iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

COMPONENT-BASED UI DESIGN WITH REACT.JS, ANGULAR, AND NODE.JS FOR ENTERPRISE PAYMENT PROCESSING APPLICATIONS

A TECHNICAL REFERENCE ON ATOMIC DESIGN, MICRO-FRONTENDS, AND THE BACKEND-FOR-FRONTEND PATTERN FOR PAYMENT UIS

Dinesh Nallapareddy
Sr. Full Stack Developer, USA

ABSTRACT
Enterprise payment applications increasingly combine React.js and Angular on the client with Node.js as an intermediating backend-for-frontend layer, a stack that offers strong componentization and reuse but that also concentrates a specific set of risks, inconsistent validation, accessibility gaps, and duplicated logic across teams, if component architecture is not deliberately governed. This reference works through component-based UI design for payment applications from first principles: atomic design as a structuring discipline, state management for payment forms, a shared design system and its cross-framework theming, micro-frontend composition, and the Node.js backend-for-frontend pattern that aggregates calls to downstream payment services. It covers the operational concerns that follow, testing strategy, accessibility, performance, security, and observability, and closes with a migration case study and a worked reference architecture. Illustrative figures and tables throughout quantify representative outcomes such as bundle size, test coverage, and performance score improvements.

Keywords:
React, Angular, Node.js, component-based architecture, atomic design, micro-frontends, backend for frontend, design system, payment forms, WCAG accessibility, enterprise UI

01 Introduction
A payment form is one of the smallest surfaces in an enterprise application and one of the most expensive to get wrong. It is also, in most organizations, rebuilt more times than it should be: once for the customer-facing checkout flow, again for an internal admin console used by support staff, and sometimes a third time for a partner-facing embedded widget, each built by a different team, in a different framework, with subtly different validation rules. Component-based architecture, applied deliberately rather than adopted as a slogan, is the discipline that turns three divergent implementations into one shared, tested set of building blocks used consistently everywhere a payment amount or a card number needs to be collected.

This reference examines that discipline in the specific context of a stack combining React.js, Angular, and Node.js, a common combination in enterprises that have grown by acquisition or that maintain both a modern customer-facing application and an older but still actively developed internal tool. React and Angular rarely coexist by original design; they coexist because two different parts of an organization made two different reasonable choices at two different points in time, and the practical problem is making components, validation logic, and design language consistent across that boundary rather than pretending the boundary does not exist.

Layer	Typical technology	Primary responsibility
Customer-facing client	React.js	Checkout, account management, self-service payment flows
Internal admin client	Angular	Case management, transaction review, support tooling
Backend for frontend	Node.js	Aggregating and shaping calls to downstream payment services
Shared design system	Framework-agnostic tokens and framework-specific wrappers	Consistent look, behavior, and accessibility across clients

Table 1. A representative technology layering for the stack covered in this reference.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

Sections 2 through 9 build the architectural foundation: why component-based design matters specifically for payment interfaces, how the three technologies divide responsibility, atomic design as a structuring method, state management, the shared design system, micro-frontend composition, and the backend-for-frontend layer. Sections 10 through 18 cover the cross-cutting concerns, including validation, accessibility, performance, testing, security, localization, observability, delivery, and theming. Section 19 presents a migration case study, Section 20 a worked reference architecture, and Sections 21 through 23 close with team topology, anti-patterns, and a summary.

02 Why Component-Based Architecture for Payment UIs

A payment form touches more concerns per pixel than almost any other interface a typical enterprise application presents: input formatting, real-time validation, accessibility for a legally protected class of user interaction, localization of currency and date formats, and a security posture that assumes the browser itself is a hostile environment. Building this repeatedly, by hand, in every product surface that needs it, multiplies the surface area for each of these concerns to go wrong independently.

Concern	Risk if rebuilt independently per team
Input validation	Card number or amount validation rules drift out of sync across surfaces
Accessibility	One team's form is screen-reader compatible; another's silently is not
Visual consistency	Customers experience a visibly different brand across products
Security posture	A fix for a client-side vulnerability is applied in one codebase and missed in another

Table 2. Risks of independently rebuilding payment UI logic across teams.

A shared component library converts each of these concerns into something fixed once, in one place, and consumed everywhere, which is the core argument for component-based architecture at the organizational level, not merely the technical one. Frost's (2016) atomic design methodology, covered in depth in Section 4, gives this sharing a concrete structure, and the remainder of this reference builds on that structure to address the specific demands of payment data.

> *A payment component's correctness is not a UI concern that happens to touch money. It is a financial control that happens to be implemented in a browser, and it deserves the review rigor of the latter, not the former.*

03 Comparing React.js, Angular, and Node.js Roles in the Stack

React and Angular occupy the same architectural position, rendering an interactive client-side application, but they make different structural commitments. React, maintained as an open source project, provides a rendering library and leaves state management, routing, and form handling to a surrounding ecosystem of separately chosen libraries. Angular is a more complete platform, bundling dependency injection, routing, forms, and a strongly typed development model built around TypeScript into one cohesive framework.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

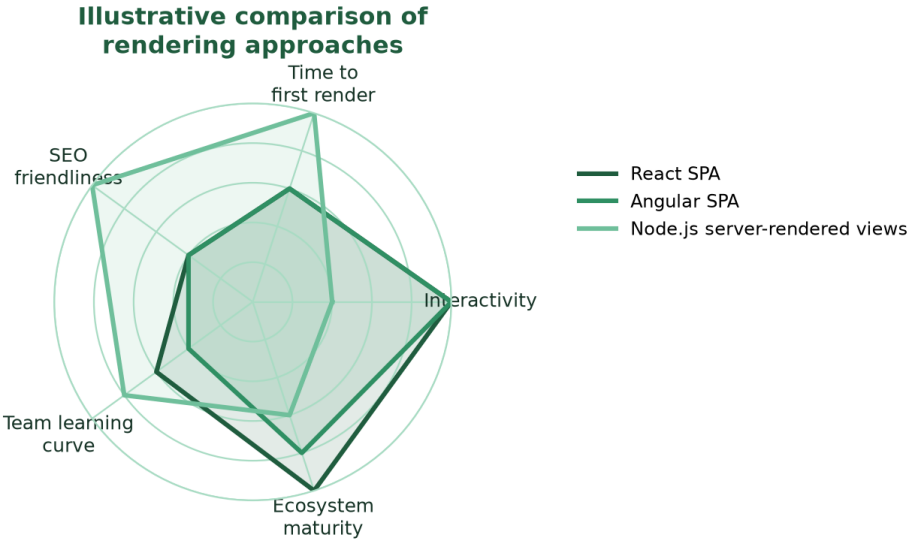

Figure 1. An illustrative comparison of React, Angular, and Node.js-rendered views across dimensions relevant to a payment UI.

