This document is deprecated; please see the version on GitHub instead.

(We'll try to keep this doc up to date, but please refer to the source code as source of truth)

Building an AMP Extension-(Deprecated)

author: mkhatib
last updated: *Oct 6, 2016*
send feedback: Leave a comment or file an issue.

A word on contributing

We suggest opening an "Intent to Implement" GitHub issue for your extension as early as you can, so we can advise on next steps or provide early feedback on the implementation or naming. See CONTRIBUTING.md for more details.

Getting Started

AMP can be extended to allow more functionality and components through building open source extensions (aka custom elements). For example, AMP provides `amp-carousel`, `amp-sidebar` and `amp-access` as extensions. If you'd like to add an extension to support your company video player, rich embed or just a general UI component like a star rating viewer, you'd do this by building an extension.

Naming

All AMP extensions (and built-in elements) have their tag names prefixed with `amp-`. Make sure to choose a proper clear name for your extension. For example, video players are also suffixed with `-player` (e.g. amp-brid-player).

Directory Structure

You create your extensions' files inside the `extensions/` directory. The directory structure is below:

- extensions/amp-my-element/

    - 0.1/
        - test/
            - **test-amp-my-element.js** (Unit test suite for your element)
            - More test JS files (Optional)
        - **amp-my-element.js** (Contains your element's implementation)
        - **amp-my-element.css** (Optional)
        - More JS files (Optional)
        - validator-amp-my-element.protoascii (Validator rules)
    - **amp-my-element.md** (Main usage documentation for your element)
    - More documentation in .md files (Optional)

In most cases you'll only create the bolded files. If your element does not need custom CSS you don't need to create the CSS file.

Extend AMP.BaseElement

Almost all AMP extensions extend AMP.BaseElement, which provides some hookups and callbacks for you to override in order to implement and customize your element behavior. These callbacks are explained below in the BaseElement Callbacks section, and are also explained inline in the BaseElement class.

Element Class

The following shows the overall structure of your element implementation file (extensions/amp-my-element/0.1/amp-my-element.js).

```
import {func1, func2} from '../src/module';
import {CSS} from '../../../build/amp-my-element-0.1.css';
// more ES2015-style import statements.

/** @const */
const EXPERIMENT = 'amp-my-element';

/** @const */
const TAG = 'amp-my-element';

class AmpMyElement extends AMP.BaseElement {

  /** @param {!AmpElement} element */
  constructor(element) {
    super(element);

    // Declare instance variables with type annotations.
  }

  /** @override */
  isLayoutSupported(layout) {
     return layout == LAYOUT.FIXED;
  }
```

```
  /** @override */
  buildCallback() {
    // Get attributes, assertions of values, assign instance variables.
    // Build lightweight DOM and append to this.element.
  }

  /** @override */
  layoutCallback() {
    // Actually load your resource or render more expensive resources.
  }
}

AMP.registerElement('amp-my-element', AmpMyElement, CSS);
```

BaseElement Callbacks

upgradeCallback

Default: Does nothing
Override: Rarely.
Vsync Context: None
Usage: If your extension provides different implementations depending on a late runtime condition (e.g. type attribute on the element, platform)
Example Usage: amp-ad, amp-app-banner

buildCallback

Default: Does nothing
Override: Almost always
Vsync Context: Mutate
Usage: If your element has UI elements this is where you should create your DOM structure and append it to the element. You can also read the attributes (e.g. width, height…) the user provided on your element in this callback.
Warning: Don't load remote resources during the buildCallback. This only circumvents the AMP resources manager, but it will also lead to higher data charges for users because all these resources will be loaded before layouting needs to happen.
Warning 2: Do the least needed work here, and don't build DOM that is not needed at this point.

preconnectCallback

Default: Does nothing.
Vsync Context: None (Neither mutate nor measure)
Override: Sometimes, if your element will be loading remote resources.
Usage: Use to instruct AMP which hosts to preconnect to and which resources to preload/prefetch this allows AMP to delegate to the browser to get a performance boost by preconnecting, preloading and prefetching resources via preconnect service.
Example Usage: Instagram uses this to preconnect to instagram hosts.

createPlaceholderCallback

Default: Does nothing.
Vsync Context: Mutate
Override: Sometimes. If your component provides a way to dynamically create a lightweight placeholder. This gets called only if the element doesn't already have a publisher-provided placeholder (through the placeholder attribute).
Usage: Create placeholder DOM and return it. For example, amp-instagram uses this to create a placeholder dynamically by creating an amp-img placeholder instead of loading the iframe, leaving the iframe loading to layoutCallback.
Warning: Only use amp-elements for creating placeholders that require external resource loading. This allows runtime to create this early but still defer the resource loading and management to AMP resources manager. Don't create or load heavyweight resources (e.g. iframe…).
Example Usage: amp-instagram.

