**NodeAns 2**

Self Intro

Related to my APIS and work

API Example: and practice

React

Clustering in node

Socket programming

Arrow function

clouser,prototype real time exmaple

Firstcry - more about db ,

pass by value and pass by ref in javascript

ng-app can be used more than once ?

node streams , pipe

buffers

promises testcases

mocha chai script writing and run

pck.json ^ and ~ means

async lib

js typecasting at runtime

node event loop runs on ram or rom

event emiter

debugging of node app

loadash

promises syntax

use of SOAP connector ..syntax

security node app

how to switch port at runtime – sftp

bouncy script - <http://substack.net/bounce_http_requests_with_bouncy>

websockets

JSLint

--Telstra Wipro

Iife - self invoking function

how to send 100s of requests to node server

Why to use .then in promises

How many types of Dependencies in pck.JSON

Peer dependencies

Diff bet let and const

Testing environments – DEV,SIT,PAT,BAT,PROD

UAT- user acceptance testing

Node modules used- express , request, mocha , chai, fis connector, mongodb, mongoose, gulp

Async,body parser,loopback,dateformat

**Clouser :**

Clouser means a function inside a function which can access variables declared outside it.

Basically it is accessing a variable which is out of scope.

**Why clouser**: With a function closure you can store data in a separate scope, and share it only where necessary.

## Lexical Scoping

First off, JavaScript has *lexical scoping* with *function scope*. In other words, even though JavaScript looks like it should have block scope because it uses curly braces { }, a new scope is created only when you create a new function.

|  |
| --- |
| **var** outerFunction  = **function**(){    **if**(**true**){  **var** x = 5;  *//console.log(y); //line 1, ReferenceError: y not defined*  }    **var** nestedFunction = **function**() {    **if**(**true**){  **var** y = 7;  console.log(x); *//line 2, x will still be known prints 5*  }    **if**(**true**){  console.log(y); *//line 3, prints 7*  }  }  **return** nestedFunction;  }    **var** myFunction = outerFunction();  myFunction(); |

In this example, the variable *x* is available everywhere inside of outerFunction(). Also, the variable *y* is available everywhere within the nestedFunction(), but neither are available outside of the function where they were defined. The reason for this can be explained by lexical scoping. **The scope of variables is defined by their position in source code**. In order to resolve variables, JavaScript starts at the innermost scope and searches outwards until it finds the variable it was looking for. Lexical scoping is nice, because we can easily figure out what the value of a variable will be by looking at the code; whereas in dynamic scoping, the meaning of a variable can change at runtime, making it more difficult.

## Closures

The fact that we can access the variable *x* might still be confusing, because, normally, a local variable inside a function is gone after a function finishes executing. We called outerFunction() and assigned its result, nestedFunction(), into myFunction(). How does the variable *x* still exist if outerFunction() has already returned?

Merely accessing a variable outside of the immediate scope (no return statement is necessary) will create something called a *closure*

***Bind() call() apply()***

***Bind method is used to bind a this value to variable***

When we use the bind() method: the JS engine is creating a new pokemonName instance and binding pokemon as its this variable. It is important to understand that ***it copies the pokemonName function.***

|  |  |
| --- | --- |
|  | var pokemon = { |
|  | firstname: 'Pika', |
|  | lastname: 'Chu ', |
|  | getPokeName: function() { |
|  | var fullname = this.firstname + ' ' + this.lastname; |
|  | return fullname; |
|  | } |
|  | }; |
|  |  |
|  | var pokemonName = function(snack, hobby) { |
|  | console.log(this.getPokeName() + 'I choose you!'); |
|  | console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby); |
|  | }; |
|  |  |
|  | var logPokemon = pokemonName.bind(pokemon); // creates new object and binds pokemon. 'this' of pokemon === pokemon now |
|  |  |
|  | logPokemon('sushi', 'algorithms'); // Pika Chu loves sushi and algorithms |

The **call()** method calls a function with a given this value and arguments provided individually.

What that means, is that we can call any function, and *explicitly specify what this should reference* within the calling function. Really similar to the bind()method! This can definitely save us from writing hacky code (even though we are all still hackerzzz).

