

Promesas

Node.js y CPS

Node.js maneja la asincronía utilizando callbacks

- Continuation Passing Style
- Continuaciones explícitas como funciones
- “*Cuando termines, ejecuta esta otra función*”

Node.js y CPS

Los callbacks tienen muchas ventajas

- Muy fáciles de entender e implementar
- Familiares para el programador JavaScript
- Extremadamente flexibles (clausuras, funciones de primer orden, etc, ...)
- Un mecanismo universal de asincronía/continuaciones

Pero...

Node.js y CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

Node.js y CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

Node.js y CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
    if (exists) {
        fs.readFile("./hola.txt", function(err, data) {
            if (err) {
                // MANEJO DE ERROR
            } else {
                fs.writeFile("./copia.txt", data, function(err) {
                    if (err) {
                        // MANEJO DE ERROR
                    } else {
                        console.log("OK!");
                    }
                })
            }
        })
    } else {
        // MANEJO DE ERROR
    }
});
```

Node.js y CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
    if (exists) {
        fs.readFile("./hola.txt", function(err, data) {
            if (err) {
                // MANEJO DE ERROR
            } else {
                fs.writeFile("./copia.txt", data, function(err) {
                    if (err) {
                        // MANEJO DE ERROR
                    } else {
                        console.log("OK!");
                    }
                })
            }
        })
    } else {
        // MANEJO DE ERROR
    }
});
```

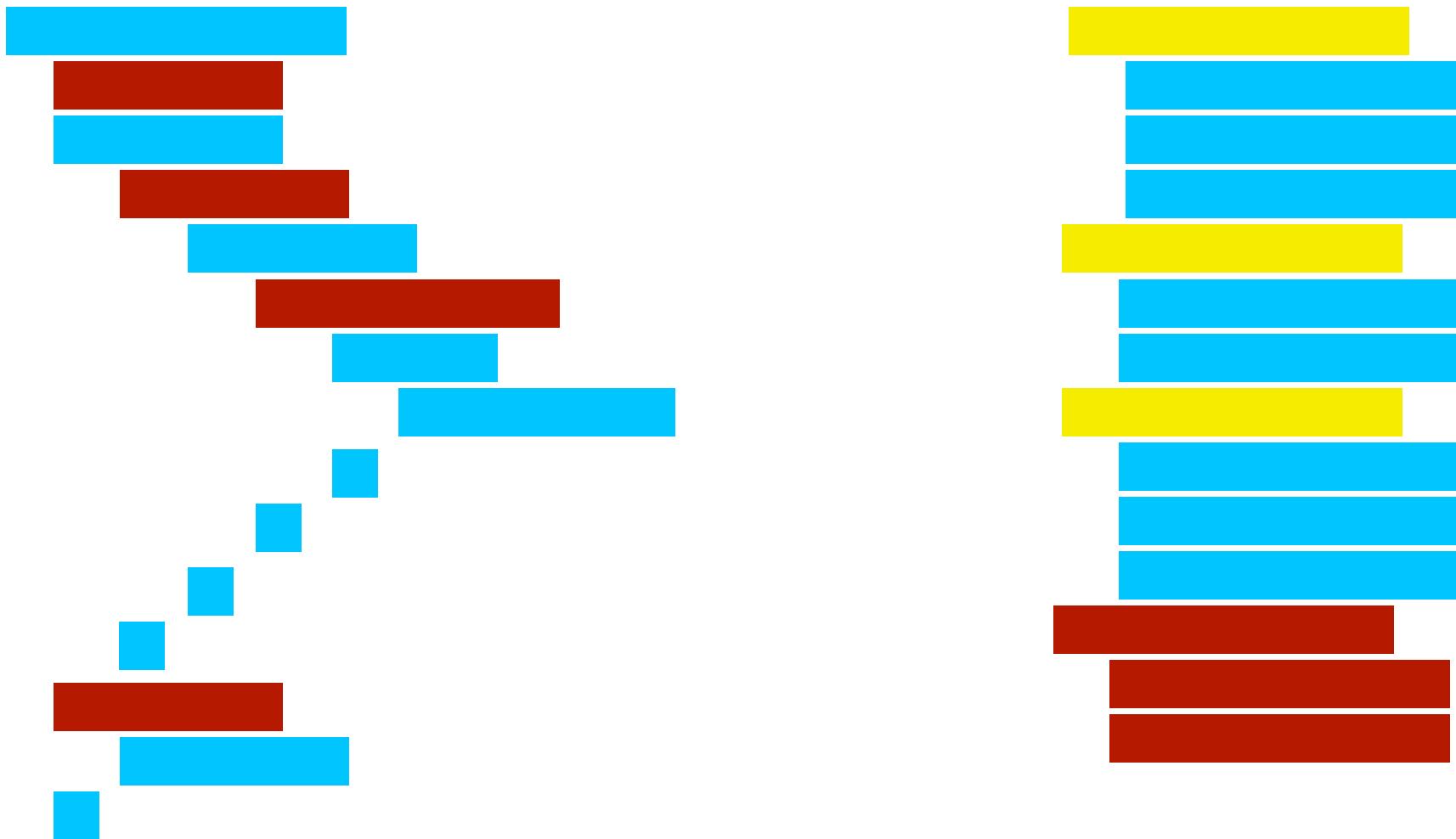
Pyramid of Doom
Callback Hell

Node.js y CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

CPS vs. Promises



Promesas

Una promesa representa el resultado *posible* de una operación asíncrona.

