XPCOM

Laurent Jouanneau
Course on Mozilla Education and Technologies @ Evry

December, 2011

What is a XPCOM

- A component system
- XPCOM is inspired from Microsoft COM and is a way to:
  - open your XUL application to the world (externals libraries...)
  - create your components into a language other than javascript
  - bring components into an existing application through extensions
    - Adding features to existing internal components (ex: protocol handlers)
    - Replacing existing components
    - Your own technical/business component for your extension/app

A component

- No need to manage instances:
  - Automatic destruction by using reference counters. Even in C++
  - A factory should be provided to declare and instancy the component
  - The component can be used as multiple instances or as a service (automatic singleton management)
- Language agnostic: a component can be implemented with any language, if the language binding is availabled
  - Actual availabled binding: C++, Javascript (and Python)

A component #2

- Should implement one or more interfaces
- Xpconnect: glue between the javascript world and all other XPCom

Interfaces

- An interface describes public properties and methods implemented by a component
- This is a « contract » between the component and other components/client code → compatibilities between components
  - It means that if a component/client code needs a component which have a specific interface, we can give to it any components which implement this interface.
- Several components can implement the same interface and a component can implement several interfaces

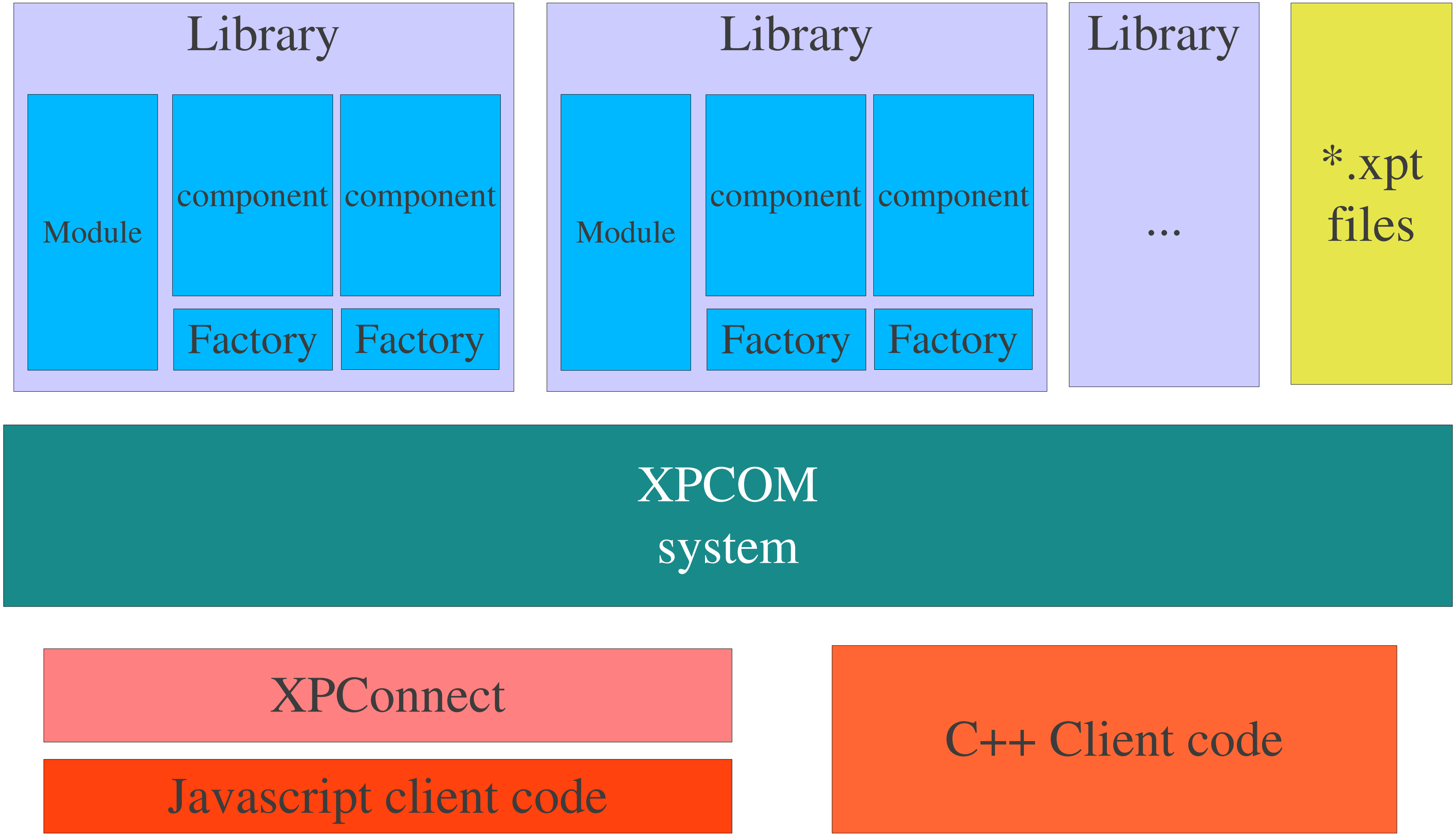
Interfaces

- Naming: **aaIBbb**
  - **aa**: vendor prefix (« ns » for Netscape, « moz » for Mozilla...)
  - **I**: for Interface
  - **bbb**: interface name
- Each interface have an UUID
- Interfaces are described into *.xpt files, which are produced from *.idl files during the development.
- Interfaces can be inherit from other interfaces
