notifications, allowing the user to view, filter, and manage them in one place. This page does not create a separate data source or open a new SSE connection. Instead, it uses the same singleton notification context provided by the layout through React Router’s `useOutletContext()` hook. This ensures that the notification state remains perfectly synchronized between the background listener, the toast pop-ups, and the main notification list.

When a user marks a notification as read (whether directly from the toast pop-up or from the Notifications page) the update is immediately reflected everywhere in the interface. For example, the unread count displayed on the header’s notification badge decreases in real time, demonstrating the shared global state between layouts and components. Similarly, when the user clicks “Mark All as Read”, the change propagates through the same context, updating both the backend and all other open UI instances without requiring a reload.

This seamless synchronization highlights the strength of using a centralized context architecture for real-time data sharing. By reusing the same hook and provider across layouts, Silah maintains a single live connection per user session, reduces redundant data fetching, and guarantees UI consistency between Buyer and Supplier dashboards.

```

on > page / buyer / Notifications.js > Notifications.js > buyerNotifications
  ...
  <!-- Notifications Layout -->
  1 import React, { useState, useEffect } from 'react';
  2 import { useTranslation } from 'react-i18next';
  3 import { useNavigate, useOutletContext } from 'react-router-dom';
  4 import { FaEnvelope, FaFileInvoice, FaTag, FaTruck, FaUsers, FaCheck } from 'react-icons/fa';
  5 import './Notifications.css';
  6
  7 const TYPE_ICONS = {
  8   CHAT: FaEnvelope,
  9   INVOICE: FaFileInvoice,
 10   OFFER: FaTag,
 11   ORDER: FaTruck,
 12   GROUP_PURCHASE: FaUsers,
 13 };
 14
 15 export default function BuyerNotifications() {
 16   const [t, setT] = useTranslation('notifications');
 17   const [selectedType, setSelectedType] = useState('');
 18   const [selectedDate, setSelectedDate] = useState('');
 19   const [loading, setLoading] = useState(true);
 20   const [typeMap, setTypeMap] = useState({ newMessages: 'CHAT', newInvoices: 'INVOICE', offers: 'OFFER', orderStatus: 'ORDER', groupPurchase: 'GROUP_PURCHASE' });
 21   const [notifications, markAllAsRead, markSingleAsRead] = useOutletContext();
 22
 23   useEffect(() => {
 24     if (notifications.length > 0) {
 25       setLoading(false);
 26     }
 27   }, [notifications]);
 28
 29   // --- Close dropdowns ---
 30   useEffect(() => {
 31     const handleClick = (e) => {
 32       if (e.target.closest('.page-dropdown')) {
 33         setTypeOpen(false);
 34         setSelectedDate('');
 35       }
 36     };
 37     document.addEventListener('click', handleClick);
 38     return () => document.removeEventListener('click', handleClick);
 39   }, []);
 40
 41 }
 42
 43 // --- Click logic ---
 44 const handleOnClick = (n) => {
 45   if (n.relatedEntityId) {
 46     if (n.type === 'newMessages') {
 47       setTypeOpen(true);
 48       setSelectedDate(n.created);
 49     }
 50   }
 51   navigate(n.route);
 52 }
 53
 54 
```

Figure 5-139: Notifications.jsx (1/5)

```

55 // --- Handle Click ---
56 const handleNotificationClick = (n) => {
57   if (n.isRead) markSingleAsRead(n.notificationId);
58   switch (n.relatedEntityType) {
59     case 'CHAT':
60       navigate('/buyer/chats');
61       break;
62     case 'INVOICE':
63       navigate('/buyer/invoices/${n.relatedEntityId}');
64       break;
65     case 'OFFER':
66       break;
67     case 'ORDER':
68       navigate('/buyer/orders/${n.relatedEntityId}');
69       break;
70     case 'GROUP_PURCHASE':
71       navigate('/buyer/invoices');
72       break;
73     default:
74       break;
75   }
76 };
77
78 // --- Filter Logic ---
79 const filtered = notifications.filter((n) => {
80   if (selectedType === t('allNotifications')) {
81     const typeMap = {
82       [t('newMessages')]: 'CHAT',
83       [t('newInvoices')]: 'INVOICE',
84       [t('offers')]: 'OFFER',
85       [t('orderStatus')]: 'ORDER',
86       [t('groupPurchase')]: 'GROUP_PURCHASE',
87     };
88     if (typeMap[selectedType] !== n.relatedEntityType) return false;
89   }
90   if (selectedDate === t('allDays')) {
91     const today = new Date();
92     const created = new Date(n.createdAt);
93     if (selectedDate === t('today') && !isSameDay(created, today)) {
94       return false;
95     }
96     if (selectedDate === t('thisWeek') && !isThisWeek(created)) return false;
97     if (
98       selectedDate === t('thisMonth') &&
99       created.getMonth() !== today.getMonth()
100     ) {
101       return false;
102     }
103   }
104 
```

Figure 5-140: Notifications.jsx (2/5)

```

105     return (
106       <div className="buyer-notif-page" data-dlr={dlr}>
107         <div className="page-notif-header">
108           <div className="page-notif-titler-center">
109             <div style={{ filter: `blur(1px)` }}>
110               <div className="page-notif-filter-inlne">
111                 <span className="page-filter-label">({`type`})</span>
112                 <div className="page-dropdown">
113                   <button onClick={() => setTypeOpen(!prev)}>({`prev`})</button>
114                   <div style={{ position: 'absolute' }}>
115                     {TypeOpen && (
116                       <div style={{ padding: 10px }}>
117                         <div style={{ display: 'flex', gap: 10px }}>
118                           {[
119                             t('allNotifications'),
120                             t('newMessages'),
121                             t('newInvoices'),
122                             t('newOffers'),
123                             t('orderStatus'),
124                             t('groupPurchase'),
125                           ].map((opt) => {
126                             <div style={{ position: 'relative' }}>
127                               <div key={opt}>
128                                 <div style={{ position: 'absolute' }}>
129                                   <span style={{ color: '#000' }}>{`selected`}</span>
130                                 <div style={{ border: 1px solid #ccc, padding: 5px }}>
131                                   {opt}
132                                 </div>
133                               </div>
134                             </div>
135                           ])
136                         </div>
137                         <div style={{ margin: 10px 0 }}>
138                           <button onClick={() => setFilterOpen(true)}>({`date`})</button>
139                           <div style={{ border: 1px solid #ccc, padding: 5px }}>
140                             {selectedDate} && (
141                               <div style={{ display: 'flex', gap: 10px }}>
142                                 {[
143                                   t('allDays'),
144                                   t('today'),
145                                   t('thisWeek'),
146                                   t('thisMonth'),
147                                 ].map((n) => {
148                                   <div style={{ position: 'relative' }}>
149                                     <div key={n}>
150                                       <div style={{ position: 'absolute' }}>
151                                         <span style={{ color: '#000' }}>{`selected`}</span>
152                                       <div style={{ border: 1px solid #ccc, padding: 5px }}>
153                                         {n}
154                                       </div>
155                                     </div>
156                                   </div>
157                                 ])
158                               </div>
159                             </div>
160                           )
161                         </div>
162                         <div style={{ margin: 10px 0 }}>
163                           <button onClick={() => setFilterOpen(true)}>({`date`})</button>
164                           <div style={{ border: 1px solid #ccc, padding: 5px }}>
165                             {selectedDate} && (
166                               <div style={{ display: 'flex', gap: 10px }}>
167                                 {[
168                                   t('allDays'),
169                                   t('today'),
170                                   t('thisWeek'),
171                                   t('thisMonth'),
172                                 ].map((n) => {
173                                   <div style={{ position: 'relative' }}>
174                                     <div key={n}>
175                                       <div style={{ position: 'absolute' }}>
176                                         <span style={{ color: '#000' }}>{`selected`}</span>
177                                       <div style={{ border: 1px solid #ccc, padding: 5px }}>
178                                         {n}
179                                       </div>
180                                     </div>
181                                   </div>
182                                 ])
183                               </div>
184                             </div>
185                           )
186                         </div>
187                         <div style={{ margin: 10px 0 }}>
188                           <button onClick={() => markAllAsRead()}>({`markAllRead`})</button>
189                         </div>
190                       </div>
191                     )
192                   </div>
193                 </div>
194               </div>
195             </div>
196           </div>
197         </div>
198       </div>
199     )
200   );
201   </div>
202   <div style={{ margin: 10px 0 }}>
203     <button onClick={() => handleNotificationClick(n)}>({`n.notificationId`})</button>
204   </div>
205   <div style={{ margin: 10px 0 }}>
206     <div style={{ display: 'flex', gap: 10px }}>
207       <div style={{ position: 'relative' }}>
208         <div style={{ position: 'absolute' }}>
209           <span style={{ color: '#000' }}>{`n.isRead`}</span>
210         <div style={{ border: 1px solid #ccc, padding: 5px }}>
211           {n.title}
212         </div>
213       </div>
214       <div style={{ position: 'relative' }}>
215         <div style={{ position: 'absolute' }}>
216           <span style={{ color: '#000' }}>{`n.message`}</span>
217         <div style={{ border: 1px solid #ccc, padding: 5px }}>
218           {n.content}
219         </div>
220       </div>
221     </div>
222   </div>
223   <div style={{ margin: 10px 0 }}>
224     <div style={{ display: 'flex', gap: 10px }}>
225       <div style={{ position: 'relative' }}>
226         <div style={{ position: 'absolute' }}>
227           <span style={{ color: '#000' }}>{`n.date`}</span>
228         <div style={{ border: 1px solid #ccc, padding: 5px }}>
229           {formatDate(n.createdAt, i18n.language)}
230         </div>
231       </div>
232     </div>
233   </div>
234 )
235 )
236 );
237 // === Helpers ===
238 const formatDate = (iso, lang) => {
239   const date = new Date(iso);
240   return date.toLocaleDateString(lang === 'ar' ? 'ar-EG' : 'en-US', {
241     day: 'numeric',
242     month: 'short',
243     year: 'numeric',
244   });
245 };
246 const formatTime = (iso) => {
247   return new Date(iso).toLocaleTimeString([], {
248     hour: '2-digit',
249     minute: '2-digit',
250   });
251 };
252 const isSameDay = (d1, d2) => d1.toDateString() === d2.toDateString();
253 const isThisWeek = (d) => {
254   const today = new Date();
255   const weekStart = new Date(today.setDate(today.getDate() - today.getDay()));
256   return d >= weekStart;
257 };
258 
```

Figure 5-141: Notifications.jsx (3/5)

```

162   <map (opt) => {
163     <div key={opt}>
164       <div style={{ position: 'relative' }}>
165         <div style={{ position: 'absolute' }}>
166           <span style={{ color: '#000' }}>{`selected`}</span>
167         <div style={{ border: 1px solid #ccc, padding: 5px }}>
168           {opt}
169         </div>
170       </div>
171     </div>
172   )
173 </div>
174 </div>
175 </div>
176 </div>
177 </div>
178 </div>
179 </div>
180 /* === Mark All as Read Button === */
181 <button className="page-mark-all-btn" onClick={markAllAsRead}>({`markAllRead`})</button>
182 </div>
183 </div>
184 </div>
185 /* === Notifications List === */
186 <div className="page-notif-list">
187   <loading ? (
188     <div className="page-loading">{t('loading')}{t('noNotifications')}

```

Figure 5-142: Notifications.jsx (4/5)

```

211   <div style={{ position: 'relative' }}>
212     <div style={{ position: 'absolute' }}>
213       <span style={{ color: '#000' }}>{`formatDate(n.createdAt, i18n.language)`}</span>
214     <div style={{ border: 1px solid #ccc, padding: 5px }}>
215       {formatTime(n.createdAt)}
216     </div>
217   </div>
218   <div style={{ position: 'relative' }}>
219     <div style={{ position: 'absolute' }}>
220       <span style={{ color: '#000' }}>{`n.notificationId`}</span>
221     <div style={{ border: 1px solid #ccc, padding: 5px }}>
222       <button onClick={() => markSingleAsRead(n.notificationId)}>({`FaCheck`})</button>
223     </div>
224   </div>
225   <div style={{ margin: 10px 0 }}>
226     <div style={{ display: 'flex', gap: 10px }}>
227       <div style={{ position: 'relative' }}>
228         <div style={{ position: 'absolute' }}>
229           <span style={{ color: '#000' }}>{`n.title`}</span>
230         <div style={{ border: 1px solid #ccc, padding: 5px }}>
231           {n.title}
232         </div>
233       </div>
234       <div style={{ position: 'relative' }}>
235         <div style={{ position: 'absolute' }}>
236           <span style={{ color: '#000' }}>{`n.message`}</span>
237         <div style={{ border: 1px solid #ccc, padding: 5px }}>
238           {n.content}
239         </div>
240       </div>
241     </div>
242   </div>
243 </div>
244 </div>
245 </div>
246 </div>
247 </div>
248 </div>
249 </div>
250 </div>
251 </div>
252 </div>
253 </div>
254 </div>
255 </div>
256 </div>
257 </div>
258 </div>
259 </div>
260 
```

Figure 5-143: Notifications.jsx (5/5)

Unlike RESTful HTTP requests, which operate in a request-response pattern and close the connection after each call, and unlike Server-Sent Events (SSE), which allow only one-way communication from the server to the client, WebSockets establish a bi-directional, persistent channel between the browser and the backend.

This enables both parties to send data to each other instantly without re-opening connections, making it ideal for a real-time chat feature.

In Silah, WebSocket communication is handled using the Socket.IO library, which abstracts low-level connection management and provides automatic reconnection, fallback transport, and room-based event handling. The shared utility file utils/socket.js initializes a single socket instance configured to connect to the backend's base URL, maintain credentials, and automatically join the user's private room upon successful connection.

```
src > utils > JS socket.js > ...
1 import { io } from 'socket.io-client';
2
3 const API_BASE = import.meta.env.VITE_BACKEND_URL || 'https://api.silah.site';
4
5 export const socket = io(API_BASE, {
6   withCredentials: true,
7   autoConnect: true,
8   transports: ['websocket'],
9 });
10
11 // === CONNECT & JOIN USER ROOM ===
12 socket.on('connect', () => {
13   console.log('SOCKET CONNECTED');
14   socket.emit('join_user');
15 });
16
17 // === USER ROOM CONFIRMATION (GLOBAL, PERSISTENT) ===
18 socket.on('joined_user_room', (userId) => {
19   console.log('Joined user room:', userId);
20 });
21
22 // === ERROR HANDLING ===
23 socket.on('connect_error', (err) => {
24   console.error('SOCKET ERROR:', err.message);
25 });
```

Figure 5-144: socket.js file

By keeping this connection in a centralized module, the same singleton socket instance can be imported across multiple React components; ensuring that the application maintains one consistent real-time channel per session.

This socket instance is then used extensively in the chat module to provide live updates. The Chats.jsx component listens for the new_message event to update the list of conversations in real time, reorder chats, and increment unread message counts without the need for manual refresh or API polling. Similarly, the dynamic Chats/[id].jsx page subscribes to the same event stream but at a more granular level, listening for incoming messages within an open conversation and displaying them instantly.

When a user sends a message, the frontend emits a send_message event through the same WebSocket connection. The server acknowledges the message via a callback, allowing the UI to replace the temporary (optimistic) message with the confirmed one

from the backend; preserving message order and consistency. Additionally, the client uses a join_chat and leave_chat pattern to ensure that only users participating in the conversation receive message events for that chat room.

```
useEffect(() => {
  if (!socketInitialized.current) return;

  socket.on('new message', (data) => {
    const msg = data.message || data;
    if (!msg.chatId) return;

    setChats((prev) => {
      const updated = [...prev];
      const chatIndex = updated.findIndex((c) => c.chatId === msg.chatId);

      if (chatIndex === -1) {
        fetchChats();
        return prev;
      }

      const chat = updated[chatIndex];
      updated[chatIndex] = {
        ...chat,
        lastMessage: msg.text || t('imageMessage'),
        lastMessageTime: msg.createdAt,
        unreadCount: chat.unreadCount + 1,
        isRead: false,
      };

      updated.splice(chatIndex, 1);
      updated.unshift(updated[chatIndex]);
      return updated;
    });
  });

  socketInitialized.current = true;
}, [t]);
```

Figure 5-145: Receiving new messages logic on Chats.jsx

```
// *** SOCKET: Receive new messages ***
useEffect(() => {
  if (!currentUserId) return;

  const handleNewMessage = (data) => {
    const msg = data.message || data;
    if (!msg.messageId) return;

    const senderId = msg.senderId || msg.sender?.userId;
    if (!senderId || senderId === currentUserId) return;

    const alreadyExists = messages.some((m) => m.messageId === msg.messageId);
    if (alreadyExists) return;

    // Add the incoming message
    setMessages((prev) => [
      ...prev,
      {
        messageId: msg.messageId,
        text: msg.text || null,
        senderId,
        createdAt: msg.createdAt,
        imageUrl: msg.imageUrl || null,
        isRead: msg.isRead ?? false,
      },
    ]);

    // Debounced mark all unread as read
    if (!msg.isRead && currentChatId) {
      const timeoutWindow = readTimeout;
      const readTimeout = setTimeout(() => {
        markReceivedMessagesAsReadBulk(currentChatId);
      }, 500);
    }
  };
});

socket.on('new message', handleNewMessage);
return () => {
  socket.off('new message', handleNewMessage);
  clearTimeout(readTimeout);
};
}, [currentUserId, currentChatId]);
```

Figure 5-146: Receiving new messages logic on Chats/[id].jsx

```
const sendMessage = (file) => {
  if (!input.value || !isCurrentUser) return;

  const tempId = `temp-${Date.now()}`;
  const template = `temp-$${Date.now()}`;
  const message = file.textContent;
  if (isCurrentUser) {
    payload.receiverId = receiverId;
    message = '';
  } else {
    payload.chatId = currentChatId;
    payload.receiverId = partnerUserId;
  }

  // 1. Optimize UI
  setMessages((prev) => [
    ...prev,
    {
      messageId: tempId,
      text: message,
      senderId: currentUserId,
      createdAt: new Date().toISOString(),
      imageUrl: null,
      isRead: false,
    },
  ]);

  // 2. CLEAR INPUT + DRAFT IMMEDIATELY
  setInput('');
  const draftKey = localStorage.getItem(`draftKey-${tempId}`);
  if (draftKey) localStorage.removeItem(draftKey);
  localStorage.removeItem(`tempDraft`);

  // 3. Send via socket
  socket.emit('send message', payload, (ack) => {
    console.log('ACK received:', ack);
  });

  if (!message) return;

  const backlogMsg = ack.message;
  const backlogIdFromServer = backlogMsg.chatId;

  // Replace temp message
  setMessages((prev) => prev.filter((m) => m.messageId === tempId));
  if (backlogIdFromServer === tempId) {
    const prev = [...prev];
    prev[0].messageId = backlogMsg.messageId;
    prev[0].text = backlogMsg.text;
    prev[0].senderId = backlogMsg.senderId;
    prev[0].createdAt = backlogMsg.createdAt;
    prev[0].imageUrl = backlogMsg.imageUrl || null;
    prev[0].isRead = true;
  }
  return update();
}

// Handle new chat + real chat
if (isCurrentUser) {
  setCurrentChatId(chatIdFromServer);
  navigate(`/user/chats/${newChatIdFromServer}`, { replace: true });
}

setInterval(() => markReceivedMessagesAsReadBulk(currentChatId), 500);
};
```

Figure 5-147: Send a message logic on Chats/[id].jsx

```
const sendImage = async (file) => {
  if (!file || isNewChat || !currentChatId) {
    alert('Send a text message first.');
    return;
  }

  if (file.size > 5 * 1024 * 1024) {
    alert('Max 5MB.');
    return;
  }

  const tempId = `temp-${Date.now()}`;
  const previewUrl = URL.createObjectURL(file);

  setMessages((prev) => [
    ...prev,
    {
      messageId: tempId,
      text: null,
      senderId: currentUserId,
      createdAt: new Date().toISOString(),
      imageUrl: previewUrl,
      isRead: true,
    },
  ]);

  const formData = new FormData();
  formData.append('file', file);

  try {
    await axios.post(`${API_BASE}/api/chats/me/${currentChatId}/upload`, formData, {
      headers: { 'Content-Type': 'multipart/form-data' },
      withCredentials: true,
    });
  } catch (err) {
    alert('Failed to send image.');
    setMessages((prev) => prev.filter((m) => m.messageId !== tempId));
  }
};
```

Figure 5-148: Send an image logic on Chats/[id].jsx

Through this architecture, Silah achieves real-time, low-latency communication for chat while maintaining clean separation of concerns and a single persistent WebSocket session.

5.2.3.3 AI-Augmented Pages

The Silah frontend integrates two artificial intelligence-powered features that enhance user experience and operational decision-making: the LaBSE-based recommendation system and the Prophet-based demand forecasting module. While the AI models themselves are hosted and executed on the backend, the frontend acts as the interface through which users interact with these intelligent services. Both components rely on RESTful APIs exposed by the backend and communicate via Axios requests, receiving structured JSON responses that are dynamically rendered on the client side.

The first AI integration, powered by LaBSE (Language-agnostic BERT Sentence Embeddings), supports the Smart Search Alternatives page. This feature allows buyers to find semantically related products and services, even when using varied terminology or language. The page accepts either a product ID or a free-text query through URL parameters and then sends an API request to `/api/smart-search`. Upon receiving the ranked list of similar items, the frontend parses and displays up to ten results, each rendered using the reusable ItemCard component. This ensures visual consistency across the marketplace and provides users with an intuitive, localized interface powered by the i18next library. Error handling and loading states are also implemented to manage cases where items are unavailable or the AI service is temporarily unreachable.

```

src/pages/Wayer/Alternatives.js
  ...
  import { useState, useEffect, useResetEffect } from 'react';
  import { useSearchParams } from 'react-router-dom';
  import { useTranslation } from 'react-i18next';
  import axios from 'axios';
  import ItemCard from './components/itemCard/itemCard';
  import './smartSearch.css';

  export default function Alternatives() {
    const { t, i18n } = useTranslation('smartSearch');
    const [searchParams] = useSearchParams();
    const rawItemId = searchParams.get('itemId')?.trim() ?? '';
    const rawText = searchParams.get('text')?.trim() ?? '';
    const text = rawText ? decodeURIComponent(rawText) : '';

    const lang = i18n.language;
    const i18nDir = i18n.dir() === 'rtl' ? 'rtl' : 'ltr';

    const [items, setItems] = useState([]);
    const [originalQuery, setOriginalQuery] = useState('');
    const [loading, setLoading] = useState(false);
    const [error, setError] = useState(null);

    // PAGE TITLE
    useEffect(() => {
      document.title = t('pageTitle');
      i18n.language;
    }, [t, i18n.language]);

    // MAIN EFFECT
    useEffect(() => {
      // 1. Validate query params
      if (!rawItemId || !rawText) {
        setError('invalidInput');
        setLoading(false);
        return;
      }

      if (rawText) {
        setOriginalQuery(text);
        setLoading(false);
        return;
      }

      // 2. Resolve original item name (only if itemId provided)
      const resolveOriginal = async () => {
        if (!rawItemId) {
          return;
        }

        const base = import.meta.env.VITE_BACKEND_URL;

        try {
          // Try product first
          const res = await axios.get(`${base}/api/products/${rawItemId}/lang-${lang}`);
          const mapped = {
            ...res.data,
            withCredentials: true,
          };
          setOriginalQuery(res.data.name);
        } catch (e) {
          ...
        }
      };
    }, [rawItemId, rawText]);
  }

```

Figure 5-149: Alternatives.jsx (1/3)

```

    ...
    ) catch (e) {
      // Not a product - try service
      try {
        const res = await axios.get(`${base}/api/services/${rawItemId}/lang-${lang}`);
        ...
        withCredentials: true,
      } catch (e2) {
        // Invalid ID (404 or unknown) - show user-friendly error
        setError('itemNotFound');
        setLoading(true);
      }
    }
  }

  // -----
  // 3. Call smart search
  ...
  const callSmartSearch = async () => {
    setLoading(true);
    setError(null);

    try {
      const base = import.meta.env.VITE_BACKEND_URL;
      const body = { rawItemId ? { itemId: rawItemId } : { text } };
      const res = await axios.post(`${base}/api/smart-search?lang=${lang}`, body, {
        ...options,
        withCredentials: true,
      });
      ...
      // Sort by rank (rank 1 = first)
      const sorted = res.data ?? [];
      sorted.sort((a, b) => (a.rank ?? Infinity) - (b.rank ?? Infinity));
      .slice(0, 10); // max 10
      options['sorted'];
    } catch (err) {
      const msg =
        err.response?.data?.error?.message ||
        err.response?.data?.message ||
        'An error occurred';
      setError(msg);
      console.error(err);
    } finally {
      setLoading(false);
    }
  };

  // -----
  // Execute flow
  ...
  if (rawItemId) {
    resolveOriginal().then(() => {
      if (error) callSmartSearch();
    });
  } else {
    setOriginalQuery(text);
    callSmartSearch();
  }
}, [rawItemId, rawText, text, lang, t, error]);

```

Figure 5-150: Alternatives.jsx (2/3)

```

  ...
  // UI HELPERS
  ...
  const noResults = () => (
    <p className='status'>
      {t('noResults')} <strong>{originalQuery}</strong>.
    </p>
  );
  ...

  const showEmpty = !originalQuery && !loading;

  return (
    <div className='smart-search-page' dir={[i18n.dir()]}>
      ...
      <div className='search-header'>
        ...
        {t('title')} <strong>{originalQuery}</strong>
      </div>
      ...
      <div className='results-grid-wrapper'>
        <main className='results'>
          {showEmpty ? (
            <p className='status'>{t('enterSearchText')}
          ) : loading ? (
            <p className='status'>{t('loading')}
          ) : error ? (
            <p className='status error'>{error}</p>
          ) : items.length === 0 ? (
            noResults()
          ) : (
            <div className='results-grid'>
              {items.map(({ item }, idx) => {
                const mapped = {
                  ...item,
                  _id: item.productId || item.serviceId,
                  name: item.name,
                  supplier: item.supplier,
                  businessName: item.supplier?.businessName ?? 'Unknown',
                  supplierId: item.supplier?.supplierId,
                };
                ...
                imagesFileUrls: item.imagesFileUrls ?? [],
                avgRating: item.avgRating ?? 0,
                ratingsCount: item.ratingsCount ?? 0,
                price: item.price ?? 0,
                type: item.productid ? 'product' : 'service',
              });
            )}
          )
        </main>
      </div>
    </div>
  );
}

```

Figure 5-151: Alternatives.jsx (3/3)

The second integration is the Demand Prediction page, designed for suppliers subscribed to the Premium plan. This module communicates with the backend endpoint /api/demand-predictions/:id, which returns the AI-generated forecast produced by the Prophet time-series model. The frontend then visualizes the returned data using Chart.js, leveraging the Line component to display predicted demand over time. In addition, the page highlights the recommended restock quantity, enabling suppliers to make data-driven inventory decisions. The interface dynamically adapts based on the supplier's plan status: active premium users view the complete forecast chart, while free or inactive users are presented with a blurred teaser and an upgrade prompt. This conditional rendering pattern enhances engagement while preserving access control.

```

1 // pages/Supplier/Demand.jsx
2 // 1.1 Load initial state
3 import React from 'react';
4 import { useAuth, useNavigate } from 'react-router-dom';
5 import { useTranslation } from 'react-i18next';
6 import { useState } from 'react';
7 import { Line } from 'react-chartjs-2';
8 import {
9   Chart as ChartJS,
10   CategoryScale,
11   LinearScale,
12   PointElement,
13   LineElement,
14   Title,
15   Tooltip,
16   Filler,
17 } from 'chart.js';
18 import './Demand.css';
19
20 ChartJS.register(
21   CategoryScale,
22   LinearScale,
23   PointElement,
24   LineElement,
25   Title,
26   Tooltip,
27   Legend,
28   Filler,
29 );
30
31 export default function DemandPrediction() {
32   const [t, setTime] = useState('predictDemand');
33   const [id] = useState();
34   const [navigate] = useNavigate();
35   const [userRole, supplierStatus] = useAuth();
36   const [auth] = useState(true);
37
38   // ...
39   // 1.1: STATE
40
41   const [plan, setPlan] = useState(null); // 'PREDICTION' | null
42   const [product, setProduct] = useState(null);
43   const [stock, setStock] = useState(null);
44   const [recommendedStock, setRecommendedStock] = useState(0);
45   const [loading, setLoading] = useState(true);
46   const [error, setError] = useState(null);
47
48   // ...
49   // 1.2: USER ACCESS (are computed on every render)
50   // ...
51   const [isSupplier] = auth === 'supplier';
52   const isActive = supplierStatus === 'ACTIVE';
53   const isPremium = plan === 'PREMIUM';
54   const canAccess = isSupplier && isActive && isPremium;
55
56   // ...
57   // 2. PREDICT PLAN = (IF PREMIUM) PATCH FORECAST
58   // ...
59   useEffect(() => {
60     // Reset everything
61     setPlan(null);
62     setProduct(null);
63     setStock(null);
64     setForecast([]);
65     setRecommendedStock(0);
66     setError(null);
67     setLoading(true);
68   }, []);

```

Figure 5-152: Demand/[id].jsx (1/5)

```

66   // 3.1 Not a supplier / inactive - show teaser
67   // ...
68   if (!isSupplier || !isActive) {
69     console.log('[Demand] Not supplier or inactive - teaser');
70     setLoading(false);
71     return;
72   }
73
74   // 3.2 Fetch supplier plan
75   // ...
76   const fetchPlan = async () => {
77     try {
78       const base = import.meta.env.VITE_BACKEND_URL;
79       console.log(`[Demand] Fetching plan from ${base}/api/suppliers/me/plan`);
80       const res = await axios.get(`${base}/api/suppliers/me/plan`, {
81         withCredentials: true,
82       });
83       console.log(`[Demand] Plan response: ${res.data}`);
84       setPlan(res.data.plan);
85     } catch (err) {
86       console.error(`[Demand] Plan fetch failed: ${err}`);
87       setError(err);
88       setLoading(false);
89     }
90   };
91
92   // ...
93   // 3.3 Fetch forecast (only if we are premium + either no plan is set)
94   // ...
95   const fetchForecast = async () => {
96     if (!plan || !isPremium) {
97       return;
98     }
99     const id = plan?.product?.id;
100     const res = await axios.get(`${base}/api/demand-predictions/${id}`);
101     setLoading(false);
102     return;
103   };
104
105   try {
106     const base = import.meta.env.VITE_BACKEND_URL;
107     console.log(`[Demand] Fetching forecast for id: ${id}`);
108     const res = await axios.get(`${base}/api/demand-predictions/${id}`);
109     setLoading(false);
110     const d = res.data;
111     const product = {
112       name: d.productName,
113       image: d.productFirstImageFileUrl,
114     };
115     const forecastId = forecast;
116     const recommendedStock = recommendedStock;
117     const forecast = {
118       id: forecastId,
119       product,
120       stock: recommendedStock,
121       recommendedStock,
122     };
123     setForecast(forecast);
124     setRecommendedStock(recommendedStock);
125   } catch (err) {
126     const status = err.response?.status;
127     const msg = err.response?.data?.message;
128     const errResponse = err.response?.data?.error?.message || err.response?.data?.error;
129     console.error(`[Demand] Forecast error: ${status}, ${msg}, ${err}`);
130   }
131 }

```

Figure 5-153: Demand/[id].jsx (2/5)

```

131     let uiMsg = t('genericError');
132     if (status === 400) uiMsg = msg || t('notEnoughData');
133     else if (status === 401) uiMsg = msg || t('upgradeRequired');
134     else if (status === 403) uiMsg = msg || t('forbidden');
135     else if (status === 404) uiMsg = t('productNotFoundOrNotYours');
136     else if (status === 502) uiMsg = t('uiUnavailable');
137     else if (msg) uiMsg = msg;
138
139     setError(uiMsg);
140   } finally {
141     setLoading(false);
142   }
143 }
144
145 // 3.4 Execute plan + then conditionally forecast
146 //
147 fetchPlan().then(() => {
148   // **Now** 'plan' state is updated - 'isPremium' will be true on next render
149   // We **don't** use 'isPremium' here. Instead, we let the next render decide.
150   // So we just stop loading.
151   setLoading(false);
152 }, [id, isSupplier, isActive, t]);
153
154 // ...
155
156 // 4. SECOND EFFECT: Run forecast *after* plan is known
157
158 useEffect(() => {
159   if (!canAccess || !id) return;
160
161   // At this point: plan is loaded + canAccess is correct
162   const fetchForecast = async () => {
163     setLoading(true);
164     try {
165       const base = import.meta.env.VITE_BACKEND_URL;
166       console.log(`[Demand] (2nd effect) Fetching Forecast for id: ${id}`);
167       const res = await axios.get(`${base}/api/demand-predictions/${id}`, {
168         withCredentials: true,
169       });
170       console.log(`[Demand] Forecast response: ${res.data}`);
171
172       const d = res.data;
173       setProduct({
174         name: d.productName,
175         image: d.productFirstImageFileUrl,
176       });
177       setForecast(d.forecast);
178       setRecommendedStock(d.recommendedStock);
179     } catch (err) {
180       const status = err.response.status;
181       const msg =
182         err.response.data.error.message || err.response.data.message;
183       console.error(`[Demand] Forecast error: ${status}, ${msg}, ${err}`);
184
185       let uiMsg = t('genericError');
186       if (status === 400) uiMsg = msg || t('notEnoughData');
187       else if (status === 401) uiMsg = msg || t('upgradeRequired');
188       else if (status === 403) uiMsg = msg || t('forbidden');
189       else if (status === 404) uiMsg = t('productNotFoundOrNotYours');
190       else if (status === 502) uiMsg = t('uiUnavailable');
191       else if (msg) uiMsg = msg;
192
193       setError(uiMsg);
194     } finally {
195       setLoading(false);
196     }
197   };
198 });

```

Figure 5-154: Demand/[id].jsx (3/5)

```

199   fetchForecast);
200   }, [canAccess, id, t]);
201
202 // ...
203 // 5. CHART CONFIG
204 //
205 const chartData = {
206   labels: forecast.map(f => f.month),
207   datasets: [
208     {
209       label: t('demandLabel'),
210       data: forecast.map(f => f.demand),
211       borderColor: '#9577ff',
212       backgroundColor: '□ #f0f0f0',
213       fill: true,
214       tension: 0.4,
215       pointBackgroundColor: '#5177ff',
216       pointRadius: 5,
217     },
218     {
219       label: t('currentStockLabel'),
220       data: forecast.map(f => 48),
221       borderColor: '#a780ff',
222       backgroundColor: '□ #f0f0f0',
223       fill: false,
224       borderDash: [5, 5],
225       pointRadius: 0,
226     },
227   ],
228 };
229
230 const chartOptions = {
231   responsive: true,
232   maintainAspectRatio: false,
233   plugins: [
234     legend: { position: 'top', rtl: isRTL },
235     tooltip: { rtl: isRTL },
236   ],
237   scales: {
238     y: {
239       beginAtZero: true,
240       ticks: { stepSize: 10 },
241       grid: { color: '□ #f0f0f0' },
242     },
243     x: { grid: { display: false } },
244   },
245 };
246
247
248 // ...
249 // 6. RENDER
250 //
251 const showPremium = canAccess && !loading && !error && product;
252 const showTeaser = !canAccess && !loading;
253
254 return (
255   <main className="predict-demand-page" dir={i18n.dir}>
256     </header>
257     <div className="page-header">
258       <h1>t('title')</h1>
259       <p className="subtitle">t('subtitle')</p>
260     </div>
261
262     <main className="content-area">
263       <loading ? >
264         <p className="status">t('loading')</p>
265       : error ? >
266         <p className="status error">t(error)</p>
267       : showPremium ? >
268         <section className="forecast-section">
269           <div className="product-card">
270             <img
271               src={product.image}
272               alt={product.name}
273               className="product-image"
274             />
275             <h2>t(product.name)</h2>
276           </div>
277           <p>t('chartLabel')</p>
278           <div className="chart-container">
279             <Line data={chartData} options={chartOptions} />
280           </div>
281
282           <p>t('restockAdvice')<br/>(t('restockAdvice', { count: recommendedStock }))</p>
283         </section>
284       : showTeaser ? >
285         <section className="teaser-section">
286           <div className="blurred-overlay">
287             <div className="teaser-content">
288               <h2>t('teaserTitle')</h2>
289               <p>t('teaserText')</p>
290               <button
291                 onClick={() => navigate('/supplier/choose-plan')}
292                 className="upgrade-btn"
293               >
294                 t('upgradeNow')
295               </button>
296             </div>
297           </div>
298         </section>
299       </div>
300     </main>
301
302     <div className="blurred-chart">
303       <Line
304         data={...chartData,
305         datasets: chartData.datasets.map(d => ({ ...d,
306           ...d,
307           borderColor: '□ #f0f0f0',
308           backgroundColor: '□ #f0f0f0',
309         })),
310       }
311       >
312         <options={...chartOptions, interaction: { mode: null }}>
313       </Line>
314     </div>
315
316     <p>t('restockAdvice', { count: 87 })</p>
317   </section>
318   <p>t('genericError')</p>
319   </div>
320 </div>
321
322   <p>t('genericError')</p>
323   </div>
324 </main>
325 </div>
326 </div>
327 </div>

```

Figure 5-155: Demand/[id].jsx (4/5)

```

262   <main className="content-area">
263     <loading ? >
264       <p className="status">t('loading')</p>
265     : error ? >
266       <p className="status error">t(error)</p>
267     : showPremium ? >
268       <section className="forecast-section">
269         <div className="product-card">
270           <img
271             src={product.image}
272             alt={product.name}
273             className="product-image"
274           />
275           <h2>t(product.name)</h2>
276         </div>
277         <p>t('chartLabel')</p>
278         <div className="chart-container">
279           <Line data={chartData} options={chartOptions} />
280         </div>
281
282         <p>t('restockAdvice')<br/>(t('restockAdvice', { count: recommendedStock }))</p>
283       </section>
284     : showTeaser ? >
285       <section className="teaser-section">
286         <div className="blurred-overlay">
287           <div className="teaser-content">
288             <h2>t('teaserTitle')</h2>
289             <p>t('teaserText')</p>
290             <button
291               onClick={() => navigate('/supplier/choose-plan')}
292               className="upgrade-btn"
293             >
294               t('upgradeNow')
295             </button>
296           </div>
297         </div>
298       </section>
299     </div>
300
301     <div className="blurred-chart">
302       <Line
303         data={...chartData,
304         datasets: chartData.datasets.map(d => ({ ...d,
305           ...d,
306           borderColor: '□ #f0f0f0',
307           backgroundColor: '□ #f0f0f0',
308         })),
309       }
310       >
311         <options={...chartOptions, interaction: { mode: null }}>
312       </Line>
313     </div>
314   </div>
315
316   <p>t('restockAdvice', { count: 87 })</p>
317 </section>
318 <p>t('genericError')</p>
319 </div>
320 </div>
321
322 <p>t('genericError')</p>
323 </div>
324 </main>
325 </div>
326 </div>
327 </div>

```

Figure 5-156: Demand/[id].jsx (5/5)

In both AI-enhanced modules, the frontend maintains the principle of clear separation between computation and presentation. All model inference occurs on the backend, ensuring scalability and security, while the client focuses on responsive visualization, user interaction, and multilingual accessibility. Together, these integrations demonstrate how Silah's frontend extends beyond static interfaces to become an intelligent, insight-driven platform that empowers both buyers and suppliers.

5.2.3.4 Integration with External Services: Tap Payments

On the frontend, Tap Payments integration was implemented to handle all user payment interactions securely, ensuring that Silah never directly processes or stores any sensitive card information. Instead, the frontend communicates with Tap's official JavaScript SDK, which handles card input, validation, and tokenization entirely on Tap's servers through an embedded iframe. This design choice isolates Silah from PCI-DSS compliance scope while maintaining a smooth user experience.

When a buyer navigates to the payment section in their settings page, the frontend first checks if the user already has a saved payment method. If no card is found, the Tap SDK is dynamically loaded and initialized within the TapCardForm component. This component is responsible for rendering the secure, sandboxed card input fields hosted by Tap, ensuring that sensitive card data never passes through Silah's codebase. Once the user submits their card details, Tap generates a one-time token that represents the card data. This token is sent to Silah's backend via the endpoint /api/buyers/me/card, along with a redirect URL used by Tap for 3D Secure authentication.

```

10 import React from 'react';
11 import { useState } from 'react';
12 import { useMutation } from 'react-query';
13 import './TapCardForm.css';
14
15 const TapCardForm = ({ onFormSubmitted, isActive, onError, customerId }) => {
16   const [t, tRef] = useState(''); // state for TapCardForm
17   const lang = '+100'; // language (en, es, fr, de, etc)
18   const amount = '1'; // amount in cents (100 for 1 SAR)
19   const transactionId = 'transactionId'; // unique transaction ID
20   const returnUrl = 'http://localhost:3001'; // return URL
21   const userIp = '127.0.0.1'; // user IP address
22
23   const formFields = [
24     {
25       field: 'customer',
26       value: 'Silah',
27     },
28     {
29       field: 'card',
30       value: 'Visa',
31     },
32     {
33       field: 'amount',
34       value: amount,
35     },
36     {
37       field: 'lang',
38       value: lang,
39     },
40     {
41       field: 'transactionId',
42       value: transactionId,
43     },
44     {
45       field: 'userIp',
46       value: userIp,
47     },
48     {
49       field: 'returnUrl',
50       value: returnUrl,
51     },
52     {
53       field: 'customerEmail',
54       value: 'silah@silah.com',
55     },
56     {
57       field: 'customerPhone',
58       value: '+91 9876543210',
59     },
60     {
61       field: 'customerAddress',
62       value: 'Silah Office, Silah',
63     },
64     {
65       field: 'customerCity',
66       value: 'Silah City',
67     },
68     {
69       field: 'customerState',
70       value: 'Silah State',
71     },
72     {
73       field: 'customerZip',
74       value: '1234567890',
75     },
76     {
77       field: 'customerCountry',
78       value: 'Silah Country',
79     },
80     {
81       field: 'customerCardNumber',
82       value: '4242424242424242',
83     },
84     {
85       field: 'customerCardExpiry',
86       value: '2025-12',
87     },
88     {
89       field: 'customerCardCvv',
90       value: '123',
91     },
92     {
93       field: 'customerCardName',
94       value: 'Silah Silah',
95     },
96     {
97       field: 'customerCardType',
98       value: 'Visa',
99     },
100    {
101      field: 'customerCardHolder',
102      value: 'Silah Silah',
103    },
104    {
105      field: 'customerCardAddress',
106      value: 'Silah Office, Silah',
107    },
108    {
109      field: 'customerCardCity',
110      value: 'Silah City',
111    },
112    {
113      field: 'customerCardState',
114      value: 'Silah State',
115    },
116    {
117      field: 'customerCardZip',
118      value: '1234567890',
119    },
120    {
121      field: 'customerCardCountry',
122      value: 'Silah Country',
123    },
124    {
125      field: 'customerCardNumber',
126      value: '4242424242424242',
127    },
128    {
129      field: 'customerCardExpiry',
130      value: '2025-12',
131    },
132    {
133      field: 'customerCardCvv',
134      value: '123',
135    },
136    {
137      field: 'customerCardName',
138      value: 'Silah Silah',
139    },
140    {
141      field: 'customerCardType',
142      value: 'Visa',
143    },
144    {
145      field: 'customerCardHolder',
146      value: 'Silah Silah',
147    },
148    {
149      field: 'customerCardAddress',
150      value: 'Silah Office, Silah',
151    },
152    {
153      field: 'customerCardCity',
154      value: 'Silah City',
155    },
156    {
157      field: 'customerCardState',
158      value: 'Silah State',
159    },
159  ];

```

Figure 5-157: TapCardForm.jsx
(1/3)

```

160   try {
161     const response = await axios.put(
162       `${import.meta.env.VITE_BACKEND_URL}/api/buyers/me/card`,
163       {
164         tokenId,
165         redirectUrl: window.location.origin + '/buyer/payment/callback?type=card',
166         withCredentials: true,
167       },
168     );
169
170     console.log(` Redirecting user to Tap OTP page: ${response.data.transactionUrl}`);
171     window.location.href = response.data.transactionUrl;
172   } catch (err) {
173     setError(
174       err.response?.data?.error?.message || t('errors.saveCardFailed'),
175     );
176   }
177   finally {
178     setLoading(false);
179   }
180 }
181
182 export default TapCardForm;

```

Figure 5-158:
TapCardForm.jsx (2/3)

Figure 5-159:
TapCardForm.jsx
(3/3)

Upon receiving this token, the backend creates a small verification charge (1 SAR) using Tap’s API, enabling the “save card” feature. Tap may then redirect the buyer to its hosted OTP page for additional verification (3D Secure). After successful verification, the buyer is redirected back to Silah through the callback page (/buyer/payment/callback?type=card), which is handled by the PaymentCallback component. This component extracts the Tap transaction ID from the query parameters and calls the backend to confirm the card save operation via the /api/buyers/me/card/confirm endpoint. Once confirmed, the user’s payment method is securely stored on Tap’s system and linked to their Silah account for future use.

```

<div className="tap-card-wrapper">
  <TapCardForm
    isActive={activeTab === 'payment'}
    onTokenGenerated={handleTokenGenerated}
    onError={handleFormError}
    customerId={user.tapCustomerId}
    t={t}
  />
  {error && <p className="error-text">{error}</p>}
</div>

```

Figure 5-160: TapCardFrom usage in
Buyer/Settings.jsx

```

const handleTokenGenerated = async (tokenId, cardId) => {
  setLoading(true);
  setError('');

  try {
    const { data } = await axios.put(
      `${import.meta.env.VITE_BACKEND_URL}/api/buyers/me/card`,
      {
        tokenId,
        redirectUrl: window.location.origin + '/buyer/payment/callback?type=card',
        withCredentials: true,
      },
    );
  } catch (err) {
    setError(
      err.response?.data?.error?.message || t('errors.saveCardFailed'),
    );
  }
  finally {
    setLoading(false);
  }
};

```

Figure 5-161: handleTokenGenerated method in
Buyer/Settings.jsx

```

src > pages > Buyer > Payment > ✎ Callback.jsx > PaymentCallback
You last week | 1 author (You)
1 import React, { useEffect, useState } from 'react';
2 import { useSearchParams, useNavigate } from 'react-router-dom';
3 import axios from 'axios';
4 import { userTranslation } from 'react-i18next';
5 import './PaymentCallback.css';
6
7 export default function PaymentCallback() {
8   const [loading, setLoading] = useState(false);
9   const [searchParams] = useSearchParams();
10  const [navigate] = useNavigate();
11
12  const [message, setMessage] = useState('');
13  const [loading, setLoading] = useState(true);
14  const [isSuccess, setIsSuccess] = useState(false);
15  const [secondsLeft, setSecondsLeft] = useState(3);
16  const [paymentConfig] = [
17    {
18      card: {
19        confirmEndpoint: `${import.meta.env.VITE_BACKEND_URL}/api/buyers/me/card/confirm`,
20        requestBody: (params) => ({ chargeId: params.get('tap_id') }),
21        redirectTo: '/buyer/settings?tab=payment',
22        successKey: 'success.cardSaved',
23        errorKey: 'errors.saveCardFailed',
24      },
25      checkout: {
26        confirmEndpoint: `${import.meta.env.VITE_BACKEND_URL}/api/cart/checkout`,
27        requestBody: (params) => ({ chargeId: params.get('tap_id') }),
28        redirectTo: (params) => `/order/confirmation/orderId=${params.get('orderId')} || ''`,
29        successKey: 'success.orderCompleted',
30        errorKey: 'errors.checkoutFailed',
31      },
32      invoice: {
33        confirmEndpoint: `${import.meta.env.VITE_BACKEND_URL}/api/cart/checkout`,
34        requestBody: (params) => ({
35          chargeId: params.get('tap_id'),
36          invoiceId: params.get('invoiceId') || '',
37        }),
38        redirectTo: (params) => `/invoices/${params.get('invoiceId')} || ''`,
39        successKey: 'success.invoicePaid',
40        errorKey: 'errors.paymentFailed',
41      },
42    },
43  ];
44}
45

```

Figure 5-162: Payment/Callback.jsx (1/3)

```

// 3. ORIGINAL PAYMENT LOGIC - UNCHANGED
// https://github.com/silah/buyer/pull/1000
46
47 const payment = async () => {
48   const chargeId = searchParams.get('tap_id');
49   const type = searchParams.get('type') || 'card';
50   const cardId = paymentConfig[type].card;
51
52   if (!chargeId) {
53     setMessage('Sorry, missing chargeId');
54     return;
55   }
56
57   try {
58     const res = await fetch(`https://api.silah.com/v1/tap/charge`, {
59       method: 'POST',
60       headers: {
61         'Content-Type': 'application/json',
62         'Authorization': `Bearer ${localStorage.getItem('token')}`,
63       },
64     });
65
66     const response = await res.json();
67     if (response.error) {
68       setMessage(response.error.message || 'Something went wrong');
69     } else {
70       setMessage(`Success! ${response.message}`);
71     }
72   } catch (e) {
73     setMessage(`Error: ${e.message}`);
74   }
75
76   finishPayment();
77 }
78
79 // 4. CONFIRMATION LOGIC
80 const finishPayment = () => {
81   if (localStorage.getItem('tap_id')) {
82     if (localStorage.getItem('otp')) {
83       // otp
84     } else {
85       // 3d secure
86     }
87   }
88 }
89
90 // 5. FINISHING LOGIC
91 const finishPayment = () => {
92   if (localStorage.getItem('tap_id')) {
93     if (localStorage.getItem('otp')) {
94       // otp
95     } else {
96       // 3d secure
97     }
98   }
99 }
100
101 // 6. SETTING UP INTERVAL
102 const interval = setInterval(() => {
103   if (secondsLeft < 1) {
104     clearInterval(interval);
105   }
106   const type = searchParams.get('type') || 'card';
107   const config = paymentConfig[type];
108   const redirectTo =
109     type === 'card'
110       ? config.redirectTo
111       : type === 'invoice'
112         ? config.redirectTo
113         : config.redirectTo;
114
115   navigate(redirectTo);
116   return;
117 }, 1000);
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145

```

Figure 5-163: Payment/Callback.jsx (2/3)

```

163  return () => clearInterval(timer);
164  if ([loading, searchParams, navigate]) {
165    // 3. READER - NEW CARD USE
166    return (
167      <div className="payment-callback-backdrop">
168        <div className="payment-card">
169          <Loading />
170          <div className="spinner-container">
171            <div className="spinner">
172              <span>Tap</span>
173            </div>
174          </div>
175          <div className="icon"><Icon icon="success" /><Icon icon="error" /></div>
176          <div style={{ display: 'flex', align-items: 'center' }}>
177            <div style={{ flex: 1 }}>
178              <div style={{ display: 'flex', align-items: 'center' }}>
179                <div style={{ flex: 1 }}>
180                  <div style={{ display: 'flex', align-items: 'center' }}>
181                    <div style={{ flex: 1 }}>
182                      <div style={{ display: 'flex', align-items: 'center' }}>
183                        <div style={{ flex: 1 }}>
184                          <div style={{ display: 'flex', align-items: 'center' }}>
185                            <div style={{ flex: 1 }}>
186                              <div style={{ display: 'flex', align-items: 'center' }}>
187                                <div style={{ flex: 1 }}>
188                                  <div style={{ display: 'flex', align-items: 'center' }}>
189                                    <div style={{ flex: 1 }}>
190                                      <div style={{ display: 'flex', align-items: 'center' }}>
191                                        <div style={{ flex: 1 }}>
192                                          <div style={{ display: 'flex', align-items: 'center' }}>
193                                            <div style={{ flex: 1 }}>
194                                              <div style={{ display: 'flex', align-items: 'center' }}>
195                                                <div style={{ flex: 1 }}>
196                                                  <div style={{ display: 'flex', align-items: 'center' }}>
197                                                    <div style={{ flex: 1 }}>
198                                                      <div style={{ display: 'flex', align-items: 'center' }}>
199                                                        <div style={{ flex: 1 }}>
200                                                          <div style={{ display: 'flex', align-items: 'center' }}>
201                                                            <div style={{ flex: 1 }}>
202                                                              <div style={{ display: 'flex', align-items: 'center' }}>
203                                                                <div style={{ flex: 1 }}>
204                                                                  <div style={{ display: 'flex', align-items: 'center' }}>
205                                                                    <div style={{ flex: 1 }}>
206                                                                      <div style={{ display: 'flex', align-items: 'center' }}>
207                                                                        <div style={{ flex: 1 }}>
208                                                                          <div style={{ display: 'flex', align-items: 'center' }}>
209                                                                            <div style={{ flex: 1 }}>
210                                                                              <div style={{ display: 'flex', align-items: 'center' }}>
211                                                                                <div style={{ flex: 1 }}>
212                                                                                  <div style={{ display: 'flex', align-items: 'center' }}>
213                                                                                    <div style={{ flex: 1 }}>
214                                                                                      <div style={{ display: 'flex', align-items: 'center' }}>
215                        </div>
216                      </div>
217                    </div>
218                  </div>
219                </div>
220              </div>
221            </div>
222          </div>
223        </div>
224      </div>
225    );
226  }
227}
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245

```

Figure 5-164: Payment/Callback.jsx (3/3)

The same integration pattern is reused during the checkout flow. When a buyer proceeds to pay for an order, the frontend verifies the presence of a saved card before initiating payment. If no card is found, the user is redirected back to the Settings page specifically to the Payment tab to save one first. Otherwise, the checkout process triggers a new Tap charge request through the backend, which in turn redirects the user to Tap's OTP or 3D Secure page if required. Upon completion, Tap redirects back to Silah's /buyer/payment/callback?type=checkout, where the PaymentCallback component again confirms the payment status with the backend through the /api/cart/checkout endpoint.

Both TapCardForm and PaymentCallback operate under strict security guidelines. TapCardForm only receives a short-lived token from Tap's iframe, never the actual card number or CVV. The PaymentCallback, on the other hand, ensures that every payment or card confirmation request is revalidated by the backend using Tap's official APIs before updating any local state. All communications between the frontend, backend, and Tap's API occur over HTTPS, further protecting users' financial information.

By integrating Tap Payments in this manner, Silah achieves a secure and compliant payment system where all sensitive operations are fully delegated to Tap's infrastructure. The frontend acts purely as a bridge between the user and the backend; responsible for collecting non-sensitive metadata, managing redirects, and providing real-time feedback through loading states and success messages. The result is a seamless user experience that combines security, usability, and regulatory compliance without exposing Silah to unnecessary financial data risks.

5.2.3.5 Deployment and DevOps

The Silah frontend was deployed to the same DigitalOcean droplet that hosts both the NestJS backend and the FastAPI AI service, named silah-site-server. Consolidating all services into a single virtual machine simplified management, eliminated cross-domain issues, and ensured a uniform deployment environment across the entire platform. This approach allowed both teams (frontend and backend) to work against the same production-like infrastructure, promoting consistency and faster debugging during integration phases.

