





# AEM Gems – AEM SPA Editor, part 1

Gabriel Walt, Product Manager & Patrick Fauchere, Engineering Manager



# Shipped: AEM 6.4 Service Pack 2

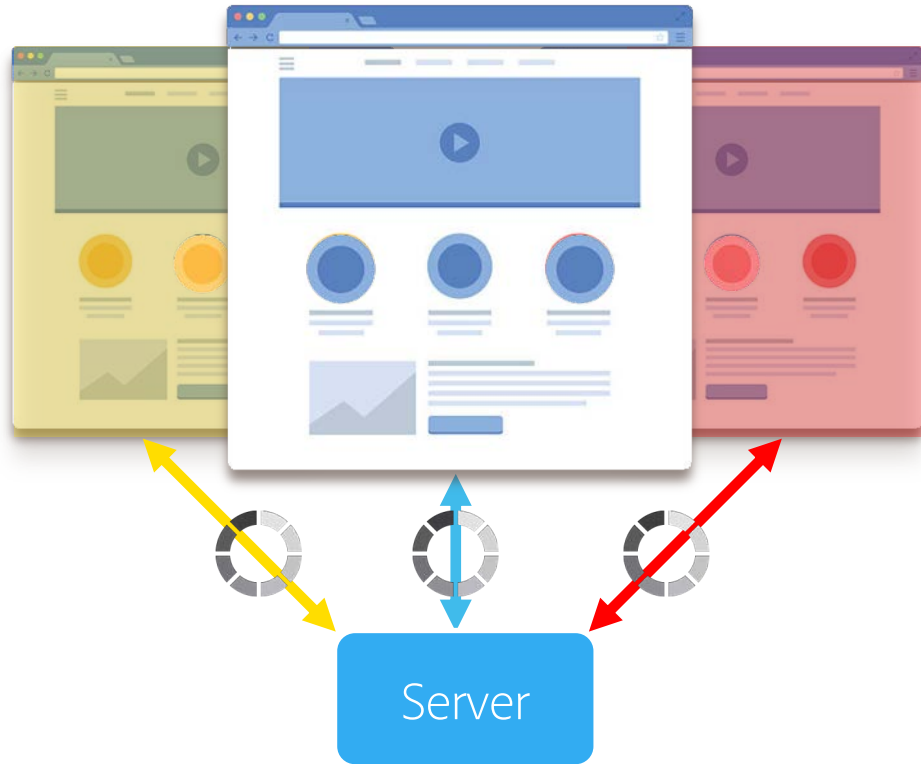
## SPA[\*] Editor Version 1.0

- Offers in-context editing for JavaScript apps
- Supports  React and  Angular
- Support of History API for routing
- Support for state library, like Redux
- Tech preview for JS Server-Side Rendering

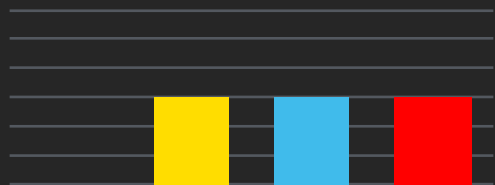
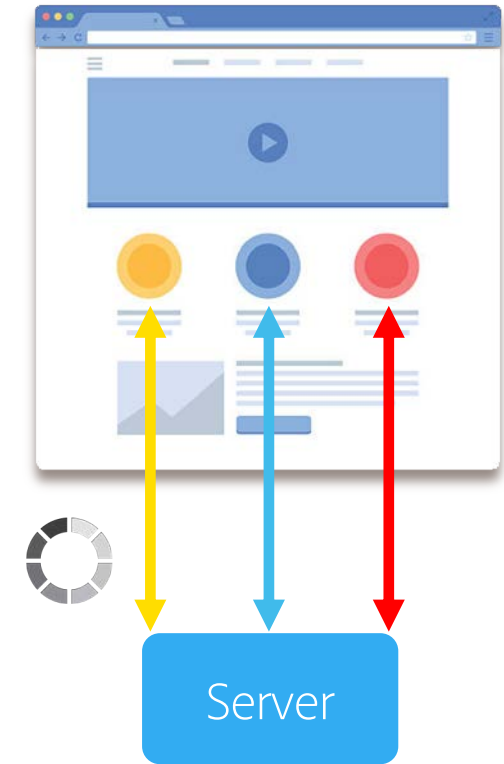
[\*] SPA = Single Page Application



