A Lightweight Presentation Server with Generic Web Front End for Developer-Centric Hypermedia Publishing

Tobias Kiertscher, Robert U. Franz
Department of Business and Management
University of Applied Sciences Brandenburg
Brandenburg a. d. Havel, Germany
e-mail: tobias.kiertscher@th-brandenburg.de,
robert.franz@th-brandenburg.de

Daniel Kiertscher
Independent Researcher
Brandenburg a. d. Havel, Germany
e-mail: daniel@kiertscher.net

***Abstract*—This paper introduces a lightweight presentation server, designed to support developer-centric hypermedia publishing with immediate visual feedback. Existing systems do not simultaneously provide a programmable publishing interface, extensible media-type support, a generic web-based front end with live updates, and flexible deployment. The proposed architecture addresses this gap through a simple HTTP API, an asynchronous processing pipeline with plugin support for text and binary formats, and real-time updates via WebSockets. Initial usage indicates that such an approach improves analytical feedback cycles, reduces overhead when publishing heterogeneous content, and streamlines workflows in development and knowledge-work scenarios. The findings suggest that lightweight presentation servers can substantially enhance interactive and non-interactive media workflows in modern development environments.**

Keywords-Hypermedia; presentation server; feedback loop; volatile presentation; data visualization.

I. INTRODUCTION

Working as software developer and data analyst requires a large palette of tools: Multiple Integrated Development Environments (IDE) for efficiently working with different programming languages, database clients, programming notebooks, command line scripts, monitoring dashboards, and more. After two decades, some aspects in our workflow remain insufficiently supported by existing tools: First, immediate feedback from most of our tools is essential, to maintain cognitive flow and minimize disruptive context switching. And second, we need to be able to present varying parts of our work to ourselves and to others, with minimal overhead on a regular basis. These two aspects share a lot of common functionality. While discussing possible solutions, the idea of some kind of display took shape, where one can easily send text, images, structured data, tabular data, and linked media items, and it is presented in a pragmatic and appealing way — on our desktop, in the intranet, or the public web.

There are many existing solutions for parts of our requirements in the established development and data analysis environments. To our knowledge, there is no attempt yet, to meet all of them in an application agnostic fashion — that it can be used from almost any development or data analysis environment, without tying the display surface to a specific graphical user interface. Further, the existing tools restrict the types of media they can present to rather narrow sets, each fitting only the prime domain of the tool. We instead, want to be able to present any digital content that current technology can produce.

The web browser, with its immense capabilities to render multimedia content on different devices, is a natural choice as rendering technology for hypermedia. And that is of course, why it is already utilized in many applications for presentation, content management, and dashboards. But these have their own drawbacks and are ill equipped to serve as a local display for immediate feedback during development, or to interactively present data from an IDE.

We propose a lightweight presentation server, that is easy to run locally but can be deployed as publicly facing web server as well. It must provide a generic front end to present multiple media items, arranged on one page or separated on multiple pages. It reduces the overhead for presenting a media item to the two following steps: starting the server, and triggering a command in the IDE or running a single shell command. We introduce an extensible, asynchronous processing pipeline for arbitrary media items in text and binary form. It yields web resources, able to present the processed media item in real-time.

Our innovative approach of combining support for arbitrary media formats with a generic pre-styled front end enables a more integrated presentation workflow than alternative tools. Fewer context switches and media format transformations reduce overhead and cognitive load during development and analysis tasks. This work contributes to the field of lightweight web middleware and developer-centric visualization systems, aligning with the conference's focus on human-web interaction.

The remainder of this paper is structured as follows. Section II establishes requirements for the proposed system. Section III analyses related systems and prior work. Section IV introduces our proposed system, detailing its specifications, architecture and key innovations. Section V presents a first implementation of the proposed server with back end, processing pipeline, and front end. It ends with a list of available clients. Section VI discusses real-world deployment experiences and performance observations. Section VII concludes with a summary and future research directions.

TABLE I. REQUIREMENTS

No	Short Description	Remarks
1	**HTTP API** back end	– HTTP as a protocol with good client support in most mainstream programming languages – Publishing via a command line tool – Publishing from a graphical desktop application or an application extension
2	Support for **arbitrary media content**	– Multi- and hypermedia types, that are supported by most web browsers at the time of writing – In extension media types that can be rendered with the help of JavaScript libraries
3	**Generic web front end** for presentation of published media content	– A set of CSS rules that target typical elements of hypertext in scientific and technical documents – Allowing for simple HTML content without CSS rules, to be rendered in an appealing way – Further includes basic layout support for arranging multiple media items on multiple pages
4	**Real-time** display of published content in the web front end	– A live experience, where the user does not need to refresh the page in the web browser to see updated or added content
5	**Navigable history** for changed media items	– A user interface in the front end, that supports navigating to previous versions of published media items by the viewer
6	**Local execution** on a desktop computer as well as **deployment on a server**	– An unceremonial way of running the HTTP server — listening on the loop-back interface – No additional runtime dependencies, like, e.g., a database server
7	**Minimal protocol** overhead	– Avoiding complex XML or JSON schemas and proprietary protocols, at least for simple publishing tasks

II. REQUIREMENTS

The requirements in Tab. 1 serve as benchmark for a system that fits our workflow. Subsequent references to them use only their assigned number.

Req. 1, 3, 4, and 6 are essential. Only when these are met, a workflow involving an IDE and shell scripts with immediate feedback, in a local and in an intranet environment, is efficiently realizable. Req. 2 can be viewed with more nuance: There are essential media types, like hypertext, images, and tabular data; and more advanced media types, like video, syntax highlighted source code, data plots, geographic maps, and others. A solution that only supports essential media types is still useful, but not that effective in the targeted use cases. The requirement for certain media types depends on the domain a presentation is created for. Req. 5 is not essential but increases the value of the system in an interactive context. Req. 7 is not essential either but simplifies direct use of the HTTP API with command line HTTP clients (e.g., cURL or xh) and is helpful in accelerating the implementation of an ecosystem with various clients for the HTTP API.