Dimension	React.js	Angular
Structural philosophy	Library plus ecosystem of choices	Opinionated, batteries-included platform
Typing	Optional, commonly added via TypeScript	Built in, TypeScript by default
Form handling	Via a chosen library or hand-rolled hooks	Built-in reactive forms module
Common enterprise fit	Fast-moving customer-facing products	Long-lived internal tools with many contributors

Table 3. React and Angular contrasted on dimensions relevant to a payment application team's choice.

Node.js occupies a different position entirely: it is not a rendering framework but a JavaScript runtime, used here specifically as the backend-for-frontend layer described by Newman (2015) and Richardson (2018), a service that sits between the client applications and the organization's downstream payment services, shaping and aggregating calls so that neither the React client nor the Angular client needs to understand the internal topology of those downstream services directly.

04 Atomic Design: Structuring Components from Atoms to Pages

Frost (2016) organizes a component library into a small hierarchy: atoms, the smallest indivisible pieces such as a label or a text input; molecules, small groups of atoms that function together, such as a labeled input with an inline validation message; organisms, larger, more complete sections built from molecules, such as a full payment form; and templates and pages, which arrange organisms into an actual screen.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

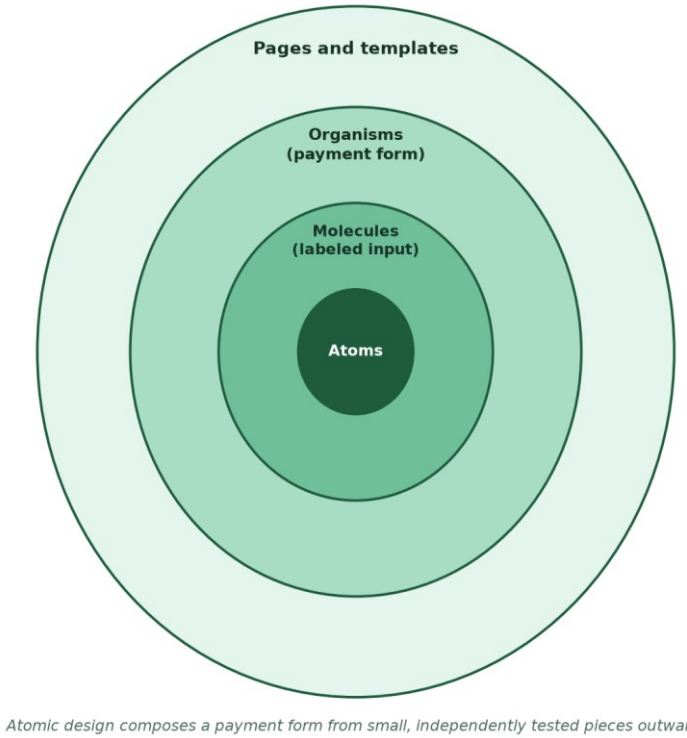

Figure 2. Atomic design's layering, shown as nested rings, applied to a payment form.

Layer	Payment UI example	Owned and tested by
Atom	Text input, currency symbol, button	Design system team
Molecule	Labeled amount field with validation message	Design system team
Organism	Complete payment form with card and amount fields	Product team, using design system pieces
Template or page	Checkout page combining the payment form with an order summary	Product team

Table 4. Atomic design layers mapped to ownership in a typical enterprise organization.

The practical value of this hierarchy is that it gives a natural boundary for both testing effort and ownership: atoms and molecules are cheap to test exhaustively because they are simple and change rarely, while organisms and pages, which change more often as product requirements evolve, can lean on the already-verified correctness of the atoms and molecules beneath them rather than re-testing that foundation every time.

05 State Management Patterns for Payment Forms

A payment form's state is not simply the current value of each field; it includes validation status per field, a submission status for the form as a whole, and often asynchronous state, such as a card-issuer lookup triggered as the user types, that changes independently of any single field's value. Modeling all of this as one large, ad hoc object invites exactly the kind of subtle bug that only appears when two of these state changes happen to occur close together.

State category	Example	Common approach
Field value	Current amount entered	Local component state or a form library's field state
Field validation	Amount exceeds account limit	Derived from field value, recomputed on change
Form-level status	Submitting, succeeded, failed	A single status enum rather than several booleans

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

Asynchronous lookup	Card network detected from card number	A dedicated async state slice with its own loading and error state

Table 5. Categories of state in a payment form and common ways to model each.

Representing form-level status as a single enumerated value, rather than a combination of independent boolean flags such as isSubmitting and hasError that can drift into an inconsistent combination, removes an entire category of bug where the form is, impossibly, both submitting and already failed at the same time. Both React and Angular support this modeling style natively; the discipline is in choosing to model state this way consistently across every payment form in the component library, not in a capability either framework lacks.

06 Component Reusability and a Shared Design System

A design system is the concrete artifact that makes atomic design's hierarchy shareable across React and Angular rather than merely a documentation exercise. In practice, this means design tokens, colors, spacing, typography, expressed in a framework-agnostic format, and a set of components built twice, once as React components and once as Angular components, but sharing the same tokens, the same visual specification, and ideally the same underlying test suite structure.

Design system layer	Shared across React and Angular	Framework specific
Design tokens	Yes, typically as a JSON or CSS custom property source	No
Visual specification and accessibility behavior	Yes, documented once	No
Component implementation	No, one implementation per framework	Yes
Component API surface	Kept as similar as each framework's idioms allow	Adapted to framework conventions

Table 6. What is genuinely shared across frameworks in a cross-framework design system, and what is not.

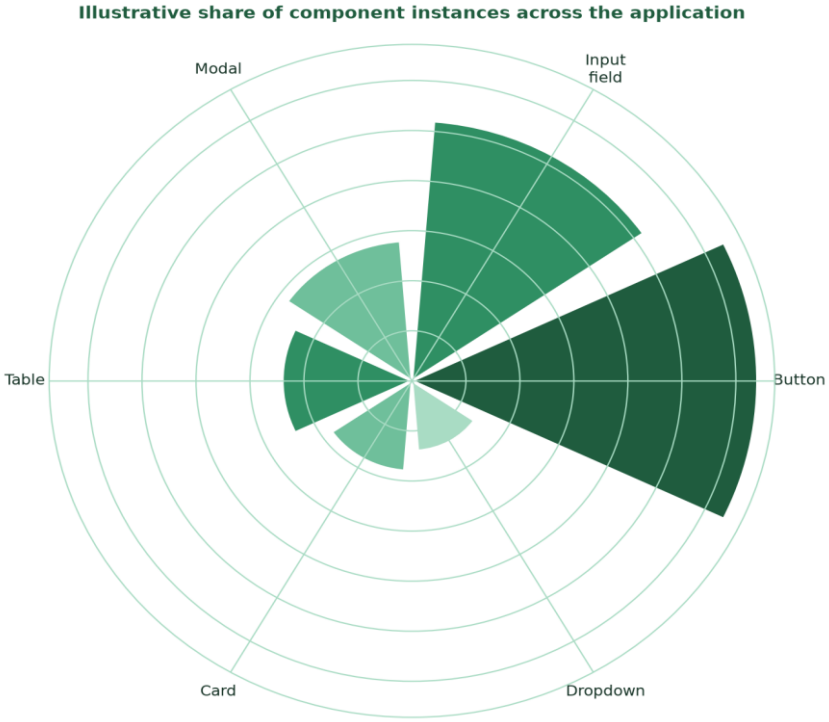

Figure 13. Illustrative share of component instances across the application, by component type.

