A lightweight Browser-Based Projection Layer for Runtime-Driven UI Systems

1. Introduction

Modern software increasingly depends on graphical user interfaces that must operate consistently across platforms, devices, and execution environments. Despite decades of progress in desktop application frameworks, building lightweight, portable, and maintainable UI layers remains an architectural challenge. Many existing solutions introduce substantial runtime dependencies, platform-specific abstractions, tightly coupled rendering systems, or deployment overhead that negatively impacts accessibility, maintainability, and long-term extensibility. Even cross-platform alternatives often exchange these problems for architectural complexity, large runtime footprints, and deeply integrated rendering pipelines. As a result, application logic, rendering, layout, lifecycle ownership, and frontend runtime behavior frequently become intertwined.

The previously introduced Data Driven UI Framework addressed many of these issues through a lightweight ECS-based UI architecture centered around stateless systems, reactive event streams, and strict separation between data, logic, and rendering. Instead of relying on traditional widget hierarchies or runtime-bound object models, the framework represents UI state as composable component data processed through deterministic layout, input, and rendering passes. Rendering backends remain interchangeable by consuming streams of primitive draw commands rather than framework-owned visual objects, enabling platform-independent UI composition with minimal runtime overhead. This paper introduces a complementary browser-based projection and display orchestration layer designed to operate on top of the previously defined runtime architecture. Its purpose is not to create another browser framework, embedded web runtime, or browser-centric application platform. Instead, it repurposes already installed system browsers as lightweight rendering and interaction runtimes capable of acting as distributed display surfaces.

Modern browsers already provide highly optimized rendering pipelines, GPU acceleration, advanced text layout systems, input handling, networking stacks, accessibility integration, session management, and broad cross-platform compatibility. These capabilities are available by default on virtually all modern operating systems and mobile devices. Many browser-based application architectures, however, place significant portions of application ownership inside the browser itself, frequently leading to duplicated state management, hydration complexity, fragmented execution lifecycles, frontend/backend synchronization issues, and heavy framework dependencies.

The model proposed in this work instead treats the browser as a remotely attachable presentation and interaction node while authoritative application state, orchestration, synchronization, and execution ownership remain entirely within the host runtime. User interfaces are projected into browser instances through synchronized runtime surfaces and reactive stream communication rather than being

executed as self-contained browser applications.

The browser is therefore not considered the application runtime itself. Instead, it becomes a reusable projection target capable of rendering runtime-controlled UI surfaces locally or remotely. By leveraging already installed browser engines such as Chromium- and Gecko-based implementations, applications avoid embedded browser distributions, large deployment sizes, and tightly coupled frontend frameworks while still benefiting from modern rendering capabilities

2. Architectural Motivation

The architecture originates from a practical attempt to reduce the complexity associated with modern graphical application development while preserving portability, modularity, and lightweight deployment characteristics. Rather than introducing another standalone browser framework or frontend runtime, the system emerged as an extension of the previously established runtime-centric UI architecture in response to increasingly common requirements such as remote tooling, browser-accessible interfaces, mobile interaction surfaces, and distributed runtime visualization.

These requirements continue to grow in relevance while modern operating systems already ship with highly capable browser engines that provide many of the low-level capabilities traditionally required by graphical applications. At the same time, many existing UI solutions still rely either on heavyweight native frameworks or on embedded browser runtimes that substantially increase deployment complexity and operational overhead

2.1 Limitations of Existing Desktop UI Frameworks

Despite significant advances in graphical application development, many desktop UI frameworks remain tightly coupled to specific operating systems, rendering pipelines, runtime environments, or frontend execution models. Consequently, even comparatively small graphical applications often inherit considerable complexity through their underlying framework dependencies.

More recent cross-platform frameworks attempt to modernize desktop UI development through platform-independent rendering systems and declarative composition models. In practice, however, these frameworks frequently require extensive runtime infrastructure, large dependency graphs, complex build pipelines, and deeply integrated rendering abstractions. While technologically capable, the resulting operational overhead can become disproportionate for smaller tools, runtime utilities, dashboards, editors, or lightweight graphical applications.

Comparable patterns can also be observed across many browser-based UI frameworks. Modern frontend ecosystems often distribute application ownership across browser runtimes, frontend frameworks, hydration layers, backend APIs, asynchronous synchronization systems, and virtual rendering abstractions. This fragmentation commonly results in duplicated state management, increased lifecycle complexity, larger deployment artifacts, and reduced determinism.

Many of these limitations originate from ownership models that tightly couple rendering, state management, layout systems, lifecycle control, and interaction

handling into a single runtime abstraction. The previously proposed ECS-based architecture instead emphasized deterministic state synchronization, renderer abstraction, and reactive runtime-controlled composition. The browser projection layer introduced here builds directly upon those concepts rather than replacing them

2.2 Runtime Complexity and Deployment Overhead

The operational complexity associated with modern UI frameworks extends beyond framework architecture itself and directly affects deployment, maintenance, and distribution. Applications relying on embedded browser runtimes, large frontend frameworks, or platform-specific rendering systems often require substantial runtime packages to be distributed alongside comparatively small application binaries.

Electron-based systems illustrate this issue particularly well. Although they provide highly capable cross-platform environments, they also bundle complete browser runtimes together with application-specific logic, introducing significant storage overhead, memory consumption, update complexity, and deployment fragmentation. Even smaller embedded webview-based solutions commonly depend on platform-specific runtime installation requirements or tightly coupled rendering backends.