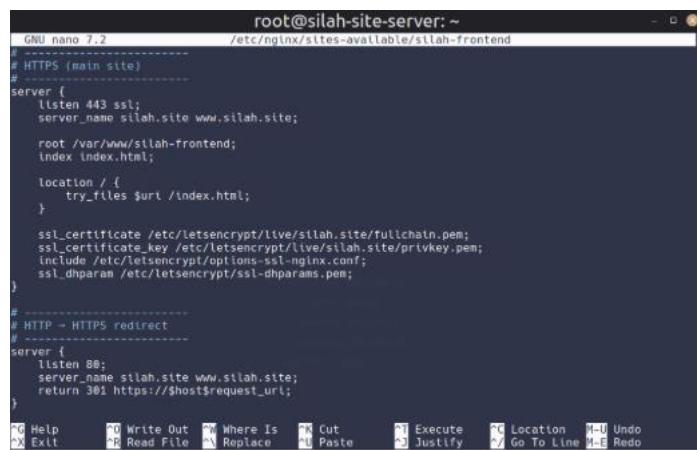
The frontend, developed with Vite and React, was built into an optimized static bundle and served through NGINX, which functions as a high-performance static file server and SSL terminator. The droplet runs Ubuntu 24.04 LTS with 2 vCPUs, 4 GB of RAM, and 80 GB SSD storage; sufficient to host all services simultaneously without noticeable latency.

After SSH access was configured using key-based authentication, the frontend repository (silah-frontend) was cloned into the server under /root/silah-frontend. During the initial setup, dependencies were installed using “npm ci” command to

guarantee reproducible builds. The project was then built with “npm run build”, producing the optimized assets inside the dist directory. These compiled files were deployed to /var/www/silah-frontend, which serves as the document root for the NGINX site configuration.

The NGINX configuration for the Silah frontend is straightforward and production-optimized. It listens on port 443 for HTTPS requests under the domains silah.site and www.silah.site, with automatic redirection from HTTP to HTTPS on port 80.

The try_files directive ensures that Vite’s client-side routing (React Router) correctly serves index.html for all frontend routes, enabling deep-linking and Single Page Application (SPA) behavior. SSL termination is managed via Let’s Encrypt, and certificates are automatically renewed by Certbot, ensuring secure, encrypted communication for all end users.



```
root@silah-site-server: ~
GNU nano 7.2
/etc/nginx/sites-available/silah-frontend
# HTTPS (main site)
#
server {
    listen 443 ssl;
    server_name silah.site www.silah.site;
    root /var/www/silah-frontend;
    index index.html;

    location / {
        try_files $uri /index.html;
    }

    ssl_certificate /etc/letsencrypt/live/silah.site/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/silah.site/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
}

# HTTP - HTTPS redirect
#
server {
    listen 80;
    server_name silah.site www.silah.site;
    return 301 https://$host$request_uri;
}
```

Figure 5-165: NGIX Configuration for frontend-silah

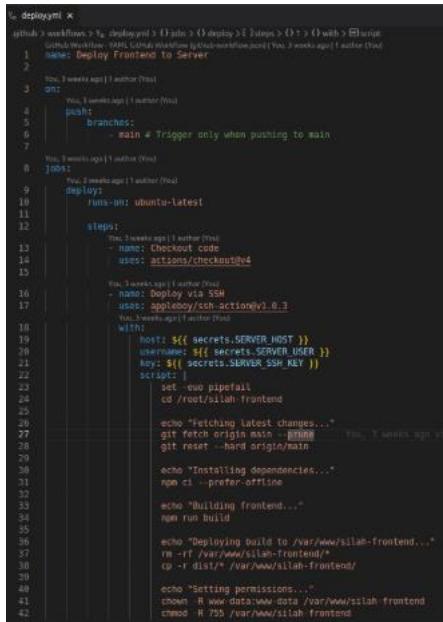
Frontend API calls (e.g., authentication, chat, notifications, demand forecasting) are made directly to https://api.silah.site, which is managed by a separate NGINX site block configured as a reverse proxy. This division ensures clean separation of concerns: one NGINX instance serves static assets for the user-facing site (silah.site), while another handles API requests (api.silah.site). Both share the same SSL infrastructure, domain ownership, and automatic certificate management via Let’s Encrypt.

To minimize manual intervention and ensure consistency across deployments, a GitHub Actions workflow was configured to automatically build and deploy the frontend upon every push to the main branch. This pipeline guarantees that the live

site always reflects the latest stable version of the codebase, reducing downtime and synchronization issues between developers.

The CI/CD workflow, defined in `.github/workflows/deploy.yml`, runs on Ubuntu-latest and uses the appleboy/ssh-action to remotely connect to the droplet via SSH. Once authenticated, the workflow performs a full deployment sequence, including fetching the latest changes, resetting the repository to the latest commit, reinstalling dependencies, building the optimized assets, and copying them to the web root directory. Permissions are then adjusted to ensure that NGINX (running under the www-data user) can properly serve the files.

This fully automated process replaces what would otherwise be a manual, error-prone procedure, making frontend releases predictable, repeatable, and instantaneous.

A screenshot of a GitHub Actions job script for a frontend deployment. The script is named `deploy.yml` and is located in the `.github/workflows` directory. It defines a single job named `Deploy Frontend to Server` with the following configuration:

```
name: Deploy Frontend to Server
on:
  push:
    branches:
      - main # Trigger only when pushing to main
  workflow_dispatch:
  pull_request:
    branches:
      - main
  schedule:
    - cron: '0 0 * * *'
  workflow_run:
    branches:
      - main
    types:
      - completed
```

The job uses the `ubuntu-latest` runner and consists of several steps:

- `steps:`
 - `- name: Checkout code`
 - `- name: actions/checkout@v4`
- `host: ${{ secrets.SERVER_HOST }}`
- `username: ${{ secrets.SERVER_USER }}`
- `key: ${{ secrets.SERVER_SSH_KEY }}`
- `script:`
 - `set -eu pipefail`
 - `cd /root/silah-frontend`
 - `echo "Fetching latest changes..."`
 - `git fetch origin main -f`
 - `git reset --hard origin/main`
 - `echo "Installing dependencies..."`
 - `npm ci --prefer-offline`
 - `echo "Building frontend..."`
 - `npm run build`
 - `echo "Deploying build to /var/www/silah-frontend..."`
 - `rm -rf /var/www/silah-frontend/`
 - `cp -r dist/* /var/www/silah-frontend/`
 - `echo "Setting permissions..."`
 - `chown -R www-data:www-data /var/www/silah-frontend`
 - `chmod -R 755 /var/www/silah-frontend`

Figure 5-166: GitHub Actions Job Script for Frontend Deployment

In essence, this deployment strategy embodies the core principles of modern DevOps (automation, reproducibility, and reliability) while maintaining the simplicity required for a university-scale yet production-quality project. By consolidating all services (the frontend, backend, and AI components) within a single DigitalOcean droplet, the architecture minimizes configuration complexity and fosters a unified development environment. Security is strengthened by exposing only ports 80 and 443, with all inter-service communication confined to localhost, effectively reducing the attack surface. Meanwhile, performance is optimized as NGINX efficiently serves cached

static assets and handles HTTPS termination with minimal overhead. Furthermore, the design remains inherently scalable, supporting seamless migration to multiple droplets or containerized environments in the future without necessitating changes to frontend code. This balanced approach achieves both operational efficiency and architectural flexibility, ensuring Silah's deployment remains maintainable, secure, and adaptable to future growth.

5.2.3.6 Frontend Design: Strengths, Trade-offs, and Path Forward

The Silah frontend represents a thoughtfully engineered, bilingual React-based application that successfully translates complex backend functionality into an intuitive, responsive, and maintainable user experience. Despite being developed by a small team with limited prior experience, it achieved a professional level of usability and cohesion, integrating real-time communication, AI-driven recommendations, and dynamic multilingual support within a consistent design system. The modular component structure, dynamic routing, and context-driven state management demonstrate sound architectural decisions that balanced code reusability, clarity, and flexibility. From an engineering perspective, the use of Vite, Axios, React Context, and i18n libraries provided a robust foundation for maintainability, performance, and localization. The decision to deploy early, working directly against live backend APIs, accelerated development and improved integration reliability; a key success factor mirrored in the backend strategy.

The frontend's strengths lie in its bilingual UX, real-time responsiveness, and consistent modularity. The integration of Right-to-Left (RTL) and Left-to-Right (LTR) layouts enabled seamless language switching between Arabic and English at runtime, offering a culturally inclusive interface uncommon in student projects. React contexts were effectively used to handle authentication, notifications, and chat, ensuring that session management and WebSocket communication remained centralized and predictable. Dynamic route and translation loading further streamlined collaboration by reducing merge conflicts and automating resource registration, a clever technical solution to early workflow challenges. Collectively, these choices reflect a solid grasp of practical frontend engineering principles and the ability to prioritize both user and developer experience.

However, the implementation also reveals natural trade-offs that emerged from limited time and evolving familiarity with React's broader ecosystem. We initially lacked awareness of powerful form libraries such as React Hook Form and Yup, leading to the creation of custom, form-specific validation logic across multiple pages. While functional, these manual implementations increased maintenance effort and introduced code duplication. Similarly, because buyer and supplier UIs were implemented as separate directory trees, several pages (such as the orders, notifications, and chats) were duplicated with only minor differences. This structure simplified role-based routing early on but made later updates and testing more time-consuming, as the same fixes or improvements had to be applied twice. A more scalable approach would have been to abstract shared logic into reusable higher-order components or parameterized routes that render role-specific data.

Testing coverage in the frontend was also minimal. Only a few basic unit tests were written, and comprehensive testing (including integration and end-to-end tests) was largely absent. This limited our ability to automatically detect regressions or ensure that complex interactions (such as real-time chat and form validation) worked as expected after updates. Introducing structured testing with Vitest and React Testing Library would significantly improve reliability and developer confidence in future iterations.

CSS organization and layout structure also presented opportunities for improvement. Styling decisions were mostly handcrafted per component, without adopting a global UI library such as Material UI or Shadcn. While this gave the project a distinctive and cohesive aesthetic, it limited consistency and increased styling overhead. Component-level duplication occasionally led to layout drift between pages, particularly when RTL adjustments were applied manually. These are not major flaws but symptoms of a frontend that prioritized functionality and delivery speed over refinement.

Looking ahead, several concrete improvements could elevate the frontend to production-grade quality. Introducing React Hook Form and Yup would drastically simplify form validation, improving both developer efficiency and input reliability. Refactoring duplicated buyer/supplier pages into shared base components would reduce maintenance costs and ensure consistent behavior. Incorporating TypeScript could improve type safety and API integration robustness. On the UX side, adopting

a component library or design system would enhance accessibility, maintain a consistent look and feel, and reduce design drift across pages. Finally, optimizing build performance through lazy loading, route-based code splitting, and improved caching would yield measurable improvements in responsiveness and load times.

In conclusion, the frontend of Silah demonstrates a strong foundation built with clarity, adaptability, and user focus. It embodies practical engineering trade-offs that favor progress over premature optimization, delivering a polished and fully functional interface that aligns with the project's vision and technical constraints. While certain abstractions and optimizations were deferred, the lessons learned (particularly around code reuse, form management, and component design) form a solid steppingstone toward more advanced frontend engineering practices in future projects.

Figure 5-167, presents an overview of the key tools, frameworks, and technologies integrated throughout Silah's development lifecycle. It aligns with the adopted Waterfall methodology, mapping each phase (design, implementation, testing, and deployment) to the tools that supported it. This summary offers a holistic view of the project's technical ecosystem, clarifying how design platforms (such as Figma and Eraser), development frameworks (React, NestJS, FastAPI), and supporting utilities (Postman, GitHub, NGINX) collectively contributed to a coherent workflow.

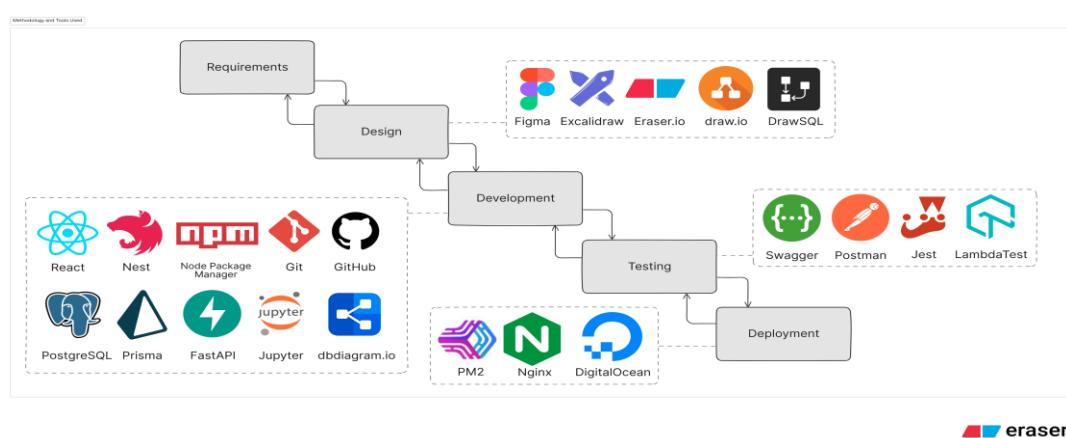


Figure 5-167: Overview of the Methodology and Tools Used in Silah's Lifecycle

5.3 Results: Interface and User Interaction Screens

This section presents the fully implemented and operational web platform as experienced by end users. The following pages illustrate the final interface across different user roles. All screens are captured directly from the deployed application in November 2025. Because the overall structure, navigation flow, and core interaction patterns remain consistent with the designs presented in Section 4.2, detailed re-description of every element is kept brief. The accompanying figures are largely self-explanatory; the text therefore highlights only notable refinements or new states that emerged during development.

5.3.1 Guest Pages

The guest user flow represents the first point of contact for visitors. These pages communicate the platform's value proposition and guide users toward registration with minimal friction.

The Landing Page

The landing page serves as the welcoming entry point, featuring a fixed header with category selection, search, language switcher, and clear “Login” and “Sign Up” calls-to-action. The hero section emphasizes the core message “Find Trusted Suppliers & Grow Your Business”, followed by the “How It Works” illustration and interactive category browsing.

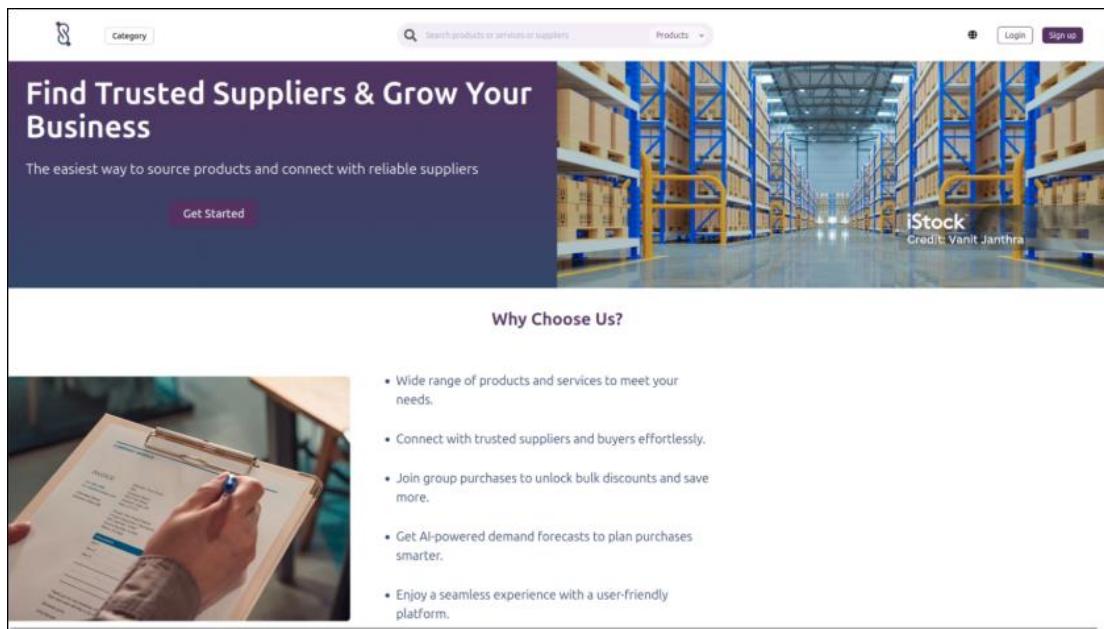


Figure 5-168: Landing Page (1/3)

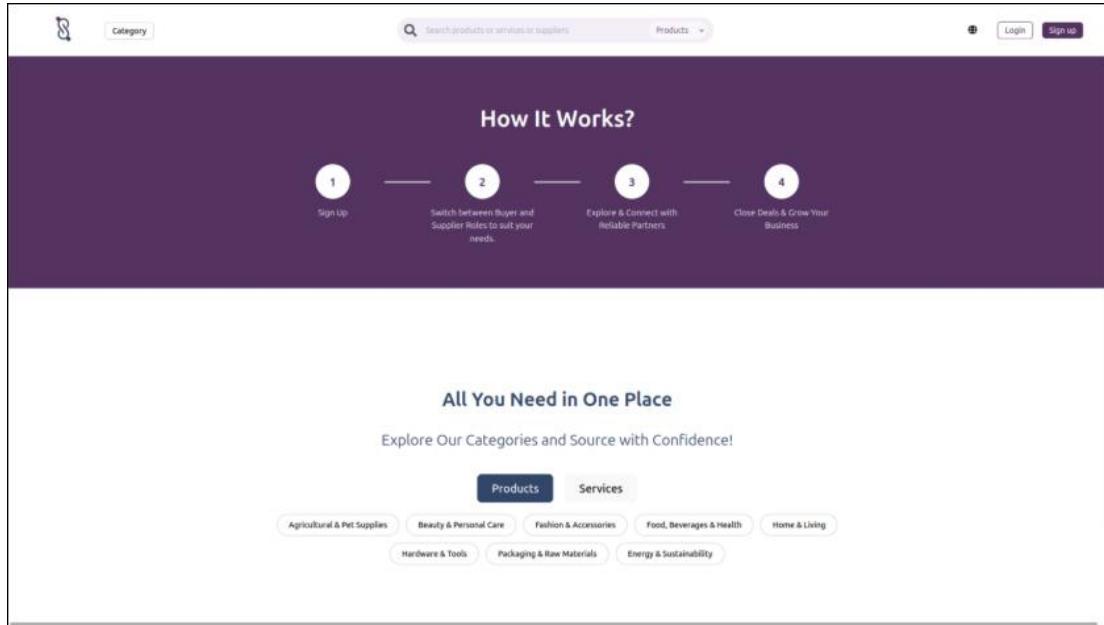


Figure 5-169: Landing Page (2/3)

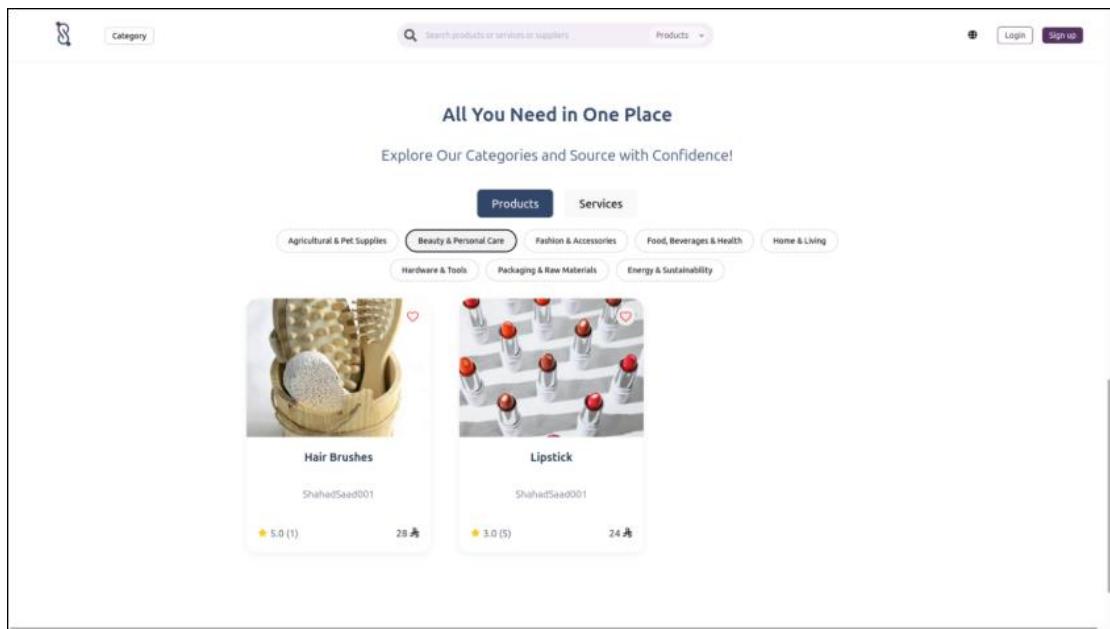


Figure 5-170: Landing Page (3/3)

The Sign-up Process

Registration follows a three-step flow: business details (name, commercial registration number, activity), personal information (name, national ID, city), and account credentials with terms acceptance. Progress indicators and validation messages ensure a smooth onboarding experience.

The screenshot shows the first step of a three-step sign-up process. The title "Join us!" is at the top. Below it is a vertical navigation bar with three items: "Business Info" (selected), "User Info", and "Account Info". The main form contains fields for "Business Name", "Commercial Registration Number", and "Select your business activity". A link "Already have an account? Log in" and a "Next" button are at the bottom.

Figure 5-171: Sign-up Page (1/3)

The screenshot shows the second step of the sign-up process. The title "Join us!" is at the top. Below it is a vertical navigation bar with three items: "Business Info", "User Info" (selected), and "Account Info". The main form contains fields for "Name", "National ID", and "City". A "Back" button and a "Next" button are at the bottom.

Figure 5-172: Sign-up Page (2/3)

The screenshot shows the third step of the sign-up process. The title "Join us!" is at the top. Below it is a vertical navigation bar with three items: "Business Info", "User Info", and "Account Info" (selected). The main form contains fields for "Email", "Password", and "Confirm Password". A checkbox "I agree on the terms and conditions" is present. A "Back" button and a "Done" button are at the bottom.

Figure 5-173: Sign-up Page (3/3)

The Login Page

Users can sign in using either their email or commercial registration number. The layout remains clean, with a visible “Forgot Password?” link and direct navigation to sign-up.

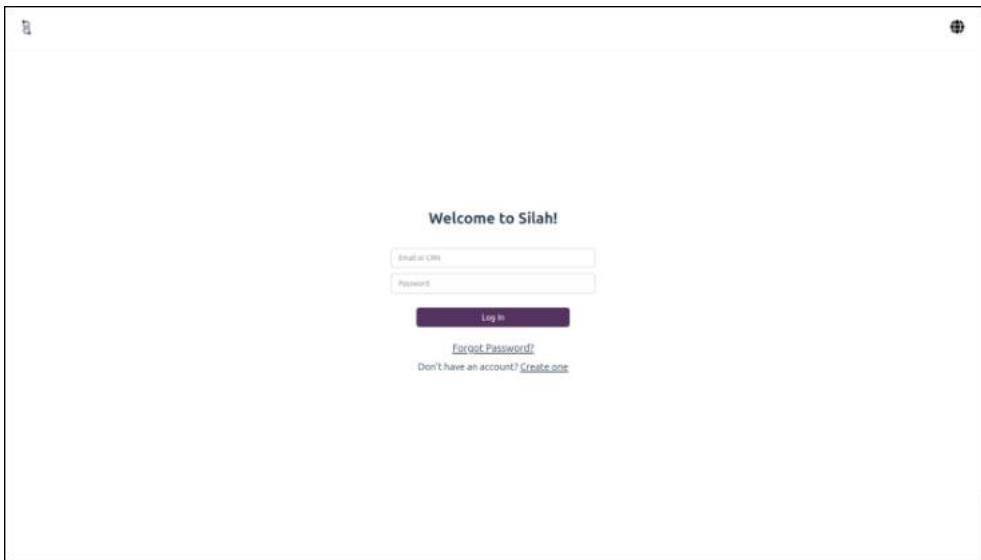


Figure 5-174: Login Page

The Email Verification Page

After registration, users are redirected to a dedicated verification screen. The page now handles four distinct states while maintaining visual consistency:

1. Initial confirmation with “Didn’t receive the email?” option
2. Resend requested state with prominent resend button
3. Confirmation that a new email has been sent
4. Success message after clicking the verification link, with direct “Log in” button

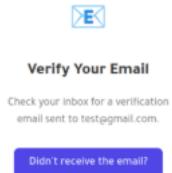


Figure 5-175: Initial Verify Email Dialog

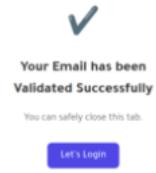


Figure 5-176: Email Successfully Verified Dialog

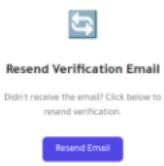


Figure 5-177: Request Resend Verification Email



Figure 5-178: Resent Verification Email Successfully

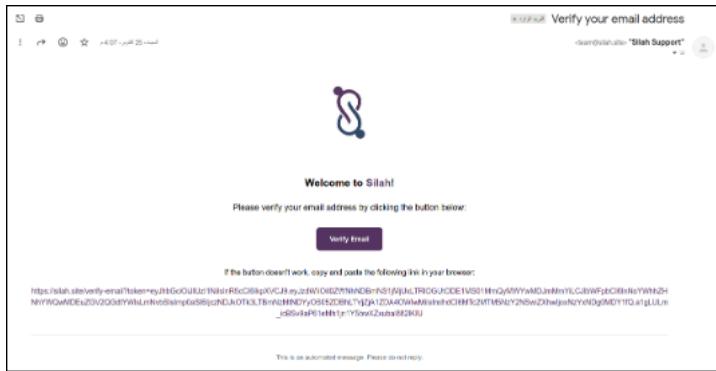


Figure 5-179: Sample verification email received by the user

5.3.2 Supplier Pages

The supplier interface provides a comprehensive environment for managing storefronts, listings, orders, bids, invoices, analytics, and buyer communication. Access is immediate upon login (or via “Switch Role”), with a persistent left sidebar for navigation.

The Overview Page

The dashboard presents key metrics via quick-action cards. A small but helpful addition clarifies store status explicitly: “Your store and listings are currently hidden from buyers” (when closed) or “Your store is visible” (when open).

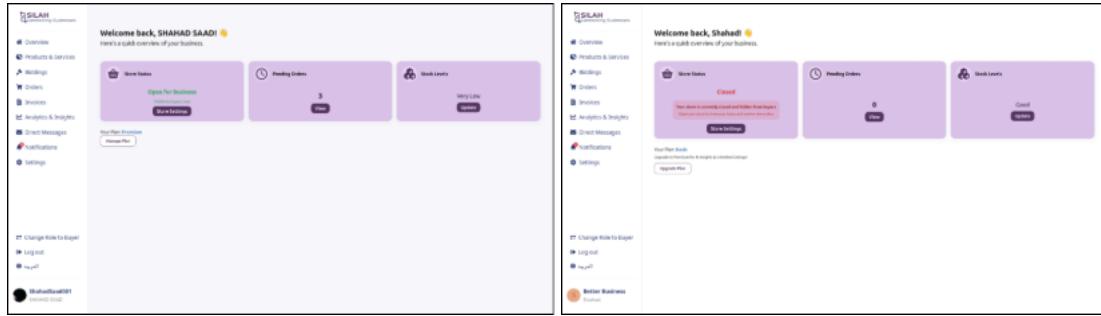


Figure 5-180: Overview Page (Opened Store)

Figure 5-181: Overview Page (Closed Store)

Manage Listings Process

The original table layout was replaced with a card-based design for better visual clarity and easier scanning. Radio-button filters were redesigned as modern toggles. Wishlist counts are shown only to Premium subscribers; Basic users see a tooltip prompting upgrade.

Select Item(s) to take an action						
	Image	Item Name	Unit Price	Stock	Status	Predict Demand
<input type="checkbox"/>		Product: Lipstick	24	1116	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: Hair Brushes	28	0	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: REAL DIAMOND RING	5807.5	20	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: شوكولاتة آيس كريم	10	0	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: Bread	4	170	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: خبز رجالي مصنوع من العسل	220	2000	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: سلطة سلطة	450	2000	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: App Development	3000	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: Web Development	3000	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: Database Design	1500	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: دعم فني أو تكنولوجيا	45	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: Technical Repair	500	—	Published	<button>Predict Demand</button>

Figure 5-182: Listings Page (Premium Plan)

Select Item(s) to take an action						
	Image	Item Name	Unit Price	Stock	Status	Predict Demand
<input type="checkbox"/>		Product: Lipstick	24	1116	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: Hair Brushes	28	0	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: REAL DIAMOND RING	5807.5	20	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: شوكولاتة آيس كريم	10	0	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: Bread	4	170	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: خبز رجالي مصنوع من العسل	220	2000	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Product: سلطة سلطة	450	2000	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: App Development	3000	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: Web Development	3000	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: Database Design	1500	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: دعم فني أو تكنولوجيا	45	—	Published	<button>Predict Demand</button>
<input type="checkbox"/>		Service: Technical Repair	500	—	Published	<button>Predict Demand</button>

Figure 5-183: Listings Page (Basic Plan)

The screenshot shows the SILAH platform's interface. At the top, there is a navigation bar with the SILAH logo and a search bar containing the text "devel". Below the search bar are buttons for "All", "Products", and "Services", along with "Create a New Product" and "Create a New Service". On the left, a sidebar lists various menu items: Overview, Products & Services, Biddings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, and Settings. Under "Products & Services", there are links for "Change Role to Buyer", "Log out", and "العربي". A user profile section at the bottom shows a placeholder profile picture and the name "ShahadSaad001 / SHAHAD SAAD". The main content area displays a table titled "Selected item(s) to take an action". The table has columns for "Image", "Item Name", "Unit Price", "Stock", "Status", and "Predict Demand". Three items are listed:

Image	Item Name	Unit Price	Stock	Status	Predict Demand
[Placeholder]	Service: App Development	1 3000	—	Published	
[Placeholder]	Service: Web Development	1 3000	—	Published	
[Placeholder]	Service: Database Design	0 1500	—	Published	

Figure 5-184: Listings Page (Search Results)

The Product Details and Service Details pages adopt the same card grouping for images, pricing, stock, categorization, and group-purchase settings, making forms far easier to complete than the original dense layout.

SILAH
Connecting Businesses

حذا رجالي مصنوع من الجلد
Created On 16 Nov 2025 9:16 pm

Product Status: Published

Basic Information

Product Details
Build buyers confidence with a clear, detailed product listing.

Name
حذا رجالي مصنوع من الجلد

Description
حذا رجالي أنيق مصنوع من جلد ينبع على الجلد، سهل
الสวม، ذات تصميم فاخر في كل طرف. ينبع هذا الجلد من
العنادن والمرجع، يعطي حذاء الـ Derby تشكيلًا مع مظهر
القدم هو مظهر الجوف دون أن يفقد ملائتها. ينبع من
كلاسيكي تفاصيل الأطوال (العصمة والخواص)، مما يجعل حذاء
 Derby مناسب لـ "البساطة والروعة" في نفس الوقت.

Category
Footwear

Images

Image Tips

- Use at least 1050px by 1050px.
- Use a white or natural background.
- Ensure bright lighting with minimal shadows.
- Include front, side, and close-up views.

Price

ShahadSaad001
SHAHAD SAAD

Figure 5-185: Supplier's Product Details Page (1/2)

Order Requirements

Case Quantity
6

Minimum Order Quantity
6

Maximum Order Quantity
Unlimited

Case Quantity
Case quantity refers to the number of units that come in a single case or package. Buyers are required to purchase products in this case quantity, meaning orders must be in multiples of the case quantity. For example, if the case quantity is 10, buyers must order in increments of 10 units (e.g., 10, 20, 30 units, etc.).

Minimum Order Quantity
The minimum order quantity is the smallest number of units that a buyer must order from you. This ensures that you can sell products in bulk, optimizing your production, inventory, and shipping processes.

Maximum Order Quantity
The maximum order quantity is the largest number of units or items a buyer can order from you at one time. This limit is set to ensure that products are distributed fairly, prevent market saturation, manage inventory effectively, and avoid stock shortages.

Group Purchasing Settings

Would you like to enable group purchasing?

Minimum Group Orders Quantity
160

Order Deadline
After 3 days

Group Purchase Price Per Unit
180

Group Purchasing
Group purchasing allows five buyers from the same city to collectively place a bulk order for a product, enabling them to access discounted pricing. This feature helps you increase sales volume, attract more buyers, and move inventory more efficiently while offering competitive pricing.

Minimum Group Orders Quantity
The minimum group order quantity is the smallest number of units or packages that must be collectively ordered by all buyers in a group purchase. This ensures that you can meet your production and inventory requirements efficiently while offering a discount to buyers.

Order Deadline
The order deadline is the specified time by which the minimum number of buyers (minimum 5) and the minimum group orders quantity must be met for the group purchase to be completed. If the requirements are not fulfilled by the deadline, the group purchase will be canceled.

ShahadSaad001
SHAHAD SAAD

Figure 5-186: Supplier's Product Details Page (2/2)

The screenshot shows a product management interface for a product named "Lipstick". The main page displays basic information like "Created On 23 Oct 2025 10:07 pm" and status "Published". A sidebar on the left includes links for Overview, Products & Services, Bidings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, and Settings. The user is logged in as "ShahadSaad001".

A modal window titled "Update Stock Levels" is open, prompting the user to change the current stock level from 1116 to 2000. The modal includes "Update" and "Cancel" buttons.

Other visible sections include "Basic Information", "Product Details" (describing the product as building buyer confidence), "Product Category" (set to "Makeup & Fragrances"), "Description" (listing "Red Lipstick"), "Quantity" (set to 2000), and "Images" (with tips for image quality).

Figure 5-187: Update Stock Modal

SILAH
Connecting Businesses

App Development
Created On 25 Oct 2025 5:02 pm
Service Status: Published

Basic Information

Service Details
Build buyers confidence with a clear, detailed service listing.

Name
App Development

Description
Applications

Service Category
This information will help us categorize your service.

Category
Web & App Development

This Service will appear to buyers in Software & IT Solutions > Web & App Development.

Images

Image Tips

- Use at least 1050px by 1050px.
- Use a white or natural background.
- Ensure bright lighting with minimal shadows.
- Include front, side, and close-up views.

You can upload up to 3 images (png, jpg, jpeg, webp).

Upload Image (png / jpg / jpeg / webp)

Price

Description
Applications

Images

Image Tips

- Use at least 1050px by 1050px.
- Use a white or natural background.
- Ensure bright lighting with minimal shadows.
- Include front, side, and close-up views.

You can upload up to 3 images (png, jpg, jpeg, webp).

Upload Image (png / jpg / jpeg / webp)

Price

Price
3000 Price negotiable
Price negotiation based on project scope

Service Details

Service Availability
24/7

Service availability refers to the times an user when the service is offered to you. This ensures that buyer know when they can expect the service to be provided, allowing them to plan accordingly. For instance, if a service is available 24/7, the buyer can request the service at any time of the day or week.

Save

ShahadSaad001
SHAHAD SAAD

Figure 5-188: Supplier's Service Details Page (1/2)

SILAH
Connecting Businesses

App Development
Created On 25 Oct 2025 5:02 pm
Service Status: Published

Basic Information

Service Details
Build buyers confidence with a clear, detailed service listing.

Name
App Development

Description
Applications

Service Category
This information will help us categorize your service.

Category
Web & App Development

This Service will appear to buyers in Software & IT Solutions > Web & App Development.

Images

Image Tips

- Use at least 1050px by 1050px.
- Use a white or natural background.
- Ensure bright lighting with minimal shadows.
- Include front, side, and close-up views.

You can upload up to 3 images (png, jpg, jpeg, webp).

Upload Image (png / jpg / jpeg / webp)

Price

Price
3000 Price negotiable
Price negotiation based on project scope

Service Details

Service Availability
24/7

Service availability refers to the times an user when the service is offered to you. This ensures that buyer know when they can expect the service to be provided, allowing them to plan accordingly. For instance, if a service is available 24/7, the buyer can request the service at any time of the day or week.

Save

ShahadSaad001
SHAHAD SAAD

Figure 5-189: Supplier's Service Details Page (2/2)

View Stock Demand

The forecasting page retains the original line-chart layout and three-month horizon. New contextual messages were added: positive feedback when stock is sufficient, encouragement when fewer than 10 sales days are recorded, and clear guidance when no sales have occurred yet. Basic-plan users still see the blurred chart with upgrade prompt.



Figure 5-190: Demand Predictions Page (Needs Restocking)

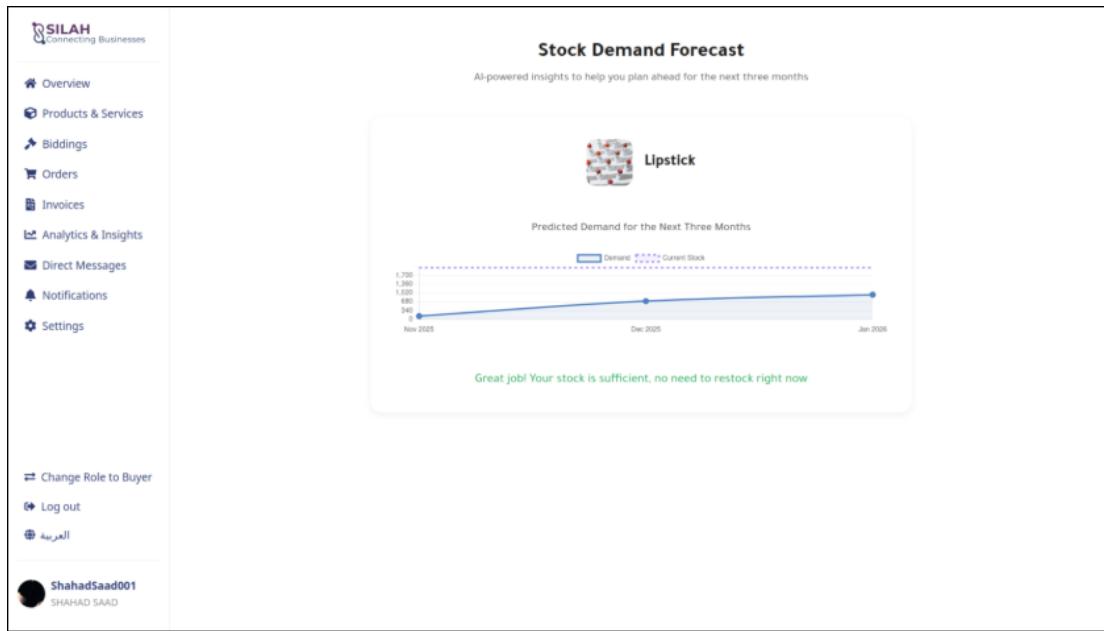


Figure 5-191: Demand Predictions Page (No Restocking Needed)

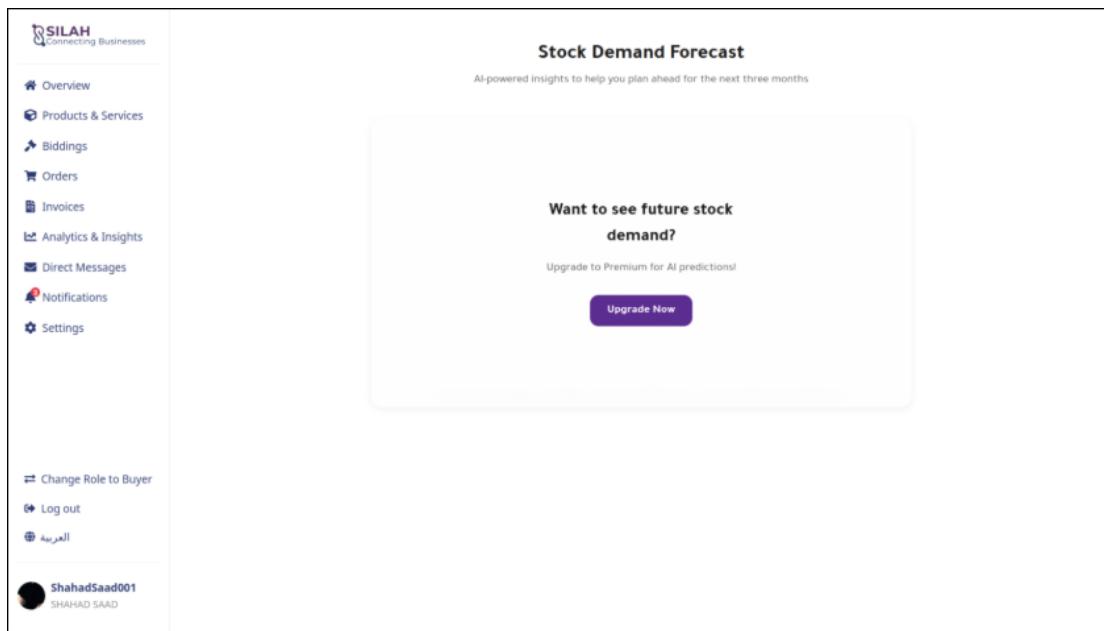


Figure 5-192: Demand Predictions Page (Basic Plan)

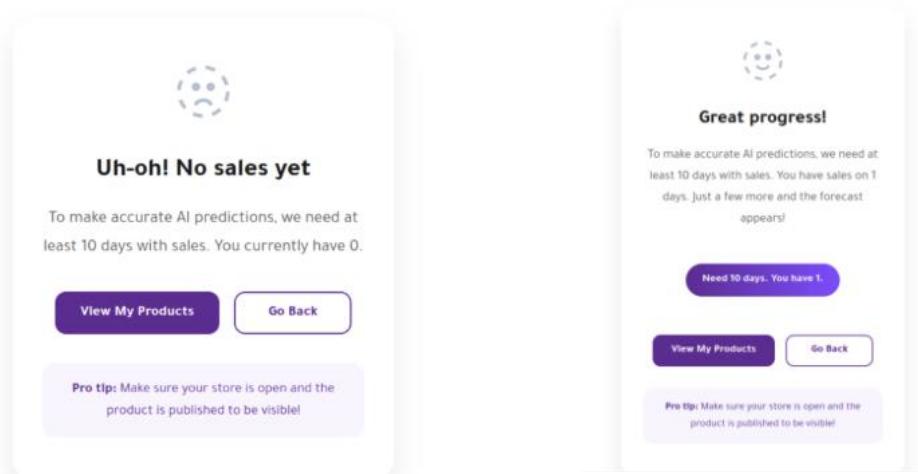


Figure 5-193: Demand Predictions Page (No Sales)

Figure 5-194: Demand Predictions Page (Not Enough Sales)

Participating in Biddings

Bid cards and page spacing were adjusted slightly for better proportions in the production environment; all content and interactions remain unchanged.

The screenshot shows the 'Open Bids' section of the SILAH platform. On the left is a sidebar with navigation links: Overview, Products & Services, Biddings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, and Settings. At the bottom of the sidebar is the user profile 'ShahadSaad001 SHAHAD SAAD'. The main content area is titled 'Open Bids' and includes a checkbox 'Show only bids I joined'. Three bid cards are listed:

- Bid 1:** Published on: 20 Nov 2025. Main Activity: توريد أدوات صحية (Supply of medical tools). Reference #: 5026679460. Submission Deadline: 27 Nov 2025. [View Details](#)
- Bid 2:** Published on: 20 Nov 2025. Main Activity: توريد عدد 100 حاتم قصبة (Supply of 100 endotracheal cuffs). Reference #: 8744892462. Submission Deadline: 25 Dec 2025. [View Details](#)
- Bid 3:** Published on: 22 Nov 2025. Main Activity: توريد مستلزمات مدرسية (Supply of school supplies). Reference #: 3960640913. Submission Deadline: 01 Dec 2025. [View Details](#)

Figure 5-195: Supplier's Biddings Page (All)

The screenshot shows the 'Open Bids' section of the SILAH platform, filtered to show only joined bids. The sidebar and user profile are identical to Figure 5-195. The main content area is titled 'Open Bids' and includes a checkbox 'Show only bids I joined'. Three bid cards are listed:

- Bid 1:** You have joined this bid. Published on: 18 Nov 2025. Main Activity: I will decline you, sorry!. Reference #: 4517944305. Submission Deadline: 18 Nov 2025. [View Details](#)
- Bid 2:** You have joined this bid. Published on: 18 Nov 2025. Main Activity: I will accept you!. Reference #: 9093054683. Submission Deadline: 18 Nov 2025. [View Details](#)
- Bid 3:** You have joined this bid. Published on: 09 Nov 2025. Main Activity: Submit 1. Reference #: 4306158646. [View Details](#)

Figure 5-196: Supplier's Biddings Page (Joined Only)

The screenshot shows the 'Bid Details' page for a supplier. The left sidebar includes links for Overview, Products & Services, Biddings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, Settings, Change Role to Buyer, Log out, and Account. The main content area displays bid information: Bid Name (Arabic: سوق المزادات), Main Activity (naifshabbab), Organization (naifshabbab), Reference # (5026679460), Time Remaining (4 day(s) remaining), Published On (20 Nov 2025), Submission Deadline (27 Nov 2025), and Expected Response Time (2 weeks after submission). Buttons for Back and Participate are at the bottom.

Figure 5-197: Supplier's Bid Details Page

The screenshot shows the 'Submit Your Offer' page. The left sidebar is identical to Figure 5-197. The main form has fields for Proposed Amount (with placeholder 'Write your proposed amount here'), Expected Completion Date (mm / dd / yyyy), Technical Details (with placeholder 'Describe your technical approach, materials, and standards...'), Execution Plan (with placeholder 'Explain timeline, phases, and delivery plan...'), and Additional Notes (Optional) (with placeholder 'Any special conditions or remarks...'). A note indicates 6/500 characters available. Buttons for Back and Submit Offer are at the bottom.

Figure 5-198: Write an Offer Page

Supplier Settings

Instead of one long scrolling page, settings are now organized under clean horizontal tabs (General, Account, Notifications, Store, Support), each using card containers for improved readability.

The screenshot shows the 'Settings' page for a supplier. On the left is a sidebar with navigation links: Overview, Products & Services, Biddings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, and Settings. Below these are links for Change Role to Buyer, Log out, and Case Closed (Shahad Saad). The main content area is titled 'Settings' and has tabs for General, Account, Notifications, Store, and Support. The 'General' tab is selected. It contains two sections: 'User Info' (Name: Shahad Saad, National ID: [redacted]) and 'Business Info' (Business Name: [redacted], Commercial Registration Number: 0065432179, Business Activity: Shopping & Logistics, Legal & Compliance Services). A 'Save Settings' button is at the bottom.

Figure 5-199: Supplier's Settings Page (General)

The screenshot shows the 'Settings' page for a supplier, specifically the 'Account' tab. The sidebar and layout are identical to Figure 5-199. The 'Account' tab is selected. The main content area contains two sections: 'Account Info' (Email: shahad.saad@hotmail.com, Password: [redacted], Current Password: [redacted], New Password: [redacted], Confirm New Password: [redacted]), and 'Subscription' (Your Current Plan: BASIC, Manage Plan button). A 'Done' button is located near the top right of the account info section. A 'Save Settings' button is at the bottom.

Figure 5-200: Supplier's Settings Page (Account)

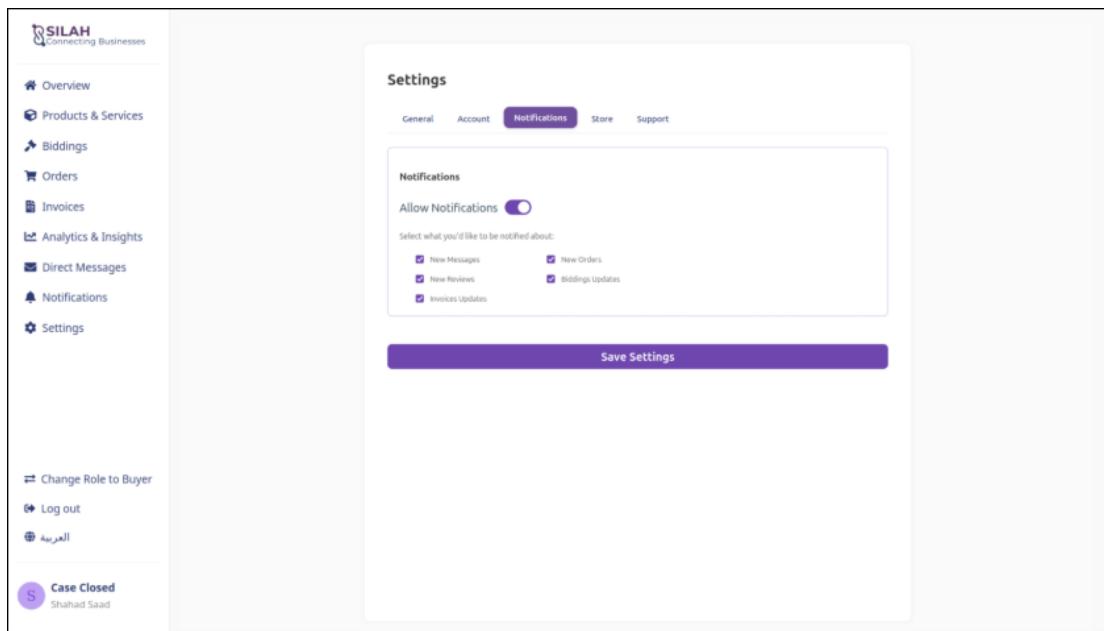


Figure 5-201: Supplier's Settings Page (Notifications)

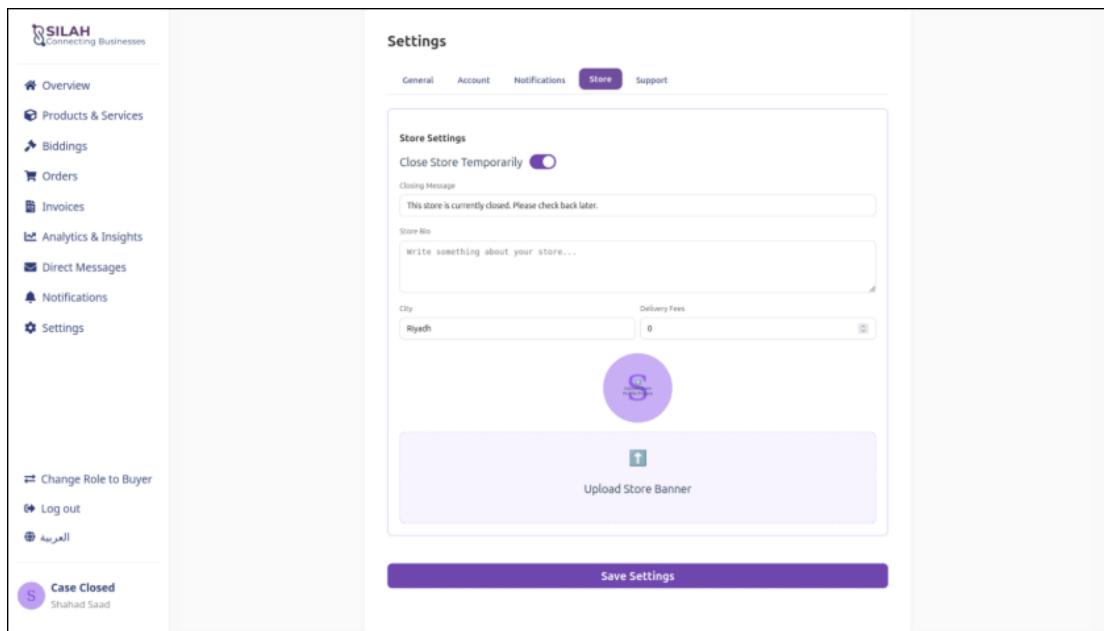


Figure 5-202: Supplier's Settings Page (Store)

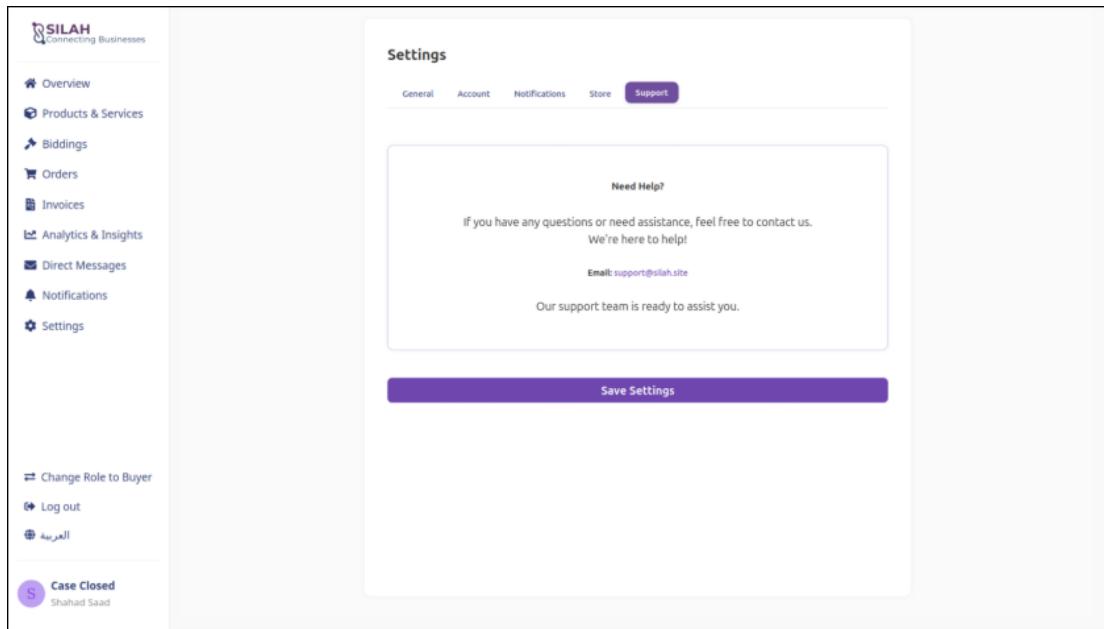


Figure 5-203: Supplier's Settings Page (Support)

Subscription Plan Management

The page presents both plans with feature comparison and a 30-day Premium trial option, exactly as designed.

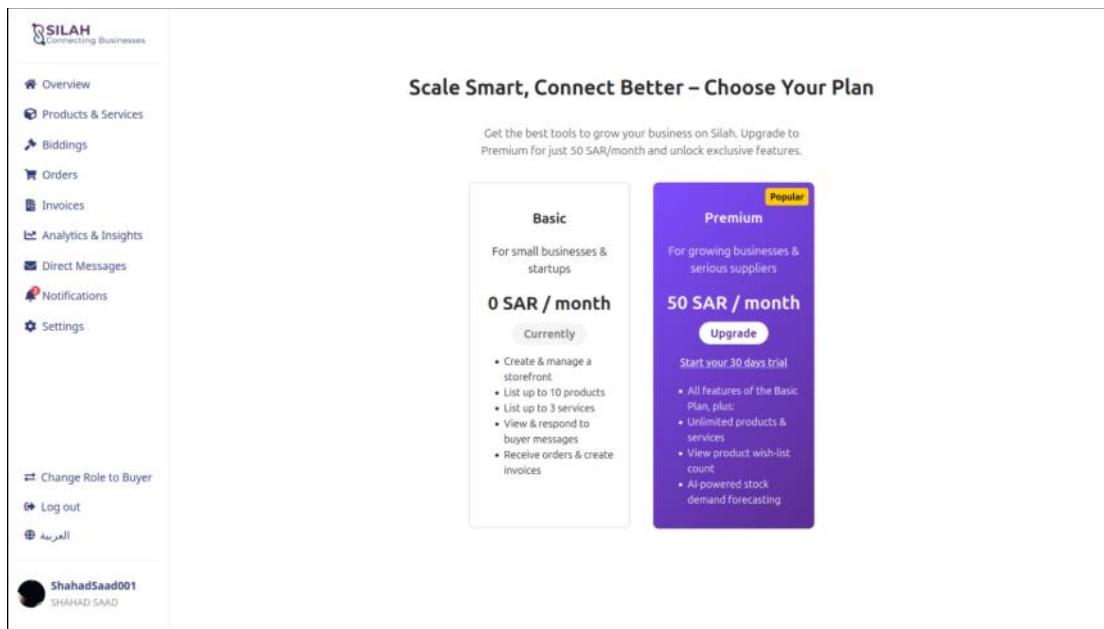


Figure 5-204: Choose Plan Page

Supplier Notifications

Unread items are marked with a purple dot. A “Mark all as read” button and brief top-left toast messages for new events were added for better real-time awareness.

