EWIS AEGIS
Modernize with Certainty.

CASE STUDY

MODERNIZING A LEGACY BANKING APPLICATION

ASP.NET WebForms → .NET 10 + React 19

Enterprise letter-management system — runtime, security, and test-coverage modernization

`EWIS AEGIS engagement reference • Mission-critical banking`

156	677+	0	53
SQL INJECTION	AUTOMATED TESTS	DATA MIGRATION	REST ENDPOINTS
sites closed	added (from zero)	database untouched	across 7 controllers

✓ ENGAGEMENT OUTCOME — MODERNIZED RUNTIME AND SECURITY; BEHAVIOR PRESERVED VERBATIM

A business-critical letter-management application running on end-of-life **ASP.NET WebForms / .NET Framework 3.5** was rebuilt on **.NET 10 + React 19** under Clean Architecture, with stateless JWT authentication, Kubernetes deployment, and a 677-test parity-and-regression suite. **156 SQL-injection sites were closed**, the fragile keyword blacklist retired, and the underlying IBM DB2 / AS400 database was deliberately left untouched — same schemas, same tables, zero migration risk. Behavioral fidelity was proven by construction, not by sign-off.

Domain	Enterprise banking — customer letter management (balance & advance letters)
Migration	ASP.NET WebForms (.NET Framework 3.5) → ASP.NET Core (.NET 10) + React 19 SPA
Database	IBM DB2 / AS400 — unchanged, no migration
Deployment	IIS on-prem → Docker images on Kubernetes (DEV / UAT / PREPROD)
Architecture	Clean Architecture (API / Application / Core / Infrastructure / Web)
Test coverage	**677+ automated tests** — unit, parity, regression, end-to-end (legacy: none)
Engagement type	Like-for-like behavioural rebuild on modern stack
Prepared by	EWIS AEGIS

Modernize with Certainty.

Overview

A business-critical letter-management application — used by branch and back-office staff to create, authorize, and print customer-facing balance and advance letters — was running on an end-of-life ASP.NET WebForms stack. EWIS AEGIS rebuilt the system on a modern .NET 10 + React 19 architecture, **modernizing the runtime and security posture while preserving the exact business logic** of the original system.

The application was re-engineered functional-area by functional-area. The underlying core-banking database was deliberately left untouched: same schemas, same tables, same data. **No data migration was required**, eliminating an entire class of cutover risk.

Domain	Enterprise banking — customer letter management
Migration	ASP.NET WebForms → .NET 10 (ASP.NET Core) + React 19
Database	IBM DB2 / AS400 — unchanged, no migration
Test coverage	677+ automated tests (unit, parity, regression, E2E)
Original test coverage	**None**

The Challenge

The legacy application carried the typical risk profile of a long-lived enterprise WebForms system:

- **End-of-life runtime.** .NET Framework 3.5 on IIS, with page-centric postbacks and ViewState — increasingly difficult to secure, host, and staff for.
- **Monolithic data access.** A single ~7,367-line data-access class held 100+ database methods, with code-behind business logic embedded directly in each page.
- **String-concatenated SQL.** Data access built queries by string concatenation, with **156 documented SQL-injection sites**. The only mitigation was a runtime keyword blacklist attempting to catch tokens such as `--`, `;`, `select`, and `update`.
- **Opaque workflow state.** Letter status was written inconsistently across many update methods, forcing consumers to inspect several columns to infer the true state.
- **Stateful, session-bound auth.** Authentication relied on server-side session state and an HTTP-referer trust check.
- **No automated tests.** Every change was a manual-regression gamble.

> **THE DEFINING CONSTRAINT**
>
> The rebuilt system had to behave **identically** to the original at the business-logic level, against the **same production database**, with zero schema migration. Modernization was a runtime, security, and architecture exercise — not a behavioural one.

