



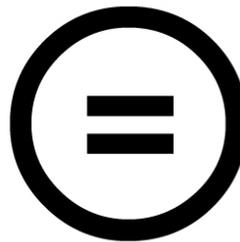
Libre
atix
F u n d a c i ó n



2018

10 Años Aportando al Conocimiento

Distribuido bajo:



2018 - Bolivia



<http://revista.atixlibre.org>

Twitter: @atixlibre

Facebook: facebook.com/Atix.Libre





DIRECCION GENERAL
Esteban Saavedra Lopez



DIAGRAMACION
Jenny Saavedra Lopez
Esteban Saavedra Lopez



REVISION
Jenny Saavedra Lopez



CONTACTO
info@atixlibre.org
<http://revista.atixlibre.org>



Atix Libre



**EL QUE LO
INTENTA**

**EL QUE LO
SABE**

**EL QUE LO
PUEDE**

**EL QUE LO
LOGRA**

Dicen que empezar nuevos proyectos es fácil, la tarea difícil es mantenerlos; nos costo mucho sacrificio pero después de 10 años continuamos firmes en poder brindar un medio que permita compartir conocimientos y experiencias entre nuestros autores y lectores.

Lo maravilloso es que siempre cada etapa se inicia con lo aprendido en la anterior, nosotros al igual que nuestros lectores aprendimos muchísimo durante todo este tiempo, pero lo más importante, supimos mantener nuestra esencia y nuestro compromiso.

Enseñar es dejar una huella en las vidas de otras personas, en **AtixLibre** estamos seguros que cumplimos ese objetivo, ya que brindamos lo que se encontraba en nuestras manos para poder aportar a su saber y sobre todo a la ardua tarea de difundir conocimiento y cultura de forma libre, sin distinciones ni prejuicios.

10 años atrás nos pusimos un lema "**Atix hacia un futuro innovador, por un mundo ético, libre y justo**", estoy seguro que con lo que realizamos hasta ahora, cumplimos a cabalidad lo propuesto; aunque tuvimos varios contratiempos, aun seguimos con el espíritu firme e intacto y con muchas ganas de continuar.

En nuestra vida aprendimos que el conocimiento no es nada, sino lo compartes y lo difundes, ahora es grande la satisfacción de haber dado los pasos que dimos; pocos tal ves, lentos muy posible, pero de seguro lo hicimos con la mayor predisposición y enorme placer.

Sean bienvenidos a nuestra edición número 23.



Esteban Saavedra L.

Presidente Fundación AtixLibre

Contenido

Número 23 - Mayo 2018

1

Node.JS
Javascript en el Servidor

2

LibrePlan
Gestión de Proyectos

3

Ansible
Aprovisionamiento de SW

4

Pensando en DDD
para aplicaciones MVC

5

Python 3
Estructura de Datos



Node.JS

Javascript en el servidor

Node.js es un entorno de ejecución para Javascript en el lado del servidor. Se basa en el uso de operaciones asíncronas para las operaciones de I/O (lectura/escritura) lo que evita bloqueos haciéndolo una elección a tomar en cuenta cuando se piensa en construir una aplicación de red escalable.

Introducción

Las operaciones de I/O¹ son pesadas (toma mucho tiempo realizarlas comparadas con otras operaciones), y en los lenguajes tradicionales esto implica utilizar un hilo para cada operación que se realiza. Node.js afronta este problema de otra manera ya que solamente cuenta con un hilo de ejecución, por lo cual “terciariza” dichos procesos al kernel del sistema operativo.



Desarrollo

La forma “tradicional” de realizar una operación en la mayoría de los lenguajes de programación es secuencial, por ejemplo, en una aplicación web se suele obtener información de una base de datos, esperamos a que la información nos sea devuelta para posteriormente tratarla y dar una respuesta al cliente. Este proceso se realiza secuencialmente y se espera a que cada línea de ejecución termine para continuar con la siguiente.

```
TAREA1();
RESULTADO_TAREA_2 = TAREA2_PETICION_BASE_DE_DATOS_O_RED();
TAREA3(RESULTADO_TAREA_2);
TAREA4();
```

El problema con esta forma de manejar las operaciones es que no todas las tareas tienen un

1 I/O se refiere a la interacción de un programa con el exterior, como por ejemplo con los dispositivos de almacenamiento (disco duro u otro) ó la red (solicitud de información a otros sistemas).

tiempo de respuesta corto; usualmente las operaciones que requieren I/O (en el ejemplo la lectura de la base de datos) toman un tiempo mayor que otras y nuestra aplicación (un hilo usualmente) se encuentra bloqueada hasta que la respuesta sea retornada. Esta forma de manejar las operaciones se denomina síncrona y se puede ver como se ejecutan las tareas una tras otra en el siguiente diagrama:

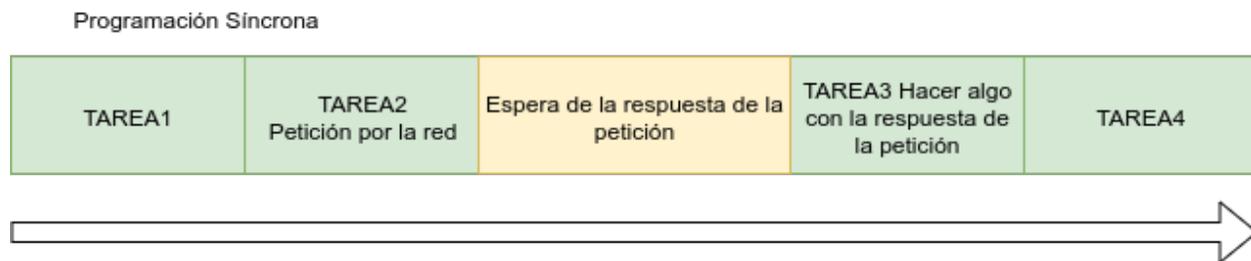


Ilustración 1: Programación síncrona

Programación asíncrona

La programación asíncrona es una forma de realizar operaciones de forma paralela donde una unidad de trabajo corre separadamente del hilo principal de ejecución.

Node.js es single threaded (se ejecuta solamente sobre un hilo), por lo tanto, si tengo una aplicación web y esta se queda esperando a que una operación se realice, muchas otras peticiones no se podrían atender.

Para solucionar este problema, se utiliza la programación asíncrona, donde se evita bloquear el hilo principal de ejecución dejando las operaciones de I/O al kernel del sistema operativo siempre que sea posible. Por ejemplo, un diagrama de la ejecución del ejemplo 1 en node.js pero de manera asíncrona sería:

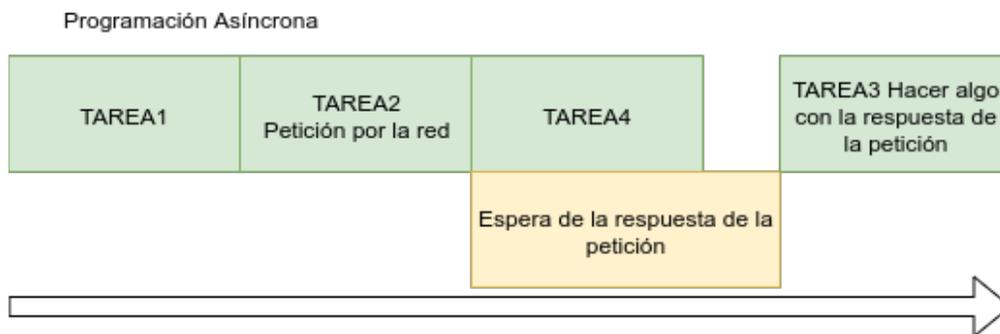


Ilustración 2: Programación asíncrona

Como se puede observar, la petición de la TAREA2 se realiza sin bloquear la ejecución de nuestro programa, una vez terminada la TAREA2 se puede continuar con la TAREA3.

Algunos de los conceptos que debemos conocer para comprender realmente el funcionamiento de node.js son:

Heap

Los objetos y variables creadas se almacenan en un espacio de memoria denominado heap.



Call stack

El call stack es la cola donde se almacenan de manera ordenada las funciones que se están ejecutando y/o que faltan ejecutarse.

Cuando comenzamos a ejecutar una función, esta se inserta en el call stack y cuando termina su ejecución (returns) se saca del call stack.

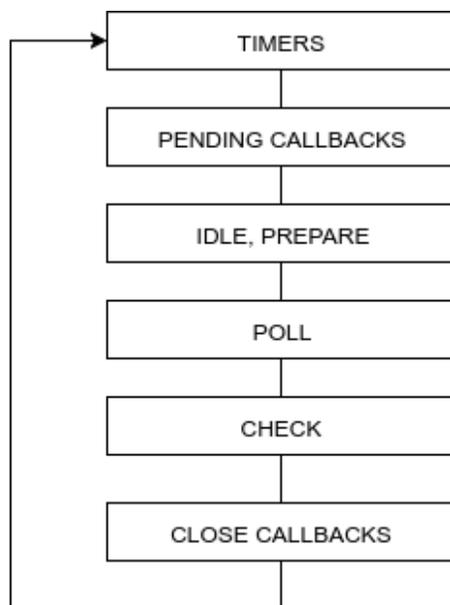
Muchas veces, cuando creamos un ciclo de llamadas a funciones que no tiene fin, este call stack se llena y luego se desborda lo que provoca el clásico error denominado “stack overflow”.

Los errores que la mayoría de los lenguajes imprimen vienen con una lista del call stack, lo que permite saber en que función falló nuestro programa.

Event loop

El event loop permite a node.js realizar operaciones I/O. Dado que el kernel es multi-threaded, se pueden ejecutar muchas operaciones en segundo plano, cuando una de las operaciones se completa, el kernel informa que la operación ha concluido y que se puede ejecutar la operación que le sigue.

Son varias fases las que se ejecutan en el event loop, y cada fase cuenta con una cola de funciones que deben ejecutarse (callbacks). Cuando en la cola de una fase no existen más funciones para ejecutar entonces se pasa a la siguiente fase.



Veamos un poco a detalle de que se realiza en cada fase:

- **Timers:** en esta fase se ejecutan las funciones programadas con `setInterval` y `setTimeout`.
- **Pending callbacks:** en esta fase se ejecutan las funciones en las que intervienen I/O y/o que fueron aplazadas en el último ciclo. Como la mayor parte del código que escribimos son callbacks, esta es la fase donde se ejecuta nuestro código.
- **Idle, prepare:** fase utilizada internamente por node.js.
- **Poll:** acá node.js se detiene para obtener nuevos mensajes de los eventos I/O. Los eventos que han sido retornados por el kernel del sistema son ejecutados en esta fase.

- **Check:** En esta fase se ejecutan las funciones programadas por `setImmediate`.
- **Close callbacks:** En esta fase se emiten eventos `close`. Estos eventos son lanzados por ejemplo cuando un `socket` se ha cerrado abruptamente.

A cada ciclo del event loop se le denomina tick.

Dado que `node.js` es `single threaded`, en máquinas que tienen varios núcleos se debe utilizar el módulo `cluster`, lo que permite generar un proceso hijo (`child process`) por cada núcleo. Cada proceso hijo tiene su propio event loop y el proceso maestro (`master process`) transparentemente distribuye la carga entre los procesos hijos. También es posible utilizar otras herramientas como `pm2`², `phusion passenger`³ u otros.

Si una tarea tarda demasiado en ejecutarse en el event loop, este se bloquea por lo cual no se ejecuta ninguna tarea adicional hasta que la tarea que se encuentra bloqueando el event loop finalice su ejecución.

Pero, si `node.js` es `single threaded`, ¿a dónde van las tareas que se ejecutan asincrónicamente?

Las tareas asíncronas se van a un `worker pool` y son implementadas por `libuv`⁴.

Lo más importante si se quiere poder construir un servidor que responda a una gran cantidad de solicitudes al mismo tiempo es no bloquear el event loop.

`Node.js` utiliza el motor `v8`⁵ de google para ejecutar todo el código javascript que se encuentra optimizado para la mayoría de las operaciones, sin embargo, existen expresiones regulares y otras operaciones síncronas que pueden resultar desastrosas para la ejecución de cualquier programa (se deja de responder a solicitudes).

Algunos módulos a tomar en cuenta para no bloquear

Varios de los módulos del núcleo de `node.js` tienen operaciones síncronas que son bastante pesadas (bloquean el event loop) por lo cual deben utilizarse conociendo sus ventajas y desventajas.

Las librerías de cifrado (`crypto`), del sistema de archivos (`fs`) y compresión (`zlib`), en general las funciones que no deben utilizarse suelen terminar con la palabra `sync`, por ejemplo `readFileSync`.

Por ejemplo, una función síncrona puede utilizarse cuando el servidor de `node.js` está levantando pero debe evitarse su uso cuando se está respondiendo a solicitudes de los clientes.

Ejemplos para comprender la naturaleza asíncrona de node.js

Ahora que ya conocemos un poco del núcleo de `node` y varios conceptos, veamos algunos ejemplos para comprender mejor como se ejecutan las solicitudes realmente y conocer algunas funciones básicas del lenguaje.