Existe un estándar independiente que las librerías e implementaciones nativas deben seguir

<https://promisesaplus.com>

Promesas

Una manera alternativa de modelar asincronía

- Construcción explícita del flujo de ejecución
- Separación en bloques consecutivos
- Manejo de errores más controlado
- Combinación de diferentes flujos asíncronos

Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```

Promesas

Una promesa = Un flujo de ejecución

```
promesa.then( function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
}  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
}  
.then(function() {  
    console.log("listo!");  
}  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

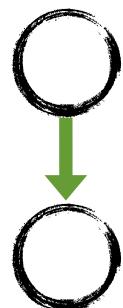
```
promesa.then(function() {  
  // bloque  
  return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
  // bloque  
  return writeFilePromise("./copia.txt");  
})  
.then(function() {  
  console.log("listo!");  
})  
.catch(function(err) {  
  // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

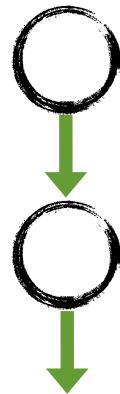
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

```
promesa.then( function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

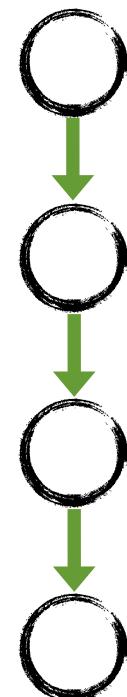
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

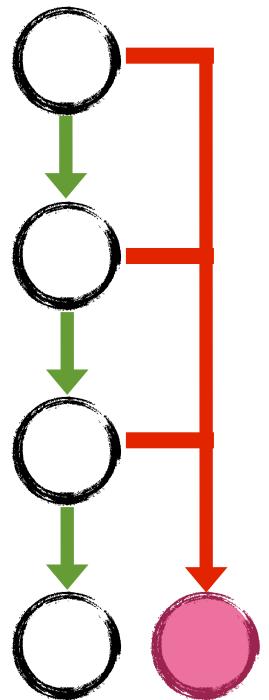
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

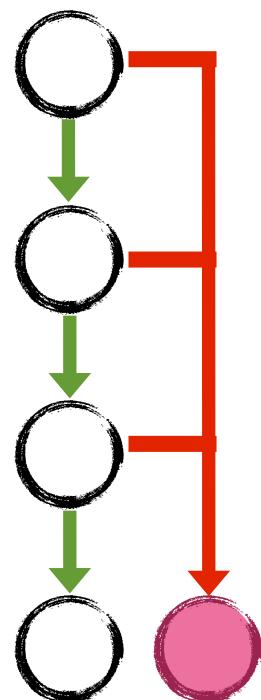
Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.catch(function(err) {  
    // MANEJO DEL ERROR  
});
```



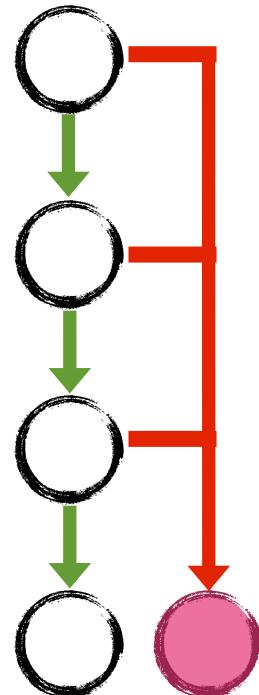
Promesas

Una promesa = Un flujo de ejecución



Promesas

¡Pero aún no hemos ejecutado nada! Solamente hemos construido el flujo



Promesas

¿Ventajas?

- Código mucho más ordenado y más legible
- Mejor control de errores
- **Podemos manipular el flujo**
 - Añadir nuevas etapas
 - Devolverlo en funciones
 - Pasarlo como parámetro
- **Podemos combinar varios flujos**

Promesas

```
function copyFile(from, to) {
  return readFilePromise(from);
  .then(function(data) {
    // bloque
    return writeFilePromise(to);
  });
}

copyFile("./hola.txt", "./copia.txt")
  .then(function() {
    return copyFile("./otraCosa.txt", "./copia2.txt");
  })
  .then(function() {
    console.log("listo!");
  })
  .catch(function(err) {
    console.log("Oops!");
  })
}
```

Promesas

```
function copyFile(from, to) {
  return readFilePromise(from);
  .then(function(data) {
    // bloque
    return writeFilePromise(to);
  });
}

copyFile("./hola.txt", "./copia.txt")
  .then(function() {
    return copyFile("./otraCosa.txt", "./copia2.txt");
  })
  .then(function() {
    console.log("listo!");
  })
  .catch(function(err) {
    console.log("Oops!");
  })
}
```

Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  });  
}  
}
```



```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .catch(function(err) {  
    console.log("Oops!");  
  })
```

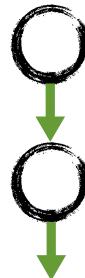
Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
  
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .catch(function(err) {  
    console.log("Oops!");  
  })
```