The main differences between bind() and call() is that the call()method:

1. Accepts additional parameters as well
2. Executes the function it was called upon right away.
3. The call() method does not make a copy of the function it is being called on.

call() and apply() serve the **exact same purpose.** The ***only difference between how they work is that*** call() expects all parameters to be passed in individually, whereas apply() expects an array of all of our parameters. Example:

|  |
| --- |
| var pokemon = { |
|  | firstname: 'Pika', |
|  | lastname: 'Chu ', |
|  | getPokeName: function() { |
|  | var fullname = this.firstname + ' ' + this.lastname; |
|  | return fullname; |
|  | } |
|  | }; |
|  |  |
|  | var pokemonName = function(snack, hobby) { |
|  | console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby); |
|  | }; |
|  |  |
|  | pokemonName.call(pokemon,'sushi', 'algorithms'); // Pika Chu loves sushi and algorithms |
|  | pokemonName.apply(pokemon,['sushi', 'algorithms']); // Pika Chu loves sushi and algorithms |

var logPokemon = pokemonName.**bind**(pokemon); // creates new object and binds pokemon. 'this' of pokemon === pokemon now

|  |  |
| --- | --- |
|  | pokemonName.**call**(pokemon,'sushi', 'algorithms'); // Pika Chu loves sushi and algorithms |
|  | pokemonName.**apply**(pokemon,['sushi', 'algorithms']); // Pika Chu loves sushi and algorithms |

**Async.waterfall syntax**

**Async.waterfall ( [ f1,f2,f3 ] ,cb );**

example

async.waterfall ( [

function(next) {

requestValidator.requestValidation(params, function(err, result1) {

if (err) {

next(err, null);

} else {

next(null, result1);

}

});

},

function(result1, next) {

requestMapper.requestTransformer(params, function(err, result2) {

if (err) {

return next(err, null);

} else {

next(null, result2);

}

});

},

function(result2, next) {

fisPagination.execute(params, result2, function(err, result3) {

if (err) {

return next(err, null);

} else {

next(null, result3);

}

});

}

], function(err, result) {

if (err) {

logger.error('Error while executing async waterfall,', err, metadata);

callback(err);

} else {

callback(null, result);

}

});

**Promises :**

**Promise is like normal promise ..if some one says you that he will do this task ..its a promise ..that task will be done. Where it will be done OR not….that means promises can be resolved or rejected .**

Promises are used to make an asynchronous call to a remote web service.

a Promise object represents the data that will be returned by the web service in future.

The actual data is not available yet. It will become available when the request completes and a response comes back from the web service. In the meantime the Promise object acts like a proxy to the actual data

The core component of a promise object is its then method. The then method is used to get the return value (known as the *fulfillment value*) or the exception thrown (known as the *rejection reason*) from an asynchronous operation. then takes two optional callbacks as arguments, which we’ll call onFulfilled and onRejected:

var promise = doSomethingAync()

promise.then(onFulfilled, onRejected)

onFulfilled and onRejected are called when the promise is resolved (the asynchronous processing has completed). Only one will ever be triggered since only one resolution is possible.

**SYNTAX**

**Var p= new Promise**

**????**

**Pass by value and pass by ref in javascript**

It's always pass by value, but for objects the value of the variable is a reference. Because of this, when you pass an object and change its *members*, those changes persist outside of the function. This makes it *look* like pass by reference. But if you actually change the value of the object variable you will see that the change does not persist, proving it's really pass by value.

Example:

function changeObject(x) {

x = {member:"bar"};

alert("in changeObject: " + x.member);

}

function changeMember(x) {

x.member = "bar";

alert("in changeMember: " + x.member);

}

var x = {member:"foo"};

alert("before changeObject: " + x.member);

changeObject(x);

alert("after changeObject: " + x.member); /\* change did not persist \*/

alert("before changeMember: " + x.member);

changeMember(x);

alert("after changeMember: " + x.member); /\* change persists \*/

Output:

before changeObject: foo

in changeObject: bar

after changeObject: foo

before changeMember: foo

in changeMember: bar

after changeMember: bar

**ng-app can be used more than once ?**

Only one AngularJS application can be auto-bootstrapped per HTML document. The first ngApp found in the document will be used to define the root element to auto-bootstrap as an application. To run multiple applications in an HTML document you must manually bootstrap them using [angular.bootstrap](https://docs.angularjs.org/api/ng/function/angular.bootstrap) instead.

