AMP Bind: Design Doc

Published on GitHub

Authors: willchou@ (@choumx)

Last Updated: 2016-12-09

Objective

Background

Overview

Detailed Design

- Installation
  - Default values
- Digest
  - Data flow diagrams
  - Syntax choice details
    - Option A
    - Option B
- DOM mutations
  - Mutation flow diagram
- Expressions
  - Whitelisted functions
- Validation
  - Runtime validation
- XHR state update

Caveats

Scope

Security Considerations
Vulnerability classes
XSS
XSRF

Media queries

Testing Plan

Work Estimates

Document History

Objective

Bind is a new AMP extension that allows elements to mutate in response to user actions or data changes via data binding and simple JS-like expressions.

Final syntax and naming for Bind are still being deliberated. This document describes the implementation of the system that would support either syntax.

Background

- Original design doc by dvoytenko@
- Use case collation by elindley@
- Hackathon PR by aghassemi@
- Syntax proposal by willchou@
- Syntax proposal by dvoytenko@

Overview

Bind is a new AMP extension that will follow the normal extended component life cycle. Once installed, Bind scans the DOM for bindings -- an element may have attributes, CSS classes or textContent bound to a JS-like **expression**.

The **scope** is implicit, mutable document state that Bind expressions may reference. The scope may be initialized with a new AMP component <amp-state>. Changes to the scope happen through user actions, e.g. clicking a <button> or switching slides on an <amp-carousel>.

A **digest** is a periodic evaluation of all binding expressions. Since scope is mutable and expressions can reference the scope, the evaluated result of expressions may change over

time. Bound elements are updated as a result of a digest.

Data flow in Bind is unidirectional, i.e. only one-way bindings. See the data flow diagrams section below for details.

A simple example:

```
<amp-state id="foo">
  <script type="application/json">{ message: "Hello World" }</script>
</amp-state>

<p [text]="foo.message">Placeholder text</p>
```

1. Bind scans the DOM and finds the <p> element's binding to "text" (textContent)
2. During the next digest, Bind reevaluates the expression "foo.message" to "Hello World"
3. Bind queues the UI update for <p> on the next frame
4. During the next frame, the <p> has its "Placeholder text" replaced with "Hello World"

Detailed Design

Installation

On installation, the Bind script will install the <amp-state> component and the Bind service. The <amp-state> component contains a JSON object with arbitrary data, e.g.

```
<amp-state id="product">
  <script type="application/json">
    {
      "colors": {
        "black": {
          "sizes": ["S", "L"]
        }
      }
    }
  </script>
</amp-state>
```

The Bind service will initialize the set of bindings and expressions by scanning the DOM for all elements that contain attributes that conform to the Bind syntax, e.g.

```
<a [text]="foo" [href]="bar" href="hello.html">Hello World</a>
```

Bindings may only mutate an element's textContent, attributes and CSS classes. The set of

bindable attributes is whitelisted per element since attribute mutation may require relayout.

TBD: Which attributes are whitelisted/blacklisted?

Default values

The default values of bound values (attributes, classes or textContent) should match the initial evaluations of their corresponding binding expressions. In the example above, this means that the expression “bar” must evaluate to “hello.html” and “foo” must evaluate to “Hello World”. This avoids unexpected UI changes due to initial data being inconsistent with default element state. Bind will detect and generate runtime warnings for such inconsistencies, but only in development mode. Publishers will be responsible for maintaining consistency in production.

Note: Scanning the DOM and mutating non-descendent elements seems to violates the Extended Component spec and will be an exception to the rule:

- *Not attempt to access or manipulate objects outside of the component's immediate ownership - e.g. elements that are not specified by or children of the component.*

Digest

Digests are triggered by **scope** mutations, but not necessarily invoked immediately following a change for performance reasons.

Digests occurs at most once per frame, and element mutations caused by a digest are applied in a single vsync block.