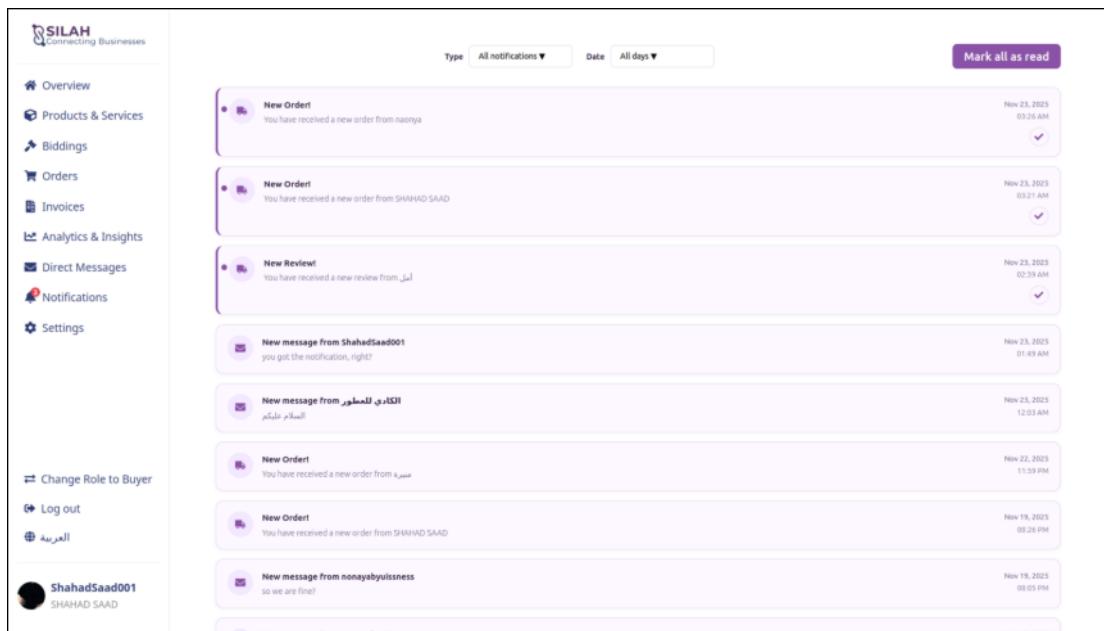


Figure 5-205: Supplier's Notifications Page

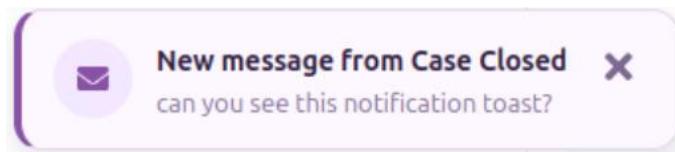
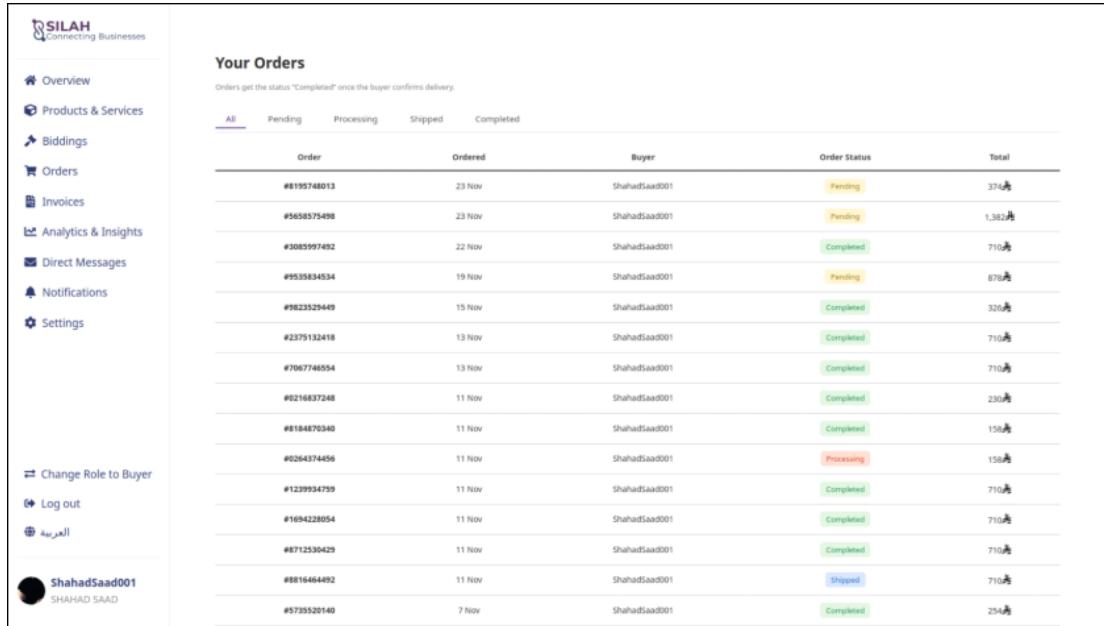


Figure 5-206: Supplier's Notification Toast

Order Management

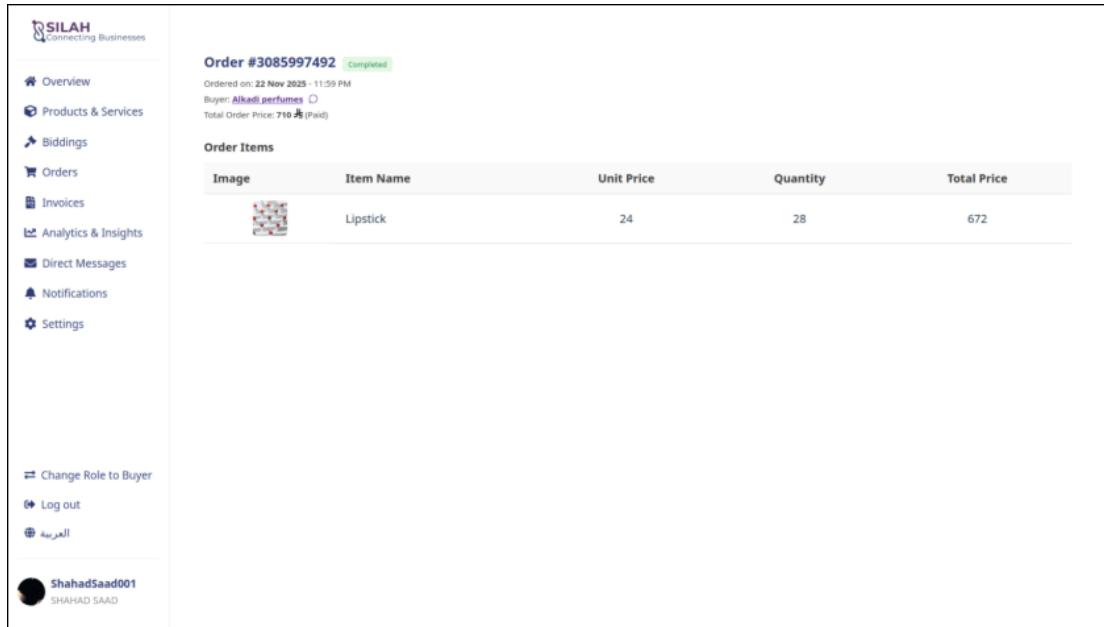
Order listing and detailed views follow the original design completely.



The screenshot shows the 'Your Orders' section of the SILAH platform. On the left is a sidebar with navigation links: Overview, Products & Services, Biddings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, and Settings. Below these are two more sections: Change Role to Buyer and Log out. At the bottom of the sidebar is a user profile for 'ShahadSaad001 SHAHAD SAAD'. The main content area is titled 'Your Orders' with a subtitle 'Orders get the status "Completed" once the buyer confirms delivery.' It includes tabs for All, Pending, Processing, Shipped, and Completed. A table lists 15 orders with columns for Order ID, Ordered date, Buyer, Order Status, and Total amount. The table shows various statuses like Pending, Completed, and Processing.

Order	Ordered	Buyer	Order Status	Total
#8199748013	23 Nov	ShahadSaad001	Pending	374 ₩
#3658575498	23 Nov	ShahadSaad001	Pending	1,382 ₩
#3085997492	22 Nov	ShahadSaad001	Completed	710 ₩
#9535834834	19 Nov	ShahadSaad001	Pending	878 ₩
#9823529449	15 Nov	ShahadSaad001	Completed	320 ₩
#2375132418	13 Nov	ShahadSaad001	Completed	710 ₩
#7067746554	13 Nov	ShahadSaad001	Completed	710 ₩
#0216837248	11 Nov	ShahadSaad001	Completed	230 ₩
#8184870340	11 Nov	ShahadSaad001	Completed	150 ₩
#0264374456	11 Nov	ShahadSaad001	Processing	150 ₩
#1239934739	11 Nov	ShahadSaad001	Completed	710 ₩
#1694228054	11 Nov	ShahadSaad001	Completed	710 ₩
#8712530429	11 Nov	ShahadSaad001	Completed	710 ₩
#8816464492	11 Nov	ShahadSaad001	Shipped	710 ₩
#5735520140	7 Nov	ShahadSaad001	Completed	250 ₩

Figure 5-207: Supplier's Orders Page



The screenshot shows the details for Order #3085997492. The top part displays basic order information: Ordered on: 22 Nov 2025 - 11:59 PM, Buyer: Alkadi perfumes, and Total Order Price: 710 ₩ (Paid). Below this is a table titled 'Order Items' with columns for Image, Item Name, Unit Price, Quantity, and Total Price. It shows one item: Lipstick at 24 per unit, quantity 28, and total price 672.

Order Items				
Image	Item Name	Unit Price	Quantity	Total Price
	Lipstick	24	28	672

Figure 5-208: Supplier's Order Details Page

Direct Messaging & Search for Chats

Conversation filtering is now inline using the same dropdown style as notifications. When an unsaved invoice draft exists for a chat, a “Continue creating invoice” button appears directly on the conversation row.

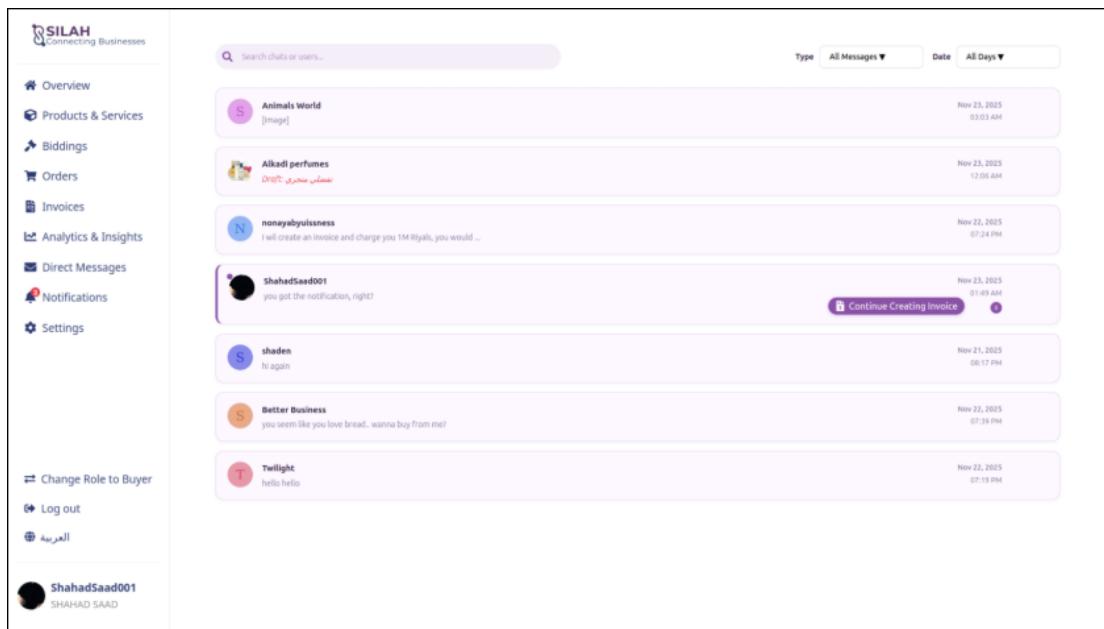


Figure 5-209: Supplier's Chats Page

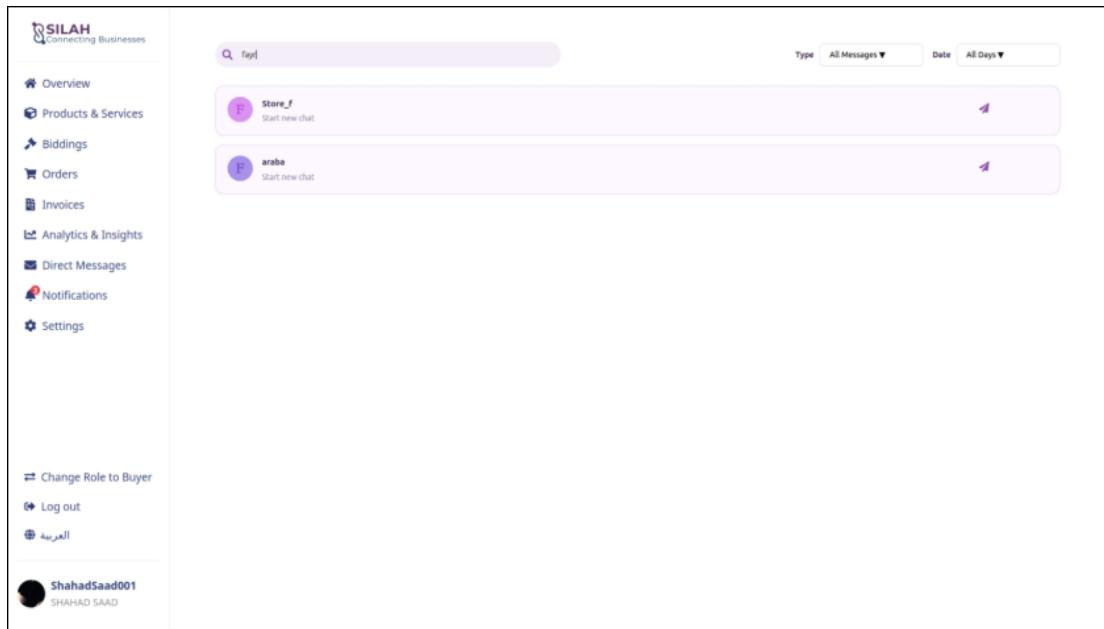


Figure 5-210: Supplier's Chats Page (Search Results)

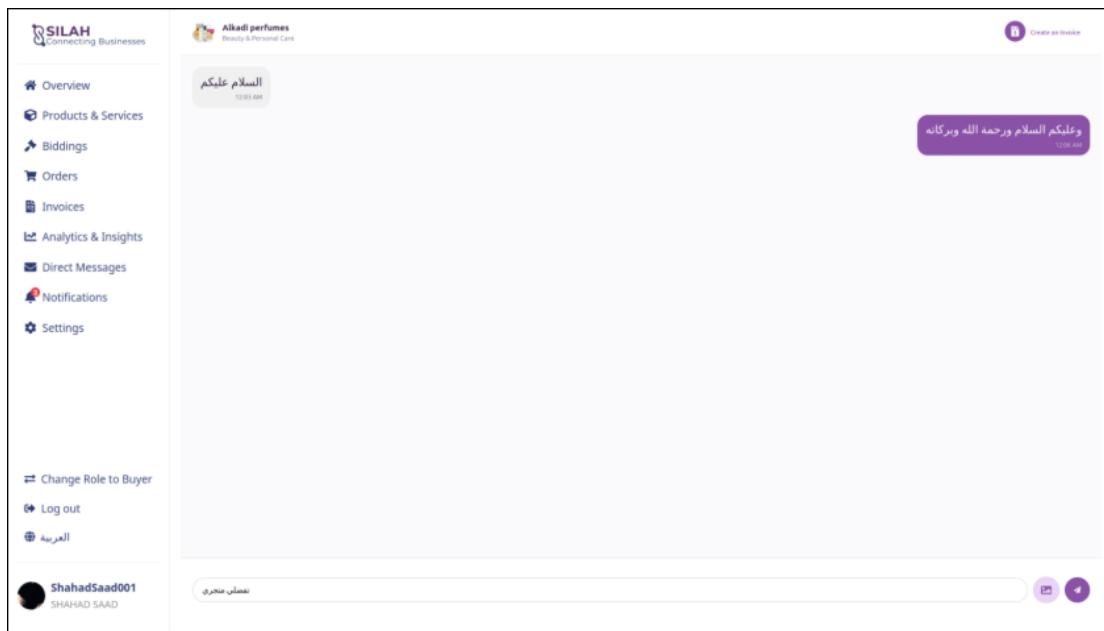


Figure 5-211: Supplier's Chat Page

Invoice Management

The most visible change concerns the Create Invoice and Invoice Details pages. Instead of the original table-based layout with compact rows, the final implementation adopts a clean, modern card-based design that is visually consistent across both creation and viewing modes. Information is grouped into clearly defined cards (buyer details, items, payment terms, totals, etc.), dramatically improving readability and reducing cognitive load. This unified card approach was intentionally applied to both pages to make the transition from invoice creation to review feel seamless and intuitive. The “Link to Listing” dialog is fully implemented and allows suppliers to associate invoice items with existing product or service listings. This linkage enables buyer reviews submitted for that invoice item to be automatically displayed on the public or service details page, thereby increasing trust and visibility for future customers.

SILAH
Connecting Businesses

Create an Invoice

Overview	Invoice Number Will be generated upon creation	Issue Date 23/11/2025						
Products & Services	Delivery Date * 11 / 30 / 2025	Terms of Payment * Fully Paid						
Biddings	Upon Delivery Amount * 200							
Orders	Supplier ShahadSaad001 SHAHAD SAAD Riyadh shahadsaad001.dev@gmail.com	Buyer ShahadSaad001 SHAHAD SAAD Riyadh shahadsaad001.dev@gmail.com						
Invoices	Items							
Analytics & Insights	Item	Description	Agreed Details	Qty	Unit Price	Total Price	Link	Actions
Direct Messages	test	test	test	1	200	200.00	✓	
Notifications	Item nam	Description	Agreed det	1	0	0.00		
Settings								
Change Role to Buyer								
Log out								
Arabic								
ShahadSaad001 SHAHAD SAAD								

Figure 5-212: Create an Invoice Page (1/2)

SILAH
Connecting Businesses

Overview	Supplier ShahadSaad001 SHAHAD SAAD Riyadh shahadsaad001.dev@gmail.com	Buyer ShahadSaad001 SHAHAD SAAD Riyadh shahadsaad001.dev@gmail.com						
Products & Services	Items							
Biddings	Item	Description	Agreed Details	Qty	Unit Price	Total Price	Link	Actions
Orders	test	test	test	1	200	200.00	✓	
Invoices	Item nam	Description	Agreed det	1	0	0.00		
Analytics & Insights								
Direct Messages								
Notifications								
Settings								
Change Role to Buyer								
Log out								
Arabic								
ShahadSaad001 SHAHAD SAAD								

Notes & Terms (Optional)
Add any additional terms or notes...
0/500

Create Invoice

Figure 5-213: Create an Invoice Page (2/2)

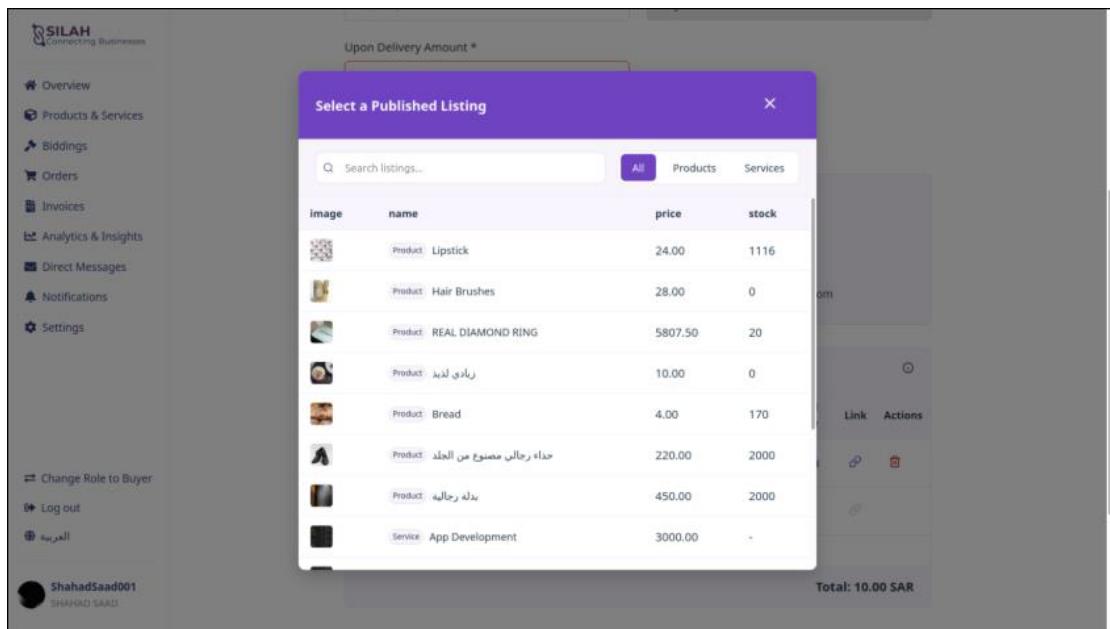


Figure 5-214: Link an Item Modal

The Invoices page retains the overall structure and filtering concept from Figma. A smart enhancement was added to the status filters: rather than always displaying every possible status, only the relevant statuses are shown depending on the selected invoice source (All, Products, Services, Bids, Group Purchases). This context-aware filtering prevents confusion and ensures suppliers can only select statuses that apply to the current view (e.g., “Partially Paid” and “Fully Paid” appear only when Products or Services are selected, while Bids and Group Purchases show only Pending, Failed, and Successful statuses).

Your Invoices						
	Invoice	Created	Buyer	Invoice Status	Pre-invoice Status	Total
	#0371446940	23 Nov	ShahadSaad001	Fully Paid	—	200₼
	#0635911840 PRE	23 Nov	ShahadSaad001	—	Pending	2,000₼
	#1326252426 PRE	23 Nov	ShahadSaad001	—	Pending	2,000₼
	#1294948349 PRE	23 Nov	ShahadSaad001	—	Pending	4,830₼
	#6783871048 PRE	23 Nov	ShahadSaad001	—	Pending	730₼
	#4773148457 PRE	22 Nov	ShahadSaad001	—	Pending	2,430₼
	#9363470498	18 Nov	ShahadSaad001	Pending	—	2₼
	#3881873438 PRE CANCELED	18 Nov	ShahadSaad001	—	Failed	1,000,000₼
	#5318416249 PRE	18 Nov	ShahadSaad001	—	Successful	2₼
	#8168066409 PRE CANCELED	18 Nov	ShahadSaad001	—	Failed	200₼
	#8090763234 PRE CANCELED	18 Nov	ShahadSaad001	—	Failed	590₼
	#4797249988	18 Nov	ShahadSaad001	Fully Paid	—	200₼
	#1377240142 PRE CANCELED	17 Nov	ShahadSaad001	—	Failed	590₼
	#5550023429	17 Nov	ShahadSaad001	Fully Paid	—	10₼
	#9174837462	17 Nov	ShahadSaad001	Fully Paid	—	10₼

Figure 5-215: Supplier's Invoices Page

Invoice #4797249988									
Supplier		Buyer		Items					
ShahadSaad001	SHAHAD SAAD	nonayabyuissness	naonya	Issue Date: 18/11/2025 Delivery Date: 19/11/2025 Terms of Payment: Fully Paid Total Amount: \$ 200.00 ₼					
Riyadh	shahadsaad001.dev@gmail.com			Item	Description	Agreed Details			
				test	test	test			
				Qty	Unit Price	Total Price			
				1	200.00	200.00			
				Total: 200.00 ₼					
Notes & Terms									
FOR YOU									

Figure 5-216: Supplier's Invoice Details Page

Pre-invoice details page (shown for Group Purchases and Bids) uses the same card-based design as regular invoices; the only visible differences are the title at the top of the page changed to “Pre-invoice” and additional contextual information explaining the underlying group purchase or offer.

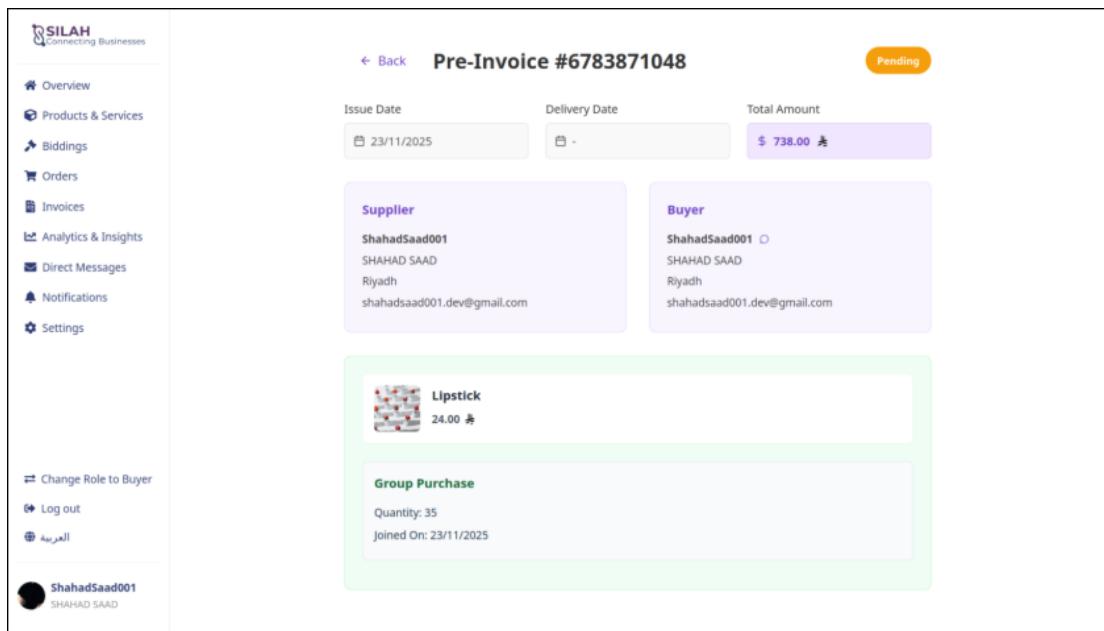


Figure 5-217: Supplier's Pre-Invoice Details Page (Group Purchase)

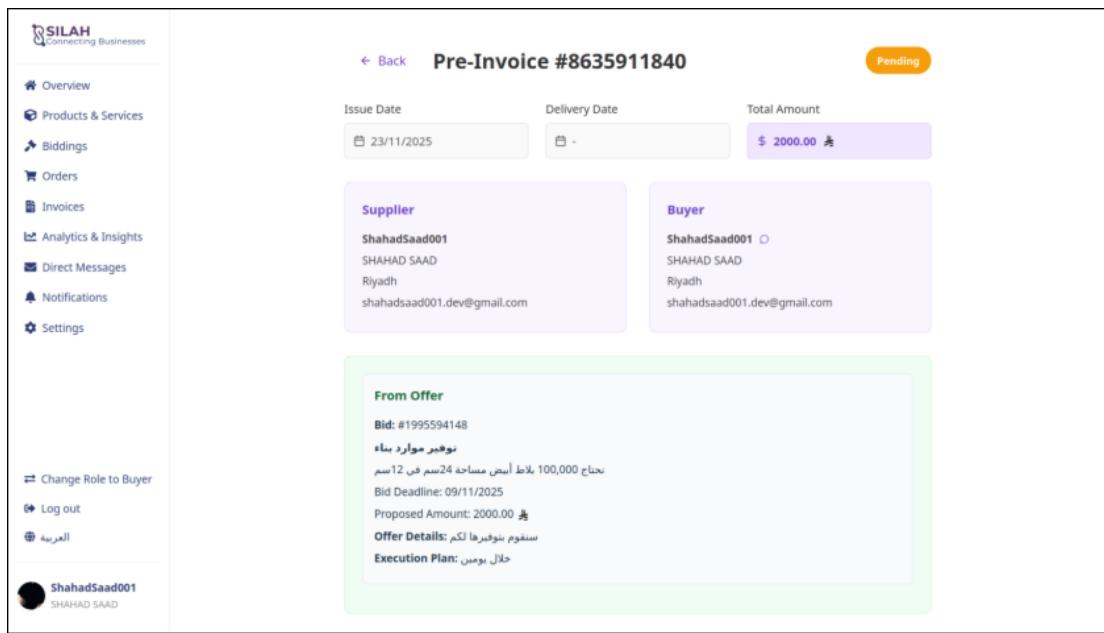


Figure 5-218: Supplier's Pre-Invoice Details Page (Offer)

Store Analytics & Insights

The dashboard shows total sales, ratings, and top items. Premium users see the most wish-listed item; Basic users receive an upgrade prompt in its place.

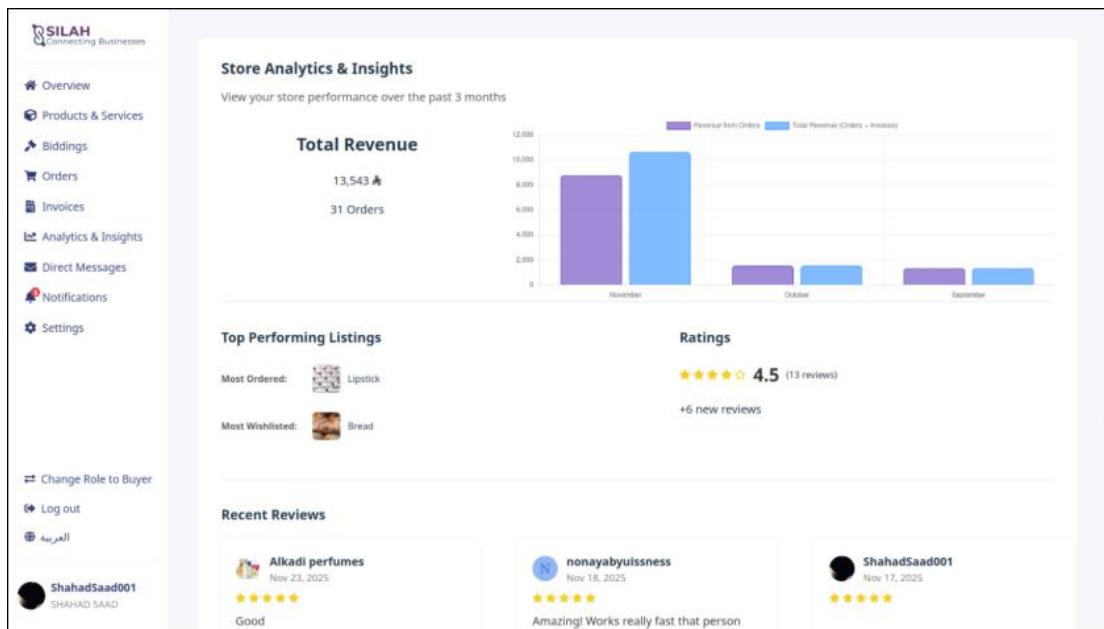


Figure 5-219: Analytics Page (Premium Plan)

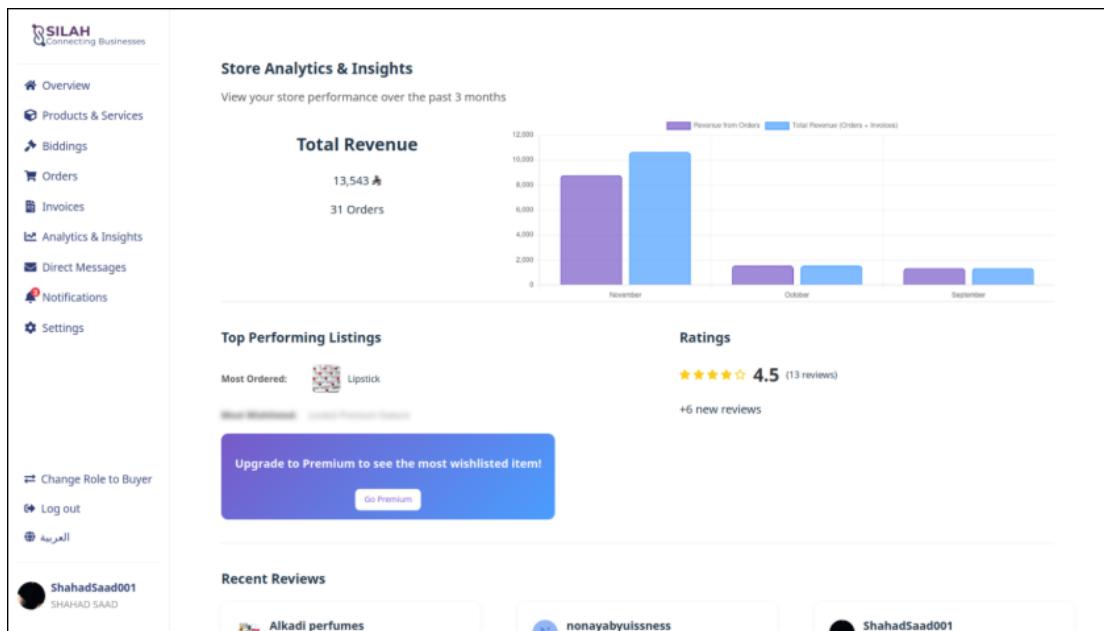


Figure 5-220: Analytics Page (Basic Plan)

5.3.3 Buyer Pages

The buyer interface is the default view for new users and offers discovery, group purchases, bidding, ordering, and direct supplier communication. A persistent top header provides global navigation.

Buyer Header & Global Navigation

The header includes category browsing, search, cart, notifications, and profile menu. Unread notifications show a red badge; the dropdown displays “No new notifications” when empty.



Figure 5-221: Buyer Header

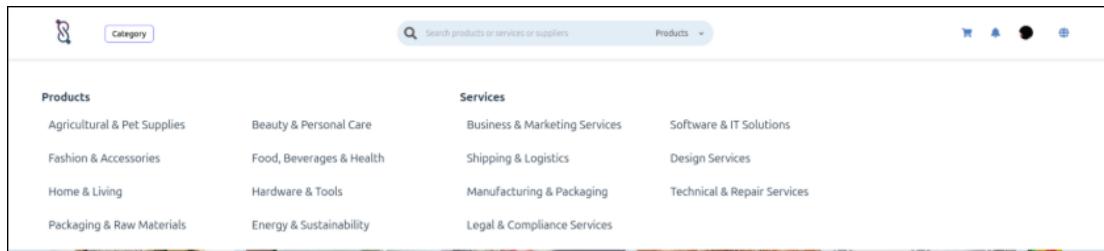


Figure 5-222: Buyer Header (Opened Category Megamenu)

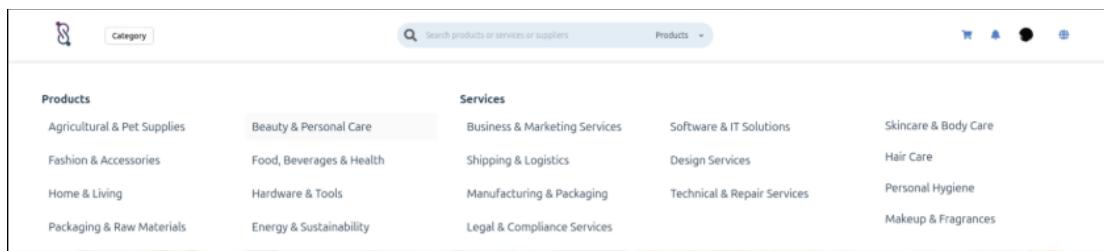


Figure 5-223: Buyer Header (Opened Category Megamenu + Hovered on Main Menu List Item)

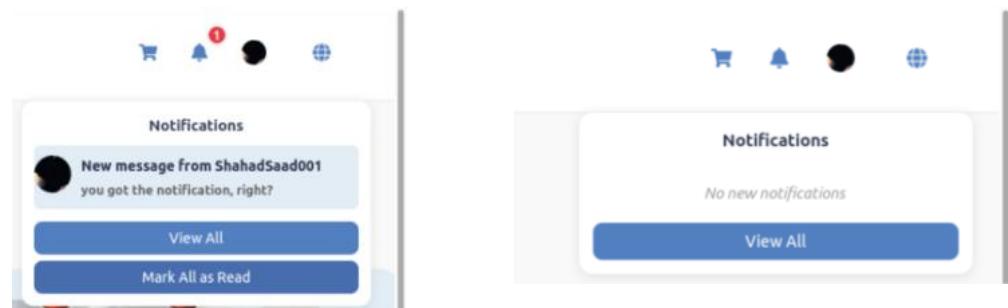


Figure 5-224: Buyer Header (Opened Notifications Menu)

Figure 5-225: Buyer Header (Opened Notifications Menu + No Unread Notification)

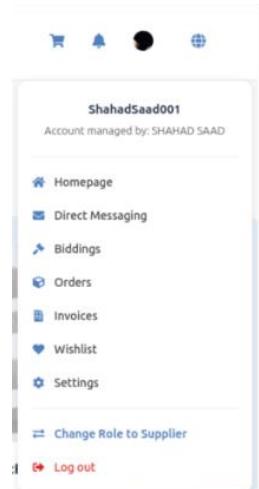


Figure 5-226: Buyer Header (Opened Profile Menu)

The Homepage

Two horizontal carousels highlight featured products and services. The AI-powered search bar at the bottom accepts natural-language queries.

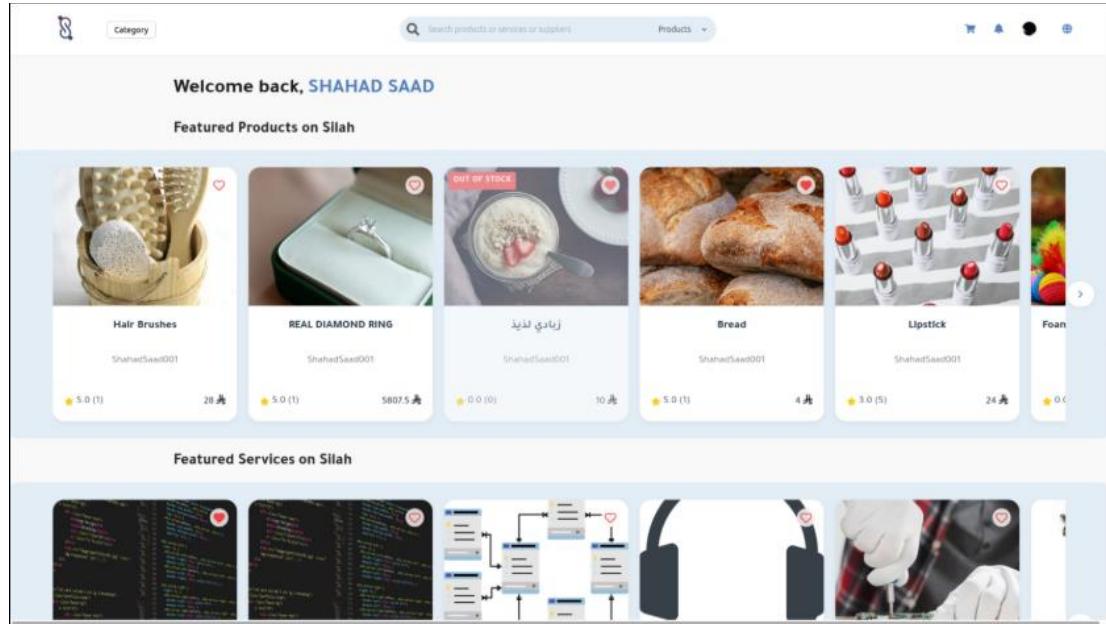


Figure 5-227: Homepage (1/2)

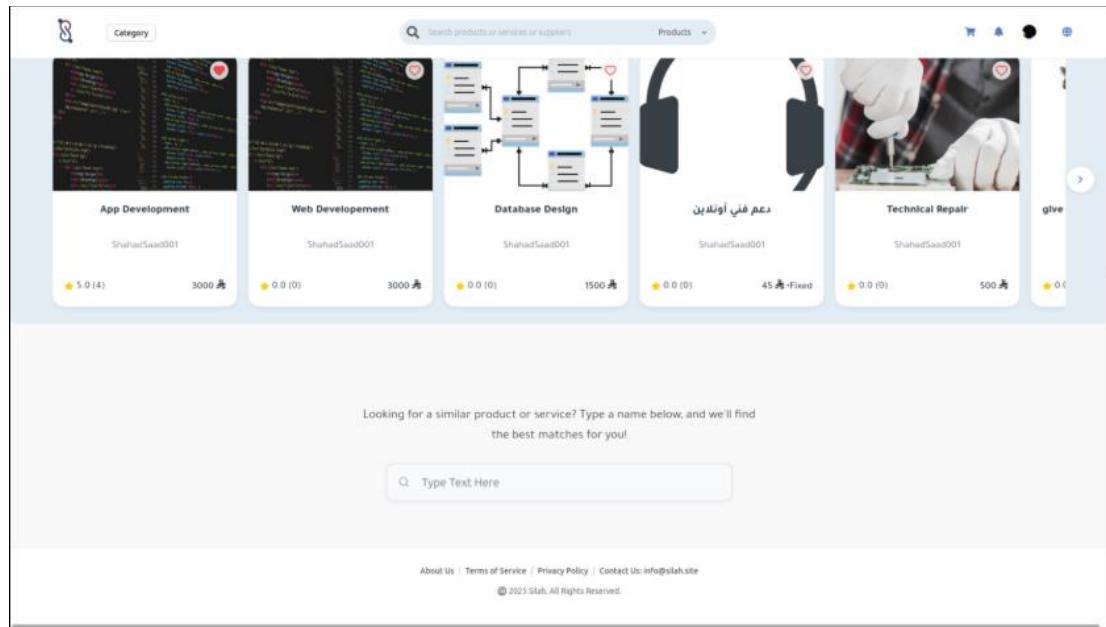


Figure 5-228: Homepage (2/2)

Alternatives Page

The Alternatives page is a central AI-powered feature reachable from multiple points across the buyer journey:

- Directly from the homepage search bar.
- From a Product Details page when the product is out of stock. The usual “Add to Cart” button is replaced by a “Find Alternatives” button.
- From the Cart when an out-of-stock product is detected, displaying a “Find similar products” link.
- From the Wishlist, where every card includes a dedicated “Find Alternatives” button.

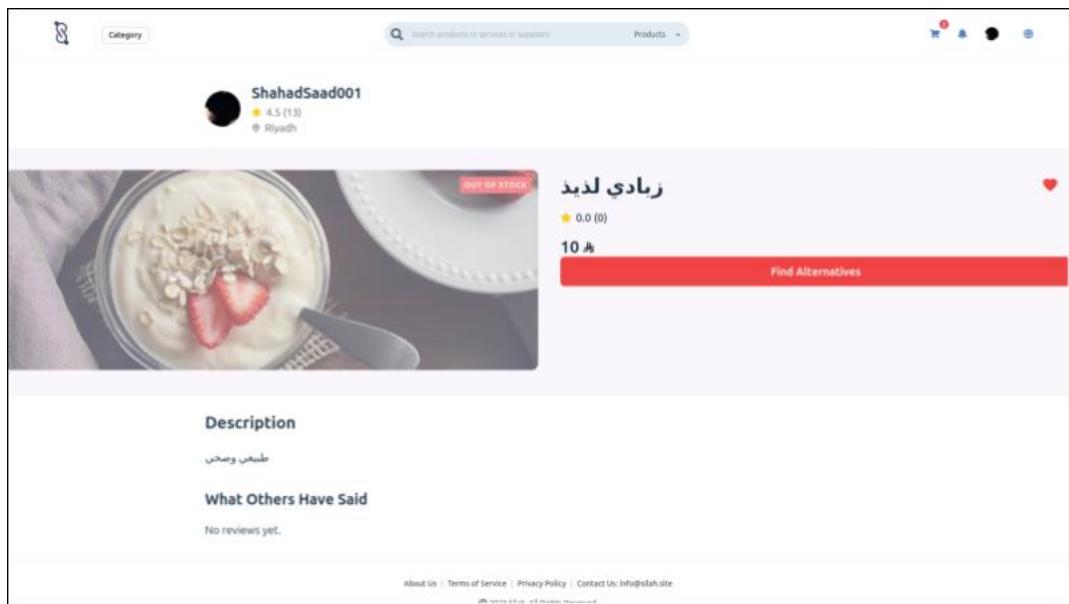


Figure 5-229: Buyer's Product Details Page (Out of Stock)

The results page itself is implemented exactly as designed in Figma: a ranked, card-based list showing the product/service image, name, supplier, price and rating. Buyers can immediately wishlist any result or proceed to its full details.

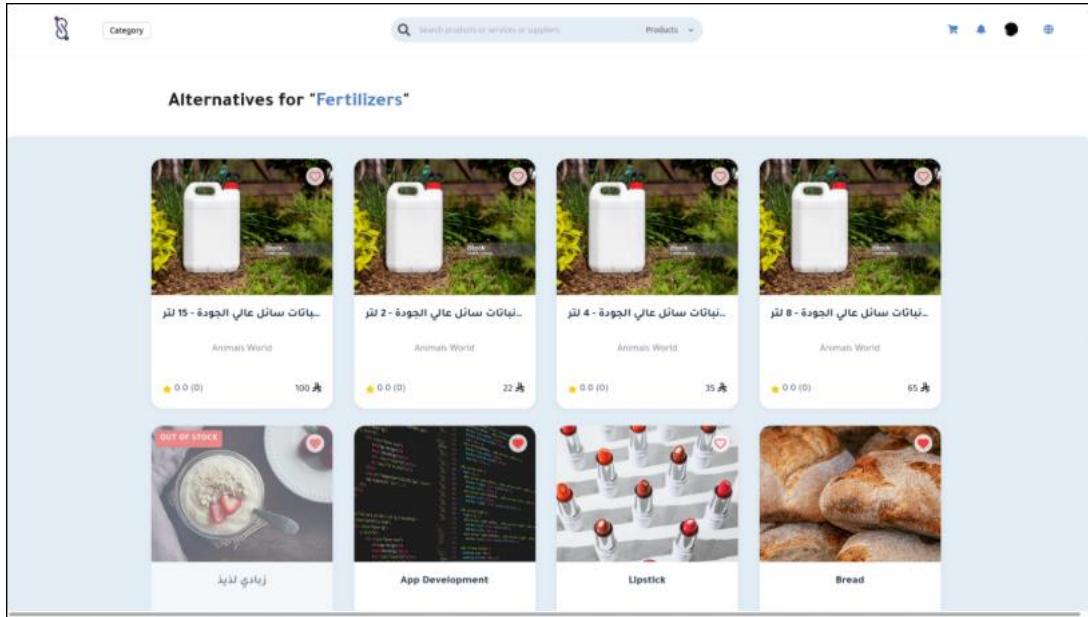


Figure 5-230: Alternatives Page

Product Details & Join a Group Purchase

The Product Details page displays all essential information (images, description, price, stock, supplier details, reviews, etc.) in a clean layout. The group-purchase functionality introduces several states that enhance the buyer experience:

- Active group purchase:** When a group purchase is already in progress, the page shows the current number of participants, required total, time remaining, and discounted price, together with a “Join Group Purchase” button and quantity input field.
- No active group purchase but allowed:** If the supplier enabled group purchases for the product but none is currently active, the buyer sees a “Start a Group Purchase” option that initiates the process.
- Group purchases disabled by supplier:** When the supplier has turned off group purchasing for the product, no related messages or buttons are shown.

All states use consistent styling, clear messaging, and the same visual hierarchy, ensuring a seamless experience regardless of the product’s group-purchase configuration.