onLayoutMeasure

Default: Does nothing.
Vsync Context: Measure.
Override: Rarely.
Usage: Use to measure layouts for your element.
Example Usage: amp-iframe

layoutCallback

Default: Does nothing.
Vsync Context: Mutate
Override: Almost always.
Usage: Use this to actually render the final version of your element. If the element should load a video, this is where you load the video. This needs to return a promise that resolves when the element is considered "laid out" - usually this means load event has fired but can be different from element to element. Note that load events usually are fired very early so if there's another event that your element can listen to that have a better meaning of ready-ness, use that to resolve your promise instead - for example: amp-youtube uses the *playerready* event that the underlying YT Player iframe sends to resolve the layoutCallback promise.

firstLayoutCompleted

Default: Hide element's placeholder.
Vsync Context: Mutate
Override: Sometimes. If you'd like to override default behavior and not hide the placeholder when the element is considered first laid out. Sometimes you wanna control when to hide the placeholder.
Example Usage: amp-anim

pauseCallback

Default: Does nothing.
Vsync Context: Mutate
Called: When you swipe away from a document in a viewer. Called on children of lightbox when you close a lightbox instance, called on carousel children when the slide is not the active slide. And possibly other places.
Override: Sometimes. Most likely if you're building a player.
Usage: Use to pause video, slideshow auto-advance...etc
Example Usage: amp-video, amp-youtube

resumeCallback

Default: Does nothing.
Vsync Context: Mutate
Override: Sometimes.
Usage: Use to restart the slideshow auto-advance.
Note: This is not used widely yet because it's not possible to resume video playback for example on mobile.

unlayoutOnPause

Default: Returns false.
Vsync Context: Mutate
Override: If your element doesn't provide a pausing mechanism, instead override this to unlayout the element when AMP tries to pause it.
Return: True if you want unlayoutCallback to be called when paused.
Usage Example: amp-brightcove

unlayoutCallback

Default: Does nothing.
Vsync Context: Mutate
Override: Sometimes.
Usage: Use to remove and unload heavyweight resources like iframes, video, audio and others that your element has created.
Return: **True** if your element need to re-layout.
Usage Example: amp-iframe

viewportCallback

Default: Does nothing.
Override: Rarely.
Usage: Use if your element need to know when it comes into viewport and when it goes out of it for finer control.

Usage Example: amp-carousel, amp-anim

Element Styling

You can write a stylesheet to style your element to provide a minimal visual appeal, your element structure should account for whether you want users (publishers and developers using your element) to customize the default styling you're providing and allow for easy CSS classes and/or well-structure DOM elements.

Element styles are loaded when the element script itself is included in an AMP doc. You tell AMP which CSS belongs to this element when registering the element, see next.

Class names prefixed with `-amp-` are considered private and publishers are not allowed to use to customize (enforced by AMP validator).

Register Element with AMP

Once you have implemented your AMP element, you need to register it with AMP, all AMP extensions are prefixed with `amp-`. This is where you tell AMP which class to use for this tag name and which CSS to load.

```
AMP.registerElement('amp-carousel', CarouselSelector, CSS);
```

Element Usage Documentation and Tests

Make sure to write pretty comprehensive unit tests for your element. Also don't forget to write an .md file documenting how to use this element along with its attributes requirements and examples.

An extra step that you can do is to write a small AMP doc inside examples/ directory to provide easy examples for users to preview and for developers to manually test and visually inspect your element.

Actions and Events

AMP provides a framework for elements to fire their own events to allow users of that element to listen and react to the events. For example, amp-form extension fires a few events on <form> elements like `submit-success`. This allow publishers to listen to that event and react to it, for example, by launching a lightbox to display a message.

The other part of the event-system in AMP is actions. When listening to an event on an element usually you'd like to trigger an action (possibly on other elements). For example, in the example above, the publisher is executing the `open` action on `lightbox`.

The syntax for using this on elements is as follow:

```
<form on="submit-success:my-success-lightbox.open;submit-error:my-error-lightbox.open">
</form>
```

To fire events on your element use AMP's action service and the `.trigger` method.

```
actionServiceForDoc(doc.documentElement).trigger(this.form_, 'submit-success', null);
```

And to expose actions use `registerAction` method that your element inherits from `BaseElement`.

```
this.registerAction('close', this.close.bind(this));
```

Your element could also choose to override the `activate` method inherited from BaseElement that would define the default action for your element. For example amp-lightbox overrides activate to define the open default case.