- All interfaces have the **nsISupports** interface as ancester.

XPConnect

- This is the glue between a javascript context and the XPCom system.
- When a component is explicitely instanciated, XPconnect maps properties and methods of a component on a javascript object.
- Some « native » JS object are also mapped to some XPCom components (ex: DOM, xmlHttpRequest...)
- Is able to « decorate » a simple JS object as a component when we give this object to a XPCOM component which expect to receive a component with a specific interface.
- Provides the object « Components »

Architecture



Library

Module

component

component

Factory

Factory

Library

Module

component

component

Factory

Factory

Library

...

*.xpt files

XPCOM system

XPConnect

Javascript client code

C++ Client code

Components object

- Its main properties
  - Classes: contain the list of available factories
  - Interfaces: contain the list of available interfaces
- To use a component, we have to:
  - Retrieve the factory corresponding to the component
  - Call createInstance or getService method of the factory to instanciate the component
  - Indicate the interface we want to use on this component
  - We can then use the component by calling methods or properties corresponding on the interface we want to use

Creating an instance of a component

```
// Retrieve the factory of the component by using the contract ID
// of the component
var factory = Components.classes["a_contract_id"];

// call the createInstance method of the factory, and indicate the interface
// we want to use on the component
var myObject = factory.createInstance(Components.interfaces.nsITheInterface);

// later, if we want to use an other interface of the component
myObject = myObject.QueryInterface(Components.interfaces.nsIOtherInterface);

// shorter call
var myObject = Components.classes["a_contract_id"]
                         .createInstance(Components.interfaces.nsITheInterface);
```