This becomes problematic in lightweight tooling scenarios where applications are expected to remain portable, self-contained, rapidly deployable, or directly executable from source-driven build environments.

The architectural goals established by the previously discussed data-driven runtime system emphasize modular composition, lightweight compilation pipelines, deterministic execution, and minimal external dependencies. Introducing large embedded browser runtimes would directly conflict with those goals. At the same time, modern applications increasingly benefit from browser-grade rendering capabilities. The system presented here attempts to reconcile those competing requirements by utilizing browser runtimes already available on the host system

2.3 Browser Runtimes as Existing System Infrastructure

Modern operating systems already provide highly capable browser runtimes that implement a large portion of the functionality commonly required by graphical applications. Browser engines such as Chromium and Gecko include GPU-accelerated rendering pipelines, advanced text layout systems, networking stacks, accessibility integration, session management, hardware input handling, and increasingly sophisticated security models. From a technical perspective, modern browsers already function as optimized cross-platform rendering environments. Conventional browser application architectures, however, typically treat the browser as the primary application runtime itself rather than as reusable rendering infrastructure.

The approach proposed here reinterprets the browser as reusable system infrastructure. Rather than embedding browser runtimes directly into applications, the system utilizes already installed browsers as externally attachable display and interaction nodes.

This distinction fundamentally changes the relationship between the application runtime

and the browser runtime. In conventional web applications, the browser frequently owns application lifecycle management, substantial portions of application state, rendering orchestration, routing logic, execution scheduling, and frontend interaction behavior.

Within the proposed model, these responsibilities remain entirely under host runtime ownership. The browser instead acts as a synchronized projection target capable of rendering runtime-controlled UI surfaces. Browser runtimes are therefore repurposed as lightweight projection environments capable of presenting synchronized runtime surfaces through standardized communication mechanisms.

One of the most important technologies enabling this model is the WebSocket protocol. Unlike traditional stateless HTTP request-response communication, WebSockets provide persistent bidirectional communication channels between browser clients and host runtimes. Runtime-controlled UI graphs, interaction events, synchronization messages, and display updates can therefore be exchanged continuously with relatively low overhead. Browser instances effectively become reactive display nodes rather than autonomous frontend runtimes

2.4 Separation of Runtime and Presentation

The separation between runtime ownership and presentation ownership represents one of the central concepts of the system.

Traditional browser applications frequently place substantial portions of execution behavior inside the browser itself. Frontend frameworks commonly manage application state, interaction routing, lifecycle scheduling, layout synchronization, asynchronous communication, and rendering orchestration directly within browser-controlled execution environments. Over time this tends to introduce fragmented ownership models, duplicated application state, increased synchronization complexity, frontend/backend divergence, reduced determinism, and complicated reconnect or session persistence behavior.
The projection system instead treats the browser exclusively as a presentation and interaction surface. Authoritative application state remains entirely controlled by the host runtime. Browser clients receive synchronized projections of runtime-controlled UI graphs and emit interaction events back into the runtime through reactive communication channels.

As a result, browser clients no longer behave as self-contained applications but instead operate as synchronized runtime surfaces attached to a centralized orchestration layer. This aligns closely with the renderer abstraction principles established within the earlier Data Driven UI Framework. Rendering backends remain interchangeable because the browser projection layer consumes synchronized runtime state rather than framework-owned visual objects or embedded frontend logic.

The browser therefore becomes only one possible rendering backend among multiple potential projection targets

2.5 Motivation for Distributed Display Surfaces

The growing demand for lightweight remote tooling and multi-device interaction environments further motivates the separation between runtime orchestration and display projection. Many modern workflows benefit from distributed

interaction surfaces extending beyond traditional desktop windowing models. Examples include:

- remote dashboards
- mobile control panels
- browser-accessible runtime tooling
- secondary monitoring surfaces
- collaborative runtime interfaces
- and tablet-based interaction systems

Within development and production environments, lightweight remote surfaces can provide substantial usability advantages without requiring full remote desktop systems or dedicated native applications. A tablet device operating as a browser-based control surface may provide large interaction elements, readable monitoring information, or runtime management interfaces while remaining synchronized with a centralized host runtime.

Because authoritative runtime ownership remains entirely outside the browser, display nodes may connect, disconnect, reconnect, or move between devices without fundamentally disrupting application execution. Runtime state persists independently of individual display surfaces. This naturally extends toward distributed runtime interaction scenarios in which multiple browser clients simultaneously observe or interact with synchronized runtime surfaces

2.6 Transition Toward a Display Server Architecture

As browser instances increasingly function as externally attachable runtime surfaces rather than self-contained applications, the overall system begins to resemble a modernized display server model.

Traditional display servers such as X11 historically focused on window composition, graphical output management, and remote display capabilities. The architecture discussed here extends those ideas toward runtime-centric orchestration of synchronized interaction surfaces.

The resulting display server abstraction is therefore not limited to window management alone. Instead, it acts as a runtime-controlled orchestration layer responsible for synchronized UI surface projection, browser client management, capability negotiation, interaction routing, session persistence, reactive stream synchronization, and distributed display coordination.

Browser engines such as Chromium and Gecko effectively become interchangeable projection runtimes attached to this orchestration layer rather than directly embedded application frameworks. This enables lightweight graphical applications, runtime dashboards, remote tooling environments, and browser-accessible interaction systems to operate without shipping dedicated browser runtimes or depending on heavyweight frontend stacks

3. Display Server Abstraction

The separation between runtime orchestration and graphical presentation forms the foundation of the proposed

display architecture. Rather than tightly coupling rendering behavior, frontend execution, synchronization logic, and application ownership into a single framework runtime, the system introduces a generalized display server abstraction focused exclusively on projection, synchronization, interaction routing, and display surface management.