To avoid a jarring layout change after page load, relayouts are only possible during digests triggered by a user action. Scroll position should also remain steady during relayouts because a change in size in out-of-viewport elements could affect layout of in-viewport elements.

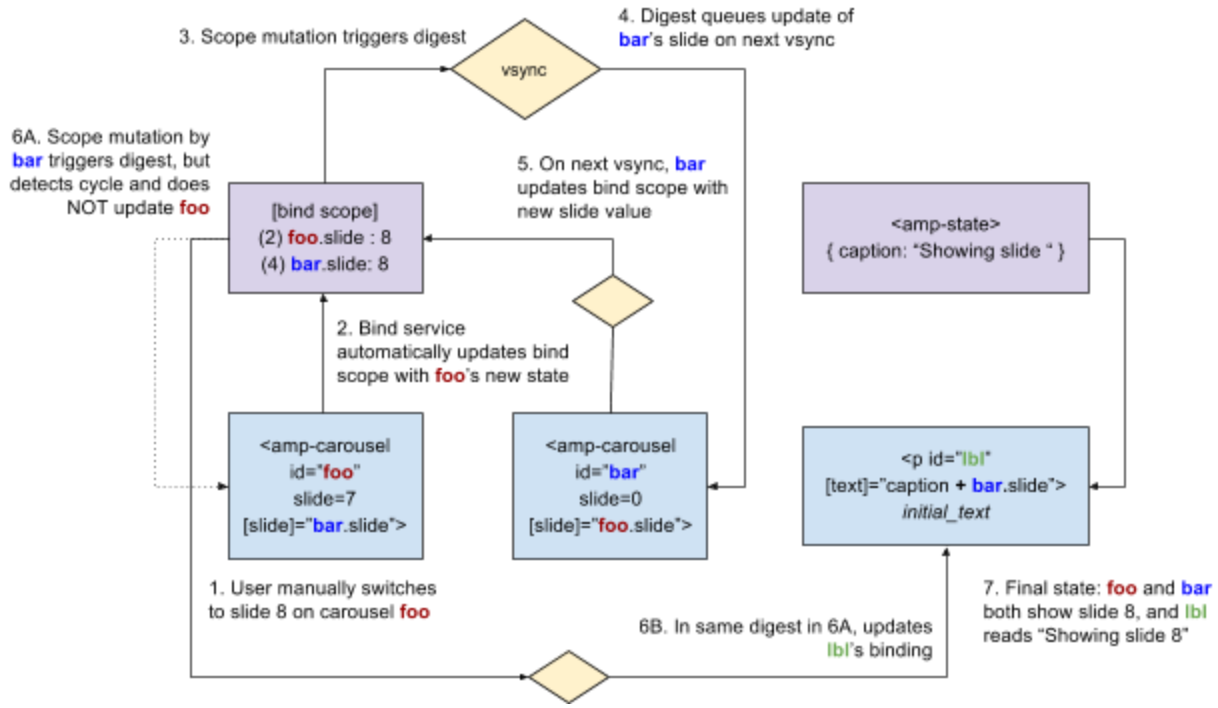
Digests are paused in pauseCallback and resumed in resumeCallback.

If digests turn out to be expensive in practice, a single digest can be amortized over several frames. Digest of in-viewport elements can be prioritized over out-of-viewport elements.

