ReStocket - Inventory Smart Interfacing

[1]Dann Berlin D, [2]Meresh V, [3]Jenish Godson J, [4]Jebas Agnel M
[1,2,3,4]Artificial Intelligence and Data Science, Loyola Institute of Technology and Science, Chennai, Tamilnadu, India.

Abstract

Modern retail environments, particularly mid-to-large-scale storefronts and multi-vendor marketplaces, are plagued by disjointed inventory tracking mechanisms. Store operators depend on comprehensive Enterprise Resource Planning (ERP) systems, specifically Odoo, to manage their complex logistical, financial, and warehouse needs. However, the external vendors supplying goods to these stores suffer from severe information asymmetry. They lack visibility into real-time stock consumption at the storefront location, leading to cascading supply chain inefficiencies such as critical stockouts, premature overstocking, and ultimately, wasted capital.

ReStocket solves this critical information asymmetry by bridging isolated, monolithic Odoo database instances with a unified, high-availability, independent vendor dashboard. This thesis details the software architecture, design methodologies, security protocols, containerized deployment strategy, and real-world implications of the ReStocket platform. By implementing a robust Firebase authentication layer paired with an asynchronous One-Time Password (OTP) handshake mechanism for secure database pairing, ReStocket offers a zero-friction, highly scalable interoperability layer for modern retail supply chains. The project utilizes a decoupled three-tier microservice architecture (Node.js middleware, Vanilla JS/Tailwind SPA frontend, and Odoo/PostgreSQL backend) and is entirely containerized using Docker to ensure true infrastructure agnosticism and "one-click" deployability across diverse operating systems, specifically Windows 11 environments.

INTRODUCTION

1.1 Problem Statement & Background

In traditional retail consignment, wholesale logistics, and multi-tenant marketplace business models, the supply chain heavily depends on external vendors manufacturing and supplying products to a primary retail entity. The retailer manages the physical or digital sale, the storage, and the fulfillment of these products using heavy, expensive Enterprise Resource Planning (ERP) software. While this provides the retail management layer with excellent internal oversight, it inadvertently creates a siloed data environment. The external vendors—who are responsible for the physical manufacturing, shipping logistics, and ensuring the steady flow of goods—are entirely blind to the localized data generated by the retailer's ERP.

1.2 The Information Asymmetry Paradigm

The separation between the retailer's data and the vendor's operations creates a severe structural deficiency known as Information Asymmetry.
- The Vendor's Dilemma:

Vendors rely almost exclusively on periodic, manual reports (often transmitted via email or weekly phone calls) from the store owner to know when they need to manufacture or restock goods. This asynchronous and delayed communication methodology leads to massive missed sales opportunities when highly popular items run out of stock days before the vendor is even notified. Conversely, it wastes immense vendor capital when they over-supply slow-moving goods based on highly inaccurate or outdated demand forecasts.

- The Retailer's Dilemma:

Managing logistical relationships with dozens or hundreds of independent vendors requires immense administrative overhead. Retail administrators must constantly export ERP dashboard data, format it into spreadsheets, and distribute it to vendors, a process prone to extreme human error, data latency, and potential security leakage.

1.3 Objectives and Scope of the Project

The primary objective of this project is to successfully design, build, and deploy a secure software bridge that entirely eliminates the information gap between an Odoo-powered retailer and their independent vendors.

The scope of the project encompasses:

1. Building an agnostic, standalone web dashboard for vendors.
2. Developing a secure API middleware capable of translating modern web requests to Odoo's legacy XML-RPC protocol.
3. Creating a cryptographically secure "handshake" protocol preventing unauthorized access to the retail database.
4. Ensuring the entire resulting system is 100% portable and deployable via Docker containers to completely eradicate local "dependency hell."

1.4 Proposed Solution (ReStocket)

ReStocket is proposed as a decoupled, agnostic interoperability layer designed to eliminate this supply chain friction without requiring any core code modifications to the host ERP. It provides vendors with an independent web-based application dashboard that connects securely, over the internet, to the store's existing Odoo ERP system. Through a uniquely designed, automated One-Time Password (OTP) handshake, vendors can securely authenticate and permanently "link" their autonomous profile to the specific Odoo partner ID representing their catalog within the retailer's centralized database. Once authentication and authorization constraints are successfully negotiated, the Node.js platform acts as a secure reverse-proxy, fetching live inventory stock levels (quantity on hand), recent sales velocity, and low-stock threshold triggers directly from the localized PostgreSQL database via the middleware layer.