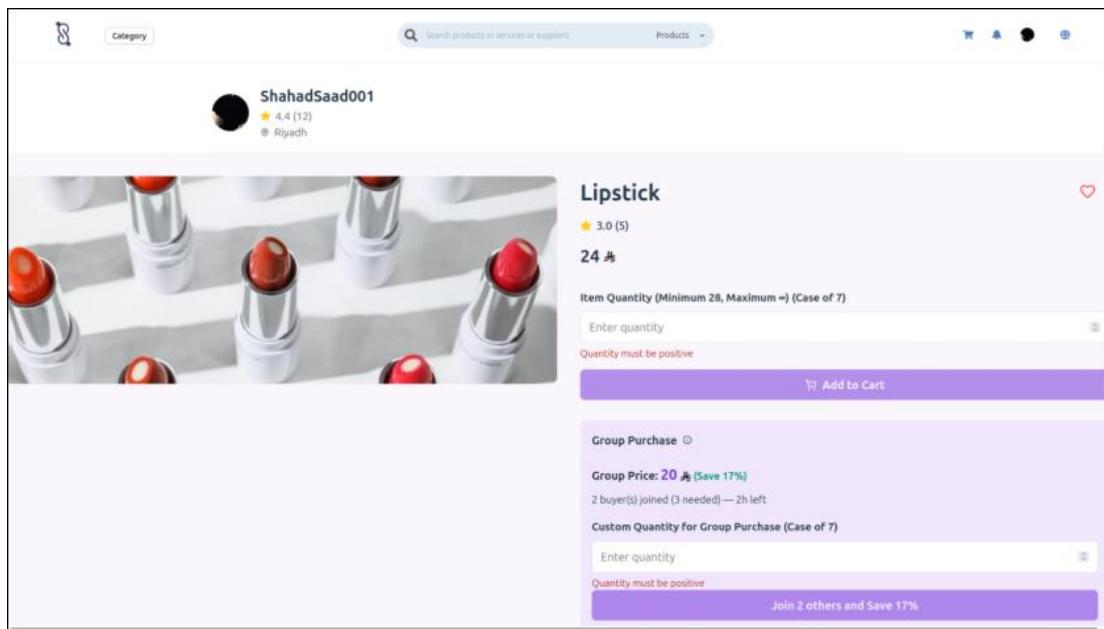


Figure 5-231: Buyer's Product Details Page (1/2) + Join Group Purchase Section

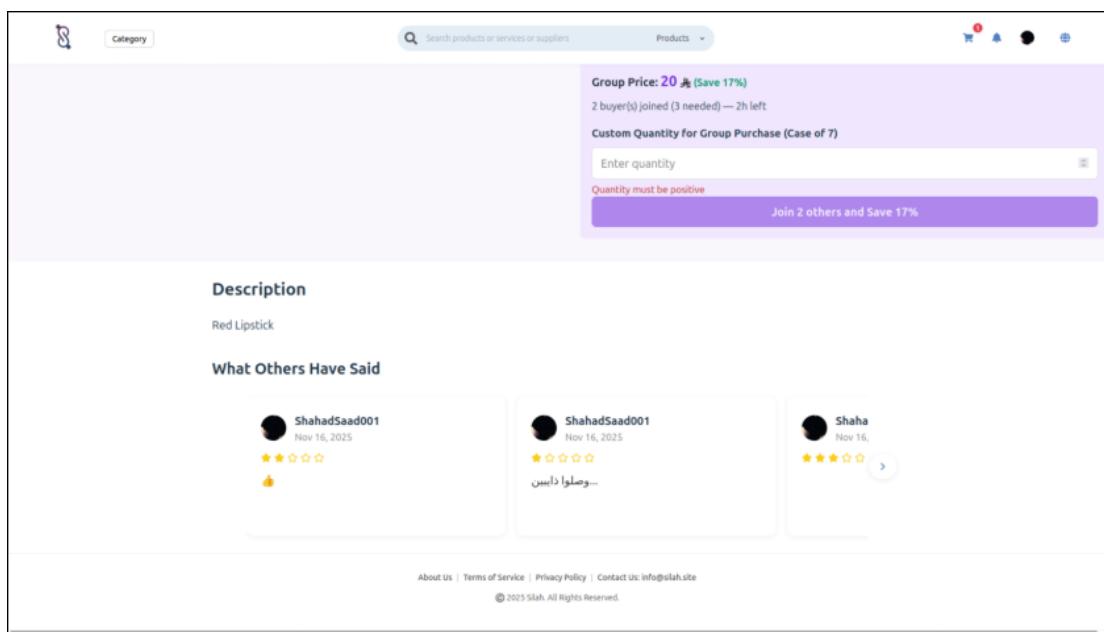


Figure 5-232: Buyer's Product Details Page (2/2)

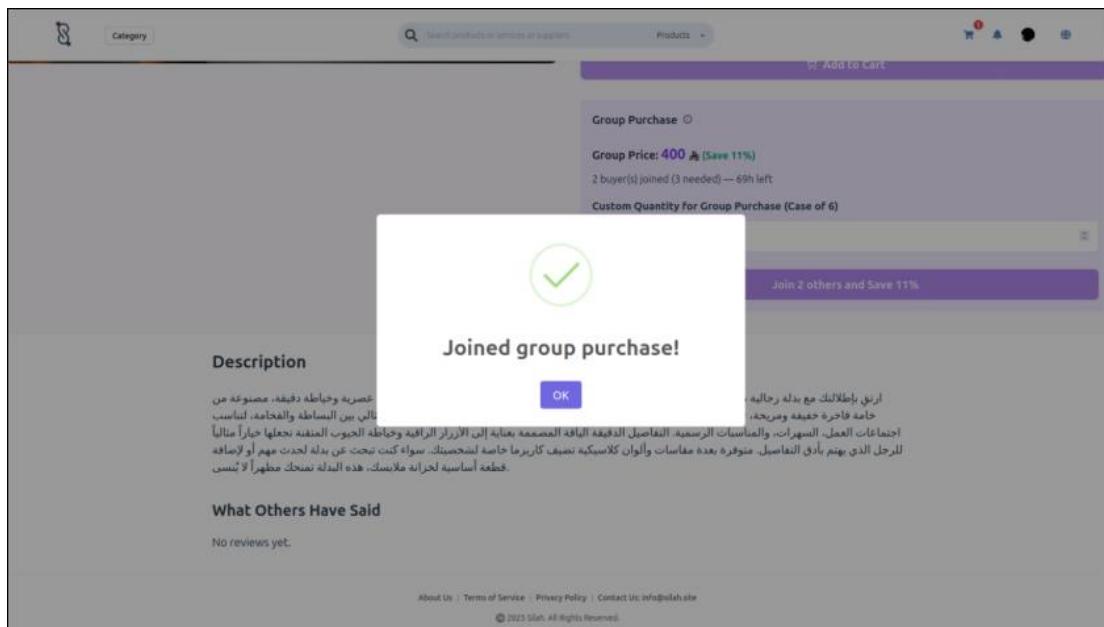


Figure 5-233: Joined Group Purchase Dialog

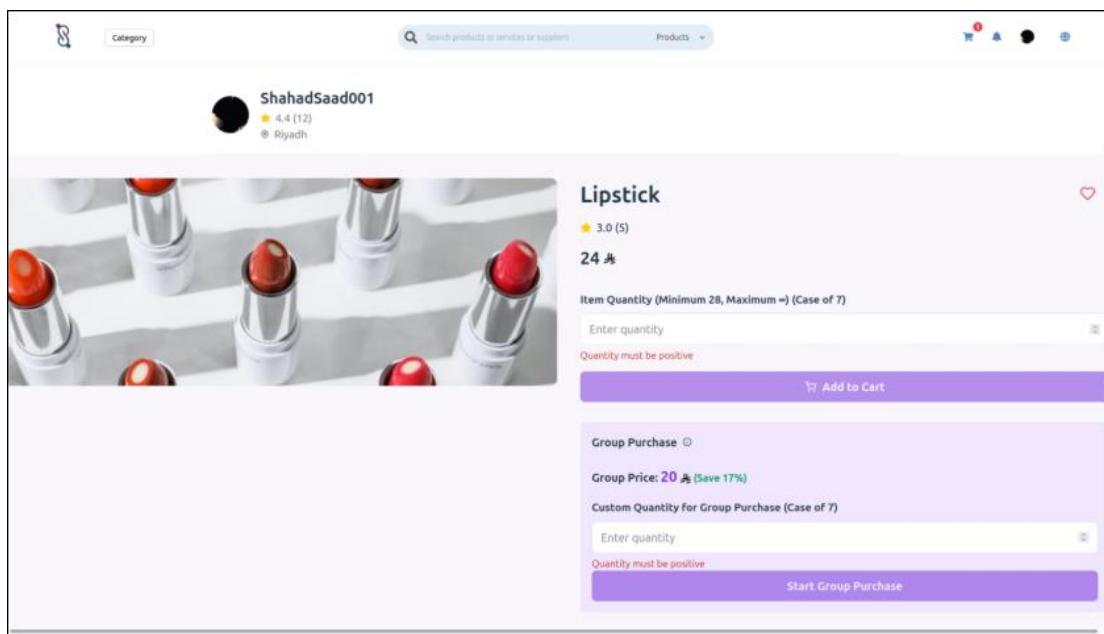


Figure 5-234: Start Group Purchase Section

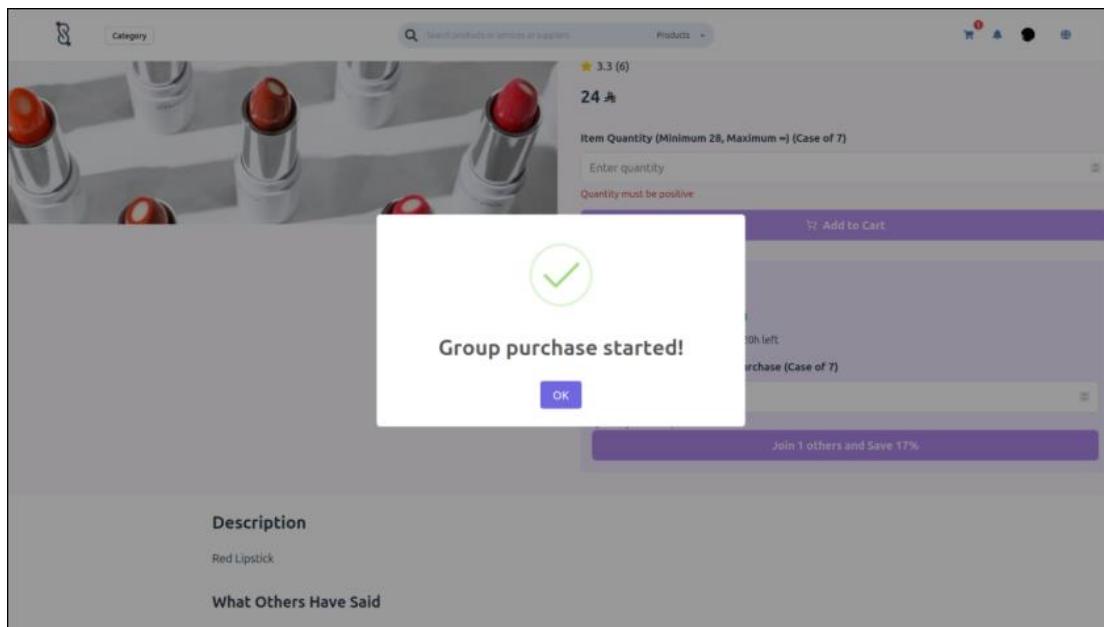


Figure 5-235: Started Group Purchase Dialog

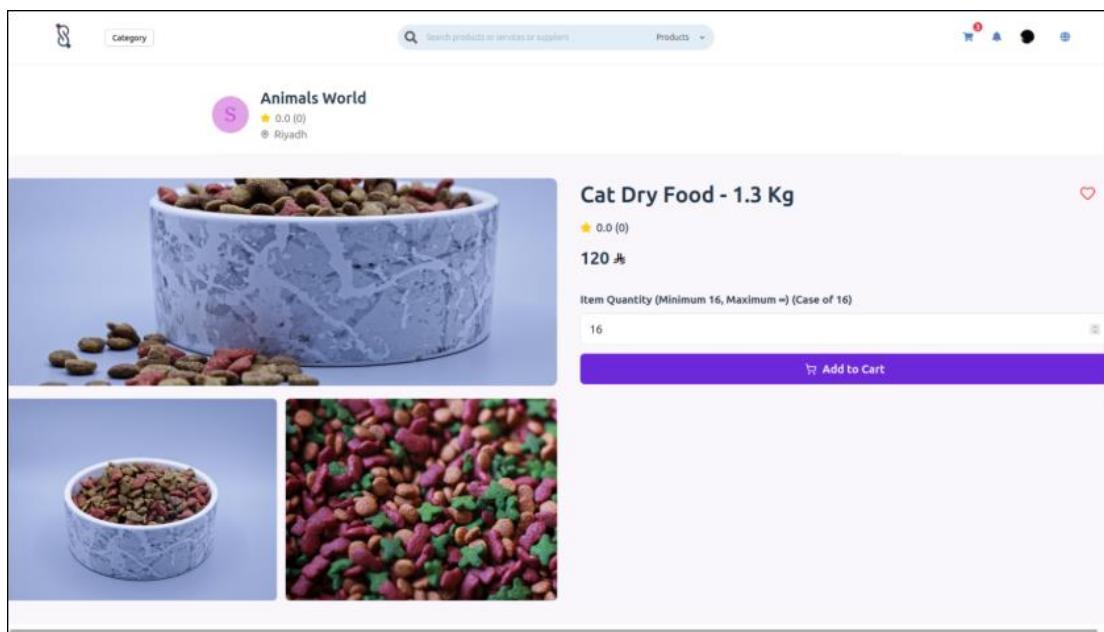


Figure 5-236: Buyer's Product Details Page (No Group Purchase)

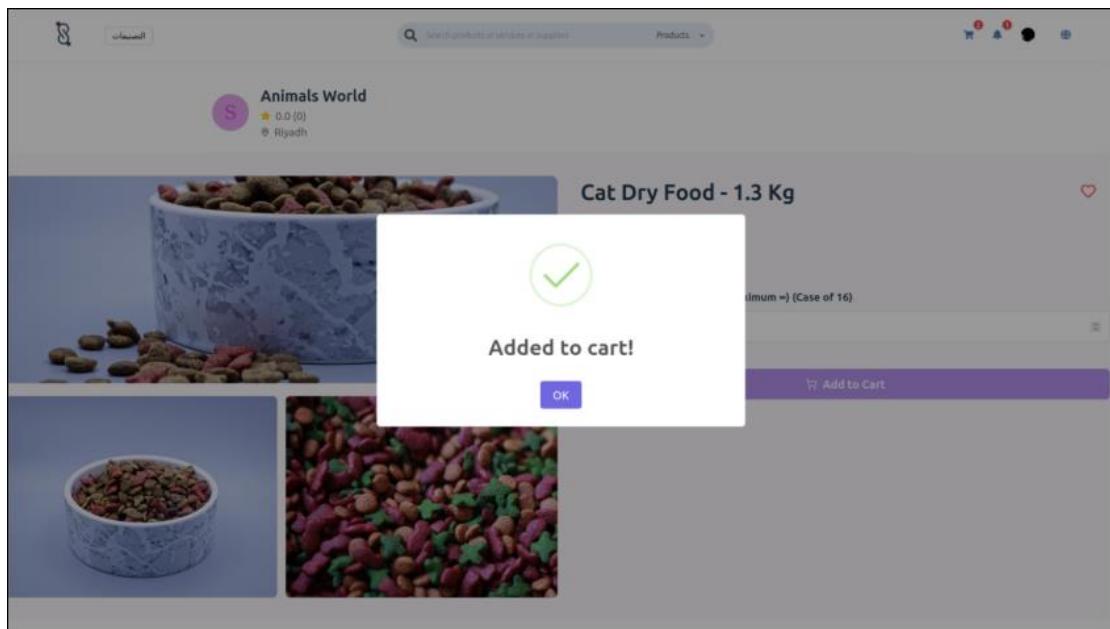


Figure 5-237: Added Item to Cart Dialog

Cart

Items are grouped by supplier with delivery fees and totals. Out-of-stock items include a direct “Find similar products” link.

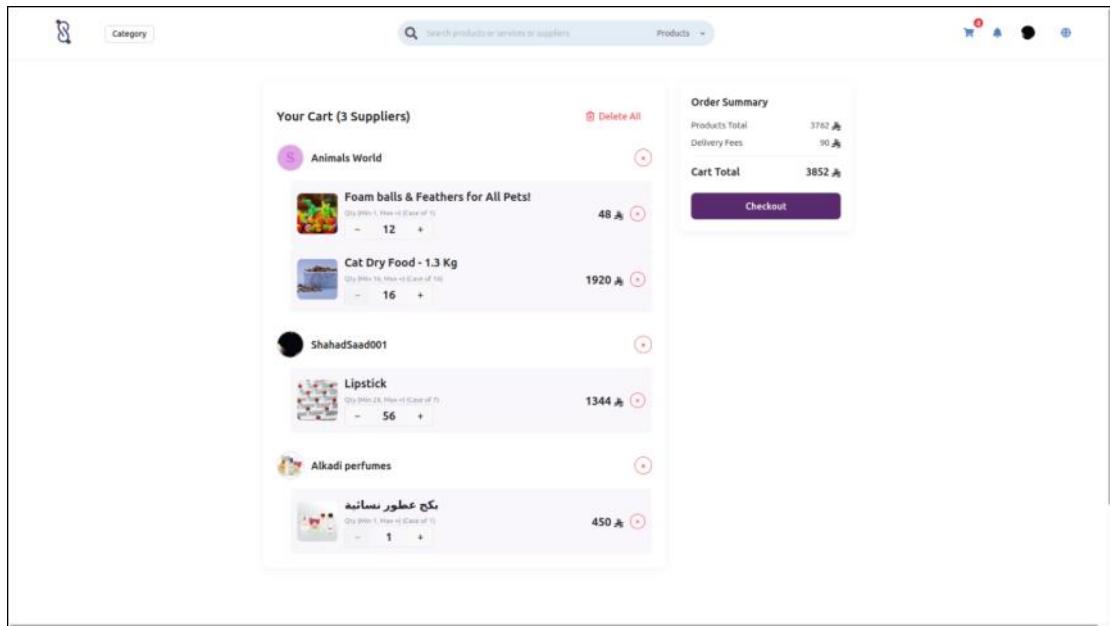


Figure 5-238: Cart Page

Payment Flow (Tap Payments Integration & Test Mode)

All payment-related actions (checkout from cart, paying an invoice, or saving a new card) are processed through Tap Payments in test mode. Upon initiating payment, the buyer is redirected to Tap’s ACS (Access Control Server) emulator, which allows

manual selection of the 3-D Secure authentication outcome for testing purposes. After the emulator step, Tap redirects the user to Silah's unified Payment Callback page, which handles three successful scenarios with distinct, clear messages:

1. Successful order payment (from cart checkout)
2. Successful invoice payment
3. Successful card addition (for future use)

This page displays a clear success message followed by an automatic “Redirecting in 3... 2... 1...” countdown. Upon completion of the countdown, the buyer is seamlessly returned to the appropriate section (orders page, invoices page, or settings respectively).

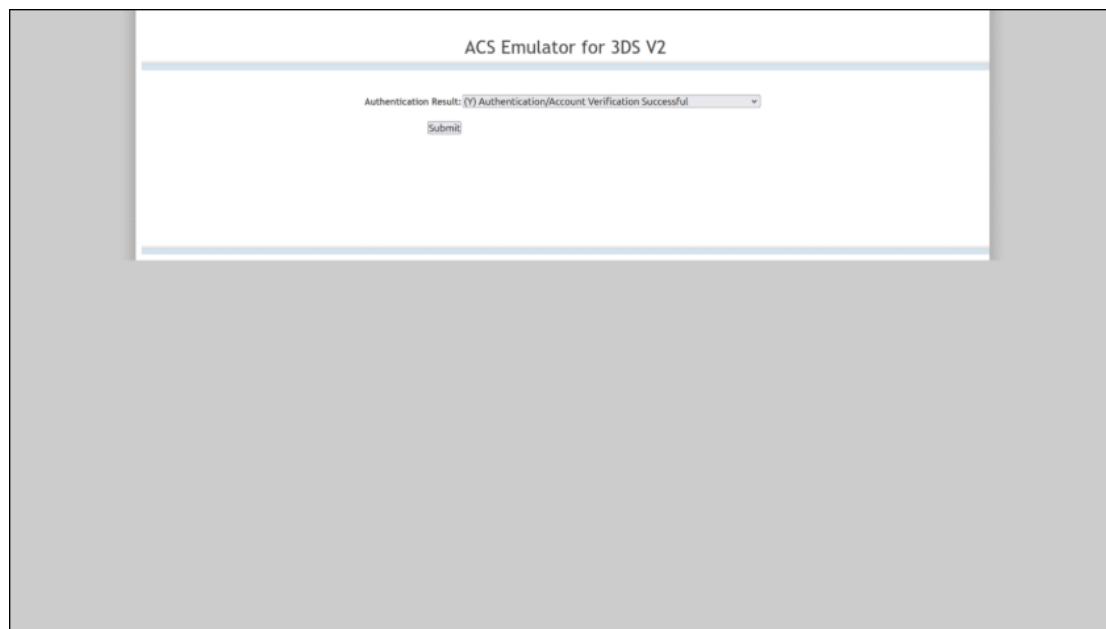


Figure 5-239: Tap ACS Emulator

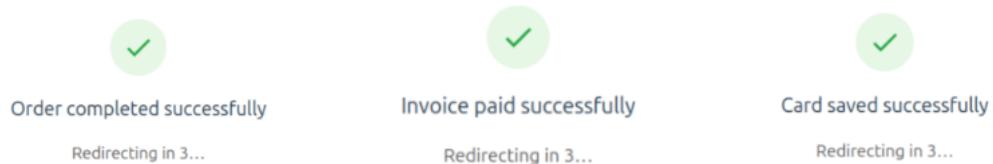


Figure 5-240: Payment Callback
(Cart Paid + Order Made)

Figure 5-241: Payment Callback
(Invoice Paid)

Figure 5-242: Payment Callback
(Card Saved)

Service Details

The layout mirrors the Product Details page, highlighting availability settings and price negotiability, with a “Request Service” button opening direct messaging.

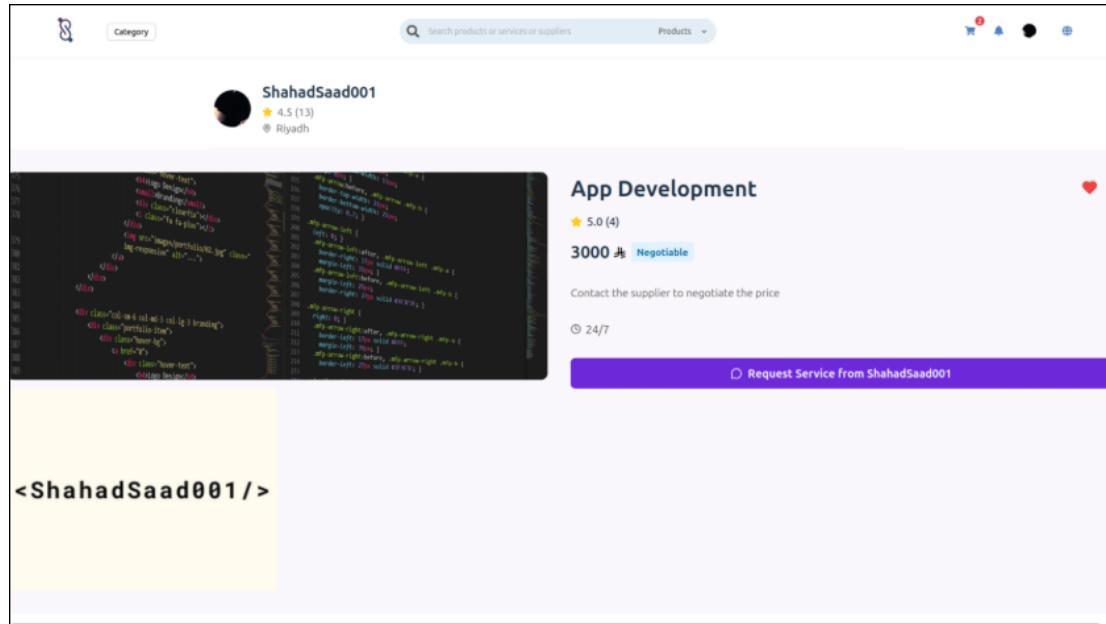


Figure 5-243: Buyer's Service Details Page (1/2)

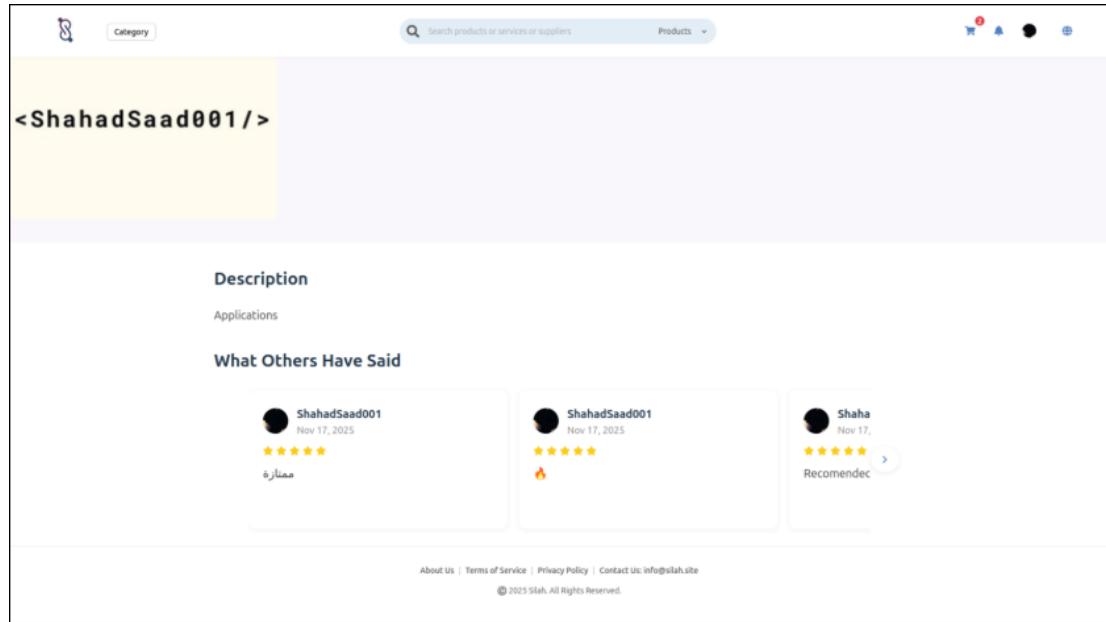


Figure 5-244: Buyer's Service Details Page (2/2)

Supplier Storefront

Filters were moved from the left sidebar to the top, allowing wider card grids. Dialogs gracefully handle closed or inactive stores.

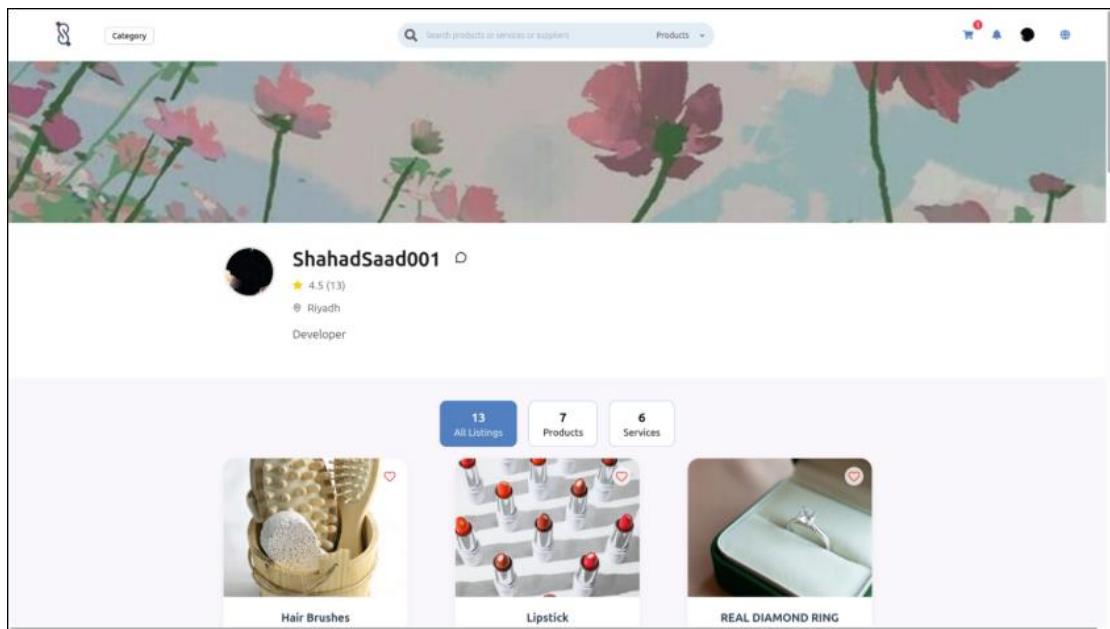


Figure 5-245: Storefront Page (1/2)

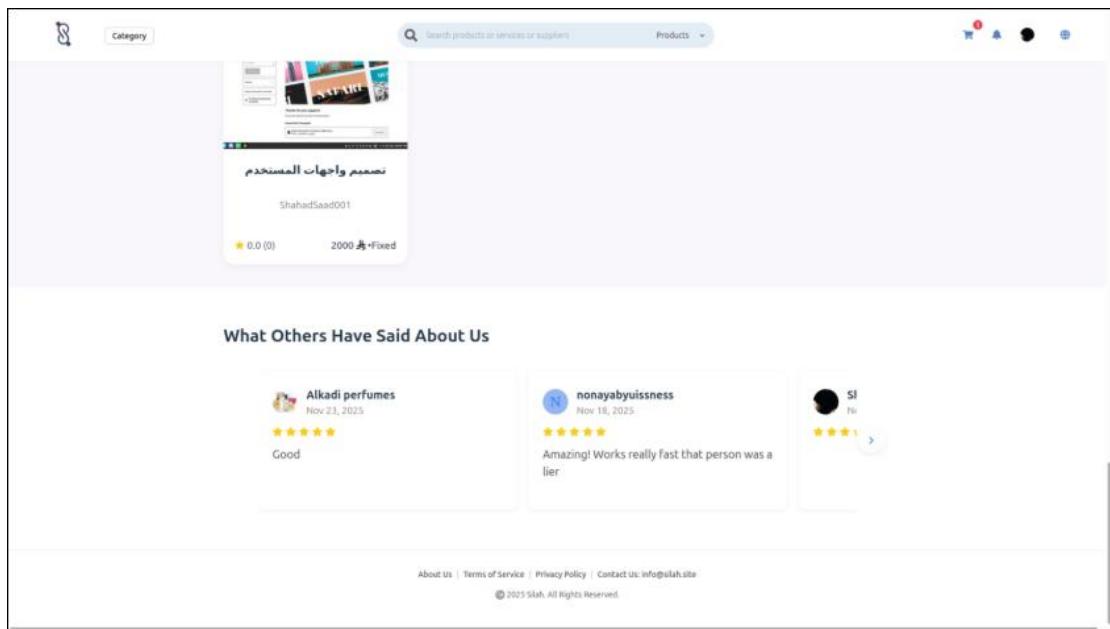


Figure 5-246: Storefront Page (2/2)

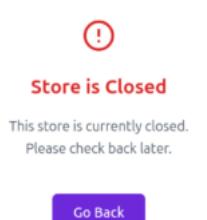


Figure 5-247: Storefront Page (Store Closed)

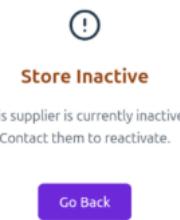


Figure 5-248: Storefront Page (Inactive Supplier)

Search Results & Browsing Category

Buyers initiate a search by first selecting the desired target from a dropdown menu on the search bar (Products, Services, or Suppliers), ensuring precise and unambiguous results even with minor typos. Once submitted, the page instantly presents a clean, responsive card grid tailored to the chosen type. Product results include images, prices, ratings, supplier name, and a left-side price-range filter. Service results use the same card layout as products but without the price filter. Supplier results show store profile cards with description, rating, and onclick it directs the buyer to the storefront page. All variants share the same modern styling for intuitive discovery experience.

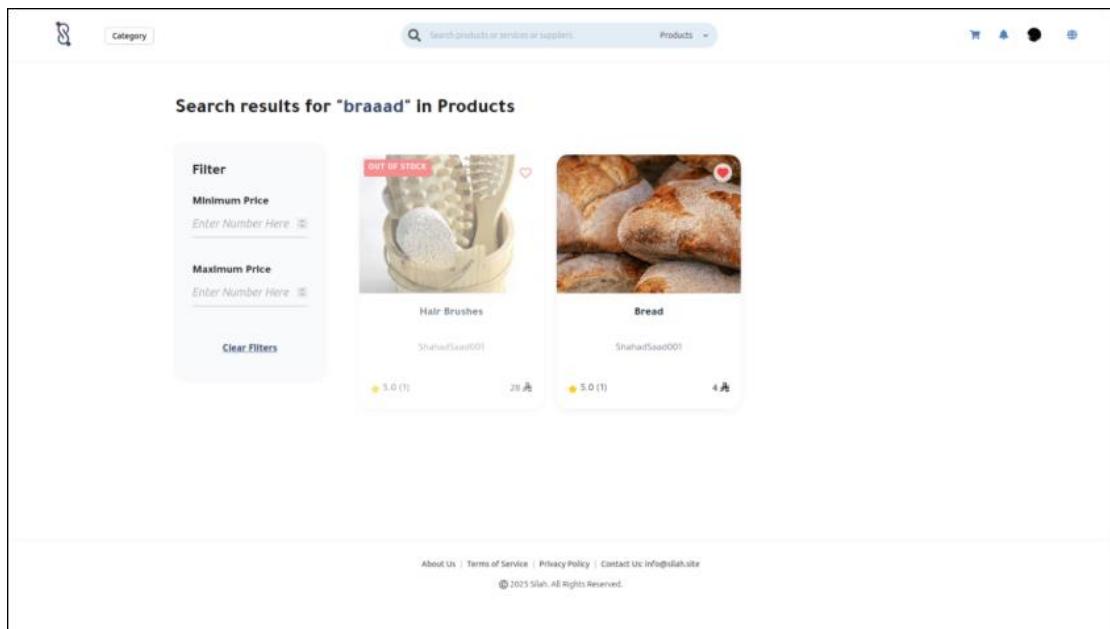


Figure 5-249: Search Results Page (Products)

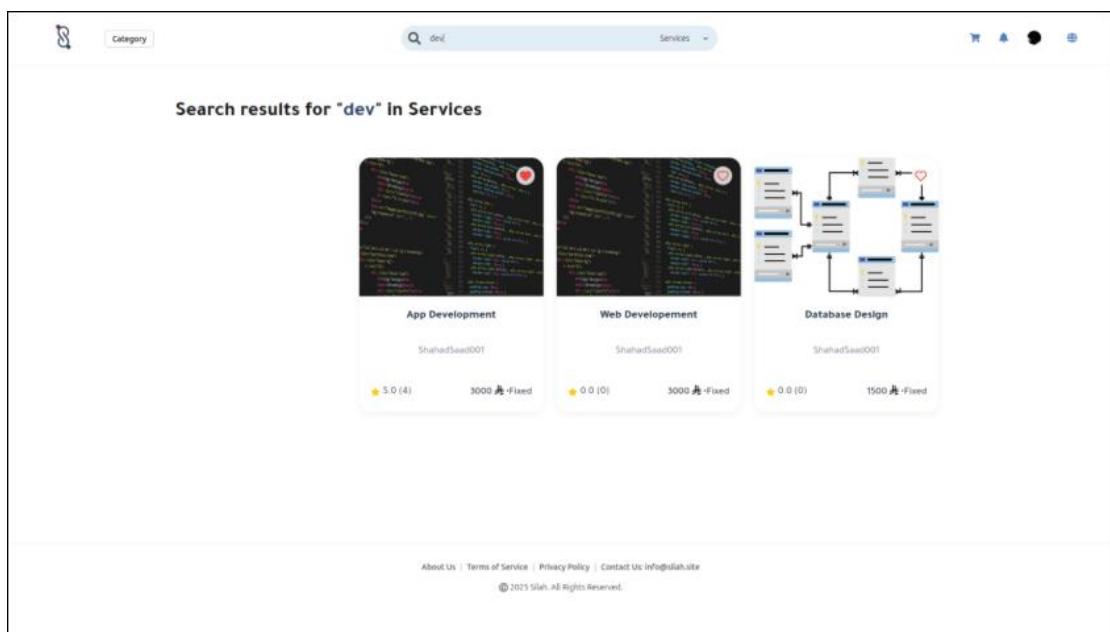


Figure 5-250: Search Results Page (Services)

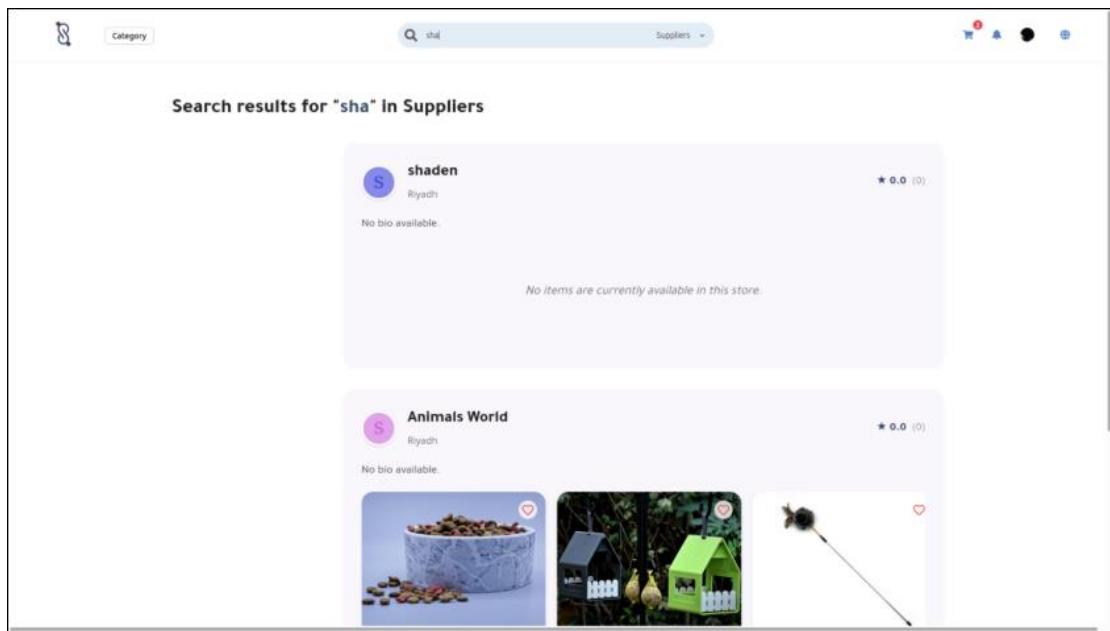


Figure 5-251: Search Results Page (Suppliers)

The category browsing experience offers two hierarchical levels.

Entering a main category (e.g., “Agriculture & Pet Supplies”) displays all items within that category and its subcategories, with a left sidebar that lists the main category followed by its specific subcategories (e.g., Animal Feed, Fertilizers, Pet Accessories & Toys, Pet Food & Treats). Selecting a subcategory refines the view to only items in that subcategory, while the page title and breadcrumb clearly indicate the path (e.g., “Agriculture & Pet Supplies > Fertilizers”). The same card-based grid and filtering options are used throughout, ensuring consistency and easy navigation.

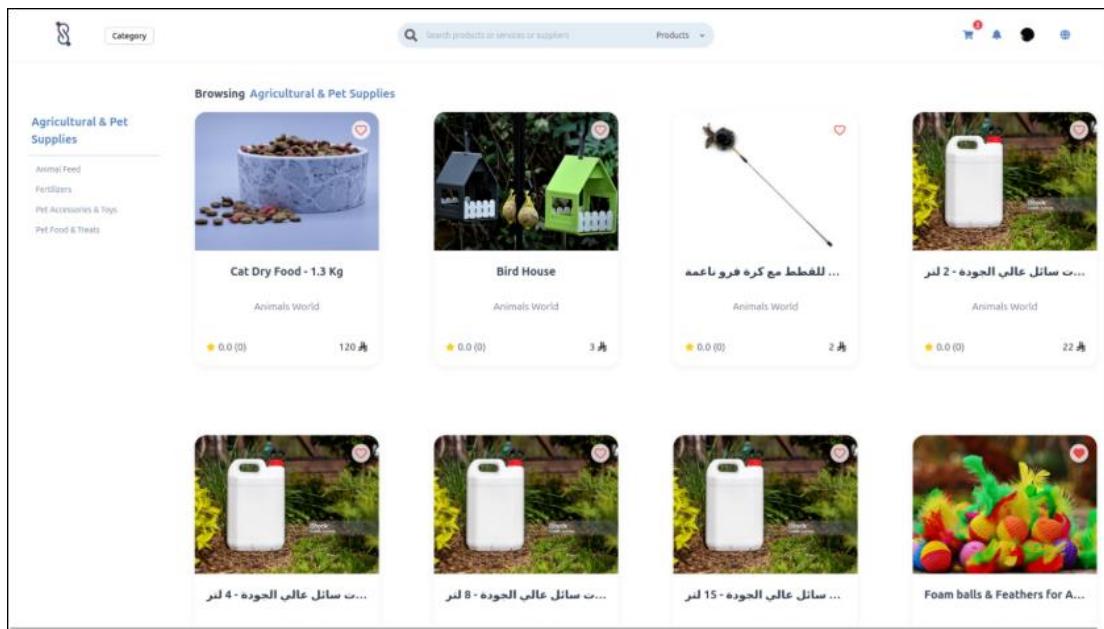


Figure 5-252: Browse by Category Page (Main Category)

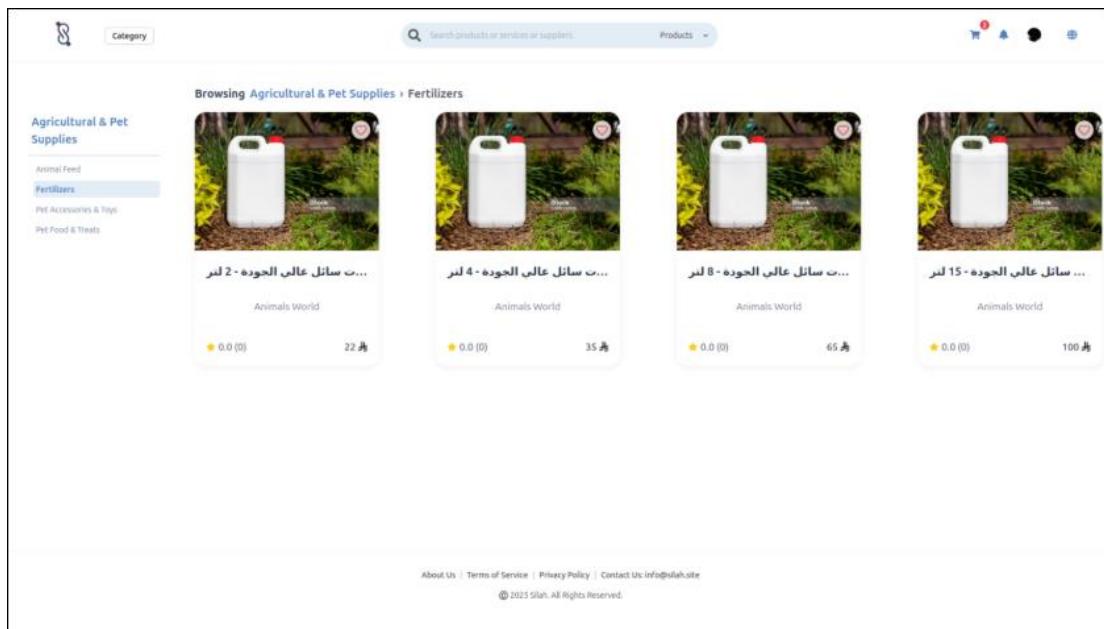


Figure 5-253: Browse by Category Page (Subcategory)

Wishlist

Displays saved items with real-time stock status and one-click alternative search.

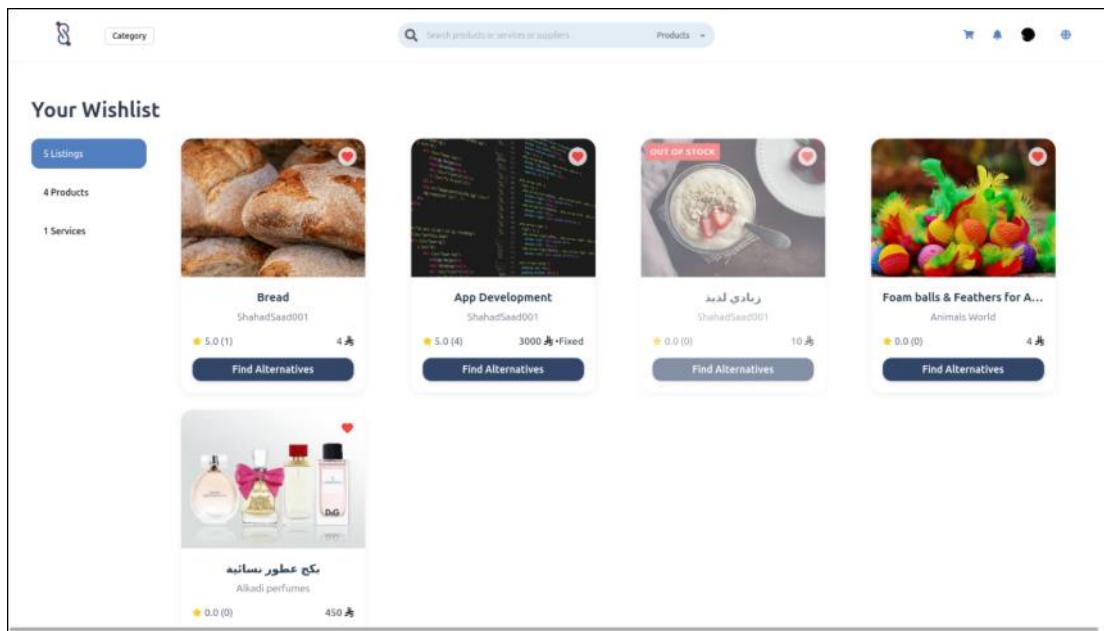


Figure 5-254: Wishlist Page

Creating a New Bid

The buyer bidding system is fully implemented as a clean, card-based interface that makes creating, tracking, and responding to bids intuitive.

From the My Bids page, buyers get an at-a-glance overview of all their published bids, each shown as a compact card listing the bid title, reference number, main activity, submission deadline, and status. A “Create New Bid” button opens the Create a New Bid page where the buyer fills in the bid name, main and secondary activities, submission deadline, and expected response timeframe before publishing with a single click.

The screenshot shows a web-based bidding platform interface. At the top, there are navigation tabs for 'Category' and 'Products'. A search bar is located at the top center. On the right side, there are several small icons. The main content area is titled 'My Bids' and displays three separate bid cards:

- Bid 1:** Published on: 23 Nov 2025. Main Activity: محتاج ببط ابيض مساحة 24 متر مربع ف مساحة 100.000 ف مساحة 24 متر مربع. Reference #: 1995594148. Submission Deadline: 24 Nov 2025. Buttons: View Details, Offers are locked until the deadline.
- Bid 2:** Published on: 18 Nov 2025. Main Activity: Is it dup?. Reference #: 4915531757. Submission Deadline: 18 Nov 2025. Buttons: View Details, View Offers.
- Bid 3:** Published on: 18 Nov 2025. Main Activity: I will decline you, sorry!. Reference #: 4517944305. Buttons: View Details.

Figure 5-255: Buyer's Biddings Page

The screenshot shows the 'Create a New Bid' page. At the top, there are navigation tabs for 'Category' and 'Products'. A search bar is located at the top center. On the right side, there are several small icons. The main content area is titled 'Create a New Bid' and contains the following form fields:

- Bid Name:** Supply and installation of Smart Street Lighting Systems. A note below says "0/100".
- Main Activity:** Electrical Works & Lighting - Installation and Maintenance of Electrical Systems. A note below says "0/100".
- Submission Deadline:** A date input field with placeholder "mm / dd / yyyy" and a calendar icon.
- Response Deadline for Offers:** A dropdown menu showing "2 weeks after submission".
- Publish Bid:** A purple button at the bottom of the form.

Figure 5-256: Create a New Bid Page

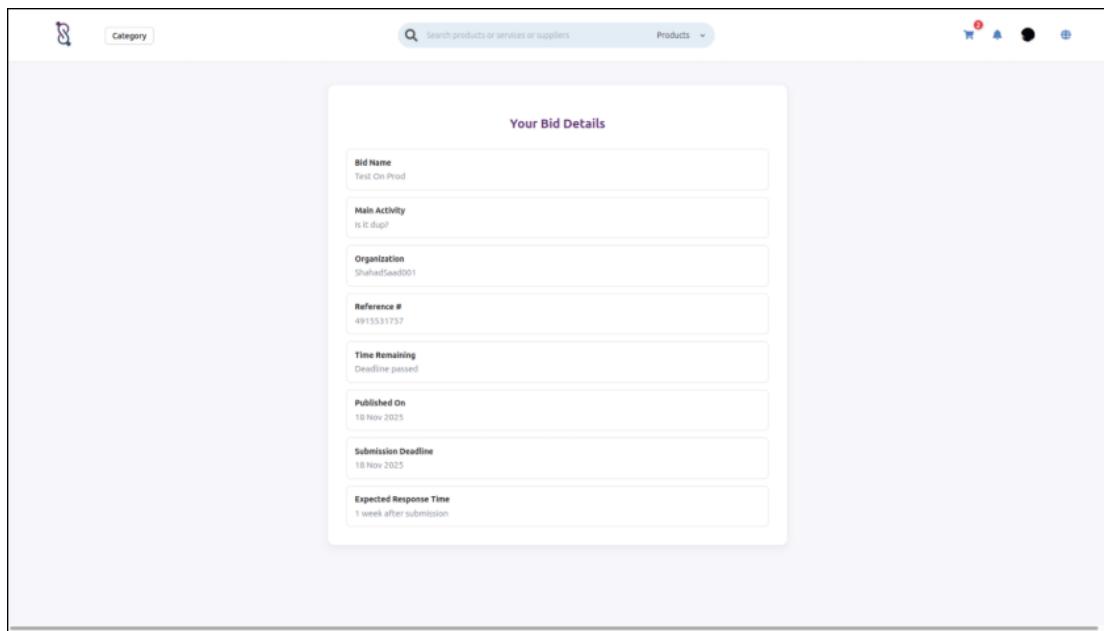


Figure 5-257: Buyer's Bid Details Page

Accepting an Offer

Once the submission deadline passes, every bid card displays a “View Offers” button. Clicking it opens the Received Offers page, where each supplier’s offer appears as a clear card showing the supplier’s name, proposed amount, expected completion date, and action buttons. Buyers can immediately Decline or Accept an offer or click “View Details” to open the Offer Details page that presents the complete technical and commercial proposal submitted by the supplier. After an offer is accepted or declined, its card instantly updates with a clear visual status while remaining in the list for reference.

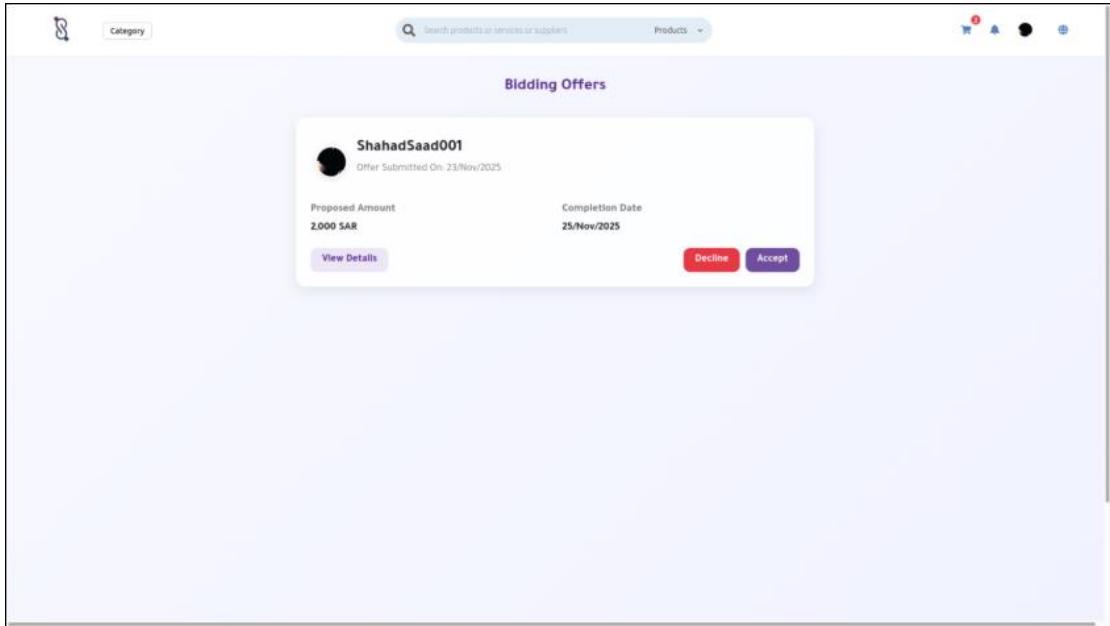


Figure 5-258: Received Offers Page

A screenshot of the 'Received Offers Page' showing a declined offer. The offer is from 'ShahadSaad001' submitted on 18/Nov/2025. The proposed amount is 1,000,000 SAR, and the completion date is 30/Nov/2025. The status is 'DECLINED'. A note below the card states 'This offer has already been reviewed.' There is a 'View Details' button.

Figure 5-259: Received Offers Page (Declined Offer)

A screenshot of the 'Received Offers Page' showing an accepted offer. The offer is from 'ShahadSaad001' submitted on 18/Nov/2025. The proposed amount is 2 SAR, and the completion date is 19/Nov/2025. The status is 'ACCEPTED'. A note below the card states 'This offer has already been reviewed.' There is a 'View Details' button.

Figure 5-260: Received Offers Page (Accepted Offer)

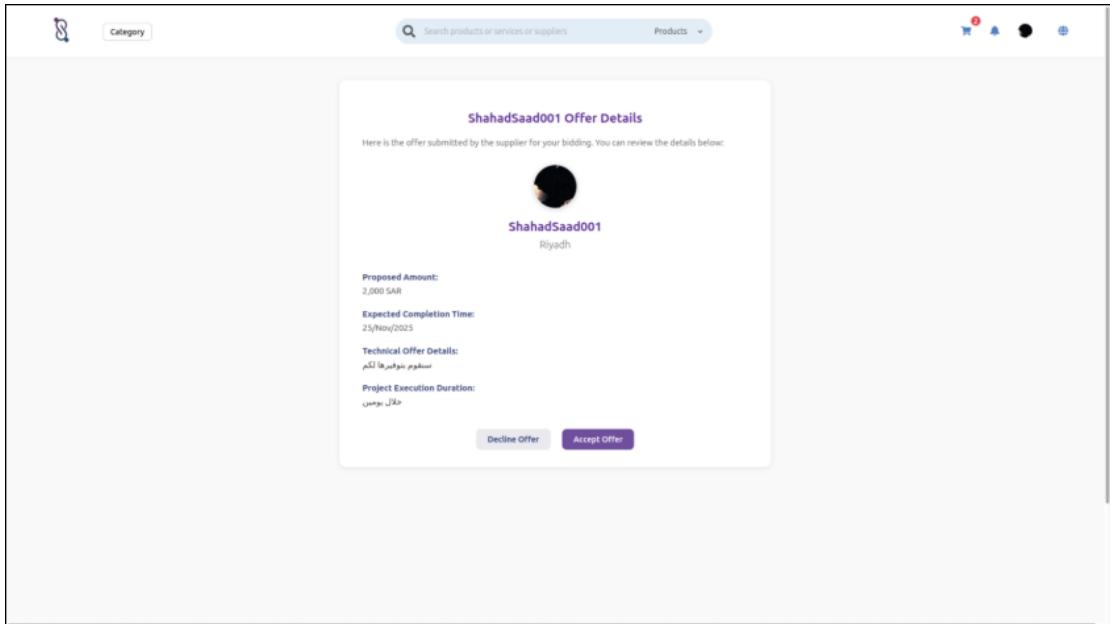


Figure 5-261: Offer Details Page

Buyer Settings

The Buyer Settings page adopts the same polished, tabbed layout used for Supplier Settings, with horizontal tabs (General, Account, Notifications, Payment Method, Support) and all content presented in clean, well-spaced card containers. The General, Account, Notifications, and Support tabs mirror their supplier's equivalents in design and behavior. The buyer-specific Payment Method tab replaces the store settings tab and offers two states. When a card is already saved, it displays the cardholder's name, last four digits, and expiry date with clear options to remove or replace it. When no card is on file, a friendly message reads “You don't have a saved card yet. Add one to make your payments easier” and an inline form appears for entering card number, expiry month/year, and CVV. Saving the card triggers the Tap Payments test-mode flow described earlier.

