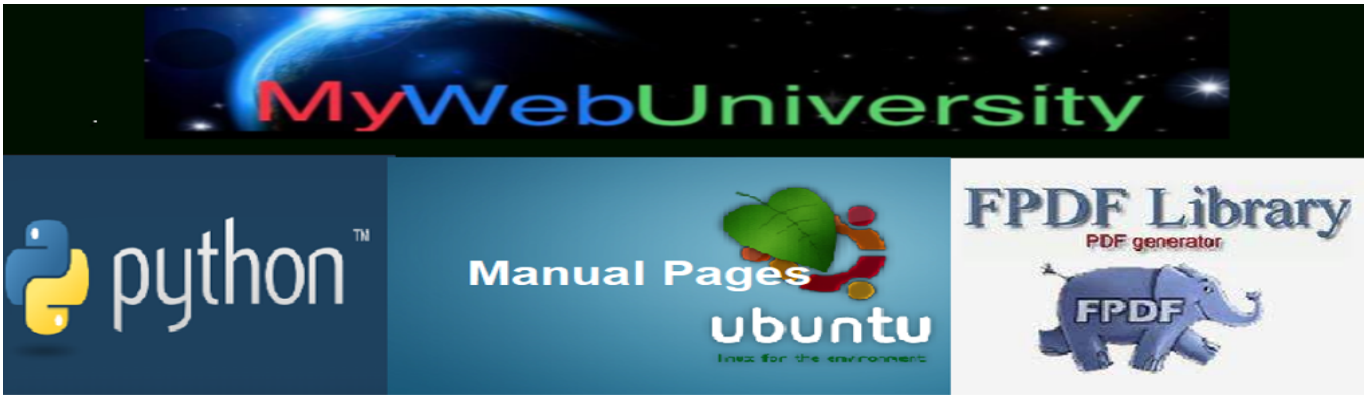

Rocky Enterprise Linux 9.2 Manual Pages on command 'Net::DBus::Tutorial::ExportingObjects.3pm'

$ man Net::DBus::Tutorial::ExportingObjects.3pm

Net::DBus::Tutorial::ExportingOUsertContributed Perl DoNet::DBus::Tutorial::ExportingObjects(3pm)

NAME

Net::DBus::Tutorial::ExportingObjects - tutorials on providing a DBus service

DESCRIPTION

This document provides a tutorial on providing a DBus service using the Perl Net::DBus application bindings. This examples in this document will be based on the code from the Music::Player distribution, which is a simple DBus service providing a music track player.

CREATING AN OBJECT

The first step in creating an object is to create a new package which inherits from Net::DBus::Object. The Music::Player::Manager object provides an API for managing the collection of music player backends for different track types. To start with, lets create the skeleton of the package & its constructor. The constructor of the super type, Net::DBus::Object expects to be given to parameters, a handle to the Net::DBus::Service owning the object, and a path under which the object shall be exported. Since the manager class is intended to be a singleton object, we can hard code the path to it within the constructor:

```
package Music::Player::Manager;
use base qw(Net::DBus::Object);
sub new {
    my $class = shift;
    my $service = shift;
    my $self = $class->SUPER::new($service, "/music/player/manager");
    bless $self, $class;
```

```
    return $self;
}
1;
```

Now, as mentioned, the manager with handle a number of different player backends. So we need to provide methods for registering new backends, and querying for backends capable of playing a particular file type. So modifying the above code we add a hash table in the constructor, to store the backends:

```
sub new {
    my $class = shift;
    my $service = shift;
    my $self = $class->SUPER::new($service, "/music/player/manager");
    $self->{backends} = {};
    bless $self, $class;
    return $self;
}
```

And now a method to register a new backend. This takes a Perl module name and uses it to instantiate a backend. Since the backends are also going to be DBus objects, we need to pass in a reference to the service we are attached to, along with a path under which to register the backend. We use the "get_service" method to retreieve a reference to the service the manager is attached to, and attach the player backend to this same service: When a method on DBus object is invoked, the first parameter is the object reference ($self), and the remainder are the parameters provided to the method call. Thus writing a method implementation on a DBUs is really no different to normal object oriented Perl (cf perltoot):

```
sub register_backend {
    my $self = shift;
    my $name = shift;
    my $module = shift;
    eval "use $module";
    if ($@) {
        die "cannot load backend $module: $@" ;
    }
    $self->{backends}->{$name} = $module->new($self->get_service,
```

```
                                "/music/player/backend/$name");
}
```

Looking at this one might wonder what happens if the "die" method is triggered. In such a scenario, rather than terminating the service process, the error will be caught and propagated back to the remote caller to deal with.

The player backends provide a method "get_track_types" which returns an array reference of the music track types they support. We can use this method to provide an API to allow easy retrieval of a backend for a particular track type. This method will return a path with which the backend object can be accessed

```
sub find_backend {
    my $self = shift;
    my $extension = shift;
    foreach my $name (keys %{$self->{backends}}) {
      my $backend = $self->{backends}->{$name};
      foreach my $type (@{$backend->get_track_types}) {
        if ($type eq $extension) {
            return $backend->get_object_path;
        }
      }
    }
    die "no backend for type $extension";
}
```

Lets take a quick moment to consider how this method would be used to play a music track. If you've not already done so, refresh your memory from Net::DBus::Tutorial::UsingObjects. Now, we have an MP3 file which we wish to play, so we search for the path to a backend, then retrieve the object for it, and play the track:

```
...get the music player service...
# Ask for a path to a player for mp3 files
my $path = $service->find_backend("mp3");
# $path now contains '/music/player/backend/mpg123'
# and we can get the backend object
my $backend = $service->get_object($path);
# and finally play the track
```

```
$backend->play("/vol/music/beck/guero/09-scarecrow.mp3");
```

PROVIDING INTROSPECTION DATA

The code above is a complete working object, ready to be registered with a service, and since the parameters and return values for the two methods are both simple strings we could stop there. In some cases, however, one might want to be more specific about data types expected for parameters, for example signed vs unsigned integers. Adding explicit data typing also makes interaction with other programming languages more reliable.