Make sure your element documentation documents the events and actions it exposes.

Sub-elements Ownership

AMP elements are usually discovered and scheduled by the AMP runtime automatically and managed through Resources. In some cases an AMP element might want to control and own when its sub-elements gets scheduled and not leave that to the AMP runtime. An example to this is the <amp-carousel>, where it wants to schedule preloading/pre-rendering or layouting of its cells based on the window the user is in.

AMP provides a way for an element to control this by setting the owner on the element you want to control. For carousel example, carousel loops over all its elements and sets itself as the owner of these elements. AMP runtime will not manage scheduling layouting for elements that have owners.

```
this.cells_ = this.getRealChildren();

this.cells_.forEach(cell => {
  this.setAsOwner(cell);
  cell.style.display = 'inline-block';
  this.container_.appendChild(cell);
```

```
});
```

An element can then later call *schedulePreload* or *scheduleLayout* to schedule preload or layout respectively. For example, <amp-carousel type=slider> (Slider instance of amp-carousel) calls *schedulePreload* for the next/previous slide when the user moves forward/backward in the slides and then calls *scheduleLayout* for the current slide when the user moves to it.

```
this.updateInViewport(oldSlide, false);
this.updateInViewport(newSlide, true);
this.scheduleLayout(newSlide);
this.setControlsState();
this.schedulePause(oldSlide);
this.schedulePreload(nextSlide);
```

It's important to understand that the parent/owner element is responsible for managing all of its children (except for placeholders, see below). This means you need to make sure your element updates whether the child is in viewport and when to schedule different phases for the element.

Nested Sub-elements

Your element should anticipate its sub-elements to nest some more amp-elements and schedule preload or layout for these as well, otherwise the element will never be preloaded or laid out. This is true to all nested amp-elements that are not placeholders. AMP runtime will schedule nested amp-elements that are placeholders.

```
<amp-carousel> ← Parent element
  <amp-figure>  ← Parent needs to schedule this element
    <amp-img placeholder></amp-img> ← AMP will schedule this when amp-figure is scheduled
    <amp-img></amp-img>  ← Parent needs to schedule this element
    <amp-fit-text></amp-fit-text> ← Parent needs to schedule this element
  </amp-figure>
</amp-carousel>
```

Allowing Proper Validations

One of AMP's features is that a document can be checked against validation rules to confirm it's AMP-valid. When you implement your element, AMP validator needs to be updated to add rules for your element to keep documents using your element valid. In order to do that you need to file an issue on the github repo select "Related to: Validator" and mention what rules the validator needs to validate. This usually includes

- Your element tag-name
- Required attributes for the element
- Specific values that an attribute accept (e.g. myattr="TYPE1|TYPE2")

- Layouts your element supports (see Layout specs and Layouts Supported in your Element)
- If there are restrictions where your element can or can't appear (e.g. disallowed_ancestory, mandatory_parent...)

For more details take a look at Contributing Component Validator Rules.

Performance Considerations

Pre-rendering and Placeholders

Another enabling feature of instant-web in AMP is support for prerendering in a way that does not consume loads of data and does not waste too much of the user's device resources. AMP does this by strictly controlling resource loading and rendering.

If your extension is lightweight, it might be worth enabling pre-rendering of your elements so that users will be able to see it appear instantly when they click on an article.

Sometimes fully pre-rendering the element isn't an option because it is heavyweight. Your element might want to opt into creating a dynamic placeholder for itself (in case a placeholder wasn't provided by the developer/publisher who is using your element). This allows elements to display content as fast as possible and allow prerendering that placeholder. Learn more about placeholder elements.

NOTE: Make sure not to request external resources in the pre-render phase. Requests to the publisher's origin itself are OK. If in doubt, please flag this in review.

