Tareas simples

Tiempo estimado: 25min

Una vez tenemos una ligera imagen de Justo, se hace necesario ir presentando cada uno de sus componentes individualmente. Comencemos con las tareas simples.

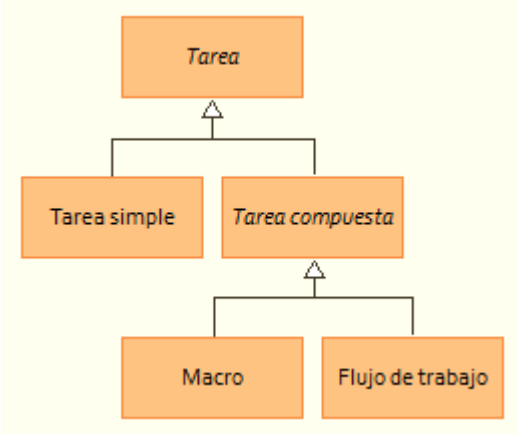
Primero, mostramos el concepto de tarea simple, la unidad de trabajo más pequeña e indivisible. A continuación, cómo definirlas e invocarlas. Después, exponemos cómo crear tareas parametrizadas, síncronas y/o asíncronas. Seguimos con el mecanismo de inyección, usado por Justo, para pasar datos a las operaciones de tarea, facilitando así su escritura. Finalmente, introducimos el catálogo de tareas, independientemente de su tipo, invocables directamente desde la línea de comandos.

Al finalizar la lección, el estudiante sabrá:

- Qué es una tarea.
- Qué es una tarea simple.
- Cómo definir una tarea simple.
- Cómo invocar una tarea.
- Cómo definir tareas simples parametrizadas, síncronas o asíncronas.
- Cómo registrar tareas en el catálogo.
- Cómo invocar tareas registradas en el catálogo.

Introducción

Una tarea (*task*) representa un trabajo, actividad o proceso a ejecutar como, por ejemplo, la operación de compilación, la minimización de código JavaScript, HTML o CSS, la realización de una copia de seguridad, el alta de un nuevo usuario, etc. Se distingue entre tareas simples y compuestas, estas últimas clasificadas en macros y flujos de trabajo.



Una tarea simple (*simple task*) es una operación que realiza un determinado trabajo indivisible. Mientras que una tarea compuesta (*composite task*) representa un trabajo formado por varias tareas. Una macro (*macro*) es una secuencia de cero, una o más tareas ejecutadas secuencialmente. Y un flujo de trabajo (*work flow*), una operación que puede ejecutar unas tareas u otras según un determinado flujo de ejecución.

En esta lección, centramos nuestra atención en las tareas simples.

Tareas simples

Una tarea simple (*simple task*) es la unidad de operación más pequeña, implementada mediante una

función JavaScript, que realiza un determinado trabajo. Su ejecución la lanza el ejecutor de tareas de Justo, el cual recopilará información sobre su duración y su resultado final. Cualquier operación o función, cuya ejecución deseamos sea supervisada por el motor de Justo, debe definirse como una tarea simple.

Por ejemplo, supongamos un sencillo trabajo relacionado con la generación del código publicable de un paquete Node.js. Básicamente, encontramos que es una operación de cinco fases:

1. Comprobar que el código cumple con las buenas prácticas de programación.
2. Borrar o vaciar el directorio donde construir el paquete.
3. Compilar el código JavaScript.
4. Borrar o vaciar el directorio donde depositar el código a distribuir.
5. Crear el paquete a distribuir o publicar.

La idea es que deseamos poder monitorizar cada uno de los pasos anteriores y recopilar su estado final, esto es, si acabaron bien o en fallo. Para ello, lo que hacemos es definir cada una de las operaciones como una tarea simple. En nuestro caso, lo que hacemos es definir una tarea compuesta con el nombre build, la cual estará formada por cinco tareas simples, cada una de las cuales realiza una de las operaciones anteriores. Un ejemplo de su ejecución es el siguiente:

```
> justo build

  build
   build
    [ OK ] Clean build directory (9 ms)
    [ OK ] Best practices and grammar (410 ms)
    [ OK ] Transpile (250 ms)
    [ OK ] Clean dist directory (0 ms)
    [ OK ] Create package (0 ms)

  OK 5 | Failed 0 | Ignored 0 | Total 5

>
```

Como puede observar, Justo ejecuta y monitoriza la invocación de cada tarea simple. Para así, recopilar información de cada una de ellas. Para finalmente, mostrársela al usuario. Básicamente, cómo finalizó y cuánto tiempo llevó su ejecución.