Providing explicit data type definitions for exported method is known in the DBus world as "Introspection", and it makes life much more reliable for users of one's service whom may be using a strongly typed language such as C.

The first step in providing introspection data for a DBus object in Perl, is to specify the name of the interface provided by the object. This is typically a period separated string, by convention containing the domain name of the application as its first component. Since most Perl modules end up living on CPAN, one might use "org.cpan" as the first component, followed by the package name of the module (replacing :: with .), eg "org.cpan.music.player.manager". If it is not planned to host the module on CPAN, a personal/project domain might be used eg "com.berrange.music.player.manager". The interface for an object is defined by loading the Net::DBus::Exporter module, providing the interface as its first parameter. So the earlier code example would be modified to look like:

```
package Music::Player::Manager;
use base qw(Net::DBus);
use Net::DBus::Exporter qw(com.berrange.music.player.manager)
```

Next up, it is necessary to provide data types for the parameters and return values of the methods. The Net::DBus::Exporter module provides a method "dbus_method" for this purpose, which takes three parameter, the name of the method being exported, an array reference of parameter types, and an array reference of return types (the latter can be omitted if there are no return values). This can be called at any point in the module's code, but by convention it is preferable to associate calls to "dbus_method" with the actual method implementation, thus:

```
dbus_method("register_backend", ["string", "string"]);
sub register_backend {
    my $self = shift;
```

```
    my $name = shift;
    my $module = shift;
    .. snipped rest of method body ...
}
```

And, thus:

```
dbus_method("find_backend", ["string"], ["string"])
sub find_backend {
    my $self = shift;
    my $extension = shift;
    ... snip method body...
}
```

DEFINING A SERVICE

Now that the objects have been written, it is time to define a service. A service is nothing more than a well known name for a given API contract. A contract can be thought of as a definition of a list of object paths, and the corresponding interfaces they provide.

So, someone else could come along a provide an alternate music player implementation using the Python or QT bindings for DBus, and if they provided the same set of object paths & interfaces, they could justifiably register the same service on the bus.

The Net::DBus::Service module provides the means to register a service. Its constructor expects a reference to the bus object (an instance of Net::DBus), along with the name of the service. As with interface names, the first component of a service name is usually derived from a domain name, and then suffixed with the name of the application, in our example forming "org.cpan.Music.Player". While some objects will be created on the fly during execution of the application, others are created upon initial startup. The music player manager object created earlier in this tutorial is an example of the latter. It is typical to instantiate and register these objects in the constructor for the service. Thus a service object for the music player application would look like:

```
package Music::Player;
use base qw(Net::DBus::Service);
sub new {
    my $class = shift;
    my $bus = shift;
    my $self = $class->SUPER::new($bus, "org.cpan.music.player");
```

```
    bless $self, $class;
    $self->{manager} = Music::Player::Manager->new($self);
    return $self;
}
```

The Net::DBus::Service automatically provides one special object to all services, under the path "/org/freedesktop/DBus/Exporter". This object implements the "org.freedesktop.DBus.Exporter" interface which has a method "ListObject". This enables clients to determine a list of all objects exported within a service. While not functionally necessary for most applications, it is none-the-less a useful tool for developers debugging applications, or wondering what a service provides.

CONNECTING TO THE BUS

The final step in getting our service up and running is to connect it to the bus. This brings up an interesting conundrum, does one export the service on the system bus (shared by all users & processes on the machine), or the session bus (one per user logged into a machine). In some cases the answer, with only one of the two buses conceptually making sense. In other cases, however, both the session & system bus are valid. In the former one would use the "session" or <system> methods on Net::DBus to get a handle to the desired bus, while in the latter case, the "find" method would be used. This applies a heuristic to determine the correct bus based on execution environment. In the case of the music player, either bus is relevant, so the code to connect the service to the bus would look like:

```
use Net::DBus;
my $bus = Net::DBus->find;
my $player = Music::Player->new($bus);
```

With the service attached to the bus, it is merely necessary to run the main event processing loop to listen out for & handle incoming DBus messages. So the above code is modified to start a simple reactor:

```
use Net::DBus;
use Net::DBus::Reactor;
my $bus = Net::DBus->find;
my $player = Music::Player->new($bus);
Net::DBus::Reactor->main->run;
exit 0;
```

Saving this code into a script "/usr/bin/music-player.pl", coding is complete and the service ready for use by clients on the bus.

SERVICE ACTIVATION

One might now wonder how best to start the service, particularly if it is a service capable of running on both the system and session buses. DBus has the answer in the concept of "activation". What happens is that when a client on the bus attempts to call a method, or register a signal handler against, a service not currently running, it will first try and start the service. Service's which wish to participate in this process merely need stick a simple service definition file into the directory "/usr/share/dbus-1/services". The file should be named to match the service name, with the file extension ".service" appended. eg, "/usr/share/dbus-1/services/org.cpan.music.player.service" The file contains two keys, first the name of the service, and second the name of the executable used to run the service, or in this case the Perl script. So, for our simple service the data file would contain:

```
[D-BUS Service]
Name=org.cpan.music.player
Exec=/usr/bin/music-player.pl
```

SEE ALSO

Net::DBus::Tutorial for details of other tutorials, and Net::DBus for API documentation

AUTHORS

Daniel Berrange <dan@berrange.com>

COPYRIGHT