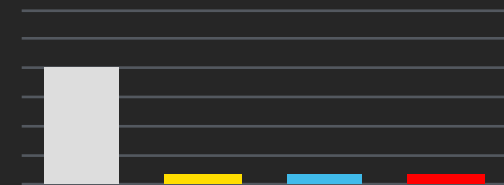
# Multi-Page Experience



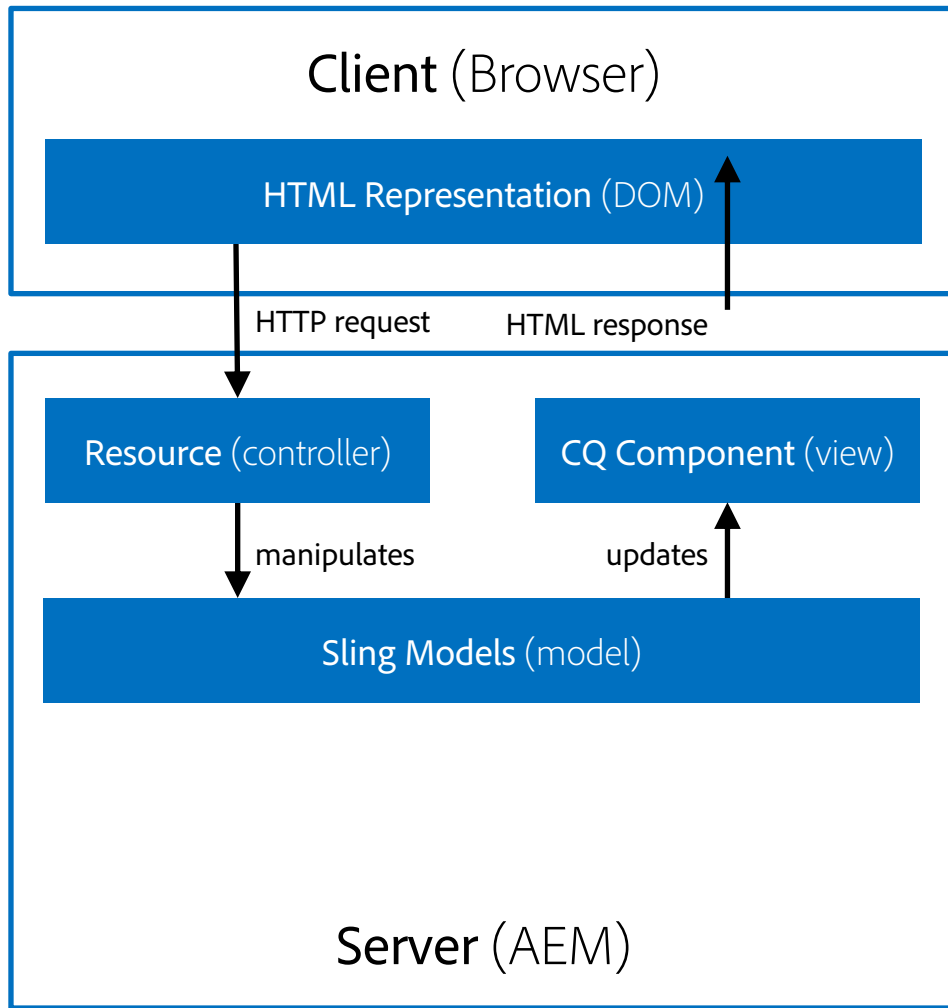
# Single-Page Experience



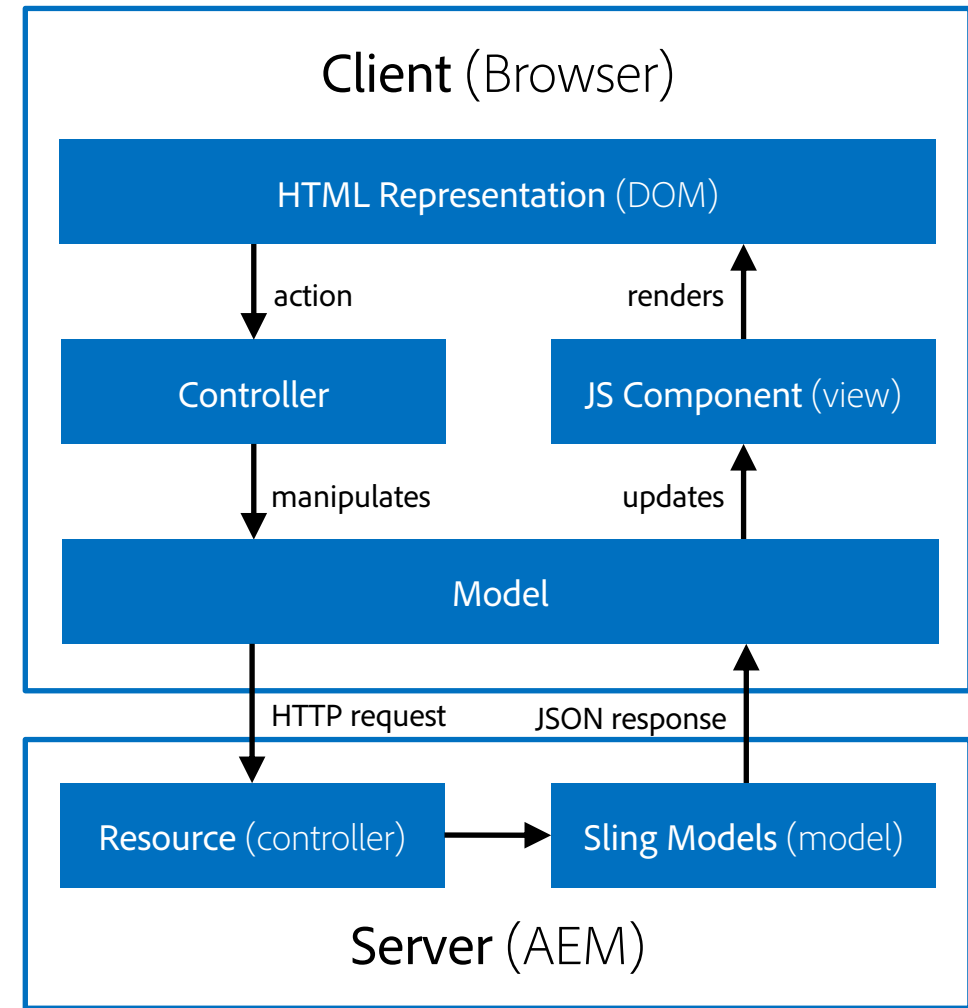
Time Loading



# Architecture of Multi-Page vs Single-Page Experiences



Multi-Page Experience



Single-Page Experience

# Benefits of Multi-Page vs Single-Page Experiences

	Multi-Page Experience	Single-Page Experience
Response time	<p>⊖ Slower Usually 1s to 5s</p>	<p>⊕ Fast Usually 100ms to max 1s</p>
Complexity of implementation	<p>⊕ Simple Mostly just HTML + CSS</p>	<p>⊖ More complex Framework-specific JS implementation</p>
Technological obsolescence	<p>⊕ More sustainable Uses more standardized technologies</p>	<p>⊖ Rapid framework evolution New frameworks appear every year</p>
Development workflow	<p>⊖ Complex Front-end handing off HTML to back-end engineer is not optimal</p>	<p>⊕ Simple Front &amp; back-end engineer agree on JSON and can work much more independently</p>