A usage distribution like the one in Figure 13 is a practical prioritization tool for the design system team: the button and input field atoms, together accounting for well over half of all component instances in this illustrative breakdown, justify a disproportionate share of review and testing investment relative to a less frequently used

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

component such as a dropdown, simply because a defect in a high-usage atom is encountered by far more of the application than a defect in a rarely used one.

> *Attempting to share the actual component implementation between React and Angular, rather than sharing the specification and tokens behind two separate implementations, is the single most common design system mistake in a mixed-framework organization. The frameworks' component models are different enough that a genuinely shared implementation usually degrades to the lowest common denominator of both.*

Governance matters here as much as tooling: a design system with no clear owner and no review process for new components tends to accumulate near-duplicate components, three slightly different button variants, for example, that each solve one team's immediate need without contributing to a coherent shared library. Section 21 returns to the ownership model that keeps this from happening.

07 Micro-Frontend Architecture with Module Federation

As an organization's payment product surface grows, a single monolithic client application, whether built in React or Angular, eventually faces the same coupling problems Section 2 described for duplicated logic, but in the opposite direction: one enormous codebase where every team's changes compete for the same build, the same deployment, and the same release calendar. Micro-frontend architecture, described in depth by Geers (2020), addresses this by decomposing the client application itself into independently built and deployed pieces, composed together at runtime.

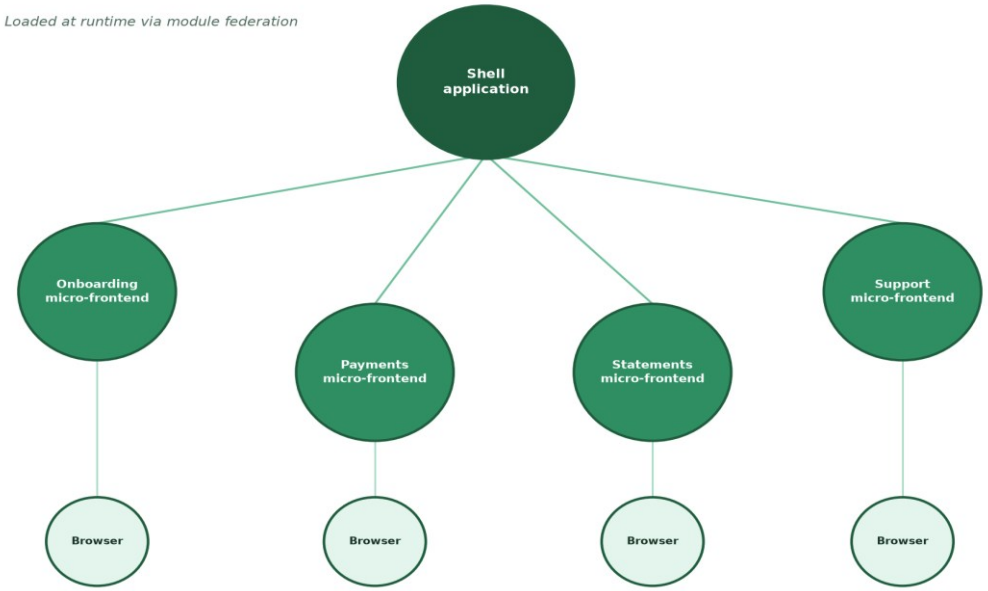

Figure 3. A shell application composing four independently deployed micro-frontends via module federation.

Property	Monolithic client	Micro-frontend composition
Deployment unit	One application, every team's code together	One deployable unit per micro-frontend
Technology consistency	Enforced by a single build configuration	Each micro-frontend may vary, within shared design system constraints
Failure isolation	A defect in one area can block the whole build	A defect is scoped to its own micro-frontend
Coordination overhead	Low per change, high in aggregate as teams grow	Higher per integration, lower in aggregate at scale

Table 7. Monolithic and micro-frontend client architectures compared.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

Module federation, the specific mechanism shown in Figure 3, allows the shell application to load a remote micro-frontend's compiled bundle at runtime rather than at the shell's own build time, which is what lets the Payments micro-frontend deploy independently of the Onboarding micro-frontend even though both are ultimately rendered inside the same shell. The shared component library from Section 6 becomes especially important in this architecture, since it is the mechanism that keeps four independently built micro-frontends visually and behaviorally coherent to the end user, who has no reason to know they are looking at four different deployments.

08 Node.js as the Backend-for-Frontend Layer

The backend-for-frontend pattern, documented by Newman (2015) and Richardson (2018), places a dedicated service between a client application and the organization's downstream services, shaped specifically around what that client needs rather than around the downstream services' own data models. Node.js is a common choice for this layer in a JavaScript-heavy stack because it lets frontend engineers, already fluent in JavaScript or TypeScript, own the BFF's code without a language switch.

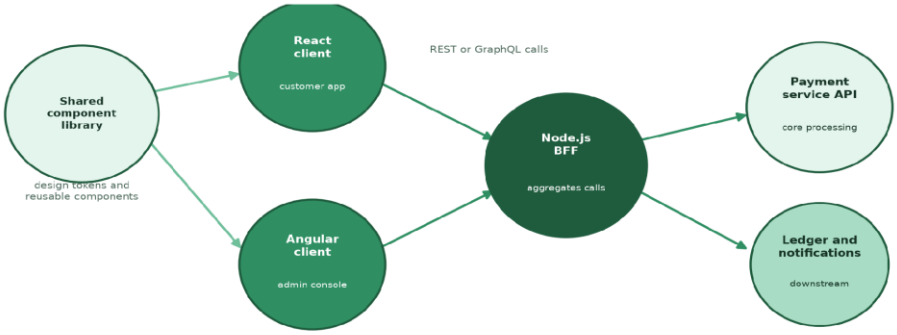

Figure 4. React and Angular clients both routed through a shared Node.js backend-for-frontend layer.

A single BFF shared by both the React and Angular clients, as shown in Figure 4, avoids duplicating aggregation logic between two separate backend layers, one per client framework, which would otherwise recreate the exact duplication problem Section 2 described for UI components, just one layer down the stack. The BFF is also a natural place to apply cross-cutting concerns once rather than per client: authentication token exchange, response shaping, and rate limiting all belong here rather than duplicated in both the React and Angular applications.

