

Vamos a introducir a **Justo.js** formalmente.

Comenzamos presentando **Justo.js** y su arquitectura. A continuación, cómo instalarlo. Y finalmente, cuáles son los principales archivos de un proyecto **Justo.js**.

Al finalizar la lección, el estudiante sabrá:

- Qué es **Justo.js**.
- Cuáles son sus principales características.
- Cómo instalarlo.
- Cuáles son sus principales componentes.
- Cuál es la diferencia entre un proyecto **Justo.js** independiente y uno integrado.

Introducción

Justo.js, o simplemente **Justo**, es una herramienta de automatización, esto es, una aplicación para la ejecución automática de tareas repetitivas como, por ejemplo, la compilación de una aplicación, la instalación personalizada de un producto o la realización de copias de seguridad. Está desarrollada íntegramente en **JavaScript** usando **Node.js**.

Su sitio web oficial es justojs.org.

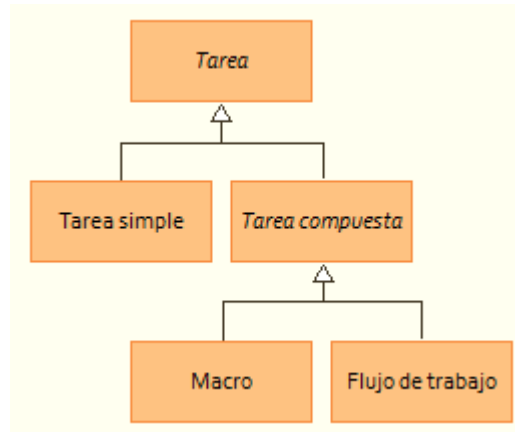
Sus principales características son las siguientes:

- Es sencillo y fácil de usar, dificultando cometer errores.
- Es fácil de aprender, reduciendo la curva de aprendizaje.
- Está bien documentado, facilitando el aprendizaje de los usuarios.
- Se ha desarrollado en **JavaScript**, concretamente bajo la plataforma **Node.js**, ampliamente aceptada y usada hoy en día por empresas como **eBay**, **Facebook**, **LinkedIn**, **Microsoft**, **Netflix**, **The New York Times**, **Uber** y **Yahoo!**.
- Es rápido. Utiliza el motor de **JavaScript V8** de **Google**.
- Es gratuito tanto para su uso personal como profesional.
- Distingue entre distintos tipos de tareas, simples, macros o flujos de trabajo, lo que facilita su flexibilidad y adaptabilidad a cada situación.
- Soporta tareas síncronas y asíncronas.
- Proporciona generadores para la creación de archivos y/o directorios plantillas, reduciendo los tiempos de arranque de los proyectos.
- Es multiplataforma. Se puede utilizar en **Windows** y **Linux**. Y en dispositivos diversos como computadoras personales, servidores y **Raspberry Pis**.
- Es muy fácil extender su funcionalidad mediante *plugins* y generadores escritos en **JavaScript**.
- Utiliza inyección de dependencia, facilitando su uso y la escritura de nuevos *plugins*.
- Proporciona soporte nativo para la automatización de pruebas. Permitiendo utilizar la misma herramienta de automatización en la realización de pruebas de unidad e integración en proyectos de software, así como de instalación. Reduciendo así la curva de aprendizaje y el número de herramientas de automatización a usar.

Arquitectura

Los componentes básicos de **Justo** son las tareas, el ejecutor de tareas, los archivos **package.json**, **Justo.js** y **Justo.json** así como la utilidad de línea de comandos **justo**.

Una **tarea** (*task*) representa un trabajo o actividad a ejecutar como, por ejemplo, el proceso de compilación, la minimización del 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, concretamente una función de **JavaScript**. 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. Y un **flujo de trabajo** (*work flow*) representa un trabajo que puede ejecutar unas tareas u otras según un flujo de ejecución y determinadas condiciones. Poco a poco, las iremos detallando a lo largo del curso.

Tal como veremos, la tarea simple representa una determinada funcionalidad independiente e indivisible. Esto quiere decir que es la unidad más pequeña de trabajo. Mientras que las tareas compuestas son unidades de trabajo que invocan otras tareas, ya sea simples o compuestas.