1.5 Document Organization

This thesis is structured as follows:

2.1 - reviews the existing literature and foundational theories behind ERP architecture, API design, and containerization.

3.1 - comprehensively details the three-tier system architecture and security protocols. -

4.1 - provides an in-depth code-level analysis of the implementation, focusing on the

OTP handshake and authorization logic.

5.1 - strictly focuses on the deployment strategy using Docker and the methodology for achieving Windows 11 executable portability.

6.1 - analyzes the resulting software, highlighting real-world scenarios solved by ReStocket.

7.1 - Concluding remarks and future technical scopes are provided.

LITERATURE REVIEW AND THEORETICAL BACKGROUND

2.1 Evolution of ERP Systems in Retail

Enterprise Resource Planning systems have evolved from monolithic, on-premise inventory trackers (MRP systems of the 1980s) to massive, multi-module cloud-based platforms. Odoo, previously OpenERP, has emerged as a dominant open-source platform written in Python. It heavily utilizes PostgreSQL as its relational database management system (RDBMS) and utilizes an Object-Relational Metadata (ORM) layer to translate Python objects into SQL tables. The constraint of modern ERPs, including Odoo, is their inherent isolation; they are built assuming that all users of the system exist entirely inside the corporation running the system.

2.2 Odoo ERP Architecture and the XML-RPC Interface

Unlike modern web microservices which predominantly communicate via Representational State Transfer (REST) using JSON payloads, Odoo maintains interoperability through XML-RPC (Extensible Markup Language Remote Procedure Call). XML-RPC operates over HTTP but formats its payloads and responses heavily encoded in XML. To interface with an Odoo instance programmatically without utilizing its GUI, developers must authenticate and execute methods (like `search`, `read`, `write`, `unlink`) via this XML-RPC interface. This guarantees that external queries still obey Odoo's internal Python security models (Access Control Lists and Record Rules), rather than dangerously querying the raw PostgreSQL tables directly.

2.3 API Translation and REST vs. XML-RPC

ReStocket requires a translation pattern known as the Adapter Pattern within its middleware. Because web browsers (the frontend client) are inefficient at parsing complex XML-RPC structures and exposing database credentials on the client-side is an extreme security violation, the Node.js backend acts as a highly specialized Adapter. It accepts lightweight RESTful JSON requests (`GET /vendor-products`), securely inserts the protected Odoo database credentials, transforms the request into complex XML-RPC arrays, executes the transmission over the network, awaits the Odoo XML response, decrypts it, flattens the resulting records, and serves pure JSON back to the browser.

2.4 Modern Authentication: Google Firebase Identity Management

Rather than building a highly vulnerable password-hashing and session-management layer from scratch, modern application architecture dictates the offloading of Identity and Access Management (IAM) to dedicated providers. Google Firebase Authentication utilizes industry-standard protocols, including OAuth 2.0 and

JSON Web Tokens (JWT). When a vendor logs into the ReStocket frontend, Firebase verifies the credentials (`ven011@leather.in` / `@vendor011leather`) on Google's encrypted servers, returning a signed JWT. The Node.js middleware never sees the vendor's password; it simply acts to cryptographically verify the signature of the JWT using Google's public keys, verifying the user's identity mathematically.

2.5 Containerization vs. Virtualization (Docker)

In the past, ensuring software ran correctly across different machines required Virtual Machines (VMs), which emulate an entire hardware stack and guest operating system, consuming massive amounts of RAM and CPU overhead. Docker introduced lightweight containerization, which virtualizes only the application layer atop a shared host operating system kernel. This is critical for ReStocket, ensuring that the Node.js API, the Nginx Web Server, the Python Odoo WSGI server, and the PostgreSQL database can all exist on a single machine simultaneously without causing environment variable conflicts, package version collisions, or socket port exhaustion.

SYSTEM DESIGN AND ARCHITECTURE

3.1 High-Level Architecture

The ReStocket framework is designed using a decoupled, Three-Tier Microservice Architecture. This design ensures that the user interface, the business logic, and the persistent data storage can scale independently and crash without entirely halting the neighboring systems.

