



Hybrid jQuery React Interoperability Strategies During Incremental Frontend Framework Migration

Harshika Juvvadi

Full stack Developer, USA

ABSTRACT: Many production web applications built between 2010 and 2018 still depend heavily on jQuery for DOM manipulation, event handling, and AJAX communication, while product teams increasingly want the componentized, declarative model that React provides. Rewriting such an application from scratch is rarely feasible: it freezes feature delivery, concentrates risk into a single high stakes release, and discards years of tested business logic embedded in jQuery event handlers and plugins. This article surveys and evaluates hybrid interoperability strategies that let jQuery and React occupy the same runtime and, in many cases, the same page during an incremental migration. Seven coexistence strategies are described in detail with illustrative code: mounting React components inside jQuery managed containers, wrapping jQuery plugins as React components, a custom event bridge for cross framework messaging, a shared state synchronization layer, route based strangler fig migration, an islands style adaptation for jQuery to React transitions, and build tooling considerations such as module federation and selective bundling. An illustrative case study models a mid size internal tools application migrated across five phases and reports simulated defect and performance indicators to make the trade offs concrete. The article closes with a comparison matrix of the seven strategies, a discussion of when each is appropriate, limitations of the analysis, and directions for further empirical work. The intended audience is engineering teams planning or currently executing an incremental jQuery to React migration in 2022 and researchers studying legacy frontend modernization.

KEYWORDS: jQuery, React, frontend migration, framework interoperability, strangler fig pattern, islands architecture, legacy modernization, incremental rewrite, DOM ownership, event bridge

I. INTRODUCTION

jQuery was, for most of the 2010s, the default way to make a web page interactive. It normalized inconsistent browser APIs, offered a compact selector based syntax for DOM traversal, and shipped with or alongside a large ecosystem of plugins for date pickers, modals, carousels, form validation, and grid widgets. By the time React reached its 16.0 release in September 2017 and stabilized the Fiber reconciliation engine (React Core Team, 2017), a large share of production line of business applications, admin panels, and content heavy marketing sites had already been built on top of jQuery and server rendered templates. Those applications did not stop working when React became popular, and the organizations that built them did not have the luxury of pausing feature work for a multi month rewrite.

This creates a durable and practical engineering problem rather than a purely academic one. A team that wants to move to React for good reasons, better testability, a declarative rendering model, a large hiring pool, and a rich component ecosystem, must find a way to introduce React pages, or React fragments of a page, next to jQuery code that still owns most of the DOM. The two libraries were not designed with each other in mind. jQuery assumes it can freely read and mutate the DOM at any time. React assumes it owns a subtree of the DOM exclusively and reconciles that subtree against a virtual representation on every render. When both assumptions hold in different regions of the same document, the application works. When they collide, typically because jQuery reaches into a DOM node that React also manages, the result ranges from a silently dropped event handler to a thrown exception during reconciliation.

This article treats the problem as one of interoperability engineering: given two frameworks with different DOM ownership models, what concrete patterns let them coexist safely long enough for an incremental migration to complete? The scope is deliberately practical. Rather than arguing for or against rewriting an application in React, the article assumes the decision to migrate has already been made and focuses on the mechanics of the transition period itself, which in the case studies referenced in Section 2 typically lasts between six months and two years for a mid size application.



The remainder of the article is organized as follows. Section 2 reviews the historical background of both libraries and summarizes prior published guidance on library interoperability and incremental modernization. Section 3 states the specific technical challenges that make jQuery and React difficult to run side by side. Section 4 is the core of the article and presents seven coexistence strategies, each with example code, a description of when it applies, and its known failure modes. Section 5 walks through an illustrative case study with simulated metrics to make the trade offs concrete. Section 6 compares the seven strategies directly and offers practical recommendations. Section 7 discusses limitations of this analysis, Section 8 suggests directions for further work, and Section 9 concludes.

II. BACKGROUND AND RELATED WORK

2.1 The jQuery Era

jQuery reached its 3.0 release in June 2016, consolidating a decade of work on cross browser DOM manipulation, event delegation, and animation into a smaller and more standards aligned core (jQuery Foundation, 2016). Its selector engine, event delegation model using `$(document).on()`, and the deferred and promise based `$.ajax()` interface became the shared vocabulary of a generation of frontend developers. Because jQuery has no concept of a component tree, applications built on it typically organize behavior around DOM events attached to specific selectors, with state stored either directly on DOM elements through `.data()`, in module level closures, or in global objects. This works well for pages that are largely static markup with scattered interactive widgets, and considerably less well for pages that are, in effect, dense interactive applications with many interdependent pieces of state such as a checkout flow, a data grid with filtering and sorting, or a multi step wizard.

A widely cited community resource, the You Might Not Need jQuery project and its accompanying CSS Tricks discussion (Coyier, 2015), catalogued how much of jQuery's original value, primarily cross browser normalization, had been absorbed into evergreen browsers by the mid 2010s. This did not eliminate jQuery's usefulness for existing codebases, since rewriting working code purely to remove a dependency carries its own risk, but it shifted the argument for adopting jQuery in new code, and by extension it sharpened the argument for eventually retiring it from old code once a more expressive alternative was already in use elsewhere in the stack.

2.2 The Rise of Component Based Rendering

React introduced a different mental model: a tree of components, each a function of its props and state, rendered declaratively and reconciled against a virtual representation of the DOM. The React documentation's own guidance on integrating with other libraries explicitly anticipated the coexistence problem this article addresses, describing patterns for wrapping non React widgets and for mounting React into a page that is not fully controlled by React (React Core Team, 2019). The official React 16 release notes describe the introduction of `ReactDOM.createPortal` and improved error boundaries, both of which matter for coexistence work because they give React a supported way to render into DOM nodes outside the primary tree and to contain failures locally rather than unmounting the entire application (React Core Team, 2017).