A continuación, se muestra un ejemplo de un proyecto de desarrollo de software que utiliza **Justo** para compilar, generar el paquete final y realizar las pruebas de unidad:

```
default
  default
    build
      [ OK ] Clean build directory <15 ms>
      [ OK ] Best practices and grammar <439 ms>
      [ OK ] Transpile <269 ms>
      [ OK ] Clean dist directory <16 ms>
      [ OK ] Create package <15 ms>
    test
      test/unit/index.js
        API
          render
            [ OK ] Test function <76 ms>
          test/unit/lib/render.js
            #op<>
              op<config> - condition is true
                [ OK ] init(*) <1 ms>
                [ OK ] Test function <38 ms>
                [ OK ] fin(*) <2 ms>
              op<config> - condition is false
                [ OK ] init(*) <0 ms>
                [ OK ] Test function <0 ms>
                [ OK ] fin(*) <0 ms>
OK 12 | Failed 0 | Ignored 0 | Total 12
```

El ejemplo anterior muestra dos trabajos. Uno que realiza la construcción del software, que consiste en cinco tareas simples: suprimir los directorios de compilación y distribución, comprobar que el código cumple con las buenas prácticas, realizar la compilación y finalmente generar el paquete a publicar para su uso por otros usuarios. El segundo realiza las pruebas de unidad.

Como puede observar, las tareas pueden acabar en tres estados: **OK**, todo ha ido bien; **Failed**, se produjo algún error; o **Ignored**, la tarea se ha ignorado, por ejemplo, porque se debe ejecutar sólo en un entorno **Windows** o **Linux**.

Finalmente, tenemos el **ejecutor de tareas** (*task runner*), el componente que se encarga de ejecutar las tareas e informar de su resultado: **OK**, **Failed** o **Ignored**. El ejecutor es la utilidad de línea de comandos **justo**.

Instalación

Una vez tenemos claro que vamos a utilizar **Justo** para la automatización de tareas, actividades, trabajos o procesos, ya sea de software, de sistemas o de cualquier otro tipo, hay que realizar una pequeña instalación, siguiendo los siguientes pasos:

1. Instalar el paquete **justo-cli** globalmente en la máquina.
2. Instalación del paquete **justo** localmente al proyecto.
3. Crear los archivos **package.json**, **Justo.json** y **Justo.js** en la raíz del proyecto.

Justo está desarrollado en **JavaScript**, con la plataforma **Node.js**. No hay que olvidar tenerla instalada en la máquina en la que se ejecutará. Mediante la utilidad de paquetes de **Node.js**, **npm**, se procederá a instalar los paquetes de **Justo**: **justo-cli**, **justo** y cualquier *plugin* que vayamos a utilizar.

Instalación del paquete justo-cli

El paquete **justo-cli** proporciona la línea de comandos de **Justo**, esto es, el programa **justo**. Debe instalarse *globalmente*, una única vez por máquina, por ejemplo, mediante:

```
npm install -g justo-cli
```

Una vez instalado, es buena práctica comprobar que tenemos acceso a la utilidad **justo**, generalmente consultando la versión instalada o mostrando la ayuda:

```
justo --version
justo --help
```

Instalación del paquete justo

A continuación, hay que instalar el paquete **justo** en el proyecto. Generalmente, se añade como dependencia de desarrollo. Según el proyecto, utilizaremos las propiedades **dependencies** o **devDependencies** del archivo **package.json**. Ambas cosas se pueden hacer de una vez mediante el siguiente comando **npm**:

```
npm install --save justo
npm install --save-dev justo
```

La primera instala **justo** y añade el paquete como dependencia principal. Mientras que la segunda, como dependencia de desarrollo.

Archivo package.json

Una vez instalado el paquete **justo**, lo siguiente es crear los archivos **package.json**, **Justo.json** y **Justo.js**. Estos archivos son específicos de cada proyecto.

El archivo **package.json** es específico de la plataforma **Node.js**. Como **Justo.js** está desarrollado bajo esta plataforma, todo proyecto **Justo** debe disponer de este archivo. Grosso modo, es un archivo de texto en formato **JSON** que contiene los metadatos específicos del proyecto. Entre las propiedades principales que se debe configurar, encontramos:

- **name** (string). Nombre del proyecto, componente o aplicación.
- **version** (string). Versión del proyecto según la especificación <http://semver.org/>.
- **description** (string). Breve descripción del proyecto.
- **keywords** (string[]). Array de palabras claves que describen el proyecto.
- **homepage** (string). Sitio web oficial del proyecto.

