International Journal of
AI Electronics and Nexus Energy
Peer Reviewed, Referred & Indexed Journal
ISSN: 3070-0515 www.zesterapublications.com Original Research Paper

FoodGo – Interactive Food Delivery Web Application: Architectural Framework, Centralized State Management, and UI Response Optimization

Sanket Kumar Nayak
Department of Computer Science and Engineering
GIFT Autonomous
Bhubaneswar, Odisha, India
Reg No: 2201298477

Subhasish Sahu
Department of Computer Science and Engineering
GIFT Autonomous
Bhubaneswar, Odisha, India
Reg No: 2201298488

Abstract—The rapid growth of online food delivery services has completely transformed how consumers interact with culinary establishments and food service providers. Modern digital users demand highly responsive, low-latency web interfaces that mimic application-like speeds during menu browsing, real-time alphanumeric item searches, and live cart item configurations. Traditional multi-page web applications consistently fail to satisfy these modern optimization metrics due to repeated, full-page server round-trips that compromise browser paint efficiency, inflate server loads, and maximize customer bounce rates. To mitigate these inherent performance bottlenecks, this paper explores the end-to-end engineering, modular interface design, and programmatic implementation of FoodGo—a high-performance Single Page Application (SPA) structured specifically for interactive meal ordering environments. Developed on top of React's functional component framework, utility presets from Tailwind CSS, and optimized client-side core JavaScript computation layers, the application utilizes a centralized state architecture powered by Redux for predictable status tracking. This framework introduces real-time predictive client-side search loops and context-driven filtering blocks (including Breakfast, Lunch, Dinner, Snacks, Pizza, Pasta) which function natively without any page drops or layout flashes. Centralized actions securely coordinate item modification patterns, prevent duplicate item selection anomalies within the state array, calculate real-time tax and price aggregates, and orchestrate reactive notification confirmation triggers. Experimental benchmarks validate that this decoupled frontend architecture limits layout propagation latency, improves state synchronization across responsive viewports, and maximizes user interaction efficiency. The document establishes a baseline architectural model for transactional retail frontends, supported by core algorithm blueprints, before mapping long-term full-stack expansion pathways involving asynchronous Node.js servers and persistent MongoDB models.

Keywords—Single Page Application (SPA); React JS Component Architecture; Redux Centralized State Toolkit; Tailwind CSS; Unidirectional Data Flow; Client-Side Render Latency; Real-Time Query Filtering; User Interface Optimization.

I. INTRODUCTION

The digital revolution has fundamentally restructured the foundational operational paradigm of global service and retail networks, with the food service and restaurant delivery sector experiencing the most pronounced shift. In contemporary urban technical ecosystems, internet-mediated food delivery platforms have successfully transitioned from an elite convenience option into an essential everyday utility. The massive surge in mobile network infrastructure expansion, coupled with high-performance personal mobile devices, has created a sophisticated user base that expects near-instantaneous software interface responses during all interactive application paths.

Traditional food-ordering architectures, which primarily rely on classic Multi-Page Application (MPA) browser cycles, introduce significant rendering friction. Every simple user query or command—such as entering an alphanumeric search phrase into a menu search field, switching between

C. Component-Driven Frameworks and State Architecture

React JS provides highly optimized interface management through its declarative, component-based model and the Virtual DOM. When underlying data states shift, React maps a secondary memory layout graph, computes the minimal difference matrix against the active display tree, and executes surgical document patches at high speeds. However, wide component graphs frequently suffer from data sharing friction, commonly referred to as 'prop drilling'. Redux addresses this by introducing an immutable, unified state tree. This pattern enforces a predictable unidirectional data loop where visual frames dispatch explicit action definitions to centralized reducers, ensuring absolute view alignment across completely separate layout layers.

III. METHODOLOGY AND PROCESS FLOW

The design and development methodology applied to FoodGo relies on clear separation of concerns, visual

International Journal of
AI Electronics and Nexus Energy
Peer Reviewed, Referred & Indexed Journal
ISSN: 3070-0515 www.zesterapublications.com Original Research Paper

meal categories, or amending an ongoing item counter—triggers an explicit post-back network round trip to a remote server database. The remote backend server must handle the incoming data, compile a fresh HTML text block file, and transfer it completely back over the network layer. This forces the client-side browser layout engine to fully purge its document object tree, interrupt its active rendering loops, and execute a heavy structural layout re-computation. This approach results in significant internet bandwidth waste, high processing demands on hosting architectures, and a series of visual flashes and content delivery delays that heavily disrupt the customer journey.