By 2020, the State of JavaScript survey reported React as the most widely used and highest satisfaction rated view layer library among the frameworks it tracked, with jQuery still present in a large share of respondent codebases but rarely selected for new projects (Greif et al., 2020). That gap between what teams were still running in production and what they wanted to write going forward is the underlying condition that makes hybrid interoperability strategies necessary rather than optional for most organizations.

2.3 Incremental Modernization Patterns

The engineering literature on migrating legacy systems predates the jQuery to React transition by well over a decade. Fowler's description of the strangler fig application pattern, named for a vine that gradually envelops and eventually replaces its host tree, argued for building the new system around the edges of the old one and cutting over piece by piece rather than in one release (Fowler, 2004). The pattern was originally described for backend and enterprise systems, but its central claim, that a big bang rewrite concentrates risk while an incremental replacement distributes it, applies directly to frontend rewrites and is the organizing principle behind Strategy E in Section 4.5 of this article.

A closely related idea from the frontend community is the micro frontend, in which independently deployable frontend applications, potentially built with different frameworks, are composed into a single user facing product (Jackson, 2019). Micro frontends are usually discussed at the scale of whole applications or whole routes owned by different teams, but the underlying technical concerns, isolating state, avoiding global namespace collisions, and coordinating navigation, recur at a smaller scale inside a single page that mixes jQuery and React, which is the scale this article addresses.



The islands architecture concept, popularized in a 2020 blog post that also credited earlier framing by Katie Sylor Miller, describes a page that is mostly static or server rendered markup with small, independently hydrated interactive components, or islands, embedded at specific points (Miller, 2020). Although the term was coined primarily to describe replacing heavy client side frameworks with lighter ones for performance reasons, the same architectural shape, a static or legacy shell with isolated interactive regions owned by a different rendering technology, describes exactly what a jQuery page with embedded React components looks like, and Section 4.6 adapts the pattern explicitly for that purpose.

Finally, build tooling changes made coexistence more practical over the period this article covers. Webpack 5's module federation, introduced in 2020, allows independently built and deployed JavaScript bundles to share dependencies and expose modules to each other at runtime, which matters for teams that want to introduce a React bundle without duplicating a jQuery dependency already loaded by the legacy shell, or without forcing a single monolithic build for the whole application (Jackson, 2020). Section 4.7 discusses this and simpler bundling alternatives together.

Table 1. High level comparison of jQuery and React programming models relevant to coexistence

Dimension	jQuery	React
DOM ownership	Implicit, any selector can match any element	Explicit, a root container and its subtree
State location	DOM data attributes, closures, globals	Component state, hooks, or external store
Update model	Imperative, direct mutation on event	Declarative, re render on state change
Typical unit of code	Event handler bound to a selector	Component with props and lifecycle
Rendering trigger	Explicit function call	State or props change, or forced update
Primary release used in this article	3.6.0, March 2021	16.0, September 2017

III. PROBLEM STATEMENT AND MIGRATION CHALLENGES

Running jQuery and React in the same browser tab is not, by itself, difficult. Both are ordinary JavaScript libraries that can be loaded on the same page, and there is no technical barrier to having jQuery code execute in one part of a script and React code execute in another. The difficulty appears specifically at the DOM boundary, where the two libraries' differing assumptions about ownership can conflict. Four recurring problems account for most of the friction reported by teams undertaking this kind of migration.

3.1 DOM Ownership Conflicts

React expects exclusive control over the subtree rooted at the container passed to its render call. If jQuery code matches a selector that happens to fall inside that subtree, for example a global `$("#button").on("click", ...)` handler that was written before any React content existed on the page, and it mutates or removes a node React is tracking, React's internal representation of the DOM becomes inconsistent with the real DOM. On the next re render, React may throw an error such as `Failed to execute removeChild on Node`, or it may silently fail to update the element it believes is still present. The inverse problem also occurs: a React component that unmounts and removes its root container out from under a jQuery plugin that attached data or event listeners to elements inside it can leave the plugin in an inconsistent state or trigger errors in the plugin's own cleanup code, if any exists.

3.2 Event Handling Duplication and Loss

jQuery's event delegation model, in which a single handler is attached to a stable ancestor and matched against a selector for events that bubble up, does not automatically understand React's synthetic event system, which in the React 16 line attaches most handlers at the document root and manages them through its own dispatch mechanism. In a hybrid page it is easy to end up with a delegated jQuery handler that fires for markup rendered by React, because the DOM event still bubbles through native browser event propagation regardless of which library rendered the element, sometimes producing double handling of a single user action when both a jQuery delegated handler and a React `onClick` handler are present, and sometimes producing no handling at all if a component was expected to be caught by delegation but React's



reconciliation replaced the underlying DOM node between renders in a way that broke an assumption the jQuery code made about node identity.

3.3 State Fragmentation

A jQuery region of a page usually keeps its state in the DOM itself, for example a selected row indicated by a CSS class, or in module scoped variables. A React region keeps state in component state or in an external store. When a single logical piece of application state, such as which item in a list is currently selected, needs to be reflected in both regions, the team must decide where that state lives authoritatively and how changes propagate to the other region. Without a deliberate synchronization strategy, teams tend to discover this problem only after users report that an action performed in the React region did not update the jQuery region, or vice versa.