III. RELATED SYSTEMS

There are multiple classes of systems, which partially support the stated requirements; however, to our knowledge, none of them support all. We briefly look at each of the following system classes: Presentation Server, Hypermedia Display Server, Content Management System, Programming Notebook, Research Document Generator, Data Visualization View in an IDE, Monitoring and Data Visualization Dashboard, Cloud-based Presentation App.

A. Headless Presentation Server

A presentation server [1] focusses on managing content with a graphical back end and/or an HTTP API, then publishing it with a web front end. A *headless* presentation server, however, does not provide a front end, but instead publishes the content by an HTTP API. That is then consumed by an additional custom front end. The support for different media types depends on the capabilities of the custom front end. Layout and styling are implemented in the custom front-end also. Immediately refreshing part of a web page is only possible if the API does support life events for changed content. A navigable history is not a typical feature, and support depends on the schema of the content. Some presentation servers like Strapi [2] can be used locally without too much overhead. The API for publishing a media item is very lean.

Unmet requirements: 3, 4, (5)

B. Hypermedia Display Server

A Hypermedia Display Server (HDS), like OpenMHEG [3], is a system that manages hyper media content together with a model for relationships and playback timing. Its main task is composition and rendering media items for TV broadcast on different devices. As a component in broadcast networks, it provides an API with standardized protocols like MHEG-5 [4]. Formats, typical in a professional broadcast environment, are supported. A special role takes SMIL [5] for animated and interactive content. The rendering surface is usually a TV, or alternatively, a player in the browser. Displaying media items in real-time is supported under the constraints of the configured timeline. Interactively navigating through the history of updated media items is not supported. An HDS is not designed for local deployment.

Unmet requirements: 2, 3, 5, 6, 7

C. Content Management System

The class of Content Management Systems (CMSs) [8] includes a wide range: simple applications for small websites, like WordPress [6], to scalable enterprise systems with many dependencies and integration endpoints, like the Adobe Experience Manager [7]. A lot of CMSs provide an HTTP API for publishing, even though the graphical back end is typically the main in-route for content. Support for different media types depends on the individual implementation. But with extensions, almost every format that is supported by a browser can be published with a CMS. The template-based web front end usually provides a great deal of freedom, regarding the page layout. And simple templates could be utilized as a generic platform for ad-hoc publishing.

Refreshing parts of a rendered web page in real-time, when its content is updated, is not supported. Although most CMSs track the changes of articles in a version history, navigating it from the front end is not supported. Local execution of a CMS is possible but usually require an additional database server, and local deployment is not a focus.

Unmet requirements: 4, (5), 6

D. Programming Notebook

Programming notebooks, like Jupyter Notebook [9], Wolfram Mathematica [10], provide an integration of code, for running data processing steps, and visualization of output produced in each step. The server component of a running notebook typically provides an API to manipulate code blocks and control their execution on a kernel. But publishing arbitrary media items is not supported. As output of the executed code, on the other hand, a rather large variety of formats is supported. A main feature of notebooks is the unity of code editor and graphical output in a flowing document. But this is typically the only layout option available. Notebooks typically provide immediate feedback when executing a code block. A notebook can be exported as a static HTML document as a means of sharing or archiving. Changes of a code block and its output are usually not tracked; therefore, navigating through the history of a specific output is impossible. Notebooks are primarily executed locally. However, some systems allow the setup of a shared workspace, where the code is executed on a server. But this kind of setup is not intended as a public presentation surface, rather than a closed collaboration space.

Unmet requirements: 1, 3, (4), 5

E. Data Visualization View in an IDE

Data visualization views in IDEs, like DataGraph [11], ILNumerics Array Visualizer [12], and Data Wrangler for VS Code [13], are either an integrated part of the IDE, or implemented in some kind of extension. Because of their tight integration into the GUI of an IDE, they do not provide an HTTP API. They often support markup text, tabular or binary data but lack the broad media support a web browser has. The views are typically isolated, and do not allow the arrangement of multiple media items in one, or even multiple pages. They provide very good integration and immediate feedback but lack the support for publishing and sharing the output. If some of them provide a way of navigating through the history of an evolving computation result, is unknown to the authors. Because they are integrated in an IDE they are only deployed and executed locally.

Unmet requirements: 1, 2, 3, 5, 6

F. Monitoring and Data Visualization Dashboard

Monitoring and data visualization dashboards, like Grafana [14] and Microsoft Power BI Report Server [15], focus on presenting frequently updated data, from several query-driven data sources. They do not provide an API for directly publishing media items. However, by updating the data source, they allow indirect publishing. A variety of data plots are supported very well but a larger set of multimedia types is typically not supported. They provide a generic front end, which allows arranging multiple views over multiple pages. The layout is constructed in a special editor mode of the graphical front end. The history of changing data can be presented as data plot. Changes in the definition of a dynamic view, or updates of static content, are not tracked at all or are not navigable in a comfortable way. Local execution is not a design goal for web dashboards, because they are usually run in concert with other servers, as part of a larger infrastructure.

Unmet requirements: (1), 2, 5, 6

G. Cloud-Based Presentation App

Cloud-based presentation software, like Prezi [16], and Google Slides [17], is the evolution of classic presentation software like Microsoft PowerPoint or LibreOffice Impress for the web. They provide very good support for manual visual design with an online editor but typically do not provide an API for publishing arbitrary multimedia items. Formatted text, geometric shapes, images, and videos are the primary media types that are supported. Any other format must be converted into an image, before publishing. They provide templates, with placeholders for typical elements of a presentation. Displaying media items in real-time, is not supported. The presentation is either in editing or in publishing mode. The history of individual items is not navigable. Local execution of a cloud service is impossible by nature.

