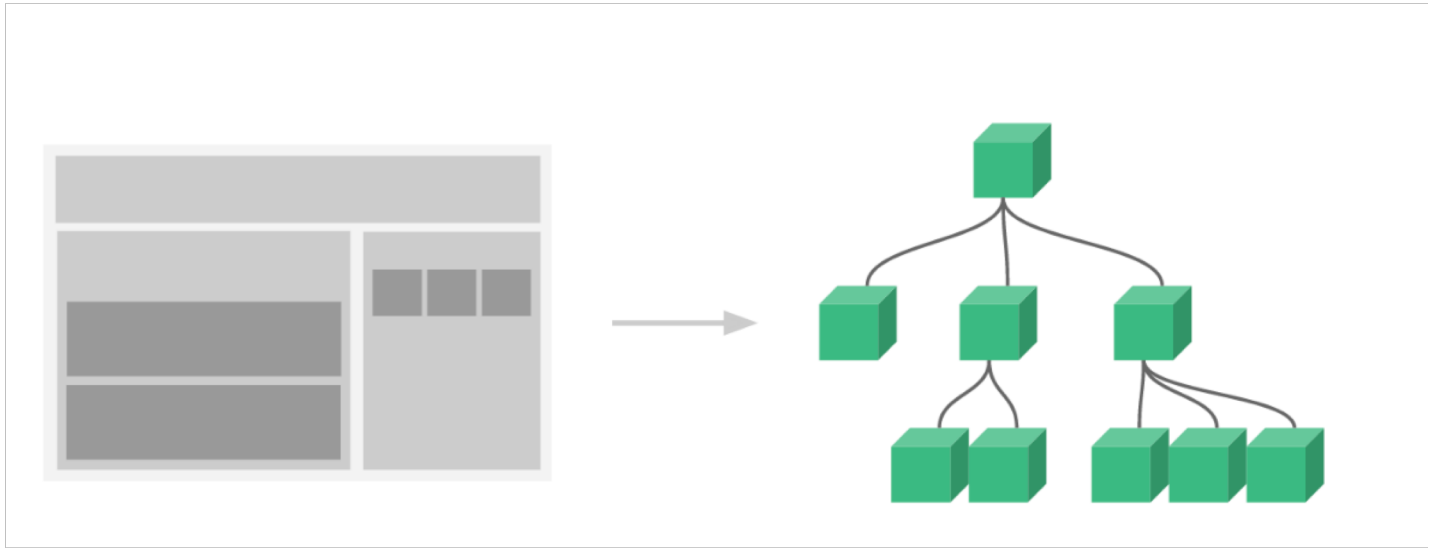
Module 2

Vue Components

Vue components

- It’s common for an app to be organized into a **tree of nested components:**
  - For example, you might have components for a header, sidebar, and content area, each typically containing other components for navigation links, blog posts, etc.



Vue components

- Components are reusable Vue instances with a name: in this case, <button-counter>.
- We can use this component **as a custom element inside a root Vue instance** created with new Vue
- The data is private to the component
- We cannot directly access any parent data in a child component
- Each time you use a component (<button-counter>)**, a new instance of it is created**, so each button-counter has a individual count value

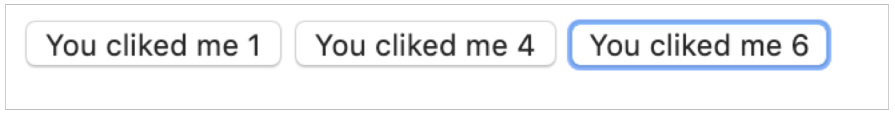
```
Vue.component('button-counter',{
    data: function (){
        return {
          count:0
        }
    },
    template: '<button v-on:click="count++">You cliked me {{ count }}</button>'
  }
)
```

```
<div id="components-demo">
  <button-counter></button-counter>
</div>
```

```
new Vue({ el: '#components-demo' })
```

Vue components

- **A component's data option must be a function**, so that each instance can maintain an independent copy of the returned data object:



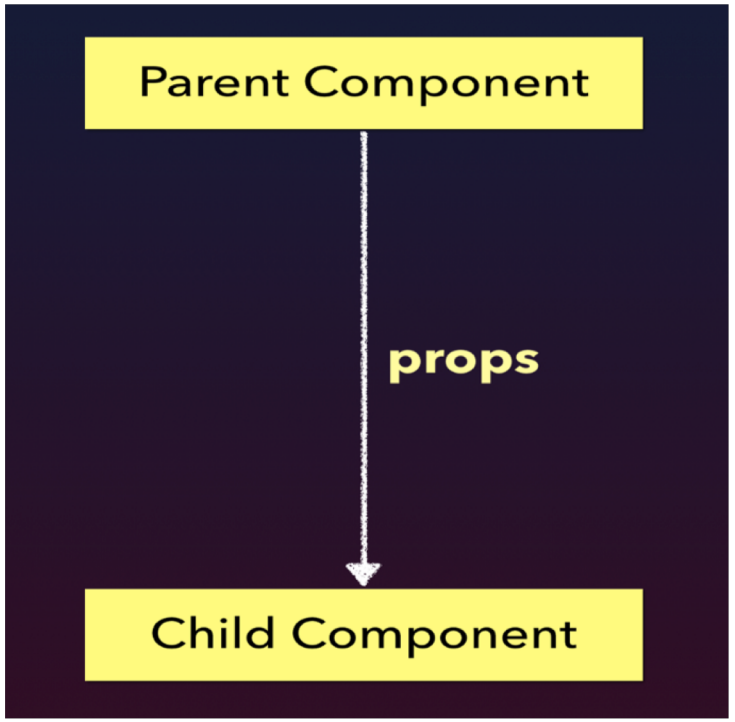
- If the component code in the data section is not a function:

```
Vue.component('button-counter',{
    data: {
        count:0
    },
    template: '<button v-on:click="
})
```

```
[Vue warn]: The "data" option should be a function that returns a per-instance value in component definitions.  vue.js:634

[Vue warn]: Property or method "count" is not defined on the instance but referenced during render. Make sure that this property is reactive, either in the data option, or for class-based components, by initializing the property. See: https://vuejs.org/v2/guide/reactivity.html#Declaring-Reactive-Properties.  vue.js:634

found in

---> <ButtonCounter>
       <Root>
```

Vue props - Passing data (parent to child)

- Vue gives us the ability to use **props** to pass data from the parent down to the child.



