

# **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**

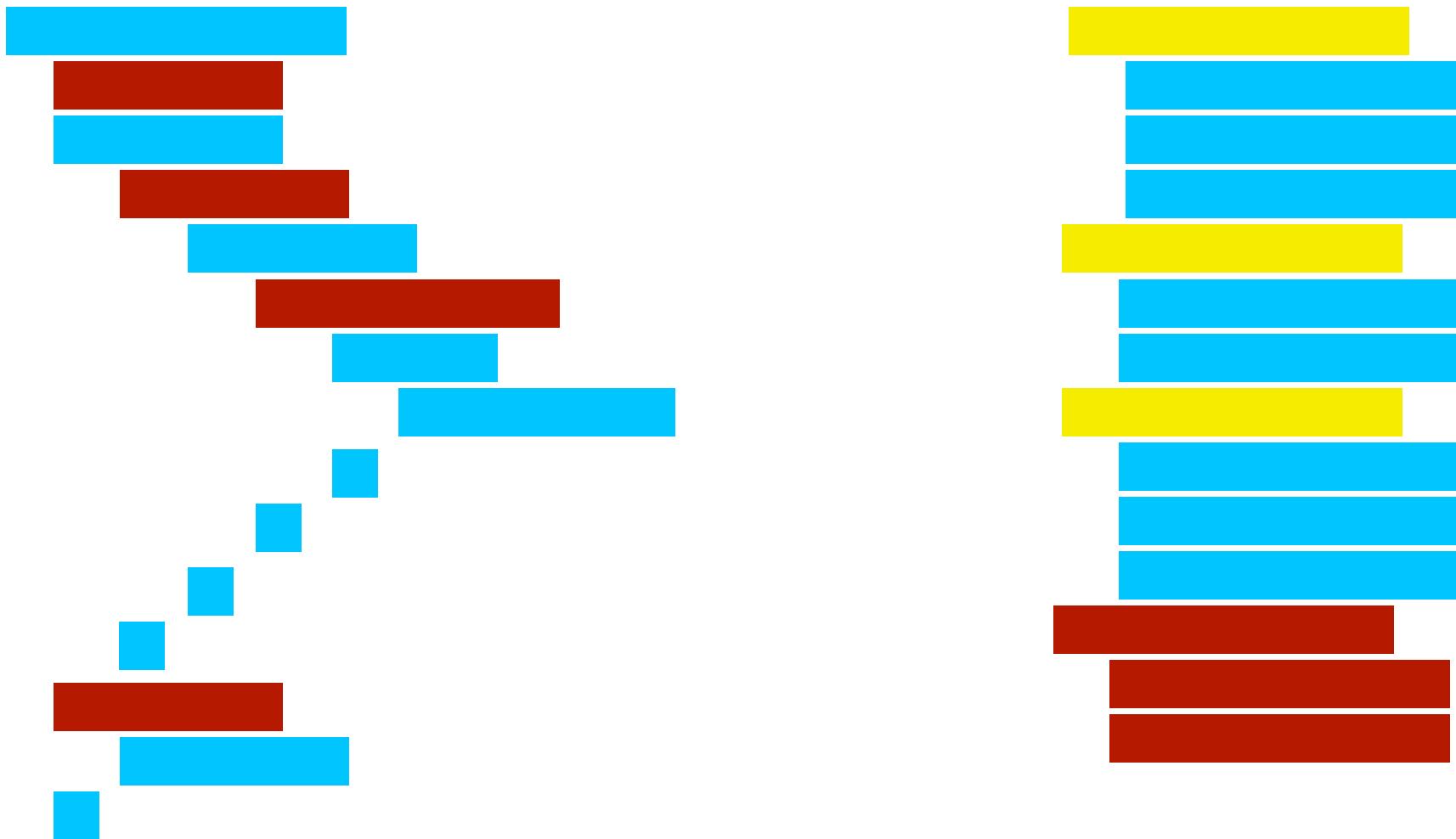
*Callback Hell*

# 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 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!");  
})  
.fail(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!");  
}  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



# Promesas

Una promesa = Un flujo de ejecución

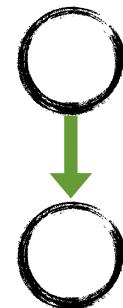
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