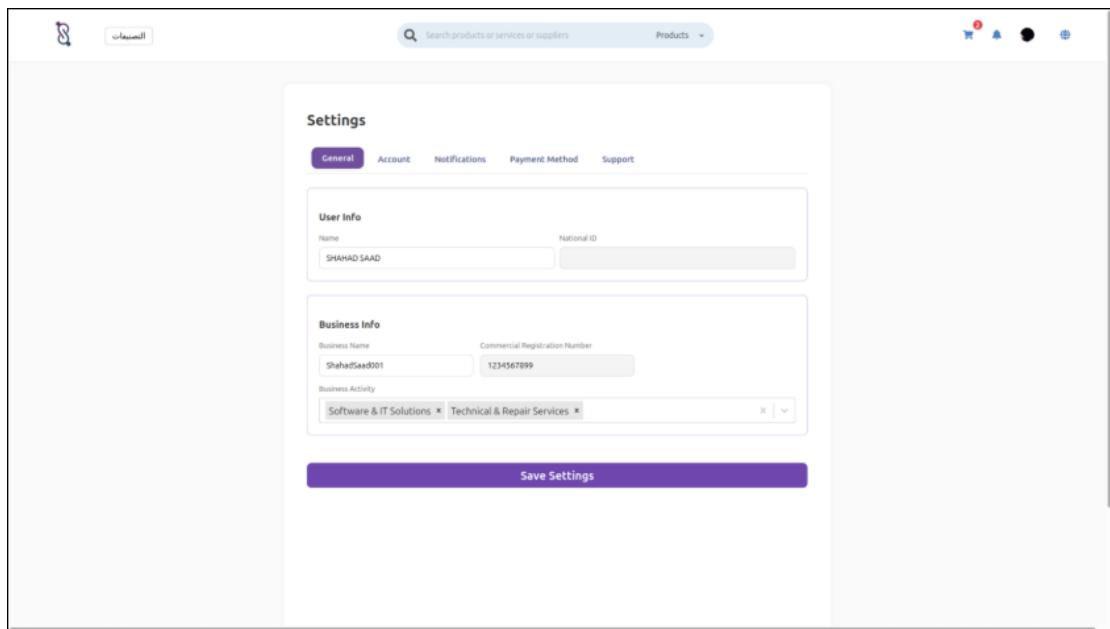


Figure 5-262: Buyer's Settings Page (General)

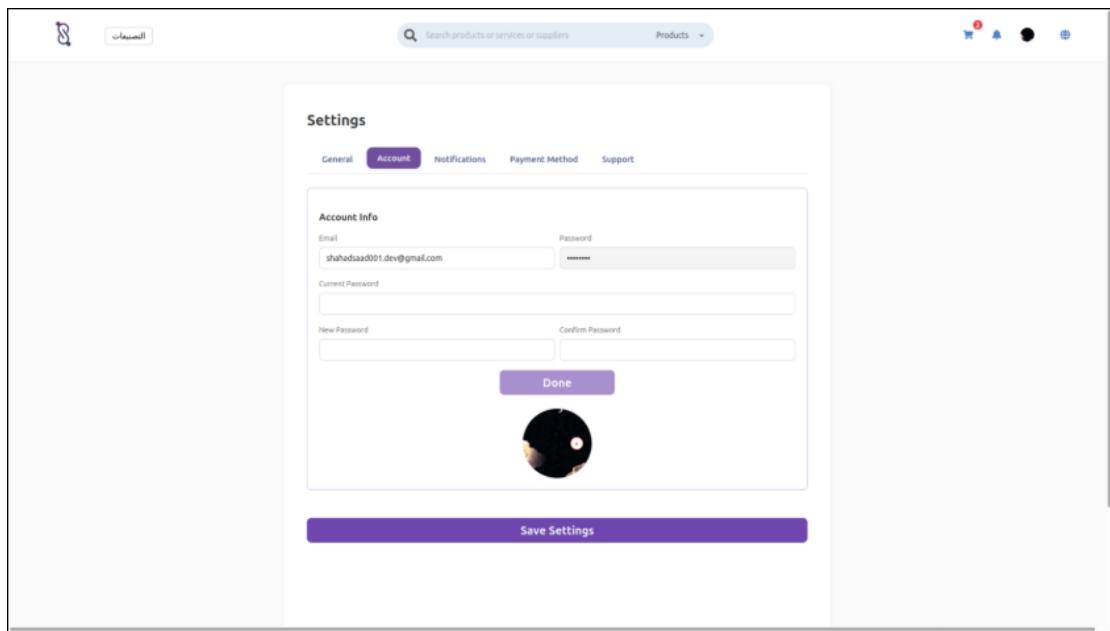


Figure 5-263: Buyer's Settings Page (Account)

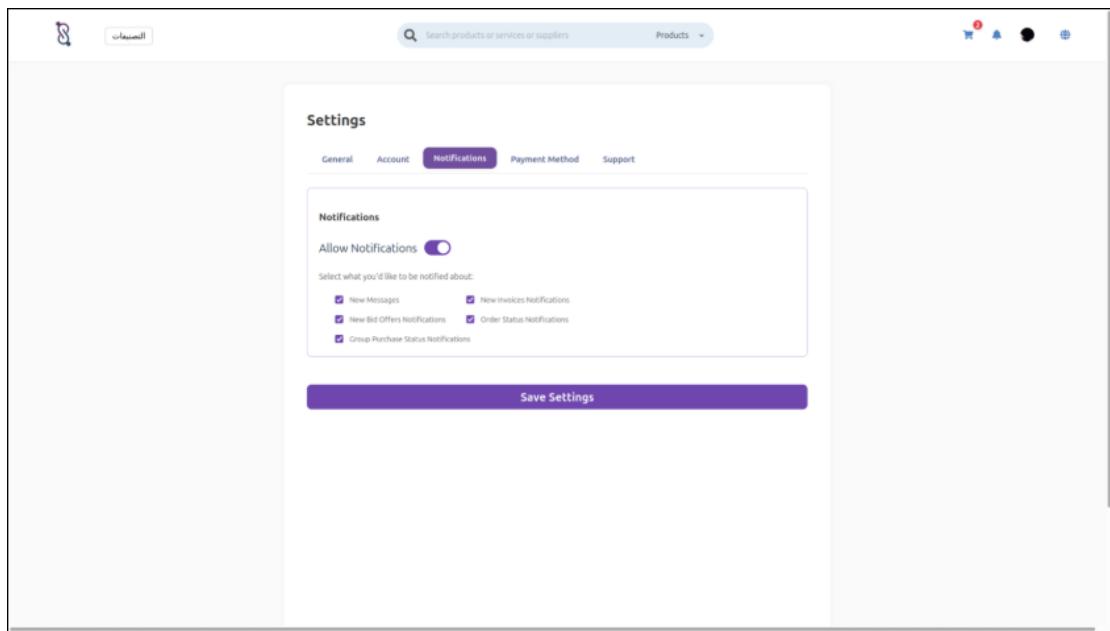


Figure 5-264: Buyer's Settings Page (Notifications)

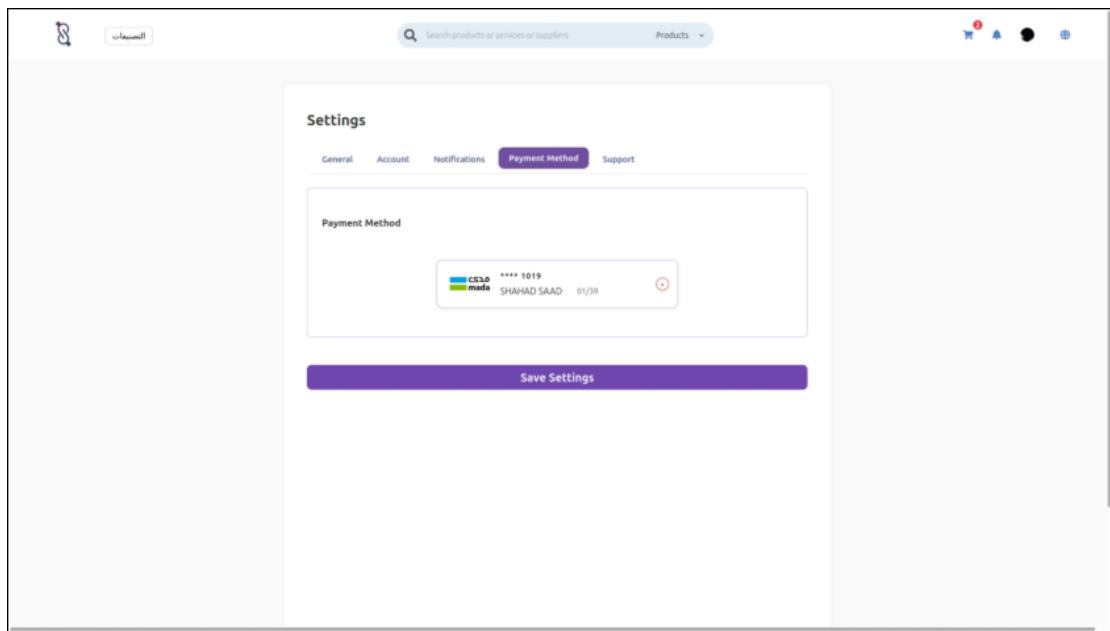


Figure 5-265: Buyer's Settings Page (Payment Method + Stored Card)

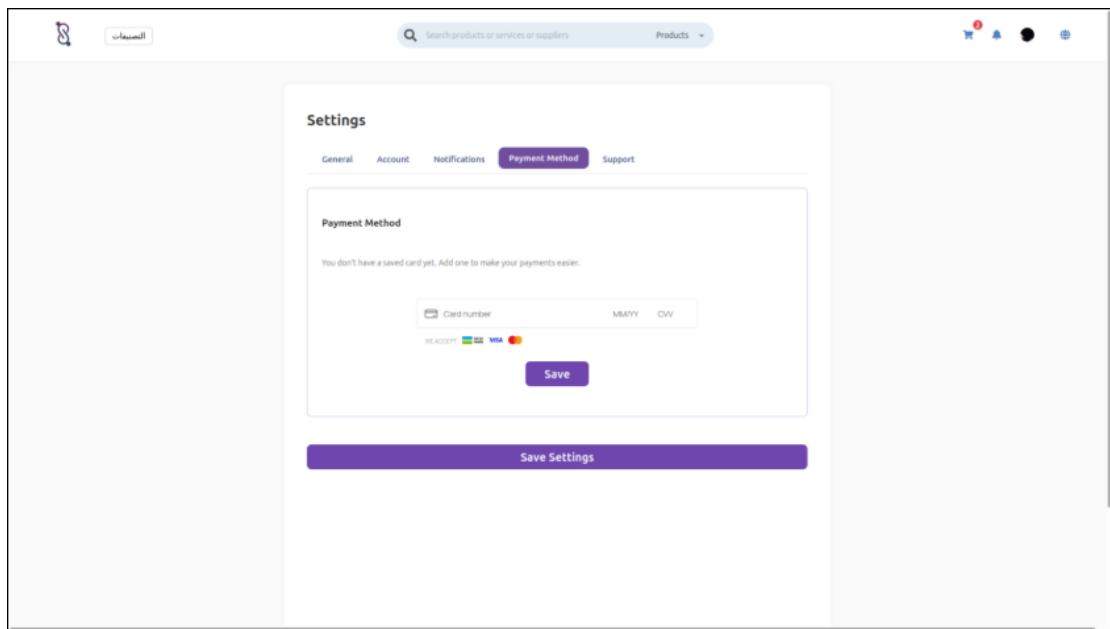


Figure 5-266: Buyer's Settings Page (Payment Method + No Stored Card)

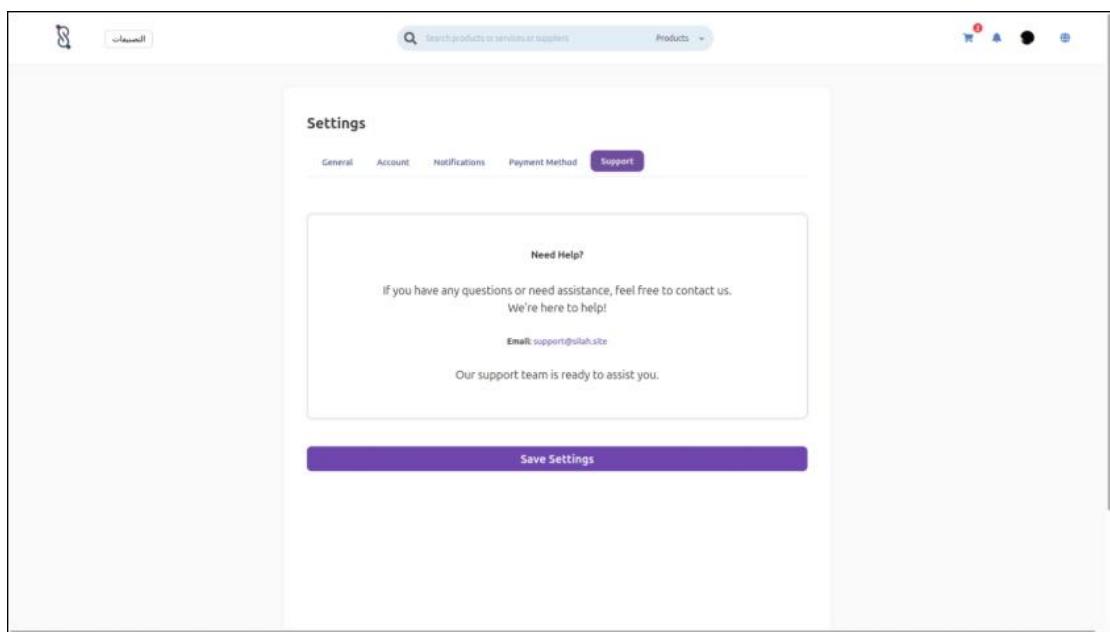


Figure 5-267: Buyer's Settings Page (Support)

Buyer Notifications

Identical behavior to supplier notifications (blue dot for unread, toast messages, “Mark all as read”).

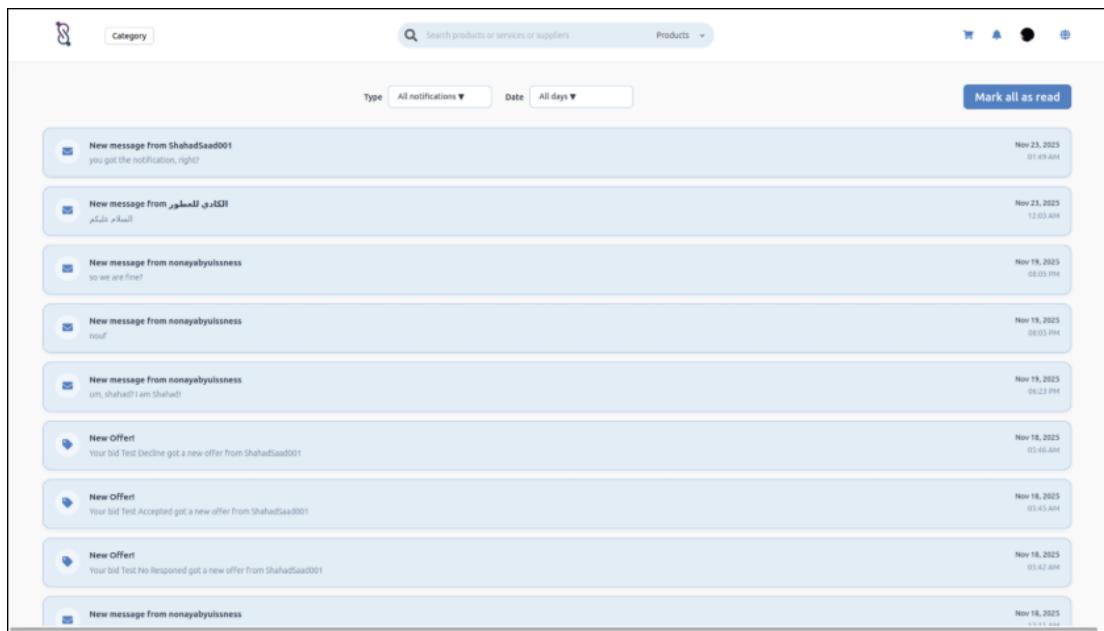


Figure 5-268: Buyer's Notifications Page

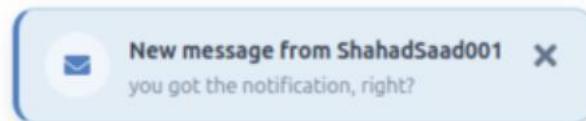


Figure 5-269: Buyer's Notification Toast

Order History

The buyer Orders page mirrors the supplier's implementation in both layout and interaction patterns. The Orders page presents a clean, filterable list of all buyer orders with status, date, supplier name, and total amount. Clicking any order opens the Order Details page, which displays complete order information (items, quantities, pricing, delivery address, tracking info when available, and full status history) using the same easy-to-scan design.

A buyer-specific interaction appears when an order reaches the “Shipped” status. A “Confirm Delivery” button is shown at the bottom of the details page, clicking it instantly updates the order status to “Completed”. This button is hidden for all other statuses.

All	Pending	Processing	Shipped	Completed	Total
#5658575498	23 Nov	ShahadSaad001	Pending	1,382 ₦	
#8899454169	23 Nov	Animals World	Pending	2,005 ₦	
#8898234813	23 Nov	Alkadi perfumes	Pending	915 ₦	
#6963554848	22 Nov	Animals World	Pending	15 ₦	
#4957624848	22 Nov	Animals World	Pending	15 ₦	
#9535834534	19 Nov	ShahadSaad001	Pending	878 ₦	
#9235259449	15 Nov	ShahadSaad001	Completed	320 ₦	
#2375132418	13 Nov	ShahadSaad001	Completed	710 ₦	
#7087746554	13 Nov	ShahadSaad001	Completed	710 ₦	
#0216837248	11 Nov	ShahadSaad001	Completed	250 ₦	
#8184870340	11 Nov	ShahadSaad001	Completed	150 ₦	
#0264374456	11 Nov	ShahadSaad001	Processing	150 ₦	
#1239934759	11 Nov	ShahadSaad001	Completed	710 ₦	
#1694228054	11 Nov	ShahadSaad001	Shipped	710 ₦	
#8713910629	11 Nov	ShahadSaad001	Pending	710 ₦	

Figure 5-270: Buyer's Orders Page

Order #5658575498 Pending				
Ordered on:	23 Nov 2025 - 3:21 AM			
Supplier:	ShahadSaad001			
Total Order Price:	1,382 ₦ (Paid)			
Order Items				
Image	Item Name	Unit Price	Quantity	Total Price
	Lipstick	24	56	1,344

Figure 5-271: Buyer's Order Details Page

The screenshot shows a web-based order management interface. At the top, there are navigation links for 'Category' (with a magnifying glass icon), 'Search products or services or suppliers' (with a magnifying glass icon), 'Products' (with a 'x' icon), and a user profile icon with a red notification dot. Below the header, the order details are displayed:

Order #1694228054 Shipped

Ordered on: 11 Nov 2023 - 2:42 PM
Supplier: [ShahadSaad001](#)
Total Order Price: 710 (Paid)

Order Items

Image	Item Name	Unit Price	Quantity	Total Price
	Lipstick	24	28	672

Confirm Delivery
Click this button if the order has been delivered. This will change the status to "Completed".

Figure 5-272: Buyer's Order Details Page (Shipped)

Direct Messaging & Search for Chats

Shares the same layout and filtering as the supplier version (invoice features omitted).

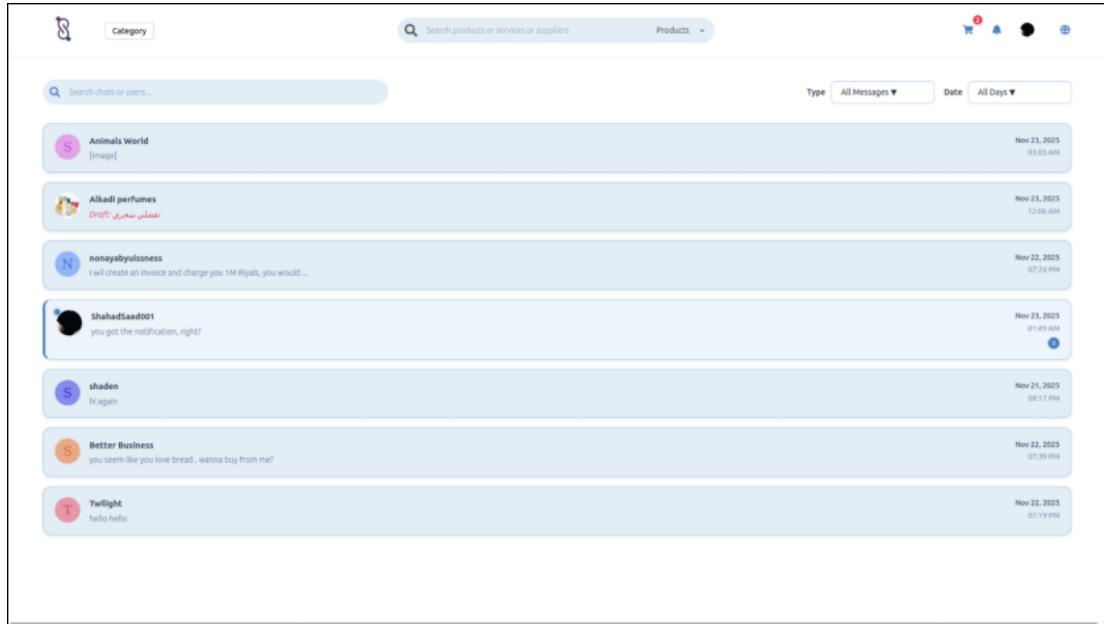


Figure 5-273: Buyer's Chats Page

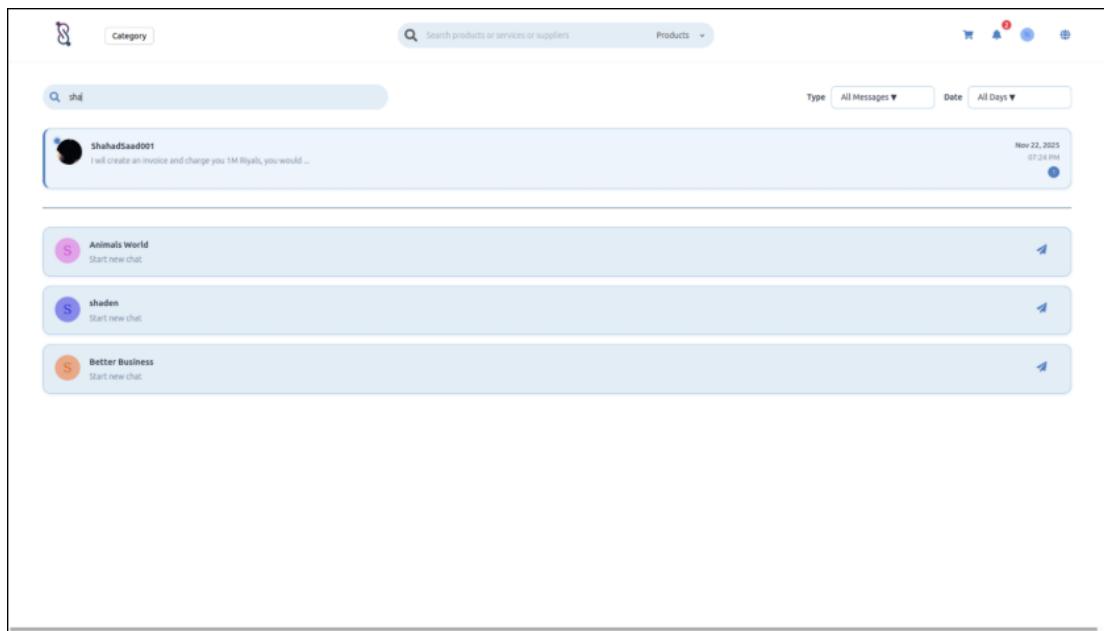


Figure 5-274: Buyer's Chats Page (Search Results)

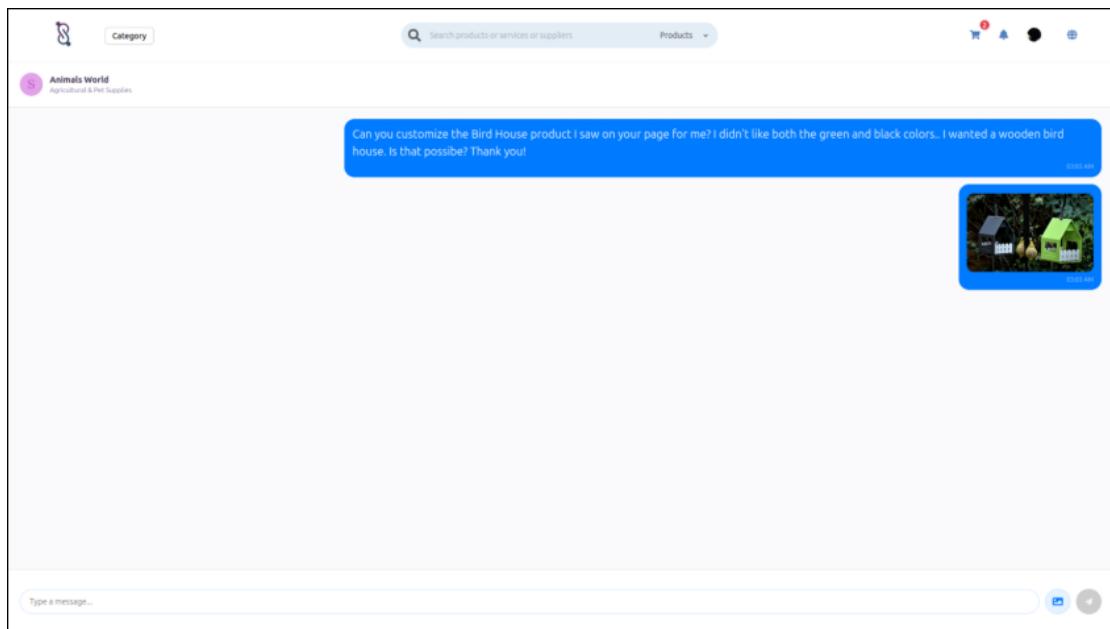


Figure 5-275: Buyer's Chat Page

Invoice Management

Uses the same card-based design and context-aware status filtering introduced for suppliers. Pending invoices offer “Accept/Reject” actions; accepted invoices show “Pay Now”.

Show invoices for: All Products Services Bids Group Purchases

Your Invoices

Invoice	Created	Supplier	Invoice Status	Pre-Invoice Status	Total
#1234567890	23 Nov	ShahadSaad001	—	Pending	4,800 ₩
#6789012345	23 Nov	ShahadSaad001	—	Pending	7,500 ₩
#9876543210	18 Nov	ShahadSaad001	Pending	—	2,500 ₩
#3210987654	18 Nov	ShahadSaad001	—	Failed	1,000,000 ₩
#5432109876	18 Nov	ShahadSaad001	—	Successful	2,500 ₩
#8765432109	18 Nov	ShahadSaad001	—	Failed	200 ₩
#0987654321	18 Nov	ShahadSaad001	—	Failed	500 ₩
#7654321098	18 Nov	nonayabusiness	Accepted	—	2,000 ₩
#5550023429	17 Nov	ShahadSaad001	Fully Paid	—	10 ₩
#9174837462	17 Nov	ShahadSaad001	Fully Paid	—	100 ₩
#5863348138	17 Nov	ShahadSaad001	Fully Paid	—	120 ₩
#2205914264	13 Nov	ShahadSaad001	Fully Paid	—	15 ₩
#6372673465	13 Nov	ShahadSaad001	Fully Paid	—	210 ₩

Figure 5-276: Buyer's Invoices Page

Invoice #0371446940 Pending

Issue Date: 23/11/2025 | **Delivery Date**: 30/11/2025 | **Terms of Payment**: Fully Paid | **Total Amount**: \$ 200.00 ₩

Supplier		Buyer	
ShahadSaad001	SHAHAD SAAD	Riyadh	shahadsaad001.dev@gmail.com
ShahadSaad001	SHAHAD SAAD	Riyadh	shahadsaad001.dev@gmail.com

Items

Item	Description	Agreed Details	Qty	Unit Price	Total Price	Linked
test	test	test	1	200.00	200.00	

Total: 200.00 ₩

Accept Invoice | **Reject Invoice**

Figure 5-277: Buyer's Invoice Details Page (Pending)

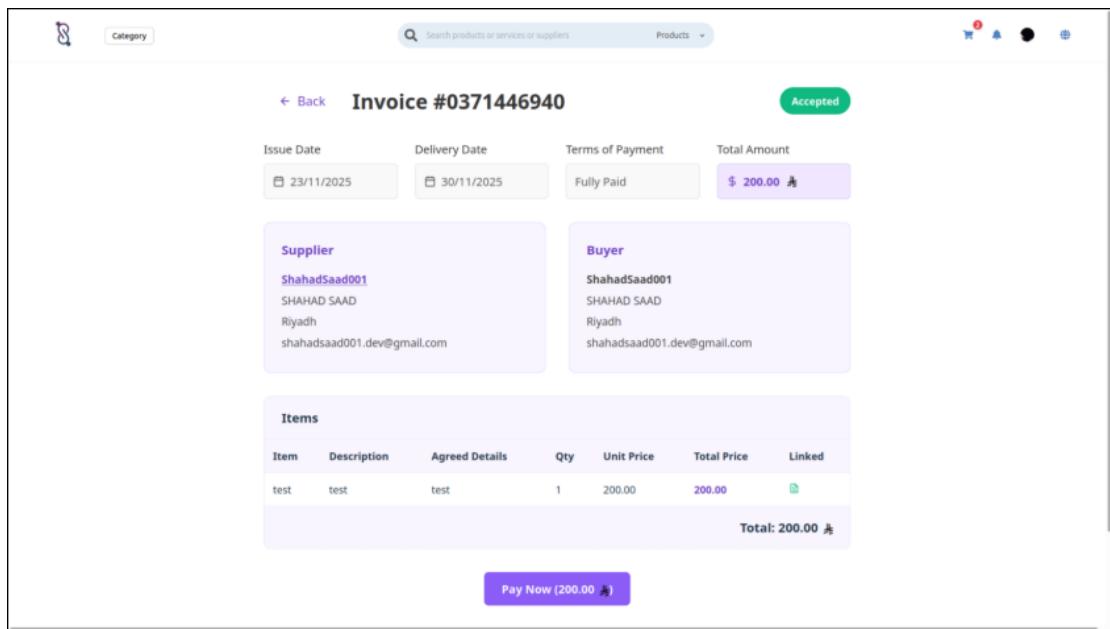


Figure 5-278: Buyer's Invoice Details Page (Accepted)

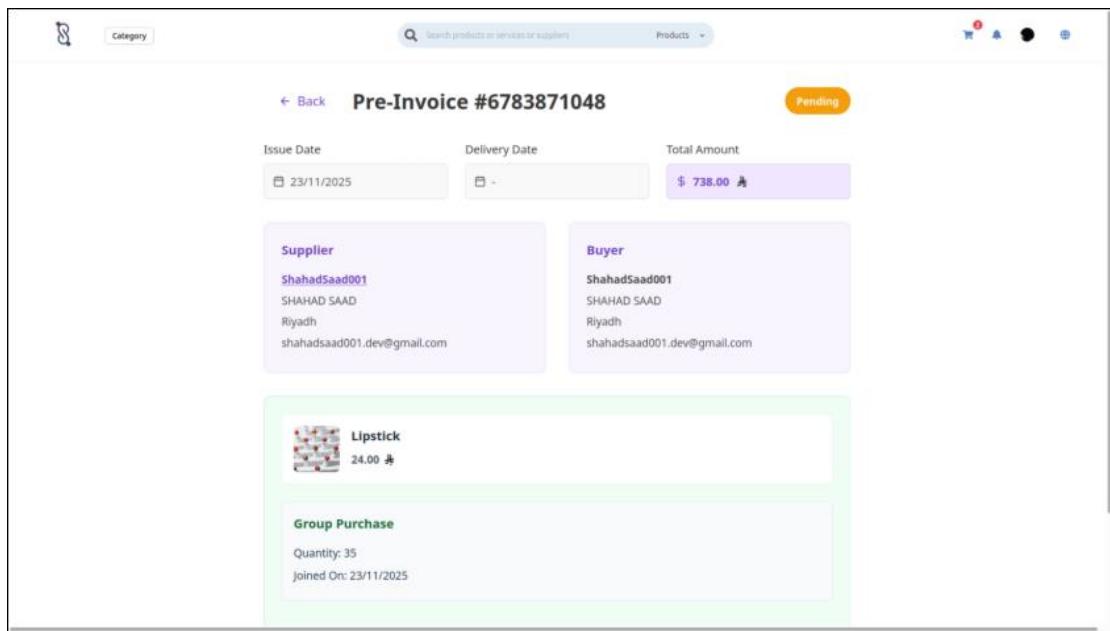


Figure 5-279: Buyer's Pre-Invoice Details Page (Group Purchase)

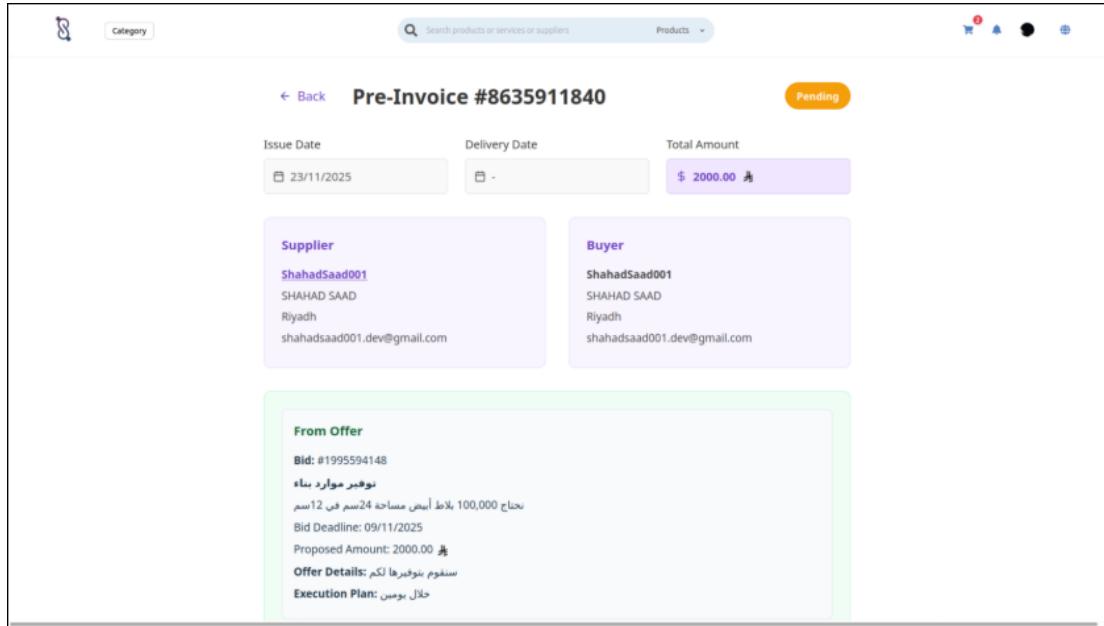


Figure 5-280: Buyer's Pre-Invoice Details Page (Offer)

Write a Review

The Write a Review page is only accessible when the buyer has either confirmed delivery of an order (status: Completed) or fully paid an invoice (status: Fully Paid). In these cases, the corresponding Order Details or Invoice Details page displays a clear “Write a Review” button. If a draft review already exists for that order or invoice, the button text changes to “Continue Writing Review” so the buyer can resume where they left off. The review form itself provides a straightforward interface for rating the supplier and each item and allows optional comments.

Order #1694228054 Completed

Ordered on: 11 Nov 2023 - 2:42 PM
Supplier: ShahadSaad001
Total Order Price: 710 ₩ (Paid)

Order Items

Image	Item Name	Unit Price	Quantity	Total Price
	Lipstick	24	28	672

You Confirmed The Delivery of This Order
[Write a Review](#)
You can write a review after confirming the delivery.

Figure 5-281: Buyer's Order Details Page (Completed)

[← Back](#) **Invoice #0371446940** Fully Paid

Issue Date	Delivery Date	Terms of Payment	Total Amount
23/11/2025	30/11/2025	Fully Paid	\$ 200.00 ₩

Supplier
 ShahadSaad001
 SHAHAD SAAD
 Riyadh
 shahadsaad001.dev@gmail.com

Buyer
 ShahadSaad001
 SHAHAD SAAD
 Riyadh
 shahadsaad001.dev@gmail.com

Items

Item	Description	Agreed Details	Qty	Unit Price	Total Price	Linked
test	test	test	1	200.00	200.00	

Total: 200.00 ₩

[Write a Review](#)

Figure 5-282: Buyer's Invoice Details Page (Fully Paid)

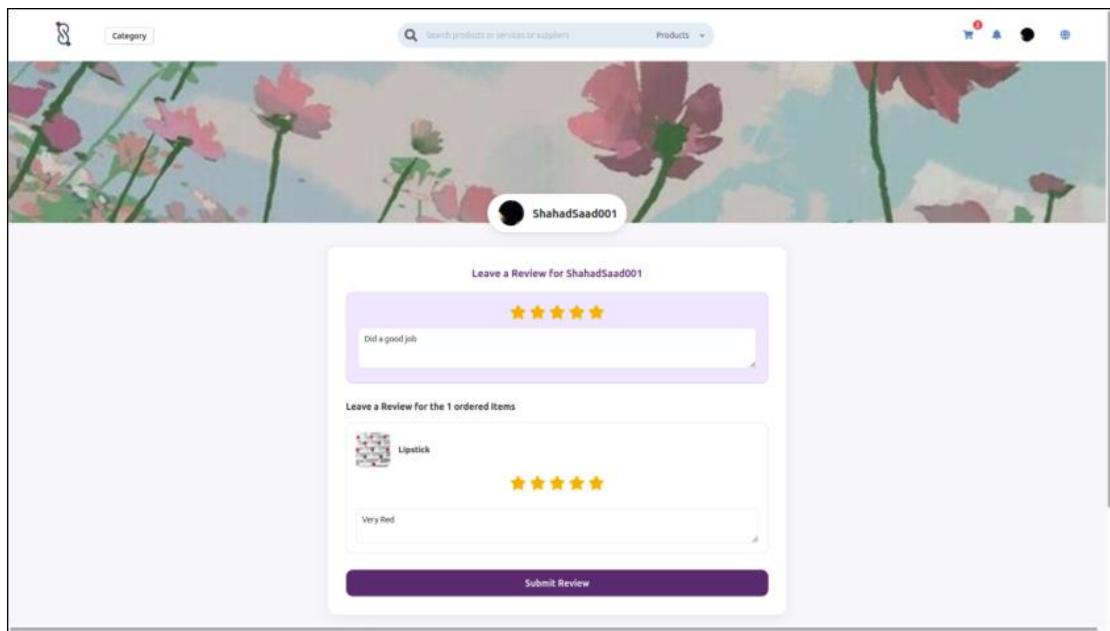


Figure 5-283: Write a Review Page

5.3.4 Language Support & Empty States

All implemented screens fully support bilingual operation (English left-to-right and Arabic right-to-left). Switching languages via the language selector instantly mirrors the entire interface, including text direction, alignment, and all UI elements, while preserving the exact same layout and functionality.

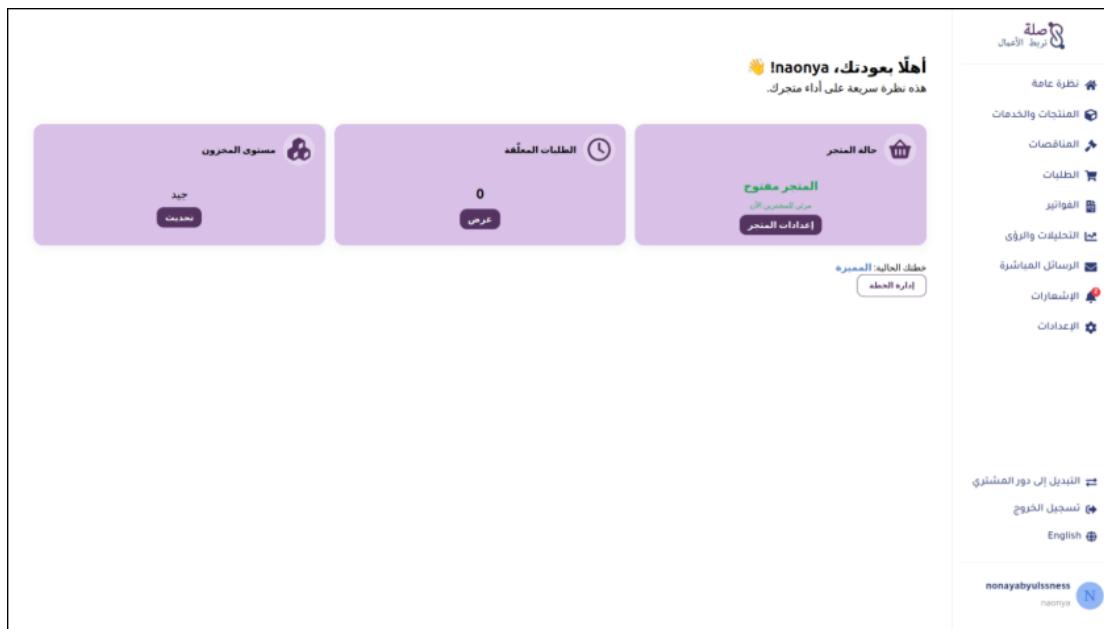


Figure 5-284: Overview in RTL



Figure 5-285: Analytics in RTL

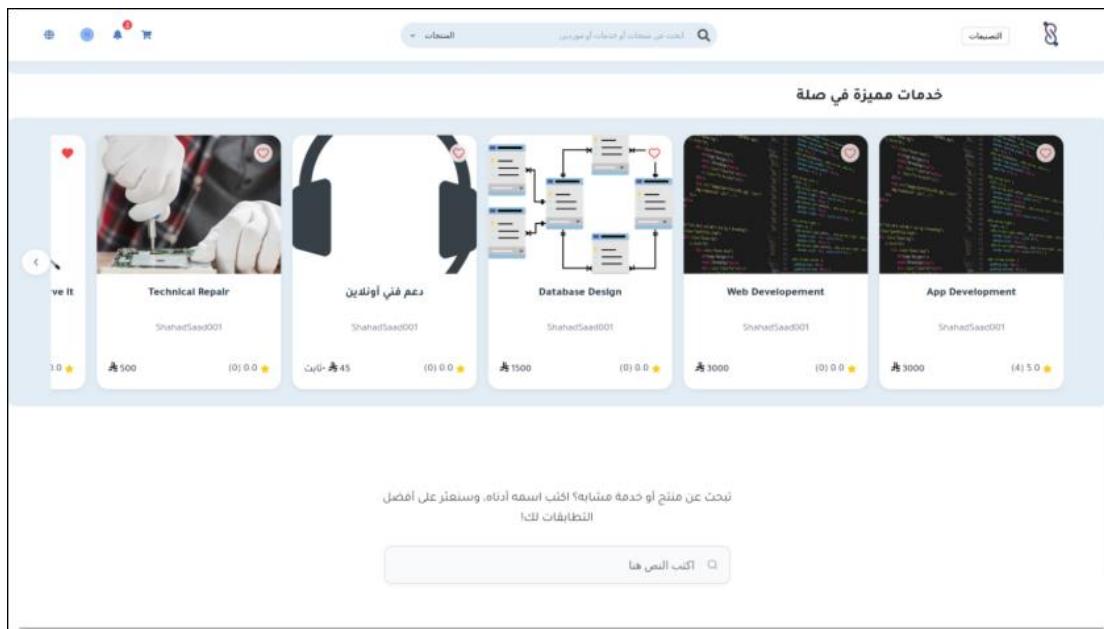


Figure 5-286: Homepage in RTL

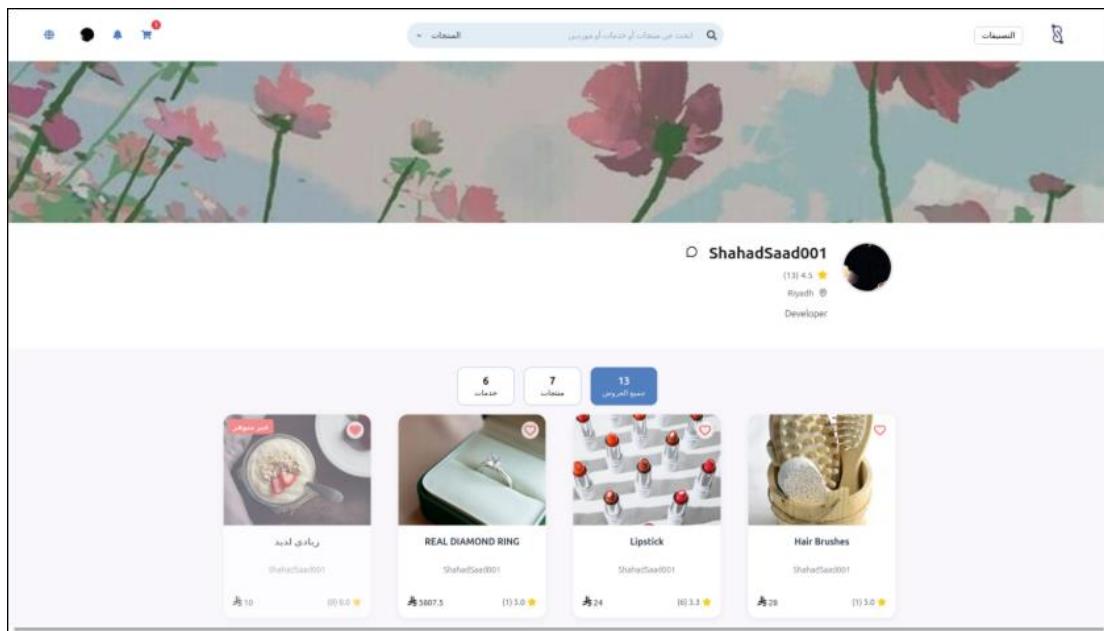


Figure 5-287: Storefront in RTL

Equally important attention was given to graceful empty and no-data states across the platform. Instead of blank pages or raw error messages, the pages display friendly, contextual illustrations and clear guidance when no content is available. These thoughtful empty states maintain visual consistency, reduce user confusion, and gently encourage next actions throughout both buyer and supplier experiences.

The screenshot shows the SILAH platform's Listings page. The left sidebar contains navigation links: Overview, Products & Services, Biddings, Orders, Invoices, Analytics & Insights, Direct Messages, Notifications, and Settings. Below these are links for changing role, logging out, and switching language. The main content area has a search bar at the top with the placeholder "Q - Nothing". A table below it is titled "Select Item(s) to take an action" and includes columns for Image, Item Name, Unit Price, Stock, Status, and Predict Demand. A message at the bottom of the table says "No Items to Show". At the top right, there are buttons for "Create a New Product" and "Create a New Service".

Figure 5-288: Listings (No Search Result)

The screenshot shows the SILAH platform's Your Orders page. The left sidebar is identical to Figure 5-288. The main content area is titled "Your Orders" and includes a sub-instruction: "Orders get the status 'Completed' once the buyer confirms delivery." Below this is a navigation bar with tabs: All (which is selected), Pending, Processing, Shipped, and Completed. A central message states "No orders yet" with the sub-instruction "When you place or receive orders, they'll appear here." At the bottom right, there is a user profile section for "nonayabyuissness" (Naomya).

Figure 5-289: Supplier's Orders Page (No Orders)

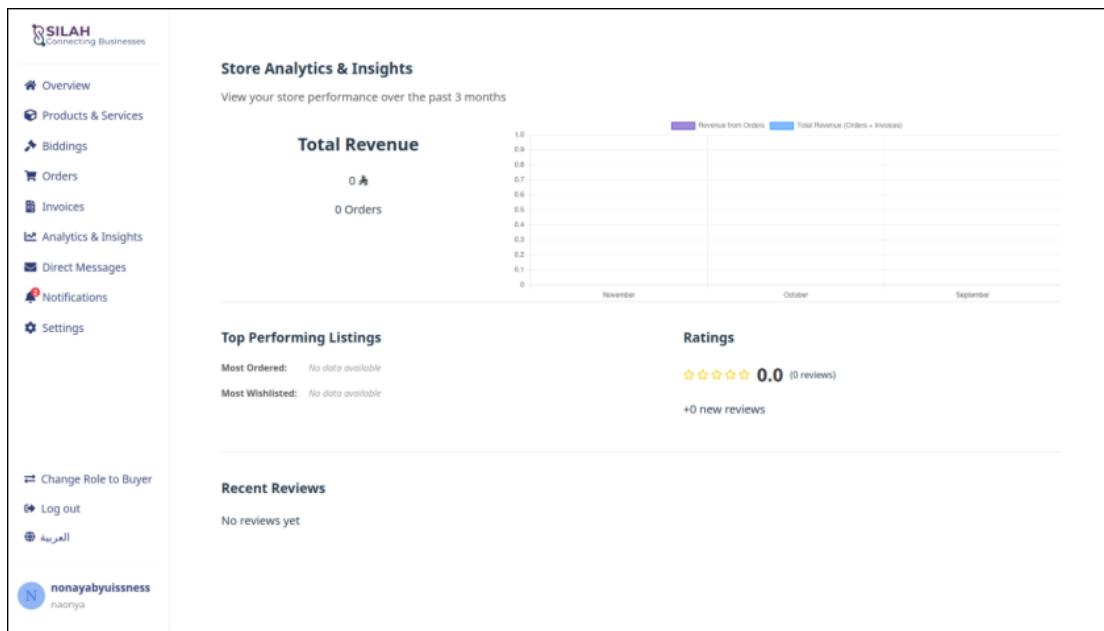


Figure 5-290: Analytics (No Data)

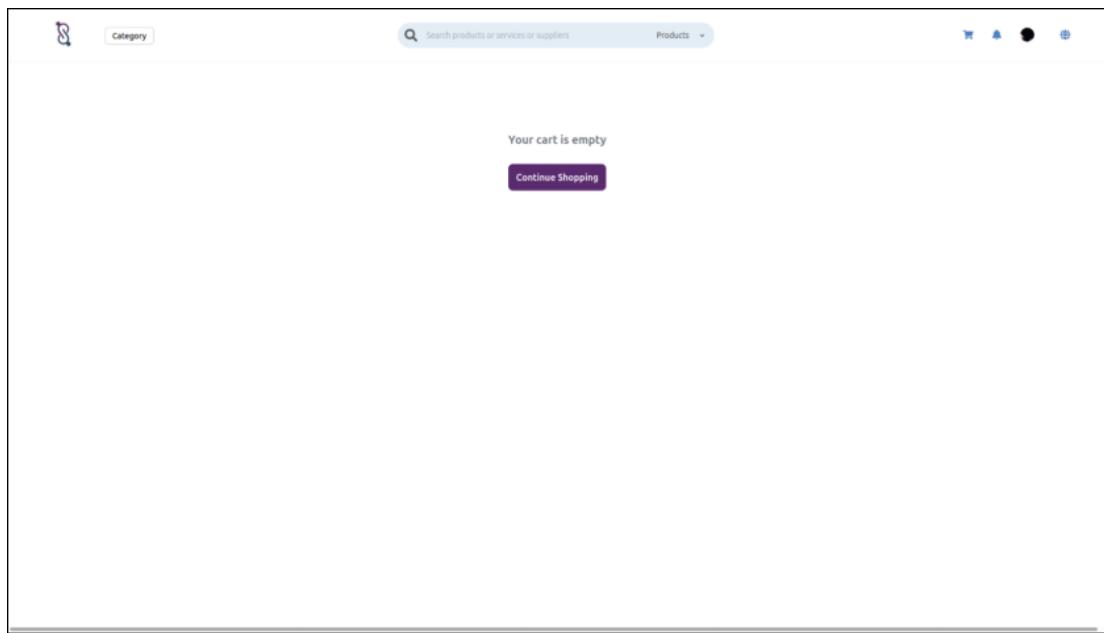


Figure 5-291: Cart (Empty)

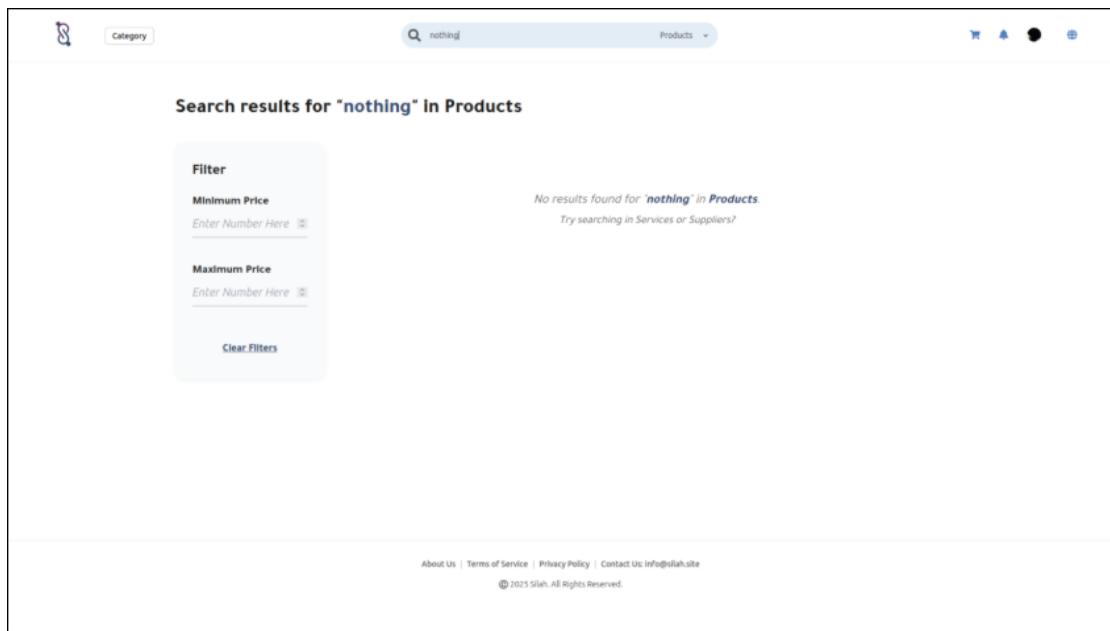


Figure 5-292: Search Results Page (Products + No Results)

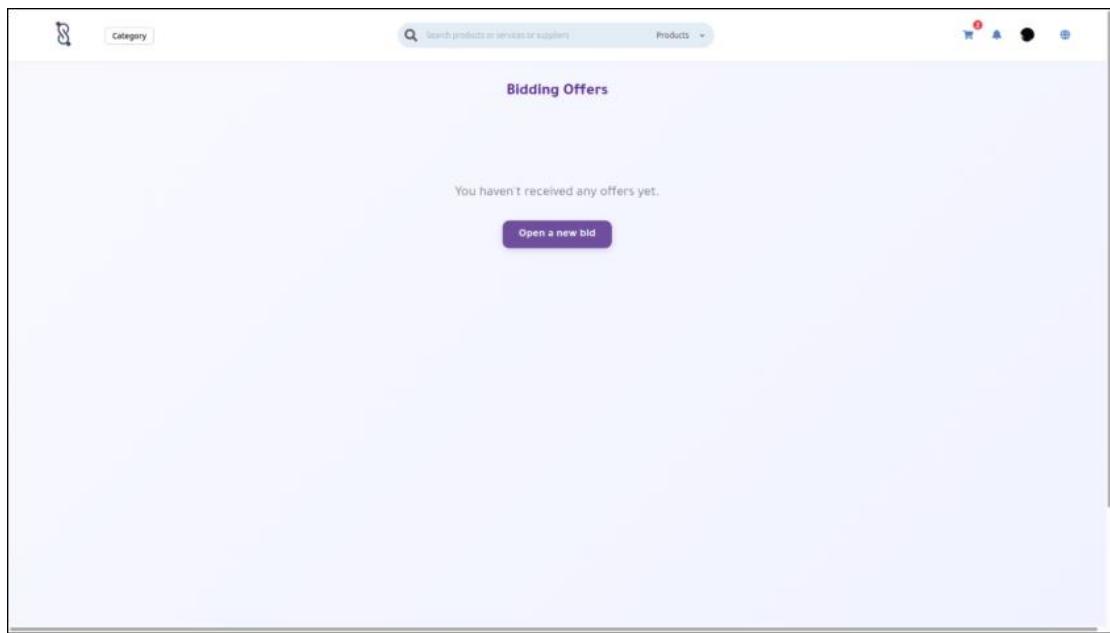


Figure 5-293: Received Offers Page (No Offers)

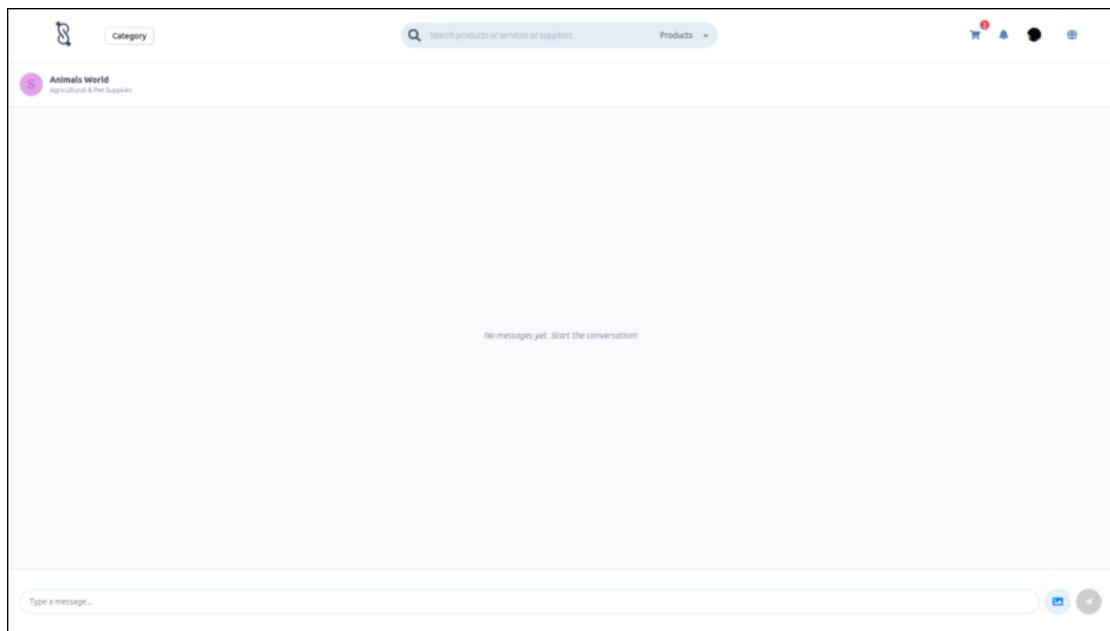


Figure 5-294: Buyer's Chat Page (New Chat)

This chapter has detailed the complete realization of the system described in Chapters 3 and 4. From the requirements, class diagrams, sequence diagrams, database schema, and architecture to the final running application. The development process successfully translated these specifications into a fully functional web platform that supports the entire guest, buyer, and supplier workflows.