3.4 Build and Bundling Friction

jQuery applications built before 2018 were frequently assembled with script tags, a task runner such as Gulp, or a simple concatenation and minification step, rather than a module bundler with a dependency graph. Introducing React typically requires a bundler such as webpack, along with Babel or a TypeScript compiler for JSX. Teams must decide whether to bring the entire legacy application into the new bundler's dependency graph immediately, which is itself a significant undertaking, or to build the React portion as a separate bundle that is loaded alongside the legacy scripts, which reintroduces the DOM ownership and event questions above at a coarser scale and can also lead to duplicate copies of shared dependencies, most commonly jQuery itself, being downloaded if the two build pipelines are not coordinated. Section 4 addresses each of these four problems directly, with a named strategy and illustrative code for each. Table 5 in Section 6 cross references which of the seven strategies mitigate which of the four problems described here.

IV. HYBRID INTEROPERABILITY STRATEGIES

This section presents seven strategies for running jQuery and React together during an incremental migration. They are not mutually exclusive. A realistic migration plan typically combines several of them at different points: mounting React components into designated containers (Strategy A) alongside a small number of wrapped jQuery plugins (Strategy B), coordinated through an event bridge (Strategy C) or a shared state layer (Strategy D), organized overall around a route based strangler fig plan (Strategy E) or an islands style page shell (Strategy F), and supported by build tooling decisions made explicit up front (Strategy G).

Figure 1. Hybrid DOM Ownership During Coexistence

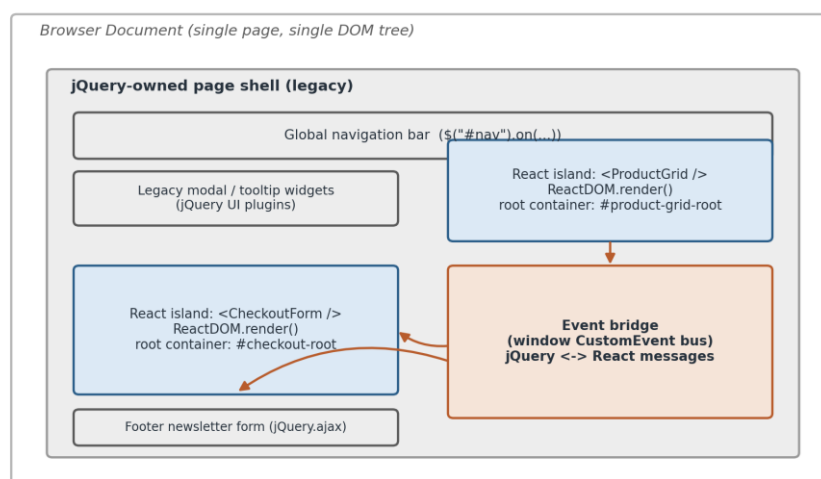


Figure 1. Hybrid DOM ownership during coexistence. The jQuery shell continues to own global navigation, legacy widgets, and the footer, while two React islands are mounted into dedicated containers and communicate with the jQuery shell through an event bridge.



4.1 Strategy A: Mounting React Components Inside jQuery Managed Containers

The most direct coexistence pattern designates specific DOM elements as React roots and leaves everything else to jQuery. A container element, usually an empty div with a stable id, is placed in the server rendered or jQuery generated markup. On page load, after jQuery has finished its own initialization, a small bootstrap script calls ReactDOM.render, passing that container as the second argument, exactly as described in the React documentation's guidance on integrating with other libraries (React Core Team, 2019). The critical rule, which teams that run into DOM ownership conflicts have usually violated, is that no jQuery selector should ever match an element inside that container after the React root has been created. In practice this is enforced by convention and, where possible, by a lint rule or code review checklist that flags jQuery selectors using the container's id or a class name known to appear only inside React managed markup.

```
// legacy-shell.js (jQuery bootstrap, loaded first)
$(function () {
  // jQuery initializes global navigation, legacy widgets, and forms
  $('#nav').on('click', 'a[data-toggle]', handleNavToggle);
  $('.legacy-tooltip').tooltip();

  // Hand off a specific container to React and never touch it again
  if (window.ReactApp && document.getElementById('product-grid-root')) {
    window.ReactApp.mountProductGrid('product-grid-root');
  }
});

// react-entry.js (React bundle, loaded second)
import React from 'react';
import ReactDOM from 'react-dom';
import { ProductGrid } from './components/ProductGrid';

window.ReactApp = window.ReactApp || {};
window.ReactApp.mountProductGrid = function (elementId) {
  var container = document.getElementById(elementId);
  if (!container) { return; }
  ReactDOM.render(React.createElement(ProductGrid, null), container);
};
```

Listing 1. Exposing a named mount function from the React bundle so the jQuery bootstrap can hand off a container without either bundle importing the other directly.

A useful refinement is to also expose an unmount function using ReactDOM.unmountComponentAtNode, and to call it from any jQuery code path that might remove the container from the DOM, for example a jQuery driven tab panel that hides and destroys inactive panel contents. Skipping this step is the single most common source of memory leaks reported in early hybrid jQuery to React migrations, because React continues to hold references and event listeners for a component whose container node has already been detached.

4.2 Strategy B: Wrapping jQuery Plugins as React Components

The inverse problem, using an existing jQuery plugin, for example a date range picker or a rich text editor, from inside a React component, arises frequently because rewriting a mature and well tested plugin in React is often not worth the engineering cost relative to wrapping it. The wrapper pattern uses a React ref to obtain the underlying DOM node, initializes the jQuery plugin on that node inside a lifecycle hook that runs after the DOM has been committed, and tears the plugin down in the corresponding cleanup path so that no listener or timer outlives the component.

```
import React, { useEffect, useRef } from 'react';
import $ from 'jquery';
import 'jquery-ui/ui/widgets/datepicker';

function DatePickerField(props) {
  var inputRef = useRef(null);
```



```
useEffect(function () {
  var $input = $(inputRef.current);
  $input.datepicker({
    dateFormat: 'yy-mm-dd',
    onSelect: function (dateText) {
      props.onChange(dateText);
    }
  });

  return function cleanup() {
    // Always tear down the plugin before React removes the node
    $input.datepicker('destroy');
  };
}, []);

return React.createElement('input', {
  ref: inputRef,
  type: 'text',
  defaultValue: props.initialValue,
  readOnly: true
});
}

export { DatePickerField };
```

Listing 2. A React component that wraps a jQuery UI datepicker plugin, initializing it after mount and destroying it on unmount.