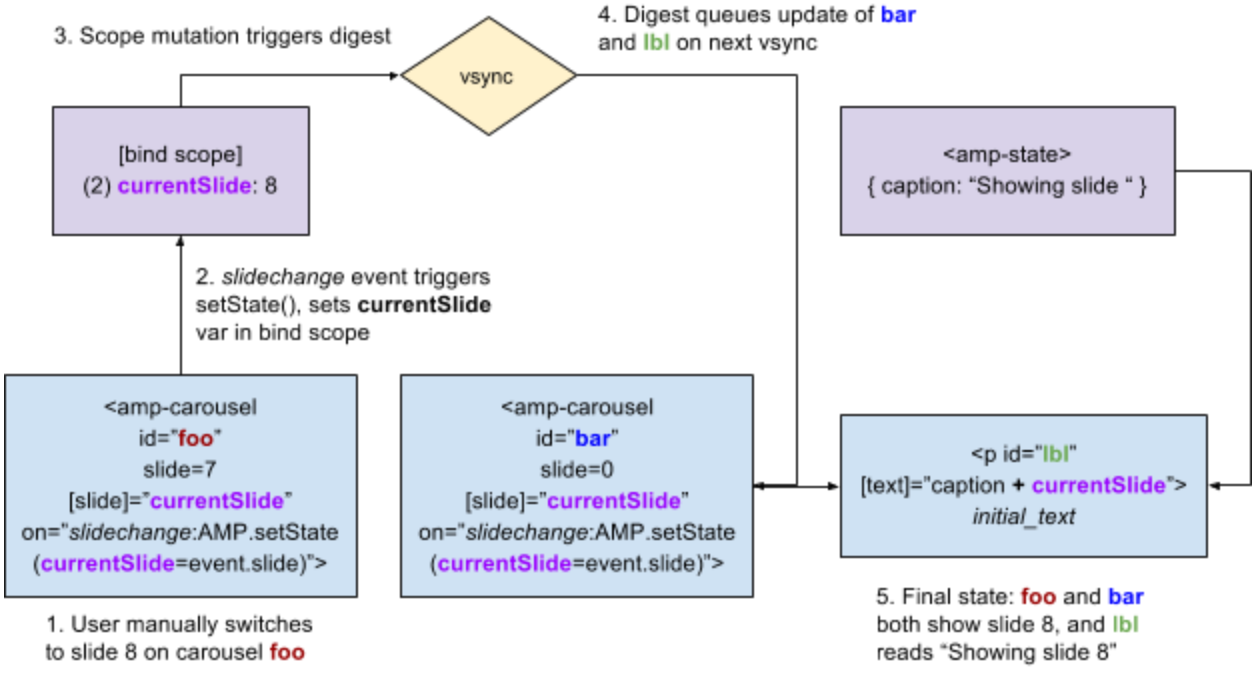
Data flow diagrams

The two diagrams shows data flow from components to/from data for the “dependent carousels” example with a caption displaying the current slide.

Option A: With binding to element state



Option B: With setState() and binding to scope data only



Syntax choice details

Option A

If binding to implicit element state is possible in the choice of syntax, e.g.

```
<p [text]="someElement.someState"></p>
```

Then, Bind will need to initialize the set of **targets** (elements whose state are being bound to) during the first digest cycle. This allows the scope to conservatively determine when it should trigger a digest. This wouldn't be necessary given a different syntax; forcing all scope mutations to occur through events and a setState() call would not require it.

If binding cycles are possible in the choice of syntax, e.g.

```
<amp-carousel id="x" [slide]="y.slide + 1"></amp-carousel>
<amp-carousel id="y" [slide]="x.slide + 1"></amp-carousel>
```

Then, Bind won't traverse the cycle, i.e. Bind will not queue digests due to scope mutations during a digest. This prevents infinite loops and performance problems with long binding chains. The tradeoff is that legitimate binding chains of N depth will take N digests (at least N frames) to update. By contrast, Angular will traverse binding chains up to [10 iterations].

Option B

On the other hand, using setState() syntax avoids cycles entirely since a bind digest cannot cause a state update. Note that this requires the constraint that component events cannot be triggered by digest.

Option B relies on extending the AMP action system in several ways:

- Add AMP.setState() as a valid target and method
- Add event triggers to AMP extensions with relevant event data
- Add support for variables in method arguments to allow accessing of event data

For example:

```
<amp-carousel on="slidechange:AMP.setState(foo=event.slide)">
```

When a user changes slides on the carousel above, AMP.setState() will be called and set the `foo` local state variable to the slide index that was changed to.