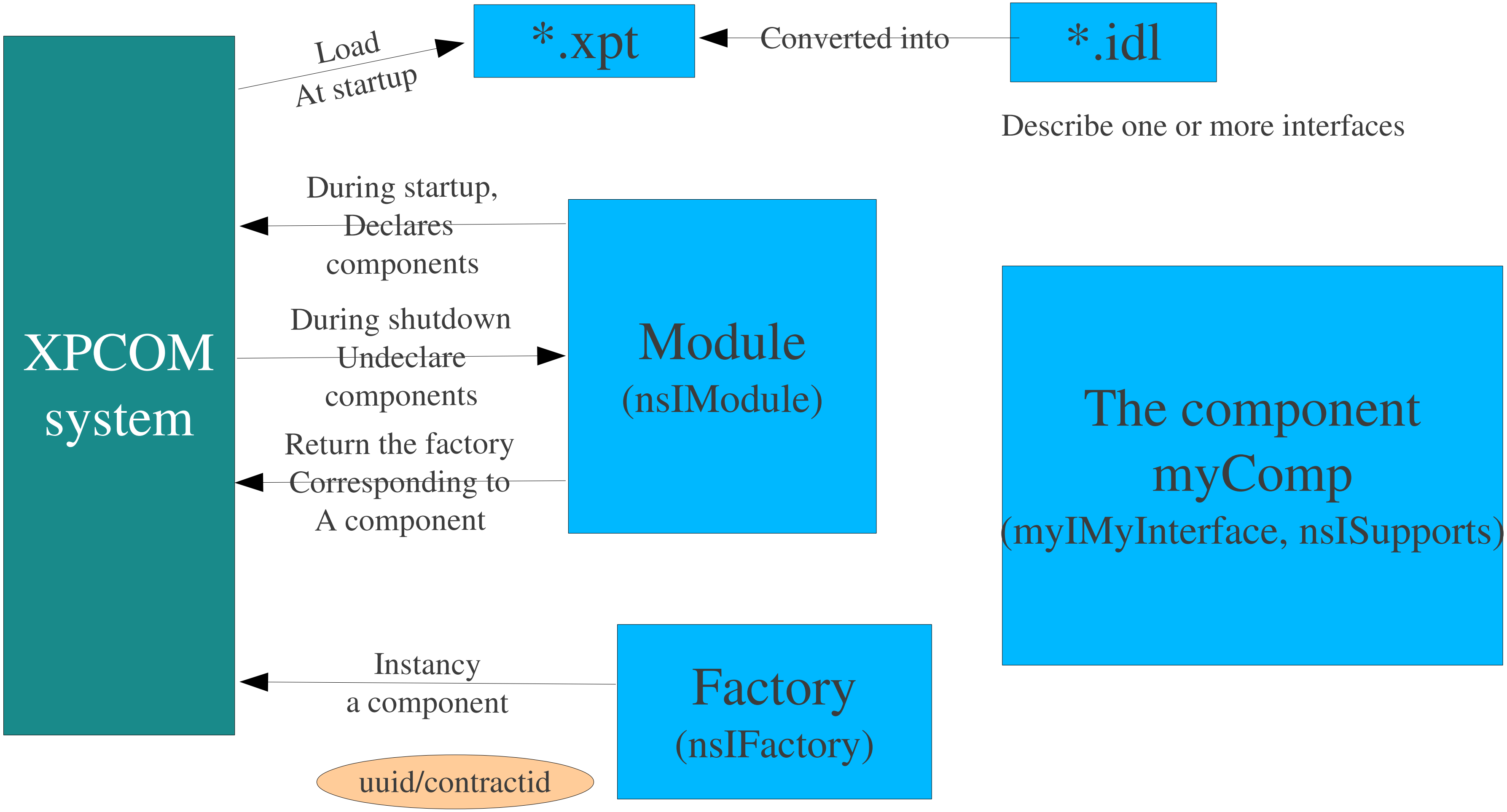
Getting a component as a service

- Use getService instead of createInstance
- If we call getService several time for the same component, we will have the same instance. A factory use a singleton.

```
// Retrieve the factory of the component by using the contract ID of the component
var factory = Components.classes["a_contract_id"];

// call the getService method of the factory, and indicate the interface
// we want to use on the component
var myObject = factory.getService(Components.interfaces.nsITheInterface);
```

Objects to provide for a component



XPCOM system

*.xpt

*.idl

Converted into

Load At startup

Describe one or more interfaces

During startup, Declares components

During shutdown Undeclare components

Return the factory Corresponding to A component

Module (nsIModule)

The component myComp (myIMyInterface, nsISupports)

Instancy a component

Factory (nsIFactory)

uuid/contractid

Creating a component in Javascript

Creating an XPCOM

- Choose a contract id and a uuid
- Choose an existing interface you want to implement, or create a new one
- If you want to create a new interface:
  - Describe the interface in an IDL file
  - Create an XPT file from the IDL file
- Develop your component, and implement all interfaces
- Declare your component in chrome.manifest (gecko > 2.0)
- Develop the factory (nsIFactory) (< gecko 1.8)
- Develop the module (nsIModule) (< gecko 1.8)

UUID et contract id

- contract ID : human readeable ID
  - Ex: « @mydomain.org/modulename/compname;1 »
- uuid/CID/class ID : unique hexadecimal number
  - Ex: c0273bbc-ef55-4557-a276-cda221eeac03
  - To create a UUID: uuidgen, guidgen.exe