Search Engine Optimization	<p>⊕ Works as-is Even with invalid HTML, the indexation will work for all search engines</p>	<p>⊖ Might require some effort Google and Bing can execute JS when it doesn't throw errors, SSR can mitigate</p>

# Experience Management for Single-Page Experiences

AEM Experience Management UI  
(SPA Editor & Site Admin)

Your Single-Page Experiences  
(using the AEM JS SDK)

JSON APIs

AEM as a  
Headless CMS  
(Content Services)

Your other  
(micro) services

# SPA Editor – In-context capabilities

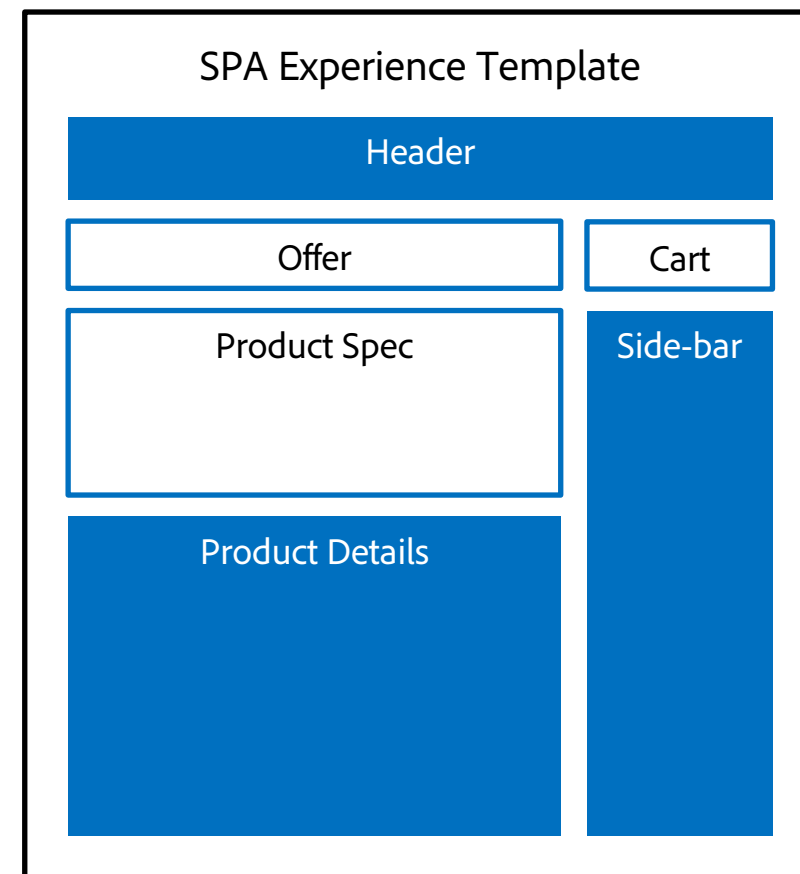
For content from AEM's JSON APIs:

1. **Editing:** define texts, images and other content of JS components.

For content from all services (AEM & your other services):

2. **Composition:** define which JS components are shown.
3. **Configuration:** define the settings of the JS components.
4. **Layout:** define how the JS components flow on the responsive grid.
5. **Template:** define and edit what is shared among app states.

→ As marketers see the same experience than visitors, they can immediately evaluate and optimize the end-user experience in-context during authoring.