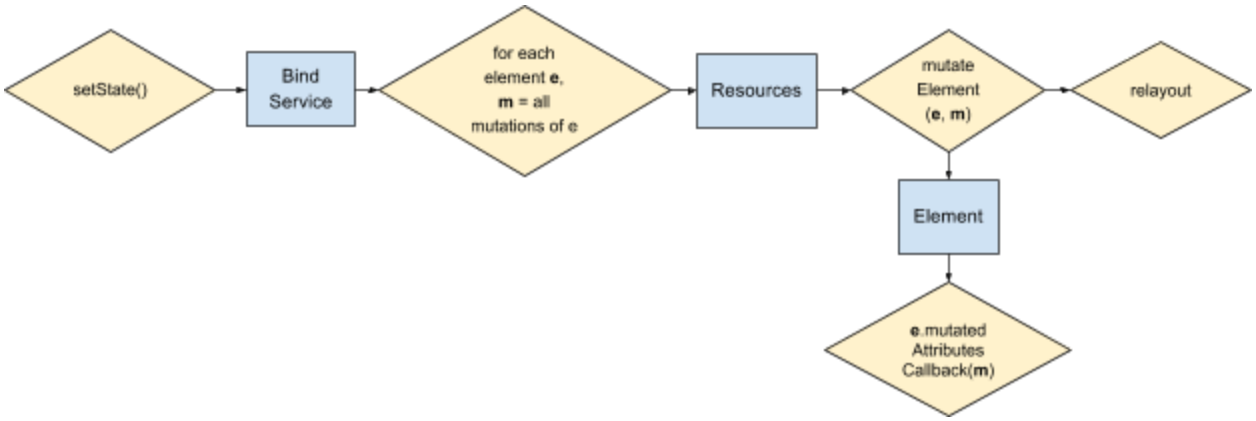
DOM mutations

Applying new expression results to bound elements can affect the layout of neighbor and child elements. For example, dramatically reducing the size of a <p> element's textContent, adding a CSS class to an element containing the rule "display: none" or changing the value of the "height" attribute binding of an AMP element.

Therefore, all elements with bounding rectangles below the topmost element with a changed binding value must relayout. This will happen by calling `Resources#mutateElement()` with the specific mutations for each bound element, which will remeasure and relayout all affected elements on a later frame. A closure containing all mutations to the element will be passed into the `mutator` parameter in `mutateElement()`. Note that N calls to `mutateElement()` in a single frame will only trigger a single relayout.

Mutated AMP elements will perform internal mutations necessary per the bound attribute via a new callback `BaseElement#mutateAttributesCallback()`. For example, an <amp-img> will change its internal <img> element when its [src] binding changes. This callback assumes invocation within a `mutateElement()` call.

Mutation flow diagram



Expressions

Bind expressions are JS-like but with significant constraints. For example, most types of control flow, custom functions, invocation of non-whitelisted functions and multiple expressions are disallowed. These constraints form an effective sandbox that prevents arbitrary JS execution via Bind expressions and discourages developers from writing application logic into Bind.

Undefined vars and properties, and array-index-out-of-bounds returns null instead of throwing errors. This allows more leniency at runtime for unexpected XHR data (Angular does something similar).

EBNF-like grammar for Bind expressions:

```
expression =
        literal
      | identifier
      | unaryOp expression
      | expression binaryOp expression
      | "(" expression ")"
```

```
      | expression "?" expression ":" expression  // Ternary op
      | expression "." identifier                 // Object access
      | expression "[" expression "]"             // Array/object access
      | expression "." function "(" args ")"      // Function call
literal =
        null | true | false | <string> | <number>
      | "[" expression ("," expression)? "]"         // Array literal
      | "{" (keyValuePair ("," keyValuePair)?)? "}"  // Object literal
unaryOp = "-" | "+" | "!"
binaryOp =
        "+" | "-" | "*" | "/" | "%" | "^" | ">" | ">=" | "<" | "<="
      | "&&" | "||"
function = <whitelisted function>
args = expression ("," expression)?
keyValuePair = expression ":" expression
```

Note that keys in object literals can be expressions (unlike JS).