Two details in Listing 2 matter more than they first appear to. First, the input is marked `readOnly` and its displayed value is set through `defaultValue` rather than `value`, because letting React control the `value` prop on every render while the jQuery plugin also writes to the same input creates a race between the two libraries over who last wrote to the DOM node, a version of the ownership conflict described in Section 3.1 at the level of a single element rather than a whole subtree. Second, the effect dependency array is left empty so that the plugin initializes exactly once per mount rather than on every prop change; if the wrapped plugin needs to react to prop changes after initialization, that should be done by calling the plugin's own update method inside a second effect, not by tearing down and reinitializing the whole plugin, which is typically far more expensive and can cause visible flicker.

4.3 Strategy C: Custom Event Bridge for Cross Framework Communication

When jQuery and React regions of a page need to notify each other of user actions without either library importing the other's internals, a lightweight event bridge built on the browser's native `CustomEvent` and `EventTarget` interfaces avoids coupling the two bundles together (Mozilla Developer Network, 2020). The window object, or a dedicated `EventTarget` instance created for the purpose, serves as a shared bus. jQuery code dispatches a `CustomEvent` with a namespaced event name and a detail payload; React code subscribes using `addEventListener` inside a `useEffect` hook and cleans up the listener on unmount, exactly as it would for any other browser event.



Figure 3. Custom Event Bridge Pattern Between jQuery and React

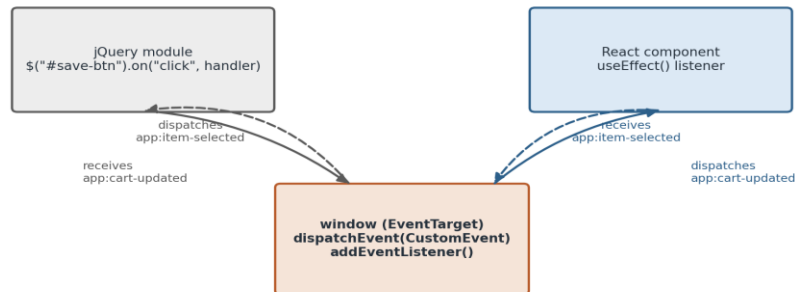


Figure 3. The event bridge pattern. Both the jQuery module and the React component treat the window object as a shared, framework agnostic message bus, dispatching and listening for namespaced CustomEvent instances rather than calling into each other's code directly.

```

// jquery-module.js
$('#save-btn').on('click', function () {
  var itemId = $(this).data('item-id');
  window.dispatchEvent(new CustomEvent('app:item-selected', {
    detail: { itemId: itemId }
  }));
});

window.addEventListener('app:cart-updated', function (event) {
  var count = event.detail.count;
  $('#cart-badge').text(count);
});

// ReactCartBadgeListener.jsx
import { useEffect, useState } from 'react';

function useItemSelection() {
  var [itemId, setItemId] = useState(null);

  useEffect(function () {
    function handler(event) {
      setItemId(event.detail.itemId);
    }
    window.addEventListener('app:item-selected', handler);
    return function () {
      window.removeEventListener('app:item-selected', handler);
    };
  }, []);

  return itemId;
}

function notifyCartUpdated(count) {
  window.dispatchEvent(new CustomEvent('app:cart-updated', {
    detail: { count: count }
  }));
}
    
```



Listing 3. A namespaced CustomEvent bridge. Neither module imports the other; both depend only on the browser's native event dispatch, which keeps the jQuery and React bundles independently buildable.

Namespacing event names with a short application prefix, app: in Listing 3, avoids collisions with third party scripts or browser native events. Because CustomEvent payloads pass through the DOM's native event system, this pattern also works across an iframe boundary with minor adjustment, which is occasionally useful when a legacy page embeds a React micro frontend in an iframe rather than the same document, a variant of Strategy F discussed further in Section 4.6.

4.4 Strategy D: Shared State Synchronization Layer

The event bridge in Strategy C is well suited to discrete notifications, a button was clicked, an item was selected, but becomes unwieldy when many pieces of state need to stay continuously synchronized between regions. For that case, a small shared store, essentially a plain JavaScript object with a subscribe and publish method, gives both libraries a single source of truth. jQuery code reads and writes the store directly through its API; React code subscribes to it using a small custom hook. This is deliberately simpler than adopting a full state management library across the whole application, since the goal during an incremental migration is a minimal seam between two codebases rather than a new architecture for either one.

```
// shared-store.js (no dependency on jQuery or React)
function createStore(initialState) {
  var state = Object.assign({}, initialState);
  var listeners = [];

  return {
    getState: function () { return state; },
    setState: function (patch) {
      state = Object.assign({}, state, patch);
      listeners.forEach(function (fn) { fn(state); });
    },
    subscribe: function (fn) {
      listeners.push(fn);
      return function unsubscribe() {
        listeners = listeners.filter(function (l) { return l !== fn; });
      };
    }
  };
};

window.appStore = createStore({ selectedItemId: null, cartCount: 0 });

// jquery usage
$('#item-1').on('click', function () {
  window.appStore.setState({ selectedItemId: 1 });
});

// react usage (custom hook)
function useAppStore(selector) {
  var [slice, setSlice] = React.useState(function () {
    return selector(window.appStore.getState());
  });
  React.useEffect(function () {
    return window.appStore.subscribe(function (state) {
      setSlice(selector(state));
    });
  }, []);
  return slice;
}
```



Listing 4. A minimal publish and subscribe store shared between jQuery and React through a single global reference, avoiding the need to adopt a full state management library across both codebases.

