The Jasmine Effect

Automate; Don't Panic.

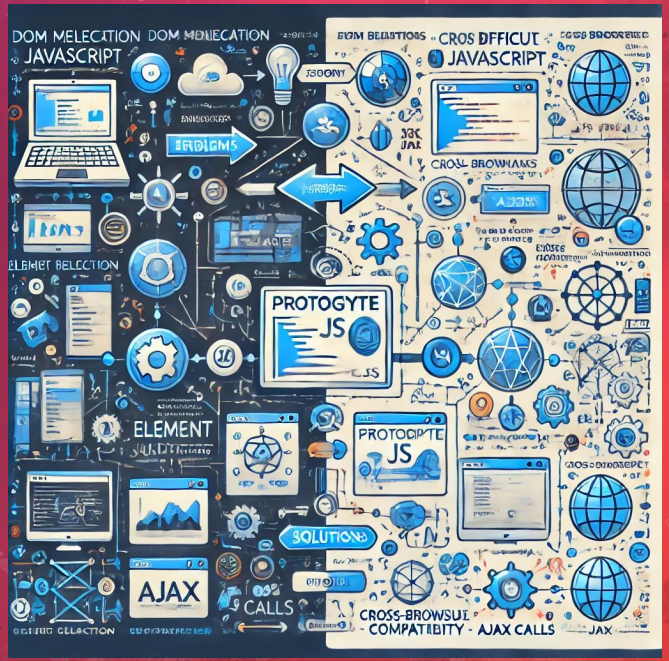
Jasmine's Got Your Back

- Jasmine is a lightweight testing framework for testing JavaScript code.
- It does not depend on any other JavaScript frameworks.
- It runs in browsers and in Node.js.
- And it has a clean, obvious syntax so that you can easily write tests.
- Think of it as a detective who catches bugs before they cause trouble.
    - Automated, consistent testing.
    - Catching bugs early in development.
    - Improving script quality and maintainability.

Prototype

Created by **Sam Stephenson** in 2005 to simplify and improve the process of writing JavaScript for web development.
It provided various utilities and extensions to the JavaScript language, such as:

- Extending built-in JavaScript objects like Array, String, and Function.
- Easier DOM traversal and manipulation.
- Cross-browser support.



ServiceNow's Secret Sauce

Enter Prototype.js Code

```
var Person = Class.create();
Person.prototype = {
  initialize: function(name) {
    this.name = name;
  },
  say: function(message) {
    return this.name + ': ' + message;
  }
};

var guy = new Person('Miro');
guy.say('hi');
```

```
Miro: hi
```

```
1  var Person = Class.create();
2  Person.prototype = {
3      initialize: function(personName) {
4          this.personName = personName;
5      },
6      say: function(message) {
7          return this.personName + ': ' + message;
8      },
9
10     type: 'Person'
11 };
```

```
1  var guy=new Person("Miro");
2  gs.info(guy.say("hi"));
```

```
*** Script: Miro: hi
```

ServiceNow's Secret Sauce

Enter Prototype.js Code

```
var Person = Class.create();
Person.prototype = {
  initialize: function(name) {
    this.name = name;
  },
  say: function(message) {
    return this.name + ': ' + message;
  }
};

var Pirate = Class.create();

// inherit from Person class:
Pirate.prototype = Object.extend(new Person(), {

  // redefine the speak method
  say: function(message) {
    return this.name + ': ' + message + ', yarr!';
  }

});

var john = new Pirate('Long John');
john.say('ahoy matey');
```