The display server does not attempt to function as a complete UI framework, rendering engine, or application runtime. Instead, it operates as an independent orchestration layer capable of managing synchronized runtime surfaces independently of the systems responsible for generating them.

A particularly important consequence of this separation is that incoming runtime representations are translated into a generalized surface model understood by the display orchestration layer. The display infrastructure therefore remains intentionally decoupled from any specific framework implementation. This allows lightweight tools, dashboards, control surfaces, runtime utilities, and remote interaction systems to operate on top of the display server abstraction even while surrounding runtime systems continue to evolve independently

3.1 Generalized Display Server Model

Traditional display servers historically focused on low-level graphical composition, display synchronization, window management, and remote rendering. While the proposed system inherits several conceptual similarities from those architectures, its primary responsibility is not direct rendering ownership but runtime surface orchestration.

The display server therefore acts as an intermediary layer between runtime-generated surface descriptions and externally attachable projection environments such as browser-based display clients.

At a conceptual level, the architecture may be represented as:

Runtime Sources ->
Translation Layer ->
Display Server ->
Projection Runtime ->
Display Nodes

Within this model, the display server manages synchronized surfaces, runtime surface registration, session ownership, interaction routing, state synchronization, capability negotiation, projection lifecycle management, and distributed display coordination. Importantly, authoritative application state and execution remain outside the display server itself. The display layer only manages synchronized projections of externally owned runtime surfaces, allowing it to remain lightweight, modular, and independent from the application-specific runtime architecture generating those surfaces

3.2 Runtime Surface Abstraction

In order to remain independent from specific frontend frameworks or ECS ownership models, the display server operates on a generalized runtime surface abstraction.

A surface represents a synchronized projection target capable of exposing runtime-controlled visual state while receiving interaction events. Surfaces may

represent complete application windows, remote dashboards, embedded control panels, runtime overlays, mobile interaction layouts, or arbitrarily nested presentation regions. The display server itself does not require knowledge regarding the internal runtime structures responsible for generating these surfaces. Instead, surfaces are represented through generalized state descriptions containing hierarchical element structures, layout information, styling data, interaction bindings, synchronization metadata, and capability descriptors.

This abstraction intentionally avoids direct dependence on framework-owned widget hierarchies or renderer-specific object models. A surface therefore behaves as a synchronized state projection rather than as a self-contained application instance.

Because surfaces remain detached from specific rendering implementations, the same synchronized surface representation may be projected into browser runtimes, native rendering backends, remote display clients, mobile devices, embedded overlays, or alternative presentation environments

3.3 Translation Layer Architecture

The display server intentionally separates runtime-specific UI generation from generalized display orchestration through a dedicated translation layer. This layer acts as an adapter between arbitrary runtime architectures and the generalized surface model consumed by the display server.

Within ECS-driven environments, entities, components, layout systems, interaction systems, and rendering systems may collectively generate synchronized surface state which is then translated into the generalized surface representation understood by the display server.

The same infrastructure may also receive surface descriptions originating from standalone runtime utilities, manually constructed surface definitions, scripting environments, JSON-based surface descriptions, remote runtime systems, reactive stream pipelines, or custom application-specific generators. The display server itself remains completely unaware of the originating runtime model.

This separation preserves framework independence, modular extensibility, renderer abstraction, lightweight tooling support, and incremental runtime adoption. Most importantly, the translation layer functions as a compatibility boundary rather than a framework dependency, allowing the display infrastructure to remain usable even while surrounding runtime systems evolve independently

3.4 Runtime-Owned Surface Graphs

Internally, synchronized runtime surfaces are managed through retained surface graphs representing the hierarchical relationship between synchronized surfaces, visual elements, interaction regions, session ownership information, and synchronization metadata. Unlike traditional widget hierarchies, the surface graph does not contain framework-owned rendering logic or autonomous frontend behavior. Instead, it acts purely as a synchronized projection structure representing externally owned runtime states.

Each node within the graph maintains stable identity information allowing deterministic synchronization, incremental patch generation, interaction routing, and

reconnect-safe state restoration. This retained graph structure enables the system to synchronize only incremental state changes, preserve interaction continuity across reconnects, support distributed display projections, maintain deterministic synchronization order, and coordinate multiple projection runtimes simultaneously.

Because the graph itself remains renderer-independent, identical runtime surfaces may be projected into multiple browser clients or alternative rendering environments without duplicating runtime ownership

3.5 Projection Surfaces and Display Nodes

Projection runtimes act as externally attachable environments responsible for rendering synchronized runtime surfaces. Within the proposed model, browser instances function as display nodes connected to the display server through persistent synchronization channels. A display node simultaneously represents a projection runtime, an interaction endpoint, and a synchronized rendering surface.

Display nodes may operate locally or remotely and may dynamically attach or detach during runtime execution. Because authoritative state ownership remains outside the browser runtime, display nodes may reconnect or migrate between devices without fundamentally affecting runtime execution.

This differs significantly from conventional browser application models where browser instances frequently own substantial portions of application state and lifecycle management. Within the projection architecture, browser runtimes instead behave as lightweight reactive projection environments synchronized through externally controlled runtime states.

