Flujos de trabajo

Tiempo estimado: 10min

En las lecciones anteriores, presentamos los conceptos de tarea simple y compuesta. Vimos que, además, las compuestas se clasifican en macros y flujos de trabajo. A continuación, vamos a mostrar los flujos de trabajo.

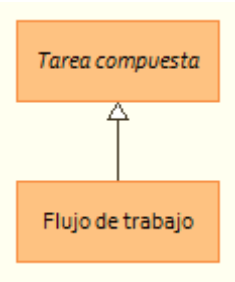
Primero, presentamos el concepto de flujo de trabajo. Después, cómo definirlos y registrarlos. Finalmente, como invocarlos. Tal como hemos hecho con los anteriores tipos de tareas.

Al finalizar la lección, el estudiante sabrá:

- Qué es un flujo de trabajo.
- Cómo definir un flujo de trabajo.
- Cómo invocar un flujo de trabajo.
- Cómo registrar un flujo de trabajo en el catálogo.

Introducción

Un flujo de trabajo (*work flow*) es una tarea compuesta que permite ejecutar unas tareas u otras atendiendo a un determinado flujo de ejecución. Este flujo se define mediante una función JavaScript, donde se puede definir sentencias if, while, for, etc., y donde se puede invocar cualquier tarea. Cada tarea invocada dentro de esta función se asociará al flujo y así se mostrará en el informe final.



A diferencia de las macros de tareas, donde sus tareas asociadas se invocan una detrás de otra, según el orden en el que se indican, en un flujo de trabajo las tareas se invocan cuando es necesario y si es necesario.

Por ejemplo, supongamos que deseamos definir un flujo de trabajo para crear un grupo de cuentas de usuario. Para cada cuenta, deseamos comprobar primero si existe y, entonces, si no existe, crearla. Algo así como lo siguiente:

```
for (let cuenta of cuentas) {
  if (!existe(`Comprobar si existe la cuenta '${cuenta}'`, cuenta)) {
    crear(`Crear la cuenta '${cuenta}'`, cuenta);
  }
}
```

Supongamos tres cuentas de usuario, a, b y c, donde a y c existen ya, no así b. Cuál sería el resultado de ejecutar el flujo de trabajo anterior:

```
Crear cuentas de usuario
[ OK ] Comprobar si existe la cuenta 'a' (0 ms)
[ OK ] Comprobar si existe la cuenta 'b' (1 ms)
[ OK ] Crear la cuenta 'b' (2 ms)
[ OK ] Comprobar si existe la cuenta 'c' (0 ms)
```

Sólo se invoca la tarea crear cuando la cuenta no existe.

Ahora, veamos el resultado para el caso en que las cuentas de usuario a y c no existen, pero sí b:

```
Crear cuentas de usuario
```



```
[ OK ] Comprobar si existe la cuenta 'a' (0 ms)
[ OK ] Crear la cuenta 'a' (1 ms)
[ OK ] Comprobar si existe la cuenta 'b' (0 ms)
[ OK ] Comprobar si existe la cuenta 'c' (0 ms)
[ OK ] Crear la cuenta 'c' (1 ms)
```

Como se puede observar, las tareas ejecutadas dependen del flujo de ejecución seguido en cada ocasión. Este tipo de situaciones no se puede implementar mediante tareas simples ni macros. Para este fin, se han definido los flujos de trabajo. Es más, una macro de tareas puede definirse como un flujo de trabajo donde se ejecuta cada una de sus tareas una detrás de otra.