- **bugs** (object). Información sobre cómo notificar al desarrollador los posibles bugs que encuentren los usuarios.
- **author** (object). Autor del proyecto.
- **contributors** (object[]). Miembros participantes del equipo del proyecto.
- **files** (string[]). Archivos y/o directorios que se copiarán en cada instalación.
- **main** (string). Ruta relativa al archivo que se cargará cuando se importe el paquete.
- **dependencies** (object). Dependencias principales del proyecto.
- **devDependencies** (object). Dependencias específicas para el desarrollo del proyecto.
- **engines** (object). Versiones mínimas de las aplicaciones que requiere el proyecto como, por ejemplo, la de **node** y la de **npm**.

Para más información sobre este archivo, puede remitirse a su página oficial, docs.npmjs.com/files/package.json.

El archivo **package.json** se puede crear mediante los generadores **justo-generator-packagejson** y **justo-generator-justo**. El primero sólo genera el archivo **package.json**, mientras que el segundo también **Justo.json** y **Justo.js**. Por lo general, se utiliza el segundo.

Un **generador** (*generator*) es un componente que crea archivos y/o directorios. Se dedica un curso específico a los generadores. Grosso modo, se instalan globalmente y se ejecutan mediante **justo** con la opción **-g**. Así pues, si deseamos usar el generador **packagejson**, primero debe instalar su generador globalmente:

```
npm install -g justo-generator-packagejson
```

Y a continuación, crear el archivo, invocando el generador como sigue:

```
justo -g packagejson
```

Recordemos, que en **Node.js** las instalaciones globales sólo hay que hacerlas una única vez. Aunque no hay que olvidar comprobar lo instalado, de vez en cuando, por si hubiera nuevas versiones de nuestro interés para actualizarlas.

Archivo Justo.json

El archivo **Justo.json** contiene la configuración de **Justo** específica del proyecto. Usa el formato **JSON**. Es opcional, sólo se recomienda cuando se tiene que configurar algo de **Justo** explícitamente.

Se puede generar mediante el generador **justo-generator-justo**. Primero, hay que instalar el generador globalmente:

```
npm install -g justo-generator-justo
```

Y a continuación, generar el archivo:

```
justo -g justo
```

Este comando crea los archivos **Justo.js** y **Justo.json** en el directorio actual. Debemos hacerlo en el directorio raíz del proyecto. He aquí un ejemplo ilustrativo de este archivo:

```
{
  "runner": {
    "main": "Justo.js",
    "onError": "continue",
    "logger": {
      "minLevel": "info",
      "maxLevel": "fatal"
    }
  },
  "reporter": {
    "type": "coloredConsole",
    "coloredConsole": {
      "theme": {
        "task": {
          "title": {
            "simple": "gray",
            "composite": "cyan"
          }
        }
      }
    }
  }
}
```

```
    },  
    "result": {  
      "between": ["[ ", " ]"],  
      "ok": {  
        "text": "OK",  
        "color": "green"  
      },  
      "failed": {  
        "text": "ER",  
        "color": "red"  
      },  
      "ignored": {  
        "text": "IG",  
        "color": "white"  
      }  
    }  
  }  
}  
  
}  
  
}  
  
}
```

La propiedad `runner` contiene la configuración del ejecutor de tareas. Mientras que `reporter`, del impresor de tareas y resultados.

Mediante la propiedad `main`, se indica el archivo `JavaScript` en el que se define las tareas específicas del proyecto. De manera predeterminada, es `Justo.js` y no se recomienda cambiarlo. La propiedad `onError` indica qué debe hacer el ejecutor de tareas si alguna de ellas acaba en error: `continue`, continuar con la siguiente tarea; o `break`, detener la ejecución. Finalmente, la propiedad `logger` indica los niveles de registro a mostrar:

- **debug**. Mensajes de depuración.
- **info**. Mensajes de información.
- **warn**. Mensajes de aviso. Por ejemplo, algo que está deprecado.
- **error**. Mensajes de error. Por ejemplo, algo que va mal, pero el componente puede continuar sin problemas.
- **fatal**. Mensajes de error fatales. Algo ha ido mal y el componente no debería continuar trabajando porque podría acabar en un estado inconsistente.