Ejemplo 1: realizamos la impresión de “uno”, “dos” y “tres”.

- `console.log` permite imprimir a `stdout` con un salto de línea.

```
console.log('uno');
console.log('dos');
console.log('tres');
```

2 <https://github.com/Unitech/pm2>

3 <https://github.com/phusion/passenger>

Cuando este código sea ejecutado se imprimirá en la consola:

```
uno  
dos  
tres
```

Nada nuevo o diferente comparado con otro lenguaje, las operaciones se ejecutan una tras otra sin problemas.

Ejemplo 2: realizamos la impresión de “uno”, “dos” y “tres”, pero esperando un segundo (1000ms) para la impresión de “dos”.

setTimeout: permite ejecutar una función después de determinado tiempo. (Debe recordarse que los timers son asíncronos)

```
console.log('uno');  
setTimeout(function () {  
    console.log('dos');  
}, 1000);  
console.log('tres');
```

Cuando este código sea ejecutado se imprimirá en la consola:

```
uno  
tres  
dos
```

La explicación es simple, se ejecuta el primer **console.log** (uno), luego se programa una función para ejecutarse en un segundo, entonces se imprime el tercer **console.log** (tres); por último luego de un segundo de espera se ejecuta la función que tiene el segundo **console.log** (dos).

Ejemplo 3: realizamos la impresión de uno, dos y tres, y establecemos una función que se ejecute después de cero segundos.

```
console.log('uno');  
setTimeout(function () {  
    console.log('dos');  
}, 0);  
console.log('tres');
```

Ahora sin embargo suceden algunas cosas interesantes, veamos la respuesta que el programa devuelve:

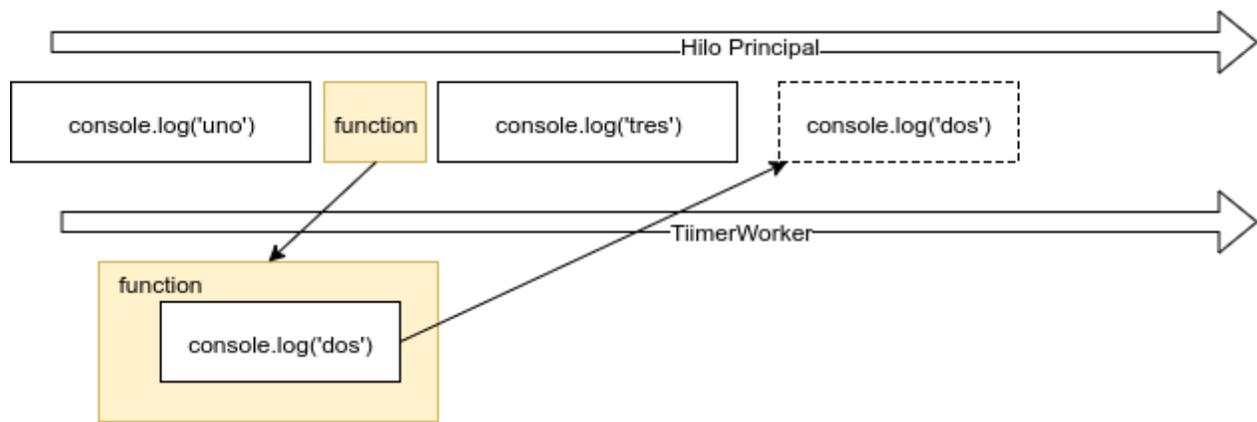
```
uno  
tres  
dos
```

Pero, ¿porqué “dos” se imprimió al final si el tiempo de espera era 0 segundos?

Esto es debido a lo que se explicó previamente del event loop y se debe a que las funciones asíncronas se ejecutan en un worker. Veamos las partes involucradas:

4 <http://docs.libuv.org/en/v1.x/threadpool.html>

5 <https://developers.google.com/v8/>



Pasos que realiza el motor de node.js:

1. Ejecuta `console.log('uno')`
2. Envía la función `setTimeout` al worker de timers.
3. Ejecuta `console.log('tres')`
4. Como todavía quedan funciones que deben ejecutarse en el call stack, se obtiene el resultado del timer que ya se ha ejecutado y se ejecuta la función que imprime `console.log('dos')`.
5. El programa finaliza porque no quedan operaciones pendientes.

Como se indicó previamente, la función con el timer se ejecuta en el siguiente tick, por lo cual pese a que tiene un tiempo de cero se ejecutará al final.

Conclusiones

- Node.js es un lenguaje de programación con el cual podemos ejecutar javascript en el servidor y es ejecutado por el motor v8 de chrome, y las tareas asíncronas son ejecutadas por libuv⁶.
- En un principio es complicado comprender como funciona internamente node.js y especialmente la programación asíncrona, pero una vez que se entiende los procesos internos, es muy sencillo programar con node.js.
- El problema de adopción de node.js, es cultural, ya que a todos nos enseñan como realizar las operaciones una tras otra y pasar a un estilo de programación asíncrona suele presentar dificultades e incluso miedo en un principio.
- Si bien node.js permite realizar un procesamiento eficiente de los procesos que involucran I/O, no es un remedio para todo tipo de problema, siempre debe utilizarse la herramienta (o lenguaje) que permita resolver los problemas de manera más sencilla y eficiente. Es importante que los procesos que necesitan hacer uso alto del procesador (CPU) sean trasladados a un worker externo para que no bloqueen el hilo principal de node.js, o incluso, posiblemente que sean implementados en otro lenguaje.

El siguiente artículo introducirá el uso de promesas y varias funciones útiles de node.js.

6 <http://libuv.org/>



Referencias

- [1] The node craftsman book - <https://leanpub.com/nodecraftsman>
- [2] Documentación oficial de node.js - <https://nodejs.org/en/>
- [3] <https://aprendiendo-nodejs.blogspot.com/2017/07/entendiendo-el-event-loop.html>
- [4] Call stack - https://es.wikipedia.org/wiki/Pila_de_llamadas
- [5] Queue - [https://es.wikipedia.org/wiki/Cola_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica))



Ariel Alvarado
Desarrollador
alvarado.ariel.contact@gmail.com

BOLIVIA



LibrePlan

Gestión de Proyectos

El éxito final de un proyecto está en la adecuada gestión de sus variables, como ser recursos, tiempo, costos, etc. Las herramientas más utilizadas para la gestión de proyectos son EDT, Diagrama Gantt, Costos, entre otros; pero lo más relevante no es es el poder gestionar un solo proyecto a la vez, sino todo un Portafolio de Proyectos. LIBREPLAN es un software que nos ayuda en la gestión de Proyectos individuales y en la gestión de un Portafolio de Proyectos.

Introducción

Para la correcta gestión de proyectos es necesario tener una vista rápida de la EDT (Estructura de Descomposición/Desglose de Trabajo) y su respectivo Diagrama Gantt, con las consideraciones necesarias respecto a recursos, tiempo y costos asociados. Adicionalmente, es necesario también una vista rápida del Portafolio de Proyectos que nos permite tener de forma óptima y oportuna el control de los proyectos que conforman el portafolio y de los recursos que normalmente son los mismos en su mayoría para los proyectos que conforman el portafolio.



Características

Entre las características de LIBREPLAN están:

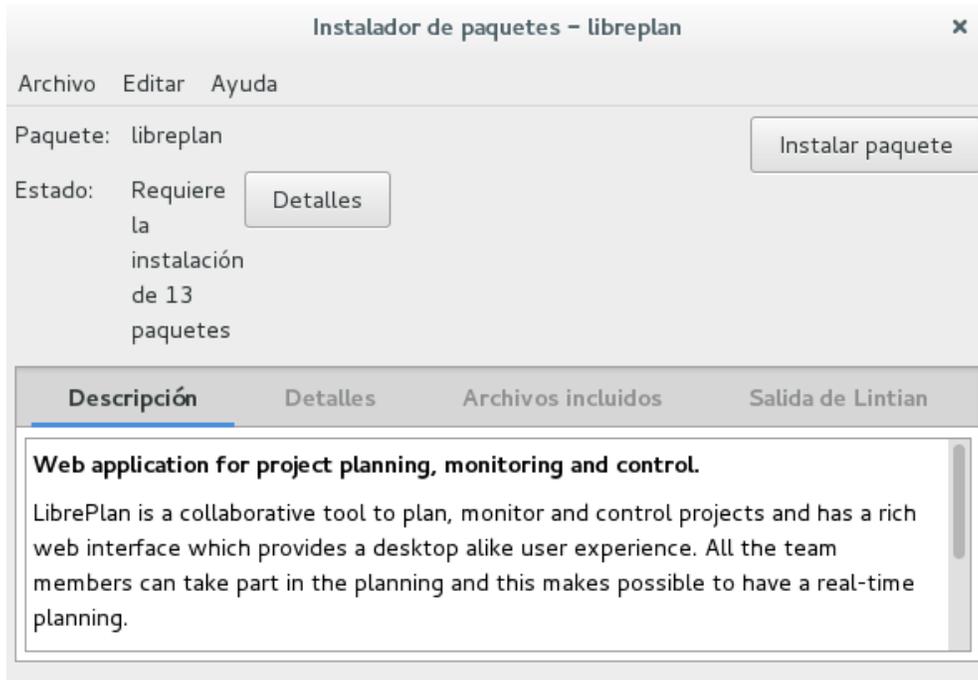
- Multiproyecto, permite la gestión de un Portafolio de Proyectos.
- Gestión de recursos.
- Planificación, a través de EDT, plantillas, reasignación de recursos automática, etc.
- Asignación avanzada de tareas.
- Simulación Monte Carlo, para la estimación de fechas de conclusión de proyectos.
- Control y seguimiento de proyectos.
- Subcontrataciones y gestión de materiales por tarea.
- Colaborativo, ya que es una plataforma web la cual permite la interacción de todos los gestores del o los proyectos y puede también integrarse con autenticación LDAP.

Instalación

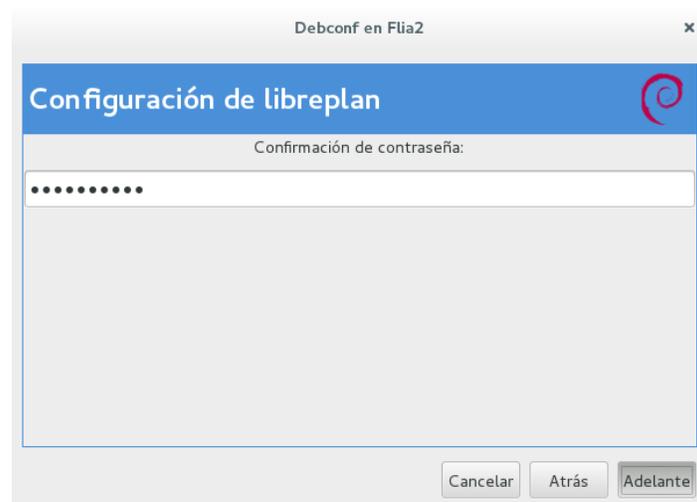
Para la versión LIBREPLAN v1.4 es necesario como prerequisite la instalación del Web Server TOMCAT v6.

LIBREPLAN v1.4 se lo puede bajar de <https://sourceforge.net/projects/libreplan/> (paquetes en formato DEB).