Unmet requirements: 1, 2, 4, 5, 6

H. Performance Analysis

In Tab. 2, the considered system classes are matched against the requirements. A cross marks an unmet requirement. A cross in parenthesis marks a requirement, that is usually unmet or unmet by default but could be fulfilled with extra effort for specific systems. If Req. 1 is not met, Req. 7 cannot be met, therefore, the cell is barred. A score is created by counting the unmet requirements. If a requirement could be fulfilled with extra effort half of a point is conceded. The lower the score, the better a system class meets the targeted use cases.

Two system classes share the best score of 2.5: Headless Presentation Server and Content Management System. This indicates that the feature set of them is closest to fulfilling the stated requirements.

But at the same time, it is obvious that neither of them is a good match. A headless presentation server would require the implementation of a complete front end to suit our use cases. It could however serve as a technical basis, given that the real-time aspect is supported in its API. Content management systems are not designed for real-time publishing or for local execution. And navigating through older versions of a piece of content, is usually not supported in the front end.

The next candidates are programming notebooks, and monitoring and data visualization dashboards, with a shared score of 3.5. This indicates that they do support some aspects of the targeted use cases but are overall an even worse match. Programming notebooks do not offer an API, are not designed for real-time publishing, and do not offer any flexibility, regarding the layout in the front end.

TABLE II. RELATED SYSTEMS

System Class	Unmet Requirements							Score
	1	2	3	4	5	6	7	
Headless Presentation Server			×	×	(×)			2.5
Hypermedia Display Server		×	×		×	×	×	5.0
Content Management System				×	(×)	×		2.5
Programming Notebook	×		×	(×)	×			3.5
Data Visualization View in an IDE	×	×	×		(×)	×		4.5
Monitoring/Data Visualization Dashboard	(×)	×			×	×		3.5
Cloud-Based Presentation App	×	×		×	×	×		5.0

(×): a requirement is usually unmet or unmet by default but could be fulfilled with extra effort for specific systems

Monitoring and data visualization dashboards use data sources, like query-driven databases or APIs, for their visualizations. They do not support push-style publishing of arbitrary media items via an HTTP API. Local deployment is generally not well supported either.

In conclusion, two system classes are close but not sufficiently aligned with our use cases to match all requirements. Two more system classes can serve as an inspiration source for possible design decisions or implementation strategies as well. But none of the considered system classes is a good match. That is why we decided to design a lightweight presentation server, optimized for the stated requirements.

IV. PROPOSED SYSTEM

We propose a lightweight presentation server with the following capabilities, derived and concretized from the requirements:

- A back end, accepting display requests via HTTP POST call
- A front end, including a set of general-purpose Cascading Style Sheet (CSS) rules for hypertext
- A generic layout system for filling the screen with a limited number of media items without scrolling
- A generic layout system for an unbounded number of media items with scrolling
- Immediate update of a targeted layout position, in all browsers that show the respective web page, when processing of an arriving media item is completed
- Selecting and configuring the layout systems through management requests via HTTP calls to the backend
- An extensible mechanism for transforming arbitrary text formats into hypertext
- An extensible mechanism for rendering binary media
- Pragmatic defaults for the presentation of a wide range of media types
- Process and present large binary media items efficiently, e.g., videos of multiple gigabytes in size
- Tracking of the history of layout positions, with navigation controls at the front end
- Respecting limits for memory and disk space, when storing media items

In the following sections, we first specify the behaviour and appearance of the system followed by a description of the system's architecture. Third, we highlight the key innovations of the proposed system, regarding related systems.

A. Specifications

The system accepts a *display request* with a media item as input from the back end (HTTP API), and presents the media item at a target position in an HTML-based front end.

With every display request, a presentation intent is conveyed, entailing the media type of the transmitted content, the target position in the front end's layout, a title for the media item, and, optionally, customizing information about the way the media item is to be presented. This customizing information is subsequently referred to as *display options*. The information for specifying the presentation intent should be encoded with well-established web standards. The structure of display options certainly needs some kind of system-specific schema, but a display request without them should not be much more than a very basic HTTP POST call. Specifically, no proprietary headers and no wrapping or re-encoding of media data should be required.

The front end is structured in multiple *panels* (web pages), which in turn contain *slots* (target positions). A *grid layout* system allows an arbitrary number of slots to be arranged inside the area of the viewport, schematically visualized in Fig. 1. A *document layout* system arranges slots in a vertical sequence with unbounded space to the bottom, schematically visualized in Fig. 2. The content history of a slot is tracked. Tracking the history can be disabled for each slot individually. Slots provide a configurable toolbar, with buttons for navigating the slots history, clearing the slot, and opening the slot isolated in a new browser tab. In a grid layout, an additional button does toggle maximizing a slot to the full viewport, hiding all other slots in the panel.

Fig. 3 shows a slot with its toolbar at the top. The history navigation controls on the left, the media item title in the middle, and all other buttons on the right.

The back end accepts *management requests*, in addition to display requests. Creating, updating, and deleting panels is supported by management requests.