# Promesas

Una promesa = Un flujo de ejecución

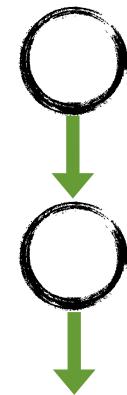
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



# 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!");  
})  
.fail(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!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



# Promesas

Una promesa = Un flujo de ejecución

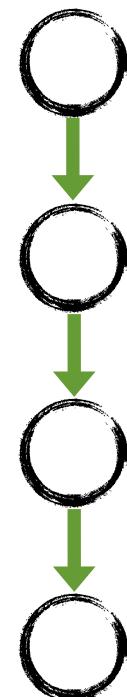
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



# Promesas

Una promesa = Un flujo de ejecución

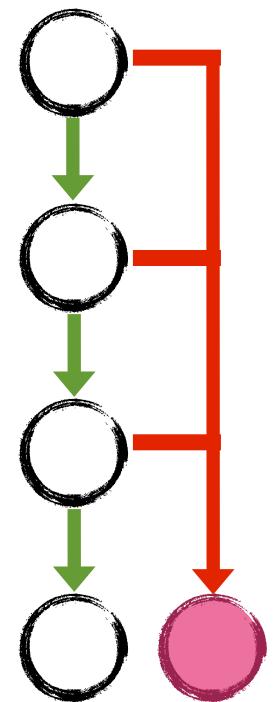
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



# Promesas

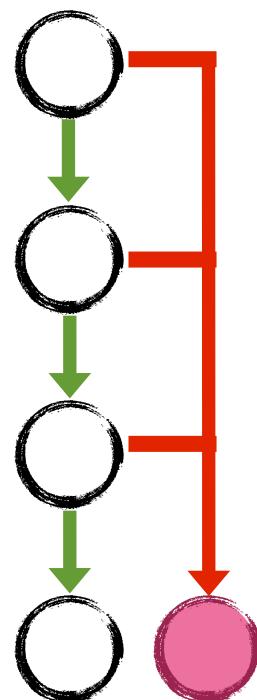
Una promesa = Un flujo de ejecución

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



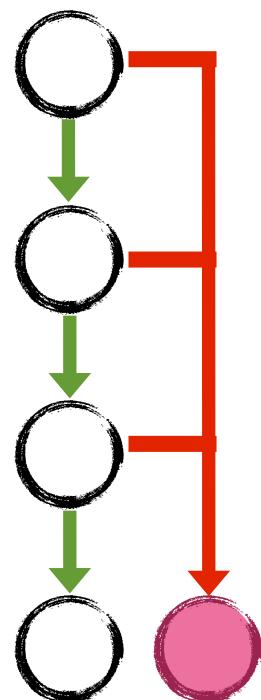
# Promesas

Una promesa = Un flujo de ejecución



# Promesas

¡Pero aún **no hemos ejecutado nada!** Solamente hemos construído 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!");
  })
  .fail(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!");  
  })  
  .fail(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!");  
})  
.fail(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!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



# Promesas

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



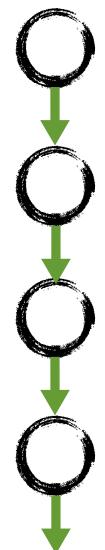
```
copyFile("./hola.txt", "./copia.txt")  
.then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    console.log("Oops!");  
})
```



# 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!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



# Promesas

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

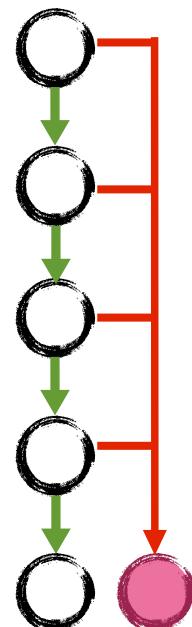
```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



# 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!");  
  })  
  .fail(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 .fail(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

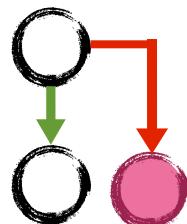
```
var promesa = readFilePromise("./hola.txt");