Definición de tareas simples

Para definir una tarea simple, hay que utilizar la función simple() definida en el paquete justo que, recordemos, se debe instalar localmente a cada proyecto:

```
function simple(name : string, fn : function) : function
function simple(props : object, fn : function) : function
```

Parámetro	Tipo de datos	Descripción
name	String	Nombre identificativo de la tarea.
props	Object	Propiedades de la tarea: • name (string). Nombre identificativo de la tarea. • ns (string). Espacio de nombres en el que se encuentra ubicada la tarea. • title (string). Título predeterminado de la tarea. • desc (string). Descripción predeterminada de la tarea. • ignore (boolean). ¿Ignorar la ejecución de la tarea? true, sí; false, no. • onlyIf (boolean). Ejecutar si se cumple lo indicado. • mute (boolean). ¿Ejecución silenciosa de la tarea? true, sí; false, no.
fn	Function	Operación de tarea.

En Justo, la ejecución de una tarea es como ejecutar una función. Así pues, lo que devuelve simple() es la

función a utilizar para invocar la tarea. No hay que utilizar la función pasada a simple(), sino la devuelta por simple(). Por esta razón, distinguimos dos conceptos. La operación de tarea (*task operation*), aquella que implementa la operación asociada a la tarea y que se pasa a la función simple(). Y la función de tarea (*task function*) o función de envoltura (*wrapper function*), la que representa la tarea y devuelve simple().

A continuación, se muestra un ejemplo ilustrativo de la definición de una tarea simple:

```
//imports
const simple = require("justo").simple;

//tasks
const wrapper = simple({ns: "com.myco", name: "task"}, function op() {
  ...
});
```

Identificación de las tareas

Se recomienda nombrar las tareas mediante un nombre identificativo y un espacio de nombres. El espacio de nombres (*namespace*) es un contenedor lógico de tareas. Como un paquete de Java o un espacio de nombres de .Net. Se utiliza para mantener organizadas las tareas según su propietario o funcionalidad. Los espacios de nombres org.justojs y com.justojs se encuentran reservados para tareas *oficiales* implementadas por el equipo de Justo.js y no debe definirse ninguna tarea de usuario bajo estos espacios de nombres.

Por otra parte, las tareas deben tener un nombre (*name*), un identificador único dentro del espacio de nombres al que pertenece. Se utiliza únicamente para su organización lógica.

Título de las tareas

Tal como veremos en breve, la ejecución de una tarea consiste en ejecutar la función devuelta por simple(). Cada ejecución de la función de tarea puede tener asociada su propio título (*title*), palabra o frase que describe una ejecución o invocación particular. Cuando definimos una tarea, podemos indicar el título predeterminado, para aquellos casos en los que no se indique ninguno explícitamente en la invocación.

Por ejemplo, supongamos que disponemos de una tarea que crea usuarios en una base de datos PostgreSQL. A cada ejecución particular, le podríamos asociar como título específico algo como *Crear usuario dba*, *Crear usuario secadmin*, etc. Si por ejemplo estamos usando una tarea para copiar archivos, podríamos utilizar el título para indicar los archivos origen y destino, por ejemplo, *Copiar postgresql.conf a postgresql.conf.bak*.

Descripción de tareas

La descripción (*description*) es un texto que detalla el objeto de la tarea. Por ejemplo, una tarea que copia archivos podría tener como descripción *Copiar archivos*; la de *ping* a una máquina, *Realizar ping*; etc. Es opcional, pero se recomienda su uso.

Omisión de tareas

La omisión de tareas (*task skip*) es una marca que indica si la ejecución de la tarea debe ignorarse. Se puede particularizar a nivel de invocación de la tarea, pero si se desea, se puede indicar el valor predeterminado de esta marca. Por ejemplo, para definir una tarea que sólo debe ejecutarse cuando el sistema operativo es Windows podríamos utilizar la propiedad ignore u onlyIf como sigue:

```
task = simple({name: "task", ignore: !os.platform().startsWith("win")}, function() {
  //...
});

task = simple({name: "task", onlyIf: os.platform().startsWith("win")}, function() {
  //...
});
```

Ambas propiedades son similares, aunque desde puntos de vista distintos. En caso de indicarse ambas, prevalecerá onlyIf.