BFF responsibility	Why it belongs here rather than in the client
Aggregating multiple downstream calls into one response	Keeps client code simple and reduces round trips over the public internet
Translating downstream error codes into a stable client contract	Shields both clients from downstream service implementation changes
Enforcing authentication and session handling	Centralizes a security-sensitive concern in one reviewed codebase

Table 8. Representative backend-for-frontend responsibilities and their rationale.

09 API Design Between Clients and Node.js Services

The contract between a client component and the BFF deserves the same design discipline as any public API, even though both sides may be maintained by closely collaborating teams. A payment form component that directly encodes assumptions about a specific BFF endpoint's response shape becomes difficult to reuse across products with slightly different backend integrations, which undermines exactly the reusability goal the component library exists to serve.

- Define the data a payment component needs as a framework-agnostic interface, independent of the exact BFF endpoint that happens to supply it today, so the component can be reused against a different endpoint shape without changing the component itself.
- Version BFF endpoints deliberately when a breaking change is unavoidable, following the same compatibility discipline any service-to-service API would apply.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

- Keep validation rules that affect what the user sees, such as a minimum payment amount, available to the client synchronously, rather than requiring a round trip to the BFF for feedback the user expects instantly.

This last point matters specifically for payment forms: a validation rule enforced only on the server produces a noticeably worse experience than one that can also run client-side for immediate feedback, but the server-side check must remain authoritative regardless, since a client-side check can always be bypassed. Section 10 covers this dual-validation requirement in more depth.

10 Form Validation and PCI-Aware Input Handling

Client-side validation in a payment form serves user experience, not security, a distinction worth stating plainly because it is easy to conflate the two. Every validation rule that matters for correctness or compliance must be enforced again on the server, specifically in the Node.js BFF or the downstream payment service, regardless of what the client already checked, because a client can be manipulated in ways a server-side check cannot.

Validation concern	Client-side role	Server-side role
Card number format	Immediate feedback via a check digit algorithm	Authoritative validation before submission to a processor
Amount limits	Immediate feedback against a known limit	Authoritative check against the current account state
Required fields	Prevents an obviously incomplete submission	Authoritative rejection of any incomplete payload

Table 9. Client and server validation roles for representative payment form fields.

The Payment Card Industry Data Security Standard (PCI Security Standards Council, 2018) constrains how card data may be handled well before validation even occurs: a payment component should, wherever possible, tokenize card details directly with a payment processor's own client-side library rather than passing raw card numbers through the application's own React or Angular state and Node.js BFF at all, which keeps the broader application out of the strictest scope of PCI compliance entirely.

> *The safest place for a card number to exist in your own application code is nowhere. A component that hands raw card data directly to a processor's tokenization library, rather than storing it in component state even briefly, removes an entire category of compliance and security risk by construction.*

11 Accessibility in Payment Components

Payment forms are, in many jurisdictions, subject to legal accessibility requirements in addition to the general usability argument for accessible design, which makes the Web Content Accessibility Guidelines (W3C, 2018) a compliance concern as much as a design one for an enterprise payment application.

WCAG concern	Payment component implication
Keyboard operability	Every field, including a card network selector, must be usable without a mouse
Error identification	A validation error must be programmatically associated with its field, not only styled in color
Sufficient contrast	Error and success states must remain legible for users with low vision
Focus management	Submitting a form with errors should move focus to the first invalid field

Table 10. Selected WCAG 2.1 concerns and their implication for payment component design.

Building these behaviors into the atoms and molecules described in Section 4, rather than leaving each product team to reimplement focus management and error association independently, is what makes accessibility compliance achievable at scale. A design system that gets focus management right once, in one labeled-input molecule used everywhere, guarantees the behavior everywhere that molecule is used; a policy document asking every team to implement it correctly on their own does not.

12 Performance: Bundle Size, Lazy Loading, and Code Splitting

A component library's reusability can work against performance if consumed carelessly: a page that imports an entire component library to use one button pays the download cost of every other component in that library as well, unless the build tooling can eliminate unused code through tree shaking or the application deliberately splits its bundle.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

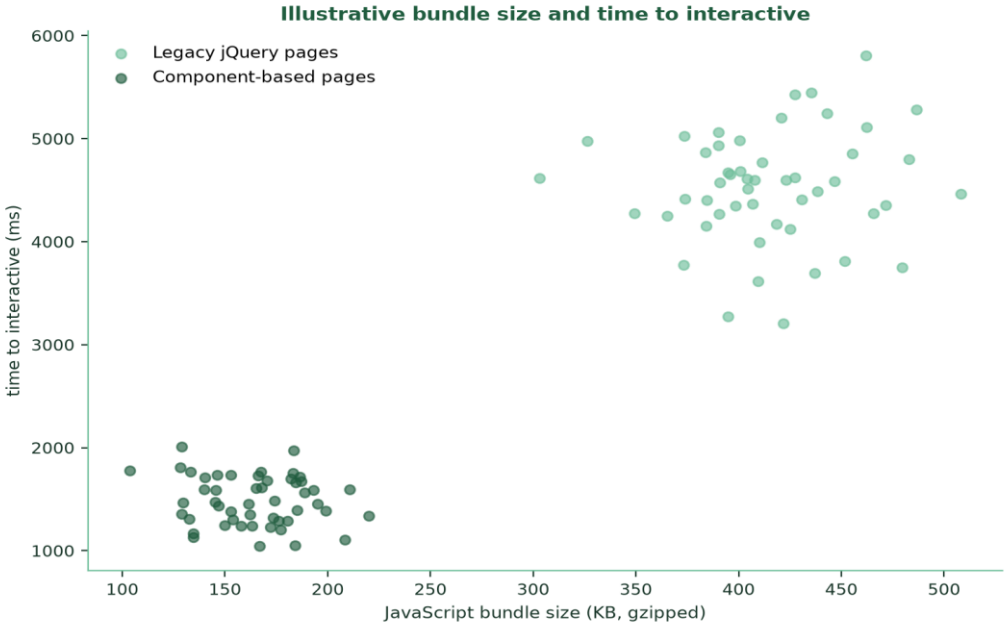

Figure 5. Illustrative bundle size and time to interactive, legacy pages compared with component-based pages.

Technique	Effect
Tree shaking	Removes unused exports from the final bundle at build time
Code splitting by route	Loads a page's code only when the user navigates to it
Lazy loading a heavy component	Defers loading a rarely used organism, such as a rich transaction export dialog, until requested

Table 11. Bundle size reduction techniques applicable to both React and Angular applications.