Whitelisted functions

Function invocation is gated on the type of the caller, which is determined via:

```
Object.prototype.toString.call(obj) === '[object <Type>]'
```

Whitelisted functions per object type:

Array	**String**
Array.prototype.concat, Array.prototype.includes, Array.prototype.indexOf, Array.prototype.join, Array.prototype.lastIndexOf, Array.prototype.slice,	String.prototype.charAt, String.prototype.charCodeAt, String.prototype.concat, String.prototype.includes, String.prototype.indexOf, String.prototype.lastIndexOf, String.prototype.repeat, String.prototype.slice, String.prototype.split, String.prototype.substr, String.prototype.substring, String.prototype.toLowerCase, String.prototype.toUpperCase,

Validation

Since Bind has activation overhead, bound attributes can't substitute default attributes in a valid AMP document. I.e. Stripping all bindings from a document should yield valid AMP.

The AMP validator will need to be updated to permit attributes that conform to the chosen bind syntax with arbitrary values. For example, for every whitelisted attribute “attr” allowed in the validator, also whitelist the attribute “[attr]”. This will forbid binding to banned attributes like “onload”.

Runtime validation

Bind allows runtime modification of an AMP document, which means it allows circumvention of the AMP validator (which is performed a priori). For security reasons, and to prevent invalid AMPs from “hiding” within Bind, Bind itself will need to implement validation rules during evaluation of Bind expressions. For example, Bind’s interpreter must block binding to banned attributes (e.g. “onload”) as well as ignore expression evaluations that have security implications, e.g. “javascript:” in src tags.

Ideally, Bind’s runtime sanitizer would use the same implementation as AMP validator. We have a couple options:

1. Reuse AMP validator’s implementation. This avoids code dupe and maintains a consistent definition of “valid AMP” in Bind.
2. Roll our own sanitation. This is similar to amp-template’s approach.

Unfortunately, early experimentation suggests that (1) is too slow for runtime execution[1].

A. Generating the complete validation spec takes **17ms**[2] and generating the lookup table for tags and attribute specs takes an additional **33ms**.
B. Validating a single URL attribute takes **3ms**, with URL parsing taking about 50% of that time. By contrast, copying and running the core logic from the validator’s implementation takes less than **0.3ms**.

(A) may be eventually feasible since it’s a one-time cost that can be amortized across multiple frames. Additionally, restricting the rules to the subset that Bind cares about and generating the lookup table at compile-time could reduce the cost to O(milliseconds).

Current plan is to go with (B). To keep it fast, we’ll only implement the most important validation rules with respect to security and user experience. The downside is a larger maintenance cost due to duplication of logic and, relatedly, a somewhat larger security risk.

XHR state update

To support use cases like “smart buttons”, updates to <amp-state> must be possible via XHR.

---

[1] To test this, I imported the validator rules created in `validator/dist/validator-generated.js` and created a lookup table that mapped tag and attribute names to spec objects. During a digest, I passed the appropriate spec object into “ParsedAttrSpec” from `validator/engine/validator.js`.
[2] Profiled in Chrome on my 2015 MBP with 2x CPU throttling.

To do this, we can extend the functionality of <amp-form> to allow passing of JSON response data to <amp-state> via actions. This can be simply done by passing the response data as an event in the action invocation and implementing the corresponding action handler in the amp-state component.

