A Minimalist Design Ontology for Document-Centric Web Systems: HTML as a Primary Artifact

Alexey A. Nekludoff

AstraVerge Research

E-mail: an@astraverge.org
ORCID: 0009-0002-7724-5762

5 February 2026

Abstract

HTML was originally conceived as a self-contained document format. Over time, however, it has increasingly been treated as a transient output of complex and layered toolchains. This paper revisits a document-first design philosophy and articulates a minimalist design ontology in which HTML is preserved as a primary artifact. Rather than generating documents from scratch, dynamic behavior is expressed through carefully constrained declarative adaptations of existing structure, allowing HTML to retain its role as a stable carrier of structured meaning rather than a disposable byproduct of execution.

Contents

1 Introduction

Over the past decades, the role of HTML in web systems has undergone a profound transformation. Originally conceived as a self-contained document format, HTML is today commonly treated as a transient output produced by increasingly complex software pipelines. In many contemporary architectures, HTML no longer exists as a stable artifact; it is generated, streamed, diffed, and discarded as part of a larger execution process.

This transformation was not accidental. As web applications grew in complexity, purely static documents proved insufficient. Conditional structures, repeated elements, data binding, and request-driven behavior became necessary even for relatively simple systems. A wide range of dynamic technologies emerged in response, from early CGI-based approaches to server-side includes, template engines, and eventually full client-side rendering frameworks. Each step addressed real engineering needs and enabled new classes of applications.

However, this evolution also introduced a subtle conceptual shift. HTML gradually ceased to be treated as a document in its own right and instead became an intermediate representation or final byproduct of program execution. The primary artifact of development moved elsewhere: to template languages, component systems, build configurations, and runtime frameworks. As a result, the HTML document itself often cannot be meaningfully inspected, validated, or even understood outside of the full execution environment that produces it.

This paper does not treat this shift as either a mistake or the inevitable form of true dynamism. Modern frameworks solve problems that early document-centric approaches could not, particularly in the domains of rich client-side interaction and large-scale application state. Rather, the aim here is to observe that the dominance of generation-oriented models has obscured an alternative design space: one in which HTML remains a primary, durable artifact, and dynamic behavior is expressed as a constrained adaptation of an existing document, rather than as a program that emits HTML as output.

We revisit a document-first perspective in which an HTML file is considered complete and meaningful even prior to execution. In this view, execution is not a process of generating markup from scratch, but a process of bringing an existing document into correspondence with available data and execution context. Dynamic behavior is limited to a small set of structural transformations: the conditional presence of elements, controlled repetition, data binding, and static inclusion of document fragments.

The motivation for this approach is an explicit methodological step toward identifying the primary source of meaning in document-centric systems. This inquiry is guided by a minimalist design philosophy closely aligned with the Unix tradition: each component should perform a single, well-defined function, and unnecessary ontological layers should be avoided. Applied to web documents, this principle suggests avoiding the introduction of secondary languages or parallel representations where a constrained extension of HTML itself is sufficient.

The contribution of this paper is therefore definitional rather than prescriptive. We articulate an ontological and design rationale for treating HTML as a primary artifact and for expressing limited forms of dynamism declaratively within the document itself. We outline the implications of this perspective for execution models, tooling boundaries, and system longevity, and clarify the scope in which such an approach is both appropriate and advantageous.

The remainder of the paper proceeds as follows. Section 2 revisits the original assumptions underlying HTML as a document format. Section 3 briefly surveys the emergence of early dynamic extensions that operated directly within HTML. Section 4 describes the later shift toward generation-oriented models. Section 5 introduces a minimalist ontology of document-centric web systems. Section 6 discusses declarative structural constraints as an alternative to template generation. Section 7 examines the role of an executor designed in accordance with the Unix philosophy. Finally, Section 8 outlines the limitations of this approach and situates it within the broader landscape of web architectures.

2 HTML as a Document: Original Assumptions

HTML was never intended to be a language of meaning in itself. It does not define semantics in the sense of propositional content, logical relations, or conceptual truth. Rather, HTML was designed as a language for *marking up meaning*: for structuring, organizing, and annotating content whose semantic substance originates elsewhere.

In this sense, HTML is fundamentally derivative. The meaning of an HTML document is not created by its tags, but by the textual, symbolic, or referential content to which those tags are applied. Markup does not generate meaning; it frames it. It provides cues about structure, emphasis, hierarchy, and relation, allowing meaning to be navigated, interpreted, and processed.