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

# Promesas

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

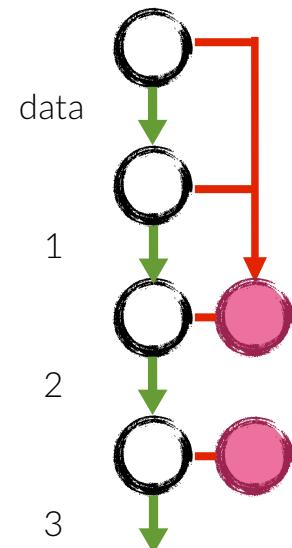
promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```



# 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;
})
.fail(function(err) {
    console.log("Oops!");
});
```

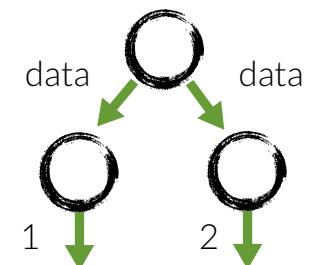


# Promesas

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

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

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



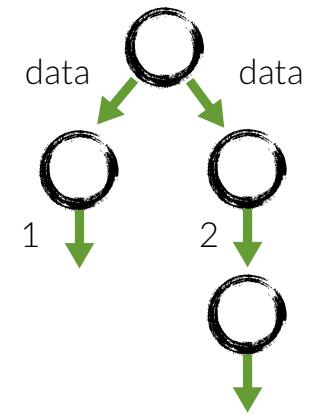
# Promesas

```
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

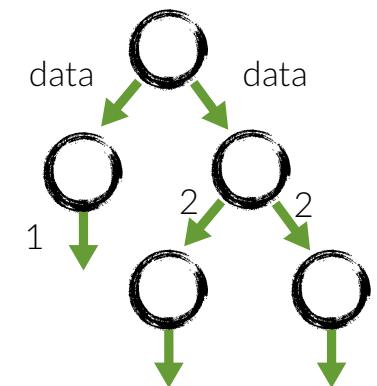
```
var promesa = readFilePromise("./hola.txt");

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!");
});
```



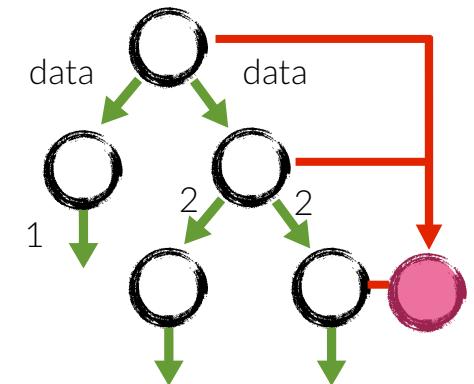
# Promesas

```
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: walled garden

Vamos a empezar a trastear con promesas...

- Pero, de momento, con una librería de mentira
- Para asentar conceptos
- (y perder el miedo)

# Promesas: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

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

# Promesas: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

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

# Promesas: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

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: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

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

¿Por qué?

# Promesas: walled garden

Una promesa = un flujo de ejecución

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

# Promesas: walled garden

Una promesa = un flujo de ejecución

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

# Promesas: walled garden

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: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

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

promise.resolve(42);
```

# Promesas: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

promise.then(function() {
    return "holo";
})
.then(function(msg) {
    console.log(msg);
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

promise.reject(new Error("Oops!"));
```

# Promesas: walled garden

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: walled garden

```
var fakePromise = require("./fakePromise");

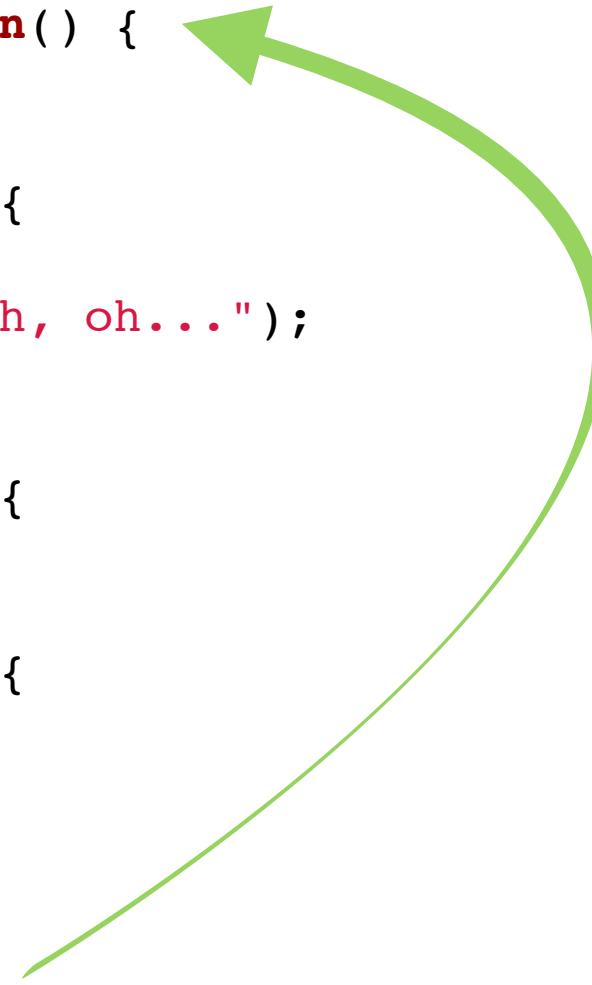
var promise = fakePromise.gimmePromise();