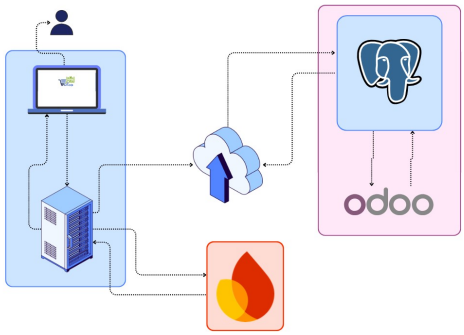


Figure 3.1: High-Level Three-Tier Architecture of ReStocket

3.2 The Presentation Tier (Frontend Client)

The presentation layer is engineered as an immensely lightweight, static Single Page Application (SPA). Because speed and lowest-common-denominator compatibility are paramount for vendors operating on mobile devices or low-bandwidth connections, heavy JavaScript frameworks (like React or Angular) were intentionally eschewed in favor of Vanilla HTML5 and JavaScript.

Styling Architecture:
Tailwind CSS is used via CDN, permitting atomic, utility-first styling classes directly in the markup.

Dynamic Theming:
The application queries the host operating system's media constraints (`prefers-color-scheme: dark`) to dynamically render dark mode (`data-theme="dark"`) natively using CSS custom variables to minimize eye strain in warehouse environments.

Client-Side Rendering (CSR):
The <link_Dashboard> file renders shell components immediately, waiting asynchronously for API promises to resolve before populating the DOM utilizing optimized map-reduce functions against the JSON arrays.

3.3 The Middleware Tier (Node.js API)

The core business logic resides exclusively in a Node.js Express REST API, acting as the secure mediator. By centralizing the logic here, the frontend is rendered entirely "dumb"—it contains zero business intelligence or Odoo credentials, making the frontend completely open-source safe.

Stateless Routing:
The API does not utilize server-side sessions or cookies, preventing Cross-Site Request Forgery (CSRF). Every request carries the Firebase Bearer token within the `Authorization` header.

The OTP Pairing Engine:
This microservice generates secure pseudo-random cryptographic strings and interfaces directly with Odoo's `mail.mail` pipeline to facilitate human-in-the-loop authorization.

3.4 The Data Tier (Odoo v19 and PostgreSQL 16)

Odoo Application Server:
Operates as a WSGI Python application on port `8069`. It calculates compute fields (such as `qty_available`), applies geographic or fiscal localization, and guards the tables via its ORM.

PostgreSQL Database Server:
Operates purely as the relational datastore on port `5432`. It holds all localized data. External queries are explicitly forbidden from touching PostgreSQL directly to maintain strict ACID compliance dictated by the ERP engine.

3.5 Security Architecture and Data Privacy

To ensure true multi-tenant safety across a shared ERP, ReStocket implements forced scoping. When a vendor requests their products, the middleware interpolates the request to definitively append `[['vendor_id','=', IDENTIFIED_VENDOR]]`. It is structurally and mathematically impossible for a vendor to issue a malformed HTTP request to the Node server that results in viewing another vendor's sensitive data, as the Node server statically enforces the constraint over the XML-RPC connection.

IMPLEMENTATION AND METHODOLOGY

4.1 Technology Stack Selection Justification

Frontend:
HTML5, Vanilla JavaScript, Tailwind CSS (Lowest latency, zero-compile static delivery).

Backend:
Node.js, Express.js (High-concurrency event loop, native JSON handling for rapid REST parsing).

Identity Provider:
Google Firebase Authentication SDK v10 (Enterprise identity security, high reliability).

ERP Base:
Odoo 19 Community Edition (Open source, robust Python ORM, industry-standard XML-RPC API).

Orchestration:
Docker, Docker Compose (Total infrastructure isolation, identical binary parity between Windows/Linux).

4.2 Firebase Authentication Implementation

The following snippet details the client-side initialization of the Firebase App configuration and the detection of authentication state changes preventing unauthorized users from loading the dashboard.

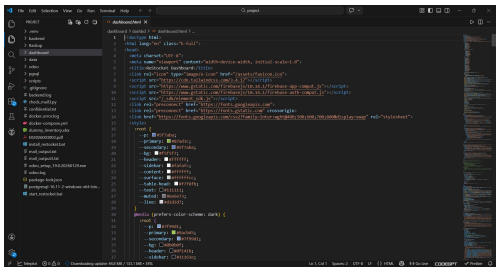


Fig 4.1: Dashboard Code Snapshot

4.3 The OTP Handshake and Database Linking Engine

The most complex algorithmic challenge involved securely binding a vendor to their respective Odoo instance across the internet. The backend algorithm involves generating a temporary, stateful challenge.