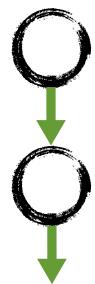


Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  });  
}  
}
```



```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .catch(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
  
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .catch(function(err) {  
    console.log("Oops!");  
  })
```



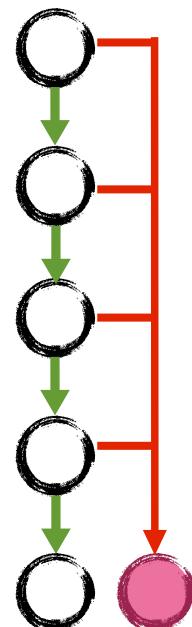
Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
  
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .catch(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  });  
}  
  
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .catch(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

.then(success, [error])

- Concatena bloques
- El nuevo bloque (success)...
 - Sólo se ejecuta si el anterior se ha ejecutado sin errores
 - Recibe como parámetro el resultado del bloque anterior
 - Devuelve el valor que se le pasará el siguiente bloque
 - ➡ Si es un **dato inmediato**, se pasa tal cual
 - ➡ Si es una **promesa**, se resuelve antes de llamar al siguiente bloque
- El segundo parámetro pone un manejador de error
 - Equivalente a llamar a .catch(error)
- **.then(...)** siempre devuelve una nueva promesa

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");  
  
promesa = promesa.then(function(data) {  
    console.log("Contenido del fichero: ", data);  
, function(err) {  
    console.log("Ooops!", err);  
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

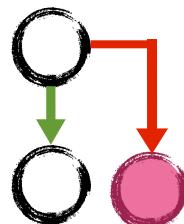
```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

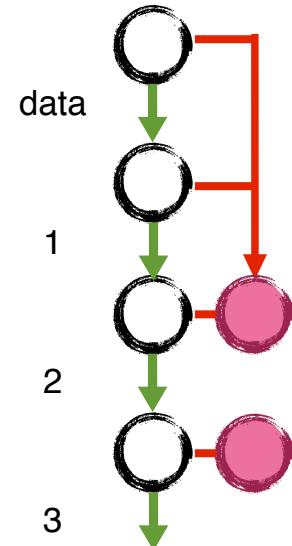
promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```



Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa.then(function(data) {
    return 1;
})
.then(function(uno) {
    return 2;
}, function(err) {
    console.log("Oh, oh...");
})
.then(function(dos) {
    return 3;
})
.catch(function(err) {
    console.log("Oops!");
});
```

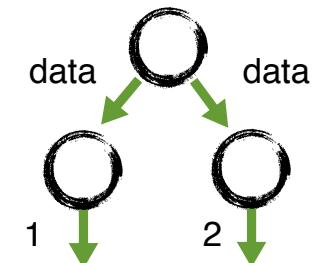


Promesas

```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});
```



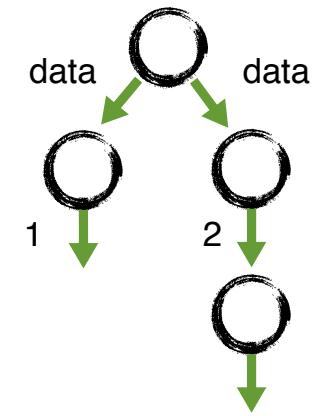
Promesas

```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});

promesa3.then(function(dos) {
    console.log("Ping!");
});
```



Promesas

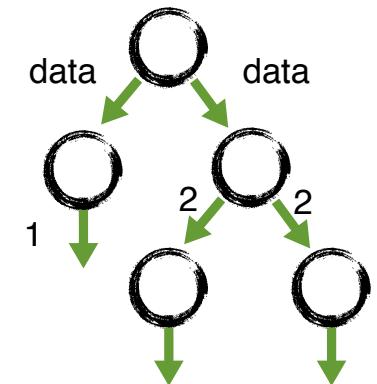
```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});

promesa3.then(function(dos) {
    console.log("Ping!");
});

promesa3.then(function(dos) {
    console.log("Pong!");
});
```



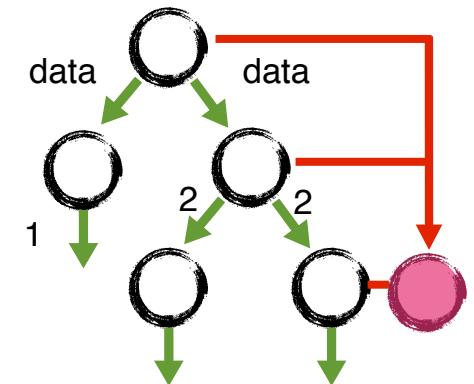
Promesas

```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});

promesa3.then(function(dos) {
    console.log("Ping!");
});
promesa3.then(function(dos) {
    console.log("Pong!");
}, function(err) {
    console.log("Oh, oh...");
});
```



Promesas

¿Cómo creamos una promesa? Con el constructor:

```
var miPromesa = new Promise(function(resolve, reject) {  
    //...  
});
```

Creamos una nueva promesa y su único argumento es una función que recibe dos funciones como argumentos, **resolve** y **reject**.

Promesas

Dentro de la función podemos ejecutar cualquier código que queramos. Usaremos **resolve(valor)** para devolver el resultado final de la promesa, o bien **reject(valor)** para rechazar finalmente la promesa.

Promesas

Una promesa tiene **tres estados**:

- Pendiente
- Resuelta - cuando llamemos a **resolve**
- Rechazada - cuando llamemos a **reject**

Promesas

Podemos usar “new Promise(..)” ...

- De forma nativa si es node.js ≥ 0.12
- Con una librería de promesas en node.js < 0.12 o en el navegador
- Las más utilizadas: **bluebird**, **q**
- Tenemos que instalarlas con npm!

Promesas

Ejemplo: una promesa que resuelve con un valor inmediatamente.