promise.then(function() {
    return "holá";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("Oh, oh...");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

promise.resolve(42);
```

# Promesas: walled garden

```
var fakePromise = require("./fakePromise");  
  
var promise = fakePromise.gimmePromise();  
  
promise.then(function() {  
    return "holo";  
})  
.then(function(msg) {  
    console.log(msg);  
    throw new Error("Oh, oh...");  
    return "mundo";  
})  
.then(function(msg) {  
    console.log(msg);  
})  
.fail(function(err) {  
    console.log(err);  
    return "MAL!"  
})  
  
promise.resolve(42);
```



# Promesas: walled garden

```
var fakePromise = require("./fakePromise");
```

```
var promise = fakePromise.gimmePromise();
```

```
promise.then(function() {  
    return "holá";  
})  
.then(function(msg) {  
    console.log(msg);  
    throw new Error("Oh, oh...");  
    return "mundo";  
})  
.then(function(msg) {  
    console.log(msg);  
})  
.fail(function(err) {  
    console.log(err);  
    return "MAL!"  
})  
  
promise.resolve(42);
```



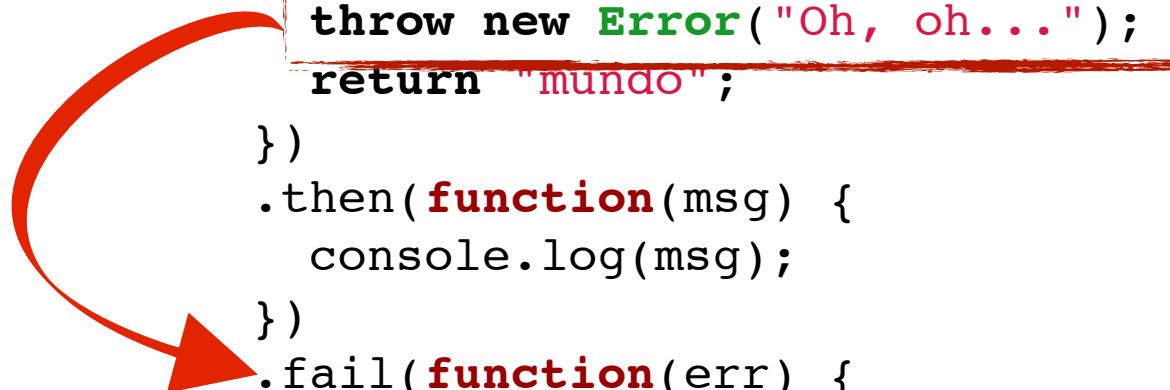
# Promesas: walled garden

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise();

promise.then(function() {
    return "holo";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("Oh, oh...");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

promise.resolve(42);
```



# Promesas: walled garden

```
var fakePromise = require("./fakePromise");
var promise = fakePromise.gimmePromise();

promise.then(function(msg) {
  console.log(msg);
  throw new Error("Oh, oh...");
  return "mundo";
})
.then(function(msg) {
  console.log(msg);
  return "OK!";
})
.fail(function(err) {
  console.log(err);
  return "MAL!";
})
.then(function(msg) {
  console.log(msg);
})

promise.resolve("holo");
```

# Promesas: walled garden

## Propagación de promesas

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise(),
    promise2 = fakePromise.gimmePromise();