- content from AEM (as a headless CMS)
- data from your other services



Demo Time

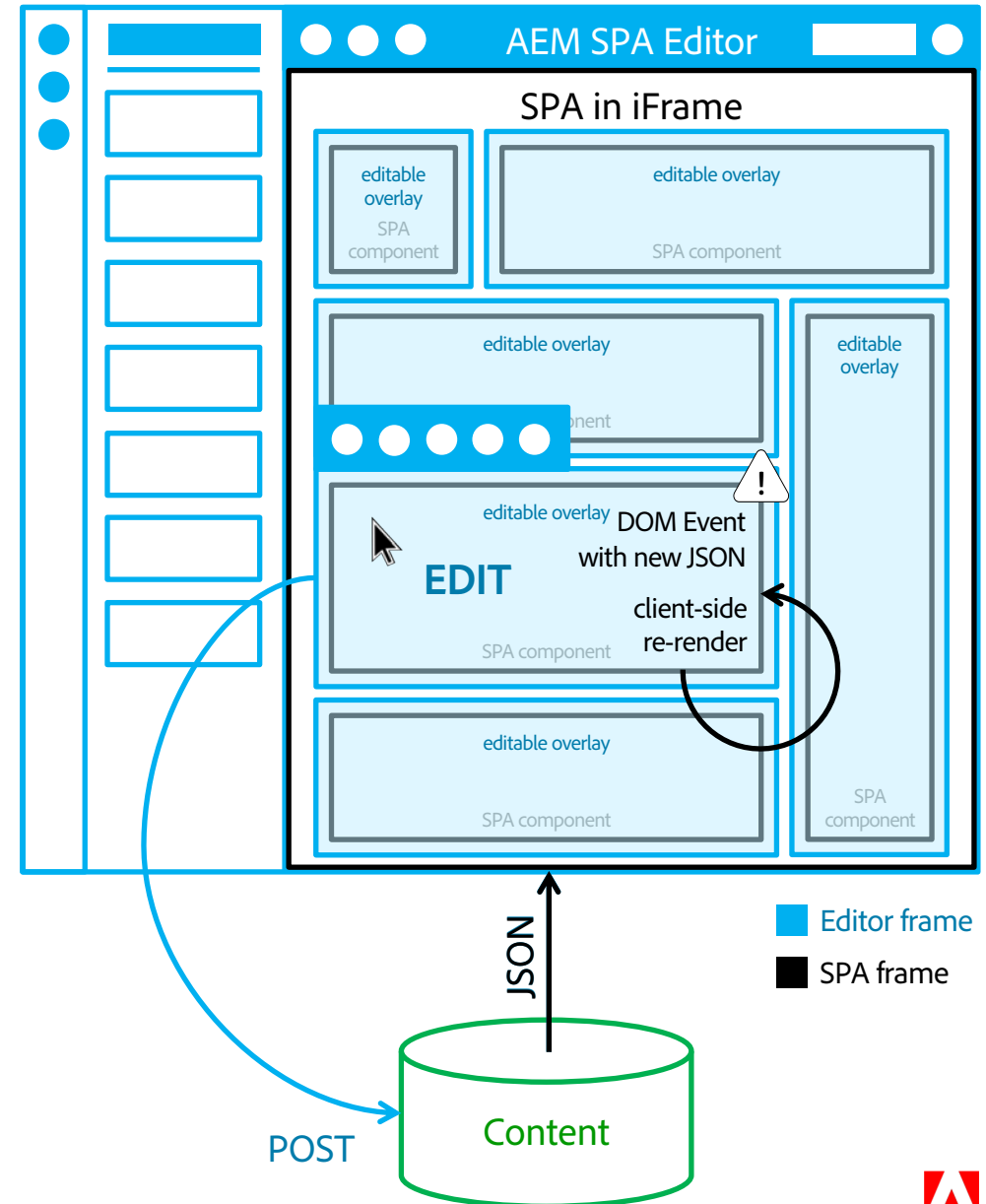




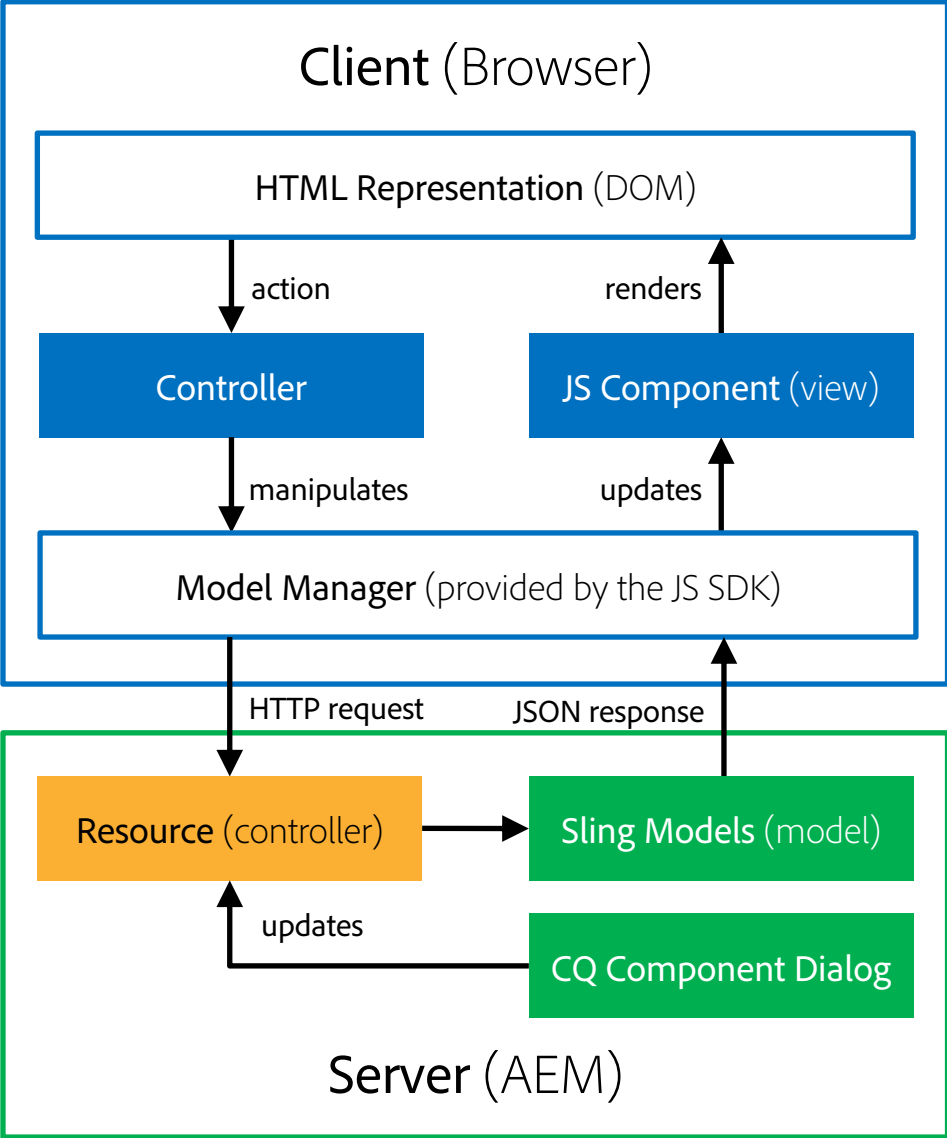
# SPA Editor – How it works

1. SPA Editor loads.
2. SPA is loaded in a separated frame.
3. SPA requests JSON content and renders components client-side.
4. SPA Editor detects rendered components and generates overlays.
5. Author clicks overlay, displaying the component's edit toolbar.
6. SPA Editor persists edits with a POST request to the server.
7. SPA Editor requests updated JSON to the SPA Editor, which is sent to the SPA with a DOM Event.
8. SPA re-renders the concerned component, updating its DOM.

- The SPA is isolated from the editor and always in charge of its display.
- In production (publish), the SPA editor is never loaded.



# SPA Editor – Separation of Concerns



Front-End Engineer

← Good separation of concerns based on JSON contract



Back-End Engineer



Marketing Author