**Package.Json related**

* **version** Must match **version** exactly
* **>version** Must be greater than **version**
* **>=version** etc
* **<version**
* **<=version**
* **~version** "Approximately equivalent to version" See [semver](https://docs.npmjs.com/misc/semver)
* **^version** "Compatible with version" See [semver](https://docs.npmjs.com/misc/semver)
* For example, these are all valid:

{ "dependencies" :

{ "foo" : "1.0.0 - 2.9999.9999"

, "bar" : ">=1.0.2 <2.1.2"

, "baz" : ">1.0.2 <=2.3.4"

, "boo" : "2.0.1"

, "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"

, "asd" : "http://asdf.com/asdf.tar.gz"

, "til" : "~1.2"

, "elf" : "~1.2.3"

, "two" : "2.x"

, "thr" : "3.3.x"

, "lat" : "latest"

, "dyl" : "file:../dyl"

}

}

**Types of dependencies**

Dependencies serve many different purposes. Some dependencies are needed to build your project, others are needed when you’re running your program. As such there are a number of different types of dependencies that you can have

{

"name": "my-project",

"dependencies": {

"package-a": "^1.0.0"

},

"devDependencies": {

"package-b": "^1.2.1"

},

"peerDependencies": {

"package-c": "^2.5.4"

},

"optionalDependencies": {

"package-d": "^3.1.0"

}

"bundledDependencies": {

"package-d": "^3.1.0"

}

}

##### dependencies

These are your normal dependencies, or rather ones that you need when running your code (e.g. React or ImmutableJS).

##### devDependencies

These are your development dependencies. Dependencies that you need at some point in the development workflow but not while running your code (e.g. Babel or Flow).

##### peerDependencies

Peer dependencies are a special type of dependency that would only ever come up if you were publishing your own package.

Having a peer dependency means that your package needs a dependency that is the same exact dependency as the person installing your package. This is useful for packages like react that need to have a single copy of react-dom that is also used by the person installing it.

##### optionalDependencies

Optional dependencies are just that: optional. If they fail to install, Yarn will still say the install process was successful.

This is useful for dependencies that won’t necessarily work on every machine and you have a fallback plan in case they are not installed (e.g. Watchman).

##### bundledDependencies

Array of package names that will be bundled when publishing the package.

Bundled dependencies should be inside your project. The functionality is basically the same as normal dependencies. They will also be packed when running yarn pack.

Normal dependencies are usually installed from the npm registry. Bundled dependencies are useful in cases normal dependencies are not sufficient:

* When you want to re-use a third party library that doesn’t come from the npm registry or that was modified.
* When you want to re-use your own projects as modules.
* When you want to distribute some files with your module.

|  |
| --- |
|  |
|  |
|  | "**dependencies**": {  "async": "~0.9", |
|  | "biased-opener": "~0.2.2", |
|  | "debug": "^2.2.0", |
|  | "express": "^4.12.3", |
|  | "glob": "^5.0.5", |
|  | "path-is-absolute": "^1.0.0", |
|  | "rc": "^1.0.1", |
|  | "semver": "^4.3.4", |
|  | "serve-favicon": "^2.1.1", |
|  | "strong-data-uri": "^1.0.0", |
|  | "v8-debug": "~1.0.0", |
|  | "v8-profiler": "~5.7.0", |
|  | "which": "^1.1.1", |
|  | "ws": "^1.0.1", |
|  | "yargs": "^3.9.0" |
|  | }, |
|  | "**devDependencies**": { |
|  | "chai": "^3.5.0", |
|  | "fs-extra": "~0.8.1", |
|  | "install": "^0.8.7", |
|  | "jshint": "^2.4.4", |
|  | "mocha": "^1.21", |
|  | "npm": "^4.4.1", |
|  | "promise": "^7.0.3", |
|  | "rimraf": "^2.6.1" |
|  | } |