Para la instalación se puede utilizar los diferentes métodos y/o programas de gestión de paquetes, en este caso utilizamos Gdebi:

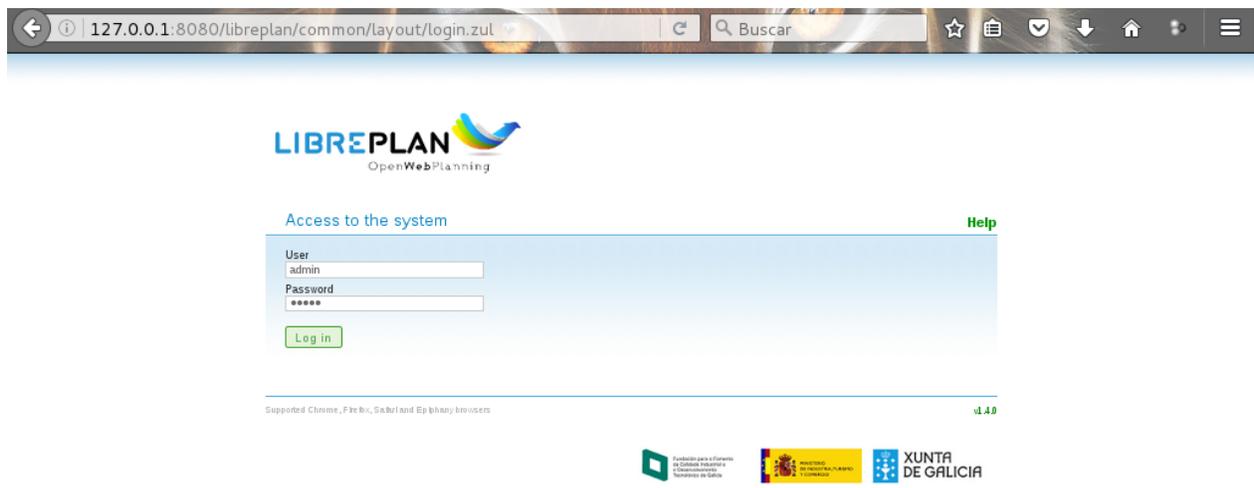


En el proceso de instalación se creará la clave de administración de la base de datos y aplicación:



Después de la instalación, se puede ingresar con un Web Browser a:

<http://127.0.0.1:8080/libreplan/common/layout/login.zul>



La documentación o manual de usuario, se habilita con la instalación de la aplicación en el mismo Web Server y se encuentra en:

<http://127.0.0.1:8080/libreplan/help/en/index.html>



LibrePlan: User documentation

LIBREPLAN OpenWebPlanning Following document contains necessary help for using LibrePlan, the opensource web application for project planning. This document is divided in three main sections:

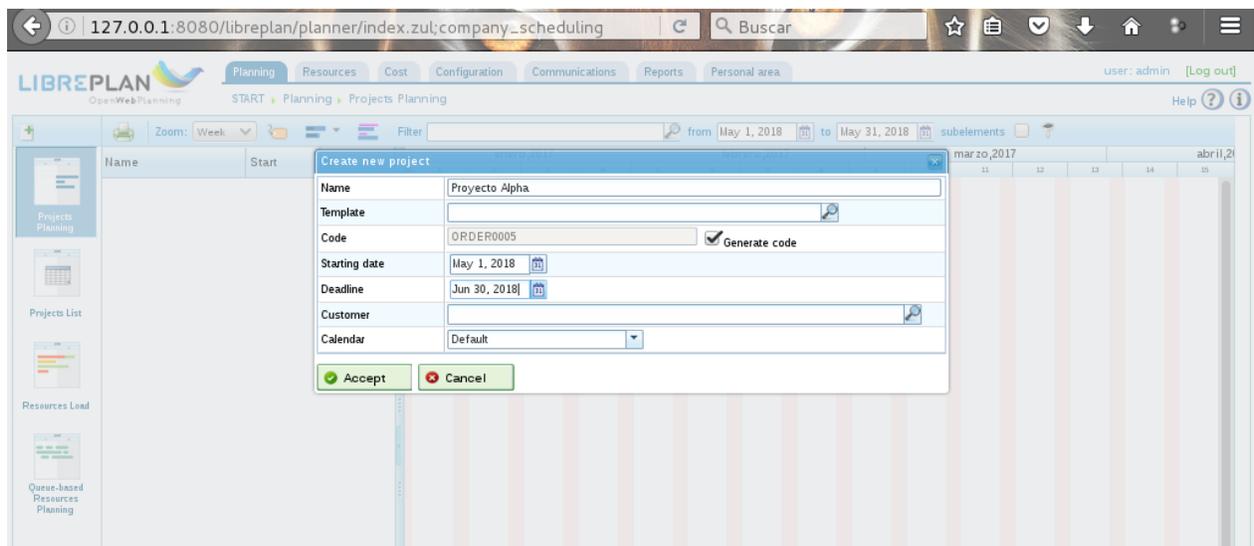
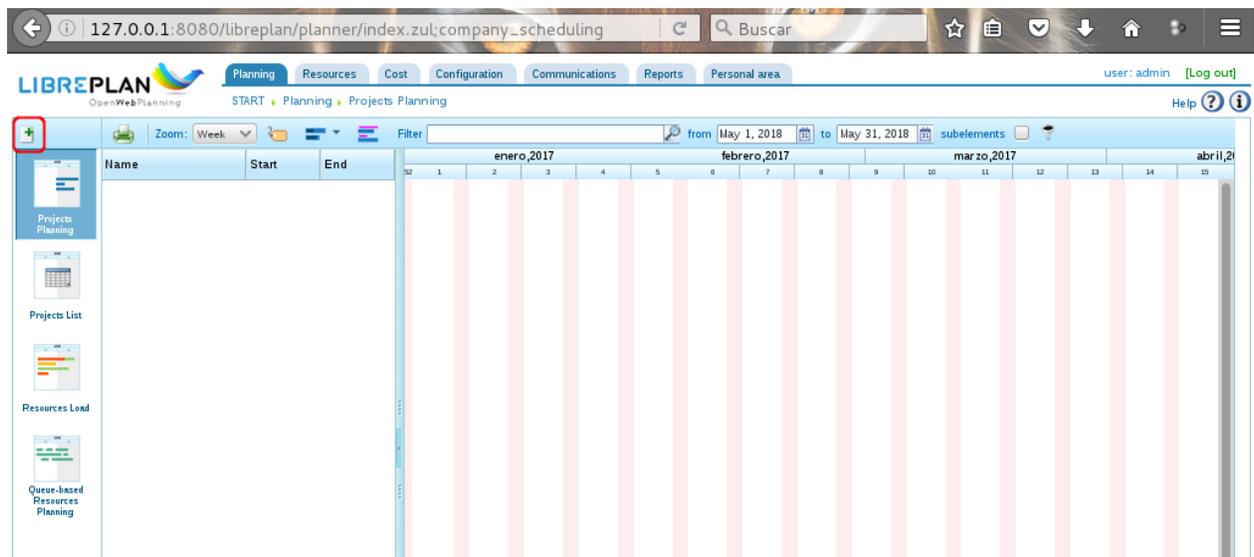
- First, global goals and behaviour.
- Second, basic entities to understand the minimum concepts to use LibrePlan.
- Finally, complete processes description to create orders, projects, project planning, resources assignment, advance assignment and result extraction.

1. [Introduction](#)
2. [Criteria](#)
3. [Calendars](#)
4. [Progress](#)
5. [Management of resources](#)
6. [Orders and order elements](#)
7. [Task planning](#)
8. [Assignment of resources](#)
9. [Work reports](#)
10. [Tags](#)
11. [Materials](#)
12. [Quality forms](#)
13. [Users](#)
14. [Cost management](#)
15. [Reports](#)
16. [LDAP Configuration](#)
17. [Project dashboard](#)
18. [Connectors](#)
19. [Scheduler](#)
20. [About](#)

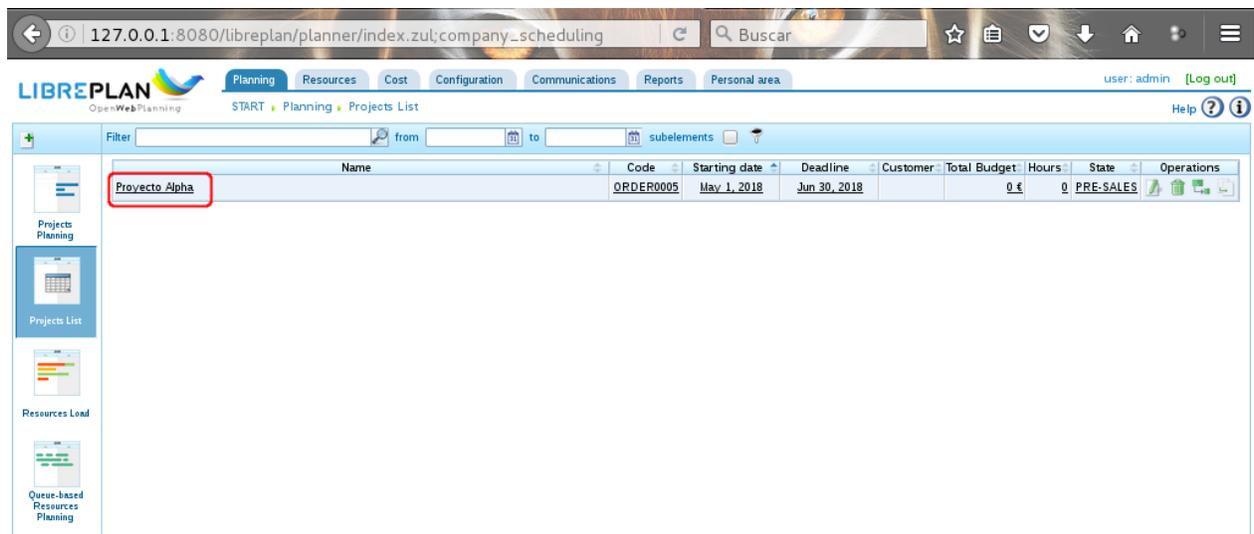
Ejemplo

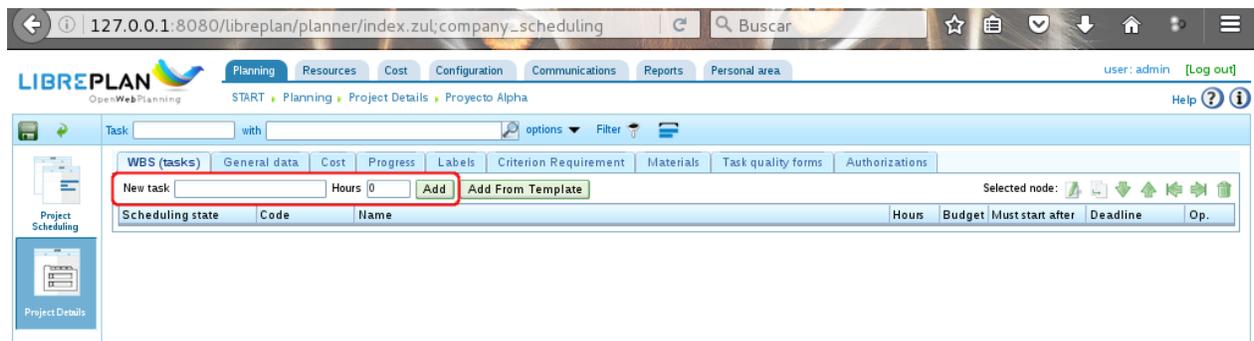
A continuación, explicamos en un ejemplo la creación de dos proyectos y sus recursos asociados.

- 1) Primero creamos un proyecto llamado "Proyecto ALPHA":

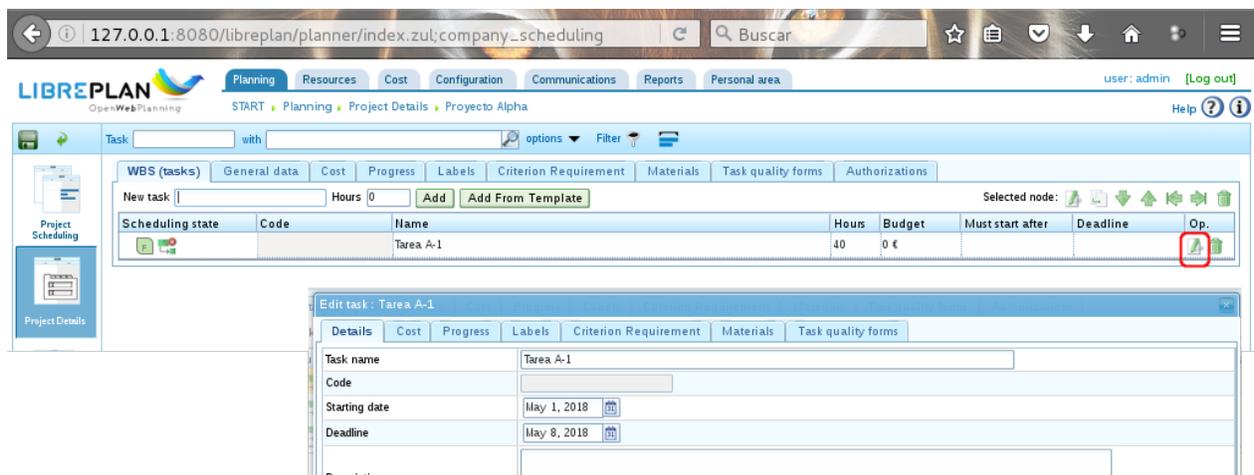


2) A continuación, creamos tres tareas:



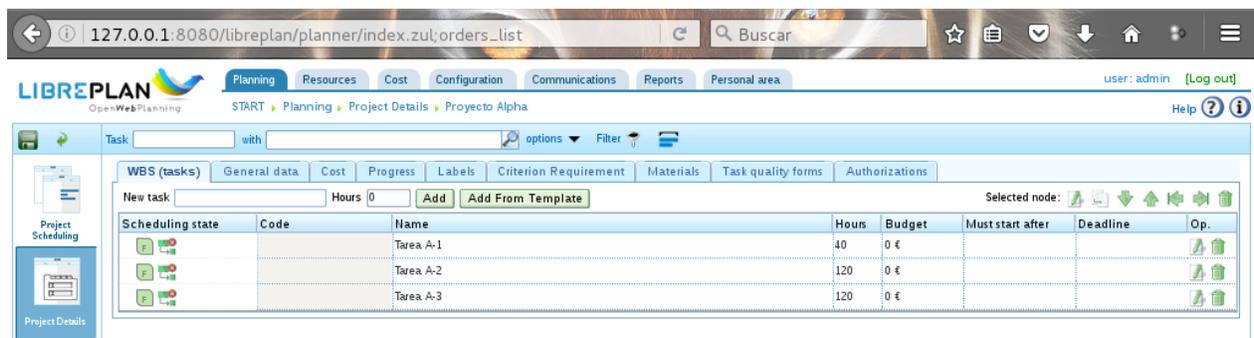


Editamos la tarea creada para agregar el costo y fechas:



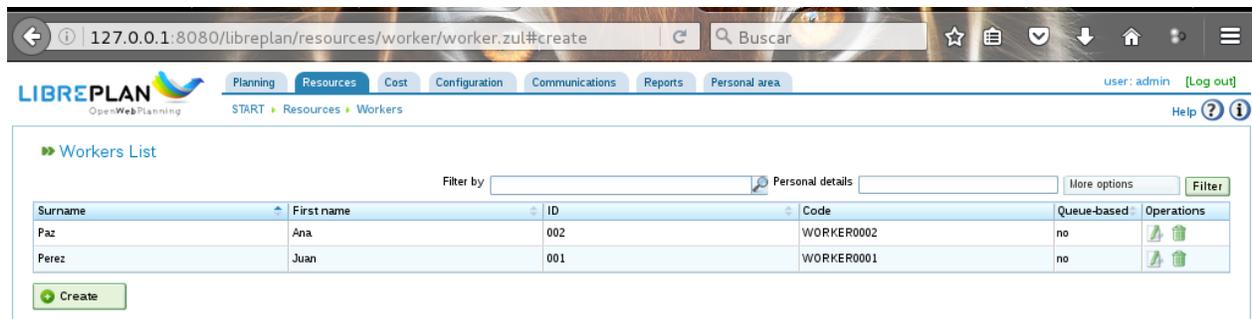
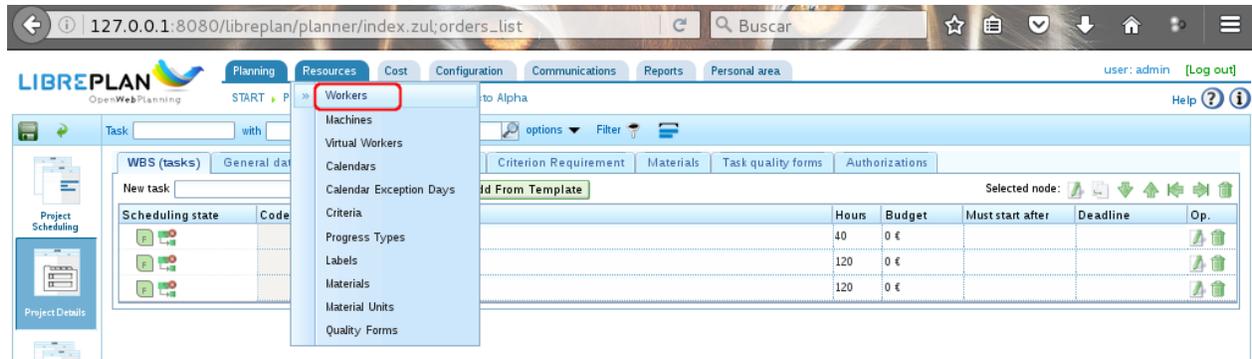
Scheduling state	Code	Name	Hours	Budget	Must start after	Deadline	Op.
		Tarea A-1	40	0 €			

Agregamos dos tareas adicionales de ejemplo:

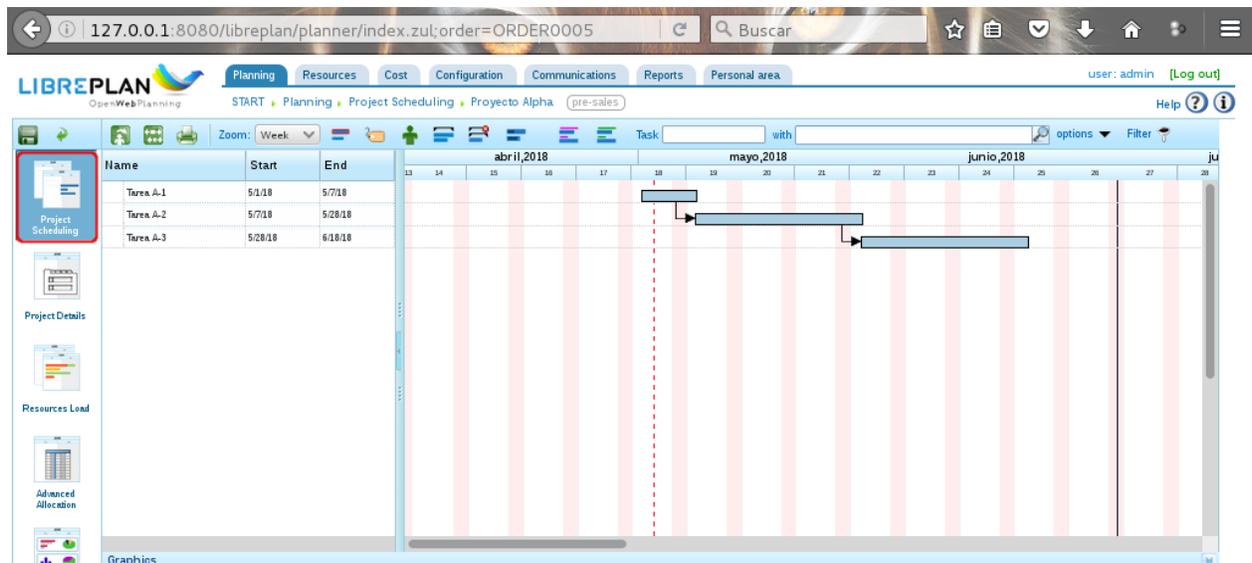


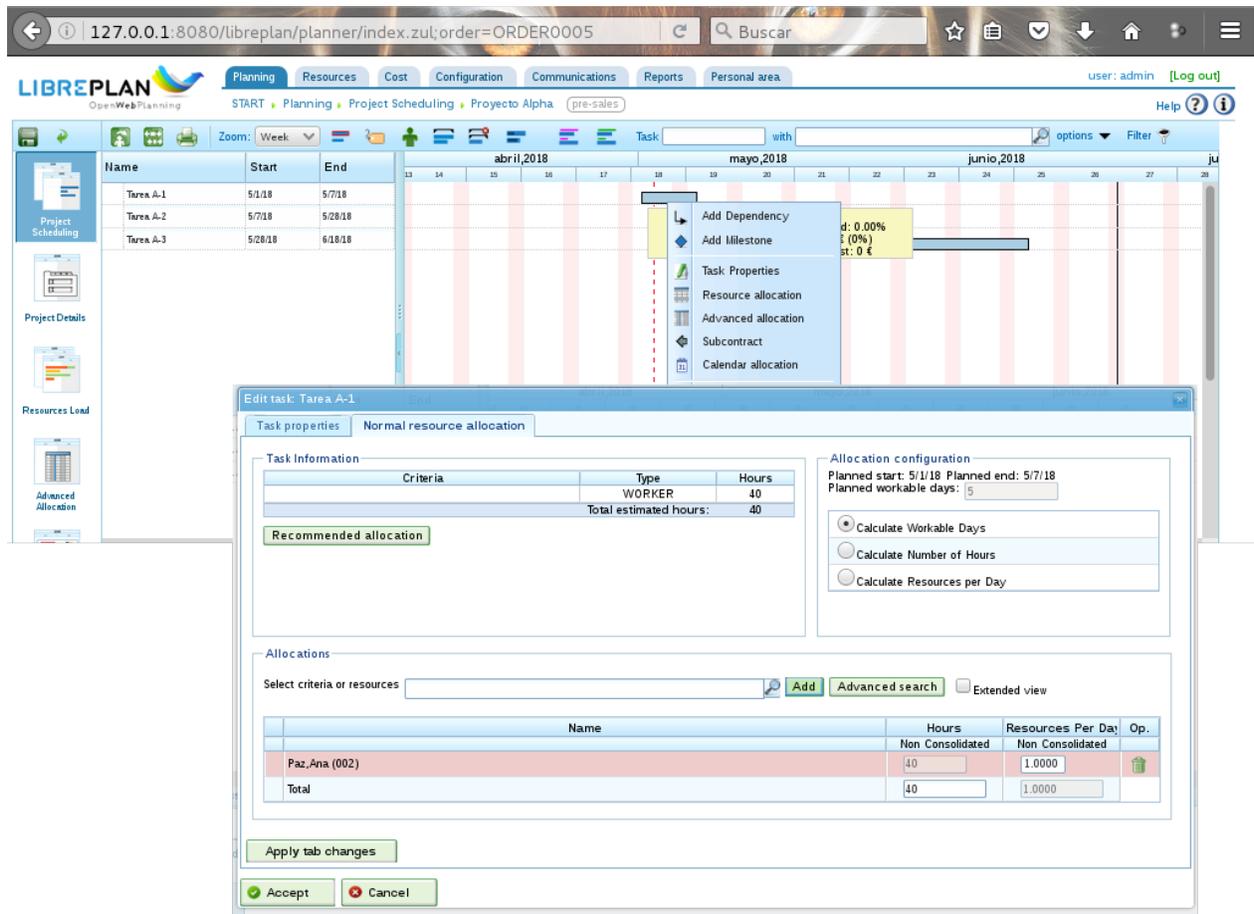
Scheduling state	Code	Name	Hours	Budget	Must start after	Deadline	Op.
		Tarea A-1	40	0 €			
		Tarea A-2	120	0 €			
		Tarea A-3	120	0 €			

3) Ahora, creamos dos recursos llamados “Ana Paz” y “Juan Perez”:



4) Asignamos los recursos a las tareas:

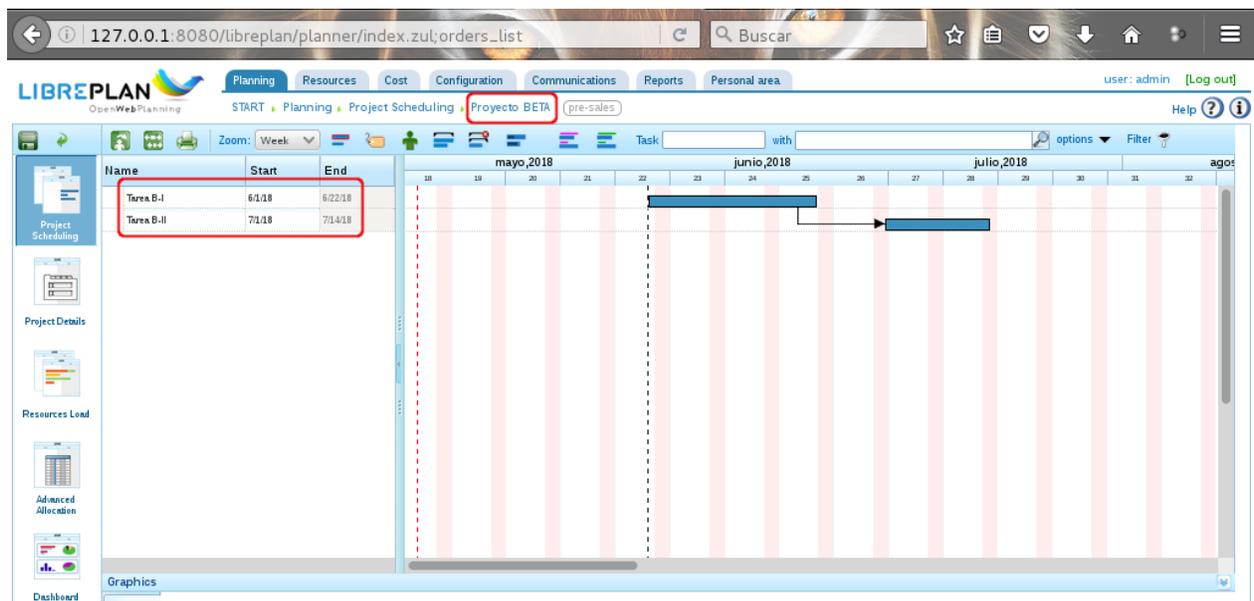




The screenshot shows the LibrePlan web application interface. At the top, the browser address bar displays '127.0.0.1:8080/libreplan/planner/index.zul;order=ORDER005'. The application header includes navigation tabs: Planning, Resources, Cost, Configuration, Communications, Reports, and Personal area. The user is logged in as 'admin'. The main view is a Gantt chart for 'Proyecto Alpha' (pre-sales) spanning from April to June 2018. A context menu is open over a task bar, showing options like 'Add Dependency', 'Add Milestone', 'Task Properties', 'Resource allocation', 'Advanced allocation', 'Subcontract', and 'Calendar allocation'. A dialog box titled 'Edit task: Tarea A-1' is open, showing 'Task properties' and 'Normal resource allocation'. The 'Task Information' section includes a table with columns 'Criteria', 'Type', and 'Hours', showing a total of 40 hours for a 'WORKER' type. The 'Allocation configuration' section shows 'Planned start: 5/1/18', 'Planned end: 5/7/18', and 'Planned workable days: 5'. The 'Allocations' section has a table with columns 'Name', 'Hours', 'Resources Per Day', and 'Op.', showing an allocation for 'Paz, Ana (002)' with 40 hours and 1.0000 resources per day. The dialog box has 'Accept' and 'Cancel' buttons at the bottom.