The Approach

EWIS AEGIS treated the legacy monolith as the authoritative specification and rebuilt it under Clean Architecture, validating equivalence with an **automated parity suite** rather than relying on manual sign-off.

Stack Transition

Layer	Legacy	Modernized (V2)
UI framework	ASP.NET WebForms, Master Pages, postbacks + ViewState	React 19 SPA + TypeScript + Vite + Tailwind CSS + TanStack Query
API style	Page-centric (each page returns rendered HTML)	REST / JSON — 53 endpoints across seven controllers
Backend runtime	.NET Framework 3.5 on IIS	.NET 10 (ASP.NET Core Web API) in a Linux container
Architecture	Monolithic; code-behind drives a single large data-access class	Clean Architecture: API / Application / Core / Infrastructure / Web
DB driver	OLE DB	Modern DB2 provider via Dapper
Database	IBM DB2 / AS400	Same database, same schemas, same tables — no migration
Authentication	Server-side session + HTTP-referer check	JWT bearer tokens + React auth context; referer check preserved
Deployment	IIS on-prem	Docker images on Kubernetes
Automated tests	None	677+ tests across unit, parity, regression, and E2E suites

Clean Architecture Mapping

The monolith was decomposed layer by layer:

- The single large data-access class became a set of **repository interfaces** (Core) with concrete implementations (Infrastructure) — enabling dependency inversion and unit-testable services.
- Session- and directory-bound logic was centralized into dedicated **authentication and core-banking services**.
- Business logic embedded in page code-behind moved into **application services**, reusable across multiple controllers.
- Inline HTML letter assembly was replaced with a proper **PDF document-generation layer**.

Engineering Highlights

Closing 156 SQL-Injection Sites

Every data-access method in the legacy system built SQL by string concatenation, guarded only by a runtime keyword blacklist. In the rebuilt system, **every statement executes through parameterized queries**. All 156 injection sites are closed, the brittle blacklist was removed entirely, and a dedicated regression suite asserts the vulnerable patterns cannot return.

Critically, the DB2-specific SQL idioms the business depends on — `SELECT … FROM FINAL TABLE (INSERT …)`, `CURRENT DATE` / `CURRENT TIME`, `FETCH FIRST N ROWS ONLY`, zero-padded sequence generation, and `TRIM(…)` on padded CHAR columns — were preserved **byte-identically** across both codebases.

> **VULNERABILITY CLASS ELIMINATED**
>
> A regression test suite encodes the legacy injection patterns and asserts they cannot return. The brittle keyword blacklist that attempted to catch `--`, `;`, `select`, `update` at runtime has been retired entirely — safety is now structural, not defensive.

A Deterministic Workflow State Machine

Legacy status was an opaque string column written inconsistently throughout the system. The rebuild derives status **deterministically** from five atomic columns through a single parsing function, with a whitelisted set of valid transitions:

```
Draft     → Forward (→ Initiated)  ·  Update (stays Draft)  ·  Delete
Initiated → Approve  ·  Decline
Approved  → Print
Printed   → Reprint
Reprinted → Reprint (idempotent)
Declined  → Update (→ Draft)
```

The legacy status column is still written so downstream reports that scrape it keep functioning — but it is no longer the source of truth.

Stateless Authentication, Same Trust Boundary

Server-side session state was replaced with stateless **JWT bearer tokens** and a React auth context. The external-portal login flow and its **HTTP-referer trust boundary** were preserved, but the allowed-referer list moved into environment configuration — new entries can be added per environment without a code change. Enterprise OIDC integration was scaffolded alongside for production use.

Role-Based Authorization and Dual Control

Three legacy master pages were consolidated into a single layout whose menu is filtered by the user-type claim in the JWT. A central authorization function returns the exact set of actions a user may perform for a given status and role, enforced on every mutating call. **Dual control is enforced in code**: the user who created a letter cannot approve it.

