# How to Use A-Frame

## What is A-Frame

A-Frame is a web framework for building virtual reality (VR) experiences. A-Frame is based on top of HTML, making it simple to get started. But A-Frame is not just a 3D scene graph or a markup language; the core is a powerful entity-component framework that provides a declarative, extensible, and composable structure to [three.js](https://threejs.org/).

**Entity-Component Architecture:** A-Frame is a powerful [three.js](https://threejs.org/) framework, providing a declarative, composable, reusable [entity-component structure](https://aframe.io/docs/1.3.0/introduction/entity-component-system.html). HTML is just the tip of the iceberg; developers have unlimited access to JavaScript, DOM APIs, three.js, WebVR, and WebGL.

## Import

To include A-Frame in an HTML file, we drop a <script> tag pointing to the CDN build:

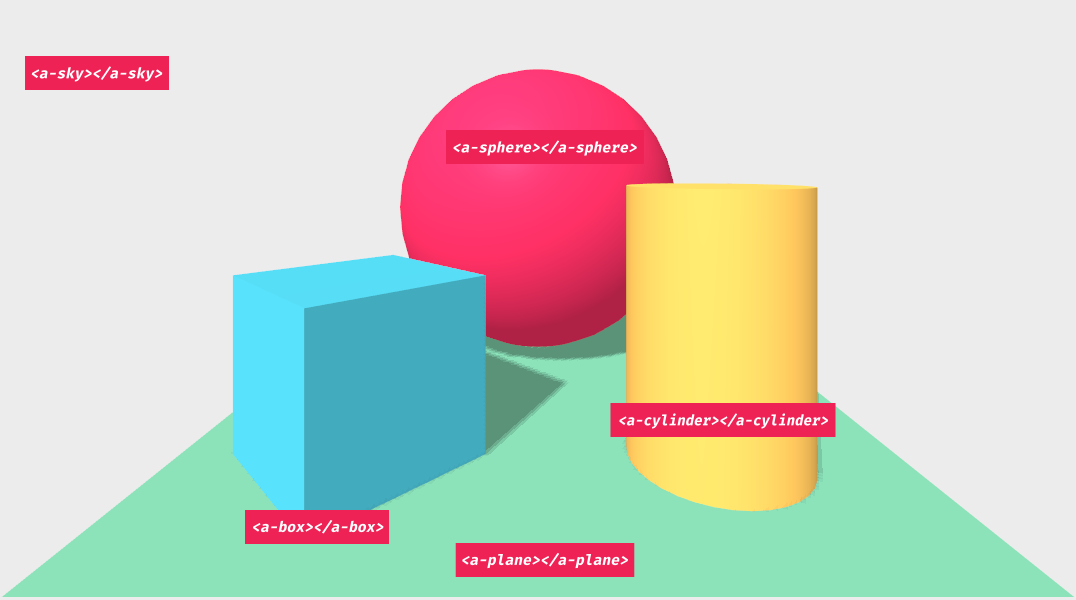
<head>  
 <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>  
</head>

Or install from npm:

$ npm install aframe

## HTML Primitives

A-Frame is based on top of [HTML](https://developer.mozilla.org/docs/Learn/HTML/Introduction_to_HTML/Getting_started) and [the DOM](https://developer.mozilla.org/docs/Web/API/Document_Object_Model) using a polyfill for Custom Elements. HTML is the building block of the Web, providing one of the most accessible computing languages around. There are no installations or build steps required, creating with HTML involves just text in an HTML file and opening the HTML file in a browser. Since most of the Web was built on top of HTML, most existing tools and libraries work with A-Frame including React, Vue.js, Angular, d3.js, and jQuery.



While the HTML layer looks basic, HTML and the DOM are only the outermost abstraction layer of A-Frame. Underneath, A-Frame is an entity-component framework for three.js that is exposed declaratively.

A-Frame provides a handful of elements such as **<a-box>** or **<a-sky>** called primitives that wrap the entity-component pattern to make it appealing for beginners. At the bottom of the documentation navigation sidebar, we can see every primitive that A-Frame provides out of the box. Developers can create their own primitives as well.

## Components

Primitives are just **<a-entity>**s under the hood. This means primitives have the same API as entities such as positioning, rotating, scaling, and attaching components.

For example, attach the physics components via HTML attributes:

<html>  
 <head>  
 <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>  
 <script src="https://unpkg.com/aframe-physics-system@1.4.0/dist/aframe-physics-system.min.js"></script>  
 </head>  
 <body>  
 <a-scene physics>  
 <a-box position="-1 4 -3" rotation="0 45 0" color="#4CC3D9" dynamic-body></a-box>  
 <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#7BC8A4" static-body></a-plane>  
 <a-sky color="#ECECEC"></a-sky>  
 </a-scene>  
 </body>  
</html>

## Create Personalized Primitives

We can register our own primitives (i.e., register an element) using **AFRAME.registerPrimitive(name, definition)**. **name** is a string and must contain a dash (i.e. **'a-foo'**). **definition** is a JavaScript object defining these properties:

| Property | Description | Example |
| --- | --- | --- |
| defaultComponents | Object specifying default components of the primitive. The keys are the components’ names and the values are the components’ default data. | {geometry: {primitive: 'box'}} |
| mappings | Object specifying mapping between HTML attribute name and component property names. Whenever the HTML attribute name is updated, the primitive will update the corresponding component property. The component property is defined using a dot syntax ${componentName}.${propertyName}. | {depth: 'geometry.depth', height: 'geometry.height', width: 'geometry.width'} |

## ECS in A-Frame

A-Frame has APIs that represents each piece of ECS:

* **Entities** are represented by the **<a-entity>** element and prototype.
* **Components** are represented by HTML attributes on **<a-entity>**‘s. Underneath, components are objects containing a schema, lifecycle handlers, and methods. Components are registered via the **AFRAME.registerComponent (name, definition)** API.
* **Systems** are represented by **<a-scene>**‘s HTML attributes. System are similar to components in definition. Systems are registered via the **AFRAME.registerSystem (name, definition)** API.