Multiple display nodes may simultaneously observe or interact with synchronized runtime surfaces depending on capability permissions and session ownership policies. This naturally enables remote dashboards, mobile control surfaces, secondary interaction displays, collaborative runtime visualization, and distributed tooling environments

3.6 Reactive Synchronization Model

Synchronization between runtime sources, the display server, and projection runtimes occurs through persistent reactive communication channels. Rather than repeatedly transferring complete frontend state snapshots, the synchronization layer distributes incremental runtime changes through continuous stream-oriented communication.

Persistent bidirectional communication channels such as WebSockets allow display nodes to receive synchronized surface updates, emit interaction events, negotiate capabilities, report display state changes, and maintain session continuity. The synchronization layer therefore behaves similarly to a reactive event stream connecting runtime-owned state with externally projected display surfaces.

Incremental synchronization substantially reduces bandwidth usage while improving responsiveness and reconnect behavior. Because display nodes do not own authoritative runtime state, synchronization remains deterministic and reconnect-safe. Browser clients may disconnect and later reconstruct synchronized surface state

directly from the runtime-owned surface graph

3.7 Input Routing and Interaction Flow

Interaction events originating from display nodes are routed back into the runtime environment through the display server orchestration layer. Display nodes therefore primarily function as rendering environments, interaction capture surfaces, and synchronization endpoints.

Input ownership remains externally controlled. Pointer events, keyboard interaction, touch gestures, focus changes, resize events, and capability state changes are transmitted back to the runtime through persistent synchronization channels where they are processed by runtime-owned interaction systems.

This preserves deterministic interaction handling while avoiding fragmented frontend/backend ownership models. The display server itself acts as the coordination boundary responsible for session-aware event routing, interaction ownership validation, capability-aware input handling, and synchronized interaction propagation. Because interaction behavior remains runtime-owned, browser runtimes remain lightweight and largely stateless beyond temporary presentation state

3.8 Browser Runtime Independence

The architecture intentionally avoids direct dependence on any single browser engine or browser-specific frontend framework. Browser runtimes are treated as interchangeable projection environments connected through standardized communication and synchronization mechanisms. This abstraction layer allows Chromium-, Gecko-, or future browser engines to operate within the same generalized synchronization model.

Particular emphasis is placed on avoiding assumptions that would tightly couple the system to engine-specific APIs or frontend execution behavior. Instead, the design prioritizes standards-based communication, browser capability negotiation, renderer-independent synchronization, and lightweight projection behavior. This preserves portability across desktop and mobile environments while maintaining long-term browser engine flexibility

4. Reactive Surface Synchronization

The display server architecture relies on continuous synchronization between runtime-owned surface graphs and externally attached projection environments. Rather than transmitting complete frontend state snapshots or executing autonomous application logic inside browser runtimes, the synchronization model preserves authoritative runtime ownership while allowing browser-based display nodes to maintain lightweight synchronized projections of runtime-controlled surfaces. The resulting behavior resembles a distributed composition system more closely than a conventional browser application architecture.

A central design goal of the synchronization layer is the minimization of frontend complexity. Projection runtimes should remain lightweight, deterministic, reconnect-safe, and largely independent from application-specific execution behavior. Synchronization is therefore

intentionally based around a small number of generalized structural operations operating on retained synchronized element graphs

4.1 Persistent Synchronization Channels

Synchronization between runtime systems, display servers, and browser-based projection runtimes occurs through persistent bidirectional communication channels. Unlike traditional stateless HTTP request-response communication models, persistent synchronization channels allow runtime state and interaction events to flow continuously between connected systems.

The architecture primarily relies on WebSockets as the transport mechanism for runtime synchronization. WebSockets provide low-overhead full-duplex communication channels capable of transmitting incremental surface updates, interaction events, capability negotiation messages, and session synchronization data without repeatedly establishing new network connections.

Within this synchronization model, browser runtimes do not behave as autonomous frontend applications but instead operate as synchronized projection environments attached to runtime-owned surface graphs. Persistent synchronization channels therefore support incremental structural synchronization, runtime interaction routing, capability negotiation, reconnect-safe session restoration, projection lifecycle coordination, and distributed surface synchronization.

Because browser runtimes do not own authoritative application state, synchronization remains deterministic and resilient against temporary connection interruptions

4.2 Retained Surface Graph Synchronization

Internally, synchronized UI structures are represented through retained hierarchical surface graphs. Each synchronized node within the graph represents either a generic UI element, a composable surface region, or a specialized projection structure. Surfaces themselves behave as synchronized elements capable of hosting nested child elements and additional projection semantics.

A display node always exposes a root surface representing the primary projection space of the connected runtime environment. Additional child surfaces may exist beneath the root surface in order to represent application windows, overlays, taskbars, popup menus, floating panels, modal regions, or embedded runtime projections.

This generalized hierarchy allows the synchronization system to represent both lightweight application layouts and more advanced workspace-oriented composition systems through the same structural model. Because synchronized surfaces behave as retained runtime-owned structures, browser runtimes may disconnect and later reconstruct synchronized state directly from the authoritative runtime graph

4.3 Handle-Based Element Identity

All synchronized structures within the projection system are identified through stable runtime-generated handles. Rather than relying on string-based identifiers or

framework-owned object references, the synchronization model uses lightweight numeric handles representing synchronized element instances, composable surfaces, interaction regions, and projection structures. This significantly reduces synchronization overhead while improving deterministic state reconstruction and runtime lookup performance.