The separation visible in Figure 5 between legacy and component-based pages reflects both a smaller starting bundle and a build process actively enforcing these techniques; a component library alone does not guarantee a smaller bundle if pages import more of it than they use, which is why bundle size budgets, checked automatically in continuous integration, matter alongside the component architecture itself.

13 Testing Strategy Across the Component Stack

Testing a component-based payment UI benefits from the same layered strategy that a service-oriented backend uses, adapted to the specifics of rendered interfaces: unit tests for pure logic, component tests that render a single component in isolation, integration tests that exercise a client against a real or faithfully mocked BFF, and end-to-end tests that drive a full checkout flow through a real browser.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

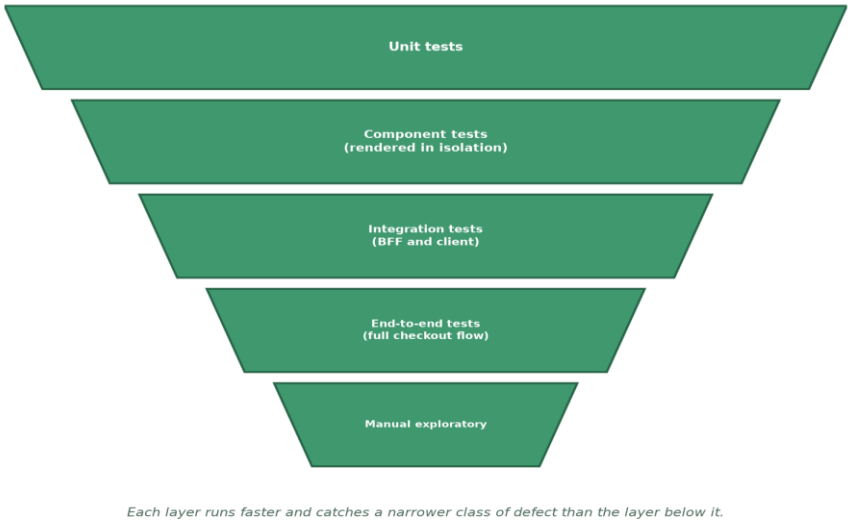

Figure 6. A testing pyramid for a component-based payment UI, from unit tests through manual exploratory testing.

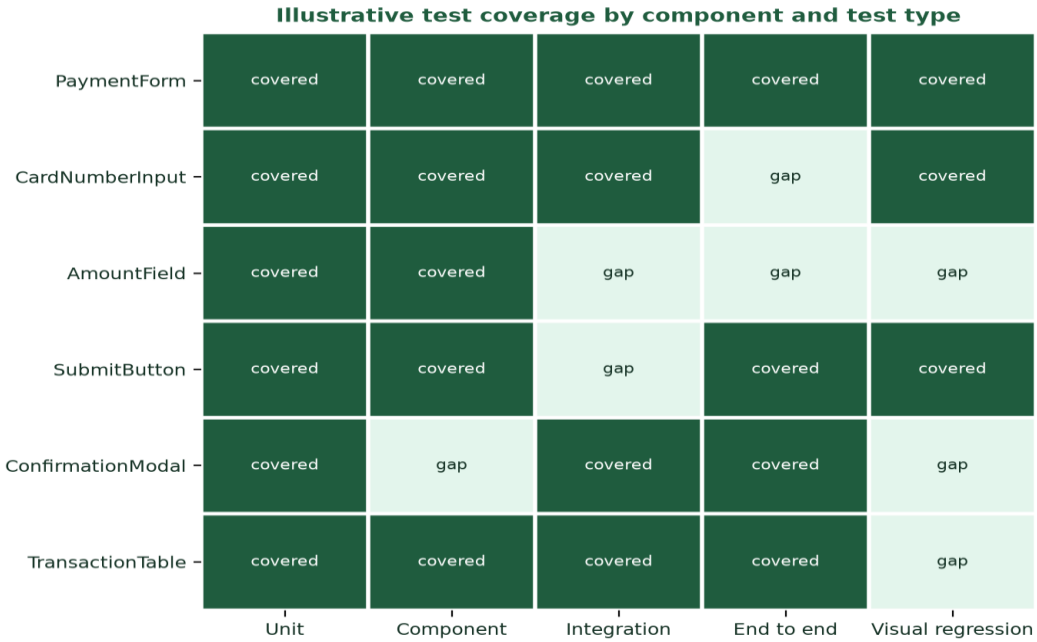

Figure 7. Illustrative test coverage by component and test type across the shared component library.

Test layer	Typical tool category	What it catches
Unit	A JavaScript test runner	Logic errors in pure functions, such as amount formatting
Component	A component testing library rendering in isolation	Rendering defects and accessibility regressions in one component
Integration	A test harness exercising client and BFF together	Contract mismatches between client expectations and BFF responses
End to end	A browser automation tool	Defects only visible across a full, realistic user flow

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

Table 12. Testing layers for a component-based payment UI, following the pyramid in Figure 6.

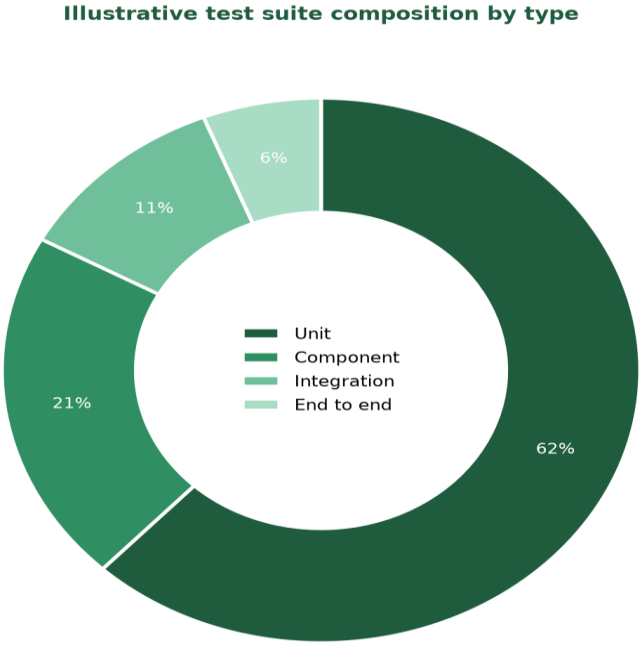

Figure 8. Illustrative test suite composition by type.

The gaps visible in Figure 7 are as informative as the coverage: a component like AmountField missing integration and end-to-end coverage is a reasonable near-term risk if its logic is simple and well covered at the unit and component level, while the same gap on PaymentForm itself, the organism orchestrating the whole submission, would be considerably more concerning given how much behavior that organism coordinates.

14 Security Considerations in Client-Side Payment Components

A browser is, from a security design perspective, an environment the application does not control: its JavaScript can be inspected, its network calls can be intercepted, and any client-side check can be bypassed by a sufficiently motivated user. Payment components need to be designed with this assumption explicit rather than incidental.