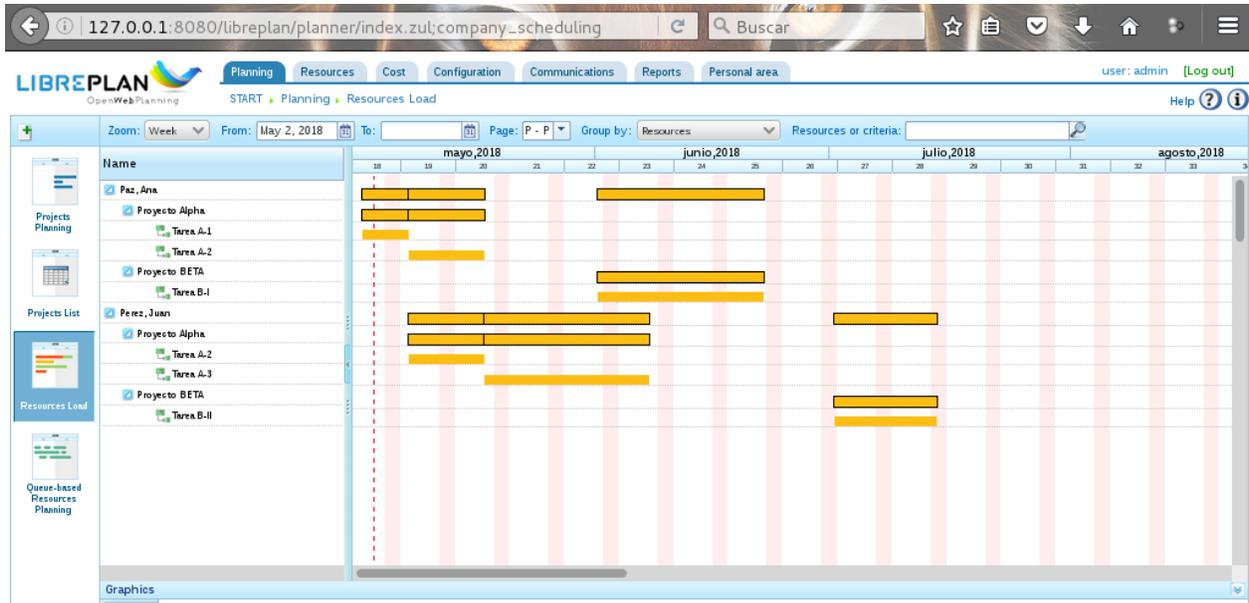
Repetimos la asignación de recursos a todas las tareas.

5) Para tener un ejemplo de portafolio de proyectos creamos un segundo proyecto llamado "Proyecto BETA" con dos Tareas y les asignamos los recursos ya creados "Ana Paz" y "Juan Perez":



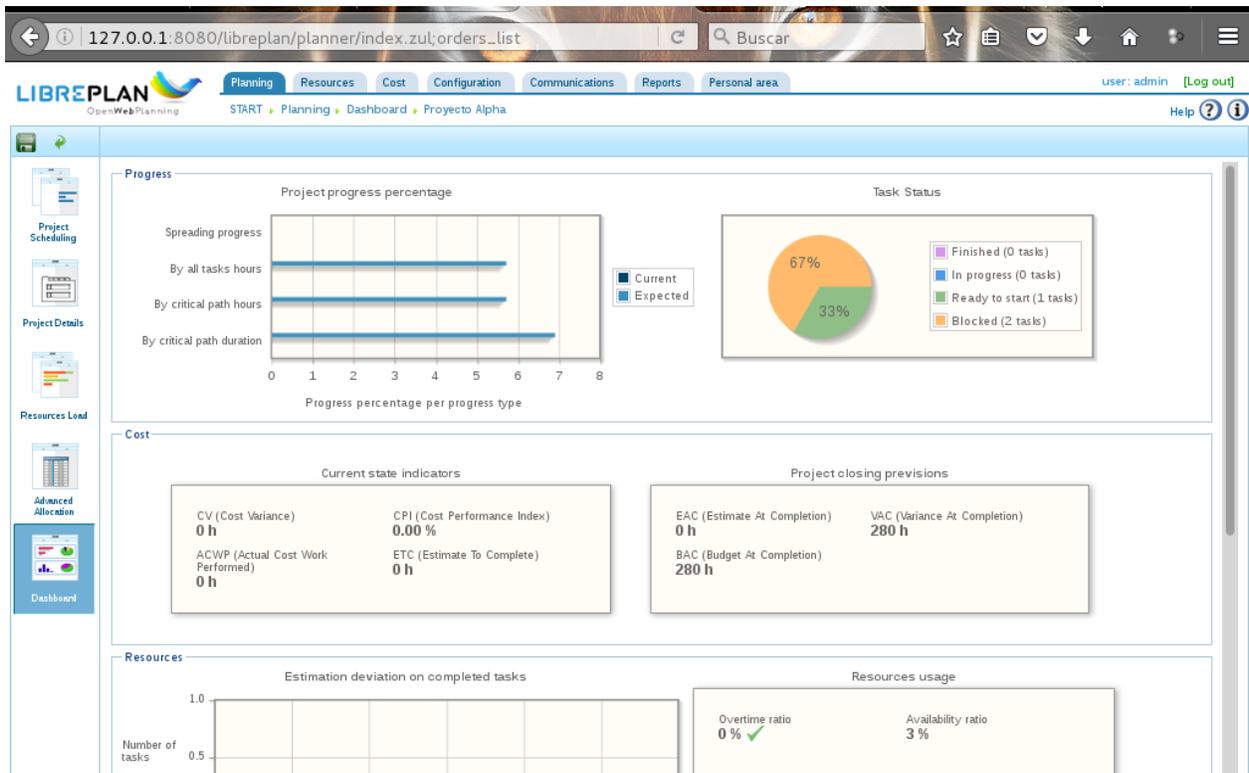
The screenshot shows the LibrePlan web application interface for 'Proyecto BETA'. The browser address bar displays '127.0.0.1:8080/libreplan/planner/index.zul;orders_list'. The application header is the same as in the previous screenshot. The main view is a Gantt chart for 'Proyecto BETA' (pre-sales) spanning from May to August 2018. Two tasks are highlighted with a red box: 'Tarea B-I' (Start: 6/1/18, End: 6/22/18) and 'Tarea B-II' (Start: 7/1/18, End: 7/14/18). The Gantt chart shows task bars and resource allocation lines. The 'Proyecto BETA' label in the breadcrumb navigation is highlighted with a red box.

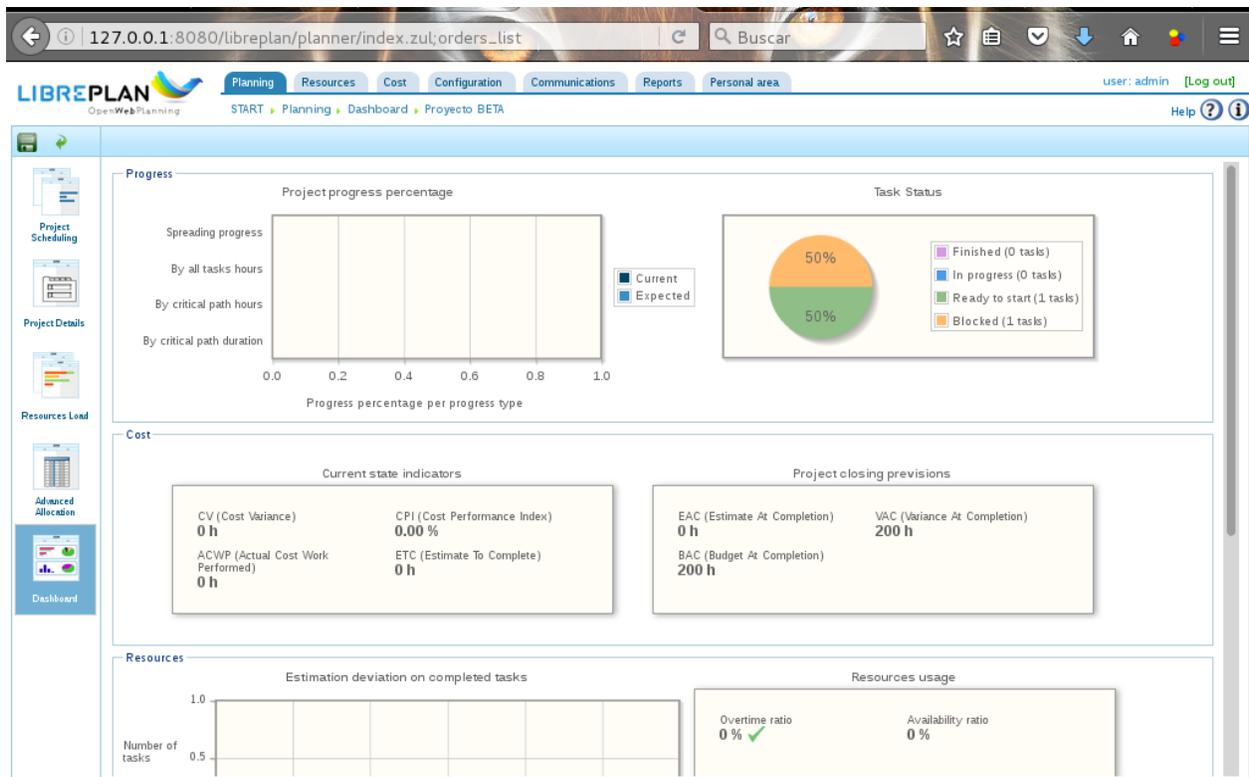
6) Ahora podemos ver los Recursos versus las Tareas de todos los proyectos:



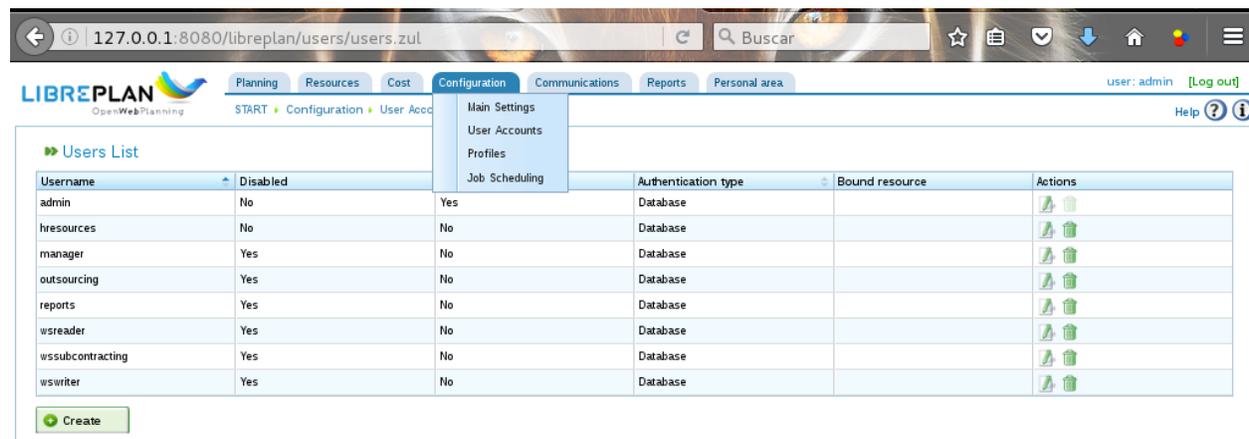
En esta vista podemos ajustar los recursos en cuanto a tiempo, en caso de que se tenga sobrecarga, etc. y podremos ver de forma automática la reasignación de recursos.

7) Podemos ver también los Dashboards de cada Proyecto:





8) Por último podemos crear usuarios de acceso al LIBREPLAN con diferentes roles y perfiles, según la necesidad de cada proyecto y/o el Portafolio de Proyectos:



The screenshot shows the 'Users List' page in LIBREPLAN. A dropdown menu is open over the 'Configuration' tab, listing options like Main Settings, User Accounts, Profiles, and Job Scheduling. The main table lists the following users:

Username	Disabled	Job Scheduling	Authentication type	Bound resource	Actions
admin	No	Yes	Database		
hresources	No	No	Database		
manager	Yes	No	Database		
outsourcing	Yes	No	Database		
reports	Yes	No	Database		
wsreader	Yes	No	Database		
wssubcontracting	Yes	No	Database		
wswriter	Yes	No	Database		

A 'Create' button is visible at the bottom left of the table.

Conclusiones

- Este tipo de herramienta nos facilita la Gestión de Proyectos y el control del Portafolio de Proyectos. Los reportes generados nos brindan información muy importante y detallada para la administración de recursos de forma oportuna.

- Al ser una aplicación con la información centralizada en un Servidor Web, la actualización del avance de tareas, demoras, nuevas tareas, etc. pueden ser realizadas por los usuarios pertenecientes a cada proyecto de forma independiente y como resultado tener los Dashboards actualizados de forma inmediata. Estos indicadores o Dashboards nos ayudan a la toma de decisiones oportuna y así poder alinear la ejecución de los proyectos según la estrategia propuesta.