The root surface of a display node is represented through a reserved null-equivalent handle comparable to root composition spaces used by traditional windowing systems. Child surfaces and synchronized elements maintain explicit parent-child relationships through retained handle references.

This structure enables reconnect-safe synchronization, incremental structural patching, deterministic element lookup, lightweight projection state reconstruction, and efficient runtime-side graph traversal. Because all synchronized structures share the same generalized handle model, normal UI elements and composable surfaces may be synchronized through identical patch operations

4.4 Incremental Structural Operations

Synchronization between runtime systems and projection runtimes is performed through batched structural update messages. Instead of transmitting individual object-based messages or standalone JSON structures, the protocol is designed around array-encoded command packets. This reduces serialization overhead, improves parsing efficiency, and aligns naturally with batch-oriented runtime updates.

Each synchronization message follows the structure:

["command", { args }]

Multiple operations may be transmitted sequentially within a single batch payload, allowing efficient aggregation of structural updates during a single transmission cycle. The synchronization layer intentionally minimizes protocol complexity by restricting interactions to a small set of generalized structural operations operating on retained element graphs

Append Operation

The append operation creates a new synchronized element instance from a registered template and attaches it to a parent element within the retained graph.

Parameters:

- **id** Registered template identifier
- **handle** Instance handle of the newly created element
- **parent** Optional parent handle. A value of 0 represents the root surface

Example:

["append", { "id": 10, "handle": 4096, "parent": 0 }]

Move Operation

The move operation reattaches an existing element to a new parent within the retained graph.

Parameters:

- **handle** Instance handle of the element being moved
- **parent** Optional new parent handle. A value of 0 represents the root surface

Example:

["move", { "handle": 4096, "parent": 2048 }]

Modify Operation

The modify operation updates properties of an existing element instance.

Parameters:

- **handle** Instance handle of the element being modified
- **properties** Key-value map of property updates

Example:

["modify", { "handle": 4096, "properties": { "text": "Connected", "visible": true } }]

Remove Operation

The remove operation deletes an existing element instance from the retained graph.

Parameters:

- **handle** Instance handle of the element being removed

Example:

["remove", { "handle": 4096 }]

Together, these operations form a minimal deterministic synchronization core capable of expressing hierarchical UI state changes while remaining independent from rendering backends or frontend frameworks

4.5 Template-Based Element Instantiation

Projection runtimes instantiate synchronized elements using pre-registered templates. HTML5 template elements provide a browser-native mechanism for storing pre-parsed DOM structures that can be cloned efficiently at runtime without repeated construction overhead.

Instead of transmitting raw HTML or imperative DOM instructions, append operations reference template identifiers resolved locally within the projection runtime. This reduces allocation overhead, minimizes layout recalculations, and enables efficient batch instantiation of synchronized UI structures

4.6 Deterministic Runtime Ownership

A strict separation exists between runtime-owned state and projection-owned state. Projection runtimes do not maintain authoritative application logic or persistent state. Instead, they render synchronized views derived from runtime-owned surface graphs.

This guarantees deterministic behavior across reconnects while eliminating frontend/backend state divergence

4.7 Session Persistence and Reconnection

Because synchronization is based on retained state graphs and incremental command streams, projection runtimes may reconnect and reconstruct UI state without requiring full application restart.

Upon reconnection, the runtime replays or rehydrates the current surface graph state using the same synchronization model

4.8 Capability Negotiation and Projection Coordination

Display nodes negotiate capabilities during session initialization. Capabilities influence rendering behavior, interaction handling, and feature activation. Examples include display constraints, input methods, and browser feature availability.

This ensures portability across different browser engines and device classes while maintaining consistent synchronization semantics

5. Browser Projection Runtime

The browser projection runtime acts as the client-side execution environment responsible for rendering synchronized runtime surfaces, forwarding interaction events, and maintaining lightweight local projection state.

Unlike conventional browser applications, the projection runtime does not behave as an autonomous frontend framework. Instead, it functions as a deterministic projection layer attached to a runtime-owned synchronization system.

A central architectural goal of the projection runtime is the strict separation between static runtime infrastructure and dynamic synchronized state. Templates, styling rules, event bindings, and local runtime functionality are generated during display server startup and delivered as a stable projection environment to connected browser clients. Runtime synchronization is therefore limited primarily to structural state updates and interaction events.

This separation substantially reduces synchronization complexity, minimizes frontend overhead, and allows the browser runtime to remain lightweight and deterministic

5.1 Static Projection Runtime Generation

During startup, the display server composes the complete browser projection runtime from registered runtime components.

These components may include:

- HTML template definitions
- CSS styling rules
- event anchor registrations
- reactive stream bindings
- projection runtime scripts
- synchronization infrastructure
- and optional runtime extensions

The resulting projection runtime is emitted as a complete static browser environment served directly by the display server. The startup sequence may therefore be represented conceptually as:

Display Server Startup ->
Component Registration ->
HTML Composition ->
CSS Composition ->
Projection Runtime Composition ->
Browser Runtime Generation ->
Browser Connection ->
Runtime Synchronization

Within this architecture, browser clients receive a stable projection runtime environment before synchronization begins. Templates, styles, event bindings, and projection runtime logic are therefore not synchronized dynamically through runtime update streams. Instead, they form the static execution environment of the projection runtime itself.

Changes to projection runtime structure typically require only a browser refresh after the display server has regenerated the runtime environment. This model intentionally prioritizes runtime simplicity and deterministic synchronization behavior over dynamic frontend reconfiguration

5.2 Projection Runtime Responsibilities

The browser projection runtime intentionally performs only a minimal set of responsibilities required for synchronized surface rendering and interaction forwarding.