```
var miPromesa = new Promise(function(resolve, reject) {  
    resolve(5);  
});  
  
miPromesa  
    .then(function(value) {  
        return value*2;  
    })  
    .then(function(value) {  
        console.log('El resultado es', value);  
    });
```

Promesas

Ejemplo: una promesa que se rechaza con un error.

```
var miPromesa = new Promise(function(resolve, reject) {
    reject(new Error("Boom!"));
});

miPromesa
    .then(function(value) {
        return value*2;
    })
    .then(function(value) {
        console.log('El resultado es', value);
    })
    .catch(function(err) {
        console.log('Algo ha ido mal', err);
}) ;
```

Promesas

Atajos para **resolve**: usamos el método estático **resolve** directamente sobre Promise.

→ `var miPromesa = Promise.resolve(5);`

```
miPromesa
  .then(function(value) {
    return value*2;
  })
  .then(function(value) {
    console.log('El resultado es', value);
  })
  .catch(function(err) {
    console.log('Algo ha ido mal', err);
  });
}
```

Promesas

Atajos para **reject**: usamos el método estático **reject** directamente sobre Promise.

→ `var miPromesa = Promise.reject(new Error('BOOM!'));`

```
miPromesa
  .then(function(value) {
    return value*2;
  })
  .then(function(value) {
    console.log('El resultado es', value);
  })
  .catch(function(err) {
    console.log('Algo ha ido mal', err);
  });
}
```

Promesas

Ejemplo: una promesa que resuelve con un valor un segundo más tarde.

```
var miPromesa = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(5);
    }, 1000);
});

miPromesa
    .then(function(value) {
        return value*2;
    })
    .then(function(value) {
        console.log('El resultado es', value);
    })
    .catch(function(err) {
        console.log('Algo ha ido mal', err);
    });
}
```

Promesas

Ventajas: cualquier error no controlado dentro de
new Promise la rechaza automáticamente!!!

```
var miPromesa = new Promise(function(resolve, reject) {  
    ➔ blublublu  
});  
  
miPromesa  
    .then(function(value) {  
        return value*2;  
    })  
    .then(function(value) {  
        console.log('El resultado es', value);  
    })  
    .catch(function(err) {  
        console.log('Algo ha ido mal', err);  
    });
```

Promesas

Ventajas: cualquier error no controlado dentro de
new Promise la rechaza automáticamente!!!

```
var miPromesa = new Promise(function(resolve, reject) {  
    blublublu  
});  
  
miPromesa  
    .then(function(value) {  
        return value*2;  
    })  
    .then(function(value) {  
        console.log('El resultado es', value);  
    })  
    .catch(function(err) {  
        console.log('Algo ha ido mal', err);  
    });
```



Algo ha ido mal [ReferenceError: blublublu is not defined]

Promesas: ejercicios

Vamos a empezar a trastear con promesas...

- Para asentar conceptos
- (y perder el miedo)

Promesas: ejercicios

```
var promise = new Promise(function(resolve,reject){  
});  
  
promise.then(function() {  
    return "hola";  
})  
.then(function(msg) {  
    console.log(msg);  
    return "mundo";  
})  
.then(function(msg) {  
    console.log(msg);  
});
```

Promesas: ejercicios

```
var promise = new Promise(function(resolve,reject){  
});  
  
promise.then(function() {  
    return "hola";  
})  
.then(function(msg) {  
    console.log(msg);  
    return "mundo";  
})  
.then(function(msg) {  
    console.log(msg);  
});
```

¿Qué se muestra por consola al ejecutar esto?

Promesas: ejercicios

```
var promise = new Promise(function(resolve,reject){  
});
```

```
promise.then(function() {  
    return "hola";  
})  
.then(function(msg) {  
    console.log(msg);  
    return "mundo";  
})  
.then(function(msg) {  
    console.log(msg);  
});
```

¿Por qué?

Promesas: ejercicios

Una promesa = un flujo de ejecución

- Configuramos un árbol de flujos de ejecución
- Añadimos bloques a promesas
- Pero... **¿Cuándo se empieza a ejecutar?**

Promesas: ejercicios

Una promesa = un flujo de ejecución

- Configuramos un árbol de flujos de ejecución
- Añadimos bloques a promesas
- Pero... ¿Cuándo se empieza a ejecutar?
- **Cuando se resuelva la primera promesa del árbol**

Promesas: ejercicios

Recuerda: las promesas se **resuelven** o se **rechazan**

Si se resuelven:

- Se resuelven a un valor (si es un bloque, su valor de retorno)
- Representan el estado “*OK, puede seguir el siguiente*”

Si se rechazan:

- Representan un error
- La ejecución cae hasta el siguiente manejador de errores
- Se saltan todos los estados desde el error hasta el manejador

Promesas: ejercicios