## A-Frame using JavaScript, Events, DOM

Since A-Frame is just HTML, we can control the scene and its entities using JavaScript and [DOM](https://developer.mozilla.org/docs/Web/API/Document_Object_Model/Introduction) APIs as we mostly would in ordinary web development.

**Important:** Before we go over the different ways to use JavaScript and DOM APIs, we prescribe encapsulating your JavaScript code within [A-Frame components](https://aframe.io/docs/1.4.0/core/component.html). Components modularize code, make logic and behavior visible from HTML, and ensure that code is executed at the correct time (e.g., after the scene and entities have attached and initialized). As the most basic example, to register a **console.log** component before **<a-scene>**:

AFRAME.registerComponent('log', {  
 schema: {type: 'string'},  
  
 init: function () {  
 var stringToLog = this.data;  
 console.log(stringToLog);  
 }  
});

And after the registration, use the component from HTML:

<a-scene log="Hello, Scene!">  
 <a-box log="Hello, Box!"></a-box>  
</a-scene>

Do **not** try to put A-Frame-related JavaScript in a raw **<script>** tag after **<a-scene>** as we would with traditional 2D scripting. If we do, we’d have to take special measures to make sure code runs at the right time (see [Running Content Scripts on the Scene](https://aframe.io/docs/1.4.0/core/scene.html#running-content-scripts-on-the-scene)).

In javascript, you can grab DOM elements from the html. For example, in an html that looks like this:

<html>  
 <a-scene>  
 <a-box id="redBox" class="clickable" color="red"></a-box>  
 <a-sphere class="clickable" color="blue"></a-sphere>  
 <a-box color="green"></a-box>  
 <a-entity light="type: ambient"></a-entity>  
 <a-entity light="type: directional"></a-entity>  
 </a-scene>  
</html>

If we want to grab just one element, we use **.querySelector()** which returns one element. Let’s grab the scene element:

const sceneEl = document.querySelector('a-scene');

Note if we were working within a component, we’d already have a reference to the scene element without needing to query. All entities have reference to their scene element:

AFRAME.registerComponent('foo', {  
 init: function () {  
 console.log(this.el.sceneEl); // Reference to the scene element.  
 }  
});

If an element has an ID, we can use an ID selector (i.e., **#<ID>**). Let’s grab the red box which has an ID. Before we did a query selector on the entire document. Here, we’ll do a query selector just within the scope of the scene. With query selectors, we’re able to limit the scope of the query to within any element:

const sceneEl = document.querySelector('a-scene');  
console.log(sceneEl.querySelector('#redBox'));  
// <a-box id="redBox" class="clickable" color="red"></a-box>

You can grab groups of elements using .querySelectorAll() and then, using iteration, access each element in the group.

#### A Note About Performanc:

Avoid using **.querySelector** and **.querySelectorAll** in **tick** and **tock** functions that get called every frame as it does take some time to loop over the DOM to retrieve entities. Instead, keep a cached list of entities, calling the query selectors beforehand, and then just loop over that.

AFRAME.registerComponent('query-selector-example', {  
 init: function () {  
 this.entities = document.querySelectorAll('.box');  
 },  
   
 tick: function () {  
 // Don't call query selector in here, query beforehand.  
 for (let i = 0; i < this.entities.length; i++) {  
 // Do something with entities.  
 }  
 }  
});

To learn about other native javascript DOM uses, refer to : <https://aframe.io/docs/1.4.0/introduction/javascript-events-dom-apis.html>

## Creating a Modifying an Entity

To create an entity, we can use **document.createElement**. This will give us a blank entity:

const el = document.createElement('a-entity');

However, this entity will not be initialized or be a part of the scene until we attach it to our scene.

To add an entity to the DOM, we can use **.appendChild(element)**. Specifically, we want to add it to our scene. We grab the scene, create the entity, and append the entity to our scene.

const sceneEl = document.querySelector('a-scene');  
const entityEl = document.createElement('a-entity');  
// Do `.setAttribute()`s to initialize the entity.  
sceneEl.appendChild(entityEl);

Note that **.appendChild()** is an asynchronous operation in the browser. Until the entity has finished appending to the DOM, we can’t do many operations on the entity (such as calling **.getAttribute()**). If we need to query an attribute on an entity that has just been appended, we can listen to the **loaded** event on the entity, or place logic in an A-Frame component so that it is executed once it is ready:

var sceneEl = document.querySelector('a-scene');  
  
AFRAME.registerComponent('do-something-once-loaded', {  
 init: function () {  
 // This will be called after the entity has properly attached and loaded.  
 console.log('I am ready!');  
 }  
});  
  
var entityEl = document.createElement('a-entity');  
entityEl.setAttribute('do-something-once-loaded', '');  
sceneEl.appendChild(entityEl);

To **remove** an entity from the DOM and thus from the scene, we call **.removeChild(element)** from the parent element. If we have an entity, we must ask its parent (**parentNode**) to remove the entity.

entityEl.parentNode.removeChild(entityEl);