These responsibilities primarily include:

- establishing synchronization channels
- maintaining synchronized handle-to-element lookup tables
- instantiating elements from registered templates
- processing structural synchronization operations
- forwarding interaction events
- exposing reactive event streams

The projection runtime does not own business logic, authoritative application state, orchestration behavior, or long-lived runtime session ownership. This strict separation prevents frontend/backend divergence while preserving deterministic synchronization behavior

5.3 Template Registry Architecture

Templates used by synchronized surfaces are registered during runtime generation and mapped to stable numeric identifiers. Projection runtimes may then instantiate synchronized structures directly from locally cached template definitions.

The use of HTML template cloning substantially reduces repeated DOM construction overhead while avoiding excessive layout invalidation.

Because synchronization payloads reference compact template identifiers instead of transmitting raw HTML structures, synchronization bandwidth remains comparatively small even for complex runtime surfaces

5.4 Structural Patch Processing

Synchronization batches received by the browser runtime are processed sequentially through a lightweight structural dispatcher. Each synchronization operation references synchronized element handles and invokes the corresponding append, move, modify, or remove behavior.

Because both elements and composable surfaces share the same generalized handle model, synchronization logic remains structurally uniform across the entire projection graph. This significantly simplifies synchronization processing while preserving deterministic state reconstruction.

The browser runtime therefore behaves less like a traditional application framework and more like a lightweight synchronized composition engine

5.5 Reactive Event Streams

Rather than relying on fragmented event listener patterns distributed throughout application-specific frontend code, the browser projection runtime translates interaction behavior into reactive event streams. Interaction events generated within the browser runtime are transformed into structured synchronization events containing originating handles, event identifiers, interaction parameters, and optional metadata.

A synchronized event payload may conceptually resemble:

```
{
  "handle": 4096,
  "event": 5,
  "parameters": {
    "x": 120,
    "y": 320
  }
}
```

Reactive event streams may represent pointer events, keyboard interaction, focus transitions, surface lifecycle events, resize notifications, touch gestures, or higher-level interaction semantics. This model integrates naturally with runtime ecosystems already utilizing Reactive Extensions or comparable stream-processing architectures.

Because interaction handling remains stream-oriented rather than callback-oriented, event processing pipelines remain composable, deterministic, and externally orchestrated

5.6 Local Projection State and Roundtrip Avoidance

Although authoritative state ownership remains entirely within the host runtime, some categories of interaction require immediate local responsiveness in order to maintain acceptable user experience.

Text input, cursor movement, selection updates, focus transitions, and pointer interactions may become perceptibly degraded if every state update depends entirely on remote synchronization roundtrips. The projection runtime may therefore maintain a limited amount of temporary local projection state for latency-sensitive interaction behavior.

Examples include optimistic text updates, local cursor positioning, temporary focus state, or immediate visual feedback during pointer interaction. These temporary local

changes may later be reconciled against authoritative runtime state through synchronized updates.

To avoid synchronization echo loops, the projection runtime may optionally attach lightweight synchronization metadata identifying locally originated changes. This preserves responsive interaction behavior without sacrificing centralized runtime ownership

5.7 Browser Runtime Composition

The browser projection runtime itself is distributed as part of the display server runtime package.

A minimal default runtime implementation may provide:

- template registration
- synchronization infrastructure
- event stream integration
- projection graph management
- browser integration behavior

Applications may either utilize this runtime directly or replace individual components during runtime composition.

Because runtime generation occurs during startup, composition remains highly modular without requiring heavyweight frontend framework infrastructure. The resulting browser runtime may therefore be embedded directly into the display server, compiled into static resources, or generated dynamically during startup

5.8 Runtime Layer Separation

The browser projection architecture is fundamentally divided into three distinct runtime layers

Static Projection Layer

The static projection layer contains:

- templates
- CSS rules
- event bindings
- reactive stream definitions
- projection runtime infrastructure
- and browser integration logic

This layer is generated during display server startup and remains stable throughout runtime execution

Dynamic Synchronization Layer

The synchronization layer transports:

- structural patch operations
- runtime state updates
- projection changes
- and synchronization metadata

This layer operates continuously throughout runtime execution

Reactive Event Layer

The reactive event layer translates browser interaction events into synchronized runtime streams. This layer bridges browser

interaction behavior with runtime-owned orchestration systems while preserving deterministic synchronization semantics.

Maintaining strict separation between these layers allows the browser runtime to remain lightweight, deterministic, modular, and largely independent from application-specific frontend frameworks

6. Browser Integration and Runtime Hosting

The system intentionally avoids custom embedded rendering engines and instead utilizes already installed system browsers as standardized rendering environments. This substantially reduces deployment complexity while leveraging mature browser engines that already provide hardware acceleration, input handling, accessibility integration, touch support, fullscreen behavior, security isolation, and modern web platform APIs.

Rather than replacing browser runtimes, the display server treats existing browser engines as externally hosted synchronized projection environments. The distinction is fundamental.

The display server does not attempt to behave as a browser engine, an embedded renderer, or an Electron-style application container. Instead, it coordinates synchronized projection sessions across independently hosted browser runtimes

6.1 Browser-Based Projection Environments

Projection runtimes may operate within conventional desktop browser windows, application-style browser windows, mobile browsers, fullscreen sessions, kiosk-style environments, remote projection displays, or headless browser environments.