Vue props - Passing data (parent to child)

- Props are **custom attributes** you can register on a component
- When a value is passed to a prop attribute, it becomes a property on that component instance
- **Parent components pass props down to their children**

```
Vue.component('blog-post', {
  props: ['title'],
  template: '<h3>{{ title }}</h3>'
})
```

```
<blog-post title="My journey with Vue"></blog-post>
<blog-post title="Blogging with Vue"></blog-post>
<blog-post title="Why Vue is so fun"></blog-post>
```

My journey with Vue

Blogging with Vue

Why Vue is so fun

- A component can have as many props as you'd like and by default, any value can be passed to any prop.
- Props are the way that we pass data from a parent component down to it's child components.

Vue – Component – A single root element

- **Every component must have a single root element**. You can fix this error by wrapping the template in a parent element, such as:

Wrap the template of the component in a single root element

```
<div class="blog-post">
  <h3>{{ title }}</h3>
  <div v-html="content"></div>
</div>
```

Defining a prop for each related piece of information could become very annoying:

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:title="post.title"
  v-bind:content="post.content"
  v-bind:publishedAt="post.publishedAt"
  v-bind:comments="post.comments"
></blog-post>
```

Key is a mandatory attribute when you use directive v-for

- Or let the template of the component display all content

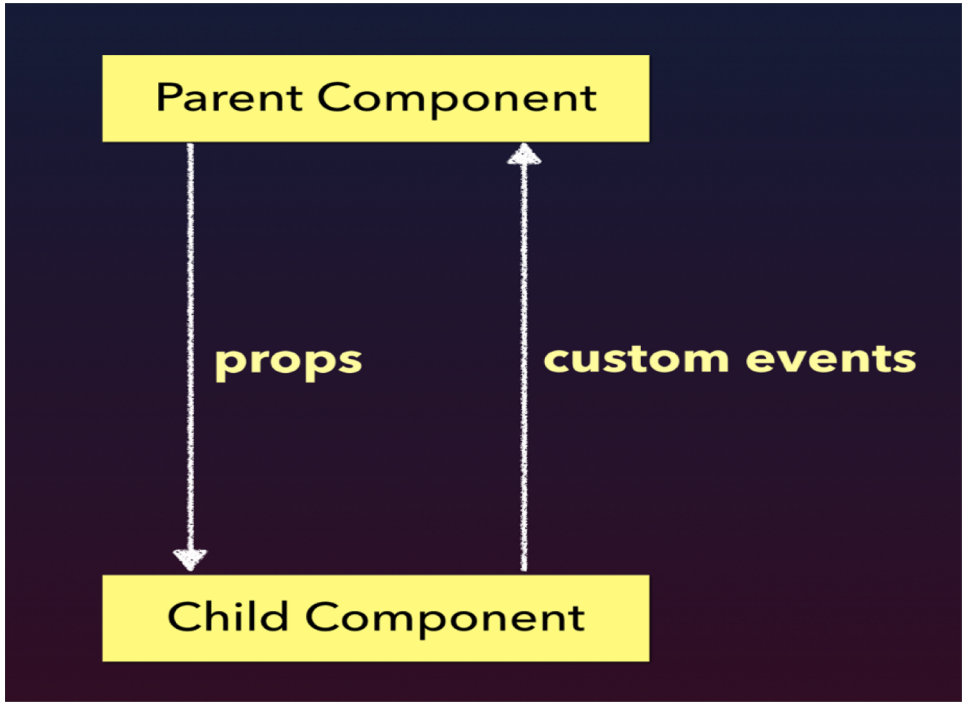
Less attributes to initialize

```
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:post="post"
></blog-post>
```

```
Vue.component('blog-post', {
  props: ['post'],
  template: `
    <div class="blog-post">
      <h3>{{ post.title }}</h3>
      <div v-html="post.content"></div>
    </div>
  `
})
```

Vue – Component – events

- To facilitate having the child component notify the parent about something, we can use Vue **custom events**.



- Custom events in Vue behave very similar to native JavaScript custom events but with one key distinction - **Vue custom events are used primarily for communication between components as opposed to communication between DOM nodes**.

Vue – Component – events

- We're able to send data down from a parent component via props (short for properties).
- But what **about sending data from a child component back up to its parent**?
  - **”Emit" a signal**—a **signal from a child component to notify a parent component** that an event has taken place (for example, a click event). Typically, the parent component will then perform some sort of action, such as the execution of a function.

Child component template

```
<button v-on:click="$emit('enlarge-text')">
  Enlarge text
</button>
```

Parent component

```
<blog-post
  ...
  v-on:enlarge-text="postFontSize += 0.1"
></blog-post>
```

Vue – Component – events

- **Emitting a value with an event**
  - We can use $emit parameters:

```
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>
```

  - Then when we listen to the event in the parent, we can access the emitted event's value with $event

```
<blog-post
  ...
  v-on:enlarge-text="postFontSize += $event"
></blog-post>
```

```
<blog-post
  ...
  v-on:enlarge-text="onEnlargeText"
></blog-post>
```

```
methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

Using a method

Using v-model on components

- Custom events can also be used to create custom inputs that work with v-model. Remember that:

```
<input v-model="searchText">
```

=

```
<custom-input
  v-bind:value="searchText"
  v-on:input="searchText = $event"
></custom-input>
```

- For this to actually work though, the <input> inside the component must:
  - Bind the **value attribute to a value prop**
  - On input, **emit its own custom input event with the new value**