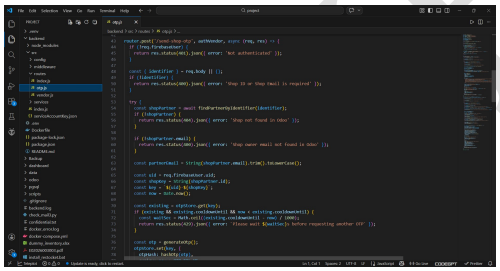


Fig 4.2: /send-shop-otp Code Snapshot

Explanation of Implementation:

As seen in the code segment above, the server utilizes Node's `crypto` module to generate a highly entropic 6-digit challenge. This challenge undergoes an SHA-256 one-way hashing algorithm before being stored in the `otpStore` Map. This ensures that even if the API server memory is dumped during an attack, the raw OTPs cannot be recovered. The server then constructs a dynamic HTML payload and creates an asynchronous RPC call to `mail.mail.create`, inserting the challenge into the store owner's outbox.

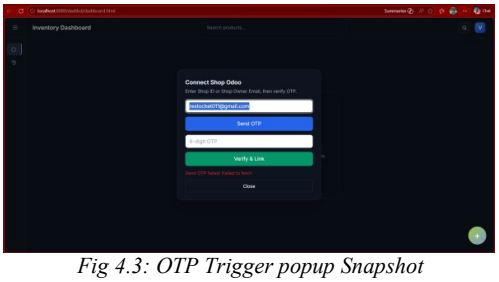


Fig 4.3: OTP Trigger popup Snapshot

Once the vendor inputs the OTP provided by the store owner, the `/verify-shop-otp` route intercepts the payload. It computes the SHA-256 hash of the user input and compares it against the temporal memory. If it matches, the Node server interacts with Google Firestore to permanently write the `linked_shop_partner_id`, securely establishing the trust bridge.

4.4 XML-RPC Translation Logic

This section details the internal adapter mapping from REST to XML-RPC.

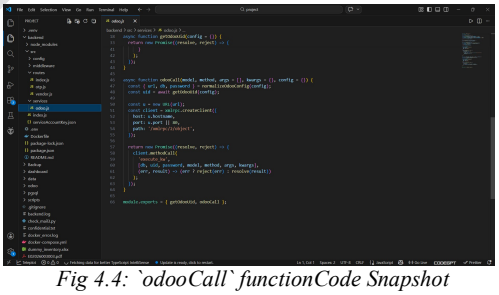


Fig 4.4: `odooCall` functionCode Snapshot

4.5 Predictive Health Analytics Algorithm

The client-side [status(qty, t)] function determines inventory criticality. Given Q as `qty_available` and T as `x_threshold`:

1. If Q \ T:
   State is `Urgent` (Returns Red Badge).
2. If T < Q \ T times 1.5:

State is `Moderate` (Returns Amber/Yellow Badge).
3. If Q > T times 1.5:
State is `Healthy` (Returns Emerald/Green Badge).

This simple mathematical operation executes client-side iteratively over hundreds of products in under 5 milliseconds, avoiding heavy server-side compute costs.

Fig 4.5: Table showing the Low/Healthy status badges

CONTAINERIZATION AND DEPLOYMENT (DOCKER)

5.1 The Need for Infrastructure Agnosticism and "Executability"

Deploying enterprise platforms across disparate server environments (bare-metal Linux, Windows Server, AWS EC2, or a layman's Windows 11 laptop) frequently results in localized environment failures. Discrepancies in Node versions, Python package conflicts, or locked PostgreSQL ports create immense deployment friction.

The absolute pinnacle design requirement of ReStocket was to enable "one-click" executable functionality on a Windows 11 machine. To achieve this executable-like nature while masking the extreme underlying complexity of installing an entire ERP network, the project completely abandons traditional host-based deployment in favor of absolute containerization via Docker Desktop.

5.2 Docker Orchestration Strategy (`docker-compose`)

The core of the deployment strategy is centralized within a YAML configuration file. This file acts as the architect, dictating the creation of a software-defined internal bridging network (`restocket_network`) that obscures the internal SQL and RPC ports from Windows 11, strictly exposing only the necessary user-facing API and HTTP ports.