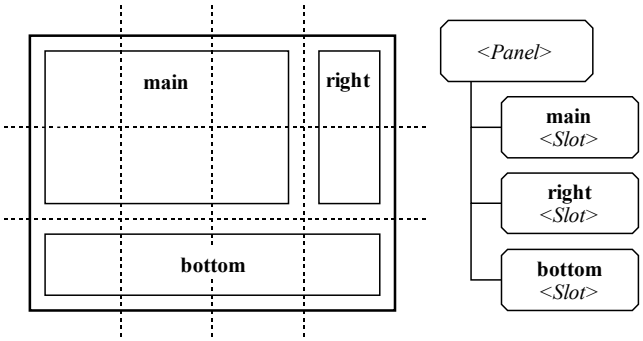

Figure 1. Grid Layout

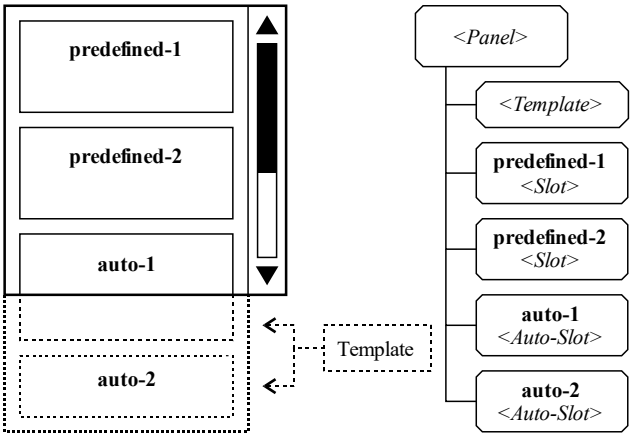

Figure 2. Document Layout

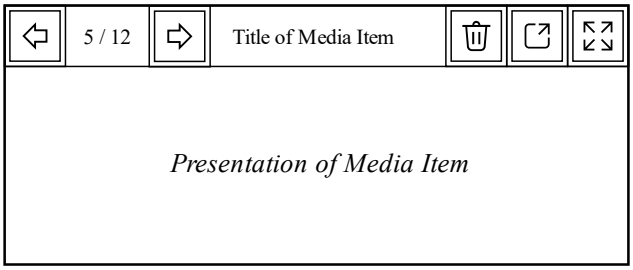

Figure 3. Slot with Toolbar

When creating or updating a panel, the layout system is selected and configured, and the available slots in the panel are specified. Management requests include clearing individual slots and whole panels, discarding all presented media items including their history.

If a display request does not specify a target panel, a special default panel serves as target. Its layout can be updated, but it cannot be deleted. If a display request specifies a target panel but not a target slot, a panel-specific default slot is used. Grid panels must specify a default slot. Document panels can specify a default slot — alternatively, a new auto-slot is created for every display request without a specified target slot. A slot configuration template is used for auto-slots.

The interpretation of transmitted media data is controlled by its media type. The media type is encoded according to [18]. The system contains a list of supported media types, each associated with default display options. For display requests with unknown media types, the system falls back to a pragmatic behaviour: If the media type indicates text, the content is presented as plain text; otherwise, the content is presented as a download button. The list of media types is customizable via system configuration, and by plugins. Creating, updating, and deleting entries in the list of media types is also supported by management requests. For user specific scenarios, where a custom media type associates media data with user-defined display options, media types with a subtype prefix "x-" are appropriate.

B. System Architecture

The system is designed as an HTTP server with two main interfaces: The back end, in the shape of an API for display and management requests, and the front end, in the shape of an HTML-based graphical user interface. An overview of the systems architecture is shown in Fig. 4. The front end consists of static web resources for styles and client code, and dynamically generated HTML for panels and slots. The front-end client code establishes a WebSocket connection [19] for bidirectional communication between the browser and the system. Presented media items are transmitted to the front end directly via WebSocket message, or, after notification via WebSocket message, through a fetch call (AJAX) from the front end to the server.

Processing requests from either back end or front end is realized asynchronously, meaning that every pending IO operation during processing, frees the CPU to process additional requests concurrently. IO operations in this context include receiving or sending data over the network and loading data from or storing data to disk. The main purpose of the system is to receive, store and deliver web resources and multimedia data — usually without intense computational tasks involved. Therefore, asynchronous processing of requests is very beneficial for the efficiency and throughput of the system.

The media type list is involved at the beginning of the display request pipeline. It associates an incoming display request with default display options. The retrieval engine provides loading media data, from the body of a display request, from a file on disk, or from a URL via HTTP GET. Text transformations can convert text formats into hypertext. This can, e.g., be utilized for syntax highlighting source code, or converting structured text formats like Markdown into HTML. The storage engine provides means for storing media data in memory or on disk. It is responsible for respecting configurable limits for consumed memory and disk space.

Media data stored in memory can optionally be persisted via regularly performed snapshots, saving the data on disk. To increase efficiency in some scenarios, streaming media data from a file on disk or an HTTP GET call, bypassing memory and disk storage, is supported.

The hypertext renderer generates the necessary HTML for displaying binary content, like images or PDF pages.

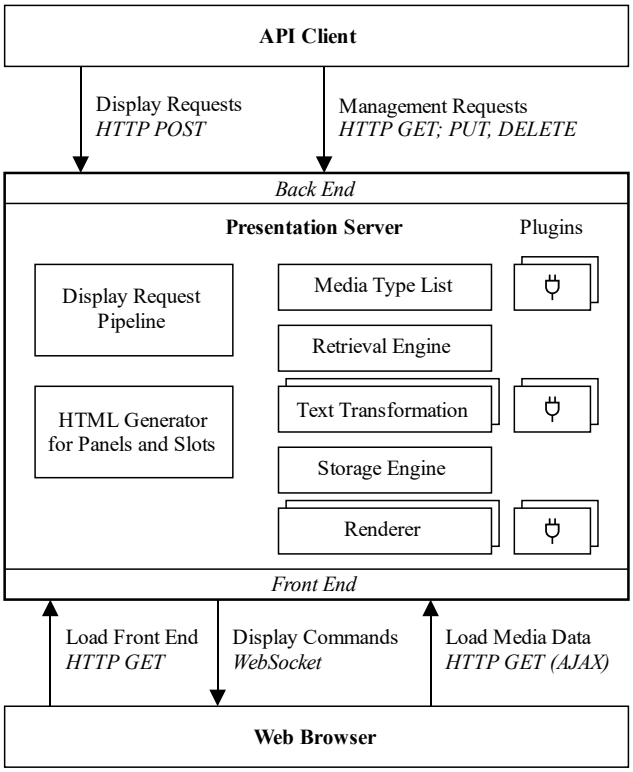

Figure 4. System Architecture

The system provides the following extension points:

1. Additional entries for the list of *Media Types*
2. *Text Transformations* for converting arbitrary text content into hypertext
3. *Renderers* to generate hypertext, capable of presenting a binary multimedia resource
4. Static web resources, supporting text transformations and renderers

Selection and parametrization of a text transformation or renderer are part of the display options. This gives full control over the way a media item is presented.

