

# Testing

# ¿Qué es?

Comprobación automática del código

- Organizada por casos
- Cada caso comprueba un aspecto
- Comparando el resultado obtenido con el esperado

# ¿Para qué sirve?

¡Para garantizar que todo funciona!

- Que el nuevo código es correcto
- Que no se ha roto nada de lo anterior
- Que una refactorización no ha introducido bugs

# ¿En JavaScript?

Una práctica que va penetrando poco a poco

- Aunque sigue sin estar muy extendida
- Necesaria para aplicaciones complejas
- En general, una garantía de calidad

# Testing

Hay muchos tipos de tests:

- Unitarios: comprueban un componente o una parte específica del código
- Integración: comprueban la interacción de componentes
- Aceptación: comprueban los requisitos del proyecto
- Regresión: comprueban la corrección de cambios
- etc...

# Tests Unitarios

La idea de test unitario es muy simple:

- Dado un componente del sistema
- Para cada caso posible
- Comprobar que se comporta de la manera adecuada

# Test Unitarios

```
var Contador = ProJS.Class.extend({  
  init: function() {  
    this.i = 0;  
  },  
  get: function() {  
    return this.i;  
  },  
  inc: function() {  
    this.i++;  
  },  
  dec: function() {  
    this.i--;  
  },  
  reset: function() {  
    this.i = 0;  
  }  
});
```

# Test Unitarios

¿Cómo podríamos comprobar, programáticamente, que Contador funciona bien?

Haciendo algo así:

➡ <http://jsbin.com/aluhid/1/edit>



# Test Unitarios

Es bastante tedioso!


- Mucha repetición de código similar
- Se puede abstraer bastante

# Test Unitarios

Segundo intento

- <http://jsbin.com/aluhid/4/edit>

# Test Unitarios



```
var ContadorTests = Test.extend({
  casos: {
    debe_empezar_a_cero: function(contador) {
      var i = contador.get();
      this.assertEqual(i, 0, "Empieza a %1".format(i));
    },
    // ...
  }
});
```

# Jasmine


Estupenda librería de testing

- Al estilo rspec
- Sencilla
- Potente
- <http://pivotal.github.com/jasmine/>

# Jasmine

¿Qué pinta tiene?

- <http://jsbin.com/emosif/4/edit>



```
describe("Conjunto de tests", function() {  
  it("debería ser un caso válido", function() {  
    expect(true).toBe(true);  
  });  
  it("debería ser un caso con error", function() {  
    expect(true).toBe(false);  
  });  
});
```

# Jasmine

Test del contador con Jasmine

- <http://jsbin.com/emosif/5/edit>

# Jasmine

## Test asíncronos

- ¿Cómo testearías que esta función llama al **cb** con **true**?

```
function asyncFn(cb) {  
  setTimeout(function() { cb(true); }, 250);  
}
```

- <http://jsbin.com/orulak/1/edit>

# Jasmine

```
describe("Test asíncrono", function() {  
  it("debería llamar al callback con true", function() {  
    var result,  
        callback = function(response) { result = response; };  
    ➡ runs(function() {  
        asyncFn(callback);  
    });  
    ➡ waitsFor(function() {  
        return result == true;  
    }, 300);  
    ➡ runs(function() {  
        expect(result).toBe(true);  
    });  
  });  
});
```



# Intermedio: Jasmine

¡Testea alguna de las funciones del tema anterior!

- La que te parezca más confusa
- Documentación y “matchers” de Jasmine en  
➡ <http://pivotal.github.com/jasmine/>

# Jasmine

Jasmine en la consola:

- Cambiar a **ConsoleReporter**
- Y un poco de magia funcional...

➡ <http://jsbin.com/udogag/3/edit>

# Jasmine

```
var lazyPrint = (function() {  
  var buffer = "",  
      print = function() {  
    console.log(buffer);  
    buffer = "";  
  };  
  print = debounce(print, 300);  
  return function(msg) {  
    buffer += msg;  
    print();  
  };  
})();
```

# Jasmine

¿Para qué sirve Jasmine en la consola?

- Dejar la página libre
- Poder cargar nuestro propio HTML
- ¡Testear interacciones e interfaces!

# Test de Integración (interfaz)

Comprobar que el UI funciona correctamente

- Simular la interacción del usuario disparando eventos DOM
- Observar el estado del programa inspeccionando el interfaz
- Asegurar la correcta integración de los componentes de la página

# Jasmine

Partimos de aquí:

➡ <http://jsbin.com/udogag/5/edit>

- Queremos testear que el intefaz funciona bien
- “Inc” incrementa el contador y el display
- “Dec” decrementa el contador y el display
- “Reset” lo pone a 0

# Jasmine

El resultado:

- ➡ <http://jsbin.com/udogag/6/edit>
- Salida de los test en la consola!

# Spam Mode: ON

Al escribir test JS acaba surgiendo un problema:

- ¡Los datos!
- ¿De dónde saco datos válidos para testear?
- ¿Del servidor?
  - No es fácil de conseguir modificar/resetear un set de datos cada vez que ejecuto un test
  - Dependencia del backend
- Lo ideal sería:
  - Factorías de datos (estilo FactoryGirl)
  - Simular la interacción con el servidor de forma inocua



# Solipsist.js

Solipsist.js es una librería auxiliar para testear

➡ <https://github.com/WeRelax/solipsist-js>

- Tests JS aislados
- Factorías
- Mocking de peticiones AJAX
- Otro uso: programar el frontend independiente del backend