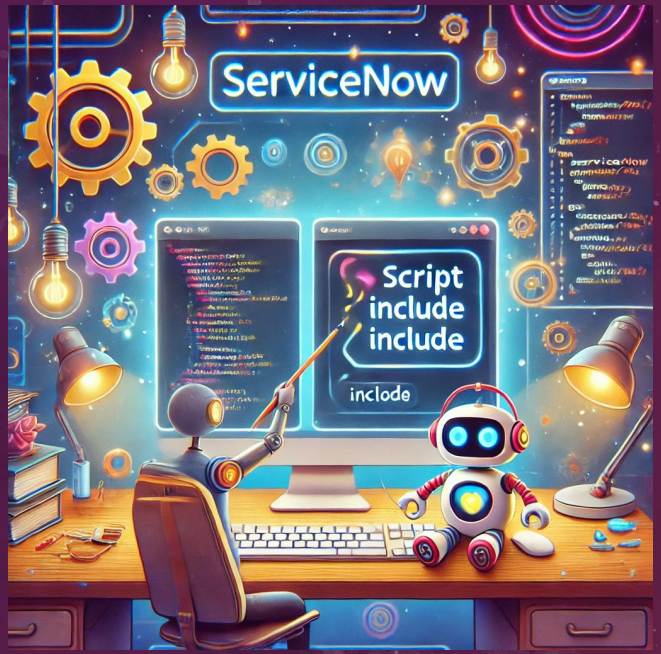
```
Long John: ahoy matey, yarr!
```

```
1  var Pirate = Class.create();
2  Pirate.prototype = Object.extendsObject(Person, {
3
4      say: function(message) {
5          return this.personName + ': ' + message + ', yarr!';
6      },
7
8      type: 'Pirate'
9  });
```

```
1  var john = new Pirate('Long John');
2  gs.info(john.say("ahoy matey"));
```

```
*** Script: Long John: ahoy matey, yarr!
```

Script Includes



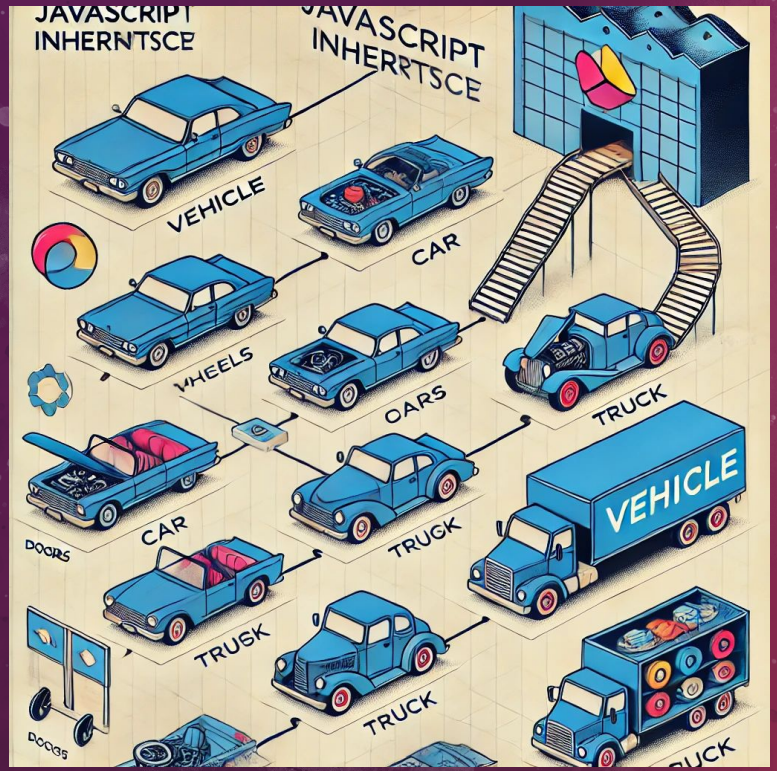
- Script Includes are reusable server-side script logic that define a function or class.
- Script Includes execute their script logic only when explicitly called by other scripts.
- They are a powerful way to write code that can be called from multiple places, making your instance more modular, maintainable, and efficient.