C. Key Innovations

The proposed system has two key innovations with regard to related systems:

The first is the ability to handle display requests with arbitrary media types. It has multiple levels for configuring the way a media item is presented. Which let it support pragmatic ad-hoc scenarios with minimal configuration overhead and, at the same time, allow deep customization.

1. Display request: The display request itself can carry display options, overruling all other levels.
2. Custom media types: Custom media types refer to display options. They can be installed up-front, as part of the system's configuration, or at runtime via management requests.
3. Media types provided by plugins: Plugins can provide text transformations and renderers. Additionally, they can provide matching media type definitions, referring to display options, which in turn select the provided text transformation or renderer.
4. Hard-coded behaviour for main media type classes: The system has default behaviour for "text/*" type as well as default renderers for the types "image/*", "audio/*", and "video/*".
5. Last resort: For unknown types, and specifically the type "application/octet-stream", the system provides a renderer that generates a download button.

The second key innovation is a generic front end with two layout systems, which supports presentation of media items in an appealing way, without the setup of a custom front end or cumbersome configuration of a templating system. The concepts of panels and slots build a layer of abstraction over web pages. They provide the following built-in features, that simplify publishing media items greatly.

Every panel has a header by default. The header contains a home button, that allows navigating to a home page with a list of all panels. Further, it contains controls for selecting a visual theme, and for downloading the panel as a static web site for archival purposes. Slots have a toolbar at the top by default. Buttons in the toolbar support interacting with a slot and its media content. Part of that is navigating through the history of media items in that slot.

The system provides a URL for every panel, and for every individual slot as well. That allows to open a specific slot on its own, without the panel header and other slots on the web page. Query parameters in the URL for a panel or an individual slot allow to hide header and toolbars, select a theme, apply a zoom factor to the page, and choose an offline mode without real-time updates. They provide a way to show the same panel or slot in different browser instances with an adjusted user interface.

V. IMPLEMENTATION

In the following sections, we describe a first implementation of the proposed system under the name "Boomack Server" [20]. It is publicly available as download on the project website, and as npm package "boomack" for the NodeJS runtime. The latest version at the time of writing is 0.15.

A. HTTP Server

As specified earlier, the system is an HTTP server, serving the HTTP routes for the back-end API, and providing the URLs and WebSockets for the front end.

The server is implemented as JavaScript application, running on NodeJS version 20 or later. Routing and request/response handling is provided by the ExpressJS library in version 5.1. WebSockets are handled by the library Socket.IO in version 4.8, which provides a fallback in form of long-polling for cases where the WebSocket connection fails.

B. Back End

The back end serves three groups of HTTP routes: Pure display requests, JSON-formatted display requests, and management requests.

Pure display requests transmit the data of a single media item to a single slot without re-encoding. The data is directly sent as body of an HTTP POST call. Media type, and data length are transmitted in the appropriate standard header fields [21, Sec. 8.3] and [21, Sec. 8.6] respectively. Target panel and slot are encoded in the path portion of the URI. Therefore, every slot has its own URI for publishing media items. Title and display options for the media item are transmitted in custom header fields with prefix "X-". The title is percent-encoded [22, Sec. 2.1] in the header "X-Boomack-Title". The display options are serialized as JSON [23] and optionally encoded with BASE64 [24] in case they include control characters or characters outside of the ASCII table. They are placed in the header "X-Boomack-Options". Pure display requests allow for efficient transmission of large media items, because the media data does not need to be loaded fully into memory when sending or receiving, but can be streamed in chunks. Referencing media data via URL is not possible in pure display requests.

Pure display requests, in simple cases, mirror typical HTTP GET calls. Without the optional media item title and display options, only basic HTTP mechanisms are used to convey the presentation intent.

JSON-formatted display requests can transport multiple media items. They have the content type of "application/json" and contain either one display request, encoded as a JSON map, or a list of display requests, encoded as a JSON array of maps. The target position is not encoded as part of the URI but instead as values for the keys "panel" and "slot" inside each display request. The title of the media item is a Unicode string under the key "title". Display options are included as map under the key "options". The media data can be embedded or referenced. If referenced, its URL is placed under the key "src". If embedded, it can be either placed under the key "text" if the media data is a Unicode character string, or under the key "data" as BASE64-encoded byte sequence. Because an HTTP POST call for publishing JSON-formatted display requests can contain multiple display requests, and the target positions are encoded inside the JSON body, only one URI for that kind of request is needed. JSON-formatted display requests are not well suited for large media items, because all data needs to be loaded into memory for JSON serialization. Additionally, the BASE64 encoding increases the amount of transmitted data by at least 33%. Which is no concern for small images but can lead to bad system performance for large images and videos.

Management requests allow creating and updating panels and media types via HTTP PUT call with the content type of "application/json". They support removing panels and media types via HTTP DELETE call. Panels and slots can be cleared from published media items with an HTTP POST call. And management requests allow the retrieval of the layout of panels and the definition of media types via HTTP GET call. The GET calls support "text/plain", "application/json" and "text/html" as response format.

C. *Display Request Processing Pipeline*

The core of the system is an asynchronous request processing pipeline, visualized in Fig. 5. The pipeline accepts a display request from the back end as input and outputs either an error message or a display command for the front end. The pipeline is implemented with asynchronous functions. That way, many display requests can be processed concurrently.