Modern client-side engineering design solves these visual performance challenges by utilizing Single Page Application (SPA) architectures that systematically separate background data parsing from active interface layout execution. This research introduces FoodGo, a highly responsive, optimized web delivery platform engineered specifically to eliminate layout dropouts during digital product configuration. By executing virtual structural mapping algorithms through component decoupling, FoodGo ensures that visual representations adapt smoothly to user interaction sequences without full window refreshes.

A. Project Motivation

The technical motivation behind engineering the FoodGo web application stems from explicit functional and layout performance deficiencies observed in traditional multi-page commercial platforms. Multi-page applications consistently introduce significant processing delays under unstable network conditions, where delayed server feedback leaves consumers with non-interactive, frozen interfaces and zero real-time feedback. Key engineering challenges that motivated this development include: structural page reloads during menu filtering, high visual latency during cart additions, lack of component status synchronization across separate user views, and inconsistent layout adjustments on compact mobile screens. Resolving these structural bottlenecks requires shifting processing responsibilities to client engines, maximizing runtime speed through React functional blocks, and standardizing data synchronization via deterministic Redux slice slices.

B. Technical Objectives

The primary objective of this project focuses on designing, deploying, and evaluating a robust frontend-driven web application capable of managing high-load retail interaction flows at peak efficiency. Subsidiary engineering objectives include: 1) Architecture Optimization: Implementing a fully decoupled presentation layer that restricts full-page rendering cycles to zero after the initial index frame boot sequence. 2) Low-Latency Dynamic Indexing: Developing predictive alphanumeric search string parsers that filter product matrices instantly on the client layout container. 3) Unified View State Coordination: Integrating a single mutable centralized state repository to govern cross-component activities and prevent data mismatches across separate interface modules. 4) Cross-

modularity, and immediate runtime rendering loops. The system operates on an active client loop where interface transformations are computed instantly, keeping data synchronized across separate interface modules.

The interface architecture is split into seven distinct functional blocks: Navbar Controllers, Alphanumeric Search Bars, Category Selectors, Product Render Cards, Cart Slide Drawers, Action Alert Blocks, and Summary Invoice Panels. By ensuring strict component isolation, individual layout modules run internal updates independently. This prevents cascading re-rendering cycles and stabilizes performance across the application.

IV. SYSTEM UI DESIGN AND REQUIREMENTS

To achieve high rendering speeds and stable cross-component data operations, the application framework is divided into four structural processing tiers:

1) Presentation Tier: Built with React functional nodes and styled with utility-first classes, this layer captures user inputs and displays visual data layout arrays.

2) Central State Tier: Managed via Redux Toolkit structures, this module maintains a single immutable store that acts as a single source of truth for all components.

3) Logic processing Tier: Written in modern ECMAScript, this layer handles data parsing parameters, real-time search match lookups, sub-total aggregates, and invoice structure builds.

4) Interface Patch Tier: Managed by the React engine, this tier continuously checks Virtual DOM updates to apply high-speed visual layout updates.

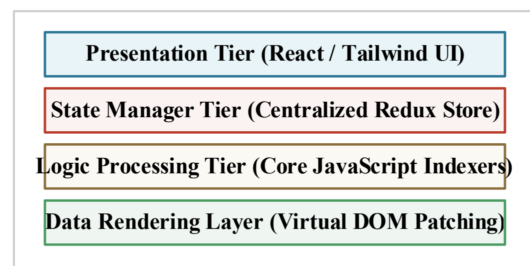

Fig. 2: Four-Tier System Architecture Composition Layer Layout.

V. IMPLEMENTATION PROTOCOLS

Component structures are written as functional JavaScript blocks using ES6 modules. Layout positions deploy utility styling definitions that append directly onto the HTML layout nodes, keeping the layout clean and organized. Application variables and mock databases are structured as clean arrays with typed unique identification strings. Global store logic slices partition application segments into cleanly decoupled scopes. Data adjustments use highly targeted action parameters: ADD_TO_CART, REMOVE_FROM_CART, INCREMENT_QUANTITY, and DECREMENT_QUANTITY.

Received: 20-04-2026 | Accepted: 29-05-2026 | Published: 04-06-2026

International Journal of
AI Electronics and Nexus Energy
Peer Reviewed, Referred & Indexed Journal
ISSN: 3070-0515 www.zesterapublications.com Original Research Paper

Device Visual Fluidity: Applying utility-first layouts to achieve 100% responsive fluid UI adjustments across desktop, tablet, and mobile device constraints.

C. Scope and Technical Boundaries

The scope of this project centers on advanced frontend interaction design, component modularity, client-side data filtering algorithms, and state architecture patterns. The system includes: fully interactive product grids, multi-tier search index logic, context-driven category filter arrays, persistent cart storage buffers, atomic quantity increment trackers, synchronized prompt modules, and client-side invoice summary outputs. While the baseline platform operates with simulated frontend-only document models and omits real-time external credit clearing nodes, database storage, and backend servers, its architecture is engineered with decoupled integration points. This allows for clean, long-term full-stack upgrades without requiring major layout redesigns.