# SPA Editor – JS Components

JS components can be implemented AEM agnostic, and follow framework-specific best practices.

→ What is needed is to map them to the AEM resource types via `MapTo()`

## React

```
import React, { Component } from "react";
import { MapTo } from
  "@adobe/cq-react-editable-components";

class MyComponent extends Component {
  render() {
    return (
      <p>
        { this.props.text }
      </p>
    );
  }
}

MapTo("myResourceType")(MyComponent);
```

## Angular

```
import { Component, Input } from "@angular/core";
import { MapTo } from
  "@adobe/cq-angular-editable-components";

@Component({
  templateUrl: "./my-component.component.html"
})
class MyComponent {
  @Input() text:string;
}

MapTo("myResourceType")(MyComponent);

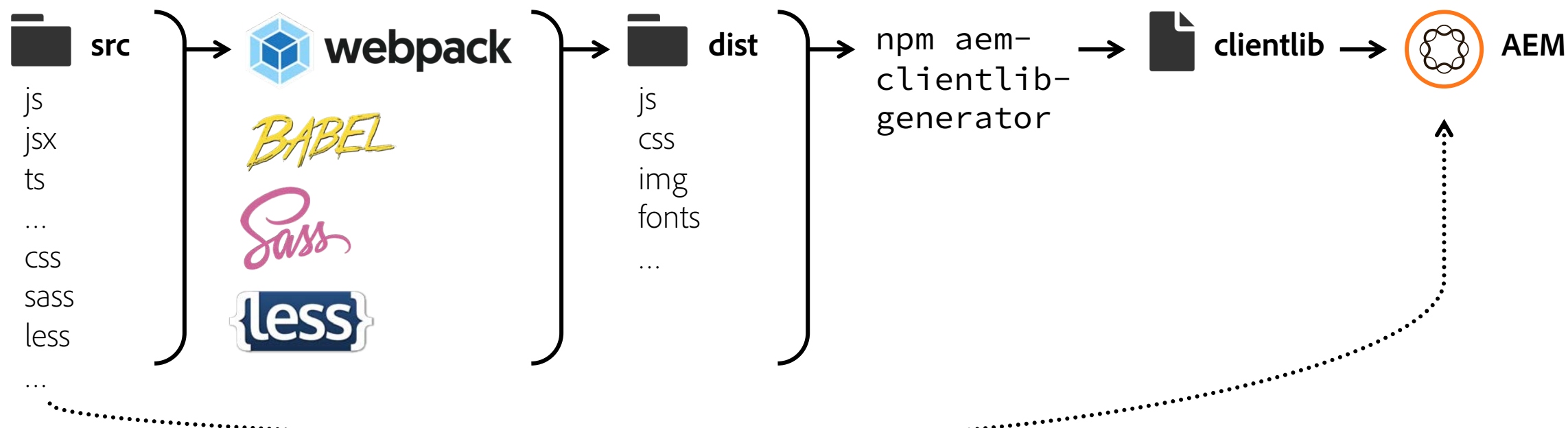
<!-- my-component.component.html -->
<ng-template>
  <p>{{text}}</p>
</ng-template>
```



# SPA Editor – Front-end compilation & tooling

Compilation of JavaScript and of CSS is done outside of AEM and then converted in a ClientLib.