Every display request, pure or JSON-formatted, is first converted into a JavaScript object called *Display Task*. The media data is represented by one of four properties "text" (String), "data" (Bytes), "src" (URL), or "stream" (Stream). The first three are mere copies from a JSON request. The last is a handle to the data stream of the HTTP body from a pure display request. This display task is then handed down through the pipeline with the following steps: Normalize, Load, Transform, Store, Format, Render, and Command. Intermediate results as well as the final display command are attached to the display task during processing.

In the **Normalize** step, optional information, like target position, media item title, and display options, including the selected text transformation or renderer, and storage mode, are completed. The media type is looked up in the media type list of the system, and the layers of display options are merged. If not already answered by explicit display options, the following questions are answered using configurable thresholds, heuristics, and hard-coded defaults:

- Where should the media item be presented?
- What content type has the media item?
- Does the media item have text or binary data?
- *If the media data is text:*
  - Must a text transformation be applied?
  - Should existing content at the target position be replaced or extended?
- *If the media data is binary:*
  - Can it be embedded in HTML as a data-URL [25], or is it served as a separate resource?
  - What renderer is the data presented with?
- Is the media data referenced or included?
- *If the media data is referenced:*
  - Must the data be loaded, or can it stay referenced for streaming through?
- *If the data is loaded and not embedded in HTML:*
  - How should the media data be stored?
- Is the media item presented in an HTML iframe?

The result of this first step is a detailed configuration for the rest of the processing pipeline.

In the **Load** step, the retrieval engine retrieves and decodes the included or referenced media data.

In the **Transform** step, text data is transformed by calling a transformation function from a plugin. The transformation function is selected with the display options for parameterization. The fallback text transformation for the generic "text/*" media type treats the media data as plain text and converts it into HTML with forced line breaks and a monotype font. Besides the generated HTML, a text transformation optionally returns a list with static web resources

(scripts, styles), that must be loaded in the web page, before the generated HTML can be rendered properly.

In the **Store** step, the media data is either stored in memory or as file on disk. Stored and referenced media data is later referred to as *Media Resource* and is identified by a UUID Version 4 [26, Sec. 5.4].

In the **Format** step, a URL for the media resource is formed. If the display request was referring to media data via an http(s) URL, and the media content is to be presented in an HTML iframe, the URL can simply pass through as external resource URL. If the media item has binary content, which is stored or referenced via a file URL, an internal resource URL is generated. If the media data can be embedded in a data-URL, then this is generated instead.

In the **Render** step, HTML code is generated by calling a renderer function from a plugin. It is parameterized with media resource URL and display options. Fallback renderer for the generic media types "image/*", "video/*" and "audio/*" generate *img*, *video*, and *audio* tags. With *video* and *audio* tags being rendered in the browser with a default user interface of a media player. Fallback for "application/octet-stream" and any unknown type is a renderer that generates a button for downloading the media resource. Besides the generated HTML, a renderer optionally returns a list with static web resources (scripts, styles), that must be loaded in the web page, before the generated HTML can be rendered properly.

In the **Command** step, intermediate results from the processing pipeline are restructured into a display command. This command is queued for immediate transmission to the front end over all open WebSocket connections, associated with the panel or slot which is targeted by the display command.

Transform and Render steps provide extension points for plugins. Plugins expand the media type support. The system contains several core plugins that are always available and do not need to be installed as an extension:

- Text Transformations
  - **plain**
    described as fallback under the Transform step
  - **highlight**
    syntax highlighting for various text formats and programming languages
  - **markdown**
    converts Markdown into HTML
  - **csv**
    converts CSV data into an HTML table
- Renderer
  - **generic**
    generates HTML from a template, containing a placeholder for the resource URL
  - **image**, **audio**, **video**
    described as fallbacks under the Render step

D. Front End

The front end consists of dynamically generated HTML for panels and slots, JavaScript code, CSS rules, and web font files. It is designed to be fully functional without an internet connection.