Threat	Mitigation in the component or BFF layer
Cross-site scripting via an untrusted rendered value	Rely on the framework's default escaping; avoid raw HTML injection APIs
Sensitive data lingering in client memory or logs	Avoid storing raw card data in component state; clear sensitive fields on unmount
Cross-site request forgery against the BFF	Standard anti-forgery tokens and same-site cookie policy at the BFF layer
Dependency supply chain risk in the component library	Pin and audit dependencies; review before upgrading a widely used shared package

Table 13. Representative client-side security threats and where they are mitigated.

The shared component library is a specific point of leverage for security as well as for consistency: a vulnerability fixed once in a shared CardNumberInput atom propagates to every product that consumes it on its next dependency update, whereas the same fix applied to three independently built implementations requires three separate, coordinated releases.

15 Internationalization and Localization

A payment component built for one locale carries assumptions that do not generalize automatically: decimal separators, currency symbol placement, and date formats vary by locale, and a hard-coded assumption baked into an atom propagates to every product using that atom, in every locale, immediately.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

Localization concern	Component-level implication
Decimal and thousands separators	Amount formatting must use a locale-aware formatting function, not a fixed pattern
Currency symbol position	Symbol placement before or after the amount varies by locale
Date format	A statement date component must render according to the active locale, not a fixed format
Text direction	A right-to-left locale changes layout direction for an entire form, not just its text

Table 14. Localization concerns relevant to payment components.

Centralizing locale-aware formatting inside the shared atoms, rather than leaving each product team to call a formatting library correctly on their own, is again the design system doing for localization what Section 11 described for accessibility: making the correct behavior the default rather than a responsibility each team must remember to discharge independently.

16 Observability: Frontend Monitoring and Error Tracking

A defect in a payment component that only manifests in one browser, one locale, or one specific combination of feature flags is difficult to reproduce from a bug report alone. Frontend error tracking, capturing an unhandled exception along with enough context, the component involved, the browser, the user's locale, to reproduce the failure, closes that gap.

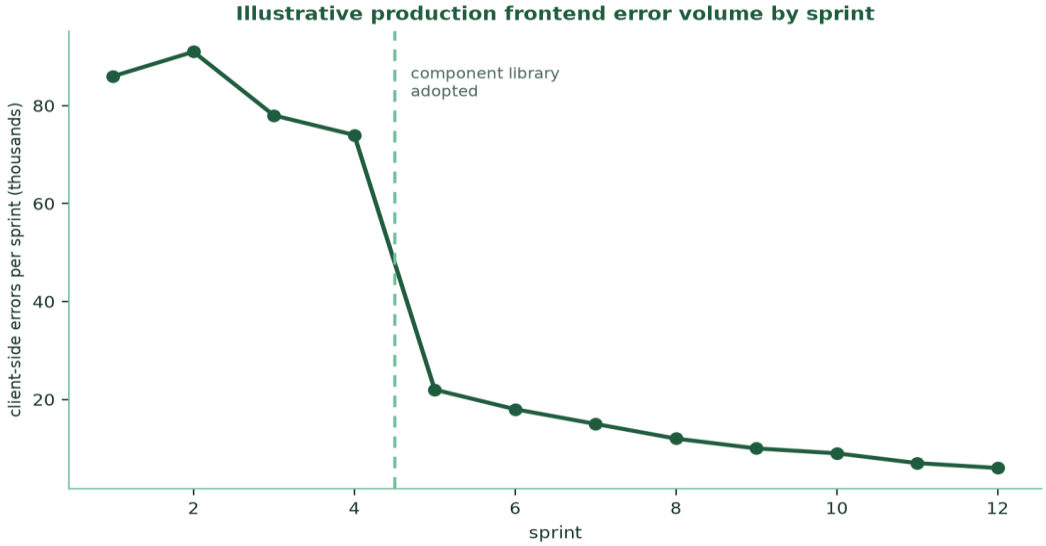

Figure 9. Illustrative production frontend error volume by sprint, before and after component library adoption.

The decline shown in Figure 9 reflects the same underlying dynamic Section 2 predicted: much of what previously reached production as a frontend defect was duplicated, inconsistent logic across several independently built forms, and consolidating that logic into a tested, shared component removed most of the opportunity for that class of defect to occur at all, leaving a smaller residual error volume concentrated in genuinely new product logic rather than in payment form fundamentals.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

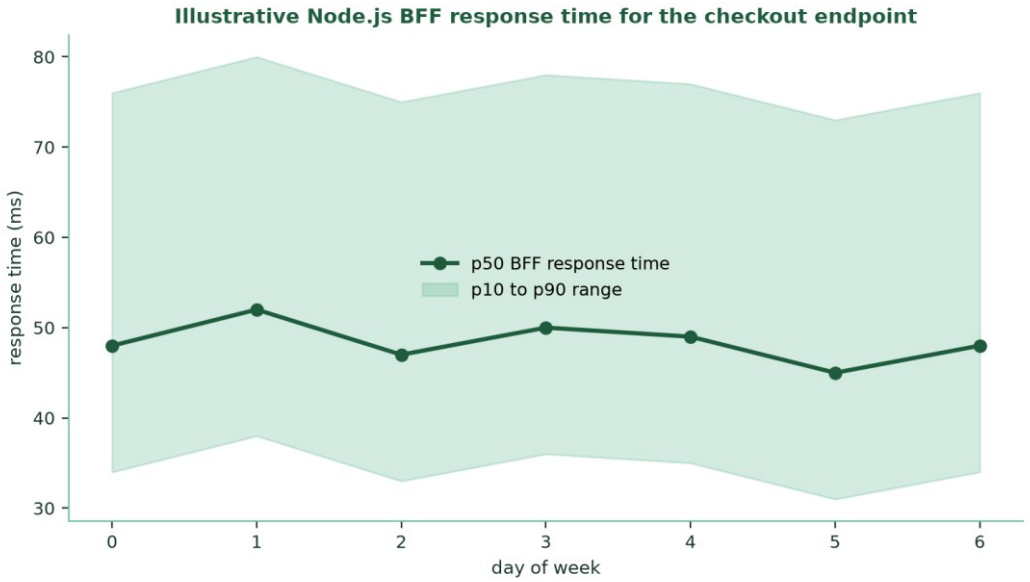

Figure 10. Illustrative Node.js backend-for-frontend response time for the checkout endpoint, with the p10 to p90 range shaded.

17 Continuous Delivery for a Component Library

Humble and Farley (2010) describe continuous delivery as keeping software in a state that can be released at any time, a standard a shared component library needs to meet just as much as any deployed service, because every consuming application's own release schedule depends on the library's releases being trustworthy.