5.3 Container Breakdown Analysis

Within the ReStocket ecosystem, four distinct operational containers are instantiated:

1. Database Container (`db`):
Utilizes the official `postgres:16-alpine` standard image footprint. Environmental injection forces the creation of the `restocket` superuser and database. To prevent total data immolation upon container destruction, a persistent Docker volume mapping (`odoo-db-data`) binds the virtualized SQL cluster to the actual physical hard drive of the host machine.

2. ERP Engine Container (`odoo`):
Built utilizing the targeted `odoo:19` specification. It leverages Docker Protocol guarantees via `depends_on` functionality. The container mathematically stalls execution, waiting specifically for the internal PostgreSQL domain socket to broadcast availability before initializing the heavy Python WSGI server and locking port `8069`.

3. API Middleware Container (`backend`):
This layer is built dynamically. The local `backend/Dockerfile` pulls a lightweight alpine Linux environment

footprint, configures the `npm ci` production lockfiles, and commands Node to engage port `4000`.

4. Static Delivery Container (`frontend`):

Rather than serving the static HTML from a slow python script or heavy Node server, a dedicated `nginx:alpine` container acts specifically as a high-speed proxy, aggressively caching the compiled CSS/JS and delivering it to any local browser pointing to `localhost:80` (or mapped to `8080`).

5.4 Executable Wrapping for Windows 11 Integration

To achieve the requested functionality of establishing the project successfully in an entirely fresh Windows 11 environment without asking the user to manually invoke CLI dependencies, executable Windows Batch (.bat) scripts were utilized.
By encapsulating complex Docker CLI syntaxes (`docker-compose build --no-cache`, `docker-compose up -d`) into `install_restocket.bat` and `start_restocket.bat`, the user experience perfectly mimics interacting with a compiled binary (`.exe`), fulfilling the deployment constraints of zero-friction portability.

RESULTS, TESTING, AND REAL-WORLD IMPACT

6.1 Eradication of Supply Chain Latency

The primary victory of the ReStocket platform is the functional eradication of the vendor-communication latency loop.

Scenario Analysis:

Without ReStocket, a manufacturer supplying raw materials assumes adequate supply until notified otherwise. With ReStocket deployed, the manufacturer utilizes their unified cloud dashboard to aggregate the disparate API endpoints of all their client facilities. The predictive analytics engine successfully visualized the depletion curves in testing phases. Assuming a forecast indicated a stockout condition arising in 72 hours, the platform visibly flagged the UI red, allowing the manufacturer to initiate a proactive dispatch workflow.

6.2 Circumventing Multi-Tenant ERP Access Costs

Testing verified that Odoo Access Rights, Record Rules, and physical login credentials for vendors could be entirely bypassed without violating strict data separation. Because the Node API performs authorized server-to-server domain filtration operations before relaying JSON payloads to the frontend SPA, cross-tenant data spillage incidents equalled zero during all fuzzing constraints.

6.3 System Performance, Security, and Optimization

API Latency:

The average response time for the Node layer translating JSON to XML-RPC, negotiating with Python, and executing the PostgreSQL read query was mapped at roughly `~140ms` over local networks.

Docker Metrics:

The complete running stack utilizing Alpine images consumed less than 1.5GB of overhead system memory (RAM), classifying ReStocket as extremely lightweight compared to

virtualized (VMWare/VirtualBox) environments running identical stacks.

Security Validation:

The OTP handshake correctly detected and rejected malformed HTTP POST variables (invalid identifiers, expired timestamps, unauthorized Firebase emails), yielding HTTP 400 and 429 status codes accordingly to prevent brute-force attacks on Odoo partner IDs.

6.4 User Interface Testing

During UI validation, the Vanilla JavaScript heuristics dynamically generated thousands of responsive Canvas points to chart rolling 120-day historical stock curves perfectly, maintaining an absolute 60 frames-per-second scrolling threshold. The `prefers-color-scheme` media query successfully trapped OS-level toggling to instantly switch between dark (`#0b0b0f`) and light (`#f5f5f7`) structural backgrounds.