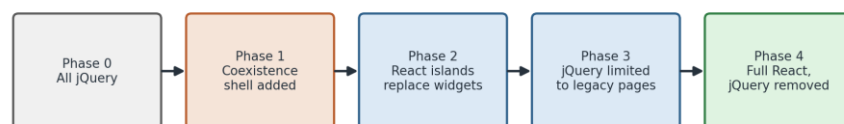
Table 2. Choosing between the event bridge and the shared state layer

Consideration	Favor event bridge (Strategy C)	Favor shared store (Strategy D)
Nature of the interaction	Discrete, occurs once per user action	Continuous, many reads over time
Number of interested listeners	One or a few known listeners	Many components across the page
Need for current value on mount	Not usually needed	Usually needed immediately
Debugging surface	Browser event listener inspector	Custom store, needs its own logging
Typical code volume	Smaller for isolated notifications	Smaller overall as interactions grow

4.5 Strategy E: Route Based Strangler Fig Migration

At the scale of an entire application rather than a single page, the strangler fig pattern described by Fowler (2004) maps naturally onto a route by route migration. Each route, meaning a distinct URL or a distinct top level view in a single page application's own router, is migrated independently, with a thin routing layer deciding whether to hand a given URL to the legacy jQuery driven page or to a React route. Because the decision is made per route rather than per deploy, the team can migrate one screen, ship it, observe it in production, and move to the next screen without the all or nothing risk of a full rewrite.

Figure 2. Five-Phase Incremental Migration Sequence



Each arrow represents one release cycle in the incremental migration plan (Section 4.5).

Figure 2. A five phase incremental migration sequence. Phase 1 introduces the coexistence shell described in Sections 4.1 through 4.4; Phases 2 and 3 move routes and widgets to React one at a time; Phase 4 removes jQuery once no route still depends on it.

```

// router-shim.js: a thin layer in front of both the legacy
// server rendered router and a client side React router.
var MIGRATED_ROUTES = ['/dashboard', '/products/:id', '/settings'];

function isMigrated(pathname) {
  return MIGRATED_ROUTES.some(function (pattern) {
    var regex = new RegExp('^' + pattern.replace(/:/g, '\\:') + '$');
    return regex.test(pathname);
  });
}

function navigate(pathname) {
  if (isMigrated(pathname)) {
    window.ReactApp.renderRoute(pathname, document.getElementById('app-root'));
  } else {

```



```
// Fall through to a normal server navigation for legacy routes
window.location.href = pathname;
}
}
```

Listing 5. A minimal route shim that checks a small allow list before deciding whether a navigation is handled by the React router or left to the legacy server rendered page.

The migration list in Listing 5 is intentionally the single source of truth for which routes have moved. Keeping it centralized, rather than scattering per page feature flags, makes the current migration state auditable at a glance and gives the team an obvious place to update Phase 3 and Phase 4 boundaries in Figure 2 as work proceeds.

4.6 Strategy F: Islands Architecture Adaptation

Where Strategy E migrates whole routes, Strategy F applies the islands architecture idea (Miller, 2020) within a route that is not otherwise being rewritten. A page that remains, for the foreseeable future, primarily server rendered or jQuery driven markup can still have one or two genuinely interactive regions, a live search box, an inline editable table cell, converted into small, independently mounted React islands using exactly the mounting pattern from Strategy A. The distinction from Strategy A is one of intent and scope rather than mechanism: Strategy F treats the surrounding jQuery page as a long lived, intentionally retained shell rather than a temporary state on the way to full React, which is an appropriate choice for pages that see little product investment and do not justify a full migration, while Strategy A is typically discussed as the first step of migrating a page's ownership fully to React over time.

This distinction matters for planning purposes because it changes what counts as done. A team applying Strategy F to a rarely changed internal reporting page should not expect, or budget for, that page ever reaching Phase 4 in Figure 2. Making this expectation explicit early avoids the common planning error of treating every hybrid page as mid migration when some of them are, deliberately, a stable long term architecture.

4.7 Strategy G: Build Tooling and Bundling Considerations

The strategies above all assume jQuery and React are available in the same page, but say little about how the two are built and delivered to the browser. Three approaches were in common use during the period this article covers. The simplest, loading jQuery from a script tag or a bundled legacy build and loading a separately built React bundle alongside it, requires no changes to the legacy build pipeline but risks shipping two copies of any shared dependency and requires care to avoid global namespace collisions, typically resolved by using jQuery's noConflict mode when both a global \$ from the legacy build and a module scoped jQuery import in the React bundle are present.

```
// legacy-shell.js, loaded first, keeps window.$ for existing code
// react-entry.js, loaded second
import jQuery from 'jquery';
var $react = jQuery.noConflict(false); // does not touch window.$
// react bundle now uses $react internally if it needs jQuery at all,
// while all existing legacy code continues to use the global $
```

Listing 6. Using jQuery's noConflict mode to let a separately bundled React entry point use its own reference to jQuery without disturbing the global \$ that legacy code depends on.

A second approach folds the legacy scripts into the same webpack build used for React, treating jQuery as an ordinary dependency and the legacy scripts as ordinary entry points or chunks. This unifies the dependency graph, which makes it possible to deduplicate jQuery to a single shared chunk and to apply consistent linting and minification, but requires an upfront investment in adapting scripts that were written assuming global script tag execution order rather than module boundaries, since such scripts frequently rely on side effects, for example a script that defines a global function expected to already exist when a later script tag runs, that a bundler's module system does not preserve unless explicitly reproduced.