Yet this derivative character should not be confused with ontological insignificance. The historical success of HTML lies precisely in the fact that its form of markup proved to be extraordinarily transferable. HTML documents could be rendered across platforms, interpreted by different user agents, indexed, archived, transformed, and preserved without requiring access to the original context of their creation.

Over time, this portability had an important consequence. The marked-up document itself became a stable carrier of meaning, even though the meaning it carries is not intrinsic to the markup language. Structure, headings, lists, tables, and links do more than merely decorate content: they select, foreground, background, and relate its parts. Through this process, markup does not merely transmit meaning, but actively shapes its presentation and interpretation.

As a result, HTML documents came to function as autonomous artifacts. They could be read, reasoned about, and reused independently of the systems that produced them. The document became a unit of exchange, a referenceable object, and a durable representation of structured knowledge. In this practical sense, HTML acquired an ontological status that exceeds its purely derivative semantics.

This status was reinforced by early design assumptions of the web. Documents were expected to be directly accessible, human-readable, and meaningful even in degraded environments. A browser failure, a missing stylesheet, or an unavailable script did not negate the existence of the document itself. Instead, the document degraded gracefully, retaining its structural core.

These assumptions positioned HTML not as an executable specification, but as a declarative artifact whose primary role was to stabilize meaning across time, tools, and contexts. Computation, where present, served to adapt or supplement the document, not to replace it. The document preceded execution both conceptually and practically.

This distinction is central to the argument of this paper. Although HTML is not a language of meaning, its role as a language for the organization and modulation of meaning has proven sufficient to support a wide range of document-centric systems. It is precisely this sufficiency that motivates a reconsideration of design approaches that treat HTML merely as an ephemeral output, rather than as a primary artifact in its own right.

3 Early Dynamic Extensions and In-Place Adaptation

As soon as HTML documents began to be used for more than static publication, the need for limited forms of dynamism became apparent. Even relatively simple systems required conditional content, repeated structures, and the inclusion of shared fragments. These requirements did not initially call for a redefinition of HTML's role; rather, they motivated mechanisms for adapting existing documents to varying contexts.

Early dynamic technologies approached this problem through *in-place extension* of HTML. Server-Side Includes (SSI) provided one of the simplest examples: documents remained largely unchanged, while a small set of directives allowed for file inclusion, environment-dependent substitution, and basic conditional logic. Importantly, the surrounding document structure remained intact, and the HTML file itself continued to be a readable and meaningful artifact.

Subsequent systems such as JSP and ASP followed a similar pattern, albeit with increased expressive power. Executable fragments were embedded directly within HTML documents, allowing request parameters, session state, and external data to influence the final output. Despite their differences in syntax and runtime behavior, these systems shared a common assumption: HTML remained the structural backbone, and dynamic behavior operated *within* the document rather than replacing it.

In these early approaches, the distinction between document and execution was still perceptible. The source files were recognizably HTML, often renderable in a degraded but intelligible form even without server-side processing. Dynamic constructs were localized and explicit, and their presence did not obscure the overall structure of the document.

Crucially, these systems treated dynamism as an auxiliary capability. They did not assume that the document itself was transient or disposable. Instead, execution was understood as a means of specializing a stable document for a particular request or context. The HTML file continued to function as a unit of understanding, maintenance, and reuse.

At the same time, these approaches revealed inherent tensions. As dynamic requirements grew, the embedded logic became more complex, and the boundaries between structure and computation began to blur. While HTML remained central, the increasing density of executable fragments made documents harder to reason about and more tightly coupled to their execution environments.

These limitations were not merely technical. They reflected the strain placed on a document-centric model when asked to accommodate ever-expanding forms of application logic. The response to this strain would eventually take the form of a more radical reorganization of web architectures, in which HTML would no longer be treated as a stable artifact, but as a generated output. It is this structural shift that the next section examines.

4 The Structural Shift: HTML as Generated Output

As dynamic requirements continued to expand, a more fundamental reorganization of web architectures gradually took place. Although many of the underlying technologies had existed earlier, it was during this period that a decisive conceptual shift became dominant: HTML ceased to be treated as a stable document and was redefined as a generated output of program execution.

