

Guía del desarrollador de Plasmoides

CREANDO PLASMOIDES

Cosmi Fotolia

Vemos cómo crear plasmoides propios para el último escritorio de KDE, disfrutando así de la capacidad de desarrollar el entorno de escritorio activo perfecto. **POR JOHAN THELIN**

Cuando el escritorio KDE hizo su transición de la versión 3 a la 4, fueron muchas las cosas que cambiaron. Uno de los cambios más importantes fue la introducción del escritorio Plasma. En lugar de tener imágenes de fondo con iconos, el escritorio alberga ahora plasmoides, también conocidos como widgets de escritorio.

Los plasmoides pueden hacer cualquier cosa, desde mostrar iconos o una presentación de fotogramas con nuestras fotos favoritas, hasta monitorizar semillas RSS o actualizar la de Twitter. En este artículo veremos cómo desarrollar plasmoides y también analizaremos la arquitectura que se esconde tras Plasma.

Arquitectura

El escritorio Plasma está formado por varios componentes. De ellos, hay dos que son de especial interés: la visualización y las fuentes de datos. Lo importante aquí no son

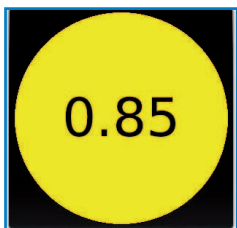


Figura 1: El applet Load Light para Plasma en acción.

estas dos partes en sí, sino el hecho de que se haga esta distinción en dos partes. El principal motivo es que la visualización del plasmoides suele estar separada de los datos. De este modo se obtiene una buena arquitectura modelovista, pero además proporciona la posibilidad de visualizar datos comunes de varios modos sin tener que reimplementar el código que concierne a la parte de la fuente de datos.

En esta separación, los motores de datos han sido equipados con varias cosas interesantes, como el soporte para colas, el control del período mínimo de encolamiento, la capacidad para soportar eventos asíncronos, un mecanismo para proporcionar múltiples valores, etcétera. Para utilizar cualquiera de estas características basta con heredar la clase base *Plasma::DataEngine*.

Del mismo modo, heredando de *Plasma::Applet* se tiene un punto de partida para la creación de applets. Estos applets, al combinarse con un archivo de escritorio, gráficos, etcétera, dan lugar a los plasmoides. Empezaremos con la creación de un plasmoides personal.

Un Plasmoides

El plan consiste en crear un applet simple con el que monitorizar la carga del

sistema. Mostrará la carga media del último minuto en una luz de semáforo, como se expone en la Figura 1. La luz indica los niveles de carga bajo, alto y demasiado alto con los colores verde, amarillo y rojo, respectivamente. Por suerte, ya contamos con un motor de datos adecuado: el monitor del sistema. Antes de pasar a los detalles, veremos cómo elaborar algo de código de base.

Para comenzar, veamos el archivo de cabecera mostrado en el Listado 1. En este archivo, la clase *PlasmaLoadLight* deriva de la clase *Plasma::Applet*. El constructor, *paintInterface*, así como *init*, proporcionan implementaciones de funciones virtuales. La idea detrás de *init* es llevarle desde el constructor las partes complejas de la inicialización. La llamada al constructor *paintInterface* se realiza cada vez que hace falta actualizar la visualización.

Luego siguen los slots *sourceAdded* y *dataUpdated*. Un slot es una función de callback conectada a una señal. Veremos más sobre esto más adelante. La última parte de la clase contiene el privado *m_load*, que a su vez contiene el nivel de carga actual.

El código interesante viene en la implementación de la clase mostrada en el Listado 2. Comenzando desde arriba,

las líneas 4-11 muestran la implementación del constructor. Aquí simplemente le decimos a Plasma que use el fondo estándar y configure un tamaño inicial razonable. La otra parte de la inicialización de la clase tiene lugar en la función *init* (líneas 13-17). Aquí se hace una petición al motor de datos *systemmonitor* y se conecta la señal *sourceAdded* a un slot con el mismo nombre en el applet. Esto implica que con cada fuente de información que añade el motor de datos del monitor del sistema, se llamará al slot *sourceAdded* del applet. Así llegamos hasta la siguiente función de la implementación: *sourceAdded* (líneas 19-27). El motor de datos emite una señal que dispara esta función cada vez que hay una fuente disponible. El motor de datos del monitor del sistema simplemente lista sus fuentes, mientras que otros motores podrían tener fuentes que aparecen y desaparecen durante el transcurso de su existencia. Algunos ejemplos incluyen motores de datos con soporte para hardware

extraíble o servicios de red que aparecen y desaparecen. Por tanto, la señal *sourceRemoved* está disponible desde la clase del motor de datos, aunque en este caso la ignoraremos.