Referencias

[1] <http://www.libreplan.org/>



3 Ansible

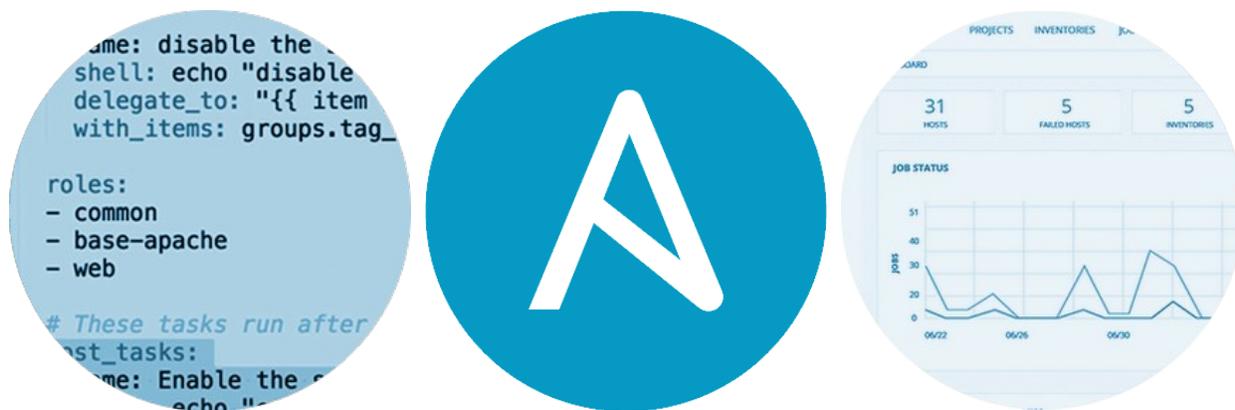
Aprovisionamiento de Software

En un mundo donde la existencia de arquitecturas variables, contextos heterogéneos y una gran cantidad de nodos por gestionar, la tarea de administrar se ha convertido en una tarea tediosa y muchas veces repetitiva, por lo que surge la necesidad de contar con herramientas e instrumentos que permitan a los SysAdmin y Developers automatizar estos procesos de forma sencilla, entendible y segura, independientemente si se encuentran en arquitecturas de nodos físicos, virtuales o en la nube.

Ansible es una plataforma de software libre para configurar y administrar ordenadores y/o servidores.

Que es ansible

Ansible es una herramienta que nos permite gestionar configuraciones, aprovisionamiento de recursos, orquestación, despliegue automático de aplicaciones y muchas otras tareas de TI de una forma limpia, sencilla y segura.



Ansible permite

- Instalaciones multi-nodo
- Ejecuciones de tareas ad hoc
- Administración de configuraciones
- Manejar nodos a través de SSH

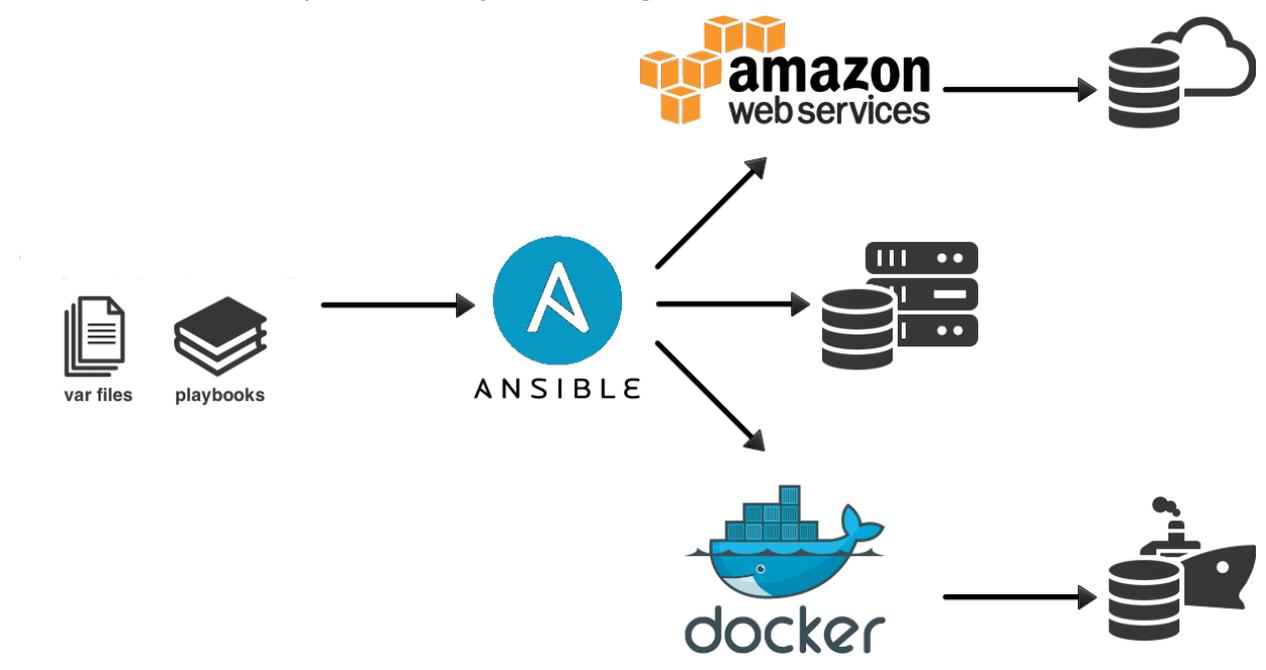
Para todo esto y mucho más, no requiere ningún software remoto adicional (excepto Python 2.4 o posterior).

Características

- **Mínimalista:** Es una sistema de administración que no impone dependencias adicionales.
- **Consistente:** Es muy robusto y fuerte en su funcionalidad
- **Seguro:** Ansible no instala agentes vulnerables en los nodos. Solamente se requiere OpenSSH que es considerado crítico y altamente testeado.
- **Alta confiabilidad:** El modelo de idempotencia es aplicado para las instalaciones y configuraciones, para prevenir efectos secundarios en la ejecución repetitiva de scripts.
- **Curva de aprendizaje mínima:** ya que hace uso de de los playbooks, los cuales usan un lenguaje descriptivo simple, basado en YAML.

Integración con diferentes arquitecturas

Independientemente de la arquitectura que disponga nuestra infraestructura, Ansible puede instalarse en ambientes físicos, virtualizados, nubes públicas y privadas; también puede instalarse en entornos para analizar y archivar big data.



¿En qué se diferencia de otras herramientas similares?

- No necesita agentes
- No requiere de configuraciones engorrosas y complicadas
- Flexibilidad (API, Módulos, Plugins)
- Facilidad de uso

Componentes

- **Máquina de Administración:** es la máquina donde tenemos instalado Ansible, y desde donde podremos ejecutar las tareas y/o gestionar los nodos.
- **Inventario:** es una descripción de los nodos que pueden ser accedidos por Ansible (máquina de administración). El inventario está descrito por un archivo de configuración, en formato INI, cuya ubicación por defecto es `/etc/ansible/hosts`. En el archivo de configuración se listan las direcciones IP o hostname de cada nodo que es accesible por Ansible. Además, los nodos pueden ser asignados a grupos.
- **Playbook:** describen configuraciones, despliegue, y orquestación; el formato del Playbook es YAML. Cada Playbook asocia un grupo de hosts a un conjunto de roles, cada rol está representado por llamadas a lo que Ansible define como Tareas. En resumen un PlayBook lista las tareas que deben ejecutarse en los diferentes nodos.
- **Tarea:** son bloques dentro del Playbook donde se define una acción específica a realizar (instalar paquetes, ejecutar scripts, verificar estado de actualizaciones, etc).
- **Módulo:** son las unidades de trabajo en Ansible; cada módulo es auto-suficiente y puede ser escrito en lenguaje estándar de scripting, como ser Python, Perl, Ruby, Bash, etc. Una de las propiedades principales de los módulos es la idempotencia la cual asegura que ninguna operación se realizará una vez que el sistema ha alcanzado el estado deseado.
- **Roles:** permiten ordenar los diferentes Playbooks.
- **Facts:** variables dentro de Ansible que contienen información sobre los servidores (Sistema Operativo, Cantidad de Ram, Direcciones IP, etc)
- **Handlers:** código que se usa cuando algo cambia o existen excepciones (si actualizas el archivo de configuración de Apache, un Handler re-iniciará el servicio httpd).

Como funciona Ansible

- Ansible no precisa instalar ni Agentes, solo un equipo a partir del cual se realizará la gestión de los diferentes nodos.
- Ansible se conecta a los nodos que se desea gestionar, aprovisionar u orquestar utilizando una conexión SSH, mediante la cual le envía una serie de instrucciones a ejecutar y/o configuraciones que se desee aplicar.

¿Quiénes usan Ansible actualmente?

Por su simplicidad Ansible es utilizado por muchísimas empresas. Entre ellas:

- Atlassian
- NASA
- Evernote
- Spotify
- Verisign
- RedHat
- Amazon

Instalación

- RedHat y Centos: `yum install ansible`
- Debian: `apt-get install ansible`

Pasos para realizar la gestión de nodos

- Conectar el equipo de administración con los nodos
- Configurar el inventario de nodos
- Realizar pruebas de conectividad
- Realizar la gestión y/o aprovisionamiento

Conectar el equipo de administración con los nodos

Generar la llave

Generar la llave en el nodo administrador

```
ssh-keygen
```

Esto genera

```
~/.ssh/id_rsa  
~/.ssh/id_rsa.pub
```

Copiar la llave pública

Copiar la llave del nodo administrador a los nodos a administrar

```
ssh-copy-id -i ~/.ssh/id_rsa.pub root@web01.atixlibre.org  
ssh-copy-id -i ~/.ssh/id_rsa.pub root@web01.atixlibre.org  
ssh-copy-id -i ~/.ssh/id_rsa.pub root@web01.atixlibre.org
```

Archivos de configuración

```
/etc/ansible/
```

Configurar el inventario de nodos

```
/etc/ansible/hosts
```

```
[servidoresweb]  
web01.atixlibre.org  
web02.atixlibre.org  
web03.atixlibre.org
```

```
[basesdedatos]
db01.atixlibre.org
db02.atixlibre.org
db03.atixlibre.org
```

Realizar pruebas de conectividad

Ejecución ad hoc

```
ansible all -m ping -u root
```

Resultado

```
192.168.101.104 | success >> {
  "changed": false,
  "ping": "pong"
}
```

Ejecución sin -u root

```
192.168.100.10 | FAILED => SSH encountered an unknown error during the connection. We
recommend you re-run the command using -vvvv, which will enable SSH debugging output to
help diagnose the issue
```

Ejecución desde un playbook

```
playbook_test.yml
```

```
- hosts: servidoresweb
  remote_user: root
tasks:
- name: test de conexión
  ping:
```

```
ansible-playbook playbook_test.yml
```

Partes de una tarea

Name: Cada tarea tiene un nombre, que sirve para identificar la tarea y también para el seguimiento de la misma.

Ejemplo

```
tasks:
- name: Verificar si apache esta corriendo
  service: name=httpd state=running
```

notify: son las acciones que se ejecutarán al final de cada tarea en el Playbook. Sólo serán ejecutadas una vez, incluso cuando sean llamadas por diferentes tareas. Por ejemplo: Muchas tareas pueden solicitar el reinicio de algún servicio específico, pero este sólo será reiniciado una sola vez.

Ejemplo

```
notify:
- restart apache
```

Ejecución

```
ansible-playbook playbook.yml
```

Ejemplo de playbook

Crear un Playbook llamado `webserver.yml`, con el que vamos a instalar las herramientas necesarias para publicar una página web en varios servidores.

```
#####
# Hosts donde se realizarán las tareas
#####

- hosts: servidoresweb
  user: root

#####
# Instalación de paquetes necesarios.
#####

tasks:
- name: General | Instalación de paquetes requeridos.

action: apt pkg={{ item }} state=installed with_items:
- php5
- apache2
- mysql-server
- mysql-client
- php5-mysql
- php-apc
- php5-xmllrpc
- php-soap
- php5-gd
- unzip
- python-mysqldb

#####
# Configuración de Apache2.
#####

- name: Apache2 | Habilitar módulos
  action: command a2enmod rewrite vhost_alias

#####
# Reinicio de servicios
#####

- name: Reiniciar Apache
  action: service name=apache2 state=restarted
```

Ejecución del playbook

ansible-playbook webserver.yml

```
PLAY [webserver] *****
GATHERING FACTS *****
ok: [192.168.101.104]

TASK: [General | Instalación de paquetes requeridos.] *****
changed: [192.168.101.104] => (item=php5,apache2,mysql-server,mysql-client,php5-
mysql,php-apc,php5-xmlrpc,php-soap,php5-gd,unzip,python-mysqldb)

TASK: [Apache2 | Habilitar modulos] *****
changed: [192.168.101.104]

TASK: [Restart Apache] *****
changed: [192.168.101.104]

PLAY RECAP *****
192.168.101.104      : ok=4    changed=3    unreachable=0    failed=0
```

Referencias

[1] <http://www.ansible.com>



Esteban Saavedra L.
Presidente Fundación AtixLibre
esteban.saavedra@atixlibre.org

BOLIVIA

4 Pensando DDD para aplicaciones MVC

Este artículo habla sobre la reestructuración de una aplicación web Rest utilizando el patrón Modelo Vista Controlador (MVC) con conceptos de Diseño Orientado al Dominio (DDD o Domain Driven Design) aislando cada parte del sistema en base a las reglas de negocio que definen su dominio y así hacerlo mas entendible, acoplable y mantenible.