Choosing the interface

We will create a component which implements the nsISupportsString interface, already defined in Gecko.

```
[scriptable, uuid(d0d4b136-1dd1-11b2-9371-f0727ef827c0)]
interface nsISupportsPrimitive : nsISupports
{
    const unsigned short TYPE_ID                = 1;
    const unsigned short TYPE_CSTRING           = 2;
    const unsigned short TYPE_STRING            = 3;
    const unsigned short TYPE_PRBOOL            = 4;
(...)
    const unsigned short TYPE_INTERFACE_POINTER = 17;

    readonly attribute unsigned short type;
};

[scriptable, uuid(d79dc970-4a1c-11d3-9890-006008962422)]
interface nsISupportsString : nsISupportsPrimitive
{
    attribute AString data;
    wstring toString();
};
```

Implementing the component in JS

We create a javascript file mystring.js into the components/ directory of the extension or the xulrunner application.

Let's first declare the contract id and the class id in some constants:

```
/**
 * contract ID of the component
 */
const MYSTRING_CONTRACTID = '@xulfr.org/tutorial/mystring;1';


/**
 * UUID of the component
 */
const MYSTRING_CLASSID = Components.ID('{df53456e-0484-4098-9353-ee22661e6819}');
```

JS Component: the main object

Let's implement the object with methods and properties corresponding to the interface we choose.

```
function myStringImpl(){
    this._str = '';  // a « private » membre
}

myStringImpl.prototype = {

    //-------------- implementation of the nsISupportsPrimitive interface
    get type() {
      return Components.interfaces.nsISupportsPrimitive.TYPE_STRING;
    },

    //-------------- implementation of the nsISupportsString interface
    get data() { return this._str; },
    set data(aValue) { return this._str = aValue; },

    toString: function () { return this._str; }


}
```

JS Component: QueryInterface

NsISupports should be implemented in all components. It declares only one method for javascript components, QueryInterface, and two other methods for C++ components, AddRef and Release

```
[scriptable, uuid(00000000-0000-0000-c000-000000000046)]
interface nsISupports {
  void QueryInterface(in nsIIDRef uuid,
                      [iid_is(uuid),retval] out nsQIResult result);
  [noscript, notxpcom] nsrefcnt AddRef();
  [noscript, notxpcom] nsrefcnt Release();
};
```

QueryInterface is a function which should check if the requested interface is one of the interface the component implements, and if yes, should return the object which implements it (most of time, the object itself).

JS Component: QueryInterface (2)

Let's add the QueryInterface method (old way, with gecko < 1.8) :

```
myStringImpl.prototype = {

    (...)


    //-------------- implementation of the nsISupports interface

    QueryInterface: function(iid) {
        if (!iid.equals(Components.interfaces.nsISupportsPrimitive) &&
            !iid.equals(Components.interfaces.nsISupportsString) &&
            !iid.equals(Components.interfaces.nsISupports))
            throw Components.results.NS_ERROR_NO_INTERFACE;
        return this;
    }
}
```

XPCOMUtils: simpler JS components

- Since gecko 1.8.0, a javascript module (called also a JSM, do not confuse with xpcom module objects) is provided to facilitate the development of javascript XPCOM components : XPCOMUtils
- It provides helpers to create some object like factories and modules, or to implement some function like QueryInterface
- We use it by writing this instruction at the beginning of the javascript file:

```
Components.utils.import("resource://gre/modules/XPCOMUtils.jsm");
```

XPCOMUtils: the component

Properties to indicate the class id, the contract id and a description:

```
myStringImpl.prototype = {

  classDescription: "myString2 JS component",
  classID:          MYSTRING_CLASS_ID,
  contractID:       MYSTRING_CONTRACTID,
...
```

To implement, QueryInterface, call generateQI method by given the list of interfaces you implement:

```
QueryInterface: XPCOMUtils.generateQI ([
      Components.interfaces.nsISupportsString
      ])
```

JS component : the factory

A function NSGetFactory should exists in the file (gecko > 2.0).
This function is responsible to instantiate the component.
XPCOMUtils provide a method to generate it.

```
const NSGetFactory = XPCOMUtils.generateNSGetFactory([myStringImpl]);
```

JS component: registration

Since Gecko 2.0, declare the JS component into the chrome.manifest file, by indicating its class ID and its contract ID.

```
component CLASSID FILE
contract CONTRACTID CLASSID
```