Document Generation

In-page HTML assembly that relied on browser CSS for print layout was replaced with a **programmatic PDF generator** offering three templates (balance confirmation, advance letter, and an Islamic-banking variant). Amount-to-words conversion was reimplemented as a dedicated utility and unit-tested against the legacy outputs.

Auditing

The original inline audit-log and fee-transaction writes were wrapped in dedicated repository methods invoked at each state transition. In addition, structured JSON logs are now emitted for every request and mutation, consumable by centralized log aggregation outside the database.

Testing: A New Capability

The legacy codebase had **no automated tests**. The rebuilt system ships with **677+ tests** across four categories:

Category	Coverage
Unit	Services, DTOs, entities, workflow, PDF generation, amount-to-words
Feature parity	Maps legacy method signatures to new service methods and asserts equivalent behavior
Regression	SQL-injection prevention, CHAR-padding handling
End-to-end	Full workflow: create → forward → approve → print (browser-driven)

> **THE PARITY SUITE IS THE LINCHPIN**
>
> The feature-parity suite is what enables the "modernize without changing behavior" guarantee — it pins the new implementation to the documented legacy contract. Every method signature in the old data-access class has a corresponding parity test that exercises the new service and asserts equivalent input-to-output behaviour.

Deployment

	Legacy	Modernized (V2)
Packaging	Deployed to IIS as a web application	Two Docker images (API + web)
Orchestration	Manual IIS configuration	Kubernetes across DEV / UAT / PREPROD
Configuration	Hardcoded connection string and keys in a config file	Environment variables via ConfigMap + Secret
Secret management	Plaintext in a config file	Kubernetes Secret / managed secret store
External dependencies	Accessed over the enterprise network	

	Legacy	Modernized (V2)
		Unchanged — database, directory, and identity services remain outside the cluster

All environments share the same two images, with only configuration and secrets differing per environment.

Business Logic Preserved Verbatim

The following rules were carried across identically, validated by the parity suite:

- **Letter lifecycle:** Draft → Initiated → Approved → Printed (Reprinted on re-issue; Declined branches back to Draft).
- **Dual control:** The creator of a letter cannot approve it.
- **Fee collection:** Optional at approval; when a fee is charged, a debit account is required and a transaction row is written; when no fee is charged, a waiver reason is required.
- **Waiver reasons:** Drawn from an admin-managed reference table; an "Other" selection accepts a free-text qualifier stored in the legacy format.
- **Signatory eligibility:** Trainee and junior designations are excluded uniformly across both the forward and approve surfaces.
- **Customer auto-population** on identifier lookup during letter creation.
- **Property-data re-use** for follow-on letters for the same customer.
- **Industry-specific special handling** driven by a branch-code configuration setting.

Outcomes

156
SQL INJECTION
sites closed

677+
AUTOMATED TESTS
where none existed

0
DATA MIGRATION
database untouched

- **Eliminated an entire vulnerability class.** 156 SQL-injection sites closed and locked down by regression tests; the fragile blacklist retired.
- **Replaced an end-of-life runtime** with a containerized, horizontally scalable .NET 10 + React 19 platform on Kubernetes.
- **Introduced 677+ automated tests** where none existed, converting every future change from a manual gamble into a verifiable one.
- **Zero data migration.** The same production database, schemas, and tables continue to serve both downstream consumers and the new application.
- **Behavioral fidelity by construction**, proven by a feature-parity suite rather than asserted by sign-off.
- **Modernized security posture** — stateless JWT auth, externalized secrets, structured centralized logging — without disturbing the established trust boundaries staff already rely on.

ENGAGEMENT REFERENCE

End of Case Study. • Banking Application Modernization • ASP.NET WebForms → .NET 10 + React 19

EWIS AEGIS • Modernize with Certainty.

Engineered for enterprise legacy modernization.

156 vulnerabilities closed • 677+ automated tests • 0 data migration • REFERENCE MATERIAL — DISTRIBUTED UNDER NDA