Elements as a Game Runtime

Fundamentals · https://namazustudios.com/docs/fundamentals/elements-as-a-game-runtime/

Elements is not just a backend service that you call from your game - it is a server-side runtime capable of hosting your game's authoritative logic.

This page explains what that means, how it differs from traditional architectures, and when you might choose to run your game logic directly inside Elements.

The Big Idea

In most game architectures, you write game logic in one place and run it somewhere else:

- a custom backend service
- a cloud function platform
- a managed game server

With Elements, you have another option - your game's authoritative logic can run *inside* Elements itself.

This is done by writing Custom Elements, which are full server-side applications hosted and executed by the Elements platform.

What a Custom Element Is

A Custom Element is:

- a server-side Java application
- loaded and executed by Elements
- running in a trusted, non-player-controlled environment
- isolated from other Elements at the classpath level

A Custom Element is *not*:

- client code
- a scripting sandbox
- a restricted rules engine

You write real Java, with full control over logic and data access.

Acting as the Game Server

A Custom Element can act as the core game server.

Depending on your game, this may include:

- handling REST requests for turn-based gameplay
- managing real-time sessions via WebSockets or WebRTC
- validating player actions
- enforcing game rules
- mutating authoritative game state

From the player's perspective, the client still communicates with "the server" - the difference is that the server logic lives inside Elements.

Direct Access to Game State

Custom Elements have unrestricted access to the Elements data model.

This allows you to:

- read and write directly to the database
- manage custom tables and relationships
- integrate tightly with (and even expand upon) Elements systems, such as inventory, rewards, missions, and leaderboards

Because this code runs authoritatively, there is no need to proxy through an external backend just to apply state changes.

Real-Time and Turn-Based Games

Elements supports both interaction models:

Turn-Based Games

- Clients make REST calls
- Custom Elements validate and apply moves
- Game state is stored and enforced by Elements

This model is well-suited for:

- async games
- strategy games
- games with simple state transitions

Real-Time Games

- Clients establish persistent connections (WebSockets / WebRTC)
- Custom Elements manage sessions and events
- Elements acts as the authoritative host

This model is well-suited for:

- real-time multiplayer
- synchronous co-op or PvP
- fast-paced interactions requiring low latency

Composability and Multiple Elements

Multiple Custom Elements can be loaded at the same time.

This enables a composable architecture, where:

- one Element handles core gameplay
- another handles purchases or receipt validation
- another manages promotions or live events

Each Element:

- runs independently
- is classpath-isolated
- can be developed, tested, and deployed separately

Example: Payment Processing as a Custom Element

A payment provider can ship a Custom Element that:

- validates receipts against external app stores
- verifies transactions
- issues digital goods using Elements inventory APIs

For game developers, this means:

- no custom purchase-validation backend
- no duplicated logic across games
- a trusted, reusable integration

Using Elements Alongside External Services

Running your game inside Elements does *not* prevent you from using other services.

Common hybrid setups include:

- Custom Elements for authoritative gameplay logic
- external services for analytics, matchmaking, or social features

Elements does not force a single architecture - it provides a runtime where authority can safely live.

Choosing the Right Model

You might choose to run your game logic inside Elements if:

- you want to minimize backend infrastructure
- you want authoritative access to game state
- your logic is tightly coupled to Elements features
- you want composable, reusable server-side modules

You might choose an external server if:

- you already have significant backend investment
- your architecture depends on specialized infrastructure
- you want to keep Elements focused on state and services

Many teams generally prefer to keep everything within Elements, but sometimes it makes sense to use both.

How This Fits Into the Rest of the Docs

When other documentation refers to:

- “server-side code”
- “authoritative logic”
- “Custom Elements”

It may be referring to logic that runs inside Elements itself.

Understanding Elements as a game runtime will make feature-level documentation clearer and more flexible.

What to Read Next

- **Why You Need Authority** – understanding trust and permanence

- **Where Your Authoritative Code Runs** – choosing between deployment models
- **Lifecycles and Flows** – seeing how systems work end-to-end