Throughout implementation, the team maintained strict fidelity to the approved designs whenever they proved effective in practice. At the same time, continuous internal review and hands-on testing revealed opportunities for meaningful improvement; these were addressed through measured refinements such as card-based layouts, context-aware filtering, tabbed settings, and clearer real-time feedback. Each change was introduced only where it demonstrably increased usability and clarity, resulting in an interface that feels significantly more polished and intuitive than the original prototypes while remaining completely faithful to the defined functional requirements.

The system presented in this chapter is the exact platform that underwent comprehensive testing. Chapter 6 documents the full test plan, test cases, and results, confirming that the implemented solution not only satisfies but, in numerous aspects, exceeds the requirements and expectations originally established.

Chapter 6 Testing

With the system fully developed and deployed, this chapter evaluates how well it performs against its defined requirements and the expectations set throughout the project. We present the testing strategy adopted for the Silah System, covering automated backend tests, functional validation, and non-functional assessments. Each testing category examines whether the system delivers on its promises in terms of correctness, reliability, portability, and user experience. By reviewing the methods used, the results obtained, and the overall quality of the system, this chapter provides a comprehensive assessment of the system's readiness for real-world use.

6.1 Test Plan

The purpose of this test plan is to verify that the developed system meets its specified functional and non-functional requirements and performs reliably after deployment. The testing aims to ensure that the implemented features work as intended, that the system remains available and stable, and that it delivers a satisfactory user experience across different browsers and environments.

The testing scope covers both the frontend and backend components of the system. The backend, developed using NestJS, and the frontend, built with React, were both deployed on DigitalOcean for testing. The scope includes all core functionalities such as authentication, CRUD operations, and role-based access control, as well as usability and non-functional aspects like availability, reliability, and portability. End-to-end testing was excluded due to limited project time and because major integrations had already been validated during development.

The overall testing strategy combines manual and automated testing approaches. Manual testing was carried out continuously throughout the development process to verify that each page and workflow behaved as expected during user interaction. We manually checked different flows in the frontend and backend to identify usability and logic issues early. Although these results are not formally documented, manual testing played an important role in maintaining system quality.

Automated testing focused on the backend and was implemented using Jest to perform both unit and integration testing. Unit testing validated individual modules and functions in isolation, such as authentication, validation logic, and utility methods, while integration testing confirmed the correct interaction between

connected modules, including database operations and service communication. Because the system includes a large number of test cases, only the most significant or unique ones will be presented in later sections, while coverage percentages and summary results will be reported at the end.

Functional testing was performed to ensure that the system's behavior aligns with its intended requirements. Rather than testing every single feature, the focus was placed on key and representative scenarios, such as verifying that a buyer cannot access supplier pages, that the signup and login functions operate correctly, and that role-based restrictions are properly enforced. These tests were demonstrated with screenshots as proof of correct functionality.

Non-functional testing focused on evaluating quality attributes rather than system behavior. This included availability, reliability, portability, and usability testing. Availability testing verified that the deployed system maintained consistent uptime using DigitalOcean's monitoring tools. Reliability testing examined whether the system could restore user data after unexpected closures or crashes. Portability, or cross-browser compatibility testing, ensured that the system displayed and functioned correctly on major browsers such as Google Chrome, Mozilla Firefox, and Safari. Finally, usability testing measured the ease of use and user satisfaction by collecting feedback from participants through observation, with a target satisfaction score of 70% or higher. The evaluation was conducted using an observable method known as the Think-Aloud technique, where participants were asked to perform predefined tasks while verbalizing their thoughts and reactions in real time. This approach allowed us to identify usability challenges and understand user perceptions directly during interaction. Each of these non-functional aspects is discussed and evidenced in later subsections.

All testing activities were performed in the deployed DigitalOcean environment to ensure that the results reflect realistic operating conditions. Backend tests were conducted through the live API endpoints, and frontend tests were performed on multiple browsers and devices. Testing took place during the final phase of development and was distributed over several weeks: early weeks were devoted to manual verification, followed by automated unit and integration testing, then functional and non-functional testing toward the end. The resulting deliverables

include documented test cases, execution results, coverage summaries, screenshots, and a final evaluation of how well the system satisfies its defined requirements.

6.2 Test Cases

This section presents the different categories of tests conducted to validate the system's functionality and performance against the specified requirements. The tests were organized into unit testing, integration testing, functional testing, and non-functional testing. Each testing level addressed a specific objective to ensure that the system was developed according to quality standards and met both user and business expectations.

6.2.1 Unit Testing

Unit testing focused on verifying the correctness of individual components within the backend system. Each function was tested in isolation to confirm that it produced the expected output given defined inputs. The tests were implemented and executed using Jest, providing a fast and automated approach to detect regressions early in development.

Key unit tests included validation logic, data formatting, and utility functions that support the system's main operations. For example, one distinctive case ensured that when a user deleted their profile picture, the system automatically generated a default avatar image. This test verified that user profiles remain consistent even when optional data is removed.

In total, a significant number of unit tests were successfully executed on the backend. While the testing phase did not cover the entire system, the tests performed were sufficient to ensure that the core features functioned correctly.

The table below summarizes the main unit test scenarios, expected outcomes, actual outcomes, and their status:

Table 6-1: Unit Testing Results

Feature	Test Scenario	Expected Outcome	Actual Outcome	Status

signUp	Sets partitioned + httpOnly cookie	res.cookie called with token, { httpOnly: true, partitioned: true, sameSite: 'lax' }	res.cookie called with token, { httpOnly: true, partitioned: true, sameSite: 'lax' }	Pass
resetPassword	Returns success message	{ message: 'Password reset successfully' }	{ message: 'Password reset successfully' }	Pass
switchUserRole	BUYER to SUPPLIER	{ message: 'Role switched successfully', newRole: 'SUPPLIER' }	{ message: 'Role switched successfully', newRole: 'SUPPLIER' }	Pass
deleteProfilePicture	Attempt to delete when already default	Throws BadRequestException with message 'Profile picture already default'	Throws BadRequestException with message 'Profile picture already default'	Pass
deleteProfilePicture	Deletes profile picture successfully when not default	{ message: 'Profile picture deleted successfully' }	{ message: 'Profile picture deleted successfully' }	Pass
generateDefaultAvatar	Generates default avatar using	Returns path to uploaded default avatar (e.g., 'default-	Returns path to uploaded default avatar (e.g., 'default-	Pass

	sharp mock	avatars/user- 123.png')	avatars/user- 123.png')	
--	---------------	----------------------------	----------------------------	--

Overall, the unit testing phase contributed significantly to the reliability and stability of the backend system.

6.2.2 Integration Testing

Integration testing verifies that the individual modules of the system interact correctly when combined. These tests, if fully implemented, ensure that data flows correctly between controllers, services, and the database, and that dependent modules handle unexpected situations gracefully.

Due to time constraints and the focus on implementing the core features of the system, we were only able to create automated integration tests for the User module. This limitation was not a matter of choice, but rather a necessity given the project deadlines and the priority of delivering functional features.

Despite this, we deployed the backend early, which provided a practical environment for manual testing and allowed the frontend team to work with a fully functioning backend. This early deployment enabled prompt feedback, early identification of issues, and rapid fixes; demonstrating that, even without extensive automated integration tests, the system was tested effectively in practice.

While we acknowledge that automatic integration testing is crucial, and we would have ideally written more, the combination of feature implementation, early deployment, and manual testing ensured the reliability of the system during development. The implemented integration tests still verified critical interactions within the User module, and they serve as a foundation for expanding tests across other modules in the future.

Table 6-2: Integration Testing Results

Feature	Test Scenario	Expected Outcome	Actual Outcome	Status
---------	---------------	------------------	----------------	--------

GET /users/email/:e mail	Return 404 for unkno wn email	404 with message “not found”	404 with message “not found”	Pass
GET /users/:id/profi le-picture	Return correct URL	{ pfpUrl: 'https://r2.mock.dev/bu yer.png' }	{ pfpUrl: 'https://r2.mock.dev/bu yer.png' }	Pass
GET /users/:id/profi le-picture	404 when no picture	404	404	Pass
POST /users/profile- pictures/batch	Return correct URLs for multipl e users	Array with correct pfpUrls	Array with correct pfpUrls	Pass
GET /users/me	Reject missin g token	401 Unauthorized	401 Unauthorized	Pass

Although the scope of automated integration testing was limited, we prioritized delivering working features and ensuring early deployment. This allowed for effective manual testing, early feedback, and verification of key system interactions. Future work will expand automated integration testing to cover additional modules, further strengthening system reliability.

6.2.3 Functional Testing

Functional testing was conducted across the entire system to verify that all major features operate correctly and that the implemented functionalities satisfy the project's requirements. This included validating user registration, authentication, data management operations, and all other core features expected of the platform. The complete system was tested end-to-end to ensure that users can perform all tasks as intended, and these successful flows were already demonstrated extensively in Section 5.3 through detailed screenshots and result evidence.

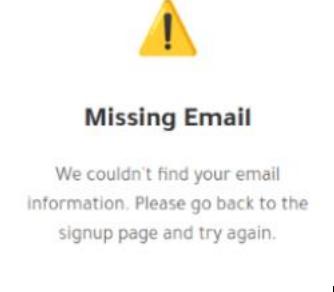
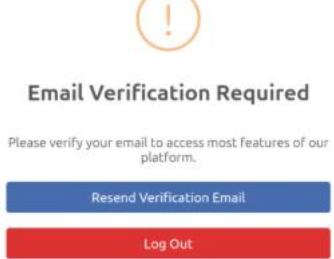
For documentation purposes, this chapter does not repeat every successful scenario already proven. Instead, the focus is placed on how the system responds to incorrect or restricted actions, including invalid inputs, violations of business rules, boundary conditions, and access-control restrictions. These scenarios provide clearer evidence of the system's robustness by showing that validation rules are enforced, improper operations are blocked, and unauthorized behavior is prevented.

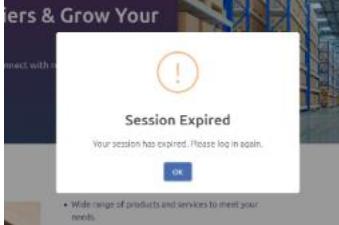
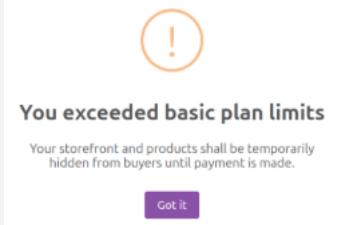
Evaluating these exception and boundary cases not only confirms that the system handles errors correctly, but also indirectly evidences that the corresponding valid scenarios succeed, as both rely on the same underlying validation and business logic layers. The following documented tests therefore highlight the system's most critical exception-driven cases, each supported by focused screenshots of the exact message or dialog shown to the user, providing clear, objective evidence that the system behaves correctly even when things go wrong.

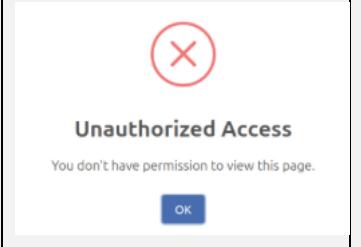
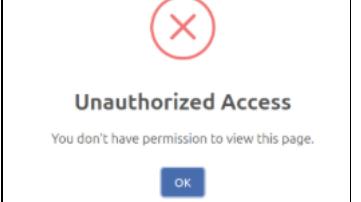
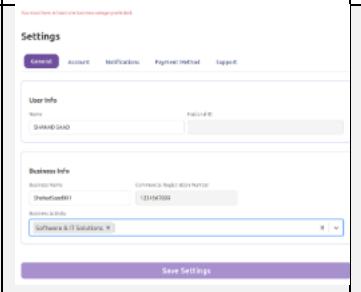
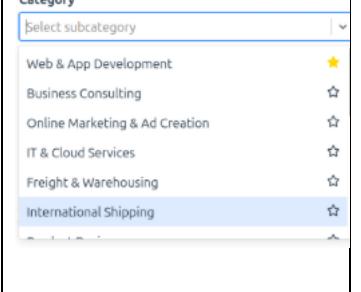
Table 6:- Functional Testing Results

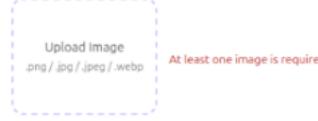
Feature	Test Scenario	Expected Outcome	Actual Outcome	Status
Sign-up	Enter a CRN that already exists in the system	Error: "A company with this commercial registration already exists"	 Figure 61: CRN exists	Pass

Sign-up	Enter a NID that already exists in the system	Error: “A user with this National ID already exists”	 Figure 62: NID exists	Pass
Sign-up	Enter a CRN that is fewer or more than exactly 10 digits	Error: “CRN must be exactly 10 digits”	 Figure 63: CRN must be 10 digits	Pass
Sign-up	Enter a NID that is not exactly 10 digits	Error: “National ID must be 10 digits”	 Figure 64: NID must be 10 digits	Pass
Sign-up	Enter an invalid email address format	Error: “Please enter a valid email address”	 Figure 65: Enter valid email	Pass
Sign-up	Enter a password missing uppercase, lowercase, number, or < 8 characters	Error: “Password must include uppercase, lowercase, number and be 8–28 characters long”	 Figure 66: Weak password error	Pass
Sign-up	Confirmation password does not match the original password	Error: “Passwords do not match”	 Figure 67: Passwords does not match	Pass

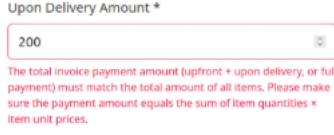
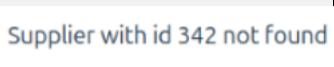
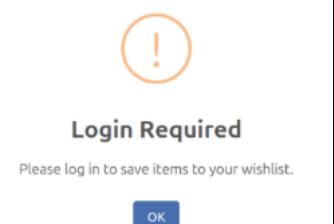
Sign-up	Enter a city name containing numbers or special characters (e.g. “Riyadh-1”)	Error: “City name can only contain letters”	 Figure 68: City name must be letters only	Pass
Sign-up	Enter a fake or non-existent CRN	Error: “The provided CRN does not exist”	 Figure 69: CRN is not real error	Pass
Sign-up	Enter a real CRN that belongs to an inactive business	Error: “The CRN exists but is not active”	 Figure 610: CRN is not active error	Pass
Email Verification	User manually types the email verification page URL (direct URL attack or copy-paste after sign-up)	Modal dialog: “Missing email”	 Figure 611: Verify Email Page (Missing Email Dialog)	Pass
Post-Login (any user)	User has signed up but did not verify their email	Modal dialog: “Email verification is required.” with two buttons: “Resend verification email” and “Log out”	 Figure 612: Email Verification Required Modal Dialog	Pass

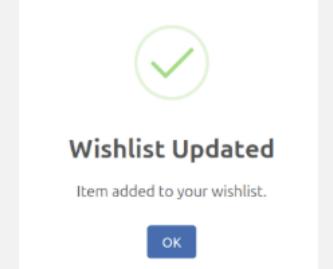
Post-Login (any user)	User's session token has expired (token lifetime = 1 day)	Modal dialog: “Session expired” and user is automatically logged out and redirected to landing page		Pass
Supplier Account Inactive	Logged-in supplier has exceeded basic plan limits	Modal dialog: “You exceeded the Basic plan limits”		Pass
Supplier Account Inactive	Inactive supplier attempts to access a restricted page (e.g., Add Product, Analytics, etc.)	Toast error message: “This page is not available while your plan is inactive” and user is redirected to the supplier overview		Pass

Role-Based Access Control	Buyer attempts to directly access a supplier-only page (e.g., /supplier/overview, /supplier/orders)	Modal dialog: “Unauthorized access” and redirect user to buyer homepage	 <p>Figure 616: Unauthorized Access Dialog</p>	Pass
Role-Based Access Control	Supplier attempts to directly access a buyer-only page (e.g., /search, /buyer/cart)	Modal dialog: “Unauthorized access” and redirect user to supplier overview	 <p>Figure 617: Unauthorized Access Dialog</p>	Pass
Profile (Business Info)	User attempts to remove the last selected business category in Settings	Error: “You must have at least one business category selected”	 <p>Figure 618: Business Activity is Required (Settings Page Error)</p>	Pass
Supplier Favorite Categories	Supplier opens the category dropdown while creating or editing a product or service	Previously favorited subcategories appear at the top of the dropdown list with a star icon (e.g., “Web & App Development”)	 <p>Figure 619: Supplier Favoriting Categories</p>	Pass

Listing Creation	Supplier submits a new product or service without entering a price	Inline validation: “Price is required”	 Figure 620: Price is required	Pass
Listing Creation	Supplier submits a new product or service without uploading any image	Inline validation: “At least one image is required”	 Figure 621: At least one image is required	Pass
Listing Creation	Supplier removes the last (or only) image while editing a listing	Temporarily Inline validation: (5 seconds): “At least one image is required”	 Figure 622: Can't delete last image	Pass
Product Creation (Quantities)	Minimum order quantity entered is not a multiple of the selected case quantity	Inline validation: “Minimum order quantity must be a multiple of the case quantity”	 Figure 623: Minimum order requirement must be of the case quantity	Pass
Product Creation (Quantities)	Maximum order quantity entered is not a multiple of the case quantity or is lower than the minimum	Inline validation: “Maximum order quantity must be a multiple of	 Figure 624: Maximum order requirement must be of the case quantity	Pass

		the case quantity”		
Product Creation (Group Purchase)	Group purchase is enabled, and the supplier sets the group purchase price \geq standard price	Inline validation: “Group purchase price cannot be equal to or higher than the standard price”	 Figure 625: Group purchase price must be less than standard price	Pass
Invoice Creation	Supplier tries to create an invoice without adding any items	Validation message: “Add at least one item”	 Figure 626: Add at least one item to the invoice	Pass
Invoice Creation	Supplier attempts to delete the last remaining item from an invoice	Toast: “Cannot delete the last item”	 Figure 627: Can't delete last item Toast	Pass
Invoice Creation	An invoice item is not linked to any published product/service	Validation message: “Each item must be linked to a published product or service”	 Figure 628: Must link the item	Pass

Invoice Creation	Upon-delivery amount (or combination of upfront + upon-delivery) does not equal the calculated total of all items	Inline validation: “The total invoice payment amount (upfront + upon-delivery or upon-delivery only) must match the total amount of all items”	 <p>The total invoice payment amount (upfront + upon delivery, or full payment) must match the total amount of all items. Please make sure the payment amount equals the sum of item quantities x item unit prices.</p>	Pass
Resource Access (Any Page)	User (or guest) accesses a URL containing an invalid, deleted or non-existent ID	Page displays friendly error message: “X with id Y not found”	 <p>Supplier with id 342 not found</p>	Pass
Wishlist (Guest)	Unauthenticated user clicks the wishlist heart icon on any product/service	Modal dialog: “Login required”	 <p>Login Required Please log in to save items to your wishlist. OK</p>	Pass

Wishlist (Buyer)	Logged-in buyer adds an item to wishlist or removes an item from wishlist	Modal dialog: “Wishlist updated”		Pass
Add to Cart (Quantity Validation)	Buyer enters a quantity that is not a multiple of the product's case quantity (e.g. enters 2 when case = 7)	Inline error: “Must be a multiple of case quantity (7)”		Pass
Add to Cart (Minimum Order)	Buyer enters a quantity that is a multiple of case quantity but below the product's minimum order quantity (e.g. enters 7 when min = 28)	Inline error: “Minimum order quantity: 28”		Pass

Add to Cart (Stock Availability)	Buyer tries to add more units than currently available in stock	Modal dialog: “Only XX units available in stock”	 Error Only 96 units available in stock OK	Pass
Cart	Cart contains an item that went out of stock after it was originally added	Item is visually marked “Out of stock”, checkout button disabled, message shown: “Remove the out-of-stock item to proceed with checkout” and a “Find similar products” link is provided		Pass
Checkout	Buyer reaches checkout without any saved payment method	Modal dialog: “You must add a payment card before checking out”	 Card Required You must add a payment card before checking out. Go to Payment Settings cancel	Pass

		(with options to go to payment settings or cancel)		
Payment Callback	Payment gateway returns a failure, and the user is redirected back to the site	Modal dialog: “Charge not successful. Redirecting in ...”	 Charge not successful yet. Status: DECLINED Redirecting in 2...	Pass
Group Purchase	Buyer attempts to join a group-purchase deal they have already joined	Modal dialog: “You have already joined this group purchase”	 Error You already joined this group purchase <input type="button" value="OK"/>	Pass

Test Results Summary

Table 6:- Functional Test Results Summary

Total Test Cases Executed	Passed	Failed	Pass Rate
35	35	0	100%

All critical exception scenarios and boundary conditions tested were successfully handled by the system. Every validation rule triggered the exact expected user-facing error message, dialog, toast, or redirect, and no invalid or unauthorized actions were permitted.

The complete absence of failures in these deliberately adversarial test cases, combined with the extensive successful workflow demonstrations already presented in Section 5.3, provides strong evidence that the platform fully satisfies all functional requirements and behaves robustly in both normal and erroneous conditions. Resulting in a system that is production-ready from a functional perspective and exceeds the original project requirements in terms of input validation, business-rule enforcement, and graceful error handling.

6.2.4 Non-Functional Testing

Non-functional testing focused on evaluating the system's performance and quality attributes beyond functionality. These included availability, reliability, portability, and usability. Each aspect was tested using methods appropriate to its metric and purpose, as described below.

6.2.4.1 Availability

Availability testing verified that the deployed system met its defined uptime requirements, ensuring consistent accessibility and performance. According to the system specifications, the platform shall maintain a minimum uptime of 98% per month and shall not experience downtime exceeding two consecutive hours.

The test was conducted using DigitalOcean's built-in monitoring and reporting tools, which provide real-time insights into droplet uptime, CPU load, memory usage, disk performance, and bandwidth activity. Historical incident data were also reviewed to ensure that no unexpected service interruptions occurred during the deployment period.

DigitalOcean's incident history (Figure 6-40) reported no incidents during the months of September, October, or November, confirming the stability of the hosting provider during the testing period. Additionally, the droplet activity log (Figure 6-41) indicated that since its creation two months prior, the server had only undergone a few intentional administrative operations (such as power-offs and resizes) each lasting under two minutes, with no recorded unplanned downtime. The latest operational record showed that the system had been continuously active for over three weeks without interruption.

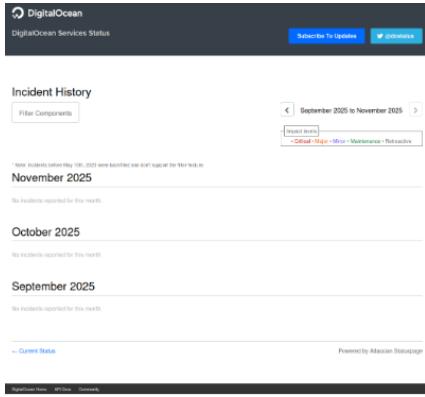


Figure 6-40: DigitalOcean Incident History

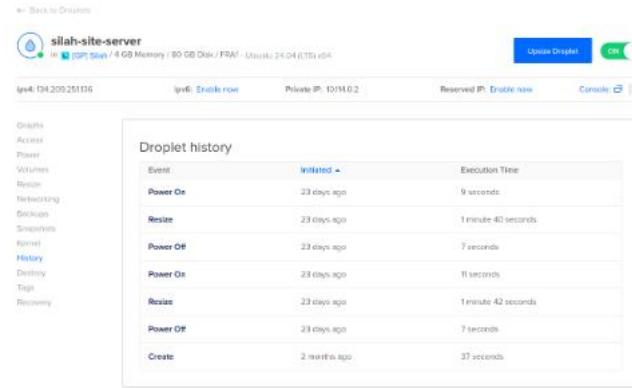


Figure 6-41: Droplet Activity History

Furthermore, resource monitoring graphs for CPU usage, load average, memory utilization, disk I/O, bandwidth, and storage capacity (Figures 6-42, 6-43, and 6-44) showed stable and consistent performance with no indicators of system inactivity or degradation. The combined evidence demonstrates that the deployed instance sustained 100% uptime during the monitoring period, thereby exceeding the availability requirement of 98% uptime and complying fully with the maximum downtime constraint.



Figure 6-42: Monitoring Graphs (CPU)

Figure 6-43: Monitoring Graphs (Load, Memory, Disk I/O)

Figure 6-44: Monitoring Graphs (Disk Usage, Bandwidth)

6.2.4.2 Reliability

Reliability testing aimed to measure the system's stability and its ability to recover from unexpected conditions. This included observing system behavior during simulated interruptions, ensuring that data remained consistent after restarts, and verifying that critical operations could resume without loss of information. The results will be summarized in tables to demonstrate the robustness and fault tolerance of the system.

This feature was implemented using the browser's localStorage API, which automatically saves form drafts in real time as the user types. When the user returns to the page, the system detects the saved draft and pre-fills all fields. Once the user successfully submits the form (or sends a message), the draft is automatically cleared from localStorage to keep the storage clean and up to date.

This functionality was applied to the following parts of the system:

- Sign-up Form
- Supplier's Create an Invoice Form
- Chat (unsent messages for both users) Page
- Buyer's Open a New Bid Form
- Supplier's Participating in Bidding (Write an Offer) Form
- Supplier's Product Details Page
- Supplier's Service Details Page
- Buyer's Write a Review Form

Because the visual appearance of the page is identical before closure and after reopening (the whole point of the feature), it is not meaningful to show separate "before" and "after" screenshots of the UI. Instead, we will show a screenshot of the browser's DevTools → Application → Local Storage panel showing the exact key-value pair that stores the draft for that specific page.

Sign-up Form

All fields are saved continuously. When the user returns, the form is pre-filled exactly as it was left.

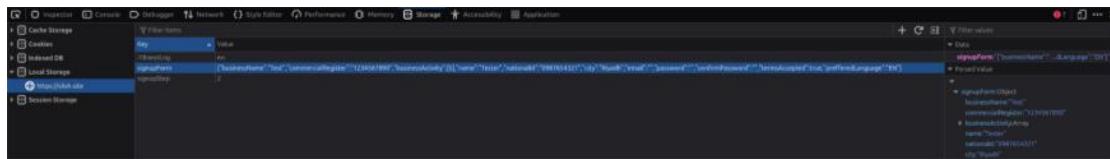


Figure 6-45: Corresponding draft stored in localStorage (keys: signupForm and signupStep)

Supplier's Create an Invoice Form

The invoice draft is saved in real time. On the supplier Chat page, the button changes to “Continue Writing an Invoice” when a draft exists instead of the “Create an Invoice” button.

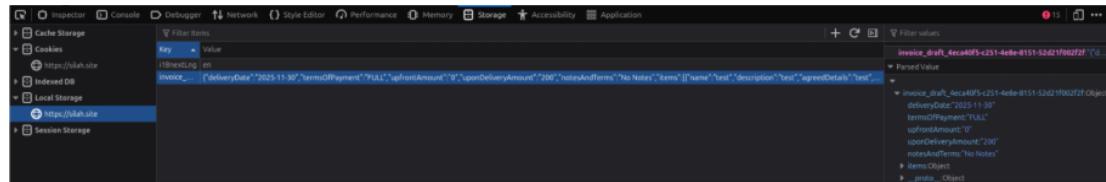


Figure 6-46: Draft in localStorage (key format: invoice_draft_{invoiceId})

Chat (unsent messages for both users) Page

Any text typed in the message input (but not yet sent) is preserved.

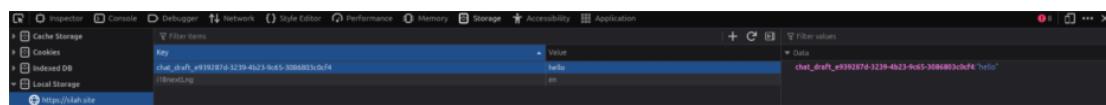


Figure 6-47: Unsent message stored in localStorage (key format: chat_draft_{conversationId})

Buyer's Open a New Bid Form

Simple form fields are saved in real time.

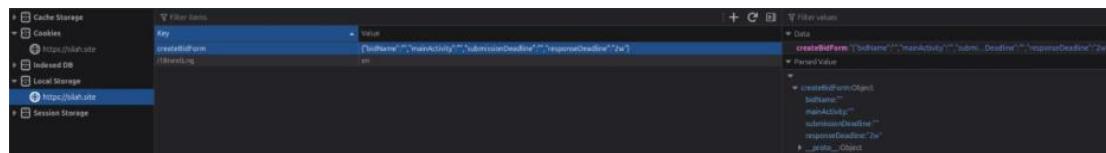


Figure 6-48: Draft in localStorage (key: createBidForm)

Supplier's Participating in Bidding (Write an Offer) Form

Offer amount, completion date, and other feilds are saved instantly.



Figure 6-49: Corresponding entry in localStorage (key: bidOfferDraft)

Supplier's Product Details Page

When creating or editing a product, all fields (title, description, pricing, images metadata, etc.) are preserved.

The screenshot shows the Chrome DevTools Network tab with the Local Storage section selected. A key named "newProductForm" is expanded, showing its value as an object with properties: id, name, description, category, price, currency, baseQTY, minOrderQty, maxOrderQty, unitPrice, groupEnabled, groupMinQty, groupDeadline, and idStr. The "Parsed Value" pane on the right shows the same object structure.

Figure 6-50: Draft in localStorage (key: newProductForm)

Supplier's Service Details Page

It uses the same mechanism as the Product Details page.

The screenshot shows the Chrome DevTools Network tab with the Local Storage section selected. A key named "newServiceForm" is expanded, showing its value as an object with properties: id, name, description, category, price, currency, isPriceNegotiable, status, serviceAvailability, createdAt, and updatedAt. The "Parsed Value" pane on the right shows the same object structure.

Figure 6-51: Draft in localStorage (key: newServiceForm)

Buyer's Write a Review Form

Both the star rating and the written review text are persisted. A “Continue Writing a Review” button appears when a draft exists instead of the “Write a Review” button.

The screenshot shows the Chrome DevTools Network tab with the Local Storage section selected. A key named "review_draft_0fb14305-3c82-4eaa-a690-c5ed6bfef46" is expanded, showing its value as an object with properties: supplierRating, supplierReview, and itemRatings. The "Parsed Value" pane on the right shows the same object structure.

Figure 6-52: Draft in localStorage (key: review_draft_{orderId})

Test Results Summary

Table 6-: Reliability Test Results

Component	Real-time Saving	Restoration on Return	Draft Cleared on Submit	User Guidance Button
Sign-up Form	Yes	Yes	Yes	-
Create an Invoice Form	Yes	Yes	Yes	Continue Writing an Invoice
Chat (unsent messages)	Yes	Yes	Yes (on Send)	-

Open a New Bid Form	Yes	Yes	Yes	-
Participating in Bidding	Yes	Yes	Yes	-
Supplier Product Details	Yes	Yes	Yes (on Save)	-
Supplier Service Details	Yes	Yes	Yes (on Save)	-
Write a Review Form	Yes	Yes	Yes	Continue Writing a Review

In conclusion, the tests confirmed that the system fully meets the reliability requirement across all specified components. Users experience no data loss during unexpected interruptions, and clear visual cues (such as “Continue Writing” buttons) guide them back to their unfinished work. This significantly improves user satisfaction and reduces frustration in real-world usage scenarios.

6.2.4.3 Portability (Cross-Browser Testing)

Portability testing (also known as cross-browser testing) was carried out to ensure that the web application delivers a consistent visual appearance and full functionality across the major modern desktop browsers: Google Chrome, Apple Safari, and Mozilla Firefox.

Firefox compatibility was already extensively verified through the functional testing screenshots presented in Section 6.2.3, and the screenshots presented in Section 5.3. In this round, the testing effort was directed toward Chrome and Safari, as they were the final key browsers on our list.

The testing was performed manually in live interactive sessions using LambdaTest, a cloud-based platform that provides real devices and browsers. This allowed direct navigation through every page of the application on both Chrome (Windows 11) and Safari (macOS Sequoia).

During the sessions, all pages were systematically examined to ensure correct rendering of layout, spacing, and alignment, consistent font styles and text readability, proper scaling of images and icons, and accurate colors and background treatments. The test also included checking the hover, focus, and active states of buttons and links, verifying the visibility and styling of form elements along with validation and error messages, and confirming the functionality of navigation menus, modals, and dropdown menus, and testing both RTL and LTR modes.

Although the entire site was tested, 12 representative pages that cover the main user flows and interface complexity were selected for detailed documentation and screenshot evidence in this report.

Landing Page

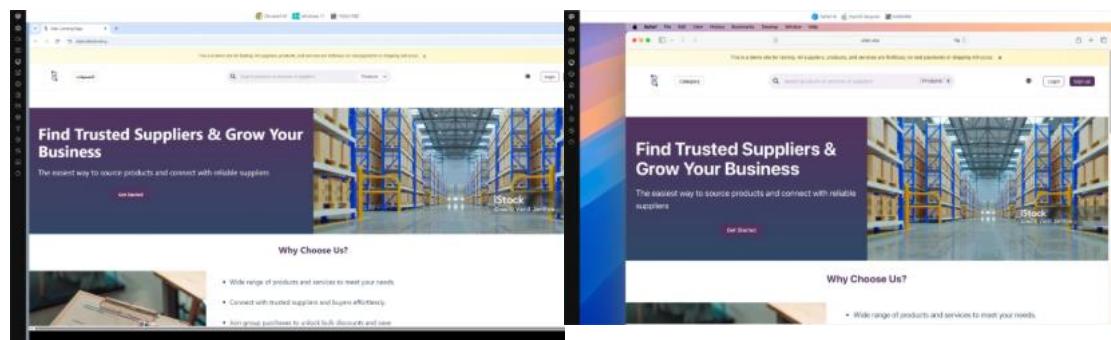


Figure 6-53: Landing page (1/2) On Chrome

Figure 6-54: Landing page (1/2) On Safari

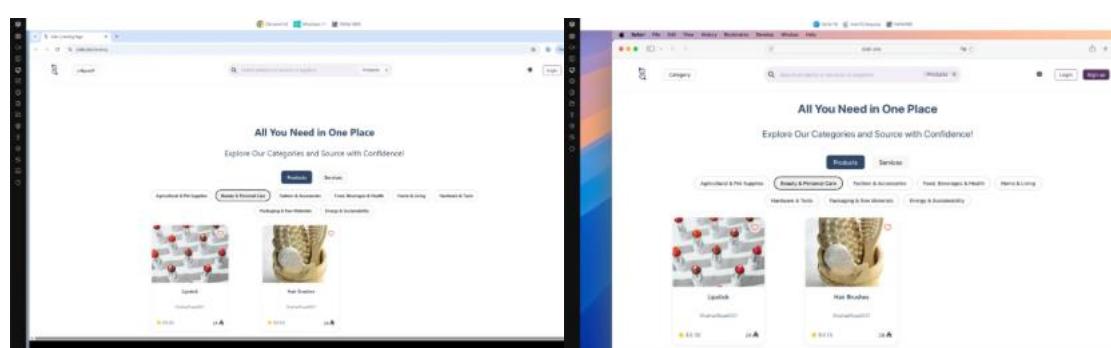


Figure 6-55: Landing page (2/2) On Chrome

Figure 6-56: Landing page (2/2) On Safari

Buyer Homepage

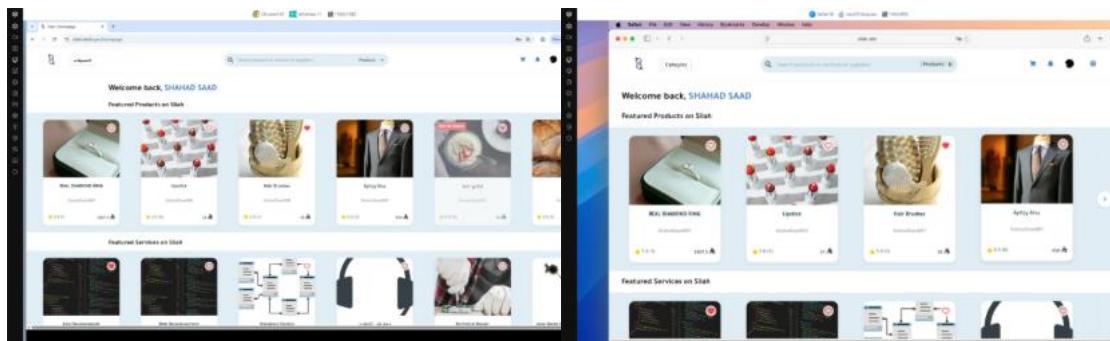


Figure 6-57: Homepage on Chrome

Figure 6-58: Homepage on Safari

Buyer Supplier's Storefront Page

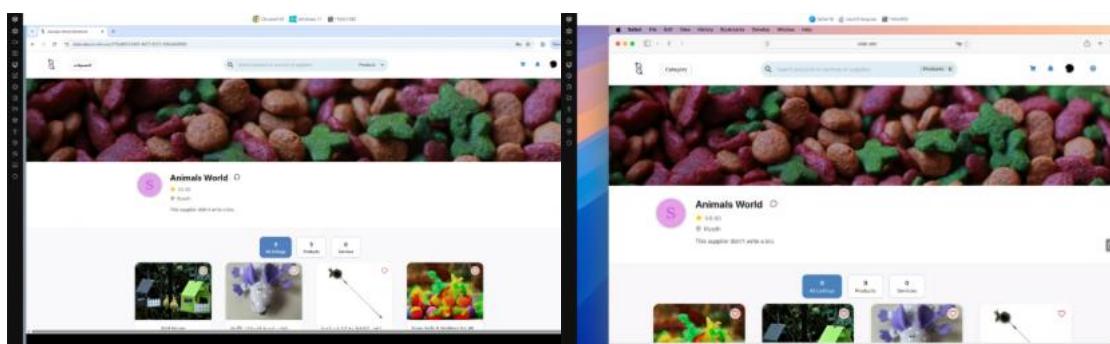


Figure 6-59: Storefront (1/2) On Chrome

Figure 6-60: Storefront (1/2) On Safari

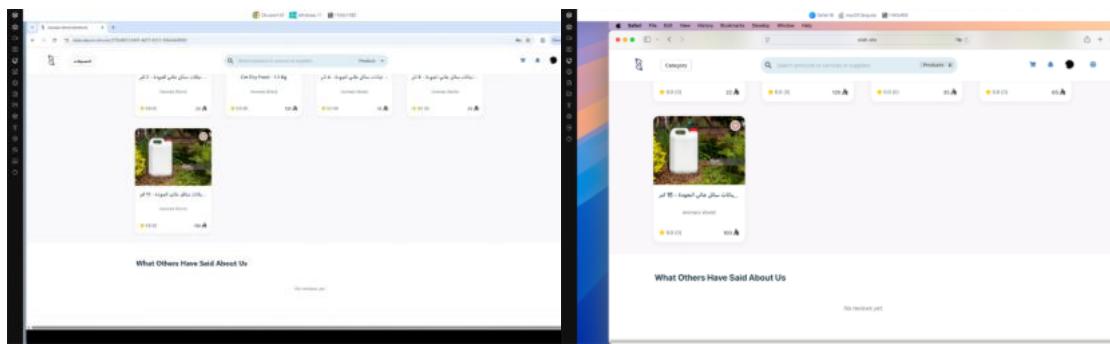


Figure 6-61: Storefront (2/2) On Chrome

Figure 6-62: Storefront (2/2) On Safari

Buyer Product Details Page

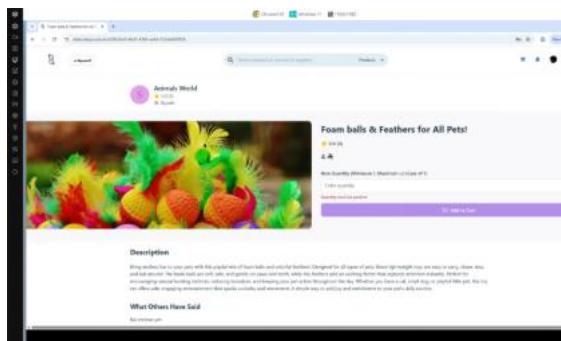


Figure 6-63: Buyer Product Details on Chrome

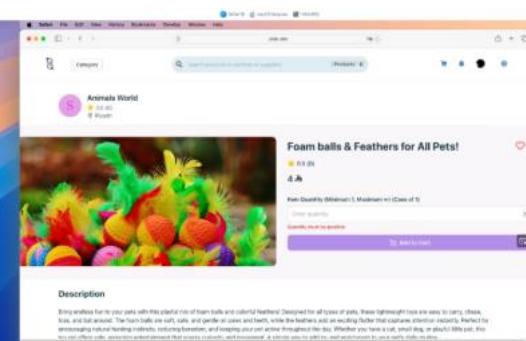


Figure 6-64: Buyer Product Details on Safari

Buyer Cart Page

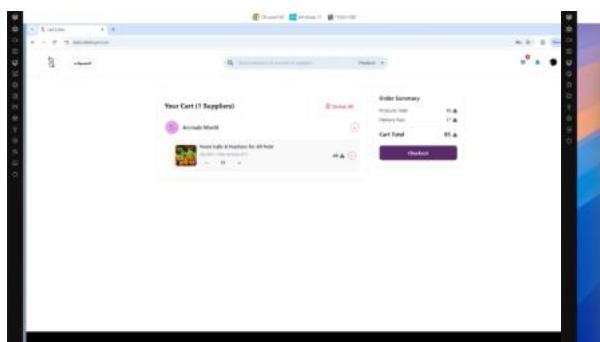


Figure 6-65: Cart on Chrome

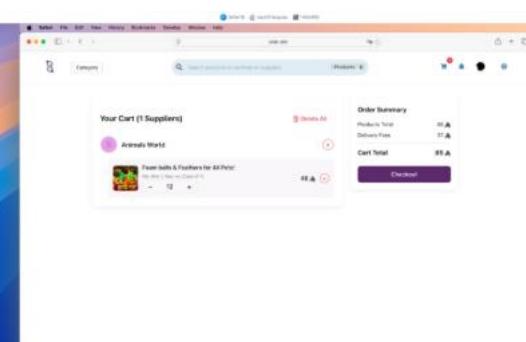


Figure 6-66: Cart on Safari

Buyer Chats Page

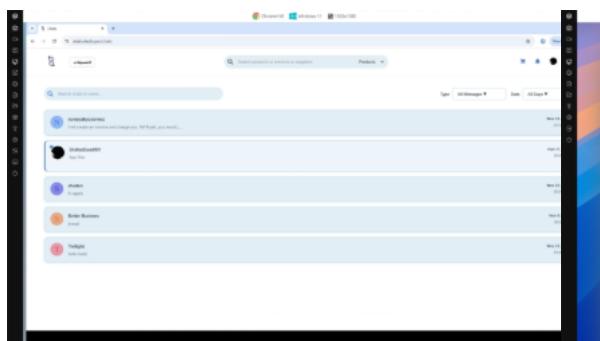


Figure 6-67: Chats on Chrome

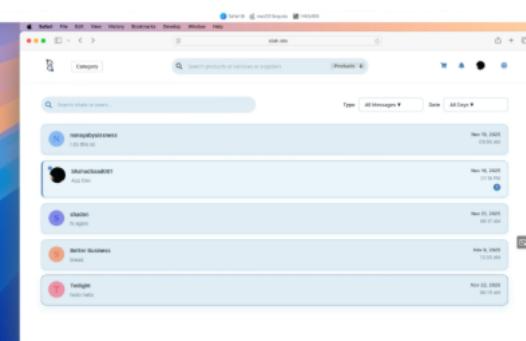


Figure 6-68: Chats on Safari

Buyer Invoices Page

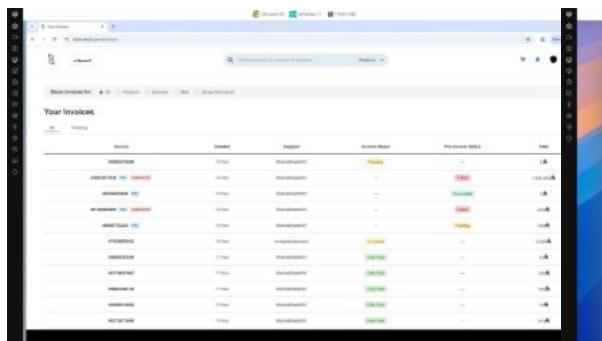


Figure 6-69: Invoices on Chrome

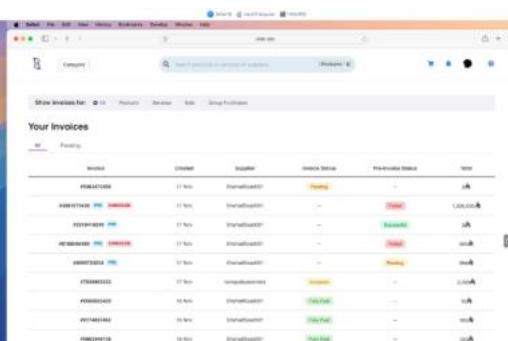


Figure 6-70: Invoices on Safari

Supplier Overview

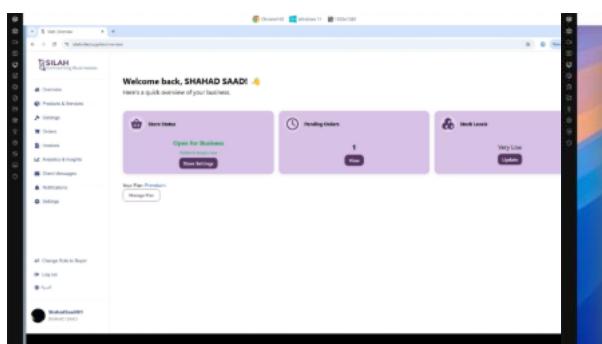


Figure 6-71: Overview on Chrome

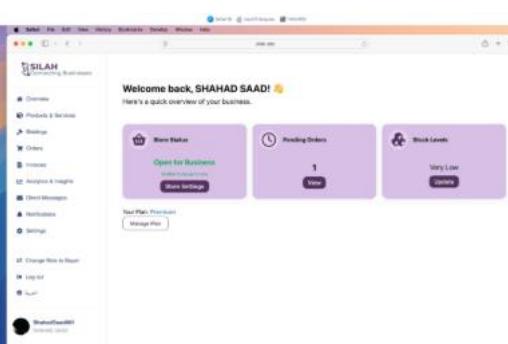


Figure 6-72: Overview on Safari

Supplier Listings Page

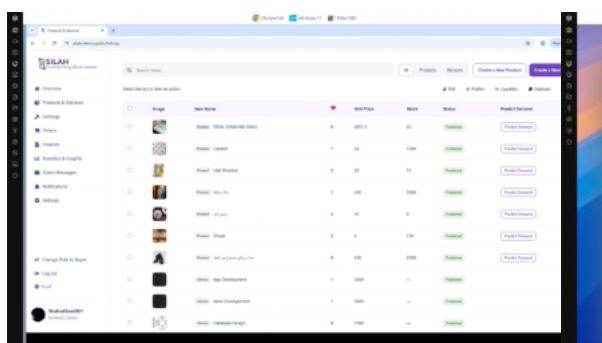


Figure 6-73: Listings on Chrome

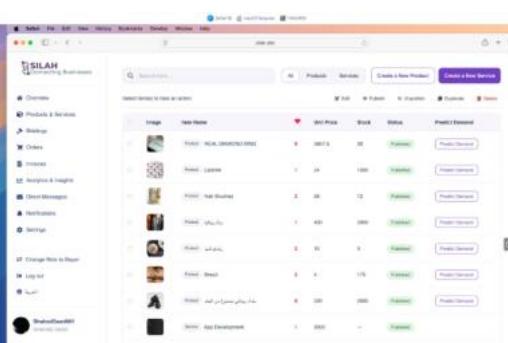


Figure 6-74: Listings on Safari

Supplier Create a Service Page

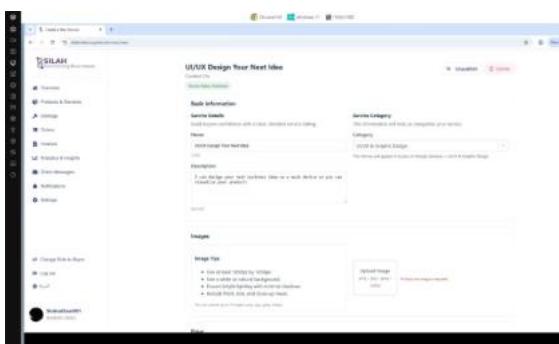


Figure 6-75: Create a Service on Chrome

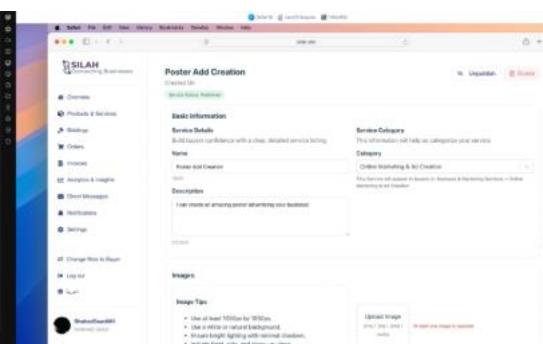


Figure 6-76: Create a Service on Safari

Supplier Create an Invoice Page

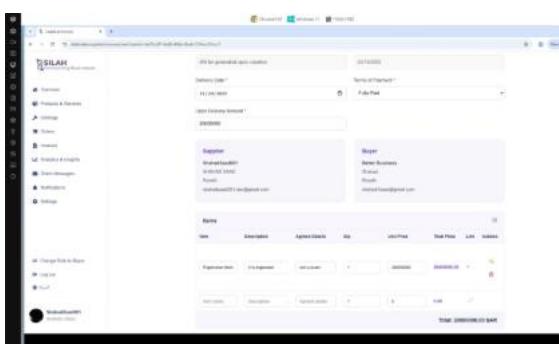


Figure 6-77: Create an Invoice on Chrome

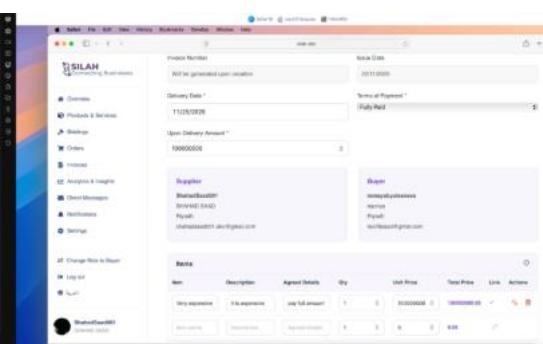


Figure 6-78: Create an Invoice on Safari

Supplier Orders Page

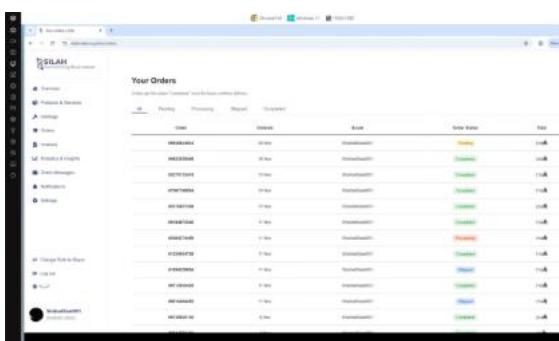


Figure 6-79: Orders on Chrome

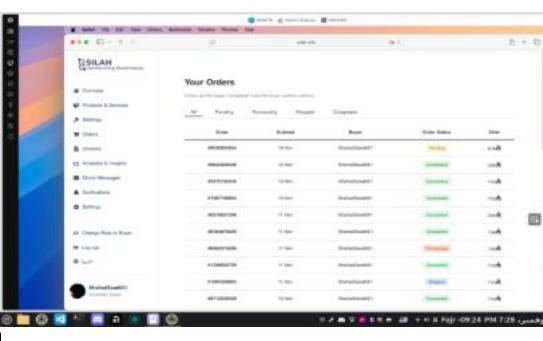


Figure 6-80: Orders on Safari

Test Results Summary

Table 6-: Portability Test Results

Browser	Pages Tested	Visual Differences	Functional Issues	Overall Result
Chrome	Entire site	None	None	Fully compatible
Safari	Entire site	None	None	Fully compatible
Firefox	Entire site	None	None	Fully compatible

The site demonstrates excellent cross-browser compatibility across all major modern desktop browsers. Layout, styling, interactivity, and functionality are consistent with no discrepancies that would affect the user's experience. This confirms that the implementation follows current web standards effectively and requires no browser-specific workarounds.

6.2.4.4 Usability

Usability testing focused on evaluating how easily participants could navigate and interact with the system's interface. We used an observable assessment method called the Think-Aloud technique, where participants performed specific tasks while verbalizing their thoughts, expectations, and difficulties. Feedback and observations were later compiled to identify areas for enhancement. The target satisfaction score was set at 70% or higher.

Table 6-5 presents the aggregated results across all ten participants, including average completion time, total errors, and task success rate (percentage of participants who completed the task without any errors).

Table 6-: Aggregated Usability Testing Results (10 participants & 20 tasks)

Task Name	Average Time	Total Errors (out of 10)	Success Rate %
Sign up	2 min 9 s	1	90 %

Add a payment method	1 min 8 s	1	90 %
Start/Join a group purchase	42 s	3	70 %
View Invoices history	21 s	3	70 %
Add a product to the cart	12 s	1	90 %
Pay for cart	32 s	1	90 %
Check order status	19 s	0	100 %
Write a review	51 s	2	80 %
Smart search a product	20 s	0	100 %
Contact a supplier	20 s	0	100 %
View invoice details	26 s	0	100 %
Pay for invoice	36 s	0	100 %
Create a new bid	47 s	3	70 %
Switch role to supplier	10 s	0	100 %
Add a new product/service	2 min 7 s	1	90 %
Open the store	46 s	1	90 %
Create an invoice	1 min 49 s	2	80 %
View notifications	8 s	0	100 %

Join a bid	47 s	1	90 %
View received orders	25 s	0	100 %
Overall	N/A	20 / 200 tasks	90 %

The platform achieved an overall task success rate of 90%, exceeding the predefined target of 70%. Seventeen out of twenty tasks achieved 90–100% success, demonstrating excellent usability in core flows such as searching for products, contacting suppliers, checking order status, paying invoices, viewing notifications, and switching roles.

Only three tasks fell to 70% (Start/Join a group purchase, View Invoices history, Create a new bid), primarily due to minor discoverability issues that did not prevent completion.

Most of the qualitative feedback was overwhelmingly positive: participants repeatedly described the interface as “beautiful”, “very easy”, “extremely fast”, “intuitive”, “professional”, and “enjoyable”. Many explicitly stated they loved the design and were impressed by features such as smart search speed, creating a new product or service, and the flexibility of the sign-up process. The few points of confusion that emerged were limited to the following areas:

- Some participants did not initially realize that every registered buyer can instantly become a supplier without creating a new account.
- Several users were unsure about the exact review rules (reviews are only possible on fully paid invoices or on orders whose status has been marked as “Completed”).
- A few participants found invoice-related fields scattered and requested clearer visual grouping on both the “Create Invoice” and “View Invoice Details” pages.

- When entering unit prices on the invoice creation form, the field starts with a “0” that is not automatically cleared on focus, which accidentally added extra zeros to the price.
- Minor frustration occurred with the Settings page when success and error messages appeared only at the very top, requiring scrolling to see them.