Al añadir la fuente *cpu/system/loadavg1*, le estamos pidiendo al motor de datos que nos conecte al método *connectSource*. Los argumentos para esta función son, de izquierda a derecha: nombre de la fuente, objeto receptor y período de muestreo en milisegundos. Por lo que, en la línea 23, le estamos diciendo que se actualice con *cpu/system/loadavg1* a cada segundo.

El método *connectSource* sólo acepta un puntero al objeto receptor, y no un slot con el que conectar. Esto se debe a que da por hecho que el receptor tendrá un slot con la siguiente firma:

```
dataUpdated(
const QString &sourceName,
const
Plasma::DataEngine::Data &data)
```

dataUpdated. Cabe destacar el período de muestreo dado al conectar a una fuente (el motor de datos puede forzar el período y aumentar su tamaño). Además, así se controla el período entre peticiones, y no sólo el período transcurrido entre las llegadas de los nuevos datos.

Por ejemplo, imaginemos un motor de datos que monitoriza semillas de RSS. Puede que no tenga sentido hacer peticiones de datos a intervalos menores de diez segundos en este tipo de motores de datos. Además, cuando se hacen peticiones a través de la red, se trata de respuestas asíncronas, con lo que el tiempo de respuesta de los resultados es arbitrario. Por tanto, el slot *dataUpdated* se debería disparar a intervalos irregulares conforme van llegando los datos.

Lo siguiente que vemos en el código fuente es la implementación del método *dataUpdated*, en las líneas 29 a la 38. Dado que se llama a este slot en todas las fuentes de datos, primero nos aseguraremos de que realmente se trata de la fuente *cpu/system/loadavg1*. Los datos se pasan entonces en forma de tabla de hashes con pares clave-valor. Una peculiaridad o rareza del motor de datos del

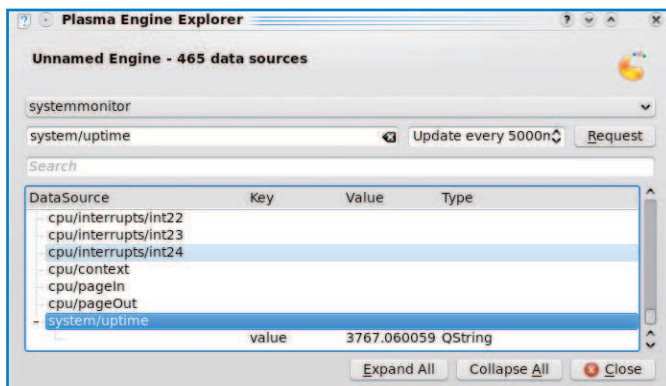


Figura 2: La herramienta Engine Explorer de Plasma mostrando el motor de datos *systemmonitor*.

Cada vez que la fuente solicitada tiene disponibles nuevos datos, se dispara el slot

Listado 1: loadlight.h

```
01 #ifndef LOADLIGHT_H
02 #define LOADLIGHT_H
03
04 #include <Plasma/Applet>
05 #include <Plasma/DataEngine>
06
07 class QSizeF;
08
09 class PlasmaLoadLight : public Plasma::Applet
10 {
11     Q_OBJECT
12
13 public:
14     PlasmaLoadLight( QObject *parent,
15                     const QVariantList &args );
16     void init();
17     void paintInterface( QPainter *painter,
18                         const
19                         QStyleOptionGraphicsItem *option,
20                         const QRect
21                         &contentsRect );
22 protected slots:
23     void sourceAdded( const QString &name );
24     void dataUpdated( const QString &sourceName,
25                       const
26                       Plasma::DataEngine::Data &data );
27 private:
28     double m_load;
29 };
30 #endif // LOADLIGHT_H
```

PlasMate

En este artículo sólo hemos visto el desarrollo de plasmoides con C++. Sin embargo, se pueden desarrollar plasmoides fácilmente utilizando otros lenguajes, incluidos *JavaScript*, *Ruby* y *Python*. Con el fin de que todo el mundo comience con el desarrollo de nuevos plasmoides, la gente de KDE ha empezado a trabajar en un editor llamado *PlasMate*. En la Figura 3 se muestra la ventana de edición principal de *PlasMate* con los componentes en su parte superior, el archivo actual en el centro y una previsualización en la parte inferior. La herramienta no es completamente utilizable aún, pero el proyecto está descrito en la KDE TechBase, y el código actual se puede obtener a través de Subversion desde KDE playground:

```
svn co svn://anonsvn.kde.org/
home/kde/trunk/playground/base/
/plasma/plasmate
```