Using v-model on components

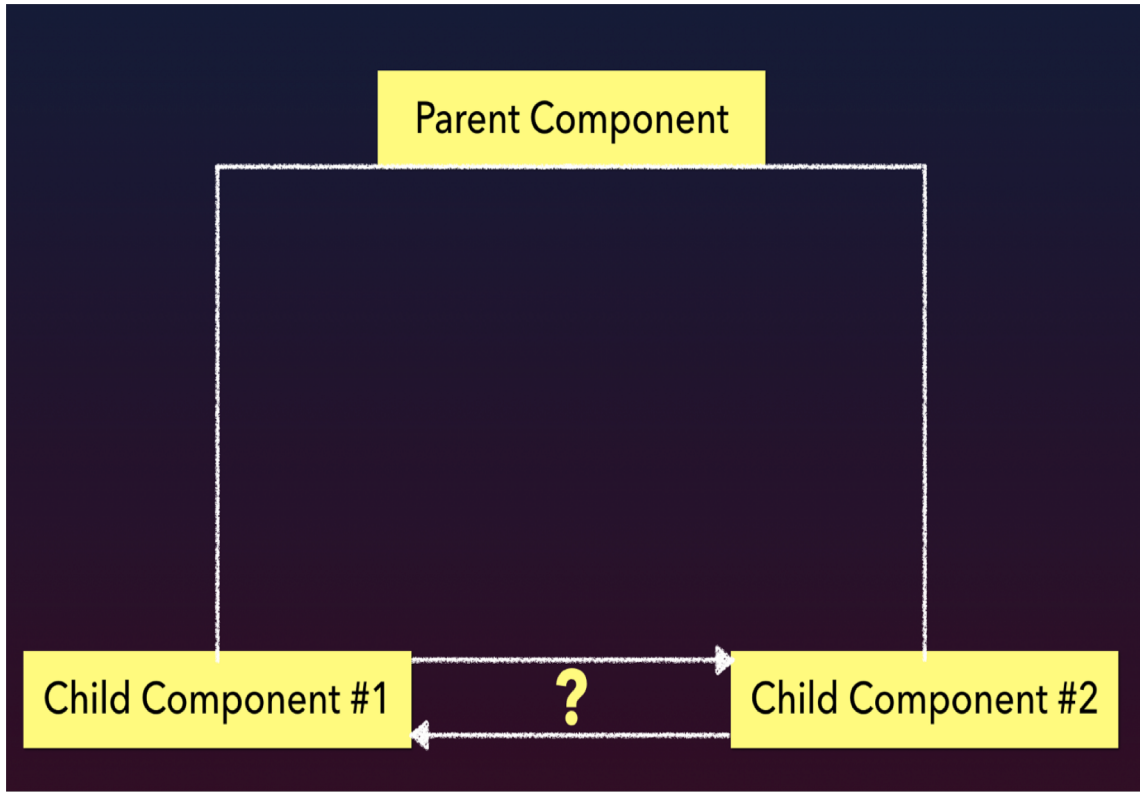
```
Vue.component('custom-input', {
  props: ['value'],
  template: `
    <input
      v-bind:value="value"
      v-on:input="$emit('input', $event.target.value)"
    >
  `
})
```

```
<custom-input v-model="searchText"></custom-input>
```

NOTE: searchText is the initial text value of the input component

Vue – Component – events between sibling

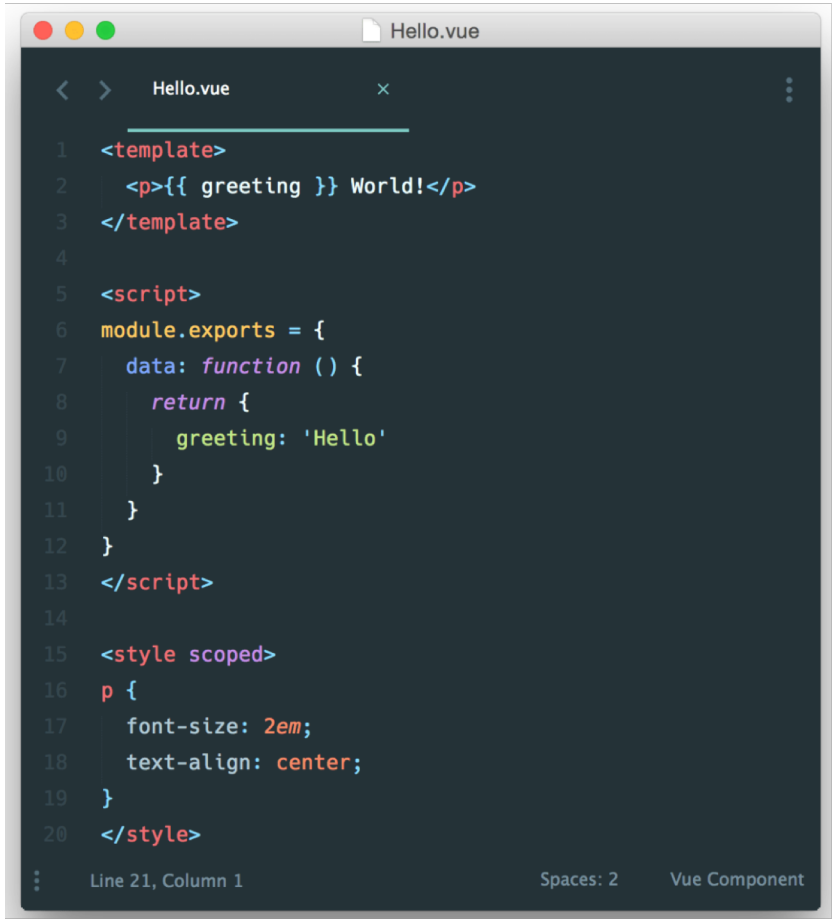
- ***Custom events*** *can be used to create communication from child to parent components.*
- There are roughly 3 main ways we can begin to manage information between sibling components and as a result start to handle application wide state management:
  - Use a **global EventBus**
  - Use a **simple global store**
  - Use the **flux-like library Vuex**