Pipeline stage	Purpose for a component library
Automated visual regression testing	Catches an unintended visual change to a widely used component before release
Semantic versioning enforcement	Ensures a breaking change to a component's API is never released as a minor version
Cross-framework build verification	Confirms both the React and Angular packages build successfully from shared tokens
Automated changelog generation	Gives every consuming team a clear, reviewable record of what changed

Table 15. Continuous delivery pipeline stages specific to a shared, cross-framework component library.

Semantic versioning discipline matters more for a shared library than for an application, because a library has many independent consumers who cannot all coordinate an upgrade at the same moment; a breaking change released without a major version bump effectively breaks every consumer's build simultaneously and without warning, a failure mode Section 22 returns to as a specific anti-pattern.

18 Design Tokens and Cross-Framework Theming

Design tokens, the smallest named values behind a design system, a specific shade of green, a spacing unit, a border radius, are what let React and Angular components share a visual language without sharing implementation code. Defined once in a framework-neutral format and compiled into whatever consumption format each framework prefers, CSS custom properties, a TypeScript constants file, or both, tokens keep a rebrand or a theme change a single-source-of-truth edit rather than a search-and-replace exercise across two codebases.

Token category	Example	Consumed by
Color	color-brand-primary	Buttons, links, focus indicators
Spacing	space-md	Padding and margin across every component
Typography	font-size-body	Text rendering across both frameworks
Elevation	shadow-modal	Modal and dropdown surfaces

Table 16. Representative design token categories and their consumers.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

A theming requirement that only affects tokens, supporting a white-labeled partner-facing version of the checkout flow with different brand colors, for example, can be satisfied without touching either framework's component code at all, which is the clearest practical demonstration of why separating tokens from implementation pays for itself.

19 Case Study: Migrating a Legacy jQuery Payment Form

A representative migration case starts from a payment form built years earlier in jQuery, directly manipulating the document object model, with validation logic duplicated across several pages and no systematic accessibility support. The goal is not a rewrite in one step but an incremental strangler-style migration, echoing the strategy applied to backend systems elsewhere in the literature, that replaces the legacy form with React and Angular equivalents built from the shared component library, page by page.

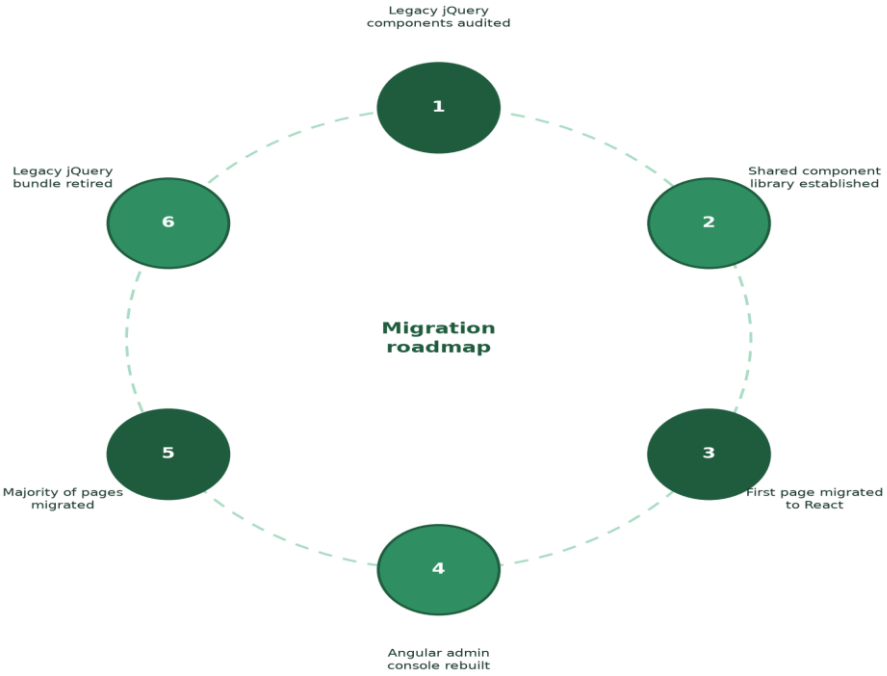

Figure 11. An illustrative migration roadmap from a legacy jQuery payment form to the shared component library.

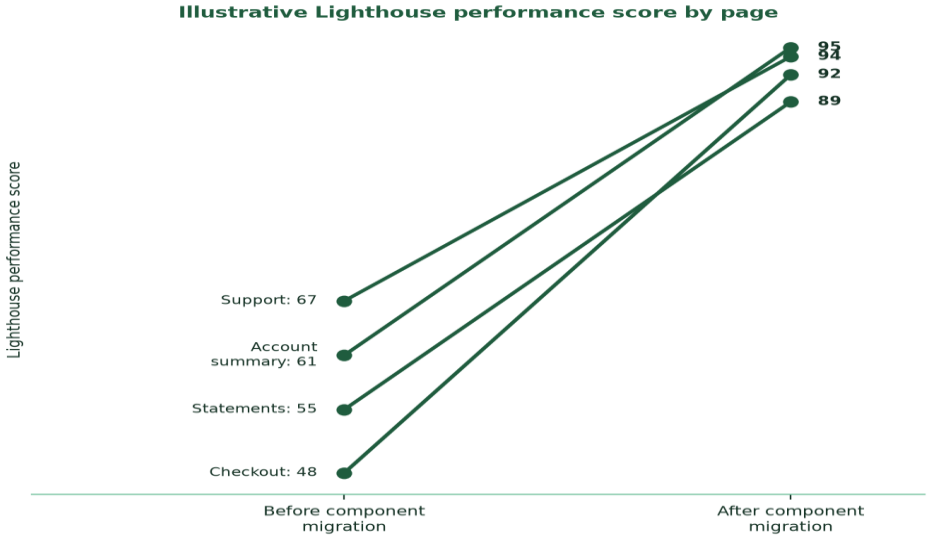

Figure 12. Illustrative Lighthouse performance score by page, before and after migration.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

Migration phase	Key activity	Primary risk addressed
Audit	Catalog every existing jQuery form and its validation rules	Unknown scope causing an incomplete migration
Library established	Build atoms and molecules covering the audited rules	Inconsistent behavior across migrated pages
Parallel run	New and legacy forms both live behind a feature flag	A regression reaching every customer at once
Cutover	Feature flag defaults to the new form for all traffic	Delaying indefinitely without a forcing function
Retirement	Legacy jQuery bundle removed from the build	Ongoing maintenance cost of two parallel implementations

Table 17. Migration phases for the jQuery-to-component-library case study.

20 A Worked Reference Architecture

Bringing the preceding sections together, a representative enterprise payment UI architecture serves a React customer-facing client and an Angular admin client, both built from the same atomic-design component library and design tokens, both consuming a shared Node.js backend-for-frontend layer, with validation enforced at both the component level for immediate feedback and the BFF level for authoritative correctness.