Una vez obtenido el código, se puede compilar con CMake. Sin embargo, el código fuente requiere KDE 4.3, por lo que probablemente tenga que actualizarse también la versión de KDE de la distro en la que se vaya a instalar.

monitor del sistema es que tiende a devolver un par de resultados sin datos válidos antes de que comience a servir los datos reales útiles. No se trata de un problema demasiado serio, pero puede evitarse comprobando que la clave realmente existe en los datos. En este caso en particular sólo se obtiene un valor, por lo que si en él hay una clave, cogemos el primer valor y actualizamos la variable miembro *m_load*.

Una vez actualizada *m_load*, llamamos a *update* para disparar el evento de redibujado. Este evento llamará a su vez al método *paintInterface*, donde se usarán los datos para dibujar un círculo con los datos en forma de texto dentro de él.

El último paso en la implementación pasa por exportar la clase como applet de Plasma en la línea 60. La macro *K_EXPORT_PLASMA_APPLET* recibe dos argumentos: un nombre para la librería seguido del nombre de la clase a

exponer. Después de eso, el archivo *.moc* del compilador de metaobjetos se incluye en el archivo, ya que en este caso sólo hay una clase.

La exportación de la clase como applet es la primera parte en la elaboración de un plasmoide. Para que el escritorio Plasma sea capaz de encontrar nuestro applet, hemos de crear un archivo de escritorio para él. El archivo de escritorio es una lista de pares clave-valor que se puede descargar junto con el código fuente de este proyecto [1].

Las claves usadas son:

- **Name:** El nombre del plasmoide tal y como se mostrará al usuario. En este caso, *Load Light*.
- **Comment:** Un comentario descriptivo del plasmoide.
- **Type:** Ha de ser *Service*.
- **ServiceTypes:** Debe ser *Plasma/Applet*.
- **X-KDE-Library:** El nombre de la librería.

- **X-KDE-PluginInfo-Name:** El nombre de librería del plugin.
- **X-KDE-PluginInfo-License:** La licencia utilizada por la librería, por ejemplo GPL.
- **KDE-PluginInfo-EnabledByDefault:** Ponemos *true*.
- **X-KDE-PluginInfo-Author:** Nombre de contacto del autor.
- **X-KDE-PluginInfo-Email:** Email de contacto del autor.

Ahora todo lo que resta es compilar e instalar el plasmoide en el entorno KDE. En el proyecto KDE, para compilar se usa la herramienta *CMake*. Para controlar *CMake*, se utiliza un archivo *CMake-Lists.txt*. Este archivo es más o menos un trozo de código base con el nombre de la librería adecuado en él y se puede descargar con el código fuente de este proyecto.

Para compilar el proyecto hay que conocer el prefijo de la instalación de KDE. Para averiguarlo se puede usar la

Listado 2: loadlight.cpp

```

01 #include "loadlight.h"
02 #include <QPainter>
03
04 PlasmaLoadLight::PlasmaLoadLight( QObject
05     *parent,
06     const
07     QVariantList &args )
08     : Plasma::Applet( parent, args )
09     , m_load( 0.75 )
10 {
11     setBackgroundHints( DefaultBackground );
12     resize( 100, 100 );
13 }
14 void PlasmaLoadLight::init()
15 {
16     connect( dataEngine( "systemmonitor" ),
17             SIGNAL(sourceAdded(QString)),
18             this, SLOT(sourceAdded(QString)) );
19 }
20 void PlasmaLoadLight::sourceAdded( const QString
21     &name )
22 {
23     if( name == "cpu/system/loadavg1" )
24     {
25         dataEngine( "systemmonitor" )->connectSource(
26             name, this, 1000 );
27         disconnect( dataEngine( "systemmonitor" ),
28             SIGNAL(sourceAdded(QString)),
29             this, SLOT(sourceAdded(QString))
30         );
31     }
32 }
33 void PlasmaLoadLight::dataUpdated( const QString
34     &sourceName,
35     const
36     Plasma::DataEngine::Data &data )
37 {
38     if( sourceName != "cpu/system/loadavg1" )
39         return;
40     if( data.keys().count() == 0 )
41         return;
42     m_load = data[data.keys()[0]].toDouble();
43     update();
44 }
45 void PlasmaLoadLight::paintInterface( QPainter
46     *painter,
47     const
48     QStyleOptionGraphicsItem *option,
49     const QRect
50     &contentsRect )
51 {
52     if( m_load < 0.5 )
53         painter->setBrush( Qt::green );
54     else if( m_load > 0.95 )
55         painter->setBrush( Qt::red );
56     else
57         painter->setBrush( Qt::yellow );
58     painter->drawEllipse( contentsRect );
59
60     QFont f;
61     f.setPixelSize( contentsRect.height()/4 );
62     painter->setFont( f );
63     painter->drawText( contentsRect,
64         Qt::AlignCenter,
65         QString::number( m_load,
66             'f', 2 ) );
67 }
68
69 K_EXPORT_PLASMA_APPLET(loadlight,
70     PlasmaLoadLight)
71 #include "loadlight.moc"