Based on the testing results, the following targeted improvements are recommended:

- After registration, display a one-time modal clearly stating: “You are registered as a buyer. You can open your supplier store and start selling anytime.”
- Show “Write Review” buttons directly on eligible orders and invoices, and immediately after successful payment or delivery confirmation, display a friendly message explaining the review rule (e.g., “Order placed! Once you receive the items and mark them as delivered, you’ll be able to leave a review.” or “Invoice fully paid! Please leave a review once you receive the items.”).
- Group related fields on invoice creation and detail pages into clearly labelled cards or sections and auto-clear placeholder zeros from numeric inputs.
- Add a small “Group Purchase Available” badge on product cards, show a visible success toast (“Store is now open!” or “Settings saved”) after saving changes, and display relative time for bid deadlines (e.g., “3 days 5 hours left”) instead of absolute dates only.

Detailed individual performance tables for all ten participants are provided in Appendix E.

With an overall success rate of 90% and overwhelmingly positive qualitative feedback, the platform proves to be highly usable, intuitive, and user-friendly. The testing confirms that both buyer and supplier journeys are efficient and enjoyable, comfortably surpassing the original 70% target and reaching a level typical of mature, polished commercial platforms. The few remaining friction points are minor and can be addressed quickly in the future.

6.3 Summary of Test Results

The testing phase covered four major categories: automated backend testing (unit and integration), functional testing, non-functional testing, and usability testing. Table 6-6 below presents an overview of the methods used and the results achieved.

Table 6-: Overall Summary of Test Results

Testing Type	Tool or Method Used	Scope	Total Test Cases	Success Rate
Unit	Jest	NestJS controllers and services	96	100 %
Integration	Jest	NestJS services and database integration	5	100 %
Functional	Manual testing	All website	N/A	100 %
Availability	DigitalOcean monitoring	Continuous uptime monitoring	N/A	100 %
Reliability	Manual testing	Draft recovery on critical pages	8 pages	100 %
Portability	LambdaTest	Chrome, Firefox, and Safari	N/A	100 %
Usability	Think-Aloud	10 participants & 20 tasks	200 tasks	90 %

The system successfully passed all functional, availability, reliability, and portability tests with 100% success. Usability testing delivered an outstanding 90% task success rate and extremely positive qualitative feedback, far exceeding the original target of 70%.

These outcomes demonstrate that the system fully meets its functional and non-functional requirements and delivers a mature, production-ready user experience. A complete evaluation of the project is provided in the next chapter.

Chapter 7 Conclusion

This final chapter brings together the outcomes of the Silah project, evaluating the platform in relation to the aims and goals established at the start of the journey. It reflects on the system's overall performance, the effectiveness of its technical and business features, and the extent to which it addresses the challenges of digital procurement in the Saudi market. The chapter highlights the project's achievements, acknowledges its limitations, and maps each objective to the tangible results delivered throughout development.

In addition to evaluating the system, this chapter outlines potential future enhancements that could further strengthen Silah's functionality, intelligence, and operational value if additional development time were available. Finally, we conclude with personal reflections that mark the end of our journey with Silah; summarizing the lessons learned, the challenges overcome, and the growth experienced throughout the process.

7.1 Evaluation

The Silah platform has successfully achieved its core objectives as a bilingual B2B marketplace connecting buyers and suppliers in Saudi Arabia, directly tackling the challenges of inefficient procurement and low digital adoption in the local market. All key goals from the project proposal were met, including the development of a fully functional web platform, an e-bidding system for RFPs and proposals, group purchasing for collective negotiations, and an AI-powered alternative product recommendation system. The platform supports real-time communication, inventory management, and demand forecasting, providing a transparent and efficient ecosystem that aligns with Saudi Vision 2030's emphasis on digital transformation and SME empowerment.

From a technical standpoint, the system demonstrates a solid full-stack implementation using React for the frontend and NestJS for the backend, deployed on DigitalOcean with Prisma ORM for database management and WebSockets and SSE for live interactions. The early deployment strategy allowed for continuous testing against live APIs, ensuring seamless integration. Testing results further validate the platform's readiness: functional, availability, reliability, and portability tests achieved 100% success, while usability evaluation with 10 real users across 200 tasks yielded a

90% success rate with participants describing the interface as “intuitive,” “beautiful,” and “professional”.

Achievements

The project delivered tangible value through innovative features and practical engineering:

- **User-centric design:** The platform offers a highly intuitive experience tailored for business users, with seamless workflows like one-click role switching between buyer and supplier modes, real-time notifications for order updates and chat messages, and a unified dashboard that adapts dynamically to user roles without page reloads.
- **AI-driven efficiency:** Integration of fine-tuned AI models for smart search, demand forecasting, and alternative product suggestions provides actionable insights, such as predicting stock needs based on historical data.
- **Real-time collaboration:** WebSocket-powered chat and SSE for live updates (e.g., bid notifications or group purchase progress) creates a responsive environment that feels like a modern app.
- **Bilingual inclusivity:** Full RTL/LTR support with dynamic language switching ensures accessibility for Arabic-speaking users, including localized validation messages and culturally adapted layouts (e.g., right-aligned forms for invoices).
- **Deployment resilience:** Cloud hosting with CI/CD pipelines and draft recovery on critical pages (e.g., bids, listings) minimizes data loss, making the system reliable for daily business use.

Limitations

While the platform is production-ready for a prototype, several limitations arose from time constraints and scope priorities, focusing on core functionality over advanced refinements:

- **Deletion and data management:** Users cannot delete accounts or associated media (e.g., product images in R2 storage), limiting privacy options; this was a

deliberate simplification to avoid complex cascading deletes and ensure auditability for invoices and bids.

- **Performance optimization:** No caching layers (e.g., Redis) or lazy loading were implemented, which could slow queries on large product catalogs; this trade-off prioritized rapid feature delivery over premature scaling, as load times remain under 2 seconds for typical use.
- **Frontend testing absence:** No automated unit or integration tests were added for React components, relying solely on manual verification; this increases regression risk but was planned from the start to allocate time to UX polish.
- **Accessibility features:** The system does not include specialized support for visually or hearing-impaired users (e.g., screen-reader optimizations beyond basic semantic HTML, high-contrast modes, or keyboard-only navigation enhancements).
- **Theme support:** Only light mode is implemented. Dark mode was excluded to avoid additional design and testing overhead.

These are not critical flaws but expected outcomes of an academic timeline, where delivering a working system outweighed exhaustive refinements. They do not impact core usability, as confirmed by the 90% success rate.

Alignment with Aims & Goals

The project successfully delivered on every single aim and goal that was set at the very beginning. The table below maps each one exactly to how it was actually achieved.

Table 7-1: Achievement of Project Aims and Goals

Aim or Goal	How It Was Achieved
Understanding and applying full-stack web development, covering both front-end and back-end technologies, as well as core web development languages.	Built the complete system using React + Vite for the frontend and NestJS + Prisma for the backend, with WebSockets, SSE, and full REST API integration.

Exploring B2B business models and market structures to understand how to build an effective intermediary platform.	Designed and implemented group purchasing, e-bidding, supplier subscriptions, role-based dashboards, and real-time buyer-supplier communication.
Applying architecture patterns and design patterns to improve performance and maintenance.	Applied modular architecture in NestJS with dependency injection and singleton pattern for shared services, React Context as global state, dynamic route loading to reduce bundle size, and separate buyer/supplier route trees for clearer maintenance.
Implementing an AI-powered alternative recommendation system to suggest alternative products, while learning how to prepare data and train models using fine-tuning techniques.	Fine-tuned a LaBSE model for smart search and alternative suggestions (Alternatives page), and fine-tuned Facebook Prophet for demand forecasting using real procurement data.
Managing databases efficiently using SQL to ensure effective data storage and analysis.	Designed and managed a PostgreSQL database via Prisma with soft deletes and complex relationships.
Exploring cloud deployment and web hosting techniques to ensure the platform is operational and secure.	Deployed the full stack on a DigitalOcean Droplet, set up CI/CD with GitHub Actions, used Cloudflare R2 for secure media storage, and configured NGINX as reverse proxy.
Enhancing knowledge of procurement systems and e-Bidding management to optimize digital purchasing processes.	Implemented the complete e-bidding flow, automated bid and group-purchase deadline handling using cron jobs, and full order/invoice status tracking.

Developing a fully functional web platform that acts as an intermediary between businesses.	Delivered a live, bilingual platform used by real testers with all core buyers and suppliers' journeys working end-to-end.
Building an e-Bidding system that enables companies to submit RFPs and receive bids in an organized and transparent manner.	Created the full bidding module with deadlines, supplier offers, acceptance flow, and automatic invoice generation.
Implementing a group purchasing feature to allow buyers to negotiate better deals collectively.	Built group purchase creation, joining, real-time participant counter, automated success/failure handling, and final order splitting.
Developing an “Alternative-Product” recommendation system to help businesses discover competitive alternatives to well-known products.	Integrated AI-powered alternative suggestions displayed on the dedicated Alternatives page.

Every aim was addressed through hands-on implementation, fostering deep technical growth while producing a practical solution.

In summary, Silah exceeds expectations as a cohesive, innovative B2B platform that solves real procurement and serves as a strong foundation for expansion. The high testing scores and positive user feedback affirm their viability, while the acknowledged limitations highlight opportunities for iteration. This project underscores the potential of student-led development to contribute meaningfully to Saudi Arabia's digital economy.

7.2 Future Work

If given another six months, several important improvements could be made to enhance the system, both functionally and technically. Each suggested feature below contributes directly to the system's main goals; improving usability, trust, automation, and data-driven insights between suppliers and buyers.

Intelligent Features

- Product/Service Recommendations Before Bidding

Suggest relevant items or services to buyers before they create a bid, reducing waiting time for proposals and helping them discover available options instantly.
- Supplier Ranking Page

Display top suppliers for each product category, allowing buyers to make informed choices and promote visibility and competition among suppliers.
- AI-Based Group Purchase Suggestions

Use an AI model to suggest forming groups among buyers with similar needs, purchase habits, or categories; improving efficiency and pricing for bulk orders.
- Recommendation Section on Product and Service Details Pages

Display recommended or similar products/services after the reviews section, enhancing the browsing experience and cross-selling potential.

Operational and Business Enhancements

- Real Payment Integration

Make the subscription feature real through integration with Tap, supporting payments, auto-renewal, and cancellation flows.
- Invoice Sharing in Chat

Enable suppliers to send invoices directly through the chat, ensuring faster communication and transaction tracking.
- Deletion of Accounts and Media

Implement true deletion of user accounts and associated media files (from R2 bucket), ensuring data privacy and storage optimization.
- Cancellations and Refunds Management

Add cancellation flows for orders, group purchases, invoices, and subscriptions.

- Follow Suppliers
Allow buyers to follow specific suppliers to strengthen business relationships and receive updates.
- Email Notifications and Invoices
Add automated email notifications for important actions (order status changes, payment updates, invoices, etc.).

Supplier Experience & Data Management

- Product Import/Export
 - Upload products via .csv files or through APIs of other stores (e.g., Salla).
 - Export sales and reviews data via .csv for integration with external CRM tools.
- Enhanced CRM System
 - Provide built-in customer relationship management tools.
 - Allow suppliers to see returning customers, total profit per customer, and order histories.
 - Add notes for customers to track preferences and behaviors.
 - Allow export of customer and order data for analysis.
- Supplier Analytics and Preview
 - Let suppliers preview their storefront and product/service details pages from a buyer's perspective.
 - Display analytics like customer activity, cart items, and purchase behavior insights.

Technical & Documentation Improvements

- Refactoring & Code Optimization
Improve maintainability by refactoring modules developed in early stages.

- Comprehensive Documentation
Add technical documentation, business documentation, and user manuals.
- Extensive Testing
Implement unit and integration tests for both frontend and backend.
- Enhanced Demand Prediction
Extend the AI demand forecasting to consider similar products, supplier visibility, and other contextual factors.
- Business Model Revision
Reassess the subscription pricing (e.g., 50 SAR) to ensure profitability while maintaining fairness for suppliers.

These enhancements, prioritized by their potential impact and return on investment (such as implementing AI-driven recommendations for immediate user value), would transform Silah from a proof-of-concept into a competitive marketplace; directly amplifying its mission of fostering AI-driven efficiency, transparency, and trusted B2B connections.

Final Words

الحمد لله أولاً وأخراً، وبفضل الله تمت رحلتنا مع صلة، المشروع الذي كان فكرة صغيرة وأصبح ثمرة جهد وتعاون وتعلم طويل.

This report marks the completion of our journey with Silah, a journey that taught us more than just code or design. It taught us patience, teamwork, and the beauty of turning ideas into reality through persistence and trust in Allah's plan.

Every step of this project carried its own story: the challenges that shaped us, the long nights that strengthened us, and the lessons that stayed with us.

We thank Allah for giving us the strength, clarity, and perseverance to complete this journey. And we thank everyone who supported us (our families, our supervisors, our colleagues, and especially each other) for their kindness, effort, and encouragement.

Silah will always be more than a system we built; it reflects who we became through the process; learners, developers, and dreamers who grew with every challenge.

With this, we close our report (and our journey with Silah) carrying gratitude, pride, and hope for what lies ahead.

الحمد لله على التمام، وعلى ما أنعم به من توفيق. الحمد لله حمدًا كثيراً طيباً مباركاً فيه.

- The Silah Team

References

- [1] J. Alzahrani, "The impact of e-commerce adoption on business strategy in Saudi Arabia small and medium enterprises," *Review of Economics and Political Science*, 2018.
- [2] U. N. C. o. T. a. D. (UNCTAD), "Digital Economy Report 2021: Cross-border data flows and development: For whom the data flow," United Nations, 2021.
- [3] P. Verma, "Transforming Supply Chains Through AI: Demand Forecasting, Inventory Management, and Dynamic Optimization," 2024. [Online]. Available:
https://www.researchgate.net/publication/385098771_Transforming_Supply_Chains_Through_AI_Demand_Forecasting_Inventory_Management_and_Dynamic_Optimization.
- [4] M. Intelligence, "Saudi Arabia E-Commerce Market - Growth, Trends, and Forecasts (2025 - 2030)," Mordor Intelligence, 2024. [Online]. Available:
<https://www.mordorintelligence.com/industry-reports/saudi-arabia-ecommerce-market>.
- [5] Investopedia, "Business," Investopedia Dictionary, 2024. [Online]. Available:
<https://www.investopedia.com/terms/b/business.asp>.
- [6] V. K. a. G. Raheja, "BUSINESS TO BUSINESS (B2B) AND BUSINESS TO CONSUMER (B2C) MANAGEMENT," *International Journal of Computer Applications*, 2012. [Online]. Available:
<https://citeseerx.ist.psu.edu/document?doi=ac36aa8328563f01ea924b45966691c46602062c&repid=rep1&type=pdf>.
- [7] K. R. a. L. Pilelienė, "Principle Differences between B2B and B2C Marketing Communication Processes," 2019. [Online]. Available:

- https://www.researchgate.net/publication/336973845_Principle_Differences_between_B2B_and_B2C_Marketing_Communication_Processes.
- [8] "B2B (business to business)," TechTarget, 2023. [Online]. Available: <https://www.techtarget.com/searchcio/definition/B2B>.
- [9] M. E. I. M. T. A. Md. Rasidul Islam, "Supply Chain Management and Logistics: How Important Interconnection Is for Business Success," Scientific Research, 2023. [Online]. Available: <https://www.scirp.org/journal/paperinformation?paperid=128009>.
- [10] S. S. a. S. Serdar Asan, "Dealing with Complexity in the Supply Chain: The Effect of Supply Chain Management Initiatives," 2012. [Online]. Available: https://www.researchgate.net/publication/256019231_Dealing_with_Complexity_in_the_Supply_Chain_The_Effect_of_Supply_Chain_Management_Initiatives.
- [11] L. X. L. a. J. M. Swaminathan, "Supply Chain Management," 2015. [Online]. Available: https://www.researchgate.net/publication/304194361_Supply_Chain_Management.
- [12] "What is HTTP?," GeeksforGeeks, 2024. [Online]. Available: <https://www.geeksforgeeks.org/what-is-http/>.
- [13] Scaler, "What is WebSocket?," Scaler Topics, 2024. [Online]. Available: <https://www.scaler.in/websocket/>.
- [14] M. Jovanović, "What Is a Modular Monolith?," 2024. [Online]. Available: <https://www.milanjovanovic.tech/blog/what-is-a-modular-monolith>.
- [15] B. M. Skalický, "Analysis and Comparison of Application Architecture: Monolith, Microservices, and Modular Approach," 2024. [Online]. Available: <https://raw.githubusercontent.com/founek2/diploma-thesis/master/thesis.pdf>.

- [16] B. L. Sean J Taylor, "Forecasting at scale," 2017. [Online]. Available: <https://peerj.com/preprints/3190v2/>.
- [17] S. Kanankearachchi, "Time series Forecasting in Machine Learning," Medium, 2017. [Online]. Available: <https://engineering.99x.io/time-series-forecasting-in-machine-learning-3972f7a7a467>.
- [18] S. W. Y. L. ,. Y. ,. ,. L. Cheng Yang, "CoRe: An Efficient Coarse-refined Training Framework for BERT," 2020. [Online]. Available: https://www.researchgate.net/publication/346475867_CoRe_An_Efficient_Coarse-refined_Training_Framework_for_BERT.
- [19] Y. Y. D. C. N. A. W. W. Fangxiaoyu Feng, "Language-agnostic BERT Sentence Embedding," Google AI, 2022. [Online]. Available: <https://arxiv.org/pdf/2007.01852>.
- [20] F. Karabiber, "Cosine Similarity," [Online]. Available: <https://www.learndatasci.com/glossary/cosine-similarity/>.
- [21] D. A. D. Gonçalves, "Development of a Marketing Automation Platform to Integrate Online E-Commerce Services," 2022. [Online]. Available: <https://repositorium.sdum.uminho.pt/handle/1822/80045>.
- [22] F. D. D. P. Mario Angos Mediavilla, "Review and analysis of artificial intelligence methods for demand forecasting in supply chain management," 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2212827122004036>.
- [23] J. Feizabadi, "Machine learning demand forecasting and supply chain performance," 2020. [Online]. Available: <https://www.tandfonline.com/doi/full/10.1080/13675567.2020.1803246>.
- [24] U. S. G V Radhakrishnan, "Predictive Analytics in Supply Chain Management: The Role of AI and Machine Learning in Demand Forecasting," 2024. [Online]. Available: <https://jier.org/index.php/journal/article/view/1879>.

- [25] I. J. M. M. S. L. T. R. D. I. Benjamin Rolf, "A review on reinforcement learning algorithms and applications in supply chain management," 2022. [Online]. Available: <https://www.tandfonline.com/doi/full/10.1080/00207543.2022.2140221>.
- [26] T. F. P. B. F. D. Mehran Nasseri, "Applying Machine Learning in Retail Demand Prediction—A Comparison of Tree-Based Ensembles and Long Short-Term Memory-Based Deep Learning," 2023. [Online]. Available: <https://www.mdpi.com/2076-3417/13/19/11112>.
- [27] M. E. G. M. L. Fayçal Messaoudi, "Demand Prediction Using Sequential Deep Learning Model," 2023. [Online]. Available: https://www.researchgate.net/publication/373502439_Demand_Prediction_Using_Sequential_Deep_Learning_Model.
- [28] H. Y. X. W. T. L. Z. L. T. W. B. Z. Jiatu Shi, "Relation-aware Meta-learning for E-commerce Market Segment Demand Prediction with Limited Records," 2021. [Online]. Available: <https://arxiv.org/pdf/2008.00181>.
- [29] V. Chowdhury, "Superstore Dataset," Kaggle, 2021. [Online]. Available: <https://www.kaggle.com/datasets/vivek468/superstore-dataset-final/data>.
- [30] D. ... U. K. Hariprasath Gnanasekaran, "Time-series Forecasting of Web Traffic Using Prophet Machine Learning Model," 2023. [Online]. Available: https://www.researchgate.net/publication/376596105_Time-series_Forecasting_of_Web_Traffic_Using_Prophet_Machine_Learning_Model.
- [31] K. J, "Amazon Sales Dataset," Kaggle, 2023. [Online]. Available: <https://www.kaggle.com/datasets/karkavelrajaj/amazon-sales-dataset/data>.
- [32] M. K. M. G. Anna Glazkova, "Fine-tuning of Pre-trained Transformers for Hate, Offensive, and Profane Content Detection in English and Marathi," 2021. [Online]. Available: <https://arxiv.org/pdf/2110.12687>.

- [33] J. Atwood, "UI is Hard," 2005, [Online]. Available: <https://blog.codinghorror.com/ui-is-hard/>.
- [34] J. Atwood, "UI-First Software Development," 2008. [Online]. Available: <https://blog.codinghorror.com/ui-first-software-development/>.
- [35] E. Yipis, "Developers want more, more, more: the 2024 results from Stack Overflow's Annual Developer Survey," Stack Overflow, 2025. [Online]. Available: <https://stackoverflow.blog/2025/01/01/developers-want-more-more-more-the-2024-results-from-stack-overflow-s-annual-developer-survey/>.
- [36] "REST API vs GraphQL API vs gRPC API," GeeksforGeeks, 2023. [Online]. Available: <https://www.geeksforgeeks.org/system-design/rest-api-vs-graphql-api-vs-grpc-api/>.
- [37] S. Omojola, "GraphQL vs. gRPC vs. REST: Choosing the right API," LogRocket, 2022. [Online]. Available: <https://blog.logrocket.com/graphql-vs-grpc-vs-rest-choosing-right-api/>.
- [38] S. Olusola, "Server-sent events vs. WebSockets," LogRocket, 2022. [Online]. Available: <https://blog.logrocket.com/server-sent-events-vs-websockets/>.
- [39] DeepL, "Unbeatable translation quality for global business," DeepL, [Online]. Available: <https://www.deepl.com/en/quality>.
- [40] S. R. James Robertson, "Volere Requirements Specification Template," 2012. [Online]. Available: <https://www.cs.uic.edu/~i440/VolereMaterials/templateArchive16/c%20Volere%20template16.pdf>.
- [41] P. Gonzalez, "Design Patterns for Salesforce Git Branching Strategies," 2023. [Online]. Available: <https://www.pablogonzalez.io/salesforce-git-branching-strategies/>.
- [42] R. Sahoo, "Superstore Sales Dataset," Kaggle, 2020. [Online]. Available: <https://www.kaggle.com/datasets/rohitsahoo/sales-forecasting>.

- [43] H. K. P. T. Yanrui Ning, "A comparative machine learning study for time series oil production forecasting: ARIMA, LSTM, and Prophet," 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/S009830042200084X>.

Appendix

Appendix A: Survey Questions

Table A - 1: Survey Questions – All Sections

Survey Question	Possible Responses
Basic Information Section:	
What industry does your business operate in?	<ul style="list-style-type: none"> <input type="checkbox"/> Manufacturing & Production <input type="checkbox"/> Wholesale & Trade <input type="checkbox"/> Retail & E-commerce <input type="checkbox"/> Logistics & Supply Chain <input type="checkbox"/> Energy & Utilities <input type="checkbox"/> Finance & Banking <input type="checkbox"/> Technology & Telecommunications <input type="checkbox"/> Healthcare & Pharmaceuticals <input type="checkbox"/> Agriculture & Food <input type="checkbox"/> Media & Entertainment <input type="checkbox"/> Travel & Hospitality <input type="checkbox"/> Education & Training <input type="checkbox"/> Other (Specify)
What is your business role in the B2B marketplace?	<ul style="list-style-type: none"> <input type="radio"/> Supplier (We sell products to other businesses) <input type="radio"/> Buyer (We purchase products from suppliers) <input type="radio"/> Both (We act as both a supplier and a buyer) <input type="radio"/> Service Provider (We offer services for other business)
How often does your business buy or sell products/services?	<ul style="list-style-type: none"> <input type="radio"/> Daily <input type="radio"/> Weekly <input type="radio"/> Monthly <input type="radio"/> Quarterly <input type="radio"/> Annually
What are the biggest challenges your business faces in purchasing products or outsourcing services?	<ul style="list-style-type: none"> <input type="checkbox"/> Difficulty finding suitable suppliers or service providers <input type="checkbox"/> High costs and lack of bulk discounts <input type="checkbox"/> Slow response time from suppliers/buyers <input type="checkbox"/> Lack of transparency in pricing and deals <input type="checkbox"/> Limited access to alternative product options <input type="checkbox"/> Other (Specify)
How does your business currently handle procurement?	<ul style="list-style-type: none"> <input type="checkbox"/> Manually (phone calls, emails, in-person deals)

	<ul style="list-style-type: none"> <input type="checkbox"/> Through an Online B2B marketplace (e.g., Forsah, Etimad, Alibaba) <input type="checkbox"/> Through direct supplier/buyer partnerships <input type="checkbox"/> Other (Specify)
Would your business consider shifting from traditional procurement methods to a digital platform?	<ul style="list-style-type: none"> <input type="radio"/> We are already using digital platforms <input type="radio"/> Yes, we are actively looking for a solution <input type="radio"/> Maybe, if it improves efficiency and reduces costs <input type="radio"/> No, we prefer traditional procurement methods
Which features do you think would be most useful in a B2B procurement platform?	<ul style="list-style-type: none"> <input type="checkbox"/> E-Bidding system <input type="checkbox"/> Similar product discovery <input type="checkbox"/> Group purchasing to get bulk discounts <input type="checkbox"/> Real-time messaging between suppliers and buyers <input type="checkbox"/> Other (Specify)
Supplier-Focused Section:	
What are the biggest challenges your company faces when selling to businesses?	<ul style="list-style-type: none"> <input type="checkbox"/> Unclear payment terms or delays <input type="checkbox"/> Low order volumes from small buyers <input type="checkbox"/> Difficulty in negotiating bulk orders <input type="checkbox"/> Competition from larger suppliers <input type="checkbox"/> Difficulty finding buyers
Do you offer bulk discounts based on order size?	<ul style="list-style-type: none"> <input type="radio"/> Yes, for all orders above a certain quantity <input type="radio"/> Yes, but only for long-term buyers <input type="radio"/> No, we have fixed pricing
Would you be interested in a system that groups multiple small buyers together to help them meet your bulk discount requirements?	<ul style="list-style-type: none"> <input type="radio"/> Yes, it would help increase sales <input type="radio"/> Maybe, if there is a clear mechanism for managing group orders <input type="radio"/> No, we prefer direct transactions
Buyer-Focused Section:	
What are the biggest challenges your company faces when sourcing suppliers?	<ul style="list-style-type: none"> <input type="checkbox"/> Finding verified and trustworthy suppliers <input type="checkbox"/> High prices and lack of bulk discounts <input type="checkbox"/> Difficulty finding product alternatives <input type="checkbox"/> Long procurement cycles and slow supplier responses

	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Lack of suitable suppliers for specific business needs
Would you benefit from a feature that suggests alternative suppliers for the same product?	<ul style="list-style-type: none"> <input type="radio"/> Yes, it would improve procurement efficiency <input type="radio"/> Maybe, if suppliers offer competitive pricing <input type="radio"/> No, we are satisfied with our current suppliers
Would your company be interested in a system that allows multiple buyers to form a group purchase to secure bulk discounts?	<ul style="list-style-type: none"> <input type="radio"/> Yes, we frequently purchase in bulk <input type="radio"/> Yes, but only if the cost savings are significant <input type="radio"/> Maybe, it depends on the process and supplier options <input type="radio"/> No, we prefer independent purchases

Appendix B: Detailed System Requirements Tables

The requirements table follows a structured format inspired by the Volere Requirements Specification Template, an industry-standard framework for requirements engineering [40]. This table provides a structured breakdown of the system requirements, helping prioritize features and track dependencies.

Table B - 1: Requirements Table Format and Usage Guidance

Requirement Title	A short, clear name for the requirement.
Description	A brief but precise explanation of what the requirement entails.
User Beneficiary	Identifies who benefits from this requirement (Supplier, Buyer, or Shared).
Priority	The level of importance (High, Medium, Low) to indicate its criticality.
Rationale	Justifies the need for the requirement in the system.
Dependencies	Lists other requirements, APIs, or services that must exist for this one to function properly.

Table B - 2: Availability

Requirement Title	Availability
Description	The system shall maintain a minimum uptime of 95% per month, with no downtime exceeding 6 consecutive hours.
User Beneficiary	Shared
Priority	High
Rationale	Ensures high availability and reliability of the platform for users.
Dependencies	Hosting platform (DigitalOcean).

Table B - 3: Reliability

Requirement Title	Reliability
Description	In the event of a crash or accidental page closure, the system shall restore the most recently saved user input.
User Beneficiary	Shared
Priority	High
Rationale	Prevents loss of user data and improves user experience.
Dependencies	localStorage for storing user input, frontend input persistence.

Table B - 4: Usability

Requirement Title	Usability
Description	The system shall offer an intuitive and user-friendly interface with clear navigation. The Usability Test score shall not be below 70%.
User Beneficiary	Shared
Priority	Medium
Rationale	Ensures that users can navigate the platform easily and perform tasks efficiently.
Dependencies	Frontend framework (React.js), UI/UX design.

Table B - 5: Portability

Requirement Title	Portability
Description	The system shall be compatible with the major web browsers: Google Chrome, Mozilla Firefox, and Safari.
User Beneficiary	Shared
Priority	Medium
Rationale	Ensures the platform is accessible across different devices and browsers.
Dependencies	Frontend code compatibility with major web browsers.

Table B - 6: User Registration & Authentication

Requirement Title	User Registration & Authentication
Description	The system shall allow users to register, login, and manage their authentication details. Users must enter their business name, email, password, location, industry type, and Commercial Registration number during sign-up. The system shall confirm that the Commercial Registration is valid and active. Additionally, users must agree to the platform's terms and conditions before completing registration. For authentication, users shall log in using their Commercial Registration number or email and password. The system shall enforce secure password complexity rules (8-28 characters, with at least one uppercase letter, one lowercase letter, and one number). Users must also have the option to reset their password if forgotten.
User Beneficiary	Shared
Priority	High
Rationale	Authentication is required to ensure secure access to the platform and appropriate role-based permissions.
Dependencies	Wathq API (for Commercial Registration validation), and frontend and backend validation.

Table B - 7: Account Management

Requirement Title	Account Management
Description	The system shall allow users to update their business details, including business name, the name of the person responsible for the company account, location, industry type, and password. Users shall also have the option to log out at any time. Additionally, users shall be able to toggle between roles (e.g., switching from Buyer to Supplier and vice versa), gaining access to the corresponding functionalities depending on their role.
User Beneficiary	Shared
Priority	High
Rationale	This feature is necessary for users to keep their information up to date, maintain control over their accounts, and switch roles based on their needs.
Dependencies	Frontend for role-switching functionality, backend validation, session management.

Table B - 8: Notifications & Alerts

Requirement Title	Notifications & Alerts
Description	The system shall notify users via in-platform notifications about important updates such as new messages, bid requests, order updates, and invoice status changes. Users shall also have the option to manage notification settings, enabling them to turn on or off specific alerts according to their preferences.
User Beneficiary	Shared
Priority	High
Rationale	Notifications are critical for keeping users informed about important actions on their account, such as accepting or rejecting invoices and receiving status updates. Allowing users to manage their preferences ensures they receive relevant notifications without being overwhelmed.
Dependencies	Notification service (backend infrastructure for notifications), WebSocket for real-time notifications, backend settings management.

Table B - 9: Messaging & Communication

Requirement Title	Messaging & Communication
Description	The system shall allow buyers and suppliers to communicate via an internal direct messaging system. Users shall be able to send and receive messages in real-time. Additionally, both suppliers and buyers shall be able to send images in chat to facilitate product/service discussions and inquiries.
User Beneficiary	Shared
Priority	High
Rationale	Messaging functionality is essential for communication between buyers and suppliers, enabling quick and direct interaction. It is crucial for suppliers to create invoices, making it a necessary component for the platform to function effectively.
Dependencies	Messaging system infrastructure, real-time communication service (WebSocket), file upload service.

Table B - 10: Product & Service Management

Requirement Title	Product & Service Management
Description	The system shall allow suppliers to manage their product and service listings. Specifically, suppliers can: Add a new listing by providing necessary details (name, description, pricing, images). Edit an existing listing to update information. Duplicate an existing listing for quicker listing creation. Delete an existing listing if it is no longer available. The system shall also allow suppliers to categorize their listings by industry when adding a new product or service.
User Beneficiary	Supplier
Priority	High

Rationale	Product and service management is essential for suppliers to maintain up-to-date inventory and manage their offerings effectively. The ability to add, edit, duplicate, and delete listings gives suppliers full control over their products and services.
Dependencies	Database for storing product/service details, frontend for product management interface, image storage service.

Table B - 11: Product Order Management

Requirement Title	Product Order Management
Description	The system shall notify suppliers upon order placement by buyers. The default status of a new order will be set to "Pending". Suppliers will be able to view the order details, including the products ordered and buyer information. The system shall allow suppliers to change the status of an order at different stages, such as: "Processing" when they begin working on the order. "Shipped" once the order has been dispatched. Order status changes to "Completed" once the buyer confirms delivery.
User Beneficiary	Supplier
Priority	High
Rationale	Order management is crucial for suppliers to fulfill buyer requests and track received orders.
Dependencies	Order database for order tracking, frontend order management interface, order status management logic.

Table B - 12: Bidding Management from Supplier Perspective

Requirement Title	Bidding Management
Description	The system shall allow suppliers to view bid requests, including relevant details such as project title, main activity, and submission deadline. Suppliers shall also be able to submit offers for bid requests. The system will notify suppliers when their offer is accepted or rejected by the buyer. If the buyer does not respond within the specified expected response time, the system shall send a rejection notification to all suppliers who participated in the bid.
User Beneficiary	Supplier
Priority	High
Rationale	Bidding management is important for competing with other similar platforms that already offer this feature. It enables suppliers to engage with potential buyers and submit competitive offers. This feature is essential for ensuring the platform remains competitive and meets industry standards.
Dependencies	Bidding system infrastructure, notification service for bid acceptance/rejection, backend logic for bid management.

Table B - 13: Invoice Management from Supplier Perspective

Requirement Title	Invoice Management
--------------------------	--------------------

Description	The system shall allow suppliers to create invoices for buyers via direct messages. Invoices will include the product/service name, agreed details (e.g., customization, service duration), prices, delivery date, and payment terms (Full Payment or Partial Payment). Suppliers will be able to receive invoices for successful group purchases and view their invoice history, which includes the statuses: Accepted, Rejected, Partially Paid, Fully Paid. The system will notify suppliers of any status changes to their invoices.
User Beneficiary	Supplier
Priority	High
Rationale	Invoice management is essential for processing payments, ensuring financial transparency, and facilitating payment tracking. This enables suppliers to create invoices for buyers, receive invoices for group purchases and biddings, and keep track of payment statuses.
Dependencies	Messaging system for invoice creation, database for storing invoice details, notification service for status updates.

Table B - 14: Storefront Management

Requirement Title	Storefront Management
Description	The system shall allow the supplier to set a fixed delivery fee that will be applied to all orders placed. Suppliers will be able to customize their storefront by adding a banner and a business bio. The system shall allow suppliers to temporarily close their store, and when a store is closed, the storefront page will display a customizable "Out of Order" message, defined by the supplier. Additionally, the system will notify the supplier of new reviews.
User Beneficiary	Supplier
Priority	Medium
Rationale	Storefront management allows suppliers to personalize their presence on the platform and manage their business visibility.
Dependencies	Frontend for storefront customization, backend for store status management, notification service for new reviews.

Table B - 15: Business Insights & Analytics

Requirement Title	Business Insights & Analytics
Description	The system shall display key business insights for suppliers, including total sales, overall supplier rating, most ordered product, and most wish-listed product. The system will also display the option to view AI-powered stock demand forecasts for each product on the products and services page, with predictions for the next three months. Based on these forecasts, the system will provide recommendations for restocking as well.
User Beneficiary	Supplier
Priority	Medium (High for Demand Forecasting)
Rationale	Business insights and stock demand forecasting help suppliers make informed decisions about inventory management, sales

	strategies, and restocking needs. For demand forecasting, the priority is high as it is a key feature that differentiates our system from others.
Dependencies	Prophet AI model for demand forecasting, database for sales and product data, frontend for displaying insights and recommendations.

Table B - 16: Subscription Management

Requirement Title	Subscription Management
Description	The platform shall offer two subscription plans for suppliers: a Basic Free Plan with limited features and a Premium Plan (50 SAR/month) that provides additional benefits. The Basic Free Plan shall include the following features: creation of a storefront, receiving orders, listing up to 10 products and 3 services, and the ability to view and respond to buyer messages. The Premium Plan includes unlimited product and service listings, and stock demand forecasting powered by AI-driven insights. New suppliers will receive a one-month free trial of the Premium Plan, with no credit card required for activation. If a supplier fails to renew their subscription, their storefront and products will be temporarily hidden from buyers until payment is made.
User Beneficiary	Supplier
Priority	Low
Rationale	Subscription management is a useful addition to the platform, helping to provide a more realistic feel and creating a more authentic experience. This feature enables tiered access to tools, such as AI-driven insights, motivating suppliers to upgrade for more advanced benefits. While it's a valuable addition, the platform can function without subscriptions, and we could offer all features for free.
Dependencies	Subscription management system, database for storing subscription details, and ensuring suppliers can't access premium features without a subscription (this will be managed through implementing appropriate access control mechanisms).

Table B - 17: Supplier & Product Discovery

Requirement Title	Supplier & Product Discovery
Description	The system shall allow buyers to search for products, services, and suppliers by entering keywords or selecting relevant categories. Buyers will be able to filter search results based on criteria such as minimum order price, excluding products with a total order value exceeding the specified amount. When displaying search results, the system shall present each product with its image, name, price, and the supplier's name. For products, the system will also display the minimum order quantity requirement. When selecting a product, the system shall display the full product description, product rating, and group purchasing details (if applicable), such as minimum required

	units per group, the discount offered, and the order deadline. Additionally, the system will allow buyers to save products to a wishlist for future reference and offer a "Find Alternatives" option to discover similar products.
User Beneficiary	buyer
Priority	High (Low for Wishlist)
Rationale	Supplier and product discovery is crucial for the buyer's experience, enabling efficient search and navigation through products and services, improving the overall usability of the platform. The wishlist feature is low priority because the platform can function without it, and its absence does not impede the core functionality of the system.
Dependencies	Search functionality, database for products and suppliers, frontend for displaying search results and filters, suppliers adding products and services to the platform.

Table B - 18: Order Placement & Management

Requirement Title	Order Placement & Management
Description	The system shall allow buyers to add items to their cart. Buyers will be able to proceed to checkout and complete the purchase. The system shall prevent buyers from ordering products that are out of stock by automatically marking them as unavailable. The system will track and display the status of orders (Pending, Processing, Shipped, Completed) and notify buyers when the order status changes. Once an order is delivered, the buyer can mark it as "Completed" and submit reviews and ratings for the product and supplier.
User Beneficiary	Buyer
Priority	High (Low for Ratings and Reviews)
Rationale	Efficient order management ensures smooth transaction processing and enables buyers to track the status of their orders, enhancing the overall user experience. Writing reviews is of low priority, as it is not a critical feature for order completion and can be added after the core functionality is established.
Dependencies	Cart system, Tap payment gateway, order status tracking, notification system, review and rating system.

Table B - 19: Finding Product Alternatives

Requirement Title	Finding Product Alternatives
Description	The system shall offer a "Find Alternatives" option next to wish-listed products, allowing buyers to discover similar products. Additionally, the system shall allow buyers to enter a product name they want to find alternatives for. This option will be available at the bottom of the homepage, and upon submission, the user will be redirected to a "Product Similar-To" page. If a product in the buyer's cart goes out of stock, the system shall also provide a link to this "Product Similar-To" page, enabling the buyer to find alternatives.

User Beneficiary	Buyer
Priority	High
Rationale	Finding product alternatives is a core feature that enhances the buyer's experience by providing a convenient way to explore other products and suppliers. This feature is essential for product discovery, which increases user engagement and satisfaction.
Dependencies	LaBSE AI model for semantic search, product database, and frontend for displaying similar products.

Table B - 20: Bidding Management from Buyer Perspective

Requirement Title	Bidding Management
Description	The system shall allow buyers to create bids, where they can specify details such as the Bidding Project Title and Response Deadline for Offers. If the buyer does not respond to supplier offers within the selected timeframe, the system shall automatically reject all pending offers and notify them. The system shall notify buyers when a supplier submits an offer, and buyers can review the offers before accepting one. Once an offer is accepted, the system shall automatically generate an invoice based on the agreed price and details.
User Beneficiary	Buyer
Priority	High
Rationale	Bidding management is essential for allowing buyers to gather competitive offers for their projects. This feature improves the efficiency and transparency of the procurement process.
Dependencies	Bidding system infrastructure, notification system for bid updates, invoice system.

Table B - 21: Invoice Management from Buyer Perspective

Requirement Title	Invoice Management
Description	During negotiations, buyers shall be able to choose between two payment options: full payment or partial payment. The system shall notify buyers when a new invoice is created, and display a list of the buyer's invoices with statuses such as Accepted, Rejected, Partially Paid, and Fully Paid. The system will allow buyers to review and accept invoices and proceed with their chosen payment option. For group purchases, a Pre-Invoice will be created to track the status of the group purchase. Once the group purchase is successful, the Pre-Invoice will be converted into a standard invoice.
User Beneficiary	Buyer
Priority	High
Rationale	Invoice management ensures transparency in transactions, enabling buyers to manage payments effectively and track the status of their purchases.
Dependencies	Invoice system, Tap payment gateway, group purchase functionality, notification system for updates.

Appendix C: Development Workflow and GitHub Repositories

This appendix documents the development workflow followed throughout the Silah project and provides an overview of the GitHub organization, repositories, and project management practices used by us. Since all the code developed for Silah is fully open-source, readers are encouraged to visit the project organization on GitHub to explore the repositories, source code, issues, documentation, and workflows in more detail. The main GitHub organization can be accessed at:

<https://github.com/GP-Silah>

while the three primary repositories of the project are available at:

<https://github.com/GP-Silah/silah-ai>

<https://github.com/GP-Silah/silah-backend>

<https://github.com/GP-Silah/silah-frontend>

GitHub Organization & Repositories

The GitHub organization created for the project serves as the central collaborative space for all development work. The landing page of the organization presents an overview of the repositories, available documentation, and the GitHub Projects used for planning.

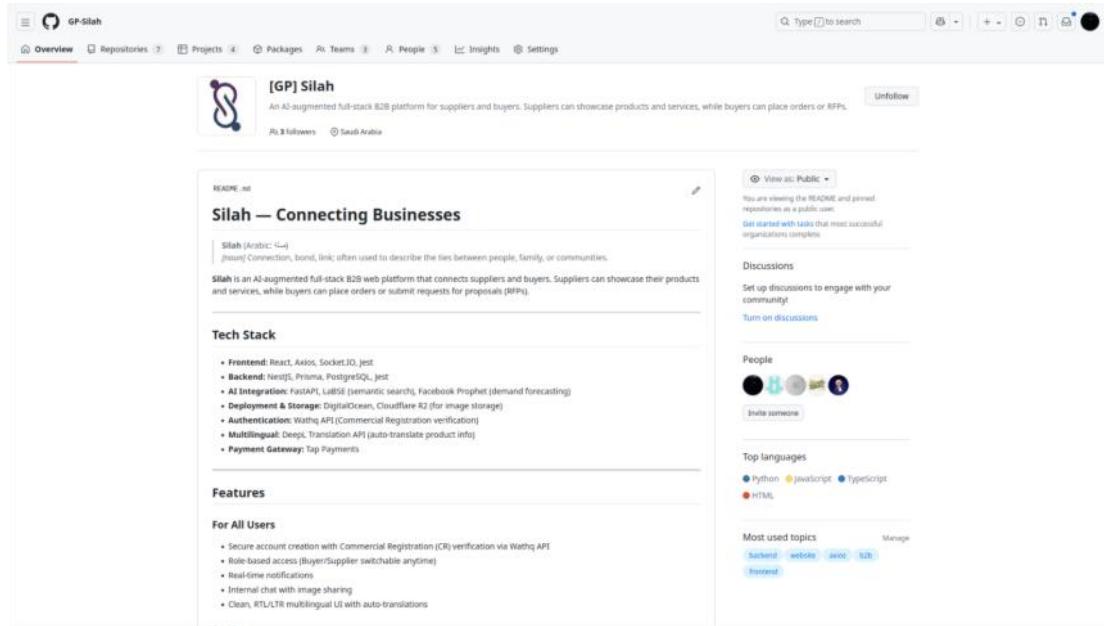


Figure C - 1: GitHub Project Organization Landing Page

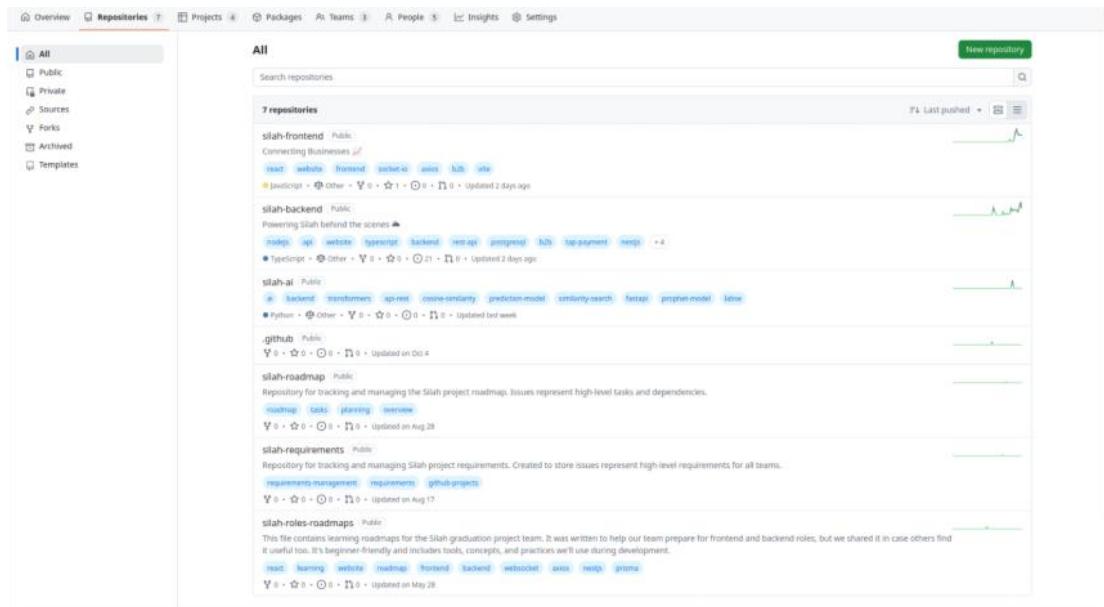


Figure C - 2: GitHub Repositories

The GitHub organization includes three primary repositories: the AI backend, the NestJS backend, and the frontend web application. Each repository is dedicated to a specific subsystem of Silah and was developed independently while still adhering to a shared workflow.

The AI repository contains a production-ready FastAPI backend responsible for exposing the intelligent features of Silah to the rest of the system. This repository represents the operational layer of the intelligence module; the part that the

NestJS backend communicates to obtain forecasts, embeddings, and similarity results.

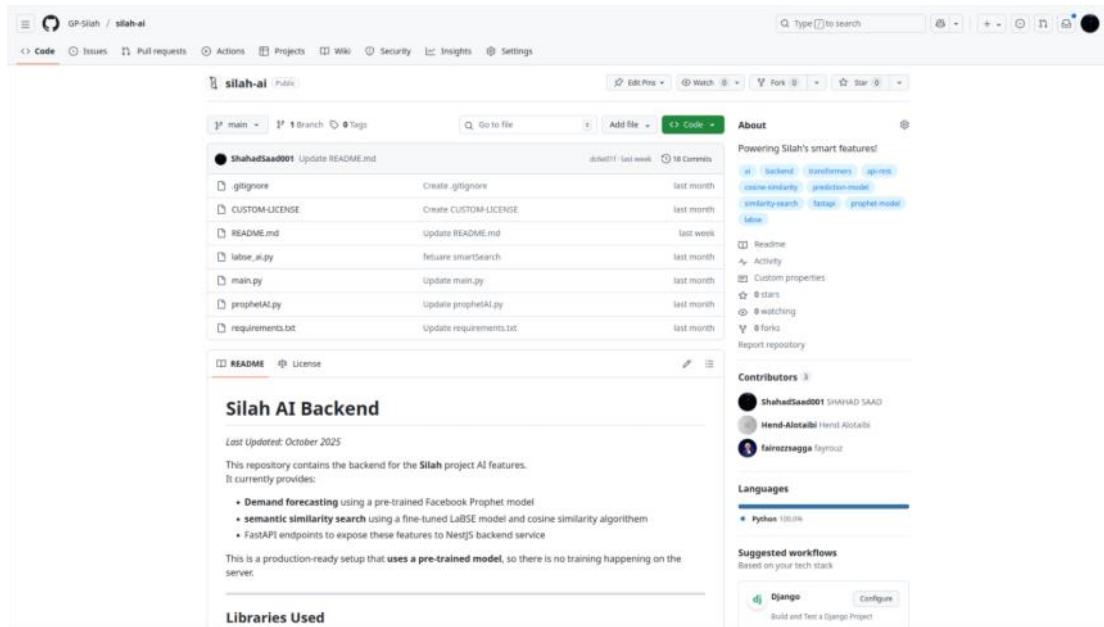


Figure C - 3: AI Repository

The backend repository contains the NestJS application that coordinates authentication, user management, business rules, and API routing. It communicates directly with the AI FastAPI service through internal HTTP calls. This repository also includes pull requests, automated checks, and GitHub Actions workflows.

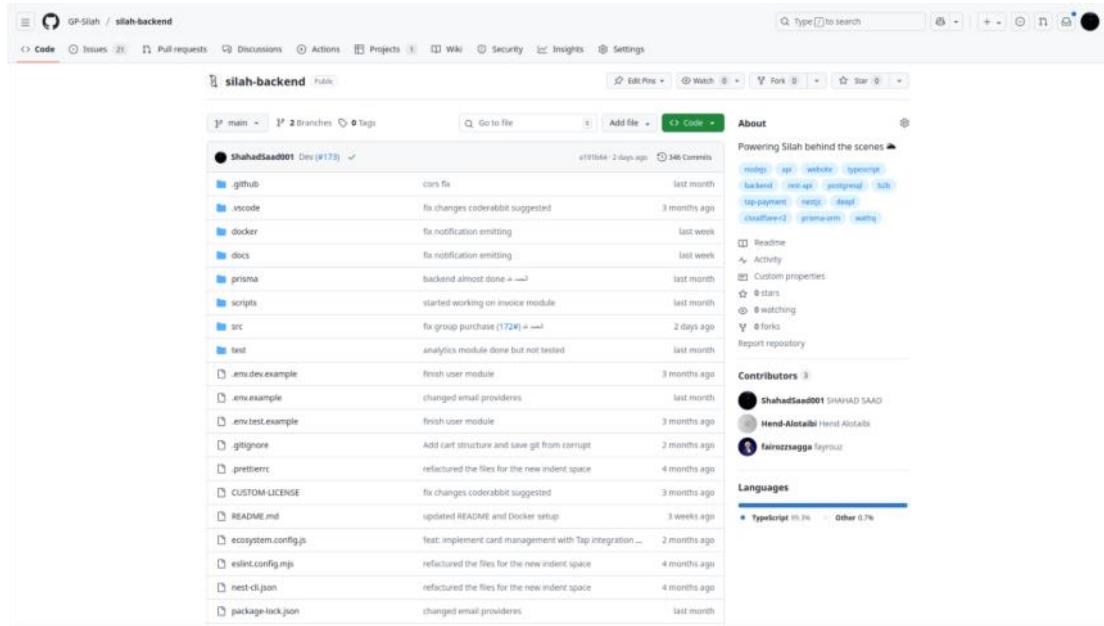


Figure C - 4: Backend Repository

The frontend repository includes the React-based web application. It implements the user interface, integrates with the NestJS backend, and manages client-side routing and state.

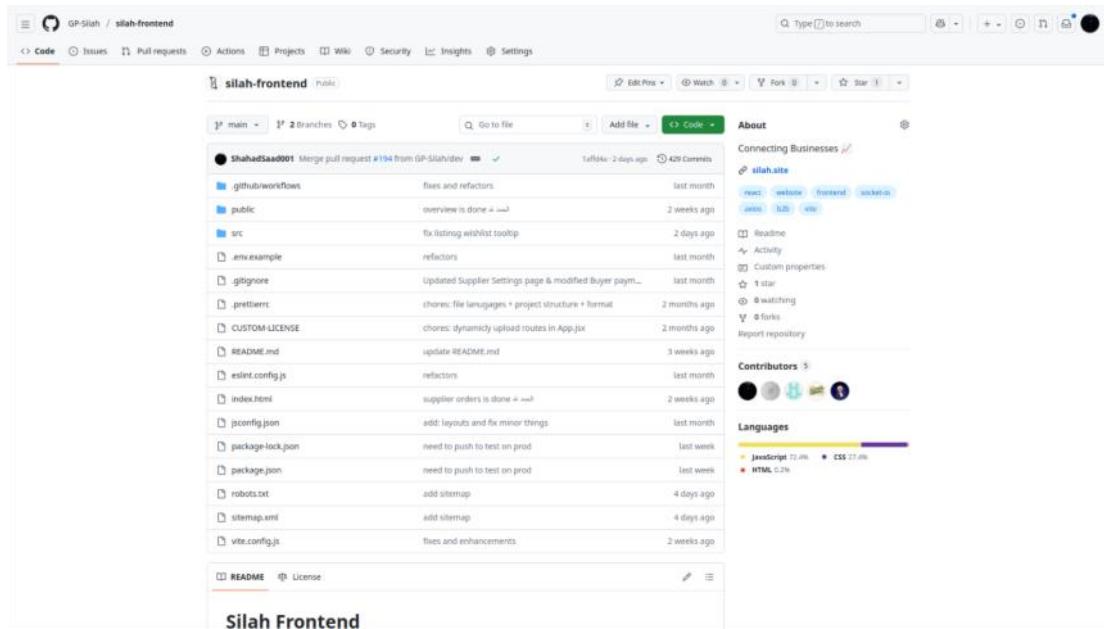


Figure C - 5: Frontend Repository

Together, these repositories form the complete development space for Silah. They also reflect how the system is modularized: the AI logic runs independently, the backend orchestrates the core business logic, and the frontend presents a clean interface to end users.

GitHub Workflow & Collaborative Practices

Throughout development, all repositories in the Silah organization followed a unified branching strategy designed to maintain structure and reduce integration issues. Each repository consisted of three primary branches: a main branch representing the stable production-ready state, a dev branch used for integrating ongoing work, and multiple feature branches where individual tasks were implemented. Figure C-6 illustrates this branching workflow, which served as the foundation for our development process.

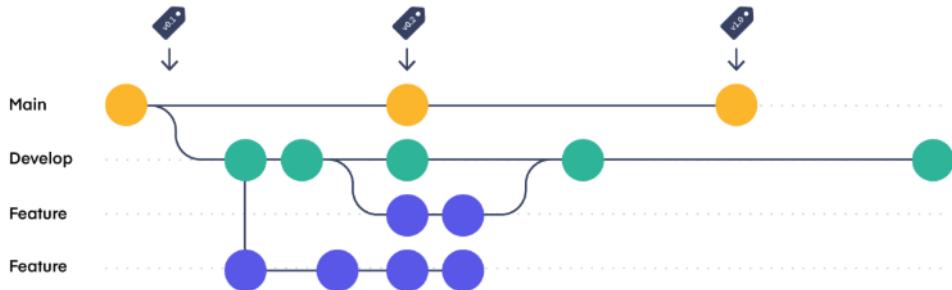


Figure C - 6: GitHub Workflow Strategy [41]

This workflow enabled us to work independently without interfering with each other's progress. New features and bug fixes were implemented on isolated branches, allowing us to develop and test our work locally before integration. When a feature reached a stable condition, a pull request was opened to merge the changes into the dev branch. These pull requests acted as clear documentation of what each change introduced, making it easier to track progress and understand the rationale behind different updates.