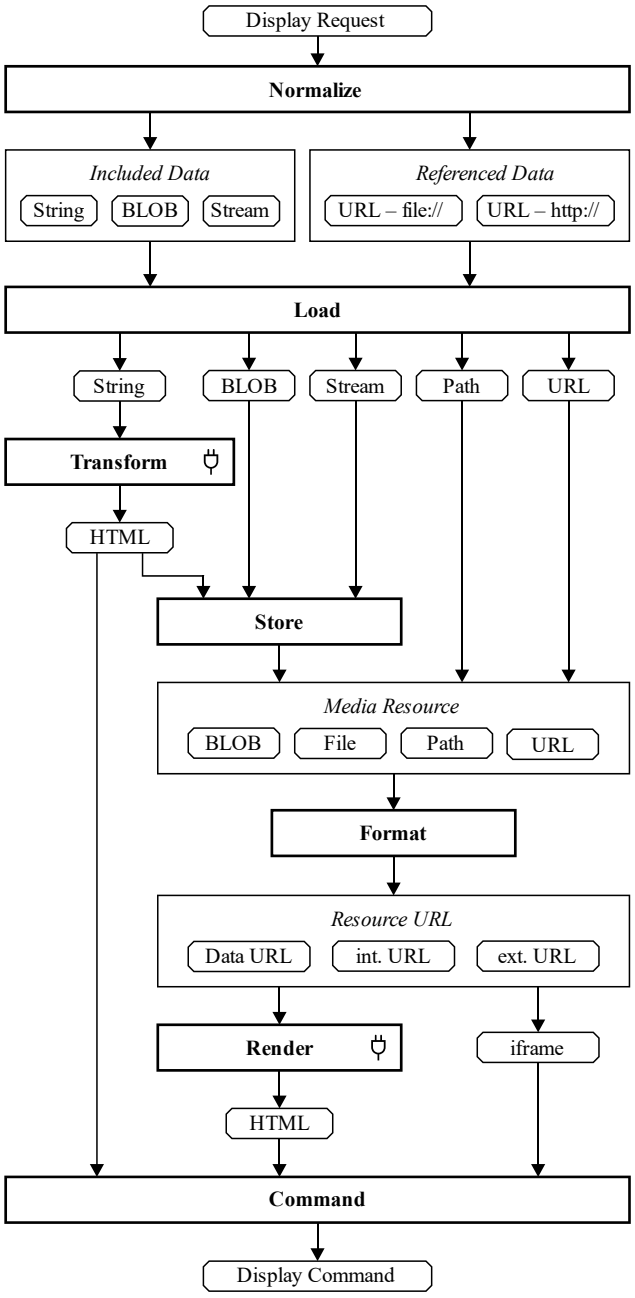

Figure 5. Display Request Processing Pipeline

That means that no resources are loaded from Content Delivery Networks (CDN); instead, all required resources are served by the HTTP server itself.

The HTML code for home page, panel page and slot pages is generated server-side with EJS templates. The JavaScript code facilitates WebSocket management, dynamic update of slot content, lazy loading of required web resources, and interactive features of the front end. It further provides an API, exposing UI features to script blocks in presented hypertext. The CSS rules consist of a customized Fomantic UI stylesheet and a set of system specific rules for the layout of home page, panels, and slots. Fomantic UI, a

fork of Semantic UI, is a CSS framework with a wide range of supported components, of which only a subset is used by the front end for menus and toolbars. The framework is compiled in two variations: A full version with all components, and a slim version with the tailored subset. The layout of a panel specifies if the full or slim variation is to be loaded on the page. If presented hypertext is supposed to contain sub-layouts, visually structured elements, buttons, or forms, it can make use of the full set.

E. Ecosystem

In parallel with developing Boomack Server, an ecosystem of API clients was initiated. At the time of writing the following clients are publicly available.

Dedicated libraries for the programming languages JavaScript, C#, and Rust allow comfortable interaction with a Boomack Server — currently, other programming languages must utilize an HTTP client directly to interact.

Two command line clients: The first is implemented in JavaScript for the NodeJS runtime. It is available as npm package "boomack-cli" and is used for prototyping and functional evaluation during development of the back-end API and the command line interface. The second, implemented in Rust, is focused on performance and is used to evaluate integration with shell scripts. It is built on top of the Boomack Library for Rust and lags behind the JavaScript implementation feature-wise, as only mature features are ported to Rust.

A prototypical implementation of a background service for monitoring tasks was implemented. It regularly performs queries against a relational database. The results are presented as tabular data, as Vega Lite plot, or as hypertext, generated via template.

A first IDE extension, targeting Visual Studio Code, is available. It is used for evaluating interactive workflows and integration strategies into IDEs.

The following plugins for the Boomack Server are available as npm packages: "boomack-plugin-vega" for rendering Vega and Vega Lite plots, "boomack-plugin-mermaid" for rendering mermaid charts, "boomack-plugin-leaflet" for rendering markers on geographic maps from the OpenStreetMap project, and "boomack-plugin-pdf" for rendering a PDF page via the Mozilla PDF.js library.

VI. CURRENT EXPERIENCE

First and foremost, the ability to define a presentation with formatted text, highlighted source code, tabular data, data plots, charts, and images in source code, in the programming language of our choice, is liberating for our workflow. We can structure, re-use, automate and version control presentations from within our IDE or the command line. Having immediate visual feedback for pieces of information during development and analysis yields a better understanding of our data and a workflow, less interrupted by mental context switches. Getting used to a common interface for different publishing tasks increases the efficiency for communicating insights to ourselves and to others.

We observed reduced time spent transforming and preparing media items for presentation, due to the system's support for a broader range of media formats compared to other presentation tools. This factor is amplified when presentations recur regularly, with partial content changing each time. The integrated and official plugins, implemented at the time of writing, cover all media formats we use regularly. Data plots are chiefly supported by Vega and Vega Lite, though their JSON-based syntax necessitates a significant learning curve. A set of templates for typical plots helps mitigate this complexity.

The supported abstractions (panel, slot, grid and document layout) give a lot of freedom to arrange, structure, and link media items when publishing. The concepts remain accessible through intentional simplicity yet accommodate diverse publishing scenarios. The capabilities of the storage engine together with the slot history provide good support for volatile short-term presentations and long-term presentations as well.

Specifically, we use the system in the following scenarios: Data analysis with Python within Visual Studio Code, and with Clojure within Emacs; in both environments a thin abstraction over the HTTP API, tailored to the plot types required for data exploration, is used. In this scenario each developer uses a local instance of the Boomack Server. Panels are usually set up with just the layout, and no further visual customization. Here, we benefit greatly from the system's pragmatic defaults and integrated themes.

We use the Boomack Server on our intranet, to communicate asynchronously. Each colleague can have a panel to regularly publish information about individual progress or open questions. In this scenario we profit from the ability to seamlessly integrate the presentation aspect into our development work.

A publicly deployed Boomack Server is used as dashboard for system metrics. A .NET application publishes statistics about user activity and data state in the shape of key figures and plots. Some panels on the dashboard are configured with custom style rules for a refined visual appearance.

Besides the professional work, the Boomack Server is used in a smart home environment to present temperature curves and web cam feeds.

When the API of the system is used by software developers or data scientists proficient in programming, we have positive feedback regarding the flexibility of the system. However, creating a presentation that deviates from the themes provided requires at least solid knowledge about HTML and CSS.

The most recent addition to the ecosystem of API clients is the Visual Studio Code extension. By exposing display and management features of a Boomack Server through a graphical user interface, the capabilities of the system are more accessible to a user group less comfortable with programming or command line use.