Vue Components – Single file vue component

- The **single file Vue Component**:
  - **Each single file Vue component** is treated as its own “mini application” within itself.
  - You can, of course, nest components within components.
  - The **Vue.js application is one large component** that is bootstrapped and **injected into one single Vue component**. which is then bootstrapped into a single Vue instance.
  - Your **HTML is not in a separate file**; **you don’t need to define an el for Vue.js** to control. Instead, the logic in the script tag and the CSS in the style tag directly affect the HTML above in the template tag.
  - The **data** in the single file component ***must*** **be a function that returns an object**

Vue component SFC - Example



- Inside a component, its **template, logic and styles are inherently coupled**, and collocating them actually makes the component more cohesive and maintainable.
- Even if you don't like the idea of Single-File Components, you can still leverage its hot-reloading and pre-compilation features by separating your JavaScript and CSS into separate files:

```
<!-- my-component.vue -->
<template>
  <div>This will be pre-compiled</div>
</template>
<script src="./my-component.js"></script>
<style src="./my-component.css"></style>
```

Important considerations

```
new Vue({
  render: h => h(App),
}).$mount('#app')
```

Your root Vue instance that the rest of the application descends from

```
</noscript>
<div id="app"></div>
<!-- built files will be auto injected -->
```

```
<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>
```

Declaring a component which can be registered and reused later

```
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>
```

Folder structure vue-cli application

- **Node modules directory** - Contains **all of the npm packages needed for your application to run**. Every time you run the command **npm install some-package**; the package some-package will download and be stored in this folder. From here, you can import dependencies into your Vue.js project or reference them manually in an HTML page.
- **Public directory** - Contains your **index.html** file that *everything* gets bootstrapped and injected in to. If you ever have the need to add a **dependency** like Bootstrap 4 into your application **via a CDN, for example, you can add it's respective tags in this file**. However, it's **best practice to always import them via the node_modules folder**.

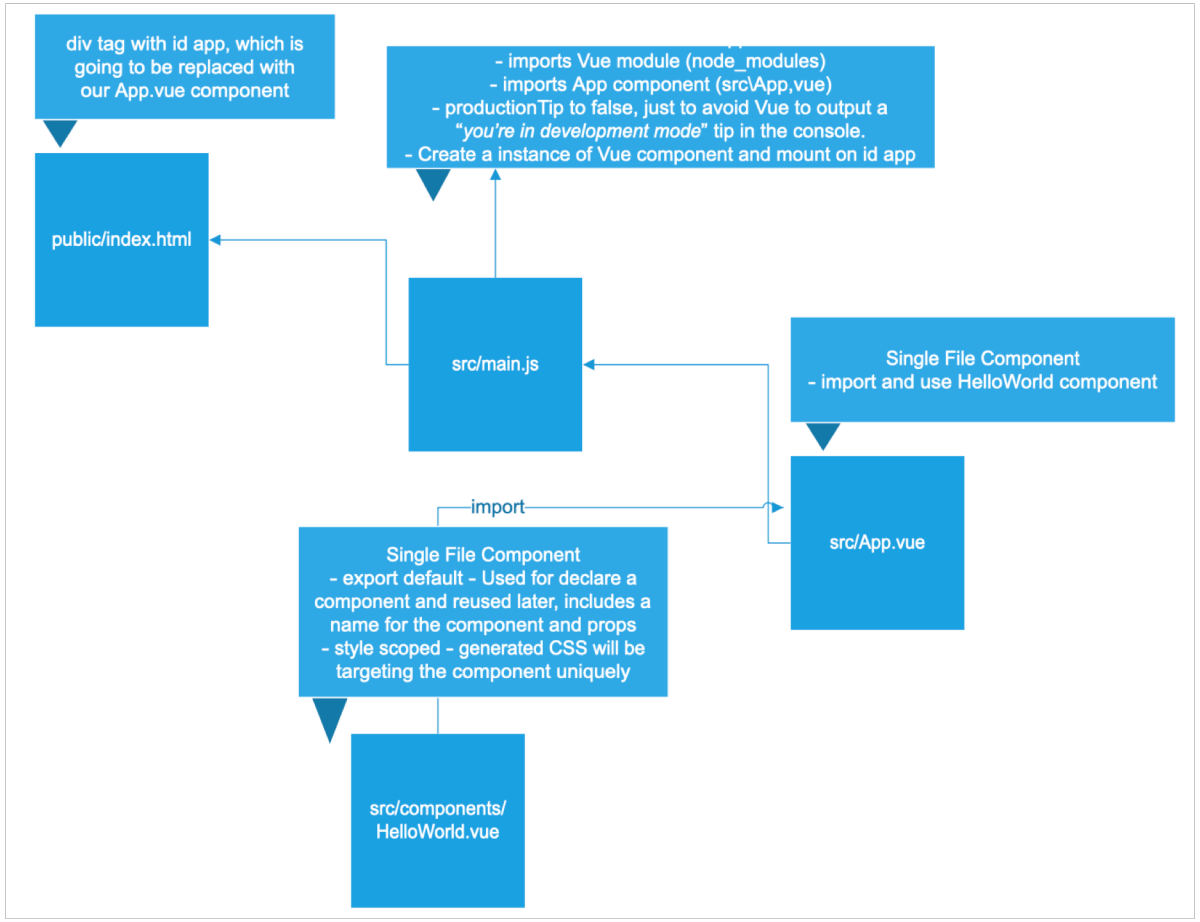
Folder structure vue-cli application

- **Source directory** - This is the most important directory in the generated project. In this folder, ***all*** **of your single file components**, stylesheets, assets and more are stored here.
  - **assets** - Stores all of your application's assets like images, CSS, and scripts
  - **components** - Stores all of the application's components (*.vue files)
  - **views** - Views are **single file components** that act as "pages" or containers that structure their child components
  - **App.vue** - The **single component** in which all other views and components get injected into. **This is a great place to add global components that should be shared across the app like Header.vue and Footer.vue.**