```
var promise = new Promise(function(resolve,reject){  
    setTimeout(function(){  
        resolve();  
    }, 2000);  
});  
  
promise.then(function() {  
    return "hola";  
})  
.then(function(msg) {  
    console.log(msg);  
    return "mundo";  
})  
.then(function(msg) {  
    console.log(msg);  
});
```

Promesas: ejercicios

```
var promise = new Promise(function(resolve,reject){  
    setTimeout(function(){  
        reject("Fallo");  
    }, 1000);  
});
```

```
promise.then(function() {  
    return "hola";  
})  
.then(function(msg) {  
    console.log(msg);  
    return "mundo";  
})  
.catch(function(err) {  
    console.log(err);  
});
```

Promesas: ejercicios

Otra manera de rechazar una promesa es lanzar una excepción desde el interior de un bloque

```
promise.then(function() {  
    return "hola";  
})  
.then(function(msg) {  
    console.log(msg);  
    throw new Error("Oh, oh...");  
    return "mundo";  
})
```

Promesas: ejercicios

¿Qué muestra por consola lo siguiente?

```
var promise = Promise.resolve(45);

promise.then(function() {
    return "holo";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("oops");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.catch(function(err) {
    console.log('Fail!');
    return "mal";
})
.then(function(msg) {
    console.log(msg);
}) ;
```

Promesas: ejercicios

Propagación de promesas (coordinar 2 diferentes)

```
var promise = Promise.resolve(42);
promise2 = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(12);
    }, 2000);
}) ;

promise.then(function(val) {
    console.log("promise:", val);
    promise2.then(function(val) {
        console.log("promise2:", val);
    }) ;
}) ;
```

Promesas: ejercicios

Propagación de promesas

```
var promise = Promise.resolve(42);
promise2 = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(12);
    }, 2000);
}) ;

promise.then(function(val) {
    console.log("promise:", val);
promise2.then(function(val) {
    console.log("promise2:", val);
}) ;
}) ;
```



Promesas: ejercicios

Propagación de promesas - bien hecho

```
var promise = Promise.resolve(42);
promise2 = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(12);
    }, 2000);
}) ;

promise.then(function(val) {
    console.log("promise:", val);
    return promise2;
})
.then(function(val) {
    console.log("promise2:", val);
}) ;
```



Promesas: ejercicios

¿Y al revés?

```
var promise2 = Promise.resolve(42);
promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve(12);
    }, 2000);
});
```

```
promise.then(function(val) {
    console.log("promise:", val);
    return promise2;
})
.then(function(val) {
    console.log("promise2:", val);
});
```

Diferidos

A veces queremos aplazar o delegar la resolución de una promesa. Podemos usar **defer** (aplazar) para obtener dos objetos independientes:

- La **promesa** en sí
 - Interfaz limitada a la *construcción de flujos*
 - .then, .fail y algún método más
- Su **diferido**
 - Interfaz limitada a *controlar el estado*
 - .reject y .resolve

Diferidos

```
//solo si no hay Promise nativo
//var Promise = require('bluebird');

var defer = Promise.defer(),
      promise = defer.promise;

promise.then(function(val) {
    console.log('Valor:', val);
})
  .catch(function(err) {
    console.log('Error:', err);
});

defer.resolve("hey!");
```

Diferidos

```
//solo si no hay Promise nativo  
//var Promise = require('bluebird');
```

```
var defer = Promise.defer(),  
    promise = defer.promise;
```

```
promise.then(function(val) {  
    console.log('Valor:', val);  
})  
.catch(function(err) {  
    console.log('Error:', err);  
});
```

```
defer.resolve("hey!");
```

Diferidos

```
//solo si no hay Promise nativo  
//var Promise = require('bluebird');
```

```
var defer = Promise.defer(),  
    promise = defer.promise;
```

```
promise.then(function(val) {  
    console.log('Valor:', val);  
})  
.catch(function(err) {  
    console.log('Error:', err);  
});
```

```
defer.reject("BOOM");
```

Diferidos

Cumplen dos roles diferentes:

- Diferido lo controla el **gestor** del recurso/proceso que se está modelando
- Promesa es el interfaz para que el **consumidor** del recurso pueda construir el flujo que necesita

Promesas

Un ejemplo realista: `readFilePromise(file)`

- Implementa la función
- Basándote en `fs.readFile(ruta, callback)`
- Devuelve una promesa
- Se resuelve con el contenido del fichero
- O se rechaza con el error

Promesas

```
function readFilePromise(filePath) {  
  // ???  
}  
  
readFilePromise("./hola.txt").then(function(contenido) {  
  console.log("contenido:", contenido.toString());  
, function(err) {  
  console.log("ERROR!", err);  
});
```

Promesas

```
var fs = require("fs");
//Sin diferidos
function readFilePromise(filePath) {
    return new Promise(function(resolve, reject) {
        fs.readFile(filePath, function(err, data) {
            if(err) {
                return reject(err);
            }
            resolve(data);
        });
    });
}

readFilePromise("./hola.txt")
.then(function(contenido) {
    console.log("contenido:", contenido.toString());
})
.catch(function(err) {
    console.log("ERROR!", err);
});
```

Promesas