VI. ALGORITHMIC LOGIC AND PSEUDOCODE

Algorithm 1: FoodGo Core Runtime Loop

```
1: START
2: INITIALIZE application container framework wrapper
3: PARSE static food array database list references
4: INITIALIZE global state slice with empty cart layout
5: WHILE application instance remains active inside viewport DO
  6: CAPTURE active character lookup inputs
  7: IF string query content changes THEN
    8: EXECUTE client array filter match checking arrays
    9: PATCH Virtual DOM and repaint target cards
  10: END IF
  11: IF user executes 'Add to Cart' click event THEN
    12: DISPATCH lookup operation to register index changes
    13: RE-EVALUATE dynamic price math structures
  14: END IF
15: END WHILE
16: STOP
```

Received: 20-04-2026 | Accepted: 29-05-2026 | Published: 04-06-2026

International Journal of
AI Electronics and Nexus Energy
Peer Reviewed, Referred & Indexed Journal
ISSN: 3070-0515 www.zesterapublications.com Original Research Paper

II. LITERATURE SURVEY

The historical evolution of web-based digital interaction graphs tracks a continuous push to minimize transaction execution latencies and optimize browser rendering mechanics. Early electronic food delivery structures operated on completely static document networks or basic server-side procedural scripts, where the web page served as a passive container for remote server operations.

A. Multi-Page System Limitations and Manual Ordering Friction

Traditional Multi-Page Applications (MPAs) are fundamentally bottlenecked by their architectural reliance on synchronous request-response network loops. When a user executes a minor change, the backend engine recalculates relational tables and re-renders the complete view file. This introduces distinct performance penalties: high data transmission sizes over network channels, excessive hosting compute cycles due to view script regeneration, and noticeable visual layout resets that degrade the customer experience. Before digital solutions became widespread, manual operations using telephone communications introduced even greater failure vectors, such as verbal misunderstandings, zero data logging, long waiting queues, and a complete absence of real-time transactional monitoring.

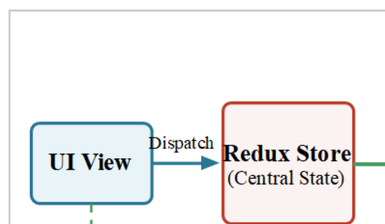

Fig. 1: FoodGo Unidirectional Architectural Data State Cycle.

VII. EXPERIMENTAL EVALUATION

Performance profiling captured page load parameters, event latency, component re-rendering counts, and interface scaling behaviors across diverse viewport setups. The client-side filter engine processed extensive text strings efficiently, updating layouts in milliseconds. This confirms that local query parsing runs smoothly without network request round trips. Cart data management achieved complete transactional accuracy with zero component status drift under consecutive action sequences.

VIII. CONCLUSION AND ENHANCEMENT ROADMAP

The development of FoodGo successfully demonstrates the exceptional efficiency of component-based Single Page Application architectures in high-frequency commercial scenarios. Centralizing transactional variables within an immutable state store eliminates component desynchronization anomalies and full-page layout reloads, providing a lightweight, robust architecture for interactive web interfaces. Future full-stack paths will connect secure Node.js routing environments with persistent MongoDB document collections and JWT validation structures.

REFERENCES

[1] Alex Banks and Eve Porcello, Learning React: Functional Web Development with React and Redux, O'Reilly Media, 2020.

[2] Jon Duckett, HTML and CSS: Design and Build Websites, Wiley Publications, 2011.

[3] Jon Duckett, JavaScript and JQuery: Interactive Front-End Web Development, Wiley Publications, 2014.

[4] David Flanagan, JavaScript: The Definitive Guide, O'Reilly Media, 2020.

B. The Transition toward Single Page Application Architecture

Single Page Application (SPA) designs resolve layout disruptions by transmitting a single skeletal HTML structure along with compiled client bundle scripts during the initial boot sequence. Subsequent user adjustments trigger asynchronous data transmissions that exchange lightweight data payloads, such as JSON blocks. The client engine processes these structures natively and alters targeted nodes within the document tree without full-page reloads. This approach minimizes data transfer sizes, dramatically reduces backend processing load, and delivers a highly responsive, app-like environment.

[5] Ethan Brown, Web Development with Node and Express, O'Reilly Media, 2019.

[6] Robin Wieruch, The Road to React, Self-Published, 2022.

[7] Stoyan Stefanov, React: Up and Running, O'Reilly Media, 2021.

[8] William S. Vincent, Modern JavaScript for the Impatient, Self-Published, 2020.

[9] React and Redux Official Documentation, Meta Open Source / Redux Toolkit, 2025.