→ Potentially any tooling can be used.



**aemfed** – allows for live coding of js and less files using aemsync, Browsersync and Sling Log Tracer

# Debunking Some Myths

As the SPA Editor disrupts how authoring has been done for years.

→ We need to debunk some deeply rooted myths about authoring for SPA.

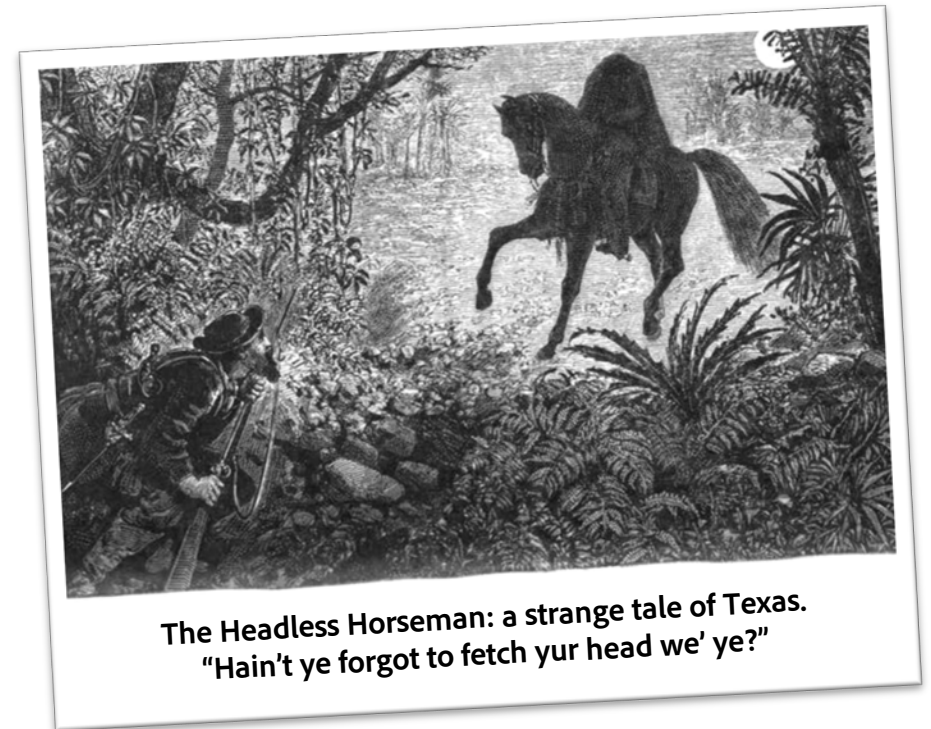


# Myth #1 – Headless Only

**MYTH:** Separating the content from the presentation allows to better reuse it in multiple channels, therefore authors should work headless only.

**BUSTED:** It's true that such semantic content can be well reused, but when publishing it to a channel, context-specific changes must be possible.

→ Therefore, authors must also have access to some good in-context editing capabilities.





## Myth #2 – CSS Job Security

**MYTH:** Front-end engineers are the most efficient to define the layout of the SPA content with just some CSS, therefore I need no experience management system.

**BUSTED:** That's probably true for the initial release, but font-end engineers will then endlessly have to adapt their CSS layout to content changes made by authors, becoming a real bottleneck to the publication process.

→ Therefore, authors must have a way to self-service layout changes too.