Ejecución silenciosa de tareas

Cada vez que ejecutamos una función de tarea, Justo monitorizará su invocación, recopilando información de su ejecución. Cuando se desea ejecutar una tarea en modo silencioso, omitiendo esta recopilación de información, sin mostrarse nada en el informe resultado, se puede utilizar la propiedad mute. La cual puede definirse de manera predeterminada en la definición de la tarea y sobrescribirse, al igual que title, ignore y onlyIf, en cada ejecución.

Invocación de tareas

Una función de tarea se puede invocar tantas veces como sea necesario. Cada una de ellas representa una ejecución particular de la operación de tarea. En resumen, es una unidad de ejecución reutilizable. Esta función, recordemos, la devuelve la función simple() y es lo que deberemos utilizar para su ejecución. Pero a diferencia de la operación de tarea, la que pasamos a la hora de crearla, tiene su propia signatura de ejecución:

```
task(title : string)
task(props : object)
task(title : string, arg1, arg2, arg3…)
task(props : object, arg1, arg2, arg3…)
```

A cada ejecución le debemos de asignar un título. En caso de no indicarse explícitamente, se utilizará el título asignado en la creación de la tarea. Si no se fijó ninguno, se usará el nombre de la tarea. Se recomienda encarecidamente indicar un título explícitamente.

Las ejecuciones pueden tener las siguientes propiedades, las cuales siempre se deben pasar como primer argumento:

- title (string). Título asignado a la ejecución y con el que se identificará en el informe resultado.
- ignore (boolean). ¿Ignorar la ejecución? true, sí; false, no.
- onlyIf (boolean). Ejecutar si se cumple lo indicado.
- mute (boolean). ¿Ejecutar la tarea silenciosamente? true, sí; false, no.

Veamos un ejemplo:

```
//imports
const justo = require("justo");
const babel = require("justo-plugin-babel");

babel("Transpile", {
  comments: false,
  retainLines: true,
  preset: "es2015",
  files: [
    {src: "index.js", dst: "build/es5/"},
    {src: "lib/", dst: "build/es5/lib/"}
  ]
});
```

La función de tarea devuelta por simple() la asociamos a la variable babel. En nuestro caso, utilizamos el *plugin* justo-plugin-babel, el cual define la tarea con la que ejecutar el comando babel. El título de la ejecución es *Transpile*. Mientras que el resto de argumentos se pasará a la operación de tarea.

Otro ejemplo ilustrativo:

```
//imports
const justo = require("justo");
const copy = require("justo-plugin-fs").copy;

copy(
  "Create package",
  {
    src: "build/es5/lib/",
    dst: "dist/es5/nodejs/justo-plugin-browserify/lib"
  },
  {
    src: ["package.json", "README.md"],
    dst: "dist/es5/nodejs/justo-plugin-browserify/"
```



```
  }
);
```

Tareas parameterizadas

Tal como acabamos de ver, el primer argumento pasado a una función de tarea lo usa internamente el ejecutor o motor de Justo. El resto de argumentos se pasan a la operación de tarea. Cuando a una operación se le puede pasar argumentos, el ejecutor se los pasará mediante el parámetro params. Para ir abriendo boca, veamos un ejemplo ilustrativo de una tarea que puede recibir argumentos de usuario en cada invocación:

```
task = simple({...}, function(params) {
  ...
});
```

Recordemos que el parámetro params, de tipo *array*, tiene un significado especial para Justo. A través de él, pasará los argumentos pasados en la llamada de la tarea. Cuando una operación define este parámetro, la tarea se conoce formalmente como tarea parametrizada (*parameterized task*).

Por ejemplo, supongamos la tarea copy del *plugin* justo-plugin-fs para la copia de archivos y/o directorios. La operación de tarea se define como sigue:

```
function copy(params) {
  //...
}
```

Pero cuando la invocamos, lo hacemos como sigue:

```
copy(
  "Create package",
  {
    src: "build/es5/lib/",
    dst: "dist/es5/nodejs/justo-plugin-browserify/lib"
  },
  {
    src: ["package.json", "README.md"],
    dst: "dist/es5/nodejs/justo-plugin-browserify/"
  }
);
```

Justo utiliza el primer argumento de la llamada a la función de tarea para cuestiones internas. En nuestro caso, el texto *Create package*. El resto de argumentos se los pasará en un *array* al parámetro params. Siguiendo con nuestro ejemplo, la operación de tarea recibirá lo siguiente:

```
[
  {src: "build/es5/lib/", dst: "dist/es5/nodejs/justo-plugin-browserify/lib"} ,
  {src: ["package.json", "README.md"], dst: "dist/es5/nodejs/justo-plugin-browserify/"}
]
```

Tareas asíncronas

Las tareas se clasifican en síncronas o asíncronas, atendiendo a si llevan a cabo E/S. Cuando una operación de tarea *no* lleva a cabo ninguna E/S, por ejemplo, a disco o a red, se conoce formalmente como tarea síncrona (*synchronous task*). En cambio, aquella que sí lo hace, como tarea asíncrona (*asynchronous task*).