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

promise.resolve(42);
setTimeout(promise2.resolve.bind(promise2, 12), 2000);
```

# Promesas: walled garden

## Propagación de promesas

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise(),
    promise2 = fakePromise.gimmePromise();

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

promise.resolve(42);
setTimeout(promise2.resolve.bind(promise2, 12), 2000);
```

# Promesas: walled garden

Propagación de promesas

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise(),
    promise2 = fakePromise.gimmePromise();

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

promise.resolve(42);
setTimeout(promise2.resolve.bind(promise2, 12), 2000);
```

# Promesas: walled garden

Propagación de promesas

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise(),
    promise2 = fakePromise.gimmePromise();

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

promise.resolve(42);
setTimeout(promise2.resolve.bind(promise2, 12), 2000);
```

# Promesas: walled garden

¿Y al revés?

```
var fakePromise = require("./fakePromise");

var promise = fakePromise.gimmePromise(),
    promise2 = fakePromise.gimmePromise();

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

setTimeout(promise.resolve.bind(promise, 42), 2000);
promise2.resolve(12);
```

# Diferidos

En el mundo real, las cosas son un poco distintas

Las “promesas” están divididas en dos objetos:

- 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

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

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

// flujo

promise.then(function(val) {
  console.log("val:", val);
}, function(err) {
  console.log("Error!");
});

// estado

defer.resolve(42);
```

# Diferidos

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

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

// flujo

promise.then(function(val) {
  console.log("val:", val);
}, function(err) {
  console.log("Error!");
});

// estado

defer.resolve(42);
```

# 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()`
- 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"),
Q = require("q");

function readFilePromise(filePath) {
  var defer = Q.defer();
  fs.readFile(filePath, function(err, data) {
    err ? defer.reject(err) : defer.resolve(data);
  })
  return defer.promise;
}

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

# Promesas

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

function readFilePromise(filePath) {Gestor del recurso
    var defer = Q.defer();
    fs.readFile(filePath, function(err, data) {
        err ? defer.reject(err) : defer.resolve(data);
    })
    return defer.promise;
}Consumidor del recurso
```

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

# Promesas: paréntesis

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

- fileExistsPromise
- readFilePromise

# Promesas: combinaciones

Q es una librería muy potente

- Trae un montón de funcionalidad
- Muy recomendable leer la documentación
- Nosotros vamos a ver dos cosas esenciales:
  - Combinación de promesas en paralelo
  - Adaptadores de llamadas con formato Node.js

# **Q.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

# Q.all([promise, promise, ...])

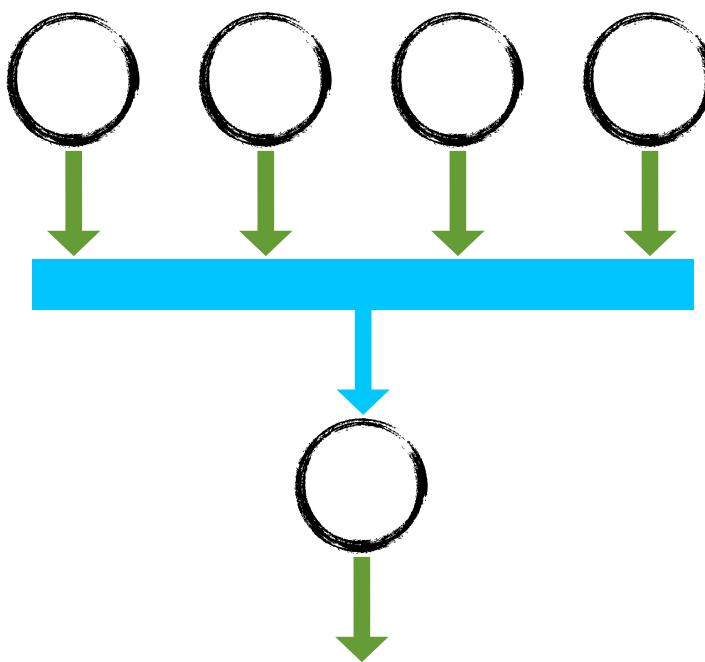
```
var Q = require("q");

var def1 = Q.defer(),
    prom1 = def1.promise,
    def2 = Q.defer(),
    prom2 = def2.promise;

Q.all([prom1, prom2]).then(function(v) {
  console.log("Todas resueltas!");
  console.log(v); // [42, 13]
})

def1.resolve(42);
setTimeout(def2.resolve.bind(def2, 13), 1000);
```

# **Q.all([promise, promise, ...])**



# **Q.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).

# **Q.spread([v1, v2], callback)**

“Reparte” valores de un array en parámetros

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

Q.spread([1,2,3,4], function(a, b, c, d) {
  console.log(a, b, c, d); // 1 2 3 4
})
```

# **Q.spread(...)**

Reescribe **readAllFiles(...)** para que utilice  
**Q.spread(...)**.