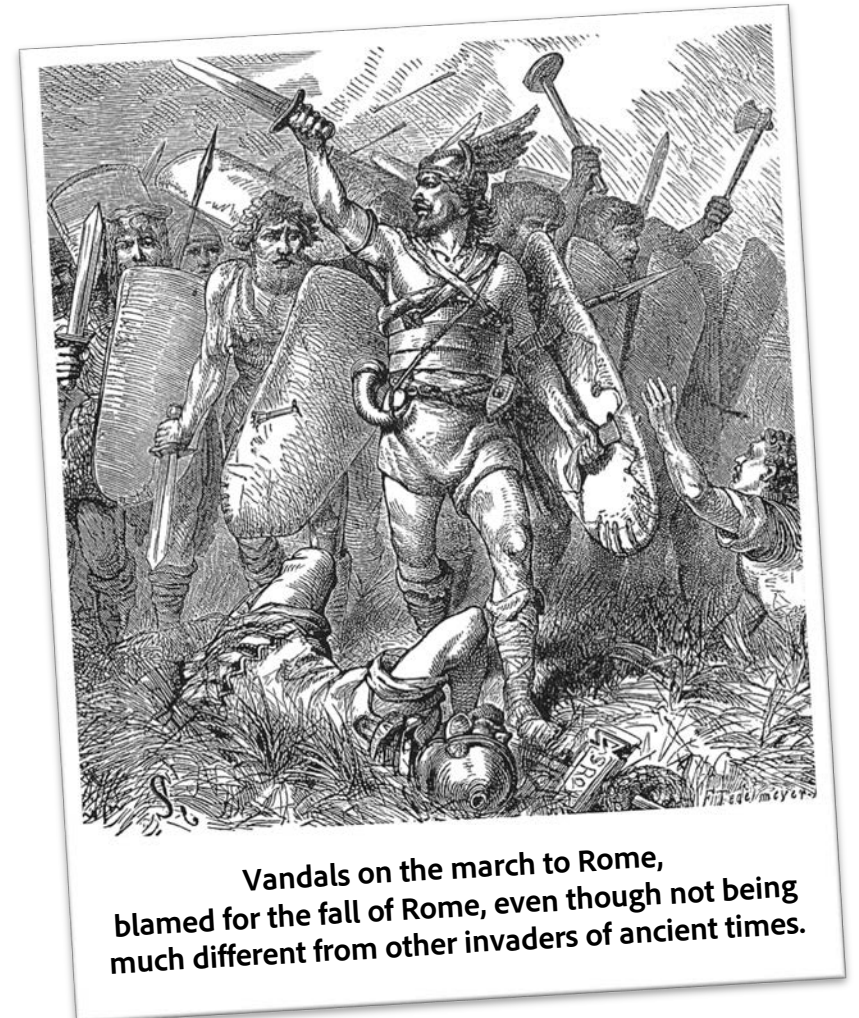
**Sisyphus: punished by Thanatos to roll a giant boulder up a hill, only for it to roll down again when nearing the top, repeating this forever.**

# Myth #3 – Dangerous Authors

**MYTH:** Authors will break my SPA if I let them compose, configure, and layout the SPA components.

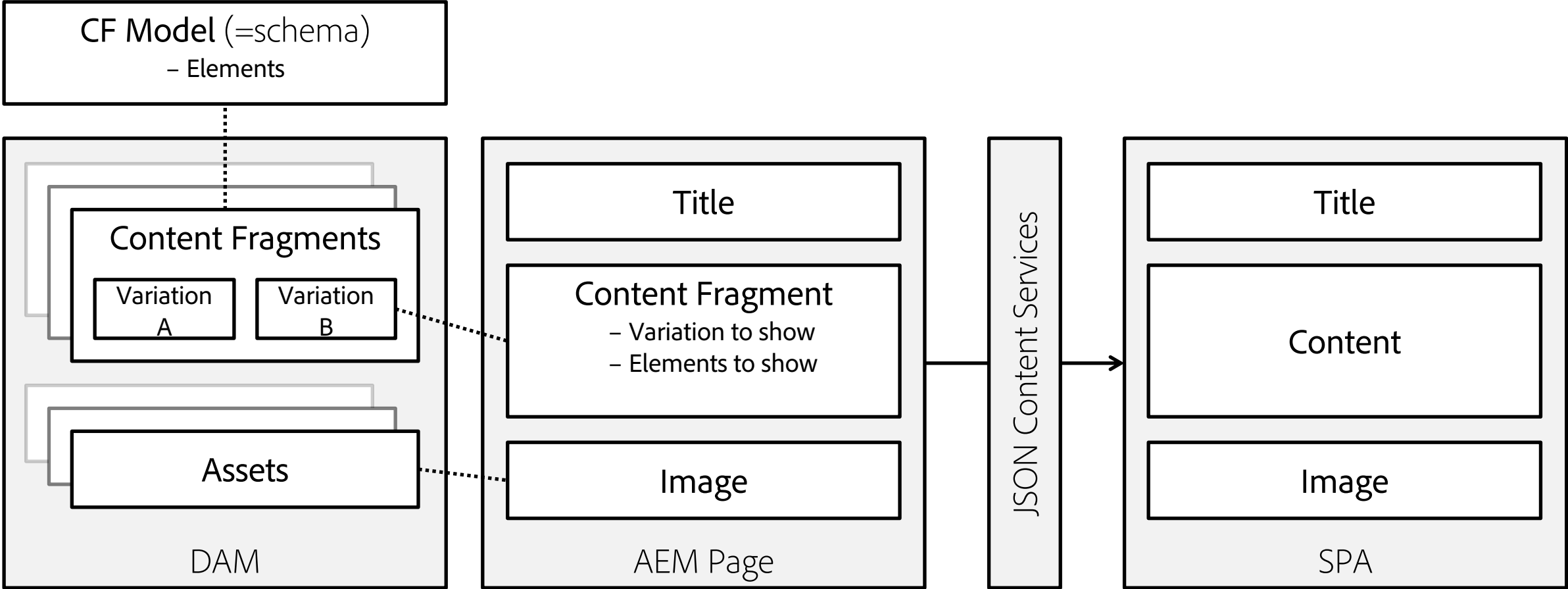
**BUSTED:** It's true that authors could create ugly layouts, but fundamentally, managing the layout of content for apps is not different than managing it for traditionally rendered websites.

→ The same tools and strategies that allow authors to successfully manage the layout for websites can and should also be applied to SPA.



# SPA Editor – Headless or Headful?

Each is a tool that is best for its specific purpose → combining them works best.