Una tarea síncrona finaliza cuando finaliza su cuerpo. En cambio, una tarea asíncrona lo hace cuando finaliza toda su ejecución. En este último caso, la finalización de la ejecución del cuerpo de la operación de tarea no implica que haya terminado, por lo que es necesario indicarle al ejecutor, de alguna manera, que ha acabado.

Una tarea asíncrona define su operación con la presencia del parámetro especial done. Cuando se define una operación con este parámetro, automáticamente se tratará como asíncrona. Cualquiera que *no* la defina, como síncrona. Mediante este parámetro, el ejecutor pasará una función que debe invocar la operación cuando haya finalizado. Veamos un ejemplo ilustrativo:

```
task = simple({...}, function(params, done) {
  ...
  done();
});
```


La función done() puede invocarse como sigue:

```
done()
done(error)
done(undefined, resultado)
```

Cuando se invoca sin argumentos, el ejecutor considerará que la ejecución de la tarea asíncrona finalizó correctamente. Pero si se produjo algún error, se debe pasar éste como primer argumento a la función. Ejemplos:

```
done()                              //OK
done("mensaje de error")            //fallida
done(new Error("mensaje de error")) //fallida
```

Ahora bien, si la función debe devolver algo, debe pasarlo como segundo parámetro:

```
done(undefined, 123456)
```

Inyección de dependencias

A diferencia de otros automatizadores, donde el orden de los parámetros de la operación de tarea es importante, en Justo no lo es. Justo utiliza inyección para pasar datos específicos a la operación de tarea. Para ello, usa convenio de nomenclatura. Por ejemplo, a través del parámetro params se marca la tarea como parametrizada y en él se pasará los argumentos pasados en la invocación. El parámetro done marca la tarea como asíncrona y en él se pasará la función que debe invocar la operación cuando haya terminado, para así continuar con la siguiente tarea.

En Justo, el orden de los factores no afecta al producto. Da lo mismo indicar primero params y después done o viceversa. Lo único es que la función los defina. Así pues, las siguientes operaciones tendrán el mismo significado para el motor de ejecución:

```
function(params, done) { ... }
function(done, params) { ... }
```

Registro de mensajes

Justo utiliza un registro de mensajes con el que mostrar mensajes adicionales al usuario. Cada mensaje puede tener un nivel de importancia: debug, info, warn, error o fatal. De manera predeterminada, Justo sólo muestra los mensajes con niveles igual o superior a info, descartando los de depuración. Se puede configurar los niveles a mostrar mediante las propiedades de configuración runner.logger.minLevel y runner.logger.maxLevel, recordemos, ubicadas en el archivo Justo.json. Por lo general, sólo durante el desarrollo de un *plugin* se suele mostrar los mensajes de depuración.

Una operación de tarea puede definir el parámetro logger o, generalmente, log a través del cual el motor de ejecución le pasará el registro de mensajes. Este objeto contiene los siguientes métodos, cada uno para un nivel distinto:

```
log.debug(msg)
log.info(msg)
log.warn(msg)
log.error(msg)
log.fatal(msg)
```

A continuación, se muestra una operación de tarea parametrizada, asíncrona y con posibilidad de escribir mensajes en el registro de mensajes:

```
function(params, log, done) { ... }
```

Recordemos que el orden de los parámetros no altera el producto. Sus valores los inyecta el motor atendiendo a sus nombres.

Consola

Justo proporciona un objeto console que proporciona las funciones log(), info() y error(). Cuando tengamos que mostrar algo, se recomienda solicitarle a Justo que se lo pase a nuestra operación de tarea. Se consigue mediante el parámetro console. Gracias a este objeto, los mensajes mostrados en la consola quedarán alineados en el informe resultado.