Folder structure vue-cli application

- **Source Directory**
  - **main.js** - This is **your single *Vue Instance*** in which the App.vue, routes, and all their components get injected into.
  - **router.js** - Define **your URL routes and which component get's loaded** when the URL address is visited.
  - **store.js** - Your Vuex store that contains state, mutations, and actions.
  - **package.json** - JSON file that **lists your NPM dependencies** and small project configurations.
  - **vue.config.js** - A file to add Webpack configs *without* ejecting!

vue-cli application template



ES2015 – Quick review

- ES2015 is a significant update to the language JavaScript, and the first major update to the language since ES5 was standardized in 2009.
- Implementation of these features in major JavaScript engines is underway now.
- Some of the features are:

  - Arrows and lexical this
  - Class
  - Enhanced Object Literals
  - Template string
  - Destructuring
  - Default + Rest + Spread
  - Let + Const
  - Iterators + For..Of
  - Generators
  - Modules
  - Module loaders
  - Map + Set + WeakMap + WeakSet
  - Proxy
  - Symbols
  - Subclassable Built-ins
  - Math + Number + String + Object APIs
  - Binary and octal literals
  - Promises
  - Reflect API
  - Tail call optimization

ES2015 arrow

- Arrows **are a function shorthand** using the **=> syntax**

```
// An arrow function expression is a syntactically compact alternative
// to a regular function expression
var materials = [
    'Hydrogen',
    'Helium',
    'Lithium',
    'Beryllium'
  ];
var pares=[2,4,6,8];
var impares=pares.map(v=>v+1);
var fives=[];
// The map() method creates a new array with the results of calling a function
// for every array element.
console.log('Lenght of strings: '+ materials.map(material => material.length));
// Imprimir impares
impares.forEach(
    v=> {
       if(v % 5 ===0 )
          fives.push(v);
    }
);
console.log("Impares: "+impares);
console.log("Fives: "+fives);
```

ES2015 lexical this

- Within an arrow function, this is the enclosing context

```
// Lexical this
// Without an arrow function this is defined as a global object
// Fat arrow functions don't have a this reference so they use the reference
//    of the enclosing context
var cliente = {
    _nombre: "Juan",
    _amigos: ['Pedro','David','Javier'],
    imprimirAmigos() {
        this._amigos.forEach(f => console.log(this._nombre+" conoce a "+f));
    }
};

cliente.imprimirAmigos();
```

ES2015 lexical this

- If an arrow is inside another function, it shares the "arguments" variable of its parent function.

```
// Lexical arguments
// If an arrow is inside another function, it shares the "arguments" variable
//    of its parent function.
function square() {
    // () we receive the arguments of the parent
    let example = () => {
      let numbers = [];
      for (let number of arguments) {
        numbers.push(number * number);
      }

      return numbers;
    };

    return example();
}
// We call square with multiple arguments
var resultado=square(2, 4, 7.5, 8, 11.5, 21);
console.log (resultado);// returns: [4, 16, 56.25, 64, 132.25, 441]
```

ES2015 Class

- Classes are a simple sugar over the prototype-based OO pattern
- Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability
- Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

```
class Polygon {
    //If one is not supplied, a default constructor is used instead:
    // constructor() { }
    constructor(height, width) {
      this.name = 'Polygon';
      this.height = height;
      this.width = width;
    }

    // Simple class instance methods using short-hand method
    // declaration
    sayName() {
      console.log('Hi, I am a ', this.name + '.');
    }

    sayMessage(){
        console.log("Polygon is derived from the Greek polus (many)" +
                    " and gonia (angle)");
    }
    // static method
    static convertCentimeterToInch(cm){
        console.log(cm+" centimeter "+"is equivalent to "+cm*0.39370 + " inches");
    }

  }
```

ES2015 Class - inheritance

```
class Square extends Polygon {
    constructor(length){
        super(length,length);
        this.name="Square";
    }
    get area(){
        return this.height * this.width;
    }

    set side(value){
        this.width = value;
        this.height = value;
    }

    sayName() {
      console.log('Hi, I am a ', this.name + '.');
    }

    // Override superclass method
    sayMessage(){
      console.log("Square extends Polygon ")
      super.sayMessage();
  }
}
```

ES2015 - Enhanced Object Literals

- Combine variables in a single object

```
// Step 1
// Combine variables in a single object
var x = 10, y = 20;
point1 = {x:x,y:y};
console.log("Point1: "+point1.x+","+point1.y);
// In ES2015 we can skip duplication
point2 = {x,y};
console.log("Point2: "+point2.x+","+point2.y);
```

ES2015 - Enhanced Object Literals

- Include functions in an object

```
// Step2
// Include functions in an object
var MATH1 = {
    add: function(a,b){
        return (a+b);
    }
}
console.log("SumA :"+MATH1.add(8,6));
//Include functions in an object using ES2015
//Constants are block-scoped,The value of a constant cannot change through reassignment,
//   and it can't be redeclared.
const MATH2 = {
    add(a,b){
        return a+b;
    }
}
console.log("SumB :"+MATH2.add(8,6));
```

ES2015 - Enhanced Object Literals

- Define async functions and generators

```
// Step 3
// ES2015 - Define async function and generator
const MATH3 = {
    async theAnswer(){ return 10; },
    *tenInts(){ // * to define a generator function, returns a Generator object
        for (let i=0;i<4;i++){
            yield i; // returns i to the generator next() method
        }
    }
}
// Returns a Promise object
console.log("MATH3 theAnswer:"+MATH3.theAnswer());
var numbers = MATH3.tenInts(); // Use a generator function
console.log(numbers.next());
console.log(numbers.next());
console.log(numbers.next());
console.log(numbers.next());
console.log(numbers.next());
// Print the content of the promise object that returned by an async function
var promiseA = MATH3.theAnswer().then(v => { console.log("Promise object: "+v);});
```

ES2015 - Enhanced Object Literals

- Arrow functions in an object

```
// Step4
// It is possible to include arrow functions ?
// How can we call?, arrow functions are anonymous, so the following code is bad
//const MATH4 = {
//    (a,b) => a+b
//}

// We need to change the code to the following:
const MATH4 = {
    add: (a,b) => a+b
}

console.log("MATH4: " + MATH4.add(20,6));
```