# SPA Editor – Roadmap

0 = we don't need it  
 2 = we would use it  
 4 = we must have it

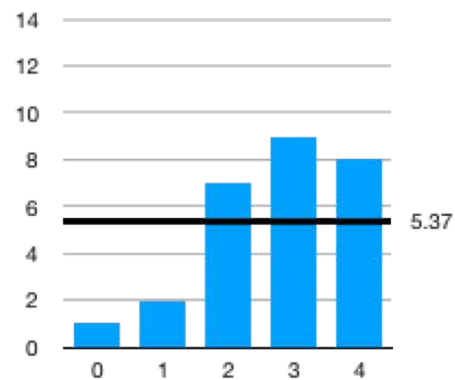
Redux State Management



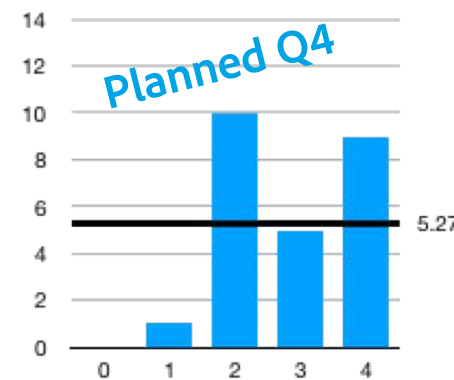
JS Server-Side Rendering



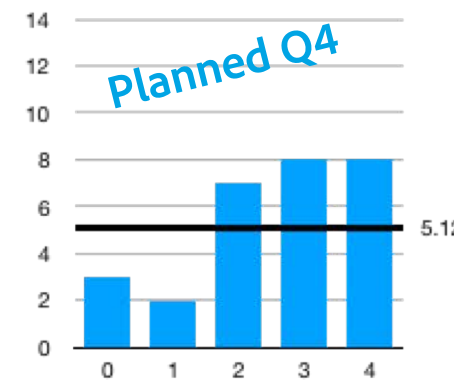
Front-End Dialog Definition



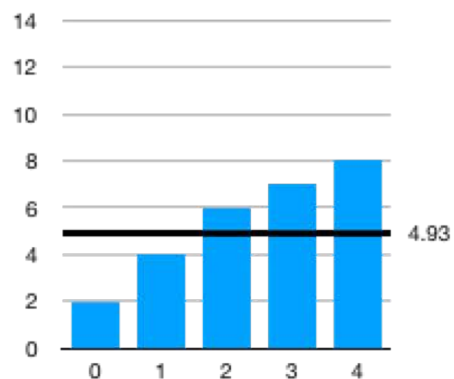
XF Support



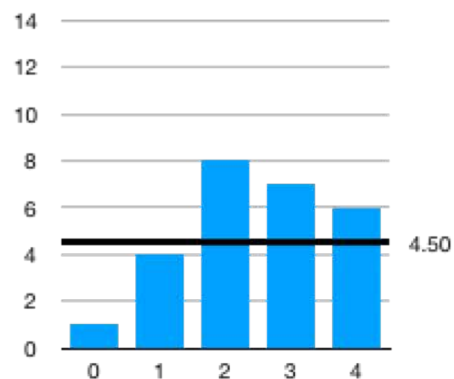
Mixed Rendering



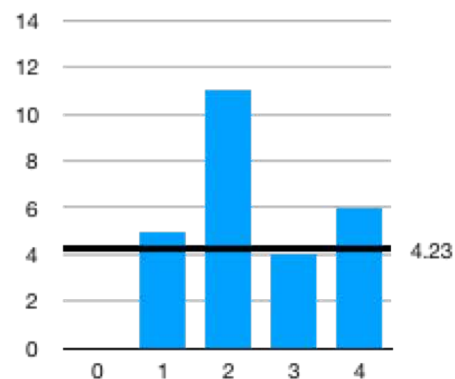
JS Core Components



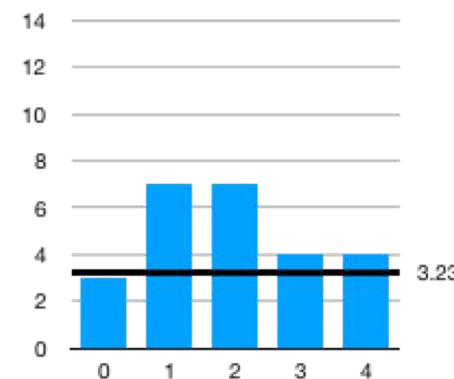
AEM Target Mode



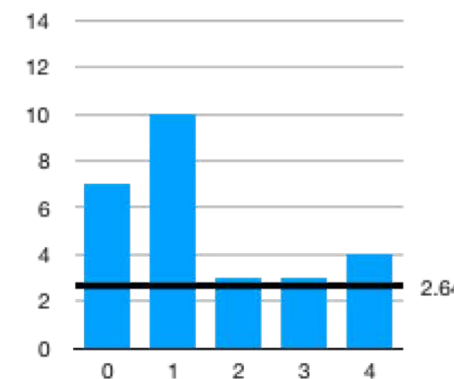
JSON Data Transformation



MSM LiveCopy Support



Vue.js Support





**Adobe**