# **Q.spread([v1, v2], callback)**

Invoca automáticamente a Q.all(...)!

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

var def1 = Q.defer(), def2 = Q.defer(), def3 = Q.defer(),
    pro1 = def1.promise, pro2 = def2.promise, pro3 = def3.promise;

pro1.then(function(v1) {
  console.log(v1);
  return [pro2, pro3];
})
.spread(function(v2, v3) {
  console.log(v2, v3);
});

def1.resolve(42);
setTimeout(def2.resolve.bind(def2, 13), 1000);
setTimeout(def3.resolve.bind(def3, 71), 200);
```

# **Q.spread([v1, v2], callback)**

Invoca automáticamente a Q.all(...)!

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

var def1 = Q.defer(), def2 = Q.defer(), def3 = Q.defer(),
    pro1 = def1.promise, pro2 = def2.promise, pro3 = def3.promise;

pro1.then(function(v1) {
    console.log(v1);
    return [pro2, pro3];
})
.spread(function(v2, v3) {
    console.log(v2, v3);
});

def1.resolve(42);
setTimeout(def2.resolve.bind(def2, 13), 1000);
setTimeout(def3.resolve.bind(def3, 71), 200);
```

# **Q.spread([v1, v2], callback)**

Invoca automáticamente a Q.all(...)!

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

var def1 = Q.defer(), def2 = Q.defer(), def3 = Q.defer(),
    pro1 = def1.promise, pro2 = def2.promise, pro3 = def3.promise;

pro1.then(function(v1) {
  console.log(v1);
  return [pro2, pro3];
})
.spread(function(v2, v3) {
  console.log(v2, v3);
});

def1.resolve(42);
setTimeout(def2.resolve.bind(def2, 13), 1000);
setTimeout(def3.resolve.bind(def3, 71), 200);
```

# Paréntesis: invocadores

Casi todos los módulos de node.js que realizan una operación asíncrona siguen el mismo convenio:

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

# Paréntesis: invocadores

Por ej, escribe la función `fsStatPromise(filePath)`

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

# Paréntesis: invocadores

Otro: escribe la función `fsDirPromise(filePath)`

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

# Paréntesis: invocadores

Surge un patrón común:

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

function fsDirPromise(path) {
  var defer = Q.defer();
  fs.readdir(path, function(err, files) {
    return err? defer.reject(err) : defer.resolve(files);
  });
  return defer.promise;
}
```

# Paréntesis: invocadores

Surge un patrón común:

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

function fsDirPromise(path) {
    var defer = Q.defer();
    fs.readdir(path, function(err, files) {
        return err? defer.reject(err) : defer.resolve(files);
    });
    return defer.promise;
}
```

# Paréntesis: invocadores

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)

# Paréntesis: invocadores

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

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);
    });
```

# Paréntesis: invocadores

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)

# 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);
  });
});
```

# **Q(promiseOrValue)**

Homogeneizar valores:

- Si es una promesa, se queda tal cual
- Si es un valor, se convierte en una promesa que se resuelve a ese valor

```
Q(42).then(function(v) {  
  console.log(v); // 42  
})
```

```
Q.promise.then(function(v) {  
  console.log(v); // resolución de promise  
})
```

# Promesas: gotcha

¿Qué pasa aquí?