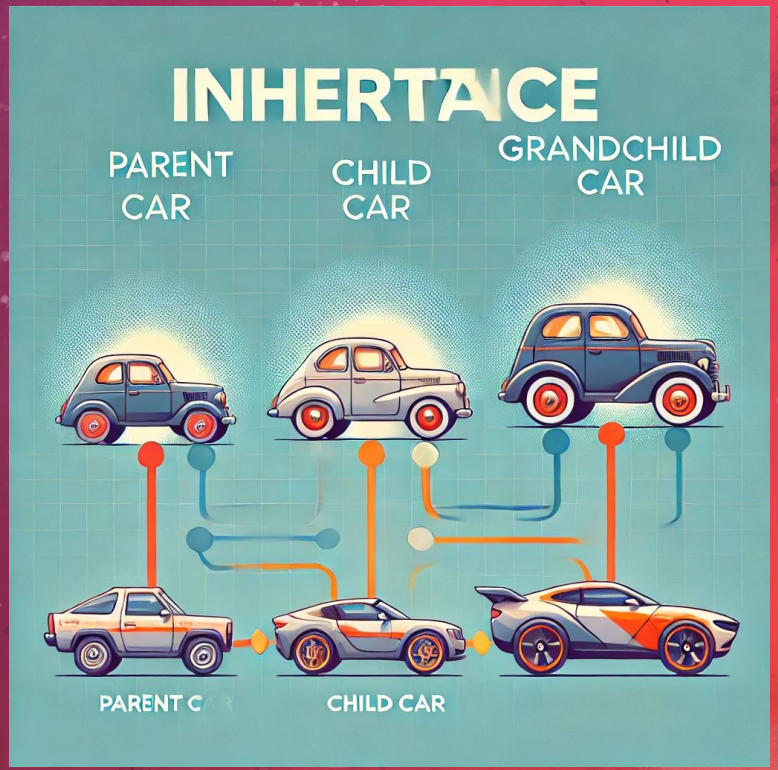
Script Includes Patterns

```
var ParentSI = Class.create();
ParentSI.prototype = {
    initialize: function() {
        gs.info("initialize: ParentSI method.");
    },
    parentFunc: function() {
        gs.info("parentFunc: ParentSI method.");
    },
    funcToBeOverriden: function() {
        gs.info("funcToBeOverriden: ParentSI method.");
    },

    type: 'ParentSI'
};
```

```
function sumTwoNums(int1, int2) {
    var mySum = int1 + int2;
    gs.info("The sum of " + int1 + " + " + int2 + "= " + mySum);
}
```

Inheritance





Script Includes Patterns

```
var Person = Class.create();
Person.prototype = {
    initialize: function(personName) {
        this.personName = personName;
    },
    say: function(message) {
        return this.personName + ': ' + message;
    },
    type: 'Person'
};
```

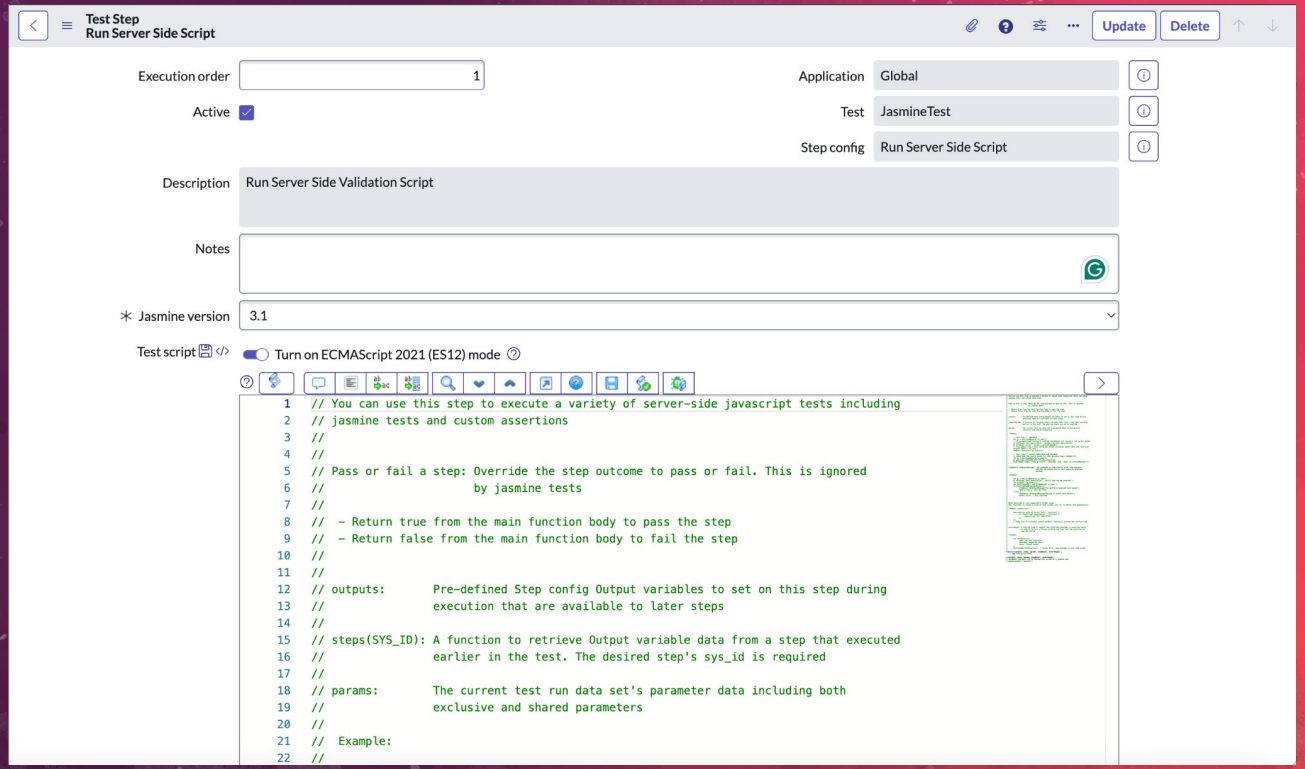
```
var Pirate = Class.create();
Pirate.prototype = Object.extendsObject(Person, {
    say: function(message) {
        return this.personName + ':' + message + ', yarr!';
    },
    type: 'Pirate'
});
```