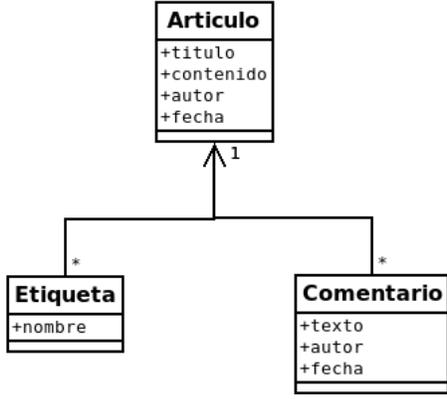
Introducción

Para la implementación del servicio utilizaremos las siguientes tecnologías:

- **NodeJS**, como entorno de ejecución para nuestra aplicación
- **Patrón MVC** (Modelo-Vista-Controlador) como el patrón principal para la estructura inicial de la aplicación
- **REST** (Transferencia de estado representacional) como estilo de arquitectura del sistema
- **DDD** (Diseño orientado al dominio) como el enfoque que se propone utilizar para estructurar de mejor manera la aplicación.

Desarrollo

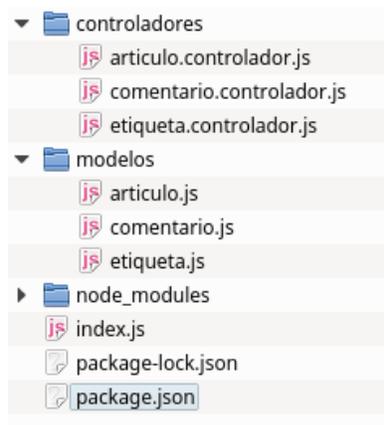
La aplicación que se desarrollará será la de un blog, con artículos, etiquetas y comentarios bajo el siguiente esquema:



El código se encuentra disponible en github en el siguiente enlace: <https://github.com/nardhar/nodejs-ddd>, y se tendrán tags para cada uno de los pasos de este artículo

Estructura Básica de aplicación REST con Express y Sequelize con el Patrón MVC (1 pagina)
 Primero creamos nuestra aplicación MVC siguiendo el ejemplo indicado por Sequelize (enlace)

pero cambiando la estructura de las carpetas para aplicar el modelo MVC de la siguiente manera:



En `index.js` se describe la lectura de los archivos de los modelos y controladores

Los archivos de la carpeta `modelos` son los típicos archivos de definición de Modelo de Sequelize

En la carpeta `controladores` tenemos toda la lógica de guardado de artículos en conjunto con sus etiquetas, así como el guardado de comentarios, como ejemplo veremos la estructura del archivo `controladores/articulo.controlador.js`

```

1  module.exports = (router, models) => {
2    router.get('/articulo', (req, res, next) => {
3      return models.Articulo.findAndCountAll()
4        .then(articulos) => {
5          return res.status(200).json(articulos);
6        })
7      .catch(next);
8    });
9
10   router.post('/articulo', (req, res, next) => {
11     return models.Articulo.create(req.body)
12       .then(articulo) => {
13       if (articulo) {
14         return res.status(201).json(articulo);
15       }
16       return res.status(400).json({});
17     })
18     .catch(next);
19   });
20
21   // metodos get/:id, put y delete acortados para facilitar lectura
22   router.get('/articulo/:id', (req, res, next) => {=
23
24
25
26
27
28
29
30
31
32
33   router.put('/articulo/:id', (req, res, next) => {=
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50   router.delete('/articulo/:id', (req, res, next) => {=
51
52
53
54
55
56   };
57
  
```

El código de la aplicación se encuentra en el `tag v0.1-inicial`

Observaciones

- Si queremos modificar el método de creación de artículo (e.g.: para agregar validaciones), debemos cambiar el código del controlador, debiendo verificar que no se altere el código correspondiente a la respuesta que se envía al cliente.
- En caso de que se quiera agregar una funcionalidad para guardar artículos con etiquetas opcionales, se debe combinar parte de la lógica creada en `articulo.controlador.js` y `etiqueta.controlador.js` en otro archivo o hacer que `articulo.controlador.js` se comporte condicionalmente de acuerdo a los parámetros enviados, lo cual ya suena algo complicado. ¿Y si quisiéramos incluir un método que permita guardar artículos, etiquetas y comentarios en conjunto? (más complicado aún).
- El esquema actual no nos permita cambiar fácilmente el modo de almacenamiento, dependemos demasiado del ORM Sequelize, es decir que si necesitáramos cambiar a otro tipo de almacenamiento (mongoDB), debemos modificar todos los controladores, arriesgándonos a modificar la lógica del negocio.
- No existe una capa que se pueda definir como el corazón del negocio del sistema, aquel archivo o archivos que permitan identificar como crear un artículo y sus relaciones. Aunque podría decirse que los controladores asumen ese rol, pero también cumplen otras funciones como la de recopilar los parámetros de la petición y también de darle formato a la respuesta para cumplir con el estilo de arquitectura REST, por lo que son algo más que el corazón del negocio del sistema (demasiada responsabilidad).

Reestructuración de la aplicación con Diseño Orientado al Dominio

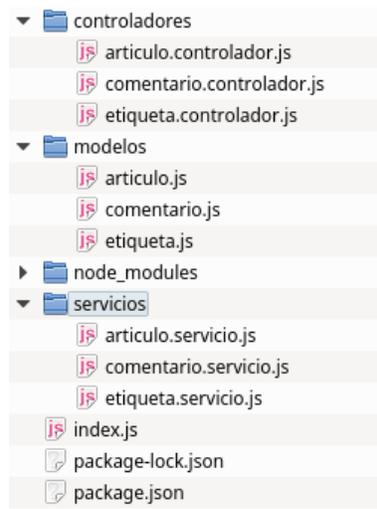
El Diseño Orientado al Dominio es una forma de desarrollar software estableciendo como principal objetivo el desarrollo de la lógica del dominio que vendría a ser el corazón del sistema.

Los artefactos que deben construirse son los siguientes:

- **Entidad**, un objeto que esta definido por una identidad en vez de los atributos que lo definen.
- **Objeto Valor**, un objeto que no tiene una identidad y esta definido por los atributos que lo componen.
- **Fábrica**, métodos para la creación de objetos que permiten ser fácilmente cambiables por otros dependiendo de la necesidad de la lógica del negocio.
- **Agregados**, cuando una colección de objetos corresponden a una entidad raíz que los controla, esta es un agregado.
- **Servicios**, cuando una operación no corresponde conceptualmente a ningún objeto se debe crear un método utilitario para implementarla y así resolver el problema.
- **Repositorios**, métodos para obtener objetos de un repositorio especial de forma que se pueda alterna entre implementaciones del almacenamiento.
- **Eventos de Dominio**, un objeto que define un evento que ocurre dentro del alcance del sistema.

¿Y como aplicamos estos patrones en nuestra aplicación?, primeramente vemos que Entidad y Objeto Valor en realidad se aplican con la capa del modelo (solo en el caso de nuestra aplicación), pero el resto de capas los aplica en su totalidad el controlador, por lo que se debe dividir sus responsabilidades en al menos una capa mas, la cual denominaremos servicio para

mayor comodidad (no confundir con el patrón Servicio de DDD), modificando nuestra aplicación de la siguiente manera:



Modificando los métodos de los controladores hacia los servicios de la siguiente manera:

Controlador:

```

1  module.exports = (router, servicios) => {
2  router.get('/articulo', (req, res, next) => {
3    return servicios.articulo.listar()
4    .then(articulos) => {
5      return res.status(200).json(articulos);
6    }
7    .catch(next);
8  });
9
10 router.post('/articulo', (req, res, next) => {
11   return servicios.articulo.crear(req.body)
12   .then(articulo) => {
13     return res.status(201).json(articulo);
14   }
15   .catch(next);
16 });
17
18 router.get('/articulo/:id', (req, res, next) => {
19   return servicios.articulo.obtener(req.params.id)
20   .then(articulo) => {
21     return res.status(200).json(articulo);
22   }
23   .catch(next);
24 });
25
26 router.put('/articulo/:id', (req, res, next) => {
27   return servicios.articulo.actualizar(req.params.id, req.body)
28   .then(articulo) => {
29     return res.status(200).json(articulo);
30   }
31   .catch(next);
32 });
33
34 router.delete('/articulo/:id', (req, res, next) => {
35   return servicios.articulo.eliminar(req.params.id)
36   .then(() => {
37     return res.status(200).json({});
38   })
39   .catch(next);
40 });
41 };
42
  
```



Servicio:

```
32 // metodos Factory
33 articuloServicio.crear = (params) => {
34   return articuloServicio.guardar(articuloServicio.construir(params));
35 };
36
37 articuloServicio.actualizar = (id, params) => {
38   return articuloServicio.obtener(id)
39     .then((articulo) => {
40       return articuloServicio.guardar(articulo, params);
41     });
42 };
43
44 articuloServicio.obtener = (id) => {
45   return articuloServicio.encontrarUno({ id })
46     .then((articulo) => {
47       if (!articulo) throw new Error('Artículo no encontrado');
48       return articulo;
49     });
50 }
51
52 articuloServicio.eliminar = (id) => {
53   return articuloServicio.obtener(id)
54     .then(articuloServicio.destruir);
55 };
56
57 // metodos Aggregate
58 articuloServicio.crearConEtiquetas = (params) => {
59   return articuloServicio.crear(params.articulo)
60     .then((articulo) => {
61       const etiquetaList = params.etiquetas.map((etiqueta) => {
62         return servicios.etiqueta.crear(params.etiqueta);
63       });
64
65       return Promise.all(etiquetaList).then(() => {
66         return articuloServicio.obtener(articulo.id);
67       });
68     });
69 };
70
71 return articuloServicio;
72 };
```

Utilizando los patrones del Diseño orientado al Dominio de la siguiente forma :

- **Repositorio**, los métodos que recuperan u operan directamente con el almacenamiento de los datos: Base de Datos.
- **Fábrica**, los métodos que construyen un objeto desde los parámetros enviados o desde el repositorio.
- **Agregado**, los métodos que unen varias operaciones a través de una raíz, usando los métodos de fábrica y de repositorio.
- **Servicios**, los métodos que componen una operación compleja entre dos o más dominios.

El código de esta versión está en el **tag v0.2-ddd**



Conclusiones

Los controladores tienen responsabilidades limitadas convirtiéndose en una capa para recuperar los datos de la petición, enviarlos a los servicios y posteriormente enviar la respuesta al cliente.

Los métodos de repositorio permiten limitar el acceso al almacenamiento (base de datos) haciendo que un cambio sobre los mismos no cambie el núcleo del sistema.

Los métodos de fábrica (construcción y manipulación de objetos) permiten ser modificados fácilmente sin que se tenga que cambiar el acceso a la base de datos o la comunicación con el cliente.

Al agregar la capa de servicio orientada al dominio, se tiene aislado el núcleo del sistema en métodos pequeños y escalables, de forma que ahí se enfoque el desarrollo principal del sistema.

Referencias

- [1] DDD Community
- [2] Domain Driven Design Quickly – InfoQ 2006



Felix Tribeño
Desarrollador
nardhar.mure@gmail.com

BOLIVIA



Python 3

Estructura de Datos

Hora de dejar de usar Python 2, y actualizar los conocimientos de la inmensa biblioteca estándar para Python 3. En las próximas entregas de esta columna introduciré un módulo de la biblioteca estándar de Python en casa número de esta revista. El texto es la traducción al español del sitio Python 3 Module of the Week de Doug Hellman, que ha sido publicado como el libro "The Python 3 Standard Library by Example".

Estructuras de datos

Python incluye varias estructuras de datos de programación estándar, como list, tuple, dict, y set, como parte de sus tipos incorporados. Muchas aplicaciones no requieren otras estructuras, pero cuando lo hacen, la biblioteca estándar proporciona versiones poderosas y bien probadas que están listas para ser usadas.

El módulo enum proporciona una implementación del tipo enumeración, con capacidades de iteración y comparación. Se puede usar para crear símbolos bien definidos para valores, en lugar de usar cadenas literales o enteros.

El módulo collections incluye implementaciones de varias estructuras de datos que amplían las encontradas en otros módulos. Por ejemplo, Deque es una cola de doble extremo, que permite la adición o eliminación de elementos de ambos extremos. El defaultdict es un diccionario que responde con un valor predeterminado si falta una clave, mientras que OrderedDict recuerda la secuencia en la que se agregan los elementos a ésta. namedtuple extiende el tuple normal para dar a cada elemento un nombre de atributo además de un índice numérico.