He aquí un ejemplo ilustrativo de operación de tarea que solicita la inyección del objeto console:

```
function(params, console) { ... }
```


Catálogo de tareas

Cada vez que se crea una tarea, no se registra en ninguna parte. Se crea un objeto función que actúa a modo de intermediario entre la tarea, el usuario y el ejecutor de tareas. A pesar de ello, el ejecutor viene con un catálogo (*catalog*), un contenedor donde registrar tareas que pueden invocarse directamente desde la línea de comandos. Las tareas registradas en este contenedor se conocen formalmente como tareas catalogadas (*cataloged tasks*).

Registro de tareas catalogadas

Por lo general, cada proyecto dispone de sus propias tareas específicas, o sea, tareas que automatizan algún tipo de trabajo, actividad o proceso. Por ejemplo, es muy común que los proyectos de desarrollo de software definan y registren principalmente tres tareas, build, test y default. El registro de una tarea se hace mediante el objeto catalog, ubicado en el paquete justo. En el caso de una tarea simple, mediante catalog.simple():

```
function catalog.simple(name : string, fn : function) : function
function catalog.simple(props : object, fn : function) : function
```

A la hora de realizar un registro, se puede especificar las siguientes propiedades:

- name (string). Nombre con el que registrar la tarea en el catálogo. Debe ser único y se utiliza para invocarlo.
- desc (string). Descripción de la tarea.

He aquí unos ejemplos ilustrativos:

```
//imports
const catalog = require("justo").catalog;

//tasks
catalog.workflow({name: "build", desc: "Build the package."}, function() {
  //...
});

catalog.simple({name: "package", desc: "Package the module."}, function() {
  //...
});

catalog.macro("default", ["build", "test"]);
```

El primer ejemplo define una tarea de flujo de trabajo con el nombre build. El segundo es una tarea simple. Y el tercero, una macro, con nombre default y consiste en la invocación de las tareas catalogadas, build y test, en ese orden. La tarea registrada como default se conoce formalmente como tarea predeterminada (*default task*) y es la que invoca justo cuando no indicamos ninguna específicamente.

Listado de tareas catalogadas

Para conocer las tareas registradas o catalogadas y que, por tanto, podemos invocar directamente con el comando justo, se utiliza el comando justo con la opción -c o --catalog. Ejemplo:

```
> justo -c
Name     Description
build    Build the package.
default  Default task.
publish  NPM publish
test     Unit test.
>
```

Invocación de tareas registradas o catalogadas

Para invocar una determinada tarea catalogada, hay que pasar su nombre. Podemos indicar varias. A continuación, un ejemplo:

```
> justo build test

   build
    build
```



```
    [ OK ] Clean build directory (0 ms)
    [ OK ] Best practices and grammar (417 ms)
    [ OK ] Transpile (250 ms)
    [ OK ] Clean dist directory (0 ms)
    [ OK ] Create package (15 ms)

  OK 5 | Failed 0 | Ignored 0 | Total 5


   test
    test
      test/unit/index.js
        API
          render
          [ OK ] Test function (86 ms)
      test\unit\lib\render.js
        #op()
          op(config) - condition is true
          [ OK ] init(*) (0 ms)
          [ OK ] Test function (31 ms)
          [ OK ] fin(*) (0 ms)
          op(config) - condition is false
          [ OK ] init(*) (0 ms)
          [ OK ] Test function (0 ms)
          [ OK ] fin(*) (0 ms)

  OK 7 | Failed 0 | Ignored 0 | Total 7

>
```

Si deseamos invocar la tarea predeterminada, basta con invocar justo sin ninguna opción:

```
> justo

   default
    default
      build
      [ OK ] Clean build directory (0 ms)
      [ OK ] Best practices and grammar (455 ms)
      [ OK ] Transpile (241 ms)
      [ OK ] Clean dist directory (17 ms)
      [ OK ] Create package (15 ms)
      test
        test/unit/index.js
          API
            render
            [ OK ] Test function (78 ms)
        test\unit\lib\render.js
          #op()
            op(config) - condition is true
            [ OK ] init(*) (0 ms)
            [ OK ] Test function (31 ms)
            [ OK ] fin(*) (0 ms)
            op(config) - condition is false
            [ OK ] init(*) (0 ms)
            [ OK ] Test function (0 ms)
            [ OK ] fin(*) (14 ms)

  OK 12 | Failed 0 | Ignored 0 | Total 12

>
```

Cuando invocamos una tarea desde la línea de comandos, podemos pasarle argumentos si es necesario. Los añadiremos a continuación del nombre de la tarea separados por dos puntos:

```
tarea:arg1:arg2:arg3…
```

Los argumentos son de tipo cadena.