In this emerging paradigm, the primary locus of meaning was relocated away from the document. Meaning was increasingly identified with unstructured or minimally structured data, typically stored in databases, while HTML was reduced to a presentational layer responsible for rendering that data for display. From this perspective, HTML was understood as "mere markup": a formatting mechanism applied to content whose semantic substance was assumed to exist independently of any particular document form.

This shift was not arbitrary. In many respects, it represented a return to a classical separation of concerns. Data models, business logic, and presentation were clearly distinguished, and the meaning of a system was grounded in its underlying data structures rather than in its rendered representations. For complex applications, this reorientation enabled greater flexibility, reuse, and abstraction.

At the same time, the consequences of this shift were not purely technical. By treating HTML as an ephemeral byproduct, the accumulated ontological role of markup was effectively disregarded. HTML had already proven to be more than a neutral formatting language: through headings, lists, tables, links, and structural grouping, it had become a means of selecting, emphasizing, suppressing, and relating elements of meaning. In practice, documents were not merely containers for content, but active instruments in shaping how that content was understood.

The generation-oriented paradigm largely ignored this development. By assuming that meaning resided exclusively in raw data and that documents were disposable renderings, it severed the connection between structure and interpretation that HTML had gradually come to embody. As a result, the document itself lost

its status as an object of reasoning, maintenance, and preservation, becoming instead a transient artifact of execution.

This redefinition of HTML gave rise to entire classes of languages and frameworks explicitly oriented toward markup generation. In these systems, HTML is constructed from scratch, often through intermediate representations or component abstractions, and exists only momentarily as the final output of a rendering pipeline. The source of the system is no longer an HTML document, but a program whose purpose is to produce HTML.

While this approach proved powerful, it also introduced a conceptual discontinuity. The expressive capacity of markup to organize and modulate meaning was subordinated to the logic of generation. Documents could no longer be inspected independently of their generators, and the stability previously afforded by document-centric artifacts was replaced by dependence on increasingly elaborate toolchains.

It is this discontinuity that motivates a re-examination of document-centric design assumptions. The question is not whether generation-oriented models are valid—they clearly are— but whether their underlying ontology should be treated as exclusive. The following sections explore an alternative perspective, in which HTML is neither reduced to mere formatting nor elevated to a language of meaning, but acknowledged as a structurally expressive artifact whose role warrants careful preservation.

5 A Minimalist Ontology of Document-Centric Web Systems

The preceding sections describe a historical movement away from documents as stable artifacts toward generation-oriented models of web architecture. This section introduces an alternative ontological framework for document-centric web systems, grounded in a minimalist conception of structure, execution, and meaning.

At the core of this ontology lies a simple distinction: between artifacts that *carry* structured meaning and processes that *adapt* such artifacts to specific contexts. HTML documents belong to the former category. They are not executable specifications, nor repositories of semantic truth, but stable carriers of structured representation. Their role is not to compute meaning, but to preserve and expose it in a form that remains interpretable across tools and time.

Within this framework, HTML is treated as a primary artifact not because it originates meaning, but because it stabilizes meaning through structure. The document exists independently of any particular execution, and its identity is not exhausted by the process that renders or transforms it. Execution may alter a document's realization, but it does not constitute the document itself.

Minimalism, in this context, is not an aesthetic preference but an ontological constraint. Each layer introduced into a system must justify its existence by providing a function that cannot be realized elsewhere without loss of clarity or stability. From this perspective, the proliferation of parallel languages, intermediate representations, and opaque generation stages represents not merely technical complexity, but an expansion of ontological commitments.

A minimalist document-centric ontology therefore rejects the introduction of secondary languages whose sole purpose is to regenerate HTML from scratch. Instead, it favors constrained extensions that operate directly on existing documents, preserving their structure while allowing limited adaptation. Such extensions do not replace the document; they qualify it.

Within this ontology, dynamism is reconceptualized. Dynamic behavior is not understood as the generation of documents, but as the conditional realization of structural possibilities already present in the artifact. Repetition, omission, and binding are treated as modes of structural variation, not as instances of programmatic construction.

This distinction has practical consequences. When a document is primary, it remains inspectable, diffable, archivable, and meaningful even in the absence of execution. Failure modes degrade the realization of

the document, not its existence. The system retains a core artifact that can be reasoned about independently of its runtime environment.