Para grandes cantidades de datos, un array puede hacer un uso más eficiente de memoria que una list. Como el array está limitado a un solo tipo de datos, puede usar una representación de memoria más compacta que una list de propósito general. Al mismo tiempo, las instancias de array se pueden manipular usando muchos de los mismos métodos que list, por lo que es posible reemplazar una list con un array en una aplicación sin muchos cambios.

Ordenar elementos en una secuencia es un aspecto fundamental de la manipulación de datos. La list de Python incluye un método sort(), pero a veces es más eficiente mantener una lista ordenada sin volver a clasificarla cada vez que su contenido cambia. Las funciones en heapq modifican el contenido de una lista mientras preservan el orden de la lista con poca sobrecarga.

Otra opción para crear listas ordenadas o arrays es bisect. Éste utiliza una búsqueda binaria para encontrar el punto de inserción para nuevos elementos, y es una alternativa a ordenar repetidamente una lista que cambia frecuentemente.

Aunque la list incorporada puede simular una cola usando los métodos insert() y pop(), no es segura para subprocesos. Para una comunicación ordenada entre hilos de verdad usa el módulo queue. multiprocessing incluye una versión de Queue que funciona entre procesos, haciendo más fácil convertir un programa multi-hilo para que use procesos en su lugar.

struct es útil para decodificar datos de otra aplicación, quizás viniendo de un archivo binario o una secuencia de datos, en tipos Python nativos para una manipulación más fácil.

Este capítulo cubre dos módulos relacionados con la gestión de la memoria. Para estructuras de datos altamente interconectadas, como gráficos y árboles, usa weakref para mantener referencias mientras se permite que el recolector de basura limpie los objetos una vez que ya no son necesarios. Utiliza las funciones en copy para duplicar estructuras de datos y sus contenidos, incluida la realización de copias recursivas con deepcopy().

La depuración de las estructuras de datos puede llevar mucho tiempo, especialmente cuando recorriendo la salida impresa de grandes secuencias o diccionarios. Utiliza pprint para crear representaciones fáciles de leer que pueden ser impresas en la consola o escritas en un archivo de registro para una depuración más fácil.

Finalmente, si los tipos disponibles no cumplen con los requisitos, subclassifica uno de los tipos nativos y personalízalo, o construye un nuevo tipo de contenedor utilizando una de las clases base abstractas definidas en collections como punto de partida.

enum – Tipo de Enumeración

El módulo enum define un tipo de enumeración con capacidades de iteración y comparación. Puede ser usado para crear símbolos bien definidos para valores, en lugar de usar enteros literales o cadenas.

Creando enumeraciones

Una nueva enumeración se define utilizando la sintaxis class creando una subclase de Enum y agregando atributos de clase que describen los valores.

```
# enum_create.py
```

```
import enum
class BugStatus(enum.Enum):
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1
print('\nMember name: {}'.format(BugStatus.wont_fix.name))
print('Member value: {}'.format(BugStatus.wont_fix.value))
```

Los miembros del Enum se convierten en instancias cuando la clase es analizada. Cada instancia tiene una propiedad name que corresponde al nombre del miembro y una propiedad value correspondiente al valor asignado al nombre en la definición de la clase.

```
$ python3 enum_create.py
```

```
Member name: wont_fix
Member value: 4
```

Iteración

Iterar sobre la clase emun produce los miembros individuales de la enumeración.

```
# enum_iterate.py
```



```
import enum
class BugStatus(enum.Enum):
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

Los miembros se producen en el orden en que se declaran en la definición de la clase. Los nombres y valores no se usan para ordenarlos de ninguna manera.

```
$ python3 enum_iterate.py
```

```
new          = 7
incomplete   = 6
invalid      = 5
wont_fix     = 4
in_progress  = 3
fix_committed = 2
fix_released = 1
```

Comparando Enums

Como los miembros de la enumeración no están ordenados, solo admiten comparación por identidad e igualdad.

```
# enum_comparison.py
```

```
import enum
class BugStatus(enum.Enum):
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1
actual_state = BugStatus.wont_fix
desired_state = BugStatus.fix_released
print('Equality:',
      actual_state == desired_state,
      actual_state == BugStatus.wont_fix)
print('Identity:',
      actual_state is desired_state,
      actual_state is BugStatus.wont_fix)
print('Ordered by value:')
try:
    print('\n'.join(' ' + s.name for s in sorted(BugStatus)))
except TypeError as err:
    print(' Cannot sort: {}'.format(err))
```

Los operadores de comparación mayor que y menor que elevan excepciones TypeError.

```
$ python3 enum_comparison.py
```

```
Equality: False True
Identity: False True
Ordered by value:
  Cannot sort: '<' not supported between instances of 'BugStatus' and 'BugStatus'
```

Usa la clase IntEnum para enumeraciones donde los miembros necesitan comportarse más como números – por ejemplo, para ofrecer comparaciones.

```
# enum_intenum.py
```

```
import enum
class BugStatus(enum.IntEnum):
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1
print('Ordered by value:')
print('\n'.join(' ' + s.name for s in sorted(BugStatus)))
```

```
$ python3 enum_intenum.py
```

```
Ordered by value:
  fix_released
  fix_committed
  in_progress
  wont_fix
  invalid
  incomplete
  new
```

Valores de enumeración únicos

Los miembros de Enum con el mismo valor se tratan como referencias de alias al mismo objeto miembro. Los alias no hacen que los valores repetidos estén presentes en el iterador para el Enum.

```
# enum_alias.py
```

```
import enum
class BugStatus(enum.Enum):
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1
    by_design = 4
    closed = 1
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
print('\nSame: by_design is wont_fix: ',
      BugStatus.by_design is BugStatus.wont_fix)
print('Same: closed is fix_released: ',
      BugStatus.closed is BugStatus.fix_released)
```

Porque by_design y closed son alias para otros miembros, no aparecen por separado en el

resultado al iterar sobre el Enum. El nombre canónico para un miembro es el primer nombre unido al valor.

```
$ python3 enum_alias.py
```

```
new          = 7
incomplete   = 6
invalid      = 5
wont_fix     = 4
in_progress  = 3
fix_committed = 2
fix_released = 1
Same: by_design is wont_fix: True
Same: closed is fix_released: True
```

Para requerir que todos los miembros tengan valores únicos, agregue el decorador `@unique` al Enum.

```
# enum_unique_enforce.py
```

```
import enum
@enum.unique
class BugStatus(enum.Enum):
    new = 7
    incomplete = 6
    invalid = 5
    wont_fix = 4
    in_progress = 3
    fix_committed = 2
    fix_released = 1
    # This will trigger an error with unique applied.
    by_design = 4
    closed = 1
```

Los miembros con valores repetidos elevan una excepción `ValueError` cuando la clase Enum está siendo interpretada.

```
$ python3 enum_unique_enforce.py
```

```
Traceback (most recent call last):
  File "enum_unique_enforce.py", line 11, in <module>
    class BugStatus(enum.Enum):
  File ".../lib/python3.6/enum.py", line 834, in unique
    (enumeration, alias_details))
ValueError: duplicate values found in <enum 'BugStatus'>:
by_design -> wont_fix, closed -> fix_released
```

Creando enumeraciones programáticamente

En algunos casos, es más conveniente crear enumeraciones programáticamente, en lugar de codificarlos en una definición de clase. Para esas situaciones, Enum también admite pasar los nombres y valores de los miembros al constructor de la clase.

```
# enum_programmatic_create.py
```



```
import enum
BugStatus = enum.Enum(
    value='BugStatus',
    names=('fix_released fix_committed in_progress '
          'wont_fix invalid incomplete new'),
)
print('Member: {}'.format(BugStatus.new))
print('\nAll members:')
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

El argumento value es el nombre de la enumeración, que se usa para compilar la representación de los miembros. El argumento names lista los miembros de la enumeración. Cuando se pasa una sola cadena, es dividida en espacios en blanco y comas, y los tokens resultantes se utilizan como nombres para los miembros, que son valores asignados automáticamente comenzando con 1.

```
$ python3 enum_programmatic_create.py
```

```
Member: BugStatus.new
All members:
fix_released      = 1
fix_committed     = 2
in_progress       = 3
wont_fix          = 4
invalid           = 5
incomplete        = 6
new               = 7
```

Para un mayor control sobre los valores asociados con los miembros, la cadena names se puede reemplazar por una secuencia de tuplas de dos partes o un diccionario mapeando nombres a valores.

```
# enum_programmatic_mapping.py
```

```
import enum
BugStatus = enum.Enum(
    value='BugStatus',
    names=[
        ('new', 7),
        ('incomplete', 6),
        ('invalid', 5),
        ('wont_fix', 4),
        ('in_progress', 3),
        ('fix_committed', 2),
        ('fix_released', 1),
    ],
)
print('All members:')
for status in BugStatus:
    print('{:15} = {}'.format(status.name, status.value))
```

En este ejemplo, se proporciona una lista de tuplas de dos partes en lugar de una sola cadena que contiene solo los nombres de los miembros. Esto hace posible reconstruir la enumeración BugStatus con los miembros en el mismo orden como la versión definida en enum_create.py.

```
$ python3 enum_programmatic_mapping.py
```

```
All members:
new           = 7
incomplete   = 6
invalid      = 5
wont_fix     = 4
in_progress  = 3
fix_committed = 2
fix_released = 1
```

Valores miembros no enteros

Los valores de miembro de Enum no están restringidos a enteros. De hecho, cualquier tipo de objeto se puede asociar con un miembro. Si el valor es una tupla, los miembros se pasan como argumentos individuales a `__init__()`.

```
# enum_tuple_values.py
```

```
import enum

class BugStatus(enum.Enum):

    new = (7, ['incomplete',
              'invalid',
              'wont_fix',
              'in_progress'])
    incomplete = (6, ['new', 'wont_fix'])
    invalid = (5, ['new'])
    wont_fix = (4, ['new'])
    in_progress = (3, ['new', 'fix_committed'])
    fix_committed = (2, ['in_progress', 'fix_released'])
    fix_released = (1, ['new'])

    def __init__(self, num, transitions):
        self.num = num
        self.transitions = transitions

    def can_transition(self, new_state):
        return new_state.name in self.transitions

print('Name:', BugStatus.in_progress)
print('Value:', BugStatus.in_progress.value)
print('Custom attribute:', BugStatus.in_progress.transitions)
print('Using attribute:',
      BugStatus.in_progress.can_transition(BugStatus.new))
```

En este ejemplo, cada valor de miembro es una tupla que contiene el ID numérico (tal como podría estar almacenado en una base de datos) y una lista de transiciones a partir del estado actual.

```
$ python3 enum_tuple_values.py
```

```
Name: BugStatus.in_progress
Value: (3, ['new', 'fix_committed'])
Custom attribute: ['new', 'fix_committed']
Using attribute: True
```

Para casos más complejos, las tuplas pueden volverse difíciles de manejar. Como cualquier tipo de objeto puede ser valor miembro, los diccionarios se pueden usar para los casos donde hay muchos atributos separados para seguir por cada valor enum. Los valores complejos se pasan directamente a `__init__()` como único argumento que no sea `self`.



```
# enum_complex_values.py
```

```
import enum
class BugStatus(enum.Enum):
    new = {
        'num': 7,
        'transitions': [
            'incomplete',
            'invalid',
            'wont_fix',
            'in_progress',
        ],
    }
    incomplete = {
        'num': 6,
        'transitions': ['new', 'wont_fix'],
    }
    invalid = {
        'num': 5,
        'transitions': ['new'],
    }
    wont_fix = {
        'num': 4,
        'transitions': ['new'],
    }
    in_progress = {
        'num': 3,
        'transitions': ['new', 'fix_committed'],
    }
    fix_committed = {
        'num': 2,
        'transitions': ['in_progress', 'fix_released'],
    }
    fix_released = {
        'num': 1,
        'transitions': ['new'],
    }

    def __init__(self, vals):
        self.num = vals['num']
        self.transitions = vals['transitions']

    def can_transition(self, new_state):
        return new_state.name in self.transitions

print('Name:', BugStatus.in_progress)
print('Value:', BugStatus.in_progress.value)
print('Custom attribute:', BugStatus.in_progress.transitions)
print('Using attribute:',
      BugStatus.in_progress.can_transition(BugStatus.new))
```

Este ejemplo expresa los mismos datos que el ejemplo anterior, utilizando diccionarios en lugar de tuplas.

```
$ python3 enum_complex_values.py
```

```
Name: BugStatus.in_progress
Value: {'num': 3, 'transitions': ['new', 'fix_committed']}
Custom attribute: ['new', 'fix_committed']
Using attribute: True
```



Referencias

- [1] Documentación de la biblioteca estándar para enum
- [2] PEP 435 – Agregar un tipo Enum a la biblioteca estándar de Python
- [3] fluf.enum – La inspiración original para enum, por Barry Warsaw.



Ernesto Rico Smith
Usuario GNU/Linux desde 1994
e.rico.schmidt@gmail.com

BOLIVIA

AtixLibre

Hacia un Futuro Innovador

Etico

Libre

Justo