ES2015 - Enhanced Object Literals

- Dynamic properties

```
// Step 5
// We can dynamically define properties within an object, using ES2015 syntax
const NAME = "nombre",
obj = {
    [NAME]: "Javier Morales",
    ["prop_"+(()=>42)()]:42
};
console.log("Nombre: " + obj[NAME]);
console.log("Prop: "+obj["prop_42"]);
```

ES2015 – Template strings

```
// Template string
// Basic literal string creation
console.log(`This is a pretty little template string.`);

// Interpolate variable bindings
var name = "Bob", time = "today";
console.log(`Hello ${name}, how are you ${time}?`);

console.log(`In ES5 "\n" is a line-feed.`);
console.log(String.raw`In ES5 "\n" is a line-feed.`);
```

ES2015 – Destructuring

- Destructuring assignment allows you to assign the properties of an array or object to variables using syntax that looks similar to array or object literals.

```
var getASTNode = {op:10,lhs:{op:12},rhs:'Data1'};
// List matching
var [a, ,b] = [1,2,3];
console.log (a === 1);
console.log(b === 3);

// Object matching
var { op: a, lhs: { op: b }, rhs: c } = getASTNode;
console.log("a:"+a+" b:"+b+" c:"+c);

// Object matching shorthand
var {op,lhs,rhs} = getASTNode;
console.log("op:"+op+" lhs:"+lhs.op+" rhs:"+rhs);

// Can be used in parameter position
function g({name:x}){
  console.log(x);
}
g({name:5})

// Fail-soft destructuring
var [a] = [];
console.log(a === undefined);

// Fail-soft destructuring with defaults
var [a = 1] = [];
console.log(a === 1);

// Destructuring + defaults arguments
function r({x, y, w = 10, h = 10}) {
    return x + y + w + h;
}
console.log(r({x:1, y:2}) === 23);
```

ES2015 - Default + Rest + Spread

```
// Default + Rest + Spread
function f1(x, y=12) {
    // y is 12 if not passed (or passed as undefined)
    return x + y;
}
console.log(f1(3) == 15);

function f2(x, ...y) {
    // y is an Array
    return x * y.length;
}
console.log(f2(3, "hello", true) == 6);

function f3(x, y, z) {
    return x + y + z;
}
// Pass each elem of array as argument
console.log(f3(...[1,2,3]) == 6);
```

ES2015 Let + const

- Block-scoped binding constructs

```
//let + const
function f() {
    {
      let x; // block scoped
      {
        // this is ok since it's a block scoped name
        const x = "sneaky";
        // error, was just defined with `const` above
        x = "foo";
      }
      // this is ok since it was declared with `let`
      x = "bar";
      // error, already declared above in this block
      let x = "inner";
    }
  }
  f();
```

ES2015 Iterators

- The **Symbol.iterator** well-known symbol specifies the default iterator for an object. Used by for...of.

```
//Iterators + For..Of
//Iteration is based on the following interfaces
//interface IteratorResult {
//    done: boolean;
//    value: any;
// }
// interface Iterator {
//    next(): IteratorResult;
//  }
//  interface Iterable {
//    [Symbol.iterator](): Iterator
//  }
let fibonacci = {
    // Symbol.iterator() returns default iterator
    [Symbol.iterator]() {
      let pre = 0, cur = 1;
      return {
        // return each individual element using next()
        next() {
          //[pre,cur] = [1,1]
          //[pre,cur] = [1,2]
          //[pre,cur] = [2,3]
          [pre, cur] = [cur, pre + cur];
          return { done: false, value: cur }
        }
      }
    }
  }

  for (var n of fibonacci) {
    // truncate the sequence at 1000
    if (n > 1000)
      break;
    console.log(n);
  }
```

ES2015 - Generators

- Generators simplify iterator-authoring using function* and yield.
- A function declared as function* returns a Generator instance.
- Generators are subtypes of iterators which include additional next and throw.
- These enable values to flow back into the generator, so yield is an expression form which returns a value (or throws).

ES2015 - Generators

```
//interface Generator extends Iterator {
//    next(value?: any): IteratorResult;
//    throw(exception: any);
//}

var fibonacci = {
    // function*, generates a iterator instance
    [Symbol.iterator]: function*() {
      var pre = 0, cur = 1;
      for (;;) {
        var temp = pre;
        pre = cur;
        cur += temp;
        yield cur;
      }
    }
  }

  for (var n of fibonacci) {
    // truncate the sequence at 1000
    if (n > 1000)
      break;
    console.log(n);
  }
```

ES2015 - Modules

- Language-level support for modules for component definition
- Node.js doesn't support ES6 imports yet, However, you can use them today with the help of Babel.

Any variables declared inside a module aren't available to other modules unless they're *explicitly exported* as part of the module's API *(and then imported in the module that wants to access them)*.

JavaScript

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js
import * as math from "lib/math";
console.log("2π = " + math.sum(math.pi, math.pi));
```

```
// otherApp.js
import {sum, pi} from "lib/math";
console.log("2π = " + sum(pi, pi));
```

ES2015 - Modules

- Babel can transpile ES2015 Modules to several different formats including Common.js, AMD, System, and UMD
- Some additional features include export default and export *:

JavaScript

```
// lib/mathplusplus.js
export * from "lib/math";
export var e = 2.71828182846;
export default function(x) {
    return Math.exp(x);
}
```

```
// app.js
import exp, {pi, e} from "lib/mathplusplus";
console.log("e^π = " + exp(pi));
```

- There are two different types of export, named and default. You can have multiple named exports per module **but only one default export**[...]
- Named exports are useful to export several values. During the import, it is mandatory to use the same name of the corresponding object. **But a default export can be imported with any name**

ES2015 – Module loaders

- This is left as **implementation-defined within the ECMAScript 2015** specification. The **eventual standard will be in WHATWG's** Loader specification, but that is currently a work in progress.
- Module loaders support:
  - Dynamic loading
  - State isolation
  - Global namespace isolation
  - Compilation hooks
  - Nested virtualization

ES2015 - Map + Set + WeakMap + WeakSet