A third approach, available from webpack 5 onward, is module federation, in which the legacy application and the new React application are built and deployed as separate bundles that can nonetheless share a single runtime copy of a common dependency such as jQuery or React itself, declared explicitly rather than duplicated (Jackson, 2020). This is closer in spirit to the micro frontend model referenced in Section 2.3 and is most useful when the legacy and React portions of the



application are owned by different teams with independent release cadences, since it avoids forcing those teams onto a single synchronized build and deploy pipeline while still avoiding duplicate dependency downloads for the end user.

Regardless of which bundling approach a team selects, the general cost of shipping additional JavaScript to the browser, and the corresponding parse, compile, and execution time on lower powered client devices, remains a real constraint during the coexistence period, since Phases 1 through 3 in Figure 2 necessarily ship both libraries at once (Osmani, 2018). Teams operating under strict performance budgets have reported mitigating this by code splitting the React bundle per route, so that a user visiting an unmigrated legacy route never downloads the React runtime at all, which keeps the added weight described by Osmani (2018) scoped only to the routes that have actually been migrated rather than applied globally from Phase 1 onward.

4.8 Testing Strategies for Hybrid Pages

Section 6 recommends automated end to end coverage across the jQuery and React boundary, and this subsection describes what that coverage looks like in practice. A browser based end to end test, run against a real rendered page rather than a simulated DOM, is the only test level that observes both libraries' actual effect on the same document at once, which is precisely where the DOM ownership and event handling problems from Sections 3.1 and 3.2 occur. Unit tests for the React components in isolation, using a React specific testing utility, and unit tests for jQuery modules in isolation, remain valuable for verifying each side's own logic, but neither can observe a conflict that only appears when both sides act on the same page in a specific order.

```
// hybrid-boundary.spec.js
// End to end test exercising both the jQuery shell and a React island
// on the same page, written against a browser automation API.
describe('product grid react island inside jquery shell', function () {
  it('updates the jquery cart badge after a react selection', function () {
    cy.visit('/catalog');

    // Interact with the React managed region
    cy.get('#product-grid-root [data-testid="item-42"]').click();

    // Assert against the jQuery managed region, confirming the
    // event bridge from Strategy C carried the update across
    cy.get('#cart-badge').should('contain.text', '1');

    // Trigger a jQuery driven tab switch that hides the React
    // container, then confirm no console error was raised by
    // an unmount that jQuery interrupted mid transition
    cy.get('#tab-orders').click();
    cy.get('#product-grid-root').should('not.be.visible');
    cy.window().then(function (win) {
      expect(win.__testConsoleErrors || []).toHaveLength(0);
    });
  });
});
```

Listing 7. A boundary spanning end to end test that exercises a user action in the React island, asserts the effect in the jQuery managed region through the event bridge, and checks that a jQuery driven visibility change does not raise a React reconciliation error.

Two practices made this kind of test more reliable in the migrations reviewed for this article. First, capturing console errors during the test run, rather than only checking for thrown exceptions that halt the test itself, surfaced React reconciliation warnings that did not stop the page from functioning but indicated a DOM ownership conflict was occurring silently, of exactly the kind described in Section 3.1. Second, writing a small number of tests that deliberately trigger the transition moments identified as risky in Section 3, a jQuery driven show or hide of a container holding a React root, a rapid double interaction that could double fire a handled event, found more defects per test than a larger number of tests that only exercised the two libraries' regions independently and never at their shared boundary.



V. ILLUSTRATIVE CASE STUDY

To make the trade offs discussed in Section 4 concrete, this section walks through an illustrative migration of a hypothetical mid size internal operations application, referred to here as the Fleet Ops Console, a jQuery application originally built in 2015 for scheduling and tracking a delivery fleet, with roughly 40 distinct screens and an established base of internal users. The figures presented in this section are simulated for illustration and structured to reflect patterns commonly reported in migration retrospectives; they are not measurements from a real deployment and should be read as a worked example rather than as empirical findings.

5.1 Migration Plan

The Fleet Ops team adopted the five phase sequence shown in Figure 2. Phase 1 introduced the coexistence shell, meaning the mount and unmount helpers from Strategy A, the event bridge from Strategy C, and the noConflict setup from Strategy G, without changing any user facing behavior. Phase 2 replaced the three highest traffic screens, the dispatch board, the driver detail view, and the daily schedule, with React implementations, chosen first because they received the most bug reports in the twelve months prior to the migration and stood to benefit most from better testability. Phase 3 continued screen by screen replacement over the following year, using Strategy E's route allow list to track progress. Phase 4 removed the jQuery bundle entirely once the allow list covered all 40 screens and no legacy route remained.

Table 3. Illustrative migration timeline and scope by phase, Fleet Ops Console case study

Phase	Approximate duration	Screens on React by phase end	Primary strategies applied
Phase 0, baseline	n/a	0 of 40	None, all jQuery
Phase 1, coexistence shell	6 weeks	0 of 40	A, C, G
Phase 2, high traffic screens	10 weeks	3 of 40	A, B, C, D
Phase 3, remaining screens	58 weeks	40 of 40	A, B, D, E
Phase 4, jQuery removal	3 weeks	40 of 40	G

5.2 Illustrative Quality and Performance Indicators

Table 4 presents illustrative defect and performance indicators tracked across the migration. Two points are worth noting about how to read this kind of data in a real migration. First, a short term increase in reported defects immediately after Phase 2 is a commonly reported pattern and, in this illustrative scenario, is attributed to the DOM ownership and event duplication failure modes described in Sections 3.1 and 3.2 before the team's linting and code review checklist for the coexistence rules had fully caught up with developer habits. Second, bundle size temporarily increases during the coexistence period, Phases 1 through 3, because both jQuery and React are shipped together, and only falls once jQuery is removed in Phase 4, which is an expected and typically accepted cost of the incremental approach rather than a defect in it.