```
var d = Q.defer(), promise = d.promise;

promise.then(function(v) {
    console.log(v);
    throw new Error("Vaya por Dios!");
})
.then(function() {
    console.log("Hola?");
});

d.resolve(42);
```

# promise.done()

Finaliza el flujo, levantando los errores que no se hayan manejado

```
var d = Q.defer(), promise = d.promise;

promise.then(function(v) {
    console.log(v);
    throw new Error("Vaya por Dios!");
})
.then(function() {
    console.log("Hola?");
})
.done();

d.resolve(42);
```

# **promise.done()**

La regla es:

- Si vas a devolver la promesa, déjala abierta
- Si eres el consumidor final de la promesa, asegúrate de cerrarla con `.done()`

# ¡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);
})
```

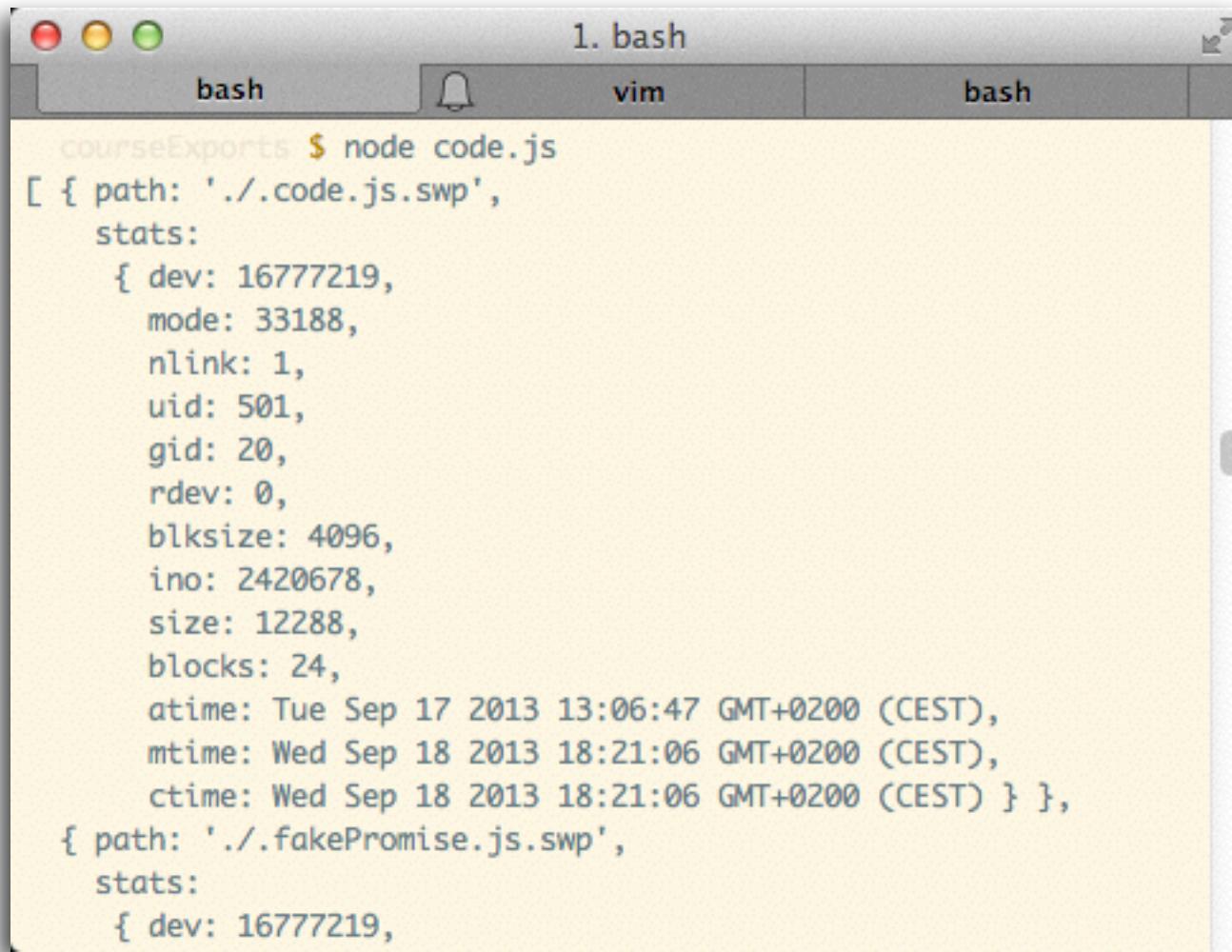
# Primer paso: listado recursivo

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 with a terminal window titled "1. bash". The window has three tabs: "bash", "vim", and another "bash" tab which is currently active. 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 opcionales

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);
})
```