Los flujos de trabajo se utilizan principalmente para automatizar trabajos o procesos que involucran varias tareas, donde no siempre se debe ejecutar todas ellas o debe hacerlo con parámetros distintos. He aquí un ejemplo del flujo de trabajo de una instalación de Redis:

```
catalog.workflow({name: "install", desc: "Install Redis manually."}, function() {
  var uid, gid;

  //(1) uninstall if needed
  if (apt.installed("Check if Redis installed", {names: "redis-server"})) {
    apt.uninstall("Uninstall Redis", {names: ["redis-server", "redis-sentinel", "redis-
tools"], purge: true});
  }

  //(2) create user and group if needed
  if (!user.exists(`Check if ${REDIS_USER} user created`, {name: REDIS_USER})) {
    user.add(`Create ${REDIS_USER} user`, {
      name: REDIS_USER,
      home: "/var/lib/redis",
      shell: "/bin/false"
    });
  }

  if (!group.exists(`Check if ${REDIS_GROUP} group created`, {name: REDIS_GROUP})) {
    group.add(`Create ${REDIS_GROUP} group`, {name: REDIS_GROUP});
  }

  uid = user.info(`Get ${REDIS_USER} UID`, {name: REDIS_USER})[0].uid;
  gid = group.info(`Get ${REDIS_GROUP} GID`, {name: REDIS_GROUP})[0].gid;

  //(3) download Redis
  download("Download Redis from redis.io", {
    src: "http://download.redis.io/releases/redis-stable.tar.gz",
    dst: "./tmp/"
  });

  tar.extract("Extract redis-stable.tar.gz", {
    src: "./tmp/redis-stable.tar.gz",
    dst: "./tmp/"
  });

  //(4) compile
  apt.update("Update package index");
  apt.install("Install compilation requisites", {names: ["make", "gcc", "tcl"]});

  cli("Compile dependencies", {
    cmd: "make",
    args: ["make", "hiredis", "lua", "jemalloc", "linenoise", "geohash-int"],
    wd: "./tmp/redis-stable/deps"
  });

  cli("Compile Redis", {cmd: "make", wd: "./tmp/redis-stable"});

  //(5) install
  cli("Install Redis", {cmd: "make", arg: "install", wd: "./tmp/redis-stable"});
  fs.copy("Generate /etc/redis/redis.conf", {src: "./tmp/redis-stable/redis.conf", dst:
"/etc/redis/redis.conf"});
  fs.clean("Remove tmp dir", {src: "./tmp/"});

  //(6) postinstall
```



```
  fs.chown("Set ownership to /etc/redis/ and /var/lib/redis/", {
    src: ["/etc/redis/", "/var/lib/redis/"],
    user: uid,
    group: gid,
    recurse: true
  });

  fs.chown("Set ownership to /usr/local/bin/redis-*", {
    src:  [
      "/usr/local/bin/redis-benchmark",
      "/usr/local/bin/redis-check-aof",
      "/usr/local/bin/redis-check-rdb",
      "/usr/local/bin/redis-cli",
      "/usr/local/bin/redis-sentinel",
      "/usr/local/bin/redis-server"
    ],
    user: uid,
    group: gid
  });

  fs.chmod(`Set permissions to /etc/redis/redis.conf`, {src: "/etc/redis/redis.conf", mode:
"640"});

  fs.chmod(`Set permissions to /usr/local/bin/redis-*`, {
    src: [
      "/usr/local/bin/redis-benchmark",
      "/usr/local/bin/redis-check-aof",
      "/usr/local/bin/redis-check-rdb",
      "/usr/local/bin/redis-sentinel",
      "/usr/local/bin/redis-server"
    ],
    mode: "750"
  });

  fs.chmod(`Set permissions to /usr/local/bin/redis-cli`, {
    src: "/usr/local/bin/redis-cli",
    mode: "755"
  });

  fs.chmod(`Set permissions to /var/lib/redis`, {src: "/var/lib/redis", mode: "750"});

  //(7) start service
  cli("Start Redis server", {
    cmd: "runuser",
    args: ["-u", REDIS_USER, "-g", REDIS_GROUP, "redis-server"],
    wd: "/var/lib/redis",
    bg: true
  });
});
```

Definición de flujos de trabajo

Para definir un flujo de trabajo, hay que utilizar la función workflow() definida en el paquete justo:

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

Parámetro	Tipo de datos	Descripción
name	String	Nombre identificativo de la tarea.
props	Object	Propiedades de la tarea. Las mismas que las presentadas en simple() y macro().
fn	Function	Operación de flujo.

La operación de flujo es similar a la operación de una tarea simple. Puede ser síncrona, asíncrona y/o parametrizada. Al igual que simple() y macro(), workflow() devuelve la función de tarea o envoltura que debemos usar para invocar el flujo cada vez que sea necesario.

Recordemos que para definir un flujo de trabajo asíncrono, basta con definir el parámetro done en la operación de flujo, a través del cual Justo inyectará la función que debe invocar cuando haya finalizado.

A continuación, se muestra un ejemplo ilustrativo de la definición de un flujo de trabajo síncrono:

```
const wrapper = workflow({name: "crear", desc: "Crear cuentas de usuario."},
function(params) {
  for (let cuenta of params) {
    if (!existe(`Comprobar si existe la cuenta '${cuenta}'`, cuenta)) {
      crear(`Crear la cuenta '${cuenta}'`, cuenta);
    }
  }
});
```

Registro de flujos de trabajo catalogados

Para registrar un flujo de trabajo en el catálogo, hay que utilizar la función catalog.workflow():

```
function catalog.workflow(name : string, tasks : function) : function
function catalog.workflow(props : object, tasks : function) : function
```

Registran el flujo de trabajo en el catálogo y devuelven su función de tarea o envoltorio.

Invocación de flujos de trabajo

La invocación de un flujo de trabajo es similar al de una tarea cualquiera, tal como vimos con las tareas simples y las macros. Se utiliza la función devuelta por la función workflow(). El primer argumento pasado en la invocación, recordemos, lo utiliza internamente el motor de Justo, mientras que el resto se pasan, mediante un *array*, al parámetro params de la operación de flujo.