The ontology proposed here does not deny the utility of generation-oriented approaches. Rather, it situates them within a broader design space and argues that their assumptions should not be universalized. For a large class of document-centric systems, a minimalist ontology centered on primary artifacts offers greater conceptual economy, long-term stability, and transparency of structure.

The next section examines how this ontology can be operationalized through declarative structural constraints, providing limited forms of dynamism without abandoning the primacy of the document.

6 Declarative Structural Constraints

If HTML is treated as a primary artifact, the question of dynamism must be reformulated. Rather than asking how documents can be generated, one must ask how an existing document can be selectively realized under varying conditions. This section outlines a class of mechanisms that enable such adaptation while preserving the primacy and integrity of the document.

The central idea is to express dynamic behavior not through imperative code, but through declarative constraints placed upon the structure of the document itself. These constraints do not describe how to construct HTML; they describe under which conditions parts of an existing structure become present, repeated, or bound to data. Execution, in this context, is an act of interpretation, not of generation.

Three forms of structural constraint are sufficient for a wide range of document-centric systems. The first concerns *presence*. Certain elements may be conditionally included or omitted depending on available data or execution context. Importantly, the structure of the document remains visible, even when particular realizations suppress parts of it.

The second form concerns *repetition*. Document fragments may represent collections by indicating that a given structural unit can be instantiated multiple times. Here, repetition is not an instruction to construct markup, but a declaration that a particular structural pattern admits multiple realizations. The document encodes the shape of repetition independently of the number of instances.

The third form concerns *binding*. Specific positions within the document may be associated with external data values. Binding does not introduce computation into the document; it associates already structured placeholders with data supplied by the execution environment. The document determines where meaning appears, while execution determines which values are supplied.

A fourth, auxiliary mechanism concerns *static inclusion*. Larger documents may be composed from smaller fragments through explicit inclusion points. Such inclusion operates at the level of document structure, not at the level of control flow, and serves primarily as a means of reuse and organization.

However, unlike presence, repetition, or binding, static inclusion introduces a qualified dependency on the execution environment. When interpretation is unavailable, the included fragments may not be materialized, and parts of the intended document may remain absent. This trade-off is accepted deliberately: inclusion is treated as an optional convenience, rather than a fundamental requirement for the document's intelligibility.

For this reason, static inclusion occupies a secondary position within the proposed ontology. Its use presupposes a willingness to sacrifice a portion of structural completeness in exchange for modularity and reuse, while preserving the document's overall coherence in degraded or non-executed contexts.

Together, these mechanisms form a minimal set. They enable conditional variation, multiplicity, and data association without introducing a secondary language or a parallel representation of structure. Crucially, they operate on the assumption that the document already exists and already expresses its intended organization.

Because these constraints are declarative, their presence does not obscure the document. An HTML file remains readable and meaningful even when the constraints are ignored or unsupported. The document degrades gracefully: structural cues remain visible, fallback content persists, and the artifact retains its identity

independently of execution.

This approach contrasts sharply with generation-oriented models, in which the document exists only as a product of execution. By limiting dynamism to structural constraints, the document-centric model preserves a clear boundary between artifact and process. Execution realizes one of many possible interpretations of a single, stable document.

The use of declarative structural constraints thus operationalizes the minimalist ontology introduced in the previous section. It provides a means of accommodating necessary dynamism while preserving the document as a primary artifact, even under partial or absent execution. The following section examines the implications of this approach for the design of executors and the boundaries of their responsibility.

7 The Executor as a Unix-Style Tool

Within a document-centric ontology, the role of execution must be sharply delimited. If the document is treated as a primary artifact, the executor cannot be understood as a system that generates or owns the document. Instead, it functions as an interpreter that realizes a particular structural interpretation of an already existing artifact.

This conception aligns closely with the Unix design tradition. In that tradition, tools are not defined by the complexity of the problems they address, but by the clarity of the single function they perform. A tool reads an input, applies a well-defined transformation, and produces an output, without asserting ownership over the broader system context.

Applied to web documents, the executor performs exactly one task: it brings a document into correspondence with available data and execution conditions, according to explicitly declared structural constraints. It does not define application architecture, manage long-lived state, or prescribe interaction patterns. Its responsibility begins and ends with the interpretation of the document.

This strict limitation of scope has important consequences. Because the executor does not generate documents from scratch, it does not replace HTML with an alternative source language. Because it does not embed a general-purpose programming model, it avoids becoming a parallel ontology alongside the document itself. The executor remains subordinate to the artifact it interprets.