Layer	Technology	Testing emphasis
Customer client	React, shared component library	Component and end-to-end tests on the checkout flow
Admin client	Angular, shared component library	Component and integration tests on case management flows
Backend for frontend	Node.js	Integration tests against real downstream contracts
Shared component library	Framework-agnostic tokens, React and Angular wrappers	Unit, component, and visual regression tests

Table 18. Layer-by-layer summary of the worked reference architecture.

No single layer in this architecture is unusual on its own; its coherence comes from the shared component library and design tokens running through every layer, which is what keeps a React checkout flow and an Angular case management screen feeling like parts of one product rather than two unrelated applications that happen to share a color palette.

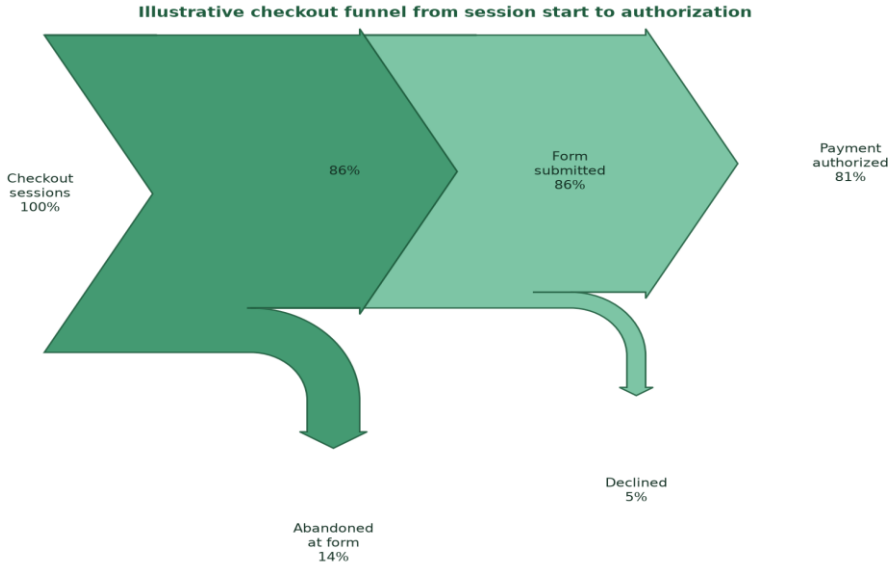

Figure 14. An illustrative checkout funnel from session start through payment authorization, the end-to-end flow this reference architecture exists to support.

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

The abandonment and decline points visible in Figure 14 are where the architecture's individual layers earn their keep in combination: client-side validation from Section 10 reduces abandonment at the form itself, the backend-for-frontend layer from Section 8 shapes a clear decline reason back to the client rather than an opaque failure, and the observability practices from Section 16 make both rates visible to the team responsible for improving them.

21 Team Topology and Component Ownership

A shared component library needs a team that owns it as a product in its own right, with its own backlog, its own release cadence, and its own accountability for the quality of what it publishes, rather than being treated as a shared folder anyone can commit to without review.

Ownership model	Strength	Weakness
Dedicated platform team owns the library	Consistent quality bar and clear accountability	Can become a bottleneck if under-resourced
Rotating ownership across product teams	Distributes knowledge of the library widely	Inconsistent quality bar between rotations
No dedicated ownership, open contribution	Low coordination overhead initially	Tends toward the near-duplicate component problem in Section 6

Table 19. Component library ownership models compared.

A dedicated platform team does not need to be large to be effective; its primary function is review and curation, deciding what belongs in the shared library and what should remain specific to one product, more than it is building every component from scratch itself. Contributions from product teams are welcome and often necessary, provided they pass through the same review bar the platform team applies to its own work.

22 Common Anti-Patterns in Component-Based UI Development

Anti-pattern	Symptom	Remedy
Prop drilling through many layers	A value is passed through components that do not use it themselves	Use a scoped state mechanism appropriate to the framework
Shared implementation forced across frameworks	A React component wrapped awkwardly to pretend to be Angular	Share tokens and specification; implement natively per framework
Breaking change released without a major version	Every consuming application breaks on its next dependency update	Enforce semantic versioning in the library's release pipeline
Client-only validation for a compliance-relevant rule	A bypassed client check reaches a downstream service unvalidated	Always enforce the authoritative check server-side as well
No component library ownership	Near-duplicate components accumulate across product teams	Establish a platform team accountable for the library

Table 20. Recurring anti-patterns in component-based enterprise UI development.

Prop drilling deserves particular mention because it is a symptom rather than a root cause: it usually signals that a value genuinely belongs in shared, scoped state rather than in props threaded through several intermediate components that have no use for it themselves. Both React and Angular offer purpose-built mechanisms for this, context in React and dependency injection or a shared service in Angular, and reaching for one of those mechanisms rather than continuing to thread props is usually the correct fix once the pattern is recognized.

23 Conclusion

Component-based architecture does not, by itself, make a payment UI correct, accessible, or performant; it makes those properties achievable at a scale and consistency that rebuilding logic independently across React, Angular, and every product surface never can. Atomic design gives that architecture a structure, the shared design system gives it a cross-framework visual and behavioral language, and the Node.js backend-for-frontend layer gives both clients a consistent, well-shaped contract with the organization's downstream payment services.

The practices covered across this reference, deliberate state modeling, dual client and server validation, accessibility and localization built into shared atoms, layered testing, and clear component library ownership, are individually familiar to most frontend engineers. Applied consistently across every payment surface an enterprise maintains, rather than selectively on whichever surface happens to get the most attention, they are what turns a

iJETRM
INTERNATIONAL JOURNAL OF ENGINEERING TECHNOLOGY RESEARCH & MANAGEMENT (IJETRM)
Peer-Reviewed | Open Access | International Journal
https://ijetrm.com/

payment UI from a recurring source of inconsistency and rework into a shared, trusted foundation the whole organization builds on.

REFERENCES

1) Frost, B. (2016). Atomic Design.
2) Geers, M. (2020). Micro Frontends in Action. Manning Publications.
3) Google. (2022). Angular: The Modern Web Developer's Platform.
4) Humble, J., and Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley.
5) Meta Open Source. (2022). React: A JavaScript Library for Building User Interfaces.
6) Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.
7) Nielsen, J. (1993). Usability Engineering. Academic Press.
8) OpenJS Foundation. (2022). Node.js Documentation.
9) PCI Security Standards Council. (2018). Payment Card Industry Data Security Standard, version 3.2.1.
10) Richardson, C. (2018). Microservices Patterns: With Examples in Java. Manning Publications.
11) W3C. (2017). WAI-ARIA Authoring Practices.
12) W3C. (2018). Web Content Accessibility Guidelines (WCAG) 2.1.