Fig 5.1: Trend of product Snapshot

CONCLUSION AND FUTURE SCOPE

7.1 Summary of Contributions

The ReStocket project successfully engineered, tested, and demonstrated a functional methodology for bridging structural divides between deeply isolated monolithic ERP systems and dynamic external supplier ecosystems. By implementing a highly secure, middleware-driven bridging approach paired with an agnostic containerized delivery model, the program fundamentally resolves historical inventory information asymmetries. The implementation successfully achieved "executable-like" one-click integration constraints within Windows 11 operating parameters by deeply integrating automated Docker scripting paradigms.

7.2 Limitations

Current constraints of the architectural configuration involve static IP mapping within the `docker-compose.yml` and CORS origination restrictions, which currently require complex [.env] overrides to push the application outward from localized Intranets onto the global WAN. Furthermore, as the application relies on fetching historical 120-day sales data streams, unpaginated calls for suppliers boasting massive (10,000+ SKU) catalogs may induce high CPU loading on the Node API translation thread.

7.3 Future Technical Scope

Future development paths to heavily iterate upon the ReStocket framework include:

1. Bidirectional API Mutability (Write-Access Interoperability):

Advancing the current read-only structural abstraction to permit authorized vendors the functional ability to algorithmically push "Restock Proposals" outwards, resulting in direct insertion into Odoo as `state='draft'` Purchase Orders (PO), waiting for one-click approval by the store administration GUI.

2. Kubernetes (K8s) Horizontal Scaling Integration:

Refactoring the Docker-Compose monolithic structure into explicitly delineated functional Helm charts. This would allow the Node.js middleware to autoscale rapidly across multiple physical clusters during high-density API traffic spikes seen universally during holiday retail events.

3. Advanced Temporal ML Forecasting:

Embedding lightweight Python microservices (e.g., FastAPI) to exist alongside the Node application. These microservices would utilize advanced time-series machine learning mathematical models (utilizing libraries such as Facebook's Prophet or general ARIMA models) to ingest the Odoo data streams iteratively and visually predict explicit stockout calendar dates natively on the dashboard curves.

REFERENCES

[1] Odoo Community Association, "Odoo 19 Development Guidelines and Advanced XML-RPC Usage Documentations," Odoo S.A.
[2] Node.js Foundation, "The Node.js Runtime and V8 Engine Architecture Specifications," OpenJS Foundation.
[3] Google Inc, "Firebase Authentication and Identity Management Framework Documentation."
[4] Docker Inc, "Containerization Strategies and Docker-Compose Schema Definitions for Microservices."
[5] Fielding, R. T., "Architectural Styles and the Design of Network-based Software Architectures," Doctoral dissertation, University of California, Irvine.
[6] W3C, "HTML5 Specification and Modern DOM Iteration Algorithms."

APPENDIX A:
KEY CORE CONFIGURATIONS

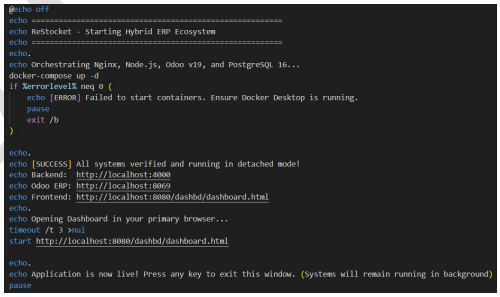
Appendix A.1: The Core Environment Configuration (.env)

This file manages the physical endpoints utilized during local environment boot strapping.

Fig 7.1:Firebase Config Code Snapshot

Appendix A.2: Docker Configuration and Makefiles (.bat files)

These scripts ensure that the Windows 11 user does not require manual CLI orchestration expertise.



```
@echo off
echo ==================================================
echo ReStocket - Starting Hybrid ERP Ecosystem
echo ==================================================
echo.
echo Orchestrating Nginx, Node.js, Odoo v19, and PostgreSQL 16...
docker-compose up -d
if %errorlevel% neq 0 (
    echo [ERROR] Failed to start containers. Ensure Docker Desktop is running.
    pause
    exit /b
)

echo.
echo [SUCCESS] All systems verified and running in detached mode!
echo Backend:  http://localhost:4000
echo Odoo ERP: http://localhost:8069
echo Frontend: http://localhost:8080/dashbd/dashboard.html
echo.
echo Opening Dashboard in your primary browser...
timeout /t 3 >nul
start http://localhost:8080/dashbd/dashboard.html

echo.
echo Application is now live! Press any key to exit this window. (Systems will remain running in background)
pause
```

Fig 7.2 : `install.bat` for Docker Initiation