From this perspective, execution is not an act of creation, but an act of selection. Among the structural possibilities encoded in the document, the executor realizes those that are compatible with the current context. The document specifies what may appear; execution determines what does appear.

This division of responsibility supports a clear separation between stability and variability. The document embodies stable structure and intent. Execution supplies contingent data and conditions. When execution fails or is unavailable, the document persists as a meaningful artifact, even if only partially realized.

By constraining the executor to a single, well-defined role, the system resists the accumulation of hidden responsibilities. There is no implicit framework logic, no secondary control flow, and no requirement that understanding the document depends on understanding the executor's internal machinery.

In this sense, the executor is not a framework, but a tool. Its purpose is not to abstract the document away, but to make its declared structural possibilities explicit. Such an executor exemplifies a minimalist approach to system design, in which complexity is not eliminated, but localized and bounded.

8 Scope, Limitations, and Non-Goals

The ontology and design principles articulated in this paper are intentionally limited in scope. They do not aspire to provide a universal foundation for all forms of web application development, nor do they seek to replace generation-oriented frameworks where those frameworks are well justified. Instead, the approach is explicitly situated within the domain of document-centric web systems.

The primary scope of applicability includes systems in which documents retain a stable structural identity across requests and contexts. Typical examples include form-based interfaces, tabular views, reports, administrative panels, and other applications where the core interaction consists in presenting, collecting, and organizing information. In such systems, the document itself constitutes a significant part of the system's enduring structure.

Conversely, the proposed ontology is not designed for applications whose primary complexity lies in rich client-side interaction, highly dynamic state synchronization, or continuous user-driven reconfiguration of interface structure. In these domains, the document is often transient by necessity, and generation-oriented or component-based models provide capabilities that exceed the expressive intent of a document-centric approach.

A further limitation concerns expressive power. By restricting dynamism to declarative structural constraints, the approach deliberately excludes general-purpose computation within the document. This exclusion is not a technical deficiency, but a design choice. It preserves the intelligibility and inspectability of the document at the cost of limiting the range of behaviors that can be expressed in situ.

Similarly, the executor described in this framework is not intended to function as an application framework. It does not provide abstractions for routing, state management, authorization models, or interaction lifecycles. Such concerns are assumed to reside outside the document and outside the executor's responsibility. The executor's role is confined to interpretation, not orchestration.

It is also a non-goal of this approach to optimize for maximal performance or minimal latency in all deployment scenarios. While document-centric execution can be efficient, its primary motivation lies in conceptual economy, system transparency, and long-term maintainability rather than in raw throughput.

Finally, this work does not claim historical exclusivity or normative superiority. The ontology proposed here coexists with other architectural paradigms, each grounded in different assumptions about the nature of documents, execution, and interaction. The intent is not to prescribe a single correct model, but to clarify an alternative design space that remains viable and underexplored.

Within its intended scope, a minimalist document-centric ontology offers a stable foundation for systems that prioritize durability, inspectability, and clarity of structure. Outside that scope, other models may be more appropriate, and no conflict is implied.

9 Conclusion

This paper has argued for a document-centric perspective in which HTML is treated as a primary artifact rather than as a transient output of program execution. Without denying the derivative nature of markup with respect to meaning, we have shown how HTML has acquired ontological stability through its capacity to structure, modulate, and preserve meaning across tools, contexts, and time.

By revisiting early document-oriented assumptions and examining the subsequent shift toward generation-based models, we have identified a largely overlooked design space. Within this space, dynamism is expressed not through the construction of documents from scratch, but through declarative constraints that govern the realization of an existing structure. This approach preserves the intelligibility and durability of documents while accommodating necessary variation.

The minimalist ontology proposed here does not compete with contemporary frameworks on the basis of expressive power or generality. Instead, it offers a principled alternative grounded in ontological economy, clear separation of responsibilities, and alignment with long-standing design traditions that favor simple tools over comprehensive systems.

For document-centric web systems, treating HTML as a primary artifact provides a stable foundation for transparency, maintainability, and graceful degradation. More broadly, it illustrates how restraint in abstraction can serve not as a limitation, but as a deliberate and productive design choice.