With our example :

```
component {df53456e-0484-4098-9353-ee22661e6819} components/mystring.js
contract @xulfr.org/tutorial/mystring;1 {df53456e-0484-4098-9353-ee22661e6819}
```

For binary components, only do this :

```
binary-component components/mycomponent.dll
```

Old Gecko, JS Component: the factory

The factory is responsible to instancy the object corresponding to the requested interfaces. The factory should implement the nsIFactory interface.

```
var myStringFactory = {

    createInstance: function(outer, iid) {
        if (outer != null)
            throw Components.results.NS_ERROR_NO_AGGREGATION;

        return (new myStringImpl()).QueryInterface(iid);
    }

    // in our javascript file, the factory is in a global variable
    // so it is always in memory, we don't need to implement this
    // method
    LockFactory: function (lock) { }
}
```

Most of time, there is one factory per component

Old Gecko, JS Component: the module

The module is responsible to register and instancy the factory corresponding to the requested contract id. The module should implement the nsIModule interface.

```
var myStringModule = {

    registerSelf: function(compMgr, fileSpec, location, type) {
        compMgr.QueryInterface(Components.interfaces.nsIComponentRegistrar)
               .registerFactoryLocation(MYSTRING_CLASSID,
                                        "myString JS component",
                                        MYSTRING_CONTRACTID,
                                        FileSpec, location, type);
    },

    getClassObject: function(compMgr, cid, iid) {
        if (!iid.equals(Components.interfaces.nsIFactory))
            throw Components.results.NS_ERROR_NOT_IMPLEMENTED;

        if (!cid.equals(MYSTRING_CLASSID))
            throw Components.results.NS_ERROR_FACTORY_NOT_REGISTERED;

        return myStringFactory;
    },

    canUnload: function(compMgr) { return true; }
};
```

Old Gecko, JS Component: declaring the module

Each javascript file for components should have one specific function, NSGetModule, which is called during the load of the file by the XPCOM system. This function should return the module object we have implemented.

```
function NSGetModule(comMgr, fileSpec) {
  return myStringModule;
}
```

Old Gecko, XPCOMUtils: the module

For Gecko 1.8 and 1.9, no need to create a factory and a module by hand. Just call the generateModule method. It will generate a module object and a factory object for each object you provide:

```
function NSGetModule(aCompMgr, aFileSpec) {
  return XPCOMUtils.generateModule([myStringImpl]);
}
```

Creating an interface

Example of a file xfrIBankAccount.idl

```
#include "nsISupports.idl"

[scriptable, uuid(163CD99E-45DB-4717-A4AA-8FC660956310)]
interface xfrIBankAccount : nsISupports
{
   readonly attribute long balance;

   void credit (in long aAmount);

   void charge (in long aAmount);
};
```

Interfaces: generating an xpt file

- Use the xpidl program of the SDK. Here are main parameters:
  - "-m typelib" to indicate that you want to generate a xpt file
  - "-I path/to/sdk/idl" to indicate the directory which contains all the idl of the SDK
  - "-o path/to/store" to indicate the directory where to create the xpt file
  - your_file.idl
- example:

```
xpidl -m typelib -w -I ../idl/ -o components/bankaccount  dev/xfrIBankAccount.idl
```

Creating a C++ component

The SDK

- You can use the official SDK to build your library, however it has some limitations:
  - it contains only « frozen » and few other useful interfaces, so you have only corresponding headers and idl files to these interfaces.
  - It doesn't use the build system of Mozilla : it needs more effort to build cross-plateform libraries.
- Instead, you can use sources of XulRunner or Firefox, and build your component with them.
- Compilation with Firefox/XulRunner: http://developer.mozilla.org/en/docs/Build_Documentation

Compilation with firefox

- Summary
  - Create a directory (« myextension » for instance) into the « extensions » directory of the mozilla sources.
  - Create appropriate makefiles into « myextension » with the sources of your component
  - In the .mozconfig file used to build firefox or, add the name of the extension in the –enable-extension instruction
  - Compile firefox
- See details in the xpcom_cpp example, and in https://developer.mozilla.org/en/Creating_Custom_Firefox_Extensions_with_the_Mozilla_Build_System