```

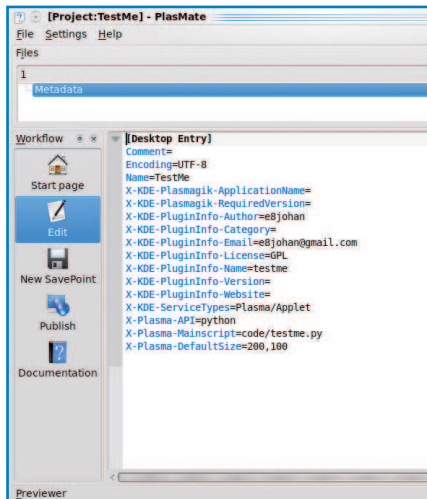


Figura 3: Ventana principal de edición de PlasMate.

aplicación `kde4-config`. En nuestro caso el prefijo es `/usr`.

```
$ kde4-config --prefix ↵
/usr
```

Entonces entramos en el directorio de las fuentes y ejecutamos `cmake` y `make` para compilar el proyecto. Añadimos el

prefijo como valor de `CMAKE_INSTALL_PREFIX`:

```
cmake ↵
-D CMAKE_INSTALL_PREFIX=/usr
make
```

Para instalar el plasmoid, ejecutamos `make install` con el nivel de privilegios necesario y ejecutamos la utilidad `kbuildsysco4`:

```
sudo make install
kbuildsysco4
```

Luego, basta con reiniciar la sesión Plasma o usar la utilidad `plasmoidviewer` para probar el nuevo plasmoid. Para encontrar todos los plasmoides conocidos, así como los comentarios asociados al archivo de escritorio, utilizaremos el parámetro `--list`. O, simplemente, le pasamos el nombre del plasmoid para que lo muestre:

```
$ plasmoidviewer ↵
--list | grep loadlight
plasma-applet-loadlight - ↵
```

```
Plasmoid Load Light, ↵
monitoriza la ↵
carga media del sistema

$ plasmoidviewer
plasma-applet-loadlight
```

Esto es cuanto se necesita para añadir un applet al escritorio Plasma. Como se puede apreciar, se trata de un proceso bastante lineal en el que se recopilan los datos, se implementa la interfaz del applet y se pone el código a disposición de Plasma.

De cualquier manera, para añadir realmente funcionalidad al escritorio se han de proveer motores de datos propios.

Motores de Datos

A la hora de desarrollar motores de datos, nuestro mayor aliado es la utilidad `plasmaengineexplorer`. Esta herramienta nos permite movernos por los motores de datos disponibles y hacerles peticiones, disponiendo así de un mecanismo con el que explorar las fuentes de datos existentes, así como probar nuestras propias implementaciones.

Barcelona
6-9 September, 2010



FOSS4G es "la conferencia" anual organizada por OSGeo, sobre Software Geoespacial de Código Abierto. Se trata de una "reunión de tribus" de las diferentes comunidades de Código Abierto Geoespacial, donde desarrolladores y usuarios muestran las novedades y proyectos punteros.

La edición de 2010 tendrá lugar en Barcelona, España.

<http://2010.foss4g.org>



FOSS4G 2010 Barcelona

Antes de comenzar con un motor de datos propio, exploremos los motores de datos disponibles. Por ejemplo, en la Figura 2 se muestra el tiempo total de ejecución obtenido a partir del motor de datos de monitorización del sistema.

El de creación de un motor de datos propio es un proceso muy similar al de creación de un nuevo plasmoid. Por ejemplo, los archivos *CMakeLists.txt* y de escritorio permanecen. La única diferencia es que el archivo de escritorio declara el tipo de servicio como *Plasma/DataEngine*.