Script Includes Patterns

```
var ChildSI = Class.create();
ChildSI.prototype = Object.extendsObject(ParentSI, {

    initialize: function() {
        gs.info("initialize: ChildSI method.");
        ParentSI.prototype.initialize.call(this);
    },

    childFunc: function() {
        gs.info("childFunc: ChildSI method.");
    },
    funcToBeOverriden: function() {
        gs.info("funcToBeOverriden: ChildSI method.");
    },


    type: 'ChildSI'
});
```

```
var ConstantsSI = Class.create();
ConstantsSI.myConstVar = "myConstVar";
ConstantsSI.prototype = {
    myVar: "myVar",
    initialize: function() {},
    myFunc: function() {
        gs.info("myFunc");
    },

    type: 'ConstantsSI'
};
ConstantsSI.prototype.varFunc = function() {
    gs.info("varFunc");
};
ConstantsSI.constFunc = function() {
    gs.info("constFunc");
};
```

Run Server Side Script



Suites: describe Your Tests

- The **describe** function is for **grouping related specs;** typically, each test file has one at the top level.
- The string parameter is for naming the collection of specs, and will be concatenated with specs to make a spec's full name.
  - This aids in finding specs in a large suite.
  - If you name them well, your specs read as full sentences in traditional BDD style.

Specs

- Specs Specs are defined by calling the global Jasmine function **it**, which, like describe takes a string and a function.
- The string is the title of the spec and the function is the spec, or test.
- A spec contains one or more expectations that test the state of the code.
    - An expectation in Jasmine is an assertion that is either true or false.
    - A spec with all true expectations is a passing spec.
    - A spec with one or more false expectations is a failing spec.

Expectations

- Expectations are built with the function **expect** which takes a value, called the actual.
- It is chained with a Matcher function, which takes the expected value.

Matchers

- Each matcher implements a boolean comparison between the actual value and the expected value.
- It is responsible for reporting to Jasmine if the expectation is true or false.
  - Jasmine will then pass or fail the spec.
- Any matcher can evaluate to a negative assertion by chaining the call to expect with a **not** before calling the matcher.
- Jasmine has a rich set of matchers included, you can find the full list in the API docs.

Setup and Teardown

- To help a test suite **DRY** up any duplicated setup and teardown code, Jasmine provides the global **beforeEach** , **afterEach** , **beforeAll** , and **afterAll** functions.
- As the name implies, the **beforeEach** function is **called once before each spec** in the describe in which it is called and the **afterEach** function is **called once after each spec** .
- The **beforeAll** function is **called only once before all the specs** in describe are run, and the **afterAll** function is **called after all specs finish** .
- beforeAll and afterAll can be used to speed up test suites with expensive setup and teardown.