```
// Map + Set + WeakMap + WeakSet
// Sets - Store unique values of any type
var s = new Set();
s.add("hello").add("goodbye").add("hello");
console.log(s.size === 2);
console.log(s.has("hello") === true);


// Maps - Holds key-value pairs and remembers the original insertion order
//  of the keys. Any value (both objects and primitive values) may be used
//  as either a key or a value
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
console.log(m.get(s) == 34);
```

ES2015 - Map + Set + WeakMap + WeakSet

```
// Weak Maps - Is a collection of key/value pairs in which the keys are
//  weakly referenced.  The keys must be objects and the values can
//   be arbitrary values
// "Weak" references to key objects, which means that they do not prevent garbage
//   collection in case there would be no other reference to the key object.
//   This also avoids preventing garbage collection of values in the map
// Because of references being weak, WeakMap keys are not enumerable
//   (i.e. there is no method giving you a list of the keys)
var wm = new WeakMap();
wm.set(s, { extra: 42 });
console.log(wm.size === undefined); //true


// Weak Sets
// Store collections or iterable objects
// If an iterable object is passed, all of its elements will be added to the
// new WeakSet. null is treated as undefined.
var ws = new WeakSet();
ws.add({ data: 42 });
// Because the added object has no other references, it will not be held in the set
```

ES2015 - Proxy

```
//Proxies
//Proxy - used to define custom behavior for fundamental operations
//        (e.g. property lookup, assignment, enumeration, function invocation, etc).
// traps - The methods that provide property access
// handler - Placeholder object which contains traps.
// target - Object which the proxy virtualizes.

// Property lookup
var target = {};
var handler = {
  // The handler.get() method is a trap for getting a property value.
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};
var p = new Proxy(target, handler);
//p.world is a get call to property world, received in name argument
console.log(p.world === "Hello, world!"); //true
//-----------------
// Proxying a function invocation
var target = function () { return "I am the target"; };
var handler = {
  // The handler.apply() method is a trap for a function call.
  apply: function (receiver, ...args) {
    return "I am the proxy";
  }
};

var p = new Proxy(target, handler);
console.log(p() === "I am the proxy"); //true
//-----------------
```

- SEE: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

ES2015 - Symbols

- Symbols are new **primitive** type introduced in ES6. Symbols are completely unique identifiers
- They can be created using the factory function Symbol() which returns a Symbol.
- The optional string-valued parameter is a descriptive string that is shown when printing the symbol.

```
const symbol1 = Symbol();
const symbol2 = Symbol(22);
const symbol3 = Symbol('foo');
console.log(typeof symbol1); //symbol
console.log(symbol2); // Symbol(22)
console.log(symbol3.toString()); // Symbol(foo)
console.log(Symbol('foo')  === Symbol('foo')); // false, each Symbol is unique
```

ES2015 - Symbols

```
const _counter = Symbol('counter');
const _action  = Symbol('action');
const _method1 = Symbol('method');
class Countdown {
    constructor(counter, action,id) {
        this[_counter] = counter;
        this[_action] = action;
        this.id = id;  // It does not use a Symbol

    }

    dec() {
        let counter = this[_counter];
        if (counter < 1) return;
        counter--;
        this[_counter] = counter;
        if (counter === 0) {
            this[_action]='stopDrink';
        }
    }

}
```

```
Symbols....
symbol
Symbol(22)
Symbol(foo)
false
Countdown....
Symbols are hidden
Array(1) ["id"]
Array(1) ["id"]
Symbols are listed
Array(3) ["id", Symbol(counter), Symbol(action)]
Array(2) [Symbol(counter), Symbol(action)]
#Coffee: 3, Action: drinkCoffee , id:1
#Coffee: 2, Action: drinkCoffee , id:1
#Coffee: 1, Action: drinkCoffee , id:1
#Coffee: 0, Action: stopDrink , id:1
```

ES2015 – Sub classable Built-ins

- In ES2015, built-ins like Array, Date and DOM Elements can be subclassed

```
1   // User code of Array subclass
2   class MyArray extends Array {
3       constructor(...args) { super(...args); }
4   }
5
6   var arr = new MyArray();
7   arr[1] = 12;
8   console.log(arr[0]); //undefined
9   console.log(arr.length == 2); //true
10  for (let j in arr){
11      console.log(arr[j]); //12
12  }
13
```

ES2015 - Math + Number + String + Object APIs

- **Many new library additions**, including core Math libraries, Array conversion helpers, and Object.assign for copying

```
console.log(Number.EPSILON) // 2.220446049250313080847263336181 6E-16, or 2-52
console.log(Number.isInteger(Infinity)) // false, Infinity is a global property
console.log(Number.isInteger(Math.PI)) // false
console.log(Number.isInteger(5)) // true
// isNaN - This method returns true if the value is of the type Number
//   and equates to NaN. Otherwise it returns false.
console.log(Number.isNaN('NaN')) // false
console.log(Number.isNaN(NaN)) // true
console.log(Number.isNaN(123)) // false
console.log(Number.isNaN(0/0)) // true

console.log(Math.acosh(3)) // 1.762747174039086
console.log(Math.hypot(3, 4)) // 5, returns the square root of the sum of squares

console.log("abcde".includes("cd")) // true
console.log("abc".repeat(3)) // "abcabcabc"

console.log(Array.from('My dog')) // ["M", "y", " ", "d", "o", "g"]
console.log(Array.from([1, 2, 3], x => x + x)); //   [2, 4, 6]

console.log(Array.of(1, 2, 3)) // Similar to new Array(...), [1, 2, 3]
console.log([0,0,0].fill(7,1)) // [0,7,7], fill with 7 from position 1
// The findIndex() method returns the index of the first element in the array
//  that satisfies the provided testing function. Otherwise, it returns -1
console.log([1,2,3].findIndex(x => x == 2)) // 1
```

ES2015 - Math + Number + String + Object APIs