AMP will automatically call your element's `createPlaceholderCallback` during build step if it didn't detect a placeholder was provided. This allows you to create your own placeholder. Here's an example of how `amp-instagram` element used this callback to create a dynamic placeholder of an `amp-img` element to avoid loading the heavyweight instagram iframe embed during pre-rendering and instead loads just the image directly from instagram media endpoint.

```
class AmpInstagram extends AMP.BaseElement {
  // ...
  /** @override */
  createPlaceholderCallback() {
    const placeholder = this.getWin().document.createElement('div');
    placeholder.setAttribute('placeholder', '');
    const image = this.getWin().document.createElement('amp-img');
    // This is always the same URL that is actually used inside of the embed.
    // This lets us avoid loading the image twice and make use of browser cache.
    image.setAttribute('src', 'https://www.instagram.com/p/' +
        encodeURIComponent(this.shortcode_) + '/media/?size=l');
    image.setAttribute('width', this.element.getAttribute('width'));
    image.setAttribute('height', this.element.getAttribute('height'));
```

```
    image.setAttribute('layout', 'responsive');
    setStyles(image, {
      'object-fit': 'cover',
    });
    const wrapper = this.element.ownerDocument.createElement('wrapper');
    // This makes the non-iframe image appear in the exact same spot
    // where it will be inside of the iframe.
    setStyles(wrapper, {
      'position': 'absolute',
      'top': '48px',
      'bottom': '48px',
      'left': '8px',
      'right': '8px',
    });
    wrapper.appendChild(image);
    placeholder.appendChild(wrapper);
    this.applyFillContent(image);
    return placeholder;
  }
  // …
}
```

Important: One thing to keep in mind is that when you create a placeholder, use the amp-provided elements when loading external resources. This is most likely going to be `amp-img` like in the instagram case above. This still allows AMP resource manager to control when these resources get loaded and rendered as oppose to using the HTML-native `img` tag which will be out of AMP resource management.

Loading Indicators

Consider showing a loading indicator if your element is expected to take a long time to load (for example, loading a GIF, video or iframe). AMP has a built-in mechanism to show a loading indicator simply by whitelisting your element to show it. You can do that inside layout.js file in the LOADING_ELEMENTS_ object.

```
Export const LOADING_ELEMENTS_ = {
  ...
  'AMP-YOUTUBE’: true,
  'AMP-MY-ELEMENT': true,
}
```

Destroying Heavyweight Resources

To stay good to our promise of lowering resources usage especially on mobile, elements that create and load heavyweight resources (e.g. iframes, video, very large images, an expensive timer or computation...) need to be destroyed when they are no longer needed.

AMP signals to your element that it needs to do that with `unlayoutCallback`. AMP calls this when the document becomes inactive; like when the user swipes away from the document to another one or when they switch tabs.

This might be also be called in special cases like if your element is used as an amp-carousel cell and it was swiped away to become outside the viewport. This will only happen if your element sets `unlayoutOnPause`. Carousel by default only pauses the elements that are outside its viewport.

Here's an example of how `amp-instagram` destroys the iframe it has embedded when `unlayoutCallback` is called.

```
/** @override */
unlayoutCallback() {
  if (this.iframe_) {
    removeElement(this.iframe_);
    this.iframe_ = null;
    this.iframePromise_ = null;
    setStyles(this.placeholderWrapper_, {
      'display': '',
    });
  }
  return true;  // Call layoutCallback again.
}
```

Note if your element unlayoutCallback destroys the resources, it probably wants to return true in order to signal to AMP the need to call `layoutCallback` again once the document is active. Otherwise your element will never be re-laid out.

vsync, mutateElement, deferMutate and changeSize

AMP provides multiple utilities to optimize many mutations and measuring for better performance. These include vsync service with a mutate and measure utility method that will synchronize all measuring happening in short period of time together and then do all the mutating in a requestAnimationFrame or similar cycles.

Loading External Resources

If your extension needs to load external resources (like an sdk) then you might need to add proper third party integration for it to work and use the proper third party iframe. Loading external resources is only allowed inside a 3p iframe which AMP serves on a different domain for security and performance reasons. Take a look at adding <amp-facebook> extension PR for examples of 3p integration.

Read about Inclusion of third party software, embeds and services into AMP.

For contrast, take a look at amp-instagram which does NOT require an SDK to be loaded in order to embed a post, instead it provides an iframe-based embedding allowing amp-instagram

extension to use a normal iframe with no 3p integration needed, similarly, amp-youtube and others.

Layouts Supported in your Element

AMP defines different layouts that elements can choose whether or not to support Your element needs to announce which layouts it supports through overriding the `isLayoutSupported(layout)` callback and returning true if the element supports that layout. Read more about AMP Layout System and Layout Types.

What Layout should your Element Support?

After understanding each layout type, if it makes sense, support all of them. Otherwise choose what makes sense to your element. A popular support choice is to support size-defined layouts (Fixed, Fixed Height, Responsive and Fill) through using the utility `isLayoutSizeDefined` in `layout.js`.