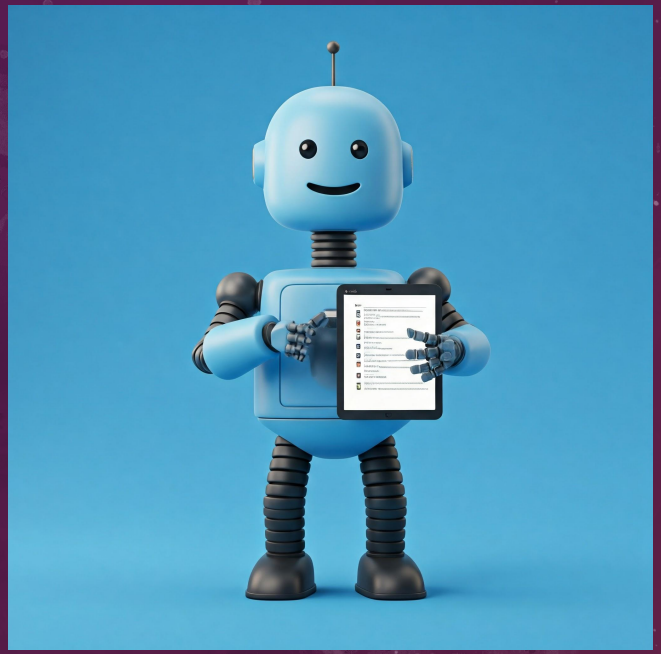
The this keyword

- A way to **share variables** between a **beforeEach** , **it**, and **afterEach** is through the this keyword.
- Each spec's beforeEach/it/afterEach has the this as the same empty object that is set back to empty for the next spec's beforeEach/it/afterEach.
- **Note:** If you want to use the this keyword to share variables, **you must use the function keyword and not arrow functions** .

Disabling Suites

- Suites can be disabled with the **xdescribe** function.
- These suites and any specs inside them are skipped when run and thus their results will show as pending.
- Suites can also be focused with the **fdescribe** function.
  - That means only fdescribe suits will run.

Pending Specs



- Pending specs do not run, but their names will show up in the results as pending.
- Any spec declared with **xit** is marked as pending.
- Any spec declared without a function body will also be marked pending in results.
- And if you call the function **pending** anywhere in the spec body, no matter the expectations, the spec will be marked pending.
  - A string passed to pending will be treated as a reason and displayed when the suite finishes.
- Tests can also be focused with the **fit** function.
  - That means only fit tests will run.

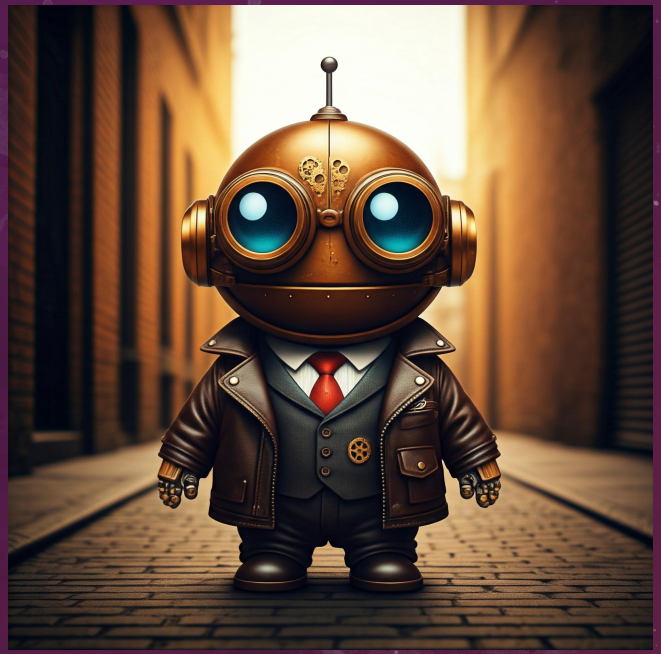
Manually failing a spec with fail

- The fail function causes a spec to fail.
- It can take a failure message or an Error object as a parameter.

Spies

- Jasmine has test double functions called **spies**.
- A spy can stub any function and tracks calls to it and all arguments.
- A spy **only exists in the describe or it block in which it is defined**, and will be removed after each spec.
- You can define what the spy will do when invoked with **and**.
- There are special matchers for interacting with spies.
  - The **toHaveBeenCalled** matcher will pass if the spy was called.
  - The **toHaveBeenCalledTimes** matcher will pass if the spy was called the specified number of times.
  - The **toHaveBeenCalledWith** matcher will return true if the argument list matches any of the recorded calls to the spy.

Spies: createSpy



- When there is not a function to spy on, jasmine.createSpy can create a "bare" spy.
- This spy acts as any other spy - tracking calls, arguments, etc; But there is no implementation behind it.

References

- http://prototypejs.org/
- https://beingfluid.github.io/ptototypejstest/
- Objects: Making It A Little Easier...
- Script Includes
- NOWCommunity Live Stream - Code Decoded - Using a Constants script include
- Live Coding Happy Hour for 2017-02-10 - Server Scripts with Automated Test Framework
- Unit testing your Javascript with jasmine
- https://jasmine.github.io/
- Unlearn Series - Unit Test Your Script Include Using ATF
- Learning Unit Testing with Jasmine
- https://beingfluid.github.io/ptototypejstest/code.html