Caveats

- Preliminary analysis indicates that DOM scan should be performant given typical element/attribute counts on popular e-commerce and news publisher sites, but need to verify this with actual implementation
- Requiring Bind expressions to prevent arbitrary JS injection is a strong security guarantee that may have performance and maintenance costs
- Current design was targeted to accommodate use cases in this document. Actual publisher needs may vary

Scope

- To focus on delivery of MVP, Bind in A4A is out of scope for now

Security Considerations

Vulnerability classes

XSS

Classic JS-injection XSS is not possible in Bind since it only mutates the DOM via Element.attributes, Element.classList and Node.textContent:

- Element.classList and Node.textContent don't allow arbitrary script execution by default
- Event Attributes are blacklisted from binding (e.g. "onload")
- There is no way to bind Element.innerHTML (e.g. injecting <script>)
- "javascript:" etc. are blocked from evaluated bound `href` values via a protocol whitelist

XSS via template expressions is more interesting. For example, Angular's expression sandbox has a long history of escape vectors. Angular 1.6+ removed the sandbox and instead advises developers to avoid rendering user input in server-generated Angular templates (or use ngNonBindable).

Compared to Angular, Bind's featureset is much more constrained and expressions can be sandboxed to prevent arbitrary script execution:

- Expressions are narrowly scoped and do not have access to the global namespace
    - Scopes only contain JSON data corresponding to <amp-state>, local or element state
- JS operators with side-effects are disallowed, e.g. assignment, `new`, i++, etc.
- Function declarations are disallowed
- Only whitelisted, non-mutating functions may be invoked, e.g. Array.indexOf and NOT Array.splice
- Property access is disallowed if Object.prototype.hasOwnProperty.call(obj, prop) === false
    - Prevents access to built-in properties like `__proto__`, `constructor`, etc.
    - Similar to eslint's no-prototype-builtins rule
- Disallow plain JS objects in function arguments to prevent functions from being confused by strange objects (e.g. standard built-in mimics)

Denial of service XSS (e.g. `while(1)`) should not be possible since control flow statements are disallowed.

XSRF

Bind does not add new methods for sending server requests from an AMP page and thus does not introduce new XSRF vectors.

- Any XHR that Bind interacts with should follow the existing AMP CORS guidelines
- <amp-state> avoids JSON hijacking (XSSI) by requiring top-level JSON data to be an object, not an array (similar to <amp-list>)

molnarg@: Is there a way to infer AMP validity at runtime? This would reduce CSP bypass on non-AMP pages due to a Bind sandbox escape.

Media queries

dvoytenko@: Bind protocol allows to change attributes of an element based on a set of expressions and internal model state. This is a deferred procedure since (a) runtime initialization is expensive; and (b) `amp-bind` extension takes time to download.

We also have a family of tools to update style based on media queries, such as `media`, `sizes`, `heights`, etc.

There are some additional requests to be able to update attributes based on the responsive media queries. E.g. a carousel can show several slides at a time on wider screens; a sidebar could start opened on a wide screen; etc.

The idea is to combine media expressions and amp-bind.

```
<amp-sidebar>
  <amp-set
    attr="open"
    media="(min-width:800px)"
    default-value="true"
    expr="..."
    ></amp-set>
</amp-sidebar>
```

In this system, `amp-set` (or whichever name) will be a built-in. The system will be able to executed `media` as needed. But `expr` will be deferred to Bind protocol.

Testing Plan

A thorough suite of unit tests will be needed to make sure that the expression parser enforces all of the constraints it specifies, e.g. access to Function constructor, prototypes of builtins, etc.

Work Estimates

Task	Estimate
Syntax and design review	3 weeks
Bind component/service implementation	2 weeks
Parser implementation and unit tests	2 weeks
Adding binding support to existing AMP components	2 weeks
Work with select publisher to adopt	Variable

Document History

Date	Author	Description	Reviewed by	Signed off by
10/27/2016	willchou@	First draft.		
12/09/2016	willchou@	Added validation experiment results and suggested approach.		
12/20/2016	willchou@	Added “DOM mutations” section.		

12/27/2016	willchou@	Added “XHR state update” section.		