Se puede configurar el nivel más bajo a mostrar, mediante `logger.minLevel` y el más alto, mediante `maxLevel`. De manera predeterminada son `info` y `fatal`, respectivamente.

Archivo Justo.js

Por otro lado, tenemos el archivo `Justo.js`, en el que se define las tareas específicas del proyecto. Consiste en un archivo `JavaScript`. He aquí un ejemplo ilustrativo:

```
//imports
const catalog = require("justo").catalog;
const babel = require("justo-plugin-babel");
const clean = require("justo-plugin-fs").clean;
const copy = require("justo-plugin-fs").copy;
const jshint = require("justo-plugin-jshint");
const publish = require("justo-plugin-npm").publish;

//catalog
catalog.workflow({name: "build", desc: "Build the package."}, function() {
  jshint("Best practices", {
    output: true,
    src: ["index.js", "lib/"]
  });

  clean("Clean build directory", {
    dirs: ["build/es5"]
  });
});
```

```

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

clean("Clean dist directory", {
  dirs: ["dist/es5"]
});

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

catalog.macro({name: "test", desc: "Unit test."}, {
  require: "justo-assert",
  src: ["test/unit/lib", "test/unit/index.js"]
});

catalog.workflow({name: "publish", desc: "NPM publish"}, function() {
  publish("Publish", {
    who: "justojs",
    src: "dist/es5/nodejs/justo-plugin-browserify/"
  });
});

catalog.macro({name: "default", desc: "Default task."}, ["build", "test"]);

```

Cada proyecto tiene sus tareas específicas, aquellas que automatizan ciertas actividades, trabajos o procesos. Se dan a conocer en este archivo.

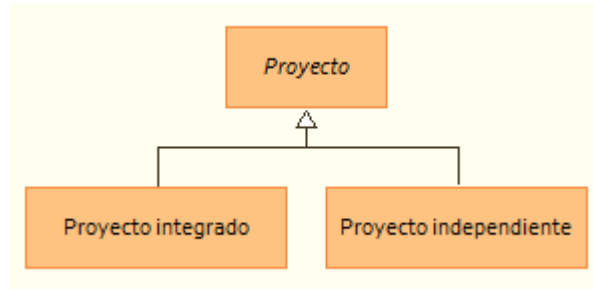
Recordemos que un archivo plantilla se puede crear con el generador [justo-generator-justo](#).

Importación del paquete justo

El archivo `Justo.js` debe importar `justo` como primer paquete. Para que pueda ser usado por cualquier *plugin* que vayamos a utilizar. En el ejemplo anterior, se utiliza algunos *plugins* como el que permite la invocación de `Babel`, `JSHint` o la ejecución de ciertas tareas del sistema de archivos como la copia y la supresión de archivos y/o directorios. Pero antes de la importación de *todos* ellos, se debe importar *necesariamente* `justo`.

Tipos de proyectos Justo.js

Un *proyecto* (*project*) es un trabajo para el desarrollo de algo. Debido a la naturaleza de `Justo`, básicamente podremos utilizarlo en dos tipos de proyectos: independientes e integrados.



Un **proyecto independiente** (*standalone*) es aquel que es autónomo y se dedica exclusivamente a **Justo.js**. Por ejemplo, para instalar **CouchDB**, **PostgreSQL** o **Redis** en una máquina, para instalar y desinstalar paquetes de una **Raspberry Pi** tras una instalación predeterminada, etc. En cambio, un **proyecto integrado** (*embedded project*) es aquel en el que **Justo** asiste a su proyecto principal. Se utiliza para automatizar ciertas tareas, pero no es la razón de ser del proyecto. Por ejemplo, en el desarrollo de un paquete de **Node.js**, se puede utilizar **Justo** para automatizar la compilación, el empaquetamiento, la publicación y las pruebas del paquete. En este caso, el proyecto principal es el paquete de software.

Los proyectos independientes tendrán el paquete **justo** entre sus dependencias principales en el archivo **package.json**, esto es, en la propiedad **dependencies**. Mientras que uno integrado, generalmente, como dependencia de desarrollo, **devDependencies**.