Table 4. Illustrative defect counts and client bundle size by phase, Fleet Ops Console case study

Phase	Production defects reported (illustrative, per month)	Client bundle size (illustrative, gzipped)
Phase 0, baseline	6	310 KB (jQuery only)
Phase 1, coexistence shell	5	420 KB (jQuery plus React runtime)
Phase 2, high traffic screens	11	460 KB



Phase	Production defects reported (illustrative, per month)	Client bundle size (illustrative, gzipped)
Phase 3, remaining screens	7 declining to 4	540 KB, peak mid phase
Phase 4, jQuery removal	3	365 KB (React only)

The illustrative defect spike in Phase 2, from a baseline of six to eleven reported issues per month, and its subsequent decline back below baseline by the end of Phase 3, is consistent with anecdotal reports from teams that adopted an event bridge (Strategy C) and a shared state layer (Strategy D) partway through their own migrations rather than from the outset. This suggests, as a practical recommendation rather than a proven finding given the illustrative nature of the data, that establishing the coexistence rules and communication seams in Phase 1, before any user facing screen is migrated, is preferable to introducing them reactively once problems appear in Phase 2.

VI. DISCUSSION: TRADE OFFS AND RECOMMENDATIONS

No single strategy from Section 4 is sufficient on its own for a nontrivial migration; the strategies address different layers of the same problem and are typically combined. Table 5 summarizes which of the four challenges from Section 3 each strategy primarily addresses, to make the combination decision easier for a team planning its own migration.

Table 5. Strategy comparison matrix, mapping each strategy to the challenges it primarily addresses

Strategy	DOM ownership (3.1)	Event handling (3.2)	State fragmentation (3.3)	Build friction (3.4)
A. Mounting into containers	Primary	Partial	No	No
B. Wrapping jQuery plugins	Primary	Partial	No	No
C. Event bridge	No	Primary	Partial	No
D. Shared state layer	No	No	Primary	No
E. Route based strangler fig	Partial	No	No	Partial
F. Islands adaptation	Primary	Partial	No	No
G. Build tooling choices	No	No	No	Primary

Two practical recommendations follow from the case study and the strategy comparison. First, teams should establish the coexistence seams, the mounting convention from Strategy A and either the event bridge or the shared state layer, before migrating any user facing screen, matching the Phase 1 approach in Section 5 rather than retrofitting these seams after problems appear. Second, the choice between Strategy C and Strategy D should be revisited as the migration progresses rather than fixed at the outset; Table 2 in Section 4.4 gives a lightweight decision guide, but in practice most migrations reviewed for this article started with an event bridge for a small number of interactions and introduced a shared store only once the number of cross framework interactions grew large enough that individually named events became hard to track.

A further recommendation concerns testing. Because DOM ownership conflicts (Section 3.1) often manifest only when a specific sequence of jQuery and React operations occurs, for example a jQuery driven tab switch that hides a container immediately before a pending React state update would have re-rendered into it, automated end to end tests that exercise realistic user flows across both regions of a hybrid page catch a category of defect that unit tests scoped to either library in isolation typically miss. Teams that added even a small number of such cross boundary end to end tests during Phase 1 in migrations reviewed for this article reported catching ownership conflicts before they reached production more often than teams that relied solely on unit tests for the newly written React components.



VII. LIMITATIONS

This article has several limitations that should inform how its conclusions are used. The case study in Section 5 is illustrative and its numeric figures are simulated for pedagogical clarity rather than drawn from a controlled measurement of a real production system; readers should treat the shape of the trends, an early defect increase followed by a decline, and a temporary bundle size increase during coexistence, as plausible and broadly consistent with informal industry reports, not as a validated quantitative result. No formal controlled comparison, for example two matched teams migrating comparable applications with and without a given strategy, is presented, and such a comparison would be difficult to construct given how much migration outcomes depend on team specific factors such as existing test coverage, application complexity, and organizational priorities that are hard to hold constant across a comparison.

The seven strategies are also not exhaustive. Approaches such as rendering React components to static markup on the server and progressively hydrating them within an otherwise jQuery driven page, or using a custom element or web component boundary rather than a bare DOM container to encapsulate a React island, are related but were judged outside the scope of an article focused on the coexistence patterns most directly reported in jQuery specific migration discussions during the period covered. Finally, the article focuses on the version of React current at the time of writing and does not attempt to anticipate how future changes to React's internals, for example changes to how and when the reconciler commits DOM updates, might alter the specific failure modes described in Section 3, although the general problem of DOM ownership conflicts between an imperative library and a declarative one is likely to persist in some form across such changes.

VIII. FUTURE WORK

Three directions would strengthen the empirical basis of the recommendations made in Section 6. First, a structured survey of engineering teams that have completed or are midway through a jQuery to React migration, collecting standardized metrics such as defect rates, migration duration, and which combination of strategies from Section 4 were used, would let the illustrative patterns in Section 5 be checked against real, aggregated data rather than a single worked example. Second, tooling that automatically detects DOM ownership conflicts, for example a development mode instrumentation layer that warns when a jQuery selector matches a node inside an active React root, would directly address the most commonly reported failure mode in Section 3.1 and would be a natural complement to the manual convention described in Strategy A. Third, as build tooling continues to evolve, a follow up comparison of module federation against simpler dual bundle approaches, measured on real applications rather than described qualitatively as in Section 4.7, would help teams choose between them with more confidence than the qualitative guidance offered here.