For example, `amp-pixel` only supports fixed layout.

```
class AmpPixel extends BaseElement {
  /** @override */
  isLayoutSupported(layout) {
    return layout == Layout.FIXED;
  }
}
```

While `amp-carousel` supports all size-defined layouts.

```
class AmpSlides extends AMP.BaseElement {
  /** @override */
  isLayoutSupported(layout) {
    return isLayoutSizeDefined(layout);
  }
}
```

Experiments

Most newly created elements are initially launched as experiments. This allows people to experiment with using the new element and provide the author(s) with feedback. It also provides the AMP Team with the opportunity to monitor for any potential errors. This is especially required if the validator hasn't been updated yet to allow your newly created extension, otherwise people using it in production will invalidate all their AMP documents.

You can add your extension as an experiment in the `amphtml/tools/experiments` file by adding a record for your extension in EXPERIMENTS variable.

```
/** @const {!Array<!ExperimentDef>} */
```

```
const EXPERIMENTS = [
  // ...
  {
    id: 'amp-my-element',
    name: 'AMP My Element',
    spec: 'https://github.com/ampproject/amphtml/blob/master/extensions/' +
        'amp-my-element/amp-my-element.md',
    cleanupIssue: 'https://github.com/ampproject/amphtml/issues/XXXYYY',
  },
  // ...
];
```

And then protecting your code with a check `isExperimentOn(win, ‘amp-my-element’)` and only execute your code when it is on.

```
import {isExperimentOn} from ‘../src/experiments;

/** @const */
const EXPERIMENT = ‘amp-my-element’;

/** @const */
const TAG = ‘amp-my-element’;

Class AmpMyElement extends AMP.BaseElement {

  /** @param {!AmpElement} element */
  constructor(element) {
    super(element);

    // declare instance variables with type annotations.
  }

  /** @override */
  isLayoutSupported(layout) {
     return layout == LAYOUT.FIXED;
  }

  /** @override */
  buildCallback() {
    if (!isExperimentOn(this.getWin(), EXPERIMENT)) {
      user.warn('Experiment %s is not turned on.', EXPERIMENT);
      return;
    }
    // get attributes, assertions of values, assign instance variables.
    // build lightweight dom and append to this.element.
  }

  /** @override */
  layoutCallback() {
    if (!isExperimentOn(this.getWin(), EXPERIMENT)) {
      user.warn('Experiment %s is not turned on.', EXPERIMENT);
      return;
    }
    // actually load your resource or render more expensive resources.
  }
}
```

```
AMP.registerElement(‘amp-my-element’, AmpMyElement, CSS);
```

Enable Experiment

Users wanting to experiment with your element can then go to the experiments page and enable your experiment.
If test on localhost, use the command `AMP.toggleExperiment(id, true/false)` to enable.

Example of Using your Extension

This is optional but it greatly helps users to understand and demo how your element works and provides an easy start-point for them to experiment with it. This is basically where you actually build an AMP HTML document and use your element in it by creating a file in the `examples/` directory, usually with the `my-element.amp.html` file name. Browse that directory to see examples for other elements and extensions.

Also consider contributing to ampbyexample.com on github.

Updating Build Configs

In order for your element to build correctly you would need to make few changes to gulpfile.js to tell it about your extension, its files and its examples.

```
...
declareExtension('amp-kaltura-player', '0.1', /* hasCss */ false);
declareExtension('amp-carousel', '0.1', /* hasCss */ true);
...
```

Versioning

AMP runtime is currently in v0 major version. Extensions versions are maintained separately. If your changes to your non-experimental extension makes breaking changes that are not backward compatible you should version your extension. This would usually be by creating a 0.2 directory next to your 0.1.

If your extension is still in experiments breaking changes usually are fine so you can just update the same version.

Unit Tests

Make sure you write good coverage for your code. We require unit tests for all checked in code. We use the following frameworks for testing:

- Mocha, our test framework
- Karma, our tests runner
- Sinon, spies, stubs and mocks.

For faster testing during development, consider using --files argument to only run your extensions' tests.

```
$ gulp test --files=extensions/amp-my-element/0.1/test/test-amp-my-element.js --watch
```

Type Checking

We use Closure Compiler to perform type checking. Please see Annotating JavaScript for the Closure Compiler and existing AMP code for examples of how to add type annotations to your code. The following command should be run to ensure no type violations are introduced by your extension.

```
$ gulp check-types
```

Example PRs

Adding new Ad-provider
Teads
EPlanning
Taboola
Adding Embeds that's not iframe-based (requires JS SDK)
amp-facebook
Adding iframe based embeds
amp-soundcloud
amp-vine
amp-brightcove
Adding non-visual elements
amp-font
amp-install-serviceworker
Adding General UI components
amp-sidebar
amp-image-lightbox
amp-accordion