VII. CONCLUSION AND FUTURE WORK

This paper proposes a presentation server as versatile hypermedia publishing tool for software developers and knowledge workers comfortable with source code and the command line. Our requirements analysis identified no suitable preexisting candidate across seven different hypermedia related system classes. The proposed architecture emphasizes

a simple publishing interface and flexible deployment options with three main components: HTTP API as back end, display request processing pipeline with extension points as core, and browser-based front end with two layout systems.

The key innovations of the proposed system are extensible support for arbitrary media types and a generic but customizable front end.

Pragmatic defaults throughout the processing pipeline reduce the necessary overhead for publishing a media item to a minimum. A first implementation of the proposed system does support our workflow for data science and software development better than evaluated alternatives.

In future work the system's performance should be evaluated under different load patterns. Alternative layout systems should be evaluated. Further, the system's performance should be assessed for specific workflows in combination with different API clients. Especially for interactive presentation, the user interface for publishing has a significant impact on efficiency. The ecosystem of API clients and plugins should be validated and extended on demand.

ACKNOWLEDGMENT

We are grateful to Theresa Schulz for her critical review of the manuscript. We thank the technical staff at the Department of Business and Management for their support in deploying the system for intranet and internet.

REFERENCES

[1] R. N. Taylor, N. Medvidovic, and E. Dashofy, Software Architecture: Foundations, Theory, and Practice. Chichester, England: John Wiley & Sons, 2009.

[2] Strapi Team, “Strapi – Open-source headless CMS.” [Online]. Available: https://github.com/strapi/strapi. (accessed Jan. 6, 2026)

[3] G. Group, “OpenMHEG.” Aug. 2008, [Online]. Available: https://code.google.com/archive/p/openmheg/. (accessed Jan. 6, 2026)

[4] “Information technology – Coding of multimedia and hypermedia information – Part 5: Support for base-level interactive applications,” ISO13522-5, 1997.

[5] D. Bulterman et al., “Synchronized Multimedia Integration Language (SMIL 3.0).” Dec. 2008, [Online]. Available: https://www.w3.org/TR/SMIL3/. (accessed Jan. 6, 2026)

[6] WordPress Developer, “WordPress.” Jan. 2004, [Online]. Available: https://wordpress.org/. (accessed Jan. 6, 2026)

[7] Adobe, “Adobe Experience Manager (AEM).” Oct. 2025, [Online]. Available: https://experienceleague.adobe.com/en/browse/experience-manager. (accessed Jan. 6, 2026)

[8] J. T. Hackos, Content management for dynamic web delivery. Nashville, TN: John Wiley & Sons, 2002.

[9] T. Kluyver et al., “Jupyter Notebooks - a publishing format for reproducible computational workflows,” in Positioning and Power in Academic Publishing: Players, Agents and Agendas, 2016, pp. 87–90.

[10] S. Wolfram, The Mathematica Book, 5 ed. Champaign, IL: Wolfram Media, 2003.

[11] A. Shuvaev, “DataGraph.” Sep. 2023, [Online]. Available: https://plugins.jetbrains.com/plugin/22472-datagraph. (accessed Jan. 6, 2026)

[12] ILNumerics GmbH, “ILNumerics - Array Visualizer.” Oct. 2025, [Online]. Available: https://ilnumerics.net/array-visualizer-c.html. (accessed Jan. 6, 2026)

[13] Microsoft, “Data Wrangler Extension for Visual Studio Code.” 2021, [Online]. Available: https://code.visualstudio.com/docs/datascience/data-wrangler. (accessed Jan. 6, 2026)

[14] T. Ödegaard, “Grafana.” 2014, [Online]. Available: https://grafana.com. (accessed Jan. 6, 2026)

[15] A. Bansal and A. K. Upadhyay, “Microsoft Power BI,” in International Journal of Soft Computing and Engineering (IJSCE), Jul. 2017, vol. 7-3.

[16] Prezi Inc., “Prezi.” Apr. 2009, [Online]. Available: https://prezi.com/. (accessed Jan. 6, 2026)

[17] Google LLC, “Google Slides.” Mar. 2006, [Online]. Available: https://www.google.com/slides/about/. (accessed Jan. 6, 2026)

[18] N. Freed and D. N. S. Borenstein, “Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types,” Request for Comments, no. 2046. RFC Editor, Nov. 1996, doi: 10.17487/RFC2046.

[19] A. Melnikov and I. Fette, “The WebSocket Protocol,” Request for Comments, no. 6455. RFC Editor, Dec. 2011, doi: 10.17487/RFC6455.

[20] T. Kiertscher, “Boomack - A Lightweight Presentation Server for Research and Development.” [Online]. Available: https://boomack.com/. (accessed Jan. 6, 2026)

[21] R. T. Fielding, M. Nottingham, and J. Reschke, “HTTP Semantics,” Request for Comments, no. 9110. RFC Editor, Jun. 2022, doi: 10.17487/RFC9110.

[22] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” Request for Comments, no. 3986. RFC Editor, Jan. 2005, doi: 10.17487/RFC3986.

[23] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” Request for Comments, no. 8259. RFC Editor, Dec. 2017, doi: 10.17487/RFC8259.

[24] S. Josefsson, “The Base16, Base32, and Base64 Data Encodings,” Request for Comments, no. 4648. RFC Editor, Oct. 2006, doi: 10.17487/RFC4648.

[25] L. M. Masinter, “The “data” URL scheme,” Request for Comments, no. 2397. RFC Editor, Aug. 1998, doi: 10.17487/RFC2397.

[26] K. R. Davis, B. Peabody, and P. Leach, “Universally Unique IDentifiers (UUIDs),” Request for Comments, no. 9562. RFC Editor, May 2024, doi: 10.17487/RFC9562.