En lo que respecta a la clase C++, la clase base usada es *Plasma::DataEngine*, en vez de *Plasma::Applet*; por tanto, tenemos otra interfaz que implementar. Para ilustrarlo, implementaremos un motor de datos que sirve números aleatorios y es consultado por la utilidad *Engine Explorer*. Veamos la declaración de clase que se presenta en el Listado 3. En ella se muestra una de las implementaciones de interfaces de motores de datos más sencillas posibles. Continuando con la implementación de la clase, Listado 4, podemos ver que la implementación es igual de simple.

Comienza con un constructor muy sencillo (líneas 4 a 7). Se suele llamar a *setMinimumPollingInterval* cuando se desea evitar un encolamiento demasiado frecuente. En este caso, sin embargo, podremos servir una cantidad casi ilimitada de números aleatorios por segundo, por lo que no vamos a establecer ningún límite.

El método *sources* que sigue en las líneas que van de la 9 a la 12 es también bastante trivial. Aquí, simplemente se plantea que la fuente “Número” está soportada. En las dos funciones que siguen, *sourceRequestEvent* y *updateSourceEvent*, es donde se lleva a cabo el verdadero trabajo.

Los eventos de petición se dan la primera vez que se consulta una fuente. Aún se espera que se establezca un valor, por lo que la implementación inicializa primero el generador de números aleatorios mediante *qsrand()* antes de llamar al método del evento de actualización.

Los eventos de actualización se producen cada vez que se van a actualizar los datos. Nótese que el método *updateSourceEvent* no devuelve ningún valor real; en lugar de eso, utiliza el método *setData* para anunciar que hay nuevos datos disponibles. Mediante este procedimiento podemos implementar motores de datos asíncronos fácilmente utilizando exactamente los mismos mecanismos.

Basta con establecer la petición de datos desde el evento de actualización para llamar entonces a *setData* desde el código en el que se recibirán los datos.

Para compilar e instalar el motor de datos se sigue exactamente el mismo procedimiento que para los applets. Para probarlo se puede utilizar *Engine Explorer* y pedir un par de números.

Crear un Escritorio

Como se puede apreciar, la elaboración de plasmoides y motores de datos no es demasiado compleja. El objetivo es implementar una API determinada. Para saber más sobre el arte de crear plasmoides recomendamos visitar el sitio de KDE TechBase [2]. Allí se pueden encontrar tutoriales, así como documentación detallada de referencia de la API. ■

RECURSOS

- [1] Listados de este artículo: <https://www.linux-magazine.es/Magazine/Downloads/62/KDE4Plasmoids>
 [2] KDE TechBase: <http://techbase.kde.org>

Listado 3: randomnumberengine.h

```
01 #ifndef RANDOMNUMBERENGINE_H
02 #define RANDOMNUMBERENGINE_H
03
04 #include <Plasma/DataEngine>
05
06 class RandomNumberEngine : public
    Plasma::DataEngine
07 {
08     Q_OBJECT
09
10 public:
11     RandomNumberEngine( QObject *parent, const
        QVariantList &args );
12     QStringList sources() const;
13
14 protected:
15     bool sourceRequestEvent( const QString &name
        );
16     bool updateSourceEvent( const QString &name
        );
17 };
18
19 #endif // RANDOMNUMBERENGINE_H
```

Listado 4: randomnumberengine.cpp

```
01 #include "randomnumberengine.h"
02 #include <QDateTime>
03
04 RandomNumberEngine::RandomNumberEngine( QObject
    *parent, const QVariantList &args )
05     : Plasma::DataEngine( parent, args )
06 {
07 }
08
09 QStringList RandomNumberEngine::sources() const
10 {
11     return QStringList() << "Number";
12 }
13
14 bool RandomNumberEngine::sourceRequestEvent(
    const QString &name )
15 {
16     if( name != "Number" )
17         return false;
18
19     qsrand(
        QDateTime::currentDateTime().toTime_t() );
20     return updateSourceEvent( name );
21 }
22
23 bool RandomNumberEngine::updateSourceEvent( const
    QString &name )
24 {
25     if( name != "Number" )
26         return false;
27
28     setData( name, qrand() );
29     return true;
30 }
31
32 K_EXPORT_PLASMA_DATAENGINE(randomnumber,
    RandomNumberEngine)
33
34 #include "randomnumberengine.moc"
```