```
var fs = require("fs");
//Con diferidos
function readFilePromiseDefer(filePath) {
    var defer = Promise.defer();          Gestor del recurso
    fs.readFile(filePath, function(err, data) {
        err ? defer.reject(err) : defer.resolve(data);
    })
    return defer.promise;
}
```

```
readFilePromise("./hola2.txt")
    .then(function(contenido) {           Consumidor del recurso
        console.log("contenido:", contenido.toString());
    })
    .catch(function(err) {
        console.log("ERROR!", err);
    });
}
```

Promesas

Es preferible usar el constructor de Promise en lugar de defer...

Porque cualquier error no controlado dentro de **readFilePromise** rechazaría la promesa automáticamente.

Haz la prueba, pon **fs.readFoo** en la versión con diferido y en la versión “estándar”.

Promesas: paréntesis

Intenta modificar el servidor de ficheros estáticos para que utilice promesas:

- fileExistsPromise
- readFilePromise

Promesas: combinaciones

bluebird es una librería muy potente

- Trae un montón de funcionalidad
- Muy recomendable leer la documentación
- <https://github.com/petkaantonov/bluebird/blob/master/API.md>
- Nosotros vamos a ver dos cosas esenciales:
 - Combinación de promesas en paralelo (no necesaria librería si se tiene Promise nativo)
 - Adaptadores de llamadas con formato Node.js (necesaria librería)

Promise.all([promise, promise, ...])

Crea una promesa que representa al conjunto

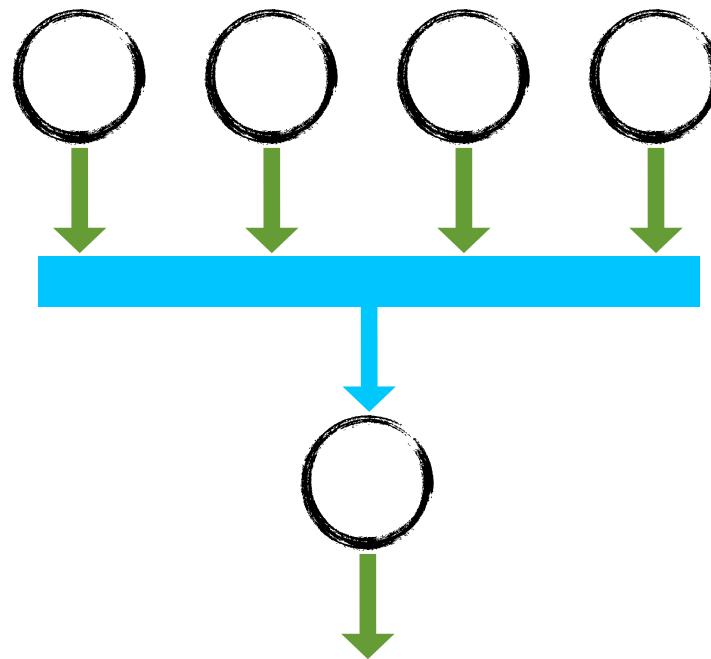
- La nueva promesa se resuelve cuando todas las del conjunto se hayan resuelto
- Su valor es un array con los valores de las promesas
- Si una del conjunto es rechazada, la nueva promesa también es rechazada

Promise.all([promise, promise, ...])

```
var p1 = new Promise(function(resolve) {  
    resolve('Promises');  
});  
var p2 = new Promise(function(resolve) {  
    setTimeout(function() {  
        resolve('are cool');  
    }, 500);  
});
```

→ `Promise.all([p1, p2]).then(function(values) {
 console.log('Todas resueltas!');
 console.log(values); //['Promises', 'are cool']
});`

Promise.all([promise, promise, ...])



Promise.all(...)

Escribe una función `readAllFiles(...)` que lea varios ficheros a la vez y devuelva un array con los contenidos (si todos se han podido leer correctamente).

Adaptadores para node.js

Casi todos los módulos de node.js que realizan una operación asíncrona siguen el mismo convenio (nuestro `readFilePromise` anterior):

- Reciben un callback como ultimo parametro
- El callback recibe 1 o más parámetros:
 - El primero indica si ha habido un error
 - ➔ Valor *falsy* => todo OK!
 - ➔ Valor *truthy* => el error

Adaptadores para node.js

Por ej, escribe la función

`fsStatPromise(filePath)`

- recibe una ruta como parámetro
- Devuelve una promesa con el resultado de llamar a `fs.stat()`

Adaptadores para node.js

Otro: escribe la función fsDirPromise(filePath)

- recibe una ruta como parámetro
- Devuelve una promesa con el resultado de llamar a `fs.readdir()`

Adaptadores para node.js

Surge un patrón común:

```
var fs = require('fs');

function fsDirPromise(path) {
  return new Promise(function(resolve, reject) {
    fs.readdir(path, function(err, files) {
      return err? reject(err) : resolve(files);
    });
  });
}
```

Adaptadores para node.js

Surge un patrón común:

```
var fs = require('fs');

function fsDirPromise(path) {
  return new Promise(function(resolve, reject) {
    fs.readdir(path, function(err, files) {
      return err? reject(err) : resolve(files);
    });
  });
}
```

Adaptadores para node.js

Escribe una función `nodefcall`

- Que reciba una función con “formato node”
- Devuelva una promesa
- Si la función devuelve error, la promesa falla
- Si la función se ejecuta bien, resuelve la promesa con todos los parámetros que haya recibido la función (sin el error)

Adaptadores para node.js

```
function nodefcall(fn) {  
    // rellena el hueco!  
}  
  
var fs = require("fs");  
  
nodefcall(fs.readdir, ".")  
    .then(function(result) {  
        console.log(result);  
    });  
  
nodefcall(fs.stat, ".")  
    .then(function(result) {  
        console.log(result);  
    });
```

Adaptadores para node.js

Escribe una función `nodeInvoke`

- Que reciba
 1. un objeto
 2. el nombre de un método a invocar
 3. parámetros para pasarle al método
- Devuelva una promesa
- Si la función devuelve error, la promesa falla
- Si la función se ejecuta bien, resuelve la promesa con todos los parámetros que haya recibido la función (sin el error)

Adaptadores para node.js

Las librerías externas ya proporcionan estos métodos, por ejemplo:

- Q (npm install q)
 - ➔ `Q.ninvoke(context, method, arg [,args] ...)`
 - Ejecuta la función y devuelve una promesa
- bluebird (npm install bluebird)
 - ➔ `Promise.promisify(function, [context])`
 - Devuelve una nueva función que devuelve promesas

Q.ninvoke(ctx, method, arg, [arg])

Adaptador para convertir llamadas Node.js en promesas

```
var Q = require("q"),
    fs = require("fs");

Q.ninvoke(fs, "readFile", "./hola.txt")
  .then(function(data) {
    console.log("Contenido: ", data.toString());
  })
  .fail(function(err) {
    console.log("Oops!", err);
  });
});
```

bluebird.promisify(fn, [context])

Adaptador para convertir llamadas Node.js en
llamadas que devuelven promesas

```
var Promise = require("bluebird"),  
    fs = require("fs");  
var readFileAsync = Promise.promisify(fs.readFile);  
  
readFileAsync("./hola.txt")  
.then(function(data) {  
    console.log("Contenido: ", data.toString());  
})  
.fail(function(err) {  
    console.log("Oops!", err);  
});
```

Adaptadores para node.js

Bluebird además nos da la opción de *promisificar* un objeto entero:

```
Promise.promisifyAll(fs); //una vez en toda la app
```

Eso creará una nueva versión de cada función, con el sufijo *Async*

```
fs.readFileAsync("foo.txt").then(...)
```

¡A teclear!

Vamos el primer ejercicio complicadillo: un servidor de ficheros versionado

- La herramienta que hemos estado utilizando para compartir código
- Con promesas
- (Si alguien se atreve, que lo intente hacer sin promesas...)

Necesitas saber...

Manejar rutas: require("path")

- path.resolve(base, ruta): ruta relativa a ruta absoluta (partiendo de base)
- path.relative(base, ruta): ruta absoluta a ruta relativa (desde base)

Necesitas saber...

fs.readdir(ruta): Listar ficheros de un directorio

- Devuelve un array de strings con los nombres de los ficheros
- No es recursivo
- No hay manera de saber si una entrada es un fichero o un directorio

```
var fs = require("fs"),
    Q = require("q");

Q.ninvoke(fs, "readdir", ".").then(function(list) {
    console.log(list);
})
```

Necesitas saber...

fs.stat(ruta): Info sobre un fichero/directorio

- stats.isDirectory(): true si es un directorio
- stats.mtime: Date de la última modificación

```
var fs = require("fs"),
    Q = require("q");