Because synchronization behavior remains browser-engine independent, the same display server may expose synchronized projection environments simultaneously across Chromium-based browsers, Gecko-based browsers, desktop systems, tablets, and mobile devices.

Each browser instance effectively behaves as a synchronized projection node attached to the display server. This allows runtime surfaces to remain accessible from multiple independent devices without requiring platform-specific frontend implementations

6.2 Launching Browser App Windows

Modern browser engines provide several mechanisms allowing browser sessions to behave similarly to standalone application windows.

Chromium-based browsers expose native application-window support through the *--app* parameter:

---

chrome --app=https://localhost:8080

This launches the synchronized projection environment inside a dedicated application-style window without traditional browser navigation controls. Additional launch parameters may further reduce browser UI visibility:

---

chrome \

```
  --app=https://localhost:8080 \
  --disable-session-crashed-bubble \
  --disable-infobars \
  --disable-features=TranslateUI
```

Firefox and other Gecko-based browsers currently expose fewer dedicated application-window features compared to Chromium-based runtimes. Rather than depending primarily on fullscreen kiosk hosting, Gecko runtime integration is more reliably implemented through isolated runtime profiles combined with browser chrome reduction.

Projection runtimes may therefore generate temporary runtime-specific Firefox profiles during display server startup. Such profiles may contain runtime-specific browser preferences, projection-oriented UI overrides, wake lock configuration, fullscreen permissions, synchronization policies, and optional extension configuration.

Firefox projection sessions may then be launched through:

```
firefox \
  --no-remote \
  --profile ./runtime-profile \
  --new-window https://localhost:8080
```

The --no-remote parameter is particularly important because Firefox otherwise tends to attach new windows to already running browser sessions.

Optional browser chrome reduction may additionally be implemented through generated *userChrome.css* configuration files inside the runtime profile.

Example:

```
#TabsToolbar
{
   visibility: collapse !important;
}

#nav-bar
{
   visibility: collapse !important;
}

#PersonalToolbar
{
   visibility: collapse !important;
}

#sidebar-box
{
   visibility: collapse !important;
}
```

Custom browser stylesheet support must additionally be enabled through:

```
user_pref(

"toolkit.legacyUserProfileCustomizations.stylesheets",
   true
);
```

---

inside the generated *prefs.js* runtime profile configuration.

This approach allows Gecko-based runtimes to behave similarly to lightweight

synchronized application windows while remaining fully standards-compliant browser sessions. Unlike Chromium application windows, Firefox still internally behaves as a complete browser environment. However, the visible runtime surface may be reduced substantially.

Popup-based runtime hosting may additionally be implemented through browser-side runtime initialization logic.

Example:

```
window.open(
    "https://localhost:8080",
    "projection",
    "popup,width=1280,height=720"
);
```

Because browser capabilities differ significantly between engines, the display server architecture intentionally treats browser windows as loosely standardized projection environments rather than fully identical runtime hosts

6.3 Chromium Runtime Integration

Chromium-based browsers currently provide the most complete runtime hosting environment for browser-based synchronized projection systems. Available functionality includes application-style windows, fullscreen projection support, stable wake lock behavior, Progressive Web Application integration, advanced compositor acceleration, robust WebSocket support, and mature mobile compatibility.

Chromium browsers additionally expose comparatively predictable window-management behavior across desktop operating systems. This allows synchronized projection environments to behave similarly to lightweight native desktop applications while still remaining fully browser-hosted.

The synchronization architecture itself, however, does not depend on Chromium-specific rendering behavior. Chromium engines simply provide one of the most complete currently available environments for browser-hosted synchronized runtimes

6.4 Gecko Runtime Integration

Gecko-based browsers provide several architectural advantages including strong privacy tooling, extension flexibility, and broad platform availability. However, runtime hosting behavior differs substantially from Chromium-based browsers.

Firefox currently exposes fewer dedicated application-window-oriented runtime hosting features and places greater emphasis on conventional browser interaction models. This affects standalone application window behavior, popup persistence, fullscreen integration, browser chrome visibility, extension integration, and runtime-oriented window management.

Despite these differences, Gecko-based browsers remain fully capable synchronized projection environments. Projection runtimes hosted within Firefox continue to support WebSocket synchronization, fullscreen projection, reactive event streams, touch interaction, mobile projection workflows, wake lock support, and hardware-accelerated rendering.

The synchronization model therefore avoids dependence on browser-engine-specific assumptions. Only standards-compliant rendering behavior, modern JavaScript

execution, DOM template support, WebSocket communication, and modern browser API availability are fundamentally required. This preserves portability across both Chromium- and Gecko-based environments.

Firefox projection hosting is therefore implemented primarily through isolated runtime-generated browser profiles combined with projection-oriented browser chrome reduction.

This strategy avoids dependence on experimental browser features, undocumented startup modes, or fullscreen-only kiosk environments. Projection runtimes may therefore remain compatible with conventional desktop multitasking workflows while still behaving similarly to lightweight synchronized application windows

6.5 Mobile Browser Integration

Because the projection architecture relies entirely on standard browser runtimes, synchronized surfaces may be accessed directly from mobile devices without requiring separate native application implementations.

This is particularly useful for remote tooling environments, secondary control surfaces, tablet-based dashboards, streamer control clients, portable runtime management panels, and auxiliary interaction devices.

The display server may therefore expose synchronized projection environments directly over local networks through HTTPS-secured browser sessions.