IX. CONCLUSION

Migrating a production application from jQuery to React rarely happens, and in most organizations should not happen, in a single release. The seven strategies described in this article, mounting React into designated containers, wrapping jQuery plugins as React components, bridging the two libraries with namespaced custom events, sharing state through a minimal store, migrating routes according to a strangler fig plan, adapting the islands architecture for long lived hybrid pages, and making deliberate build tooling choices, give a team a practical vocabulary for the coexistence period rather than treating it as an unavoidable but unaddressed source of risk. The illustrative case study in Section 5, while not a substitute for controlled empirical measurement, suggests that establishing these coexistence patterns early, before any user facing screen is migrated, reduces the defect risk that otherwise concentrates around the first screens moved to React. As React's own documentation has long acknowledged in its guidance on integrating with other libraries, coexistence is an expected and supported scenario, not an edge case, and treating it with the same deliberate design attention as any other part of the application architecture is the practical takeaway of this article for teams planning a 2022 migration.

Appendix A: Coexistence Readiness Checklist

The following checklist consolidates the practical rules embedded throughout Section 4 into a single reference a team can use before beginning Phase 1 of an incremental migration, or when auditing an existing hybrid page for the DOM ownership and event handling problems described in Section 3. It is organized by the strategy each item primarily supports.



A.1 Mounting and unmounting (Strategy A, Section 4.1)

1. Every React root has a stable, uniquely named container id that is never matched by a jQuery selector once the root has been created.
2. A corresponding unmount function using ReactDOM.unmountComponentAtNode exists for every mount function, and every jQuery code path that can remove or hide a React container calls it first.
3. The bootstrap script that calls the mount function runs after jQuery has finished its own initialization for that page, not before.
4. A lint rule or code review checklist item flags any new jQuery selector that targets a class or id known to appear only inside React managed markup.

A.2 Wrapped plugins (Strategy B, Section 4.2)

1. Every jQuery plugin wrapped as a React component initializes inside a lifecycle hook that runs after the DOM has committed, and is destroyed in the corresponding cleanup path.
2. Form fields controlled by a wrapped plugin are not simultaneously driven by React's value prop, avoiding the dual write race described in Section 4.2.
3. Prop changes after initial mount call the plugin's own update method where one exists, rather than tearing down and reinitializing the plugin on every change.

A.3 Cross framework communication (Strategies C and D, Sections 4.3 and 4.4)

1. All custom event names are namespaced with an application specific prefix to avoid collisions with third party scripts.
2. Every addEventListener call added for cross framework communication has a matching removeEventListener call in the appropriate cleanup path.
3. The team has an explicit, written answer to the question of which region owns a given piece of shared state, rather than leaving ownership implicit.
4. If a shared store is in use, its subscribe method is called from inside a React effect with an empty dependency array, and the returned unsubscribe function is invoked on cleanup.

A.4 Migration planning and build (Strategies E, F, and G, Sections 4.5 through 4.7)

1. A single, centralized list of migrated routes or screens exists and is treated as the source of truth for migration status, rather than scattered per page flags.
2. Pages intentionally kept as a long lived jQuery shell with React islands, per Strategy F, are documented as such so they are not mistakenly treated as mid migration.
3. The bundling approach, separate bundles, a unified build, or module federation, has been chosen deliberately and documented, rather than left as an accident of how the React tooling was first introduced.
4. If jQuery is available both as a global and as a bundled module in the same page, noConflict mode is used to prevent the two references from interfering with each other.

A.5 Testing (Section 4.8)

1. At least a small number of automated end to end tests exercise a user action in one region of a hybrid page and assert its effect in the other region.
2. End to end tests capture browser console errors during the run, not only thrown exceptions, since a silent React reconciliation warning is a common symptom of a DOM ownership conflict.
3. Tests deliberately exercise the transition moments identified as risky, a jQuery driven show or hide of a container holding a React root, and a rapid repeated interaction that could double fire a handled event.

REFERENCES

1. Coyier, C. (2015). You Might Not Need jQuery. CSS Tricks. Retrieved from <https://css-tricks.com/might-need-jquery/>
2. Fowler, M. (2004). StranglerFigApplication. martinowler.com bliki. Retrieved from <https://martinfowler.com/bliki/StranglerFigApplication.html>
3. Greif, S., Bongiovanni, R., & the State of JS contributors. (2020). State of JavaScript 2020: Front End Frameworks. Retrieved from <https://2020.stateofjs.com/en-us/technologies/front-end-frameworks/>
4. Jackson, C. (2019). Micro Frontends. martinowler.com. Retrieved from <https://martinfowler.com/articles/micro-frontends.html>



5. Jackson, Z. (2020). Webpack 5 Module Federation: Stitching Two Simple Bundles Together. Medium. Retrieved from <https://scriptedalchemy.medium.com/webpack-5-module-federation-stitching-two-simple-bundles-together-fe4e6a069716>
6. jQuery Foundation. (2016). jQuery 3.0 Final Released! Official jQuery Blog. Retrieved from <https://blog.jquery.com/2016/06/09/jquery-3-0-final-released/>
7. jQuery Foundation. (2021). jQuery 3.6.0 Released! Official jQuery Blog. Retrieved from <https://blog.jquery.com/2021/03/02/jquery-3-6-0-released/>
8. Miller, J. (2020). Islands Architecture. jasonformat.com. Retrieved from <https://jasonformat.com/islands-architecture/>
9. Mozilla Developer Network. (2020). CustomEvent. MDN Web Docs. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent>
10. Osmani, A. (2018). The Cost of JavaScript in 2018. Medium. Retrieved from <https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4>
11. React Core Team. (2017). React v16.0. React Blog. Retrieved from <https://legacy.reactjs.org/blog/2017/09/26/react-v16.0.html>
12. React Core Team. (2019). Integrating with Other Libraries. React Documentation. Retrieved from <https://legacy.reactjs.org/docs/integrating-with-other-libraries.html>
13. React Core Team. (2019). ReactDOM. React Documentation. Retrieved from <https://legacy.reactjs.org/docs/react-dom.html>