Q.ninvoke(fs, "stat", ".").then(function(stats) {
    console.log("es dir?", stats.isDirectory());
    console.log("última modificación:", stats.mtime);
})
```

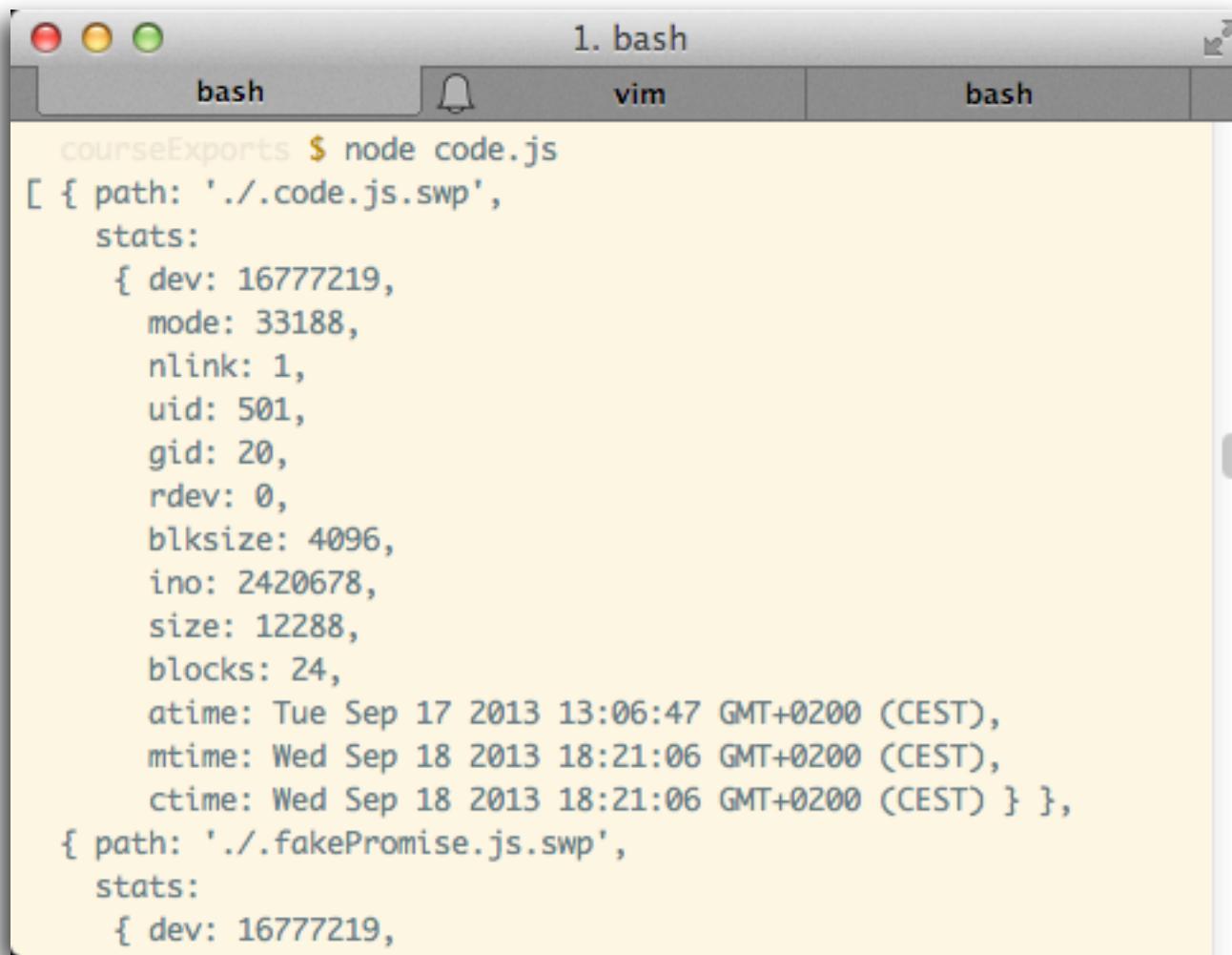
Primer paso: listado recursivo

Escribe una función `listAllFiles(ruta)` que:

- Devuelva una promesa
- La promesa se resuelva con un listado recursivo de todos los ficheros que hay dentro del directorio
- Para cada fichero, genere un objeto del tipo
`{path: “/ruta/absoluta.txt”, stats: statsDelFichero}`

```
listAllFiles(".").then(function(list) {  
    console.log(list);  
})  
.done()
```

Primer paso: listado recursivo



The screenshot shows a Mac OS X desktop environment with a terminal window titled "1. bash". The window has three tabs at the top: "bash" (selected), "vim", and "bash". The terminal output is as follows:

```
courseExports $ node code.js
[ { path: './.code.js.swp',
  stats:
   { dev: 16777219,
     mode: 33188,
     nlink: 1,
     uid: 501,
     gid: 20,
     rdev: 0,
     blksize: 4096,
     ino: 2420678,
     size: 12288,
     blocks: 24,
     atime: Tue Sep 17 2013 13:06:47 GMT+0200 (CEST),
     mtime: Wed Sep 18 2013 18:21:06 GMT+0200 (CEST),
     ctime: Wed Sep 18 2013 18:21:06 GMT+0200 (CEST) } ],
  { path: './.fakePromise.js.swp',
    stats:
     { dev: 16777219,
```

Segundo paso: listener

Función somethingChanged(ruta):

- Devuelve true si algún fichero ha sido modificado desde la última vez que se invocó
- Impleméntalo utilizando stats.mtime como referencia

Utilizando esa función, escribe un “demonio” que monitorize un directorio y escriba un mensaje por consola cada vez que hay cambios

Tercer paso: volcado a memoria

Función readAllFiles(ruta):

- Completa el resultado de listAllFiles()
añadiendo una tercera propiedad “contents” con los contenidos del fichero

Haz que el demonio lea todos los ficheros si detecta algún cambio y guarda el resultado en posiciones consecutivas de un array. Este va a ser nuestro control de versiones.

Cuarto paso: sirve los datos

El interfaz web tiene las siguientes rutas:

- /: listado de versiones
- /list?version=<n>: listado de ficheros de la versión n
- /ruta/al/fichero?version=<n>: busca el fichero con la ruta correspondiente en la versión n y lo sirve
- la versión “latest” siempre apunta a la versión más reciente

Virguerías optionales

Escribe un módulo simpleRoute de modo que podamos definir rutas así:

```
var routes = require("./simpleRoute");

routes.get("/", function(req, res) {
})

routes.get("/list", function(req, res) {
})

routes.default(function(req, res) {
})
```

Virguerías optionales

Además, simpleRoute modifica el parámetro req.url y lo sustituye por la url parseada

```
var routes = require("./simpleRoute");

routes.get("/list", function(req, res) {
  console.log(req.url.pathname);
  console.log(req.url.query.version);
})
```