A Conceptual Reference Design

This appendix presents a minimal conceptual reference design that illustrates how the document-centric ontology developed in the main body of the paper may be operationalized in practice. The intent is not to specify an implementation, but to clarify roles, responsibilities, and boundaries at the level of design concepts.

A.1 Core Artifacts

The reference design is organized around three primary artifacts: the document, the data context, and the executor.

Document. The document is a stable HTML artifact. It exists independently of execution and remains meaningful when interpreted as plain markup. The document encodes structural intent: hierarchy, grouping, emphasis, repetition patterns, and potential variation points. It does not contain general-purpose computation and does not prescribe control flow.

Data Context. The data context is external to the document. It provides contingent values that may be associated with positions explicitly identified by the document's structure. The data context does not define the document and does not determine its organization; it merely supplies values for an already articulated structural form.

Executor. The executor is an interpretive mechanism. It does not generate documents and does not replace the document as a source artifact. Its sole responsibility is to interpret declared structural constraints and to realize one admissible instantiation of the document given a particular data context and execution environment.

A.2 Structural Constraints

Structural constraints define the admissible variations of a document without introducing a secondary language. They operate directly on the existing structure and are limited to a small, closed set of categories.

Presence. A document may declare that certain elements are conditionally present. Presence constraints determine whether a structural unit is realized or omitted in a given execution, without altering the surrounding structure.

Repetition. A document may declare that a structural fragment admits multiple instantiations. Repetition expresses multiplicity without prescribing how markup is constructed. The document defines the pattern; execution determines the number of realizations.

Binding. Binding associates specific positions within the document with external data values. Binding does not perform computation; it assigns contingent values to structurally predefined placeholders.

Inclusion. Documents may reference other documents for purposes of composition and reuse. Inclusion is treated as an auxiliary mechanism whose absence under non-executed conditions is an accepted and explicit trade-off. The document remains intelligible even when included fragments are unavailable.

A.3 Execution Model

Execution proceeds as an act of interpretation. Given a document and a data context, the executor evaluates declared structural constraints and realizes a concrete document instance compatible with those constraints.

Execution does not construct structure; it selects among structural possibilities already encoded in the document. The executor neither introduces new organization nor modifies the document's intent. Its output is one of many admissible realizations of a single, stable artifact.

A.4 Degradation and Failure Modes

A central property of the reference design is graceful degradation. When execution is unavailable or incomplete, the document remains a valid artifact.

If structural constraints are ignored, fallback content persists, structural cues remain visible, and the document retains coherence. If inclusion mechanisms fail, local intelligibility is preserved even when modular completeness is reduced. Failure affects realization, not existence.

This behavior reflects a deliberate prioritization: the persistence of the document over the completeness of execution.

A.5 Non-Responsibilities

The reference design explicitly excludes a range of concerns commonly associated with web application frameworks. These exclusions are not accidental; they follow directly from the minimalist ontology.

The executor does not: provide routing models, manage application state, define authorization schemes, or prescribe interaction lifecycles. It does not mediate client-side reactivity and does not introduce component hierarchies.

Such concerns are assumed to reside outside the document and outside the executor's scope. The executor's responsibility is limited to the interpretation of structural constraints within an existing artifact.

A.6 Summary

This conceptual reference design demonstrates that a document-centric ontology can be operationalized without introducing additional languages, opaque generation stages, or expansive execution responsibilities.

By preserving a clear separation between artifact and process, the design maintains HTML as a stable carrier of structured meaning while permitting controlled variation. It exemplifies how minimalist principles can yield systems that are conceptually economical, transparent in structure, and resilient over time.

References

[1] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[2] T. Berners-Lee, *Html tags*, `https://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html`, Accessed 2026-02-05, 1992.

[3] Microsoft, *Active server pages overview*, `https://learn.microsoft.com/en-us/previous-versions/iis/6.0-sdk/ms525870(v=vs.90)`, Historical documentation, 1998.

[4] Oracle, *Javaserver pages specification*, `https://jcp.org/en/jsr/detail?id=152`, JSR 152, 2003.

[5] E. S. Raymond, *The Cathedral and the Bazaar*. O'Reilly Media, 2003.

[6] E. S. Raymond, *The unix philosophy*, http://www.catb.org/esr/writings/taoup/html/ch01s06.html, Accessed 2026-02-05, 2003.

[7] W3C, *Html 4.01 specification*, https://www.w3.org/TR/html401/, W3C Recommendation, 1999.