Mobile browsers effectively behave as additional synchronized display nodes connected to the retained runtime graph. This enables lightweight multi-device projection workflows without introducing separate mobile frontend stacks

6.6 Wake Lock and Long-Lived Sessions

Long-lived synchronized projection environments may require prevention of automatic display suspension or device sleep behavior. Modern browser environments expose the Screen Wake Lock API, allowing browser runtimes to request temporary prevention of automatic display sleep.

Example:

```
let wakeLock = null;

async function acquireWakeLock()
{
   try
   {
      wakeLock = await
navigator.wakeLock.request("screen");
   }
   catch (error)
   {
      console.error(error);
   }
}
```

Wake lock functionality is particularly important for dashboard displays, remote control panels, mobile projection devices, fullscreen control surfaces, and long-running synchronized sessions.

Projection runtimes may negotiate wake lock availability during capability negotiation. Because browser support differs between engines and platforms,

wake lock integration remains capability-driven rather than mandatory

6.7 Local Network Projection Access

Display servers may expose synchronized projection environments directly through local network HTTPS endpoints. Desktop systems, tablets, laptops, and mobile devices may therefore connect directly to synchronized runtime surfaces without requiring cloud infrastructure or externally hosted frontend deployments.

Authentication may be implemented through session tokens, application passwords, runtime-generated access keys, or local network trust policies.

Because browser runtimes themselves remain lightweight and largely stateless, remote devices effectively behave as temporary synchronized projection clients. This is especially useful for local development environments, remote tooling systems, portable dashboards, streamer control interfaces, and multi-device runtime management

6.8 Security and Runtime Isolation

The projection model intentionally leverages existing browser security infrastructure instead of bypassing it through embedded rendering runtimes. Modern browser engines already provide mature process isolation, sandboxing, permission systems, origin restrictions, HTTPS enforcement, and resource isolation.

The display server therefore benefits directly from mature browser security models while avoiding many of the security concerns commonly associated with custom embedded web runtimes.

Because browser runtimes remain externally hosted, strong separation is preserved between runtime orchestration, projection rendering, and client interaction environments. This separation aligns naturally with the modular runtime architecture used throughout the broader runtime ecosystem

6.9 Headless and Embedded Runtime Modes

Although the primary architecture targets externally hosted browser runtimes, the synchronization system may also operate within headless browser environments.

Headless execution is particularly useful for automated testing, runtime validation, surface snapshot generation, projection verification, and server-side rendering workflows. Chromium-based browsers currently provide mature headless runtime support while Firefox additionally exposes headless execution capabilities.

Runtime behavior may still differ between browser engines, particularly regarding fullscreen support, extension behavior, compositor acceleration, and runtime lifecycle management. Headless execution is therefore treated primarily as an auxiliary runtime environment rather than the primary deployment target

7. Roadmap and Extensibility

The projection architecture described throughout this work intentionally focuses on establishing a lightweight, modular, and browser-independent synchronized runtime foundation.
Rather than attempting to provide a complete monolithic frontend framework,

the system defines a generalized projection model capable of supporting a wide range of higher-level runtime environments.

Because runtime ownership, synchronization, projection composition, browser integration, and reactive interaction flows remain structurally separated, additional runtime systems may be layered on top of the synchronization model without fundamentally changing the underlying architecture.

Potential future runtime extensions include popup and overlay systems, docking environments, graph-oriented runtime projections, visual scripting systems, workspace-oriented tooling surfaces, multi-device runtime dashboards, remote control interfaces, collaborative projection environments, and browser-hosted desktop-style runtime systems.

The generalized handle model, retained projection graph, and reactive synchronization layer provide a stable foundation upon which these higher-level runtime environments may be implemented incrementally.

Because projection runtimes remain browser-hosted and synchronization-oriented, advanced runtime systems may additionally operate across desktop systems, mobile devices, tablets, remote development environments, and distributed runtime sessions without requiring separate frontend implementations.

This extensibility emerges directly from the intentionally modular runtime architecture underlying the system

8. Summary and Vision

Modern application development continues to suffer from fragmentation between native UI frameworks, browser runtimes, platform-specific rendering systems, and increasingly heavyweight frontend ecosystems. The system presented throughout this work approaches the problem from a fundamentally different direction.

Instead of treating the browser as the application itself, the browser becomes a universally available synchronized projection runtime attached to a runtime-owned display server. This distinction significantly changes the role of the browser within the overall system.

The browser runtime no longer owns application state, orchestration logic, or runtime authority. Instead, it behaves as a lightweight synchronized projection environment responsible for rendering retained runtime surfaces and forwarding interaction streams.

By leveraging standardized browser technologies, lightweight synchronization protocols, retained projection graphs, reactive runtime streams, and modular runtime composition, browser-hosted runtime environments become possible without introducing embedded browser runtimes or heavyweight frontend abstraction layers.

Synchronized runtime systems may therefore remain lightweight, modular, browser-independent, remotely accessible, multi-device capable, and incrementally extensible.

The long-term objective is not the creation of yet another browser framework. Instead, the goal is the establishment of a generalized synchronized runtime projection system capable of supporting tooling environments, runtime dashboards, editor systems, visual scripting

environments, remote interaction surfaces, and future browser-hosted runtime ecosystems through a unified synchronization architecture.

As browser capabilities continue evolving across desktop and mobile environments, the browser increasingly becomes not merely a document renderer but a universally available runtime projection environment.

The architecture presented throughout this work attempts to utilize that capability directly while preserving runtime ownership, deterministic synchronization, modularity, and deployment simplicity