Once features were integrated into the dev branch and verified to be functioning correctly, the dev branch was merged into main. This two-step merging process reduced the likelihood of conflicts, minimized accidental overwrites, and maintained a clean history of the project's evolution. It also ensured that the main branch only contained stable and confirmed work, aligning with standard industry practices for version control and collaborative software development.

Although we did not conduct formal code reviews due to time constraints and the academic nature of the project, GitHub still played a central role in keeping contributions organized. Pull requests served as a lightweight form of documentation and integration management, while day-to-day coordination occurred through WhatsApp. We used messages, screenshots, and occasional recorded explanations to clarify changes and align on decisions when necessary. This combination of GitHub's structured workflow and continuous informal communication allowed us to collaborate effectively despite the fast-paced timeline of the project.

Together, the branching strategy, pull request workflow, and ongoing communication formed a practical development process that supported consistent progress and maintained project stability.

GitHub Projects & Project Management

In addition to the repositories themselves, the GitHub organization included two major GitHub Projects that supported planning, progress tracking, and requirement alignment. Both Projects were essential for keeping the team synchronized and for ensuring that implementation work consistently reflected the goals outlined in the Software Requirements Specification (SRS).

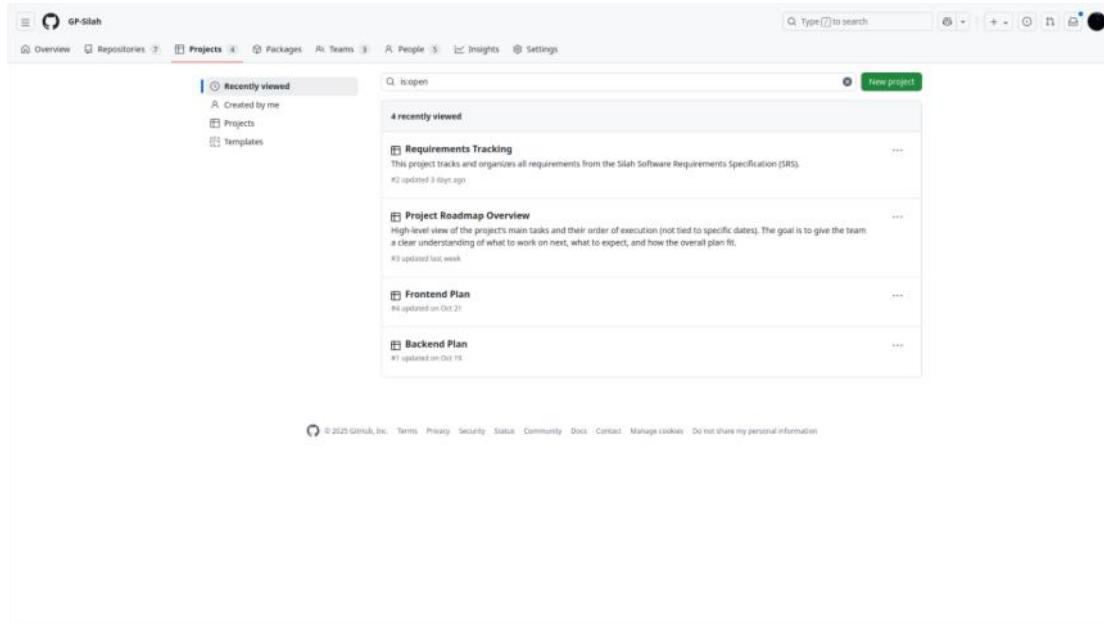


Figure C - 7: GitHub Projects

The first project, Requirements Tracking, served as a centralized space for documenting all requirements extracted from the SRS. Instead of mixing these high-level requirements with the technical issues inside each code repository, we created a separate tracking board to organize them clearly. This approach ensured that we remained aligned with the SRS and avoided accidental fragmentation of requirement knowledge across different repositories.

The screenshot shows a Requirements Tracking board with the following columns:

Title	Status	Responsibility	Req Type	Req Status	Assignee	Notes	
1 Software Requirements #1	Done	All Teams	Software Req	Done			
2 Hardware Requirements #2	Done	All Teams	Hardware Req	Done			
3 Availability #3	Done	Backend Team	Non-Functional Req	Done			
4 Reliability #4	Done	Frontend Team	Non-Functional Req	Done			
5 Usability #5	Done	Frontend Team	Non-Functional Req	Done			
6 Portability #6	Done	Frontend Team	Non-Functional Req	Done			
7 1.1 User Registration #7	Done	Frontend + Backend Te...	Functional Req, Shared	Done			
8 1.2 User Authentication #8	Done	Frontend + Backend Te...	Functional Req, Shared	Done			
9 1.3 Account Management #9	Done	Frontend + Backend Te...	Functional Req, Shared	Done			
10 1.4 Notifications #10	Done	Frontend + Backend Te...	Functional Req, Shared	Done			
11 1.5 Messaging #11	Done	Frontend + Backend Te...	Functional Req, Shared	Done			
12 2.1 Listings Management #12	Done	Frontend + Backend Te...	Functional Req, Supplier	Done			
13 2.2 Product Order Management #13	Done	Frontend + Backend Te...	Functional Req, Supplier	Done			
14 2.3 Bidding Management #14	Done	Frontend + Backend Te...	Functional Req, Supplier	Done			
15 2.4 Invoice Management #15	Done	Frontend + Backend Te...	Functional Req, Supplier	Done			
16 2.5 Storefront Management #16	Done	Frontend + Backend Te...	Functional Req, Supplier	Done			
17 2.6 Business Insights & Analytics #17	Done	All Teams	Functional Req, Supplier	Done			
18 2.7 Subscription Management #18	Done	Frontend + Backend Te...	Functional Req, Supplier	Done			
19 3.1 Product Discovery #19	Done	All Teams	Functional Req, Buyer*	Done			
20 3.2 Order Placement #20	Done	All Teams	Functional Req, Buyer*	Done			
21 3.3 Order Management #21	Done	Frontend + Backend Te...	Functional Req, Buyer*	Done			
22 3.4 Bidding Management #22	Done	Frontend + Backend Te...	Functional Req, Buyer*	Done			
23 3.5 Invoice Management #23	Done	Frontend + Backend Te...	Functional Req, Buyer*	Done			

+ You can use `Control + Space` to add an item

Figure C - 8: Requirements Tracking Board

The second project, Project Roadmap Overview, provided a high-level plan of the major development tasks and the expected sequence in which they should be approached. The roadmap was not tied to fixed deadlines but rather served as a guide for identifying what to work on next and how different modules connected to each other. While the issues inside the repositories contained detailed code-level tasks, the roadmap allowed us to understand the broader narrative of the project.

The screenshot shows a GitHub Project Roadmap Overview board. At the top, there are navigation links for 'The Plan', 'Status', 'Table', and '+ New view'. A search bar with placeholder 'Type to search' and a 'Save' button are also present. The main area is a table with the following columns: Title, Assignees, Status, and Responsibility. The table contains 13 rows of tasks, each with a small profile icon, a title, an assignee name, a status (Done), and a responsibility team (Backend Team or Frontend Team). The tasks include various project modules and user-related pages.

Title	Assignees	Status	Responsibility
1 Backend User Preferences & Interaction Modules #6	ShahadSaeed001	Done	Backend Team
2 Backend Purchase-Related Modules #7	ShahadSaeed001	Done	Backend Team
3 Backend Bidding-Related Modules #8	ShahadSaeed001	Done	Backend Team
4 AI-Related Tasks #3	A201525, fairozz...	Done	All Teams
5 Backend Listings-Related Modules #5	ShahadSaeed001	Done	Backend Team
6 Learn Git & GitHub #1	A201525, fairozz...	Done	All Teams
7 Frontend Guest-Facing Pages #2	A201525, fairozz...	Done	Frontend Team
8 Backend User-Related Modules #4	ShahadSaeed001	Done	Backend Team
9 Frontend Listings-Related Pages #9	Hend-Alotabi, Ma...	Done	Frontend Team
10 Frontend User Actions Pages #10	Hend-Alotabi and ...	Done	Frontend Team
11 Frontend Purchase-Related Pages #11	A201525, Hend-Al...	Done	Frontend Team
12 Frontend Bidding-Related Pages #12	Hend-Alotabi and ...	Done	Frontend Team
13 Frontend Supplier-Related Pages #13	A201525, fairozz...	Done	Frontend Team

Figure C - 9: Project Roadmap Overview Board

These GitHub Projects were extremely valuable, particularly in preventing requirement drift and in keeping the development process organized across multiple teams. They also helped ensure transparency for anyone reviewing the project, since both the requirements documentation and the implementation progress remain publicly accessible.

Importance of GitHub in the Silah Development Lifecycle

GitHub played a central role in every stage of the Silah development lifecycle. It supported source control, collaboration, and deployment. It also served as the single source of truth for requirements, implementation progress, system architecture, and release history.

By making the repositories and project boards publicly accessible, the project maintains complete transparency. Readers of this report can freely explore the repositories, examine commit history, inspect code examples, review open and closed issues, and analyze pull requests. The workflows, pipelines, and project boards offer insight into how we organized and executed the development of Silah from start to finish.

Appendix D: Full Prisma Schema Code

This appendix presents the complete schema.prisma file that defines all database models, relations, and enumerations used in the Silah backend. It serves as a technical reference for developers and readers who wish to explore the full structure of the database beyond the excerpts shown in Chapter 5.

For clarity and readability, the schema is divided into multiple figures, each representing a logical section of the data model (e.g., Users, Suppliers, Orders, etc.).

```

schema.prisma ✘
prisma > schema.prisma > ...
You, 3 weeks ago | 2 authors (You and one other)
1 // This is your Prisma schema file! You, 4 weeks ago via PR #62 · remove Settings module and re...
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 // Looking for ways to speed up your queries, or scale easily with your serverless or edge functions?
5 // Try Prisma Accelerate: https://pris.ly/cli/accelerate-init
6
7 generator client {
8   provider = "prisma-client-js"
9   output   = "../node_modules/.prisma/client"
10 }
11
12 Shabed Said, 4 months ago | 1 author (Shabed Said)
13 datasource db {
14   provider = "postgresql"
15   url      = env("DATABASE_URL")
16 }
17 // ===== USER =====
18 model User {
19   id          String    @id @default(uuid())
20   tapCustomerId String   @unique
21   email       String   @unique
22   crn        String   @unique
23   password    String
24   name        String
25   role        UserRole @default(BUYER)
26   businessName String
27   city         String
28   nid         String   @unique
29   agreedToTerms Boolean @default(true)
30   isEmailVerified Boolean @default(false)
31   pfpFileName String?
32   isPfpDefault Boolean @default(true)
33   preferredLanguage Languages @default(EN)
34   createdAt    DateTime @default(now())
35   updatedAt    DateTime @updatedAt
36
37   categories   UserCategory[]
38   supplier     Supplier?
39   buyer        Buyer?
40   notificationPreference NotificationPreference?
41   sentNotifications Notification[] @relation("SentNotifications")
42   receivedNotifications Notification[] @relation("ReceivedNotifications")
43   firstChatUser Chat[] @relation("FirstUser")
44   secondChatUser Chat[] @relation("SecondUser")
45   sentMessages  Message[]
46   receivedMessages Message[] @relation("ReceivedMessages")
47 }

```

Figure D - 1: Prisma Schema (Connection String and User model)

```

49 // ===== SUPPLIER =====
50 You, 3 weeks ago | 1 author (You)
51 model Supplier {
52   id          String    @id @default(uuid())
53   userId      String   @unique
54   user        User      @relation(fields: [userId], references: [id], onDelete: Cascade, onUpdate: Cascade)
55   plan        SupplierPlan @default(BASIC)
56   status      SupplierStatus @default(ACTIVE)
57   usedFreeTrial Boolean @default(false)
58   isStoreClosed Boolean @default(true)
59   storeClosedMsg String   @default("This store is currently closed. Please check back later.")
60   storeBio    String?
61   storeBannerFileName String?
62   deliveryFees Float    @default(0.0)
63   avgRating   Float    @default(0.0)
64   ratingsCount Int     @default(0)
65   createdAt    DateTime @default(now())
66   updatedAt    DateTime @updatedAt
67   isDeleted    Boolean @default(false)
68   deletedAt   DateTime?
69
70   categories   SupplierFavoriteCategory[]
71   subscriptions SupplierSubscription[]
72   products     Product[]
73   services     Service[]
74   cartGroups   CartBySupplier[] // buyer's cart is grouped based on suppliers
75   ordersReceived Order[]
76   preInvoices  PreInvoice[]
77   invoices     Invoice[]
78   receivedReviews Review[]
79   groupPurchases GroupPurchase[]
80   writtenOffers Offer[]
81 }
82
83 // ===== SUBSCRIPTIONS =====
84 You, 4 weeks ago | 1 author (You)
85 model SupplierSubscription {
86   id          String    @id @default(uuid())
87   supplierId String
88   supplier    Supplier @relation(fields: [supplierId], references: [id], onDelete: Cascade)
89   startDate   DateTime @default(now())
90   endDate     DateTime @default(dbgenerated("now() + interval '1 month'")) // postgres interval syntax
91   createdAt   DateTime @default(now())
92   updatedAt   DateTime @updatedAt
93 }

```

Figure D - 2: Prisma Schema (Supplier and Subscription models)

```

93 // ===== BUYER =====
94 You, 3 weeks ago | 1 author (You)
95 model Buyer {
96   id      String  @id @default(uuid())
97   userId String  @unique
98   user    User    @relation(fields: [userId], references: [id], onDelete: Cascade, onUpdate: Cascade)
99   card    Card?
100  createdAt DateTime @default(now())
101  updatedAt DateTime @updatedAt
102  carts      Cart[]
103  orders     Order[]
104  wishlist   Wishlist[]
105  preInvoices PreInvoice[]
106  invoices   Invoice[]
107  writtenReviews Review[]
108  writtenItemsReviews ItemReview[]
109  joinedGroupPurchases GroupPurchaseBuyer[]
110  createdBids Bid[]
111 }
112
113 // ===== CARD =====
114 You, 4 weeks ago | 1 author (You)
115 model Card {
116   id      String  @id @default(uuid())
117   tapCardId String  @unique
118   buyerId String  @unique
119   buyer    Buyer   @relation(fields: [buyerId], references: [id], onDelete: Cascade)
120   cardHolderName String
121   last4    String
122   brand    String
123   expMonth String
124   expYear  String
125   createdAt DateTime @default(now())
126   updatedAt DateTime @updatedAt
127 }

```

Figure D - 3: Prisma Schema (Buyer and Card models)

```

128 // ===== CATEGORY =====
129 You, 4 weeks ago | 2 authors (You and one other)
130 model Category {
131   id      Int      @id @default(autoincrement())
132   name    String   @unique
133   parentCategoryId Int?
134   parentCategory Category? @relation("CategoryToSubcategories", fields: [parentCategoryId], references: [id], onDelete: Cascade)
135   subcategories Category[] @relation("CategoryToSubcategories")
136   usedFor    ItemType // products or services
137   createdAt   DateTime @default(now())
138   updatedAt   DateTime @updatedAt
139
140   users     UserCategory[]
141   suppliers SupplierFavoriteCategory[]
142   products   Product[]
143   services   Service[]
144 }
145
146 // ===== USER_CATEGORY =====
147 You, 4 weeks ago | 1 author (You)
148 model UserCategory {
149   userId   String
150   user     User    @relation(fields: [userId], references: [id], onDelete: Cascade)
151   categoryId Int
152   category  Category @relation(fields: [categoryId], references: [id], onDelete: Cascade)
153   createdAt DateTime @default(now())
154   updatedAt DateTime @updatedAt
155
156   @@unique([userId, categoryId])
157 }
158
159 // ===== ★ SUPPLIER_FAVORITE_CATEGORY =====
160 You, 4 weeks ago | 1 author (You)
161 model SupplierFavoriteCategory {
162   supplierId String
163   supplier   Supplier @relation(fields: [supplierId], references: [id], onDelete: Cascade)
164   categoryId Int
165   category   Category @relation(fields: [categoryId], references: [id], onDelete: Cascade)
166   createdAt   DateTime @default(now())
167
168   @@id([supplierId, categoryId]) // composite PK ensures uniqueness
169 }

```

Figure D - 4: Prisma Schema (Category, UserCategory, SupplierFavoriteCategory models)

```

168 // ===== PRODUCT =====
169 You, 3 weeks ago | 1 author (You)
170 model Product {
171   id          String      @id @default(uuid())
172   supplierId String?
173   supplier    Supplier?  @relation(fields: [supplierId], references: [id], onDelete: SetNull)
174   name        String      @db.VarChar(60)
175   description String      @db.VarChar(1000)
176   price       Float
177   stock       Int         @default(0)
178   categoryId Int
179   category    Category   @relation(fields: [categoryId], references: [id])
180   imagesFileNames String[]
181   caseQuantity Int        @default(1)
182   minOrderQuantity Int        @default(1)
183   maxOrderQuantity Int? // when null it means unlimitid
184   allowGroupPurchase Boolean   @default(false)
185   minGroupOrderQuantity Int?
186   groupPurchasePrice Float?
187   groupPurchaseDuration GroupPurchaseDeadline?
188   isPublished Boolean   @default(false)
189   wishlistCount Int        @default(0)
190   avgRating   Float      @default(0.0)
191   ratingsCount Int        @default(0)
192   isDeleted   Boolean   @default(false)
193   deletedAt  DateTime?
194   createdAt   DateTime   @default(now())
195   updatedAt   DateTime   @updatedAt
196   cartItems   CartItem[]
197   orderItems  OrderItem[]
198   invoiceItems InvoiceItem[]
199   preInvoices PreInvoice[]
200   groupPurchases GroupPurchase[]
201 }

```

Figure D - 5: Prisma Schema (Product model)

```

203 // ===== SERVICES =====
204 You, 4 weeks ago | 1 author (You)
205 model Service {
206   id          String      @id @default(uuid())
207   supplierId String?
208   supplier    Supplier?  @relation(fields: [supplierId], references: [id], onDelete: SetNull)
209   name        String      @db.VarChar(60)
210   description String      @db.VarChar(1000)
211   price       Float
212   isPriceNegotiable Boolean   @default(false)
213   categoryId Int
214   category    Category   @relation(fields: [categoryId], references: [id])
215   imagesFileNames String[]
216   serviceAvailability ServiceAvailability
217   isPublished Boolean   @default(false)
218   wishlistCount Int        @default(0)
219   avgRating   Float      @default(0.0)
220   ratingsCount Int        @default(0)
221   isDeleted   Boolean   @default(false)
222   deletedAt  DateTime?
223   createdAt   DateTime   @default(now())
224   updatedAt   DateTime   @updatedAt
225   invoiceItems InvoiceItem[]
226 }
227
228 // ===== ITEM EMBEDDING =====
229 You, 4 weeks ago | 1 author (You)
230 model ItemEmbedding {
231   id          String      @id @default(uuid())
232   itemId     String
233   itemType   ItemType
234   embedding  Float[] // array of numbers representing the vector
235   createdAt  DateTime   @default(now())
236   updatedAt  DateTime   @updatedAt
237   @unique([itemId, itemType]) // each item has exactly one embedding
238 }

```

Figure D - 6: Prisma Schema (Service and ItemEmbedding models)

```

240 // ===== CART =====
241 You, 4 weeks ago | 1 author (You)
242 model Cart {
243   id      String    @id @default(uuid())
244   buyerId String
245   buyer   Buyer    @relation(fields: [buyerId], references: [id])
246   productsTotal Float
247   deliveryFees Float
248   cartTotal  Float
249   createdAt  DateTime @default(now())
250   updatedAt  DateTime @updatedAt
251   isBought   Boolean @default(false)
252   isDeleted   Boolean @default(false)
253   deletedAt  DateTime?
254
255   orders   Order[]
256   // A cart can have many suppliers inside it
257   suppliers CartBySupplier[]
258 }
259
260 // ===== CART BY SUPPLIER =====
261 You, 4 weeks ago | 1 author (You)
262 model CartBySupplier {
263   id      String    @id @default(uuid())
264   cartId String
265   cart    Cart      @relation(fields: [cartId], references: [id], onDelete: Cascade)
266   supplierId String
267   supplier Supplier @relation(fields: [supplierId], references: [id], onDelete: Cascade)
268   deliveryFee  Float // of the supplier
269   subTotal    Float // sum of this supplier's items (without delivery fee)
270   supplierTotalPrice Float // subTotal + deliveryFee
271   createdAt  DateTime @default(now())
272   updatedAt  DateTime @updatedAt
273
274   cartItems CartItem[]
275 }
276
277 // ===== CART ITEM =====
278 You, 4 weeks ago | 1 author (You)
279 model CartItem {
280   id      Int       @id @default(autoincrement())
281   cartBySupplierId String
282   cartBySupplier CartBySupplier @relation(fields: [cartBySupplierId], references: [id], onDelete: Cascade)
283   productId  String
284   product   Product @relation(fields: [productId], references: [id])
285   quantity   Int
286   itemTotalPrice Float // product.price * quantity at time of adding
287 }

```

Figure D - 7: Prisma Schema (Cart, CartBySupplier, CartItem models)

```

286 // ===== ORDER =====
287 You, 4 weeks ago | 1 author (You)
288 model Order {
289   id      String    @id @default(uuid())
290   tapChargeId String // same across all orders from the same checkout
291   buyerId String?
292   buyer   Buyer?    @relation(fields: [buyerId], references: [id], onDelete: SetNull)
293   cartId  String
294   cart    Cart      @relation(fields: [cartId], references: [id])
295   supplierId String
296   supplier Supplier @relation(fields: [supplierId], references: [id])
297   finalPrice Float
298   status   OrderStatus @default(PENDING)
299   createdAt  DateTime @default(now())
300
301   items   OrderItem[]
302   review  Review?
303 }
304
305 // ===== ORDER ITEM =====
306 You, 4 weeks ago | 1 author (You)
307 model OrderItem {
308   id      Int       @id @default(autoincrement())
309   orderId String
310   order   Order    @relation(fields: [orderId], references: [id], onDelete: Cascade)
311   productId String?
312   product  Product? @relation(fields: [productId], references: [id], onDelete: SetNull)
313   quantity  Int     @default(1)
314   unitPrice Float // price at purchase time
315   totalPrice Float // quantity * unitPrice
316   createdAt  DateTime @default(now())
317   itemReview ItemReview?
318 }

```

Figure D - 8: Prisma Schema (Order and OrderItem models)

```

319 // ===== GROUP PURCHASE =====
320 You, 3 weeks ago | 1 author (You)
320 model GroupPurchase {
321   id          String      @id @default(uuid())
322   productId   String
323   product     Product    @relation(fields: [productId], references: [id])
324   supplierId String
325   supplier    Supplier   @relation(fields: [supplierId], references: [id])
326   city        String
327   minGroupQuantity Int
328   actualGroupQuantity Int
329   totalPrice   Float // sum of all buyers totalPrice
330   deadline    DateTime
331   status      GroupPurchaseStatus @default(OPEN)
332   createdAt   DateTime   @default(now())
333   updatedAt   DateTime   @updatedAt
334
335   joinedBuyers GroupPurchaseBuyer[]
336 }
337
338 // ===== GROUP PURCHASE BUYER =====
338 You, 3 weeks ago | 1 author (You)
339 model GroupPurchaseBuyer {
340   id          String      @id @default(uuid())
341   groupPurchaseId String
342   groupPurchase GroupPurchase @relation(fields: [groupPurchaseId], references: [id])
343   buyerId    String
344   buyer       Buyer      @relation(fields: [buyerId], references: [id])
345   quantity    Int
346   priceBasedQuantity Float
347   totalPrice   Float // priceBasedQuantity + supplier deliveryFees
348   joinedAt   DateTime   @default(now())
349
350   preinvoice PreInvoice?
351
352   @@unique([groupPurchaseId, buyerId])
353 }

```

Figure D - 9: Prisma Schema (GroupPurchase and GroupPurchaseBuyer models)

```

355 // ===== ⚡ BID =====
355 You, 3 weeks ago | 1 author (You)
356 model Bid {
357   id          String      @id @default(uuid())
358   buyerId    String
359   buyer       Buyer      @relation(fields: [buyerId], references: [id])
360   bidName    String      @db.VarChar(100)
361   mainActivity String    @db.VarChar(500)
362   submissionDeadline DateTime
363   expectedResponseTime BidExpectedResponseTime
364   status      BidStatus  @default(OPEN)
365   createdAt   DateTime   @default(now())
366   updatedAt   DateTime   @updatedAt
367
368   offers Offer[]
369 }
370
371 // ===== 💰 OFFER =====
371 You, 3 weeks ago | 1 author (You)
372 model Offer {
373   id          String      @id @default(uuid())
374   bidId      String
375   bid        Bid        @relation(fields: [bidId], references: [id], onDelete: Cascade)
376   supplierId String
377   supplier    Supplier   @relation(fields: [supplierId], references: [id])
378   proposedAmount Float
379   expectedCompletionTime DateTime
380   offerDetails String    @db.VarChar(500)
381   executionDetails String  @db.VarChar(500)
382   notes      String?
383   status      OfferStatus @default(PENDING)
384   createdAt   DateTime   @default(now())
385   updatedAt   DateTime   @updatedAt
386
387   preinvoice PreInvoice?
388 }

```

Figure D - 10: Prisma Schema (Bid and Offer models)

```

390 // ===== INVOICE =====
391 You, 4 weeks ago | author (You)
392 model Invoice {
393   id           String      @id @default(uuid())
394   buyerId     String?
395   buyer        Buyer?      @relation(fields: [buyerId], references: [id], onDelete: SetNull)
396   supplierId  String?
397   supplier     Supplier?   @relation(fields: [supplierId], references: [id], onDelete: SetNull)
398   deliveryDate DateTime
399   termsOfPayment InvoiceTermsOfPayment
400   upfrontAmount Float?
401   tapChargeIdForUpfront String?
402   tapChargeId   String? // can be null if invoice just created or partially paid
403   uponDeliveryAmount Float
404   amount        Float
405   notesAndTerms String?
406   status        InvoiceStatus @default(PENDING)
407   createdAt    DateTime   @default(now())
408
409   items       InvoiceItem[]
410   preInvoice  PreInvoice? // if created from a preinvoice
411   review      Review?
412 }
413 // ===== INVOICE ITEM =====
414 You, 3 weeks ago | author (You)
415 model InvoiceItem {
416   id           Int        @id @default(autoincrement())
417   invoiceId   String
418   invoice      Invoice?   @relation(fields: [invoiceId], references: [id])
419   name         String     @db.VarChar(60)
420   description  String     @db.VarChar(100)
421   agreedDetails String     @db.VarChar(100)
422   quantity     Int
423   unitPrice    Float
424   priceBasedQuantity Float // price * quantity
425   relatedServiceId String?
426   relatedService Service? @relation(fields: [relatedServiceId], references: [id], onDelete: SetNull)
427   relatedProductId String?
428   relatedProduct Product? @relation(fields: [relatedProductId], references: [id], onDelete: SetNull)
429   itemReview   ItemReview?
430 }

```

Figure D - 11: Prisma Schema (Invoice and InvoiceItem models)

```

432 // ===== PRE-INVOICE =====
433 You, 3 weeks ago | author (You)
434 model PreInvoice {
435   id           String      @id @default(uuid())
436   invoiceId   String?     @unique
437   invoice      Invoice?    @relation(fields: [invoiceId], references: [id])
438   groupPurchaseBuyerId String?     @unique
439   groupPurchaseBuyer GroupPurchaseBuyer? @relation(fields: [groupPurchaseBuyerId], references: [id])
440   offerId     String?     @unique
441   offer        Offer?      @relation(fields: [offerId], references: [id])
442   buyerId     String?
443   buyer        Buyer?      @relation(fields: [buyerId], references: [id], onDelete: SetNull)
444   supplierId  String?
445   supplier     Supplier?   @relation(fields: [supplierId], references: [id], onDelete: SetNull)
446   productId    String?
447   product      Product?   @relation(fields: [productId], references: [id], onDelete: SetNull)
448   amount       Float
449   status       PreInvoiceStatus @default(PENDING)
450   createdAt    DateTime   @default(now())
451
452 // ===== REVIEW =====
453 You, 4 weeks ago | author (You)
454 model Review {
455   id           String      @id @default(uuid())
456   orderId     String?     @unique
457   order        Order?      @relation(fields: [orderId], references: [id], onDelete: SetNull)
458   invoiceId   String?     @unique
459   invoice      Invoice?    @relation(fields: [invoiceId], references: [id], onDelete: SetNull)
460   buyerId     String?
461   buyer        Buyer?      @relation(fields: [buyerId], references: [id], onDelete: SetNull)
462   supplierId  String?
463   supplier     Supplier?   @relation(fields: [supplierId], references: [id], onDelete: SetNull)
464   supplierRating Int        @default(5)
465   writtenReviewOfSupplier String? @db.VarChar(150)
466   createdAt    DateTime   @default(now())
467   itemsReview  ItemReview[]
468 }
469
470 // ===== ITEM REVIEW =====
471 You, 4 weeks ago | author (You)
472 model ItemReview {
473   id           Int        @id @default(autoincrement())
474   reviewId    String
475   review      Review?    @relation(fields: [reviewId], references: [id], onDelete: Cascade)
476   orderItemId Int?      @unique
477   orderItem    OrderItem? @relation(fields: [orderItemId], references: [id], onDelete: SetNull)
478   invoiceItemId Int?      @unique
479   invoiceItem  InvoiceItem? @relation(fields: [invoiceItemId], references: [id], onDelete: Cascade)
480   buyerId     String?
481   buyer        Buyer?      @relation(fields: [buyerId], references: [id], onDelete: SetNull)
482   itemRating   Int        @default(5)
483   writtenReviewOfItem String? @db.VarChar(150)
484   createdAt    DateTime   @default(now())

```

Figure D - 12: Prisma Schema (PreInvoice, Review, ItemReview models)

```

486 // ===== ❤ WISHLIST =====
487 You, 4 weeks ago | Author (You)
488 model Wishlist {
489   id      String @id @default(uuid())
490   buyerId String @relation(fields: [buyerId], references: [id], onDelete: Cascade)
491   buyer   Buyer
492   itemId  String
493   itemType ItemType
494   createdAt DateTime @default(now())
495   @unique([buyerId, itemId, itemType])
496 }
497
498 // ===== 🔍 NOTIFICATION PREFERENCE =====
499 You, 4 weeks ago | Author (You)
500 model NotificationPreference {
501   userId          String @id @relation(fields: [userId], references: [id], onDelete: Cascade)
502   user             User
503   allowNotifications Boolean @default(true) // for all users
504   newMessageNotify Boolean @default(true) // for suppliers only
505   newOrderNotify  Boolean @default(true) // for buyers only
506   newInvoiceNotify Boolean @default(true) // for buyers only
507   newOfferNotify  Boolean @default(true) // for suppliers only
508   biddingStatusNotify Boolean @default(true) // for suppliers only
509   invoiceStatusNotify Boolean @default(true) // for suppliers only
510   orderStatusNotify Boolean @default(true) // for buyers only
511   groupPurchaseStatusNotify Boolean @default(true) // for buyers only
512   createdAt        DateTime @default(now())
513   updatedAt        DateTime @updatedAt
514 }
515
516 // ===== 📢 NOTIFICATION =====
517 You, 3 weeks ago | Author (You)
518 model Notification {
519   id      String @id @default(uuid())
520   senderUserId String @relation("SentNotifications", fields: [senderUserId], references: [id], onDelete: Cascade)
521   sender   User
522   receiverUserId String @relation("ReceivedNotifications", fields: [receiverUserId], references: [id], onDelete: Cascade)
523   receiver  User
524   type     NotificationType
525   title    String
526   content   String
527   isHead   Boolean @default(false)
528   readAt   DateTime?
529   entityId String
530   entityType NotificationEntityType
531   isDeleted Boolean @default(false)
532   createdAt DateTime @default(now())
533   updatedAt DateTime @updatedAt
}

```

Figure D - 13: Prisma Schema (Wishlist, NotificationPreference, Notification models)

```

535 // ===== 💬 CHAT =====
536 You, 3 weeks ago | 1 author (You)
537 model Chat {
538   id      String @id @default(uuid())
539   userId  String @relation("FirstUser", fields: [userId], references: [id])
540   user1   User
541   user2Id String @relation("SecondUser", fields: [user2Id], references: [id])
542   createdAt DateTime @default(now())
543   updatedAt DateTime @updatedAt
544   messages Message[]
545
546   @@unique([userId, user2Id])
547 }
548
549 // ===== 📬 MESSAGE =====
550 You, 3 weeks ago | 1 author (You)
551 model Message {
552   id      String @id @default(uuid())
553   chatId String @relation(fields: [chatId], references: [id], onDelete: Cascade)
554   chat    Chat
555   senderId String @relation("SentMessages", fields: [senderId], references: [id])
556   sender   User
557   receiverId String @relation("ReceivedMessages", fields: [receiverId], references: [id])
558   receiver  User
559   text    String? // null if it is just an image
560   imageFileName String?
561   isRead   Boolean @default(false)
562   readAt   DateTime?
563   createdAt DateTime @default(now())
}

```

Figure D - 14: Prisma Schema (Chat and Message models)

```

565 // ----- X ENUMS -----
566
567 enum UserRole {
568   GUEST
569   SUPPLIER
570   BUYER
571 }
572
573 enum TokenType {
574   EMAIL_VERIFICATION
575   PASSWORD_RESET
576   TOKEN
577 }
578
579 enum Languages {
580   AR
581   EN
582 }
583
584 enum SupplierPlan {
585   BASIC
586   PREMIUM
587 }
588
589 enum ItemType {
590   PRODUCT
591   SERVICE
592 }
593
594 enum SupplierStatus {
595   ACTIVE
596   INACTIVE
597 }
598
599 enum StoreStatus {
600   OPEN
601   CLOSED
602 }
603
604 enum GroupPurchaseDeadline {
605   THREE_DAYS
606   FIVE_DAYS
607   SEVEN_DAYS
608 }
609
610 enum ServiceAvailability {
611   TWENTY_FOUR_SEVEN
612   EVERYDAY
613   WEEKDAYS
614   WEEKENDS
615   APPOINTMENT
616 }
617

```

Figure D - 15: Prisma Schema
(Enums Part 1)

```

618 enum OrderStatus {
619   PENDING
620   PROCESSING
621   SHIPPED
622   COMPLETED
623 }
624
625 enum PreInvoiceStatus {
626   PENDING
627   FAILED
628   SUCCESSFUL
629 }
630
631 enum InvoiceStatus {
632   PENDING
633   ACCEPTED
634   REJECTED
635   FULLY_PAID
636   PARTIALLY_PAID
637 }
638
639 enum InvoiceTermsOfPayment {
640   PARTIAL
641   FULL
642 }
643
644 enum NotificationType {
645   NEW_MESSAGE
646   NEW_ORDER
647   NEW_REVIEW
648   NEW_INVOICE
649   NEW_OFFER
650   BID_STATUS_CHANGED
651   INVOICE_STATUS_CHANGED
652   ORDER_STATUS_CHANGED
653   GROUP_PURCHASE_STATUS_CHANGED
654 }
655
656 enum NotificationEntityType {
657   CHAT
658   ORDER
659   REVIEW
660   INVOICE
661   OFFER
662   RTD
663   GROUP_PURCHASE
664 }
665
666 enum GroupPurchaseStatus {
667   OPEN
668   CLOSED
669 }

```

Figure D - 16: Prisma Schema
(Enums Part 2)

```

671 enum BidStatus {
672   OPEN
673   CLOSED
674 }
675
676 enum BidExpectedResponseTime {
677   ONE_WEEK
678   TWO_WEEKS
679   FOUR_WEEKS
680   SIX_WEEKS
681 }
682
683 enum OfferStatus {
684   PENDING
685   ACCEPTED
686   DECLINED
687 }

```

Figure D - 17: Prisma Schema
(Enums Part 3)

Appendix E: Detailed Individual Usability Testing Results

This appendix presents the complete raw data collected from the ten usability testing sessions conducted using the Think-Aloud method. Each participant's performance is shown in a separate table, including task completion times, number of errors, participant comments, and specific suggestions for improvement.

A high-level summary, aggregated metrics, and overall satisfaction scores are provided in Chapter 6 (Section 6.2.4.4). The individual tables below offer full transparency and serve as supporting evidence for the findings and recommendations discussed in the main report.

Table E - 1: Usability Testing Results – Participant 1

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	2 min	0	They wondered about character limits for the business name. Expected a city dropdown instead of free-text field.	Replace free-text city with searchable dropdown, and display character counters for name fields
Add a payment method	48 s	1	Error message appeared far at the top, they had to scroll to see it, which felt disconnected from the form	Move validation errors inline next to relevant fields
Start/Join a group purchase	23 s	0	They were confused when	Display a message

			group purchase section was missing on some product pages	“Group purchasing not enabled by supplier” when the feature is disabled
View Invoices history	37 s	1	They could not find Invoices quickly; they expected direct header link instead of inside profile menu	Add direct header icons/links for navigation options inside the profile menu (like the Cart)
Add a product to the cart	17 s	1	They wanted an explanation of “case quantity”	Add tooltips next to case quantity, minimum and maximum order quantity fields
Pay for cart	34 s	1	Initially missed the cart icon in header despite it being visible	Add flying-item animation from “Add to Cart” button to cart icon to guide the eye
Check order status	11 s	0	“Completed” status felt unclear after confirming delivery	Change final status from “Completed” to “Delivered” for clearer meaning

Write a review	52 s	1	Hard to locate where to leave a review, they are unaware reviews are tied to delivered orders and fully paid invoices	Create dedicated “My Reviews” page showing pending, drafted, and submitted reviews with clear action buttons
Smart search a product	53 s	0	Everything was clear, they appreciated the clean interface	No improvements needed
Contact a supplier	4 s	0	Very intuitive, they found chat button instantly	No improvements needed
View invoice details	26 s	0	Clear and straightforward	No improvements needed
Pay for invoice	23 s	0	Process was smooth	No improvements needed
Create a new bid	38 s	1	Did not know where to create bids	Add quick-action cards or section on buyer homepage for recent/open bids with

				“Create New Bid” button
Switch role to supplier	8 s	0	Everything worked as expected	No improvements needed
Add a new product or service	3 min 21 s	1	Description field is plain text, the participant expected rich-text formatting (bold, lists, etc.)	Upgrade description editor to support basic rich-text formatting
Open the store	45 s	0	Found the toggle quickly	No improvements needed
Create an invoice	2 min	1	They felt confused because of the error message about total amount when price fields appear before items	Move total upfront and upon-delivery fields to the bottom, after the items table
View notifications	9 s	0	Clear and easy to access	No improvements needed
Join a bid	1 min 9 s	0	Process was clear, but term “bid/offer”	Add small tooltip icons explaining “Bid

			could be unfamiliar	= Request for Proposal” and “Offer = Supplier Proposal”
View received orders	14 s	0	Intuitive navigation	No improvements needed

Table E - 2: Usability Testing Results – Participant 2

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	2 min 17 s	0	They wanted password visibility toggle, and they expected input fields to have actual labels instead of placeholders only	Add “show password” eye icon, and add proper field labels above or beside input fields
Add a payment method	48 s	0	Process was smooth	No improvements needed
Start/Join a group purchase	49 s	0	Everything was clear	No improvements needed

View Invoices history	22 s	0	Found quickly and easily	No improvements needed
Add a product to the cart	12 s	0	They wanted to add to cart directly from product cards without entering product details page	Add “Quick Add to Cart” button on product cards
Pay for cart	23 s	0	Everything was clear	No improvements needed
Check order status	9 s	0	Everything was clear	No improvements needed
Write a review	47 s	1	They were unsure which invoices or orders allow reviews and where to find the review option	Add “Write Review” action buttons on allowed invoices and orders
Smart search a product	34 s	0	Initially unclear what “smart search” meant	Add a short guide or highlighted banner for AI-augmented features

Contact a supplier	6 s	0	Very fast and obvious	No improvements needed
View invoice details	15 s	0	Clear layout	No improvements needed
Pay for invoice	38 s	0	Smooth process	No improvements needed
Create a new bid	1 min 6 s	0	Completed without issues	No improvements needed
Switch role to supplier	15 s	0	Did not realize they could also act as suppliers, they assumed registration locked them as buyers only	Show a one-time modal after registration explaining users can become suppliers anytime
Add a new product or service	4 min 2 s	0	They wanted to preview how the listing appears to buyers	Add “Preview as Buyer” button during or after listing creation
Open the store	24 s	0	Expected a quick toggle directly on the supplier overview	Add “Open/Close Store” toggle button on supplier overview page

Create an invoice	1 min 19 s	1	The invoice item unit price numeric field start with 0 and is not auto cleared on focus which accidentally added extra zero to the actual price	Clear initial 0 when user focuses on the numeric input field
View notifications	8 s	0	Easy to access	No improvements needed
Join a bid	55 s	0	Completed successfully but unsure where to track status afterwards	After joining, show a confirmation dialog mentioning a pre-invoice is created and visible in the Invoices page
View received orders	15 s	0	Intuitive and clear	No improvements needed

Table E - 3: Usability Testing Results – Participant 3

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design

Sign up	1m	1	They were unsure whether the account was buyer-only or could later become supplier	Add a short note during or after registration: “You are registered as a buyer. You can open a supplier store anytime”
Add a payment method	55s	0	Process was straightforward	Didn't find a suggestion
Start/Join a group purchase	1m 20s	3	They needed time to understand how group purchases work and what benefit they provide	Add a tooltip icon on the group-purchase section explaining the concept and buyer benefit
View Invoices history	30s	1	They took a moment to locate Invoices in the profile dropdown	Add direct header icons/links for navigation options inside the profile menu (like the Cart)
Add a product to the cart	25s	0	Completed smoothly	Didn't find a suggestion
Pay for cart	50s	0	They noted the order summary lacks tax and per-supplier fee breakdown	Enhance order summary with tax line and supplier-specific subtotals
Check order status	30s	0	They found the status clear and easy to read	Didn't find a suggestion

Write a review	30s	1	They were unsure when and for what exactly a review could be written	After payment, display a friendly success message with a brief reminder (e.g., “Invoice paid. Please leave a review!” or “Order placed. Remember to confirm delivery to review it!”).
Smart search a product	40s	0	It was clear	Didn't find a suggestion
Contact a supplier	50s	0	They felt the contact button could be more emphasized	Add a larger, colored “Contact Supplier” button on product details page
View invoice details	25s	0	They found the information clear, but sections not visually grouped	Group related fields into labelled cards or sections (Invoice Info, Items, Notes & Terms)
Pay for invoice	45s	0	Process worked smoothly	Didn't find a suggestion
Create a new bid	1m	2	They did not understand what some fields were asking for	Add “!” tooltip icons next to each field with a short, clear explanation of what information is required

Switch role to supplier	10s	0	They did not immediately realize they were now acting as supplier	Display a small indicator near the profile: “Acting as Supplier” (or Buyer)
Add a new product or service	1m 20s	0	They felt image upload and description entry took slightly long	Offer AI-assisted title and description generation from images
Open the store	50s	0	It was clear	Didn't find a suggestion
Create an invoice	55s	2	They found invoice sections scattered (same layout issue as invoice details)	Group fields into clearly labelled cards or sections (same improvement as invoice details)
View notifications	10s	0	It was clear and well organized	Didn't find a suggestion
Join a bid	1m	1	The bidding deadline may be shown in relative time for improved clarity	Show relative time next to deadline (e.g., “3 days 5 hours left”)
View received orders	35s	0	It was clear	Didn't find a suggestion

Table E - 4: Usability Testing Results – Participant 4

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design

Sign up	1 m	0	Everything was clear and intuitive	Nothing to adjust
Add a payment method	30 s	0	Very fast and straightforward	Nothing to adjust
Start/Join a group purchase	1 m	0	Understood quickly	Nothing to adjust
View Invoices history	40 s	0	Found easily	Nothing to adjust
Add a product to the cart	10 s	0	Super fast and intuitive	Nothing to adjust
Pay for cart	20 s	0	Smooth and clear process	Nothing to adjust
Check order status	30 s	0	Status was very clear	Nothing to adjust
Write a review	1 m	0	Everything was clear	Nothing to adjust
Smart search a product	10 s	0	They expected autocomplete results	Enable autocomplete suggestions in smart search
Contact a supplier	15 s	0	Chat button was obvious	Nothing to adjust
View invoice details	45 s	0	Page was clear	Nothing to adjust
Pay for invoice	20 s	0	Payment went smoothly	Nothing to adjust
Create a new bid	10 s	0	Very quick and intuitive	Nothing to adjust

Switch role to supplier	10 s	0	Role switch was clear	Nothing to adjust
Add a new product or service	15 s	0	Form was easy to fill	Nothing to adjust
Open the store	1 m	0	Everything was clear	Nothing to adjust
Create an invoice	1 m	0	Invoice creation was straightforward	Nothing to adjust
View notifications	5 s	0	Notifications very easy to access	Nothing to adjust
Join a bid	5 s	0	Joining process was fast	Nothing to adjust
View received orders	20 s	0	Orders page clear and intuitive	Nothing to adjust

Table E - 5: Usability Testing Results – Participant 5

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	1 m	0	Everything was clear	Auto-login after email verification instead of returning to the login page
Add a payment method	1 m	0	Very smooth	Nothing to adjust

Start/Join a group purchase	40 s	0	Understood perfectly	Nothing to adjust
View Invoices history	30 s	0	Found instantly	Nothing to adjust
Add a product to the cart	10 s	0	Super fast	Nothing to adjust
Pay for cart	30 s	0	Payment process was clear	Nothing to adjust
Check order status	45 s	0	Status very visible	Nothing to adjust
Write a review	1 m	0	Review page easy to fill	Nothing to adjust
Smart search a product	10 s	0	Search worked perfectly	Nothing to adjust
Contact a supplier	30 s	0	Chat feature was easy	Nothing to adjust
View invoice details	50 s	0	Details well presented	Nothing to adjust
Pay for invoice	1 m	0	Payment went smoothly	Nothing to adjust
Create a new bid	15 s	0	Bid creation was quick	Nothing to adjust
Switch role to supplier	15 s	0	Role switch intuitive	Nothing to adjust
Add a new product or service	20 s	0	Form easy and clear	Nothing to adjust

Open the store	30 s	0	Store toggle found fast	Nothing to adjust
Create an invoice	1 m	0	Invoice creation straightforward	Nothing to adjust
View notifications	15 s	0	Notifications very clear	Nothing to adjust
Join a bid	15 s	0	Joining was instant	Nothing to adjust
View received orders	1 m	0	Orders page very intuitive	Nothing to adjust

Table E - 6: Usability Testing Results – Participant 6

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	1 m	0	Everything clear and fast	Nothing to adjust
Add a payment method	2 m	0	Process was smooth	Nothing to adjust
Start/Join a group purchase	1 m	0	Everything was clear	Nothing to adjust
View Invoices history	1m	0	Found the page beautiful	Nothing to adjust
Add a product to the cart	15 s	0	Very fast and intuitive	Nothing to adjust
Pay for cart	40 s	0	Payment clear	Nothing to adjust

Check order status	30 s	0	Status very visible	Nothing to adjust
Write a review	2 m	0	Review was easy	Nothing to adjust
Smart search a product	20 s	0	Search worked perfectly	Nothing to adjust
Contact a supplier	20 s	0	Chat button obvious	Nothing to adjust
View invoice details	1 m	0	Page looked nice	Nothing to adjust
Pay for invoice	1 m	0	Payment smooth	Nothing to adjust
Create a new bid	20 s	0	They wanted to upload an image with the bid	Allow image attachment when creating a bid
Switch role to supplier	20 s	0	Role switch clear	Nothing to adjust
Add a new product or service	30 s	0	Form very intuitive	Nothing to adjust
Open the store	20 s	0	Toggle found fast	Nothing to adjust
Create an invoice	1 m	0	Invoice creation easy	Nothing to adjust
View notifications	15 s	0	Notifications clear	Nothing to adjust

Join a bid	15 s	0	They expected to be able to search or filter available bids	Add search and filter options on the biddings page
View received orders	1 m	0	Orders page very clear	Nothing to adjust

Table E - 7: Usability Testing Results – Participant 7

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	2 m 40 s	0	Completed registration very smoothly	None
Add a payment method	56 s	0	Completed registration very smoothly	None
Start/Join a group purchase	5 s	0	Joined a group purchase in seconds	None
View Invoices history	10 s	0	Invoices list appeared immediately	None
Add a product to the cart	10 s	0	Added to cart very fast	None
Pay for cart	27 s	0	Checkout was quick and clear	None
Check order status	9 s	0	Status was very easy to see	None

Write a review	26 s	0	Found the review button instantly	None
Smart search a product	11 s	0	Search results were clear	None
Contact a supplier	26 s	0	Chat opened right away	None
View invoice details	15 s	0	All invoice information was perfectly organized	None
Pay for invoice	50 s	0	Invoice payment went perfectly	None
Create a new bid	1 m 17 s	0	Created the bid very quickly	None
Switch role to supplier	6 s	0	Switched role in one click	None
Add a new product or service	2 m 23 s	0	Listing form was very easy to complete	None
Open the store	1m 16s	0	Store opened immediately	None
Create an invoice	2m 37s	0	Invoice created without any issues	None
View notifications	10 s	0	Notifications loaded instantly	None
Join a bid	51 s	0	Joined the bid smoothly	None
View received orders	30 s	0	Orders page was very clear	None

Table E - 8: Usability Testing Results – Participant 8

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	2m 43s	0	Registration felt fast and simple	None
Add a payment method	1 m	0	Card added in seconds	None
Start/Join a group purchase	7 s	0	Group purchase option was obvious	None
View Invoices history	7 s	0	Found invoices instantly	None
Add a product to the cart	7 s	0	Added to cart with one click	None
Pay for cart	29 s	0	Payment process was very smooth	None
Check order status	9 s	0	Status stood out clearly	None
Write a review	21 s	0	Review button was easy to locate	None
Smart search a product	13 s	0	Smart search worked perfectly	None
Contact a supplier	11 s	0	Chat button very visible	None
View invoice details	8 s	0	Invoice details loaded fast and clean	None

Pay for invoice	35 s	0	Invoice paid without any problem	None
Create a new bid	1m 17s	0	Bid created quickly and easily	None
Switch role to supplier	6 s	0	Role changed instantly	None
Add a new product or service	1m 35s	0	Product form was intuitive	None
Open the store	50 s	0	Store opened with a single click	None
Create an invoice	2m 44 s	0	Invoice creation was straightforward	None
View notifications	6 s	0	Notifications appeared immediately	None
Join a bid	38 s	0	Joined the bid in seconds	None
View received orders	9 s	0	Orders very well organized	None

Table E - 9: Usability Testing Results – Participant 9

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	3m 30s	0	The user appreciated the flexibility; if they needed to go back to change a specific option, they returned to the step they were	No suggestions needed

			on, and all the data was preserved.	
Add a payment method	1m 41s	0	Process was fast and clear	No suggestions needed
Start/Join a group purchase	30s	0	Very quick and intuitive	Show a small “Group Purchase Available” badge on product cards (visible before clicking)
View Invoices history	6s	0	The user Loved everything and whole design	No suggestions needed
Add a product to the cart	7s	0	Very easy	No suggestions needed
Pay for cart	28s	0	Extremely easy	No suggestions needed
Check order status	12s	0	The user Loved the design of the interface	No suggestions needed
Write a review	1m 3s	0	The interface design is very beautiful, and the user liked the way the review was split between the product and the supplier on the same page.	No suggestions needed

Smart search a product	5s	0	The user was impressed by the high speed and accuracy of the search.	No suggestions needed
Contact a supplier	15s	0	Excellent experience	When buyer starts a new chat, show quick pre-written questions set by the supplier
View invoice details	15s	0	Everything clear and well presented	No suggestions needed
Pay for invoice	20s	0	Smooth and fast	No suggestions needed
Create a new bid	1m 9s	0	Very straightforward	Allow adding more optional details or images to bids for richer submissions
Switch role to supplier	5s	0	Very clear	No suggestions needed
Add a new product or service	2m 32s	0	The user found adding a service to be very easy and enjoyable	No suggestions needed

Open the store	48s	0	No clear feedback after clicking “Save Changes” when opening the store, left them unsure if it saved	Show a visible success toast (“Store is now open!”) after saving changes
Create an invoice	2m 42s	0	The user expected this process to take longer than signing up, but was surprised that it took less time	No suggestions needed
View notifications	4s	0	It was immediate	No suggestions needed
Join a bid	1m 35s	0	Process was smooth	No suggestions needed
View received orders	10s	0	Very clear and well organized	No suggestions needed

Table E - 10: Usability Testing Results – Participant 10

Task Name	Task Time	Number of Errors	Comments	Suggestion to improve Task Design
Sign up	3m 20s	0	The process of creating an account felt smooth and professionally guided.	No suggestions needed
Add a payment method	1m 50s	0	Card added quickly	Show a short tip: “Check your bank’s online payment

				policy before adding a card”
Start/Join a group purchase	32s	0	The feature is very simple to initiate and understand.	No suggestions needed
View Invoices history	7s	0	The history loaded immediately, and the titles were very clear.	No suggestions needed
Add a product to the cart	8s	0	The button location was intuitive and easy to find.	No suggestions needed
Pay for cart	26s	0	The payment gateway integrated smoothly and confirmed the purchase instantly.	No suggestions needed
Check order status	10s	0	The status update was prominently displayed and easy to read.	No suggestions needed
Write a review	1m 5s	0	Review process was clear	The user suggested that the platform should offer general tips on how to write a balanced and constructive review.

Smart search a product	5s	0	The search results page was remarkably quick and relevant.	No suggestions needed
Contact a supplier	14s	0	Chat opened instantly	No suggestions needed
View invoice details	16s	0	The level of detail provided in the invoice was exactly what the user needed.	No suggestions needed
Pay for invoice	22s	0	Payment was fast and smooth	The user mentioned that having a personal reminder on their private calendar for payment deadlines is a useful practice.
Create a new bid	1m 12s	0	The interface for creating a bid was organized and logical.	No suggestions needed
Switch role to supplier	6s	0	The role switch was extremely fast, and the change in interface was noticeable.	No suggestions needed
Add a new product or service	2m 35s	0	The user found the form design to be engaging and efficient.	No suggestions needed

Open the store	45s	0	The user praised the clear structure of the store settings page.	Show a celebratory modal (“Your store is now open, good luck!”) after first opening
Create an invoice	2m 30s	0	It was very enjoyable, and the steps were logically arranged.	No suggestions needed
View notifications	5s	0	The notifications badge updated instantly upon arrival.	No suggestions needed
Join a bid	1m 30s	0	The steps to join were clear and required minimal effort.	No suggestions needed
View received orders	11s	0	The list of received orders was immediately accessible and sortable.	No suggestions needed