```
// The entries() method returns a new Array Iterator object that contains
// the key/value pairs for each index in the array.
var iterator1 = ["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
console.log(iterator1.next().value)
console.log(iterator1.next().value)

//["a", "b", "c"].keys() // iterator 0, 1, 2
//["a", "b", "c"].values() // iterator "a", "b", "c"

// Object.assign() method is used to copy the values of all enumerable
//  own properties from one or more source objects to a target object.
//  It will return the target object.
//  If the source value is a reference to an object,
//   it only copies that reference value.
// Object.assign(target, ...sources)
var obj = {a:1,b:'Text1',c:{d:0}}
var copy = Object.assign({},obj)
console.log(JSON.stringify(copy)) // {"a":1,"b":"Text1","c":{"d":0}}
obj.b='Text2'
console.log(JSON.stringify(obj)) // {"a":1,"b":"Text2","c":{"d":0}}
console.log(JSON.stringify(copy)) // {"a":1,"b":"Text1","c":{"d":0}}
obj.c.d=10
console.log(JSON.stringify(obj)) // {"a":1,"b":"Text2","c":{"d":10}}
console.log(JSON.stringify(copy)) // {"a":1,"b":"Text1","c":{"d":10}}
```

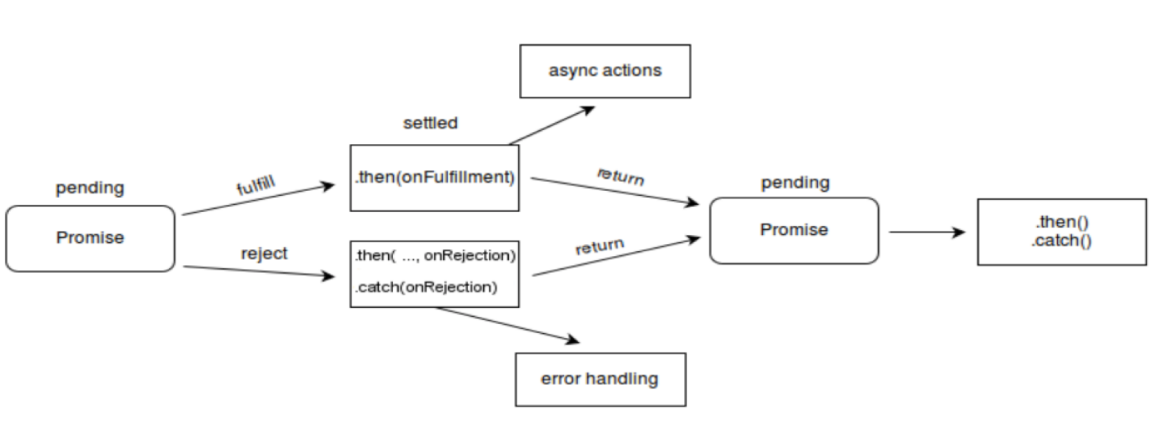
ES2015 - Binary and octal literals

- Two new numeric literal forms are added for binary (b) and octal (o).

```
console.log(0b111110111 === 503) // true
console.log(0o767 === 503) // true
```

ES2015 - Promises

- Promises are a library for asynchronous programming. Promises are a first class representation of a value that may be made available in the future. Promises are used in many existing JavaScript libraries
- A Promise is in one of these states:
  - *pending*: initial state, neither fulfilled nor rejected.
  - *fulfilled*: meaning that the operation completed successfully.
  - *rejected*: meaning that the operation failed.
- As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.



ES2015 - Promises

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
  });
}
```

ES2015 - Promises

```
function timeout(duration = 0) {
    return new Promise((resolve, reject) => {
        // new Promise()
        // We call resolve(...) when what we were doing asynchronously was successful,
        //   and reject(...) when it failed.

        // In this example, we use setTimeout(...) to simulate async code.
        // executes resolve, after waiting a duration
        //setTimeout(resolve, duration);
        setTimeout(function(){
            resolve("Exito "+ duration);  // Send to the resolve a string as argument
        }, duration);

    })
}

var p = timeout(5000).then((message) => {
    // This is the resolve for the timeout(5000)
    console.log('Resolve executed :' + message ); // after 5s
    return timeout(2000); //2000ms, return another Promise
}).then((message) => {
    // This is the resolve for the second call to timeout(2000), or a second Promise
    // This resolve generates an exception
    console.log('Resolve executed :'+ message)
    throw new Error("hmm");
}).catch(err => {
    // This is the failureCallback
    console.log(err);
    // Promise.all - returns a single Promise that resolves when all of the promises
    // passed as an iterable have resolved or when the iterable contains no promises.
    // It rejects with the reason of the first promise that rejects.
    Promise.all([timeout(100), timeout(200)]).then((values) => {
        console.log(values); // ["Exito 100", "Exito 200"]
    });
})
```

ES2015 – Reflect API

- Full reflection API exposing the runtime-level meta-operations on objects. This is effectively the inverse of the Proxy API, and allows making calls corresponding to the same meta-operations as the proxy traps. Especially useful for implementing proxies.

```
1   var O = {a: 1};
2   // Add a property to an object
3   Object.defineProperty(O, 'b', {value: 2});
4   O[Symbol('c')] = 3;
5
6   console.log(Reflect.ownKeys(O)); // ['a', 'b', Symbol(c)]
7
8   function C(a, b){
9     this.c = a + b;
10  }
11  // Reflect.construct - It is equivalent to calling new target(...args).
12  // It gives also the added option to specify a different prototype.
13  var instance = Reflect.construct(C, [20, 22]);
14  console.log(instance.c); // 42
```

Additional information

- Component basics - https://vuejs.org/v2/guide/components.html
- Single file components - https://vuejs.org/v2/guide/single-file-components.html
- Emit events - https://www.telerik.com/blogs/how-to-emit-data-in-vue-beyond-the-vuejs-documentation
- Form input binding - https://vuejs.org/v2/guide/forms.html
- Vue style guide - https://vuejs.org/v2/style-guide/
- ES2015 - https://babeljs.io/docs/en/learn