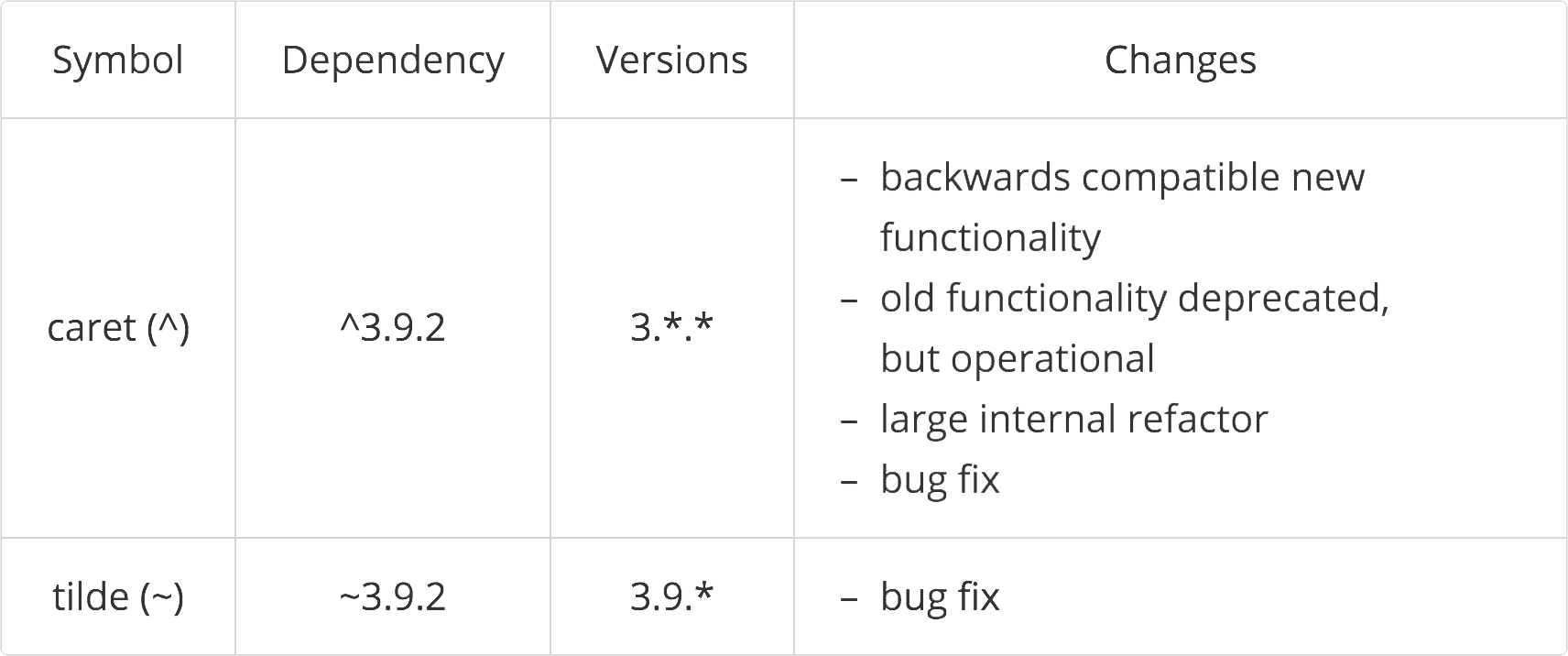
What is the difference between tilde(~) and caret(^)

In the simplest terms, the tilde matches the most recent minor version (the middle number). ~1.2.3 will match all 1.2.x versions but will miss 1.3.0.

The caret, on the other hand, is more relaxed. It will update you to the most recent major version (the first number). ^1.2.3 will match any 1.x.x release including 1.3.0, but will hold off on 2.0.0

Npm allows installing newer version of a package than the one specified. Using tilde (~) gives you bug fix releases and caret (^) gives you backwards compatible new functionality as well.

The problem is old versions usually don't receive bug fixes that much, so npm uses caret (^) as the default for --save.



**Freeze major-level**

~0 (0) 0.0 <= v < 1

0.2 0.2 <= v < 1 // Can't do that with ^ or ~

~1 (1, ^1) 1 <= v < 2

^1.2 1.2 <= v < 2

^1.2.3 1.2.3 <= v < 2

^1.2.3-beta.4 1.2.3-beta.4 <= v < 2

**Freeze minor-level**

^0.0 (0.0) 0 <= v < 0.1

~0.2 0.2 <= v < 0.3

~1.2 1.2 <= v < 1.3

~0.2.3 (^0.2.3) 0.2.3 <= v < 0.3

~1.2.3 1.2.3 <= v < 1.3

So if you see ~1.0.2 it means to install version 1.0.2 or the latest patch version such as 1.0.4. If you see ^1.0.2 it means to install version 1.0.2 or the latest minor or patch version such as 1.1.0

* **Bouncy**

pipe raw http traffic from incoming http requests to remote endpoints

var bouncy = require('bouncy');

bouncy(function (req, bounce) {

if (req.headers.host === 'bouncy.example.com') {

bounce(8000);

}

else if (req.headers.host === 'trampoline.example.com') {

bounce(8001)

}

}).listen(80);

Since bouncy is just parsing the http headers and sending along the raw tcp stream, you can use websockets on the place you bounce() to without writing any special code!